

ПРОГРАММИСТУ

Р. Круз

# СТРУКТУРЫ ДАННЫХ И ПРОЕКТИРОВАНИЕ ПРОГРАММ



БИНОМ

# **СТРУКТУРЫ ДАННЫХ И ПРОЕКТИРОВАНИЕ ПРОГРАММ**

# **DATA STRUCTURES AND PROGRAM DESIGN**

THIRD EDITION

**Robert L. Kruse**

St. Mary's University  
Halifax, Nova Scotia

PRENTICE HALL  
Upper Saddle River, New Jersey 07458

ПРОГРАММИСТУ

Р. Круз

# СТРУКТУРЫ ДАННЫХ И ПРОЕКТИРОВАНИЕ ПРОГРАММ

Перевод 3-го английского издания  
К. Г. Финогенова

2-е издание (электронное)



Москва  
БИНОМ. Лаборатория знаний  
2014



УДК 004.7  
ББК 32.973.202  
К84

*Серия основана в 2005 г.*

**Круз Р. Л.**

**К84** Структуры данных и проектирование программ [Электронный ресурс] / Р. Л. Круз ; пер. с англ. — 2-е изд. (эл.). — М. : БИНОМ. Лаборатория знаний, 2014. — 765 с. : ил. — (Программисту).

ISBN 978-5-9963-1308-2

В качестве фундаментальных средств разработки программ рассматриваются такие вопросы, как структурное решение задач, абстракция данных, принципы программной инженерии и сравнительный анализ алгоритмов. Дано полное освещение большинства модулей знаний, касающихся структур данных и алгоритмов.

Большая часть глав начинается основной темой и сопровождается примерами, приложениями и практическими исследованиями.

Это учебное пособие дает основательные знания, которые позволяют студентам по ходу своей дальнейшей работы использовать ее также в качестве справочного пособия.

УДК 004.7  
ББК 32.973.202

**По вопросам приобретения обращаться:  
«БИНОМ. Лаборатория знаний»**

**Телефон: (499) 157-5272**

**e-mail: [binom@Lbz.ru](mailto:binom@Lbz.ru), <http://www.Lbz.ru>**

Authorized Translation from the English language edition, entitled DATA STRUCTURES AND PROGRAM DESIGN, 3rd Edition; by ROBERT KRUSE; and by BILL ZOBRIST; published by Pearson Education, Inc, publishing as Prentice Hall. Copyright © 1994 by Prentice Hall, Inc. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. Electronic RUSSIAN language edition published by BKL PUBLISHERS. Copyright © 2014.

Авторизованный перевод издания на английском языке, озаглавленного DATA STRUCTURES AND PROGRAM DESIGN, авторы ROBERT KRUSE и BILL ZOBRIST, опубликованного Pearson Education, Inc, осуществляющим издательскую деятельность под торговой маркой Prentice Hall. Copyright © 1994 by Prentice Hall, Inc. Все права защищены. Воспроизведение или распространение какой-либо части/частей данной книги в какой-либо форме, какими-либо способами, электронными или механическими, включая фотокопирование, запись и любые поисковые системы хранения информации, без разрешения Pearson Education, Inc запрещены. Электронная русскоязычная версия издана BKL Publishers. Copyright © 2014.

© Перевод, оформление. БИНОМ. Лаборатория знаний, 2008

ISBN 978-5-9963-1308-2

# Оглавление

---

<b>Предисловие</b>	14
Краткий обзор	15
Изменения в третьем издании	17
Структура курса	18
Разработка книги	19
Благодарности	20
<b>Глава 1. Принципы программирования</b>	21
1.1. Введение	21
1.2. Игра «Жизнь»	24
1.2.1. Правила игры «Жизнь»	24
1.2.2. Примеры	25
1.2.3. Решение	26
1.2.4. Life: главная программа	27
1.3. Стил ь программирования	29
1.3.1. Имена	29
1.3.2. Документация и форматы	32
1.3.3. Детализация программы и модульность	33
1.4. Кодирование, тестирование и дальнейшая детализация	39
1.4.1. Заглушки	39
1.4.2. Подсчет соседей	40
1.4.3. Ввод и вывод	42
1.4.4. Драйверы	46
1.4.5. Трассировка программы	47
1.4.6. Принципы тестирования программы	48
Подсказки и ловушки	52
Обзорные вопросы	54
Литература для дальнейшего изучения	55
Pascal	55
Программистские принципы	55
Игра «Жизнь»	56
<b>Глава 2. Введение в программную инженерию</b>	57
2.1. Поддержка программ	57
2.1.1. Обзор программы Life	58
2.1.2. Новый старт и новый метод для программы Life	60
2.2. Разработка алгоритма: второй вариант программы Life	63
2.2.1. Списки: спецификации для структуры данных	63

2.2.2. Программа Main .....	68
2.2.3. Скрытие информации .....	70
2.2.4. Детализация: разработка подпрограмм .....	71
2.2.5. Верификация и алгоритмы .....	75
2.3. Кодирование .....	78
2.3.1. Пакет обработки списков .....	79
2.3.2. Обработка ошибок .....	80
2.3.3. Демонстрация и тестирование .....	81
2.4. Кодирование процедур программы Life .....	85
2.5. Анализ и сравнение программ .....	89
2.6. Заключение и предварительный просмотр .....	91
2.6.1. Игра «Жизнь» .....	92
2.6.2. Разработка программы .....	93
2.6.3. Pascal .....	96
Подсказки и ловушки .....	99
Обзорные вопросы .....	99
Литература для дальнейшего изучения .....	99

<b>Глава 3. Стеки и рекурсия .....</b>	<b>101</b>
3.1. Стеки .....	101
3.1.1. Введение .....	101
3.1.2. Первый пример: реверсирование строки .....	102
3.1.3. Скрытие информации .....	104
3.1.4. Спецификации для стека .....	105
3.1.5. Реализация стеков .....	107
3.1.6. Связные стеки .....	109
3.2. Введение в рекурсию .....	116
3.2.1. Кадры стека для подпрограмм .....	116
3.2.2. Дерево вызовов подпрограмм .....	117
3.2.3. Факториалы: рекурсивное определение .....	119
3.2.4. Метод разбиения: башни Ханоя .....	121
3.3. Принципы рекурсии .....	128
3.3.1. Разработка рекурсивных алгоритмов .....	128
3.3.2. Как работает рекурсия .....	129
3.3.3. Хвостовая рекурсия .....	134
3.3.4. Когда не следует использовать рекурсию .....	136
3.3.5. Рекомендации и заключения .....	140
Подсказки и ловушки .....	142
Обзорные вопросы .....	143
Литература для дальнейшего изучения .....	143

<b>Глава 4. Примеры рекурсии .....</b>	<b>145</b>
4.1. Алгоритмы с отходом: откладывание работы .....	145
4.1.1. Решение задачи о восьми ферзях .....	146
4.1.2. Пример: четыре ферзя .....	146
4.1.3. Алгоритм с отходом .....	148
4.1.4. Детализация: выбор структур данных .....	148
4.1.5. Анализ алгоритма с отходом .....	152

4.2. Древовидные программы: прогнозирование в играх .....	154
4.2.1. Деревья игр .....	154
4.2.2. Метод минимакса .....	156
4.2.3. Разработка алгоритма .....	157
4.2.4. Детализация .....	159
4.3. Компиляция методом рекурсивного спуска .....	161
4.3.1. Главная программа .....	162
4.3.2. Объявления типов .....	163
4.3.3. Синтаксический анализ предложений .....	164
4.3.4. Синтаксический анализатор предложений языка Pascal ....	168
Подсказки и ловушки .....	179
Обзорные вопросы .....	180
Литература для дальнейшего изучения .....	180

<b>Глава 5. Очереди</b> .....	182
5.1. Определения .....	182
5.2. Реализация очередей .....	185
5.3. Кольцевые очереди в языке Pascal .....	190
5.4. Приложения очередей: Моделирование .....	195
5.4.1. Введение .....	195
5.4.2. Моделирование работы аэропорта .....	195
5.4.3. Случайные числа .....	197
5.4.4. Главная программа .....	198
5.4.5. Шаги моделирования .....	199
5.4.6. Пример результатов .....	202
5.5. Связные очереди .....	205
5.6. Приложения: полиномиальная арифметика .....	210
5.6.1. Цель проекта .....	210
5.6.2. Главная программа .....	211
5.6.3. Структуры данных и их реализация .....	214
5.6.4. Чтение и вывод полиномов .....	216
5.6.5. Сложение полиномов .....	218
5.6.6. Завершение проекта .....	220
5.7. Абстрактные типы данных и их реализации .....	222
5.7.1. Введение .....	222
5.7.2. Общие определения .....	224
5.7.3. Детализация спецификации данных .....	227
Подсказки и ловушки .....	228
Обзорные вопросы .....	230
Литература для дальнейшего изучения .....	230

<b>Глава 6. Списки</b> .....	231
6.1. Спецификации списков .....	231
6.2. Реализация списков .....	233
6.2.1. Непрерывная реализация .....	234
6.2.2. Реализация простого связывания .....	235
6.2.3. Вариация: сохранение текущей позиции .....	239
6.2.4. Дважды связные списки .....	240
6.2.5. Сравнение реализаций .....	243

6.3. Цепочки символов .....	246
6.3.1. Операции над цепочками символов .....	246
6.3.2. Реализация цепочек символов .....	248
6.4. Приложение: текстовый редактор .....	253
6.4.1. Спецификации .....	253
6.4.2. Реализация .....	254
6.5. Связные списки в массивах .....	261
6.6. Генерирование перестановок .....	271
Подсказки и ловушки .....	276
Обзорные вопросы .....	277
Литература для дальнейшего изучения .....	278

<b>Глава 7. Поиск .....</b>	<b>279</b>
7.1. Введение и обозначения .....	279
7.2. Последовательный поиск .....	281
7.3. Гардеробы: проект .....	287
7.3.1. Введение и спецификации .....	287
7.3.2. Демонстрационная и тестирующая программы .....	291
7.4. Двоичный поиск .....	295
7.4.1. Разработка алгоритма .....	296
7.4.2. Вариант с забыванием .....	297
7.4.3. Распознавание равенства .....	299
7.5. Деревья сравнений .....	302
7.5.1. Анализ для $n=10$ .....	303
7.5.2. Обобщение .....	306
7.5.3. Методы сравнения .....	309
7.5.4. Общее отношение .....	310
7.6. Нижние границы .....	313
7.7. Асимптотика .....	318
7.7.1. Введение .....	318
7.7.2. О большое .....	319
7.7.3. Неточность определения О большого .....	322
7.7.4. Порядки распространенных функций .....	323
Подсказки и ловушки .....	324
Обзорные вопросы .....	325
Литература для дальнейшего изучения .....	326

<b>Глава 8. Сортировка .....</b>	<b>327</b>
8.1. Введение и обозначения .....	327
8.2. Сортировка включением .....	329
8.2.1. Упорядоченные списки .....	329
8.2.2. Сортировка методом включения .....	330
8.2.3. Связный вариант .....	332
8.2.4. Анализ .....	334
8.3. Сортировка методом выбора .....	339
8.3.1. Алгоритм .....	339
8.3.2. Непрерывная реализация .....	340
8.3.3. Анализ .....	342
8.3.4. Сравнения .....	342

8.4. Упорядочение методом Шелла .....	344
8.5. Нижние границы .....	347
8.6. Сортировка методом разбиения .....	350
8.6.1. Базовые идеи .....	350
8.6.2. Пример .....	351
8.7. Сортировка слиянием для связанных списков .....	357
8.7.1. Процедуры .....	357
8.7.2. Анализ метода сортировки слиянием .....	359
8.8. Метод быстрой сортировки для непрерывных списков .....	365
8.8.1. Главная процедура .....	365
8.8.2. Разделение списка .....	366
8.8.3. Анализ метода быстрой сортировки .....	368
8.8.4. Анализ метода быстрой сортировки для среднего случая ..	371
8.8.5. Сравнение с методом слияния .....	373
8.9. Пирамиды и пирамидальная сортировка .....	377
8.9.1. Двухвариантные деревья как списки .....	377
8.9.2. Пирамидальная сортировка .....	379
8.9.3. Анализ пирамидальной сортировки .....	382
8.9.4. Очереди с приоритетами .....	383
8.10. Обзор: сравнение методов .....	386
Подсказки и ловушки .....	390
Обзорные вопросы .....	391
Литература для дальнейшего изучения .....	392
<b>Глава 9. Таблицы и извлечение информации .....</b>	<b>394</b>
9.1. Введение: переход через барьер lgn .....	394
9.2. Прямоугольные массивы .....	395
9.3. Таблицы различных форм .....	398
9.3.1. Треугольные таблицы .....	398
9.3.2. Рваные таблицы .....	400
9.3.3. Инвертированные таблицы .....	401
9.4. Таблицы: новый абстрактный тип данных .....	404
9.5. Приложение: поразрядная сортировка .....	407
9.5.1. Идея метода .....	407
9.5.2. Реализация .....	408
9.5.3. Анализ .....	411
9.6. Хеширование .....	412
9.6.1. Разреженные таблицы .....	412
9.6.2. Выбор хеш-функции .....	414
9.6.3. Разрешение конфликтов с помощью открытой адресации ..	417
9.6.4. Разрешение столкновений посредством связанных цепочек ..	422
9.7. Анализ хеширования .....	428
9.8. Заключение: сравнение методов .....	434
9.9. Приложение: снова игра «Жизнь» .....	435
9.9.1. Выбор алгоритма .....	435
9.9.2. Спецификация структур данных .....	435
9.9.3. Главная программа .....	437
9.9.4. Процедуры .....	438
Подсказки и ловушки .....	442

Обзорные вопросы .....	443
Литература для дальнейшего изучения .....	444
<b>Глава 10. Двоичные деревья .....</b>	<b>445</b>
10.1. Двоичные деревья .....	445
10.1.1. Определения .....	445
10.1.2. Просмотр двоичных деревьев .....	448
10.1.3. Связная реализация двоичных деревьев .....	453
10.2. Деревья двоичного поиска .....	457
10.2.1. Упорядоченные списки и реализации .....	459
10.2.2. Поиск по дереву .....	460
10.2.3. Включение в дерево двоичного поиска .....	464
10.2.4. Древовидная сортировка .....	467
10.2.5. Удаление из дерева двоичного поиска .....	469
10.3. Построение дерева двоичного поиска .....	477
10.3.1. Начинаем .....	478
10.3.2. Объявления и главная процедура .....	479
10.3.3. Включение узла .....	480
10.3.4. Завершение задачи .....	481
10.3.5. Оценка .....	483
10.3.6. Случайные деревья поиска и оптимизация .....	483
10.4. Балансирование по высоте: AVL-деревья .....	486
10.4.1. Определение .....	487
10.4.2. Включение узла .....	488
10.4.3. Удаление узла .....	494
10.4.4. Высота AVL-деревя .....	498
10.5. Скошенные деревья: самонастраивающиеся структуры данных ..	501
10.5.1. Введение .....	501
10.5.2. Шаги скашивания дерева .....	502
10.5.3. Алгоритм скашивания .....	505
10.5.4. Амортизационный анализ алгоритмов: введение .....	509
10.5.5. Амортизационный анализ скашивания .....	514
Подсказки и ловушки .....	519
Обзорные вопросы .....	520
Литература для дальнейшего изучения .....	522
<b>Глава 11. Многовариантные деревья .....</b>	<b>524</b>
11.1. Сады, деревья и двоичные деревья .....	524
11.1.1. Классификация видов .....	524
11.1.2. Упорядоченные деревья .....	525
11.1.3. Леса и сады .....	528
11.1.4. Формальное соответствие .....	529
11.1.5. Повороты .....	530
11.1.6. Резюме .....	531
11.2. Деревья лексикографического поиска: трай-деревья .....	533
11.2.1. Трай-деревья .....	533
11.2.2. Поиск ключа .....	533
11.2.3. Алгоритм на языке Pascal .....	534
11.2.4. Включение в трай-дерево .....	535

11.2.5. Удаление из трай-дерева	536
11.2.6. Оценка трай-деревьев	537
11.3. Внешний поиск: В-деревья	538
11.3.1. Время доступа	538
11.3.2. Многовариантные деревья поиска	539
11.3.3. Сбалансированные многовариантные деревья	539
11.3.4. Включение в В-дерево	540
11.3.5. Алгоритмы на языке Pascal: поиск и включение	542
11.3.6. Удаление из В-дерева	548
11.4. Красно-черные деревья	557
11.4.1. Введение	557
11.4.2. Определения и анализ	558
11.4.3. Включение	560
11.4.4. Включение на языке Pascal	563
Подсказки и ловушки	566
Обзорные вопросы	567
Литература для дальнейшего изучения	568

<b>Глава 12. Графы</b>	<b>569</b>
12.1. Математические основы	569
12.1.1. Определения и примеры	569
12.1.2. Неориентированные графы	570
12.1.3. Ориентированные графы	571
12.2. Компьютерное представление	572
12.3. Просмотр графа	576
12.3.1. Методы	576
12.3.2. Алгоритм просмотра в глубину	577
12.3.3. Алгоритм просмотра в ширину	578
12.4. Топологическая сортировка	579
12.4.1. Постановка задачи	579
12.4.2. Алгоритм упорядочения в глубину	581
12.4.3. Алгоритм упорядочения в ширину	582
12.5. Алгоритм экономного продвижения: кратчайшие маршруты	584
12.6. Графы как структуры данных	589
Подсказки и ловушки	591
Обзорные вопросы	591
Литература для дальнейшего изучения	592

<b>Глава 13. Конкретный пример: польская нотация</b>	<b>593</b>
13.1. Постановка задачи	593
13.1.1. Формула корней квадратного уравнения	593
13.2. Идея	595
13.2.1. Дерево выражения	595
13.2.2. Польская нотация	597
13.2.3. Метод для языка Pascal	599
13.3. Оценка выражений в польской нотации	599
13.3.1. Оценка выражений в префиксной форме	599
13.3.2. Соглашения языка Pascal	600
13.3.3. Pascal-процедура для префиксной оценки	601



13.3.4. Оценка постфиксных выражений .....	602
13.3.5. Доказательство правильности программы: подсчет элементов в стеке .....	603
13.3.6. Рекурсивная оценка постфиксных выражений .....	607
13.4. Преобразование из инфиксной формы в польскую .....	611
13.5. Интерактивная программа оценки выражений .....	617
13.5.1. Общая структура .....	618
13.5.2. Представление данных .....	619
13.5.3. Инициализация и вспомогательные задачи .....	623
13.5.4. Преобразование выражения .....	627
13.5.5. Оценка выражения .....	637
13.5.6. Графическое отображение выражения .....	639
Литература для дальнейшего изучения .....	642

<b>Приложение А. Математические методы .....</b>	<b>643</b>
A.1. Суммы степеней целых чисел .....	643
A.2. Логарифмы .....	645
A.2.1. Определение логарифмов .....	646
A.2.2. Простые свойства .....	646
A.2.3. Выбор основания .....	647
A.2.4. Натуральные логарифмы .....	648
A.2.5. Обозначения .....	649
A.2.6. Изменение основания логарифмов .....	649
A.2.7. Логарифмические графики .....	650
A.2.8. Гармонические числа .....	650
A.3. Перестановки, сочетания, факториалы .....	652
A.3.1. Перестановки .....	652
A.3.2. Сочетания .....	653
A.3.3. Факториалы .....	653
A.4. Числа Фибоначчи .....	655
A.5. Числа Каталана .....	656
A.5.1. Основной результат .....	656
A.5.2. Доказательство посредством однозначного соответствия ..	657
A.5.3. История вопроса .....	659
A.5.4. Численные результаты .....	659
Литература для дальнейшего изучения .....	661

<b>Приложение В. Случайные числа .....</b>	<b>663</b>
B.1. Введение .....	663
B.2. Метод .....	664
B.3. Разработка программы .....	665
Литература для дальнейшего изучения .....	670

<b>Приложение С. Модули, включаемые файлы и утилиты .....</b>	<b>671</b>
C.1. Модули Turbo Pascal .....	671
C.1.1. Введение .....	671
C.1.2. Синтаксис модулей .....	672
C.2. Включаемые файлы .....	674
C.2.1. Замена модулей включаемыми файлами .....	674
C.2.2. Родовые средства .....	675

С.3. Модули, используемые в тексте .....	677
С.3.1. Структуры данных .....	677
С.3.2. Модуль утилит Utility .....	678
С.3.3. Модуль анализа процессорного времени .....	680
С.3.4. Модуль для обслуживания файлов .....	682
С.3.5. Модуль случайных чисел .....	685
С.4. Программы поиска и сортировки .....	685
С.4.1. Демонстрационная программа .....	685
С.4.2. Создание файлов данных для тестирования программ .....	686
<b>Приложение D. Свойства языка Pascal</b> .....	<b>694</b>
D.1. Записи в языке Pascal .....	694
D.2. Процедуры .....	700
D.2.1. Процедуры в качестве параметров .....	700
D.2.2. Упреждающие объявления .....	702
D.3. Указатели и связные списки .....	703
D.3.1. Введение и обзор .....	703
D.3.2. Указатели и динамическая память в языке Pascal .....	707
D.3.3. Основы связных списков .....	711
D.3.4. Связная реализация простых списков .....	715
D.3.5. Советы для программистов .....	718
D.4. Синтаксические диаграммы .....	721
D.5. Общие правила .....	730
D.5.1. Идентификаторы .....	730
D.5.2. Правила использования пробелов .....	731
D.5.3. Указания по формату программы .....	732
D.5.4. Пунктуация .....	733
D.5.5. Альтернативные знаки .....	733
D.6. Стандартные объявления .....	733
D.6.1. Константы .....	733
D.6.2. Типы .....	735
D.6.3. Переменные .....	735
D.6.4. Процедуры .....	736
D.6.5. Функции .....	736
D.7. Операторы .....	737
<b>Предметный указатель</b> .....	<b>738</b>

# Предисловие

---

Ученик плотника довольствуется лишь молотком и пилой, однако мастер использует более замысловатые инструменты. Также и компьютерное программирование, чтобы удовлетворить все возрастающим требованиям современных прикладных задач, прибегает к разнообразным утонченным средствам, и только практическая работа с ними позволяет программисту приобрести необходимую сноровку в их эффективном использовании. В этой книге в качестве фундаментальных средств разработки программ рассматриваются такие вопросы, как структурное решение задач, абстракция данных, принципы программной инженерии и сравнительный анализ алгоритмов. Для того чтобы показать, как все эти средства совместно используются для построения законченных программ, в книге детально разбирается несколько примеров значительной сложности.

Многие изучаемые нами алгоритмы и структуры данных обладают внутренней элегантностью, проявляющейся в том, что внешняя простота скрывает богатый диапазон и мощь их применений. Очень быстро обучающийся обнаруживает, что наивные методы, обычно предлагаемые во вводных курсах, нуждаются в очень серьезных улучшениях. Однако элегантность метода сочетается с неопределенностью. Вы вскоре сталкиваетесь с тем, что выбор наилучшего из нескольких возможных подходов к решению конкретной задачи далеко не очевиден. Вот здесь-то и становится оправданным рассмотрение ряда непростых проблем, интересных самих по себе и имеющих большое практическое значение, а также показ применимости математических методов к верификации и анализу алгоритмов.

Студенты, приступая к изучению программирования, часто затрудняются в практическом приложении абстрактных идей. Поэтому в настоящей книге особое внимание уделяется вопросам преобразования идей в алгоритмы, а также дальнейшего уточнения этих алгоритмов с целью получения конкретных программ, которые уже можно использовать для решения практических задач. Также на первое место перед выбором структур данных и их реализации выступает процесс спецификации и абстракции данных.

Лично я являюсь сторонником движения от конкретного к абстрактному, последовательной разработки мотивирующих примеров, дающих возможность представить идеи в более общей форме. Большинству студентов на ранних этапах их обучения требуется помощь в освоении непосредственного приложения идей, с которыми они знакомятся, и лучше

всего, если они будут писать и выполнять программы, иллюстрирующие изучаемые ими концепции. С этой целью книга содержит большое количество программных примеров, как коротких процедур, так и законченных программ значительной длины. Более того, упражнения и программные проекты составляют неотъемлемую часть книги. Многие из этих упражнений иллюстрируют изучаемую в настоящий момент тему, и их выполнение требует написания и отладки реальных программ с целью анализа и сравнения используемых алгоритмов. Другие представляют собой более объемные программные проекты, а некоторые предназначены для использования небольшой группой работающих совместно студентов.

## Краткий обзор

Принципы  
программирования

Посредством рассмотрения первого большого проекта (игры «Жизнь» Дж. Конвея) глава 1 вводит принципы нисходящей детализации, проектирования программы, критического обзора и тестирования; демонстрация этих принципов позволит студентам использовать их при изучении последующих разделов. В то же время первый проект позволит студентам освежить свои знания языка Pascal, который выбран в качестве базового языка программирования для этой книги.

Введение  
в программную  
инженерию

В главе 2 вводятся несколько базовых принципов программной инженерии, включая проблемы спецификации и анализа, прототипирования, абстракции данных, а также разработки, детализации, верификации и анализа алгоритма. Эти принципы используются при разработке второго варианта игры «Жизнь», основанного на алгоритме, достаточно тонком для наглядной демонстрации необходимости в точной спецификации и верификации, и, кроме того, позволяющем показать важность правильного выбора структур данных.

Стеки и рекурсии

Глава 3 продолжает разъяснение концепций абстракции данных и разработки алгоритмов посредством изучения стеков как абстрактного типа данных и рекурсии как метода решения задач, а также внутренних связей между стеками, рекурсией и определенными типами деревьев. В главе 4 эти концепции иллюстрируются рассмотрением нескольких важных приложений рекурсии, включая алгоритмы с отходом, программы с древовидной структурой и рекурсивно-нисходящий синтаксический анализ.

Примеры  
рекурсий

Очереди

Списки

Центральной темой двух следующих глав являются очереди и списки. Здесь рассматриваются различные реализации каждого абстрактного типа данных, строятся большие прикладные программы, демонстрирующие относительные преимущества различных реализаций, и весьма неформальным образом вводятся идеи анализа алгоритмов. Основная цель этих глав — подвести студента к пониманию важности абстракции данных и к приложению методов нисходящего проектирования как к данным, так и к алгоритмам.

Поиск  
и сортировка

В главах 7, 8 и 9 рассматриваются алгоритмы поиска, сортировки и работы с таблицами (включая хеширование). Эти главы демонстрируют

взаимосвязь между алгоритмами и соответствующим им абстрактными типами данных, структурами данных и реализациями. Здесь вводится понятие «*O* большого» для элементарного анализа алгоритмов и подчеркивается критическая важность выбора, позволяющего наиболее эффективным образом использовать пространство памяти, время и программистские усилия.

Этот выбор требует применения аналитических методов оценки алгоритмов, причем арсенал для проведения такого анализа должна нам предоставить комбинаторная математика. На начальных стадиях обучения мы не можем требовать от студентов ни достаточно глубоких знаний, ни математической зрелости, необходимых для оттачивания до совершенства их навыков. Моя цель, таким образом, — помочь студентам осознать важность приобретения этих навыков при изучении ими в дальнейшем соответствующих разделов математики.

Безусловно, к наиболее элегантным и полезным структурам данных следует отнести двоичные деревья. Их изучение, которому посвящена глава 10, связывает вместе концепции списков, поиска и сортировки. Относясь к рекурсивно определяемым структурам данных, двоичные деревья предоставляют студентам превосходную возможность изучить рекурсии применительно как к структурам данных, так и к алгоритмам. Глава начинается с элементарных сведений и постепенно подводит читателя к скошенным деревьям и амортизационному анализу алгоритмов.

В главе 11 продолжается изучение более сложных структур данных, включая трай-структуры, В-деревья и красно-черные деревья. В следующей главе вводятся графы, как наиболее общие структуры, используемые при решении задач.

Конкретное исследование в главе 13 позволяет весьма детально рассмотреть польскую нотацию и исследовать взаимосвязь рекурсии, деревьев и стеков в качестве средств решения задач и разработки алгоритмов. Некоторые аспекты этого исследования могут служить неформальным введением в проектирование компиляторов. Как и в других местах книги, представленные алгоритмы разработаны до деталей и включены в работоспособную Pascal-программу. Эта программа принимает входные данные в виде обычного (инфиксного) выражения, преобразует это выражение в постфиксную форму и оценивает его применительно к заданным значениям переменных.

Приложения посвящены темам, строго говоря, не относящимся к предмету книги, но часто отсутствующим в начальных курсах.

В приложении А дан обзор ряда вопросов дискретной математики. Последние два раздела, посвященные числам Фибоначчи и Каталана, являются более продвинутыми и необязательны для понимания основного текста книги, однако они включены с целью развития интереса к комбинаторике у студентов с математическим складом ума.

В приложении В обсуждаются псевдослучайные числа, программы генерации таких чисел и их приложения. Этот предмет может заинтересовать многих студентов, хотя часто он выпадает из типичных программ обучения.

Таблицы  
и извлечение  
информации

Двоичные  
деревья

Многовариантные  
деревья

Графы

Конкретное  
исследование:  
польская нотация

Математические  
методы

Случайные  
числа

Модули,  
включаемые файлы  
и процедуры-  
утилиты

Средства языка  
Pascal

Приложение С детально описывает использование модулей системы программирования Turbo Pascal и включаемых (include-) файлов при реализации абстрактных типов данных. В основном тексте книги в ряде мест используются специально разработанные процедуры-утилиты и модули. В приложении С даны развернутые описания этих программных средств.

Наконец, приложение D посвящено некоторым средствам программирования на языке Pascal, обычно не входящим в начальные курсы обучения этому языку, именно, записям, процедурным параметрам и упреждающим объявлениям. Далее, в приложение D включены сведения о типах указателей языка Pascal и элементарных операциях над указателями и связными списками, поскольку эти средства являются неотъемлемой частью языка Pascal, а также с целью придания разделу книги, посвященному связным спискам, большей общности. В завершающей части приложения D приведены стандартные диаграммы и таблицы, описывающие синтаксис языка Pascal, а также дополнительная информация, которая может помочь читателю в решении программистских проблем.

## Изменения в третьем издании

При подготовке книги к третьему изданию весь текст был тщательно просмотрен в плане улучшения представления материала, а также с целью отражения предложений многочисленных читателей, изучавших эту книгу. Принципиальные изменения заключаются в следующем.

- Все программы были переработаны и улучшены с целью подчеркивания абстракции данных, получения повторно используемого кода и обеспечения однородности и элегантности стиля.
- Для облегчения этого процесса в книге широко используются модули Turbo Pascal, хотя и в ненавязчивой форме, чтобы обеспечить беспрепятственное преобразование в стандартный Pascal или, при необходимости, в другой диалект языка.
- Расширена документация к программам путем включения во все подпрограммы неформальных спецификаций (пред- и постусловий).
- Рекурсия стала использоваться в книге значительно раньше, а ее повторное постоянное использование подчеркивает важность этого средства.
- В книге расширено представление современных продвинутых тем включением разделов, посвященных скошенным и красно-черным деревьям и амортизационному анализу алгоритмов.
- Включены новые примеры конкретных исследований, в частности, синтаксический анализатор предложений языка Pascal и миниатюрный текстовый редактор.
- Добавлено много новых упражнений и программных проектов, включая последовательные варианты проектов, посвященных извлечению информации, которые требуют от студента сравнительного анализа производительности различных структур данных и алгоритмов.
- Материал по теории графов и алгоритмам реализации графов выделен в отдельную главу.

- Рассмотрение списков стало более логичным, во-первых, за счет упрощения спецификаций списков, и, во-вторых, перенесением элементарного обзора типов указателей языка Pascal в приложение, где этот материал изложен более подробно, чем раньше. Такое расположение материала позволило в основном тексте книги сконцентрировать внимание на основополагающих идеях, не отвлекаясь на детали реализации.
- Изъяты некоторые устаревшие темы, например, удаление рекурсии.
- Поставлявшийся ранее с книгой программный диск теперь можно приобрести у издателя; весь программный пакет также доступен в Интернете. Пакет включает в себя исходные тексты всех программ и программных фрагментов, вошедших в книгу, вместе с выполнимыми файлами (для компьютеров IBM PC и совместимых с ними), всех демонстрационных программ и почти всех программных проектов этой книги.
- Для преподавателей имеется также отдельное (бесплатное) пособие, включающее в себя полные решения всех упражнений и программных проектов, диск с упомянутым выше программным пакетом, а также второй диск с полными исходными текстами всех программных проектов.

## Структура курса

Для работы над этой книгой студент должен овладеть начальным курсом программирования и иметь опыт использования элементарных средств языка Pascal. В приложении D собраны некоторые продвинутые средства языка Pascal, обычно не включаемые в вводные курсы. Для анализа почти всех алгоритмов достаточно хорошего знания школьной математики, хотя знакомство с дискретной математикой (например, параллельно с чтением книги) будет весьма полезным. В приложении A дан обзор всех необходимых математических понятий.

Книга предназначена в качестве учебного пособия по таким курсам, как ACM Course CS2 (*Program Design and Implementation*, Проектирование и реализация программ), ACM Course CS7 (*Data Structures and Algorithm Analysis*, Структуры данных и анализ алгоритмов), а также курсам, являющимся комбинацией упомянутых выше. В книге дано полное освещение большинства модулей знаний ACM/IEEE<sup>1</sup>, касающихся структур данных и алгоритмов. Сюда входят модули:

- AL1 Базовые структуры данных: массивы, таблицы, стеки, очереди, деревья и графы;
- AL2 Абстрактные типы данных;
- AL3 Рекурсии и рекурсивные алгоритмы;
- AL4 Анализ сложности с использованием нотации «*O* большого»;
- AL6 Сортировка и поиск;
- AL8 Практические стратегии решения задач с примерами серьезных конкретных исследований.

требуемый  
исходный  
уровень  
подготовки

содержание

<sup>1</sup> См. *Computing Curricula 1991: Report of the ACM/IEEE-CS Curriculum Task Force*, ACM Press, New York, 1990.



Три наиболее продвинутых модуля знаний, AL5 (complexity classes and NP-complete problems, классы сложности и НП-полные задачи), AL7 (computability and undecidability, вычислимость и неразрешимость) и AL9 (parallel and distributed algorithms, параллельные и распределенные алгоритмы) не освещаются в этой книге.

Большая часть глав этой книги построена таким образом, что сначала представляются основные темы главы, сопровождаемые примерами, приложениями и практическими исследованиями. Таким образом, при нехватке времени на серьезное изучение можно без потери связности быстро переходить от главы к главе, знакомясь лишь с наиболее существенными понятиями. В дальнейшем, когда время позволит, и студент, и преподаватель смогут обращаться вразбивку к дополнительным темам и комплексным примерам.

двухсеместровый  
курс

Двухсеместровый курс практически покрывает всю книгу, удачно интегрируя в себе многие темы таких разделов, как решение задач, структуры данных, разработка программ и анализ алгоритмов. Для понимания общих методов студентам требуется время и практика. Комбинируя изучение абстракции данных, структур данных и алгоритмов с их реализациями в проектах реального размера, такой интегрированный курс будет служить твердой базой, на которой в дальнейшем могут основываться курсы, имеющие более теоретический характер.

Даже при не вполне исчерпывающем изучении, эта книга даст достаточно основательные знания, которые позволят интересующимся студентам по ходу своей дальнейшей работы использовать ее в качестве справочного пособия. В любом случае необходимо поручить студентам разработку основных программных проектов и предоставить им достаточное время для доведения их до рабочего состояния.

## Разработка книги

Эта книга вместе с дополнительным материалом была написана с помощью собственного программного обеспечения автора под названием PreT<sub>E</sub>X, объединяющего в себе препроцессор и макропакет для полиграфической системы T<sub>E</sub>X<sup>2</sup>. Пакет PreT<sub>E</sub>X, используя контекстную зависимость, автоматически обеспечивает значительную часть разметки текста, требуемую программой T<sub>E</sub>X. PreT<sub>E</sub>X также предоставляет некоторые полезные для автора средства, в частности, мощную систему перекрестных ссылок, существенное упрощение набора математических выражений и листингов компьютерных программ, и автоматическую генерацию предметного указателя и содержания. В то же время PreT<sub>E</sub>X позволяет поэтапно обрабатывать текст книги, содержащийся в относительно небольших по размеру файлах. Решения, размещенные вместе с упражнениями и проектами, автоматически убираются из текста и помещаются в отдельные документы. В сочетании с языком PostScript описания страниц, PreT<sub>E</sub>X предоставляет удобные средства для цветоделения, обработки растровых изображений и других специальных приемов.

<sup>2</sup> Система T<sub>E</sub>X была разработана Доналдом Кнутом (Donald E. Knuth), который внес весьма значительный вклад в наше сегодняшнее понимание структур данных и алгоритмов. (См. ссылки на его имя в предметном указателе).



Для книг, подобных этой, наиболее важной чертой PreTeX является возможность обработки им компьютерных программ. Программы не включаются в основной текст книги; они размещаются в отдельных вторичных файлах вместе с любыми поясняющими текстами и разметочной информацией. Помещая теги в соответствующие места вторичных файлов, PreTeX может извлекать в любом порядке требуемые фрагменты вторичного файла для вывода их на печать вместе с основным текстом. С помощью еще одной утилиты (с именем StripTeX) можно обработать вторичный файл, удалив из него все теги, текст и разметку, получив в результате программу, годную для компиляции. Таким образом, один и тот же исходный файл автоматически предоставляет и листинг программы для печати, и компилируемый программный код. В этом случае автор приобретает уверенность в безусловной правильности листинга компьютерной программы, включаемой в текст книги.

Для настоящего издания книги все диаграммы и рисунки были подготовлены в виде кода PostScript с помощью программы Adobe Illustrator. Такая процедура позволяет автоматически включать все иллюстрации в предварительные варианты рукописи, что сокращает конечные этапы подготовки книги к печати, устраняя необходимость ручной обработки оригинал-макета.

## Благодарности

В течение ряда лет эта книга существенно улучшалась стараниями многих людей: моей семьи, друзей, коллег по работе и студентов. В первом и втором издании были перечислены некоторые из них, чей вклад в создание книги был особенно важен. После выхода в свет второго издания были опубликованы две производные книги, *Programming with Data Structures* и *Data Structures and Program Design in C*, а также переводы этих книг на ряд языков. Читатели, которых слишком много, чтобы их можно было перечислить, присылали мне свои замечания и предложения. Я выражаю всем им свою искреннюю благодарность и рад сообщить, что многие из полученных предложений нашли свое отражение в настоящем издании книги.

Пол Мейлхот (Paul Mailhot), сначала мой студент, а теперь — постоянный ассистент, внес значительный вклад в настоящее издание, преданно работая вместе со мной на всех этапах создания книги. Он написал все программы, включенные в книгу, а также в сопровождающее ее пособие *Instructor's Resource Manual*, улучшив их стиль и элегантность и обеспечив возможность повторного использования. Он также внимательно вычитал весь текст, внося много ценных предложений. На нем лежала вся ответственность за выпуск как самой книги, так и приложений к ней; готовя книгу к печати, он в полной мере использовал возможности системы PreTeX, а с помощью Adobe Illustrator он довел все иллюстрации до совершенного состояния. Работа с таким надежным помощником доставляет истинное удовольствие.

Роберт Л. Круз

## Принципы программирования

---

Эта глава представляет собой обзор важных принципов правильного программирования, в особенности, применительно к проектам значительного объема, и иллюстрирует методы разработки эффективных алгоритмов. По ходу изложения материала мы поставим ряд вопросов, касающихся программного проектирования, ответы на которые будут получены в последующих главах. Мы также коснемся многих важных средств языка Pascal посредством использования их при написании программ.

### 1.1. Введение

Основная трудность при разработке больших компьютерных программ заключается не в определении цели разрабатываемой программы и даже не в поиске методов, позволяющих достичь поставленной цели. Президент какой-нибудь фирмы мог бы сказать: «Давайте возьмем компьютер, и пусть он учитывает все наши товары, счета и списки сотрудников, и пусть он будет оповещать нас о необходимости произвести переучет товаров, и о перерасходе средств, и пусть он еще выписывает ведомости на зарплату, и все будет хорошо!» При достаточных затратах времени и усилий группа системных аналитиков и программистов могла бы определить, как различные сотрудники фирмы выполнят всю эту работу, и разработать программы, делающие то же самое.

Такой подход, однако, почти неминуемо приведет к разрушительным последствиям. Беседуя со служащими, системные аналитики обнаружат, что некоторые задачи можно запрограммировать без особого труда; они напишут соответствующие программы и поставят их на компьютер. Продолжая действовать таким же образом и устанавливая на компьютер следующую порцию программ, они увидят, что новые программы опираются на старые. Однако выходные данные первой порции программ не вполне соответствуют требованиям второй порции. Эту проблему можно преодолеть, написав дополнительные программы преобразования данных, полученных от первых программ, в форму, требуемую вторыми. Программный проект начинает напоминать лоскутное одеяло. Некоторые его участки достаточно прочны, другие слабее. Одни кусочки аккуратно пришиты к соседним, другие едва приметаны. Если программистам повезет, их детище сможет в течение некоторого времени достаточно хорошо выполнять текущую работу. Если, однако, в проект потребуется внести хоть какое-то изменение, это приведет к непредсказуемым последствиям для работоспо-

проблемы  
больших  
программ

способности всей системы. Потом потребуются новое изменение или возникнет неожиданная проблема, возможно даже критическая ситуация, требующая срочного решения, и весь этот программный проект окажется столь же эффективен, как лоскутное одеяло, используемое для спасения людей, прыгающих из горящего здания.

цель книги

Главной целью этой книги является описание программных методов и средств, позволяющих создавать проекты реалистичного размера, программы, имеющие значительно больший объем, чем те, которые используются для демонстрации элементарных средств программирования. Поскольку «кусочный» метод поиска решения больших задач оказывается неприемлемым, мы должны прежде всего разработать последовательный, единообразный и логичный подход, в котором необходимо соблюсти важные принципы разработки программ, принципы, часто нарушаемые при написании небольших программ, но несоблюдение которых для больших проектов окажется фатальным.

спецификация  
задачи

Первое серьезное препятствие при решении сложной задачи заключается в необходимости точной формулировки, в чем именно эта задача заключается. Необходимо преобразовать неясные цели, противоречивые требования и, возможно, туманные пожелания в четко сформулированный проект, который можно реализовать с помощью программ. При этом методы разделения работы, используемые людьми, не обязательно будут эффективны для компьютера. Таким образом, прежде всего следует точно описать конечные цели, а затем постепенно дробить работу на меньшие по размеру задачи до тех пор, пока они не окажутся обозримого размера.

разработка  
программы

Принцип, исповедуемый многими программистами, «сначала заставь свою программу работать, а после этого можешь ее совершенствовать», часто оказывается эффективным для простых программ, однако он неприменим к программам большого размера. Каждый фрагмент большой программы должен быть хорошо организован, ясно написан и понят до мельчайших подробностей, в противном случае вы рискуете через какое-то время забыть структурные особенности этого фрагмента и в результате не сможете объединить его с другими частями проекта, что часто выполняется значительно позже и, возможно, другими программистами. Поэтому стиль программирования надо рассматривать как один из важных составных элементов проектирования программы, и с самого начала работы воспитывать в себе аккуратность и тщательность.

структура  
данных

Даже при реализации очень крупных проектов трудности могут возникнуть не из-за неспособности найти решение, а, наоборот, из-за того, что решить поставленную задачу можно самыми разными методами с применением различных алгоритмов, и трудно бывает оценить, какой из них является наилучшим, а какой, возможно, приведет к программистским проблемам или окажется безнадежно неэффективным. При этом наибольший вклад в разнообразие возможных алгоритмов вносит способ организации данных, используемых программой:

- как данные организованы по отношению друг к другу;
- какие данные хранятся в памяти;

- какие данные вычисляются по мере необходимости;
- какие данных хранятся в файлах, и как эти файлы организованы.

Таким образом, вторая цель этой книги заключается в представлении нескольких элегантных, хотя и фундаментально простых идей, используемых при организации и обработке данных. Прежде всего мы рассмотрим такие методы организации данных, как списки, стеки и очереди. Позже мы разработаем несколько мощных алгоритмов обработки данных, именно, средства сортировки и поиска данных.

анализ

Поскольку существуют различные способы организации данных и большое разнообразие алгоритмов, становится важной разработка критериев для выбора наилучшего метода. Поэтому мы обратим особое внимание на анализ поведения алгоритмов при различных условиях.

тестирование  
и верификация

Трудности, возникающие при отладке программы, возрастают значительно быстрее ее размера. Например, если некоторая программа в два раза длиннее другой, ее отладка, скорее всего, займет не в два раза больше времени, а в три или четыре. Многие очень большие программы (например, операционные системы) сдаются в эксплуатацию, несмотря на то, что они все еще содержат ошибки, найти которые программистам так и не удалось из-за непреодолимой сложности процесса отладки. Бывает так, что проекты, на разработку которых ушли годы труда, приходится отбрасывать, так как не представляется возможным понять, почему они не функционируют. Если мы не хотим, чтобы наши проекты постигла такая же судьба, мы должны использовать методы, которые:

правильность  
программы

- уменьшат количество ошибок, что облегчит поиск оставшихся;
- позволят нам заблаговременно удостовериться в правильности наших алгоритмов;
- предоставят нам способы тестирования наших программ, чтобы мы могли быть более или менее уверены, что программы будут работать правильно.

Разработка таких методов является еще одной нашей целью, которая, впрочем, пока еще в полной мере будет нам недоступна.

поддержка

Неформальные исследования показывают, что к тому моменту, когда сложная и важная программа полностью отлажена и сдана в практическую эксплуатацию, оказывается израсходованной менее половины программистского труда, который придется затратить для полного завершения проекта. *Поддержка* программ, другими словами, модификации, необходимые для удовлетворения возникающих требований пользователей или новых операционных сред, отнимают более половины программистских усилий. По этой причине важно разрабатывать большие проекты таким образом, чтобы их было легко понимать и модифицировать.

Pascal

Язык программирования Pascal обладает рядом черт, оправдывающих его выбор для выражения алгоритмов, которые мы будем разрабатывать. Pascal был специально разработан так, чтобы способствовать написанию тщательно структурированных программ, реализующих принципы программного проектирования. Pascal, если сравнивать его с большинством языков высокого уровня, содержит относительно мало

программных средств, поэтому его легко освоить, однако в то же время он обладает мощными возможностями обработки данных, которые облегчают переход от общих алгоритмов к конкретным программам их реализации.

Несколько разделов этой и последующих глав неформально вводят некоторые средства языка Pascal по мере того, как они будут встречаться в составляемых нами программах. Строгое описание синтаксиса (грамматики) языка Pascal можно найти в приложении D или в учебниках по программированию на этом языке.

## 1.2. Игра «Жизнь»

Возьму на себя смелость перефразировать старую поговорку:

*Одна конкретная задача стоит тысячи нереализованных абстракций.*

конкретное  
исследование

В этой и последующей главах мы сконцентрируемся на конкретном исследовании, которое, будучи не очень крупным по критериям реальной жизни, позволит тем не менее проиллюстрировать и методы проектирования программ, и подводные камни, которых нам следует научиться избегать. Иногда пример подвигнет нас на формулировку общих принципов; в других случаях мы начнем, наоборот, с дискуссии на общие темы; однако всегда конечной целью будет обнаружение общих методов, ценность которых будет проявляться в определенном диапазоне практических применений. В дальнейших главах мы применим тот же подход при рассмотрении более крупных проектов. Здесь же мы воспользуемся в качестве примера игрой, получившей название «Жизнь», которая была предложена английским математиком Дж. Х. Конвеем (J. H. Conway) в 1970 г.

### 1.2.1. Правила игры «Жизнь»

описания

Программа представляет собой модель жизни некоторого сообщества, а не игру нескольких игроков. Действия разворачиваются в неограниченном пространстве решетки, или сетки, в которой каждая ячейка может быть либо занята организмом, либо нет. Занятые ячейки будут называться *живыми*; незанятые — *мертвыми*. Расположение и количество живых ячеек изменяется от поколения к поколению в соответствии с количеством соседних живых ячеек следующим образом:

правила  
переходов

1. Соседями данной ячейки являются восемь ячеек, соприкасающихся с ней по горизонтали, вертикали или диагонали.
2. Если ячейка является живой, но либо имеет лишь одну живую ячейку-соседа, либо не имеет живых соседей вообще, в следующем поколении она умирает от одиночества.
3. Если ячейка является живой и имеет четырех или более живых соседей, в следующем поколении она умирает от перенаселенности.
4. Живая ячейка, имеющая двух или трех живых соседей, остается живой в следующем поколении.

5. Если ячейка мертва, то в следующем поколении она станет живой, если она имеет точно трех (не больше и не меньше) живых соседей. Все остальные мертвые ячейки остаются мертвыми в следующем поколении.
6. Все рождения и смерти имеют место точно в одни и те же моменты, поэтому умирающая ячейка может помочь родиться другой, но не может предотвратить смерть других ячеек из-за уменьшения перенаселенности; рождающаяся ячейка не может предотвратить смерть или, наоборот, убить ячейки, живые в предыдущем поколении.

### 1.2.2. Примеры

В качестве первого примера рассмотрим сообщество

		•	•		

Число живых соседей для конкретных ячеек выглядит следующим образом:

0	0	0	0	0	0
0	1	2	2	1	0
0	1	•1	•1	1	0
0	1	2	2	1	0
0	0	0	0	0	0

пример гибели

В соответствии с правилом 2 обе живые ячейки умрут в следующем поколении, а по правилу 5 ни одна ячейка не оживет; таким образом, в данном примере сообщество вымирает.

С другой стороны, сообщество

0	0	0	0	0	0
0	1	2	2	1	0
0	2	•3	•3	2	0
0	2	•3	•3	2	0
0	1	2	2	1	0
0	0	0	0	0	0

стабильность

имеет, как это показано на рисунке, другое распределение числа живых соседей. Каждая живая ячейка имеет трех живых соседей и в силу этого остается живой, но все мертвые ячейки имеют в качестве соседей две или меньше живых ячеек, в силу чего остаются мертвыми.

Два сообщества

0	0	0	0	0
1	2	3	2	1
1	•1	•2	•1	1
1	2	3	2	1
0	0	0	0	0

и

0	1	1	1	0
0	2	•1	2	0
0	3	•2	3	0
0	2	•1	2	0
0	1	1	1	0

изменения

будут изменяться от поколения к поколению, в чем можно убедиться, изучив распределение живых соседей для отдельных ячеек.

многообразие

Любопытно, что из очень простых начальных конфигураций могут развиваться весьма сложные живые сообщества, существующие на протяжении многих поколений, причем обычно очень не просто предугадать, какие изменения возникнут в следующем поколении. Некоторые очень маленькие начальные конфигурации развиваются в большие сообщества; другие медленно вымирают; многие достигают стадий, когда они перестают изменяться или проходят через повторяющиеся конфигурации каждые несколько поколений.

популярность

Вскоре после изобретения этой модели Мартин Гарднер (Martin Gardner) начал обсуждать игру «Жизнь» на своей колонке в журнале *Scientific American*, и с тех пор она очаровывала многих людей, отчего в течение нескольких лет даже существовал ежеквартальный информационный бюллетень, посвященный этой игре. Она идеально подходит для демонстрации на домашних компьютерах.

Разумеется, нашей первой задачей будет написание программы, которая покажет, как исходное сообщество будет изменяться от поколения к поколению.

1.2.3. Решение

метод

Несколько минут размышлений покажут, что решение задачи «Жизнь» настолько простое, что оно вполне может служить упражнением для членов кружка начинающих программистов, которые только что освоили понятие массива. Все, что нам нужно — это объявить большой прямоугольный массив<sup>1</sup>, члены которого соответствуют ячейкам «Жизни», и которые должны обладать состоянием, т. е. помечаться либо как живые, либо как мертвые. Тогда для определения того, что произойдет в следующем поколении, мы должны просто посчитать число живых ячеек в окрестностях каждой ячейки и применить описанные выше правила. Поскольку, однако, для продвижения по прямоугольному массиву мы будем использовать циклы, нам важно не нарушить правило 6, т. е. не позволить произведенным уже изменениям повлиять на распределение числа живых соседей для ячеек, обрабатываемых позже. Простейший способ избежать этой ловушки заключается в объявлении второго прямоуголь-

<sup>1</sup> Массив с двумя индексами называется **прямоугольным**. Первый индекс определяет **строку** массива, а второй — его **столбец**.



ного массива, который будет отображать сообщество в следующем поколении и после полного завершения вычисления этого поколения будет копироваться в исходный массив.

Теперь давайте изобразим на бумаге этот метод в виде неформального алгоритма.

алгоритм

Инициализируем прямоугольный массив, который мы назовем `map`, и который будет содержать начальную конфигурацию живых ячеек.

Будем повторять следующие шаги произвольное число раз:

Для каждой ячейки в прямоугольном массиве выполним следующее:

Подсчитаем число живых соседей данной ячейки.

Если число живых соседей равно 0, 1, 4, 5, 6, 7 или 8, то установим соответствующую ячейку в другом прямоугольном массиве, названном `newmap`, в живое состояние; если число живых соседей равно 2, установим соответствующую ячейку в `newmap` в то же состояние, что и в прямоугольном массиве `map` (поскольку состояние ячейки с числом живых соседей 2 не изменяется).

Скопируем прямоугольный массив `newmap` в прямоугольный массив `map`.

Выведем прямоугольный массив `map` на экран пользователя.

### 1.2.4. Life: главная программа

Приведенная выше схема алгоритма для игры «Жизнь» приводит к такой Pascal-программе, которую мы будем называть `Life` (жизнь), а в этом, первом, варианте — `Life1`.

главная  
программа

```

program Life1 (input,output);                                { Игра "Жизнь", вариант 1 }
{ Pre: Пользователь должен предоставить начальную конфигурацию
      живых ячеек.
  Post: Программа выводит последовательность решеток,
          отображающих изменения в конфигурации живых ячеек
          согласно правилам игры "Жизнь".
  Uses: Использует функции UserSaysYes, NeighborCount и процедуры
          WriteMap, Initialize }
uses Utility;                                                { модуль, содержащий процедуры-утилиты,
                                                                специфичные для Turbo Pascal }
const                                { maxrow и maxcol определяются размерами экрана }
  maxrow = 20;                          { максимально допустимое число строк }
  maxcol = 60;                          { максимально допустимое число столбцов }
type
  rowrange = 0 .. (maxrow + 1);
  colrange = 0 .. (maxcol + 1);
  { Эти диапазоны увеличены на 1, чтобы представить место
    для границы решетки. Необходимость этого действия будет
    пояснена в разделе 1.4.2. }
  state = (dead, alive);                                { состояние ячейки }
  grid = array [rowrange, colrange] of state;          { прямоугольный массив }
var
  map,                                { конфигурация текущего поколения }
  newmap: grid;                        { конфигурация следующего поколения }

```



```

row: rowrange;
col: colrange;
{ Объявления процедур и функций размещаются здесь. }

begin                                     { главная программа Life1 }
  Initialize(map, newmap);                { получить начальную конфигурацию }
  WriteMap(map);                          { вывод начальной конфигурации для контроля }
  writeln("Это начальная выбранная вами конфигурация.");
  writeln("Для продолжения нажмите любую клавишу.");
  readln;
  repeat                                  { начнем главный цикл по поколениям }
    for row := 1 to maxrow do             { цикл по всему массиву }
      for col := 1 to maxcol do
        case NeighborCount(map, row, col) of
          0, 1: newemap [row, col] := dead; { умирает от одиночества }
          2:   newemap [row, col] := map [row, col]; { остается тем же }
          3:   newemap [row, col] := alive;      { оживает }
          4 .. 8: newemap [row, col] := dead
                                     { умирает от перенаселенности }
        end;
      map := newmap;
      WriteMap(map);
      write("Хотите продолжать наблюдение следующих поколений?");
    until not UserSaysYes
      { определим, хочет ли пользователь продолжать }
  end.                                     { главная программа Life1 }

```

спецификация  
программы

Документация этой программы начинается с ее **спецификаций**, другими словами, точных предложений с условиями, которые должны выполняться, когда программа начинает свою работу, и условиями, выполняемыми после ее завершения. Эти условия называются соответственно **предусловиями (Pre)** и **постусловиями (Post)** для данной программы. Включение точных пред- и постусловий для каждой подпрограммы не только поясняет назначение подпрограммы, но и помогает избежать ошибок при взаимодействии подпрограмм (как говорят, ошибок межпрограммного интерфейса). Включение спецификаций настолько полезно, что мы выделяем это действие в качестве нашего первого программистского принципа:

### **Программистский принцип**

*В каждую написанную вами программу, процедуру и функцию включайте точные пред- и постусловия*

подпрограммы

Третья часть спецификаций для нашей программы представляет собой список подпрограмм, которые она будет использовать (**Uses**). Такой список также следует включать в каждую программу, процедуру или функцию.

Для программы Life нам еще надо написать процедуры и функции:

- процедура Initialize(map,newmap) инициализирует решетку и вводит исходную конфигурацию;
- процедура WriteMap(map) выполняет вывод;

функционирование  
программы

- функция `UserSaysYes` будет запрашивать пользователя, следует ли выполнить переход к следующему поколению;
- процедура `NeighborCount(map, row, col)` будет определять число населенных ячеек, соседних с ячейкой, находящейся в клетке `row,col` прямоугольного массива `map` (под населенными ячейками понимаются ячейки с живыми обитателями).

Функционирование программы `Life` осуществляется очевидным образом. Прежде всего мы считываем исходную ситуацию, чтобы установить первую конфигурацию занятых ячеек. Затем мы начинаем цикл, выполняющий один проход для каждого поколения. Внутри этого цикла мы сначала выполняем вложенную пару циклов вдоль строк `row` и столбцов `col`, которые просмотрят все ячейки в прямоугольном массиве `map`. Тело этих вложенных циклов состоит из предложения множественного выбора **case ... end**. В настоящем варианте функция `NeighborCount(map, row, col)` вернет одно из значений 0, 1, ..., 8, и для каждого из этих случаев (**case**) мы можем предусмотреть отдельное действие или, как в нашей программе, некоторые случаи могут приводить к одному и тому же действию. Вы должны удостовериться в том, что действие, предписанное для каждого случая, точно соответствует правилам 2, 3, 4 и 5 раздела 1.2.1. Наконец, после выполнения вложенных циклов и предложения **case**, которые заполняют значениями прямоугольный массив `newmap`, предложение присваивания `map := newmap` копирует его в прямоугольный массив `map`, и процедура `WriteMap(map)` выводит результат.

### Упражнения 1.2

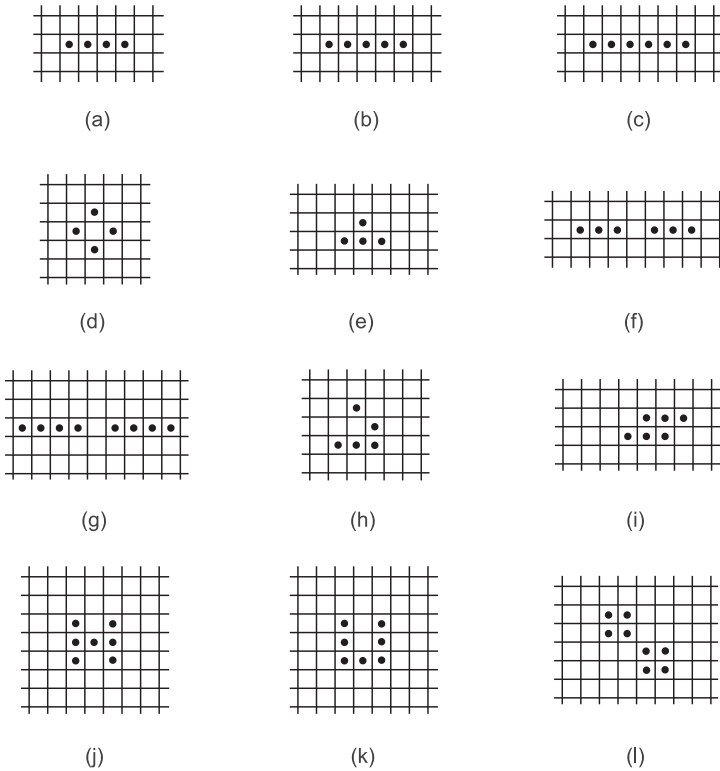
Определите путем вычислений вручную, что будет происходить с сообществами, изображенными на рис. 1.1, на протяжении пяти поколений. [Подсказка: расставьте конфигурацию программы `Life` на шашечной доске. Используйте для живых ячеек шашки одного цвета, а шашками второго цвета обозначайте ячейки, которые в следующем поколении родятся или умрут.]

## 1.3. Стиль программирования

Перед тем, как начать писать подпрограммы для игры `Life`, рассмотрим несколько принципов, которые надо будет обязательно использовать в программировании.

### 1.3.1. Имена

В легенде о создании мира (Книга Бытия 2:19) Всевышний привел всех животных к Адаму, чтобы тот дал им имена. Согласно древнему иудейскому верованию, животные ожили только после того, как Адам назвал их. Эта легенда позволяет сформулировать важный принцип компьютерного программирования: даже если данные или алгоритмы уже существуют, только после того, им будут даны значащие имена, можно будет определить их расположение в программе, и только после этого они начнут жить самостоятельной жизнью.



**Рис. 1.1.** Простые конфигурации для программы Life

цель наглядных  
наименований

Чтобы программа функционировала правильно, чрезвычайно важно точно знать, что представляет собой каждая переменная, и отчетливо представлять, что делает каждая подпрограмма. Поэтому в программу всегда необходимо включать документацию с описанием назначения переменных и подпрограмм. Имена переменных и подпрограмм следует выбирать со всей тщательностью, чтобы они ясно и точно отражали свое назначение. Найти разумные имена не всегда просто, однако это достаточно важная задача, чтобы выделить ее в качестве второго программистского принципа:

### ***Программистский принцип***

*Всегда именуйте свои переменные и подпрограммы с максимальным тщанием и детально объясняйте их назначение*

Язык Pascal даже навязывает нам выполнение этого принципа, требуя, чтобы в программе содержалась секция с объявлениями переменных. При этом Pascal, в отличие от большинства других языков программирования, допускает более интенсивное использование имен. Константам, используемым в различных местах программы, следует давать имена, и то же относится к типам данных, чтобы компьютер мог обнаружить ошибки, которые в противном случае найти было бы затруднительно.

рекомендации

При тщательном выборе наглядных имен программа становится яснее и понятнее, что способствует устранению опечаток и других обычных ошибок. Некоторые полезные правила выбора имен заключаются в следующем:

1. Обращайте особое внимание на выбор имен для процедур, функций, констант и всех глобальных переменных и типов, используемых в различных частях программы. Эти имена должны быть значащими и ясно отражать назначение подпрограмм, переменных и других программных объектов.
2. Для локальных переменных, используемых всего несколько раз, давайте простые имена. Математики обычно используют для именования переменных одну букву, и при написании математических программ вполне допустимо во многих случаях использовать для обозначения математических переменных однобуквенные имена. С другой стороны, даже для переменной, управляющей циклом **for**, обычно удастся найти короткое, но значащее слово, лучше описывающее назначение этой переменной.
3. Для обозначения объектов, относящихся к одной категории, используйте префиксы или суффиксы. Например, используемым в программе файлам можно дать такие имена:

InputFile TransactionFile TotalFile OutFile RejectFile

4. Избегайте умышленных нарушений орфографии и бессмысленных суффиксов для различения схожих по смыслу имен. Из всех приведенных ниже имен

index indx ndex indexx index2 index3

обычно следует использовать только одно (первое). Если у вас появляется искушение ввести в программу много схожих имен такого рода, примите это желание за напоминание о своей лености и постарайтесь придумать имена, лучше описывающее их назначение.

5. Избегайте смешных имен, смысл которых имеет мало общего с решаемой задачей. Предложения

```
while TV in hock do study;
if not sleepy then play else nap;
[Текст этого фрагмента с вольным переводом имен:
while телевизор in ломбард do учеба;
if not сонный then играй else спи;]
```

возможно, смешны, но они иллюстрируют плохой стиль программирования!

6. Избегайте использования имен, близких друг к другу по написанию, или которые легко спутать по другим причинам.
7. С осторожностью используйте символы «l» (строчная латинская буква L), «O» (прописная O) и «0» (ноль). Используемые внутри слов или чисел, эти символы обычно легко распознаются по контексту и не вызывают проблем, но знаки «l» и «O» никогда не следует использовать в качестве законченных имен. Взгляните на такие примеры:

l:=1; x:=1; x:=l; x:=O; O:=0

### 1.3.2. Документация и форматы

назначение  
документации

Многие студенты полагают, что документацией следует заниматься лишь после завершения работы над программой, и то лишь для того, чтобы преподаватель мог ознакомиться с программой, и чтобы удачные находки студента не остались бы незамеченными. Разумеется, автор небольшой программы может держать все детали в уме, и документация ему нужна лишь для того, чтобы объяснить роль программы кому-либо другому. Однако в случае больших программ (да и малых по истечении нескольких месяцев) становится невозможным запомнить, как каждая деталь программы соотносится с другой деталью, и поэтому при написании больших программ существенно включать в каждый фрагмент программы соответствующую документацию. Полезной привычкой является составление документации по мере разработки программы, и еще лучшей, как это будет показано позже — подготовка части документации еще до того, как программист приступит к написанию текста программы.

Не любые пояснения годятся в качестве документации. Почти так же часто, как программы с малым объемом документации или с пометками, понятными только для их автора, встречаются программы с многословными пояснениями, которые в действительности мало дают для понимания программы. Отсюда следует третий программистский принцип:

**Программистский принцип**  
*Старайтесь, чтобы ваша документация была  
краткой, но содержательной*

Стиль документации, как и вообще стиль любого письменного документа, зависит от личности автора, и самые разные стили могут оказаться вполне удовлетворительными. Тем не менее, имеются общепринятые рекомендации, которых желательно придерживаться:

рекомендации

1. В начало каждой подпрограммы помещайте пролог, который включает в себя
  - (a) Идентификационные данные (имя программиста, дату, номер версии)<sup>2</sup>.
  - (b) Назначение программы и используемый в ней метод.
  - (c) Результаты работы подпрограммы и используемые ею данные.
  - (d) Ссылки на другие виды документации, внешние по отношению к программе.
2. При объявлении каждой переменной, константы или типа объясняйте, что собой представляют эти объекты и как они будут использоваться. Еще лучше, если эти сведения можно извлечь из имени объекта.
3. Перед каждой существенной секцией (блоком или подпрограммой) программы включайте краткий комментарий со сведениями о назначении или действии данной секции.
4. Отмечайте конец каждой существенной секции, если это не очевидно.

<sup>2</sup>

Ради экономии места программы, приведенные в этой книге, не содержат идентификационных строк или других частей пролога, поскольку необходимую информацию можно найти в окружающем тексте.

5. Избегайте комментариев, которые просто повторяют действие кода, как в этом примере

```
count:=count+1; { Увеличение count на 1 }
```

или бессмысленного жаргона, например

```
{ horse string length into correctitude }
```

(Пример этого невразумительного комментария взят из некоторой системной программы.)

6. Поясняйте все предложения, в которых используются программные трюки или назначение которых неясно. Еще лучше избегать таких предложений.
7. Из кода программы должно быть ясно, *как* программа работает. Документация же должна пояснять, *зачем* программа создана и *что* она делает.
8. Если программа как-либо модифицируется, не забывайте вносить соответствующие коррективы в документацию.

формат

Пробелы, пустые строки и отступы в программе являются важной частью документации. Они облегчают чтение программы, позволяют с первого взгляда определить, какие части программы связаны друг с другом, разделяют крупные программные блоки, позволяют показать, какие предложения содержатся в каждом цикле или выделить альтернативы условных предложений. Существуют различные системы (в том числе автоматизированные) включения в программу отступов и пробелов, основная цель которых состоит в том, чтобы помочь читателю определить структуру программы.

утилиты  
prettyprinter

Системная утилита *prettyprinter* читает Pascal-программу, перемещая текст между строками и настраивая отступы так, чтобы улучшить внешний вид программы и сделать ее структуру более очевидной. Если эта утилита имеется в вашей системе, можете поэкспериментировать с ней, чтобы посмотреть, улучшит ли она вашу программу.

единообразие

Поскольку формат программы является ее важным элементом, вы должны остановиться на каких-либо разумных правилах использования пробелов и отступов, и единообразно использовать эти правила во всех создаваемых вами программах. Многие группы профессиональных программистов принимают для себя единую систему и настаивают, чтобы все программы были написаны единообразно. Так же поступают многие студенческие группы и бригады. При такой системе любой программист будет с большей легкостью читать и понимать работу своих товарищей.

#### **Программистский принцип**

*Чтение программы занимает гораздо больше времени, чем ее написание. Старайтесь облегчить чтение программы.*

### **1.3.3. Детализация программы и модульность**

решение  
задачи

Компьютеры не решают задачи; это делают люди. Обычно наиболее важной частью этого процесса является разделение всей задачи на задачи меньшего размера, которые уже можно разработать детально. Если эти

разделение  
на меньшие блоки

задачи все еще оказываются слишком сложными, можно продолжить их разделение и т. д. В любой большой организации высшее руководство не может беспокоиться о всех деталях каждой операции; высшее руководство должно заниматься глобальными целями и проблемами и возлагать ответственность за конкретные участки работы на своих подчиненных. Однако и управляющие среднего звена не могут все делать сами: они должны разделить работу на блоки и поручить выполнение этих блоков другим людям. Точно так же дело обстоит и в программировании. Даже если проект достаточно прост, чтобы его мог выполнить один человек от начала до конца, чрезвычайно важно структурировать работу, начав с глобального осознания существования задачи, и, после разделения ее на подзадачи, приступить к последовательному решению каждой из них, не забывая об остальных.

Определим этот принцип словами классической поговорки:

**Программистский принцип**  
*Остерегайтесь за деревьями не увидеть леса.*

нисходящая  
детализация

Принцип, носящий название нисходящей детализации или детализации сверху вниз, является ключевым правилом разработки больших программ. Этот принцип предполагает откладывание на некоторое время детальной проработки, не допуская при этом отказа от точности и строгости. Принцип нисходящей детализации совсем не означает, что главная программа становится некоторой туманной сущностью без ясно определенного назначения. Наоборот, главная программа передает почти всю работу различным подпрограммам (процедурам или функциям), и по мере того, как мы пишем главную программу (что надо делать в первую очередь), мы *точно* определяем, как эта работа будет распределена между ними. Позже, когда мы перейдем к написанию конкретной подпрограммы, мы еще до начала работы будем знать, что именно эта подпрограмма должна делать.

спецификации

Часто распределение работы между подпрограммами оказывается не очень простой задачей, и иной раз принятое решение в дальнейшем приходится пересматривать. Тем не менее, два правила могут помочь в распределении работы:

**Программистский принцип**  
*Каждая подпрограмма должна выполнять только одну задачу, но делать это хорошо.*

Другими словами, мы должны уметь кратко определить назначение подпрограммы. Если оказывается, что для задания предусловий или постусловий вы вынуждены составлять длинный текст, это значит, что либо вы вдаетесь в излишние детали (т. е. вы пишете подпрограмму, когда для этого еще не наступило время), либо вы неудачно распределили работу и надо заняться ее перераспределением. Сама подпрограмма будет безусловно содержать массу деталей, но они должны проявиться лишь на следующей стадии разработки.



**Программистский принцип**

*Каждая подпрограмма должна что-нибудь скрывать.*

Управляющий среднего звена в большой компании отнюдь не передает всю получаемую им от своих отделов информацию вышестоящему начальству; он обобщает, сопоставляет и сортирует информацию, ряд запросов обрабатывает сам, а начальству посылает только то, что нужно на более высоких ступенях руководства. Точно так же он не передает все получаемые им от начальства инструкции своим подчиненным. Каждому своему сотруднику он передает только то, в чем тот нуждается для выполнения своей работы. Наши подпрограммы должны вести себя таким же образом.

Одной из важнейших составляющих процесса разработки является определение задач, решаемых каждой подпрограммой, с точным указанием ее предусловий и постусловий или, другими словами, определение ее входных данных и получаемого от нее результата. Ошибки, допускаемые в этих спецификациях, чаще всего служат причиной скрытых дефектов в программе и их же труднее всего вылавливать. Прежде всего, данные, используемые подпрограммой, должны быть точно определены. Эти данные могут быть пяти разных видов:

- **Входные параметры** используются подпрограммой, но не изменяются ею. В языке Pascal входные параметры обычно являются параметрами-значениями. (Исключения: файлы должны быть всегда **var**-параметрами, даже если они используются только для ввода; часто и большие массивы передаются, как **var**-параметры, чтобы избежать затрат времени и памяти, требуемых для создания локальной копии.)
- **Выходные параметры** содержат результаты вычислений, выполняемых подпрограммой. В языке Pascal выходные параметры должны быть **var**-параметрами.
- **Параметры входа-выхода** используются и для ввода, и для вывода; исходное значение такого параметра используется подпрограммой и затем ею модифицируется. В языке Pascal выходные параметры входа-выхода должны быть **var**-параметрами.

параметры

**Замечание**

Программисты, работающие на языке Pascal, часто отождествляют входные параметры с параметрами-значениями, а выходные параметры и параметры входа-выхода с **var**-параметрами. Хотя выходные параметры и параметры входа-выхода *должны быть* **var**-параметрами, входные параметры не обязательно представляют собой значения. Фактически, чтобы избежать затрат на создание локальных копий, мы обычно рассматриваем параметры, содержащие большие записи или массивы, как **var**-параметры, хотя они будут использоваться только для ввода.

переменные

- **Локальные переменные** объявляются в подпрограмме и существуют только пока эта подпрограмма выполняется. Они не инициализируются до вызова подпрограммы и отбрасываются, когда подпрограмма завершается.



побочные  
эффекты

- **Глобальные переменные** используются в подпрограмме, но не объявляются в ней. Использование в подпрограмме глобальных переменных может привести к серьезным неприятностям, поскольку после того, как подпрограмма была написана, автор может забыть, какие именно глобальные переменные им использовались и с какой целью. Если главная программа в дальнейшем претерпит изменения, подпрограмма может таинственным образом начать работать неправильно. Если подпрограмма изменяет значение глобальной переменной, говорят, что она производит **побочный эффект**. Побочные эффекты даже еще более опасны, чем использование глобальных переменных в качестве входных параметров подпрограммы, потому что побочные эффекты могут привести к изменению поведения других подпрограмм и направить внимание программиста, отлаживающего программу, на уже отлаженные и правильные участки программы.

#### **Программистский принцип**

*Организируйте возможно более простые связи. Избегайте, если это только возможно, использования глобальных переменных.*

#### **Программистский принцип**

*Никогда не создавайте побочные эффекты, если их можно избежать. Если вам приходится использовать в качестве входных параметров глобальные переменные, тщательно документируйте их.*

Для функций определение **побочного эффекта** расширяется и включает в себя изменения как глобальных переменных, так и параметров. Функция должна вычислять лишь один результат, возвращаемый как значение этой функции. Если от подпрограммы требуется получить более одного результата, она должна быть оформлена как процедура, а не как функция.

Хотя все эти принципы нисходящего программирования могут показаться почти очевидными, единственным способом их усвоения является практика. Поэтому на протяжении этой книги мы будем скрупулезно применять их к большим программам, которые мы будем писать, а чуть позже мы вернемся к нашему первому программному проекту.

### **Упражнения 1.3**

**E1.** Пусть даны объявления

```
var A: array [1 .. n, 1 .. n] of integer;  
    i, j: integer;
```

где  $n$  является константой. Определите, что делают приведенные ниже предложения, и перепишите эти предложения, чтобы они давали тот же результат, но выглядели не так запутанно.

```
for i := 1 to n do  
  for j := 1 to n do  
    A[i, j] := (i div j) * (j div i);
```

**Е2.** Перепишите приведенные ниже предложения, чтобы они давали тот же результат, но выглядели не так запутанно.

```
procedure DoesSomething(var first, second: integer) ;{ Делает что-то }
begin
    first := second – first;
    second := second – first;
    first := second + first
end;
```

**Е3.** Определите, что делает каждая из приведенных ниже процедур. Перепишите каждую процедуру, используя разумные имена переменных и лучший формат и удалите переменные и предложения, без которых можно обойтись.

(a) **function** Calculate(apple, orange: integer): integer; { *Вычислить* }  
**var** peach, lemon: integer;  
**begin** peach := 0; lemon := 0; **if** apple < orange **then**  
**begin** peach := orange **end else if** orange <= apple **then**  
**begin** peach := apple **end else begin** peach := maxint;  
 lemon := maxint **end; if** lemon <> maxint **then**  
**begin** Calculate := peach **end end**;

(b) Для этого упражнения считайте, что имело место объявление  
**type** vector = **array** [1 .. max] **of** real. { *Фигура* }  
**function** Figure(**var** vector1: vector): real;  
**var** loop1: integer; loop2: real; loop3: real; loop4: integer;  
**begin** loop1 := 1; loop2 := vector1 [loop1]; loop3 := 0.0;  
 loop4 := loop1; **for** loop4 := 1 **to** max **do begin** loop1 := loop1 + 1;  
 loop2 := vector1 [loop1 – 1];  
 loop3 := loop2 + loop3 **end; loop1** := loop1 – 1; loop2 := loop1;  
 loop2 := loop3/loop2; Figure := loop2 **end**;

(c) **procedure** question(**var** a17: integer; **var** stuff: integer); { *Вопрос* }  
**var** another, yetanother, stillonemore: integer;  
**begin** another := yetanother; stillonemore := a17;  
 yetanother := stuff; another := stillonemore;  
 a17 := yetanother; stillonemore := yetanother;  
 stuff := another; another := yetanother;  
 yetanother := stuff; **end**;

(d) **function** mystery(apple, orange, peach: integer): integer; { *Тайна* }  
**begin if** apple > orange **then if** apple > peach **then if**  
 peach > orange **then** mystery := peach **else if** apple < orange **then**  
 mystery := apple **else** mystery := orange **else** mystery := apple **else**  
**if** peach > apple **then if** peach > orange **then** mystery := orange **else**  
 mystery := peach **else** mystery := apple **end**;

**Е4.** Приведенное ниже предложение предназначено для анализа относительных значений трех целочисленных переменных, которые отличаются друг от друга.

```
if x < z then if x < y then if y < z then c:= 1 else c:= 2 else  

if y < z then c:= 3 else c:= 4 else if x < y then  

if x < z then c:= 5 else c:= 6 else if y < z then c:= 7 else  

if z < x then if z < y then c:= 8 else c:= 9 else c:= 10;
```

(a) Перепишите это предложение в более понятной форме.

(b) Поскольку три переменные могут упорядочиваться лишь шестью возможными способами, из десяти рассмотренных выше случаев реально могут иметь место только шесть. Найдите случаи, которые никогда не реализуются, и устраните из предложения лишние проверки.

(c) Напишите более простое и более короткое предложение, дающее тот же результат.

**E5.** Приведенная ниже Pascal-функция вычисляет кубический корень из действительного числа (с помощью аппроксимации Ньютона), используя то обстоятельство, что если  $y$  является одной аппроксимацией кубического корня из  $x$ , тогда

$$z = \frac{2y + x/y^2}{3}$$

является лучшей аппроксимацией.

```
function Fcn(stuff: real): real;
var April, Tim, Tiny, Shadow, Tom, Tam, Square: real; flag: Boolean;
begin Tim := stuff; Tam := stuff; Tiny := 0.00001;
  if stuff <> 0 then repeat Shadow := Tim + Tim;
    Square := Tim * Tim;
    Tom := (Shadow + stuff / Square);
    April := Tom / 3.0;
    if April * April * April - Tam > - Tiny then
      if April * April * April - Tam < Tiny then flag := true
    else flag := false else flag := false;
    if flag = false then Tim := April else Tim := Tam until flag = true;
  if stuff = 0 then Fcn := stuff else Fcn := April end;
```

(a) Перепишите эту функцию, предусмотрев разумные имена переменных, удалив лишние переменные, не помогающие пониманию смысла функции, улучшив взаимное размещение строк программы и обойдясь без лишних и бессмысленных предложений.

(b) Напишите функцию для вычисления кубического корня из  $x$  посредством математической формулы, начав с присваивания  $y := x$  и затем повторяя операцию

$$y := (2 * y + (x / \text{sqr}(y))) / 3$$

до тех пор, пока не начнет выполняться условие

$$\text{abs}(y * y * y - x) < 0.00001.$$

(c) Какая из поставленных выше задач проще?

**E6.** *Средним значением* последовательности действительных чисел называется их сумма, деленная на количество чисел в последовательности. *Дисперсией* последовательности называется среднее значение квадрата всех чисел в последовательности минус квадрат среднего значения чисел в той же последовательности. *Стандартным отклонением* называется квадратный корень из дисперсии. Напишите хорошо структурированную Pascal-программу, вычисляющую стандартное отклонение последовательности из  $n$  чисел, где  $n$  есть константа, а числа хранятся в массиве с индексами от 1 до  $n$ , причем  $n$  выступает в качестве параметра функции. Для вычисления среднего значения и дисперсии сначала используйте, а затем напишите вспомогательные функции.

вывод графика

- Е7. Разработайте программу, которая выводит график, состоящий из заданного числа точек. Входным для программы будет текстовый файл, каждая строка которого содержит два числа, представляющие собой  $x$ - и  $y$ -координаты отображаемой на графике точки. Детализация программы приводит нас к необходимости включения в нее процедуры вывода точек, которая не может быть реализована, поскольку в нее должны входить неизвестные нам особенности оборудования, рисующего график на бумаге. Для аккуратного отображения графика программа должна знать максимальные и минимальные значения чисел  $x$  и  $y$ , содержащихся во входном файле. Таким образом, в программу должна входить еще одна процедура *Bounds*, которая читает весь файл и определяет эти четыре граничные значения. Далее другая процедура используется для рисования осей и надписей на них; наконец, файл читается заново и на график выводятся последовательные точки.
- (a) Напишите главную программу, не включая в нее процедуры.
  - (b) Напишите процедуру *Bounds*.
  - (c) Сформулируйте пред- и постусловия для оставшихся процедур вместе с соответствующей документацией, описывающей назначение этих процедур и требования к ним.

## 1.4. Кодирование, тестирование и дальнейшая детализация

Три процесса, обозначенные в заголовке этого раздела, тесно связаны друг с другом и должны выполняться совместно. Однако мысленно мы должны их разделять, так как каждый требует своего подхода и метода. **Кодирование**, разумеется, — это процесс написания алгоритма с использованием правильного синтаксиса (грамматики) языка программирования, например, языка Pascal, а **тестирование** — это процесс выполнения программы с тестовыми данными, выбранными так, чтобы обнаружить в программе возможные ошибки. При дальнейшей **детализации** программы мы обращаемся к еще ненаписанным программам и повторяем эти шаги.

### 1.4.1. Заглушки

отладка и  
тестирование  
по ходу  
разработки

Завершив кодирование главной программы, большинство программистов захочет сразу приступить к разработке и кодированию подпрограмм, чтобы посмотреть, как будет работать весь проект. Для таких небольших проектов, как игра «Жизнь», такой подход вполне применим, но для больших проектов разработка и кодирование всех подпрограмм может потребовать столько времени, что к моменту окончания этой работы многие детали главной программы и написанных ранее подпрограмм могут позабыться. К тому же, отдельные подпрограммы могли разрабатываться разными людьми, а кто-то из начинавших проект мог покинуть команду еще до того, как завершится написание всех подпрограмм. Зна-

чительно проще отлаживать программу, пока она еще свежа в вашей памяти. Поэтому для больших проектов гораздо эффективнее отлаживать и тестировать каждую подпрограмму сразу же после ее написания, а не ждать, пока будет закодирован весь проект.

Даже для небольших проектов имеются веские основания отлаживать подпрограммы по мере их написания. Например, мы можем сомневаться в некоторых деталях синтаксиса языка Pascal, которые будут использоваться в нескольких местах программы. Если мы сможем компилировать каждую подпрограмму в отдельности, то эти сомнения будут быстро устранены, и мы избежим ошибок при разработке следующих подпрограмм. В качестве другого примера предположим, что мы приняли определенный порядок выполнения основных действий программы. Если мы протестируем главную программу сразу же после ее написания, мы, возможно, обнаружим, что в каких-то случаях эти действия выполняются в неправильном порядке, и сможем исправить свою ошибку с большей легкостью, чем если бы мы ждали полного окончания разработки, когда основные действия программы могут стать не столь очевидными из-за массы содержащихся в них деталей.

Для того чтобы программу можно было откомпилировать, на месте каждой используемой, но еще не написанной подпрограммы должно что-то быть, и, следовательно, мы должны включить в программу короткие фиктивные подпрограммы, называемые *заглушками*. Простейшие заглушки ничего не делают:

```
procedure Initialize(var map, newmap: grid); begin end; { Инициализировать }
procedure WriteMap(map: grid); begin end           { Вывести конфигурацию }
function NeighborCount(map: grid; row: rowrange;   { Число соседей }
                      col: colrange): integer; begin end;
```

Даже с такими заглушками мы сможем по меньшей мере откомпилировать программу и убедиться в синтаксической правильности объявления типов и переменных. Обычно, однако, каждая заглушка выводит сообщение о вызове соответствующей подпрограммы. При выполнении программы мы можем обнаружить, что некоторые переменные используются без их инициализации, и чтобы устранить эти ошибки, мы можем добавить код в процедуру Initialize. В результате заглушка будет постепенно увеличиваться в размерах, пока не примет форму окончательной подпрограммы. Для такого простого проекта, как игра «Жизнь», мы можем просто писать подпрограмму за подпрограммой, подставлять их на место своих заглушек и смотреть, как это влияет на выполнение программы.

### 1.4.2. Подсчет соседей

функция  
NeighborCount

Приступим к дальнейшей обработке нашей программы. Функция, которая подсчитывает соседей ячейки [row, col], должна исследовать восемь соседних позиций. Для выполнения этой операции мы используем пару циклов **for**, один в пределах от row – 1 до row + 1, и другой от col – 1 до col + 1. При этом следует проявлять осторожность в тех случаях, когда ячейка [row, col] прилегает к границе решетки; в этих случаях мы должны исследовать только позиции, входящие в решетку. Чтобы не выйти за

ограда

границы решетки, можно использовать несколько предложений **if**, но мы поступим иначе: создадим вокруг решетки *ограду*. Для этого мы увеличим решетку, добавив две дополнительных строки, одну перед первой строкой решетки, а вторую после последней, и два дополнительных столбца, один до первого столбца решетки, а другой после последнего. Ячейки в столбцах и строках этой ограды всегда будут мертвы, поэтому они не повлияют на подсчет количества живых соседей. Их присутствие, однако, позволит циклам **for**, подсчитывающим соседей, не проводить различия между ячейками на границе решетки и любыми другими. См. примеры на рис. 1.2.

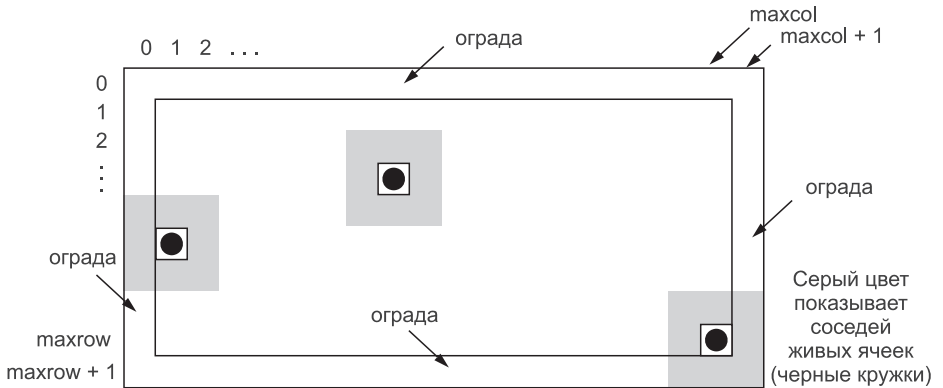


Рис. 1.2. Решетка игры «Жизнь» с оградой

сигнальная  
метка

Вместо слова «ограда» часто используется другой термин: *сигнальная метка*. Сигнальная метка — это дополнительный элемент, вводимый в структуру данных для того чтобы избежать специального рассмотрения граничных условий.

```
function NeighborCount (var map: grid; { Число соседей }
                        row: rowrange; col: colrange): integer;
{ Pre: Пара (row, col) является допустимой ячейкой в конфигурации Life.
  Post: Функция возвращает число живых соседей для данной ячейки. }
var
  nbrrow, { строка соседа ячейки (row, col) }
  nbrcol, { столбец соседа ячейки (row, col) }
  count: integer; { счетчик живых соседей }
begin { функция NeighborCount }
  count := 0;
  for nbrrow := (row - 1) to (row + 1) do { используем вложенные циклы }
    for nbrcol := (col - 1) to (col + 1) do { для подсчета соседей }
      if (map [nbrrow, nbrcol] = alive) then
        count := count + 1;
  if map [row, col] = alive then { саму ячейку не включаем в число соседей }
    count := count - 1;
  NeighborCount := count;
end; { функция NeighborCount }
```

### 1.4.3. Ввод и вывод

тщательно  
продумывайте  
ввод и вывод

Теперь нам осталось только написать процедуры Initialize, WriteMap и UserSaysYes, которые выполняют ввод и вывод. В компьютерных программах, предназначенных для использования многими людьми, процедуры, выполняющие ввод и вывод, часто оказываются самыми длинными. Ввод в программу должен тщательно проверяться, чтобы отбросить неправильные или несовместимые данные, причем ошибки ввода должны обрабатываться таким образом, чтобы избежать катастрофических последствий для программы или вывода смехотворных результатов. Выводимые данные должны быть тщательно организованы и отформатированы, при этом следует хорошо продумать, что следует выводить, и что не следует, а также предусмотреть различные альтернативные варианты вывода, удовлетворяющие различным возможным обстоятельствам. К сожалению, программные средства, используемые для организации ввода и вывода, различаются в разных компьютерных системах, и, к тому же, в них больше внимания уделяется деталям языка и другим частным проблемам, а не общим идеям. Поэтому работая только в рамках стандартного языка Pascal, невозможно включить в процедуры ввода-вывода всю необходимую проверку данных. При реализации программы в конкретной компьютерной системе в нее, как правило, включаются дополнительные средства проверки ошибок.

#### **Программистский принцип**

*Организуйте ввод и вывод в отдельных процедурах,  
чтобы их было легко изменить  
и подстроить под конкретную компьютерную систему.*

### 1. Соглашения языка Pascal

экскурс  
в историю

И входные, и выходные файлы в (исходном) стандартном языке Pascal являются абстракциями магнитных лент, и операции над ними выполняются только строго последовательно. Pascal также устанавливает буферы, которые иной раз при вводе и выводе взаимодействуют с терминалом довольно странным образом. В те времена, когда разрабатывался Pascal, практически не было интерактивных терминалов; ввод данных в программу обычно осуществлялся с магнитных лент или перфокарт. Тогда требовалось, чтобы система еще перед выполнением первого предложения программы получила первое данное из входного файла и разместила это данное в «файловом окне». Отсюда следует, что в исходном варианте языка Pascal нельзя было до ввода в программу первого данного с терминала вывести на экран запрашивающее ввод сообщение. Поскольку такое ограничение, очевидно, является неприемлемым, были предложены различные способы решения этой проблемы для различных систем, и, к сожалению, в языке Pascal не существует универсального метода организации интерактивного ввода-вывода.

В настоящей книге мы рассмотрим соглашения, которые являются на сегодня стандартными и которые правильно работают на многих, хотя, к



«ленивый»  
ввод

сожалению, не на всех системах. Одно из этих соглашений носит название «ленивого», или отсроченного ввода. В справочном руководстве по вашей системе Pascal должны описываться соглашения ввода-вывода, и, ознакомившись с этим руководством, вы сможете определить, какие изменения потребуются (и потребуются ли вообще), чтобы запустить программу Life на вашем компьютере. Мы будем использовать для получения входных данных стандартную процедуру `read`, а для ввода одной входной строки — процедуру `readln`. Мы также используем стандартную функцию `eof`, возвращающую булево значение, которое равно `true` в том и только в том случае, если встречается метка конца файла. При использовании отсроченного ввода система не делает попыток заполнить свое файловое окно до тех пор, пока в этом не возникнет необходимость. В результате не составляет труда вывести запрашивающее ввод сообщение (или выполнить любые другие действия) еще до первого ввода с терминала. Однако, когда мы выполняем проверку на конец файла, использование функции `eof` заставляет систему заполнить свое файловое окно. Таким образом, мы должны вывести запрос на ввод до проверки `eof`. В итоге чтение входных данных принимает такую общую форму:

```
write('Первый запрос');
while not eof do begin
  read(data);
  Process(data);
  write('Запрос на продолжение')
end;
```

## 2. Инициализация

метод ввода

Задача процедуры `Initialize` — установить в массиве `map` исходную конфигурацию. Для выполнения этой задачи мы можем рассмотреть каждую возможную координатную пару отдельно и запросить пользователя, следует ли заселить эту ячейку. Этот метод потребует от пользователя ввода

`maxrow * maxcol = 20 * 60 = 1200`

данных, что довольно утомительно. Вместо этого мы будем вводить только те координатные пары, которые соответствуют населенным ячейкам.

```
procedure Initialize(var map, newmap: grid); { Инициализация }
{ Pre: Предусловия отсутствуют.
  Post: Все ячейки в прямоугольных массивах map и newmap
        установлены в пустое состояние, и их ограды также сделаны
        пустыми. В прямоугольном массиве map установлена начальная
        конфигурация живых ячеек. }
```

инициализация

```
var
  row, { координаты ячейки }
  col: integer; { используем integer вместо поддиапазона
                для отлавливания ошибок пользователя }
begin { процедура Initialize }
  writeln('Эта программа представляет собой модель игры "Жизнь" ');
  writeln('Решетка имеет размер ', maxrow: 2, 'строк');
  writeln('Решетка имеет размер ', maxcol: 2, 'столбцов');
  for row := 0 to (maxrow + 1) do
    for col := 0 to (maxcol + 1) do
```



```

begin
  map [row, col] := dead;      { установим все ячейки, включая ограды,
                                в пустое состояние }

  newmap [row, col] := dead;

end;
writeln('На последовательных строках введите пары координат
        живых ячеек');
writeln('Завершите список специальной парой   0   0');
readln(row, col);
while (row <> 0) or (col <> 0) do      { проверка условия завершения }
begin                                { проверка ввода на допустимость }
  if (row >= 1) and (row <= maxrow) and (col >= 1) and (col <= maxcol)
  then
    map [row, col] := alive;
  else
    writeln('Значения выходят из допустимого диапазона');
    readln(row,col);
  end
end;                                { цикл обработки map[row, col] }
end;                                { процедура Initialize }

```

вывод

Для процедуры вывода WriteMap мы принимаем простой метод отображения всего прямоугольного массива в каждом поколении, причем населенные ячейки помечаются звездочками, а пустые — пробелами.

```

procedure WriteMap (map: grid);      { Вывод конфигурации }
{ Pre:   Прямоугольный массив map содержит текущую конфигурацию
        игры Life.
  Post: Текущая конфигурация Life выводится на экран пользователя. }
const
  full = '*';
  empty = ' ';
var
  row: rowrange;
  col: colrange;
begin                                { процедура WriteMap }
  writeln('Ниже следует конфигурация:');
  for row := 0 to (maxrow + 1) do
  begin
    for col := 0 to (maxcol + 1) do
      if map [row, col] = alive then
        write(full)
      else
        write(empty);
      writeln
    end
  end                                { обработка строки }
end;                                { процедура WriteMap }

```

ответ  
пользователя

Наконец мы переходим к функции UserSaysYes, которая определяет, желает ли пользователь вычислять следующее поколение. Задача функции UserSaysYes — потребовать от пользователя ответить yes или no. Чтобы сделать программу невосприимчивой к ошибкам ввода, запрос пользователю помещается в цикл, который повторяется до тех пор, пока от пользователя не поступит приемлемый ответ. *Множества* языка Pascal предоставляют удобную возможность за один раз проверить несколько возможных ответов.

```

function UserSaysYes: Boolean;           { пользователь ответил 'да' }
{ Pre:  Предусловия отсутствуют.
  Post: Функция возвращает true, если пользователь отвечает любым
        словом, начинающимся с 'y' или 'Y', и false, если пользователь
        отвечает любым словом, начинающимся с 'n' или 'N'.
  Uses: Не использует ничего. }
var
  response: char;
  valid: Boolean;
begin                                     { функция UserSaysYes }
  repeat
    write(' (y,n)?');
    readln(response);
    valid := (response in ['n', 'N', 'y', 'Y'] );
    if not valid then
      write('Пожалуйста, ответьте вводом одной из букв y или n')
  until valid;
  UserSaysYes := (response in ['y', 'Y'])
end;                                     { функция UserSaysYes }

```

Легко видеть, что функция `UserSaysYes` может оказаться полезной не только для программы `Life`, но и во многих других наших программах. Другими словами, мы будем рассматривать эту функцию как **утилиту**, которая должна быть доступна в любой момент, когда она нам понадобится.

пакет утилит

Фактически мы вскоре создадим целый ряд процедур и функций, которые можно использовать в разных приложениях. Все эти программы мы соберем вместе в **пакет утилит**, который при необходимости можно включить в любую программу. Большинство компиляторов предоставляют средства копирования одного файла в другой в процессе компиляции. Это часто выполняется с помощью такого рода директив компиляции:

```
%include utility.seg   или   { $I utility.seg }.
```

модули системы  
Turbo Pascal

Конкретная форма директивы зависит от компилятора. В системах Turbo Pascal<sup>3</sup> версии 4.0 или более поздних имеется даже еще лучший метод. В этих системах мы можем преобразовать пакет утилит в модуль **unit**, коллекцию объявлений и процедур, которые могут компилироваться отдельно (но которая не имеет главной программы и поэтому не может выполняться сама по себе). После того как модуль откомпилирован, он может использоваться любой программой просто путем включения объявления

```

uses Utility;           { модуль, содержащий процедуры-утилиты,
                          только в Turbo Pascal }

```

в начале программы.

По мере разработки нами в последующих главах новых программ, мы встретимся с рядом процедур и функций, которые целесообразно поместить в пакет утилит, чтобы в любой момент мы могли ими воспользоваться. В приложении С перечислены все подпрограммы пакета утилит; там же описаны некоторые другие модули и включаемые файлы.

<sup>3</sup> Turbo Pascal является зарегистрированной торговой маркой фирмы Borland International, Inc.

Мы теперь имеем все подпрограммы для моделирования жизни. Наступило время сделать паузу и проверить, как все работает.

#### 1.4.4. Драйверы

раздельная  
отладка

В небольших программных проектах каждая подпрограмма обычно устанавливается на свое место сразу же после написания, и после этого результирующая программа может отлаживаться и тестироваться. В больших проектах, однако, компиляция всего проекта может изменить взаимодействие отлаживаемой подпрограммы с другими элементами проекта, и часто изучение общего хода выполнения программы не позволяет сказать, правильно ли работает конкретная подпрограмма, или нет. Даже в небольших проектах выход одной подпрограммы может использоваться другой подпрограммой таким образом, что сразу трудно оценить, правильно ли передаются данные.

драйвер  
подпрограммы

Один из способов отладки и тестирования простых подпрограмм заключается в написании коротких вспомогательных программ, назначением которых является предоставление подпрограмме входных данных, вызов этой подпрограммы и оценка результата. Такие вспомогательные программы называются **драйверами** подпрограмм. При использовании драйверов каждую подпрограмму можно изолировать и изучать саму по себе, что во многих случаях ускоряет поиск ошибок.

В качестве примера давайте напомним драйверы для подпрограмм проекта Life. Сначала рассмотрим функцию NeighborCount. В главной программе вывод этой подпрограммы используется, но не отображается так, чтобы его можно было видеть, поэтому мы не можем быть уверены в его правильности. Для тестирования функции NeighborCount мы предоставим ей прямоугольный массив map, будем вызывать ее для каждой ячейки прямоугольного массива и выводить результаты. Наш драйвер, таким образом, использует процедуру Initialize для исходной установки прямоугольного массива и в какой-то мере напоминает исходную главную программу.

```

program DriveNeighborCount (input, output);
{ Программа-драйвер для функции NeighborCount в программе Life. }
{ Pre: Предусловия отсутствуют.
  Post: Удостоверяет, что функция NeighborCount возвращает
        правильные значения.
  Uses: Использует процедуры Initialize. }
{ Объявления констант, типов и переменных могут быть взяты
  из главной программы. }
begin
  Initialize(map);
  for row := 1 to maxrow do begin
    for col := 1 to maxcol do
      write(NeighborCount(row, col): 3);
      writeln
    end
  end.
```

Иногда две подпрограммы могут быть использованы для проверки друг друга. Например, простейший способ проверки процедур Initialize и WriteMap заключается в использовании драйвера, объявления которого совпадают с объявлениями главной программы, а выполняемая часть выглядит так:

```
begin Initialize(map); WriteMap(map) end.
```

Для тестирования обеих процедур мы запускаем этот драйвер и должны убедиться в том, что выводимая конфигурация совпадает с той, которая задана в качестве исходной.

### 1.4.5. Трассировка программы

групповое  
обсуждение

После того, как подпрограммы собраны в законченную программу, наступает время проверить все в целом. Одним из наиболее эффективных способов обнаружения скрытых дефектов является *структурированный разбор программы*. В этом случае программист показывает законченную программу другому программисту или небольшой группе программистов и подробно объясняет, что происходит в программе, начиная от главной программы и переходя затем последовательно ко всем подпрограммам. Такой структурированный разбор полезен по трем причинам. Во-первых, программисты, незнакомясь с фактическим кодом, часто обнаруживают дефекты или концептуальные ошибки, которые не замечаются автором программы. Во-вторых, вопросы, задаваемые другими людьми, могут помочь вам прояснить собственные идеи и обнаружить свои ошибки. В-третьих, в процессе структурированного разбора часто возникают идеи тестирования, которые могут пригодиться на дальнейших стадиях разработки программного обеспечения.

предложения  
write  
для отладки

Довольно редко большая программа при первом же запуске начинает работать правильно, причем локализация ошибок обычно оказывается весьма затруднительной. Многие системы программирования включают *средства трассировки*, которые позволяют вести учет вызовам подпрограмм, изменениям переменных и т. д. Однако более простым и в то же время эффективным средством отладки является *фиксация текущего состояния* выполняемой программы путем включения в важных точках главной программы предложений write. При каждом вызове подпрограммы на экран может выводиться сообщение, а перед и после вызова каждой подпрограммы — значения важных переменных. Такие «мгновенные снимки» состояния программы помогают программисту быстро сузить область поиска и подойти к тому месту в программе, где возникает ошибка.

временные  
вставки

Для описания кода, включаемого в программу с целью ее отладки, часто используется термин «вставка». Не бойтесь включать в программу по мере ее написания такие вставки. Когда в них отпадет необходимость, их можно легко удалить, и в то же время они существенно помогают в отладке. Некоторые компиляторы с языка Pascal рассматривают конструкции { ... } и (\* ... \*) как различные формы комментариев, которые даже могут вставляться друг в друга. Если ваш компилятор обладает этим свойством, вы можете оставить вставки в программе навсегда, поместив их в знаки комментария, и используя при этом один вид скобок для

обычных комментариев, а другой — для помещения в них отслуживших свою службу вставок.

Если в вашей программе обнаруживается таинственная ошибка, которую вы никак не можете локализовать, поместите в главную программу вставки, которые распечатывают значения существенных для поведения программы переменных. Такие вставки следует разместить на границах важных по значению секций, на которые делится главная программа. (Если вы написали программу заметного размера, которая не подразделяется на секции, вы уже сделали серьезную ошибку в проектировании структуры вашей программы, и ее следует исправить.) Имея значения данных в главных разделяющих точках, вы сможете определить, какая из секций программы ведет себя неправильно, после чего перейти к изучению этой секции, помещая вставки между ее подсекциями.

защитное  
программирование

Другим важным методом обнаружения ошибок является «**защитное**», или «**безопасное**» программирование. Поместите предложения **if** перед началом каждой процедуры, чтобы проверить выполнение предусловий. Если предусловие не выполняется, выведите сообщение. В этом случае вы получите предупреждение, как только возникнет потенциально неправильная ситуация; если все идет нормально, эта проверка на ошибки останется для пользователя совершенно невидимой. Разумеется, особенно важно проверять соблюдение предусловий, когда ввод в программу осуществляется пользователем, или поступает из файла или какого-то другого источника вне самой программы. Даже удивительно, как часто проверка предусловий позволяет обнаружить ошибки в местах, в которых по вашему мнению все совершенно правильно.

статический  
анализатор

Для очень больших программ иногда используется еще одно средство. Это статический анализатор, программа, которая анализирует исходный текст программы (написанной, например, на языке Pascal) в поисках неинициализированных или неиспользуемых переменных, невыполняемых секций кода или других элементов, которые могут оказаться неправильными.

### 1.4.6. Принципы тестирования программы

выбор  
тестовых данных

До сих пор мы ничего не говорили о выборе данных, используемых для тестирования программ или подпрограмм. Этот выбор, разумеется, целиком зависит от существа разрабатываемого проекта, поэтому мы здесь можем сделать только самые общие замечания. Прежде всего, мы должны иметь в виду, что

**Программистский принцип**  
*Качество тестовых данных важнее их количества*

Множество пробных прогонов, которые выполняют те же вычисления в тех же ситуациях, не более эффективны, чем один пробный прогон.

**Программистский принцип**  
*Тестирование программы может показать наличие ошибок,  
но никогда не их отсутствие*

Вполне возможно, что существуют другие ситуации, которые так и не были протестированы даже после многих пробных прогонов. Для любой программы значительного размера невозможно выполнить полностью исчерпывающие тесты, хотя тщательный выбор тестовых данных существенно повышает вашу уверенность в ее правильной работе. Например, все, конечно, уверены, что типичный компьютер может правильно сложить два числа с плавающей точкой, однако эта уверенность, разумеется, не основана на исчерпывающем тестировании компьютера, когда его заставляют складывать все возможные пары чисел с плавающей точкой с проверкой каждого результата. Если число с плавающей точкой двойной точности занимает 64 бит, то всего могут существовать  $2^{128}$  различных пар складываемых чисел. Это число астрономически велико: все компьютеры, выпущенные до настоящего времени, выполнили лишь ничтожную долю такого количества операций сложения. Наша уверенность в том, что компьютер выполняет сложение правильно, основана на тестировании отдельных компонентов схемы суммирования, т. е. проверки правильности сложения каждого из 64 разрядов и правильности учета переноса в следующий разряд.

методы  
тестирования

Существуют по меньшей мере три общих принципа, используемых при выборе тестовых данных.

### 1. Метод черного ящика

Большинству пользователей большой программы не интересны детали ее функционирования; им нужно только получить результат. Другими словами, они желают рассматривать программу как черный ящик, откуда и произошло название метода отладки. Точно так же при проверке правильности работы программы тестовые данные должны выбираться исходя из спецификаций решаемой задачи, безотносительно к внутренним деталям программы. Тестовые данные должны выбираться, по меньшей мере, следующим образом:

выбор данных

1. **Простые значения.** Программу следует отлаживать с помощью легко проверяемых данных. Не один студент, пытавшийся проверить свою программу только со сложными, запутанными данными, испытывал смущение, когда преподаватель предъявлял ему результат тривиального примера.
2. **Типичные, реалистичные значения.** Всегда проверяйте работу программы с типичными для ее применения данными. При этом данные должны быть достаточно просты, чтобы результаты можно было проверить вручную.
3. **Крайние значения.** Многие программы имеют тенденцию ошибаться на границах своей применимости. Например, превышение счетчиком предельного значения или выход за пределы массива может иметь драматические последствия.
4. **Недопустимые значения.** «Мусор на входе, мусор на выходе» — это известное в компьютерных кругах изречение не следует воспринимать как руководство к действию. Если на вход правильно написанной программы поступает мусор, на выходе программы по меньшей мере

должно появиться разумное сообщение об ошибке. Разумеется, программа должна иметь средства обнаружения типичных ошибок ввода и выполнять только те вычисления, которые имеют смысл после отбраковки ошибочно введенных входных данных.

## 2. Метод стеклянного ящика

Второй подход к выбору тестовых данных основывается на очевидном утверждении, что программу вряд ли можно считать оттестированной, если в ней остаются участки кода, которые никогда не выполнялись. В методе *стеклянного ящика* анализируется логическая структура программы, и для каждой возможной альтернативы ее поведения выбирают тестовые данные, которые порождают эту альтернативу. Так, например, тестовые данные должны обеспечить обработку всех вариантов в каждом предложении **case**, всех предложений в каждой конструкции **if** и условия завершения каждого цикла. Если в программе имеется несколько предложений выбора или итераций, для проверки всех возможных путей выполнения потребуются различные комбинации тестовых данных. На рис. 1.3 показан короткий программный сегмент с возможными путями выполнения.

проверка путей  
выполнения

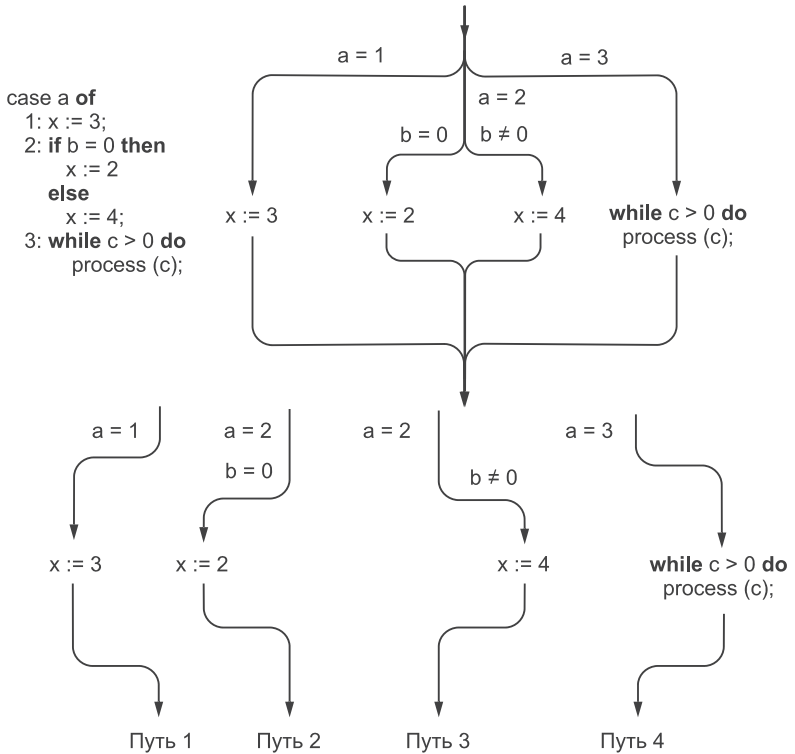


Рис. 1.3. Программный сегмент и возможные пути его выполнения



средством отладки и тестирования. В хорошо разработанной программе в каждом модуле будет лишь небольшое количество циклов и альтернативных путей выполнения. В результате нескольких тщательно выбранных комбинаций тестовых данных будет достаточно для тестирования каждого модуля по отдельности.

модульное  
тестирование

При тестировании методом стеклянного ящика преимущества модульной структуры программы становятся очевидными. Рассмотрим типичный пример проекта, включающего 50 подпрограмм, каждая из которых реализует 5 различных случаев или альтернатив. Если бы мы тестировали такую программу как единое целое, то для проверки всех возможных альтернатив потребовалось бы  $5^{50}$  наборов тестовых данных. Каждый же модуль в отдельности требует только 5 тестовых наборов, а всего таких наборов потребуется  $5 \times 50 = 250$ . В результате проблема немыслимого размера была сведена к процедуре тестирования вполне умеренной, для большой программы, сложности.

сравнение

Если вы уверились в том, что метод стеклянного ящика всегда является предпочтительным, мы должны заметить, что на практике тестирование методом черного ящика обычно оказывается более эффективным в плане обнаружения ошибок. Возможно, одна из причин этого связана с тем, что большая часть тонких программных ошибок возникает не внутри подпрограмм, а на их границах, в интерфейсах между подпрограммами, в силу неправильного или неполного понимания условий и правил обмена информацией между подпрограммами. Поэтому при отладке крупных проектов наиболее разумной стратегией тестирования будет приложение метода стеклянного ящика к каждому небольшому модулю сразу после его написания, и использование метода черного ящика с соответствующими тестовыми данными для тестирования более крупных секций программы по мере их завершения.

ошибки  
в интерфейсе

### 3. Метод тикающего ящика

В заключение этого раздела упомянем еще одну стратегию тестирования программ, стратегию, которая, к сожалению, распространена очень широко. Эту стратегию можно было бы назвать методом *тикающего ящика*. Она состоит в том, что после достаточно тщательной отладки проекта он совсем не тестируется, а вместо этого передается потребителю для пробного использования и приемки. Результатом, разумеется, будет бомба замедленного действия.

### Упражнения 1.4

- E1.** Если вы подозреваете, что программа Life содержит ошибки, где в тексте главной программы лучше всего разместить вставки? Какую информацию следует вывести на экран?
- E2.** Обратитесь к своему решению упражнения E7 из раздела 1.3 (разработка программы для вывода точечного графика) и укажите наилучшие места включения вставок, если они потребуются.
- E3.** Найдите наилучшие наборы тестовых данных при тестировании методом черного ящика для каждого из следующих случаев:



- (a) Функция возвращает наибольший из своих трех параметров, являющихся действительными числами.
  - (b) Функция возвращает квадратный корень из действительного числа.
  - (c) Функция возвращает наименьшее общее кратное своих двух параметров, которые должны быть целыми положительными числами (*Наименьшее общее кратное* есть наименьшее целое число, которое кратно обоим параметрам. Например, наименьшее общее кратное для 4 и 6 есть 12, для 3 и 9 — 9, а для 5 и 7 — 35.)
  - (d) Процедура упорядочивает три целых числа, задаваемые в качестве ее параметров, в восходящем порядке.
  - (e) Процедура упорядочивает массив A целых чисел, индексируемых от 1 до переменной n, в восходящем порядке; значения A и n являются параметрами процедуры.
- E4.** Найдите подходящий набор тестовых данных при тестировании методом стеклянного ящика следующих фрагментов:
- (a) Предложения
 

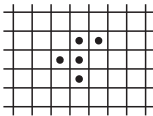
```
if a < b then if c > d then x := 1 else if c = d then x := 2
else x := 3 else if a = b then x := 4 else if c = d then x := 5
else x := 6;
```
  - (b) Функции NeighborCount(map, row, col).

### Программные проекты 1.4

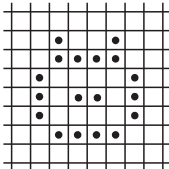
- P1.** Реализуйте программу Life из этой главы на своем компьютере и убедитесь, что она работает правильно.
- P2.** Протестируйте программу Life с примерами, показанными на рис. 1.1.
- P3.** Выполните прогоны программы Life с исходными конфигурациями, приведенными на рис. 1.4. Некоторые из этих конфигураций проходят через большое число поколений до достижения стабильной конфигурации или конфигурации с предсказуемым поведением.

## Подсказки и ловушки

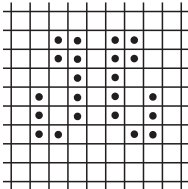
1. Приступая к решению задачи, удостоверьтесь в том, что вы ее понимаете.
2. Приступая к написанию программы, удостоверьтесь в том, что вы понимаете, какой алгоритм надо в ней использовать.
3. При возникновении затруднений разделите программу на секции и проанализируйте функционирование каждой секции в отдельности.
4. Старайтесь, чтобы ваши подпрограммы были простыми и короткими; одиночная подпрограмма редко занимает более одной страницы текста.
5. При написании подпрограмм включайте в них тщательно продуманную документацию, как это описано в разделе 1.3.2.
6. Следите за тем, чтобы для каждой подпрограммы точно описывались пред- и постусловия.



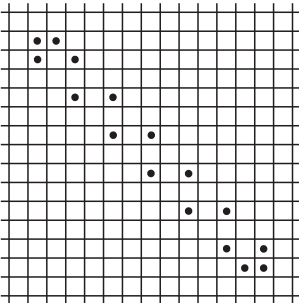
Пентамино для символа г



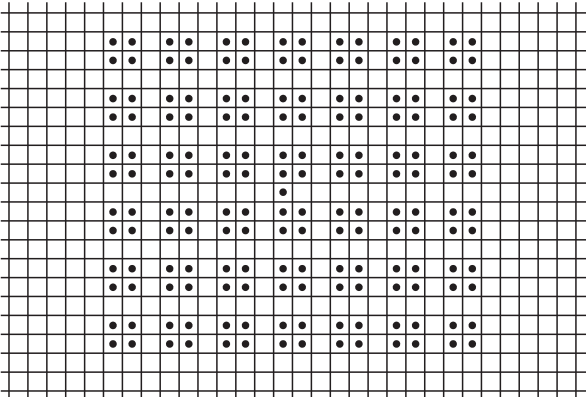
Чеширский кот



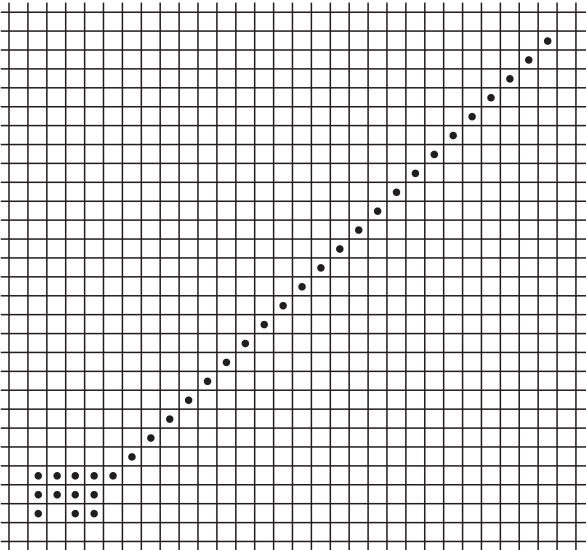
Бокал



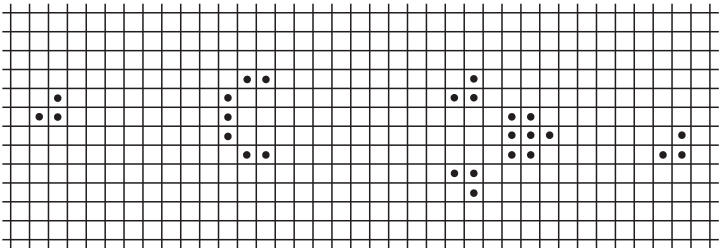
Столбик-символ парикмахерской у ее вход



Вирус



Уборка урожая



Катапульта для планера

Рис. 1.4. Конфигурации для программы Life

7. В начало каждой подпрограммы включайте проверку на фактическое выполнение указанного в ней предусловия.
8. При каждом использовании подпрограммы спрашивайте себя, почему вы уверены в выполнении ее предусловий.
9. Для упрощения отладки используйте заглушки и драйверы, а также тестирование методами черного ящика и стеклянного ящика.
10. Для локализации ошибок повсеместно используйте вставки.
11. Программируя массивы, опасайтесь выхода индексов за пределы массивов хотя бы на 1. Всегда при отладке программ, использующих массивы, прибегайте к проверке на граничные значения.
12. По мере написания программы придерживайтесь аккуратного форматирования ее исходного текста; это серьезно облегчит дальнейшую отладку.
13. Следите за тем, чтобы ваша документация соответствовала вашему коду, и, уточняя текст программы, вносите изменения в код, а не только в комментарии.
14. Постарайтесь объяснить вашу программу кому-нибудь еще. При этом вы сами поймете ее лучше.
15. Помните программистские принципы!

## Обзорные вопросы

Большинство глав этой книги завершаются набором вопросов, подобранных так, чтобы помочь вам вспомнить основные идеи данной главы. Ответы на все эти вопросы фактически содержатся в тексте книги; если вы затрудняетесь в ответе, прочитайте заново соответствующий раздел.

1. В каких случаях разумно использовать однобуквенные имена переменных?
2. Назовите четыре вида информации, которую следует включать в документацию к программе.
3. В чем состоит различие между *внешней* и *внутренней* документацией?
4. Что представляют собой пред- и постусловия?
5. Назовите три вида параметров. В чем состоят особенности их обработки в языке Pascal?
6. Почему следует избегать появления в программе побочных эффектов?
7. Что представляет собой программная заглушка?
8. В чем заключается разница между заглушками и драйверами, и когда следует использовать те и другие?
9. Что представляет собой структурированный разбор программы?
10. Что такое вставка, и когда надо включать вставки в программу?
11. Опишите способ реализации безопасного программирования.
12. Назовите два метода тестирования программы и опишите, когда следует использовать каждый из них.

13. Как вы подойдете к решению задачи, если вы не можете сразу осознать все детали этой задачи?

## Литература для дальнейшего изучения

### Pascal

Язык программирования Pascal был разработан Никлаусом Виртом (Niklaus Wirth), который впервые опубликовал его описание в 1971 г. Для старых версий языка стандартным руководством была книга

K. Jensen and N. Wirth, *PASCAL User Manual and Report*, second edition, Springer-Verlag, Berlin, Heidelberg, New York, 1974, 167 pages.

Имеется русский перевод: Йенсен К., Вирт Н. Паскаль: Руководство для пользователя. — М.: Компьютер, 1993. — 255 с.

В дальнейшем Международная организация по стандартизации (International Standards Organization, ISO) разработала спецификацию стандартной версии языка Pascal, которой придерживается большинство современных компиляторов. Этот стандартный Pascal кратко, но точно описан в следующей книге, к которой вы можете обращаться при возникновении у вас тонких языковых проблем:

Doug Cooper, *Standard Pascal User Reference Manual*, W. W. Norton, New York, 1983, 176 pages.

Многие хорошие учебники посвящены более развернутому описанию языка Pascal, причем их слишком много, чтобы их можно было здесь перечислить. В этих учебниках можно также найти массу примеров и приложений. Однако некоторые книги, предназначенные для начинающих, не содержат описания важных «продвинутых» средств языка Pascal, которые будут использоваться в этой книге. Поэтому проследите за тем, чтобы выбранный вами учебник описывал полный синтаксис стандартного языка Pascal.

### Программистские принципы

Приведем три книги, содержащие много полезных советов, касающихся стиля программирования и правильного составления программ, а также примеры хорошего и плохого программирования:

Brian Kernighan and P. J. Plauger, *The Elements of Programming Style*, second edition, McGraw-Hill, New York, 1978, 168 pages.

Henry F. Ledgard, Paul A. Nagin, and John F. Hueras, *Pascal with Style: Programming Proverbs*, Hayden Book Company, Hasbrouk Heights, N.J., 1979, 210 pages.

Dennie Van Tassel, *Program Style, Design, Efficiency, Debugging, and Testing*, second edition, Prentice-Hall, Englewood Cliffs, N.J., 1978, 323 pages.

Эдсгер У. Дейкстра (Edsger W. Dijkstra) был родоначальником движения, известного под названием структурного программирования, которое требует применения к проектированию и написанию программ тща-

тельно разработанного нисходящего подхода. Эти принципы были им сформулированы в опубликованном им в марте 1968 г. письме под названием «Go To Statement Considered Harmful» («О вреде предложения Goto») в журнале *Communications of the ACM* (том 11, стр. 147–148). С тех пор Дейкстра опубликовал несколько статей и книг, весьма полезных с точки зрения изучения правильного метода программирования. Одной из наиболее интересных является

Edsger W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1976, 217 pages.

Имеется русский перевод: Э. Дейкстра. Дисциплина программирования. — М.: Мир, 1978.

## Игра «Жизнь»

Выдающийся английский математик Дж. Х. Конвей (J. H. Conway) внес значительный вклад в столь различающиеся области математики, как теория конечных простых групп, логика и комбинаторика. Он придумал игру «Жизнь», начав с изучения клеточных автоматов и разработав правила воспроизводства, которые затрудняют безграничное разрастание популяции, но при которых многие конфигурации проходят через интересные последовательности. Конвей, однако, не опубликовал свои наблюдения, а только сообщил о них Мартину Гарднеру (Martin Gardner). Популярность игры стремительно возросла, когда появилось ее обсуждение в работе

Martin Gardner, «Mathematical Games» (regular column), *Scientific American* 223, no 4 (October 1970), 120–123; 224, no 2 (February 1971), 112–117.

Примеры в конце разделов 1.2 и 1.4 взяты из этой работы, которая была перепечатана со включением дальнейших результатов в

Martin Gardner, *Wheels, Life and Other Mathematical Amusements*, W. H. Freeman, New York, 1983, pp 214–257.

Имеется русский перевод: Гарднер М., Математические досуги. — 2-е изд., испр. и доп. — М.: Мир, 2000.

В настоящей книге приведена библиография статей, посвященных игре «Жизнь». В течение нескольких лет даже выпускался ежеквартальный бюллетень под название *Lifeline*, который публиковал для приверженцев этой игры последние разработки и статьи на смежные темы.

В этой главе продолжается обсуждение принципов разработки надежных программ со специальным акцентом на методики, позволяющие создавать большие программные системы. Эти методики включают спецификацию задачи, разработку алгоритма, верификацию и анализ, а также тестирование и поддержку программ. Эти общие принципы вводятся в контексте разработки второго варианта программы для игры «Жизнь», варианта, основанного на более утонченных методах, чем те, которые были использованы в предыдущей главе.

Программная инженерия представляет собой дисциплину в рамках вычислительной техники, посвященную методам разработки и поддержки больших программных систем. Наша цель при описании некоторых из этих методов заключается в демонстрации их важности при решении задач реалистичного размера.

Хотя значительная часть обсуждений этой главы инициирована проблемами игры «Жизнь» и относится непосредственно к программе Life, в действительности весь материал предназначен для иллюстрации общих методов, приложимых к значительно более широкому кругу задач практической важности.

## 2.1. Поддержка программ

Небольшие программы, написанные в качестве упражнений или демонстраций, обычно запускаются всего несколько раз и затем отбрасываются, но жизнь больших практически важных программ протекает совсем иначе. Программа, имеющая практическое значение, будет запускаться много раз, обычно многими разными людьми, и написание и отладка такой программы являются только начальными стадиями ее использования. Эти стадии к тому же характеризуют лишь начало длительной работы, призванной сделать эту программу полезной и сохранить ее полезность в течение заметного времени. Для того, чтобы программа соответствовала предъявляемым к ней требованиям, необходимы *обзор* и

анализ программы; ее необходимо *адаптировать* к изменяющимся условиям использования; наконец, для лучшего удовлетворения требований пользователей программу необходимо *модифицировать*.

Проиллюстрируем эти действия рассмотрением заново программы для игры «Жизнь», написанной и протестированной в главе 1. Наш первоначальный проект будет полностью переделан. Эта работа будет, в известной степени, стрельбой из пушек по воробьям, поскольку игровой проект, каковым является программа Life, не стоит тех усилий, которые нам придется затратить. Однако в процессе работы мы изучим методы программирования, важные для многих других приложений.

### 2.1.1. Обзор программы Life

проблемы

Запустив программу Life, вы сразу обнаружите ее слабое место — неудачный метод ввода начальной конфигурации. Для пользователя неестественно вычислять и вводить числовые координаты каждой живой ячейки. Форма ввода должна отображать ту же решетку, на которой выводятся результаты моделирования жизни.

неудачная  
форма ввода

низкая скорость

Есть также и вторая проблема, с которой вы столкнетесь, если будете запускать программу на очень медленной машине или в сильно загруженной системе разделения времени: скорость работы программы может оказаться обескураживающе низкой. Между выводом изображения очередного поколения и началом вывода следующего может возникнуть заметная задержка.

Наша цель заключается в улучшении программы, чтобы она могла эффективно выполняться на небольших микрокомпьютерах. Проблема улучшения формы ввода рассматривается в качестве упражнения; в тексте обсуждается задача повышения скорости работы.

#### 1. Анализ программы Life

число операций

Прежде всего мы должны выяснить, в каком месте программа потребляет наибольший объем вычислительного времени. Если мы внимательно рассмотрим программу, мы сразу же обнаружим, что неприятности не могут быть связаны с процедурой `Initialize`, поскольку она выполняется лишь один раз, перед началом основного цикла. В цикле, который вычисляет поколения, мы имеем пару вложенных циклов, которые вместе будут выполняться

$$\text{maxrow} \times \text{maxcol} = 20 \times 60 = 1200$$

раз. Программные строки внутри этих циклов будут давать значительный вклад в используемое время.

вложенные циклы

Внутри циклов мы прежде всего активизируем функцию `NeighborCount(map, row, col)`. Эта функция сама содержит пару вложенных циклов (заметьте, что мы уже опустили на уровень вложенности 5), которые выполняют свое внутреннее предложение 9 раз. В функции также имеются 3 предложения вне циклов, что дает в сумме 12 предложений.

Внутри вложенных циклов главной программы имеется, помимо вызова функции, только сравнение, с помощью которого определяется, какой вариант выбрать, и соответствующее предложение присваивания.

Таким образом, имеется всего 2 предложения дополнительно к 12 в функции. Вне вложенных циклов имеется предложение присваивания массивов `map=newmap`, которое, выполняя копирование 1200 данных, приблизительно эквивалентно еще одному предложению в цикле. Еще имеется вызов процедуры `WriteMap`, тот или иной вариант которой очевидно необходим, так как пользователь должен видеть результаты работы программы. Однако наша основная забота касается вычислительной части программы, поэтому пока мы не будем думать о времени, требуемом для выполнения функции `WriteMap`. Итак, для каждого поколения вычисления требуют выполнения около

$$1200 \times 15 = 18000$$

предложений, из которых около

$$1200 \times 12 = 14400$$

выполняются в функции.

На медленном микрокомпьютере или в сильно загруженной системе разделения времени каждое предложение запросто может потребовать для выполнения от 100 до 500 микросекунд, и тогда время вычисления одного поколения вполне может дойти до 20 секунд, что, разумеется, для большинства пользователей будет неприемлемо. С другой стороны, на более быстром компьютере все вычисления вероятно будут выполняться так быстро, что пользователь едва заметит задержку между выводом одного поколения и началом вывода следующего.

Поскольку львиная доля времени используется функцией определения числа живых соседей данной ячейки, для ускорения программы мы должны сконцентрировать свое внимание на более эффективном выполнении этой работы. Однако перед тем, как мы приступим к обсуждению новых идей, давайте сделаем паузу и сделаем следующее программное заявление:

### ***Программистский принцип***

*Большинство программ 90 процентов своего времени тратят на выполнение 10 процентов своих команд. Найдите эти 10 процентов и направьте свои усилия на повышение их эффективности.*

Только длительная практика и опыт позволят вам решать, какие детали алгоритма важны с точки зрения эффективности, а на что можно не обращать внимания, однако вы должны стремиться развить в себе это умение, чтобы научиться выбирать альтернативные методы или направлять свои усилия в том направлении, которое обеспечит максимальное повышение эффективности.

## **2. Альтернативные способы решения задачи**

После того, как мы поняли, какая именно часть программы выполняет большую часть работы, мы можем начать рассматривать альтернативные методы в надежде повысить ее эффективность. Для нашей программы `Life` зададимся вопросом, каким образом можно уменьшить время, тре-



степень занятости  
массива

буемое для подсчета числа живых соседей в окрестностях каждой ячейки. Необходимо ли нам подсчитывать число соседей каждой ячейки в любом поколении? Очевидно нет, если это число не изменяется от поколения к поколению. Если вы потратите некоторое время на эксперименты с программой Life, вы несомненно заметите, что для многих интересных конфигураций число занятых ячеек в любом поколении заметно меньше общего числа наличных ячеек. Из 1200 позиций в типичных случаях занятыми оказываются не более 40. Наша программа тратит массу времени на трудоемкие вычисления для установления того очевидного факта, что ячейки, изолированные от живых ячеек, очевидно не имеют живых соседей и не смогут сами стать живыми. Если мы сможем предотвратить или существенно сократить эти бессмысленные вычисления, мы получим гораздо лучшую программу.

В качестве первого приближения давайте попытаемся ограничить вычисления ячейками, находящимися в ограниченной области вокруг занятых ячеек. Если эта занятая область (которую мы должны будем точно задать) приблизительно прямоугольна, тогда мы можем легко реализовать эту схему, изменив пределы в циклах другими значениями, которые будут описывать занятую область. Однако эта схема окажется крайне неэффективной, если занятая область выглядит в виде большого кольца, или, например, имеются только две небольшие занятые области в противоположных углах очень большого прямоугольника. Для реализации нашего плана в случае сильно непрямоугольных областей, возможно, потребуется выполнения такого количества сравнений, а также и циклов, что ни о каком повышении эффективности не будет и речи.

### 2.1.2. Новый старт и новый метод для программы Life

массивы  
и функции

Вернемся на секунду назад. Если мы теперь решим хранить массив с числом живых соседей для каждой ячейки, то от поколения к поколению в массиве будут изменяться только отсчеты, соответствующие непосредственным соседям тех ячеек, которые умирают или рождаются.

Мы можем существенно улучшить время выполнения нашей программы, преобразовав функцию NeighborCount в массив и добавив в предложение **case** соответствующие предложения для обновления массива в то время, как мы выполняем изменения между поколениями или (что, возможно, концептуально проще) поступить по-другому: копируя пептар в map, мы можем отмечать, где происходили рождения и смерти, и в это время обновлять массив.

Чтобы подчеркнуть, что теперь мы вместо функции NeighborCount используем массив, мы изменим имя и будем называть массив numbernbrs.

Предложенный нами метод все еще предполагает проход по крайней мере однажды по всему массиву map в каждом поколении, что, очевидно, приводит к значительному объему бесполезной работы. Проявив дальнейшую смекалку, мы можем устранить необходимость просмотра незанятых областей.

разработка  
алгоритма

Когда ячейка рождается или умирает, она изменяет значение `numbernbrs` для каждого из своих непосредственных соседей. Выполняя эти изменения, мы можем отмечать, когда мы находим ячейку с таким отсчетом, что она родится или умрет в следующем поколении. Так мы будем отслеживать ячейки, которые, так сказать, обречены на гибель или ожидают в следующем поколении. После того, как мы закончили выполнение изменений в текущем поколении и выводим `map`, нас уже ожидают все ячейки, которые станут живыми или умрут в очередном поколении.

Теперь должно быть ясно, что в действительности нам нужны две структуры для хранения рождений и две для смертей, по одной для изменений, выполняемых сейчас, и по одной (их мы добавим), содержащей изменения для следующего поколения. После завершения изменений для текущего поколения, мы выводим на экран массив `map` и затем определяем изменения, которые необходимо выполнить для следующего поколения. Этот процесс проиллюстрирован на рис. 2.1.

Начальная конфигурация

	1	2	3	4	5
1					
2			•		
3		•	•	•	
4			•		
5					

кандидаты	
на оживление	на гибель
(2, 2)	(3, 3)
(2, 4)	
(4, 2)	
(4, 4)	

После одного поколения

	1	2	3	4	5
1					
2		•	•	•	
3		•	×	•	
4		•	•	•	
5					

кандидаты			
		на оживление	на гибель
ожили	умерли	(1, 3)	(2, 3)
(2, 2)	(3, 3)	(3, 1)	(3, 2)
(2, 4)		(3, 5)	(3, 4)
(4, 2)		(5, 3)	(4, 3)
(4, 4)			

После двух поколений

	1	2	3	4	5
1			•		
2		•	×	•	
3	•	×		×	•
4		•	×	•	
5			•		

кандидаты			
		на оживление	на гибель
ожили	умерли	(2, 3)	(пусто)
(1, 3)	(2, 3)	(3, 2)	
(3, 1)	(3, 2)	(3, 4)	
(3, 5)	(3, 4)	(4, 3)	
(5, 3)	(4, 3)		

Рис. 2.1. Программа Life отслеживает изменения

Теперь давайте просуммируем наши решения, описав неформальным образом программу, которую мы будем разрабатывать.

инициализация

Получаем начальную конфигурацию живых ячеек и используем ее для вычисления массива, содержащего отсчеты числа соседей для всех ячеек. Определяем ячейки, которые оживут или умрут в первом поколении;

главный цикл

Будем повторять следующие шаги произвольное число раз:

Оживим все ячейки, которые готовы стать живыми;

Убьем все ячейки, которые готовы умереть;

Распечатаем массив `map` для сведения пользователя;

готовим  
для следующего  
поколения

Увеличим отсчеты числа соседей для каждого соседа каждой оживающей ячейки; если отсчет числа соседей достигает соответствующей величины, отмечаем эту ячейку как кандидата на оживление или смерть в следующем поколении;

Уменьшаем отсчеты числа соседей для каждого соседа каждой умершей ячейки; если отсчет числа соседей достигает соответствующей величины, отмечаем эту ячейку как кандидата на оживление или смерть в следующем поколении;

Очевидно, что в этом обзоре необходимо еще уточнить массу деталей. В следующем разделе мы перейдем к этим деталям, начав с точной спецификации структуры данных, предназначенной для отслеживания состояния ячеек, вслед за чем разработаем спецификации для каждой из процедур и функций.

## Упражнения 2.1

- E1.** Пользователь может захотеть запустить программу `Life` (из этой главы) в решетке меньшего, чем  $20 \times 60$ , размера. Придумайте, каким образом можно сделать `maxrow` и `maxcol` переменными, которые пользователь мог бы устанавливать после запуска программы. Постарайтесь при этом внести в программу минимум изменений.
- E2.** Один из способов ускорения выполнения функции `NeighborCount(map, row, col)` заключается в удалении из решеток `map` и `newmap` ограды (дополнительных строк и столбцов, которые всегда мертвы). В этом случае, когда ячейка прилегает к границе, `NeighborCount` будет анализировать меньше восьми соседних ячеек, поскольку некоторые из них находятся за границами решетки. Для реализации такого подхода функция должна будет определять, находится ли ячейка `(row,col)` на границе, но эту операцию можно выполнить вне цикла. Как это повлияет на число предложений, выполняемых в `NeighborCount`?

## Программные проекты 2.1

- P1.** Перепишите процедуру `Initialize` так, чтобы она принимала занятые позиции в виде последовательности пробелов и символов `X` в соответствующих строках, вместо того чтобы требовать ввода занятых позиций в виде координатных пар.
- P2.** Добавьте в процедуру `Initialize` средство чтения начальной конфигурации из файла. Первой строкой файла будет комментарий с именем

конфигурации. Каждая оставшаяся строка файла будет соответствовать строке конфигурации. Каждая строка будет содержать  $x$  в каждой живой позиции и пробел в мертвой.

- Р3.** При использовании медленного терминала вывод всей решетки для каждого поколения будет осуществляться слишком медленно. Если на вашем терминале позиция курсора может контролироваться программой (прямая адресация курсора), перепишите функцию `WriteMap` так, чтобы она только обновляла решетку, а не рисовала ее целиком заново в каждом поколении.

## 2.2. Разработка алгоритма: второй вариант программы Life

После выбора базового метода и общих очертаний структур данных, требуемых для решения нашей задачи, наступило время начать разработку алгоритма: написать тщательные спецификации структур данных, затем главную программу, после чего постепенно вносить детализацию, пока все подпрограммы не будут определены, а весь продукт не будет сформулирован на компьютерном языке.

### 2.2.1. Списки: спецификации для структуры данных

#### 1. Списки и массивы

Вскоре после введения циклов и массивов любая студенческая группа, изучающая элементарное программирование, переходит к упражнениям на написание программ вроде следующего:

*Прочитайте целое число  $n$ , которое не должно быть больше 25, затем прочитайте список из  $n$  чисел и напечатайте этот список в обратном порядке.*

Это простое упражнение, возможно, окажется для некоторых студентов довольно трудным. Большинство, конечно, сообразит, что здесь нужно воспользоваться массивом, однако кое-кто попытается объявить массив из  $n$  элементов и с удивлением обнаружит, что попытка использовать в качестве размера массива переменную, а не константу, приведет к выводу сообщения об ошибке. Другие студенты скажут: «Я могу решить эту задачу, если я знаю, что чисел будет 25, но я не понимаю, как можно обработать меньшее количество чисел». Или: «Скажите мне до написания программы, чему равно  $n$ , и я решу задачу».

Трудности, которые испытывают эти студенты, проистекают не от их глупости, а в силу логики мышления. В курсе для начинающих иногда подчеркиваются различия двух совершенно разных понятий. Первое — это понятие *списка* из  $n$  чисел. Размер списка может изменяться, т. е. в него можно помещать числа или удалять их, так что если  $n = 3$ , то список содержит только 3 числа, а если  $n = 19$ , в нем будет 19 чисел. Второе понятие — это программное средство, называемому *массивом* или вектором. Массив содержит фиксированное число позиций, т. е. его размер

известен, когда программа компилируется. Список представляет собой *динамическую* структуру данных, потому что его размер может изменяться, в то время как список является *статической* структурой, у которой размер фиксирован.

реализация

Понятия списка и массива, разумеется, взаимосвязаны в том отношении, что список переменного размера может быть реализован в компьютере как часть массива фиксированного размера, в котором некоторые ячейки остаются неиспользуемыми. Позже, однако, мы выясним, что имеются различные способы реализации списков, и поэтому мы не будем смешивать вопросы реализации с более фундаментальным вопросом выбора конкретной структуры данных.

## 2. Базовые операции над списками

Для обработки ячеек в программе Life удобно использовать списки, поскольку число ячеек, ожидающих обработки, изменяется от поколения к поколению.

Просматривая обзор нового алгоритма программы Life в разделе 2.1.2, мы можем определить, какие операции над используемыми списками нам будут нужны. Прежде всего мы должны *создать* список, и мы должны уметь *очистить* список, чтобы сделать его пустым. Нам потребуются операции определения состояния списка, т. е. выяснения того, *пуст* ли список или *заполнен*, или функция, возвращающая *размер* списка. Наконец, нам нужно уметь *просматривать* список, где под просмотром понимается прохождение по списку и выполнение над каждый его элементом заданной операции.

простые списки

Хотя для списков допустимы и многие другие операции, перечисленные семь действий — это все, что нужно не только для игры «Жизнь», но и для удивительно большого разнообразия других приложений. Давайте теперь дадим точные спецификации списков и действий над ними, ограничившись перечисленными семью операциями.

## 3. Объявления типов

тип элементов списка

Для того, чтобы записать спецификации строгим образом, мы должны выбрать имена для используемых в программе типов. Для максимальной общности будем использовать имя *listentry* для типа элементов нашего списка. Для одного приложения *listentry* может быть *integer*; для другого элементы могут быть типа *char*. В проекте Life элементы списка должны быть координатами ячеек, поэтому мы включим в нашу программу объявление

**type** listentry = cell

Имея такой обобщенный тип *listentry*, мы сможем использовать один и тот же пакет для многих различных приложений.

родовые конструкции

Конструкции, позволяющие использовать обобщенные структуры данных и операции с данными разных типов носит название *родовых*. Некоторые языки программирования (например, Ada и C++), но не Pascal, позволяют программисту объявить структуры данных, процедуры и функции в виде родовых конструкций и затем использовать их без вся-

модули  
в Turbo Pascal

включаемые  
файлы

ких изменений для самых различных типов данных. Язык Pascal не допускает таких действий, поэтому нам придется потрудиться, чтобы сконструировать объявления, которые можно было бы использовать в различных приложениях. В этой книге мы рассмотрим два способа достижения этой цели. Если тип данного принадлежит одному из стандартных типов языка Pascal, мы сконструируем специальный пакет, который можно будет использовать в любом приложении. Если мы работаем в Turbo Pascal, мы даже можем откомпилировать такой пакет отдельно, оформив его в виде *модуля*. В тех случаях, когда тип данного не является стандартным, как, например, ячейки в проекте Life, мы не конструируем полностью отдельный пакет (поскольку он будет использован лишь однажды). Вместо этого мы будем хранить код для различных операций в файле, который будет подключаться к прикладной программе.

Таким образом, в проекте Life главная программа будет включать объявление типа

**type** listentry = cell

а затем — директиву компилятора, выполняющую копирование в программу пакета обслуживания списков. Форма этой директивы не описана в стандартном языке Pascal и зависит от компилятора. Типичными являются формы

%include list.seg или {\$I list.seg}

Мы будем использовать расширение «.seg» для файлов, содержащих программные сегменты, которые не могут быть откомпилированы сами по себе, т. е. не являются законченными программами или модулями.

Если, с другой стороны, мы работаем в системе Turbo Pascal с символьным списком, тогда, с целью повторного использования пакета во всех случаях, где нам может понадобиться символьный список, мы должны создать модуль Turbo Pascal с именем CharList. В интерфейсную часть модуля тогда следует включить объявление

**type** listentry = char

а в секцию реализации поместить директиву компилятора {\$I list.seg}, чтобы включить в программу все процедуры и функции для обработки списков.

Для упрощения процедуры написания наших программ мы разработаем для различных целей несколько отдельных модулей и включаемых файлов.

#### 4. Первые операции с простыми списками

Первой операцией, которую нужно выполнить перед тем, как начать работать со списком, является операция его создания:

инициализация

<p><b>procedure</b> Createlist (<b>var</b> L: list);</p> <p><i>предусловие:</i> Отсутствует.</p> <p><i>постусловие:</i> Список L был создан и инициализирован как пустой.</p>	{ Создать список }
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------

Следующая операция получает уже имеющийся список и очищает его.

повторная  
инициализация

**procedure** ClearList (**var** L: list); { Очистить список }  
*предусловие:* Список L уже создан.  
*постусловие:* Все элементы списка L удалены; список L пуст.

Далее идут операции проверки состояния списка.

операции  
проверки  
состояния списка

**function** ListEmpty (L: list): Boolean; { Список пуст? }  
*предусловие:* Список L уже создан.  
*постусловие:* Функция возвращает true или false в зависимости от того, пуст ли (empty) список L или нет.

**function** ListFull (L: list): Boolean; { Список полон? }  
*предусловие:* Список L уже создан.  
*постусловие:* Функция возвращает true или false в зависимости от того, заполнен ли (full) список L или нет.

**function** ListSize (L: list): integer; { Размер списка }  
*предусловие:* Список L уже создан.  
*постусловие:* Функция возвращает число элементов в списке L.

Заметьте, что в этих спецификациях некоторые процедуры и функции, предназначенные для работы с простыми списками, принимают L как **var**-параметр, в то время как в других L является параметром-значением. Если L объявляется как **var**-параметр, как, например, в процедуре ClearList, процедура может изменять список. Если же L объявляется как параметр-значение, как в функции ListSize, подпрограмма не может вносить в список абсолютно никаких изменений.

параметры  
и эффективность

Определяя спецификации для операций над данными любого типа, мы, как правило, должны следить за тем, чтобы **var**-параметры использовались исключительно в тех случаях, когда подпрограмма должна как-то изменять параметр. Однако когда мы приступаем к деталям реализации и пишем фактический текст подпрограммы, мы обычно ради повышения эффективности принимаем иную точку зрения: если в языке Pascal используется параметр-значение, каждый раз при активизации подпрограммы создается новая копия этого параметра. Если мы работаем с большим списком или другой подобной структурой данных, такое копирование требует значительного времени и расхода памяти. Поэтому мы следуем рекомендациям, приведенным в разделе 1.3.3, и при фактической реализации простых списков используем L как **var**-параметр для всех подпрограмм. В будущем мы по той же причине будем использовать **var**-параметр и для простых структур данных.

#### Замечание

## 5. Дальнейшие операции над простыми списками

Работая с нашими простыми списками, мы будем включать новые элементы только в конец списка:



**procedure** AddList (x: listentry; **var** L: list); { *Добавить в список* }  
*предусловие:* Список L уже создан и не полон; x является элементом типа listentry.  
*постусловие:* Элемент x был включен в список L в качестве его последнего элемента.

просмотр  
и посещение

Обычно над списками выполняется еще одна операция, носящая название *просмотра* списка. *Просмотр* обозначает, что мы проходим список, начав с его первого элемента, и выполняем некоторую операцию над каждым элементом списка по очереди, вплоть до самого последнего. Какая именно операция выполняется над элементами списка, зависит от приложения. Для общности мы будем говорить, что мы *посещаем* (visit) каждый элемент списка. Итак, последняя процедура для обработки списка:

**procedure** TraverseList (**var** L: list; **procedure** Visit(**var** x: listentry));  
 { *Просмотр списка* }  
*предусловие:* Список L уже создан. Visit является процедурой, обрабатывающей элементы типа listentry, но Visit не может выполнять включения или исключения из списка L.  
*постусловие:* Действие, задаваемое процедурой Visit, было по очереди выполнено над всеми элементами списка L, начиная с первого элемента и до последнего.

параметры-  
процедуры

Да, это объявление процедуры выполнено по правилам стандартного языка Pascal; процедуры допускаются в качестве формальных параметров других процедур, хотя это свойство языка не часто используется в элементарном программировании и реализовано не во всех компиляторах. Как и в случае любых других параметров, параметр Visit представляет собой просто формальное имя, которое заменяется другой процедурой, фактическим параметром, когда активизируется операция TraverseList. Процедура Visit не обязательно должна существовать под этим именем; имя Visit обозначает ту процедуру, которая будет использоваться в процессе просмотра списка для обработки каждого его элемента. Если, например, у нас есть две процедуры Update(**var** x: listentry) и Modify(**var** x: listentry), и L есть список, тогда прикладная программа, для того чтобы выполнить ту или иную из этих операций над списком L, может содержать команды

TraverseList(L,Update) или TraverseList(L,Modify)

Если в процессе отладки понадобилось распечатать все элементы списка, то все, что нужно сделать — это включить в программу предложение TraverseList(L,Print), где Print(**var** x: listentry) выводит на экран один элемент списка.

В дальнейшем, продолжая разрабатывать программу Life, мы еще встретимся с несколькими приложениями операции Traverse.

**Замечание**

Turbo Pascal версии 5 и более поздней допускает использование процедур в качестве параметров, однако с синтаксисом, отличным от стандартного синтаксиса языка Pascal.



Использование имен процедур в качестве параметров — это еще одно усложнение, введенное нами в технику программирования. В качестве формального параметра процедуры Visit мы указываем только один параметр, элемент обрабатываемого списка. Однако может оказаться, что фактическая процедура, используемая вместо Visit, требует дополнительных переменных, которые в обычном случае были бы указаны в качестве ее дальнейших параметров. Поскольку фактический смысл этих параметров зависит от приложения, мы не можем указать их как часть объявления процедуры TraverseList и ее формального параметра Visit. Поэтому мы вынуждены использовать только один параметр (listentry) и в Visit, и в фактической процедуре, которая подставляется на место Visit. Эта фактическая процедура должна затем использовать вместо дополнительных параметров глобальные переменные. В некоторых случаях такая операция может привести к побочным эффектам, если эти глобальные переменные модифицируются процедурой.

### **Программистский принцип**

*Если использование глобальных переменных, а следовательно, и вероятность возникновения побочных эффектов, представляется неизбежным, тщательно документируйте и спецификации, и правила использования подпрограмм.*

## **2.2.2. Программа Main**

Опираясь на структуры данных, специфицированные для проекта Life, мы можем воспользоваться описанием метода, данным в разделе 2.1.2, и преобразовать это описание в текст главной программы, написанный на языке Pascal. С некоторыми исключениями объявления констант, типов и переменных будут такими же, как в разделах 2.1 и 2.2, вместе с соответствующими объявлениями из первой версии программы Life. Списки maylive и maydie содержат ячейки, являющиеся в текущем поколении кандидатами на оживление и гибель. Некоторые из этих ячеек, однако, могут попасть в список по ошибке. Поэтому после того, как «правильные» ячейки были переведены в состояние живых или мертвых, списки newlive и newdie содержат ячейки, которые были фактически оживлены или убиты.

Life2, главная  
программа

```
program Life2 (input, output);           {Программа "Жизнь", вариант 2}
{ Pre: Пользователь предоставляет начальную конфигурацию живых
      ячеек.
  Post: Программа выводит последовательность решеток,
        показывающих изменения в конфигурации живых ячеек согласно
        правилам игры Life. Вариант 2.
  Uses: Использует пакеты утилит и обслуживания списков; процедуры
        Vivify, Kill, Initialize, Writemap, AddNeighbors и SubtractNeighbors. }
uses Utility; {откомпилированный модуль, специфичный для Turbo Pascal }
const
  maxrowbound = 21;                      { максимальный размер решетки }
  maxcolbound = 61                        { зависит от размера экрана }
  maxlist = 300; { максимальный допустимый размер каждого из четырех
                  списков }
```

объявления

```

type
  rowrange = 0..maxrowbound;      { включает на концах строки ограды }
  colrange = 0..maxcolbound;
  state = (alive, dead);           { состояние ячейки }
  grid = array [rowrange, colrange] of state;
  gridcount = array [rowrange, colrange] of integer; { число соседей }
  cell = record row: rowrange; col: colrange end; { координаты решетки }
  listentry = cell;
{ $I simplist.seg }                { специфично для Turbo Pascal: включает пакет
                                   для простых списков }

```

**var**

```

  map: grid;      { глобальная: квадратный массив, содержащий ячейки }
  numbernbrs: gridcount; { глобальная: квадратный массив,
                                   содержащий отсчеты числа соседей }
  newlive,        { глобальная: ячейки, которые только что ожили }
  newdie,         { глобальная: ячейки, которые только что умерли }
  maylive, { глобальная: кандидаты на оживление в следующем поколении }
  maydie: list; { глобальная: кандидаты на гибель в следующем поколении }
  maxrow: rowrange; { глобальная: определяемый пользователем размер
                                   решетки }
  maxcol: colrange;

```

инициализация

```

{ Объявления процедур будут включены сюда }
begin { главная программа, Life2 }

```

главный цикл

```

  Initialize(map, numbernbrs, newlive, newdie, maylive, maydie);
  WriteMap(map);
  write('Продолжим демонстрацию');
  while UserSaysYes do begin
    TraverseList(maylive, Vivify); { использует numbernbrs, изменяет
                                   map и newlive }
    TraverseList(maydie, Kill);    { использует numbernbrs, изменяет
                                   map и newdie }

    WriteMap(map);
    ClearList(maylive);
    ClearList(maydie);
    TraverseList(newlive, AddNeighbors); { изменяет numbernbrs,
                                           maylive, maydie }

    TraverseList(newdie, SubtractNeighbors);
    ClearList(newlive);
    ClearList(newdie);
    write('Хотите продолжить наблюдение новых поколений?')
  end

```

```

end. { главная программа, Life2 }

```

описание

Большую часть операций главная программа передает различным процедурам. После инициализации всех списков и массивов программа входит в главный цикл. В каждом поколении мы сначала проходим по ячейкам, ожидающим в списках `maylive` и `maydie`, в целях обновления массива `map`, который, как и в первой версии `Life`, отслеживает живые ячейки. Эти действия выполняются в процедурах `Vivify` и `Kill`. После записи исправленной конфигурации мы обновляем отсчеты числа соседей для каждой ячейки, которая либо родилась, либо погибла, с помощью процедур `AddNeighbors` и `SubtractNeighbors` и массива `numbernbrs`. В качестве

ве части той же процедуры, когда отсчет числа соседей достигает соответствующего значения, ячейка добавляется к списку `maylive` или `maydie`, что указывает на ее переход в живое или мертвое состояние в следующем поколении. Используя элементы списка, мы очищаем его, готовя для работы со следующим поколением.

Следует особо отметить использование процедуры `TraverseList`, вызываемой четыре раза для просмотра списка и выполнения одного из действий `Vivify`, `Kill`, `AddNeighbors` или `SubtractNeighbors` над каждым элементом списка. Каждая из этих четырех процедур, в свою очередь, является фактическим параметром, который заменяет формальный параметр `Visit`, указанный в объявлении процедуры `TraverseList`.

Заметьте также, что каждая из этих четырех процедур не только использует элемент из просматриваемого списка, но также и модифицирует (конкретно, добавляет элемент) один или несколько других списков. `Vivify`, например, использует элемент из `maylive` и добавляет ожившие ячейки к `newlive`. Параметры для этих четырех процедур не указываются в главной программе, но синтаксис языка Pascal требует, чтобы все четыре процедуры имели *те же* параметры, что и формальная процедура `Visit`, а `Visit` была объявлена с единственным параметром, именно, обрабатываемым ею элементом списка. Отсюда следует, что все четыре процедуры `Vivify`, `Kill`, `AddNeighbors` и `SubtractNeighbors` могут иметь только один этот параметр, даже хотя они модифицируют и другие списки. Другими словами, синтаксис языка Pascal *заставляет* нас написать тексты этих процедур так, чтобы они изменяли списки, имена которых им известны как глобальные переменные. т. е. мы вынуждены использовать плохой стиль программирования, приводящий к побочным эффектам. Если такой прием оказывается необходим, становится вдвойне важным тщательно документировать наши действия, особенно при составлении спецификаций для каждой процедуры.

параметры-  
процедуры

#### Замечание

### 2.2.3. Соккрытие информации

использование  
процедур

Обратите внимание на то, что мы смогли написать нашу главную программу для проекта Life и даже использовали пакет обработки списков, хотя мы еще и не думали о том, каким образом будут фактически реализованы списки в памяти и не знаем, каковы будут детали различных процедур и функций. Это пример **сокрытия информации**: если кто-то уже написал процедуры и функции для обработки списков, мы можем использовать их, и при этом нам не надо знать, как хранятся этих списки в памяти или как фактически выполняются операции над списками.

Фактически мы уже использовали принцип сокрытия информации в наших предыдущих программах, хотя и делали это бессознательно. Когда мы писали программы, использующие массив или запись, мы довольствовались применением операций над этими структурами, не рассуждая о том, как компилятор языка Pascal фактически располагает эти данные в битах и байтах памяти компьютера, или какие машинные команды выполняет процессор для поиска элемента по индексу или выбора поля за-

встроенные  
структуры  
данных

писи. Единственным реальным отличием сокрытия информации, касающейся массивов и записей, от сокрытия информации относительно списков, является то обстоятельство, что Pascal предоставляет встроенные операции для массивов и записей и не имеет таковых для списков.

альтернативные  
реализации

Любопытно, что некоторые компьютерные языки имеют списки в числе встроенных типов данных. Если бы мы использовали один из этих языков, мы могли бы выполнять операции над списками так же просто, как операции над массивами или записями. Имеется, однако, веская причина, по которой Pascal не предоставляет списки в качестве встроенного типа. В последующих главах мы увидим, что для списков (как и почти для всех типов данных, которые мы будем изучать) можно предложить несколько различных способов представления данных в памяти компьютера и несколько различных способов выполнения операций.

В некоторых приложениях оказывается лучше один метод, в то время как в других приложениях может иметь преимущества другой метод. В языке Pascal мы можем выбрать, какой метод лучше для нашего приложения, и затем подключить к программе процедуры и функции, реализующие выбранный метод.

смена  
реализации

Даже в одиночной большой программе мы можем сначала остановиться на представлении списков одним способом, а затем, приобретая опыт работы с программой, решить, что другой способ будет лучше. Если бы при этом код управления списками писался заново при каждом использовании списка, тогда пришлось бы изменить все включения этого кода в программу. Если же мы скрываем информацию путем использования отдельных процедур и функций управления списками, для замены метода достаточно изменить объявления.

ясность  
программы

Еще одно преимущество сокрытия информации при операциях над списками заключается в том, что любое появление в программе слов вроде `AddList` или `ListEmpty` сразу же дает представление тому, кто читает программу, о выполняемых в программе действиях, в то время как смысл самих команд может оказаться значительно более туманным.

нисходящее  
проектирование

В качестве последнего преимущества укажем на то, что отделение использования структур данных от их реализации окажет помощь в совершенствовании методом нисходящего проектирования как структур данных, так и наших программ.

## 2.2.4. Детализация: разработка подпрограмм

спецификации  
и решение задачи

После того, как мы набросали план решения поставленной задачи, наступает время обратиться к отдельным частям этого плана с целью включения в него больших деталей и получения тем самым более точной спецификации решения. Выполняя, однако, эту детализацию, программист часто обнаруживает, что назначение каждой подпрограммы не было определено с достаточной точностью, что интерфейс между подпрограммами нуждается в переработке и более детальном описании, чтобы каждая подпрограмма могла выполнить поставленную перед ней задачу, не дублируя действия других подпрограмм и не вступая с ними в противоре-

чия. Можно сказать, что процесс детализации возвращает нас назад к этапу решения задачи, где мы должны найти наилучший способ распределения всей работы между различными подпрограммами. В идеале этот процесс детализации и спецификации должен быть завершен перед тем, как мы приступим к написанию конкретного кода.

Проиллюстрируем эти действия рассмотрением требований для подпрограмм программы Life.

### 1. Задача процедуры AddNeighbors

Большая часть работы нашей программы выполняется в процедурах AddNeighbors и SubtractNeighbors. Разработаем первую из этих процедур, оставив вторую для упражнения. Процедуре AddNeighbors передается (в качестве ее параметра) один элемент из списка newlive. AddNeighbors находит ближайших соседей этой ячейки (как это делалось и в исходной функции NeighborCount), увеличивает отсчет в массиве numbernbrs для каждого из этих соседей, после чего должна поместить некоторых из соседей в списки maylive и maydie. Для определения конкретных претендентов давайте обозначим символом  $n$  обновленный отсчет для одного из соседей и обсудим возможные случаи.

1.  $n$  не может быть равным 0, поскольку мы только что увеличили  $n$  на 1.
2. Если  $n = 1$  или  $n = 2$ , тогда ячейка уже мертва и должна оставаться мертвой в следующем поколении. Следовательно, мы ничего не делаем.
3. Если  $n = 3$ , тогда ячейка, если она ранее была живой, продолжает жить; если она была мертвой, ее следует добавить в список maylive.
4. Если  $n = 4$ , тогда ячейка, если она ранее была живой, умирает; добавим ее в список maydie. Если ячейка мертва, она остается мертвой.
5. Если  $n > 4$ , тогда ячейка уже мертва (или уже находится в списке maydie) и остается мертвой.

### 2. Проблемы

При выполнении этой процедуры возникает одна тонкая проблема. Когда отсчет числа соседей для мертвой ячейки достигает 3, мы добавляем ее в список maylive, однако вполне может получиться так, что позже в процедуре AddNeighbors отсчет числа соседей для этой ячейки снова будет увеличен (и станет больше 3), и тогда она не должна ожить в следующем поколении. Точно так же, когда отсчет числа соседей для живой ячейки достигнет 4, мы добавляем ее в список maydie, но процедура SubtractNeighbors вполне может уменьшить для нее отсчет числа соседей, сделав его меньше 4, и мы должны будем удалить ее из maydie. Таким образом, окончательное решение относительно состава списков maylive и maydie не может быть принято до полного обновления массива numbernbrs, однако пока что мы должны в предварительном порядке добавлять элементы в списки.

Оказывается, если мы отложим решение этой проблемы, она заметно упрощается. Давайте добавлять ячейки в списки maylive и maydie в процедурах AddNeighbors и SubtractNeighbors, не заботясь об их возможном по-

случаи для  
AddNeighbors

ложные элементы

отложенные  
трудности

следующем удалении. Тогда, используя `maylive` и `maydie` при выполнении процедур `Vivify` и `Kill`, соответственно, мы можем проверять правильность отсчета числа соседей (например, в `maylive` могут находиться мертвые ячейки только с отсчетом числа соседей, в точности равным 3), и без труда игнорировать ошибочные элементы.

Однако после выполнения всех этих действий остаются еще более тонкие ошибки. Вполне возможно появление в списке `maylive` (или `maydie`) одной и той же ячейки более одного раза. Например, мертвая ячейка может первоначально иметь отсчет числа соседей, равный 2, что при увеличении отсчета добавит эту ячейку в список `maylive`. После этого отсчет для этой ячейки может еще возрасти, а в `SubtractNeighbors` уменьшиться один или несколько раз, остановившись, возможно на числе 3, в результате чего `SubtractNeighbors` опять добавит эту ячейку в список `maylive`. В этом случае в процессе обновления отсчетов числа ячеек в следующем поколении это рождение ошибочно добавит к отсчету не 1, а 2. Мы могли решить эту проблему, если перед тем, как использовать список, просматривать его в поисках дубликатов, однако на это уйдет много времени, и здесь, как и ранее, проблема упрощается, если отложить ее решение. Если в следующем поколении мы захотим оживить ячейку, мы должны сначала проверить, не жива ли она уже. Если это так, то мы знаем, что данный элемент является дубликатом уже имеющегося в списке `maylive`, и, опять же, как и ранее, мы игнорируем этот элемент.

На рис. 2.2 показана трассировка хода программы Life2 для одной простой конфигурации с демонстрацией появления и удаления ошибочных и дублирующих элементов в различных списках. Поведение этой конфигурации рассмотрено для решетки, имеющей всего четыре строки.

### Программистский принцип

Иногда откладывание задачи упрощает ее решение

Приняв все эти решения, мы можем наконец привести точные спецификации для четырех ключевых процедур. Спецификации для `SubtractNeighbors` и `Kill` оставлены для упражнений. Для `AddNeighbors` и `Vivify` мы получаем:

```
procedure AddNeighbors(var current: cell);           { Добавить соседей }
```

*предусловие:* `current` — это ячейка, которая только что стала живой.

*постусловие:* В списке numbernbrs увеличились отсчеты для всех ячеек, соседних с current. Если увеличенный отсчет числа соседей делает ячейку кандидатом на оживление [соответственно на гибель], тогда эта ячейка добавлена в список maylive [соответственно в maydie].

**использование:** Использует процедуру `AddList`; модифицирует массив `numberrbrs` и списки `maylive` и `maydie` как глобальные переменные (побочные эффекты).

дублированные  
элементы

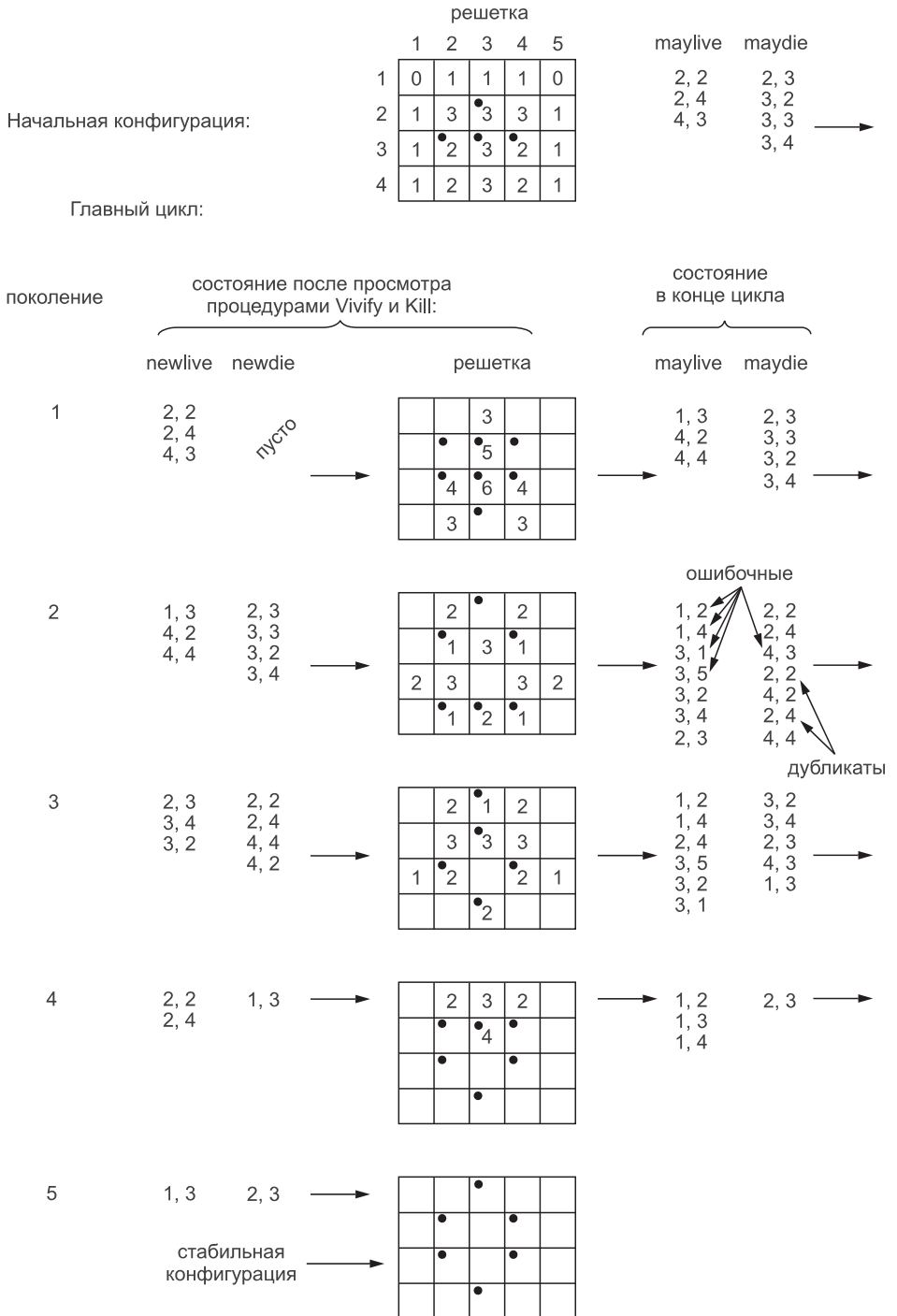


Рис. 2.2. Трассировка программы Life2



<p><b>procedure</b> Vivify(<b>var</b> current: cell); <span style="float: right;">{ Оживить }</span>  <i>предусловие:</i> Ячейка cell является кандидатом на оживление.  <i>постусловие:</i> Проверяет, удовлетворяет ли ячейка current всем требованиям на оживление. Если нет, модификация не выполняется. Если да, ячейка current добавляется в список newlive, и массив map обновляется.  <i>использование:</i> Процедура AddList, массив numbernbrs, модифицирует массив map и список newlive как глобальные переменные (побочные эффекты).</p>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### 2.2.5. Верификация и алгоритмы

цель

Другим важным аспектом разработки больших программ является верификация алгоритма, т. е. *доказательство* того, что алгоритм выполняет свою задачу. Это доказательство обычно формулируется путем просмотра спецификаций подпрограмм и выяснения, приведет ли объединение всех этих спецификаций к выполнению задачи, решаемой алгоритмом в целом. Формулируя доказательство, мы можем обнаружить, что для реализации требуемого алгоритма необходимо изменить спецификации, и, таким образом, верификация алгоритма помогает нам с большей точностью сформулировать спецификации для каждой подпрограммы. Отсюда следует, что верификация и разработка алгоритма должны идти рука об руку, и иногда верификация может даже направлять разработку. В любом случае верификация алгоритма должна предшествовать кодированию.

Давайте опять проиллюстрируем использование этих принципов на примере программы Life, сначала для того, чтобы убедиться в правильности ее алгоритма, а затем в качестве содействия при разработке последней оставшейся процедуры Initialize.

тщательность  
разработки  
алгоритмов

Тот факт, что наша первоначальная попытка организации работы процедур Vivify, Kill, AddNeighbors и SubtractNeighbors встретила со скрытыми трудностями, должен послужить предостережением о возможном наличии в программе других ошибок или, по крайней мере, о необходимости относиться к разработке алгоритмов со значительно большей внимательностью.

#### 1. Возможные проблемы

Отложив проверку отсчетов числа соседей, мы смогли преодолеть трудности решения проблем как дублирующих, так и ошибочных элементов массивов. Однако не может ли случиться, что, например, одна и та же ячейка будет ошибочно включена в оба массива, и maylive, и maydie? Если это произойдет, она может сначала ожить, а затем сразу же погибнуть в том же поколении (событие, очевидно, недопустимое). Ответ на этот конкретный вопрос будет отрицательным, поскольку главная программа использует процедуры Vivify и Kill перед вызовом процедур AddNeighbors и SubtractNeighbors. В результате ячейка сохраняет свое состояние (живая или мертвая) от конца просмотра процедурой Kill и до следующего поколения, а процедуры AddNeighbors и SubtractNeighbors



проверяют только мертвые ячейки, добавленные в `maylive`, и только живые, добавленные в `maydie`.

Каким образом можно убедиться, что по ходу выполнения программы не возникнут еще какие-то тонкие вопросы такого же рода, на некоторые из которых нам, возможно, найти ответ будет не так легко? Единственным способом получить уверенность в отсутствии таких проблем является *доказательство* того, что наша программа выполняет в каждом случае правильные действия.

## 2. Главный цикл

Основная проблема, которая может возникнуть в нашей программе, заключается в том, что происходящее в одном поколении может повлиять на следующее поколение некоторым неожиданным образом. Отсюда следует, что мы должны внимательно рассмотреть большой цикл в главной программе. Когда цикл начинает свою работу, то все, что случится в дальнейшем, определяется списками `maylive` и `maydie`. Давайте подытожим все, что знаем об этих списках, исходя из предыдущего рассмотрения.

инвариант цикла

*В начале главного цикла список `maylive` содержит только мертвые ячейки, а список `maydie` только живые, однако списки могут включать дублирующие элементы, а также ошибочные элементы с неправильными отсчетами числа соседей. Списки `newlive` и `newdie` пусты.*

В самом начале программы задачей процедуры `Initialize` является обеспечение правильной установки всех четырех списков, поэтому предыдущие утверждения верны в начале первого поколения. Мы, следовательно, должны доказать, что если эти утверждения верны в начале любого поколения, то после девятикратного вызова процедуры внутри цикла они опять окажутся верными для следующего поколения.

## 3. Доказательство посредством математической индукции

начальный случай

Легко сообразить, что наши текущие действия являются применением метода *математической индукции* для доказательства правильности программы. Согласно этому методу доказательства, мы начинаем с того, что устанавливаем правильность результата для начального случая. Далее мы доказываем правильность результата для более позднего случая, скажем, случая  $n$ , используя результат ранних случаев (расположенных между начальным случаем и случаем  $n - 1$ ). Для программы `Life` в качестве номера случая используем номер поколения.

Верификация начального случая сводится к верификации того, что процедура `Initialize` работает правильно. Эта часть доказательства сводится к написанию правильной спецификации для этой процедуры.

шаг индукции

В качестве второй части доказательства правильности шага индукции, проанализируем действия внутри главного цикла, предположив, что наши утверждения справедливы в его начале. Просмотр с помощью процедуры `Vivify` использует только список `maylive`, и, согласно спецификации для `Vivify`, функция тщательно проверяет каждый элемент перед тем, как оживить ячейку, помещая в список `newlive` только истинно оживлен-

ные ячейки. Таким образом, при завершении первого просмотра список `newlive` содержит только те ячейки, которые были должным образом оживлены, при этом дубликаты в списке отсутствуют.

Просмотр с помощью процедуры `Kill` аналогично использует список `maydie` и устанавливает список `newdie` так, что в нем содержатся только погибшие перед этим ячейки. Поскольку списки `maylive` и `maydie` первоначально ни имели общих ячеек, и такие ячейки в эти списки не добавлялись, можно утверждать, что в них не будет ячеек, одновременно оживленных и погибших.

Далее вызывается процедура `WriteMap`, но она не изменяет списки.

Процедура `AddNeighbors` получает данные только из списка `newlive` и помещает в список `maylive` только мертвые ячейки, а в список `maydie` — только живые. Аналогично, процедура `SubtractNeighbors` надежно разделяет мертвые и живые ячейки. Вместе эти две процедуры добавляют в списки все ячейки, состояние которых изменится в следующем поколении, но могут также добавить дублирующие или ложные элементы.

Наконец, списки `newlive` и `newdie` очищаются, в результате чего условия наших утверждений в начале следующего поколения снова становятся истинными. Следовательно, логика нашей программы правильна.

#### 4. Инварианты и утверждения

**Утверждения** вроде того, которое мы подтвердили предыдущим доказательством, называются **инвариантами цикла**. В общем случае инвариант цикла — это утверждение, справедливое в начале каждой итерации цикла. Применительно к программе `Life` утверждениями являются состояния наших списков в разных точках цикла.

#### 5. Инициализация

Наша неформальная верификация главного цикла программы `Life` имеет еще одно полезное качество: сформулированный нами инвариант цикла естественным образом приводит к спецификациям для процедуры инициализации, последней нерассмотренной еще существенной процедуры проекта.

```
procedure Initialize(var map: grid; var numbrnbrs: gridcount;
                    { Инициализировать }
                    var newlive, newdie, maylive, maydie: list);
```

*предусловие:* Отсутствует.

*постусловие:* Массив `map` содержит начальную конфигурацию живых и мертвых ячеек. Массив `numbrnbrs` содержит отсчеты числа живых соседей, соответствующие конфигурации в массиве `map`. Список `maylive` содержит только мертвые ячейки и включает всех кандидатов, которые могут быть оживлены в следующем поколении. Список `maydie` содержит только живые ячейки и включает всех кандидатов, которые могут умереть в следующем поколении. Списки `newlive` и `newdie` пусты.

*использование:* Использует пакет для простых списков, функцию `ListSize`, процедуры `ReadMap`, `AddNeighbors`.

## 6. Заключение

упрощение

Назначением инвариантов цикла и утверждений является выражение сущности динамического процесса. Не всегда легко найти такие инварианты цикла и утверждения, которые приведут к доказательству правильности программы, однако поиск их является весьма полезным упражнением. Попытка найти инварианты цикла и утверждения иногда приводят к упрощению структуры алгоритма, что облегчает доказательство его правильности. Всегда надо ставить перед собой цель сделать наши алгоритмы максимально простыми и ясными, чтобы правильность их логики была очевидна, и использование инвариантов цикла помогает в достижении этой цели.

Верификация алгоритмов является предметом активных исследований, и много важных вопросов в этой области остаются пока без ответа. Для большого количества широко используемых на практике алгоритмов еще не найдены доказательства правильности. Иногда возникают исключительные ситуации, в которых алгоритм работает неправильно; доказательства правильности могли бы выделить эти исключительные ситуации и обеспечить их обработку.

Одним из важнейших применений этих формальных методов является получение информации о том, как данный алгоритм будет себя вести при точном задании спецификаций, что мы как раз и сделали для процедуры `Initialize`.

### Упражнения 2.2

Составьте и запишите пред- и постусловия для каждой из следующих процедур:

- E1. procedure** `SquareRoot(x: real; var y: real)`; присваивает переменной `y` значение квадратного корня из `x`.
- E2. function** `Mean (var A: list): real`; вычисляет среднее значение элементов списка, представляющих собой действительные числа (другими словами, `listentry = real`).
- E3. procedure** `Kill(var current: cell)`;
- E4. procedure** `SubtractNeighbors (var current: cell)`;
- E5. procedure** `WriteMap (var map: grid)`;
- E6. procedure** `ReadMap (var newlive: list; var map: grid)`;

## 2.3. Кодирование

спецификации  
определены

Теперь, когда мы разработали полные и точные спецификации для каждой процедуры, наступило время описания их на выбранном нами языке программирования. При разработке большого программного проекта важно начать кодирование в нужный момент времени, не слишком рано, но и не слишком поздно. Ошибкой большинства программистов является то, что они начинают кодирование проекта слишком рано. Если к кодированию приступить еще перед тем, как разработаны точные спецификации, то неизбежно в процессе кодирования придется делать необоснованные предположения касательно спецификаций, и эти предположения

приведут к несовместимости отдельных подпрограмм друг с другом или к существенному усложнению процесса программирования.

**Программистский принцип**

*Никогда не приступайте к кодированию  
до полного и точного описания спецификаций.*

нисходящее  
кодирование

С другой стороны, возможно, хотя и маловероятно, что вы слишком промедлите с кодированием. Точно так же, как мы проектировали программы в нисходящем направлении, сверху вниз, нам следует и кодирование выполнять в том же направлении. Как только спецификации верхнего уровня будут отработаны, нам следует заняться кодированием подпрограмм этого уровня и затем выполнить их тестирование, включив соответствующие заглушки. Если мы обнаружим ошибку в проекте, мы сможем его модифицировать, не платя чрезмерную цену за процедуры нижнего уровня, которые теперь могли бы оказаться бесполезными. В настоящем разделе мы займемся программированием пакета обработки простых списков, а в следующем завершим проект Life.

### 2.3.1. Пакет обработки списков

Обратите внимание на то, что хотя мы уже в течение некоторого времени использовали операции со списками, у нас не возникало необходимости рассматривать вопросы реализации пакета обработки списков. Фактически имеются две стандартных реализации для списков, и обе можно использовать в проекте Life. В этом разделе мы разработаем **непрерывную** реализацию, в которой элементы списка хранятся в памяти друг за другом внутри массива. **Связная** реализация использует указатели языка Pascal и динамическое выделение памяти вместо массива. Простые связанные списки рассматриваются в приложении D.

#### 1. Хранение данных

При непрерывной реализации списков мы объявляем массив для хранения элементов, и нам нужен еще отдельный счетчик для указания, сколько позиций массива занято элементами списка. Эти данные можно объединить в одну структурированную запись<sup>1</sup> следующим объявлением:

```
type position = 1..maxlist;  
listcount = 0..maxlist;  
list = record  
    count: listcount;  
    entry: array [position] of listentry;  
end;
```

#### 2. Пробные операции

Кодирование операций над простыми списками выполняется достаточно просто, и большую часть их мы оставим для упражнений. Ниже приведен текст процедуры включения в список нового элемента.

<sup>1</sup> Структурированные записи языка Pascal рассматриваются в приложении D.

```

procedure AddList(x: listentry; var L: list);           { Добавить в список }
{ Pre:   Список L уже создан и не полон; x является элементом типа
        listentry.
  Post:  Элемент x включен в качестве последнего элемента L. }
begin
  if ListFull(L) then
    Error('Попытка включения в конец полного списка.')
  else begin
    L.count := L.count + 1;
    L.entry [L.count] := x;
  end
end;

```

Процедура для просмотра списка использует процедурный параметр Visit; формы объявления такого параметра различаются в языках Turbo Pascal и стандартный Pascal:

```

{ Для Turbo Pascal: }
procedure TraverseList(var L: list; Visit: processentry); { Просмотр списка }
{ Для стандартного языка Pascal:
procedure TraverseList(var L: list; procedure Visit(var x: listentry)); }
{ Pre:   Список L уже создан.
  Post:  Действие, задаваемое процедурой Visit, было выполнено
        над каждым элементом списка L, начиная от первого элемента
        и далее над каждым элементом по очереди. }

var p: position;
begin
  for p := 1 to L.count do
    Visit(L.entry [p])
end;

```

### 2.3.2. Обработка ошибок

Обратите внимание на то, что процедура AddList вызывает другую процедуру, с именем Error, в случае, если она не может выполнить свою работу из-за того, что список заполнен. Здесь мы впервые встречаемся с примером обработки ошибок, важным средством, которое следует по возможности включать в наши программы.

Существуют два способа обработки ошибок. В коммерческих программах, которые проходят процедуру тщательной отладки, возникновение ошибки обычно указывает на какую-то очень серьезную неисправность, так что ошибки чаще всего оказываются фатальными для дальнейшего выполнения программы. В этом случае программа выведет сообщение об ошибке и аварийно завершится. Однако при разработке и отладке новой программы вероятность возникновения ошибок весьма велика. Мы, следовательно, должны принять другой подход к обработке ошибок. Вместо аварийного завершения программы ошибка должна заставить программу вывести информативное сообщение с указанием, в чем заключается причина сбоя. После этого программа пропускает дефектную операцию и переходит к следующей. Такая процедура позволит нам продолжить изучение программы и больше узнать о причинах сбоя.

**Замечание**

Разумеется, программист отвечает за исправление программы, так что в конце концов она будет выполняться без всяких сообщений об ошибках. После достижения такого положения подпрограммы проекта можно переработать в коммерческий вариант, в котором при возникновении ошибки программа аварийно завершается.

Мы будем во всех случаях при возникновении ошибки вызывать процедуру `Error`, которая в качестве единственного параметра принимает строку, выводит сообщение об ошибке и возвращается в вызвавшую программу. Соответственно, мы включим эту процедуру обработки ошибок в пакет утилит, который обсуждался в разделе 1.4.3, чтобы она была доступна для всех программ, которые мы будем писать. При использовании языка Turbo Pascal, для передачи сообщения об ошибке в процедуру следует включить объявление `type errmsg = string [80]`.

```

procedure Error (message: errmsg);                                { Ошибка }
{ Pre:  Предусловия отсутствуют.
  Post: Выводит на экран строку ERROR с сообщением message,
         задаваемым в качестве параметра. }
begin
    writeln('ERROR', message)
end;                                                                { процедура Error }

```

### 2.3.3. Демонстрация и тестирование

Написав пакет процедур и функций для обработки структур данных, мы должны немедленно приступить к тестированию этого макета, чтобы убедиться, что все его составляющие работают должным образом. Одним из лучших способов выполнения такого тестирования является написание *демонстрационной программы, управляемой меню*, которая будет инициализировать структуры данных и позволять пользователю выполнять над ними все допустимые операции в любом порядке, выводя при этом на экран результаты по заказу пользователя на любом этапе тестирования. Давайте разработаем такую программу для пакета обработки простых списков. Эта программа послужит нам в будущем основой для аналогичных программ тестирования других структур данных, с которыми мы будем иметь дело в этой книге.

Мы можем определить элементы списка как данные любого типа, поэтому для простоты будем считать, что мы используем списки символов. Таким образом, в качестве элементов списков будут выступать одиночные буквы, цифры, знаки препинания и прочие знаки. Обозначим именем `CharSL` пакет обработки простых списков с входными данными типа `char`, откомпилированный в качестве модуля Turbo Pascal. Для других систем потребуется вместо модулей использовать включаемые файлы.

В каждом шаге главного цикла программа запрашивает у пользователя выбор требуемой операции. Затем программа (если это возможно) выполняет указанную операцию над структурой данных и выводит результат.

демонстрационная  
программа,  
управляемая меню





В программе имеется также процедура `Introduction`, используемая только один раз при запуске программы. Эта процедура кратко описывает назначение программы и подсказывает пользователю, как начать с ней работать. Дальнейшие команды поступают либо от `Help`, либо от `ReadCommand`.

Процедура `GetCommand` выводит меню, получает от пользователя команду, проверяет ее допустимость и преобразует ее в символ нижнего регистра (т. е. в строчную букву) с помощью процедуры из пакета `Utility`, описанного в приложении С.

```

procedure GetCommand(var ch: char);           { Получить команду }
{ Pre:  Предусловия отсутствуют.
  Post:  Получает допустимую команду с клавиатуры и возвращает ее
         через переменную ch.
  Uses: Использует процедуру LowerCase из пакета Utility. }
begin                                           { процедура GetCommand }
  writeln(' [I]nsert Ввод элемента           [P]rint Вывод списка');
  writeln(' [S]ize Размер списка             [D]elete Удаление списка');
  writeln(' [C]lear Очистка списка           [H]elp Справка');
  writeln(' [T]raverse Просмотр списка      [Q]uit Выход');
  repeat
    readln(ch);
    LowerCase(ch);
    if not (ch in ValidSet) then
      write('Пожалуйста, введите допустимую команду или H ');
      write('для получения справки');
    until ch in ValidSet
  end;                                           { процедура GetCommand }

```

Действия по анализу и выполнению команд выполняются процедурой `DoCommand`. Часто желательно иметь программу, в которую можно включать дополнительные вставки, и процедура `DoCommand` поэтому написана в виде цикла, допускающего такое расширение.

```

procedure DoCommand(var ch: char; var L: list); { Выполнить команду }
{ Pre:  ch содержит допустимую команду.
  Post: Выполняет команду ch над списком L.
  Uses: Использует пакет обслуживания списков. }
var x: listentry;                               { используется для ввода нового элемента }
begin                                           { процедура DoCommand }
  case ch of
    'I': if ListFull(L) then                   { ввод нового элемента }
      writeln('Извините, список полон.');
```

```

    else begin
      write('Новый ключ (ключи) для включения:');
      while not (eoln or ListFull(L)) do begin
        read(x);
        AddList(x,L)
      end;
      if not eoln then
        writeln('Список полон, ввод новых элементов невозможен.');
```

```

      readln
    end;

```



```

'd', 'c': begin                                     { удаление элементов }
    ClearList(L);
    writeln('Список очищен.')
end;
'p', 't': if ListEmpty(L) then                     { вывод элементов на экран }
    writeln('Список пуст.')
else begin
    writeln;
    writeln(' Просмотр списка; список содержит: ');
    TraverseList(L, Print);      { вызов процедурного параметра }
    writeln;
end;
's': writeln(' Размер списка равен ', ListSize(L): 2');
'h': Help;                                         { справочный кадр }
'q':;                                             { не делать ничего }
end
end;                                              { процедура DoCommand }

```

Наконец, для распечатки списка по ходу его просмотра необходимо иметь процедуру в форме процедурного параметра Visit:

```

procedure Print(var x: listentry);                { Вывод на экран }
{ Pre:   x содержит допустимый элемент списка (символ).
  Post: Выводит на экран значение x. }
begin
    write(' ',x)
end;                                              { процедура Print }

```

Вы можете заметить, что во всех этих процедурах мы старались поддерживать принцип абстракции данных. Мы использовали пакет обработки списков в виде предварительно откомпилированного модуля, поэтому при желании мы можем заменить этот модуль другой реализацией, и программа будет работать точно так же. Мы также написали другие процедуры, так что мы можем использовать эту программу в будущем для тестирования других структур данных, не внося почти никаких изменений, кроме коррекции набора допустимых операций и содержания экранных кадров процедур Introduction и Help.

### Упражнения 2.3

- E1. Напишите код непрерывной реализации оставшихся операций для простых списков, следуя спецификациям, описанным в разделе 2.2.1:
  - (a) CreateList
  - (b) ClearList
  - (c) ListEmpty
  - (d) ListFull
  - (e) ListSize
- E2. Опишите изменения, которые необходимо внести в пакет обработки простых списков и в демонстрационную программу, управляемую меню, чтобы они могли работать с целыми числами вместо символов в качестве элементов списков.
- E3. Используя пакет обработки простых списков, напишите процедуру CopyList со следующими спецификациями:

**procedure** CopyList(**var** source, dest: list); { Копировать список }  
*предусловие:* Списки source и dest уже созданы.  
*постусловие:* Список dest становится точной копией списка source;  
список source не изменяется.

Напишите три версии ваших процедур:

- (a) Просто используйте предложение присваивания: `dest := source`.
- (b) Используйте процедуры из пакета обработки списков для просмотра списка source и добавления каждого его элемента в конец списка dest.
- (c) Используйте детали реализации и напишите цикл, который копирует элементы из source в dest.

Какую из этих трех процедур легче всего написать? Какая из них будет выполняться быстрее всего при почти полном списке? Какая из них будет выполняться быстрее всего при почти пустом списке? Какой метод лучше использовать, если может потребоваться изменить реализацию? Что будет неправильно работать при использовании метода (a), если список реализован как связный, а не как массив?

### Программные проекты 2.3

- P1. Доведите до рабочего состояния демонстрационную программу для простых списков, написав все отсутствующие процедуры, сформировав модуль (или включаемый файл) и оттестировав программу.
- P2. Напишите реализацию пакета обработки простых списков, которая использует связные списки вместо непрерывной реализации, разработанной в тексте. Оттестируйте проект, используя ваш модуль (или включаемые файлы) в демонстрационной программе, управляемой меню. [Связные списки описаны в приложении D, где полностью разработана большая часть процедур для этого проекта.]

## 2.4. Кодирование процедур программы Life

Имея спецификации для подпрограмм программы Life, мы легко можем преобразовать наши решения в процедуры, закодированные на языке Pascal.

### 1. Процедура Vivify

Процедура Vivify берет элемент из списка `maylive` и оживляет ячейку при условии, что она была перед этим мертва и имеет число соседей, в точности равное 3. В противном случае ячейка относится к числу ложных элементов и Vivify ее игнорирует. Если ячейка действительно оживлена, она добавляется в список `newlive`, в результате чего после завершения просмотра список `newlive` содержит в точности те ячейки, которые были оживлены, и у которых отсчет числа соседей должен быть обновлен процедурой `AddNeighbors`.

оживим ячейки

```

procedure Vivify(var current: listentry); { Оживить }
{ Pre: Ячейка current является кандидатом на оживление.
  Post: Проверяет, удовлетворяет ли ячейка current всем требованиям
        для оживления. Если это не так, не производится никаких
        изменений. Если так, тогда ячейка current добавляется в список
        newlive, а массив map обновляется.
  Uses: Использует процедуру AddList, массив numbernbrs; изменяет
        массив map и список newlive, как глобальные переменные
        (побочные эффекты). }

begin { процедура Vivify }
  with current do
    if (map[row, col] = dead) and (numbernbrs[row, col] = 3) then
      if (row in [1..maxrow]) and (col in [1..maxcol]) then { не в ограде }
        begin { оживим ячейку }
          map[row, col] := alive;
          AddList(current, newlive)
        end
    end; { процедура Vivify }

```

## 2. Процедура AddNeighbors

После завершения просмотра процедурой Vivify список newlive содержит ячейки, которые были фактически оживлены. Поэтому процедура AddNeighbors сможет без труда обновить все связанные с этой ячейкой отсчеты числа соседей. Эта процедура имеет следующий вид.

обновление  
отсчетов числа  
соседей

```

procedure AddNeighbors(var current: listentry); { Добавить соседей }
{ Pre: current является только что ожившей ячейкой.
  Post: В массиве numbernbrs увеличены отсчеты для всех ячеек,
        соседних с current. Если увеличенный отсчет числа соседей
        делает ячейку кандидатом на оживление [соответственно
        на гибель], тогда ячейка добавляется к списку maylive
        [соответственно maydie].
  Uses: Использует процедуру AddList; изменяет массив numbernbrs
        и списки maylive и maydie как глобальные переменные
        (побочные эффекты). }

var nbrrow: rowrange; { индекс цикла для циклов по строкам соседей }
    nbrcol: colrange; { индекс столбца цикла }
    neighbor: cell; { форма записи соседа }
begin { процедура AddNeighbors }
  with current do
    for nbrrow := row - 1 to row + 1 do
      for nbrcol := col - 1 to col + 1 do
        if (nbrrow <> row) or (nbrcol <> col) then begin
          { пропустим саму ячейку current }
          numbernbrs[nbrrow, nbrcol] := numbernbrs[nbrrow, nbrcol] + 1;
          case numbernbrs[nbrrow, nbrcol] of
            0: writeln('Недопустимый случай в AddNeighbors.');
            1, 2: ; { никаких действий не требуется }
            3: if map[nbrrow, nbrcol] = dead then begin
              neighbor.row := nbrrow;
              neighbor.col := nbrcol;
              { Установим запись с координатами }
              AddList(neighbor, maylive);
            end;
        end;

```

границы для цикла

найдем соседа

поместим ячейки  
в списки

```

4: if map [nbrrow, nbrcol] = alive then begin
    neighbor.row := nbrrow;
    neighbor.col := nbrcol;
                                { Установим запись с координатами }
    AddList(neighbor, maydie);
    end;
5, 6, 7, 8: ;                                { изменений нет }
end                                           { предложение case }
end                                           { обработка одного соседа }
end;                                         { процедура AddNeighbors }

```

### 3. Различные процедуры

Процедуры Kill и SubtractNeighbors схожи с Vivify и AddNeighbors; их мы оставим для упражнений. Процедуру WriteMap из первой версии можно использовать без изменений, хотя можно написать более эффективную версию, использующую для обновления экрана списки newlive и newdie вместо того, чтобы в каждом поколении перерисовывать весь кадр.

### 4. Инициализация

разработка  
процедуры  
Initialize

Переходя, наконец, к процедуре Initialize, давайте для облегчения ее составления используем постусловия. Первое постусловие требует, чтобы массив map был инициализирован начальной конфигурацией живых и мертвых ячеек. Эта задача схожа с инициализацией первой версии программы, а процедуру ReadMap мы оставляем для упражнений. Однако для вычисления отсчетов числа соседей нам понадобится список изначально живых ячеек, так что мы потребуем от процедуры ReadMap поместить этот список в список newlive. В качестве одного из постусловий процедуры ReadMap мы потребуем, чтобы эта процедура проверила этот список на отсутствие в нем дублирующих элементов. (Эту операцию легко осуществить, если чтение выполняется должным образом.)

Вторая задача заключается в инициализации отсчетов числа соседей в массиве numbernbrs. Однако мы потребовали от процедуры ReadMap такой инициализации списка newlive, при которой он содержит в точности ту информацию, которая необходима для процедуры AddNeighbors, чтобы она правильно установила отсчеты числа соседей для всех соседей живых ячеек, при условии, что перед вызовом AddNeighbors мы сначала установили 0 во всех элементах массива numbernbrs. Помимо инициализации numbernbrs, процедура AddNeighbors обнаружит все мертвые ячейки, которые станут живыми в следующем поколении, и добавит их в список maylive. В результате, установив список maylive в состояние «пуст», мы выполним другое постусловие процедуры Initialize.

Последнее постусловие заключается в том, что список maydie содержит все живые ячейки, которые должны умереть в следующем поколении. Некоторые из этих ячеек, хотя, возможно, не все, могут быть найдены процедурой AddNeighbors (в главном цикле, причем остаток будет найден процедурой SubtractNeighbor, которую мы не можем использовать в Initialize). Однако мы можем выполнить это постусловие и более простым способом, просто поместив все живые ячейки в список newdie. Вспомним, что процедура Kill может в своем входном списке иметь и

ложные элементы. Поэтому только в первом проходе мы позволяем процедуре Kill просмотреть все живые ячейки, чтобы найти те, которые должны умереть.

В результате наши постусловия приводят к следующей процедуре:

```

procedure Initialize (var map: grid; var numbbrs: gridcount;
                     var newlive, newdie, maylive, maydie: list);
                                { Инициализация }
Pre:  Предусловия отсутствуют.
Post: Решетка map содержит начальную конфигурацию живых и
        мертвых ячеек. Массив numbbrs содержит отсчеты числа
        живых соседей, соответствующие конфигурации в массиве
        map. Список maylive содержит только мертвые ячейки и
        включает всех кандидатов на оживление в первом поколении.
        Список maydie содержит только живые ячейки и включает всех
        кандидатов на гибель в первом поколении. Списки newlive и
        newdie пусты.
Uses: Использует пакет обслуживания списков, функцию ListSize,
        процедуры ReadMap, AddNeighbors. }
var row: rowrange;
        col: colrange;           { используется для установки всех элементов
                                в numbbrs в 0 }
begin                                { процедура Initialize }
    CreateList(newlive);
    CreateList(newdie);
    CreateList(maylive);
    CreateList(maydie);
    FindSize(maxrow, maxcol);      { начальное сообщение; устанавливает
                                границы для решетки. }
    ReadMap(newlive, map);         { получает начальную конфигурацию }
    for row :=0 to maxrow +1 do      { устанавливает все элементы
                                в numbbrs в 0 }

        for col :=0 to maxcol +1 do
            numbbrs [row, col] := 0;
        TraverseList(newlive, AddNeighbors); { помещает кандидатов
                                на оживление в maylive }
    maydie := newlive;             { проверяет все живые ячейки с целью
                                нахождения тех, которые могут умереть. }
    ClearList(newlive)
end;                                { процедура Initialize }

```

## Программные проекты 2.4

- P1.** Напишите недостающие процедуры для второй версии программы Life:
- (a) Kill (c) FindSize
  - (b) SubtractNeighbors (d) ReadMap
- P2.** Напишите программу-драйвер для процедуры ReadMap, чтобы убедиться в том, что она работает правильно.
- P3.** Напишите программы-драйверы для процедур:
- (a) Kill и
  - (b) SubtractNeighbors и разработайте подходящие тестовые данные для проверки работоспособности этих процедур.

## 2.5. Анализ и сравнение программ

При разработке алгоритмов нам нужно иметь методы, позволяющие отличать плохие алгоритмы от хороших. Эти методы должны помочь нам решить, какой из нескольких возможных путей разработки обеспечит наиболее эффективное решение нашей задачи.

### 1. Подсчет числа предложений

Посмотрим теперь, насколько быстрее программа Life2 будет выполняться в сравнении с предыдущей версией. Как и в случае первой версии, не будем учитывать время, требуемое главной программе для ввода и вывода, а рассмотрим только предложения внутри принципиального цикла просмотра поколений. Почти вся работа выполняется здесь в четырех предложениях, каждое из которых просматривает список. Таким образом, ключевое усовершенствование программы Life2 по сравнению со старым вариантом заключается в том, что объем вычислений теперь не пропорционален размеру решетки, а только числу изменений в ней. Для типичной конфигурации в решетке может быть около 50 занятых ячеек, и не более 25 ячеек рождаются или умирают в одном поколении. При таких предположениях можно увидеть, что каждый просмотр списка приведет приблизительно к 25 вызовам используемой им процедуры.

Поскольку почти вся работа программы Life2 выполняется внутри четырех процедур, используемых в этих просмотрах, мы должны по очереди проанализировать каждую из них. Процедура Vivify выполняет 4 предложения (два предложения `if`, одно присваивание и один вызов `AddList`). Внутри процедуры `AddNeighbors` имеются вложенные циклы с общим числом шагов, равным 9. Внутри циклов имеются: предложение `if`, присваивание и предложение `case`, которое само может включать 4 предложения. Таким образом, всего в `AddNeighbors` выполняются около 54 предложений. Объем процедур `Kill` и `SubtractNeighbors` такой же, и, следовательно, для каждого поколения мы получаем около

$$25 \times (4 + 54 + 4 + 54) = 2900$$

предложений. Число предложений, выполняемых вне циклов, незначительно (меньше 10), таким образом, 2900 является разумной оценкой числа предложений для каждого поколения.

Наша первая версия программы Life насчитывала 18000 предложений на поколение. Таким образом, пересмотренная программа будет выполняться приблизительно в 6 раз быстрее. Для медленного компьютера это даст значительное улучшение, особенно ввиду того факта, что если программа Life2 начинает работать медленнее, то это происходит из-за увеличенного количества изменений, а не потому, что она повторяет уже выполненные вычисления.

### 2. Сравнения

Однако с других точек зрения наша вторая программа не так хороша, как первая. Первый из этих аспектов касается объема программистской

число операций  
в Life2

число операций  
в Life1

объем  
программистской  
работы

работы. Первая программа была короткой и несложной в реализации, наглядной для понимания и простой в отладке. Вторая программа длиннее, в ней появились тонкие проблемы, и для их преодоления потребовались сложные рассуждения. Оправдана ли эта дополнительная работа, зависит от приложения и от того, сколько раз эта программа будет использоваться. Если простой метод позволяет получить работоспособную программу, нет особой необходимости трудиться и искать более совершенный подход. Только если простой метод не дает нужных результатов, мы должны исследовать другие подходы.

**Программистский принцип**

*Упрощайте, насколько это возможно, ваши алгоритмы.  
В случае сомнений используйте более простой способ.*

требования  
к памяти

Второй аспект касается расходования памяти. Наша первая программа использовала очень мало памяти (не считая той памяти, которую занимают команды программы) сверх двух массивов `map` и `newmap`. Эти массивы используют элементы, которые, в предположении, что в них хранятся только два значения — мертвый и живой, — могут быть упакованы так, что каждый элемент занимает всего лишь один бит. В типичном компьютере с размером слова 32 или 16 бит для решетки размера 20 на 60 оба массива потребуют не более 65 или 150 слов соответственно. С другой стороны, программа `Life2` требует, помимо места для команд, пространство для каждого массива, а также еще 1200 слов для массива `numbrnbs` и по 300 слов для каждого из ее четырех списков, что в сумме дает 2500 слов.

### 3. Компромисс между использованием памяти и временем выполнения

Мы только что встретились с одним из множества примеров, иллюстрирующих противоречивое влияние алгоритма программы на время ее выполнения и требуемое пространство памяти. Что лучше выбрать — более быструю программу, занимающую много места, или экономную в смысле расходования памяти, но более медленную программу — зависит от наличного оборудования. Если на компьютере достаточно неиспользуемой памяти, очевидно, лучше выбрать алгоритм, требующий больше места, но более быстрый. Если памяти мало, придется пожертвовать скоростью работы. Однако для важных проектов наилучшим подходом будет продумать всю проблему заново: получив опыт из первых попыток, вы, возможно, сможете найти новый алгоритм, экономно использующий и память, и время.

**Программистский принцип**

*При выборе алгоритма продумайте соотношение памяти  
и времени выполнения*



**Программистский принцип**

*Никогда не бойтесь начать разработку сначала.  
В следующий раз ваша программа может оказаться  
короче и эффективнее.*

**Упражнения 2.5**

- Е1.** Мы можем сэкономить память, выделяемую под массив `map`, выполнив небольшую модификацию способа сохранения информации в массиве `numbrnbs`. Мы можем использовать положительные значения элементов этого массива для обозначения живых ячеек и отрицательные значения для обозначения мертвых ячеек. Однако в этом случае мы не сможем сказать, обозначает ли значение 0 мертвую ячейку или живую ячейку, не имеющую живых соседей. Мы можем легко преодолеть это затруднение, изменив определение понятия «сосед», именно, считать ячейку собственным соседом (и, таким образом, число соседей для мертвой ячейки будет находиться в пределах от 0 до 8, и храниться в массиве `numbrnbs` в виде чисел от 0 до  $-8$ , а число соседей для живой ячейки — от 1 до 9).
- (а) При таком изменении определения сформулируйте измененные правила (из Раздела 1.2.1) для игры «Жизнь».
  - (б) Как вы полагаете, стоит ли затрачиваемых усилий реализация изменений, позволяющих устранить массив `map`? Почему да или почему нет?
  - (с) Если вы ответили на последний вопрос положительно, опишите детально, какие изменения придется внести в программу.

**Программные проекты 2.5**

- P1.** Если вы используете видеотерминал с прямой адресацией курсора, напишите вариант процедуры `WriteMap`, в котором используется метод обновления вывода программы с помощью списков `newlive` и `newdie` вместо полной перерисовки каждого поколения.
- P2.** Модифицируйте процедуру `ReadMap` так, чтобы она давала пользователю возможность альтернативной инициализации конфигурации чтением данных с клавиатуры или из файла. Первая строка файла содержит комментарий, описывающий конфигурацию; оставшиеся строки должны содержать последовательность пробелов для мертвых ячеек и каких-либо других символов для живых ячеек.
- P3.** Выполните прогоны законченных программ `Life2` и `Life1` и сравните скорости их выполнения.

**2.6. Заключение и предварительный просмотр**

В этой главе были описаны многие фундаментальные понятия, но главным образом с высоты птичьего полета. В последующих главах мы изучим некоторые темы значительно глубже; рассмотрение других придется



перенести на более продвинутые курсы; наконец, какие-то темы лучше всего изучать на практике.

## 2.6.1. Игра «Жизнь»

### 1. Будущие направления

Мы еще не прощаемся с игрой «Жизнь», хотя сейчас мы займемся другими темами. Вернувшись к игре «Жизнь» (в Разделе 9.9), мы найдем алгоритм, который не требует хранения в памяти больших прямоугольных решеток.

### 2. Спецификация задачи

Разрешите мне остановиться на одном наблюдении, которые вы, возможно, уже сделали сами, и, которое, наверное, вас беспокоило. Все наши действия, выполняемые по ходу этой и предыдущей глав, были в действительности неверны в том отношении, что мы разрабатывали программу игры «Жизнь» не так, как первоначально было описано в разделе 1.2. В правилах ничего не говорится о границах решетки, содержащей ячейки. В наших программах, когда движущееся сообщество приближается к границе, для соседей не остается места, и сообщество деформируется из-за самого наличия границы. Это происходить не должно.

Разумеется, в любом компьютерном моделировании имеются абсолютные границы возникающих значений, но ясно, что наш случай решетки 20 на 60 слишком уж ограничен, причем ограничен произвольно. Когда мы вернемся к этой проблеме, нашей целью будет написание более реалистичной программы. При первой попытке ограничения часто представляются разумными. Тем не менее,

#### ***Программистский принцип***

*Удостоверьтесь, что вы правильно понимаете задачу.  
Если вы хотите изменить ее условия,  
точно объясните, что именно вы сделали.*

Когда мы начинали нашу разработку в Разделе 1.4, мы ничего подобного не сделали, а начали реализовывать некоторый подход, оставив еще много неясного. Почти каждый программист приобретает опыт на своих ошибках и, наверное, согласится со следующим:

#### ***Программистский принцип***

*Поспешишь — людей насмешишь.  
Торопливость в программировании ведет к бесконечной отладке.*

Та же мысль может быть выражена более определенно:

#### ***Программистский принцип***

*Начать разработку заново часто проще,  
чем латать старую программу.*

Проверенное на опыте правило гласит, что если надо изменить более десяти процентов программы, пришло время полностью программу переписать. При включении в большую программу все новых и новых заплаток количество ошибок имеет тенденцию оставаться неизменным. Объясняется это тем, что заплатки становятся столь запутанными, что каждая новая заплатка вносит столько же новых ошибок, сколько она исправляет старых.

### 3. Прототипирование

Превосходный способ устранения необходимости заново переписывать большой программный проект заключается в разработке с самого начала двух вариантов программы. Пока программа еще не работает, часто невозможно узнать, какие части проекта приведут к наибольшим трудностям, или какие свойства программы придется изменять, чтобы удовлетворить потребности пользователей. Инженеры уже давно усвоили, что невозможно создать сложную машину непосредственно с чертежной доски. При разработке больших проектов инженеры всегда строят *прототипы*, т. е. уменьшенные модели, которые можно изучать, тестировать и иногда даже использовать ограниченным образом. Модели мостов строятся и подвергаются испытаниям в аэродинамических трубах; перед тем, как пытаться использовать новую технологию на конвейере, создаются пилотные фабрики.

прототипы  
программного  
обеспечения

Прототипирование особенно полезно для компьютерного программного обеспечения, поскольку оно упрощает связь между пользователями и разработчиками на ранних стадиях проекта, тем самым уменьшая степень взаимного недопонимания и способствуя разрешению проектных проблем ко всеобщему удовлетворению. Рабочую модель, способную выполнять некоторые из поставленных задач, можно собрать из программных блоков с минимальными затратами программистского труда. Даже если прототип не будет функционировать достаточно эффективно или выполнять все, на что рассчитана конечная система, он будет являться превосходной лабораторией, предоставляющей возможность и пользователю, и разработчику экспериментировать с различными идеями, предназначенными для конечного продукта.

#### ***Программистский принцип***

*Всегда планируйте разработку прототипа,  
который вам придется выбросить.  
Вы все равно будете вынуждены это сделать,  
хотите вы того или нет.*

## 2.6.2. Разработка программы

### 1. Критерии для программ

Основной целью этой книги является оценка алгоритмов и структур данных, предназначенных для решения поставленной задачи. Среди многих критериев, по которым мы можем судить о качестве программы, наиболее важными являются следующие:

1. Решает ли наша программа поставленную задачу в соответствии с заданными спецификациями?
2. Работает ли она правильно при всех условиях?
3. Включает ли она понятную и достаточную информацию для пользователя в форме инструкций и документации?
4. Написана ли она ясно и логично? Составлена ли она из коротких модулей и подпрограмм, решающих отдельные логические задачи?
5. Эффективна ли она в плане использования памяти и времени процессора?

Некоторые из этих критериев будут внимательно изучены для составляемых нами программ. Другие мы не будем упоминать явным образом, но отнюдь не из-за их незначительности. Эти критерии будут автоматически удовлетворены, если каждый этап разработки программы мы будем достаточно глубоко продумывать и тщательно реализовывать. Я надеюсь, что изучаемые нами примеры будут служить демонстрацией такого подхода.

## 2. Программная инженерия

**Программная инженерия** — это теоретические и практические методы, помогающие в создании и поддержке больших программных систем. Хотя программы, которые мы рассматриваем в этой главе, по нынешним меркам невелики, они иллюстрируют многие аспекты программной инженерии.

Программная инженерия основывается на признании того факта, что получение доброкачественного программного обеспечения представляет собой очень долгий процесс. Он начинается задолго до кодирования программ и продолжается в виде поддержки программ в течение многих лет после их выпуска для всеобщего использования. Этот непрерывный процесс носит название **жизненного цикла** программного обеспечения. Жизненный цикл можно разбить на следующие этапы:

1. *Проанализируйте* задачу со всей строгостью и полнотой. Обязательно тщательно *специфицируйте* весь пользовательский интерфейс.
2. *Создайте* прототип и *экспериментируйте* с ним, пока все спецификации не будут отработаны.
3. *Разработайте* алгоритм, используя структуры данных и другие алгоритмы с уже известными функциями.
4. *Верифицируйте* правильность алгоритма, или упростите его настолько, чтобы его правильность была самоочевидна.
5. *Проанализируйте* алгоритм, чтобы определить условия его выполнения и убедиться в том, что он удовлетворяет поставленным спецификациям.
6. *Закодируйте* алгоритм, преобразовав его в программу на выбранном языке.
7. *Протестируйте* и *оцените* программу с помощью тщательно выбранных тестовых данных.

фазы  
жизненного цикла

8. *Уточните и повторите* вышеперечисленные шаги применительно к дополнительным подпрограммам, пока программное обеспечение не примет вид завершенного и полностью функционирующего продукта.
9. Оптимизируйте код для повышения его производительности, но только если это действительно необходимо.
10. *Организируйте поддержку* программы, чтобы она продолжала удовлетворять изменяющимся потребностям пользователей.

Большинство этих тем обсуждалось и иллюстрировалось в различных разделах этой и предыдущих глав, однако будет не лишним сделать несколько дополнительных замечаний относительно первого этапа, именно, анализа и спецификации.

### 3. Анализ задачи

Анализ задачи часто оказывается наиболее сложным этапом жизненного цикла программного обеспечения. Это связано не с тем, что практические задачи принципиально сложнее упражнений по программированию — часто справедливо обратное — а с тем, что пользователи и программисты имеют тенденцию говорить на разных языках. Ниже перечислены некоторые вопросы, по которым аналитик и пользователь должны достичь взаимопонимания:

спецификации

1. Какова должна быть форма ввода данных в программу и вывода из нее? Каков объем этих данных?
2. Предъявляются ли какие-либо особые требования к обработке данных? Какие специальные случаи потребуют отдельной обработки?
3. Будут ли эти требования изменяться? Сколь быстро будет расти потребление разрабатываемой системы?
4. Какие части системы наиболее важны? Какие должны выполняться с максимальной эффективностью?
5. Как следует реагировать на неправильные данные? Какая еще требуется обработка ошибок?
6. Какого рода потребители будут использовать программное обеспечение? Каков будет уровень их подготовки? Какого рода пользовательский интерфейс будет для них наилучшим?
7. Насколько мобильным должно быть программное обеспечение в плане переноса его на новый тип оборудования? С каким другим программным обеспечением или аппаратными системами должен быть совместим разрабатываемый проект?
8. Какие предполагаются расширения системы и какого рода обслуживание? Какова история предыдущих изменений программного и аппаратного обеспечения?

### 4. Спецификация требований

Анализ задачи и экспериментирование над большими проектами в итоге приводят к разработке формальных требований к проекту. Эти формальные требования становятся базой, на которой пользователь и инженер-программист стараются понять друг друга и определить стандарт,

согласно которому будет оцениваться конечный продукт. Среди компонентов спецификации можно выделить следующие:

1. *Функциональные требования* к системе: что она будет делать и какие команды будут доступны пользователю.
2. *Допущения и ограничения*, накладываемые на систему: на каком аппаратном обеспечении будет работать система, какая допустима форма ввода, максимальный объем вводимых данных, наибольшее число пользователей и т. д.
3. *Требования к поддержке*: предполагаемые расширения или развитие системы, изменения в аппаратном обеспечении, изменения пользовательского интерфейса.
4. *Требования к документации*: какого рода пояснительный материал необходим пользователям того или иного рода.

Спецификации требований показывают, *что* должно делать программное обеспечение, но не *как* это будет делаться. Эти спецификации должны быть понятными и для пользователя, и для программиста. Если их подготовить достаточно тщательно, они образуют базис для всех последующих этапов проектирования, кодирования, тестирования и поддержки.

### 2.6.3. Pascal

В этой и предыдущих главах мы галопом прошли по многим средствам языка Pascal. При этом не было сделано никаких попыток последовательного или полного описания этих средств. Сжатая справка по языку Pascal приведена в приложении D, к которому вам следует обращаться при возникновении вопросов, касающихся синтаксиса языка Pascal. Другие примеры и более детальное обсуждение можно найти в учебниках по языку Pascal.

типы данных

Одним из наиболее мощных черт языка Pascal является гибкость и прочность его типов данных. Мы едва коснулись изучения этих ресурсов. По мере необходимости мы будем использовать средства языка для создания нужных нам типов данных: файлов, множеств и указателей. Возможность комбинировать определения типов гибким образом (файлы записей, записи, содержащие массивы, массивы записей и т. д.) предоставляет почти бесконечные возможности организации наших структур данных.

### Программные проекты 2.6

**P1. Магический квадрат** — это такой квадратный массив целых чисел, в котором сумма чисел в любой строке, в любом столбце и в любой из двух главных диагоналей одинаковы. Два магических квадрата изображены на рис. 2.3.<sup>2</sup>

<sup>2</sup>

Магический квадрат в левой части рисунка изображен Альбрехтом Дюрером в своей известной гравюре *Меланхолия*. Обратите внимание на включение в магический квадрат года создания картины, 1514.

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

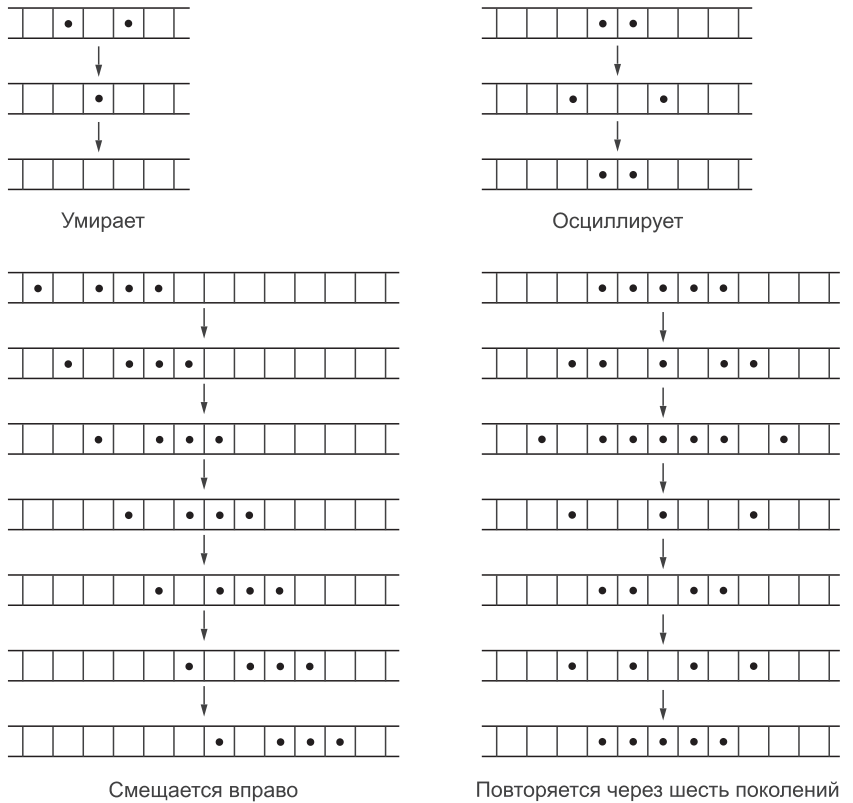
Сумма = 34

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

Сумма = 65

**Рис. 2.3.** Два магических квадрата

- (a) Напишите программу, которая читает квадратный массив целых чисел и определяет, является ли он магическим квадратом.
- (b) Напишите программу, которая генерирует магические квадраты следующим методом (метод применим, только если квадрат имеет нечетное число строк и столбцов). Начните с помещения 1 в середину верхней строки. Впишите в клетки диагонали, идущей от этой клетки вверх и вправо, последовательный ряд чисел 2, 3, ... . Когда вы достигните верхней строки (что произойдет немедленно, так как ваша 1 находится в верхней строке) продолжайте заполнение с нижней строки, как если бы она располагалась непосредственно над верхней строкой. Когда вы достигните самого правого столбца, продолжайте заполнение с самого левого столбца, как если бы он располагался непосредственно справа от самого правого. Если вы достигли уже занятой клетки, сместитесь от последней заполненной клетки на одну позицию вниз и продолжайте заполнение тем же порядком. Магический квадрат размера  $5 \times 5$ , построенный описанным методом, показан на рис. 2.3.
- P2.** Одномерная игра «Жизнь» развивается не на прямоугольной решетке, а на прямой линии. Каждая ячейка имеет четыре соседние позиции, на расстоянии 1 и 2 с каждой стороны. Правила такой игры схожи с правилами двумерной игры, за исключением: (1) мертвая ячейка с двумя или тремя живыми соседями оживает в следующем поколении и (2) живая ячейка умирает, если она имеет ноль, одного или трех живых соседей. (Отсюда мертвая ячейка, имеющая ноль, одного или четырех живых соседей, остается мертвой; живая ячейка с двумя или четырьмя живыми соседями остается живой.) Развитие нескольких конфигураций сообщества показано на рис. 2.4. Разработайте, напишите и протестируйте программу для одномерной игры «Жизнь».
- P3.** (a) Напишите программу, которая будет печатать календарь текущего года.
- (b) Модифицируйте программу таким образом, чтобы она читала номер года и печатала календарь на заданный год. Год является високосным (т. е. в феврале не 28, а 29 дней), если его номер кратен 4,



**Рис. 2.4.** Последовательные конфигурации одномерной игры «Жизнь»

за исключением того, что столетние коды (кратные 100) являются високосными, только если год делится на 400. Таким образом, год 1900 не является високосным, хотя год 2000 — високосный.

- (с) Модифицируйте программу таким образом, чтобы она принимала любую дату (день, месяц, год) и печатала день недели для этой даты.
- (d) Модифицируйте программу таким образом, чтобы она читала две даты и печатала число дней от одной до другой.
- (е) Используя правила определения високосного года, покажите, что последовательность календарей в точности повторяется каждые 400 лет.
- (f) Какова вероятность (в интервале 400 лет) того, что 13-е число месяца придется на пятницу? Почему 13-е число месяца с большей вероятностью приходится на пятницу, чем на любой другой день недели? Напишите программу, которая будет считать, сколько пятниц, приходящихся на 13-е число, будет в этом столетии.

## Подсказки и ловушки

1. Для улучшения своей программы пересмотрите ее логику. Не оптимизируйте код, основанный на неэффективном алгоритме.
2. Никогда не оптимизируйте программу, пока вы не добьетесь ее правильной работы.
3. Не оптимизируйте код, если только это не является абсолютно необходимым.
4. Для упрощения логики своей программы используйте записи.
5. При вложении процедур пользуйтесь вложенными предложениями **with**. Для каждой процедуры используйте свой уровень предложения **with**.
6. Старайтесь, чтобы ваши процедуры были короткими; редкая процедура должна иметь длину более страницы.
7. Перед тем, как вы приступите к написанию кода, удостоверьтесь в правильность вашего алгоритма.
8. Верифицируйте запутанные части ваших алгоритмов.
9. Максимально упрощайте логику своих программ.
10. Повторяйте Программистские принципы!

## Обзорные вопросы

- |     |                                                                                                                    |
|-----|--------------------------------------------------------------------------------------------------------------------|
| 2.1 | 1. Что такое поддержка программы?                                                                                  |
| 2.2 | 2. Что такое математическая индукция?                                                                              |
|     | 3. Что такое инвариант цикла?                                                                                      |
|     | 4. Что представляют собой предусловия и постусловия подпрограмм?                                                   |
| 2.3 | 5. Когда следует приступить к распределению задач между подпрограммами?                                            |
|     | 6. До какого момента следует откладывать кодирование программы?                                                    |
| 2.5 | 7. Что такое компромисс между использованием памяти и временем выполнения?                                         |
| 2.6 | 8. Что такое прототип?                                                                                             |
|     | 9. Назовите по меньшей мере шесть этапов жизненного цикла программного обеспечения и опишите, чем является каждый. |
|     | 10. Определите понятие программной инженерии.                                                                      |
|     | 11. Что представляют собой спецификации требований для программ?                                                   |

## Литература для дальнейшего изучения

программная  
инженерия

Глубокие исследования многих аспектов структурного программирования можно найти в книге:

Edward Yourdon, *Techniques of Program Structure and Design*, Prentice-Hall, Englewood Cliffs, N.J., 1975, 364 pages.



Познавательное обсуждение (в книге, от чтения которой получаешь истинное удовольствие) многих проблем, возникающих при разработке больших программных систем, содержится в:

Frederick P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Reading, Mass., 1975, 195 pages.

Имеется русский перевод: Брукс Фр., Мифический человек-месяц или как создаются программные системы. — СПб.: Символ-Плюс, 2000. — 623 с.

Хороший учебник по программной инженерии:

Ian Sommerville, *Software Engineering*, Addison-Wesley, Wokingham, England, 1985, 334 pages.

Имеется русский перевод: Соммервил И., Инженерия программного обеспечения. — 6-е изд. — М. и др.: Изд. дом «Вильямс». 2002 — 623 с.

Тестирование программ разработано до такой степени, что его методы могут составить толстый том:

William E. Perry, *A Structured Approach to Systems Testing*, Prentice-Hall, Englewood Cliffs, N.J., 1983, 451 pages.

Две книги, посвященные доказательству правильности программ и использованию утверждений и инвариантов при разработке алгоритмов:

David Gries, *Science of Programming*, Springer-Verlag, New York, 1981, 366 pages.

Имеется русский перевод: Грис Д. Наука программирования. — М.: Мир, 1984.

Suad Alagiz and Michael A. Arbib, *The Design of Well-Structured and Correct Programs*, Springer-Verlag, New York, 1978, 292 pages.

Поддержание своих программ настолько простыми, чтобы было легко доказать их правильность, является непростой, но чрезвычайно важной задачей. К. А. Р. Хоар (C. A. R. Hoare) (предложивший алгоритм быстрой сортировки; см. гл. 7) пишет: «Существуют два способа разработки программного обеспечения. По первому способу вы делаете программу настолько простой, что в ней с очевидностью отсутствуют дефекты; по второму — настолько сложной, что в ней нельзя обнаружить очевидных дефектов. Первый способ значительно сложнее второго». Это цитата из лекции, прочитанной при вручении премии Тьюринга: «The emperor's old clothes», *Communications of the ACM* 24 (1981), 75–83.

Две книги посвящены методам решения задач:

George Pólya, *How to Solve It*, second edition, Doubleday, Garden City, N.Y., 1957, 253 pages.

Имеется русский перевод: Пойа Д. Как решать задачу. — Львов: журн. «Квантор», 1991.

Wayne A. Wickelgren, *How to Solve Problems*, W. H. Freeman, San Francisco, 1974, 262 pages.

Программный проект одномерной игры «Жизнь» взят из

Jonathan K. Miller, «One-dimensional Life», *Byte* 3 (December, 1978), 68–74.

верификация  
алгоритмов

решение  
задач

## Глава 3

# Стеки и рекурсия

---

В этой главе рассматриваются стеки, которые относятся к одному из простейших, но самых важных видов структур данных. Мы увидим, что стеки тесно связаны с рекурсией, методом, позволяющим решить сложную задачу путем сведения ее к более простым случаям той же задачи. Для иллюстрации рекурсии мы рассмотрим некоторые приложения и программный пример. Далее в этой главе мы изучим вопрос о реализации рекурсии на компьютере. В процессе изучения мы разработаем руководящие принципы, касающиеся удачного и неудачного использования рекурсии, и позволяющие оценить, где рекурсия полезна и где ее следует всячески избегать.

## 3.1. Стеки

### 3.1.1. Введение

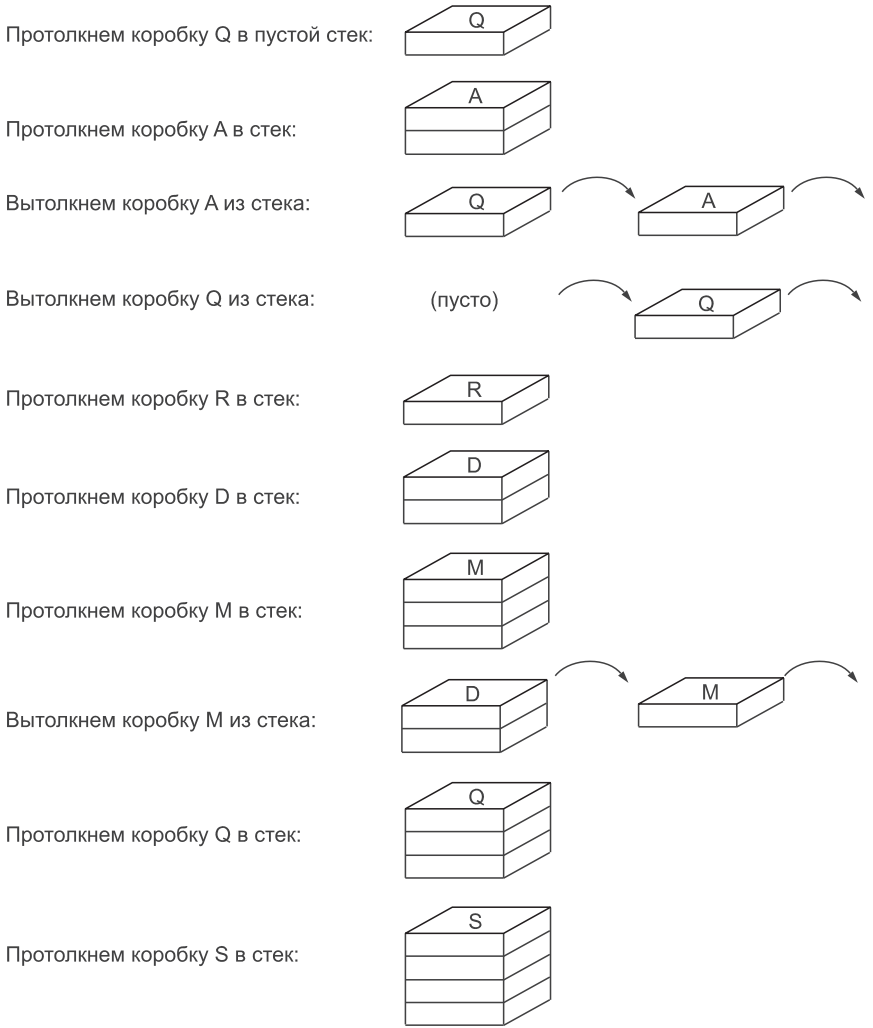
стеки

*Стек* представляет собой структуру данных, в которой все включения и исключения элементов осуществляются с одного конца, называемого *вершиной* стека. Стек можно представить (см. рис. 3.1) как стопку подносов или тарелок, стоящих на прилавке заполненного посетителями кафе. Пока длится обеденное время, посетители берут подносы с вершины стопки, а служащие кафе возвращают чистые подносы также на вершину стопки. Поднос, положенный на стопку последним будет первым снятым с нее. Дно стопки образует поднос, положенный на нее первым; он же будет самым последним используемым подносом.

Иногда в качестве аналогии стека рассматривается стопка тарелок или подносов в устройстве, оснащенном пружиной, так что вершина стопки всегда остается на одной и той же высоте. Однако такая аналогия мало соответствует действительности и ее следует избегать. Если бы мы захотели реализовать стек таким образом, нам пришлось бы перемещать все до единого элементы стека при каждом включении или удалении элемента. Значительно правильнее рассматривать стек стоящим на твердой поверхности прилавка или стола, когда при добавлении или удалении элементов перемещается только один верхний элемент. Пружинная аналогия, однако, породила пару колоритных слов, твердо вошедших в компьютерный жаргон, и которые мы будем использовать для обозначения фундаментальных операций со стеком. Когда мы добавляем элемент в стек, мы говорим, что мы *проталкиваем* его в стек, а когда мы удаляем

проталкивание  
и выталкивание





**Рис. 3.2.** Проталкивание в стек и выталкивание из стека

```

while (not eoln) and (not StackFull(line)) do
begin
    read(ch);
    Push(ch, line)
end;
readln;
writeln(" Вот строка в обратном порядке: ");
while not StackEmpty(line) do
begin{ выталкиваем каждый символ из стека и выводим его на экран }
    Pop(ch, line);
    write(ch)
end;
writeln
end;
{ процедура ReverseRead }
    
```

В этой процедуре мы не только использовали операции Push и Pop, но также и процедуру CreateStack, которая создает стек и инициализирует его так, что он оказывается пустым. У нас имеются также две булевых функции, используемые для контроля текущего состояния стека: StackEmpty проверяет, не пуст ли стек, а StackFull проверяет, не заполнен ли он до конца.

реинициализация

размер стека

вершина стека

Наконец, еще три операции иногда оказываются полезными при работе со стеками. Первая — это ClearStack, процедура, получающая в качестве параметра существующий стек и делающая его пустым. Вторая функция — StackSize; она возвращает число элементов в стеке. Третью операцию можно назвать StackTop; она возвращает верхний элемент стека, не изменяя при этом состояние стека. Мы могли бы, разумеется, сконструировать эту процедуру из операций Push и Pop, сначала выталкивая элемент из стека, сохраняя его значение и затем проталкивая его обратно в стек. Однако лучше включать процедуру StackTop в число фундаментальных операций со стеком.

### 3.1.3. Соккрытие информации

использование процедур

Обратите внимание на то, что мы смогли написать нашу процедуру для реверсирования входной строки перед тем, как стали обсуждать фактическую реализацию стека в памяти и перед разработкой деталей различных процедур и функций. Здесь мы имеем еще один пример **сокрытия информации**: если кто-то уже написал процедуры и функции для работы со стеком, мы можем использовать их, не зная деталей хранения стеков в памяти или того, как фактически выполняются операции со стеком.

альтернативные реализации

В последующих главах мы увидим, что для стеков (как и почти для всех изучаемых нами типов данных) можно предложить несколько способов представления данных в компьютерной памяти, а также несколько способов выполнения базовых операций. В некоторых приложениях может оказаться лучше один метод, в других приложениях — другой.

изменение реализации

Даже в одиночной большой программе мы можем сначала выбрать один метод представления стеков, а затем, по мере приобретения опыта работы с программой, решить, что другой метод даст лучшие результаты. Если бы команды управления стеком писались бы заново при каждом создании стека, тогда все включения этих команд в программу пришлось бы заменять на новые варианты. Если же мы практикуем сокрытие информации путем использования отдельных процедур и функций управления стеками, изменить придется только объявления.

понятность программы

Еще одно преимущество сокрытия информации применительно к стекам заключается в том, что само появление в программе слов *Push* и *Pop* немедленно покажет человеку, читающему программу, что именно здесь делается, в то время как команды, составляющие эти процедуры, могут быть значительно менее наглядными.

нисходящая разработка

Последнее преимущество заключается в том, что отделение использования структур данных от их реализации поможет нам усовершенствовать нисходящую разработку как структур данных, так и самой программы.

### 3.1.4. Спецификации для стека

Завершая этот раздел, рассмотрим точные спецификации для стека с помощью пред- и постусловий для каждой упомянутой нами ранее процедуры и функции. В этих объявлениях мы будем использовать для стека тип `stack`. В соответствии с принципами сокрытия информации и нисходящей разработки мы оставим этот тип без спецификации до следующего раздела, в котором мы рассмотрим реализацию стеков. Элементы, сохраняемые в стеке, будут иметь тип, который мы назовем `stackentry`, и который будет изменяться в зависимости от требований конкретного приложения. Для реверсирования строки ввода нам понадобится объявление `stackentry = char`. Для других приложений мы определим тип `stackentry` иным образом.

Первым шагом при работе с любым стеком будет его инициализация:

инициализация

```
procedure CreateStack (var S: stack);                                { Создать стек }
предусловие: Отсутствует.
постусловие: Стек S был создан и инициализирован так, что он пуст.
```

состояние

Далее идут операции проверки состояния стека.

```
function StackEmpty (S: stack) : Boolean;                            { Стек пуст? }
предусловие: Стек S уже создан.
постусловие: Функция возвращает true, если стек S пуст, и false в противном случае.
```

```
function StackFull (S: stack) : Boolean;                             { Стек полон? }
предусловие: Стек S уже создан.
постусловие: Функция возвращает true, если стек S полон, и false в противном случае.
```

базовые операции

Приведем теперь фундаментальные операции со стеком.

```
procedure Push (x: stackentry; var S: stack);                      { Протолкнуть }
предусловие: Стек S уже создан и не полон.
постусловие: Элемент x был сохранен в стеке в качестве его верхнего элемента.
```

```
procedure Pop (var x: stackentry; var S: stack);                  { Вытолкнуть }
предусловие: Стек S уже создан и не пуст.
постусловие: Элемент, находившийся на вершине стека, снят со стека и возвращен в качестве значения x
```

Заметьте, что согласно предусловиям было бы ошибкой проталкивать элемент в заполненный стек или выталкивать элемент из пустого. Мы должны написать процедуры `Push` и `Pop` достаточно аккуратно, чтобы они возвращали ошибку при попытке неправильного использования. Однако из объявлений не следует, что процедуры обнаружат ошибочный

вызов, хотя ошибки такого рода приведут к неправильным и непредсказуемым результатам. Отсюда следует, что аккуратный программист всегда должен внимательно следить за выполнением всех предусловий при вызове подпрограммы.

У нас остались три операции со стеком, которые иногда могут оказаться полезными.

другие операции

```
procedure ClearStack (var S: stack); { Очистить стек }
предусловие: Стек S уже создан.
постусловие: Из стека S удалены все находившиеся там элементы;
                стек S пуст.
```

```
function StackSize (S: stack): integer; { Размер стека }
предусловие: Стек S уже создан.
постусловие: Функция возвращает число элементов в стеке S.
```

```
procedure StackTop (var x: stackentry; S: stack); {Вершина стека}
предусловие: Стек S уже создан и не пуст.
постусловие: В переменную x копируется верхний элемент стека S;
                стек S не изменяется.
```

просматриваемый  
стек

В *просматриваемом стеке*, помимо всех стандартных стековых операций, предусматривается еще одна операция: просмотр (т. е. чтение) всех элементов стека по очереди без изменения каких-либо элементов. (Единственным способом изменения содержимого просматриваемого стека является использование операций Push и Pop, посредством которых все изменения выполняются на вершине стека.) Как это мы уже делали в случае просмотра списков, мы используем формальный параметр-процедуру с именем Visit для обозначения операции, которая должна быть выполнена над каждым элементом стека.

Спецификации просматриваемого стека выглядят таким образом:

```
procedure TraverseStack (S: stack; procedure Visit(x: stackentry));
                { Просмотр стека }
предусловие: Стек S уже создан.
постусловие: Процедура Visit(x: stackentry) была вызвана для всех элементов стека, начиная с элемента на его вершине и далее последовательно в направлении дна стека.
```

Строго говоря, просматриваемые стеки относятся к типам данных, отличным от обычных стеков, и для большинства приложений необходимости в операции TraverseStack не возникает. Однако в качестве инструмента для изучения стеков, а также в целях отладки, просмотр стека оказывается весьма полезной операцией, и мы во всех случаях, когда нам это понадобится, будем считать, что такая операция реализована как для стеков, так и для других типов данных, которые мы будем определять по ходу нашего изложения.

**Замечание**

Чтобы завершить рассмотрение деталей приведенных выше спецификаций, вспомним, что в разделе 2.2.1 мы условились, что при указании типа данных мы используем **var**-параметры только в тех случаях, когда подпрограмма может изменить этот параметр, в противном же случаях мы используем параметры-значения. Однако когда мы перейдем к деталям реализации в процессе фактического написания кодов подпрограмм, мы обычно в целях повышения эффективности будем объявлять структуры данных как **var**-параметры, независимо от того, изменяются ли они подпрограммой или нет.

### 3.1.5. Реализация стеков

Теперь, когда мы определили все операции со стеками, мы можем рассмотреть программистские детали их реализации. Начнем с *непрерывной* реализации, когда элементы стека размещаются в массиве. Позже мы рассмотрим *связную* реализацию с использованием указателей и динамической памяти.

#### 1. Объявления

Для непрерывной реализации мы создадим массив, в котором будут храниться элементы стека, а также счетчик, который будет фиксировать количество элементов. Используя язык Pascal, мы напомним следующие объявления, где `maxstack` — это константа, задающая максимальный размер стека, а `stackentry` — тип, описывающий данные, которые мы будем сохранять в стеке. Тип `stackentry` зависит от требований приложения и может определяться в диапазоне от простого числа или символа до больших записей со многими полями. Спецификация типа `stackentry` является частью прикладной программы.

тип стек

```
type stack = record
    top: 0..maxstack;
    entry: array [1..maxstack] of stackentry
end;
```

#### 2. Проталкивание и выталкивание

Проталкивание и выталкивание реализуются следующим образом. Мы должны внимательно отнестись к граничным случаям, именно, к попыткам выталкивания элемента из пустого стека или проталкивания в полный. Эти действия являются ошибочными. Согласно спецификациям, процедуры работы со стеком не обязаны обнаруживать такие ошибки, поскольку они являются нарушением предусловий для процедур. Тем не менее стоит придерживаться правил защищенного программирования и включать в реализацию максимально возможное число проверок соблюдения предусловий. Поэтому мы используем процедуру `Error` из модуля (или пакета) `Utility` для оповещения пользователя о нарушении заданных условий.

проталкивание

```
procedure Push(x: stackentry; var S: stack); { Протолкнуть }
{ Pre: Стек S уже создан и не полон.
  Post: Элемент x сохранен в стеке в качестве его верхнего элемента. }
```



```

begin                                     { процедура Push }
  with S do
    if top = maxstack then
      Error('Недопустимо проталкивание в полный стек.');
```

выталкивание

```

    else begin
      top := top + 1;
      entry[top] := x
    end
  end;                                     { процедура Push }
  procedure Pop(var x: stackentry; var S: stack);
    { Pre: Стек S уже создан и не пуст.
      Post: Элемент, находившийся на вершине стека, удален и возвращен
            как x. }
    { Вытолкнуть }
  begin                                     { процедура Pop }
    with S do
      if top = 0 then
        Error('Недопустимо выталкивание из пустого стека.');
```

стек пуст

```

      else begin
        x := entry[top];
        top := top - 1;
      end
    end;
  end;                                     { процедура Pop }
```

### 3. Другие операции

```

function StackEmpty (var S: stack): Boolean;
  { Pre: Стек S уже создан.
    Post: Функция возвращает true, если стек пуст, и false в противном
          случае. }
  { Стек пуст? }
begin
  StackEmpty := (S.top = 0)
end;
```

стек полон

```

function StackFull (var S: stack): Boolean;
  { Pre: Стек S уже создан.
    Post: Функция возвращает true, если стек полон, и false в противном
          случае. }
  { Стек полон? }
begin
  StackFull := (S.top = maxstack)
end;
```

Заметьте, что в этих двух функциях мы объявили S как **var**-параметр, хотя фактически он используется только как входной параметр. Причина нашего решения лежит в эффективности: если бы параметр S был объявлен как значение, система при каждом вызове процедуры выполняла бы полное копирование стека.

Еще одна процедура нам нужна для инициализации стека перед его первым использованием в программе:

инициализация

```

procedure CreateStack (var S: stack);
  { Pre: Предусловия отсутствуют.
    Post: Стек S создан и инициализирован так, что он пуст. }
  { Создать стек }
begin
  S.top = 0;
end;
```

Мы оставим процедуры StackTop, ClearStack и TraverseStack, а также функцию StackSize для упражнений.

### 3.1.6. Связные стеки

Реализация стеков в виде связанных структур в динамической памяти также не представляет особых сложностей<sup>1</sup>. Прежде всего мы должны записать объявление, устанавливающее структуру узла:

```

type
  stackpointer = 1stacknode;
  stacknode = record
    entry:    stackentry;           { связь со следующим узлом в стеке }
    nextnode: stackpointer
  end;
```

элементы и узлы

Как и в случае непрерывных стеков, мы будем *проталкивать* (push) и *выталкивать* (pop) элементы с самого верхнего узла связанного стека, называемого его вершиной. Однако здесь мы сталкиваемся с проблемой несовместимости с непрерывной организацией стека. Элементы непрерывного стека были объявлены, как имеющие тип stackentry, в то время как связный стек состоит из *узлов*, имеющих тип stacknode; как следует из сделанного выше объявления, узел stacknode состоит из элемента stackentry вместе с указателем stackpointer. Нам хотелось бы иметь возможность заменять связную реализацию стеков непрерывной реализацией, не прибегая к каким-либо изменениям остальной части программы, и, следовательно, мы должны продолжать обрабатывать *элементы* типа stackentry в различных операциях со стеком.

закрытые  
процедуры  
PushNode и  
PopNode

С этой целью мы вводим две новые операции, проталкивания узла PushNode и выталкивания узла PopNode, которые будут обрабатывать узлы связанного стека. Эти операции являются *закрытыми* для связанной реализации, однако мы можем затем использовать эти операции при написании связанных вариантов открытых процедур Push и Pop, которые, как и в предыдущих главах, будут обрабатывать элементы непосредственным образом. Если у нас есть данное *x*, которое мы хотим протолкнуть в связный стек, мы прежде всего должны создать новый узел и поместить в этот узел *x*, а затем протолкнуть узел в стек. Таким образом мы имеем:

обработка  
stackentry

```

procedure Push (x: stackentry; var S: stack);           { Протолкнуть }
{ Pre:  Стек S уже создан.
  Post: Элемент x сохранен в стеке в качестве его верхнего элемента }
begin                                                  { процедура Push }
  new(p);
  p↑.entry := x
  PushNode(p, S)
end;                                                  { процедура Push }
```

обработка node

Чтобы написать процедуру PushNode, мы должны рассмотреть еще некоторые детали реализации связанных стеков.

<sup>1</sup> Чтение этого раздела требует знакомства с концепциями динамической памяти и типом указателей языка Pascal. См. приложение D или любой учебник по программированию на языке Pascal. Материал этого раздела не используется в оставшейся части данной главы, но понадобится в последующих главах.

Прежде всего мы должны решить, будет ли вершиной стека начало или конец связной структуры. На первый взгляд может показаться, что проще (как это делалось для непрерывных стеков) добавлять узлы к концу, однако в этом случае затрудняется выталкивание из стека. Действительно, поскольку указатели, сохраняемые в структуре, указывают на следующие узлы только в одном направлении, мы не можем сразу определить узел, находящийся *перед* данным узлом. В этом случае после удаления последнего элемента нам, чтобы найти предпоследний элемент, придется просматривать весь стек от самого его начала. Выталкивание из нашего связного стека осуществляется значительно проще, если все добавления и удаления осуществлять в начале структуры. Отсюда вершиной стека всегда будет *первый* узел связной структуры.

Каждый связный список имеет переменную, служащую головным указателем и указывающую на его первый узел; для связного стека этот указатель всегда указывает на вершину стека. Поскольку каждый узел связного списка указывает на следующий узел, для работы со связным стеком требуется лишь знать местоположение его вершины. Мы, таким образом, объявим связный стек с помощью записи, хранящей вершину стека и ничего более:

**type stack = record top: stackpointer end.**

Поскольку эта запись содержит только одно поле, мы могли бы обойтись без использования структурной переменной и обращаться к вершине стека по тому же имени, которые мы назначили самому стеку.

Имеются, однако три обстоятельства, требующие использования определенной нами структуры типа записи. Наиболее важное из них – это требование сохранения логического различия между самим стеком, который содержит все его элементы (каждый в своем узле), и вершиной стека, которая указывает на единственный узел. Тот факт, что для нахождения всех элементов стека нам достаточно знать положение его вершины, не имеет отношения к логической структуре стека. Второе обстоятельство заключается в необходимости поддержки совместимости с другими структурами данных и с другими реализациями, в которых записи необходимы для объединения различных частей информации. В-третьих, определяя стек и указатель на его вершину как несовместимые типы данных, мы облегчаем отладку, поскольку компилятор может в этом случае выполнить более детальную проверку типов.

Давайте начнем с пустого стека, что теперь означает  $S.top = \text{nil}$ , и добавим к нему первый узел. Мы будем полагать, что этот узел был уже создан где-то в динамической памяти, причем мы можем получить доступ к нему с помощью переменной-указателя  $p$ . Тогда сам узел будет обозначаться  $p\uparrow$ . Проталкивание  $p\uparrow$  в стек состоит из команд

$S.top := p;$      $p\uparrow.nextnode := \text{nil}.$

Продолжая, предположим, что мы уже имеем непустой стек и что мы хотим протолкнуть в него узел  $p\uparrow$ . Требуемая настройка указателей показана на рис. 3.3. Прежде всего мы должны установить указатель, приходящий с новым узлом  $p\uparrow$ , на старую вершину стека, после чего изменить

объявление  
типа stack

пустой стек

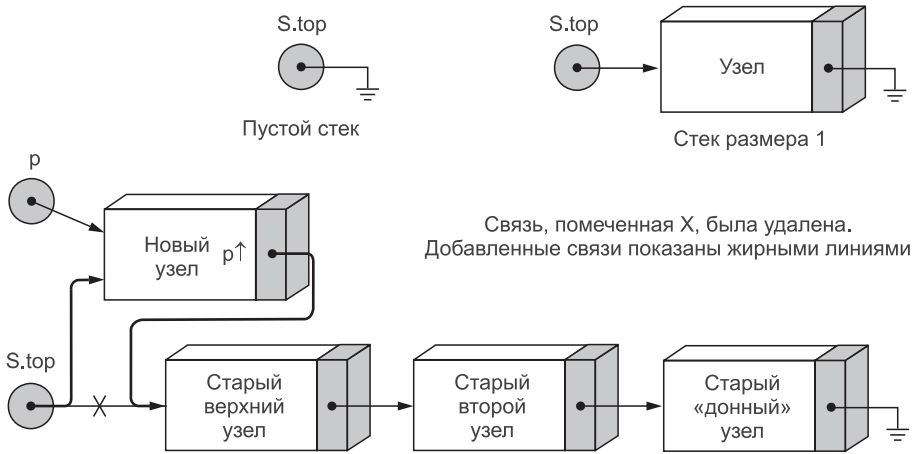


Рис. 3.3. Проталкивание узла в связный стек

вершину, чтобы ею стал новый узел. Порядок этих назначений важен. Если бы мы попытались выполнить их в обратном порядке, то изменение прежнего значения вершины приведет к тому, что мы потеряем старую часть списка. Итак, мы имеем следующую процедуру:

проталкивание  
узла

```

procedure PushNode (p: stackpointer; var S: stack); { Протолкнуть узел }
{ Pre: Связный стек S уже создан, а p указывает на узел, который
еще не помещен в стек.
Post: Узел  $p \uparrow$  протолкнут в связный стек S. }
begin
  if p = nil then
    Error('Попытка протолкнуть в стек несуществующий узел')
  else begin
     $p \uparrow$ .nextnode := S.top; {новый узел указывает на прежнюю вершину стека}
    S.top := p { установим вершину на новый узел }
  end
end; { процедура PushNode }
  
```

Во всех наших процедурах важно включать проверку на ошибки и рассматривать граничные случаи. В приведенном фрагменте фиксируется ошибка при попытке протолкнуть в стек несуществующий узел.

Одним граничным случаем для нашей процедуры является пустой стек, возникающий в случае  $S.top = \text{nil}$ . Заметьте, что в этом случае процедура выполняется правильно, проталкивая в пустой стек первый узел точно так же, как в непустой стек проталкивается дополнительный узел.

выталкивание из  
связного стека

Выталкивание узла из связного стека осуществляется так же просто. Этот процесс проиллюстрирован на рис. 3.4; детали мы оставляем для упражнений.

Остальные операции для связных стеков достаточно просты и мы их оставим для упражнений. Заметьте, однако, что для очистки связного стека мы должны не только установить головной указатель в **nil**, но и

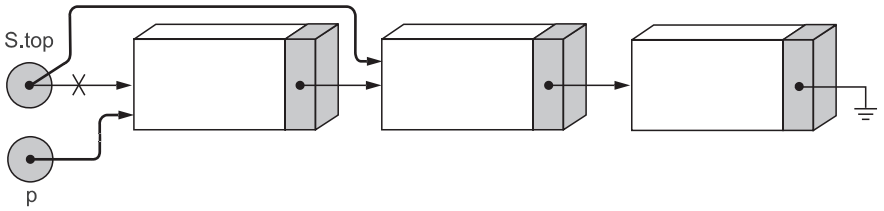


Рис. 3.4. Выталкивание узла из связанного стека

удалить все находящиеся в стеке узлы, чтобы вернуть системе занятую ими память. Эта операция, следовательно, заметно отличается от процедуры `CreateStack`. Для программирования этой операции мы должны организовать цикл по всем узлам, находящимся в стеке, удаляя каждый из них по очереди. Простейшим способом достичь этой цели является выталкивание из стека всех его узлов с освобождением занимаемой ими памяти до полного освобождения стека.

### Упражнения 3.1

**E1.** Изобразите на бумаге последовательность кадров стека аналогично рис. 3.2 для следующих сегментов кода. ( $S$  — это стек символов, а  $x$ ,  $y$ ,  $z$  — символьные переменные.)

(a) `CreateStack(S);`  
`Push('a', S);`  
`Push('b', S);`  
`Push('c', S);`  
`Pop(x, S);`  
`Pop(y, S);`  
`Pop(z, S);`

(b) `CreateStack(S);`  
`Push('a', S);`  
`Push('b', S);`  
`Push('c', S);`  
`Pop(x, S);`  
`Pop(y, S);`  
`Push('x', S);`  
`Push('y', S);`  
`Pop(z, S);`

(c) `CreateStack(S);`  
`Push('a', S);`  
`Push('b', S);`  
`ClearStack(S);`  
`Push('c', S);`  
`Pop(x, S);`  
`Push('a', S);`  
`Pop(y, S);`  
`Push('b', S);`  
`Pop(z, S);`

(d) `CreateStack(S);`  
`Push('a', S);`  
`Push('b', S);`  
`Push('c', S);`  
**while not** `StackEmpty(S)` **do**  
`Pop(x, S);`

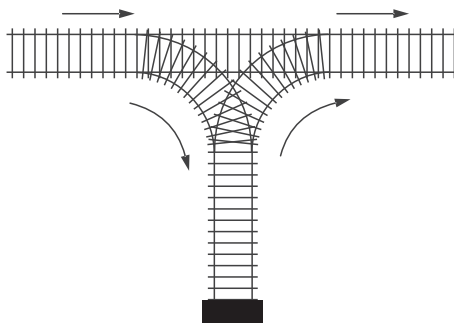
**E2.** Пусть  $S$  — это стек целых чисел, а  $x$  — целочисленная переменная. С использованием процедур `Push`, `Pop` и `CreateStack`, а также функций `StackEmpty` и `StackFull` напишите процедуры, выполняющие перечисленные ниже задачи. [Вы можете объявить дополнительные переменные и использовать в своих процедурах второй стек, если это потребуется.]

(a) Назначьте  $x$  значение верхнего элемента стека  $S$ , оставив верхний элемент  $S$  неизменным. Если  $S$  пуст, назначьте  $x$  значение `maxint`.

- (b) Назначьте  $x$  значение третьего от вершины элемента стека  $S$ , при условии, что  $S$  содержит по меньшей мере 3 целых числа. Если это не так, назначьте  $x$  значение  $\text{maxint}$ . Оставьте  $S$  неизменным.
- (c) Назначьте  $x$  значение элемента на дне стека  $S$  (или  $\text{maxint}$ , если  $S$  пуст), и оставьте  $S$  неизменным. [Подсказка: используйте второй стек.]
- (d) Удалите все вхождения  $x$  в  $S$ , оставив остальные элементы  $S$  в том же порядке.

**Е3.** Стек можно рассматривать как перестановочную ветку железнодорожных путей вроде той, что изображена на рис. 3.5. Вагоны с номерами 1, 2, ...,  $n$  подходят к ветке с левой стороны, и требуется изменить порядок вагонов, чтобы они покидали ветку с правой стороны в обратном порядке. Вагон, въехавший на ветку, может быть оставлен там или послан далее по правому пути, но не может быть послан назад на входящий путь. Например, если  $n = 3$ , и мы имеем вагоны 1, 2 и 3 на левом пути, тогда вагон 3 первым въезжает на ветку. Мы можем послать вагон 2 на ветку, а затем далее на правый путь, затем послать туда же вагон 3, затем 1, в результате чего порядок вагонов изменится на 1, 3, 2.

- (a) Для  $n = 3$  найдите все возможные перестановки вагонов, которые можно получить с помощью описанной схемы.
- (b) Для  $n = 4$  найдите все возможные перестановки вагонов, которые можно получить с помощью описанной схемы.
- (c) [Вопрос повышенной сложности] Для произвольного  $n$  найдите, сколько различных перестановок можно получить с использованием стека.

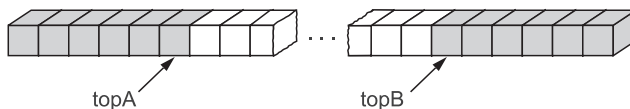


**Рис. 3.5.** Перестановочная ветка для изменения порядка элементов с помощью стека

- Е4.** Напишите функцию `StackSize` для непрерывной реализации стека, описанной в этом разделе.
- Е5.** (a) Напишите процедуру `ClearStack`, которая реинициализирует стек, делая его пустым. Используйте непрерывную реализацию.  
 (b) Учитывая сходство этой процедуры с процедурой `CreateStack`, ответьте на вопрос, почему все-таки целесообразно иметь `ClearStack` в качестве отдельной процедуры?

- E6.** Напишите процедуру `StackTop` (`var x: stackentry; var S: stack`).
- (a) При разработке `StackTop` используйте процедуры `Push` и `Pop`. Ваша процедура управления стеком должна использовать только базовые подпрограммы и при этом не должна опираться на методы реализации.
  - (b) Напишите процедуру `StackTop` на низком уровне, обращаясь непосредственно к содержимому массива. Используйте непрерывную реализацию из этого раздела.
  - (c) Каковы преимущества и недостатки каждого из этих способов построения процедуры `StackTop`?
- E7.** (a) Напишите процедуру `TraverseStack` для непрерывной реализации.
- (b) Возможно ли при разработке процедуры `TraverseStack` обойтись только базовыми операциями над стеком, не используя ни детали реализации, ни вспомогательную память (например, отдельный массив или второй стек)?
- E8.** Напишите следующие Pascal-процедуры и функции для связного стека:
- (a) процедуру `CreateStack`
  - (b) процедуру `ClearStack`
  - (c) функцию `StackEmpty`
  - (d) функцию `StackFull`
  - (e) процедуру `PopNode`
  - (f) процедуру `Pop`
  - (g) процедуру `StackTop`
  - (h) процедуру `TraverseStack`
- E9.** Функция `StackSize` для связного стека требует включения в нее цикла, который просматривает весь стек для подсчета числа его элементов, поскольку число элементов в стеке не хранится в отдельном поле записей, составляющих стек.
- (a) Напишите Pascal-функцию `StackSize` для связного стека с использованием цикла, в котором переменная-указатель перемещается от узла к узлу стека.
  - (b) Подумайте, как можно изменить объявление связного стека, чтобы преобразовать стек в структуру с двумя полями, вершиной стека и счетчиком, хранящим его размер. Какие изменения придется внести во все процедуры и функции для связных стеков? Обсудите преимущества и недостатки такой модификации по сравнению с исходной реализацией связных стеков.
- E10.** Иногда программа требует наличия двух стеков с элементами одного типа. Если два стека сохранять в отдельных массивах, то один из стеков может переполниться, в то время как в другом будет еще много неиспользованного места. Изящный способ избавиться от этой проблемы заключается в выделении достаточного пространства в *одном* массиве и организации стеков таким образом, чтобы один стек рос от одного конца массива, а другой стек начинался на другом конце и рос в противоположном направлении, к первому стеку. В этом случае, если один из стеков окажется большим, а другой

маленьким, они оба поместятся в этот массив, и переполнения не будет до тех пор, пока не используется все выделенное пространство. Объявите новый тип записи `doublestack`, включающей в себя массив и две вершины `topA` и `topB`, и напишите процедуры `CreateDoubleStack`, `PushA`, `PushB`, `PopA` и `PopB` для управления двумя стеками внутри одной записи `doublestack`.



### Программные проекты 3.1

- P1. (a)** Соберите пакет из объявлений, процедур и функций для обработки непрерывных стеков, готовый для использования прикладной программой (либо в качестве модуля Turbo Pascal, либо посредством директивы `include` или другого подобного средства). Если разрабатывается пакет модулей, возможно, придется создать несколько отдельных модулей. Разработайте модули стеков, использующих символы, действительные числа и целые. Для поддержки демонстрационных программ включите в пакет процедуру `TraverseStack`.
- (b)** Соберите пакет из объявлений, процедур и функций для обработки связанных стеков, готовый для использования прикладной программой (либо в качестве модуля Turbo Pascal, либо посредством директивы `include` или другого подобного средства). При разработке модуля Turbo Pascal процедуры `PushNode` и `PopNode` следует объявить закрытыми (`private`): они должны быть видны только в реализации части.
- (c)** Напишите демонстрационную программу, которую можно будет использовать для проверки процедур и функций этого раздела, предназначенных для управления стеками. Ваша программа должна быть построена по аналогии с программой, рассмотренной в разделе 2.3.3; используйте для своей программы максимум ее кода. Элементами вашего стека должны быть символы. Если вы используете модули, возьмите из проекта P1(a) модуль, содержащий символы в качестве элементов стека. Ваша программа должна выводить однострочное меню, из которого пользователь может выбрать любую операцию со стеком. После выполнения программой запрошенной операции она должна информировать пользователя о результате и запрашивать следующее действие. Если пользователь пожелает протолкнуть символ в стек, программа должна запрашивать, какой символ ей использовать.

Используйте непрерывную реализацию стеков и проследите за поддержкой принципов сокрытия информации. Вы не должны вносить изменений в пакет подпрограмм для операций со стеками, чтобы получить возможность заменить эти подпрограммы но-



выми вариантами; в этом случае после компиляции ваша демонстрационная программа должна работать так же правильно, как до замены.

- (d) Замените пакет для операций со непрерывными стеками эквивалентным пакетом для связанных стеков. Не вносите никаких других изменений и убедитесь, что ваша программа работает правильно.

**P2.** Напишите программу, которая использует стек для чтения числа и вывода всех его простых сомножителей в убывающем порядке. Например, для числа 2100 вывод должен быть таким:

7 5 5 3 2 2

[Подсказка: наименьший делитель, больший 1, для любого целого числа обязательно будет простым]

**P3.** Напишите программу, использующую операции со стеком, которая читает текстовый файл и проверяет парность всех скобок, встретившихся в этом файле; другими словами, для каждой открывающей скобки типа (, [ или { должна быть закрывающая скобка того же типа, причем с правильным уровнем вложенности. Если встретилась конструкция вроде ( ... [ ...] ...), или если скобка не имеет пары, программа должна вывести сообщение об ошибке вместе с номером строки, где эта ошибка обнаружена.

простые  
множители

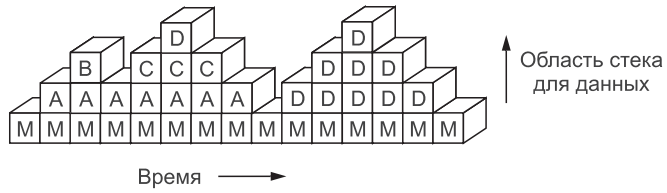
## 3.2. Введение в рекурсию

### 3.2.1. Кадры стека для подпрограмм

Одно из важных применений стеков связано с работой подпрограмм. Рассмотрим, что происходит в компьютере при вызове подпрограммы. Система (или программа) должна помнить место, где был выполнен вызов, чтобы иметь возможность вернуться туда после завершения подпрограммы. Вызывающая программа должна также помнить все локальные переменные, регистры процессора и прочие данные, чтобы эта информация не была потеряна в процессе работы подпрограммы. Мы можем рассматривать всю эту информацию как одну большую запись, временную область памяти для каждой подпрограммы.

Предположим, что у нас есть три подпрограммы с именами *A*, *B* и *C*, и пусть *A* вызывает *B*, а *B* вызывает *C*. Тогда *B* не закончит свое выполнение, пока не завершится *C* и не произойдет возврат из нее. Точно так же, *A* первой начинает свое выполнение, но завершается последней, только после того как завершится (с возвратом) *B*. Таким образом, последовательность выполнения подпрограмм может быть сформулирована как свойство «последним вошел, первым вышел» (last in, first out, LIFO). Если мы теперь рассмотрим задачу машины по выделению временных областей памяти для использования подпрограммами, то эти области должны выделяться в виде списка с описанным свойством, т. е. в виде стека (рис. 3.6). Поэтому стек играет ключевую роль при вызове подпрограмм в компьютерной системе.

области памяти  
для подпрограмм



**Рис. 3.6.** Кадры стека при вызове подпрограмм  
(М — главная программа, А, В, С и D — подпрограммы)

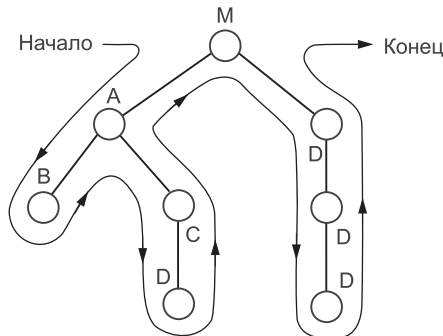
определение  
рекурсии

Из рис. 3.6 видно, что не имеет никакого значения, проталкиваются ли в стек временные области памяти для выполнения разных подпрограмм или при повторных вызовах одной и той же подпрограммы. Ситуация, когда подпрограмма вызывает саму себя или последовательность других подпрограмм, которые в конце концов снова вызывают первую подпрограмму, носит название *рекурсии*. В отношении кадров стека для вызова подпрограмм рекурсия несколько не отличается от любых других вызовов подпрограмм.

### 3.2.2. Дерево вызовов подпрограмм

Еще один графический метод подчеркивает связь между стеками и вызовами подпрограмм. Это так называемая древовидная схема, показывающая порядок, в котором вызываются подпрограммы. Такая схема, соответствующая кадрам стека из рис. 3.6, изображена на рис. 3.7. Главная программа показана как корень дерева, а все вызовы, выполняемые непосредственно в главной программе, изображены в виде узлов, расположенных сразу под корнем. Каждая из этих подпрограмм может, разумеется, вызывать другие подпрограммы, что показано в виде узлов, расположенных на следующих уровнях. В результате дерево растет и принимает форму вроде той, что изображена на рис. 3.7. Мы будем называть такое дерево *деревом вызовов подпрограмм*.

Мы начинаем с вершины дерева, которая называется *корнем* и соответствует главной программе, и последовательно обходим все узлы, как это показано на рис. 3.7 линией со стрелками. Этот путь носит название



**Рис. 3.7.** Дерево вызовов подпрограмм

*обхода* или *просмотра* дерева. Каждый кружок (называемый *вершиной* или *узлом*) соответствует вызову подпрограммы. Когда мы, двигаясь вниз, подходим к узлу, мы вызываем подпрограмму. После обхода части дерева вокруг узла мы опять, двигаясь вверх, подходим к предыдущему узлу, и это знаменует завершение подпрограммы и возврат из нее.

Нас особенно интересует рекурсия, поэтому мы часто рисуем только часть дерева, показывающую рекурсивные вызовы, и называем ее *деревом рекурсии*.

Из приведенного рисунка прежде всего следует, что нет никакой разницы между рекурсивным вызовом и вызовом любой другой подпрограммы. Последовательные рекурсивные вызовы изображаются на рисунке просто как отдельные узлы, которые имеют одно и то же имя относящейся к ним подпрограммы. Далее, очень важно отметить, что дерево вызовов показывает *вызовы* подпрограмм, но не вложения *объявлений* подпрограмм. Поэтому подпрограмма, вызываемая только в одной точке программы, но внутри цикла, выполняемого более одного раза, появится в дереве вызовов несколько раз, по одному разу на каждый шаг цикла. И наоборот, если подпрограмма вызывается из условного предложения, которое не выполняется, такой вызов не появится в дереве вызовов.

Кадры стеков вроде тех, что изображены на рис. 3.6, показывают вложенность рекурсивных вызовов, а также иллюстрируют требования к выделению памяти для рекурсии. Если процедура рекурсивно вызывает сама себя несколько раз, тогда для каждого рекурсивного вызова создаются отдельные копии переменных, объявленных в этой процедуре. При обычной реализации рекурсии эти копии сохраняются в стеке. Заметьте, что объем памяти, требуемый для такого стека, пропорционален высоте дерева рекурсии, а не полному числу узлов на дереве. Другими словами, объем памяти, требуемый для реализации рекурсивной процедуры, зависит от *глубины* рекурсии, а не от *числа* вызовов подпрограммы.

Последние два рисунка фактически можно интерпретировать в более широком контексте, чем процесс вызова подпрограмм. Эти рисунки подчеркивают простое, но важное обстоятельство, заключающееся в близкой связи между произвольными деревьями и стеками:

### Теорема 3.1

*В процессе обхода (просмотра) любого дерева узлы добавляются к пути или удаляются из него так же, как и элементы стека. И наоборот, для любого стека можно изобразить дерево, отображающее жизненную историю стека, т. е. ход проталкивания или выталкивания элементов стека.*

Теперь мы можем заняться изучением нескольких простых примеров рекурсии. После этого мы проанализируем, каким образом обычно осуществляется реализация рекурсии на компьютере. По ходу этого обсуждения мы получим рекомендации относительного удачного и неудачного применения рекурсии, когда ее следует использовать, и когда ее лучше избегать. Глава завершается примерами несколько более изощренных применений рекурсии.

дерево  
рекурсии

ход выполнения

кадры стеков

требования  
к памяти

### 3.2.3. Факториалы: рекурсивное определение

В математике **факториалом** называется функция от неотрицательного целого, которая обычно определяется формулой

неформальное  
определение

$$n! = n \times (n - 1) \times \dots \times 1.$$

Три точки в этой формуле обозначают «продолжайте таким же образом». Это определение не вполне точно, поскольку существует более одного способа заполнить промежуток с тремя точками. Для вычисления факториала нам нужно более точное определение, вроде следующего:

формальное  
определение

$$n! = \begin{cases} 1 & \text{если } n = 0 \\ n \times (n - 1)! & \text{если } n > 0. \end{cases}$$

Это определение однозначно говорит нам, как следует вычислять факториал при условии, что мы аккуратно следуем этому правилу и используем лист бумаги, чтобы не забывать, где мы в каждый момент находимся.

пример

Пусть нам надо вычислить  $4!$ . Поскольку  $4 > 0$ , определение говорит нам, что  $4! = 4 \times 3!$ . Это может нам помочь, но не очень, так как мы не знаем значения  $3!$ . Поскольку  $3 > 0$ , определение опять дает нам  $3! = 3 \times 2!$ . Опять, мы не знаем значения  $2!$ , но определение дает нам  $2! = 2 \times 1!$ . Мы все еще не знаем  $1!$ , но поскольку  $1 > 0$ , мы имеем  $1! = 1 \times 0!$ . Определение рассматривает случай  $n = 0$  отдельно, так что мы знаем, что  $0! = 1$ . Теперь мы можем подставить этот ответ в выражение для  $1!$  и получить  $1! = 1 \times 0! = 1 \times 1 = 1$ . Здесь-то нам и может пригодиться лист бумаги, на котором мы можем отображать частичные результаты. Если мы не записываем систематическим образом получаемые результаты, то, обращаясь к нашему определению несколько раз, мы неминуемо забудем ход ранних стадий процесса, еще не дойдя до самого нижнего уровня, где мы должны начать использовать результаты выполненных ранее вычислений. Для вычисления факториала нетрудно систематическим образом записать все выполняемые шаги:

$$\begin{aligned} 4! &= 4 \times 3! \\ &= 4 \times (3 \times 2!) \\ &= 4 \times (3 \times (2 \times 1!)) \\ &= 4 \times (3 \times (2 \times (1 \times 0!))) \\ &= 4 \times (3 \times (2 \times (1 \times 1))) \\ &= 4 \times (3 \times (2 \times 1)) \\ &= 4 \times (3 \times 2) \\ &= 4 \times 6 \\ &= 24. \end{aligned}$$

редукция  
задачи

Приведенный пример иллюстрирует сущность рекурсии. Для получения ответа на сложную задачу используется общий метод, который разлагает сложную задачу на одну или несколько задач, схожих по существу, но меньшего размера. Тот же общий метод используется затем для решения этих подзадач, и в том же духе рекурсия продолжается до полу-

чения подзадачи самого маленького размера, базового экземпляра, для которого решение определяется непосредственно без дальнейших рекурсий. Другими словами:

*Каждый рекурсивный процесс состоит из двух частей:*

1. Самый простой, базовый экземпляр задачи, который вычисляется без рекурсии; и
2. Общий метод, который уменьшает (редуцирует) конкретный экземпляр с получением одного или нескольких меньших экземпляров, позволяя, таким образом, последовательно редуцировать задачу до достижения базового экземпляра минимального размера.

Pascal (как и большинство других современных компьютерных языков) предоставляет простой способ организации рекурсии. Вычисление факториала на языке Pascal реализуется следующей функцией:

```
function Factorial(n: integer): integer; {Факториал}
{ Pre:  n есть неотрицательное целое число.
  Post: Значение функции представляет собой факториал от n. }
begin
  if n = 0 then Factorial := 1
  else Factorial := n * Factorial(n - 1)
end;
```

Как можно увидеть из этого примера с факториалом, рекурсивное определение и рекурсивное решение задачи могут быть точными и элегантными, но при реальном воплощении метода получение конечного результата может потребовать аккуратного прослеживания многих частичных вычислений.

Компьютеры хорошо приспособлены для прослеживания таких частичных вычислений, а вот человеческий мозг плохо справляется с такого рода задачами. Очень сложно запоминать длинную цепочку частичных результатов, а затем двигаться через них в обратном направлении до завершения всей работы. Таким образом, когда мы используем рекурсию, нам необходимо рассуждать несколько не так, как при использовании других программистских методов. Программисты должны понимать общую картину и возлагать детали вычислений на компьютер.

#### Замечание

В нашем алгоритме мы должны точно описать общий ход редукции большой задачи, т. е. преобразование ее в меньшие по размеру экземпляры задачи; мы должны определить правило остановки преобразования при получении наименьшего экземпляра, и как этот экземпляр обрабатывается. С другой стороны, за исключением нескольких простых и небольших примеров, мы *не должны* пытаться разобраться в рекурсивном алгоритме, прослеживая общий случай вниз до выполнения правила остановки или путем трассировки действий, которые компьютер предпримет при решении задачи достаточно большого размера. Мы очень быстро настолько запутаемся в отложенных подзадачах, что потеряем представление о задаче в целом и о ходе выполнения общего метода поиска ее решения.

рекурсивная  
программа

запоминание  
частичных  
результатов

Разработаны неплохие общие методы и средства, позволяющие нам сконцентрировать свое внимание на этих общих методах и ключевых шагах, одновременно оценивая объем работы, которую придется выполнить компьютеру при реализации им всех деталей алгоритма задачи. Обратимся теперь к примеру, который проиллюстрирует некоторые из этих методов и средств.

### 3.2.4. Метод разбиения: башни Ханоя

#### 1. Задача

В девятнадцатом столетии в Европе появилась игра, названная *Башни Ханоя*, вместе с рекламными материалами (несомненно, недостоверными), поясняющими, что эта игра представляет собой задачу, выполняемую жрецами в Храме Брамь. При создании мира жрецам была дана медная платформа с тремя алмазными стержнями. На первый стержень были надеты 64 золотых диска, каждый следующий слегка меньше предыдущего. (Менее экзотический вариант, продаваемый в Европе, имел 8 картонных дисков и 3 деревянные палочки.) Жрецам вменялось в обязанность переместить золотые диски с первого стержня на третий, соблюдая следующие условия: за раз можно перемещать только один диск, и никакой диск не может быть положен на диск меньшего размера. Жрецам было объявлено, что когда они закончат перемещение всех 64 дисков, наступит конец света. См. рис. 3.8.

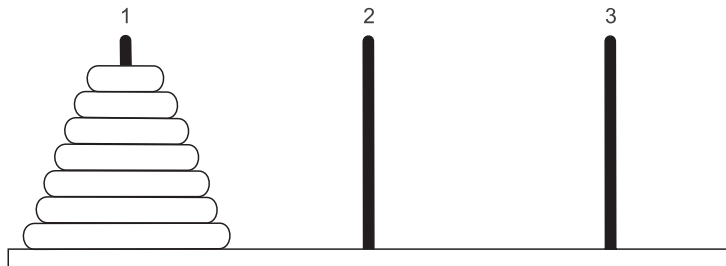


Рис. 3.8. Башни Ханоя

Нашей задача, разумеется, состоит в написании компьютерной программы, которая выведет список инструкций для жрецов. Мы можем кратко описать нашу задачу выражением

Move (64, 1, 3, 2)

что означает

*Переместить 64 диска от башни 1 к башне 3, используя башню 2 в качестве временной памяти.*

#### 2. Решение

Для того, чтобы найти решение, мы должны сконцентрировать наше внимание не на первом шаге (который заключается в перемещении

куда-либо первого диска), а на более трудной задаче: перемещение самого нижнего диска. Нижнего диска нельзя достичь, не переместив все 63 диска над ним, и, более того, все они должны быть нанизаны на стержень 2 (помещены на башню 2), чтобы мы могли переместить нижний диск от башни 1 к башне 3. Ведь за раз можно перемещать только по одному диску, а нижний (самый большой) диск не может находиться сверху любого другого, поэтому, когда мы перемещаем нижний диск, на башнях 1 или 3 не должно быть никаких других дисков. Итак, мы определяем шаги нашего алгоритма игры *Башни Ханоя* следующим образом:

```
Move(63, 1, 2, 3);
Writeln('Перемещаем диск 64 с башни 1 на башню 3.');
```

```
Move(63, 2, 3, 1)
```

обобщенная  
редукция

Мы вделали один небольшой шаг к решению, правда, очень небольшой, поскольку мы все еще должны описать, как переместить 63 диска (дважды). Тем не менее наш шаг очень важен, так как нет причин, почему бы мы не могли переместить 63 оставшихся диска тем же самым способом. (Фактически мы должны переместить их тем же способом, так как опять самый большой диск должен перемещаться последним.)

Это и есть принцип рекурсии. Мы описали, как выполнить ключевой шаг и удостоверились, что оставшаяся часть задачи решается принципиально тем же способом. Это также принцип *разбиения*: для решения задачи мы разбиваем, расщепляем ее на меньшие и меньшие части, каждую из которых решить проще, чем исходную задачу.

### 3. Детализация

Для формального определения алгоритма нам нужно знать на каждом шаге, какой стержень можно использовать в качестве временной памяти; поэтому мы вызываем процедуру перемещения Move со следующими спецификациями:

<p><b>procedure</b> Move(count, start, finish, temp);      { <i>Переместить</i> }</p> <p><i>предусловие</i>: На башне start имеются по крайней мере count дисков. Верхний диск (если таковой имеется) на каждой из башен temp и finish имеет больший размер, чем любой из верхних count дисков на башне start.</p> <p><i>постусловие</i>: Верхние count дисков на башне start перемещены на башню finish; башня temp (используемая в качестве временной памяти) вернулась в свое исходное состояние.</p>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Очевидно, что наша задача должна быть выполнена за конечное число шагов (даже если ее решение знаменует собой конец мира!), и, следовательно, должен быть какой-то способ остановить рекурсию. Очевидное правило остановки заключается в том, что если не осталось дисков для перемещения, то больше и делать ничего не надо. Мы теперь можем написать полную программу, учитывающее сформулированное правило. Главная программа выглядит следующим образом:

правило  
остановки



```

program Hanoi (output);                                { Программа Ханой }
{ Pre:  Предусловия отсутствуют.
  Post: Моделирование задачи Башен Ханоя завершено.
  Uses: Рекурсивно использует процедуру Move. }
const ndisks = 4;                                       { число дисков на первой башне }
type
  disk = 0..ndisks;
  tower = 1..3;
begin                                                    { главная программа Hanoi }
  Move(ndisks, 1, 3, 2)
end.                                                    { главная программа Hanoi }

```

Рекурсивная процедура, выполняющая всю работу:

```

procedure Move(count: disk; start, finish, temp: tower);
begin                                                    { процедура Move }
  if count > 0 then begin
    { Мы начинаем с того, что все count дисков находятся на башне start,
      а башни finish и temp пусты. Таким образом, предусловия для Move
      удовлетворяются. }
    Move(count - 1, start, temp, finish);
    { Согласно постусловиям для Move, верхние count - 1 дисков теперь
      правильно располагаются на башне temp, а башня finish пуста.
      Нижний, самый большой диск остается на башне start. }
    writeln('Перемещаем диск ', count: 2, 'с башни ', start: 2);
    writeln('на башню ', finish: 2, '.');
    { Нижний (самый большой) диск теперь перемещен в свою конечную
      позицию на башне finish; башня start пуста; и все count - 1 дисков на
      башне temp имеют меньший размер, чем единственный диск на башне
      finish. Таким образом, удовлетворяются предусловия для следующего
      вызова процедуры Move. }
    Move(count - 1, temp, finish, start)
    { Постусловия для Move теперь утверждают, что все диски находятся
      на башне finish, а остальные башни пусты. }
  end
end;                                                    { процедура Move }

```

рекурсивная  
процедура

#### 4. Трассировка программы

Весьма полезным средством изучения рекурсивной процедуры применительно к очень простому примеру является конструирование трассировки ее действий. Такая трассировка показана на рис. 3.9 для игры «Башни Ханоя» в случае, когда число дисков ndisks равно 2. Каждая рамка на диаграмме показывает, что происходит в одном вызове. Самый внешний вызов Move(2,1,3,2) (вызов, сделанный в главной программе) приводит к выполнению следующих трех предложений, показанных во внешней, самой большой рамке рисунка:

```

Move(1, 1, 2, 3);          { Перемещаем 1 диск с башни 1 на башню 2,
                           используя башню 3 }
writeln('Перемещаем диск 2 с башни 1 на башню 3');
Move(1, 2, 3, 1);          { Перемещаем 1 диск с башни 2 на башню 3,
                           используя башню 1 }

```



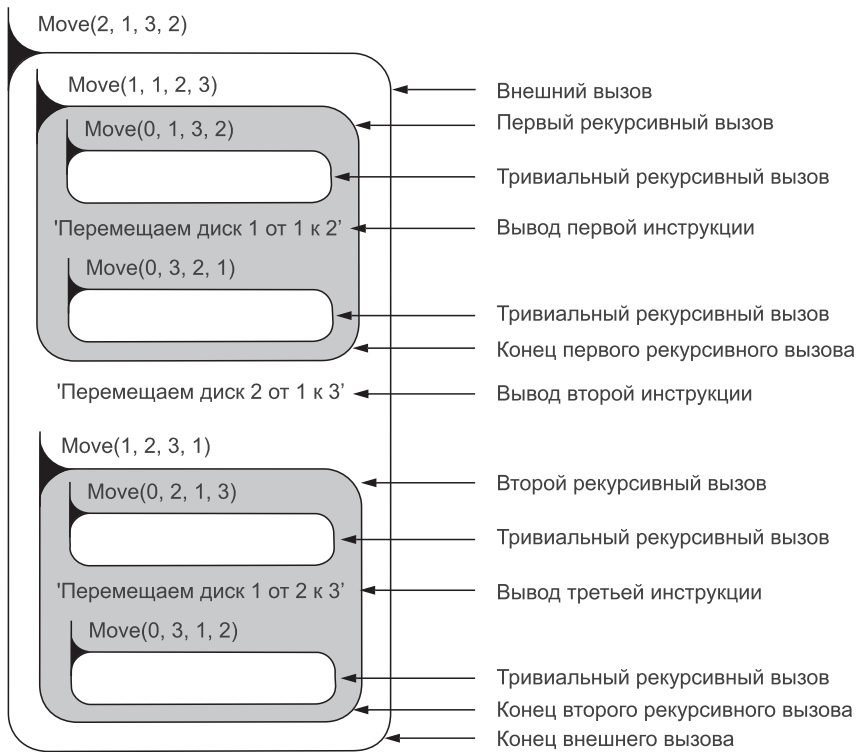


Рис. 3.9. Трассировка программы NANOI для ndisks = 2

Выполнение первого и третьего из этих предложений требует рекурсивных вызовов. Предложение `Move(1, 1, 2, 3)` означает, что процедура `Move` снова начинает выполнение сверху, но уже с новыми параметрами, что приводит к выполнению следующих трех предложений, показанных на рис. 3.9 в первой внутренней рамке (закрашенной серым цветом):

```
Move(0, 1, 3, 2);           { Перемещаем 0 дисков }
writeln('Перемещаем диск 1 с башни 1 на башню 2');
Move(0, 3, 2, 1);         { Перемещаем 0 дисков }
```

При желании вы можете считать, что эти три предложения написаны в программе вместо вызова `Move(1, 1, 2, 3)`, но лучше показать их на рисунке отдельно и придать им другой цвет, поскольку они составляют новый и отличный от первого вызов процедуры. На нашем рисунке они помещены в серую рамку.

Ниже рамки, соответствующей вызову `Move(1, 1, 2, 3)`, следует предложение `writeln`, а за ним вторая закрашенная серым внутренняя рамка, соответствующая вызову `Move(1, 2, 3, 1)`. Однако перед тем, как программа перейдет к выполнению этих предложений, она должна выполнить еще два рекурсивных вызова, исходящих из первой внутренней рамки. Следова-

тельно, мы должны были бы расширить вызов `Move(0, 1, 3, 2)`. Однако процедура `Move` ничего не делает, если параметр `count` равен 0, поэтому этот вызов `Move(0, 1, 3, 2)` не приводит к следующим вызовам процедуры или каким-либо иным программным предложениям. Мы изображаем этот вызов на рисунке в виде первой пустой рамки и называем его тривиальным рекурсивным вызовом.

После этого «пустого» вызова идет предложение `writeln`, показанное в первой внутренней серой рамке, а за ним – второй вызов `Move(0, 3, 2, 1)`, также не приводящим ни к каким действиям. На этом завершаются действия вызова `Move(1, 1, 2, 3)`, который возвращает управление в точку, из который он был сделан. Следующим является предложение `writeln` во внешней рамке и, наконец, программа переходит к выполнению процедуры `Move(1, 2, 3, 1)`. Этот вызов порождает выполнение предложений, показанных во второй внутренней рамке, которые, в свою очередь, расширяются в две пустые рамки.

Продравшись сквозь все эти рекурсивные вызовы, вы, возможно, обнаружили сходство рекурсии с известной басней об ученике мага, который, заколдовав метлу, чтобы та принесла ему воды, не знал, как ее остановить, и решил расщепить метлу пополам, после чего метла сама стала делиться, и вскоре метел, приносящих воду, стало так много, что если бы не вернулся маг, неминуемо произошло бы несчастье. Теперь мы обратимся к другому средству, позволяющему визуализировать рекурсивные вызовы, средству, отображающему множество вызовов более эффективно, чем трассировка программы. Это средство носит название дерева рекурсии.

ученик мага

5. Анализ

Дерево рекурсии для Башен Ханоя с тремя дисками показано на рис. 3.10, а ход выполнения изображен в виде линий со стрелками.

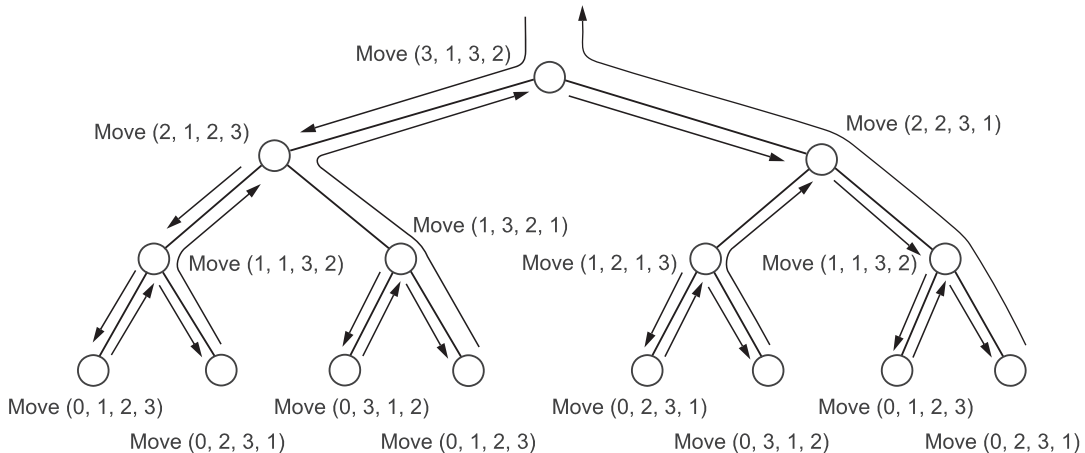


Рис. 3.10. Дерево рекурсии для трех дисков

Заметьте, что наша программы для решения задачи о Башнях Ханоя не только находит полное решение задачи, но находит лучшее из возможных решений, и, фактически, единственно возможное решение, если не считать решений с включенными избыточными и бессмысленными предложениями вроде

Перемещаем диск 5 с башни 1 на башню 2.

Перемещаем диск 5 с башни 2 на башню 3.

Перемещаем диск 5 с башни 3 на башню 1.

Чтобы показать единственность несократимого решения, заметьте, что на каждом этапе задача заключается в том, чтобы переместить определенное число дисков от одной башни к другой. Не существует никакого другого решения этой задачи, кроме как перемещения сначала всех дисков, кроме самого нижнего, затем, возможно, выполнения нескольких избыточных перемещений, далее перемещения нижнего диска, возможно, опять выполнения нескольких избыточных перемещений и, наконец, повторного перемещения верхних дисков.

глубина  
рекурсии

Далее посмотрим, сколько раз будет выполняться рекурсия перед возвратом и завершением. Когда процедура Move выполняется первый раз, count = 64, и каждый рекурсивный вызов уменьшает значение count на 1. Таким образом, если исключить вызовы с count = 0, которые не выполняют никаких действий, полная глубина рекурсии составляет 64. Другими словами, если бы мы рисовали дерево рекурсивных вызовов для программы, оно должно было бы иметь над своими листьями 64 уровня. Если исключить листья, в каждом узле выполняется по два рекурсивных вызова (и, кроме того, одна команды вывода инструкции), и, следовательно, число узлов на каждом уровне в точности в два раза больше числа узлов на уровне над ним.

Представив себе дерево рекурсии (даже если оно слишком велико для отображения на бумаге), мы можем с легкостью вычислить, сколько команд требуется для перемещения 64 дисков. Для каждого узла на дереве (за исключением листьев, которые соответствуют вызовам со значением параметра count = 0) выполняется одна команда печати. Число узлов, не являющихся листьями, равно

$$1 + 2 + 4 + \dots + 2^{63} = 2^0 + 2^1 + 2^2 + \dots + 2^{63} = 2^{64} - 1,$$

а это и есть число ходов, требуемое для перемещения всех 64 дисков.

Чтобы оценить, насколько велико это число, учтем такую аппроксимацию:

$$10^3 = 1000 < 1024 = 2^{10}$$

(Этот простой факт стоит того, чтобы его запомнить: сокращение К, как в числе 512К, означает 1024.) Один год состоит приблизительно из  $3.2 \times 10^7$  секунд. Предположим, что команды будут выполняться с бешеной скоростью одна команда каждую секунду (жрецы, в конце концов, напрактиковались). Поскольку

$$2^{64} = 24 \times 2^{60} > 2^4 \times 10^{18} = 1.6 \times 10^{19}$$

вся задача потребует приблизительно  $5 \times 10^{11}$  лет. Если астрономы оценивают возраст вселенной приблизительно в 20 миллиардов ( $2 \times 10^{10}$ )

время  
и пространство  
памяти

лет, тогда, согласно легенде, мир будет еще существовать довольно долго — в 25 раз дольше, чем он уже существует!

Следует подчеркнуть, что хотя компьютер не сможет выполнить задачу о Башнях Ханоя, это происходит из-за недостатка *времени*, а совсем не из-за недостатка *пространства памяти*. Память нужна лишь для того, чтобы хранить путь 64 рекурсивных вызовов, а вот времени потребуется столько, сколько нужно для выполнения  $2^{64}$  вычислений.

### Упражнения 3.2

**Е1.** Рассмотрите функцию  $f(n)$ , определенную следующим образом, где  $n$  является неотрицательным целым числом:

$$f(n) = \begin{cases} 0 & \text{если } n = 0; \\ f(\frac{1}{2}n) & \text{если } n \text{ четно, } n > 0; \\ 1 + f(n-1) & \text{если } n \text{ нечетно, } n > 0. \end{cases}$$

Вычислите значение  $f(n)$  для следующих значений  $n$ .

- |               |                 |
|---------------|-----------------|
| (a) $n = 1$ . | (d) $n = 99$ .  |
| (b) $n = 2$ . | (e) $n = 100$ . |
| (c) $n = 3$ . | (f) $n = 128$ . |

**Е2.** Рассмотрите функцию  $f(n)$ , определенную следующим образом, где  $n$  является неотрицательным целым числом:

$$f(n) = \begin{cases} 0 & \text{если } n \leq 1; \\ n + f(\frac{1}{2}n) & \text{если } n \text{ четно, } n > 1; \\ f(\frac{1}{2}(n+1)) + f(\frac{1}{2}(n-1)) & \text{если } n \text{ нечетно, } n > 1. \end{cases}$$

Для каждого из следующих значений  $n$  нарисуйте дерево рекурсии и вычислите значение  $f(n)$ .

- |               |               |
|---------------|---------------|
| (a) $n = 1$ . | (d) $n = 4$ . |
| (b) $n = 2$ . | (e) $n = 5$ . |
| (c) $n = 3$ . | (f) $n = 6$ . |

**Е3.** Пусть  $S(n)$  есть сумма целых чисел от 1 до  $n$  для неотрицательного целого  $n$ . Напишите Pascal-функцию, которая будет вычислять  $S(n)$  каждым из следующих способов.

- Используйте рекурсивную формулу  $S(n) = n + S(n-1)$  для  $n > 0$  и  $S(0) = 0$ .
- Используйте локальную переменную, инициализируйте ее нулем, и напишите цикл, вычисляющий сумму  $S(n) = 1 + 2 + \dots + n$ .
- Используйте формулу (все еще не доказанную)  $S(n) = \frac{1}{2}n(n+1)$ .
- Какой из указанных вариантов легче написать? Какой из них проще понять человеку, не владеющему математикой и программированием? Какой вариант будет самым быстрым для больших значений  $n$ ?

## Программные проекты 3.2

- P1.** Сравните время выполнения<sup>2</sup> рекурсивной функции для вычисления факториала, описанной в этом разделе, и нерекурсивной функции, полученной путем инициализации локальной переменной значением 1 и использованием затем цикла для вычисления произведения  $n! = 1 \times 2 \times \dots \times n$ . Для того, чтобы получить возможность достоверного сравнения времени работы процессора, вам, скорее всего, придется написать в своей программе драйвера цикл, который будет повторять одни и те же вычисления факториала несколько сотен раз. Если вы попытаетесь вычислить факториал большого числа, может произойти целочисленное переполнение. Для предотвращения такой ситуации вы можете объявить переменную  $n$  и значение функции как числа с плавающей точкой, т. е. `real` вместо `integer`.
- P2.** Подтвердите, что время выполнения программы `Nanoi` возрастает приблизительно как константа, умноженная на  $2^n$ , где  $n$  — число перемещаемых дисков. Для выполнения этой оценки сделайте величину `ndisks` переменной, закомментируйте строку, которая выводит на экран сообщение для пользователя и запустите программу для нескольких значений `ndisks`, например, для 10, 11, ..., 15. Как изменяется время процессора от одного значения `ndisks` до другого?

## 3.3. Принципы рекурсии

### 3.3.1. Разработка рекурсивных алгоритмов

Рекурсия — это средство, позволяющее программисту сконцентрировать свои усилия на ключевом шаге алгоритма, и не думать об объединении этого шага со всеми остальными. Как это обычно делается при решении задач, прежде всего вы должны рассмотреть несколько простых примеров, а после того, как они будут в достаточной степени поняты, попытаться сформулировать метод, который будет работать в более общем случае. Некоторые важные шаги при разработке рекурсивных алгоритмов описаны ниже.

- Найдите ключевое действие. Задайте себе вопрос «Каким образом эту задачу можно разделить на части?» или «Как будет выполняться ключевое действие в этой задаче?». Постарайтесь, чтобы ваши ответы были краткими, но сформулированы в общем виде. Не рассматривайте пока множество частных случаев; рассмотрите сначала задачу небольшого размера или лишь начало и конец большой задачи. Когда вы делаете простой, пусть небольшой шаг к решению задачи, спросите себя, нельзя ли решить оставшуюся часть задачи посредством таких же или схожих шагов, а затем, если понадобится, модифицируйте найденный вами метод, чтобы он приобрел достаточную общность.

<sup>2</sup>

Вам понадобится пакет процедур для анализа процессорного времени; этот пакет различен в разных системах. Пакет для `Turbo Pascal` приведен в приложении С.

- Найдите правило остановки. Это правило показывает, что проблема или какая-то ее часть решена. Правило остановки обычно представляет собой простой особый случай, достаточно тривиальный, чтобы с ним можно было справиться без рекурсии.
- Опишите в общих чертах ваш алгоритм. Скомбинируйте правило остановки и ключевое действие, используя для выбора того или другого предложение **if**. Теперь вы можете приступить к написанию главной программы и рекурсивной процедуры, которая будет определять порядок повторного выполнения ключевого действия до тех пор, пока не сработает правило остановки.
- Проверьте завершение. Следующим, очень важным шагом является проверка завершения рекурсии во всех возможных случаях. Начните с какого-то обычного случая и убедитесь в том, что через конечное число шагов правило остановки сработает и рекурсия завершится. Проверьте также, правильно ли обрабатывает ваш алгоритм крайние случаи. Любой алгоритм, если он вызывается при таких условиях, которые не требуют от него никакой работы, должен аккуратно завершаться, но по отношению к рекурсивным алгоритмам это требование особенно важно, поскольку вызов с целью ничего не делать часто является реализацией правила остановки.
- Нарисуйте дерево рекурсии. Основным средством анализа рекурсивных алгоритмов является дерево рекурсии. Как мы уже видели в случае Башен Ханоя, высота дерева тесно связана с объемом требуемой памяти, а общий размер дерева отражает число выполнений ключевого действия и, следовательно, суммарное время, используемое программой. Обычно изображение дерева рекурсии для одного или двух простых примеров, соответствующих вашей задаче, оказывается весьма поучительным.

### 3.3.2. Как работает рекурсия

этапы разработки  
и реализации

Мы должны отчетливо разделять в наших рассуждениях ход фактического выполнения рекурсии в компьютере и вопрос использования рекурсии при разработке алгоритма. На этапе разработки следует использовать все методы, применимые для поиска решения задачи, и рекурсия является одним из наиболее гибких и мощных средств такого рода. На этапе реализации нам может понадобиться задать вопрос, какой из возможных методов является наилучшим при данных обстоятельствах. Реальное воплощение рекурсии в компьютерных системах может выполняться по меньшей мере двумя способами. Первый из них в настоящее время доступен только на больших вычислительных системах, хотя по мере изменения стоимости и возможностей компьютерного оборудования он вскоре может получить широкое распространение. Важный момент при сравнении двух различных реализаций заключается в том, что хотя ограничения по времени и пространству памяти обязательно должны рассматриваться, их следует рассматривать отдельно от процесса разработки алгоритма, поскольку различные виды компьютерного оборудования в будущем могут привести к различным возможностям и ограничениям.

## 1. Многопроцессорные конфигурации: параллельные вычисления

Возможно, самым естественным способом представления рекурсии является размещение рекурсивных подпрограмм не в различных частях одного компьютера, а так, что каждая подпрограмма выполняется на отдельной машине. В этом случае, когда одна подпрограмма запускает другую, она активизирует соответствующую машину, а когда эта машина завершает свою работу, она посылает ответ назад в первую машину, которая после этого продолжить выполнение своей задачи. Если процедура выполняет два рекурсивных вызова самой себя, она просто запускает две другие машины, и те начинают выполнять те же команды, что и первая машина. Когда эти машины завершают свою работу, они посылают ответы той машине, которая их активизировала. Если они, в свою очередь, делают рекурсивные вызовы, они просто запускают добавочные машины.

стоимость

Некогда процессор был самой дорогим компонентом компьютерной системы, и предложение построить систему с несколькими процессорами рассматривалось бы как крайне экстравагантное. Однако в наше время стоимость вычислительной мощности в сравнении с другими затратами на организацию вычислительной системы радикально снизилась, и весьма вероятно, что через короткое время мы увидим большие компьютерные системы, включающие среди своих компонентов сотни, если на тысячи идентичных микропроцессоров. Когда это произойдет, реализация рекурсии посредством множества процессоров станет если и не неизбежным, то, по крайней мере, самым обычным делом.

параллельные  
вычисления

При наличии множества процессоров программисты уже не станут рассматривать алгоритмы исключительно как линейную последовательность действий, а, наоборот, осознают, что некоторые части алгоритма можно выполнять параллельно (т. е. одновременно) с другими частями. Процессы, протекающие одновременно, так и называются *параллельными*. Изучение параллельных процессов и методов из взаимодействия является в настоящее время предметом интенсивных исследований в вычислительной технике, и эти исследования несомненно в ближайшие годы приведут к новым способам описания и реализации алгоритмов.

## 2. Однопроцессорная реализация: области памяти

С целью выяснения того, как наиболее эффективно реализовать рекурсию в однопроцессорной вычислительной системе, давайте на время оставим рекурсию и рассмотрим вопрос о том, какие шаги необходимо предпринять на примитивном уровне машинных команд простого компьютера, чтобы вызвать подпрограмму.

адрес возврата

Аппаратура любого компьютера имеет ограниченный набор команд, которые включают (наряду с другими командами) команды выполнения арифметических действий над указанными словами памяти или специальными ячейками внутри центрального процессора, называемыми регистрами, перемещения данных из памяти и регистров или в них, а также переходов (ветвлений) по указанным адресам. Когда вызываемая программа передает управление на начало вызываемой подпрограммы, адрес места вызова должен быть сохранен в памяти, в противном случае



локальные  
переменные

подпрограмма не сможет узнать, куда надо возвращаться после ее завершения. Адреса или значения параметров вызова также должны быть сохранены в таком месте, чтобы подпрограмма могла их найти, и где, в свою очередь, вызывающая программа сможет найти результаты работы завершившейся подпрограммы. Подпрограмма, начавшись, будет выполнять требуемые вычисления, используя локальные переменные и области памяти. Однако после завершения подпрограммы эти локальные переменные теряются, поэтому к ним нельзя обратиться вне этой подпрограммы. Далее, подпрограмма, разумеется, будет использовать по ходу своей работы регистры процессора для выполнения вычислений, потому после выхода из подпрограммы регистры, как правило, будут содержать иные значения, чем перед входом в нее. Однако принято считать, что подпрограмма не имеет права ничего изменять в программе, за исключением параметров вызова и глобальных переменных (последнее является побочным эффектом). Поэтому обычно подпрограмма сохраняет все используемые ею регистры в начале своей работы и восстанавливает их значения перед завершением и возвратом.

область памяти  
для временного  
хранения

Итак, при вызове подпрограммы она должна иметь область памяти (возможно, разбросанную по разным участкам) для хранения временных данных; она должна сохранить в этой области памяти содержимое регистров и всего того, что она может изменить по ходу своего выполнения; она также использует эту область для хранения адреса возврата, параметров вызова и локальных переменных. В процессе возврата подпрограмма восстанавливает регистры и другие объекты, которые она использовала, но должна вернуть в том же состоянии, в каком они были перед вызовом подпрограммы. После возврата подпрограмма уже не нуждается в содержимом временной области памяти.

При такой организации вычислительного процесса мы реализуем вызовы подпрограмм путем смены временных областей памяти вместо смены процессоров, как это мы предлагали делать выше. При этом не имеет значения, вызывается ли подпрограмма рекурсивно или нет, при условии, что в случае рекурсии мы рассматриваем последовательные рекурсивные вызовы как различные, не путая тем самым области временной памяти одних вызовов с областями других, точно так же, как мы не путали бы области памяти при вызове различных подпрограмм, если одна вызывается из другой. Для нерекурсивных подпрограмм в качестве области временной памяти может быть использован один выделенный заранее участок памяти, поскольку мы знаем, что следующий вызов подпрограммы может произойти только после возврата из предыдущего вызова, а когда предыдущий вызов завершился, временно сохраненная для него информация уже не нужна. Для рекурсивных подпрограмм, однако, сохраненная информация должна оставаться в неприкосновенности до выхода из внешнего вызова, поэтому внутренний вызов должен использовать для временной хранения отдельную область памяти.

Заметьте, что широко использовавшаяся в свое время практика резервирования постоянной области памяти для нерекурсивных подпрограмм была весьма неэффективной, поскольку таким образом навсегда



изымались из обращения значительные объемы памяти, которую можно было бы использовать для других целей в то время, когда подпрограммы неактивны. Однако именно таким образом выделялась память для подпрограмм в старых языках типа Fortran или Cobol, и именно по этой причине эти языки не допускают рекурсивных вычислений.

### 3. Реентерабельные программы

Существенно, что та же самая проблема множественных областей памяти возникает в совершенно другом контексте, при использовании *реентерабельных* программ. В больших системах с разделением времени многие пользователи могут одновременно использовать интерпретатор Basic или систему обработки текстов. Эти системные программы весьма велики, и было бы неразумной тратой памяти держать в ней тридцать или сорок копий абсолютно одинаковых наборов команд, по одному набору для каждого пользователя. Вместо этого большие системные программы типа текстового редактора пишут так, что их команды располагаются в одной области памяти, а адреса всех переменных или других данных хранятся в отдельной области памяти. В этом случае в памяти системы с разделением времени будет только одна копия команд, но по отдельной области данных для каждого пользователя.

Эта ситуация несколько напоминает группу студентов, пишущих ответы на вопросы теста в комнате, где эти вопросы написаны на доске. Все студенты читают единственный набор тестовых вопросов, однако каждый студент отдельно пишет свои ответы на индивидуальных листках бумаги. Не возникает никаких проблем с одновременным чтением различными студентами различных вопросов, а при наличии отдельных листов бумаги их ответы не перемешиваются друг с другом. См. рис. 3.11.



Рис. 3.11. Пример реентерабельного процесса

### 4. Структуры данных: стеки и деревья

Мы еще не специфицировали структуры данных, которые будут использоваться в качестве временной памяти для всех подпрограмм; чтобы решить эту проблему, давайте посмотрим на дерево вызовов подпрограмм.

стеки

затраты времени  
и пространство  
памяти

Для того чтобы внутренняя подпрограмма имела доступ к переменным, объявленным во внешнем блоке, и чтобы она могла корректно вернуть управление в вызывающую программу, мы должны в каждой точке дерева помнить все узлы по пути от данной точки назад к корню. По мере того как мы движемся по дереву, в конце пути добавляются или, наоборот, удаляются узлы; другой конец (в корне дерева) остается неизменным. Отсюда следует, что узлы в пути образуют стек; соответственно и области памяти для подпрограмм должны сохраняться на стеке. Этот процесс проиллюстрирован на рис. 3.12.

Из рис. 3.12 и из нашего обсуждения можно немедленно сделать вывод, что объем памяти, требуемый для реализации рекурсии (который, разумеется, определяется числом областей памяти в текущем использовании), прямо пропорционален высоте дерева рекурсий. Программисты, не изучившие механизм рекурсии достаточно глубоко, иногда ошибочно думают, что требования к пространству памяти определяются полным числом узлов на дереве. *Время* выполнения программы определяется тем, сколько раз выполняются подпрограммы, и, следовательно, полным числом узлов на дереве, что же касается требований к *пространству памяти*, то они определяются только объемом областей памяти по пути от

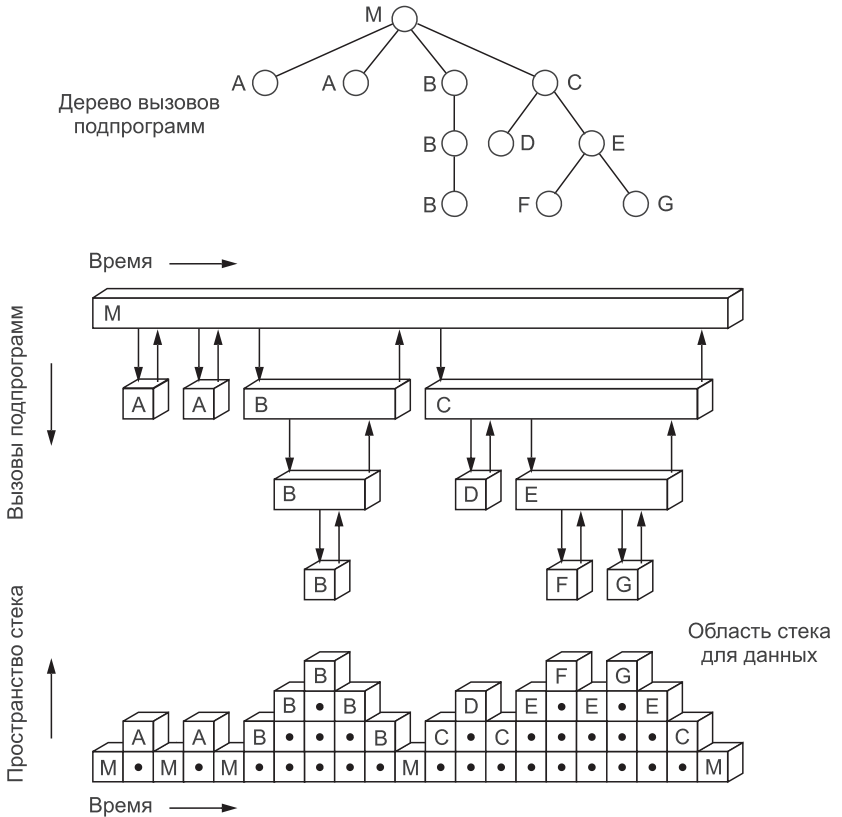


Рис. 3.12. Дерево вызовов подпрограмм и соответствующие ему кадры стека

одиноким узлом к корню. Поэтому требуемый объем памяти зависит от высоты дерева. Хорошо сбалансированное, кустистое дерево рекурсивно отражает рекурсивный процесс, который выполняет значительную работу без больших требований к дополнительной памяти.

### 3.3.3. Хвостовая рекурсия

отбрасывание  
элементов стека

Предположим, что самым последним действием процедуры является рекурсивный вызов самой себя. В случае стековой реализации рекурсии, как мы видели, при активизации рекурсивного вызова локальные переменные для процедуры будут проталкиваться в стек. Когда рекурсивный вызов завершается, эти локальные переменные будут вытолкнуты из стека и тем самым восстановят свои исходные значения. Однако это действие не имеет смысла, так как рекурсивный вызов был последним действием процедуры, так что процедура на этом завершается и только что восстановленные локальные переменные немедленно отбрасываются.

В тех случаях, когда последним действием процедуры является рекурсивный вызов самой себя, бессмысленно использовать стек, так как локальные переменные не требуется сохранять и восстанавливать. Все, что нам нужно — это назначить фиктивным параметрам вызова их новые значения и выполнить переход к началу процедуры. Мы подытожим этот принцип для дальнейших ссылок.

*Если последним выполняемым предложением процедуры является рекурсивный вызов самой себя, этот вызов может быть устранен путем назначения параметрам вызова значений, заданных в рекурсивном вызове, и затем повторения всей процедуры.*

Описанный процесс проиллюстрирован на рис. 3.13. В части (а) рисунка изображены области памяти, используемые вызывающей программой М и несколькими копиями рекурсивной процедуры Р, каждая из которых активизируется предыдущей копией. Стрелки показывают поток передачи управления от одного вызова подпрограммы к другому, а блоками обозначены области временной памяти, поддерживаемые системой. Поскольку каждый вызов процедурой Р самой себя есть ее последнее действие, то после возврата из этого вызова нет необходимости поддерживать временную память. Области временной памяти уменьшенного объема показаны на рис. 3.13, (b). Наконец, рис. 3.13, (c) соответствует повторяющимся итеративным вызовам процедуры Р; блоки памяти расположены на одном уровне рисунка.

хвостовая  
рекурсия

Описанный специальный случай, когда рекурсивный вызов является последним выполняемым предложением процедуры, особенно важен по той причине, что такая ситуация возникает довольно часто. Она носит название **хвостовой рекурсии**. Следует особо отметить, что для хвостовой рекурсии важно, чтобы последнее выполняемое предложение было рекурсивным вызовом, однако оно не обязательно должно быть последним предложением процедуры. Хвостовая рекурсия может, например, иметь место внутри одного оператора выбора предложения **case** или в предложении **if**, где за этими предложениями следуют другие строки программы.

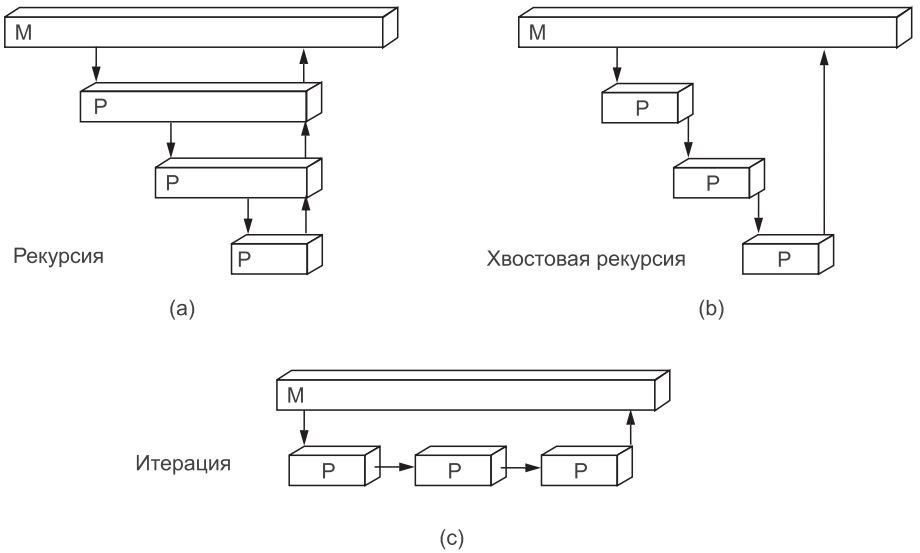


Рис. 3.13. Хвостовая рекурсия

время вычисления  
и память

Для большинства компиляторов на время выполнения программы мало повлияет сохранение в программе хвостовой рекурсии или ее удаление. Если, однако, важна экономия памяти, то хвостовую рекурсию лучше удалить. Изменив, если нужно, условие завершения, обычно бывает возможно повторять процедуру с помощью предложений **repeat** или **while**.

Рассмотрим, например, алгоритм разбиения, использованный при решении задачи о Башнях Ханоя. После удаления хвостовой рекурсии процедура **Move** исходной рекурсивной программы примет такой вид:

программа Hanoi  
без хвостовой  
рекурсии

```
procedure Move(count: disk; start, finish, temp: tower); { Переместить }
{ Pre: Диск count является диском, который можно переместить.
  Post: Перемещает count дисков с башни start на башню finish,
        используя башню temp для временного хранения. }
var
  swap: tower; { временная память для перемещения
                дисков с одной башни на другую }
begin
  while count > 0 do
  begin
    Move(count - 1, start, temp, finish);
    writeln('Перемещаем диск ', count, ' с башни ', start:2);
    writeln('на башню ', finish: 2);
    count := count - 1;
    swap := start;
    start := temp;
    temp := swap
  end
end; { процедура Move }
```

Нам было бы очень трудно придумать такой вариант процедуры, когда мы начинали решать задачу Башен Ханоя, но теперь, после всех проведенных рассуждений и анализа, он представляется вполне естественным. Считайте, что две башни, *start* и *temp*, принадлежат одному классу: мы хотим использовать их для промежуточного хранения по мере того, как мы постепенно перемещаем все диски на башню *finish*. Для перемещения стопки *count* дисков на *finish*, мы должны переместить все диски, кроме нижнего, на вторую башню нашего класса, *temp*, затем перенести нижний диск на *finish*, а далее, чередуя *start* и *temp*, повторять перемещение всех дисков, кроме нижнего, между *start* и *temp*, и в конце каждого прохода переносить новый нижний диск на башню *finish*.

### 3.3.4. Когда не следует использовать рекурсию

#### 1. Факториалы

Рассмотрим две функции для вычисления факториалов. Рекурсивную мы уже видели:

```
function Factorial(n: integer): integer; { Факториалы; рекурсивный вариант }
{ Pre:  n есть неотрицательное целое число.
  Post: Значение функции представляет собой факториал от n. }
begin                                     { функция Factorial }
  if n = 0 then Factorial := 1
    else Factorial := n * Factorial(n - 1)
end;                                     { функция Factorial }
```

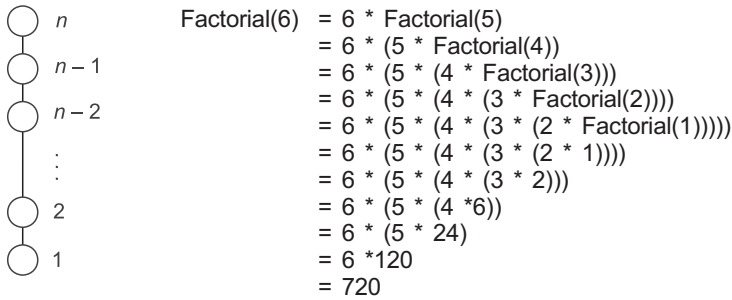
А вот почти такой же простой итеративный вариант:

```
function Factorial(n: integer): integer; { Факториалы; итеративный вариант }
{ Pre:  n есть неотрицательное целое число.
  Post: Значение функции представляет собой факториал от n. }
var
  count,
  product: integer;
begin                                     { функция Factorial }
  product := 1;
  for count := 2 to n do
    product := product * count;
  Factorial := product
end;                                     { функция Factorial }
```

Какая из этих двух программ использует меньше памяти? На первый взгляд может показаться, что рекурсивная, поскольку в ней нет локальных переменных, а в итеративной программе их две штуки. Но фактически (см. рис. 3.14) рекурсивная программа будет использовать стек, заполнив его числами

$n, n - 1, n - 2, \dots, 2$

которых будет всего  $n - 1$  штук. Эти числа являются параметрами рекурсивных вызовов процедуры; и программа, выходя из каждой рекурсии, будет перемножать эти числа в том же порядке, что и вторая, итеративная, программа. Ход выполнения для рекурсивной функции при  $n = 6$  выглядит следующим образом:



**Рис. 3.14.** Дерево рекурсии при вычислении факториалов

Таким образом, рекурсивная программа требует больше памяти, и времени она также затратит больше, поскольку она должна не только умножать все числа, но еще и сохранять, и извлекать их.

## 2. Числа Фибоначчи

Еще более затратным, чем в случае вычисления факториала, примером (его также можно встретить в некоторых учебниках в качестве рекомендуемой программы) является вычисление *чисел Фибоначчи*, которые определяются рекуррентным отношением

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2} \text{ для } n \geq 2.$$

Рекурсивная программа точно соответствует определению:

```
function Fibonacci (n: integer): integer; { Фибоначчи; рекурсивный вариант }
{ Pre: Параметр n есть неотрицательное целое число.
  Post: Функция возвращает n-е число Фибоначчи.}
begin
    if n <= 0 then          Fibonacci := 0
    else if n = 1 then      Fibonacci := 1
    else                    Fibonacci := Fibonacci(n - 1) + Fibonacci(n - 2)
end;                        { функция Fibonacci }
```

Программа кажется весьма привлекательной, поскольку она использует принцип разбиения (принцип «разделяй и властвуй»): ответ получается путем вычисления двух более простых случаев. Как мы, однако, увидим, это пример не столько «разделяй и властвуй», сколько «разделяй и усложняй».

Для оценки этого алгоритма давайте рассмотрим в качестве примера вычисление  $F_7$ ; дерево рекурсии для этого случая приведено на рис. 3.15. Сначала функция должна получить  $F_6$  и  $F_5$ . Для получения  $F_6$  функции требуется  $F_5$  и  $F_4$  и т. д. Однако после того, как  $F_5$  будет вычислено для нахождения  $F_6$ , его значение будет потеряно и недоступно, когда оно снова потребуется для вычисления  $F_7$ . Таким образом, как это видно из дерева рекурсии, рекурсивная программа излишне повторяет одни и те же вычисления снова и снова. Дальнейший анализ можно выполнить в качестве упражнения. Он показывает, что время, требуемое рекурсивной функции для вычисления  $F_n$ , растет экспоненциально с увеличением  $n$ .

Как и в случае вычисления факториалов, мы можем представить простую итеративную программу, заметив, что мы можем начать с 0 и иметь



Очевидно, что время, требуемое итеративной функции, возрастает линейно (т. е. прямо пропорционально) с ростом  $n$ , так что ее эффективность по сравнению с рекурсивным вариантом, где время возрастает экспоненциально, существенно выше.

### 3. Сравнение рекурсии и итерации

В чем фундаментальное отличие этого последнего примера от разумного использования рекурсии? Для ответа на этот вопрос мы должны вернуться к рассмотрению деревьев рекурсии. Мы уже видели, что исследование дерева рекурсии предоставляет много полезной информации, которая помогает нам решить, следует ли в конкретном случае использовать рекурсию.

цепочка

Если функция или процедура выполняет только один рекурсивный вызов самой себя, тогда ее дерево рекурсии принимает очень простую форму цепочки, в которой каждый узел имеет единственный нижележащий узел. Этот нижележащий, дочерний узел соответствует единственному выполняемому рекурсивному вызову. Такое простое дерево рекурсии легко преобразовать в программу. Для функции факториала мы имеем просто список запросов на вычисление факториалов от  $(n-1)!$  до  $1!$ . Просматривая дерево рекурсии не сверху вниз, а, наоборот, от дна к началу, мы немедленно из рекурсивной программы получаем итеративную. В тех случаях, когда дерево вырождается в цепочку, преобразование рекурсии в итерацию часто осуществляется очень просто, и такое преобразование, весьма вероятно, сэкономит и время, и память.

Заметьте, что если процедура выполняет единственный рекурсивный вызов самой себя, это совсем не то же самое, что процедура с единственным рекурсивным вызовом в некоторой своей точке, так как эта точка может находиться внутри цикла. Возможно также иметь две точки, в которых осуществляются рекурсивные вызовы (например, в ветвях **then** и **else** предположения **if**), хотя фактически в таком случае реализуется лишь один рекурсивный вызов.

дублирующие  
задачи

Дерево рекурсии для вычисления чисел Фибоначчи не является цепочкой, однако содержит большое количество узлов, выполняющих дублирующие задачи. Когда рекурсивная программа запускается, она устанавливает стек, который будет использоваться при просмотре дерева, но если результаты, помещаемые в стек, отбрасываются, а не сохраняются в какой-либо иной структуре данных для будущего использования, это может привести к значительному объему дублирующих вычислений, как это и происходит при рекурсивных вычислениях чисел Фибоначчи.

замена структуры  
данных

В таких случаях более предпочтительно вместо стека использовать другую структуру, такую, которая допускает доступ ко всем своим данным, а не только к верхнему. Для чисел Фибоначчи нам потребовались лишь две дополнительные переменные для временного хранения данных, требуемых для вычисления текущего числа.

отказ от рекурсии

Наконец, предусмотрев в программе явно объявленный стек, мы можем любую рекурсивную программу преобразовать в нерекурсивную форму. Однако такой нерекурсивный вариант почти всегда оказывается более сложным и трудным для понимания, чем рекурсивный. Единствен-



ным основанием отказа от рекурсии может быть использование вами языка программирования, не поддерживающего рекурсии, однако такая ситуация встречается все реже и реже.

### 3.3.5. Рекомендации и заключения

Итак, для принятия решения, стоит ли конкретный алгоритм писать в рекурсивной или нерекурсивной формах, большую помощь может оказать дерево рекурсии. Если оно имеет простую форму, итеративный вариант может оказаться более предпочтительным. Если задача приводит к дублированию вычислений, тогда вместо стека лучше использовать другие структуры данных, и необходимость в рекурсии может отпасть. Если же дерево рекурсии оказывается ветвистым, и в нем мало дублирующих вычислений, тогда рекурсия, скорее всего, будет наилучшим методом.

Стек, используемый для реализации рекурсии, можно рассматривать как список отложенных действий для программы. Если этот список можно легко построить заранее, тогда, возможно, итеративный метод даст лучшие результаты; если нет, лучше использовать рекурсию. Рекурсия напоминает нисходящий подход к решению задачи; в этом случае задача разделяется на части, или выделяется одно ключевое действие, а остальные действия откладываются. Итерация больше напоминает восходящий подход; она начинается с известного и затем строит искомое решение шаг за шагом.

Во всех случаях рекурсию можно заменить итерацией и стеками. Справедливо и обратное (см. ссылки на доказательства); любая (итеративная) программа, использующая стек, может быть заменена рекурсивной программой без стека. Поэтому старательный программист должен не только выяснить, не лучше ли отказаться от рекурсии, но и наоборот, в тех случаях, когда в программе имеются стеки, подумать, не приведет ли использование рекурсии к более естественной и понятной программе, что может повлечь за собой дальнейшие улучшения в подходе и результатах.

### Упражнения 3.3

- E1.** Для рекурсивного вычисления  $F_n$  определите точно, сколько раз будет вычисляться каждое меньшее число Фибоначчи. Отсюда оцените по порядку величин потребности рекурсивной функции во времени и памяти. [Вы можете выполнить эту оценку либо путем определения и решения рекуррентного отношения (нисходящий подход), либо найдя ответы для простых случаев и доказав их общность путем математической индукции (восходящий подход).]
- E2. Наибольший общий делитель** (НОД, или GCD от greatest common divisor) двух положительных чисел представляет собой наибольшее целое, на которое делятся оба эти числа. Например, НОД чисел 8 и 12 есть 4, НОД чисел 9 и 18 есть 9, а НОД чисел 16 и 25 есть 1. **(а)** Напишите рекурсивную функцию  $\text{GCD}(x, y: \text{integer}): \text{integer}$ , которая реализует **алгоритм деления**: если  $y = 0$ , тогда GCD от  $x$  и  $y$  есть  $x$ ; в против-

нисходящее  
проектирование

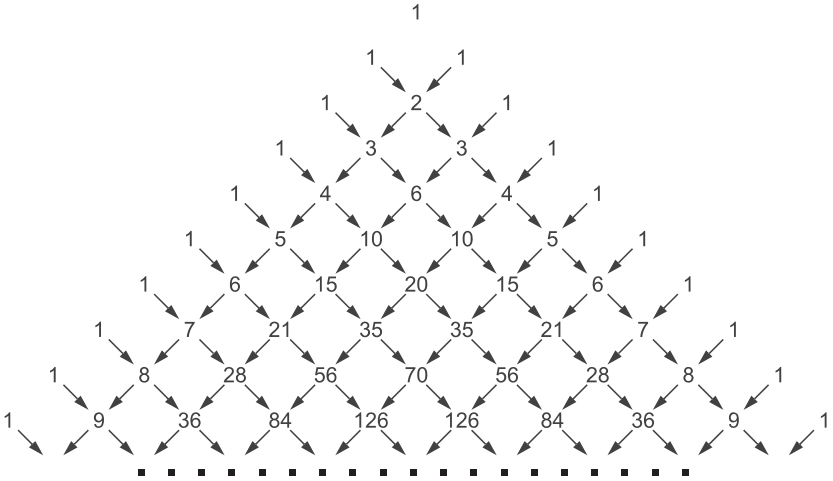
стеки  
или рекурсия

ном случае  $\text{GCD}$  от  $x$  и  $y$  есть то же самое, что и  $\text{GCD}$  от  $y$  и  $x \bmod y$ .

(b) Перепишите функцию в итеративной форме.

**Е3.** Биномиальные коэффициенты можно определить с помощью следующего рекуррентного отношения, являющегося основой *треугольника Паскаля*. Вершина треугольника Паскаля показана на рис. 3.16.

$$\begin{aligned} C(n, 0) &= 1 \quad \text{и} \quad C(n, n) = 1 && \text{для } n \geq 0, \\ C(n, k) &= C(n-1, k) + C(n-1, k-1) && \text{для } n > k > 0. \end{aligned}$$



**Рис. 3.16.** Вершина треугольника Паскаля для определения биномиальных коэффициентов

- (a) Напишите рекурсивную функцию, генерирующую коэффициенты  $C(n, k)$  по приведенной выше формуле.
- (b) Нарисуйте дерево рекурсии для вычисления  $C(6, 4)$ .
- (c) Используя квадратный массив, напишите нерекурсивную программу для генерирования треугольника Паскаля в нижней (и одновременно левой) половине массива.
- (d) Напишите нерекурсивную программу, не использующую ни массив, ни стек для вычисления  $C(n, k)$  для произвольного  $n \geq k \geq 0$ .
- (e) Оцените требования времени и памяти для каждого из алгоритмов, разработанных в упражнениях (a), (c) и (d).

**Е4.** *Функция Аккермана*, задаваемая, как показана ниже, является стандартным методом определения качества реализации рекурсии на компьютере.

$$\begin{aligned} A(0, n) &= n + 1 && \text{для } n \geq 0, \\ A(m, 0) &= A(m-1, 1) && \text{для } m > 0, \\ A(m, n) &= A(m-1, A(m, n-1)) && \text{для } m > 0 \text{ и } n > 0. \end{aligned}$$

- (a) Напишите рекурсивную функцию для вычисления функции Аккермана.

(b) Вычислите следующие значения:

$A(0, 0)$     $A(0, 9)$     $A(1, 8)$     $A(2, 2)$     $A(2, 0)$   
 $A(2, 3)$     $A(3, 2)$     $A(4, 2)$     $A(4, 3)$     $A(4, 0)$

(c) Напишите нерекурсивную функцию для вычисления функции Аккермана.

## Подсказки и ловушки

1. Практикуйте сокрытие информации. Используйте процедуры и функции для доступа к своим структурам данных и помещайте их в пакеты, отдельные от вашей прикладной программы.
2. Откладывайте разработку деталей реализации своих структур данных, насколько это возможно.
3. Стеки относятся к простейшим структурам данных; используйте стеки, когда это только возможно.
4. Избегайте хитроумных способов хранения своих данных; хитрости обычно не удается обобщить на другие случаи.
5. Не забывайте инициализировать свои структуры данных.
6. При разработке алгоритмов всегда обращайтесь внимание на граничные случаи и аккуратно их обрабатывайте. Трассируйте свои алгоритмы для определения поведения программы в граничных случаях, особенно, когда структуры данных оказываются пустыми или заполненными.
7. На начальных этапах разработки алгоритмов следует широко использовать рекурсии. Это особенно относится к таким задачам, в которых главный шаг к решению представляет собой преобразование задачи к одному или нескольким более простым действиям.
8. Рассмотрите несколько простых примеров с целью выяснения, можно ли использовать рекурсию и как она будет работать.
9. Старайтесь сформулировать метод, который будет работать в более общем случае. Спросите себя: «Каким образом можно разделить задачу на части?» или «Как следует выполнить ключевое действие в середине задачи?».
10. Выясните, нельзя ли оставшуюся часть задачи решить тем же или схожим образом, и модифицируйте при необходимости свой метод, чтобы он получил достаточную степень общности.
11. Найдите правило остановки, которое будет показывать, что вся задача или некоторая часть ее решена.
12. Обратите особое внимание на то, чтобы ваш алгоритм всегда завершался и правильно обрабатывал тривиальные случаи.
13. Ключевым средством анализа рекурсивных алгоритмов является дерево рекурсии. Нарисуйте дерево рекурсии для одного-двух простых примеров, применимых к вашей задаче.
14. Всегда следует анализировать дерево рекурсии, чтобы выяснить, не выполняется ли по ходу рекурсии бесполезная работа, или предостав-

ляет ли дерево эффективное разделение всей работы на отдельные части.

15. Рекурсивные процедуры и итеративные процедуры, использующие стек, могут решать в точности одинаковые задачи. Тщательно проанализируйте, какой из этих методов — рекурсия или итерация — позволит написать более простую программу и глубже проникнуть в существо задачи.
16. Если экономия памяти важна, ей может способствовать отказ от хвостовой рекурсии.
17. Рекурсию всегда можно преобразовать в итерацию, но обычно в этом случае структура программы становится существенно запутанной. Эту запутанность можно допустить лишь в тех случаях, когда язык программирования делает ее неизбежной, и тогда следует особенно тщательно документировать программу.

## Обзорные вопросы

- 3.1
  1. Какие операции можно выполнять над стеком?
  2. Что представляют собой кадры стека для подпрограмм? Что они показывают?
  3. Каковы преимущества оформления операций над структурами данных в виде процедур и функций?
- 3.2
  4. Определите термин *метод разбиения*.
- 3.3
  5. Назовите два различных способа реализации рекурсии.
  6. Что представляет собой *реентерабельная* программа?
  7. Как связано время выполнения рекурсивной программы с деревом рекурсии?
  8. Как связаны требования к объемам памяти для рекурсивной программы с деревом рекурсии?
  9. Что представляет собой *хвостовая рекурсия*?
  10. Опишите взаимосвязь между формой дерева рекурсии и эффективностью соответствующего ему алгоритма.
  11. Каковы основные этапы проектирования рекурсивного алгоритма?
  12. Что такое *параллельные вычисления*?
  13. Какого рода важная информация требуется компьютерной системе при реализации рекурсивных вызовов процедуры?
  14. На экономию чего больше влияет удаление хвостовой рекурсии, времени или пространства памяти?

## Литература для дальнейшего изучения

Для множества тем, касающихся структур данных, в частности, стеков, наилучшим источником дополнительной информации, исторических заметок и математического анализа является указанная ниже серия книг,

которую можно рассматривать почти как энциклопедию по ряду аспектов вычислительных наук:

энциклопедия:  
Д. Е. Кнут

Donald E. Knuth *The Art of Computer Programming*, published by Addison-Wesley, Reading, Mass.

К настоящему времени появились три тома:

1. *Fundamental Algorithms*, second edition, 1973, 634 pages.
2. *Seminumerical Algorithms*, second edition, 1980, 700 pages.
3. *Sorting and Searching*, 1973, 722 pages.

Имеется русский перевод: Кнут Д., Искусство программирования. — М. и др.: Вильямс.

T.1: Основные алгоритмы. — 3 изд. 2000. — 712 с.

T.2: Получисленные алгоритмы. — 3 изд. 2000. — 828 с.

T.3: Сортировка и поиск. — 2 изд. 2000. — 822 с.

В дальнейших главах будут часто встречаться ссылки на эту серию книг, и для краткости мы будем просто указывать имя автора ([Knuth]) вместе с томом и номером страницы. Алгоритмы в этой книге даются как в виде описаний, так и на языке ассемблера; второй вариант позволяет автору определять число выполняемых операций и, тем самым, сравнивать различные алгоритмы.

Две перечисленные ниже книги содержат детальное введение в рекурсию со многими примерами; их можно считать превосходным дополнением к настоящей книге.

Eric S. Roberts, *Thinking Recursively*, John Wiley & Sons, New York, 1986, 192 pages.

Игра «Башни Ханоя» получила широкую известность и приводится во многих учебниках. Обзор относящейся к ней литературы можно найти в статье

D. Wood, «*The Towers of Brahma and Hanoi revisited*», *Journal of Recreational Math* 14 (1981–92), 17–24.

Доказательство того, что от стеков можно избавиться путем использования рекурсии, содержится в статье

S. Brown, D. Gries and T. Szymanski, «Program schemes with pushdown stores», *SIAM Journal on Computing* 1 (1972), 242–268.

# Примеры рекурсии

Мы уже увидели полезность рекурсии и в качестве средства решения задач, и как программистский метод. В настоящей главе мы изучим еще несколько примеров и продемонстрируем многообразие задач, к которым можно плодотворно применить метод рекурсии. Некоторые из этих примеров просты, другие, наоборот, весьма изощренны.

## 4.1. Алгоритмы с отходом: откладывание работы

Рассмотрим задачу размещения на шахматной доске восьми ферзей так, чтобы ни один ферзь не находился под ударом другого. Вспомним, что согласно правилам игры в шахматы ферзь может снять другую фигуру, если та расположена в том же ряду, или в том же столбце, или на той же диагонали (в обоих направлениях), что и сам ферзь. Шахматная доска содержит восемь рядов и восемь столбцов.

Решение этой задачи отнюдь не очевидно, и ее полное решение не поддалось даже усилиям великого К. Ф. Гаусса (C. F. Gauss), который занимался этой задачей в 1850 г. Задача относится к числу головоломок, которые на вид не приспособлены для аналитического решения, а требуют либо удачи в применении метода проб и ошибок, либо изнурительных вычислений. Чтобы убедить вас, что решения этой задачи в действительности существуют, два из них приведены на рис. 4.1.

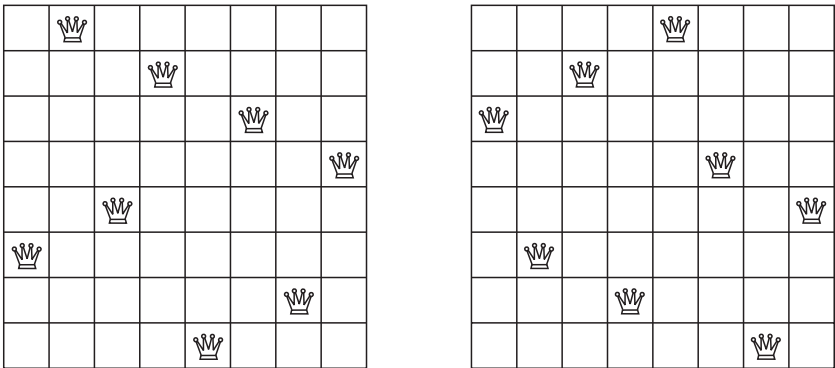


Рис. 4.1. Две конфигурации, показывающие восемь неугрожающих друг друга ферзей

### 4.1.1. Решение задачи о восьми ферзях

Некто, пытающийся решить задачу о восьми ферзях, вскоре, скорее всего, оставит попытки найти все (или хотя бы одно) решения путем умственных рассуждений и начнет просто расставлять фигуры на доске, возможно, случайным образом или по какому-то логическому плану, но всегда следя за тем, чтобы новый ферзь, помещаемый на доску, не атаковал бы ни одного уже находящегося на доске. Если этот некто достаточно удачлив, чтобы расположить на доске восемь ферзей таким методом, тогда это, конечно, решение задачи; если же нет, тогда один или несколько ферзей должны быть сняты с доски и помещены на какое-либо иное место, чтобы продолжить поиск решения. Чтобы начать формулировать задачу, обрисуем этот метод в алгоритмической форме. Мы обозначаем число ферзей на доске через  $n$ ; первоначально  $n = 0$ . Ключевое действие выглядит следующим образом:

```

procedure AddQueen;                                { Добавить ферзя }
for для каждой незащищенной позиции  $p$  на доске do
  begin
    Поместить ферзя в позицию  $p$ ;
     $n := n + 1$ ;
    if  $n = 8$  then
      Вывести на экран конфигурацию
    else
      AddQueen;
    Удалить ферзя с позиции  $p$ ;
     $n := n - 1$ 
  end.
  
```

эскиз

Этот набросок иллюстрирует использование понятия рекурсии в смысле «Перейдите к следующему шагу и повторите поставленную задачу». Помещение ферзя на позицию  $p$  является пробным действием: мы оставляем его там только в том случае, если мы можем, продолжая, добавлять ферзи, пока их не будет восемь штук. Следовательно, после возврата из внутреннего вызова мы должны удалить ферзя с позиции  $p$ , поскольку все возможности этой позиции были исследованы и отброшены.

### 4.1.2. Пример: четыре ферзя

Посмотрим, как этот алгоритм работает со значительно более простой задачей, именно, задачей размещения четырех ферзей на доске  $4 \times 4$ , как это показано на рис. 4.2.

Нам потребуется поместить по одному ферзю на каждый ряд доски. Попробуем сначала поместить ферзя в крайнюю левую позицию некоторого ряда доски. Такой ход для первого ряда показан на рис. 4.2, (а). Вопросительные знаки обозначают другие возможные ходы, которые мы еще не опробовали. Перед тем, как перейти к исследованию этих ходов, мы перемещаемся во второй ряд и пробуем поместить на него ферзя. Первые два столбца защищаются ферзем в ряде 1, как это показано на рисунке крестиками. Столбцы 3 и 4 свободны, и мы сначала помещаем

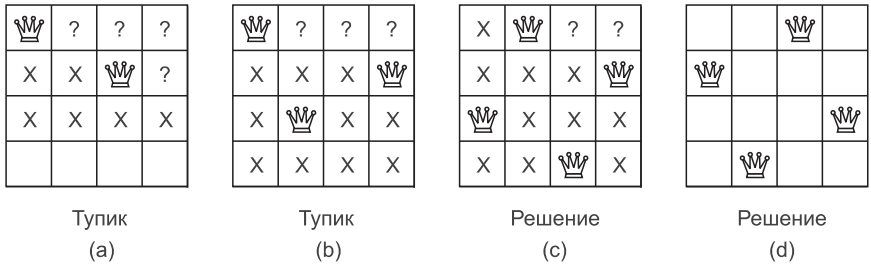


Рис. 4.2. Решение задачи о четырех ферзях

ферзя на столбец 3 и помечаем столбец 4 знаком вопроса. Далее мы перемещаемся в третий ряд, но обнаруживаем, что все четыре поля защищаются одним из ферзей в двух первых рядах. Таким образом, мы достигли тупика.

Достигнув тупика, мы должны *отойти* назад к последнему сделанному нами ходу и попробовать другую возможность. Эта позиция показана на рис. 4.2, (b), где видно, что ферзь в ряду 1 не изменил своего положения, но ферзь в ряду 2 переместился во вторую возможную позицию (а предыдущая позиция этого ферзя теперь перечеркнута, как не подходящая для использования). Теперь мы обнаруживаем, что только столбец 2 является допустимым для размещения ферзя в ряду 3, но все четыре позиции ряда 4 оказываются защищенными. Таким образом, мы опять достигли точки, в которой не может быть добавлен ни один ферзь, и, следовательно, опять должны отходить назад.

В этой точке у нас нет других ходов для ряда 2, поэтому нам придется вернуться к ряду 1 и передвинуть ферзя на следующую возможную позицию, столбец 2. Эта позиция показан на рис. 4.2, (c). Теперь выясняется, что в ряду 2 незащищенным является только столбец 4, так что ферзя мы помещаем туда. В ряду 3 такая возможность есть только в столбце 1, а в ряду 4 — в столбце 3. Это размещение ведет, однако, к решению проблемы четырех неатакующих ферзей на доске  $4 \times 4$  клетки.

Если мы хотим найти *все* возможные решения, мы можем продолжить попытки размещения тем же способом, отходя назад к последнему сделанному нами ходу и перемещая ферзя на следующую возможную позицию. На диаграмме (c) у нас не было других возможных ходов в рядах 4, 3 и 2, поэтому мы возвращаемся к ряду 1 и помещаем ферзя в столбец 3. Этот ход ведет к решению, показанному на рис. 4.2, (d).

Наконец, мы должны испытать возможность помещения ферзя в столбец 4 ряда 1, но, как и в позиции рис. 4.2, (a), в этом случае решений не обнаруживается. Фактически конфигурации с ферзем в столбцах 3 или 4 ряда 1 являются зеркальными отражениями конфигураций с ферзем в столбцах 2 или 1. Если выполнить левостороннее отражение доски, показанной на рис. 4.2, (c), вы получите доску с позицией (d), а конфигурации с ферзем в столбце 4 ряда 1 являются отражениями позиций, показанных на рис. 4.2, (a) и (b).



### 4.1.3. Алгоритм с отходом

Продemonстрированная нами процедура типична для широкого класса алгоритмов, называемых *алгоритмами с отходом*, или *с возвратом*, в которых делается попытка найти решение задачи путем конструирования частичного решения, разумеется, такого, которое удовлетворяет условиям поставленной задачи. После этого алгоритм пытается расширить частичное решение, но если возникает несовместимость с условиями задачи, алгоритм *отходит назад*, удаляя последнюю сконструированную часть решения и переходя к следующей возможности.

Отход оказывается полезен в ситуациях, когда поначалу кажется, что возможностей очень много, но лишь немногие из них проходят проверку на совместимость с условиями задачи. Например, в задачах составления расписания спортивных игр будет легко назначить первые несколько матчей, но по мере назначения следующих матчей ограничения драматически снижают число возможных вариантов. Или возьмем задачу разработки компилятора. В некоторых языках (хотя к ним не относится Pascal) невозможно определить значение предложения до тех пор, пока оно не будет прочитано почти до самого конца. Рассмотрим, например, пару предложений на языке Fortran

```
DO 17 K = 1, 6  
DO 17 K = 1. 6
```

синтаксический  
анализ

Оба предложения правильны; первое начинает цикл, а второе присваивает значение 1.6 переменной DO17K. В таких случаях, когда смысл предложения не может быть понят немедленно, отход является полезным методом *синтаксического анализа* (т. е. расщепления на части с целью расшифровки) текста программы.

### 4.1.4. Детализация: выбор структур данных

Для определения деталей нашего алгоритма решения задачи о восьми ферзях мы должны сначала решить, каким образом мы будем определять незащищенные позиции на каждом шаге, и как мы будем обрабатывать в цикле эти незащищенные позиции. Постановка этого вопроса приводит нас к необходимости принять решение о представлении данных в программе.

квадратный  
булев массив

Игрок, пытающийся найти решение задачи о восьми ферзях с помощью настоящей шахматной доски, скорее всего будет помещать ферзей на поля доски по одной фигуре за раз. Мы можем делать то же самое с помощью компьютера, введя в программу массив булевых значений размером  $8 \times 8$  и определив, что наличие ферзя на поле соответствует значению элемента массива true, а отсутствие — false. Для определения защищенности поля игрок просмотрит всю доску, отмечая поля, защищенные наличными ферзями, и мы можем сделать то же самое, однако такой просмотр потребует значительной работы.

Игрок, ищущий решение задачи на бумаге или на школьной доске, быстро обнаружит, что после того, как ферзь поставлен на доску, можно

заметно сократить время следующего этапа решения, если все поля, которые защищаются новым ферзем, отметить галочками. В этом случае для определения незащищенной позиции для следующего ферзя нужно просмотреть только поля без галочек. И здесь мы можем сделать то же самое программно, задавая каждому элементу нашего массива значение true, если поле свободно, и false, если поле защищено.

Однако здесь возникает трудность с удалением ферзя. Мы не должны однозначно изменять состояние позиции, которую этот ферзь защищал, так как эту позицию мог защищать еще и другой ферзь. Эту проблему можно решить, установив для элементов нашего массива тип integer вместо Boolean, и определив, что значение элемента массива показывает число ферзей, защищающих позицию. Тогда, добавляя ферзя, мы увеличиваем отсчет на 1 для каждой строки, столбца и диагонали, проходящих через позицию с новым ферзем, а удаляя ферзя, уменьшаем соответствующие отсчеты на 1. Позиция считается незащищенной, если и только если отсчет для нее равен 0.

Несмотря на очевидные преимущества по сравнению с первым вариантом, этот метод все еще требует значительной работы по поиску незащищенных позиций, а также некоторых вычислений для изменения отсчетов на каждом ходе. Алгоритм будет добавлять и удалять ферзи очень много раз, так что эти вычисления и операции поиска могут быть весьма затратными. Однако игрок, работающий над решением, вскоре обнаружит еще одну возможность сократить объем выполняемой работы.

После помещения ферзя на первый ряд никому не придет в голову тратить время на поиск допустимой позиции для следующего ферзя в том же ряду, поскольку этот ряд полностью защищается первым ферзем. В любом ряду никогда не может быть более одного ферзя. В то же время нашей целью является помещение восьми ферзей на доску, а на ней только восемь рядов. (Эту ситуацию можно назвать *принципом голубятни* : если у вас имеются  $n$  голубей и  $n$  закутков для них, и в каждом закутке помещается только один голубь, то все закутки должны быть заполнены голубями.)

Итак, мы продолжаем, помещая ферзей на доску по ферзю на каждый ряд, начиная с первого ряда, и мы можем хранить размещение ферзей в одномерном массиве

**var col: array [1..8] of 1..8**

где значение col[i] дает нам столбец, содержащий ферзя на строке i. Чтобы избежать размещения двух ферзей в том же столбце или на той же диагонали, нам не надо просматривать весь массив  $8 \times 8$ ; достаточно только хранить информацию о том, свободен ли или занят каждый столбец, и то же самое про каждую диагональ. Для этого достаточно трех булевых массивов: colfree, upfree и downfree, причем диагонали, идущие от левой нижней позиции к правой верхней считаются направленными вверх (массив upfree), а диагонали, идущие от левой верхней позиции к правой нижней считаются направленными вниз (массив downfree).

квадратный  
массив  
целых чисел

принцип  
голубятни

массив  
положений  
ферзей



рекурсивная  
процедура

Рекурсивная процедура такова:

```

procedure AddQueen;                                { Добавить ферзя }
{ Pre: Ферзи были правильно размещены в рядах от 1 до queencount
  (при условии, что queencount > 0), и информация занесена
  в вышеописанные массивы.
Post: Все решения, начинающиеся с этой конфигурации, выведены
  на экран. Переменной queencount и всем массивам присвоены
  начальные значения.
Uses: Использует глобальную переменную queencount,
  глобальные массивы queencol, colfree, upfree и downfree
  и процедуру WriteBoard. }
var col: 1..boardsize;
      { столбец, на который делается попытка поместить ферзя }
begin                                              { процедура AddQueen }
  queencount := queencount + 1;
  for col := 1 to boardsize do
    if colfree[col] and upfree[queencount + col] and
      downfree[queencount - col] then begin
      queencol[queencount] := col;
      { поместим ферзя в позицию [queencount, col] }
      colfree[col] := false;
      upfree[queencount + col] := false;
      downfree[queencount - col] := false;
      if queencount = boardsize then                { условие завершения }
        WriteBoard
      else
        AddQueen;                                { продолжим рекурсивно }
      colfree[col] := true;                        { теперь отход удалением ферзя }
      upfree[queencount + col] := true;
      downfree[queencount - col] := true;
    end;                                          { обработка ферзя в столбце col }
  queencount := queencount - 1;
end;                                          { процедура AddQueen }

```

## 1. Локальные и глобальные переменные

Обратите внимание на то, что в программе Queen почти все переменные и массивы были объявлены в главной программе, в то время как в программе Nani раздела 3.2.4 переменные объявлялись в рекурсивной процедуре. Если переменные объявляются внутри процедуры, они являются локальными для этой процедуры и недоступны вне ее. В частности, переменные, объявленные в рекурсивной процедуре, локальны для единственного экземпляра этой процедуры, так что если процедура повторно вызывается рекурсивным образом, все ее переменные создаются заново и отличаются от переменных предыдущего экземпляра процедуры, а эти предыдущие переменные будут снова видны, когда произойдет возврат из вложенной процедуры. Копии переменных, установленных во внешнем вызове, недоступны процедуре на протяжении внутреннего рекурсивного вызова. В программе Queen нам нужно, чтобы информация о защищенных рядах, столбцах и диагоналях была доступна всем рекурсивным экземплярам процедуры, и с этой целью мы объявляем соответствующие массивы не в процедуре, а в главной программе.

Единственным назначением массива `queencol [ ]` является передача позиций ферзей процедуре `WriteBoard`. Информация, содержащаяся в этом массиве, также сохраняется в восьми локальных копиях переменной `col`, значение которой устанавливается в процессе рекурсивного вызова, но в каждый момент времени программе доступна только одна из этих локальных копий.

#### 4.1.5. Анализ алгоритма с отходом

Теперь мы можем оценить объем работы, выполняемой нашей программой. Если бы мы приняли для программы наивный подход, помещая на доску все восемь ферзей и отбрасывая негодные конфигурации, мы должны были бы исследовать столько конфигураций, сколько может быть вариантов выбора восьми позиций из шестидесяти четырех, что равно

$$\left( \frac{64}{8} \right) = 4426165368.$$

Соображение, что в каждом ряду может быть только один ферзь, сразу же уменьшает число вариантов до

$$8^8 = 16777216.$$

Это число все еще очень велико, но наша программа не будет анализировать все эти позиции. Она отбрасывает позиции, у которых столбцы или диагонали защищены. Требование, чтобы в каждом столбце был только один ферзь, уменьшает число вариантов до

$$8! = 40320.$$

уменьшение  
числа вариантов

Исследование такого числа вариантов вполне по силам для компьютера, причем фактическое количество вариантов, которые будет рассматривать программа, будет значительно меньше этого числа (см. проект P1), поскольку позиции с защищенными диагоналями в предыдущих рядах будут отбрасываться немедленно, без бесплодных попыток заполнить следующие ряды.

эффективность  
отхода

Поведение нашей программы подчеркивает эффективность отхода: позиции, уже признанные негодными, предотвращают дальнейшее исследование бесполезных путей.

Другим способом отображения роли отхода является рассмотрение дерева рекурсивных вызовов для процедуры `AddQueen`, часть которого изображена на рис. 4.3. Два решения, показанные на этом дереве — это те же решения, которые можно было увидеть на рис. 4.1. Формально каждый узел дерева может иметь до восьми дочерних узлов соответственно рекурсивным вызовам процедуры `AddQueen` для восьми возможных значений переменной `col`. Однако даже на уровнях, близких к корню, большинство этих ветвей оказываются запрещенными, а удаление одного узла на верхнем уровне сразу же удаляет множество его потомков. Отход является наиболее эффективным средством обрезания дерева рекурсии до поддающегося обработке размера.

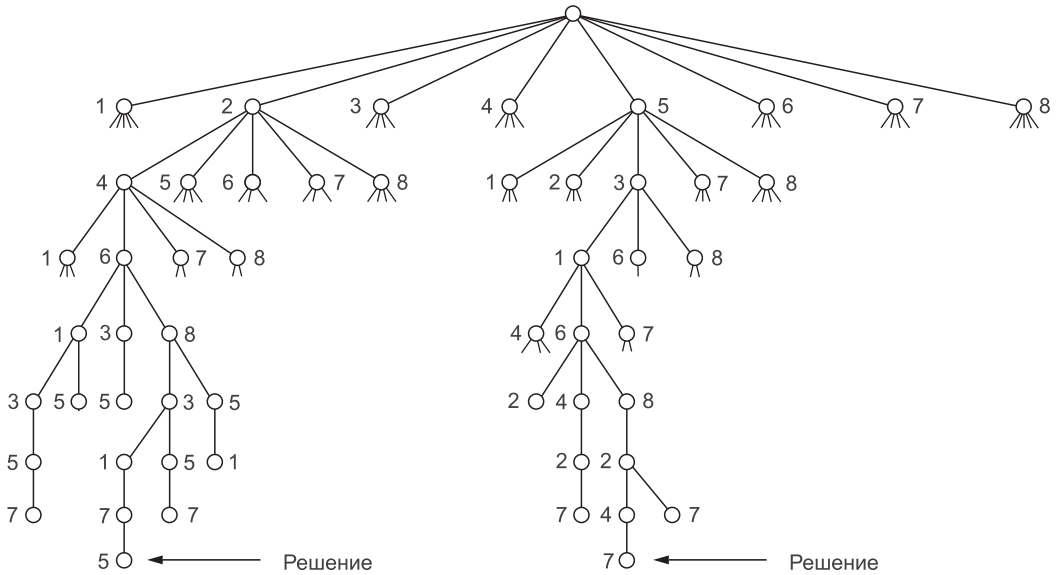
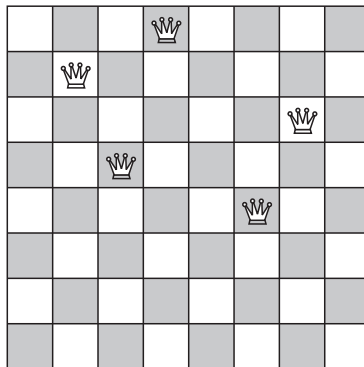


Рис. 4.3. Часть дерева рекурсии для задачи о восьми ферзях

### Упражнения 4.1

- Е1.** Какова максимальная глубина рекурсии в программе Queen?
- Е2.** Начав с приведенной ниже частичной конфигурации из пяти ферзей на доске, постройте дерево рекурсии для всех ситуаций, которые будет рассматривать программа Queen, пытаясь добавить оставшиеся три ферзя. Прекратите построение дерева в точке, где программа начнет отход с целью удаления одного из исходных пяти ферзей.



- Е3.** Выполняя отход вручную, найдите все решения задачи размещения пяти ферзей на доске размером  $5 \times 5$ . Вы можете использовать симметрию вокруг вертикали для первого ряда, рассматривая только варианты с ферзем в столбцах 1, 2 и 3 первого ряда.

## Программные проекты 4.1

- P1.** Запустите программу Queen на своем компьютере. Для выполнения вывода вам придется написать процедуру WriteBoard. Далее, включив в программу счетчик, инкрементируемый при каждом вызове процедуры AddQueen, найдите, сколько в точности исследуется позиций. [Заметьте, что метод решения задачи, когда все восемь ферзей помещались на доску перед анализом защищенных полей, эквивалентен восьми вызовам AddQueen.]
- P2.** Опишите прямоугольный лабиринт, указав его пути и стены в рамках массива. Напишите программу с алгоритмом отхода для нахождения пути сквозь лабиринт.
- P3.** Другая шахматная задача (которая, как считается, была решена Гауссом в возрасте четырех лет) заключается в нахождении последовательности ходов конем, при которой конь посетит каждую клетку шахматной доски в точности один раз. Ход коня заключается в прыжке на две клетки по горизонтали или вертикали и затем на одну клетку в перпендикулярном направлении. Первый ход реализуется путем задания переменной  $x$  значения 1 или 2, а переменной  $y$  значения  $3 - x$ , и изменения затем первой координаты на  $\pm x$ , а второй на  $\pm y$ , при этом результирующая позиция остается в пределах шахматной доски). Напишите программу с алгоритмом отхода, которая будет вводить начальную позицию и затем определять траекторию движения коня, начиная с заданной позиции так, чтобы конь побывал на каждой клетке один и только один раз. Если вы обнаружите, что программа выполняется слишком медленно, можно использовать неплохой метод упорядочения списка клеток доски, на которые конь может перейти из текущей позиции, так чтобы конь сначала пытался перейти на клетки с наименьшей доступностью, т. е. на клетки, из которых имеется минимальное число вариантов перехода на еще не посещенные клетки.

## 4.2. Древовидные программы: прогнозирование в играх

В играх, требующих интеллектуальных рассуждений, игрок, который может предугадать заранее, что может произойти через несколько ходов, имеет явное преимущество перед оппонентом, который ищет только ближайший выгодный ход. В этом разделе мы разработаем компьютерный алгоритм, позволяющий в играх разного рода смотреть на несколько ходов вперед. Этот алгоритм можно описать с помощью дерева; позже мы покажем, как можно запрограммировать такую структуру с помощью рекурсии.

### 4.2.1. Деревья игр

Мы можем изобразить последовательность возможных ходов с помощью *дерева игры*, корень которого описывает начальную ситуацию, а ветви, идущие от корня, соответствуют допустимым ходам первого игрока. На

следующем нижележащем уровне ветви соответствуют допустимым ходам второго игрока в каждой ситуации и т. д., причем ветви от узлов на нечетных уровнях описывают ходы первого игрока, а ветви от узлов на четных уровнях — ходы второго.

игра «Восемь»

Полное дерево игры для популярной игры «*Восемь*» показано на рис. 4.4. В этой игре первый игрок выбирает одно из чисел 1, 2 или 3. Второй игрок в свою очередь выбирает 1, 2 или 3, при этом нельзя выбирать число, выбранное первым игроком. Текущая сумма выбранных чисел запоминается, и если игрок доводит сумму точно до 8, он выигрывает. Если ход игрока приводит к сумме, большей восьми, выигрывает другой игрок. Ничьих в этой игре быть не может. На приведенном рисунке буква F обозначает выигрыш первого (First) игрока, буква S — второго (Second). Даже такая простая игра приводит к дереву значительного размера. Игры, действительно интересные, вроде шахмат или игры «Го», описываются деревьями гигантских размеров, так что об исследовании всех ветвей не может быть и речи, и программа, работающая обозримое время, может проверить только несколько уровней под текущим. Люди, играющие в такие игры, также не могут видеть все возможные ситуации от текущего положения до конца игры, однако они могут делать сознательный выбор ходов, поскольку с опытом игрок приобретает способность определять ходы, заметно лучшие других, даже если они и не гарантируют выигрыш.

оценочная  
функция

Для любой интересной игры, в которую мы хотим сыграть с помощью компьютера, нам нужна некоторая *оценочная функция*, которая бу-

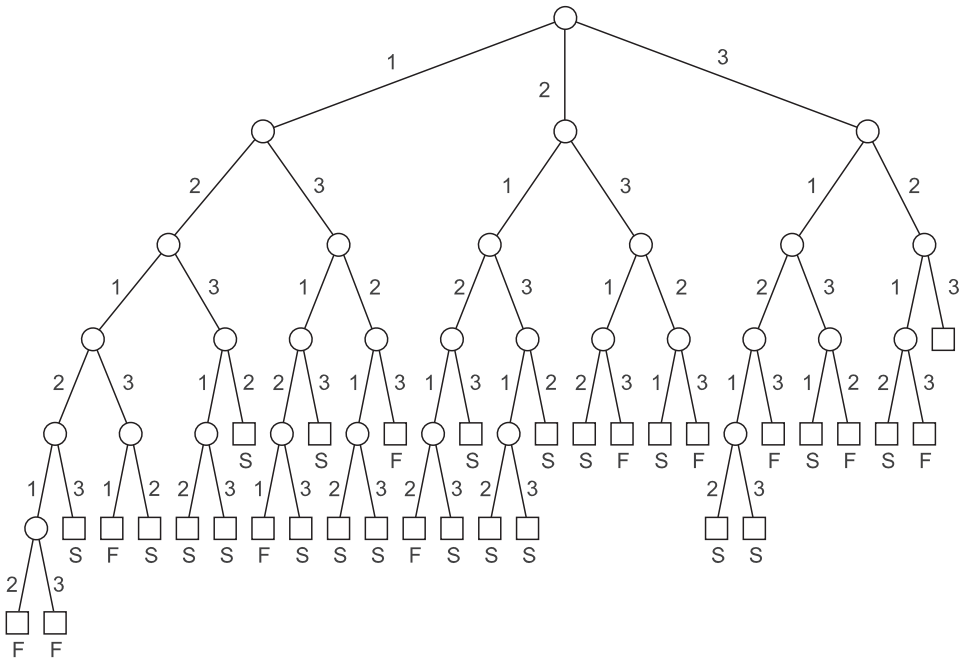


Рис. 4.4. Дерево для игры «Восемь»



дет анализировать текущую ситуацию и возвращать число, оценивающее выгоду этой ситуации. Пусть для определенности большие числа отражают ситуацию, благоприятную для первого игрока, в то время как маленькие числа (или более отрицательные) показывают преимущество второго игрока.

### 4.2.2. Метод минимакса

На рис. 4.5 изображена часть дерева для некоторой фиктивной игры. Поскольку мы занимаемся прогнозированием, нам нужна оценочная функция только для листьев дерева (т. е. позиций, от которых мы уже не будем заниматься дальнейшим прогнозом), и, исходя из полученной информации, мы хотим выбрать ход. Нарисуем листья дерева в виде квадратов, а остальные узлы в виде кружков. Ясно, что оценочные значения нам нужны только для квадратов, как это и показано на рис. 4.5.

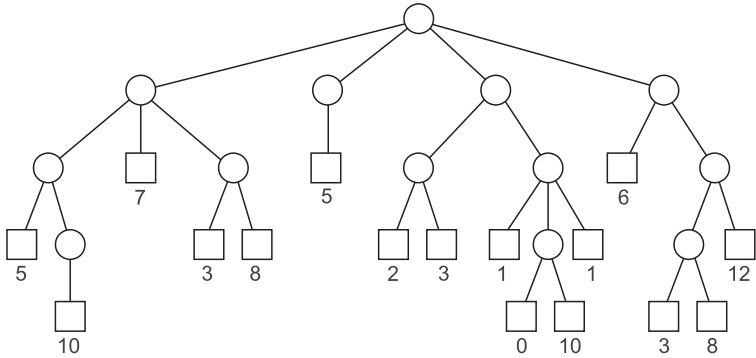


Рис. 4.5. Дерево игры со значениями, присвоенными листьям

Ход, который мы в конце концов выберем, представляет собой одну из ветвей, исходящих непосредственно из корня, самого верхнего уровня дерева. Мы рассматриваем результат оценочной функции с точки зрения этого игрока, а этот игрок очевидно выберет максимально возможное значение. На следующем уровне второй игрок выберет минимально возможное значение и т. д.

Поднимаясь вверх от дна дерева, мы назначаем определенные значения всем узлам. Давайте выполним частичную трассировку дерева, изображенного на рис. 4.5, начав с левой нижней стороны дерева. Первый непомеченный узел — это кружок над квадратиком с числом 10. Поскольку в этой точке нет выбора ходов, кружок также должен иметь значение 10. Его родительский узел имеет два дочерних узла, помеченных числами 5 и 10. Этот родительский узел находится на третьем уровне дерева и соответствует ходу первого игрока, который стремится максимизировать оценочное значение. Ясно, что этот игрок выберет ход со значением 10, поэтому значение для этого родительского узла будет также 10.

Далее переместимся на вышележащий уровень с тремя дочерними узлами. Мы знаем теперь, что самый левый из этих дочерних узлов имеет значение 10, а второй узел имеет значение 7. Значение самого правого

дочернего узла будет равно максимальному из значений его дочерних узлов, т. е. 3 и 8. Отсюда для нашего узла получаем значение 8. Узел с тремя дочерними узлами находится на втором уровне и соответствует игроку, который хочет минимизировать оценочное значение. Этот игрок выберет центральный ход из трех возможных, и значение для его узла будет, соответственно, равно 7.

Процесс продолжается таким же образом. Вам придется самостоятельно завершить оценку для оставшихся узлов рисунка 4.5. Результат этой оценки показан на рис. 4.6. Значение текущей ситуации оказывается равным 7, и текущий (первый) игрок выберет самую левую ветвь, как наилучший ход из всех возможных.

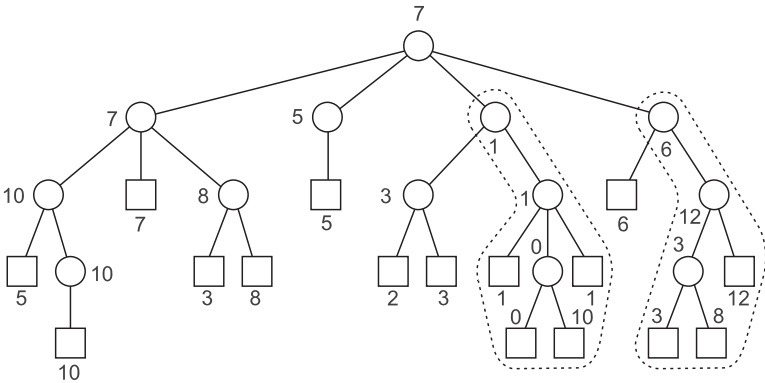


Рис. 4.6. Минимаксная оценка дерева игры

Пунктирные линии на рисунке будут пояснены позже, в одном из проектов. Оказывается, что если мы сохраняем найденные до сих пор минимумы и максимумы, нам нет необходимости оценивать каждый узел в дереве игры, и на рис. 4.6 узлы, окруженные пунктирной линией, не нуждаются в оценке. Поскольку в процессе оценки дерева игры мы попеременно берем максимальное и минимальное значения, этот процесс носит названия метода *минимакса*.

метод минимакса

### 4.2.3 Разработка алгоритма

Теперь давайте посмотрим, как метод минимакса может быть описан в виде формального алгоритма прогнозирования в игровой программе. Нам хотелось бы написать обобщенный алгоритм, который можно будет использовать в любой игре с двумя игроками; поэтому мы оставим неспецифицированными различные типы и структуры данных, поскольку их выбор зависит от условий конкретной игры.

Прежде всего нам будет нужна процедура, (назовем ее Recommend), возвращающая ходы, которые надо будет исследовать на следующем уровне прогнозирования. Для сохранения этих ходов мы должны сделать выбор из целого набора возможных структур данных. Порядок просмотра ходов не имеет значения, поэтому их можно хранить в виде множества, или списка, или стека. Ради упрощения программирования давайте

рекомендуемые  
ходы

использовать стек. Элементами стека будут служить ходы; таким образом, мы имеем **type** stackentry = move. Объявление типа move зависит от конкретной игры.

Процедура Recommend будет также возвращать значение recvalue, зависящее от текущей ситуации в игре (но еще не от той, которая будет выбрана следующим ходом). Это значение обычно является числом. Будет ли это число целым или действительным, зависит от конкретной игры.

В качестве входных данных процедура Recommend использует как текущую конфигурацию игры, так и данные о том, какой именно игрок собирается сделать ход. Для описания игрока мы используем объявление перечислимого типа **type** player = (first, second) и при этом всегда предполагаем, что первый игрок — это тот игрок, который желает максимизировать оценочное значение, а второй игрок — тот, который желает его минимизировать. Процедура Recommend выполняет также и другие операции, как это показано в приведенных ниже спецификациях.

```
procedure Recommend (var game: board; P: player; var S: stack;
                     var recvalue: value; var gameover: Boolean);
                                { Рекомендация }
```

*предусловие:* Переменная game описывает конфигурацию игры, допустимую для очередного хода игрока P.

*постусловие:* Стек S создан и содержит рекомендуемые ходы (если таковые имеются); recvalue отражает текущую ситуацию в игре, причем большее значение говорит о преимуществах первого игрока; gameover показывает, не закончена ли игра.

Перед тем, как приступить к написанию процедуры, которая с целью прогнозирования будет просматривать нижележащие ветви дерева игры, мы должны решить, когда этот алгоритм должен остановиться в своем перемещении вперед по дереву. Для игры разумной сложности мы должны ограничить величиной maxdepth число уровней просмотра. Другое условие завершения игры наступает, если переменная gameover принимает значение true. Теперь основная задача просмотра дерева с целью прогнозирования может быть описана достаточно просто посредством следующего рекурсивного алгоритма.

```
procedure LookAhead;                                { Прогнозирование }
begin                                                { процедура LookAhead }
```

Используем Recommend для получения стека S возможных ходов;

**if** рекурсия завершается (depth = 0 или gameover) **then**

Вернуть один ход и соответствующее значение

**else begin**

**for** каждого рекомендуемого хода из S **do**

Сделать пробный ход и рекурсивно вызывать LookAhead

для получения лучшего хода другого игрока;

Выбрать лучшее значение для P из значений, возвращенных в цикле;

Вернуть соответствующий ход и значение в качестве результата

**end**

**end;**

{ процедура LookAhead }

завершение

эскиз

Для определения деталей этого алгоритма мы должны, наконец, использовать еще две процедуры, зависящие от конкретной игры. Процедуры `MakeMove(P: player; m: move)` и `UndoMove(P: player; m: move)` делают и отменяют пробные ходы в соответствии с указаниями. В формальной процедуре мы также реорганизуем некоторые шаги из эскиза.

```

procedure LookAhead (var game: board; depth: integer; P: player;
                      var recmove: move; var recvalue: value);
                      { Прогнозирование }
{ Pre: Игра находится в допустимой позиции для игрока P; сейчас ход
  игрока P, и по меньшей мере один ход возможен.
Post: После просмотра вперед на depth уровней дерева игры
  процедура возвращает рекомендуемый ход recmove, который
  имеет вычисленное значение recvalue.
Uses: Использует пакет Stack; процедуры Recommend, MakeMove,
  UndoMove, LookAhead (рекурсивно), а также константу infinity
  (большую, чем значение любого хода). }

var opponent: player; { оппонент игрока P }
    oppmove: move;      { рекомендуемый ход для оппонента }
    oppvalue: value;     { значение, возвращенное для хода оппонента }
    S: stack;            { стек рекомендуемых ходов для игрока P }
    tentmove: move;      { пробный ход в дереве игры }
    gameover: Boolean;

begin { процедура LookAhead }
  Recommend(game, P, S, recvalue, gameover);
  if gameover or (depth = 0) then begin
    if not StackEmpty(S) then
      Pop(recmove, S) { вернуть любой ход в качестве ответа }
    end { значение recvalue было установлено процедурой Recommend }
    else begin { исследовать все рекомендуемые ходы }
      if P = first then begin { подготовиться к максимизации recvalue }
        opponent := second;
        recvalue := - infinity; { задать значение меньшее, чем любое
                                встречающееся }
      end
      else begin { подготовиться к минимизации recvalue }
        opponent := first;
        recvalue := infinity; { задать значение большее,
                               чем любое встречающееся }
      end;
      while not StackEmpty(S) do begin
        Pop(tentmove, S);
        MakeMove(game, P, tentmove);
        LookAhead(game, depth - 1, opponent, oppmove, oppvalue);
        UndoMove(game, P, tentmove);
        if ((P = first) and (oppvalue > recvalue)) or
          ((P = second) and (oppvalue < recvalue)) then
          begin recvalue := oppvalue; recmove := tentmove end
        end
      end
    end;
  end;
end; { процедура LookAhead }

```

## Упражнения 4.2

- E1.** Назначьте в игре «Восемь» значение  $+1$  выигрышу первого игрока и  $-1$  выигрышу второго и оцените дерево игры методом минимакса, как это было показано на рис. 4.4.
- E2.** Вариант игры «Ним» начинается с кучки палочек, из которых игроки по очереди берут 1, 2 или 3 палочки. Игрок должен взять по меньшей мере одну палочку (но не больше, чем их остается в кучке). Игрок, который возьмет последнюю палочку, проигрывает. Нарисуйте полное дерево игры, начинающейся с **(а)** 5 и **(б)** 6 палочек. Назначьте соответствующие значения листьям дерева и оцените значения других узлов методом минимакса.
- E3.** Нарисуйте верхние три уровня (показывающие первые два хода) дерева игры для игры «Крестики и нолики» и подсчитайте число узлов, из которых будет состоять четвертый уровень. Вы можете уменьшить размер дерева, учтя симметрию: на первом ходе, например, покажите только три возможности (центральное поле, угол и боковое поле), а не все девять. Учет симметрии на следующих ходах вблизи корня позволит и дальше уменьшать размер дерева игры.

## Программные проекты 4.2

- P1.** Напишите программу и другие процедуры, необходимые для игры «Восемь» против игрока-человека. Процедура `Recommend` может возвращать на каждом шаге игры все допустимые ходы.
- P2.** Напишите программу прогнозирования для игры «Крестики и нолики». В простейшем случае процедура `Recommend` возвращает в качестве рекомендуемых ходов все пустые позиции. Каково приблизительное число вариантов, просматриваемых в полном дереве игры? Реализуйте этот простой метод. Далее, модифицируйте процедуру `Recommend`, чтобы она отыскивала две метки в ряду с третьей пустой позицией и, таким образом, давала бы более интеллектуальные рекомендации. Сравните время выполнения этих двух вариантов.
- P3.** Если вы прошли по дереву на рис. 4.5 достаточно детально, вы могли заметить, что в процессе минимакса нет необходимости получать значения для всех узлов, поскольку на дереве имеются некоторые части, в которых наилучшего хода точно нет. Давайте предположим, что мы проходим по дереву, начав с низшего левого узла, и определяем значения для каждого родительского узла, как только получили значения всех его дочерних узлов. После отработки всех узлов двух главных ветвей с левой стороны, мы находим значения 7 и 5, и, следовательно, максимальное значение будет по меньшей мере 7. Когда мы переходим к следующему узлу уровня 1 и его левому дочернему узлу, мы обнаруживаем, что значение этого дочернего узла равно 3. На этом этапе мы берем минимальное значение, поэтому значение, присваиваемое родительскому узлу на уровне 1, не может быть больше 3 (фактически оно равно 1). Поскольку 3 меньше 7, первый игрок выберет самую левую ветвь, и остальные ветви мы можем исключить. Узлы, ко-

торые в результате могут оставаться неоцененными, окружены на рис. 4.6 пунктирной линией.

альфа-бета  
отсечение

Процесс такого удаления узлов носит название альфа-бета отсечения. Греческие буквы  $\alpha$  (альфа) и  $\beta$  (бета) обычно используются для обозначения найденных точек отсечения.

Модифицируйте процедуру LookAhead так, чтобы она использовала для уменьшения числа исследуемых ветвей альфа-бета отсечение. Проиграв на компьютере несколько туров, сравните производительность двух вариантов игры.

### 4.3. Компиляция методом рекурсивного спуска

Рассмотрим задачу разработки компилятора, который транслирует программу, написанную на языке Pascal, на машинный язык. По мере того, как компилятор читает строки исходной программы на языке Pascal, он должен истолковывать синтаксис и преобразовывать каждую строку в эквивалентные команды машинного языка.

идентификаторы

Первая часть Pascal-программы (или подпрограммы) содержит объявления меток, констант, типов и переменных. Компилятор использует эту информацию для выделения памяти под переменные, а также для определения того, какие операции могут выполняться над этими переменными. В то же время компилятор должен запоминать идентификаторы, которые были объявлены в качестве имен типов, переменных и прочего, чтобы интерпретировать эти идентификаторы правильным образом, когда они будут встречаться в дальнейшем тексте программы. Для того, чтобы следить за именами идентификаторов, компилятор строит **таблицу символов**. Для этой таблицы символов используются несколько различных структур данных; с некоторыми из них, например, с двоичными деревьями и хеш-таблицами, мы столкнемся в следующих главах этой книги. Сейчас, однако, мы только хотим понять в общих чертах, каким образом компилятор использует рекурсию, поэтому рассматривать детали таблиц символов мы не будем.

подпрограммы

Следующая часть Pascal-программы содержит объявления процедур и функций, а в последней части описываются действия (предложения) программы. Когда мы подходим к объявлению процедур и функций, мы находим очевидную область приложения рекурсии. Синтаксис языка Pascal разработан таким образом, что общая форма подпрограммы такая же, как и главной программы. Отсюда следует, что нет необходимости писать отдельную полную секцию компилятора ради трансляции объявлений и предложений внутри подпрограмм. После того, как все подпрограммы будут откомпилированы, компилятор приступит к трансляции предложений главной программы. Сама главная программа фактически может рассматриваться, как подпрограмма внутри вымышленного внешнего блока, в котором все стандартные идентификаторы (например, константа maxint, типы Boolean и text, процедуры writeln и dispose) уже были объявлены. При таком подходе главную программу можно рассматривать почти полностью симметрично подпрограммам.

### 4.3.1. Главная программа

В этом разделе мы будем использовать термин *модуль*, ссылаясь на любую компилируемую процедуру, функцию или программу. Общая задача компилятора выглядит следующим образом:

```
program PascalCompiler;                                {Pascal-компилятор}
begin
    Установить таблицу символов и объявлений для всех стандартных
    идентификаторов;
    Проверить, что первое введенное слово есть 'program';
    DoModule;
    Проверить, что последний символ есть точка '.'
end.
```

Процедура DoModule транслирует программу, процедуру или функцию.

Заметьте, что компилятор должен непрерывно проверять, каково следующее слово или символ в программе. В зависимости от смысла этого слова или символа будут предприниматься различные действия. Такое слово или символ называются лексемой, и для наших целей мы будем считать лексемой любое зарезервированное слово языка Pascal, идентификатор, литеральную константу, оператор или знак препинания. Заметьте, далее, что единственным способом найти конец многих предложений языка Pascal является обнаружение лексемы, *не* являющейся частью конструкции. Предложение, например, завершается, когда *следующая* лексема является любой из следующих:

- точка с запятой ';'
- ключевое слово **end**
- ключевое слово **else**
- ключевое слово **until**

В результате, когда мы начинаем обрабатывать часть программы, переменная nexttoken будет иметь значение лексемы, которая иницирует эту часть, а когда обработка закончится, переменная nexttoken получит значение первой лексемы за пределами только что обработанной части.

Разобравшись в этом вопросе, мы можем расширить процедуру DoModule. Первым шагом будет получение этой процедурой следующей лексемы, чтобы процедура могла спокойно перепрыгнуть через слова

**program**, **procedure** или **function**,

причем почти не имеет значения, какой из этих блоков обрабатывается.

```
procedure DoModule;                                { Обработать модуль }
begin
    Инициализация таблицы символов для символов этого модуля;
    GetToken(nexttoken);
    if nexttoken = '(' then DoParameters;  { возврат после парного знака ')' }
    if этот модуль есть функция then DoFunctionValue;
    if nexttoken = 'label' then DoLabelSection;
    if nexttoken = 'const' then DoConstantSection;
    if nexttoken = 'type' then DoTypeSection;
    if nexttoken = 'var' then DoVariableSection;
    while (nexttoken = 'function') or (nexttoken = 'procedure') do
        DoModule;
```

лексемы

эскиз



```

    if nexttoken = 'begin' then DoCompoundStatement;
    else Error
end;

```

### 4.3.2. Объявления типов

Чтобы познакомиться с другими приложениями рекурсии, рассмотрим чуть более детально объявление типов. В языке Pascal массивы могут содержать массивы; записи могут содержать записи. Хорошо разработанный компилятор для обработки каждой категории из стандартных категорий типов будет использовать отдельную процедуру. Таким образом, должна быть процедура DoType, которая будет по мере необходимости активизировать процедуры, которые мы назовем DoArray, DoRecord, DoSet и DoScalarType (среди прочих).

То, как эти процедуры работают, в значительной степени определяется их синтаксическими диаграммами. В частности, процедура DoArray активизирует процедуру DoScalarType, чтобы определить тип индексной переменной, и DoType, чтобы определить тип элементов в массиве. DoRecord активизирует процедуру с именем DoVariable для каждого поля внутри записи. DoVariable, в свою очередь, вызовет DoType, чтобы определить тип переменной. Эти рекурсивные вызовы, спускаясь все ниже и ниже, постепенно дойдут до простых типов, и смысл всей конструкции будет тогда определен.

Процесс расщепления текста или выражения на отдельные части, чтобы определить их синтаксис, называется *синтаксическим разбором* или *синтаксическим анализом*. Синтаксический анализ может выполняться двумя способами. В первом сначала анализируются атомарные (неделимые) блоки и способ их объединения; такой способ называется *восходящим синтаксическим анализом*. В другом способе (*нисходящим синтаксическим анализом*) текст или выражение расщепляется на простейшие компоненты. Отсюда возникло и выражение *рекурсивный спуск*: компилятор разбирает большие конструкции, расщепляя их на меньшие части, и далее рекурсивно разбирает каждую из этих частей. Таким образом, компилятор спускается к более простым и меньшим по размеру конструкциям, которые в конце концов расщепляются на свои атомарные компоненты, которые уже могут быть оценены непосредственным образом.

В качестве примера рассмотрим следующее объявление, которое приводит к дереву вызовов процедур, показанному на рис. 4.7. Предполагается, что скалярный тип index ранее уже был определен в программе.

```

type
  item = array [index] of
    record
      a: record
        u: integer;
        v: index
      end
      b: set of index;
      c: array [index] of real;
    end;

```

синтаксический  
анализ

рекурсивный  
спуск

пример



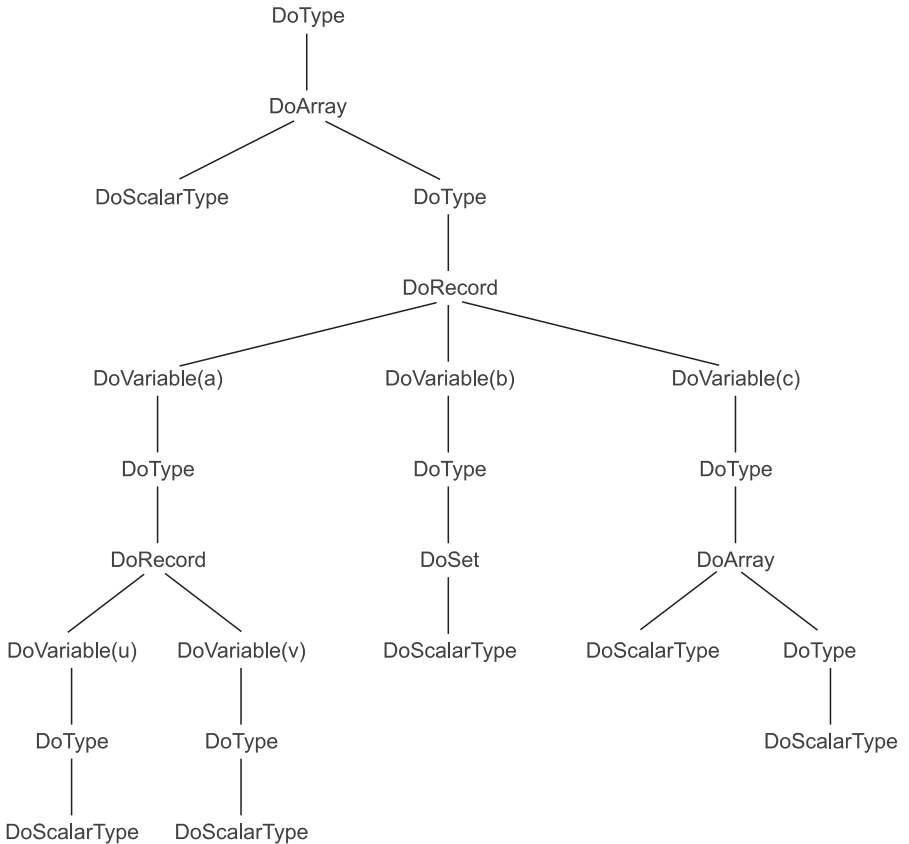


Рис. 4.7. Дерево синтаксического анализа объявления типа, использующее метод рекурсивного спуска

### 4.3.3. Синтаксический анализ предложений

Теперь обратимся к предложениям программы, т. е. к той части, которая выполняет требуемые от программы действия. Для синтаксического анализа предложений в программе мы будем использовать метод рекурсивного спуска. Активная часть программы представляет собой одно составное предложение (длящееся от **begin** до **end**), синтаксический анализ которого осуществляется процедурой `DoCompoundStatement`. Составное предложение состоит из нуля или большего числа предложений, каждое из которых будет анализироваться процедурой `DoStatement`, которая, в свою очередь, активизирует различные процедуры для каждого из возможных типов предложений.

В качестве примера рассмотрим, как может быть выполнен синтаксический анализ и трансляция на язык ассемблера (другими словами, на язык, который непосредственно соответствует командам машинного

уровня, но в то же время допускает использование символических имен и меток предложений) предложения **if**. Из его синтаксической диаграммы мы знаем, что предложение **if** состоит из лексемы **if**, за которой следует булево выражение, далее — лексема **then** и предложение и, наконец, необязательная лексема **else** с другим предложением. См. рис. 4.8.

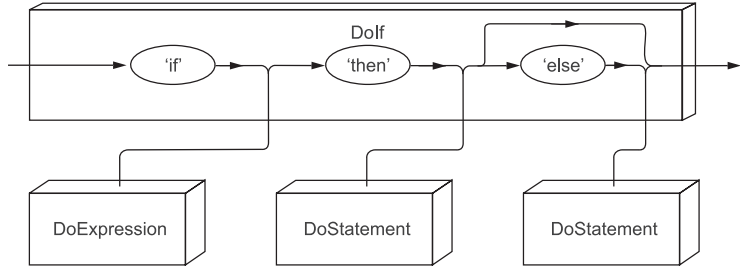


Рис. 4.8. Дерево синтаксического анализа предложения **if**

Эквивалент предложения **if** на языке ассемблера прежде всего оценит значение выражения, а затем использует команды условного перехода (предложения **goto**) для обхода фрагментов ассемблерного кода, соответствующих предложениям в конструкциях **then** или **else**, в зависимости от полученного ранее значения выражения. Синтаксическая диаграмма, таким образом, транслируется в приведенную ниже программу.

Как уже отмечалось в разделе, посвященном синтаксическому анализу объявлений, каждая часть предложения обычно распознается только когда встречается не принадлежащая этому предложению лексема. Поэтому мы снова используем процедуру `GetToken` и следим за тем, чтобы переменная `nexttoken` указывала на один шаг вперед по отношению к тому, что сейчас обрабатывается.

```

procedure Dolf;                                { Отработать if }
{ Pre: В программном коде встретилось предложение if.
  Post: Предложение if транслировано на язык ассемблера; указатель
        nexttoken перемещен на следующее зарезервированное слово.
  Uses: Использует процедуры GetToken, DoExpression, DoStatement. }
begin                                           { процедура Dolf }
  GetToken(nexttoken);
  DoExpression;                                { написать код для оценки булева выражения }
  if nexttoken <> ' then '
    then Error
  else
    begin
      Генерировать новую ассемблерную метку x;
      Написать ассемблерный код, который приведет к условному
      переходу на метку x, если булево выражение окажется ложным;
      GetToken(nexttoken);
      DoStatement;
      { написать код для вложенного предложения в предложении then }
    end;

```

```

if nexttoken = ' else' then
begin
    Генерировать новую ассемблерную метку y;
    Написать ассемблерный код для безусловного перехода на метку y;
    Включить в это место ассемблерного кода метку x;
    GetToken(nexttoken);
    DoStatement;
    { написать код для вложенного предложения в предложении else }
    Включить в это место ассемблерного кода метку y;
end
else
    Включить в это место ассемблерного кода метку x;
    { нет предложения else }
    { процедура Dolf }
end;
    
```

В качестве упражнения вы можете приложить этот алгоритм к предложению

упражнение

```

if a > 0 then b := 1
else if a = 0 then b:= 2
else b := 3;
    
```

Поскольку этот программный фрагмент состоит из двух предложений **if**, процедура синтаксического анализа будет рекурсивно вызывать сама себя и сгенерирует в целом четыре метки, по две на каждое предложение **if**. Дерево синтаксического анализа для этого предложения показано на рис. 4.9.

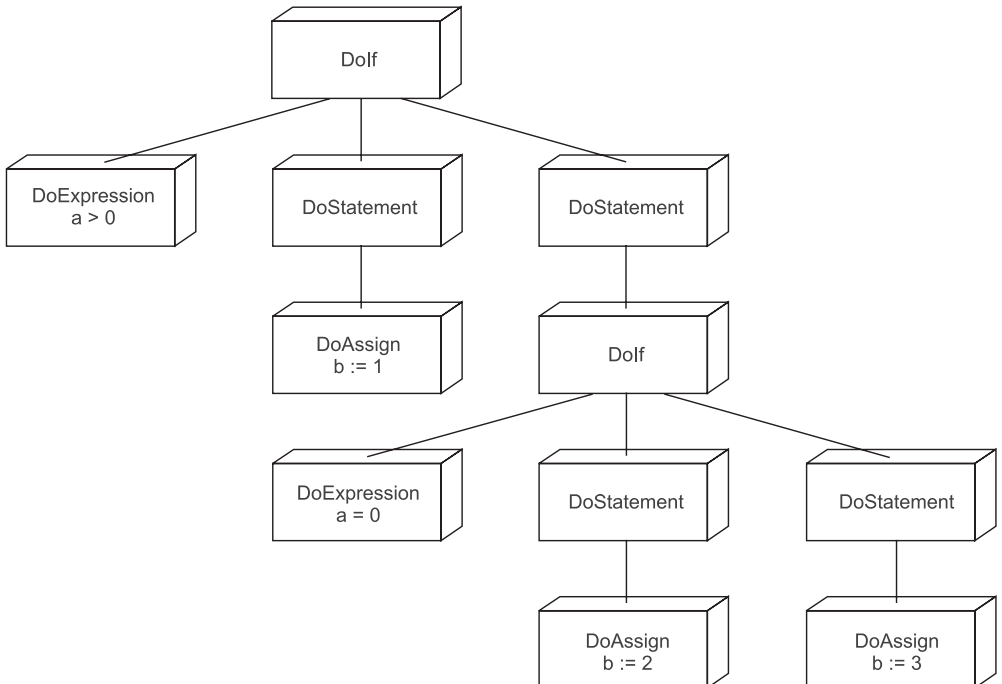


Рис. 4.9. Дерево синтаксического анализа вложенного предложения **if**

пример вывода  
компилятора

В этом конкретном примере обе конструкции **if** содержат предложения **else**. Если конструкция **if** не содержит предложения **else**, тогда в дереве синтаксического анализа будет только вызов `DoStatement`, расположенный непосредственно под соответствующим `Dolf`.

На практике реальный компилятор может реорганизовать некоторые из переходов и получить таким образом ответ, внешне отличный от результата алгоритма `Dolf`. Вывод одного из таких компиляторов, выполнившего трансляцию предыдущего предложения, приведен на рис. 4.10, где фактический вывод компилятор был несколько изменен, именно, числовые смещения, определенные компилятором, были заменены на символические метки. Используемый для этого примера компилятор генерирует дополнительные предложения с метками, потому что, в силу некоторых технических причин, он должен следовать затратной практике генерации команд перехода, которые ничего не делают, кроме обхода других команд переходов. В результирующем ассемблерном коде команда `TST` анализирует знак целого числа; команда `BEQ` осуществляет переход, если целочисленная переменная равна 0, а `BGT` выполняет переход, если она больше 0. Команда `JMP` осуществляет безусловный переход. `MOV` перемещает первое указанное число в ячейку, указанную в качестве второго операнда.

---

	TST	A	; if a > 0 then
	BGT	L0	
	JMP	L1	
L0:	MOV	#1,B	; b := 1
	JMP	L2	; else if a = 0 then
L1:	TST	A	
	BEQ	L3	
	JMP	L4	
L3:	MOV	#2,B	; b := 2
	JMP	L5	; else
L4:	MOV	#3,B	; b := 3
L5:			
L2:			

---

Рис. 4.10. Вывод компилятора с языка Pascal

компиляторы

Как видно из рассмотренного примера, разработка компилятора является весьма сложным процессом. Типичный Pascal-компилятор (если его написать на языке Pascal) может занимать от пяти до десяти тысяч строк кода, а возможно даже и значительно больше.

Наше обсуждение имело своей целью лишь в общих чертах показать полезность рекурсии при разработке компиляторов. Если вы рассмотрите этот процесс более детально, то вы увидите, что мы не только опустили многие важные шаги, но даже не упомянули целый ряд проблем (например, обработку ошибок). При рассмотрении этих проблем обнаруживает-

ся много интересных идей, и вообще следует сказать, что разработка и реализация компиляторов составляет важнейший раздел вычислительной техники, достойный самого серьезного изучения.

#### 4.3.4. Синтаксический анализатор предложений языка Pascal

В качестве завершающего этот раздел упражнения давайте разработаем программу, которая проиллюстрирует некоторые шаги синтаксического анализа Pascal-программы. Наша программа будет распознавать все предложения Pascal-программы. Ее выводом будет схема программы с указанием того, на какой строке начинается каждое предложение, к какого рода предложениям оно относится и где оно заканчивается. Эта программа продемонстрирует некоторые шаги, через которые необходимо пройти при написании компилятора методом рекурсивного спуска, но избавит нас от многих сложностей, с которыми мы столкнулись бы в реальном компиляторе. Наша программа пропустит все объявления меток, констант, типов и переменных в начале анализируемой программы. Она затем распознает предложения, входящие как в подпрограммы, так и в главную программу, и про каждое из этих предложений напечатает одну или две строки, указав вид предложения, а также номера начальной и конечной строк. Для показа вложенности предложений друг в друга мы будем использовать отступы.

Прежде всего проиллюстрируем синтаксический анализ такого рода на примере очень простого фрагмента, состоящего из единственного предложения **if** с вызовами процедур:

```

program Test(input, output); { 1 }
var number: integer;         { 2 }
begin                         { 3 }
  readln(number);             { 4 }
  if number > 0 then           { 5 }
    writeln('положительно')   { 6 }
  else                         { 7 }
    writeln('отрицательно');   { 8 }
end.                           { 9 }

```

Результат синтаксического анализа этого фрагмента, являющийся выводом нашей программы, будет выглядеть следующим образом:

```

1 PROGRAM
3 Составное предложение
4   Вызов процедуры
5   IF
6     Вызов процедуры
7     Вызов процедуры конец
7   ELSE, начало на строке 5
8     Вызов процедуры
8   IF конец, начало на строке 5
9     пусто
9 Составное предложение конец, начало на строке 3
9 Программа конец, начало на строке 1

```

Обратите особое внимание на пустое предложение в строке 9. Это не лексема **end**; пустое предложение появляется из-за того, что точка с запятой на строке 8 *разделяет* два предложения, и, следовательно, должно быть предложение между точкой с запятой и лексемой **end** на строке 9. Поскольку никаких предложений не указано, это предложение пустое. Если точку с запятой на строке 8 удалить, пустое предложение исчезнет.

Заметьте также, что вызов процедуры, начинающийся на строке 6, не заканчивается до строки 7, несмотря на то, что другие вызовы процедур заканчиваются на тех же строках, где они начинаются. Причина этого в том, что на строке 6 нет лексемы, которой закончился бы вызов процедуры, и (как думает синтаксический анализатор) на следующей строке могли бы быть дополнительные параметры.

В Pascal-программах могут возникать и многие другие осложнения, и каждый вид предложения требует индивидуального внимания. Для того, чтобы преодолеть все эти осложнения, мы должны выполнять наш анализ с большой тщательностью, следя за строгим выполнением синтаксических правил языка Pascal по мере анализа входных строк.

## 1. Главная программа

Базовая структура главной программы весьма прямолинейна, однако она должна включать объявления нескольких глобальных переменных; зачем это нужно, станет ясно только после обсуждения других частей программы.

```

program Parse(input, output, infile, outfile);           {Синтаксический анализ}
{ Pre: Пользователь предоставляет имя файла. Этот файл должен
      содержать синтаксически правильную Pascal-программу (т. е.
      она должна компилироваться без ошибок стандартным
      Pascal-компилятором). Пользователь также предоставляет имя
      выходного файла.

      Post: Программа создала выходной файл, в котором для каждого
      предложения Pascal-программы имеется строка, содержащая
      номер программной строки, на которой начинается
      предложение, а также вид предложения. Вторая строка
      показывает, где это предложение заканчивается; эта строка
      может быть подавлена, если программное предложение
      уместается на одной строке. Вложение предложений
      показывается количеством отступов. }

uses
  Utility,      { специфичные для Turbo Pascal откомпилированные модули }
  FileIO;

const
  endfile = chr(26);
              { используем ctrl-Z (в коде ASCII) в качестве символа конца файла }

var
  infile,      { входной файл, предоставляемый пользователем }
  outfile: text;      { выходной файл, создаваемый программой }
  nexttoken: string;      { текущая лексема, готовая к обработке }
  newline,      { находимся ли мы в начале следующей строки? }
  foundgets: Boolean;      { имеется ли в предложении := ? }
  parencount,      { сколько имеется уровней скобок? }
  lineno: integer;      { номер текущей строки }
  nextchar: char;      { следующий (первый необработанный) символ }

```

```

begin
    writeln('Эта программа читает Pascal-программу и находит
           все ее предложения.');
```

OpenExtensionInFile(infile, '.PAS');

OpenExtensionOutFile(outfile, '.LST');

lineno := 0;

newline := true;

parencount := 0;

nextchar := ' ';

GetToken(infile, nexttoken, lineno);

if nexttoken <> 'program' then

    Error('Первым ключевым словом файла не является слово "PROGRAM.");

else

    DoProgram(lineno);

if nexttoken <> endfile then

    Error('В программе содержится материал за ее концом.')

CloseInFile(infile);

CloseOutFile(outfile);

writeln('Завершена обработка ', lineno - 1, ' строк.');

end.

{ главная программа Parse }

Поскольку форма файлового ввода-вывода сильно зависит от компилятора, задача открытия файла передается модулю. Программа использует строки символов, которые опять же являются средством, определяемым компилятором.

## 2. Поиск единичной лексемы

Важной частью нашей задачи является чтение входного файла и разделения его на слова (т. е. строки символов, удовлетворяющие синтаксису идентификаторов языка Pascal) и на специальные символы. К последним относятся ';', ':=', ': ' и '!. Эти символы как раз и есть те самые лексемы, которые должна найти наша программа. Заметьте, однако, что в настоящем Pascal-компиляторе приходится искать и другие лексемы, например, числа и строки символов, заключенные в кавычки.

Перед тем, как начать синтаксический анализ лексемы, наша программа должна распознать начало комментария с целью игнорирования всего текста до конца комментария. Вне комментариев мы должны также распознавать начальную кавычку и игнорировать все до завершающей кавычки (или до конца строки, поскольку в языке Pascal текст в кавычках не может выходить за пределы входной строки).

Некоторым затруднением на этом этапе является необходимость исследования позиции, следующей за той, которая нас, собственно, интересует. Например, чтобы определить, нашли ли мы ':=', мы должны проанализировать символ *после* двоеточия. Этот символ может быть знаком равенства, но может быть и чем-то еще, например, буквой, с которой начинается следующее слово. Мы не можем прочесть этот следующий символ и затем отбросить его, если мы возвращаем ':', поскольку при следующем вызове процедуры, выделяющей лексему, этот символ может понадобиться. Аналогично, единственным способом убедиться в том, что мы дошли до конца идентификатора, является обнаружение символа, не являющегося ни буквой, ни цифрой, причем этот символ нам может понадобиться в следующий раз.

В стандартном языке Pascal эту проблему легко решить, если мы читаем входные данные (текст) символ за символом. Мы можем выполнить процедуру `read(F, ch)` для получения одного символа (`F` — это входной файл, а `ch` — символьная переменная), и после чтения файлового окна `F↑` есть автоматически следующий символ. Другими словами, мы можем использовать `F↑` для просмотра следующего символа без его официального чтения.

Мы, однако, вместо Pascal будем использовать Turbo Pascal, в котором такой упреждающий просмотр организовать сложнее, так как Turbo Pascal не использует файловых окон. Чтобы решить эту проблему, нам понадобится зарезервировать глобальную переменную с именем `nextchar`, и всегда читать на один символ дальше того места, где мы работаем. Для того, чтобы получить следующий текущий символ, мы копируем `nextchar` в переменную для текущего символа, а затем читаем следующий символ в `nextchar`.

Со всеми этими добавлениями процедура выглядит так:

```
procedure GetToken(var infile: text; var nexttoken: string;
                  var lineno: integer); { Получить лексему }
{ Pre: Файл infile открыт и готов к чтению.
  Post: Следующая лексема из файла возвращена как nexttoken,
        а в lineno содержится номер текущей строки, той строки, на
        которой находится nexttoken. Лексема определяется в языке
        Pascal как идентификатор, или ключевое слово (начинающееся
        с буквы и содержащее исключительно буквы и цифры), или один
        из специальных символов ' ', ':', '!' и ' '. Символ eof(infile)
        возвращается только если eof(infile) принимает значение true.
        Символы в комментариях, а также строки, заключенные
        в кавычки, при поиске следующей лексемы пропускаются.
        Уровень вложенности скобок определен.
  Uses: Использует newline и nextchar в качестве глобальных
        переменных. }
var current: char; { текущий символ, прочитанный из файла infile }
begin { процедура GetToken }
  nexttoken := ""; { начинаем с того, что возвращать нечего }
  repeat { повторять до тех пор, когда nexttoken станет непустым }
    GetNextChar;
  if current in ['a'..'z'] then { Вытащим целое Pascal-слово }
    begin
      nexttoken := current;
      while nextchar in ['0'..'9', 'a'..'z'] do begin
        nexttoken := nexttoken + nextchar; { специфично для символьных
                                           цепочек Turbo Pascal }
      GetNextChar
    end
  end
  else if (current = '{') or ((current = '(') and (nextchar = '*')) then begin
    repeat { пропустить комментарии }
      GetNextChar
    until (current = '}') or ((current = '(') and (nextchar = ')'))
      or eof(infile)
    { проверить на незавершенный комментарий }
```



```

    if current = '*' then
        GetNextChar
        { пропускаем '*', чтобы следующий проход пропустил '}' }
    end
    else if current = '"' then begin
        repeat
            GetNextChar
        until (current = '"') or newline;
        if current <> '"' then begin
            Error('Строка в кавычках в языке Pascal не может быть длиннее
                входной строки');
            writeln('Номер строки ', lineno)
        end
    end
    else if current = '(' then
        parencount := parencount + 1
    else if current = ')' then
        parencount := parencount - 1
    else if current = ';' then
        nexttoken := ';'
    else if current = ':' then
        if nextchar = '=' then begin
            GetNextChar;
            nexttoken := ':'=
        end
    else
        nexttoken := ':'
    else if current = '.' then
        nexttoken := '.'
    until nexttoken <> " { повторять, пока nexttoken не станет чем-либо,
                        что нуждается в обработке }
end; { процедура GetToken }

```

Эта процедура использует вспомогательную процедуру для получения одного символа и обновления буфера. Поскольку мы всегда читаем на один символ вперед по отношению к тому, который мы сейчас используем, условие конца строки и место, над которым мы работаем, становятся рассинхронизованными. Поэтому вместо непосредственного использования условия `eoln(infile)` мы должны ввести булеву переменную и использовать ее по мере необходимости для обновления счетчика строк.

Поскольку Pascal игнорирует регистр клавиатуры, для преобразования всех букв в строчные мы используем утилиту из приложения С.

```

procedure GetNextChar; { Получить следующий символ }
{ Pre: Файл infile открыт для чтения.
  Post: Читает один символ в current и продвигает буфер просмотра
        вперед к следующему символу; обновляет при необходимости
        lineno; встретившись с eof(infile), устанавливает
        nexttoken := endfile.
  Uses: Использует все переменные из процедуры GetToken
        как глобальные; эта процедура предназначена
        для использования исключительно как часть GetToken. }
begin { процедура GetNextChar }
    current := nextchar; { используем символ из буфера }

```

```

if newline then { поскольку лексема, завершающая предыдущую строку,
                  должна быть обработана перед обновлением lineno, счетчик строк
                  обновляется здесь, а не тогда, когда обнаруживается eoln(infile) }
begin
  lineno := lineno + 1;
  newline := false;
end;
if eof(infile) then
  nexttoken := endfile { установим флаг для завершения программы }
else if eoln(infile) then
begin
  nextchar := ' ';
  newline := true;
  readln(infile)
end
else begin
  read(infile, nextchar);
  LowerCase(nextchar)
end
end;
                                     { процедура GetNextChar }

```

На этом завершается обработка входных символов на низком уровне. Теперь все процедуры должны в качестве единиц входной информации рассматривать только лексемы.

### 3. Пропуск лексем

Поскольку мы (в этом проекте) интересуемся только распознаванием предложений, наша программа будет игнорировать большое количество лексем, которые не могут служить началом новых предложений. Например, мы можем игнорировать все списки параметров, все объявления констант, типов и переменных, а также все лексемы внутри выражений.

По этой причине мы начинаем с прямолинейной процедуры:

```

procedure Skippast(stoptoken: string); { Пройти мимо }
{ Pre: Лексема stoptoken должна обязательно встретиться где-либо
        далее в файле infile.
Post: Текущая позиция в файле продвинута вперед и установлена
        точно за первым вхождением stoptoken (вне комментариев
        и кавычек).
Uses: Использует infile, как глобальную переменную; изменяет
        nexttoken как глобальную переменную. }
begin                                     { процедура Skippast }
  while(nexttoken <> stoptoken) and (nexttoken <> endfile) do
    GetToken(infile, nexttoken, lineno); { чтобы избежать бесконечного
                                          цикла, проверяем на eof }
  if nexttoken = endfile then begin
    writeln('ERROR: поиск "', stoptoken, '" обнаружил вместо этого');
    writeln('конец файла.')
  end
  else
    GetToken(infile, nexttoken, lineno) { пропускаем саму лексему }
  end;                                { процедура Skippast }

```

Одним из важнейших приложений процедуры игнорирования лексем является обход выражения, что позволяет нам найти следующее предложение. Внутри предложения **if**, например, имеется выражение между лек-

семами **if** и **then**, за которым следует новое предложение и (необязательно) лексема **else** и другое предложение. Обычно мы можем обойти это выражение, найдя лексему, расположенную за ним. Иногда, однако, как, например, в предложении **repeat**, выражение находится в конце предложения, и тогда для того, чтобы найти конец выражения, мы должны отыскать лексему, завершающую предложение. Отложив пока решение этой проблемы, мы получаем:

```
procedure SkipExpression(stoptoken: string);      { Пропустить выражение }
{ Pre: Текущая позиция находится в начале выражения, и символьная
      строка stoptoken обязательно должна встретиться после
      выражения. Если лексема после выражения есть конец
      предложения, тогда stoptoken = ".
  Post: Символьная строка nexttoken есть первая лексема за концом
        выражения.
  Uses: Используем GetToken, Skippast, StopsStatement. }
begin                                           { процедура Skippast }
  if stoptoken <> " then
    Skippast(stoptoken)
  else    { должны проверить на лексемы, заканчивающие предложения }
    while not StopsStatement(nexttoken) do
      GetToken(infile, nexttoken, lineno)
  if nexttoken = endfile then
    Error('Внутри выражения встретился конец файла')
end;                                           { процедура SkipExpression }
```

Для того, чтобы написать функцию **StopsStatement**, нам нужно определить, какая лексема может идти непосредственно за концом предложения. Сделать это можно, только проанализировав все синтаксические диаграммы, приведенные в приложении D. Если рамка с надписью 'предложение' показана в конце синтаксической диаграммы, то она не дает никакой информации относительно того, что будет идти за ней, однако если такая рамка встречается в середине диаграммы, тогда идущая за ней лексема как раз и завершает предложение. Просматривая все синтаксические диаграммы, мы находим, что лексемами, идущими непосредственно после предложений, являются **;**, **end**, **else** и **until**. В результате мы имеем такой текст функции **StopsStatement**:

```
function StopsStatement (stoptoken: string);      {Конец предложения}
{ Pre: nexttoken является лексемой.
  Post: Функция возвращает true, если nexttoken является одной
        из лексем end, ;, else или until, которые заканчивают
        предложение. }
begin                                           { функция StopsStatement }
  StopsStatement := (nexttoken = ';')    or
                    (nexttoken = 'end') or
                    (nexttoken = 'else') or
                    (nexttoken = 'until') or
                    (nexttoken = endfile) {только для проверки на ошибку }
end;                                           { функция StopsStatement }
```

Последняя часть программы, которую мы должны обойти, включает объявления меток, констант, типов и переменных. Опять же из синтаксических диаграмм приложения D мы можем найти, что непосредственно



Первые три вида предложений на упоминавшейся выше диаграмме должны обрабатываться особым образом. Это пустое предложение (обозначаемое прямой линией в верхней части диаграммы), предложение присваивания и предложение процедуры. Остальные предложения начинаются с уникальных ключевых слов и могут быть опознаны по этим ключевым словам. Пустое предложение распознается немедленным обнаружением лексемы, завершающей предложение (и, таким образом, в этом предложении ничего нет). Два других типа предложений, присваивания и процедуры, обрабатываются одинаково, а различаются по лексеме ':' в предложении присваивания и отсутствию ее в операторе процедуры. В результате мы имеем:

```
procedure DoStatement (var lineno: integer; indent: integer);
                                { Обработать предложение }
{ Pre: Глобальная переменная nexttoken есть первая лексема предложения
  (или завершающая лексема, если предложение пусто).
  Post: Предложение полностью обработано, и nexttoken есть первая
        лексема после конца предложения. }
begin                                { процедура DoStatement }
  if nexttoken = ':' then begin
    Out('метка предложения', lineno, indent);
    GetToken(infile, nexttoken, lineno)
  end;
  if nexttoken = 'begin' then DoCompound(lineno, indent + 1)
  else if nexttoken = 'if' then DoIf(lineno, indent + 1)
  else if nexttoken = 'case' then DoCase(lineno, indent + 1)
  else if nexttoken = 'while' then DoWhile(lineno, indent + 1)
  else if nexttoken = 'for' then DoFor(lineno, indent + 1)
  else if nexttoken = 'repeat' then DoRepeat(lineno, indent + 1)
  else if nexttoken = 'with' then DoWith(lineno, indent + 1)
  else if nexttoken = 'goto' then DoGoto(lineno, indent + 1)
  else if StopsStatement(nexttoken) then DoEmpty(lineno, indent + 1)
  else DoOther(lineno, indent + 1)
end;                                { процедура DoStatement }
```

Теперь читателю должно быть понятно, как писать процедуры для синтаксического анализа конкретных предложений, и эту работу можно оставить для упражнений. Например, процедура для синтаксического анализа предложения **while**, соответствующая диаграмме в разделе предложения циклов в приложении D, выглядит так:

```
procedure DoWhile(var lineno: integer; indent: integer); { Обработать while }
{ Pre: Глобальная переменная nexttoken является словом while, которое
  начинает синтаксически правильное предложение цикла while.
  Post: Предложение while и включенные в него предложения
        обработаны, и nexttoken продвинут вперед и установлен точно
        за концом предложения while.
  Uses: Использует nexttoken как глобальную переменную. }
var startline: integer;
begin                                { процедура DoWhile }
  startline := lineno;
  Out('WHILE', lineno, indent);
  SkipExpression('do');
  DoStatement(lineno, indent + 1);
  Outend('WHILE конец', lineno, indent, startline)
end;                                { процедура DoWhile }
```

В этой процедуре также используется процедура вывода:

```
procedure Outend(message: string; lineno, indent, startline: integer)
```

которая выводит в выходной файл строку с сообщением о конце предложения.

Предложение **case**, синтаксис которого можно найти в разделе предложения выбора в приложении D, оказывается более сложным. Поскольку внутри этого предложения может быть любое количество вариантов выбора, для обработки всех этих вариантов мы должны использовать цикл. Поскольку синтаксис требует, чтобы в предложении **case** был по меньшей мере один вариант выбора, требуется цикл **repeat**. После предложения внутри варианта выбора стоит либо точка с запятой, либо лексема **end**. Последний вариант выбора, однако, может иметь точку с запятой перед **end**, но может и не иметь. (В случае точки с запятой, согласно синтаксису языка Pascal для этого типа предложений, перед **end** по-прежнему отсутствует пустое предложение.) Чтобы учесть все эти возможности, процедуру обработки предложения **case** следует писать с особой тщательностью.

```
procedure DoCase(var lineno: integer; indent: integer);
    { Обработать предложение case }
{ Pre: Глобальная переменная nexttoken является словом case,
    которое начинает синтаксически правильное предложение case.
Post: Предложение case и включенные в него предложения
    обработаны, и nexttoken продвинут вперед и установлен точно
    за концом предложения case.
Uses: Используем nexttoken как глобальную переменную. }
var startline: integer;
begin                                     { процедура DoCase }
    startline := lineno;
    Out('CASE', lineno, indent);
    SkipExpression('of');
    repeat                                { поскольку должен быть по меньшей мере
                                          один вариант выбора, используем цикл repeat }
        Skippast(':');
        DoStatement(lineno, indent + 1);
        if nexttoken = ';' then GetToken(infile, nexttoken, lineno);
        else if nexttoken <> 'end'
            then Error('Нераспознанное ключевое слово в CASE.')
        until (nexttoken = 'end') or (nexttoken = endfile);
        GetToken(infile, nexttoken, lineno);    { пропустим лексему end }
        Outend('CASE конец', lineno, indent, startline)
    end;                                     { процедура DoWhile }
```

Следующая процедура обрабатывает предложения присваивания и вызова процедур.

```
procedure DoOther(var lineno: integer; indent: integer);
    { Обработать другое }
{ Pre: Глобальная переменная nexttoken является идентификатором,
    а не ключевым словом, поэтому текущее предложение есть
    либо предложение присваивания, либо вызов процедуры.
```

**Post:** Текущее предложение обработано, и nexttoken является первой лексемой за концом предложения.

**Uses:** Использует nexttoken как глобальную переменную. }

```
var startline: integer;
    foundgets: Boolean;
begin
    startline := lineno; foundgets := false;
    repeat
        GetToken(infile, nexttoken, lineno);
        if nexttoken = ':' then foundgets := true;
    until StopsStatement(nexttoken);
    if foundgets then begin
        Out('Присваивание', startline, indent);
        if lineno <> startline then Outend('Присваивание конец', lineno, indent,
                                           startline)
    end
    else begin
        Out('Вызов процедуры', startline, indent);
        if lineno <> startline then Outend('Вызов процедуры конец', lineno, indent,
                                           startline)
    end
end;
{ процедура DoOther }
```

## 5. Программные объявления

```
procedure DoProgram(var lineno: integer); { Обработать program }
{ Pre: Глобальная переменная nexttoken является ключевым словом
      program, которым начинается программа.
  Post: Вся программа обработана, и nexttoken продвинут вперед
        и установлен за завершающей программу лексемой end
        и следующей за ней точкой '.'
  Uses: Использует nexttoken как глобальную переменную. }
var startline: integer;
begin
    startline := lineno;
    Out('PROGRAM', lineno, 0);
    SkipDeclarations;
    while (nexttoken = 'procedure') or (nexttoken = 'function') do
        DoSubprogram(lineno, 2);
    if nexttoken = 'begin' then DoCompound(lineno, 0)
    else Error('В программе не удалось найти BEGIN');
    if nexttoken <> '.' then Error('Программа не заканчивается точкой')
    else begin
        GetToken(infile, nexttoken, lineno); { пропустим последнюю точку '.' }
        Outend('Программа конец', lineno, 0, startline)
    end
end;
{ процедура DoProgram }
```

Процедуры, обрабатывающие объявления функций и процедур, весьма схожи с приведенной выше процедурой, и мы их оставим для упражнений. Для удобства мы используем еще одну процедуру, DoSubprogram, задачей которой является просто вызов, в зависимости от конкретной ситуации, одной из двух процедур: DoProcedure или DoFunction.



### Упражнения 4.3

- E1.** Для каждого из приведенных ниже объявлений нарисуйте дерево по аналогии с рис. 4.7 с указанием вызовов подпрограмм, которые будут происходить по ходу синтаксического анализа объявления. Считайте, что `index` является скалярным типом, не требующим дальнейшего анализа.
- (a) `type complex = record x: real; y: real end;`
  - (b) `type listtype = record count: index; L: array [index] of item end;`
  - (c) `var X: listtype;` (Разверните `listtype` согласно п. b)
  - (d) `var A: array [index] of set of index;`
- E2.** Для каждого из приведенных ниже предложений нарисуйте дерево по аналогии с рис. 4.9 с указанием вызовов подпрограмм, которые будут происходить по ходу синтаксического анализа предложения. Вы можете считать, что подпрограммы `DoCase`, `DoWhile` и другие, подобные им, уже существуют.
- (a) `while x > 0 do if x > 10 then x := - x else x := y - x;`
  - (b) `if a > b then if c > d then x := 1 else if e > f then x := 3;`
  - (c) `begin end;`
- E3.** Нарисуйте деревья синтаксического анализа и напишите эскизы процедур для синтаксического анализа следующих видов предложений.
- (a) `while` выражение `do` предложение
  - (b) Составное предложение: `begin` предложение (предложения) `end`
  - (c) `case ... end`
  - (d) `for` переменная := выражение `to` (или `downto`) выражение `do` предложение
- E4.** Напишите Pascal-процедуру `GetToken`, которая будет читать входной текст и возвращать по одной лексеме (как это определено в настоящей главе) при каждом вызове процедуры.

### Программные проекты 4.3

- P1.** Закончите программу распознавания предложений в файле, содержащем синтаксически правильную Pascal-программу. Вам понадобится написать процедуры `Outend`, `DoCompound`, `Dof`, `DoFor`, `DoWith`, `DoEmpty`, `DoSubprogram`, `DoProcedure` и `DoFunction`. Поскольку программа использует взаимную рекурсию между подпрограммами, вам в некоторых местах придется использовать упреждающие объявления. Если вы расположите подпрограммы в удачном порядке, упреждающие объявления понадобятся только для процедур `DoStatement` и `DoSubprogram`.

## Подсказки и ловушки

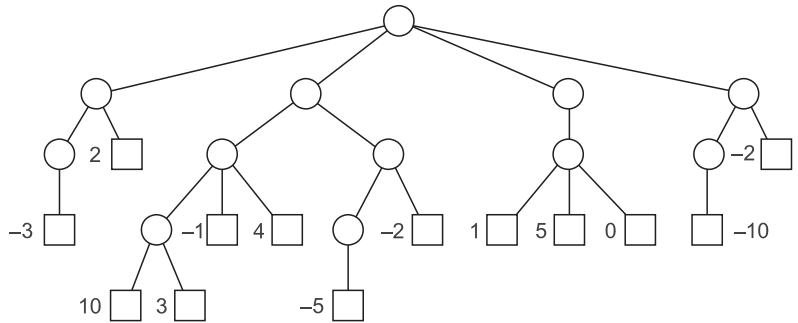
1. Рассмотрите свою задачу с точки зрения ее удовлетворения стандартным принципам рекурсивных алгоритмов, таких как метод разбиения, отход, алгоритмы с древовидной структурой и рекурсивный спуск.
2. Проследите, чтобы использование рекурсии соответствовало структуре задачи. Когда условия задачи будут полностью осознаны, структура требуемого алгоритма станет более наглядной.



3. Всегда обращайтесь особое внимание на граничные случаи. Проследите за корректным завершением вашего алгоритма при достижении им конца решаемой им задачи.
4. Всегда выполняйте максимально тщательную проверку на ошибки. Удостоверьтесь в том, что требования, необходимые для выполнения подпрограммы, указаны в ее предусловиях, и даже при этом старайтесь защитить свою подпрограмму от максимально возможного числа нарушений ее предусловий.

## Обзорные вопросы

- 4.1 1. Опишите *отход* как метод решения задачи.  
 2. Определите принцип *голубятни*.
- 4.2 3. Объясните, в чем заключается метод *минимакса* при поиске наилучших ходов в игре.  
 4. Оцените значения узлов в приведенном ниже дереве игры методом *минимакса*.



- 4.3 5. Определите термины *синтаксический анализ* и *лексема*.  
 6. Что такое *рекурсивный спуск*?  
 7. Нарисуйте дерево синтаксического анализа для объявления типа  
`list = record e: array [1 .. max] of item; n: 0 .. max end;`  
 8. Нарисуйте дерево синтаксического анализа для предложения  
`for i := 1 to 10 do if odd(i) then Process(i) else Redo(i);`

## Литература для дальнейшего изучения

В конце предыдущей главы дан перечень полезных источников с примерами и приложениями рекурсии.

Наш подход к решению задачи восьми ферзей взят из книги N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, N.J., 1976, pp 143–147.

Имеется русский перевод: Вирт Н. Алгоритмы и структуры данных. СПб. [б.и.]: 2001. — 351 с.

Эта книга Вирта также содержит решения задачи пути шахматного коня и главу, посвященную компиляции и синтаксическому анализу.

Компиляция методом рекурсивного спуска относится к базовым принципам разработки компиляторов и детально рассматривается в большинстве хороших учебников по проектированию компиляторов. См., например,

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass., 1986.

Имеется русский перевод: Ахо. А., Сети Р., Ульман Д. Компиляторы. Принципы, технологии, инструменты. — М. и др.: Вильямс, 2001. — 767 с.

Много других приложений рекурсии можно найти в разных книгах, например,

E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, 1978, 626 pp.

Эта книга содержит более подробное обсуждение и анализ деревьев игр и программ прогнозирования.

Общая теория рекурсии является предметом теоретических исследований. Доступное изложение метода рекурсии в теоретическом плане можно найти в книге

R. S. Bird, *Programs and Machines*, John Wiley, New York, 1976.

# Глава 5

## Очереди

---

Очередь представляет собой структуру данных, напоминающую очередь людей, ждущих обслуживания. Вместе со стеками очереди относятся к простейшим типам структурных данных. В настоящей главе рассматриваются свойства очередей, изучается работа с ними и приводятся примеры различных приложений очередей.

### 5.1. Определения

На обычном человеческом языке очередь обозначает линию стоящих друг за другом людей, например, желающих приобрести билет, причем первый человек в этой линии будет обслужен первым. В компьютерных приложениях мы определяем *очередь* как список, в котором все добавления делаются с одного конца, а все удаления с другого. Очереди подчиняются дисциплине обслуживания *первым вошел, первым вышел*, (first in, first out) или коротко FIFO. См. рис. 5.1.

Приложения очередей даже более распространены, чем приложения стеков, поскольку при решении компьютером своих задач, как и в повсе-

приложения



Рис. 5.1. Очередь

дневной жизни, часто оказывается необходимым ждать своей очереди перед получением доступа к чему-то. В компьютере могут существовать очереди задач, ожидающих освобождения принтера, доступа к дисковой памяти или даже, в системах разделения времени, получения времени процессора. В рамках одной программы могут существовать множественные запросы, которые приходится сохранять в очереди, или одна задача может породить другие, выполняемые друг за другом, и образующих, соответственно, очередь задач.

Элемент очереди, готовый к тому, чтобы его обслужили, т. е. первый элемент, который будет извлечен из очереди, называется *головным элементом* или просто *головой* очереди. Аналогично, последний элемент в очереди, т. е. элемент, который был добавлен в очередь последним, мы называем *хвостовым элементом* или просто *хвостом* очереди.

Для полного определения очереди мы должны задать все операции, разрешенные для этой очереди. С этой целью мы укажем для каждой операции имя процедуры или функции вместе с пред- и постусловиями, образующими ее спецификации. Читая эти спецификации, вы можете заметить сходство с соответствующими операциями над стеком.

Первым шагом при работе с любой очередью будет вызов процедуры CreateQueue для инициализации очереди с целью ее дальнейшего использования:

**procedure** CreateQueue (**var** Q: queue); { Создать очередь }  
*предусловие:* Отсутствует.  
*постусловие:* Очередь Q создана и инициализирована как пустая.

Далее идут операции для проверки состояния очереди.

**function** QueueEmpty (Q: queue): Boolean; { Очередь пуста? }  
*предусловие:* Очередь Q уже создана.  
*постусловие:* Функция возвращает true или false в зависимости от того, пуста ли очередь или нет.

**function** QueueFull (Q: queue): Boolean; { Очередь полна? }  
*предусловие:* Очередь Q уже создана.  
*постусловие:* Функция возвращает true или false в зависимости от того, полна ли очередь или нет.

Далее приведены фундаментальные операции над очередью.

**procedure** Append (x: queueentry; **var** Q: queue); { Добавить }  
*предусловие:* Очередь Q уже создана и не полна.  
*постусловие:* Элемент x сохранен в очереди в качестве ее последнего элемента.

**procedure** Serve (**var** x: queueentry; **var** Q: queue); { Обслужить }  
*предусловие:* Очередь Q уже создана и не пуста.  
*постусловие:* Первый элемент очереди извлекается из нее и возвращается в качестве значения x.

голова и хвост

операции

альтернативные  
имена

Имена *Append* (присоединить) и *Serve* (обслужить) используются для фундаментальных операций над очередью, чтобы четко обозначить смысл выполняемых операций и избежать путаницы с терминами, которые мы будем использовать для других типов данных. Однако для этих операций часто используются и другие имена, например *Insert* (вставить) или *Delete* (удалить) или парные слова *Enqueue* (поставить в очередь) и *Dequeue* (извлечь из очереди).

Как это следует из предусловий, ошибкой является попытка присоединить элемент к полной очереди или обслужить элемент пустой очереди. Если мы хотим написать корректные процедуры *Append* и *Serve*, они должны возвращать сообщения об ошибке при неправильном использовании. Объявления, однако, не гарантируют, что процедуры будут обнаруживать ошибки, и если они этого действительно не делают, их вызов может привести к ложным и непредсказуемым результатам. Таким образом, аккуратный программист должен при любом вызове подпрограммы удостовериться, что ее предусловия гарантированно удовлетворяются.

Нам осталось привести еще четыре операции над очередями, иногда оказывающиеся полезными.

<b>function</b> QueueSize (Q: queue): integer;	{ Размер очереди }
<i>предусловие:</i> Очередь Q уже создана.	
<i>постусловие:</i> Функция возвращает количество элементов в очереди Q.	

<b>procedure</b> ClearQueue (var Q: queue);	{ Очистить очередь }
<i>предусловие:</i> Очередь Q уже создана.	
<i>постусловие:</i> Из очереди Q удаляются все ее элементы, и она становится пустой.	

<b>procedure</b> QueueFront (var x: queueentry; Q: queue);	{ Голова очереди }
<i>предусловие:</i> Очередь Q уже создана и не пуста.	
<i>постусловие:</i> Переменная x является копией первого элемента в очереди Q; сама очередь Q остается без изменения.	

Последняя операция не является частью строгого определения очереди, однако она весьма полезна для отладки и демонстрации.

<b>procedure</b> TraverseQueue (Q: queue; <b>procedure</b> Visit(var x: queueentry));	{ Просмотр очереди }
<i>предусловие:</i> Очередь Q уже создана.	
<i>постусловие:</i> Процедура Visit(var x: queueentry) была выполнена над всеми элементами очереди Q, начиная от головного элемента и далее вплоть до хвостового элемента	

## Упражнения 5.1

**E1.** Пусть Q представляет собой очередь, содержащую символы, а x, y и z — символьные переменные. Покажите содержимое Q на каждом шаге следующих сегментов кода:

<b>(a)</b> CreateQueue(Q);	<b>(b)</b> CreateQueue(Q);	<b>(c)</b> CreateQueue(Q);
Append('a', Q);	Append('a', Q);	Append('a', Q);
Serve(x, Q);	Append('b', Q);	x := 'b';
Append('b', Q);	Serve(x, Q);	Append('x', Q);
Serve(y, Q);	Append('c', Q);	Serve(y, Q);
Append('c', Q);	Append(x, Q);	Append(x, Q);
Append('d', Q);	Serve(y, Q);	Serve(z, Q);
Serve(z, Q);	Serve(z, Q);	Append(y, Q);

финансовые  
операции

**Е2.** Предположим, что вы являетесь финансистом и покупаете 100 акций в акционерном капитале компании *X* в каждом январе, апреле и сентябре и продаете 100 долей в каждом июне и ноябре. Цена одной акции в эти месяцы составляет

Январь	Апрель	Июнь	Сентябрь	Ноябрь
\$10	\$30	\$20	\$50	\$30

Определите свою полную прибыль или убыток, используя **(a)** учет с помощью дисциплины FIFO (первым вошел, первым вышел) и **(b)** дисциплины LIFO (последним вошел, первым вышел). Другими словами, вы храните свои сертификаты **(a)** в очереди и **(b)** в стеке. Пусть 100 акций, которые у вас останутся в конце каждого года, не входят в расчет.

**Е3.** Используя процедуры, разработанные в тексте, напишите другие процедуры, которые будут выполнять перечисленные ниже задачи. При написании каждой процедуры обязательно проверяйте условия пустоты или заполненности структур данных.

- (a)** Перемещение всех элементов из стека в очередь.
- (b)** Перемещение всех элементов из очереди в стек.
- (c)** Освободите один стек, перемещая его элементы на вершину другого стека таким образом, чтобы элементы, находившиеся в первом стеке, не изменили своего относительного порядка.
- (d)** Освободите один стек, перемещая его элементы на вершину другого стека таким образом, чтобы элементы, находившиеся в первом стеке, изменили свой порядок на обратный.
- (e)** Начните с очереди и пустого стека и используйте стек для изменения порядка всех элементов в очереди на обратный.
- (f)** Начните со стека и пустой очереди и используйте очередь для изменения порядка всех элементов стека на обратный.

## 5.2. Реализация очередей

Теперь, после того как мы рассмотрели определение очередей и допускаемые ими операции, подойдем к вопросу с другой стороны и обсудим, каким образом очереди могут быть организованы в компьютерной памяти и Pascal-процедурах.

### 1. Физическая модель

Как и в случае стеков, мы можем без труда создать очередь в компьютерной памяти, сохраняя элементы очереди в обычном массиве. Однако мы должны следить за головой и хвостом очереди. Это можно сделать

следующим образом. Будем всегда хранить головной элемент очереди в первом элементе массива. Тогда для добавления элемента к очереди достаточно увеличить значение счетчика, указывающего на хвостовой элемент, точно так же, как это делалось для стека. Однако для удаления элемента из очереди придется выполнить весьма значительную работу, поскольку после обслуживания первого элемента все остальные элементы надо будет переместить на одну позицию вверх для заполнения свободного места. Если очередь имеет значительную длину, эта операция займет много времени. Хотя такой способ организации памяти точно моделирует поведение очереди людей, ожидающих обслуживания, для использования в компьютерах он подходит плохо.

## 2. Линейная реализация

Для эффективной обработки очереди мы должны иметь два индекса, чтобы можно было следить за обоими концами очереди — и головой, и хвостом, не прибегая к перемещению каких-либо элементов. Для добавления элемента к очереди мы просто увеличиваем значение хвостового индекса на 1 и помещаем элемент в эту позицию. При обслуживании очереди мы забираем элемент в голове очереди и увеличиваем значение головного индекса на 1. Этот метод, однако, имеет существенный недостаток. И головной, и хвостовой индексы всегда увеличиваются, но никогда не уменьшаются. Даже если в любой момент в очереди содержится не более двух элементов, для размещения очереди в памяти потребуется неограниченный объем памяти, если выполняется длинная последовательность операций такого типа:

Append, Append, Serve, Append, Serve, Append, ...

Проблема здесь заключается в том, что по мере того, как очередь перемещается вниз по массиву, пространство памяти в начале массива отбрасывается и никогда не используется снова. Такую структуру данных можно уподобить змее, ползущей по памяти компьютера. Иногда змея оказывается длиннее, иногда короче, но если она будет всегда ползти в одном направлении, она рано или поздно достигнет конца пространства памяти.

Заметьте, однако, что для приложений, в которых очередь периодически очищается (например, если приходящие запросы накапливаются до достижения некоторого их количества, а затем активизируется задача, которая перед возвратом очищает все запросы), в тот момент, когда очередь оказывается пустой, голова и хвост очереди могут быть переустановлены на начало массива, и наша простая схема с двумя индексами и прямолинейным запоминанием элементов становится весьма эффективной реализацией.

## 3. Кольцевые массивы

В принципе мы можем преодолеть неэффективное использование памяти, если рассматривать массив как кольцо, а не как прямую линию. См. рис. 5.2. В этом случае по мере того, как элементы добавляются в очередь и удаляются из нее, голова очереди непрерывно преследует хвост,

дефект

преимущество

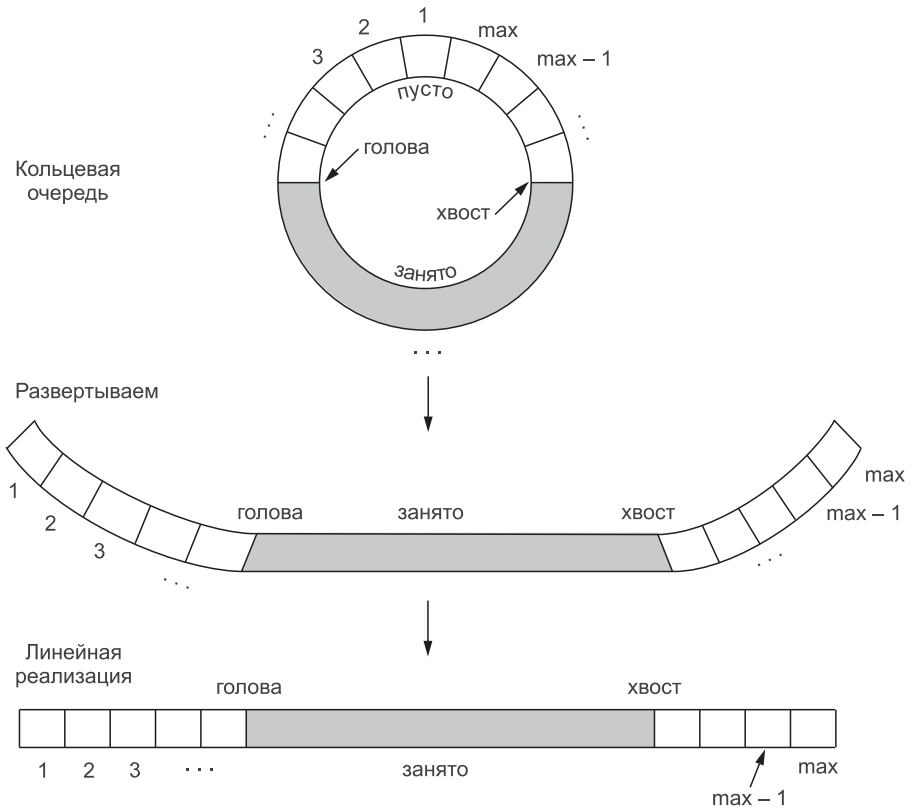


Рис. 5.2. Очередь в кольцевом массиве

перемещаясь по кольцу массива, так что змея может ползти неопределенно долго, оставаясь в ограниченном пространстве кольца. В различные моменты времени очередь будет занимать разные части массива, но нам никогда не надо будет бояться ее выхода из отведенного под массив пространства памяти, за исключением случаев полного расходования памяти массива, когда мы столкнемся с ситуацией истинного переполнения.

#### 4. Реализация кольцевых массивов

Нашей следующей задачей будет реализация кольцевого массива в виде обычного (т. е. линейного) массива. Представим себе, что позиции по кругу кольца пронумерованы от 1 до  $\text{max}$ , где  $\text{max}$  — это полное число элементов в кольцевом массиве, и для реализации кольцевого массива мы используем так же пронумерованные элементы линейного массива. Тогда перемещение индексов подчиняется правилам операций по модулю: когда значение индекса становится большим  $\text{max}$ , мы начинаем снова с 1. Это схоже с выполнением арифметических действий с помощью



круглого циферблата часов; часы пронумерованы от 1 до 12, и если мы добавляем четыре часа к десяти часам, в сумме получается 2 часа.

Возможно, наглядной житейской аналогией этого линейного представления является священник, причащающий прихожан, стоящих на коленях перед церковью. Причащающиеся не двигаются до тех пор, пока священник не пройдет весь ряд и не причастит всех прихожан. Когда священник дойдет до конца переднего ряда, он возвращается назад и начинает обряд причащения снова, поскольку к этому времени вперед продвинулся новый ряд прихожан.

## 5. Кольцевые массивы в языке Pascal

В языке Pascal мы можем увеличивать индекс  $i$  в кольцевом массиве на 1 с помощью такого предложения

```
if i = max then i:= 1 else i := i + 1;
```

или даже более просто (но возможно, менее эффективно в плане времени выполнения, поскольку здесь имеет место операция деления) с помощью оператора **mod**:

```
i := (i mod max) + 1;
```

Более естественно выразить эту последнюю операцию, индексируя массив от 0 до  $\text{max} - 1$ , чтобы сложение выполнялось перед операций **mod**:

```
i := (i + 1) mod max;
```

(Можете проверить, что результат последнего выражения всегда лежит в пределах от 0 до  $\text{max} - 1$ , в то время как в предшествующем выражении результат изменяется от 1 до  $\text{max}$ .) Индексация от 0 оказывается даже еще более естественной, если индекс  $i$  необходимо увеличивать на произвольную величину  $k > 0$ . Если мы индексируем массив от 0 до  $\text{max} - 1$ , мы просто пишем

```
i:=(i + k) mod max
```

в то время как при индексации от 1 до  $\text{max}$  нам придется написать:

```
i:=(i + k - 1) mod max + 1
```

К сожалению, в языке Pascal при индексировании от 0 мы вынуждены объявлять две константы, и  $\text{max}$ , и другую, равную  $\text{max} - 1$ , и это иногда приводит к путанице. Поэтому для простоты мы, реализуя очередь в виде массива, оставляем традиционное индексирование от 1.

## 6. Граничные условия

Перед тем, как приступить к написанию формального алгоритма добавления и удаления элементов очереди, рассмотрим граничные условия, т. е. ситуации, когда очередь пуста или полна. Если в очереди имеется в точности один элемент, тогда головной индекс будет равен хвостовому. Если этот единственный элемент удалить, головной индекс увеличится на 1, поэтому пустая очередь характеризуется таким состоянием индексов, при котором хвостовой индекс указывает на ближайшую позицию перед головным. Теперь предположим, что очередь почти полна. В про-

пустая  
или полная?

цессе заполнения очереди хвостовой индекс перемещался в направлении от головного по кольцу, и когда очередь заполнится, хвостовой индекс будет указывать в точности на ближайшую позицию перед головным. Таким образом, у нас возникает другая проблема: головной и хвостовой индексы и для пустой, и для полной очереди находятся в одних и тех же относительных позициях! Рассматривая только лишь индексы, невозможно отличить полную очередь от пустой. Эта ситуация проиллюстрирована на рис. 5.3.

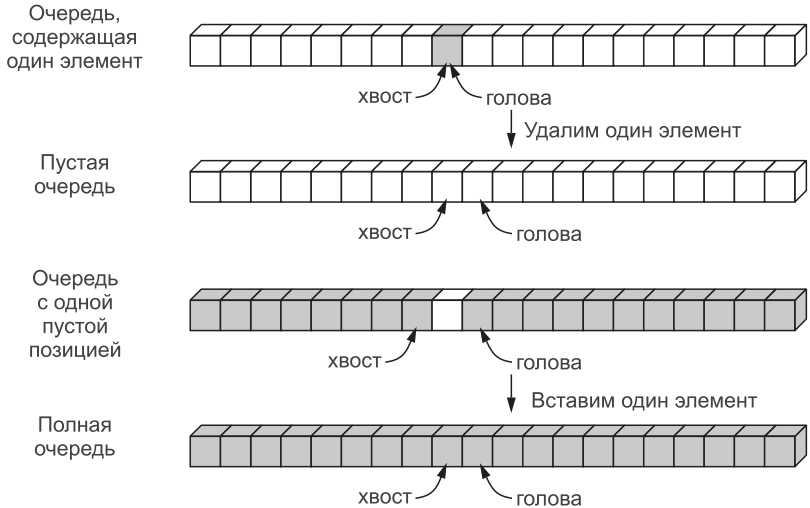


Рис. 5.3. Пустая и полная очереди

## 7. Возможные решения

Можно предложить по меньшей мере три существенно разных способа решить эту проблему. Один заключается в оставлении в массиве одной пустой позиции; в этом случае очередь считается полной, когда хвостовой индекс находится за две позиции до фронтového. Второй способ требует введения новой переменной. Это может быть булева переменная, устанавливаемая, когда хвост подходит к самой голове, и характеризующая тем самым ситуацию, когда очередь полна (булева переменная, характеризующая пустоту очереди, также вполне годится), или целочисленная переменная, которая отсчитывает число элементов в очереди. Третий способ индикации полноты (или пустоты) очереди заключается в присваивании одному или обоим индексам некоторого значения или значений, которые сами по себе никогда реализоваться не могут. Если, например, элементы массива индексируются от 1 до  $\max$ , тогда на пустую очередь может указывать значение хвостового индекса, равное 0.

## 8. Обзор реализаций

Чтобы подытожить обсуждение вопроса об очередях, перечислим все рассмотренные нами методы реализации очередей.

1. пустая позиция

2. флаг

3. специальные значения

- Физическая модель: линейный массив, головной элемент которого всегда занимает первую позицию, и все элементы перемещаются по массиву при каждом удалении головного элемента. Как правило, это неэффективный метод организации очереди в компьютере.
- Линейный массив с двумя индексами, которые во всех случаях увеличивают свои значения. Это неплохой метод, если очередь может быть очищена одномоментно.
- Кольцевой массив с головным и хвостовым индексами и одной пустой позицией.
- Кольцевой массив с головным и хвостовым индексами и булевой переменной, индицирующей заполненность (или пустоту) очереди.
- Кольцевой массив с головным и хвостовым индексами и целочисленной переменной, отсчитывающей число элементов.
- Кольцевой массив с головным и хвостовым индексами, принимающими особое значение для индикации пустоты очереди.

Позже в этой главе мы рассмотрим еще один способ реализации очередей с помощью связанной структуры. Самый главный вывод, который следует сделать из приведенного списка вариантов реализации, заключается в том, что мы всегда должны отделять вопросы использования различных структур данных, например, очередей, от вопросов их реализации; и в программировании мы должны всегда рассматривать на каждом этапе проектирования только одну из этих категорий вопросов. После того, как мы решим, как именно очереди будут использованы в нашем приложении, и после того, как мы напишем процедуры, использующие эти очереди, мы получим дополнительную информацию, которая поможет нам выбрать наилучшую реализацию очередей для конкретного приложения.

откладывайте  
решения  
относительно  
реализации

#### ***Программистский принцип***

*Практикуйте сокрытие информации: разделяйте прикладное использование структур данных и их реализацию*

### **5.3. Кольцевые очереди в языке Pascal**

Теперь давайте напишем формальные Pascal-процедуры для реализации очереди. Из последнего раздела должно быть ясно, что существует большое разнообразие реализаций, из которых некоторые представляются просто небольшими вариациями других. Поэтому сконцентрируем наше внимание лишь на одной реализации, оставив остальные для упражнений.

Реализация в виде кольцевого массива, который использует счетчик для отслеживания количества элементов в очереди, иллюстрирует технику работы с кольцевыми массивами и, к тому же, упрощает программирование некоторых операций. Поэтому мы будем работать только с этой реализацией.

Пусть очередь хранится в массиве, индексируемом в диапазоне  
1..maxqueue

и хранящем элементы типа, называемого `queueentry`. Переменные `front` и `rear` будут указывать на головную (`front`) и хвостовую (`rear`) позиции в массиве. Переменная `count` используется для отслеживания числа элементов в очереди. Объявление записи для очереди примет в этом случае такой вид:

тип очередь

```
type queue = record
  count,
  front,
  rear: 0..maxqueue;
  entry: array [1..maxqueue] of queueentry;
end;
```

Первое, что мы должны сделать — это разработать процедуру для инициализации пустой очереди.

## инициализация

[illegible]

Процедуры для добавления и удаления элементов очереди точно соответствуют проведенным ранее рассуждениям. Заметьте, что мы защищаемся от нарушения предусловий с активизацией в необходимых случаях процедуры `Error` обработки ошибок.

включение

[illegible]

```

procedure Serve (x: queueentry; var Q: queue);      { Извлечь из очереди }
{ Pre: Очередь Q уже создана и не полна.
  Post: Первый элемент в очереди был удален и возвращен в качестве
        значения x.
  Uses: Использует функцию QueueEmpty, процедуру Error. }
begin                                              { процедура Serve }
  with Q do
    if QueueEmpty(Q) then
      Error('Невозможно извлечь элемент из пустой очереди.')
    else begin
      count := count - 1;
      x := entry [front];
      front := (front mod maxqueue) + 1
    end
  end;                                              { процедура Serve }

```

извлечение

Три функции, касающиеся размера очереди, в нашей реализации пишутся без труда.

```

function QueueSize (var Q: queue): integer;      { Размер очереди }
{ Pre: Очередь Q уже создана.
  Post: Функция возвращает число элементов в очереди Q.
begin
  QueueSize := Q.Count
end;                                              { функция QueueSize }

function QueueEmpty (var Q: queue): Boolean;      { Очередь пуста? }
{ Pre: Очередь Q уже создана.
  Post: Функция возвращает true, если очередь Q пуста, и false
        в противном случае.
begin
  QueueEmpty := (Q.Count = 0)
end;                                              { функция QueueEmpty }

function QueueFull (var Q: queue): Boolean;      { Очередь полна? }
{ Pre: Очередь Q уже создана.
  Post: Функция возвращает true, если очередь Q полна, и false
        в противном случае.
begin
  QueueFull := (Q.Count = maxqueue)
end;                                              { функция QueueFull }

```

размер

очередь пуста?

очередь полна?

Заметьте, что очередь в каждой из этих функций указывается в виде параметра **var**, хотя ни в одной из этих функций она не модифицируется. Это дань повышению эффективности программы: в этом случае экономится время для создания локальной копии всей очереди при каждом вызове функции.

Процедуры ClearQueue, QueueFront и TraverseQueue мы оставим для упражнений.

### Упражнения 5.3

**E1.** Напишите оставшиеся процедуры для очередей в соответствии с способом реализации, предложенным в настоящем разделе.

(a) ClearQueue      (b) QueueFront      (c) TraverseQueue

- Е2.** Напишите процедуры и функции, необходимые для реализации очереди в виде линейного массива для случая, когда очередь при необходимости может целиком очищаться. Напишите процедуру `Append`, которая будет добавлять элемент в очередь при наличии в ней места, если же места нет, будет вызывать другую процедуру (`ServeAll`), которая будет очищать очередь. Разрабатывая эту вторую процедуру, можете принять существование вспомогательной процедуры `Service(x: queueentry)`, которая будет обрабатывать один элемент, только что удаленный вами из очереди.
- Е3.** Напишите на языке Pascal процедуры и функции, реализующие очередь простейшим, хотя и медленным методом, когда голова очереди всегда находится в первой позиции линейного массива.
- Е4.** Напишите на языке Pascal процедуры и функции, реализующие очередь с помощью линейного массива с двумя индексами, `front` (голова) и `rear` (хвост), и с условием, что при достижении индексом `rear` конца массива все элементы очереди сдвигаются к голове массива.
- Е5.** Напишите процедуры и функции для обработки очереди в реализации, которая не содержит числа элементов в очереди, но использует специальные условия

`rear = 0` и `front = 1`

для индикации опустошения очереди.

- |                              |                            |                                |
|------------------------------|----------------------------|--------------------------------|
| (a) <code>CreateQueue</code> | (d) <code>QueueFull</code> | (g) <code>Serve</code>         |
| (b) <code>ClearQueue</code>  | (e) <code>QueueSize</code> | (h) <code>QueueFront</code>    |
| (c) <code>QueueEmpty</code>  | (f) <code>Append</code>    | (i) <code>TraverseQueue</code> |

- Е6.** Перепишите набор Pascal-процедур для обработки очереди, приведенных в тексте, используя булеву переменную `full` вместо счетчика числа элементов в очереди.
- Е7.** Напишите на языке Pascal процедуры и функции, реализующие очередь в кольцевом массиве с одним неиспользуемым элементом. Т. е. считайте массив заполненным, когда его хвост находится за две позиции перед головой; когда хвост находится за одну позицию перед головой, считайте очередь пустой.

Термин *deque* (произносится «дек» или «дикью») является сокращением от *double-ended queue* и обозначает очередь с двусторонним доступом или, короче, двустороннюю очередь. Такая очередь представляет собой список, элементы которого могут добавляться и удаляться как с первой, так и с последней позиций списка, хотя изменения не могут касаться никаких внутренних элементов. Таким образом, дек является обобщением и стека, и очереди.

- Е8.** Напишите процедуры и функции, необходимые для реализации дека в виде линейного массива.
- Е9.** Напишите процедуры и функции, необходимые для реализации дека в виде кольцевого массива.
- Е10.** Как лучше реализовывать дек, в виде линейного массива или в виде кольцевого массива? Почему?

- E11.** Напишите демонстрационную программу, управляемую меню, для обслуживания дека символов, аналогичную демонстрационной программе для стека из упражнения E1(c) раздела 3.1.
- E12.** Из рис. 3.5 видно, что стек можно образно представить как подъездную ветку, ответвляющуюся от прямого железнодорожного пути. Очередь, разумеется, напоминает просто прямой железнодорожный путь. Разработайте и нарисуйте железнодорожный маневровый путь, который будет олицетворять дек. Маневровый путь должен иметь только один въезд и один выезд.
- E13.** Пусть элементы данных с номерами 1, 2, 3, 4, 5, 6 поступают во входном потоке именно в таком порядке. Другими словами, сначала приходит элемент 1, за ним элемент 2 и т. д. При использовании (1) очереди и (2) дека, какую из приведенных ниже перегруппировок можно получить в выходном потоке? Элементы покидают дек в порядке слева направо.
- |                        |                        |
|------------------------|------------------------|
| <b>(a)</b> 1 2 3 4 5 6 | <b>(d)</b> 4 2 1 3 5 6 |
| <b>(b)</b> 2 4 3 6 5 1 | <b>(e)</b> 1 2 6 4 5 3 |
| <b>(c)</b> 1 5 2 4 3 6 | <b>(f)</b> 5 2 6 3 4 1 |

### Программные проекты 5.3

- P1.** Напишите демонстрационную программу для обслуживания очередей. Программа должна быть схожа с той, что была написана для демонстрации стеков в упражнении E1(c) раздела 3.1 или простого списка в разделе 2.3.3. Элементами вашей очереди должны быть символы. Ваша демонстрационная программа должна выводить меню, из которого пользователь выбирает нужную ему операцию над очередью. После того как программа выполнит запрошенную операцию, она должна информировать пользователя о ее результате и запрашивать следующую операцию. Если пользователь желает добавить символ в очередь, программа должна запрашивать этот символ.
- P2.** Напишите процедуру, которая читает одну строку входных данных с терминала. Предполагается, что эта строка содержит две части, разделенные двоеточием ':'. В качестве результата своей работы процедура должна выводить один символ согласно следующим правилам:
- N На строке нет точки с запятой
  - L Левая часть строки (перед символом точки с запятой) длиннее правой.
  - R Правая часть строки (после символа точки с запятой) длиннее левой.
  - D Левая и правая части имеют одну длину, но различаются.
  - S Левая и правая части абсолютно тождественны.

*Примеры:*

Ввод	Вывод
Sample Sample	N
Short:Long	L
Sample:Sample	S

Для хранения левой части строки в процессе чтения правой части используйте очередь.

## 5.4. Приложения очередей: Моделирование

### 5.4.1. Введение

*Моделированием*, или *имитацией* называется использование одной системы для имитации поведения другой системы. Моделирование часто используется в тех случаях, когда экспериментирование с реальной системой было бы слишком дорогим или опасным. Физические системы моделирования, например, аэродинамические трубы, используются для экспериментов с кузовами автомобилей, а имитаторы полета применяются для тренировки летчиков гражданской и военной авиации. Математические модели представляют собой системы уравнений, описывающие некоторые системы, а компьютерное моделирование использует программные шаги для имитации поведения изучаемой системы.

компьютерное  
моделирование

В случае компьютерного моделирования изучаемые объекты обычно представляются данными, часто структурами данных вроде записей, элементы которых описывают свойства объектов. Изучаемые действия представляются операциями над данными, и правила, описывающие эти действия, преобразуются в компьютерные алгоритмы. Изменяя значения данных или модифицируя эти алгоритмы, мы можем наблюдать изменения в компьютерной модели, после чего сделать ценные заключения о поведении реальной системы.

Пока один объект в системе вовлечен в некоторые действия, другие объекты и действия часто вынуждены находиться в состоянии ожидания. Поэтому очереди относятся к важным структурам данных в программах компьютерного моделирования. Мы рассмотрим один из наиболее распространенных и полезных видов компьютерного моделирования, в котором очереди являются базовыми структурами данных. Такие системы компьютерного моделирования имитируют поведение систем (часто называемых *системами массового обслуживания*), в которых имеются очереди объектов, ожидающих обслуживания теми или иными процессами.

### 5.4.2. Моделирование работы аэропорта

В качестве конкретного примера рассмотрим небольшой, но активно используемый аэропорт с одной взлетно-посадочной полосой (рис. 5.4). В каждый интервал времени только один самолет может приземляться или только один самолет может взлетать, но не то и другое вместе. Самолеты, готовые к посадке или взлету, поступают в случайные моменты времени, так что в некоторый заданный промежуток времени взлетно-посадочная полоса может быть пустой, или самолет может садиться или взлетать, и, кроме того, несколько самолетов могут ожидать разрешения как на посадку, так и на взлет. Таким образом, нам понадобятся две очереди, которые мы назовем *landing* (посадка) и *takeoff* (взлет), для хранения этих самолетов. Лучше, конечно, держать самолет в состоянии ожидания на земле, чем в воздухе, поэтому небольшой аэропорт позволяет самолетам взлетать, только если нет самолетов, ожидающих посадки. Таким обра-

правила





Рис. 5.4. Аэропорт

зом, после получения запросов от новых самолетов на посадку или взлет, наша модель должна сначала обслужить головной элемент очереди самолетов, ожидающих посадки, и только если очередь на посадку пуста, разрешить самолетам взлетать. Мы будем выполнять моделирование в течение многих интервалов времени, и поэтому главное действие программы мы поместим в цикл, который будет отсчитывать интервалы времени в переменной *curtime* (от *current time*, текущее время) от 1 до значения переменной *endtime* (конечное время). При таких обозначениях мы можем теперь написать эскиз главной программы.

первый эскиз

```

program Airport(input, output);                                { Аэропорт }
var curplane: plane; { текущий самолет (т. е. самолет, обрабатываемый
в настоящий момент) }
    landing, takeoff: queue;                                     { очереди ожидающих самолетов }
    curtime, endtime,                                           { время начала и конца моделирования }
    counter: integer;                                           { переменная управления циклом }
begin                                                           { главная программа Airport }
    CreateQueue(landing);                                       { Установим очереди в пустое состояние }
    CreateQueue(takeoff);
    for curtime := 1 to endtime do begin                        { начнем главный цикл по интервалам времени }
        for counter := 1 to RandomNumber do begin              { добавляем самолеты в очередь на посадку }
            NewPlane(curplane);                                  { получим данные о новом самолете }
            if QueueFull(landing) then
                { включаем обработку ошибки на заполнение очереди }
                Refuse(curplane)
            else Append(curplane, landing)
                { помещаем каждый новый самолет в очередь }
        end;
        for counter := 1 to RandomNumber do begin              { добавляем самолеты в очередь на взлет }
            NewPlane(curplane);

```

новый самолет  
готов к посадкеновый самолет  
готов к взлету

	<pre>         if QueueFull(takeoff) then             Refuse(curplane)         else Append(curplane, takeoff)         end;         if not QueueEmpty(landing) then begin             { если самолет ожидает посадки, разрешим ему посадку }             Serve(curplane, takeoff);             Land(curplane)           { обработка приземляющегося самолета }         end         else if not QueueEmpty(ltakeoff) then begin           { взлет разрешается                                                                 только если нет самолетов в очереди на посадку }             Serve(curplane, takeoff);             Fly(curplane)           { обработка взлетающего самолета }         end         else             Idle { взлетно-посадочная полоса пуста; не надо делать ничего }         end;         Conclude { завершить моделирование }     end.          { главная программа Airport } </pre>
самолет садится	
самолет взлетает	
полоса простаивает	

### 5.4.3. Случайные числа

Ключевым шагом в нашем моделировании будет решение, в каждый интервал времени, сколько новых самолетов готовы к посадке или взлету. Хотя к этому решению можно прийти разными способами, наиболее интересным и полезным будет принятие случайного решения. Когда программа прогоняется повторно со случайными решениями (т. е. выполняется *стохастическое* моделирование), результаты будут отличаться от прогона к прогону, и при достаточно длительном экспериментировании моделирование может дать результаты, разброс которых отвечает поведению реальной изучаемой системы. Функция `RandomNumber` из предыдущего эскиза возвращает случайное число самолетов, ожидающих посадки или готовых к взлету в конкретный интервал времени.

Приложение В посвящено операциям с числами, называемыми **псевдослучайными числами**, и широко используемыми в компьютерных программах. Для различных приложений удобно использовать псевдослучайные числа разного рода. Для имитации работы аэропорта нам нужны весьма специальные случайные числа, которые называются *Пуассоновскими* случайными числами или, точнее, числами, распределенными по закону Пуассона.

Чтобы дать представление об идее таких чисел, прежде всего заметим, что если средняя семья имеет 2.6 ребенка, это не значит, что в каждой семье растут 2 целых ребенка и еще 0.6 третьего. Говоря о 2.6 детей, мы понимаем под этим, что при усреднении по большому количеству семей среднее число детей окажется равным 2.6. Так, если пять семей имеют по 4, 1, 0, 3 и 5 детей, то средний размер семьи будет 2.6. Аналогично, если число самолетов, прибывающих для посадки, за десять интервалов времени составит числа 2, 0, 0, 1, 4, 1, 0, 0, 0, 1, то среднее число прибывающих за один временной интервал самолетов составит величину 0.9.

псевдослучайные  
числа

математическое  
ожидание

распределение  
Пуассона

Введем фиксированное число, называемое *математическим ожиданием*  $n$  последовательности случайных чисел. Тогда, если последовательность неотрицательных целых чисел подчиняется *распределению Пуассона* с математическим ожиданием  $n$ , то в достаточно длинных последовательностях среднее значение целых чисел в последовательности стремится к значению  $n$ . В приложении В описаны функции, генерирующие случайные целые числа, подчиняющиеся распределению Пуассона с данным математическим ожиданием, а это как раз то, что нам нужно для моделирования работы аэропорта.

#### 5.4.4. Главная программа

Предыдущий эскиз показывает использование очередей для моделирования работы аэропорта, но нам нужны большие детали для прослеживания всех интересующих нас статистических характеристик процессов, происходящих в аэропорту, таких как число обслуженных самолетов, среднее время ожидания и число самолетов (если таковые обнаружатся), которым было отказано в обслуживании. Эти детали отражаются в объявлениях констант, типов и переменных, которые требуется включить в главную программу. Далее нам нужно будет написать подпрограммы для задания способов обработки этой информации.

объявления

```
uses PlaneQueue, { Модуль обслуживания очередей с queueentry = plane. }
RandomGe;        { Модуль генератора случайных чисел. }
type action = (arrive, depart); { Что желает делать этот самолет? }
var landing, takeoff: queue;
    curplane: plane;
    curtime,      { текущее время; один интервал времени = время,
                   требуемое для взлета или посадки самолета }
    nplanes,      { число обслуженных к текущему времени самолетов }
    endtime,      { полное число рассматриваемых интервалов времени }
    idletime,     { число интервалов времени, когда
                   взлетно-посадочная полоса свободна }
    nland, ntakeoff, nrefuse, { число самолетов приземлившихся,
                               взлетевших и тех, которым отказано в обслуживании }
    landwait, takeoffwait, { полное время ожидания для приземляющихся
                             и взлетающих самолетов }
    counter: integer; { переменная управления циклом }
    expectarrive, expectdepart: real; { среднее число самолетов,
                                       требующих взлета и посадки за один интервал времени }
```

статистика  
моделирования

Работоспособный вариант главной программы мало отличается от предыдущего эскиза, если не считать включения многих параметров, используемых для обновления всех только что объявленных переменных:

```
program Airport (input, output); { Аэропорт }
{ Pre: Пользователь должен предоставить число интервалов времени
        для данного прогона модели, а также ожидаемое число
        прибывающих самолетов и ожидаемое число взлетающих
        самолетов в среднем за один временной интервал.
  Post: Программа выполняет стохастическое моделирование работы
        аэропорта, показывая состояние взлетно-посадочной полосы
        в каждом временном интервале. Перед завершением программа
        выводит на экран итоговую сводку.
```

**Uses:** *Использует пакты обслуживания очередей и генерации случайных чисел, процедуры Start, NewPlane, Refuse, Land, Fly, Idle и Conclude. }*

*{ Объявления констант, типов и переменных следует вставить сюда. }*  
**begin** *{ главная программа Airport }*

инициализация

```
CreateQueue(landing);
CreateQueue(takeoff);
Start(endtime, nplanes, nland, ntakeoff, nrefuse, landwait, takeoffwait,
      idletime, expectarrive, expectdepart);
```

новый самолет(ы)  
готов к посадке

```
for curtime := 1 to endtime do begin
  for counter :=1 to PoissonRandom(expectarrive) do begin
    NewPlane(curplane, nplanes, curtime, arrive);
    if QueueFull(landing) then
      Refuse(curplane, arrive, nrefuse)
    else
      Append(curplane, landing)
  end;
```

новый самолет(ы)  
готов к взлету

```
  for counter :=1 to PoissonRandom(expectdepart) do begin
    NewPlane(curplane, nplanes, curtime, depart);
    if QueueFull(takeoff) then
      Refuse(curplane, depart, nrefuse)
    else
      Append(curplane, takeoff)
  end;
```

самолет садится

```
  if not QueueEmpty(landing) then begin { разрешаем посадку }
    Serve(curplane, takeoff);
    Land(curplane, curtime, nland, landwait)
  end
```

самолет взлетает

```
  else if not QueueEmpty(takeoff) then begin { разрешаем взлет }
    Serve(curplane, takeoff);
    Fly(curplane, curtime, ntakeoff, takeoffwait)
  end
```

полоса  
простаивает

```
  else
    Idle(curtime, idletime);
end;
```

завершение  
моделирования

```
  Conclude(nplanes, nland, ntakeoff, nrefuse, landwait, takeoffwait,
            idletime, takeoff, landing);
```

**end.** *{ главная программа Airport }*

## 5.4.5. Шаги моделирования

Действия процедур, выполняющих последовательные шаги процесса моделирования достаточно прямолинейны, так что мы начнем писать их по очереди, включая комментарии лишь там, где они действительно необходимы.

### 1. Инициализация

```
procedure Start (var endtime, nplanes, nland, ntakeoff, nrefuse, { Старт }
                 landwait, takeoffwait, idletime: integer;
                 var expectarrive, expectdepart: real);
```

**{ Pre:** *Предусловия отсутствуют.*

**Post:** *Запрашивает у пользователя данные и инициализирует все переменные, специфицированные как параметры. }*





## 7. Завершение моделирования

```

procedure Conclude (nplanes, nland, ntakeoff, nrefuse, landwait, takeoffwait,
                    idletime: integer; var takeoff, landing: queue);
                                { Завершение }
{ Pre:   Предусловия отсутствуют.
  Post:  Выводит на экран всю статистику и завершает
          моделирование. }

begin                                { процедура Conclude }
  writeln('Моделирование завершилось после ', endtime: 4, ' интервалов'.);
  writeln('Полное число обслуженных самолетов: ', nplanes: 4);
  writeln(' Число приземлившихся самолетов: ', nland: 4);
  writeln(' Число взлетевших самолетов: ', ntakeoff: 4);
  writeln(' Число самолетов, получивших отказ: ', nrefuse: 4);
  writeln(' Число оставшихся самолетов на посадку: ',
    QueueSize(landing): 4);
  writeln(' Число оставшихся самолетов на взлет: ',
    QueueSize(takeoff): 4);
  if endtime > 0 then
    writeln('Процент времени, когда полоса свободна: ',
      (idletime / endtime) * 100: 7: 2);
  if nland > 0 then
    writeln('Среднее время ожидания посадки: ',
      (landwait / nland): 7: 2);
  if ntakeoff > 0 then
    writeln('Среднее время ожидания взлета: ',
      (takeoffwait / ntakeoff): 7: 2);
end;                                { процедура Conclude }

```

### 5.4.6. Пример результатов

Мы завершаем этот раздел выводом пробного прогона программы моделирования работы аэропорта. Вы можете заметить, что имеются отрезки времени, когда взлетно-посадочная полоса простаивает, и другие отрезки, когда одна или обе очереди полны, и в которых, следовательно, некоторые самолеты должны быть отклонены. Если вы снова запустите программу моделирования, вы получите результаты, отличающиеся от приведенных здесь, однако если средние значения, задаваемые программе, останутся теми же, вы увидите некоторое сходство в числах сводных данных обоих прогонов программы.

Эта программа моделирует работу аэропорта с единственной взлетно-посадочной полосой.

В каждом интервале времени сесть или взлететь может только один самолет.

В любом интервале времени до 5 самолетов могут ждать приземления или взлета.

Сколько интервалов времени будет длиться моделирование? 30

Ожидаемое среднее число прилетающих самолетов в одном временном интервале (действительное число)? 0.47

Ожидаемое среднее число отлетающих самолетов

в одном временном интервале (действительное число)? 0.47



обе очереди пусты	Самолет	1	готов к посадке.
	1: Самолет	1	приземлился; в очереди ждал 0 интервалов.
	2: Полоса		свободна.
	Самолет	2	готов к посадке.
	Самолет	3	готов к посадке.
	3: Самолет	2	приземлился; в очереди ждал 0 интервалов.
	4: Самолет	3	приземлился; в очереди ждал 1 интервалов.
	Самолет	4	готов к посадке.
	Самолет	5	готов к посадке.
	Самолет	6	готов к взлету.
очередь на посадку пуста	Самолет	7	готов к взлету.
	5: Самолет	4	приземлился; в очереди ждал 0 интервалов.
	Самолет	8	готов к взлету
	6: Самолет	5	приземлился; в очереди ждал 1 интервалов.
	Самолет	9	готов к взлету.
	Самолет	10	готов к взлету.
	7: Самолет	6	взлетел; в очереди ждал 2 интервалов.
	8: Самолет	7	взлетел; в очереди ждал 3 интервалов.
	9: Самолет	8	взлетел; в очереди ждал 3 интервалов.
	Самолет	11	готов к посадке
очередь на взлет полна	10: Самолет	11	приземлился; в очереди ждал 0 интервалов.
	Самолет	12	готов к взлету.
	11: Самолет	9	взлетел; в очереди ждал 4 интервалов.
	Самолет	13	готов к посадке.
	Самолет	14	готов к посадке.
	12: Самолет	13	приземлился; в очереди ждал 0 интервалов.
	13: Самолет	14	приземлился; в очереди ждал 1 интервалов.
	14: Самолет	10	взлетел; в очереди ждал 7 интервалов.
	Самолет	15	готов к посадке.
	Самолет	16	готов к взлету.
очередь на посадку полна	Самолет	17	готов к взлету.
	15: Самолет	15	приземлился; в очереди ждал 0 интервалов.
	Самолет	18	готов к посадке.
	Самолет	19	готов к посадке.
	Самолет	20	готов к взлету.
	Самолет	21	готов к взлету.
	16: Самолет	18	приземлился; в очереди ждал 0 интервалов.
	Самолет	22	готов к посадке.
	17: Самолет	19	приземлился; в очереди ждал 1 интервалов.
	Самолет	23	готов к взлету.
очередь на посадку полна	Самолету	23	приказано попробовать позже.
	18: Самолет	22	приземлился; в очереди ждал 1 интервалов.
	Самолет	24	готов к посадке.
	Самолет	25	готов к посадке.
	Самолет	26	готов к посадке.
	Самолет	27	готов к взлету.
	Самолету	27	приказано попробовать позже.
	19: Самолет	24	приземлился; в очереди ждал 0 интервалов.
	Самолет	28	готов к посадке.
	Самолет	29	готов к посадке.
очередь на посадку полна	Самолет	30	готов к посадке.
	Самолет	31	готов к посадке.
	Самолет	31	направлен в другой аэропорт.
	20: Самолет	25	приземлился; в очереди ждал 1 интервалов.
	Самолет	32	готов к посадке.



сводка

Самолет 33 готов к взлету.  
Самолету 33 приказано попробовать позже.  
21: Самолет 26 приземлился; в очереди ждал 2 интервалов.  
22: Самолет 28 приземлился; в очереди ждал 2 интервалов.  
23: Самолет 29 приземлился; в очереди ждал 3 интервалов.  
Самолет 34 готов к взлету.  
Самолету 34 приказано попробовать позже.  
24: Самолет 30 приземлился; в очереди ждал 4 интервалов.  
Самолет 35 готов к взлету.  
Самолету 35 приказано попробовать позже.  
Самолет 36 готов к взлету.  
Самолету 36 приказано попробовать позже.  
25: Самолет 32 приземлился; в очереди ждал 4 интервалов.  
Самолет 37 готов к взлету.  
Самолету 37 приказано попробовать позже.  
26: Самолет 12 взлетел; в очереди ждал 15 интервалов.  
27: Самолет 16 взлетел; в очереди ждал 12 интервалов.  
28: Самолет 17 взлетел; в очереди ждал 13 интервалов.  
29: Самолет 20 взлетел; в очереди ждал 13 интервалов.  
Самолет 38 готов к взлету.  
29: Самолет 20 взлетел; в очереди ждал 14 интервалов.  
Моделирование завершилось после 30 интервалов времени.  
Полное число обслуженных самолетов: 38  
Число взлетевших самолетов: 19  
Число севших самолетов: 10  
Число самолетов, получивших отказ: 8  
Число оставшихся самолетов на посадку: 0  
Число оставшихся самолетов на взлет: 1  
Процент времени, когда полоса свободна: 3.33  
Среднее время ожидания посадки: 1.11  
Среднее время ожидания взлета: 8.60

## Программные проекты 5.4

- P1.** Соберите все подпрограммы моделирования работы аэропорта в полную программу. Используйте модули, директиву `include` или схожее средство, если то или иное из них можно использовать на вашем компьютере, и для пакета подпрограмм обслуживания очереди, и для пакета генерации случайных чисел. Поэкспериментируйте, запуская программу моделирования несколько раз и изменяя значения ожидаемого числа прибытий и отлетов в интервале времени. Найдите приближенные максимальные значения этих чисел, при которых отказ обслуживания самолета еще остается пренебрежимо малым. Что произойдет с этими значениями, если увеличить или уменьшить максимальный размер очереди?
- P2.** Модифицируйте программу моделирования, придав аэропорту две полосы, одну для взлета и другую для посадки. Сравните полное число обслуженных самолетов с той же величиной для однополосного аэропорта. Получили ли вы увеличение пропускной способности более чем в два раза?

- P3.** Модифицируйте программу моделирования, придав аэропорту две полосы, одну обычно используемую для посадки, а вторую обычно используемую для взлета. Если одна из очередей пуста, тогда обе полосы могут быть используемы для другой очереди. Кроме того, если очередь на посадку полна и прибывает новый самолет, запрашивающий посадку, взлеты приостанавливаются и обе полосы используются для устранения отказов приземляющимся самолетам.
- P4.** Модифицируйте программу моделирования, придав аэропорту три взлетно-посадочные полосы, одна из которых всегда зарезервирована для посадки, другая — для взлета, а третья используется для посадки за исключением тех случаев, когда очередь на посадку пуста, и тогда третья полоса может быть использована для взлетов.
- P5.** Модифицируйте исходный вариант программы моделирования (с одной взлетно-посадочной полосой) следующим образом. При прибытии каждого желающего приземлиться самолета ему дается (как часть его записи) определенное случайно сгенерированное количество оставшегося топлива, измеряемое в числе интервалов оставшегося времени. Если самолет не имеет достаточного количества топлива, чтобы оставаться в очереди, ему разрешается немедленная посадка. В результате самолетам в очереди на посадку придется ожидать в воздухе дополнительное число интервалов, и они в свою очередь могут исчерпать свое топливо. Проверяйте эту ситуацию в качестве части процедуры посадки, и получите данные о том, до какой степени занятости может дойти аэропорт перед тем, как самолеты начнут падать из-за исчерпания топлива.
- P6.** Напишите заглушку, чтобы использовать ее вместо функции получения случайного числа. Эта заглушка может быть использована как для отладки программы, так и для того, чтобы позволить пользователю полностью контролировать число самолетов, прибывающих в очередь в каждом временном интервале.

## 5.5. Связные очереди

При реализации очередей с использованием непрерывной памяти манипулировать ими оказывается существенно сложнее, чем стеками, и даже несколько сложнее, чем простыми списками, потому что нам пришлось рассматривать линейную память, как свернутую в кольцо, и обработка крайних случаев полной и пустой очереди была сопряжена с определенными сложностями. Именно при реализации очередей связная память проявляет все свои преимущества. Связные очереди так же просто обрабатывать, как и связные стеки. Нам нужны только два указателя, *front* и *rear*, которые будут указывать соответственно на начало и конец очереди. Операции вставки и удаления проиллюстрированы на рис. 5.5.

Для всех очередей мы считаем, что тип *queueentry* описывает элементы, хранящиеся в очереди. Тогда для связных очередей структура узла может быть объявлена весьма схоже с тем, как это мы делали для стеков.

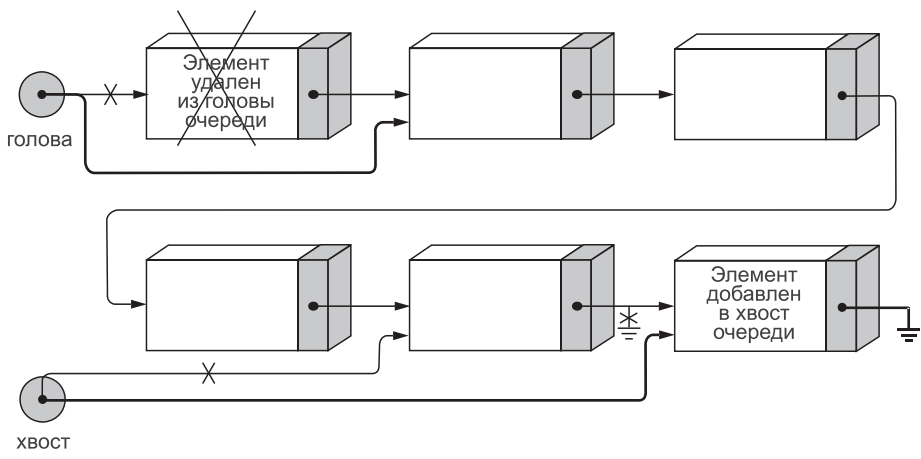


Рис. 5.5. Операции над связной очередью

```
тип queue
  queuepointer = ↑queuenode;
  queuenode = record
    entry: queueentry;
    nextnode: queuepointer
    { связь со следующим узлом в очереди }
  end;
  queue = record
    front,
    rear: queuepointer
  end;
```

инициализация      Очередь должна инициализироваться пустым состоянием с помощью процедуры:

```
procedure CreateQueue (var Q: queue); { Создать очередь }
{ Pre: Предусловия отсутствуют.
  Post: Связная очередь Q создана и инициализирована так,
        что является пустой. }
begin { процедура CreateQueue }
  Q.front := nil;
  Q.rear := nil;
end; { процедура CreateQueue }
```

Процедуры Append и Serve, которые обрабатывают элементы связной очереди, схожи с аналогичными процедурами для связных стеков, и мы оставим их для упражнений.

Обратимся теперь к процедурам обработки узлов очереди. Их следует рассматривать, как зависящие от реализации. Для добавления узла  $p \uparrow$  к хвосту очереди следует написать:

```

procedure AppendNode (p: queuepointer; var Q: queue); { Добавить узел }
{ Pre: Связная очередь Q уже создана, и p указывает на узел,
      еще не находящийся в Q.
  Post: Узел, на который указывает p, помещен в очередь в качестве
        ее последнего элемента.
  Uses: Использует процедуру Error. }
begin                                     { процедура AppendNode }
  if p = nil then
    Error('Попытка добавить в очередь несуществующий узел.');
```

**else begin**

**with** Q **do**

**if** QueueEmpty(Q) **then begin** { очередь Q пуста; установить и front,

и rear так, чтобы они указывали на p }

front := p;

rear := p

**end**

**else begin**

rear↑.nextnode := p; { поместить p↑ после

предыдущего хвоста очереди }

rear := p; { обновить хвост, чтобы он указывал на новый узел }

**end;**

p↑.nextnode := nil { указывает, что новый узел находится

в конце очереди }

**end**

**end;** { процедура AppendNode }

Заметьте, что эта процедура включает проверку на ошибки, чтобы предотвратить включение в очередь несуществующего узла. Случаи, когда очередь пуста или не пуста, должны обрабатываться отдельно, поскольку добавление узла к пустой очереди требует установки и головного, и хвостового указателей на новый узел, в то время как добавление к непустой очереди требует изменения только хвостового указателя.

Для удаления узла из головы очереди мы используем следующую процедуру:

```

procedure ServeNode (var p: queuepointer; var Q: queue);
                                     { Обслужить узел }
{ Pre: Связная очередь Q уже создана, и не пуста.
  Post: Первый узел в очереди изъят, и выходной параметр p
        указывает на этот узел.
  Uses: Использует процедуры QueueEmpty, Error. }
begin                                     { процедура ServeNode }
  with Q do
    if QueueEmpty(Q) then
      Error('Попытка удалить узел из пустой очереди');
```

**else begin**

p := front;

{ вытаскиваем головной элемент, как результат процедуры }

front := front↑.nextnode;

{ переместим голову очереди к следующему узлу }

**if** QueueEmpty(Q) **then** { теперь очередь пуста? }

rear := nil

**end**

**end;** { процедура ServeNode }

И здесь возможность встретиться с пустой очередью следует рассматривать особо. Попытка удалить узел из пустой очереди является ошибкой. Однако получить пустую очередь после удаления не будет ошибкой, но в этом случае оба указателя, и `rear`, и `front`, должны сделаться равными `nil`, чтобы явно показывать, что очередь пуста.

простота

Если вы сравните эти алгоритмы для связанных очередей с теми, что требуются для очередей непрерывных, вы увидите, что связанные варианты и концептуально яснее, и проще в плане программирования.

реализация

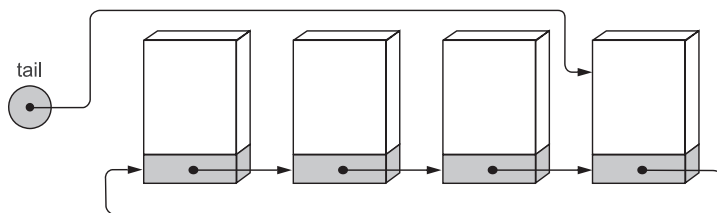
Разработанные нами процедуры обрабатывают узлы; для того, чтобы можно было легко переходить от непрерывной реализации очереди к связанной или наоборот, нам нужны варианты процедур `Append` и `Serve`, которые будут обрабатывать элементы для связанных очередей непосредственно. Мы оставляем написание этих процедур для упражнений, вместе с остающимися процедурами и функциями для обработки очередей: `CreateQueue`, `ClearQueue`, `QueueEmpty`, `QueueFull`, `QueueSize`, `QueueFront` и `QueueFrontNode`.

### Упражнения 5.5

- E1. (a)** Удаляя элемент из связанной очереди, мы проверяли, не пуста ли очередь, однако добавляя элемент, мы не проверяем очередь на переполнение. Почему?
- (b)** Напишите связный вариант функции `QueueFull`.
- (c)** В свете простоты функции `QueueFull` для связанной реализации почему все-таки важно включать ее в пакет операций со связанной очередью?
- E2.** Создавая узлы и отбрасывая узлы, напишите процедуры **(a)** `Append` и **(b)** `Serve` так, чтобы они обрабатывали элементы связанных очередей, и чтобы их можно было непосредственно заменить на непрерывные аналоги.
- E3.** Напишите следующие Pascal-процедуры и функции для связанных очередей:
- (a)** функцию `QueueEmpty`
- (b)** процедуру `ClearQueue`
- (c)** процедуру `QueueFront`
- (d)** процедуру `QueueFrontNode`, которая будет возвращать *указатель на узел* в голове очереди, в то время как `QueueFront` возвращает копию головного элемента.
- E4.** Для связанных списков функция `QueueSize` требует использования цикла, который просматривает всю очередь, подсчитывая ее элементы, поскольку число элементов в очереди не хранится в отдельном поле записи очереди.
- (a)** напишите Pascal-функцию `QueueSize` для связанной очереди, используя цикл, который перемещает переменную-указатель от узла к узлу по всей очереди.
- (b)** Продумайте изменение объявления связанной очереди с целью добавления в запись поля счетчика. Какие изменения придется вне-

сти во все процедуры и функции для связных очередей? Обсудите преимущества и недостатки такой модификации по сравнению с исходной реализацией связных очередей.

- Е5. Список с кольцевой связью**, показанный на рис. 5.6, представляет собой связный список, в котором хвостовой узел, вместо того, чтобы характеризоваться указателем `nil`, указывает назад на узел в голове списка. В этом случае нам нужен только один указатель `tail` для доступа к обоим концам списка, поскольку мы знаем, что `tail↑.nextnode` указывает назад на головной элемент.



**Рис. 5.6.** Список с кольцевой связью и хвостовым указателем `tail`

- (а) Если мы реализуем очередь в виде списка с кольцевой связью, нам нужен только один указатель `tail` (или `gear`) для обращения как к хвосту очереди, так и к ее голове. Напишите Pascal-процедуры, необходимые для обработки очереди, сохраняемой таким образом.
  - (б) Каковы недостатки и преимущества такой структуры по сравнению с вариантом, использующим два указателя?
- Е6.** Если дек хранится в виде списка с кольцевой связью с единственным указателем на хвост списка, три из четырех процедур для добавления узла к тому или иному концу дека и для удаления узла с того или иного конца заметно упрощаются в реализации, но с четвертой процедурой этого не происходит.
- (а) Какая из четырех операций оказывается наиболее сложной? Почему?
  - (б) Напишите Pascal-процедуру для создания пустого дека с кольцевой связью.
  - (с) Напишите Pascal-процедуру для трех более простых операций над связным деком.
  - (д) Напишите Pascal-процедуру для четвертой операции.
- Е7.** Напишите процедуры и функции, необходимые для реализации дека с двойной связностью.

## Программные проекты 5.5

- P1.** Соберите пакет объявлений, процедур и функций для обработки связных списков, годный для использования прикладными программами (с помощью модулей Turbo Pascal или директивы `include`, или другого подобного средства). Пакет должен содержать как подпрограммы для обработки узлов, так и подпрограммы для непосредственной обработки элементов.

- P2.** Возьмите управляемую меню демонстрационную программу для очереди символов (раздел 5.3, проект P1) и замените пакет подпрограмм для связанных очередей пакетом подпрограмм для непрерывных очередей. Если вы разработали и программы, и пакеты достаточно аккуратно, программа будет правильно работать без необходимости внесения в нее каких-либо изменений.
- P3.** В программе моделирования работы аэропорта, разработанной в разделе 5.4, замените пакет для обработки непрерывных очередей пакетом для связанных очередей. Если вы разработали пакеты достаточно аккуратно, программа будет работать в точности так же, как и раньше, без необходимости внесения в нее каких-либо изменений.

## 5.6. Приложения: полиномиальная арифметика

### 5.6.1. Цель проекта

калькулятор  
для полиномов

В качестве прикладного примера использования связанных очередей, в этом разделе будет рассмотрена программа для работы с полиномами. Наша программа будет моделировать поведение простого калькулятора, который выполняет сложение, вычитание, умножение и деление, а также, возможно, и некоторые другие операции, но не над обычными числами, а над полиномами.

обратная  
польская нотация

Существует большое количество различных видов калькуляторов, и мы можем построить нашу программу по аналогии с любым из них. Однако для того, чтобы дополнительно проиллюстрировать использование стеков, выберем для моделирования калькулятор, использующий так называемую *обратную польскую нотацию*. В таком калькуляторе операнды (обычно числа, а у нас — полиномы) вводятся *перед* указанием операции. Операнды проталкиваются в стек. Операция после своего выполнения вытаскивает из стека операнды и проталкивает в него результат. Если знаком ? обозначить проталкивание операндов в стек, +, −, \* и / представляют арифметические операции, и = обозначает печать вершины стека (но не извлечение оттуда операнда), тогда ? ? + = будет означать чтение двух операндов, а затем вычисление и печать их суммы. Инструкция ? ? + ? ? + \* = требует четырех операндов. Если обозначить их a, b, c и d, тогда выводимый результат будет выглядеть как (a + b)\*(c + d). Аналогично инструкция ? ? − = \* ? + = проталкивает a, b и c в стек, заменяет b, c на b − c и выводит полученное значение, вычисляет a \* (b − c), отправляет в стек d и, наконец, вычисляет и выводит (a \* (b − c)) + d. Преимущество калькулятора с обратной польской нотацией заключается в том, что любое выражение, каким бы сложным оно ни было, может быть записано без использования скобок.

избавление  
от скобок

Польская нотация полезна как для компиляторов, так и для калькуляторов, и ее изучение является основной темой главы 13. Здесь же несколько минут упражнений с обратной польской нотацией позволят вам приобрести навыки работы с ней.

## 5.6.2. Главная программа

### 1. Эскиз

Задача программы калькулятора в принципе весьма проста. Она должна всего лишь принимать новые команды и выполнять их. В виде предварительного эскиза программа принимает такую форму:

первый эскиз

```
program PolynomialCalculator(input, output);
                                     { Полиномиальный калькулятор }
begin
  CreateStack(S);                    { пусть S будет стеком; инициализируем его,
                                     чтобы он был пуст }
  while есть еще команды do
    begin
      GetCommand(command);           { command обозначает команду,
                                     готовую к выполнению }
      DoCommand(command)             { выполнить команду }
    end
  end.
```

### 2. Выполнение команд

Для преобразования этого эскиза в язык Pascal мы должны задать, что означает получение команд и как это будет выполняться. Прежде всего будем считать, что команды представляются символами ?, =, +, −, \*, /. Приняв такое решение, мы тут же можем написать Pascal-процедуру DoCommand, задав тем самым действие, отвечающее каждой команде:

```
procedure DoCommand (var command: char);      { Выполнить команду }
{ Pre:   Стек S уже создан, и command является одним из вариантов
          команд, допустимых в калькуляторе.
  Post:   Заданная операция command выполнена, и верхние элементы
          стека S соответствующим образом изменены. }
begin                                           { процедура DoCommand }
  case command of
    'h': Instructions                          { вывод справки для пользователя }
    '?': ReadPolynomial(S)                     { ввод полинома }
    '=': WritePolynomial(S) { вывод на экран полинома на вершине стека }
    '+': Add(S)                                { вытолкнуть из стека два полинома, сложить их
                                               и протолкнуть в стек результат }
    '-': Subtract(S)                           { вычитание полиномов }
    '**': Multiply(S)                          { умножение полиномов }
    '/': Divide(S)                             { деление полиномов }
    'q': ;                                     { завершить программу: делать ничего не надо }
  end;                                         { case }
end;                                         { процедура DoCommand }
```

Эта процедура включает две дополнительные команды: h выводит кадр со справкой для пользователя, а q завершает программу.



### 3. Команды чтения: главная программа

Решив, что команды будут обозначаться одиночными символами, мы можем без труда запрограммировать процедуру GetCommand, которая будет читать с терминала команду за командой. Часто, однако, удобно прочитать сразу строку команд, например, `?? + ?? + * =`, а затем выполнить все эти команды перед вводом следующих. Предусмотрим в программе режим чтения целой строки команд и запоминание их в простом списке. Теперь мы можем написать главную программу в своей окончательной форме за исключением дополнительных объявлений, которые мы включим в программу после того, как выберем структуры данных. Для чтения списка команд мы будем использовать процедуру ReadCommand; эта процедура будет также ответственна за обработку ошибок.

```

program PolynomialCalculator (input, output);
                                { Полиномиальный калькулятор }
{ Pre:  Предусловия отсутствуют.
  Post: Предоставляет возможность пользователю выполнять
        простые арифметические операции над полиномами. }
{ Все объявления и процедуры должны быть включены сюда }
begin                                { главная программа PolynomialCalculator }
  Introduction;
  Instructions;
  Initialize(commandlist, S, quit);
  repeat
    ReadCommand(commandlist, quit);
    TraverseList(commandlist, DoCommand);
    ClearList(commandlist);           { очистить список для ввода новой
                                     последовательности команд калькулятора }
  until quit
end.                                { главная программа PolynomialCalculator }

```

### 4. Процедура ввода

Перед тем как перейти к решению нашей главной задачи (выбора способа представления полиномов и написания процедур их обработки), давайте завершим предварительную работу, написав процедуру ввода ReadCommand. Процедуры Initialize, Introduction и Instructions мы оставим для упражнений.

Процедура ReadCommand должна проверять, что вводимые символы представляют допустимые операции. Для этого она использует набор символов validcommands (рассматриваемых как константы). Если фиксируется ошибка, строка команд должна быть введена повторно с самого начала.

```

procedure ReadCommand (var commandlist: list;
                        var quit: Boolean);      { Прочитать команду }
{ Pre:  Предусловия отсутствуют.
  Post: Параметр commandlist был заполнен списком допустимых
        команд, которые может выполнять калькулятор.
  Uses: Использует процедуры LowerCase, AddList, ClearList. }
var
  OK: Boolean; {допустимы ли все введенные к этому моменту символы?}
  ccommand: char;

```

```

begin                                     { процедура ReadCommand }
repeat                                 { главный цикл будет повторяться,
                                     пока не будет введена вся строка }
  writeln('Размер стека составляет ', StackSize(S));
  writeln('Введите список операций');
  OK := true;
  ClearList(commandlist);
  while OK and (not eoln) and (not quit) do
  begin
    read(command);
    LowerCase(command);
    if command in [' ', ','] then           { ничего не делать }
    else if command = 'q' then             { установим флаг quit }
      quit := true;
    else if command in validcommands then
      AddList(command, commandlist);
    else begin
      writeln('Введенная команда недопустима');
      writeln('введите строку повторно. ');
      OK := false;
    end;
  end;
  readln;
until OK
end;                                     { процедура ReadCommand }

```

## 5. Заглушки и тестирование

К этому моменту мы написали уже достаточно большую часть нашей программы, чтобы можно было ее откомпилировать, отладить и протестировать. Таким образом мы убедимся, что сделанное нами работает правильно.

Для того чтобы программу можно было откомпилировать, мы, разумеется, должны представить заглушки для всех отсутствующих процедур. Однако мы еще не объявили все необходимые типы: самый важный тип `polynomial` еще ждет своего определения. Мы могли бы написать объявление этого типа почти произвольным образом, и все еще иметь возможность компиляции программы, но ради возможности выполнения тестирования гораздо лучше принять такое временное объявление типа:

```
polynomial = real
```

временное  
объявление типа

и протестировать программу, запустив ее в качестве обычного калькулятора с обратной польской нотацией, выполняющего вычисления над действительными числами. Тогда приведенная ниже процедура может служить типичной заглушкой:

```

procedure Add(var S: stack);           { Добавить }
var x, y: real;
begin
  Pop(x, S);
  Pop(y, S);
  Push(x + y, S);
end;

```

Создав такой промежуточный вариант программы, мы заодно убедимся в том, что пакеты стека и утилит интегрированы в программу должным образом.

5.6.3. Структуры данных и их реализация

Обратимся теперь к нашей основной задаче и решим, как мы будем представлять полиномы и писать процедуры их обработки. Если внимательно рассмотреть полином

$$3x^5 - 2x^3 + x^2 + 4$$

существо  
полинома

то мы увидим, что важная информация об этом полиноме содержится в коэффициентах и степенях  $x$ ; сама переменная  $x$  в действительность является просто меткой-заполнителем (фиктивной переменной). Таким образом, мы можем рассматривать полином как структуру, каждый член которой включает в себя коэффициент и степень. В компьютере полином можно представить в виде *списка* из пар коэффициент-степень. Каждая из этих пар составляет запись, поэтому полином будет представлен списком записей. Далее мы определим в наших процедурах правила выполнения арифметических операций над такими списками. Однако, детализируя эти правила, мы обнаружим, что нам постоянно приходится удалять первый элемент списка, а также включать новые элементы в конец списка. Другими словами, арифметические операции обращаются со списком как с очередью, или, более точно, как с *очередью, допускающей обход (просмотр) ее элементов*, так как нам часто приходится просматривать все элементы очереди, хотя удаления осуществляются только с головы очереди, а добавления — только с хвоста.

реализация  
полинома

Должны ли мы использовать непрерывную или связную очередь? Если мы заранее знаем границу степени полиномов, с которыми мы будем иметь дело, и если интересующие нас полиномы имеют ненулевые коэффициенты почти при всех своих членах, тогда, возможно, лучше ис-

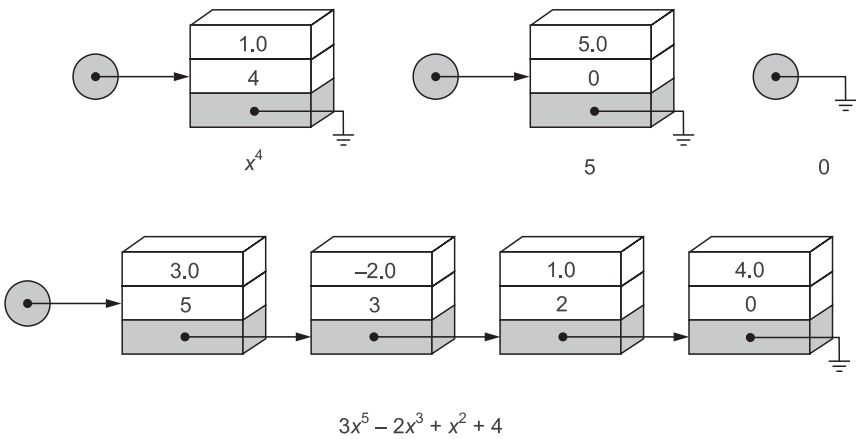


Рис. 5.7. Представление полиномов в виде связных очередей

пользовать непрерывные очереди. Но если нам неизвестна граница степени полиномов, или если у нас будут встречаться полиномы, имеющие ненулевые коэффициенты лишь при немногих своих членах, то связанная память будет предпочтительной. Примем схему представления полинома в виде связанной очереди *членов*, в которой каждый член является записью, состоящей из коэффициента и степени. Такое представление проиллюстрировано на рис. 5.7.

допущения

Каждый узел содержит один член полинома, и мы будем сохранять в очереди только ненулевые члены. Полином, который всегда равен 0 (т. е. состоит из 0 элементов) будет представлен пустой очередью. Мы будем называть такую очередь нулевым полиномом или говорить, что он *тождественно равен 0*.

Теперь мы можем сопоставить всем нашим решениям, касающимся структур данных, стандартные методы объявления стеков, списков и очередей. В результате мы получаем такие объявления для главной программы.

```

const
  maxexponent = maxint;
type
  term = record                                { полиномиальный член }
    exp: integer;                             { показатель не может быть отрицательным }
    coef: real
  end;
  queueentry = term;                          { полином представлен как очередь его членов }
  { $I linkedqu.seg }                        { включить связанные очереди,
                                           специфично для Turbo Pascal }

type
  polynomial = queue;
  stackentry = polynomial;                   { стек полиномов }
  { $I linkedst.seg }                       { включить связанные стеки,
                                           специфично для Turbo Pascal }

var
  validcommands: set of char;                { символы, допустимые в качестве
                                           команд (константы) }
  S: stack;                                  { глобальный стек полиномов }
  commandlist: list;                        { список операций, подлежащих выполнению }
  quit: Boolean;                            { ввод 'q' в качестве команды завершает программу }

```

Мы еще не определили порядок сохранения членов полинома. Если мы позволим им храниться в любом порядке, нам будет трудно распознать, что выражения

$$x^5 + x^2 - 3 \quad \text{и} \quad -3 + x^5 + x^2 \quad \text{и} \quad x^2 - 3 + x^5$$

представляют собой один и тот же полином. Отсюда следует необходимость принятия обычного соглашения о хранении членов полинома в связанной очереди в порядке убывания их степеней, и мы далее принимаем, что ни один член не имеет нулевого коэффициента. (Вспомним, что полином, тождественно равный 0, представляется пустой очередью.)

### 5.6.4. Чтение и вывод полиномов

Если мы остановимся на реализации полиномов в виде очереди, то считывание полинома становится просто просмотром очереди, как это показано ниже.

```

procedure WritePolynomial (var S: stack);                      { Вывод полинома }
{ Pre:   Стек S уже создан и не пуст.
  Post:  Коэффициенты и показатели полинома на верху стека S
          были выведены для показа пользователю.
  Uses:  Использует процедуры и функции StackEmpty, Error, StackTop,
          TraverseQueue, QueueEmpty, WriteTerm. }
var p: polynomial
begin                                     { процедура WritePolynomial }
  if StackEmpty(S) then
    Error('Вывести полином нельзя, поскольку стек пуст')
  else begin
    StackTop(p, S);      { получает копию первого полинома на стеке S }
    if QueueEmpty(p) then
      { выведем что-нибудь в случае пустой очереди }
      write('нулевой полином')
    else
      TraverseQueue(p, WriteTerm);      { вывод полинома на экран }
      writeln
    end
  end;                                     { процедура WritePolynomial }

```

Для вывода одного члена полинома в процессе просмотра очереди мы предусмотрим отдельную процедуру WriteTerm.

```

procedure WriteTerm (var currentterm: term);                  { Вывод члена }
{ Pre:   Допустимый член currentterm существует.
  Post:  Член currentterm выведен на экран. }
begin                                     { процедура WriteTerm }
  if (currentterm.coef > 0) then
    write(' + ');
    write(currentterm.coef: 5: 2, 'x^', (currentterm.exp: 1);
  end;                                     { процедура WriteTerm }

```

По мере того, как мы вводим новый полином, мы будем конструировать новую очередь, присоединяя каждый новый вводимый с терминала член к очереди в качестве ее элемента (пары коэффициент-степень). Этот процесс ввода нового члена и присоединения его к очереди будет заново выполняться почти в каждой процедуре обработки полиномов, по крайней мере, в тех, которые как-то изменяют полиномы. Поэтому с целью упрощения конструирования новых полиномов мы запишем этот процесс в виде отдельной утилиты, которую можно будет затем использовать по мере необходимости в других процедурах.

```

procedure AppendTerm (coefficient: real; exponent: integer;
                      var P: polynomial);                      { Присоединить член }
{ Pre:   Полином P существует; коэффициент coefficient и показатель
          степени exponent допустимы, причем exponent меньше
          наименьшего показателя в P.

```

**Post:** Новый член с данным коэффициентом *coefficient* и показателем *exponent* добавлен в конец полинома *P*.

**Uses:** Использует процедуру *Append*. }

```
var newterm: term;
begin
    newterm.coeff := coefficient;
    newterm.exp := exponent;
    Append(newterm, P)
end;
```

{ процедура *AppendTerm* }

{ процедура *AppendTerm* }

Теперь мы можем использовать эту процедуру для образования и включения в очередь новых членов по мере чтения с терминала пар коэффициент-степень.

Как и все процедуры, принимающие ввод непосредственно от пользователя, наша процедура для чтения новых полиномов должна тщательно проверять входные данные, чтобы они удовлетворяли требованиям нашей задачи. Проверка того, что степени полнома появляются в нисходящем порядке, является одним из самых сложных требований. Нам придется непрерывно сравнивать степень текущего члена со степенью предыдущего.

Мы будем использовать выделенное значение 0 для коэффициента или для степени, чтобы остановить процесс чтения. Вспомним, что член с коэффициентом 0 не сохраняется в очереди, и, поскольку экспоненты поступают в нисходящем порядке, член со степенью 0 всегда будет последним. В результате мы имеем такую процедуру:

```
procedure ReadPolynomial (var S: stack);
{ Чтение полинома }
{ Pre: Стек S уже создан и не полон. }
{ Post: Пользователь предоставил допустимые члены, которые
        собраны в полином newentry в порядке уменьшения показателей.
        Новый полином помещен в стек S. }
{ Uses: Использует процедуры и функции StackFull, Error, CreateQueue,
        AppendTerm, Push. }
```

```
var
    finished: Boolean;
    coefficient: real;
    exponent,
    lastexponent: integer;
    newpoly: polynomial;
```

{ чтение полинома закончилось? }

{ используется для проверки убывания показателей степеней }

{ прочитанный полином }

```
begin
    if StackFull(S) then
        Error('Стек полон: новые полиномы прочитать сейчас нельзя.')
    else begin
        CreateQueue(newpoly);
        writeln('Введите коэффициенты и показатели степеней полинома,');
        writeln('по паре на строке. ');
        writeln('Показатели должны вводиться в порядке убывания. ');
        writeln('Введите коэффициент 0 и показатель 0 для завершения');
        writeln('ввода полинома');
        lastexponent := maxexponent;
```

{ эта сигнальная метка начинает проверку того, что показатели степеней убывают }

```

repeat
  write('коэффициент?');
  readln(coefficient);
  finished :=( coefficient = 0.0);
  if not finished then
    begin
      write('показатель?');
      readln(exponent);
      if exponent >= lastexponent) or (exponent < 0) then
        begin
          finished := true;
          Error('Неверный показатель. Полином заканчивается без этого члена.')
        end
      else begin
        AppendTerm(coefficient, exponent, newpoly);
        lastexponent := exponent;
        finished := (exponent = 0)
      end
    end
  until finished;
end
end;

```

{ процедура ReadPolynomial }

### 5.6.5. Сложение полиномов

Приступим к рассмотрению одной из фундаментальных операций над полиномами, именно, сложения двух полиномов.

Требование упорядоченности членов полинома по убыванию их степеней существенным образом упрощает их сложение. Для того, чтобы сложить два полинома, мы должны просмотреть каждый из них по одному разу. Если мы находим в двух полиномах члены с одной степенью, мы складываем их коэффициенты; в противном случае мы копируем член с большей степенью в полином-сумму и переходим к следующему члену этого полинома. Как только мы достигаем конца одного из полиномов, оставшаяся часть другого копируется в полином-сумму. Мы также должны позаботиться, чтобы в сумму не включались члены с нулевыми коэффициентами.

```

procedure Add (var S: stack);

```

{ Добавить }

```

{ Pre:  Стек S уже создан и не полон.
Post:  Два верхних элемента вытолкнуты из стека и сложены
        вместе. Сумма затем протолкнута в стек.
Uses:  Использует процедуры StackSize, Error, CreateQueue, Pop,
        QueueEmpty, QueueFront, Serve, AppendTerm, MoveTerm, Push. }

var
  summand1, summand2,
  sum: polynomial;
  firstterm,
  secondterm: term;
  coefficient: real;

```

{ два верхние полинома из стека }  
 { сумма }  
 { ведущие члены обоих полиномов }  
 { коэффициент пробного члена полинома }





## 5.6.6. Завершение проекта

### 1. Опущенные процедуры

Оставшиеся процедуры для проекта калькулятора достаточно схожи с теми, что мы уже писали ранее, и их можно оставить для упражнений. Процедуры для оставшихся арифметических операций имеют ту же общую форму, что и наша процедура для сложения. Некоторые из них очень просты: вычитание, например, почти идентично сложению. Для умножения мы должны сначала написать процедуру, умножающую полином на *одночлен*, где под одночленом понимается полином с единственным членом. Эта процедура требует лишь одного просмотра очереди. Затем, чтобы выполнить умножение многочленов, мы комбинируем использование этой процедуры с процедурой сложения. Деление, однако, оказывается более сложным.

### 2. Групповой проект

Разработка логически последовательного пакета подпрограмм для манипуляций с полиномами является интересной задачей для группового проекта. Отдельные члены группы могут писать функции и процедуры для различных операций. Некоторые из этих задач включены в качестве проектов в последнюю часть этого раздела, но вы, конечно, можете включить в пакет и дополнительные средства. Любые дополнительные средства должны быть хорошо продуманы, чтобы их разработку можно было довести до конца за разумное время, не нарушая общий ход учебного процесса.

спецификации

Разделив всю работу между участниками проекта, следует принять важнейшие решения, касающиеся того, каким в точности образом процедуры и функции будут взаимодействовать друг с другом и, особенно, с вызывающей программой. Если вы захотите внести какие-либо изменения в организацию программы, удостоверьтесь в том, что все детали этих изменений точно и полностью доведены до всех членов группы.

взаимодействие

Далее, вы вскоре обнаружите, что не стоит надеяться на выполнение всеми членами группы своей работы за одно и то же время, или что все части этого проекта можно будет объединить и отладить совместно. Следовательно, для отладки и тестирования различных частей проекта вам придется использовать программные заглушки и драйверы (см. раздел 1.4). Один из членов группы может взять на себя персональную ответственность за тестирование. В любом случае, вы увидите, что работа различных членов группы по чтению, помощи в отладке и тестированию программ своих коллег окажется чрезвычайно эффективной.

координация

Наконец, необходимо позаботиться о том, чтобы все члены группы завершили свою работу вовремя, чтобы отслеживался ход реализации различных аспектов проекта, чтобы отдельные подпрограммы не были интегрированы в проект до своей полной отладки и тестирования, и чтобы все составляющие были наконец объединены в единый законченный продукт.

## Упражнения 5.6

**E1.** Pascal-функция может вернуть в качестве результата указатель. Поскольку связный список может быть описан посредством одного указателя на его начало, мы можем определить

polynomial = queuepointer

Теперь можно написать арифметические операции в виде функций, например,

**function** Subtract(p, q: polynomial): polynomial;

Можно также написать Pop(S) как функцию, которая возвращает в качестве своего результата полином, находящийся на вершине стека. Затем процедуру DoCommand можно существенно сократить, используя предложения вроде

Push(Subtract(Pop(S), Pop(S)), S);

- (a) В предположении, что это предложение работает правильно, объясните, почему его использование все же следует рассматривать, как плохой стиль программирования.
  - (b) Вполне возможно, что два различных компилятора языка Pascal, оба строго следующие правилам стандартного языка, оттранслируют это предложение по-разному, что в итоге приведет к различным ответам при выполнении программы. Объясните, как это может случиться.
- E2.** Обсудите шаги, необходимые для расширения функций полиномиального калькулятора, чтобы он мог обрабатывать полиномы от нескольких переменных.

## Программные проекты 5.6

- P1.** Соберите процедуры, разработанные в настоящем разделе, и внесите в их код необходимые изменения, чтобы получить работающую основу для программы калькулятора, которая могла бы вводить данные, складывать полиномы и выводить результат. Вам придется предоставить процедуры Introduction, Instruction и Initialize.
- P2.** Напишите процедуру Subtract и интегрируйте ее в программу калькулятора.
- P3.** Напишите процедуру
- procedure** MonomialMult(monomial: term; **var** poly, product: polynomial)
- которая умножает полином poly на одиночный член monomial, возвращая результат через product.
- P4.** Используя процедуру из предыдущего задания вместе с процедурой Add, напишите процедуру Multiply, и интегрируйте результирующую процедуру в программу калькулятора.
- P5.** Напишите процедуру Divide и интегрируйте ее в программу калькулятора.
- P6.** Процедура ReadCommand, как она написана, будет принимать любую последовательность команд, хотя некоторые из них недопустимы.

Если, например, стек в начале работы пуст, тогда последовательность  $+ \ ? \ ?$  недопустима, так как невозможно сложить два полинома, если ни один из них еще не прочитан. Модифицируйте процедуру `ReadCommand`, чтобы она принимала только допустимые последовательности команд. Процедура должна установить счетчик и инициализировать его числом полиномов на стеке. Когда во входном потоке появляется команда  $?$ , счетчик увеличивается на единицу (поскольку чтение команды  $?$  приведет к проталкиванию дополнительного полинома на стек), а когда появляется любой из символов  $+$ ,  $-$ ,  $*$  или  $/$ , счетчик уменьшается на единицу (поскольку эти команды выталкивают из стека два полинома, и один проталкивают). Если счетчик становится равным нулю или отрицательным, значит, последовательность недопустима.

- P7.** Многие калькуляторы с обратной польской нотацией используют не только стек, но и предоставляют ячейки памяти, в которых могут сохраняться операнды. Расширьте проект предоставлением для полиномов ячеек памяти и предусмотрите дополнительные команды, позволяющие сохранять вершину стека в памяти и проталкивать полиномы, хранящиеся в ячейках памяти, в стек. Следует определить массив из 100 ячеек памяти, и все 100 позиций инициализировать в начале программы нулевыми полиномами. Процедуры, обеспечивающие доступ к памяти, должны спрашивать у пользователя, какие именно ячейки памяти использовать.
- P8.** Напишите процедуру, которая будет отбрасывать верхний полином в стеке, и включите это средство как новую команду.
- P9.** Напишите процедуру, которая будет обменивать местами два верхние полинома в стеке, и включите это средство как новую команду.
- P10.** Напишите процедуру, которая будет складывать вместе все полиномы, находящиеся в стеке, и включите это средство как новую команду.
- P11.** Напишите процедуру, которая будет вычислять производную от полинома, и включите это средство как новую команду.
- P12.** Напишите процедуру, которая при заданном полиноме и действительном числе вычисляет значение полинома от этого аргумента, и включите это средство как новую команду.
- P13.** Модифицируйте процедуру `Divide` так, чтобы результатом процедуры были два новых полинома, частное и остаток, где остаток, если он не равен 0, имеет степень строго меньше, чем степень делителя. Сначала проталкивайте в стек частное, затем остаток.

## 5.7. Абстрактные типы данных и их реализации

### 5.7.1. Введение

Предположим, что расшифровывая длинную и плохо документированную программу, вы нашли такие последовательности команд:

$$xxt \uparrow .xlnk := w; \quad w \uparrow .xlnk := nil; \quad xxt := w;$$

и

```

if ((xh = xxt + 1) and (xxt > 0))
  or ((xxt = mxx) and (xh = 1))
then
  tryagain { пробуем снова }
else begin
  xxt := xxt + 1;
  if xxt > mxx then
    xxt := 1;
  xx [xxt] := wi
end;

```

Вне контекста нелегко сразу сообразить, что должны делать эти фрагменты кода, и без дальнейших пояснений вам, возможно, понадобится несколько минут, чтобы осознать, что они фактически выполняют одну и ту же функцию! Оба фрагмента предназначены для добавления элемента в конец очереди, первый — в очередь со связной реализацией, а второй — в очередь со непрерывным хранением в памяти.

аналогии

У исследователей, работающих в различных областях знаний, часто возникают идеи, фундаментально одинаковые, но разрабатываемые с различающимися целями и выраженные на разных языках. Иной раз проходят годы, перед тем как кто-нибудь осознает сходство этих работ, но когда это сходство обнаруживается, взгляд на проблему с точки зрения одной области знаний может помочь ее решению в другой области. В вычислительной технике точно так же одинаковые основополагающие идеи могут появиться в совершенно разных обликах, которые скрывают их фундаментальную схожесть. Если, однако, мы обнаружим и подчеркнем схожие черты, мы сможем обобщить идеи и найти более простые способы удовлетворения требований многих приложений.

сходство

Когда мы впервые ввели стеки и очереди, мы рассматривали их реализацию только в непрерывной памяти, однако, когда мы перешли к связным стекам и очередям, для нас не составило труда увидеть в них те же самые базовые логические структуры. Запутанность приведенного выше программного фрагмента отражает неумение программиста осознать общую концепцию очереди и воплотить эту общую концепцию в конкретной реализации, требуемой для того или иного приложения.

реализация

Способ реализации базовой структуры может в огромной степени отразиться на разработке программы и на возможностях и полезности результата. Иногда это влияние оказывается глубоко скрытым. Основополагающая математическая концепция действительного числа, например, обычно (но не всегда) реализуется в компьютере в виде числа с плавающей точкой, имеющего определенную степень точности, и внутренне присущие этой реализации ограничения часто приводят к проблемам из-за ошибок округления. Проведение четкой разграничительной линии между логической структурой наших данных и их реализацией в компьютерной памяти может помочь нам в разработке программ. Нашим первым шагом должно быть распознавание логических связей между данными и воплощение этих связей в логической структуре данных. Позже мы можем рассмотреть наши структуры данных и найти наилучший способ их реализации с точки зрения эффективности программирования и вы-

полнения. Разделяя эти два этапа, мы упрощаем анализ каждого из них и избавляемся от ошибок, сопровождающих слишком поспешный выбор решения.

Чтобы подчеркнуть важность разделения логической структуры и ее реализации и достичь большей общности подхода, мы рассмотрим некоторые уже изученные ранее структуры данных с максимально обобщенной точки зрения.

## 5.7.2. Общие определения

### 1. Математические концепции

Математику можно назвать квинтэссенцией обобщений; она представляет язык, необходимый для наших определений. Начнем с определения типа.

**Определение** *Тип* — это множество; элементы этого множества называются *значениями* типа.

Мы можем, таким образом, говорить о типе *целых чисел*, подразумевая множество всех целых чисел, типе *действительных чисел*, подразумеваемая множество всех действительных чисел или типе *символов*, подразумеваемая множество символов, с помощью которых мы будем управлять нашей программой.

Заметьте, что мы всегда можем провести разграничение между абстрактным типом и его реализацией: например, тип *integer* языка Pascal не есть множество всех целых чисел; он включает лишь набор тех целых чисел, которые можно непосредственно представить в конкретном компьютере, и наибольшее из которых равно *maxint*.

Точно так же, тип *real* языка Pascal обычно обозначает определенный набор чисел с плавающей точкой (с разделенными мантиссой и порядком), который является лишь небольшим подмножеством множества всех действительных чисел. Тип *char* языка Pascal также варьируется от компьютера к компьютеру; иногда он представляет собой набор символов ASCII, приведенный в приложении D; иногда это набор символов EBCDIC; могут использоваться и другие наборы символов. Тем не менее, все эти типы, и абстрактные типы, и их реализации, являются множествами и, следовательно, удовлетворяют определению типа.

### 2. Атомарные и структурные типы

Такие типы, как целые, действительные или символы, называются *атомарными* типами, так как мы рассматриваем их значения как одиночные данные, а не такие, которые мы могли бы разделить на части. Однако компьютерные языки и, в частности, язык Pascal, предоставляют и другие типы, такие как массивы, файлы и указатели, с помощью которых мы можем образовывать новые типы, называемые *структурными* типами. Одиночное значение структурного типа (т. е. одиночный элемент соответствующего множества) является массивом, или файлом, или связным списком. Значение структурного типа имеет две составляющих: в нем присутствуют *компонентные* элементы и, кроме того, *структура*, набор правил, объединяющий эти компоненты.

В наших обобщениях мы будем использовать математические средства для описания правил построения структурных типов. Среди этих средств будут множества, последовательности и функции. Для изучения списков разного рода нам понадобится **конечная последовательность**, и для ее определения мы воспользуемся математической индукцией. Определение индукции (как и индуктивное доказательство) содержит две части: первая представляет собой начальный случай, а вторая является определением общего случая с помощью предыдущих случаев.

**Определение** *Последовательность длины 0* пуста. *Последовательность длины  $n \geq 1$*  элементов из множества  $T$  есть упорядоченная пара  $(S_{n-1}, t)$ , где  $S_{n-1}$  есть последовательность длины  $n-1$  элементов из множества  $T$ , а  $t$  есть элемент из множества  $T$ .

С помощью этого определения мы можем строить все более и более длинные последовательности, начав с пустой последовательности и добавляя новые элементы из множества  $T$ , по одному за раз.

В дальнейших рассуждениях мы будем проводить четкое различие между словом **последовательный**, обозначающим, что элементы образуют последовательность, и словом **непрерывный**, под которым мы будем понимать, что узлы имеют примыкающие адреса в памяти. Таким образом, мы можем говорить о *последовательном* списке либо в *связной*, либо в *непрерывной* реализации.

### 3. Абстрактные типы данных

Определение конечной последовательности сразу дает нам возможность попытаться определить список: **список** элементов типа  $T$  есть просто конечная последовательность элементов из множества  $T$ . Далее нам хотелось бы определить стеки и очереди, но если вы подумаете об этих определениях, вы обнаружите, что, рассматривая стеки и очереди как последовательности элементов, мы не увидим в них ничего, что отличало бы их от списка. Единственное различие стеков, очередей и списков заключается в *операциях*, посредством которых выполняется изменение элементов или доступ к ним. Поэтому перед тем, как перейти к этим структурам, мы должны завершить определение простого списка (как мы его ранее исследовали), определив, какие операции для него допустимы. Объединив определения этих операций со структурными правилами, касающимися конечной последовательности, мы получаем

**Определение** *Простой список* элементов типа  $T$  есть конечная последовательность элементов из  $T$  вместе с операциями

1. *Создать* список, оставив его пустым.
2. Определить, *пуст* ли список, или нет.
3. Определить, *полон* ли список, или нет.
4. Найти *размер* списка.
5. *Добавить* новый элемент в конец списка в предположении, что список не полон.
6. *Просмотреть* список, выполняя над каждым его элементом заданную операцию.
7. *Очистить* список, сделав его пустым.

Стеки и очереди теперь можно определить схожим образом.

**Определение** *Стек* элементов типа  $T$  есть конечная последовательность элементов из  $T$  вместе с операциями

1. *Создать* стек, оставив его пустым.
2. Определить, *пуст* ли стек, или нет.
3. Определить, *полон* ли стек, или нет.
4. Найти *размер* стека.
5. *Протолкнуть* новый элемент на вершину стека в предположении, что стек не полон.
6. *Получить* элемент, находящийся на вершине стека, в предположении, что стек не пуст.
7. *Вытолкнуть* элемент, находящийся на вершине стека, в предположении, что стек не пуст.
8. *Очистить* стек, сделав его пустым.
9. *Просмотреть* стек, выполняя над каждым его элементом заданную операцию.

**Определение** *Очередь* элементов типа  $T$  есть конечная последовательность элементов из  $T$  вместе с операциями

1. *Создать* очередь, оставив ее пустой.
2. Определить, *пуста* ли очередь, или нет.
3. Определить, *полна* ли очередь, или нет.
4. Найти *размер* очереди.
5. *Добавить* новый элемент в хвост очереди в предположении, что очередь не полна.
6. *Получить* головной элемент очереди в предположении, что очередь не пуста.
7. *Обслужить* (и удалить) головной элемент очереди в предположении, что очередь не пуста.
8. *Очистить* очередь, сделав ее пустой.
9. *Просмотреть* очередь, выполняя над каждым ее элементом заданную операцию.

В строгом смысле слова просмотр не разрешен для стеков и очередей, поэтому, в этом строгом смысле, мы определили *просматриваемые стеки* и *просматриваемые очереди*. Заметьте, что в этих определениях ничего не говорится о способе, которым мы собираемся реализовывать абстрактные типы данных (список, стек или очередь). В предыдущих главах мы изучали различные реализации каждого из этих типов, и наши новые определения удовлетворяют всем этим реализациям в равной степени. Наши определения относятся к тому, что называется *абстрактным типом данных*, часто сокращаемым до аббревиатуры *ADT* (от abstract data type). Важно отметить, что определение любого абстрактного типа данных включает две части. Первая — это описание соотношения компонентов друг с другом; вторая — это перечень операций, которые могут выполняться над элементами конкретного абстрактного типа данных.



### 5.7.3. Детализация спецификации данных

нисходящая  
спецификация

Теперь, когда мы получили такие обобщенные определения абстрактных типов данных, наступило время разработки более детальных спецификаций, поскольку целью всей этой работы является поиск общих принципов, которые могут помочь в разработке программ, и для достижения этой цели нам нужны детали. Фактически имеется тесная аналогия между процессом нисходящей детализации алгоритмов и процессом нисходящей спецификации структур данных, к которому мы сейчас приступили. При разработке алгоритмов мы начинаем с общего, но точного определения задачи и постепенно специфицируем все больше деталей, пока не получим полностью разработанную программу. При спецификации данных мы начинаем с выбора математических концепций и абстрактных типов данных, требуемых для решения нашей задачи, и постепенно специфицируем все больше деталей, пока не получим возможность описания наших структур данных на языке программирования.

стадии  
детализации

Сколько стадий потребует этот процесс спецификации — зависит от приложения. Разработка большой программной системы потребует значительно больше стадий принятия решений, чем небольшая одиночная программа, и эти решения следует принимать на последовательных стадиях детализации. Хотя различные задачи потребуют различного числа стадий детализации, и границы между этими стадиями зачастую оказываются размытыми, мы можем очертить четыре уровня процесса детализации.

концептуальный  
уровень

1. На *абстрактном* уровне мы решаем, как данные соотносятся друг с другом, и какие операции следует предусмотреть, но мы совсем не занимаемся вопросом, как данные будут храниться в памяти или как конкретно будут выполняться эти операции.

алгоритмический  
уровень

2. На уровне *структур данных* мы определяем достаточно деталей, чтобы получить возможность анализа поведения операций и выбора, диктуемого нашей задачей. Это, например, уровень, на котором мы делаем выбор между непрерывными и связными списками. Некоторые операции легче выполнять над непрерывными списками, другие — над связными. Например, определить длину и получить  $k$ -й элемент проще для непрерывного списка, а добавление и удаление элементов проще реализуются для связных списков. В некоторых приложениях требуются частые включения и удаления в случайных позициях списка; в таких случаях мы выберем связный список. Для других задач непрерывный список может оказаться предпочтительным.

программирование

3. На уровне *реализации* мы определяем, каким образом наши структуры данных будут представлены в памяти компьютера.

4. На уровне *приложения* мы уточняем все детали, касающиеся нашего конкретного приложения, например, имена переменных или специальные требования к операциям, накладываемые приложением.

Первые два уровня часто называют *концептуальными*, потому что на этих уровнях мы в большей степени занимаемся решаемой нами задачей, чем программированием. Средние два уровня можно назвать *алго-*



*ритмическими*, потому что на них определяются конкретные методы представления данных и операций над ними. Последние два уровня связаны уже с конкретным **программированием**.

На рис. 5.8 показаны эти стадии детализации в случае очереди. Мы начинаем с математической концепции последовательности и рассматриваем далее очередь как абстрактный тип данных. На следующем уровне мы делаем выбор из ряда структур данных, показанных на диаграмме, от физической модели (в которой все элементы сдвигаются вперед, как только из очереди изымается головной элемент) и линейной модели (в которой очередь очищается одномоментно) до кольцевых массивов и, наконец, связанных списков. Некоторые из этих структур данных допускают в своей реализации дальнейшие варианты, как это показано на следующем уровне. На заключительной стадии очередь кодируется для конкретного применения.

Завершим этот раздел повторением наиболее важных принципов программирования:

#### **Программистский принцип**

*Пусть ваши данные структурируют вашу программу.  
Детализируйте свои алгоритмы и структуры данных параллельно*

#### **Программистский принцип**

*После того, как вы полностью структурируете свои данные,  
ваши алгоритмы напишутся почти сами собой*

### **Упражнения 5.7**

- E1.** Нарисуйте диаграмму, схожую с рис. 5.8, показывающую уровни детализации для стека.
- E2.** Дайте формальное определение термина *дек* (очередь с двусторонним доступом), используя в качестве модели определения, данные для стека и очереди.
- E3.** В математике *картезианское произведение* множеств  $T_1, T_2, \dots, T_n$  определяется как множество всех  $n$ -кортежей  $(t_1, t_2, \dots, t_n)$ , где  $t_i$  есть член  $T_i$  для всех  $i, 1 \leq i \leq n$ . Используйте картезианское произведение, чтобы дать точное определение *записи (record)* без вариантных полей.

## **Подсказки и ловушки**

1. Перед тем, как приступить к выбору реализации, убедитесь в том, что все ваши структуры данных и связанные с ними операции полностью специфицированы на абстрактном уровне.
2. Выбирая между связной и непрерывной реализациями, составьте полный список операций, которые будут выполняться над вашими данными. Связные структуры обладают большей гибкостью в отношении включений, удалений и перестановок; непрерывные структуры часто обеспечивают большую скорость выполнения.

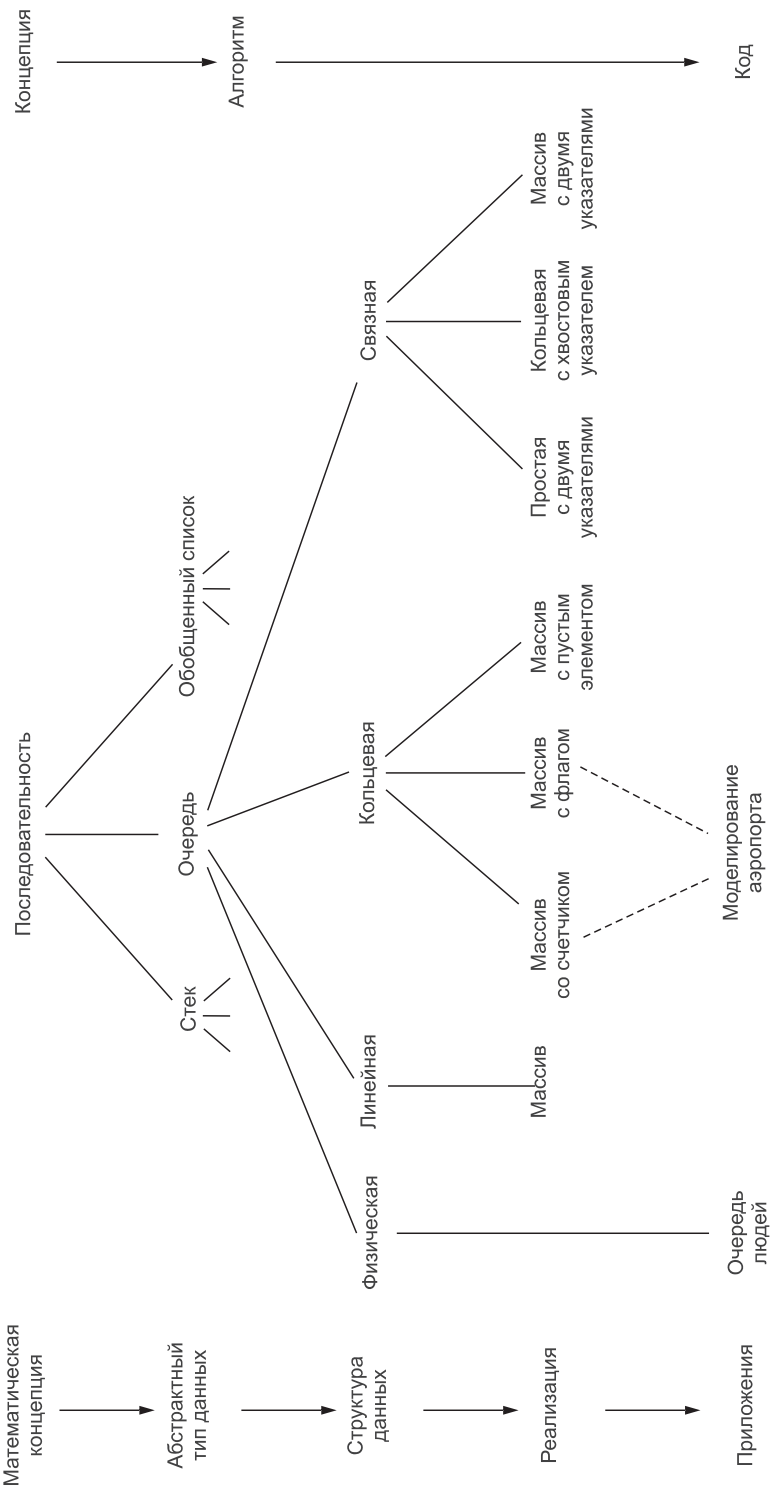


Рис. 5.8. Детализация очереди

3. Непрерывные структуры обычно требуют меньше компьютерной памяти, компьютерного времени и программистских усилий, если элементы структур малы по размеру, а алгоритмы их обработки просты. Если структура содержит большие записи, связанные структуры обычно оказываются более экономными в отношении пространства памяти, времени выполнения и программистского труда.

## Обзорные вопросы

- 5.1
  1. Определите термин *очередь*. Какие операции можно выполнять над очередью?
  2. Каким образом кольцевой список реализуется в виде линейного массива?
  3. Перечислите три различных реализации очередей.
- 5.4
  4. Определите термин *моделирование*.
- 5.6
  5. Обсудите проблемы, которые возникают при групповой работе над программным проектом, и с которыми мы не сталкиваемся при индивидуальной разработке программ. Какие преимущества имеет групповой проект перед индивидуальной работой?
- 5.7
  6. Какие две части должны присутствовать в определении любого абстрактного типа данных?
  7. Какой объем деталей реализации определяется в абстрактном типе данных?
  8. Назовите (в порядке от абстрактного к конкретному) четыре уровня детализации спецификации данных.

## Литература для дальнейшего изучения

Очереди являются неперменной темой обсуждения во всех книгах, посвященных структурам данных. Большинство современных авторов придерживается точки зрения, что свойства структур данных и операций над ними следует рассматривать отдельно от реализации структур данных. Два примера книг такого рода:

Jim Welsh, John Elder, and David Bustard, *Sequential Program Structures*, Prentice-Hall International, London, 1984, 385 pages.

Daniel F. Stubbs and Neil W. Webre, *Data Structures with Abstract Data Types and Pascal*, Brooks/Cole Publishing Company, Monterey, Calif., 1985, 459 pages.

Лучшим источником дополнительной информации, исторических замечок и математического анализа разных сторон использования очередей является [Knuth], том 1 (см. ссылку в главе 3).

Элементарный обзор компьютерного моделирования можно найти в журнале *Byte* 10 (October 1985), pp 149–251. Модель национального аэропорта в г. Вашингтон, округ Колумбия, описана на стр.186–190.

# Глава 6

## Списки

---

В этой главе мы переходим от простейших списков и схожих структур вроде стеков и очередей, где изменения происходят только на концах списка, к более общему виду списков, в которых включения, удаления и извлечения могут выполняться в любой точке списка. После рассмотрения спецификации и реализации таких списков, мы изучим списки символов, называемые символьными цепочками, разработаем простую прикладную программу текстового редактора и, наконец, обсудим реализацию связанных списков внутри массивов.

### 6.1. Спецификации списков

операции,  
сокрытие  
информации  
и реализации

Когда мы начинали изучать стеки, мы практиковали *сокрытие информации* путем разделения вопросов нашего использования стеков и фактического программирования этих операций. При изучении очередей мы также придерживались этого метода и вскоре увидели, что *реализация* допускает большое количество вариантов. Рассматривая обобщенные списки, мы имеем еще большую гибкость и свободу в доступе к любым частям списков и изменению их содержимого. Отсюда принципы сокрытия информации становятся еще более важными для обобщенных списков, чем для списков с ограниченными возможностями. Поэтому мы начнем с перечисления пред- и постусловий для всех операций, которые нам понадобятся осуществлять над списками.

операции  
с простыми  
списками

В разделе 2.2.1 мы описали несколько фундаментальных операций над списком, включая `CreateList`, `ClearList`, `ListEmpty`, `ListFull` и `ListSize`. Правда, единственными операциями, которые предоставляли доступ к элементам списка, были две процедуры: `AddList`, которая добавляла элемент в конец списка, и `TraverseList`, которая позволяла просматривать все элементы списка. Теперь нам понадобятся операции, которые позволят нам получить доступ к *любому* элементу списка. Для разработки этих операций возьмем за основу все объявления, уже сделанные в разделе 2.2.1, и будем по мере необходимости расширять этот перечень.

позиция в списке

Для того, чтобы найти элемент, мы просто используем его *позицию* внутри списка, считая, что первый элемент в списке характеризуется позицией 1, второй — позицией 2 и т. д. Тогда обнаружение элемента списка по его позиции внешне похоже на индексирование массива, однако с

существенными отличиями. Во-первых, если мы вставляем элемент в конкретную позицию, тогда номера позиций всех последующих элементов увеличиваются на 1. Далее, номер позиции для списка определяется безотносительно к его реализации. Для непрерывного списка, реализуемого в массиве, позиция будет, действительно, индексом элемента внутри списка. Но мы можем также использовать позицию для нахождения элемента в связанном списке, где индексы не используются вовсе.

Теперь мы можем дать точные спецификации для оставшихся базовых операций над списком.

**procedure** InsertList (p: position; x: listentry; **var** L: list); {Включить в список}

*предусловие:* Список L уже создан, L не полон, x есть допустимый элемент списка, p есть позиция в списке, причем  $1 \leq p \leq n + 1$ , где n есть число элементов в L.

*постусловие:* Элемент x был включен в позицию p в L; элемент, бывший ранее в позиции p (в предположении, что  $p \leq n$ ) и все последующие элементы получают номер позиции на 1 больше, чем имели ранее.

**procedure** DeleteList (p: position; **var** x: listentry; **var** L: list);

{ Удалить из списка }

*предусловие:* Список L уже создан, L не пуст, и  $1 \leq p \leq n$ , где n есть число элементов в L.

*постусловие:* Элемент в позиции p списка L возвращен как x и удален из L; элементы во всех последующих позициях (в предположении, что  $p < n$ ) получают номер позиции на 1 меньше, чем имели ранее.

**procedure** RetrieveList (p: position; **var** x: listentry; **var** L: list);

{ Извлечь из списка }

*предусловие:* Список L уже создан, L не пуст и  $1 \leq p \leq n$ , где n есть число элементов в L.

*постусловие:* Элемент в позиции p списка L возвращен как x. L не изменился.

**procedure** ReplaceList (p: position; x: listentry; **var** L: list);

{ Заменить в списке }

*предусловие:* Список L уже создан, L не пуст, x есть допустимый элемент списка, а  $1 \leq p \leq n$ , где n есть число элементов в L.

*постусловие:* Элемент в позиции p списка L заменен на x. Остальные элементы списка L остались без изменений.

Для списков возможны и другие операции, но они будут оставлены для упражнений. Позже мы перейдем к вопросам реализации.

### Упражнения 6.1

Имея функции и процедуры для операций над списками, разработанные в этом разделе и в разделе 2.2.1, напишите процедуры для выполнения каждой из приведенных ниже задач. Следите за тем, чтобы удовлетворить пред- и постусловиям для каждой процедуры. Вы можете использовать локальные переменные типов `list`, `listentry` и `position`, но не пишите какой-либо код, основывающийся на выборе реализации. Включите код для обнаружения ошибок при нарушении предусловия.

- E1.** `InsertFirst(x: listentry; var L: list)` включает элемент `x` в позицию 1 списка `L`.
- E2.** `DeleteFirst(var x: listentry; var L: list)` удаляет первый элемент из `L`, возвращая его как `x`.
- E3.** `InsertLast(var x: listentry; var L: list)` включает `x` как последний элемент `L`.
- E4.** `DeleteLast (var x: listentry; var L: list)` удаляет последний элемент из `L`, возвращая его как `x`.
- E5.** `MedianList (var x: listentry; var L: list)` возвращает в `x` центральный элемент `L`, если `L` имеет нечетное число элементов, и элемент слева от центрального, если `L` имеет четное число элементов.
- E6.** `Interchagelist (pos1, pos2: position; var L: list)` обменивает местами элементы в позициях `pos1` и `pos2` списка `L`.
- E7.** `ReverseTraverseList (var L: list; procedure Visit(var x: listentry))` просматривает список `L` в обратном порядке (от последнего элемента списка к первому).
- E8.** `CopyList (var source, dest: list)` копирует все элементы из списка `source` в список `dest`; список `source` не изменяется. Вы можете считать, что `dest` уже существует, но все элементы, если они есть в `dest`, должны быть отброшены.
- E9.** `JoinList (var L1, L2: list)` копирует все элементы из списка `L1` в конец списка `L2`; `L1` остается без изменений, как и все элементы, ранее находившиеся в `L2`. `L2` должен уже существовать.
- E10.** `ReverseList (var L: list)` изменяет порядок всех элементов в `L` на обратный.
- E11.** `SplitList (var source, oddlist, evenlist: list)` копирует все элементы из списка `source` так, что элементы с нечетными номерами позиций образуют список `oddlist`, а элементы с четными номерами позиций образуют список `evenlist`. Вы можете считать, что списки `oddlist` и `evenlist` уже существуют, но все элементы, которые могли в них находиться, должны быть отброшены.

## 6.2. Реализация списков

К этому моменту мы уже определили, как должны вести себя списки, над которыми выполняются описанные выше операции. Теперь наступило время обратиться к деталям реализации списков на языке Pascal. Для простых списков, которые мы изучали ранее, как и для стеков и очередей, мы рассмотрели два вида реализаций: непрерывные реализации, использующие массивы, и связные реализации, использующие указатели

языка Pascal. Для обобщенных списков мы будем иметь те же два вида реализаций, но обнаружим в них несколько интересных вариантов.

### 6.2.1. Непрерывная реализация

Непрерывная реализация обобщенных списков весьма схожа с той, что мы разработали для простых списков в разделе 2.3.1. Объявление списка как записи, состоящей из массива и счетчика, остается без изменений, и большая часть процедур и функций (CreateList, ClearList, ListEmpty, ListFull, ListSize и TraverseList) также оказываются идентичными рассмотренным ранее.

Вместо того, чтобы добавлять элементы с одного конца списка, как это мы делали для простых списков, мы теперь хотим иметь возможность включить новый элемент в любую позицию списка. Для этого нам придется передвинуть элементы внутри массива, чтобы освободить место для включения нового элемента. Результирующая процедура будет выглядеть так:

```

procedure InsertList (p: position; x: listentry; var L: list);
                                { Включить в список }
{ Pre:   Список L уже создан, L не полон, x есть допустимый элемент
        списка, и  $1 \leq p \leq n + 1$ , где  $n$  есть число элементов в L.
  Post: Элемент x включен в позицию p в списке L; элемент,
        находившийся перед этим в позиции p (в предположении,
        что  $p \leq n$ ) и все последующие элементы увеличили номера
        своих позиций на 1. }
var index: position;
begin                                { процедура InsertList }
  if (p <= 0) or (p > L.count + 1) then
    Error('Попытка включить элемент в позицию, отсутствующую в списке')
  else if ListFull(L) then
    Error('Попытка включить элемент в полный список')
  else with L do begin
    for index := ListSize(L) downto p do
      entry [index + 1] := entry [index];
    entry [p] := x;
    count := count + 1
  end
end;                                { процедура InsertList }

```

Какой объем работы выполняет эта процедура? Если мы включаем элемент в конец списка, процедура выполняет только небольшое и фиксированное число команд. Если, в другом крайнем случае, мы включаем элемент в начало списка, тогда процедура должна передвинуть каждый элемент в списке, чтобы освободить место для нового, так что если список имеет большую длину, процедуре придется выполнить весьма значительную работу. В среднем, считая, что включения в любое место списка равновероятны, процедура переместит около половины элементов списка. Таким образом, мы можем сказать, что объем работы, выполняемой процедурой, приблизительно **пропорционален** длине списка  $n$ .

Удаление, как и включение, должно переместить элементы списка для заполнения пустого места, оставшегося от удаленного элемента. Отсюда на удаление также требуется время, приблизительно пропорциональное  $n$ , числу элементов списка. Большинство остальных операций, с другой стороны, не используют циклы и выполняют свою работу за фиксированное время. В целом,

*При обработке непрерывного списка с  $n$  элементами:*

- *Процедурам InsertList и DeleteList требуется время, приблизительно пропорциональное  $n$ .*
- *Процедурам и функциям CreateList, ClearList, ListEmpty, ListFull, ListSize, ReplaceList и RetrieveList требуется фиксированное время.*

Мы не включили в это обсуждение процедуры TraverseList, поскольку время ее выполнения зависит от времени, используемого параметром Visit, которое в общем случае нам неизвестно.

## 6.2.2. Реализация простого связывания

### 1. Объявления

Для связной реализации обобщенного списка мы можем начать с объявления указателей и узлов, схожие с теми, что мы использовали для связных стеков, очередей и простых списков:

```

type
  listpointer = ↑listnode;
  listnode = record
    entry: listentry;
    nextnode: listpointer;           { указатель на следующий узел в списке }
  end;
  position = integer;               { позиция в списке (индекс) }
  list = record
    head: listpointer;              { указатель на головной элемент списка }
    count: integer;                 { число элементов в списке }
  end;
```

Единственное дополнительное объявление здесь — это объявление типа position, который мы будем использовать для поиска конкретного элемента в списке.

Большинство операций для обобщенных списков в простой связной реализации идентичны соответствующим операциям для простых связных реализаций простых списков, рассматриваемых в приложении D, и их нет необходимости здесь повторять.

### 2. Примеры

Для иллюстрации других видов действий, которые нам нужно будет выполнять над связными списками, давайте рассмотрим задачу редактирования текста, и будем считать, что каждый узел содержит одно слово, а также связь со следующим узлом. Предложение "Stacks are lists" [Стеки есть списки] можно увидеть на рис. 6.1 (а). Если мы *включаем* слово "simple" [простые] перед словом "lists", мы получаем конфигурацию, изо-



браженную на рис. 6.1 (b). Далее мы решаем *заменить* "lists" на "structures" [структуры] и *включить* три узла "but important data" (но важные данные) и получаем конфигурацию рис. 6.1.(c). Наконец, мы решаем *удалить* "simple but" и получаем рис. 6.1 (d).

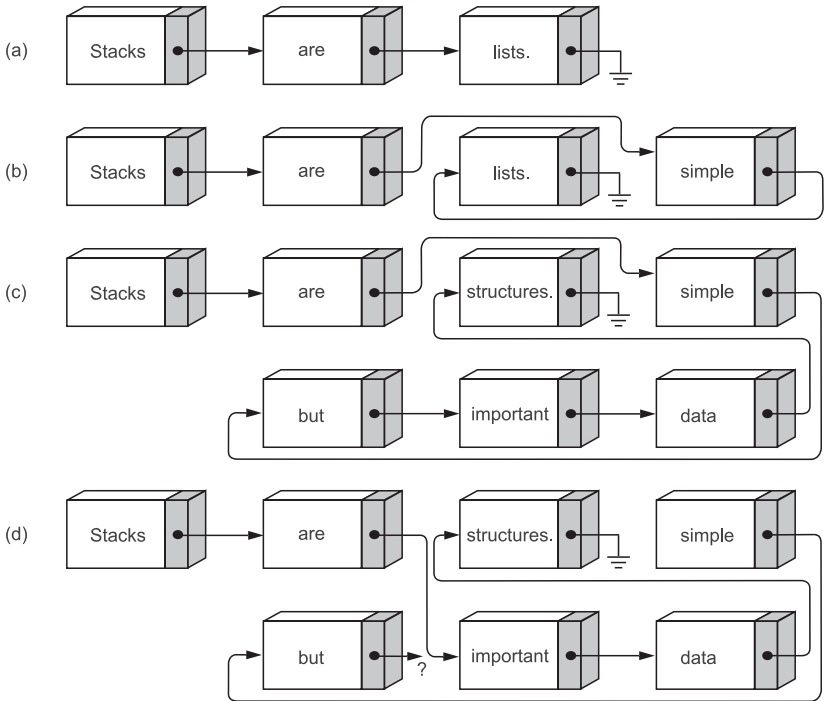


Рис. 6.1. Действия над связным списком

### 3. Поиск позиции в списке

Поскольку мы хотели бы иметь возможность заменять наш пакет для связных списков непосредственно на эквивалентный пакет для непрерывных списков, нам нужна процедура, которая в качестве входного параметра будет получать *позицию* (т. е. целочисленный индекс в списке) и возвращать *указатель* на соответствующий узел списка. Концептуально простейший алгоритм такой процедуры заключается в просмотре элементов списка с самого начала до тех пор, пока мы не достигнем требуемого узла:

```

procedure SetPosition (p: position; var L: list;           { Установить позицию }
                       var current: listpointer);
{ Pre:  p является допустимой позицией в списке L:  $1 \leq p \leq L.count$ .
  Post: Указатель списка current указывает на узел списка в позиции p. }
var i: position;
begin                                     { процедура SetPosition }
  if (p < 1) or (p > L.count) then
    Error('Попытка установить позицию, отсутствующую в списке')

```

```

else begin
  current := L.head;
  for i := 2 to p do                      { смещаемся на p - 1 позиций }
    current := current↑.nextnode
  end
end;                                       { процедура SetPosition }

```

Если запросы ко всем узлам равновероятны, тогда в среднем эта процедура для нахождения требуемой позиции будет просматривать половину списка. Отсюда время ее выполнения будет приблизительно пропорционально  $n$ , размеру списка.

#### 4. Включение

Далее мы рассмотрим задачу включения в связный список нового элемента. Если у нас есть новый узел, который мы хотим включить в середину связного списка, тогда, как это показано на рис. 6.2, мы должны иметь указатель на узел, *предшествующий* тому месту, в которое должен быть включен новый узел. Если `newnode` указывает на новый, включаемый узел, а `current` указывает на предыдущий узел, то это действие потребует двух предложений:

```

newnode↑.nextnode := current↑.nextnode;
current↑.nextnode := newnode;

```

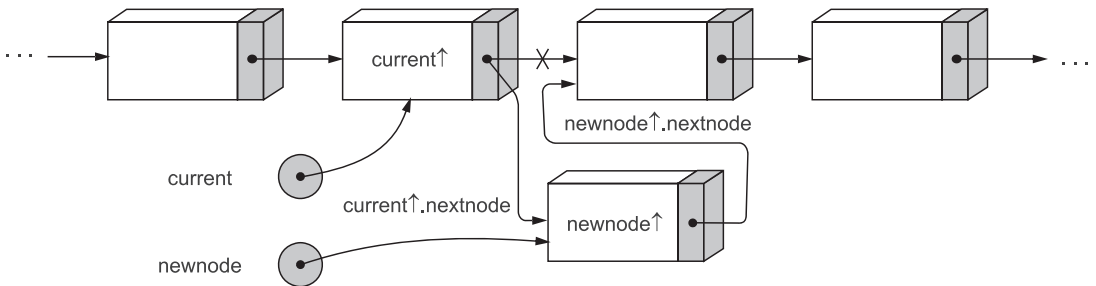


Рис. 6.2. Включение после узла

Обратите внимание на порядок, в котором выполняются операции присваивания для наших указателей. Поле `nextnode` нового узла `newnode↑` было перед этим неопределено; отсюда мы должны сначала присвоить ему новое значение, `current↑.nextnode`. Эта переменная-указатель теперь становится свободной и может принять свое новое значение. Если бы мы попытались выполнить предложения присваивания в обратном порядке, тогда значение `current↑.nextnode` потерялось бы еще перед его использованием, и не было бы никакой возможности присоединить новый узел `newnode↑` к оставшейся части списка.

Теперь мы можем преобразовать этот код в процедуру для включения нового элемента в связный список. Включение в начало списка должно рассматриваться как особый случай, поскольку новый элемент не будет следовать ни за каким другим.

```

procedure InsertList (p: position; x: listentry; var L: list);
    { Включить в список }
{ Pre: Связный список L уже создан, L не полон, и x является
допустимым элементом списка, причем  $1 \leq p \leq n + 1$ ,
где n есть число элементов в L.
Post: Элемент x включен в позицию p в списке L; элемент, ранее
находившийся в позиции p (в предположении, что  $p \leq n$ ),
а также все более поздние элементы увеличили на 1
номера своих позиций.
Uses: Используются процедуры SetPosition, Error. }
var newnode, { используется для ссылки на новый узел }
    current: listpointer; { указывает на узел
                           перед позицией для нового узла }

begin { процедура InsertList }
    if (p <= 0) or (p > L.count + 1) then
        Error("Попытка включить элемент в позицию, отсутствующую в списке")
    else begin
        new(newnode); { создадим новый узел
                       для хранения заданного элемента }

        newnode↑.entry := x;
        if p = 1 then begin { включим узел в начало списка }
            newnode↑.nextnode := L.head;
            L.head := newnode;
        end
        else begin { включим узел в список позже }
            SetPosition(p - 1, L, current);
            { найдем узел перед требуемым местом }
            newnode↑.nextnode := current↑.nextnode;
            current↑.nextnode := newnode;
        end;
        L.count := L.count + 1
    end
end; { процедура InsertList }

```

## 5. Другие операции

Все оставшиеся операции для связанных списков будут оставлены для упражнений. Те из них, которые обращаются к конкретной позиции в списке, должны использовать процедуру SetPosition, иногда для текущей позиции, а иногда, как в случае InsertList, для предыдущей. Все эти процедуры выполняются по большей части за конечное число шагов помимо тех, что входят в SetPosition, за исключением ClearList (и TraverseList), которые просматривают все элементы списка. Таким образом, мы имеем:

*При обработке связанного списка из n элементов*

- ClearList, InsertList, DeleteList, RetrieveList и ReplaceList требуют для своего выполнения время, приблизительно пропорциональное n.
- CreateList, ListEmpty, ListFull и ListSize выполняются за фиксированное время.

Как и раньше, мы не включили в это обсуждение процедуру TraverseList, поскольку время ее выполнения зависит от времени, используемого параметром Visit, которое в общем случае нам неизвестно.

### 6.2.3. Вариация: сохранение текущей позиции

Многие приложения обрабатывают элементы списка по порядку, перемещаясь от одного элемента к следующему. Другие же (и их также достаточно много) обращаются к одному и тому же элементу несколько раз, выполняя операции `RetrieveList` или `ReplaceList` перед тем, как перейти к следующему элементу. Для всех этих приложений наша текущая реализация в виде связанного списка весьма неэффективна, поскольку каждая операция обращения к элементу списка начинает просматривать список от начала до тех пор, пока не обнаружит требуемый элемент. Было бы значительно более эффективно, если бы мы *запоминали* последнюю использованную позицию в списке, и, если следующая операция обращается к той же или более поздней позиции, мы могли бы начать просмотр списка от этой последней использованной позиции.

Заметьте, однако, что запоминание последней использованной позиции не ускорит *любое* приложение, работающее со списком. Если, например, некоторая программа обращается к связанному списку в обратном порядке, начиная от его конца, тогда каждый поиск элемента списка потребует его просмотра от самого начала, поскольку связи в нашем списке однонаправленные, и запоминание последней использованной позиции не поможет найти ту, которая находится перед ней.

Расширенное объявление списка теперь принимает такой вид:

```
list = record
  head, current: listpointer; { указатель на головной элемент и на позицию,
                                использованную последней }
  currentpos: position;       { позиция текущего элемента current↑ в списке }
  count: integer;             { число элементов в списке }
end;
```

Теперь мы можем переписать процедуру `SetPosition`, чтобы она использовала эти новые поля в записи. Заметьте, что поскольку текущая позиция теперь является частью записи списка, процедуре `SetPosition` нет необходимости возвращать указатель в качестве выходного параметра; вместо этого вызывающая подпрограмма может непосредственно обратиться к переменной `current`, входящей в запись списка.

```
procedure SetPosition (p: position; var L: list); { Установить позицию }
{ Pre:  p является допустимой позицией в связанном списке L: 1 ≤ p ≤ n,
        где n есть число элементов в списке L }
Post:  Указатель списка L.current указывает на узел списка в позиции p. }
begin
  if (p < L.currentpos) then begin
    L.currentpos := 1; { настроимся на начало списка }
    L.current := L.head;
  end;
  while L.currentpos <> p do begin
    L.current := L.current↑.nextnode; { движемся к концу списка }
    { переместим указатель current
      к следующему элементу }
    L.currentpos := L.currentpos + 1; { инкремент позиции }
  end
end; { процедура SetPosition }
```

Заметьте, что при повторных ссылках на ту же позицию, ни тело предложения **if**, ни тело предложения **while** не будут выполняться, и процедура почти не потребует времени. Если мы перемещаемся вперед только на одну позицию, тело предложения **while** выполнится всего один раз, так что и в этом случае процедура оказывается очень быстрой. С другой стороны, при необходимости переместиться назад по списку, наша новая процедура будет выполняться почти так же, как и вариант `SetPosition` из предыдущей реализации.

## 6.2.4. Дважды связанные списки

дважды связанный  
список

Некоторые приложения связанных списков требуют, чтобы мы часто перемещались по списку, то вперед, то назад. В последнем разделе мы решали эту проблему путем просмотра списка от начала, пока не будет найден требуемый узел, но такое решение обычно оказывается неудовлетворительным. Программирование такого алгоритма затруднено, а время выполнения программы будет зависеть от длины списка, которая может быть весьма велика.

Существует несколько методов, которые можно использовать для решения проблемы поиска узла, предшествующего текущему. В настоящем разделе мы рассмотрим простейший и, во многих случаях, наиболее гибкий и удобный метод.

### 1. Объявления для дважды связанного списка

Идея метода, как это видно из рис. 6.3, заключается в том, что в каждом узле хранятся *две* связи, указывающие в противоположных направлениях. В результате, используя подходящую связь, мы можем двигаться по списку в обоих направлениях с равной легкостью. Такой список называется *дважды связным списком*.

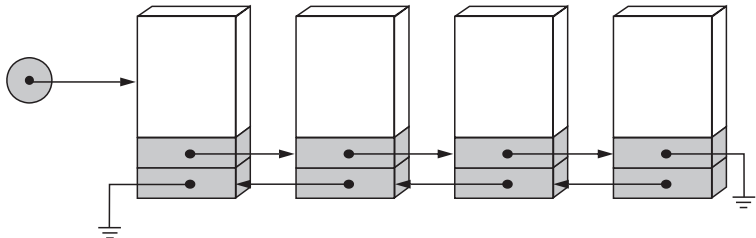


Рис. 6.3. Дважды связанный список

Для дважды связанного списка объявление узла принимает такой вид:

```
type
  listpointer = ↑listnode;
  listnode = record
    entry: listentry;
    nextnode,           { указатель на следующий узел в списке }
    previousnode: listpointer { указатель на предыдущий узел в списке }
  end;
```

а объявление списка становится таким:

```

position = integer;
list = record
    current: listpointer;           { указатель на позицию в списке,
                                    использованную последней }
    currentpos: position;           { положение указателя
                                    (аналогично индексу массива) }
    count: integer;                 { число элементов в списке }
end;

```

В результате становится возможным перемещаться по списку в обоих направлениях, пользуясь только одним указателем на элемент списка.

Заметьте, что в объявлении списка мы включили только указатель на текущий узел списка. Нам даже не нужны указатели на головной или хвостовой элементы списка, поскольку они, как и любые другие узлы, могут быть найдены просмотром назад или вперед от любого узла.

## 2. Операции над дважды связными списками

В дважды связанном списке просмотр в обоих направлениях, поиск конкретной позиции, а также включения и удаления из произвольных позиций списка выполняются без труда. Некоторые процедуры из тех, что вносят в список изменения, оказываются несколько длиннее процедур для простых связанных списков, потому что при включении или удалении элемента из списка они должны обновлять как переднюю, так и заднюю связи.

Для того, чтобы найти конкретную позицию в списке, мы должны только решить, в каком направлении — вперед или назад — мы должны двигаться от текущей позиции. Затем мы выполняем частичный просмотр списка до достижения требуемой позиции. Результирующая процедура выглядит так:

```
procedure SetPosition (p: position; var L: list); { Установить позицию }
{ Pre: p является допустимой позицией в списке L.
  Post: Позиция currentpos и указатель списка current перемещены
        к позиции p. }

begin { процедура SetPosition }
  if (p < 1) or (p > L.count) then
    Error('Попытка установить позицию, отсутствующую в списке.')
  else if L.currentpos < p then { движемся к концу списка }
    repeat
      L.currentpos := L.currentpos + 1; { инкремент позиции }
      L.current := L.current↑.nextnode { переместим указатель вперед }
    until L.currentpos = p
  else if L.currentpos > p then { движемся к началу списка }
    repeat
      L.currentpos := L.currentpos - 1; { декремент позиции }
      L.current := L.current↑.previousnode; { перемещаем указатель
                                              в обратном направлении }
    until L.currentpos = p
end; { процедура SetPosition }
```

При наличии этой процедуры мы можем теперь написать процедуру включения, которая оказывается несколько более длинной из-за необходимости перестраивать две связи. Особо следует рассмотреть случай, когда включение выполняется в конце списка или в пустом первоначально списке.

```

procedure InsertList (p: position; x: listentry; var L: list);
    { Включить в список }
{ Pre:   Связный список L уже создан, p является допустимой позицией
        в списке.
  Post:   Элемент x включен в позицию p в списке L. Указатель current
        указывает на новый элемент.
  Uses:   Используются процедуры SetPosition, Error. }
var
    following,                { указатель на узел за L.current }
    newnode: listpointer;    { указатель на новый узел, включаемый в список }
begin                        { процедура InsertList }
    if (p < 0) or (p > L.count + 1) then
        Error('Попытка включить элемент в позицию, отсутствующую в списке')
    else begin
        new(newnode);        { создадим новый узел для хранения
                                заданного элемента }

        newnode↑.entry := x;
        if p = 1 then
            begin                { включим узел в начало списка }
                newnode↑.previousnode := nil;
                if L.count = 0 then        { обслужим ранее пустой список }
                    newnode↑.nextnode := nil;
                else
                    begin            { включим узел в начало существующего списка }
                        SetPosition(1, L);
                        newnode↑.nextnode := L.current;
                        L.current↑.previousnode := newnode;
                    end;
            end;
        else
            begin                { включим узел далее по списку }
                SetPosition(p - 1, L);        { найдем узел перед требуемым местом }
                following := L.current↑.nextnode;    { включим между позициями
                                                        current и следующей }

                newnode↑.nextnode := following;
                newnode↑.previousnode := L.current;
                L.current↑.nextnode := newnode;
                if following <> nil then        { если current не есть последний элемент }
                    following↑.previousnode := newnode;
                end;
                L.current := newnode;
                L.currentpos := p;
                L.count := L.count + 1
            end
        end;
    end;                        { процедура InsertList }

```

Разумеется, за организацию дважды связанных списков нам приходится платить дополнительной памятью, требуемой в каждом узле для второй связи. Однако для большинства приложений объем памяти, занимаемой

в каждом узле информационным полем `entry`, оказывается гораздо больше, чем память, требуемая для второй связи, поэтому поле второй связи не увеличивает заметным образом полный объем памяти, занимаемой списком.

### 6.2.5. Сравнение реализаций

Теперь, когда мы получили несколько алгоритмов для манипулирования связными списками и несколько вариантов их структур и реализаций, мы можем рассмотреть сравнительные достоинства связной и непрерывной реализаций списков.

преимущества

Наибольшим преимуществом связных списков в динамической памяти является их гибкость. Нам не грозит переполнение, разве что будет исчерпана вся память компьютера. В тех случаях, когда индивидуальные записи имеют большой объем, при использовании непрерывной памяти может оказаться затруднительным заранее определить объем памяти, требуемой для хранения списков, особенно, если требуется оставить достаточный объем памяти для других нужд. При использовании динамического выделения памяти нет необходимости заранее принимать такие решения.

переполнение

изменения

Изменения, особенно, включения и удаления, могут выполняться в середине связного списка более быстро, чем в середине непрерывного списка. Если записи велики по объему, тогда гораздо быстрее изменить значения нескольких указателей, чем копировать сами записи из одного места в другое.

недостатки

Первым недостатком связных списков является расход памяти под связи, памяти, которую можно было бы использовать для других данных. В большинстве систем указатель требует столько же места (одно слово), как целое число. Поэтому список целых чисел потребует при связном хранении в два раза больше места, чем при непрерывном. С другой стороны, во многих реальных приложениях узлы списка весьма велики, и поля данных занимают в них сотни слов. Если каждый узел содержит 100 слов данных, тогда использование непрерывного списка увеличит требование к памяти лишь на один процент, т. е. на совершенно незначительную величину. При этом, если учесть необходимость резервирования дополнительной памяти в массивах непрерывных списков, чтобы иметь возможность делать включения, связная память фактически может оказаться экономнее непрерывной. Если каждый элемент занимает 100 слов, тогда непрерывная организация позволит сэкономить память, только если все массивы будут заполнены на 99 процентов своего объема.

использование  
памяти

произвольный  
доступ

Основной недостаток связных списков заключается в том, что они плохо подходят для произвольного доступа. В случае непрерывной памяти программа может обратиться к любой позиции в списке за одно и то же время. В случае же связного списка для достижения требуемого узла может понадобиться просмотреть длинный путь по списку. Даже доступ к единственному узлу в связной памяти может потребовать несколько большего компьютерного времени, поскольку сначала необходимо получить указатель, и лишь затем обратиться по найденному адресу. Это по-



следнее рассуждение, однако, обычно имеет малое практическое значение. Точно так же поначалу вам будет казаться, что написание процедур манипулирования связными списками требует больше усилий, но с приобретением опыта это различие сведется на нет.

Подытожим сказанное.

*Непрерывная память в общем случае оказывается предпочтительной*

- *если каждая запись в списке невелика по размеру;*
- *если размер списка известен заранее, когда пишется программа;*
- *если над списком выполняется мало операций включения и удаления, за исключением операций в конце списка;*
- *если важен прямой доступ к элементам списка.*

*Связная память имеет преимущества*

- *если записи имеют большой размер;*
- *если размер списка заранее неизвестен;*
- *если требуется гибкость при включении, удалении и перестановке элементов списка.*

Наконец, с целью выбора одного из многих возможных вариантов структуры и реализации, программист должен выяснить, какие именно из возможных операций будут выполняться над списком, и какие из них особенно важны. Ожидается ли *локальность* в выборке данных? Другими словами, если получен доступ к определенному элементу, велика ли вероятность того, что он понадобится снова? Обработываются ли элементы в определенном порядке, или нет? Если такая упорядоченность предполагается, может оказаться полезным ввести в структуру списка последнюю использованную позицию. Важно ли иметь возможность просмотра списка в обоих направлениях? Если да, то преимущества имеет дважды связный список.

## Упражнения 6.2

---

- E1.** Напишите Pascal-процедуры для реализации оставшихся операций для непрерывной реализации списка:
- (a) DeleteList
  - (b) ReplaceList
  - (c) RetrieveList
- E2.** Напишите Pascal-процедуры следующих операций для (первой) реализации простого связного списка:
- (a) ClearList
  - (b) DeleteList
  - (c) ReplaceList
  - (d) RetrieveList
- E3.** Напишите процедуру DeleteList для (второй) реализации простого связного списка, в котором запоминается последняя использованная позиция.

**Е4.** Укажите, какие из приведенных ниже процедур и функций не имеют различий для дважды связанных списков (как они были реализованы в настоящем разделе) и для простых связанных списков. Удостоверьтесь в том, что каждая подпрограмма удовлетворяет спецификациям, данным в разделе 6.1.

- |                |                  |                   |
|----------------|------------------|-------------------|
| (a) CreateList | (e) ListSize     | (i) DeleteList    |
| (b) ClearList  | (f) RetrieveList | (j) TraversetList |
| (c) ListEmpty  | (g) ReplaceList  | (k) SetPosition   |
| (d) ListFull   | (h) InsertList   |                   |

## Программные проекты 6.2

---

- P1.** Подготовьте пакет, состоящий из объявлений, а также всех процедур и функций для обработки непрерывного списка. Пакет должен быть составлен так, чтобы его можно было включить (посредством модулей Turbo Pascal или с помощью директивы include, если ее можно использовать на вашем компьютере) в любую прикладную программу, использующую списки.
- P2.** Напишите управляемую посредством меню демонстрационную программу для обобщенных списков, основанную на приведенной в разделе 2.2.3, но со всеми дополнительными операциями, необходимыми для обобщенных списков. Пусть элементами списка будут символы. Используйте при этом пакет операций для непрерывных списков, разработанный в проекте P1.
- P3.** Соберите пакет из объявлений, процедур и функций для обработки связанных списков, годный для включения в любую прикладную программу, использующую списки (посредством модулей Turbo Pascal, с помощью директивы include или другого подобного средства).
- (a) Используйте простые связанные списки из первой реализации.
  - (b) Используйте простые связанные списки с указателем на последнюю использованную позицию.
  - (c) Используйте дважды связанные списки, как они реализованы в настоящем разделе.
- P4.** В управляемой от меню демонстрационной программе из проекта P2 замените пакет подпрограмм для связанных списков из предыдущего проекта на пакет подпрограмм для непрерывных списков. Если вы разработали эти пакеты достаточно аккуратно, программа будет выполняться правильно без каких-либо дополнительных изменений.
- P5.**
- (a) Модифицируйте реализацию дважды связанных списков так, чтобы вместе с указателем на последнюю использованную позицию она содержала указатели на первый и последний узлы списка.
  - (b) Используйте этот пакет с демонстрационной программой, управляемой меню из проекта P2, и протестируйте результат.
  - (c) Обсудите преимущества и недостатки такого варианта дважды связанных списков в сравнении с дважды связными списками, рассмотренными в тексте.

- Р6. (a)** Напишите программу, которая будет прибавлять, вычитать, умножать и делить произвольные большие целые числа. Каждое число должно представляться списком его цифр. Поскольку целые числа могут содержать сколько угодно разрядов, потребуется применение связанных списков, чтобы избежать переполнения. Некоторые операции требуют движения по списку назад, поэтому удобнее использовать дважды связный список. Умножение и деление можно выполнить, просто повторяя сложение и вычитание.
- (b)** Перепишите операцию умножения так, чтобы она основывалась не на повторении операций сложения, а использовала стандартную процедуру умножения, когда первый сомножитель умножается на каждую цифру другого и полученные произведения складываются.
- (c)** Перепишите операцию деления так, чтобы она основывалась не на повторении операций вычитания, а использовала длинное деление. Вам может понадобиться написать дополнительную функцию, которая выделяет ситуации, когда делитель по абсолютной величине больше делимого.

## 6.3. Цепочки символов

определение

В этом разделе мы продолжим рассмотрение операций над списками на примере цепочек символов<sup>1</sup>. *Цепочка символов* (string) определяется как список, элементами которого являются символы. Примерам цепочек символов являются фразы 'This is a string' или 'Name?', причем апострофы ( ' ') не являются частью цепочки. Можно также образовать *пустую цепочку символов*, обозначаемую "".

Многие Pascal-компиляторы включают в себя средства манипулирования цепочками, схожие с операциями, описанными в настоящем разделе, но такие операции не являются частью стандартного языка Pascal, и поэтому имена, используемые для операций, как и детали их реализации, могут различаться в разных системах. Пакет функций и процедур для работы со цепочками символов, который мы здесь разработаем, можно рассматривать как полезное дополнение к Pascal-системам, которые еще не имеют такого средства.

### 6.3.1. Операции над цепочками символов

Поскольку цепочки символов представляют собой списки, мы можем, разумеется, использовать все разработанные нами ранее операции над списками и для цепочек. Однако для цепочек обычно требуются иные операции, поскольку мы, как правило, будем манипулировать ими как целыми

<sup>1</sup> Два английских вычислительных термина — *line* (строка текста, вводимая с клавиатуры, или строка текста на экране) и *string* (последовательность символов любой длины, в частности, часть входной строки или строки на экране) — обычно переводятся на русский язык одним и тем же словом «строка». Поскольку в программах этого и следующего разделов используются оба эти объекта, то чтобы не было путаницы, термин *line* мы будем переводить как «строка», а термин *string* — как «цепочка символов» или просто «цепочка». — Прим. перев.

объектами, а не обрабатывать в них отдельные элементы-символы. Цепочки символов (символьные строки) используются чрезвычайно широко, и поэтому мы разработаем отдельный набор объявлений типа для цепочек и назовем их не так, как раньше, чтобы в программе можно было работать и с другими списками, не порождая конфликта дважды определенных имен.

спецификации

Ниже приведен список операций, которые нам надо будет разработать для обслуживания цепочек символов. Как и в случае списков, мы используем два типа: `stringindex` (для обращения к позициям цепочки) и `stringlength` (чтобы хранить длину цепочки).

**procedure** CreateString (**var** s: stringtype); { Создать цепочку }  
*предусловие:* Отсутствует.  
*постусловие:* Цепочка символов s создана и инициализирована как пустая.

**function** LengthOfString (**var** s: stringtype): stringlength; { Длина цепочки }  
*предусловие:* Цепочка символов s уже создана.  
*постусловие:* Функция возвращает длину цепочки s (в виде числа символов в ней).

**procedure** ReadString (**var** F:text; **var** s: stringtype); { Прочитать цепочку }  
*предусловие:* Текстовый файл F уже установлен в исходное состояние (и, следовательно, может быть прочитан).  
*постусловие:* Цепочка символов s создана и содержит символы, которые читались из файла F до достижения конца строки ввода или до заполнения цепочки s, в зависимости от того, что встретится раньше.

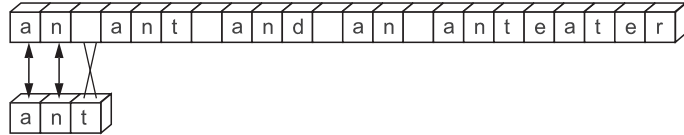
**procedure** WriteString (**var** F:text; **var** s: stringtype); { Записать цепочку }  
*предусловие:* Текстовый файл F уже создан и открыт для записи; цепочка символов s уже создана.  
*постусловие:* Цепочка символов s записана в строку (или в часть строки) текстового файла F.

**procedure** CopyString (instring: stringtype; **var** outstring: stringtype); { Копировать цепочку }  
*предусловие:* Цепочка символов instring уже создана.  
*постусловие:* Цепочка символов outstring создана и является копией instring.

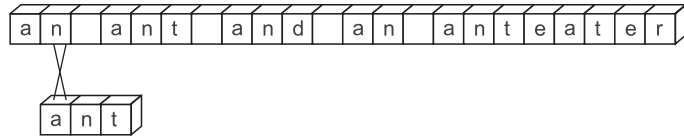


источник несколько раз, но наша функция определит только первое ее вхождение.

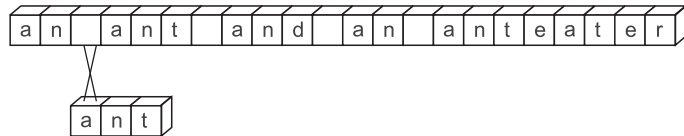
Чтобы проиллюстрировать этот подход к разработке функции `PositionString`, рассмотрим конкретный пример. Пусть мы хотим найти цепочку-мишень `'ant'` [«муравей»] в цепочке-источнике `'an ant and ant eater'` [«муравей и муравьед»]. Мы начинаем просматривать и источник, и мишень с позиции 1 и сопоставляем последовательные символы:



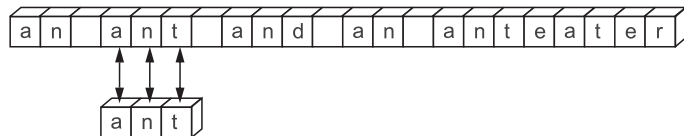
Две первые позиции совпадают, однако источник и мишень различаются в позиции 3. Тогда мы возвращаемся к началу цепочки-мишени, сдвигаемся к следующей позиции в источнике и пробуем снова:



На этот раз мы сразу же обнаруживаем несоответствие в позиции 1, поэтому мы начинаем еще раз, теперь в позиции 3 источника:



Здесь мы опять фиксируем несовпадение первого же символа, поэтому мы переходим к позиции 4 источника:



На этот раз вся цепочка-мишень совпадает с подцепочкой в источнике, и алгоритм успешно завершается. Мы не продолжаем поиск следующего соответствия, которое обнаруживается позже в первых трех буквах слова `'anteater'`.

Наш алгоритм завершится с ошибкой, если мы прогоним цепочку-мишень по всей цепочке-источнику до самого конца и не найдем ни одного соответствия; последняя позиция источника, в которой могло бы начаться соответствие, расположена перед концом источника за столько символов, какова длина мишени.

Этот метод многократно возвращает позицию сравнения назад по мишени, отбрасывая выполненный перед этим просмотр мишени, чтобы попытаться найти соответствие. Такое действие можно рассматривать как пример отхода, с которым мы уже сталкивались в разделе 4.1.

Трансляция этой программы в предложения языка Pascal приводит к следующей функции:

```

function PositionString(var target, source: stringtype): stringlength;
                                { Позиция в цепочке }
{ Pre: Цепочки target и source уже созданы.
  Post: Функция возвращает первую позицию в цепочке source,
        начинающуюся с подцепочки, совпадающей с цепочкой target;
        функция возвращает 0, если такая позиция не обнаружена. }
var
  tryposition: stringindex;      { позиция в цепочке source,
                                анализируемая в текущий момент }
  lastposition: integer;        { последняя позиция, с которой могло бы
                                начаться совпадение }
  match: stringlength; { число позиций в target с совпадающими символами }
begin                                { функция PositionString }
  lastposition := source.length – target.length + 1;
  tryposition := 1;      { начнем искать совпадение от начала источника }
  match := 0;            { первоначально мы не знаем
                        о каких-либо совпадающих символах }
  while(tryposition <= lastposition) and (match < target.length) do
    if target.character [match + 1] =
      source.character [tryposition + match] then
      match := match + 1;      { еще один символ совпал }
    else begin
      match :=0;              { снова начнем поиск совпадающих символов
                              со следующей позиции в цепочке source }
      tryposition :=tryposition +1;
    end;
    if match = target.length then
      PositionString := tryposition
    else
      PositionString :=0
  end;                                { функция PositionString }

```

### Упражнения 6.3

- Е1.** Выполните трассировку операции сравнения символов в функции PositionString, когда она ищет указанные ниже мишени в цепочках-источниках.
- Мишень 'ant' в источнике 'anteater'
  - Мишень 'and' в источнике 'anteater'
  - Мишень 'anteater' в источнике 'ant'
  - Мишень 'tear' в источнике 'anteater'
- Е2.** Какие из приведенных в этом разделе операций с цепочками символов в действительности являются операциями над списками, но с другими именами?
- Е3.** Напишите Pascal-процедуры и функции для оставшихся цепочечных операций:
- (a) Процедура CreateString (**var** s: stringtype)
  - (b) Функция LengthOfString (**var** s: stringtype): stringlength
  - (c) Процедура ReadString (**var** F: text; **var** s: stringtype)

- (d) Процедура `WriteString` (**var** F: text; **var** s: stringtype)
- (e) Процедура `CopyString` (**var** instring, outstring: stringtype)
- (f) Процедура `Concatenate` (first, second: stringtype; **var** out: stringtype)
- (g) Процедура `Substring` (instring: stringtype; start: stringindex)

**Е4.** Напишите Pascal-процедуры и функции, выполняющие приведенные ниже задачи обработки цепочек символов. Используйте, насколько это возможно, уже написанные подпрограммы, вместо того, чтобы писать код заново.

- (a) Удалите символы из цепочки символов. Первый параметр определяет цепочку, второй параметр — позицию первого удаляемого символа, а третий задает число удаляемых символов.
- (b) Вставьте одну цепочку символов (первый параметр) в другую цепочку символов (второй параметр) непосредственно после позиции, заданной третьим параметром.
- (c) Напишите функцию, которая проверяет две цепочки символов на равенство, т. е. имеют ли они одинаковую длину, и если так, совпадают ли все символы в соответствующих позициях.
- (d) Напишите функцию, которая проверяет, идет ли одна цепочка символов в точности перед другой в *лексикографическом* порядке. Это значит, что в первой позиции, где цепочки различаются, символ первой цепочки предшествует символу второй цепочки (при этом цепочки не равны друг другу).

Для следующих процедур, выполняющих чтение из файла F, вы можете использовать файловый буфер F↑ для анализа символа перед его фактическим чтением.

- (e) Прочитайте буквы из текстового файла в цепочку символов, прекратив ввод на первом прочитанном символе, который не является буквой.
- (f) Пропустите в текстовом файле символы пробела и конца строки, а затем прочитайте символы в цепочку символов, остановившись перед следующим пробелом.
- (g) Прочитайте целое число из текстового файла в цепочку символов. Следуйте правилам языка Pascal для синтаксиса целого числа. Зафиксируйте ошибку, если первый прочитанный из файла символ не может начинать целое число. Остановитесь точно перед чтением первого символа, который не может входить в целое число.

**Е5.** Палиндромом называется цепочка символов, одинаково читаемая вперед и назад, т. е. цепочка, в которой первый символ совпадает с последним, второй совпадает с предпоследним и т. д. Примерами палиндромов являются цепочки 'radar' и

'ABLE WAS I ERE I SAW ELBA'

[Последнюю фразу можно приблизительно перевести как  
'Здоров я был, пока не увидел Эльбу'.]

Напишите функцию, которая проверяет, не является ли цепочка символов палиндромом.



альтернативная  
реализация

- Е6.** Разработайте реализацию цепочки символов, которая не использует счетчик для длины цепочки, а вместо этого помещает в конец каждой цепочки специальный символ (который гарантировано не может появиться ни в каком другом месте цепочки). Перепишите процедуры и функции для обработки цепочек символов для этой новой реализации.

### Программные проекты 6.3

- P1.** Подготовьте пакет, содержащий все объявления, процедуры и функции для обработки цепочек символов. Пакет должен годиться для включения его (либо в качестве модуля Turbo Pascal, либо посредством директивы `include`, если ее можно использовать на вашем компьютере) в любую прикладную программу, использующую цепочки символов.
- P2.** Напишите демонстрационную программу, управляемую меню, которая будет осуществлять манипуляции со цепочками символов. Вводя с клавиатуры букву, обозначающую команду, пользователь сможет выбирать любую из цепочечных функций и процедур, рассмотренных в настоящем разделе.
- P3.** Разные авторы склонны использовать разный словарный состав, предложения разной длины и абзацы разного объема. Этот проект предназначен для анализа текстового файла с точки зрения некоторых из этих свойств.

анализ текста

- (a) Напишите программу, которая читает текстовый файл и подсчитывает количество слов каждой длины, встречающейся в тексте, а также общее количество слов. Затем программа выводит на экран среднюю длину слов и процентное соотношение слов каждой длины. Разрабатывая этот проект, считайте, что любое слово состоит исключительно из букв (верхнего или нижнего регистра клавиатуры) и завершается символом, не являющимся буквой.
- (b) Модифицируйте программу так, чтобы она подсчитывала еще и полное количество предложений, а также среднее число слов в предложении. Считайте, что предложение завершается одним из знаков пунктуации: точкой (.), вопросительным знаком (?) или восклицательным знаком (!).
- (c) Модифицируйте программу так, чтобы она подсчитывала еще и количество абзацев и выводила полное число абзацев и среднее число слов в абзаце. Считайте, что абзац завершается пустой строкой или строкой, начинающейся с пробельного символа.

[Этот проект показывает ограничения пакета обработки цепочек: проще написать программу, которая будет читать и обрабатывать отдельные символы, чем делать то же самой с помощью пакета обработки цепочек символов.]

## 6.4. Приложение: текстовый редактор

В этом разделе мы разработаем приложение, демонстрирующее использование и списков, и цепочек символов. Наш проект будет представлять собой миниатюрный текстовый редактор. Программа позволяет вводить лишь несколько простых команд и, следовательно, чрезвычайно примитивна по сравнению с современными текстовыми редакторами или текстовыми процессорами. Однако и в таком виде она иллюстрирует некоторые базовые идеи, используемые при разработке значительно более сложных и изощренных текстовых редакторов.

### 6.4.1. Спецификации

Наш текстовый редактор позволит читать файл в память, причем мы будем говорить, что он сохраняется в *буфере*. Каждую строку текста, поступающую из файла, мы будем рассматривать как *цепочку символов*, а буфер будет представлять собой *список* этих строк. Далее мы разработаем команды редактирования, которые будут выполнять операции над списком в буфере, а также цепочечные операции над символами отдельных строк.

Поскольку в любой момент пользователь может либо вводить символы, включаемые в строки, либо выдавать команды редактирования, текстовый редактор всегда должен с максимальной снисходительностью относиться к неправильному вводу, распознавать недопустимые команды и запрашивать подтверждение на выполнение любых решительных действий вроде удаления всего буфера.

Ниже приводится перечень команд, которые следует включить в текстовый редактор. Команды выдаются вводом буквы в ответ на запрос '?'. Командные буквы разрешается вводить как на верхнем, так и на нижнем регистре клавиатуры.

КОМАНДЫ

- 'R' (Read) Читает текстовый файл, называемый *intext*, в буфер. Любое предыдущее содержимое буфера теряется. После завершения чтения текущей строкой будет первая строка файла.
- 'W' (Write) Записывает содержимое буфера в текстовый файл, называемый *outtext*. Ни текущая строка, ни буфер не изменяются.
- 'I' (Insert) Включает новую одиночную строку, введенную пользователем с клавиатуры, по текущему номеру строки. Запрос 'I' требует ввода новой строки.
- 'D' (Delete) Удаляет текущую строку и перемещается к следующей.
- 'F' (Find) Находит первую строку, начиная с текущей строки, которая содержит цепочку-мишень, запрашиваемую у пользователя.
- 'L' (Length) Показывает длину текущей строки (число символов в ней), а также размер буфера (число строк в нем).
- 'C' (Change) Заменяет цепочку символов, запрошенную у пользователя, на текст замены, также запрашиваемый у пользователя; работает только в пределах текущей строки.



```
begin { главная программа Editor }
Introduction;
CreateList(buffer);
repeat
    GetCommand(command);
    DoCommand(command, buffer, line)
until command = 'q' { цикл завершается командой quit }
end. { главная программа Editor }
```

## 2. Получение команды

Обратимся теперь к процедуре, которая запрашивает у пользователя очередную команду. Поскольку текстовый редактор должен проявлять терпимость к неправильному вводу, мы тщательно проверяем команды, вводимые пользователем, что убедиться в их законности. Первым шагом является преобразование прописных букв в строчные, выполняемое утилитой `LowerCase` из приложения C, чтобы не заставлять пользователя думать о регистре клавиатуры при вводе команд. Процедура `GetCommand` получает команду пользователя, преобразует прописные буквы в строчные и проверяет законность ответа пользователя.

```
procedure GetCommand (var command: char); { Получить команду }
{ Pre: Предусловия отсутствуют.
Post: Процедура возвращает символ command, соответствующий
допустимой команде из множества validcommand.
Uses: Использует процедуры LowerCase, RetrieveList, WriteString
и функцию ListSize. }

var
    valid: Boolean;
    s: stringtype;

begin { процедура GetCommand }
    if ListSize(buffer) = 0 then
        writeln('Буфер пуст.')
    else
        begin
            RetrieveList(line, s, buffer);
            write(line: 1, '.');
            WriteString(output, s);
        end;
        writeln;
        write(' Пожалуйста, вводите команду: ');
        repeat
            readln(command);
            Lowercase(command);
            valid := command in validcommands;
            if not valid then
                write('Введите [H] для получения справки или допустимую команду.')
        until valid
    end; { процедура GetCommand }
```

### 3. Выполнение команд

Процедура DoCommand, предназначенная для выполнения команд, фактически представляет собой одно длинное предложение **case**, которое для каждой команды отсылает программу к соответствующей этой команде процедуре. Некоторые из этих процедур (например, NextLine и DeleteLine) основаны на процедурах обработки списков, но имеют дополнительные средства обработки ошибок. Другие процедуры мы напомним позже; эти операции могут потребовать более значительных программистских усилий.

```

procedure DoCommand (command: char; var buffer: list; var line: integer);
                                { Выполнить команду }
{ Pre:   Параметр command является допустимой командой
      для программы редактора.
Post:   Команда, соответствующая значению параметра command,
      была выполнена над строками буфера buffer или над строкой
      line (в зависимости от команды).
Uses:   Использует все процедуры, выполняющие операции редактора }
begin                                     { процедура DoCommand }
  case command of
    'b':    GoToBeginning(line, buffer);           { Перейти в начало }
    'c':    ChangeString(line, buffer);           { Заменить цепочку }
    'd':    DeleteLine(line, buffer);             { Удалить строку }
    'e':    GoToEnd(line, buffer);               { Перейти в конец }
    'f':    FindString(line, buffer);            { Найти цепочку }
    'g':    GoToLine(line, buffer);              { Перейти к строке }
    '?', 'h': Help;                             { Справка }
    'i':    InsertLine(line, buffer);             { Включить строку }
    'l':    Length(line, buffer);                { Длина строки }
    ' ', 'n': NextLine(line, buffer);            { Следующая строка }
    'p':    PrecedingLine(line, buffer);         { Предыдущая строка }
    'q':    { Quit: ничего не делать };          { Выход }
    'r':    ReadFile(line, buffer);              { Прочитать из файла }
    's':    Substitute(line, buffer);            { Подставить }
    'v':    ViewBuffer(buffer);                 { Вывести буфер на экран }
    'w':    WriteFile(line, buffer);             { Записать в файл }
  end
end;                                     { процедура DoCommand }

```

Чтобы завершить проект, мы должны последовательно написать все процедуры, вызываемые из DoCommand.

### 4. Чтение и запись файлов

Этот вариант процедуры ввода использует для открытия и закрытия входного файла пакет FileIO из приложения C. Причина этого заключается в том, что разные Pascal-системы используют разные методы для обработки файлов, если они не принадлежат к стандартному вводу-выводу через терминал. Таким образом, заключив обработку файлов в отдельный пакет, наша программа может быть перемещена с одной системы на другую без внесения изменений в саму программу.

Поскольку эта процедура уничтожает все предыдущее содержимое буфера, она перед выполнением этой операции запрашивает подтверждение, если только в начале выполнения этой процедуры буфер не был пуст.

**procedure** ReadFile (**var** line: integer; **var** buffer: list; { Прочитать из файла }  
 { **Pre:** Буфер уже создан.  
**Post:** Процедура читает файл *intext* в буфер *buffer*, прекратив чтение в конце файла или после заполнения списка *buffer*; все содержимое буфера *buffer* перед выполнением операции чтения уничтожается после подтверждения пользователя; переменная *line* устанавливается на начало буфера.  
**Uses:** Использует процедуры и функции CreateList, OpenFile, ListFull, ReadString, InsertList, CloseInfile, UserSaysYes. }

```
var
  s: stringtype;
  proceed: Boolean;           { Можно ли отбросить предыдущее
                               содержимое буфера buffer? }
  intext: text;
begin                         { процедура ReadFile }
  proceed := true;
  if not ListEmpty(buffer) then
    begin
      write('Буфер не пуст; чтение уничтожит его содержимое. Продолжить?');
      proceed := UserSaysYes;
      if proceed then           { очистить все строки буфера }
        ClearList(buffer);
      end
    if proceed then
      begin
        CreateList(buffer);
        OpenInfile(intext);
        while (not eof(intext)) and (not ListFull(buffer)) do
          begin
            ReadString(intext, s);           { Чтение строк из файла }
            readln(intext);
            InsertList(ListSize(buffer) + 1, s, buffer);
          end;
          if not eof(intext) then
            writeln(' [Буфер полон; файл не был прочитан полностью.]');
            CloseInfile(intext);
            line := 1;           { перед завершением установим текущую строку
                                на первую строку буфера }
          end;
        end;
      end;
    end;                         { процедура ReadFile }
```

запись файла

Процедура WriteFile несколько проще, чем ReadFile, и мы оставим ее для упражнений.

## 5. Включение строки

Для включения новой строки по текущему номеру строки мы прежде всего должны удостовериться, что буфер не полон, в противном случае у нас не будет места для включения. Затем мы используем цепочечную процедуру ReadString и списковую процедуру InsertList.

```

procedure InsertLine (var line: integer; var buffer: list); { Включить строку }
{ Pre: Буфер list уже создан.
  Post: Строка newline, если она является допустимой строкой,
        включена в список, и переменная line установлена на строку
        newline.
  Uses: Использует процедуры и функции ListFull, ListSize, InsertList,
        ReadString. }
var s: stringtype;
      newline: integer;
begin                                     { процедура InsertLine }
  if ListFull (buffer) then
    writeln(' [Буфер полон; включение строки невозможно.]');
  else begin
    writeln;
    writeln(' Куда вы хотите включить новую строку? : ');
    readln(newline);
    if (newline > 0) and (newline <= (ListSize(buffer) + 1)) then begin
      write(':.');
      ReadString(input, s);
      readln(input);
      InsertList(newline, s, buffer);
      line := newline;
    end
    else writeln(' [Такой номер строки не существует]');
  end;
end;                                     { процедура InsertLine }

```

## 6. Поиск цепочки символов

Теперь перед нами стоит более сложная задача поиска строки, которая содержит цепочку-мишень, предоставляемую пользователем. Сердцем процедуры является вызов цепочечной функции PositionString, однако мы включаем этот код в код списковой обработки, который будет просматривать буфер от текущей строки до конца. Если текущая строка не находится в буфере, мы просматриваем весь буфер. Если цепочка-мишень найдена, мы отображаем на экране всю строку, в которую входит мишень (теперь эта строка становится текущей), отмечая цепочку-мишень знаками стрелок (^), чтобы было наглядно видно ее местоположение.

```

procedure FindString (var line: integer; var buffer: list); { Найми цепочку }
{ Pre: Буфер list уже создан.
  Post: Процедура получает от пользователя цепочку-мишень target и
        перемещает переменную line к первой строке текста, которая
        содержит цепочку target; выводит на экран значение line, где была
        найдена искомая цепочка target, причем target выделяется
        дополнительными символами стрелок. Если target не найдена, не
        выполняется никаких изменений, и пользователь оповещается об этом.
  Uses: Использует процедуры и функции PositionString, LengthOfString,
        RetrieveList, WriteString, ReadString, ListSize, ListEmpty. }
var
  finished: Boolean;                       { Поиск завершен? }
  targetposition: stringlength; { где в строке начинается искомая цепочка? }
  count: stringindex;                { переменная цикла вывода на экран }
  searchline: integer;               { локальная переменная; текущая строка
                                     изменяется только если мишень будет найдена }

```

```

searchlinecontents, {содержимое строки, в которой осуществляется поиск}
target: stringtype;                                     { искомая цепочка-мишень }
begin                                                    { процедура FindString }
  if ListEmpty (buffer) then
    writeln(' [Буфер пуст; поиск невозможен.]');
  else begin
    searchline := line;
    write ('Цепочка, которую следует найти? ');
    ReadString(input, target);
    readln;
    finished := false;
    { Следующий далее цикл начинает просмотр с текущей строки и
      просматривает ее и каждую последующую строку пока не будет
      найдена цепочка-мишень или не будет достигнут конец буфера. }
    repeat
      { цикл от line до тех пор,
        пока не будет найдена мишень target }
      RetrieveList(searchline, searchlinecontents, buffer);
      targetposition := PositionString(target, searchlinecontents);
      { ищем в текущей строке }
      if (targetposition > 0) or (searchline = ListSize(buffer)) then
        finished := true;
      else searchline := searchline + 1;
      { продвинемся на следующую строку }
    until finished;
    if targetposition = 0 then
      writeln(' [Строка не найдена]')
    else begin
      { выделим строку и позицию,
        где найдена искомая цепочка-мишень }
      line := searchline;
      { изменим текущую строку на строку,
        где найдена цепочка-мишень }
      { Вывод на экран строки, где найдена цепочка-мишень. Выделение
        позиции цепочки-мишени внутри строки осуществляется путем вывода
        стрелок '^' под мишенью }
      WriteString(output, searchlinecontents);
      writeln;
      for count := 1 to targetposition - 1 do
        write (' ');
      for count := targetposition to targetposition + LengthOfString(target) - 1 do
        write('^');
      writeln
    end
  end
end;
{ процедура FindString }

```

## 7. Замена одной цепочки символов на другую

В соответствии с практикой многих текстовых редакторов мы сделали поиск цепочки в буфере (команда FindString) глобальным, начиная с текущей позиции и продолжая до конца буфера. Однако к команде ChangeString мы подойдем по-другому, осуществляя замены только в текущей строке. Пользователь очень легко может допустить ошибку, вводя с клавиатуры мишень и заменяющий ее текст. Команда FindString ничего не изменяет, поэтому такого рода ошибка не очень страшна. Если же разрешить команде ChangeString работать глобально, ошибка в наборе





## Программные проекты 6.4

**P1.** Напишите указанные ниже процедуры, после чего протестируйте текстовый редактор и попытайтесь поработать с ним. Если вы используете TurboPascal, импортируйте цепочечные и списковые процедуры из разных модулей. В противном случае, если позволяет ваш компьютер, используйте директиву include (или подобное ей средство) и для пакета обработки списков, и для пакета обработки цепочек символов.

- |                   |                |                    |
|-------------------|----------------|--------------------|
| (a) WriteFile     | (f) DeleteLine | (k) PrecedingLine  |
| (b) GoToBeginning | (g) ViewBuffer | (l) SubstituteLine |
| (c) GoToEnd       | (h) Length     | (m) NextLine       |
| (d) GoToLine      | (i) LengthLine |                    |
| (e) NextLine      | (j) Help       |                    |

**P2.** Добавьте к текстовому редактору средство, позволяющее размещать текст в две колонки следующим образом. Пользователь выбирает диапазон номеров строк, и соответствующие строки из буфера помещаются в две очереди, первая половина строк в одну очередь, а вторая половина строк — во вторую. Далее строки удаляются из очередей, по одной строке из каждой очереди, и объединяются с предопределенным числом пробелов между ними, формируя тем самым результирующую строку. (Пространство между колонками называется межстолбцовым промежутком.)

## 6.5. Связные списки в массивах

старые языки

Некоторые более старые языки программирования, например, Fortran, Cobol или Basic, не предоставляют средств динамического выделения памяти посредством указателей. В то же время, как уже отмечалось ранее, метод связанных списков часто оказывается предпочтительным перед методом непрерывных списков, например, в силу простоты изменения значения указателя вместо копирования большой записи. В этом разделе мы покажем, как реализовать связанные списки, не прибегая к указателям, а используя только целочисленные переменные и массивы.

### 1. Метод

Начнем с выделения рабочего массива (или нескольких массивов для хранения различных частей логической записи в случае, когда язык программирования не поддерживает записи). Далее мы разрабатываем собственные процедуры для фиксации неиспользуемых частей массива и для связи элементов массива вместе в требуемом порядке.

динамическая  
память

Единственное средство связанных списков, которое мы неминуемо потеряем при таком способе реализации — это динамическое выделение памяти, поскольку нам придется заранее решить, сколько места отвести под каждый массив. Все оставшиеся преимущества связанных списков, в частности, гибкость при перестановке больших записей и простота выполнения включений и удалений сохраняются, и такие связанные списки по-прежнему остаются ценным методом организации данных.

преимущества

множественные  
связи

Реализация связанных списков внутри массивов оказывается полезной даже в языках Pascal и других, которые предоставляют указательные типы и динамическое выделение памяти. Приложения, в которых массивы могут оказаться предпочтительнее связи через указатели:

- число элементов в списке заранее известно;
- связи часто перестраиваются, но добавления и удаления осуществляются редко;
- одни и те же данные иногда удобнее обрабатывать в виде связанного списка, а других случаях в виде непрерывного.

Пример одного из таких приложений приведен на рис. 6.4, на котором показана небольшая часть базы данных студентов. Студентам присваиваются идентификационные номера в соответствии с дисциплиной «первым вошел, первым обслужен», поэтому ни имена, ни оценки студентов по любому конкретному курсу не имеют какой-либо упорядоченности. Если известен идентификационный номер студента, запись о нем может быть найдена немедленно обращением к элементам массивов по этому индексу. Иногда, однако, может потребоваться напечатать записи с информацией о студентах в алфавитном порядке, и тогда мы следуем связям, записанным в массив `nextname` (следующее имя). Аналогично студенческие записи могут быть упорядочены в соответствии с оценками студентов по любому курсу, и тогда связи записываются в массивы `nextmath` (следующая оценка по математике) или `nextCS` (следующая оценка по вычислительной технике).

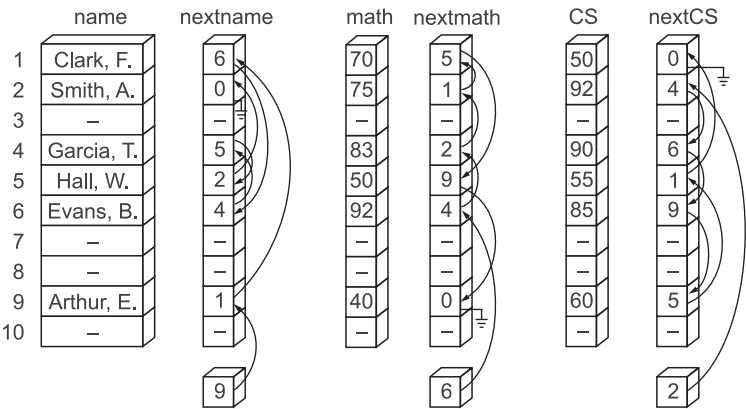


Рис. 6.4. Связные списки в массивах

Чтобы показать, как работает эта реализация, просмотрим связный список `nextname`, показанный во второй колонке рис. 6.4. Заголовок списка (изображенный под списком) содержит значение 9, которое указывает на первый элемент списка, Arthur, E. Позиция 9 массива `nextname` содержит значение 1, и это значит, что имя в позиции 1, Clark, F., является вторым элементом списка. В позиции 1 записано число 6, и элемент с индексом 6, Evans, B., идет следующим. Позиция 6 указывает на позицию

5 (Hall., W.), а позиция 5 связана с позицией 2 (Smith. A.) В позиции 2 поле `nextname` содержит 0, и это означает, что позиция 2 является последним элементом связного списка.

Аналогично этому массив `nextmath` определяет связный список, позволяющий получить оценки студентов по математике в убывающем порядке. В первом элементе записано 6, и этот элемент указывает на 4, а последующие узлы в порядке этого связного списка есть 2, 1, 5 и 9. Порядок, в котором узлы появляются в связном списке `nextCS`, определяет последовательность индексов 2, 4, 6, 9, 5 и 1.

разделяемые  
списки и  
произвольный  
доступ

Как показывает рассмотренный пример, реализация связных списков в массивах позволяет достичь *гибкости* связных списков в плане внесения изменений, возможности *разделять* одни и те же информационные поля (например, имена студентов на рис. 6.4) между несколькими связными списками и получить преимущества *произвольного доступа*, который в ином случае достигим только для непрерывных списков.

индексы

При реализации связных списков с помощью массивов, указатели превращаются в индексы относительно начала массивов, и связи в списке сохраняются в массиве, каждый элемент которого предоставляет индекс, указывающий в пределах массива на следующий элемент списка. Для того, чтобы отличить эти индексы от указателей связного списка с динамической памятью, мы будем называть связи внутри массивов **индексами**, и зарезервируем слово *указатель* для связей в списке с динамической памятью.

индексы **nil**

При составлении программ мы объявим два массива для каждого связного списка, массив `entry [ ]` для хранения информации в узлах и массив `nextnode [ ]`, в который записываются индексы следующих узлов. Для большинства приложений `entry` представляет собой массив записей, или он может быть разбит на несколько массивов, если язык программирования не имеет записей. Оба массива, `entry [ ]` и `nextnode [ ]`, индексируются от 1 до `maxlist`, где `maxlist` представляет собой константу. Поскольку мы начинаем нумерацию индексов с 1, мы вполне можем использовать индекс 0 в специальных целях, именно, для обозначения конца списка, точно так же, как указатель со значением **nil** используется при динамическом выделении памяти. Эта возможность также проиллюстрирована на рис. 6.4.

Рекомендуем вам потратить несколько минут и провести трассировку рис. 6.4, чтобы убедиться, что изогнутые стрелки действительно указывают путь от каждого узла к его последователю.

## 2. Операции: управление пространством памяти

Для получения навыков в организации связных списков в массивах, давайте перепишем некоторые из подпрограмм этой главы в новой реализации.

Нашей первой задачей будет объявление списка достаточного объема и написание процедур для создания нового узла и возврата узла в свободное пространство памяти. Ради достижения программной совместимости с программами раздела 6.2, мы несколько изменим нашу точку зрения. Все используемое нами пространство будет поступать из массива

стек доступной  
памяти

с именем `workspace`, элементами которого будут узлы связного списка. Каждый из этих узлов будет записью с двумя полями, `entry` типа `listentry` и `nextnode` типа `listindex`, причем `listindex` индексирует массив и заменяет собой тип указателя для других связных списков.

Доступное пространство в `workspace` имеет две разновидности. Во-первых, в нем содержатся узлы, которые еще ни разу не выделялись, и, во-вторых, узлы, которые ранее использовались, но теперь освобождены. Мы первоначально выделяем память от начала массива; таким образом, мы можем проследживать, сколько памяти было использовано, с помощью индекса `lastused`, который указывает на позицию последнего из использованных ранее узлов. Ячейки с индексами, большими, чем `lastused`, еще ни разу не были выделены.

Для узлов, которые одно время использовались, но потом были возвращены в доступное пространство, мы должны образовать ту или иную связную структуру, чтобы получить возможность переходить от одного узла к другому. Поскольку связные стеки представляют собой простейший вид таких структур, мы будем использовать связный стек для проследживания узлов, которые ранее использовались, но затем были возвращены в доступное пространство.

Этот стек будет связан посредством индексов в массиве `nextnode`. Поскольку индексы не только свободных узлов, но и всех связных списков будут сосуществовать в том же массиве `nextnode`, этот массив существенно глобален, и мы будем рассматривать его, как глобальную переменную. Но отсюда следует, что все процедуры, изменяющие индексы в списках, будут, модифицируя массив `nextnode`, порождать побочные эффекты.

Чтобы проследживать стек свободного пространства, нам понадобится целочисленная переменная `avail`, которая будет хранить индекс вершины стека. Если стек пуст (что будет представлено значением переменной `avail = 0`), тогда нам требуется получить новый узел, т. е. позицию в массиве, которая еще ни разу не была использована под узел. Мы выделяем этот узел, инкрементируя индексную переменную `lastused`, которая показывает общее число позиций в нашем массиве, уже использованных для хранения элементов списка. Когда `lastused` достигает значения `maxlist` (граница, которую мы приняли для размера массива), а `avail = 0`, рабочая область заполнена, и память под узлы выделить больше нельзя. Когда начинает выполняться главная программа, обе переменные `avail` и `lastused` инициализируются нулями, `avail` для индикации того, что стек ранее использованного но теперь освобожденного пространства пуст, а `lastused` для индикации того, что пространство массива еще не использовалось.

Если мы используем модуль Turbo Pascal, мы можем поступить лучше, и объявить рабочее пространство `nextnode` вместе с переменными `lastused` и `avail` внутри модуля; более точно — как глобальные для модуля, но в секции реализации, чтобы к ним нельзя было обратиться извне модуля. Они могут затем автоматически инициализироваться программой при ее запуске, что избавит нас от необходимости писать дополнительный код в вызывающей программе.

глобальный  
массив  
с побочными  
эффектами

Список доступного пространства показан на рис. 6.5. На этом рисунке, кстати, еще показано, как два связанных списка могут сосуществовать в одном массиве. Стрелки, изображенные слева от массива `nextnode` и начинающиеся с элемента, следующего за `firstname`, описывают связный список, который упорядочивает имена в массиве `info` в алфавитном порядке. Стрелки, начинающиеся с элемента, следующего за `avail`, показывают ячейки (предварительно использованного, но теперь) доступного пространства стека. Заметьте, что индексы в списке доступного пространства являются в точности индексами позиций 11 или меньших, которые не выделены для имен в массиве `info`. Наконец, ни один из элементов в позициях 12 или больших ни разу не был выделен. Этот факт отмечается значением переменной `lastused = 11`. Если нам понадобится включить дополнительные имена в список, возглавляемый переменной `firstname`, мы сначала вытолкнем из стека с вершиной `avail` все находящиеся там узлы, и только когда стек освободится, будем увеличивать `lastused`, чтобы включить новое имя в ранее неиспользованное пространство.

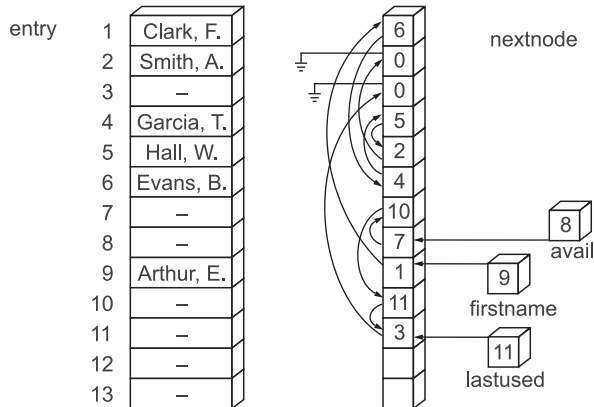


Рис. 6.5. Массив и стек доступного пространства

Принятые нами решения преобразуются в следующие объявления, которые следует поместить в модуль связанных списков:

```
unit ArrayLL;
interface
const
  maxlist = 10; { измените в соответствии с вашими требованиями }
type
  listentry = char { измените в соответствии с вашими требованиями }
                 listindex = 0..maxlist; { это вместо указателей
                                           для обращения к узлам списка }

  listinode = record
    entry: listentry;
    nextnode: listindex; { указатель на следующий узел в списке }
  end;
  position = 1..maxlist { позиция внутри списка
                         (отсчитывая от начала) }

  list = record
    head: listindex { индекс головного элемента списка }
    count: integer; { число элементов в списке }
  end;
```

процедурный  
параметр

```

processlistentry = procedure(var x: listentry); {специфично для Turbo Pascal }
procedure CreateList(var L: list);
procedure SetPosition(p: position; var L: list; var indexp: listindex);
procedure ClearList(var L: list);
function ListEmpty(var L: list): Boolean;
function ListFull(var L: list): Boolean;
function ListSize(var L: list): integer;
procedure InsertList(p: position; x: listentry; var L: list);
procedure DeleteList(p: position; var x: listentry; var L: list);
procedure RetrieveList(p: position; var x: listentry; var L: list);
procedure ReplaceList(p: position; x: listentry; var L: list);
procedure TraverseList(var L: list; Visit: processlistentry);
implementation
uses Utility; { специфичный для Turbo Pascal откомпилированный модуль }
var workspace: array [position] of listnode;
    avail,
    lastused: listindex;
{ Полные объявления всех процедур и функций следует включить в это
место. }
begin                                     { начало инициализации модуля }
    avail := 0;                            { стек ранее использованных доступных узлов пуст }
    lastused := 0;                         { в массиве еще не использовались никакие позиции }
end.                                     { модуль ArrayLL }

```

Сделав такие объявления, мы можем написать процедуры для отслеживания неиспользованного пространства. Процедуры `NewNode` и `DisposeNode` принимают такую форму:

```

procedure NewNode (var newindex: listindex);           { Новый узел }
{ Pre: Индексы списков avail и lastused уже инициализированы в модуле
и использовались или модифицировались только процедурами
NewNode и DisposeNode. Массив workspace не полон.
Post: Индекс списка newindex установлен на первое доступное место
в workspace; переменные avail, lastused и workspace
модифицированы должным образом.
Uses: Использует avail, lastused и workspace как глобальные
переменные. }
begin                                     { процедура NewNode }
    if avail <> 0 then begin                 { вытолкнуть из связанного стека
                                           доступного пространства }
        newindex:= avail;
        avail:= workspace [avail].nextnode;
        workspace [newindex].nextnode := 0;      { защита от использования
                                                    стека пользователем }
    end
    else if lastused < maxlist then begin      { выделим ранее не
                                                    использовавшийся узел }
        lastused := lastused + 1;
        newindex := lastused;
        workspace [newindex].nextnode := 0;      { защита от возможных
                                                    ошибок пользователя }
    end
    else Error('Переполнение: рабочая область для связанных списков полна.');
```

**end;**







```

procedure InsertList (p: position; x: listentry; var L: list);
                                { Включение в список }
{ Pre:   Связный список L уже создан, L не пуст, x является
          допустимым элементом, и  $1 \leq p \leq n + 1$ , где n есть число
          элементов в L.
Post:   Элемент x включен в позицию p списка L; элемент, ранее
          занимавший позицию p (в предположении, что  $p \leq n$ ) и все
          последующие элементы увеличили номера своих позиций на 1.
Uses:   Использует процедуры SetPosition, Error.
var newindex,                { используется для ссылки на новый узел }
      previous: listindex;     { указывает на узел, расположенный
                                перед местом включения нового узла }
begin
  if (p <= 0) or (p > L.count + 1) then
    Error('Включение в несуществующую позицию.')
  else begin
    NewNode(newindex);         { получить новый узел для хранения
                                указанного элемента }

    workspace [newindex].entry := x;
    if p = 1 then begin        { включим в начало списка }
      workspace [newindex].nextnode := L.head;
      L.head := newindex
    end
    else begin                { включим дальше по списку }
      SetPosition (p - 1, L, previous);
                                { найдем позицию
                                перед требуемым местом }
      workspace [newindex].nextnode := workspace [previous].nextnode;
      workspace [previous].nextnode := newindex
    end;
    L.count := L.count + 1
  end
end;                          { процедура InsertList }

```

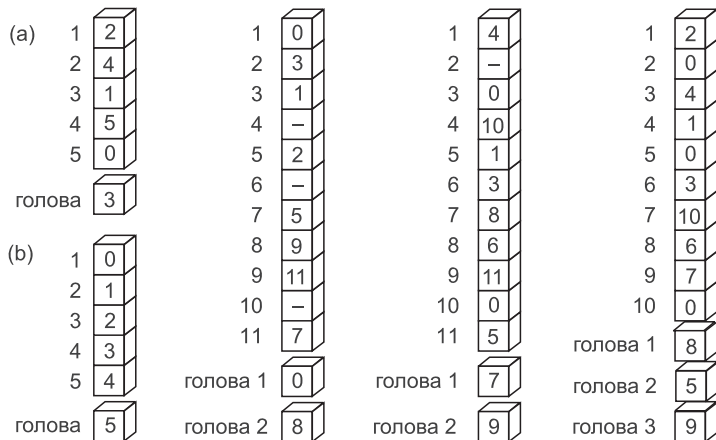
Сравните эти две процедуры с их эквивалентами для простых связанных списков с указателями и динамическим выделением памяти из раздела 6.2. Вы легко обнаружите сходство и увидите, какие изменения следует внести для преобразования из одной реализации в другую.

#### 4. Варианты связного списка

Массивы с индексами не ограничиваются реализацией простых связанных списков. Они в равной степени эффективны и в случае дважды связанных списков, а также и при других вариантах. Для дважды связанных списков возможность выполнять арифметические операции с индексами весьма удобна при реализациях (использующих как отрицательные, так и положительные значения индексов), в которых ссылки вперед и назад могут находиться в единственном поле индекса (см. упражнение E5).

#### Упражнения 6.5

**E1.** Покажите стрелками, как элементы списка связаны друг с другом в каждой из приведенных ниже таблиц nextnode. Некоторые из таблиц содержат более одного списка.



**Е2.** Сконструируйте таблицы `nextnode`, показав, как каждый из приведенных ниже списков связан в алфавитном порядке. В каждом случае укажите значение переменной `head` (головного элемента), которая начинает список.

<b>(a)</b> 1 array	<b>(c)</b> 1 the	<b>(d)</b> 1 London
2 stack	2 of	2 England
3 queue	3 and	3 Rome
4 list	4 to	4 Italy
5 deque	5 a	5 Madrid
6 scroll	6 in	6 Spain
	7 that	7 Oslo
<b>(b)</b> 1 push	8 is	8 Norway
2 pop	9 I	9 Paris
3 add	10 it	10 France
4 delete	11 for	11 Warsaw
5 insert	12 as	12 Poland

**Е3.** Для списка городов и стран в части (d) предыдущего упражнения сконструируйте таблицу `nextnode`, которая порождает два связанных списка, один, содержащий все города в алфавитном порядке, и другой, содержащий все страны в алфавитном порядке. Дайте также значения двум переменным, именуемым эти списки.

**Е4.** Напишите вариант каждой из указанных ниже процедур и функций для связанных списков в массивах. Не забудьте, что каждая подпрограмма должна удовлетворять спецификациям, данным в разделе 6.1, и объявлениям, приведенным в этом разделе.

<b>(a)</b> SetPosition	<b>(e)</b> ListFull	<b>(i)</b> DeleteList
<b>(b)</b> CreateList	<b>(f)</b> ListSize	<b>(j)</b> InsertList
<b>(c)</b> ClearList	<b>(g)</b> RetrieveList	<b>(k)</b> TraverseList
<b>(d)</b> ListEmpty	<b>(h)</b> ReplaceList	

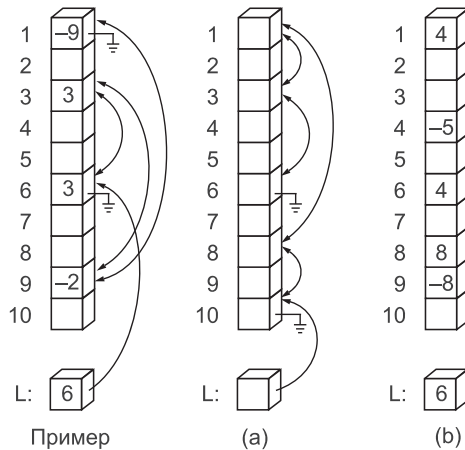
**E5.** Дважды связный список в массиве `workspace` возможно реализовать, используя лишь один индекс `nextnode`. Другими словами, нам не нужно иметь отдельное поле `previousnode` в записях, образующих этот массив, чтобы найти обратные ссылки. Идея метода заключается в размещении в поле `workspace [current].nextnode` не индекса следующего элемента списка, а индекса следующего элемента *минус* индекс элемента, предшествующего текущему. Мы должны также поддерживать два указателя на последовательные узлы в списке, индекс `current` текущего узла и индекс `previous` узла, непосредственно предшествующего текущему. Для нахождения следующего элемента списка мы выполняем такие вычисления:

$$\text{workspace}[\text{current}].\text{nextnode} + \text{previous}$$

Аналогично, чтобы найти предшествующий узел, мы вычисляем:

$$\text{current} - \text{workspace}[\text{previous}].\text{nextnode}$$

Пример такого списка показан в левой части приведенного ниже рисунка. Внутри каждого кубика показано значение, сохраняемое в `nextnode`; с правой стороны графически обозначены соответствующие вычисления для значений индексов.



- (a) Для дважды связного списка, показанного в части (a) приведенного выше рисунка, покажите значения, которые будут сохранены в полях `L.head` и `nextnode` используемых узлов рабочего пространства.
- (b) Для значений `L.head` и `nextnode`, показанных в части (b) предыдущего рисунка, нарисуйте связи для дважды связного списка.
- (c) Напишите процедуру `SetPosition` для рассмотренной в настоящем разделе реализации.
- (d) Напишите процедуру `InsertList` для такой реализации.
- (e) Напишите процедуру `DeleteList` для такой реализации.

## 6.6. Генерирование перестановок

Последний пример проекта в этой главе иллюстрирует использование как обобщенных, так и связанных списков в массивах весьма своеобразным способом, который целиком определяется спецификой приложения. В проекте генерируются все  $n!$  перестановок  $n$  различных объектов максимально эффективным образом. Вспомним, что перестановки  $n$  различных объектов представляют собой все способы их размещений в разном порядке. Более детально перестановки обсуждаются в приложении А.

Почему для  $n$  объектов имеется  $n!$  перестановок? Причина этого заключается в том, что мы можем выбрать любой объект в качестве первого, затем выбрать любой из оставшихся  $n - 1$  объектов в качестве второго и т. д. Все эти альтернативы независимы, поэтому число альтернатив перемножается. Если представить себе  $n!$  как произведение

$$n! = 1 \times 2 \times 3 \times \dots \times n,$$

то процесс умножения можно представить в виде дерева, изображенного на рис. 6.6 (не обращайтесь пока внимания на надписи у узлов дерева).

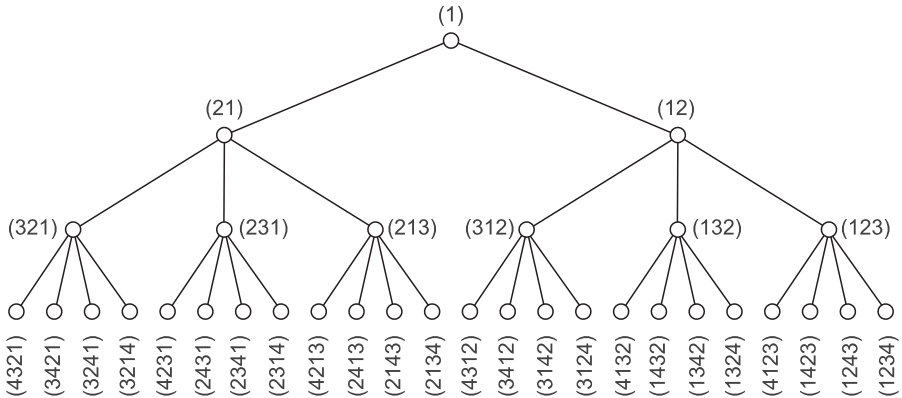


Рис. 6.6. Генерирование перестановок путем умножения,  $n = 4$

### 1. Идея

Мы можем отождествить перестановки с узлами, как это показано надписями на рис. 6.6. На самом верху находится 1 сама по себе. Мы можем получить две перестановки множества  $\{1, 2\}$ , написав сначала 2 с левой стороны от 1, а затем с правой. Аналогично шесть перестановок множества  $\{1, 2, 3\}$  могут быть получены, если начать с одной из перестановок (2, 1) или (1, 2) и затем включать 3 в одну из трех возможных позиций (слева, в середине или справа). Теперь задача генерирования перестановок множества  $\{1, 2, \dots, k\}$  может быть сформулирована таким образом:

*Возьмите данную перестановку из  $\{1, 2, \dots, k - 1\}$  элементов и поместите ее элементы в список. Включите  $k$  по очереди в каждую из  $k$  возможных позиций в этом списке, получив тем самым  $k$  различных перестановок из  $\{1, 2, \dots, k\}$  элементов.*

Этот алгоритм иллюстрирует использование рекурсии для завершения задачи, которая была временно отложена. Другими словами, мы напишем процедуру, которая сначала включит 1 в пустой список, а затем использует рекурсивный вызов для включения в этот список оставшихся чисел от 2 до  $n$ . Этот первый рекурсивный вызов включит 2 в список, содержащий только 1, и отложит дальнейшие включения до следующих рекурсивных вызовов. На  $n$ -м рекурсивном вызове будет, наконец, включено целое число  $n$ . Итак, начав с древовидной структуры в качестве наглядного представления нашего процесса, мы разработали алгоритм, который преобразует это дерево в дерево рекурсии.

## 2. Детализация

Давайте сформулируем наш алгоритм несколько более формально. Мы вызываем процедуру как

Permute(1,  $n$ )

и это означает «включить все целые числа от 1 до  $n$  для построения всех  $n!$  перестановок». Когда наступит время включения целого числа  $k$ , оставшаяся задача выглядит так:

```

procedure Permute( $k$ ,  $n$ );                                { Перестановки }
{ Требуется, чтобы числа от 1 до  $k - 1$  уже были в списке перестановок;
  включает целые числа от  $k$  до  $n$  в список перестановок. }
begin                                                       { процедура Permute }
  for каждой из  $k$  возможных позиций в списке do
    begin
      Включить  $k$  в заданную позицию;
      if  $k = n$  then
        ProcessPermutation
      else
        Permute( $k + 1$ ,  $n$ );
      Удалить  $k$  из данной позиции
    end
  end;

```

эскиз

Процедура ProcessPermutation выполняет любое требуемое действие над полным комплектом перестановок из  $\{1, 2, \dots, n\}$ . Может быть, нам достаточно распечатать полученный результат, а может быть мы захотим использовать его в качестве входных данных для какой-либо другой задачи.

## 3. Общая процедура

Для преобразования этого алгоритма в Pascal-программу мы должны изменить некоторые из наших обозначений. Мы используем объявление

**type** listentry = integer

а именем perm обозначаем список, содержащий генерируемую перестановку. Вместо  $k$  мы введем переменную newentry, которая будет обозначать включаемое целое число, и будем писать degree вместо  $n$  в качестве обозначения полного числа перестанавливаемых объектов. Тогда мы получаем такую процедуру:

```

procedure Permute(newentry: integer; degree: position; var perm: list);
                                                    { Перестановки }
{ Pre:   Список perm содержит перестановки с элементами от 1
          до newentry – 1.
  Post:   Все перестановки по числу элементов degree, которые можно
          построить на основе данной перестановки, сконструированы
          и обработаны.
  Uses:   Использует процедуру Permute рекурсивно, также
          ProcessPermutation. }
var current: position;                        { индекс для просмотра списка perm }
begin                                       { процедура Permute }
  for current := 1 to ListSize(perm) + 1 do { для всех возможных позиций }
  begin
    InsertList(current, newentry, perm);
    if newentry = degree then
      ProcessPermutation(perm)
    else
      Permute(newentry + 1, degree, perm);
    DeleteList(current, newentry, perm)
  end
end;                                       { процедура Permute }

```

Включение этой процедуры в рабочую программу мы оставим для самостоятельного проекта. В качестве пакета обработки списков можно использовать любую реализацию из раздела 6.2.

#### 4. Структуры данных: оптимизация

Число  $n!$  возрастает очень быстро с ростом  $n$ ; соответственно быстро возрастает и число перестановок. Здесь мы сталкиваемся с относительно редкой ситуацией, когда оптимизация программы по скорости ее выполнения может оправдать затраченные на это усилия, особенно, если мы хотим использовать программу для изучения интересных вопросов, касающихся генерирования перестановок.

Давайте теперь подумаем над представлением данных с точки зрения максимального увеличения скорости работы программы, даже ценой уменьшения ее наглядности. Для хранения переставляемых чисел мы используем список. Этот список является глобальным для рекурсивных вызовов процедуры, другими словами, имеется лишь одна главная копия этого списка, и каждый рекурсивный вызов обновляет элементы этой главной копии. Поскольку мы должны непрерывно включать и удалять элементы списка, связанная память будет более гибкой, чем хранение элементов в непрерывном списке. Однако полное число элементов в списке никогда не превышает  $n$ , поэтому мы (возможно) повысим эффективность, поместив связный список внутрь массива вместо того, чтобы использовать динамическое выделение памяти. В этом случае наши связи представляют собой целочисленные индексы относительно начала массива. Далее, в случае массива индекс каждого элемента после его назначения будет совпадать со значением включаемого числа, и поэтому отпадает необходимость в хранении этого числа. В результате в массиве можно хранить только индексы.

Такое представление перестановок как связного списка внутри массива изображено на рис. 6.7. Верхняя часть этого рисунка показывает перестановку (3, 2, 1, 4) в виде связного списка; ниже эта перестановка представлена как связный список в массиве. Последняя, третья часть рисунка показывает оптимизированный результат, получаемый, если удалить фактические переставляемые элементы, поскольку они совпадают с их позициями в массиве, и оставить в массиве только связи, описывающие связный список.

Представление перестановки (3214):

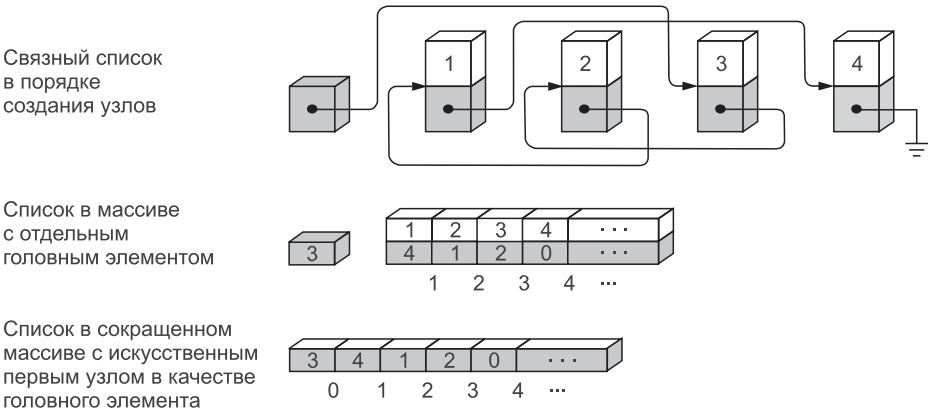


Рис. 6.7. Перестановки в виде связного списка в массиве

искусственный узел

Включения и удаления можно еще упростить, если мы поместим в начало списка искусственный первый узел, чтобы включения и удаления в начале (фактического) списка можно было выполнять точно так же, как и в других позициях, т. е. всегда как включение или удаление после предыдущего узла. В результате мы повышаем эффективность, используя все эти особые условия и поместив алгоритмы включения и удаления в процедуру `Permute` вместо того, чтобы использовать отдельный пакет обработки списков.

5. Окончательная программа

Со всеми этими улучшениями мы можем написать оптимизированный вариант процедуры `Permute`.

```
procedure Permute (newentry, degree: index; var perm: linkedpermutation);
{ Перестановки }
{ Pre: Список perm содержит связные перестановки с элементами от 1 до newentry - 1.
  Post: Все перестановки с числом элементов degree, которые можно построить на основе данной перестановки, сконструированы и обработаны.
  Uses: Использует процедуру Permute рекурсивно, также ProcessLinkedPermutation. }
```

```

var current: index;           { индекс для просмотра списка perm }
begin                         { процедура Permute }
  current := 0;
  repeat
    perm [newentry] := perm [current];
    perm [current] := newentry;           { сначала включим newentry
                                          в список после current }

    if newentry = degree then
      ProcessLinkedPermutation(perm)
    else
      Permute(newentry + 1, degree, perm);
      perm [current] := perm [newentry];   { удалим из списка newentry }
      current := perm [current]           { продвинем current на одну позицию }
  until current = 0               { current = 0 в конце списка }
end;                             { процедура Permute }

```

Главная программа почти что ничего не делает, за исключением объявлений и инициализирования процесса.

```

program PermutationGeneration(input, output); { Генератор перестановок }
{ Pre: Пользователь предоставляет целочисленную степень
      перестановок degree.
  Post: Все перестановки с числом элементов degree сконструированы
        и обработаны.
  Uses: Использует процедуру Permute рекурсивно, также
        ProcessPermutation. }
const maxdegree = 20;           { максимально допустимое число
                                элементов в списке }
type index = 0 .. maxdegree;    { 0 будет всегда обозначать индекс nil }
    linkedpermutation = array [index] of index; { рабочее пространство
                                                для связного списка }
var perm: linkedpermutation;     { конструируемая перестановка }
    degree: integer;             { число переставляемых элементов }
begin                           { главная программа PermutationGeneration }
  write("Число переставляемых элементов?");
  readln(degree);
  if (degree < 1) or degree > maxdegree then
    writeln("Число должно быть между 1 и ", maxdegree)
  else begin
    perm [0] := 0;               { сначала сделаем список пустым }
    Permute(1, degree, perm);    { включим элементы от 1 до degree }
  end
end.                             { главная программа PermutationGeneration }

```

Вспомним, что массив L описывает связный список указателей и не содержит переставляемые объекты. Если, например, мы хотим напечатать перестановки целых чисел 1, ..., n, вспомогательная процедура приобретает такой вид:

```

procedure ProcessLinkedPermutation(var perm: linkedpermutation);
                                { Обработать связные перестановки }
{ Pre: Список perm содержит перестановки в связной форме.
  Post: Перестановки строка за строкой выведены на терминал. }
var current: index;             { используется для просмотра связного списка }

```



```

begin                                     { процедура ProcessLinkedPermutation }
  current := 0;
  while perm [current] <> 0 do
    begin
      write(perm [current]);
      current := perm [current]
    end;
  writeln
end;                                     { процедура ProcessLinkedPermutation }

```

Наконец мы получили законченную программу, одну из наиболее эффективных программ для генерирования перестановок с большой скоростью.

## Программные проекты 6.6

- P1.** Завершите программу генерирования перестановок, которая использует общие пакеты обработки списков, написав главную программу и процедуру `ProcessPermutation`, выводящую на экран полученные перестановки. Протестировав программу, подавите вывод перестановок и подключите пакет анализа процессорного времени из приложения С. Сравните производительность своей программы для каждого из пакетов обработки списков, реализованных в разделе 6.2. Сравните также производительность вашего варианта и оптимизированного варианта, приведенного в тексте.
- P2.** Модифицируйте начальный вариант процедуры `Permute`, чтобы позиция, занимаемая каждым числом, не изменялась более, чем на 1 влево или вправо при генерировании следующей перестановки. [Это упрощенная форма одного из правил кампанологии (последовательных изменений тона церковных колоколов).]

## Подсказки и ловушки

1. Не путайте непрерывные списки и массивы.
2. Выбирайте структуры данных в процессе разработки алгоритмов и избегайте преждевременных решений.
3. Всегда обдумывайте крайние случаи и аккуратно их обрабатывайте. Трассируйте свои алгоритмы, чтоб определить, что случится, если структура данных окажется полной или пустой.
4. Не пытайтесь оптимизировать свой код, пока он не будет работать идеально, и вообще занимайтесь оптимизацией только если повышение эффективности действительно представляется необходимым. Сначала испытайте простые реализации своих структур данных. Переходите к более сложным реализациям, только если простые оказываются неэффективными.
5. Работая с обобщенными списками, прежде всего точно определите, какие операции вам потребуются, затем выберите реализацию, которая позволит выполнять эти операции простейшим способом.

6. Делая выбор между связной и непрерывной реализацией списков, примите во внимание операции, которые вы будете выполнять над списками. Связные списки обладают большей гибкостью для операций включения, удаления и перестановки; непрерывные списки предоставляют возможность произвольного доступа.
7. Непрерывные списки обычно требуют меньше компьютерной памяти, компьютерного времени и программистского труда, если элементы списка невелики, а алгоритмы просты. Если список содержит большие записи, связные списки обычно экономят место, время и труд программистов.
8. Динамическая память и указатели позволяют программе автоматически настраиваться на широкий диапазон размеров приложений и обеспечивают гибкость в плане выделения памяти для самых разных структур данных. Статическая память (массивы и индексы) иногда оказывается более эффективной для приложений, размер которых может быть полностью определен заранее.
9. Советы по программированию связных списков в динамической памяти приведены в приложении D.
10. Избегайте усложнений ради усложнений. Если простой метод дает удовлетворительное решение, применяйте его.
11. Не изобретайте велосипед. Если готовая процедура адекватна для вашего приложения, используйте ее.

## Обзорные вопросы

- 6.1
  1. Какие операции, возможные для обобщенных списков, допустимы также для очередей?
  2. Перечислите три операции, возможные для обобщенных списков, но недопустимые для стеков или очередей.
- 6.2
  3. Проще ли включить новый узел перед или после заданного узла в связанном списке?
  4. Если элементы списка являются целыми числами (по одному слову каждый), сравните пространство памяти, требуемое, если (а) список хранится в виде непрерывной реализации в массиве, заполненном на 90 процентов, (б) список хранится в виде непрерывной реализации в массиве, заполненном на 40 процентов, (с) список хранится в виде связанного списка (в котором каждый указатель занимает одно слово).
  5. Повторите сравнительный анализ из предыдущего упражнения, если элементами списка являются записи по 200 слов каждая.
  6. Что такое переменная-псевдоним, и в чем ее опасность?
  7. Каков основной недостаток связных списков в сравнении с непрерывными списками?
- 6.5
  8. Каковы некоторые из оснований для реализации связных списков в массивах с индексами вместо динамической памяти с указателями?

## Литература для дальнейшего изучения

Ссылки, касающиеся стеков и очередей, полезны также и для настоящей главы. В особенности для многих тем, касающихся манипуляций со списками, лучшим источником дополнительной информации, исторических замечок и математического анализа является [Knuth], том 1. (см. ссылку в Гл. 3). В этой книге, однако, не рассматриваются принципы абстракции данных.

Дополнительные детали, касающиеся реализации типов указателей в языке Pascal, лучше всего найти в справочном руководстве по языку Pascal для вашей системы. Некоторые реализации языка Pascal либо не используют стандартных процедур `new` и `delete`, либо предоставляют альтернативные методы, которые могут быть более эффективными. Поскольку детали этих методов зависят от системы, мы не можем описать их здесь более подробно.

Алгоритм генерирования перестановок путем включения в связный список был опубликован в журнале *ACM SIGCSE Bulletin* 14 (February 1982), 92–96. Полезные обзоры многих методов генерирования перестановок можно найти в работах

R. Sedgewick, «Permutation generation methods», *Computing Surveys* 9 (1977), 137–164; addenda, *ibid.*, 314–317.

R. W. Topor, «Functional programs for generating permutations», *Computer Journal* 25 (1982), 257–263.

Приложения перестановок к кампанологии порождают ряд интересных проблем для компьютерного изучения. Прекрасный источник более подробной информации по этому поводу:

F. J. Budden, *The Fascination of Groups*, Cambridge University Press, Cambridge, England, 1972, pp. 451–479.

# Глава 7

## Поиск

---

В этой главе обсуждаются проблемы поиска в списках с целью нахождения конкретного элемента. Наша дискуссия будет посвящена двум хорошо известным алгоритмам: последовательный поиск и двоичный поиск. Мы разработаем несколько довольно сложных математических средств, используемых как для демонстрации правильности алгоритмов, так и для вычисления объема выполняемой ими работы. Эти математические средства включают в себя инварианты цикла, деревья сравнений и нотацию  $O$  большого. Наконец, мы найдем нижние границы условий, при которых любой алгоритм поиска должен выполнить по меньшей мере столько же работы, как и двоичный поиск.

### 7.1. Введение и обозначения

Извлечение информации является одной из важнейших прикладных задач компьютеров. Мы указываем имя и запрашиваем соответствующий список телефонных переговоров. Мы указываем номер счета и запрашиваем перечень транзакций, осуществленных по этому счету. Мы указываем имя или номер служащего и запрашиваем его персональное дело.

#### 1. Ключи

В этих и массе других примеров мы указываем некоторую порцию информации, которую мы будем называть *ключом*, и просим найти запись, содержащую другую информацию, связанную с этим ключом. Мы должны предусмотреть варианты, когда с данным ключом связаны несколько записей, и, наоборот, когда для этого ключа никаких записей не существует (рис. 7.1).

ключи и записи

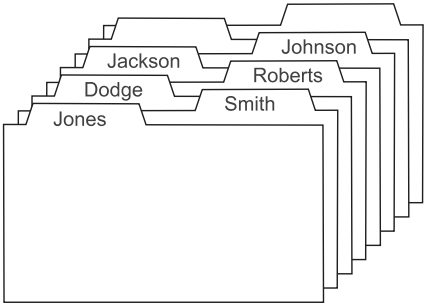


Рис. 7.1. Записи и их ключи

## 2 Анализ

Поиск ключей с целью локализации записей часто является наиболее затратным, в плане времени процессора, действием программы, и поэтому способ организации записей и выбор метода поиска может оказать существенное влияние на производительность программы. По этой причине мы потратим в этой главе некоторое время на изучение скорости выполнения разрабатываемых нами алгоритмов. Мы увидим, что определение числа сравнений одного ключа с другим дает нам превосходную оценку общего объема работы, выполняемой алгоритмом, и общего времени, затрачиваемого компьютером при выполнении программы.

## 3. Внешний и внутренний поиск

Проблема поиска естественно распадается на два случая. Если записей много, да они еще велики по объему, тогда записи придется сохранять в дисковых или ленточных файлах, т. е. во внешней компьютерной памяти. Этот случай называют **внешним** поиском. В другом случае просматриваемые записи могут целиком храниться в оперативной памяти компьютера. Такой случай называют **внутренним** поиском. В этой книге мы будем рассматривать только внутренний поиск. Хотя многие из методов, представленных нами в этой и последующих главах, полезны и для внешнего поиска, детальное изучение методов внешнего поиска выходит за рамки этой книги.

## 4. Реализация на языке Pascal

При реализации наших программ на языке Pascal мы будем следовать некоторым соглашениям. Нас в этой главе будут интересовать только непрерывные списки. Поиск в связных структурах является основной темой главы 10, и мы отложим рассмотрение связных структур до этой главы.

Поскольку мы всегда будем осуществлять поиск в непрерывном списке, мы дадим ему обобщенное имя *L*. Объекты, которые мы называем записями, действительно будут записями языка Pascal, и они будут служить элементами списка *L*. Типу языка Pascal, которому будут принадлежать эти записи, мы дадим имя *listentry*, как это было и раньше. Одно из полей каждой записи списка будет отведено под ключ и иметь тип *keytype*. Таким образом, мы принимаем, что программа будет иметь дело с объявлением такого рода:

```

type
  keytype = ...;
  listentry = record
    ...           { различные компоненты }
    key: keytype;
    ...           { еще компоненты }
  end;
```

Типичные объявления для типа ключа могут быть такими:

```

keytype = real;
keytype = integer;
keytype = string;
keytype = packed array [1 .. 8] of char;
```

только  
непрерывные  
списки

примеры

Для списка, в котором будет осуществляться поиск (мы будем называть его *списком поиска*), мы используем стандартные объявления непрерывной реализации списков из раздела 6.2.1. Ключ, по которому осуществляется поиск, будет всегда называться **мишенью** поиска.

мишень

## 5. Параметры

параметры

Каждая процедура поиска, которую мы напишем, будет иметь два входных параметра. Этими параметрами являются мишень и список поиска. Она также будет иметь два выходных параметра. Первым из них будет булева переменная *found*, указывающая, был ли в списке поиска найден элемент с указанным ключом-мишенью. Если поиск закончился успешно, выходной параметр *location* будет позицией в списке, где найдена мишень. Если поиск закончился неуспешно, выходной параметр *location* может иметь неопределенное значение или значение, зависящее от используемого метода.

# 7.2. Последовательный поиск

## 1. Алгоритм и процедура

Вне сомнения, простейшим способом поиска будет начать просмотр списка с одного конца и сравнивать мишень с ключами записей до тех пор, пока не будет найден требуемый ключ или мы не дойдем до другого конца списка. Это будет нашим первым методом.

```

procedure SequentialSearch (var L: list; target: keytype; var found: Boolean;
                             var location: position);
    { Последовательный поиск }
{ Pre: Непрерывный список L уже создан.
Post: Если элемент списка L имеет ключ, совпадающий с ключом
поиска target, тогда переменная found становится равной true,
а переменная location содержит позицию этого элемента.
В противном случае found становится равной false, а location
не имеет содержательного значения. }
begin                                     { процедура SequentialSearch }
    found := false;
    location := 1;
    while (not found) and (location <= L.count) do
        if L.entry [location].key = target then
            found := true
        else
            location := location + 1
    end;                                     { процедура SequentialSearch }

```

Цикл **while** в этой процедуре осуществляет продвижение по списку до тех пор, пока не будет найдена мишень, на чем он и завершается. Если поиск оказался неуспешным, тогда значение переменной *found* остается равным *false*, а переменная *location* сдвигается за конец списка (вспомним, что при неуспешном поиске значение *location* может оказаться неопределенным).

Поскольку *location* может сдвинуться за конец списка, необходимо объявить тип этой переменной как *integer*, а не *position*.

## 2. Анализ

Давайте теперь оценим объем работы, необходимой для последовательного поиска, чтобы позже мы могли сравнить его характеристики с характеристиками других методов. Предположим, что последовательный поиск осуществляется в длинном списке. Предложения, находящиеся вне цикла, выполняются только однажды и добавляют незначительное время к времени выполнения цикла. Для каждого шага цикла один ключ сравнивается с ключом-мишенью, выполняется еще несколько предложений, после чего осуществляется проверка нескольких условий. Все эти действия выполняются по одному разу в каждом шаге цикла, т. е. для каждой операции сравнения.

Отсюда все действия, которые мы должны учитывать, непосредственно связаны со сравнением ключей. Если кто-то еще, используя тот же метод, напишет процедуры по-другому, тогда отличия в программном подходе повлекут за собой отличия во времени выполнения программы. Однако все эти варианты по-прежнему будут выполнять одно и то же количество операций сравнения ключей. Если длина списка изменяется, работа, выполняемая любой реализацией метода поиска, также изменится пропорционально.

Мы изучим вопрос, как в разных случаях число операций сравнения ключей зависит от длины списка. Это рассмотрение даст нам чрезвычайно полезную информацию об алгоритме, информацию, которой мы сможем с равным успехом пользоваться независимо от реализации и деталей программирования.

Если мы хотим оценить, сколько компьютерного времени потребует последовательный поиск, или сравнить его с каким-либо другим методом, тогда именно знание числа операций сравнения ключей даст нам наиболее ценную информацию — фактически более ценную, чем полное время выполнения программы, которое слишком сильно зависит от того, какой вариант программного кода используется или на какой машине мы работаем.

Независимо от того, какой алгоритм поиска мы разработаем, мы можем высказать утверждение, схожее с нашей фундаментальной предпосылкой при анализе алгоритмов поиска: на полный объем работы влияет число операций сравнения ключей, выполняемых данным алгоритмом.

*Для анализа поведения алгоритма, выполняющего сравнение ключей, мы будем использовать счетчик этих операций сравнения как меру выполненной работы.*

Сколько операций сравнения ключей выполняет алгоритм последовательного поиска, когда он приложен к списку из  $n$  элементов? Поскольку при последовательном поиске выполняется сравнение мишени с каждым ключом по очереди, ответ зависит от положения мишени в списке. Если процедура находит мишень в первой позиции списка, она выполняет только одну операцию сравнения. Если мишень находится на втором месте, процедура выполняет две таких операции. Если поиск завершился неуспешно, тогда мишень будет сравниваться со всеми элементами списка, выполнив, таким образом,  $n$  операций сравнения.

Наш вопрос, следовательно, имеет несколько ответов, зависящих от того, будет ли найдена мишень в списке и если да, то где именно. Если поиск дал отрицательный результат, ответом будет  $n$  операций сравнения. Наилучшей производительностью успешного поиска будет 1 операция сравнения, наихудшей —  $n$  операций.

усредненное  
поведение

Мы получили весьма полную информацию о производительности последовательного поиска, информацию, фактически слишком детализированную для практического использования, поскольку в общем случае мы не знаем, в каком месте списка может быть найдена мишень. Было бы гораздо полезнее определить *среднее* поведение алгоритма. Однако что мы понимаем под средним поведением? Один разумный способ определения среднего поведения, которым мы всегда будем пользоваться, заключается в том, чтобы выполнить каждый из возможных вариантов один раз и усреднить результат.

оговорка

Заметьте, однако, что это определение может быть очень далеким от реальной ситуации. Например, в типичном тексте не все английские слова встречаются с равной вероятностью. Телефонный оператор получает значительно больше запросов на соединение с большой фирмой, чем звонков в обычную квартиру. Pascal-компилятор встречается со ключевыми словами **if**, **begin** и **end** значительно чаще, чем с ключевыми словами **label**, **downto** и **packed**.

Имеется значительное количество весьма интересных, но исключительно сложных проблем, связанных с анализом алгоритмов, в которых входные данные распределены по некоторому статистическому закону. Эти проблемы, однако, заведут нас слишком далеко в дебри математики, и мы их рассматривать здесь не будем. Мы ограничимся наиболее важным случаем, когда все варианты поиска равновероятны.

В предположении равновероятности вариантов мы можем найти среднее число операций сравнения ключей при последовательном поиске. Мы просто складываем число операций для каждого успешного поиска и делим эту сумму на число элементов списка  $n$ . Результат будет таким:

$$\frac{1 + 2 + 3 + \dots + n}{n}$$

Эта формула преобразована в приложении А таким образом:

$$1 + 2 + 3 + \dots + n = \frac{1}{2} n(n + 1)$$

среднее число  
операций  
сравнения ключей

Отсюда среднее число операций сравнения, выполняемых при (успешном) последовательном поиске, составит

$$\frac{n(n+1)}{2n} = \frac{1}{2} (n+1)$$



### 3. Тестирование

Разумным противовесом теоретическому анализу алгоритмов является эмпирическое тестирование разработанной программы. Мы создаем тестовые данные, запускаем процедуру и сравниваем результаты с выводами теоретического анализа.

Для процедур поиска имеются по меньшей мере две величины, которые стоит вычислить. Это среднее число операций сравнения, полученное в результате многократного повторения поиска, и требуемое для выполнения этих операций время процессора. Давайте разработаем процедуру, с помощью которой можно будет тестировать любые алгоритмы поиска.

Чтобы сохранить общность, мы должны передать процедуру поиска в качестве процедурного параметра (см. приложение D). Мы используем пакет анализа процессорного времени из приложения C, включив в программу переменную *CC* в качестве счетчика сравнения. Эта переменная должна быть глобальной, так как вряд ли целесообразно изменять наши стандартные спецификации процедуры поиска, и в то же время каждая процедура поиска должна инкрементировать *CC* при каждом сравнении ключей. В результате варианты процедур поиска, используемые в тестировании, будут отличаться от стандартных вариантов.

В качестве тестируемых данных мы будем использовать целые числа. Большинство методов поиска, рассматриваемых далее в этой главе, требуют упорядоченных данных, поэтому мы включим целые числа в наш список в возрастающем порядке. Нам интересны и успешные, и неуспешные попытки поиска, поэтому давайте включим в список только нечетные числа, а искать будем как нечетные числа (тогда поиск завершится успешно), так и четные (тогда поиск не даст положительного результата). Если список имеет  $n$  элементов, то поиск успешно завершится для чисел 1, 3, 5, ...,  $2n - 1$ , а неуспешно — для чисел 1, 2, 4, 6, ...,  $2n$ . При такой методике мы протестируем все возможные отказы, включая ключи, меньшие, чем минимальный ключ в списке, большие, чем максимальный, а также попадающие внутрь каждой пары элементов списка. Чтобы придать тестированию более реалистичные черты, мы для выбора мишени используем псевдослучайные числа, получив их с помощью функции *RandomInt* из приложения B. Пользователь предоставляет значение для переменной *searchcount*, определяющей число попыток поиска.

Для того чтобы один и тот же драйвер можно было использовать для исследования различных методов поиска, мы рассматриваем метод поиска как процедурный параметр, обозначенный именем *Look*. Чтобы это было возможно, Turbo Pascal требует, чтобы мы для задания стандартных параметров, используемых во всех алгоритмах поиска, определили тип процедуры

```
type searchmethod = procedure(var L: list; { специфично для Turbo Pascal }  
                                target: keytype; var found: Boolean;  
                                var location: position);
```

Результирующая процедура тестирования приведена ниже.

```

procedure TestSearch(var L: list; Look: searchmethod);
                                { Тестирование поиска }
{ Pre: Метод поиска Look существует.
  Post: Тестирование метода поиска Look завершено, включая анализ
    затрат процессорного времени и подсчет числа сравнений
    ключей.
  Uses: Переменная CC является глобальным счетчиком сравнений,
    инициализируемым здесь и инкрементируемым каждой
    процедурой поиска. Переменная searchcount представляет
    глобальное значение, используемое для определения числа
    повторных поисков. }

var i: integer;
    found: Boolean;
    location: position;
    average: real;
    target: keytype;

begin                                { процедура TestSearch }
  CC := 0;                            { инициализируем счетчик числа сравнений }
  SetTimer;
  if not ListEmpty(L) then begin
    for i:= 1 to searchcount do        { searchcount есть глобальное
                                          число операций поиска }
      begin                            { тест с успешными операциями поиска }
        target := 2 * RandomInt(1, L.count) - 1; { должно быть
                                                    нечетным числом }

        Look(L, target, found, location);
        if not found then writeln('Error: Ключ ', target, ' не найден.')
      end;
        average := CC/searchcount;
      end
    else
      average := 0.0;
      writeln(' Успешный поиск: ', average: 6: 2, ' сравнений. ');
      writeln(' Истекшее время для завершения ', searchcount: 1);
      writeln(' операций поиска равно ', ElapsedTime: 9: 5);
      CC :=0;
      if not ListEmpty(L) then begin
        for i:= 1 to searchcount do begin { тест с неуспешными
                                              операциями поиска }
          target := 2 * RandomInt(0, L.count); { должно быть четным числом }
          Look(L, target, found, location);
          if found then writeln('Error: Ключ ', target, 'найден неверно.')
        end;
          average := CC/(searchcount);
        end;
      else
        average := 0.0;
        writeln(' Неуспешный поиск: ', average: 6: 2, ' сравнений. ');
        writeln(' Истекшее время для завершения ', searchcount: 1);
        writeln(' операций поиска равно ', ElapsedTime: 9: 5);
      end;                                { процедура TestSearch }
  end;

```

Конкретные детали включения этой процедуры в работающую программу мы оставим для самостоятельного проекта.

## Упражнения 7.2

- Е1.** Одной из важнейших проверок любого алгоритма является выяснение, как он работает в крайних, граничных случаях. Определите, что происходит при последовательном поиске, когда
- (a) в списке имеется только один элемент;
  - (b) список пуст;
  - (c) список полон.
- Е2.** Проведите трассировку процедуры последовательного поиска в процессе поиска каждого из ключей в списке, содержащем три элемента. Определите, сколько при этом выполняется операций сравнения, и проверьте правильность формулы для среднего числа сравнений в случае успешного поиска.
- Е3.** Если мы можем предположить, что ключи выстроены в списке в каком-то порядке (например, в числовом или алфавитном), мы получим возможность быстрее завершать неуспешный поиск. Если сначала в списке идет наименьший ключ, тогда мы можем завершить поиск, как только достигнем ключа, большего или равного ключу-мишени. Если мы предположим, что ключ-мишень, отсутствующий в списке, равновероятно входит в некоторый из  $n + 1$  интервалов (перед первым ключом, между парой последовательных ключей или после последнего ключа), каково будет среднее число операций сравнения при неуспешном поиске?
- Е4.** В каждой итерации последовательного поиска проверяются два неравенства, одно при сравнении ключей, чтобы выяснить, не найден ли ключ-мишень, а другое – при сравнении индексов, чтобы определить, не достигли ли мы конца списка. Полезным способом ускорения алгоритма путем устранения второго сравнения является обеспечение условий, при которых ключ-мишень *target* будет в конце концов неминуемо найден. С этой целью список удлиняется на 1 элемент, и в конец списка помещается дополнительный элемент с ключом *target*. Такой элемент, помещаемый в список для обеспечения завершения процесса поиска, называется *сигнальной меткой*. Когда цикл завершается, поиск оказывается успешным, если ключ *target* был найден еще перед последним элементом списка, и неуспешным, если была найдена наша сигнальная метка.
- Напишите Pascal-процедуру, которая реализует идею сигнальной метки в варианте последовательного поиска для непрерывного списка, используя списки, разработанные в разделе 6.2.1.
- Е5.** Найдите число операций сравнения ключей, выполняемых процедурой, написанной в упражнении Е4, для:
- (a) неуспешного поиска;
  - (b) наилучшего случая успешного поиска;
  - (c) наихудшего случая успешного поиска;
  - (d) среднего случая успешного поиска.

сигнальная метка

## Программные проекты 7.2

- P1.** Напишите программу для тестирования последовательного поиска, а также других методов поиска, которые будут рассмотрены позже.

Используйте при этом списки, разработанные в разделе 6.2.1. Вам придется предусмотреть подходящие объявления для создания списка и помещения в него ключей. Пусть ключи будут нечетными числами от 1 до  $n$ , где значение  $n$  задает пользователь. Успешные операции поиска могут тестироваться путем поиска нечетных чисел, а неуспешные — путем поиска четных чисел. Для выполнения фактического тестирования процедуры поиска используйте процедуру `TestSearch` из текста. Модифицируйте процедуру последовательного поиска так, чтобы она обновляла счетчик `CS` при каждом сравнении ключей. Напишите соответствующую процедуру `Introduction` и драйвер с меню. Пока что в меню будут только пункты заполнения списка заданным пользователем количеством элементов, тестирования процедуры `SequentialSearch` и выхода. Позже в меню будут добавлены новые пункты, соответствующие другим методам поиска.

Определите, сколько операций сравнения выполняется в случае успешного и неуспешного поиска и сравните полученные вами результаты с анализом в тексте.

Запускайте свою программу для репрезентативных значений  $n$ , например,  $n = 10$ ,  $n = 100$ ,  $n = 1000$ .

- P2.** Возьмите программу драйвера, написанную в проекте P1 для процедуры тестирования поиска и включите в нее вариант последовательного поиска с сигнальной меткой (см. упражнение E4). Определите, для различных значений  $n$ , какой из вариантов поиска, с сигнальной меткой или без нее, оказывается быстрее. Найдите точку пересечения для этих двух вариантов (если она существует). Другими словами, в какой точке дополнительное время, необходимое для включения сигнальной метки в конец списка оказывается тем же, что и время, необходимое для дополнительного сравнения индексов в варианте без сигнальной метки?
- P3. (a)** Напишите вариант процедуры последовательного поиска для связанного списка, разработанного в разделе 6.2.2 (b). Напишите вариант тестирующий процедуры из проекта P1 для связанных списков и используйте его для тестирования поиска в связанном списке.

## 7.3. Гардеробы: проект

В этом разделе вводится абстрактный тип данных, служащий базисом для интересного продолжающегося проекта, показывающего, как можно использовать различные структуры данных и алгоритмы.

### 7.3.1. Введение и спецификации

определение

Идея гардероба проста: индивидуум, несущий громоздкий плащ, отдает этот плащ гардеробщику, получая взамен небольшой жетон, гораздо более легкий, чем плащ. С течением времени индивидуум возвращает жетон гардеробщику и получает плащ назад. В компьютерной системе также могут быть большие записи, которые необходимо держать в состоя-

нии готовности, но ненужные в течение некоторого времени. Вместо того, чтобы передавать эти большие записи от задачи к задаче или от процедуры к процедуре, будет лучше поместить их в гардероб, получить взамен жетон, и передавать от задачи к задаче только этот жетон (в вычислительной технике такой жетон называется тегом). При необходимости задача может, воспользовавшись тегом, затребовать соответствующую запись.

копии тега

Разумеется, между настоящим гардеробом для плащей и пальто и компьютерной системой имеются различия. Для компьютерной задачи создать копию тега не составляет никакого труда, в то время как сделать дубликат жетона для гардероба весьма затруднительно. Отсюда следует, что программа должна принимать специальные защитные меры, чтобы не дать возможность задаче затребовать с помощью тега-дубликата плащ, который уже был изъят другой задачей и, следовательно, не находится в гардеробе.

разница между  
ключами и тегами

Теги весьма напоминают ключи, используемые при поиске в списке. И те, и другие служат для нахождения записей, содержащих другую информацию. Между ними, однако, имеется и существенное различие: ключ является внутренней частью записи, помещаемой в нее клиентской программой; тег же назначается компьютерной системой в процессе обслуживания гардероба.

Давайте теперь специфицируем более точно интерфейс между компьютерным гардеробом и использующей его программой и с помощью этого интерфейса перечислим все операции по обслуживанию гардероба.

Клиент (вызывающая программа) должен содержать четыре объявления:

- Константа `maxcoatroom` задает максимальное количество плащей в гардеробе. Эта константа используется только для реализаций гардероба в виде списка. Для реализаций (с динамической памятью), у которых отсутствует внутренне встроенный максимальный размер, эта константа использоваться не будет.
- Тип `coat` представляет собой запись или другую структуру, зависящую от приложения. Что именно содержится в переменной типа `coat`, сколь велика она, или как ее следует обрабатывать, неизвестно, и вообще не должно интересовать систему обслуживания гардероба.
- тип `pcosat`, образуемый посредством операции приравнивания типов `pcosat = ↑coat` устанавливает один уровень *косвенности*. Мы не будем фактически перемещать плащи между гардеробом и прикладной программой; мы будем только *указывать* на плащи, поэтому фактически в гардеробе будут храниться объекты (элементы) типа `pcosat`. Если `p` есть переменная типа `pcosat`, тогда мы всегда можем получить сам плащ (`coat`) с помощью операции снятия ссылки `p↑`.
- эквивалентность типов `tag = integer` позволяет достичь единообразия поведения всех гардеробов, заставляя их возвращать целочисленные теги. В этом случае программа может как читать, так и записывать значения тегов.

Оставшиеся объявления типов и переменных, так же, как объявления всех процедур и функций для обслуживания гардеробов, следует разместить в отдельном модуле или включаемом файле. В этом случае при изменении реализации не возникнет необходимости внесения каких-либо изменений в прикладную программу.

Первое объявление, которое следует поместить в пакет гардероба — это объявление типа `coatroom`, т. е. типа самого гардероба. В этой главе `coatroom` обычно будет представлять собой список; в дальнейших главах мы будем реализовывать гардеробы с помощью других структур данных.

Далее приведены операции обслуживания всего гардероба (CR от `coatroom`, гардероб).

**procedure** CreateCR (**var** CR: `coatroom`); { Создать CR }  
*предусловие:* Отсутствует.  
*постусловие:* Гардероб CR существует и инициализирован так, что в нем находятся 0 элементов. Все возможные значения тегов недействительны для CR.

**procedure** ClearCR (**var** CR: `coatroom`); { Очистить CR }  
*предусловие:* CR уже создан.  
*постусловие:* Гардероб CR очищен. Все возможные значения тегов недействительны для CR. CR продолжает существовать, но в нем находятся 0 элементов.

**function** IsEmptyCR (**var** CR: `coatroom`): Boolean; { CR пуст? }  
*предусловие:* CR уже создан.  
*постусловие:* Функция возвращает `true` или `false`, в зависимости от того, содержит ли CR 0 указателей на плащи или любое другое количество. CR остается без изменений.

**function** IsFullCR (**var** CR: `coatroom`): Boolean; { CR полон? }  
*предусловие:* CR уже создан.  
*постусловие:* Функция возвращает `true` или `false`, в зависимости от того, содержит ли CR `maxcoatroom` указателей на плащи или меньшее количество. Если в CR не существует внутреннего ограничения на количество сохраняемых указателей на плащи, то функция во всех случаях возвращает значение `false`. CR остается без изменений.

**function** SizeCR (**var** CR: `coatroom`): integer; { Размер CR }  
*предусловие:* CR уже создан.  
*постусловие:* Функция возвращает число указателей на плащи, содержащихся в настоящий момент в гардеробе CR. CR остается без изменений.

Далее приведены подпрограммы манипулирования с тегами и указателями на плащи.

```
procedure CheckCR (p: pcoat; var CR: coatroom; var t: tag);
                                { Сдать в CR }
```

*предусловие:* p указывает на плащ coat. CR уже создан и не полон.

*постусловие:* Указатель p на плащ coat добавлен в гардероб CR. Размер CR увеличен на 1. Образованный тег t сделан допустимым для CR.

```
procedure InspectCR (t: tag; var CR: coatroom; var p: pcoat);
                                { Проверить CR }
```

*предусловие:* CR уже создан; t является допустимым тегом для CR.

*постусловие:* p становится указателем на плащ coat, который был сдан в гардероб, когда t стал допустимым тегом для CR. CR и t остаются без изменений. Плащ coat остается сданным, а тег t допустимым.

```
procedure ClaimCR (var t: tag; var CR: coatroom; var p: pcoat);
                                { Получить из CR }
```

*предусловие:* CR уже создан; t является допустимым тегом для CR.

*постусловие:* p становится указателем на плащ coat, который был сдан в гардероб, когда t стал допустимым тегом для CR. Этот указатель на coat удаляется из CR; размер CR уменьшается на 1. Тег t становится недопустимым.

```
function IsValidTagCR (t: tag; var CR: coatroom): Boolean;
                                { Тег допустим? }
```

*предусловие:* CR уже создан.

*постусловие:* Функция возвращает true или false в зависимости от того, является ли тег t в настоящий момент допустимым тегом для CR или нет. CR и t не изменяются.

Наконец, приведем две дополнительные процедуры, которые, строго говоря, не требуются для большинства гардеробов, но весьма полезны для отладки и для демонстрационных программ.

```
procedure TraverseCR (var CR: coatroom;                                { Просмотр CR }
                     procedure Visit(t: tag; p: pcoat));
```

*предусловие:* CR уже создан.

*постусловие:* Процедура, обозначенная как параметр Visit(t: tag; p: coat), выполнена однажды для каждого элемента CR, причем соответствующий тег t и указатель p на плащ coat были переданы процедуре Visit. Порядок, в котором обрабатываются теги и указатели на плащи, не определяется.

(Процедурные параметры изучаются в приложении D. Здесь используется синтаксис стандартного Pascal; Turbo Pascal требует объявления типа процедуры.)



**procedure** WriteMethodCR;  
*предусловие:* Отсутствует  
*постусловие:* Записывает часть строки, дающую краткое описание реализации гардероба.

{ Записать метод CR }

параметр **var** и  
параметр-значение

Процедура WriteMethodCR позволяет прикладной программе автоматически идентифицировать используемую ею реализацию гардероба. Ради повышения эффективности CR указан во всех приведенных процедурах и функциях как **var**-параметр. В тех приложениях, где пространство памяти и время копирования, расходуемые при определении CR как параметра-значения, не являются критическими, этот параметр следует определять как значение для всех процедур и функций, которые не изменяют содержимое гардероба.

выбор тегов

Не забывайте, что теги предоставляются процедурой CheckCR в реализации гардероба, а не прикладной программой. Мы решили, что теги будут целочисленными переменными, однако не задали, какие именно целые числа использовать. Одна из возможностей заключается в использовании небольших целых чисел, например, номеров позиций в списке. Такой метод, однако, чреват опасными последствиями, поскольку в этом случае тег выданного уже плаща может быть использован вторично для другого плаща. Другое приложение, используя дубликат тега, может осуществить проверку или получение плаща, ожидая, что это первый плащ, в то время как в действительности уже другой плащ.

случайные теги

Значительно лучшим средством конструирования тегов являются *случайные* числа. В этом случае становится чрезвычайно маловероятным, что тот же самый тег будет использован повторно. Случайные теги защищают реализацию гардероба от ошибок и повышают устойчивость системы.

### 7.3.2. Демонстрационная и тестирующая программы

#### 1. Два рода демонстраций

демонстрация,  
управляемая меню

Нашей первой программой, использующей гардероб, будет управляемая меню демонстрационная программа. Она будет очень похожа на управляемые меню демонстрации для списков, стеков и очередей. Почти единственное дополнение будет заключаться в том, что программа демонстрации работы гардероба должна в нужный момент запрашивать у пользователя номер тега (целое число), а также и содержимое соответствующего этому тегу плаща. Для нашей демонстрации в качестве содержимого плаща вполне подойдет строка символов, вводимая пользователем. Демонстрацию можно написать так, чтобы она передавала в пакет гардероба весь ввод пользователя, или чтобы она выполняла проверку ввода на допустимость, выполняя тем самым защиту от ошибок пользователя.

программа  
тестирования  
в реальных  
условиях

Вторая программа, которую мы напишем, будет представлять собой попытку более реалистичных испытаний поведения гардероба при разных условиях. Сюда будут входить действия по сдаче в гардероб, проверке наличия и получению многих плащей в различном порядке и при



различных условиях. Для придания программе большей реалистичности она будет использовать генератор псевдослучайных чисел.

Давайте теперь разработаем эскиз нашего проекта

## 2. Объявления и инициализация

размер

Запросите у пользователя ввод числа помещаемых в гардероб плащей. (Это число будет очень небольшим для начального прогона и отладки, но для окончательных тестов оно будет составлять тысячи.) Установите для `maxcoatroom` значение 5000.

вывод

Запросите у пользователя, хочет ли он, чтобы транзакции выводились на экран по мере их выполнения. Если ответ будет положительным, программа должна для каждого выполняемого ею действия (сдачи плаща в гардероб, проверки его наличия и т. д.) выводить на экран строку текста. Так естественно поступать при пробных и отладочных прогонах, когда число сдаваемых в гардероб плащей невелико. Для больших тестовых прогонов ответ, очевидно, должен быть отрицательным, и программа в этом случае выведет только итоговые данные, показывающие полное число плащей, сданных в гардероб или полученных оттуда и т. д., а также время процессора, потраченное на выполнение каждого этапа программы. Для хранения информации о том, выводить ли каждую транзакцию или нет, хорошо подходит булева переменная, например, `doprint`. Инициализируйте ее предложением `doprint := UserSaysYes` и используйте затем в предложениях вроде

```
if doprint then writeln('Сданный плащ')
```

последовательные  
номера

Примите для плащей тип целых чисел: `type coat = integer`. В качестве содержимого каждого плаща примите последовательный номер (т. е. первый сданный плащ будет иметь номер 1, второй – 2 и т. д.). Если вы забираете плащ из гардероба (т. е. избавляетесь от него) и затем создаете еще один, продолжайте увеличивать номера плащей; никогда не используйте повторно тот же последовательный номер в одном прогоне программы.

Для исследования времени выполнения каждого этапа программы используйте пакет анализа процессорного времени из приложения С.

## 3. Прослеживание тегов

список тегов

Ваша программа должна будет поддерживать список всех тегов (вспомним, что теги генерируются в реализации, а не в главной программе драйвера), чтобы она могла проверять плащи и забирать их из гардероба после их сдачи туда. Лучшим и наиболее элегантным способом организации такого списка будет использование пакета обработки списков из главы 6. После того как вы создали плащ и сдали его в гардероб, вам надо будет поместить его тег, скажем, `currenttag`, в список. Когда вам понадобится проверить все плащи в гардеробе, вы можете использовать просмотр списка. Для получения всех плащей вы опять же можете использовать просмотр списка, включив в алгоритм одно предложение удаления плаща после (возможно) вывода на экран его

последовательного номера, а в конце процедуры очистить список, поскольку теперь все теги недействительны.

#### 4. Случайное упорядочение

Большинство реализаций гардероба используют для тегов случайные числа, однако чтобы сделать программу еще более реалистичной, полезно время от времени в процессе ее выполнения перемешивать теги в списке. Это удобно сделать следующим образом. Предположим, что теги находятся в списке в позициях от 1 до  $n$ . Для каждой позиции  $i$  от 1 до  $n - 1$  генерируйте случайное число в диапазоне от  $i$  до  $n$ , включительно, и обменяйте теги в позиции  $i$  и в случайной позиции. Такое действие случайным образом перемешивает теги. Ниже приведена процедура, выполняющая эту задачу.

```

procedure Mixup(var L: list);                                { Перемешивание }
{ Pre:   Список L уже создан.
  Post:  Порядок элементов в списке L был перемешан случайным
          образом.
  Uses:  Использует пакет генерации случайных чисел. }
var curposition, newposition: position;
      temp1, temp2: listentry; { используются для обмена двух элементов }
begin                                                { процедура Mixup }
  for curposition := 1 to ListSize(L) - 1 do
    begin
      newposition := RandomInt(curposition, ListSize(L));
      RetrieveList(curposition, temp1, L);
      RetrieveList(newposition, temp2, L);
      ReplaceList(curposition, temp2, L);
      ReplaceList(newposition, temp1, L);
    end
  end;                                              { процедура Mixup }

```

#### 5. Общая структура

Ниже приведены основные этапы программы. Для каждого из них, кроме первого и пятого, с помощью пакета анализа процессорного времени выводите на экран время процессора, затраченное для выполнения данного этапа.

1. Инициализируйте глобальные переменные, получив необходимые данные от пользователя.
2. Создайте гардероб. (Для большинства реализаций требуемое для этого время будет исчезающе малым; для некоторых реализаций это может быть и не так.)
3. Сдайте в гардероб некоторое количество плащей, заданное пользователем.
4. Проверьте наличие в гардеробе всех плащей по одному за раз в порядке следования тегов в списке (совпадающем с порядком сдачи плащей).

5. Используйте процедуру Mixup для перемешивания тегов в списке. После этого опять проверьте по одному все плащи в гардеробе. Время, затраченное на выполнение процедуры Mixup, не должно включаться в анализ этого или любого другого этапа.
6. Еще раз случайным образом перемешайте список. Затем получите все плащи из гардероба. После этой операции гардероб будет пуст, поэтому удалите все теги из списка.
7. Настоящий этап выполняет большое количество транзакций, случайным образом комбинирующих три основные операции над гардеробом. Прежде всего, сдайте в гардероб плащи в количестве, ранее указанном пользователем, и сохраните их теги в списке. (Заметьте, что в начале этого этапа гардероб был пуст.) Запросите у пользователя, сколько следует выполнить транзакций. Внутри цикла, в каждом шаге которого будет выполняться одна транзакция, используйте случайное число для выбора выполняемого действия: сдача плаща, его проверка или получение, причем каждое из этих действий должно выбираться с одинаковой вероятностью, равной одной трети. Если выбранная транзакция оказалась проверкой или получением плаща, случайным образом выберите тег из имеющихся в списке. Если вы забираете плащ из гардероба, не забудьте удалить его тег из списка. Если вы начнете прогон с небольшим количеством плащей в гардеробе, программа, возможно, будет пытаться выполнять проверку или получение из пустого гардероба. Игнорируйте такие транзакции, и позаботьтесь, чтобы они не приводили к ошибкам в программе.

Любопытной проверкой будет выполнение последнего этапа при небольшом числе плащей, но большом количестве транзакций, чтобы можно было сравнить поведение больших и маленьких гардеробов.

### Программные проекты 7.3

- P1. Реализуйте пакет гардероба, используя для гардероба список. Элементами списка будут записи, состоящие из тега и указателя на плащ. Теги должны быть случайными числами (в соответствующем диапазоне, скажем, до  $1000 * \text{maxcloakroom}$ ). Для непрерывного последовательного поиска используйте стандартные операции над списком, что даст вам возможность изменять реализацию списка без необходимости внесения изменений в пакет гардероба. Добавляйте новые сданные плащи в конец списка. При проверке или получении плаща ищите тег в списке методом последовательного поиска (теги являются ключами). Выводимое процедурой WriteMethodCR сообщение может быть чем-то вроде «Реализация неупорядоченного списка».
- P2. Напишите простую управляемую меню демонстрационную программу для операций над гардеробом, которая дает возможность пользователю выбирать операции и передавать запросы пользователя пакету гардероба. Плащи должны представлять собой строки, вводимые пользователем.
- P3. Напишите более сложную управляемую меню демонстрационную программу для операций над гардеробом, которая будет обнаруживать

ошибки пользователя. Если пользователь запрашивает недопустимую операцию, задавая, например, недопустимый тег, сдачу плаща в заполненный гардероб, или получение плаща из пустого гардероба, программа должна выдать предупреждающее сообщение и не передавать неправильный запрос пакету гардероба.

- Р4.** Напишите программу тестирования гардероба, описанного в тексте, в реальных условиях. Для проверки правильности работы программы используйте пакет гардероба на базе списка, но обеспечьте возможность замены этого пакета на другой без внесения изменений в программу. Включите в итоговый отчет вывод процедуры WriteMethodCR с указанием числа транзакций и времени выполнения каждого этапа.

## 7.4. Двоичный поиск

Программы последовательного поиска легко писать, при этом они оказываются эффективными для коротких списков, но для длинных совершенно неприемлемы. Представьте себе, что вы пытаетесь найти имя «Amanda Thompson» в большой телефонной книге путем чтения имени за именем с самого начала книги! Для нахождения элементов в длинных списках разработаны значительно более эффективные методы, требующие, правда, чтобы ключи в списке были упорядочены. Одним из лучших методов является сравнение мишени со средним элементом списка, после чего поиск продолжается лишь в половине списка – левой или правой, в зависимости от того, оказался ли ключ-мишень больше или меньше сравниваемого среднего элемента. Действуя далее тем же способом, мы сокращаем на каждом шаге длину просматриваемого списка вдвое. Всего лишь за 20 шагов мы можем просмотреть список, содержащий более миллиона ключей.

Рассматриваемый нами метод носит название *двоичного поиска*. Для его использования требуется, чтобы элементы списка были скалярного или другого типа, который допускает упорядочение, и чтобы к началу поиска список был уже упорядочен.

### Определение

**Упорядоченным списком** называется список, в котором каждый элемент содержит ключ, причем ключи расположены в порядке. Другими словами, если элемент  $i$  находится в списке перед элементом  $j$ , то ключ элемента  $i$  должен быть меньше или равен ключу элемента  $j$ .

Операции над упорядоченными списками включают в себя все действия, применимые к обычным спискам. Мы имеем также несколько дополнительных операций, например, операции поиска, описанные в этой главе, и упорядоченное включение нового элемента в правильное место, определяемое порядком ключей. Эту операцию мы будем изучать в главе 8, однако здесь приведем простой, хотя и зависящий от реализации, вариант такой процедуры.

```

procedure InsertOrder (var L: list);           { Упорядоченное включение }
{ Pre:   Список L уже создан, является упорядоченным списком
        и не полон.
  Post:  Элемент x включен в списке L в такую позицию, что ключи
        всех элементов в L остались в правильном порядке. }

var
  current: position;
  currententry: listentry;
  found: Boolean;           { было ли найдено место, куда пойдет x? }
begin                     { процедура InsertOrder }
  current := 1;
  found := false;
  while not found do
    if current > listSize(L) then
      found := true;
    else
      begin
        RetrieveList(current, currententry, L);
        if x.key <= currententry.key then
          found := true;
        else
          current := current + 1
        end;
      InsertList(current, x, L)
  end;                     { процедура InsertOrder }

```

Мы предполагаем, что ключи в нашем списке могут сравниваться с помощью операций '<' и '>' (например, ключи являются числами), но алгоритмы нетрудно расширить, чтобы получить возможность обработки ключей, представляющих собой символьные строки.

Двоичный поиск не годится для связанных списков, поскольку он требует переходов взад-вперед между концами списка и его серединой, действие, легко выполняемое в массиве, но медленное применительно к связанному списку.

произвольный  
доступ

### 7.4.1. Разработка алгоритма

Хотя идея двоичного поиска представляется простой и понятной, при ее программировании удивительно легко допустить ошибки. Сам метод был предложен не позже 1946 г., однако первый вариант, свободный от ошибок и ненужных ограничений, появился лишь в 1962 г. В одном из исследований (см. список литературы в конце главы) было показано, что 90 процентов профессиональных программистов допускают ошибки при программировании двоичного поиска, даже работая над алгоритмом достаточно длительное время. Поэтому обратим особое внимание на то, чтобы ошибки не вкрались в наше программирование. Для этого мы должны точно определить, что именно обозначают наши переменные; мы должны точно сформулировать, какие условия будут удовлетворяться перед и после каждого шага цикла, содержащегося в программе; и мы должны проследить за тем, чтобы цикл закончился правильным образом.

опасности

Наш алгоритм двоичного поиска будет использовать два индекса, *top* (верх) и *bottom* (низ), между которыми будет заключен тот участок списка, в котором мы будем искать ключ-мишень. На каждом шаге цикла мы уменьшаем размер этой части списка в два раза. Чтобы облегчить себе задачу прослеживания хода алгоритма, запишем условие, удовлетворение которого мы будем требовать перед каждым следующим шагом цикла в процедуре. Такое утверждение называется *инвариантом цикла*.

инвариант цикла

*Ключ-мишень, в предположении, что он задан, должен находиться между индексами bottom и top, включительно.*

Внутри цикла мы сначала вычисляем индекс *mid*, находящийся на полдороги между *bottom* и *top*, а затем сравниваем ключ-мишень с ключом в позиции *mid*. Мы устанавливаем исходную правильность этого предложения путем присваивания индексу *bottom* значения 1, а индексу *top* — значения *L.count*, совпадающего с количеством элементов в массиве.

завершение

Далее, мы замечаем, что цикл должен завершиться, когда будет выполняться условие  $top \leq bottom$ , другими словами, когда оставшаяся часть списка содержит самое большее один элемент, в предположении, что мы не завершили цикл раньше, найдя искомую мишень. Наконец, мы должны обеспечить движение к завершению, потребовав, чтобы число элементов, остающихся после каждого частичного поиска,  $top - bottom + 1$ , строго уменьшалось в каждом шаге цикла.

При таких условиях можно разработать несколько слегка различающихся вариантов алгоритмов двоичного поиска.

### 7.4.2. Вариант с забыванием

Возможно, простейшим вариантом является тот, в котором мы забываем, что ключ-мишень в принципе может быть найден весьма быстро, и, независимо от того, найдена мишень, или нет, продолжаем делить список, пока не останется 1 элемент.

Наша первая программа так и делает. Мы используем в программе три индекса: *top* и *bottom* будут ограничивать часть списка, которая может содержать мишень *target*, а *mid* будет средней точкой этого оставшегося укороченного списка.

Эта ситуация описывается приведенным ниже рисунком:

bottom		top
< target	?	≥ target

завершение цикла

Заметьте, что предложение **if**, которое разделяет список пополам, не симметрично, поскольку проверяемое условие в каждом шаге помещает среднюю точку в меньший из двух интервалов. С другой стороны, целочисленное деление положительных чисел всегда выполняется с округлением в меньшую сторону. Только эти два факта вместе обеспечивают завершение цикла во всех случаях. Давайте рассмотрим, что происходит к концу поиска. Цикл будет повторяться только пока

$top > bottom$ . Но это условие предполагает, что после вычисления  $mid$  мы всегда имеем

$$bottom \leq mid < top$$

поскольку целочисленное деление всегда выполняет округление вниз. Далее, предложение **if** уменьшает размер интервала от  $top - bottom$  до либо  $top - (mid + 1)$ , либо  $mid - bottom$ , причем оба эти выражения, в силу неравенства, строго меньше чем  $top - bottom$ . Таким образом, в каждом шаге размер интервала строго уменьшается, поэтому цикл рано или поздно закончится.

После завершения цикла мы должны посмотреть, не найден ли ключ-мишень, поскольку все предыдущие сравнения проверяли только неравенства.

В результате мы имеем следующую процедуру, которую, ради простоты, мы сначала напомним в рекурсивной форме, приняв в качестве дополнительных параметров границы подсписка.

```
procedure RecBinary1 (var L: list; target: keytype;
                      bottom, top: integer; var found: Boolean;
                      var location: position);
    { Рекурсивный двоичный поиск 1 }
{ Pre: Непрерывный список L уже создан. Интервал от bottom до top –
диапазон в списке, где должна искаться мишень target.
Post: Если элемент entry в диапазоне от bottom до top в списке L
имеет ключ, совпадающий с target, тогда переменная found
принимает значение true, а переменная location является
позицией одного такого элемента. В противном случае found
принимает значение false, а location неопределена.
Uses: Использует RecBinary1 рекурсивно. }
var mid: integer;
begin                                     { процедура RecBinary1 }
    if bottom < top then begin             { в подсписке более одного элемента }
        mid := (top + bottom) div 2;
        if target > L.entry[mid].key then   { уменьшим подсписк до верхней
                                           половины списка }
            RecBinary1(L, target, mid + 1, top, found, location)
        else                               { уменьшим подсписк до нижней половины списка }
            RecBinary1(L, target, bottom, mid, found, location)
        end
    else if top < bottom then
        found := false;                    { пустой подсписк }
    else begin                             { top = bottom, подсписк из одного элемента }
        found := (target = L.entry[top].key); { установим выходные параметры }
        if found then
            location := top
        end
    end;                                  { процедура RecBinary }
```

Для подгонки параметров под наши стандартные условия мы делаем следующее:



```

procedure RecBinary1Search (var L: list; target: keytype;
                           var found: Boolean; var location: position);
    { Рекурсивный двоичный поиск 1 }
begin
    RecBinary1(L, target, 1, L.count, found, location);
end;
    { процедура RecBinary1Search }

```

Поскольку рекурсия, использованная в этой процедуре, является хвостовой рекурсией, мы можем легко преобразовать ее в итеративный цикл. В то же время мы можем привести параметры в соответствие с другими методами поиска.

```

procedure Binary1Search (var L: list; target: keytype); { Двоичный поиск 1 }
    var found: Boolean; var location: position);
    { Pre: Непрерывный список L уже создан.
      Post: Если элемент entry в списке L имеет ключ, совпадающий
            с target, тогда переменная found принимает значение true,
            а переменная location является позицией одного такого
            элемента. В противном случае found принимает значение false,
            а location неопределена. }
    var
        top,
        bottom,
        mid: integer;
    begin
        top := L.count;
        bottom := 1;
        while top > bottom do
            begin
                mid := (top + bottom) div 2;
                if target > L.entry[mid].key then
                    bottom := mid + 1;
                else
                    top := mid;
            end;
        if top = 0 then
            found := false;
        else
            found := (target = L.entry[top].key);
            if found then
                location := top;
        end;

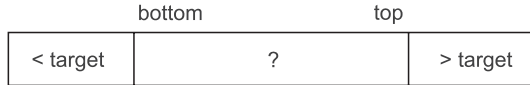
```

### 7.4.3. Распознавание равенства

Хотя процедура Binary1Search реализует простую форму двоичного поиска, она часто будет выполнять ненужные шаги, потому что перед переходом к следующему шагу она не выясняет, найдена мишень или нет. Таким образом, мы имеем возможность сэкономить компьютерное время, разработав вариант, который на каждом шаге проверяет, не найдена ли уже мишень.



Этот вариант описывается таким рисунком.



завершение цикла

Доказательство того, что Binary2Search рано или поздно завершится, оказывается проще, чем такое же доказательство для Binary1Search. В Binary2Search форма предложения **if** внутри цикла гарантирует, что длина интервала уменьшается на каждой итерации более чем в два раза.

В рекурсивной форме имеем такую процедуру:

```

procedure RecBinary2 (var L: list; target: keytype; bottom, top : integer;
                      var found: Boolean; var location: position);
    { Рекурсивный двоичный поиск 2 }
{ Pre: Непрерывный список L уже создан. Параметры bottom и top
  определяют диапазон в списке, в котором выполняется поиск
  мишени target.
Post: Если элемент entry в диапазоне от bottom до top в списке L
  имеет ключ, совпадающий с target, тогда переменная found
  принимает значение true, а переменная location принимает
  позицию одного такого элемента. В противном случае found
  принимает значение false, а location неопределена.
Uses: Использует процедуру RecBinary2 рекурсивно. }
var mid: integer;
    { mid будет позицией target,
    если target будет найдена в L }
begin
  if bottom > top then
    found := false;
  else begin
    mid := (top + bottom) div 2;
    if target = L.entry [mid].key then
      begin
        found := true;
        location := mid
      end
    else
      if target < L.entry [mid].key then
        { ограничим поиск
          нижней половиной списка }
        RecBinary2(L, target, bottom, mid - 1, found, location)
      else
        { ограничим поиск верхней половиной списка }
        RecBinary2(L, target, mid + 1, top, found, location)
      end;
    end;
  end;
    { процедура RecBinary2 }

```

Как и в случае Binary1Search, нам нужно привести параметры в соответствие с нашими стандартными условиями.

```

procedure RecBinary2Search (var L: list; target: keytype; var found: Boolean;
                           var location: position);
    { Рекурсивный двоичный поиск 2 }
begin
  { процедура RecBinary2Search }
  RecBinary2(L, target, 1, L.count, found, location);
end;
    { процедура RecBinary2Search }

```

главный вызов  
RecBinary2

Как и раньше, эту процедуру легко преобразовать в нерекурсивную форму с одними лишь стандартными параметрами:

```

procedure Binary2Search (var L: list; target: keytype); { Двоичный поиск 2 }
    var found: Boolean; var location: position;
{ Pre: Непрерывный список L уже создан.
  Post: Если элемент entry в списке L имеет ключ, совпадающий
        с target, тогда переменная found принимает значение true,
        а переменная location принимает позицию одного такого
        элемента. В противном случае found принимает значение false,
        а location неопределена. }

var
    top,                                { мишень target, если существует,
                                        всегда будет между bottom и top }
    bottom,
    mid: integer;                        { mid будет позицией target,
                                        если target будет найдена в L }
begin
    top := L.count;                     { процедура Binary2Search }
                                        { установим начальную
                                        правильность инварианта }

    bottom := 1;
    found := false;
    while (not found) and (bottom <= top) do { проверим на завершение }
    begin
        mid := (top + bottom) div 2;
        if target = L.entry[mid].key then
            found := true;
        else
            if target < L.entry[mid].key then
                top := mid - 1           { ограничим поиск нижней половиной списка }
            else
                bottom := mid + 1        { ограничим поиск верхней половиной списка }
        end;
        location := mid;                 { предыдущее предложение if гарантирует,
                                        что условие инварианта выполняется }
    end;                                { процедура RecBinary2 }

```

Какой из двух приведенных вариантов двоичного поиска обеспечит меньшее число операций сравнения ключей? Очевидно, Binary2Search, если нам повезет обнаружить мишень близко к началу поиска. Однако каждый шаг Binary2Search требует двух сравнений ключей, в то время как Binary1Search выполняет лишь одну операцию сравнения. Возможно ли, что в случае большого числа шагов Binary1Search выполнит меньше операций сравнения? Для ответа на этот вопрос мы в следующем разделе разработаем новый метод.

сравнение  
методов

## Упражнения 7.4

- E1.** Предположим, что список L содержит целые числа 1, 2, ..., 8. Выполните трассировку шагов процедуры Binary1Search, чтобы определить, какие именно сравнения ключей выполняются для каждой из следующих мишеней: (a) 3, (b) 5, (c) 1, (d) 9, (e) 4.5.
- E2.** Повторите упражнение E1 для процедуры Binary2Search.

- Е3.** [Упражнение повышенной сложности.] Предположим, что  $L_1$  и  $L_2$  есть списки, содержащие  $n_1$  и  $n_2$  целых чисел, соответственно, и оба списка уже упорядочены в числовом порядке.
- (a) Воспользуйтесь идеей двоичного поиска, чтобы описать, как найти медиану  $n_1 + n_2$  целых чисел в комбинированном списке.
  - (b) Напишите процедуру, реализующую ваш метод.

## Программные проекты 7.4

- P1.** Воспользуйтесь программой драйвера из проекта P1 раздела 7.2 и включите в нее в качестве выбираемых вариантов процедуры Binary1Search и Binary2Search. Сравните их производительность по отношению друг к другу, а также с методом последовательного поиска.
- P2.** Включите в программу тестирования из проекта P1 раздела 7.2 оба рекурсивных варианта двоичного поиска. Сравните их производительность с нерекурсивным вариантом двоичного поиска.

## 7.5. Деревья сравнений

определения

*Дерево сравнений* (также называемое *деревом решений* или *деревом поиска*) некоторого алгоритма образуется путем трассировки действий, выполняемых алгоритмом, причем каждое это действие, представляющее операцию сравнения ключей, образует *вершину*, или *узел* дерева (который мы обозначаем кружком). Внутри кружка мы помещаем индекс ключа, с которым выполняется сравнение ключа-мишени. *Ветви* (линии), протянутые вниз от кружка, представляют возможные результаты сравнения; они помечены соответствующим образом. Когда алгоритм завершается, мы рисуем в конце дерева квадратик, который называется *листом*, и помечаем этот лист либо буквой F (Failure, неудача), либо позицией, в которой была найдена мишень. Листья иногда называют *конечными вершинами* или *внешними вершинами* дерева. Оставшиеся вершины называются *внутренними вершинами* дерева.

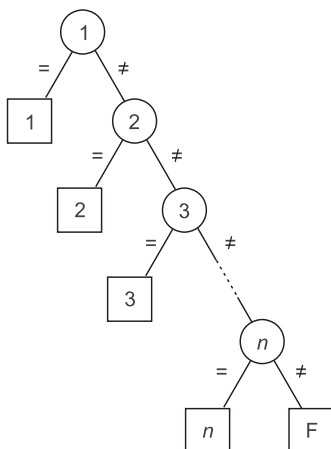
Дерево сравнений для последовательного поиска оказывается особенно простым; такое дерево изображено на рис. 7.2.

определения

Количество операций сравнения, выполненных алгоритмом в конкретном поиске, определяется числом внутренних вершин (кружков), пройденных алгоритмом от верха дерева (называемого *корнем*) вниз к конечному листу. Число ветвей, которые нужно пройти от корня дерева для достижения конкретной вершины, называют *уровнем*, или *степенью* вершины. Так, сам корень имеет уровень 0, вершины непосредственно под ним — уровень 1 и т. д.

Число вершин на самом длинном пути дерева называют *высотой* дерева. Отсюда дерево с единственной вершиной имеет высоту 1. В дальнейших главах мы иногда будем встречаться с пустыми деревьями, т. е. с деревьями, не имеющими ни одной вершины; мы будем считать, что пустое дерево имеет высоту 0.

Чтобы закончить с терминологией, используемой при обсуждении деревьев, мы прибегнем к аналогии с генеалогическим древом, и будем называть вершины, находящиеся непосредственно под вершиной  $v$ , как



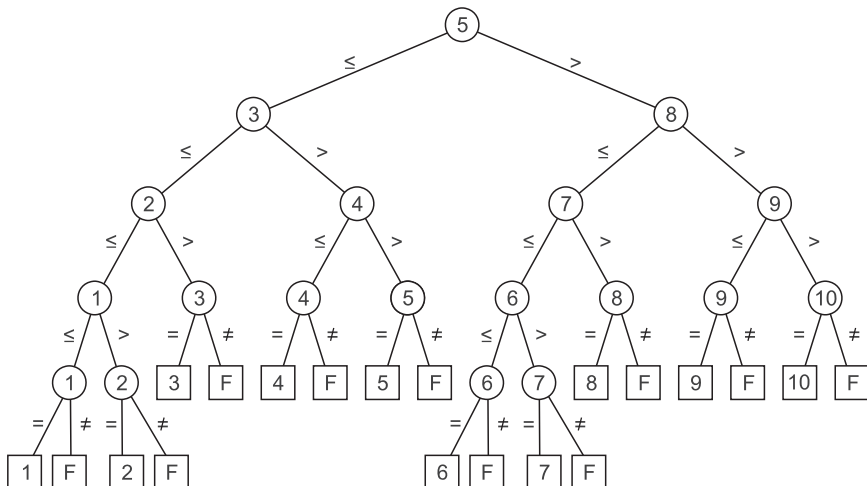
**Рис. 7.2.** Дерево сравнений для последовательного поиска

*дочерние* от  $v$ , а вершину, находящуюся непосредственно над  $v$  — как *родительскую* по отношению к  $v$ .

### 7.5.1. Анализ для $n = 10$

## 1. Форма деревьев

То, что при последовательном поиске в среднем выполняется значительно больше операций сравнения, чем при двоичном поиске, очевидно из сравнения формы его дерева с формой деревьев для процедур Binary1Search и Binary2Search, которые для  $n = 10$  приведены на рис. 7.3 и 7.4. Последовательному поиску соответствует длинное узкое дерево, что обозначает большое число операций сравнения, в то время как деревья для двоичного поиска значительно шире и короче.



**Рис. 7.3.** Дерево сравнений для процедуры Binary1Search,  $n = 10$

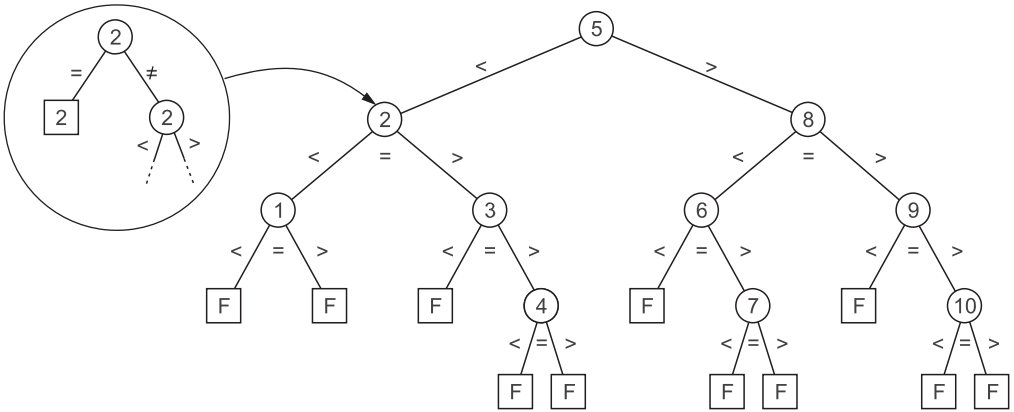


Рис. 7.4. Дерево сравнений для процедуры Binary2Search,  $n = 10$

## 2. Трехвариантное сравнение и компактное изображение

На изображении дерева для процедуры Binary2Search мы показали структуру алгоритма более отчетливо (и заодно уменьшили размер рисунка), скомбинировав две операции сравнения с получением для каждого шага цикла трехвариантного сравнения. При изображении дерева таким образом каждая вершина, не являющаяся листом, завершает тот или иной успешный поиск, а листья соответствуют неуспешным поискам. В результате рис. 7.4 становится более компактным, но не забывайте, что для каждой показанной вершины в действительности выполняются по две операции сравнения, за исключением вершины, в которой поиск завершается успешным обнаружением мишени; в такой вершине выполняется только одна операция сравнения.

Именно такой компактный метод изображения деревьев сравнений мы будем использовать в дальнейших главах.

Удобно также показывать только часть дерева сравнений. На рис. 7.5 изображена верхняя часть дерева сравнений для рекурсивного варианта Binary2Search, причем все детали рекурсивных вызовов спрятаны в

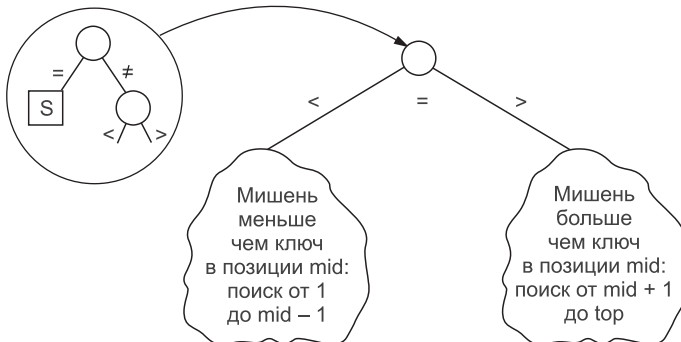


Рис. 7.5. Верхняя часть дерева сравнений для рекурсивного варианта Binary2Search

поддеревьях. Дерево сравнений и дерево рекурсий для рекурсивного алгоритма часто представляют собой два способа отображения одного и того же.

Проанализировав деревья, показанные для Binary1Search и Binary2Search при  $n = 10$ , легко определить, сколько сравнений выполнит каждый алгоритм. В худшем случае число сравнений будет на 1 больше чем высота дерева; фактически для каждой операции поиска число сравнений определяется числом внутренних вершин, лежащих между корнем и вершиной, завершающей поиск.

### 3. Число сравнений для Binary1Search

В процедуре Binary1Search каждый поиск заканчивается на листе; для получения среднего числа сравнений для случаев успешного и неуспешного поиска нам нужно знать то, что носит название *длины внешнего пути* дерева. Эта длина представляет собой сумму ветвей, пройденных на пути от корня до каждого листа дерева. Для дерева, изображенного на рис. 7.3, длина внешнего пути определится следующим образом:

$$(4 \times 5) + (6 \times 4) + (4 \times 5) + (6 \times 4) = 88.$$

Половина листьев соответствуют успешным поискам, а половина — неуспешным. Отсюда среднее число сравнений, требуемых для либо успешного, либо неуспешного поиска с помощью процедуры Binary1Search, будет равно  $44/10 = 4.4$ , если  $n = 10$ .

### 4. Число сравнений для Binary2Search

В дереве для процедуры Binary2Search, все листья соответствуют неуспешным поискам; поэтому длина внешнего пути дает число сравнений для неуспешного поиска. Для успешных поисков нам нужна *длина внутреннего пути*, которая определяется как сумма по всем ветвям, не имеющим листьев, числа ветвей от корня к конкретной вершине. Для дерева на рис. 7.4, длина внутреннего пути равна

$$0 + 1 + 2 + 2 + 3 + 1 + 2 + 3 + 2 + 3 = 19.$$

Вспомним, что процедура Binary2Search выполняет по две операции сравнения для каждой вершины, не являющейся листом, за исключением вершины, в которой обнаруживается мишень, и заметим, что число этих внутренних пройденных вершин на 1 больше числа ветвей (для каждой из  $n = 10$  внутренних вершин). В результате мы получаем среднее число сравнений для успешного поиска:

$$2 \times \left( \frac{19}{10} + 1 \right) - 1 = 4.8.$$

Вычитание 1 отвечает тому факту, что при обнаружении мишени выполняется на одну операцию сравнения меньше.

Для неуспешного поиска с помощью Binary2Search нам нужна длина внешнего пути дерева на рис. 7.4. Эта длина составляет

$$(5 \times 3) + (6 \times 4) = 39.$$

длина  
внешнего пути

длина  
внутреннего пути

среднее число  
сравнений для  
успешного поиска

среднее число  
сравнений для  
успешного поиска

Будем считать, что для неуспешных поисков все  $n + 1$  интервалов (меньше первого ключа, между парами последовательных ключей и больше самого большого ключа) являются равновероятными; для нашего рисунка мы, следовательно, предполагаем, что все 11 листьев-неудач равновероятны. Тогда среднее число сравнений для одного неуспешного поиска будет равно

$$\frac{2 \times 39}{11} \approx 7.1.$$

## 5. Сравнение алгоритмов

Для  $n = 10$  процедура Binary1Search выполняет чуть меньше сравнений и для успешных, и для неуспешных поисков. Однако справедливости ради мы должны заметить, что два сравнения, выполняемые Binary2Search в каждой внутренней вершине, тесно связаны друг с другом (сравнивается один и тот же ключ), так что оптимизирующий компилятор может здесь выполнить меньше работы, чем требуют два полноценных сравнения. В этом случае Binary2Search фактически может оказаться слегка лучшим вариантом для успешных поисков при  $n = 10$ .

## 7.5.2. Обобщение

Что будет, когда  $n$  станет больше 10? Для длинных списков может оказаться невозможным нарисовать полное дерево сравнений, но, рассматривая пример с  $n = 10$ , мы можем сделать несколько наблюдений, справедливых и для других случаев.

### 1. 2-деревья

Давайте определим **2-дерево** как дерево, в котором каждая вершина, кроме листьев, имеет в точности две дочерние вершины. Оба варианта деревьев сравнений, которые мы ранее нарисовали, удовлетворяют этому условию и являются, следовательно, 2-деревьями. Мы можем сделать несколько наблюдений, касающихся 2-деревьев, которые помогут нам установить закономерности поведения методов двоичного поиска для всех значений  $n$ .

терминология

2-деревья по-другому называются **строго двоичными деревьями** или **расширенными двоичными деревьями**, но мы не будем пользоваться этими терминами, потому что их слишком легко спутать с термином **двоичное дерево**, который (как будет рассказано в главе 10) имеет несколько другой смысл.

число вершин  
в 2-дереве

В 2-дереве число вершин на любом уровне не может быть больше, чем удвоенное число вершин на вышележащем уровне, поскольку каждая вершина имеет либо 0, либо 2 дочерних вершины (в зависимости от того, является она листом или нет). Поскольку на уровне 0 (в корне) имеется одна вершина, число вершин на уровне  $t$  не превышает  $2^t$  для всех  $t \geq 0$ . Таким образом, мы имеем следующие факты:

#### Лемма 7.1

*Число вершин на каждом уровне 2-дерева не превышает удвоенного числа вершин, находящихся на ближайшем вышележащем уровне.*

**Лемма 7.2**

В 2-дереве число вершин на уровне  $t$  не превышает  $2^t$  для  $t \geq 0$ .

## 2. Анализ процедуры Binary1Search

В процедуре Binary1Search и успешные, и неуспешные поиски заканчиваются на уровне листьев; таким образом, всегда имеются  $2n$  листьев. Далее, все эти листья должны принадлежать одному уровню или двум примыкающим уровням. (Это наблюдение можно доказать методом математической индукции: оно справедливо для списка размером 1, и когда Binary1Search делит больший список на два меньших, размеры двух половинок различаются максимум на 1, и индуктивное предположение говорит нам, что их листья находятся на одном или соседних уровнях.) Высота дерева (число уровней ниже корня) равна максимальному числу сравнений ключей, выполняемых алгоритмом, и для Binary1Search не превышает среднего числа плюс 1, поскольку все листья находятся на одном или соседних уровнях. Согласно лемме 7.2, высота есть также наименьшее целое  $t$ , для которого  $2^t \geq 2n$ . Посмотрим таблицу логарифмов по основанию 2. (Свойства логарифмов описаны в приложении А.) Мы получим, что число сравнений ключей, выполняемых процедурой Binary1Search при просмотре списка длиной  $n$  элементов, приблизительно равно

$$\lg n + 1.$$

Как можно увидеть, рассмотрев дерево, число сравнений существенно не зависит от того, успешен поиск или нет.

## 3. Обозначения логарифмов

Мы будем и дальше использовать логарифмы по основанию 2, обозначая их так, как мы это только что сделали. Иногда при анализе алгоритмов нам понадобятся также натуральные логарифмы (по основанию  $e = 2.71828 \dots$ ). Мы будем обозначать натуральные логарифмы  $\ln$ . Логарифмы по другим основаниям нужны будут редко. Итак,

### *Соглашение*

Если только не оговорено противное, все логарифмы будут браться по основанию 2. Сокращение  $\lg$  обозначает логарифм по основанию 2, а сокращение  $\ln$  обозначает натуральный логарифм. Если база логарифма не задается, (или не имеет значения), будет использоваться сокращение  $\log$ .

Работая с логарифмами, нам часто будет нужно переходить к ближайшему целому числу сверху или снизу. Для описания этого действия мы введем понятия «*пол*» (floor) действительного числа  $x$ , как максимального целого числа, меньшего или равного  $x$ , а также «*потолок*» (ceiling), как минимального целого числа, большего или равного  $x$ . Мы будем обозначать «пол» через  $\lfloor x \rfloor$ , а «потолок» через  $\lceil x \rceil$ .

число сравнений  
Binary1Search

логарифмы

«пол» и «потолок»



#### 4. Анализ Binary2Search, неуспешный поиск

Для того чтобы определить число операций сравнения, выполняемых процедурой Binary2Search при произвольном значении  $n$ , мы проанализируем дерево сравнений. По тем же причинам, что были рассмотрены для процедуры Binary1Search, у этого дерева все листья находятся не более чем на двух соседних уровнях в самом низу. Для Binary2Search все листья соответствуют неуспешным поискам, поэтому имеется в точности  $n + 1$  листьев, соответствующих  $n + 1$  неуспешным поискам: мишени, меньшей чем минимальный ключ, мишени, располагаемой между парами соседних ключей, и мишени, большей наибольшего ключа. Поскольку все листья располагаются внизу дерева, согласно лемме 7.2 число листьев приблизительно равно  $2^h$ , где  $h$  есть высота дерева. Воспользовавшись логарифмами (по основанию 2), мы получим, что  $h \approx \lg(n + 1)$ . Это значение является приблизительным расстоянием от корня до одного из листьев. Поскольку для каждой внутренней вершины выполняются по два сравнения ключей, число сравнений для неуспешного поиска приблизительно равно  $2\lg(n + 1)$ .

число сравнений  
Binary2Search,  
неуспешный  
случай

#### 5. Теорема о длине пути

Для вычисления среднего числа сравнений для успешного поиска мы прежде всего получим интересное и важное соотношение, справедливое для любого 2-дерева.

##### Теорема 7.3

*Обозначим длину внешнего пути 2-дерева через  $E$ , длину внутреннего пути через  $I$ , и пусть  $q$  есть число вершин, которые не являются листьями. Тогда*

$$E = I + 2q.$$

Для доказательства теоремы мы используем метод математической индукции.

##### Доказательство

Если дерево содержит только свой корень и в нем нет больше вершин, тогда  $E = I = q = 0$ , и для этого случая теорема тривиально верна. Теперь возьмем большее дерево, и пусть  $n$  будет некоторой вершиной, не являющейся листом, но для которого обе дочерних вершины являются листьями. Пусть  $k$  есть число ветвей на пути от корня к  $n$ . Теперь давайте удалим из 2-дерева две вершины, дочерние для  $n$ . Поскольку  $n$  не есть лист, но дочерние вершины как раз листья, число не-листьев уменьшится от  $q$  до  $q - 1$ . Длина внутреннего пути  $I$  уменьшится на расстояние до вершины  $n$  и примет значение  $I - k$ . Расстояние до каждой дочерней для  $n$  вершины есть  $k + 1$ , поэтому длина внешнего пути уменьшится от  $E$  до  $E - 2(k + 1)$ , но вершина  $n$  теперь стала листом, поэтому следует прибавить ее расстояние  $k$ , что дает новое значение длины внешнего пути:

$$E - 2(k + 1) + k = E - k - 2.$$

Поскольку новое дерево имеет меньше вершин, чем старое, по принципу математической индукции мы можем написать

$$E - k - 2 = (I - k) + 2(q - 1).$$

Приведение подобных членов в этом уравнении дает желаемый результат.

6. Анализ Binary2Search, успешный поиск

В дереве сравнений для процедуры Binary2Search расстояние до листьев составляет, как мы уже видели,  $\lg(n + 1)$ . Число листьев равно  $n + 1$ , поэтому длина внешнего пути приблизительно равна

$$(n + 1)\lg(n + 1).$$

По теореме 7.3 мы находим, что длина внутреннего пути приблизительно равна

$$(n + 1)\lg(n + 1) - 2n.$$

Для получения среднего числа сравнений, выполняемых при успешном поиске, мы должны сначала разделить эту величину на  $n$  (число не-листьев), а затем прибавить 1 и удвоить, поскольку в каждой внутренней вершине выполняются два сравнения. Наконец, мы вычитаем 1, поскольку в вершине, где обнаруживается искомая мишень, выполняется только одно сравнение. В итоге среднее число операций сравнения ключей составит приблизительно

$$\frac{2(n + 1)}{n} \lg(n + 1) - 3$$

7.5.3. Методы сравнения

упрощенные  
результаты

Обратите внимание на сходства и различия в формулировке двух вариантов двоичного поиска. Прежде всего вспомним, что мы уже ввели в наши вычисления некоторые приближения, отчего наши формулы оказываются неточными. Для больших значений  $n$  разница между  $\lg n$  и  $\lg(n + 1)$  незначительна, а величина  $(n + 1)/n$  весьма близка к 1. Поэтому мы можем упростить наши результаты следующим образом:

	Успешный поиск	Неуспешный поиск
Binary1Search	$\lg n + 1$	$\lg n + 1$
Binary2Search	$2 \lg n - 3$	$2 \lg n$

оценки

Во всех четырех случаях результат пропорционален  $\lg n$ , если не обращать внимания на небольшой постоянный член, а коэффициенты при  $\lg n$  совпадают с числом сравнений внутри цикла. Тот факт, что цикл в Binary2Search может завершиться раньше, дает обескураживающе малое повышение скорости при успешном поиске; коэффициент при  $\lg n$  несколько не уменьшается, а уменьшается лишь постоянный член, изменяясь от +1 до -3.

Посмотрев на деревья сравнений, можно сразу понять, почему это так. Более половины всех вершин расположены на нижнем уровне, поэтому их циклы не могут завершиться раньше. Более половины оставшихся циклов могут завершиться лишь на один шаг раньше. В результате при больших  $n$  число вершин, расположенных на дереве относительно высоко, скажем, в верхней половине уровней, пренебрежимо мало по сравнению с числом вершин на нижнем уровне. И только из-за этой пренебрежимой разницы процедура Binary2Search может дать лучшие результаты в сравнении с Binary1Search, при этом ценой почти удвоения

числа сравнений для всех операций поиска, как успешного, так и неуспешного.

Имея меньший коэффициент при  $\lg n$ , Binary1Search будет выполнять меньше сравнений при достаточно больших  $n$ , однако из-за меньшей величины свободного члена процедура Binary2Search, может оказаться, будет выполнять меньше сравнений, когда  $n$  невелико. Однако при использовании в условиях малого  $n$  издержки организации двоичного поиска и дополнительные усилия по разработке программы делают двоичный поиск более дорогим методом, чем последовательный. В результате мы приходим к выводу, прямо противоположному интуитивной оценке, что разработка процедуры Binary2Search вряд ли оправдана, поскольку для длинных списков Binary1Search дает лучшие результаты, а для коротких лучше оказывается SequentialSearch. Справедливости ради следует, однако, заметить, что при использовании некоторых компьютеров и оптимизирующих компиляторов два сравнения, необходимые для процедуры Binary2Search, не потребуют удвоенного времени по сравнению с одним сравнением в Binary1Search, поэтому в конечном счете Binary2Search может оказаться более эффективным средством.

Целью нашего анализа алгоритмов является помощь в принятии решения, какой алгоритм выбрать при тех или иных обстоятельствах. Проведя этот анализ, мы получили возможность сделать этот выбор осознанно, и при этом стали обладателями информации, которая поначалу не была очевидной.

На рис. 7.6 изображены графики, показывающие среднее количество сравнений ключей, выполняемых при успешном поиске программами SequentialSearch, Binary1Search и Binary2Search. Числа, показанные на графиках, получены при тестовых прогонах процедур; они не являются аппроксимациями. На первом графике рис. 7.6 три процедуры сравниваются при малых значениях  $n$  (числа элементов в списке). На втором графике мы сравниваем программы при значительно большем диапазоне  $n$ , используя при этом двойную логарифмическую шкалу, в которой каждая единица по оси представляет удвоение соответствующей координаты. На третьем графике мы попытались сравнить два варианта двоичного поиска; здесь более удобна полулогарифмическая шкала, когда по вертикальной оси отложены линейные единицы, а по горизонтальной — логарифмические.

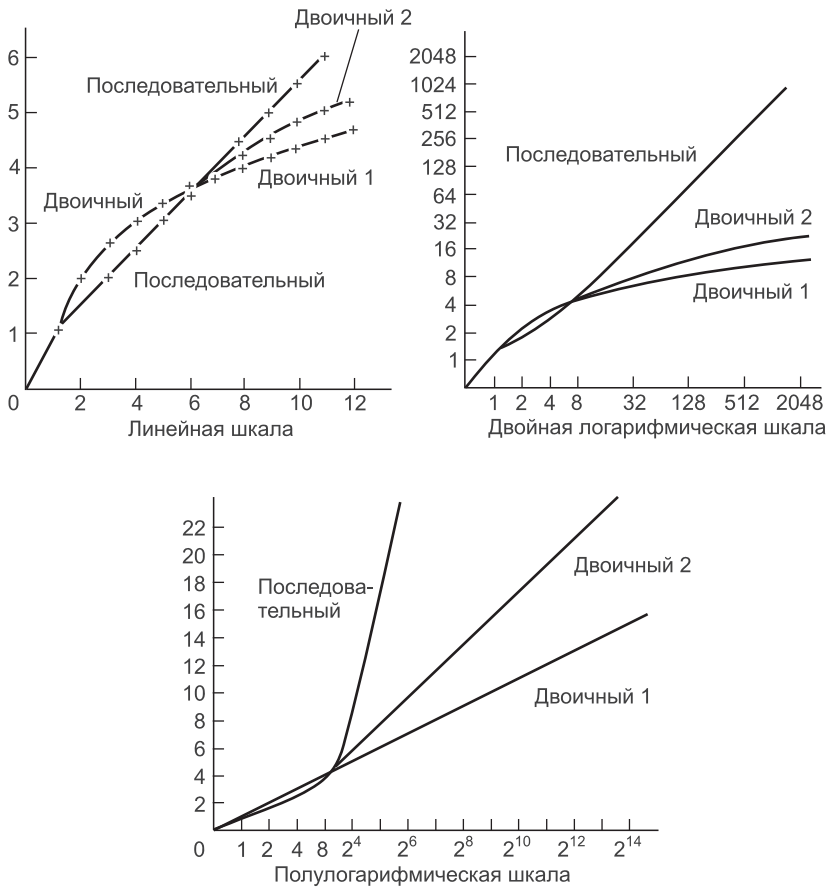
выводы

логарифмические  
графики

#### 7.5.4. Общее отношение

Перед тем, как закончить этот раздел, давайте используем теорему 7.3 для получения соотношения между средним числом операций сравнения для успешного и неуспешного поиска, соотношения, которое будет справедливым для любого метода, допускающего представление в виде дерева сравнений, как это мы сделали для Binary2Search. При этом мы будем считать, что листья дерева сравнений соответствуют неуспешным поискам, что внутренние вершины соответствуют успешным поискам, и что в каждой внутренней вершине выполняются две операции сравнения, за исключением той вершины, где в списке обнаруживается мишень, и где

гипотезы



**Рис. 7.6.** Сравнение эффективности процедур успешного поиска разными методами

выполняется только одна операция сравнения. Если  $I$  и  $E$  есть длины внутреннего и внешнего путей по дереву соответственно, а  $n$  есть число элементов в списке, так что  $n$  всегда также равно числу внутренних вершин дерева, тогда, как следует из анализа `Binary2Search`, мы можем утверждать, что среднее число сравнений при успешном поиске равно

$$S = 2 \left( \frac{I}{n} + 1 \right) - 1 = \frac{2I}{n} + 1$$

а среднее число сравнений при неуспешном поиске равно  $U = 2E/(n + 1)$ . По теореме 7.3  $E = I + 2n$ . Комбинируя эти два выражения, мы можем заключить, что

**Теорема 7.4**

*При заданных условиях среднее число операций сравнения ключей, выполненных при успешном и неуспешном поиске, соотносятся как*

$$S = \left( 1 + \frac{1}{n} \right) U - 3$$

успешные и  
неуспешные  
операции поиска

Другими словами, среднее число операций сравнения при успешном и неуспешном поиске оказывается практически одним и тем же. Сведения о наличии искомого элемента в списке очень мало помогает в его поиске, если поиск ведется путем сравнения ключей.

### Упражнения 7.5

---

- E1.** Нарисуйте деревья сравнений для (i) Binary1Search и (ii) Binary2Search, если **(a)**  $n = 5$ , **(b)**  $n = 7$ , **(c)**  $n = 8$ , **(d)**  $n = 13$ . Вычислите длины внешнего и внутреннего путей для каждого из этих деревьев и проверьте, остается ли справедливой теорема 7.3.
- E2.** Последовательный поиск имеет меньше издержек, чем двоичный, и поэтому при малых  $n$  может работать быстрее. Найдите точку пересечения эффективностей, в которой для процедур SequentialSearch и Binary1Search выполняется одно и то же число сравнений ключей. В вычислениях используйте формулу для числа сравнений при успешном поиске.
- E3.** Пусть имеется список из 10000 имен, расположенных в массиве в алфавитном порядке, и вы должны часто искать в этом списке те или иные имена. Выясняется, что 20 процентов имен покрывают 80 процентов операций поиска. Вместо того, чтобы выполнять каждый раз двоичный поиск по всем 10000 именам, рассмотрите возможность разделения списка на два, список с частым обращением, содержащий 2000 имен, и список с редким обращением с оставшимися 8000 именами. Чтобы найти имя, сначала используется двоичный поиск в списке с частым обращением, и в 80 процентах времени вам не придется переходить ко второй стадии, на которой вы используете двоичный поиск во втором списке. Даст ли такая идея практический эффект? Обоснуйте свой ответ, найдя число сравнений, выполняемых в среднем при успешном поиске, как при такой организации списков, так и при наличии одного списка на 10000 имен.
- E4.** Если вы модифицировали двоичный поиск так, что он делит список при каждом проходе не точно пополам, а на части, имеющие размер одна треть и две трети оставшейся части списка, как приблизительно изменится среднее число сравнений?

### Программные проекты 7.5

---

- P1. (a)** Напишите «тернарную» процедуру поиска, аналогичную Binary2Search, которая анализирует ключ на одной третьей всего списка и, если ключ-мишень больше ключа в списке, анализирует оставшиеся две трети списка, в результате чего после каждого прохода длина списка в любом случае уменьшается в три раза. **(b)** Включите эту процедуру в качестве еще одной возможности в тестирующую программу проекта P1 в разделе 7.2 и сравните ее производительность с производительностью других методов.
- P2. (a)** Напишите программу, которая будет выполнять «гибридный» поиск, используя Binary1Search для больших списков и переключаясь на последовательный поиск, когда в результате деления оставшийся спи-

сок станет достаточно коротким. (Из-за различных программных поддержек, ваша точка переключения не обязательно совпадет с той, что вы определили в Упражнении E2.) **(b)** Включите эту процедуру в качестве еще одной возможности в тестирующую программу Проекта P1 в разделе 7.2 и сравните ее производительность с производительностью других методов.

- P3. (a)** Разработайте реализацию гардероба (см. раздел 7.3, проект P1) на основе упорядоченного списка, где упорядочение выполнено согласно размеру тега. Используйте упорядоченное включение для сдачи в гардероб нового плаща, а также двоичный поиск для проверки наличия и получения плаща. **(b)** Протестируйте ваш пакет с помощью демонстрационной программы, управляемой меню. **(c)** Подставьте свой пакет в программу тестирования в реальном времени и сравните его производительность с производительностью других реализаций.

## 7.6. Нижние границы

Мы знаем, что в последовательном упорядоченном списке двоичный поиск оказывается значительно быстрее последовательного. В таком случае вполне естественно задать вопрос, нельзя ли найти еще какой-то метод, который окажется значительно быстрее двоичного.

### 1. Усовершенствование программ

Один из подходов заключается в таком усовершенствовании программы, чтобы она выполнялась быстрее. Можно с помощью всяких хитростей попытаться уменьшить объем работы, выполняемой в каждом шаге цикла, и ускорить тем самым выполнение алгоритма. Один из методов, так называемый *поиск Фибоначчи*, даже ухитряется заменить операции деления в каждом шаге цикла поиска операциями вычитания (без использования вспомогательных таблиц), что на некоторых компьютерах даст положительный эффект.

Тонкая настройка программы может уменьшить время ее выполнения вдвое и даже больше, но скоро вы достигнете предела, определяемого лежащим в основе программы алгоритмом. Причина того, что двоичный поиск настолько быстрее последовательного заключается не в уменьшении числа шагов внутри цикла (фактически этих шагов даже больше) или в оптимизации кода, а в том, что цикл выполняется меньшее число раз, примерно  $\lg n$  раз вместо  $n$ , а по мере увеличения  $n$  значение  $\lg n$  растет значительно медленнее, чем значение  $n$ .

В контексте сравнения базовых методов различия между Binary1Search и Binary2Search становятся незначительными. Для больших списков Binary2Search может потребовать почти вдвое больше времени, чем Binary1Search, но различие между  $2\lg n$  и  $\lg n$  пренебрежима по сравнению с различием между  $n$  и  $2\lg n$ .

### 2. Произвольные алгоритмы поиска

Давайте зададим себе вопрос, может ли существовать алгоритм поиска, который в худшем и среднем случаях сможет найти мишень, используя

значительно меньшее число сравнений ключей, чем алгоритм двоичного поиска. Мы увидим, что ответ будет отрицательным, если оставаться внутри класса алгоритмов, которые с целью поиска в упорядоченном списке основываются только на сравнении ключей.

Давайте начнем с произвольного алгоритма, который осуществляет поиск в упорядоченном списке путем сравнения ключей, и представим себе, что мы рисуем дерево сравнений этого алгоритма так же, как мы рисовали дерево сравнений для Binary1Search. Т. е. каждая внутренняя вершина дерева будет соответствовать некоторому сравнению ключей, а каждый лист — одному из возможных исходов. (Если алгоритм сформулирован как трехвариантное сравнение, как это было в процедуре Binary2Search, тогда мы расщепляем каждую внутреннюю вершину на две, как это было показано для одной вершины на рис. 7.4.) Возможные исходы, которым соответствуют листья, включают не только успешное нахождение мишени, но также и разного рода отказы, которые алгоритм может различить. Двоичный поиск в списке длиной  $n$  образует  $k = 2n + 1$  исходов, которые состоят из  $n$  успешных исходов и  $n + 1$  различных отказов (мишень меньше наименьшего ключа в списке, мишень находится внутри любой пары соседних ключей и мишень больше наибольшего ключа). С другой стороны, наша процедура последовательного поиска образует только  $k = n + 1$  возможных исходов, поскольку она фиксирует только один род отказа.

Как и для любых алгоритмов поиска, использующих сравнение ключей, высота нашего дерева будет равна числу сравнений, которые выполняются алгоритмом в наихудшем случае, и (поскольку все исходы соответствуют листьям) длина внешнего пути дерева, деленная на число возможных исходов, будет равна среднему числу сравнений, выполненных алгоритмом. Нам хотелось бы получить нижнюю границу высоты и длины внешнего пути в зависимости от числа листьев  $k$ .

### 3. Исследования 2-деревьев

Вот как выглядит результат, полученный для 2-деревьев:

#### Лемма 7.5

*Пусть  $T$  есть 2-дерево с  $k$  листьями. Тогда высота  $h$  дерева  $T$  удовлетворяет неравенству  $h \geq \lceil \lg k \rceil$ , а длина внешнего пути  $E(T)$  удовлетворяет условию  $E(T) \geq k \lg k$ . Минимальные значения для  $h$  и  $E(T)$  достигаются, когда все листья дерева  $T$  находятся на одном уровне или на двух прилегающих уровнях.*

#### Доказательство

Начнем доказательство с проверки последнего утверждения. Предположим, что некоторые из листьев  $T$  находятся на уровне  $r$ , а некоторые на уровне  $s$ , причем  $r > s + 1$ . Теперь возьмем два листа на уровне  $r$ , которые оба являются дочерними для одной и той же вершины  $v$ , отделим их от  $v$  и присоединим их в качестве дочерних к некоторому (бывшему) листу на уровне  $s$ . Мы изменили дерево  $T$ , создав новое дерево  $T'$ , которое все еще имеет  $k$  листьев, высота которого уж точно не больше высоты  $T$ , а длина внешнего пути удовлетворяет следующему равенству

$$E(T') = E(T) - 2r + (r - 1) - s + 2(s + 1) = E(T) - r + s + 1 < E(T)$$

произвольные  
алгоритмы  
и деревья  
сравнений

высота и длина  
внешнего пути



поскольку  $r > s + 1$ . Члены этого выражения получены следующим образом. Поскольку два листа на уровне  $r$  удалены,  $E(T)$  уменьшено на  $2r$ . Поскольку вершина  $v$  стала листом,  $E(T)$  увеличилось на  $r - 1$ . Поскольку два листа, ранее находившиеся на уровне  $r$ , теперь переместились на уровень  $s + 1$ , к  $E(T)$  добавлен член  $2(s + 1)$ . Этот процесс проиллюстрирован на рис. 7.7.

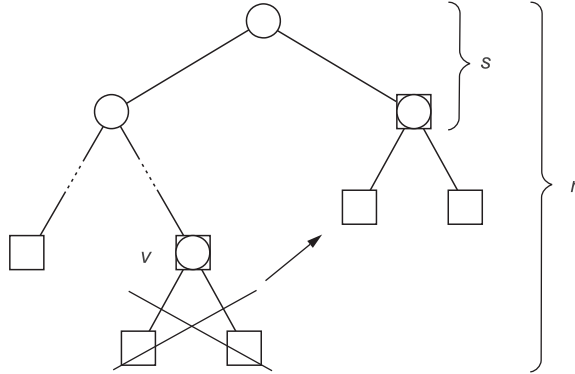


Рис. 7.7. Перемещение листьев выше по 2-дереву

Мы можем продолжать действовать в том же духе, перемещая листья выше по дереву, уменьшая каждый раз длину внешнего пути и, возможно, высоту, пока в конце концов все листья не окажутся на том же или соседних уровнях, и тогда высота и длина внешнего пути будут минимальны среди всех 2-деревьев с  $k$  листьями.

доказательство  
 $h \geq \lceil \lg k \rceil$

Для доказательства оставшихся в лемме 7.5 утверждений, давайте предположим, что  $T$  имеет минимальную высоту и длину пути среди всех 2-деревьев с  $k$  листьями, так что все листья  $T$  находятся на уровнях  $h$  и (возможно)  $h - 1$ , где  $h$  есть высота дерева  $T$ . Согласно лемме 7.2, число вершин на уровне  $h$  (которые являются листьями) не превышает величины  $2^h$ . Если все листья находятся на уровне  $h$ , тогда  $k \leq 2^h$ . Если некоторые из листьев находятся на уровне  $h - 1$ , тогда каждый из них (поскольку они не имеют дочерних вершин) уменьшает число возможных вершин на уровне  $h$  на 2, поэтому граница  $k \leq 2^h$  продолжает иметь место. Для получения  $h \geq \lg k$  мы логарифмируем выражение и, поскольку высота всегда является целым числом, округляем вверх до «потолка» и получаем  $h \geq \lceil \lg k \rceil$ .

доказательство  
 $E(T) \geq k \lg k$

Для получения границы длины внешнего пути примем, что  $x$  обозначает число листьев дерева  $T$  на уровне  $h - 1$ , так что  $k - x$  листьев находятся на уровне  $h$ . Эти вершины являются дочерними от в точности  $\frac{1}{2}(k - x)$  вершин на уровне  $h - 1$ , которые, вместе с  $x$  листьями, составляют все вершины уровня  $h - 1$ . Отсюда по лемме 7.2 имеем:

$$\frac{1}{2} (k - x) + x \leq 2^{h-1},$$



что преобразуется в  $x \leq 2^h - k$ . Теперь мы имеем

$$\begin{aligned} E(T) &= (h-1)x + h(k-x) \\ &= kh - x \\ &\geq kh - (2^h - k) \\ &= k(h+1) - 2^h. \end{aligned}$$

Из границы высоты мы уже знаем, что  $2^{h-1} < k \leq 2^h$ . Если мы положим  $h = \lg k + \varepsilon$ , тогда  $\varepsilon$  удовлетворяет  $0 \leq \varepsilon < 1$ , и подставляя  $\varepsilon$  в границу для  $E(T)$ , получим

$$E(T) \geq k(\lg k + 1 + \varepsilon - 2^\varepsilon).$$

Оказывается, при  $0 \leq \varepsilon < 1$ , величина  $1 + \varepsilon - 2^\varepsilon$  лежит между 0 и 0.0861. Таким образом, минимальная длина пути весьма близка к  $k \lg k$  и, в любом случае, не меньше  $k \lg k$ , что и требовалось доказать. Теперь лемма 7.5 полностью доказана.

#### 4. Нижние границы для поиска

Наконец, мы можем вернуться к изучению произвольного алгоритма поиска. Его дерево сравнений может и не иметь всех листьев на двух прилегающих уровнях, но даже и в этом случае границы в лемме 7.5 выдерживаются. Поэтому мы можем преобразовать эти границы в язык сравнений, как это сделано ниже.

##### Теорема 7.6

*Предположим, что алгоритм для поиска мишени в списке использует сравнения ключей. Если имеются  $k$  возможных исходов, тогда алгоритм должен выполнить по меньшей мере  $\lceil \lg k \rceil$  сравнений ключей в худшем случае и по меньшей мере  $\lg k$  в среднем случае.*

Обратите внимание на то, что разница между границами в худшем и в среднем случаях весьма невелика. Более того, согласно теореме 7.4, для многих алгоритмов при определении границ не имеет особого значения, успешен ли поиск или нет. Когда мы прикладываем теорему 7.6 к алгоритмам вроде двоичного поиска, для которого, в случае упорядоченного списка длины  $n$ , имеются  $n$  успешных и  $n + 1$  неуспешных исходов, мы получаем для границы наихудшего случая

$$\lceil \lg(2n+1) \rceil \geq \lceil \lg(2n) \rceil = \lceil \lg n \rceil + 1,$$

а для границы среднего случая  $\lg n + 1$  сравнений ключей. Сравнив эти числа с теми, что получены при анализе Binary1Search, мы получим

##### Следствие 7.7

*Алгоритм Binary1Search оптимален в классе всех алгоритмов, которые осуществляют поиск в упорядоченном списке путем сравнения ключей. И в среднем, и в наихудшем случаях алгоритм Binary1Search обеспечивает оптимальные границы.*

#### 5. Другие способы поиска

Теорема 7.6, устанавливая границы, не утверждает, что никакой алгоритм не может работать быстрее двоичного поиска; в ней речь идет только об алгоритмах, основанных на сравнении ключей. В качестве просто-

го примера предположим, что сами ключи представляют собой целые числа от 1 до  $n$ . Если мы знаем, что ключ-мишень представляет собой целое число в этом диапазоне, то вряд ли мы будем использовать алгоритм поиска для обнаружения мишени; мы просто сохраним элементы в массиве, индексируемом от 1 до  $n$ , и для нахождения нужного элемента обратимся к этому массиву по индексу  $x$ .

Эта идея может быть расширена с получением другого метода, называемого **интерполяционным поиском**. Мы предполагаем, что ключи либо являются числами, либо имеют такой характер (например, представляют собой слова), что их легко преобразовать в числа. Метод также предполагает, что ключи распределены в списке равномерно, т. е. что вероятность для ключа оказаться в конкретном диапазоне значений совпадает с вероятностью для него оказаться в любом другом диапазоне значений той же длины. Для нахождения ключа-мишени  $target$  алгоритм интерполяционного поиска оценивает по отношению величины  $target$  к первому и последнему ключу в списке, в каком приблизительно месте списка должна находиться мишень, и ищет ее там. Затем алгоритм, выяснив, оказалась ли мишень больше или меньше анализируемого ключа, уменьшает соответственно размер списка и продолжает поиск. Можно показать, что при равномерно распределенных ключах обнаружение мишени методом интерполяционного поиска потребует около  $\lg \lg n$  сравнений ключей, что при больших значениях  $n$  значительно меньше, чем требуется для двоичного поиска. Если, например,  $n = 1000000$ , процедура `Binary1Search` выполнит около  $\lg 10^6 + 1 \approx 21$  сравнение, а интерполяционный поиск может потребовать только около  $\lg \lg 10^6 \approx 4.32$  сравнения.

Наконец, стоит повторить, что даже для поиска путем сравнений наше требование, чтобы все ключи были распределены равномерно, может оказаться далеким от истины. Если один или два ключа требуются значительно чаще, чем другие, тогда даже последовательный поиск, если сначала ищутся эти ключи, может оказаться быстрее любого другого метода. Важность поиска или, в более общем плане, извлечения информации, настолько велика, что значительная часть структур данных разрабатывается исходя из методов поиска, и в последующих главах мы будем возвращаться к этим проблемами снова и снова.

### Упражнение 7.6.

**E1.** Предположим, что, как и в процедуре `Binary2Search`, алгоритм поиска выполняет трехвариантные сравнения. Пусть каждый внутренний узел дерева сравнений этого алгоритма соответствует успешному поиску, а каждый лист — неуспешному.

- (a) С помощью леммы 7.5 сформулируйте теорему наподобие теоремы 7.6, дающую нижние границы для худшего и среднего случаев поведения для неуспешного поиска таким алгоритмом.
- (b) С помощью теоремы 7.4 получите аналогичные результаты для успешных поисков.
- (c) Сравните полученные вами границы с анализом процедуры `Binary2Search`.

## Программный проект 7.6

**P1. (a)** Напишите программу, которая выполняет интерполяционный поиск и проверьте правильность ее работы (особенно, завершение программы). См. ссылки в конце этой главы, где можно найти предложения и примеры анализа программ. **(b)** Включите вашу процедуру в качестве еще одной возможности в программу тестирования из проекта P1 раздела 7.2 и сравните ее производительность с результатами испытаний других методов.

## 7.7. Асимптотика

### 7.7.1. Введение

Наступило время выделить важные обобщения из нашего анализа алгоритмов поиска. По ходу наших рассуждений мы могли более отчетливо представить, какие аспекты анализа алгоритмов имеют большую важность, а какими можно благополучно пренебречь. Если секция программы выполняется только один раз, например, вне циклов, тогда время, которое она расходует, пренебрежимо мало по сравнению с временем выполнения циклов. Мы нашли, что хотя двоичный поиск более сложен в программировании и анализе, чем последовательный поиск, и даже несмотря на то, что для коротких списков он выполняется более медленно, при обработке списков достаточной длины он оказывается существенно быстрее последовательного поиска.

Разработка эффективных методов работы с небольшими задачами является важным предметом изучения, поскольку большие задачи, возможно, выполняют те же или схожие небольшие задачи, но много раз. Однако, как мы выяснили, при решении небольших задач значительные издержки при применении изощренных методов могут заставить нас сделать выбор в пользу более простых методов. Для списка с тремя или четырьмя элементами последовательный поиск несомненно эффективнее двоичного. Для повышения эффективности алгоритма небольшой задачи программист вынужден уделить внимание деталям, специфическим для компьютерной системы или языка программирования, и в такой оптимизации нам вряд ли помогут общие наблюдения.

Разработка эффективных алгоритмов для больших задач требует совсем иного подхода. При изучении двоичного поиска мы видели, что издержки становятся относительно маловажными; базовая идея, наоборот, приобретает особое значение, и правильный выбор метода может превратить задачу, на первый взгляд слишком сложную для решения, в задачу, вполне решаемую имеющимися средствами.

Слово *асимптотика*, вынесенная в заголовок этого раздела, обозначает изучение функций параметра  $n$  по мере того, как  $n$  становится все больше и больше, ничем не ограничиваясь. Сравнивая разные алгоритмы поиска, мы видели, что число операций сравнения ключей довольно точно отражает полное время выполнения большой программы, поскольку как правило получается, что все остальные операции (например, инкре-

разработка  
алгоритмов для  
небольших задач

выбор метода для  
большой задачи

асимптотика

менты и сравнения индексов) жестко привязаны к операциям сравнения ключей.

базовые операции

Фактически частота этих базовых операций оказывается существенно более важной, чем полное число всех операций, включая организационные и служебные действия. Полное число операций, включающее организационные действия, настолько зависит от выбора языка программирования и от стиля конкретного программиста, что все эти детали могут скрыть основные методы. Вариации в организационных действиях или технике программирования легко могут утроить время выполнения программы, хотя такой результат ничего не говорит о целесообразности использования того или иного базового метода. С другой стороны, изменение базового метода может иметь огромное значение. Если число базовых операций пропорционально  $\lg n$ , тогда удвоение  $n$  ничтожно изменит общее время выполнения. Если же число базовых операций пропорционально  $n^2$ , тогда время выполнения будет расти как квадрат длины списка, и хотя программу еще можно будет использовать, большое время вычислений будет ограничивать ее применение. Если же число базовых операций оказывается пропорциональным  $2^n$ , тогда удвоение  $n$  возведет число операций в квадрат. Вычисления, длящиеся 1 секунду, могут вовлечь миллион ( $10^6$ ) базовых операций, и удвоение входных данных потребует уже  $10^{12}$  операций, увеличив время выполнения с 1 секунды до 11.5 дней.

цели

Наше желание сформулировать обобщенные принципы, которые можно приложить ко многим классам алгоритмов, сводится к стремлению выработать систему обозначений и описаний, которая будет точно отражать ход увеличения времени выполнения при увеличении размера задачи, но при этом будет игнорировать мелкие детали, вносящие небольшой вклад в общую сумму. В этом случае мы можем сконцентрировать свое внимание на одной или двух базовых операциях в нашем алгоритме, не обращая особого внимания на сопровождающие их организационные действия. Если алгоритм выполняет  $f(n)$  базовых операций при размере входных данных  $n$ , тогда полное время выполнения будет максимум  $cf(n)$ , где  $c$  есть константа, зависящая от алгоритма, способа его программирования и использованного компьютера, но  $c$  не будет зависеть от размера  $n$  входных данных (по крайней мере, когда  $n$  выходит за рамки нескольких начальных случаев).

## 7.7.2. О большое

Обсуждаемые нами идеи можно представить в виде следующего определения:

Определение

Если  $f(n)$  и  $g(n)$  есть функции, определенные в пространстве положительных целых чисел, тогда запись

$$f(n) \text{ есть } O(g(n))$$

[читается  $f(n)$  есть **О большое** от  $g(n)$ ] обозначает, что существует такая константа  $c$ , что для всех достаточно больших положительных целых  $n$   $|f(n)| \leq c |g(n)|$ .

При этих условиях мы можем также сказать, что « $f(n)$  имеет **порядок** не больший, чем  $g(n)$ » или « $f(n)$  растет не быстрее чем  $g(n)$ ».

## 1. Примеры

В качестве первого примера рассмотрим функцию  $f(n) = 100n$ . Тогда  $f(n)$  есть  $O(n)$ , поскольку  $f(n) \leq cn$  для константы  $c = 100$ . Любая константа  $c > 100$  также удовлетворяет этому определению.

Теперь рассмотрим функцию  $f(n) = 4n + 200$ . Поскольку  $f(n) < 4n$  не справедливо для больших значений  $n$ , мы не можем показать что  $f(n)$  есть  $O(n)$ , взяв  $c = 4$ . Однако мы можем выбрать для  $c$  любое большее значение. Если, например, мы выберем  $c = 5$ , тогда мы найдем что  $f(n) \leq 5n$  если  $n \geq 200$  и, следовательно, опять  $f(n)$  есть  $O(n)$ .

В качестве следующего примера возьмем  $f(n) = n^2$ . Попробуем доказать что  $f(n)$  есть  $O(n)$ . Если нам это удастся, это будет означать, что мы можем найти такую константу, для которой  $n^2 \leq cn$  для достаточно больших  $n$ . Если мы выберем в качестве  $n$  достаточно большое положительное целое и разделим обе части этого неравенства на  $n$ , мы получим  $n \leq c$  для всех целых чисел и константы  $c$ . Это неравенство очевидный нонсенс: мы можем выбрать  $n$  больше  $c$ . Отсюда мы должны заключить что  $n^2$  не есть  $O(n)$ .

Последним примером выбранных нами полиномов от  $n$  будет  $f(n) = 3n^2 - 100n$ . Для малых значений  $n$   $f(n)$  меньше  $n$ , но по причинам, схожим с анализом предыдущего примера, для любой константы  $c$   $f(n)$  будет больше  $cn$ , если  $n$  достаточно велико. Отсюда  $f(n)$  не есть  $O(n)$ . С другой стороны, если  $c = 3$ , мы имеем  $f(n) \leq 3n^2$ , откуда видно, что  $f(n)$  есть  $O(n^2)$ . Если изменить  $f(n)$  на  $3n^2 + 100n$ , мы все еще будем иметь, что  $f(n)$  есть  $O(n^2)$ . В этом случае мы не сможем использовать  $c = 3$ , но любое большее значение  $c$  нас вполне устроит.

## 2. Общие наблюдения

Рассмотренные примеры полиномов позволяют вывести первое и наиболее важное правило относительно понятия  $O$  большого. Чтобы получить порядок полиномиальной функции, мы просто извлекаем член с наибольшей степенью, отбрасывая все константы и все члены с меньшими степенями. Более формально это правило выглядит так:

полиномы

*Если  $f(n)$  есть полином от  $n$  степени  $r$ , тогда  $f(n)$  есть  $O(n^r)$ , но  $f(n)$  не есть  $O(n^s)$  для любой степени  $s$ , меньшей чем  $r$*

При исследовании алгоритмов часто приходится сталкиваться с логарифмическими функциями. Мы уже использовали логарифмы при анализе двоичного поиска и видели, что логарифм  $n$  растет гораздо более медленно, чем само  $n$ . Общее правило выглядит так:

логарифмы

*Любой логарифм от  $n$  растет медленнее (с ростом  $n$ ) чем любая положительная степень  $n$ . Отсюда  $\log n$  есть  $O(n^k)$  для любого  $k > 0$ , но  $n^k$  никогда не есть  $O(\log n)$  для любой степени  $k > 0$ .*

### 3. Распространенные порядки

Когда мы используем понятие  $O$  большого,  $f(n)$  обычно представляет собой значение времени выполнения некоторого алгоритма, и нам хотелось бы выбрать форму  $g(n)$  максимально простой. Мы тогда пишем  $O(1)$ , имея в виду время выполнения, ограниченное константой (не зависящей от  $n$ );  $O(n)$  означает, что время прямо пропорционально  $n$ , и мы тогда говорим о *линейной зависимости времени*. Мы называем  $O(n^2)$  *квадратичной зависимостью*,  $O(n^3)$  *кубической*, а  $O(2^n)$  *экспоненциальной*. Эти пять порядков вместе с *логарифмической зависимостью*  $O(\lg n)$  и  $O(n \lg n)$  используются чаще других при анализе алгоритмов.

На рис. 7.8 показано, как эти семь функций (вместе с константой 1) растут с увеличением  $n$ ; относительные величины приведены на рис. 7.9. Число в правом нижнем углу таблицы на рис. 7.9 невообразимо велико: если бы каждый электрон во вселенной (их всего насчитывается  $10^{50}$  штук) был бы суперкомпьютером, выполняющим сто миллионов ( $10^8$ ) операций в секунду от создания мира (возможно 30 миллиардов лет, или около  $10^{18}$  секунд), тогда вычисления, требующие  $2^{1000}$  операций, за это время завершили бы всего лишь  $10^{76}$  операций, т. е. им пришлось бы продолжаться еще в  $10^{225}$  раз дольше! Вычисления, требующие  $2^n$  операций имеют смысл лишь при *очень* маленьких значениях  $n$ .

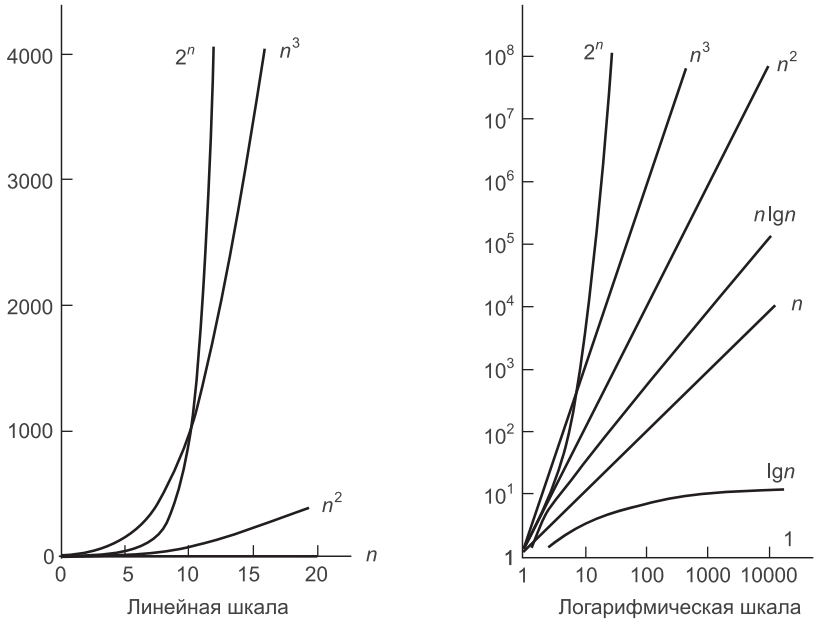


Рис. 7.8. Скорость роста распространенных функций

Обратите особое внимание на то, что  $\lg n$  растет значительно медленнее  $n$ ; это и является причиной преимущества двоичного поиска перед последовательным для больших списков. Заметьте также, что функции 1 и  $\lg n$  для больших  $n$  все более удаляются (в меньшую сторону) от остальных функций.

$n$	1	$\lg n$	$n$	$n \lg n$	$n^2$	$n^3$	$2^n$
1	1	0.00	1	0	1	1	2
10	1	3.32	10	33	100	1000	1024
100	1	6.64	100	66	10000	1000000	$1.268 \times 10^{30}$
1000	1	9.97	1000	997	1000000	$10^9$	$1.072 \times 10^{301}$

Рис. 7.9. Относительные значения функций

#### 4. Анализ алгоритмов

Теперь мы можем просто и наглядно сформулировать результаты нашего анализа алгоритмов.

- Для списка длиной  $n$  последовательный поиск характеризуется временем выполнения  $O(n)$ .
- Для упорядоченного списка длиной  $n$  двоичный поиск характеризуется временем выполнения  $O(\lg n)$ .
- Извлечение из непрерывного списка длиной  $n$  характеризуется временем выполнения  $O(1)$ .
- Извлечение из связного списка длиной  $n$  характеризуется временем выполнения  $O(n)$ .

### 7.7.3. Неточность определения $O$ большого

Заметьте, что значение константы  $c$  в определении  $O$  большого зависит от обсуждаемых функций  $f(n)$  и  $g(n)$ . Так, мы можем написать, что  $17n^3 - 5$  есть  $O(n^3)$  (здесь подойдет  $c = 17$ , как и любые большие значения  $c$ ), а также утверждать, что  $35n^3 + 100$  есть  $O(n^3)$  (здесь  $c \geq 35$ ).

#### 1. Неудачные примеры использования

Заметьте также, что в равной степени правильно написать, что  $35n^3$  есть  $O(n^7)$ , как и что  $35n^3$  есть  $O(n^3)$ . Правильно, но неинформативно утверждать, что и двоичный и последовательный поиск характеризуются временем выполнения, которое есть  $O(n^5)$ . Если  $h(n)$  есть любая функция, растущая быстрее чем  $g(n)$ , тогда функция, которая есть  $O(g(n))$ , также будет  $O(h(n))$ . Отсюда видно, что понятие большого  $O$  можно использовать неадекватно, хотя мы всегда должны стараться не делать этого, используя минимально возможную из семи функций, показанных на рис. 7.8.

Обычный, хотя и неправильный способ использования понятия  $O$  большого заключается в написании предложения вроде «Двоичный поиск работает в течение  $O(\log n)$  времени, в то время как последовательный поиск занимает  $O(n)$  времени». Это предложение, будучи формально правильным, *не* подразумевает, что двоичный поиск быстрее последовательного. Ведь столь же правильным будет замечание, что двоичный поиск также требует  $O(n)$  времени.



## 2. Сохранение доминантного члена

Обычно нам хотелось бы получить более точную меру работы, выполняемой алгоритмом, и мы можем оценить ее, используя понятие  $O$  большого в выражении, как это показано ниже. Если мы напишем

$$f(n) = g(n) + O(h(n))$$

то это означает, что  $f(n) - g(n)$  есть  $O(h(n))$ . Вместо того, чтобы думать об  $O(h(n))$  как о классе всех функций, растущих для некоторой константы  $c$  не быстрее, чем  $ch(n)$ , мы рассматриваем  $O(h(n))$  как одну, хотя и произвольную, из таких функций. Затем мы используем эту функцию для представления тех членов наших вычислений, в которых мы не очень заинтересованы, обычно всех членов кроме того, который растет наиболее быстро.

сравнение  
алгоритмов поиска

Результаты анализа некоторых из наших алгоритмов можно тогда подытожить следующим образом:

- Для случаев успешного поиска в списке длиной  $n$  алгоритм последовательного поиска затратит время  $\frac{1}{2}n + O(1)$ .
- Для случаев успешного поиска в упорядоченном списке длиной  $n$  алгоритм двоичного поиска затратит время  $2 \lg n + O(1)$ .
- Извлечение из непрерывного списка длиной  $n$  занимает время  $O(1)$ .
- Извлечение из простого связного списка длиной  $n$  занимает в среднем время  $\frac{1}{2}n + O(1)$ .

Используя понятие  $O$  большого в выражениях, всегда важно помнить, что  $O(h(n))$  обозначает не точно определенную функцию, а произвольную функцию из большого класса. Отсюда правила обычной алгебры неприменимы к выражению  $O(h(n))$ . Например, у нас могут быть два выражения

$$n^2 + 4n - 5 = n^2 + O(n) \quad \text{и} \quad n^2 - 9n + 7 = n^2 + O(n),$$

но поскольку  $O(n)$  представляет в этих выражениях различные функции, мы не можем приравнять правые части или решить, что левые части выражений совпадают.

### 7.7.4. Порядки распространенных функций

Хотя семь функций, изображенных графически на рис. 7.8, вполне удовлетворяют нашим потребностям для анализа алгоритмов, мы все же приведем простые правила, которые позволят вам определить порядок многих других видов функций.

степени

логарифмы

1. Степени  $n$  имеют порядок согласно показателю степени:  $n^a$  есть  $O(n^b)$  если и только если  $a \leq b$ .
2. Порядок  $\log n$  не зависит от основания логарифма; другими словами,  $\log_a n$  есть  $O(\log_b n)$  для всех  $a, b > 1$ .
3. Логарифм растет медленнее, чем любая положительная степень  $n$ :  $\log n$  есть  $O(n^a)$  для любых  $a > 0$ , но  $n^a$  никогда не есть  $O(\log n)$  для  $a > 0$ .



экспоненты

4. Любая степень  $n^a$  есть  $O(b^n)$  для всех  $a$  и всех  $b > 1$ , но  $b^n$  никогда не есть  $O(n^a)$  для любых  $b > 1$  или для любых  $a$ .

произведения

5. Если  $a < b$ , тогда  $a^n$  есть  $O(b^n)$ , но  $b^n$  не есть  $O(a^n)$ .

6. Если  $f(n)$  есть  $O(g(n))$  и  $h(n)$  есть произвольная функция, тогда  $f(n)g(n)$  есть  $O(g(n)h(n))$ .

цепное правило

7. Вышеприведенные правила могут применяться рекурсивно (цепное правило) путем подстановки любой функции от  $n$  вместо  $n$ . Например,  $\log \log n$  есть  $O((\log n)^{1/2})$ . Для проверки этого факта замените  $n$  на  $\log n$  в предложении « $\log n$  есть  $O(n^{1/2})$ ».

### Упражнения 7.7

**Е1.** Для каждой из приведенных ниже пар функций найдите наименьшее целочисленное значение  $n > 1$ , для которого первая функция становится больше второй.

(a)  $n^2$  и  $15n + 5$

(b)  $2^n$  и  $8n^4$

(c)  $0.1n$  и  $10 \lg n$

(d)  $0.1n^2$  и  $100n \lg n$

**Е2.** Расположите приведенные ниже функции в возрастающем порядке; другими словами,  $f(n)$  должна идти перед  $g(n)$ , если и только если  $f(n)$  есть  $O(g(n))$ .

1000000

$(\lg n)^3$

$2^n$

$n \lg n$

$n^3 - 100n^2$

$n + \lg n$

$\lg \lg n$

$n^{0.1}$

$n^2$

**Е3.** Пусть  $x$  и  $y$  есть действительные числа, причем  $0 < x < y$ . Докажите, что  $n^x$  есть  $O(n^y)$ , но  $n^y$  не есть  $O(n^x)$ .

**Е4.** Покажите, что порядок логарифмической функции не зависит от основания  $a$ , выбранного для логарифмов. Другими словами, докажите, что  $\log_a n$  есть  $O(\log_b n)$  для любых действительных чисел  $a > 1$  и  $b > 1$ .

### Программные проекты 7.7

**P1.** Напишите программу для определения того, сколько времени займет на вашем компьютере вычисление сумм  $n \lg n$ ,  $n^2$ ,  $n^5$ ,  $2^n$  и  $n!$ .

## Подсказки и ловушки

1. При разработке алгоритмов особое внимание уделяйте крайним случаям, например, пустым спискам, спискам с единственным элементом, а также полным спискам (для непрерывной реализации).
2. Удостоверьтесь в том, что все ваши переменные инициализированы должным образом.
3. Не поленитесь дважды проверить условия завершения циклов и удостоверьтесь в том, что по ходу выполнения программы циклы всегда завершаются.

4. При возникновении трудностей сформулируйте условия, которые будут выполняться как перед каждым шагом цикла, так и после каждого шага, и удостоверьтесь в том, что эти условия выдерживаются.
5. Избегайте усложнений ради усложнений. Если простой метод дает удовлетворительное решение, применяйте его.
6. Не изобретайте велосипед. Если готовая процедура адекватна для вашего приложения, используйте ее.
7. Последовательный поиск характеризуется небольшой скоростью, но высокой устойчивостью. Используйте этот алгоритм для небольших списков, а также в тех случаях, когда у вас есть сомнения, что элементы в списке упорядочены.
8. Будьте исключительно осторожны при перепрограммировании двоичного поиска. Убедитесь в правильности своего алгоритма и протестируйте его для всех крайних случаев.
9. Изображение дерева сравнений является прекрасным способом для трассировки действий алгоритма и для анализа его поведения.
10. Используйте понятие  $O$  большого для анализа алгоритмов больших приложений, но не используйте его в простых случаях.

## Обзорные вопросы

- |     |                                                                                                                                                                                                                                                                                                        |
|-----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 7.4 | 1. Назовите три условия, при которых последовательный поиск в списке предпочтительнее двоичного поиска.                                                                                                                                                                                                |
| 7.5 | 2. При поиске в списке из $n$ элементов сколько в среднем будет выполнено сравнений ключей процедурами <b>(a)</b> SequentialSearch, <b>(b)</b> Bynary1Search и <b>(c)</b> Bynary2Search?                                                                                                               |
|     | 3. Почему двоичный поиск реализуется только для непрерывных списков, но не для связных списков?                                                                                                                                                                                                        |
|     | 4. Нарисуйте дерево сравнений для Binary1Search при поиске в списке из <b>(a)</b> 1 элемента, <b>(b)</b> 2 и <b>(c)</b> трех элементов.                                                                                                                                                                |
|     | 5. Нарисуйте дерево сравнений для Binary2Search при поиске в списке из <b>(a)</b> 1 элемента, <b>(b)</b> 2 и <b>(c)</b> трех элементов.                                                                                                                                                                |
|     | 6. Если высота 2-дерева равна 3, каковы <b>(a)</b> наибольшее и <b>(b)</b> наименьшее число вершин, которые могут быть в таком дереве?                                                                                                                                                                 |
|     | 7. Определите термины <i>длина внутреннего пути</i> и <i>длина внешнего пути</i> для 2-дерева. Сформулируйте теорему о длине пути.                                                                                                                                                                     |
| 7.6 | 8. Каково в среднем наименьшее число сравнений, выполняемых любым методом, основанным на сравнении ключей, при просмотре списка из $n$ элементов?                                                                                                                                                      |
|     | 9. Если Binary2Search выполняет 20 сравнений в среднем для успешного поиска, сколько потребуется сравнений в среднем для неуспешного поиска в предположении, что вероятности для мишени оказаться меньше минимального ключа, между любой парой ключей и больше максимального ключа в списке одинаковы? |
| 7.7 | 10. В чем цель использования понятия $O$ большого?                                                                                                                                                                                                                                                     |

## Литература для дальнейшего изучения

Основным пособием для настоящей главы является книга [Knuth], том 3. (см. ссылку в главе 3) Последовательный поиск описан на стр. 389–405; двоичный поиск на стр. 406–414; затем рассмотрен поиск Фибоначчи, а далее идет раздел исторических сведений. Д. Кнут рассмотрел все затронутые нами методы, и много других. Он выполнил анализ алгоритмов значительно более подробно, чем это сделали мы, разрабатывая свои алгоритмы на псевдоассемблере и подсчитывая при этом число операций.

Проект гардероба взят из приведенной ниже ссылки, в которой можно также найти несколько интересных расширений и других приложений.

Joseph Bergin, «Coatroom: An ADT which is useful in implementation of object oriented programming (with positive pedagogical side effects)», *ACM SIGCSE Bulletin* 22 (1990), 45–46, 62.

Доказательство правильности алгоритма двоичного поиска является предметом статьи

Jon Bentley, «Programming pearls: Writing correct programs» (regular column), *Communications of the ACM* 26 (1983), 1049–1045.

В этой статье Бентли показывает, как сформулировать алгоритм двоичного поиска, удовлетворяющий поставленным условиям, замечает, что 90 процентов профессиональных программистов, которых он обучал, не смогли написать правильную программу в течение целого часа, и дает формальную верификацию правильности алгоритма.

В следующей статье изучаются 26 опубликованных вариантов двоичного поиска, указываются правильные и ошибочные рассуждения и делаются заключения, приложимые и к другим алгоритмам:

R. Lesuisse, «Some lessons drawn from the history of binary search algorithm», *The Computer Journal* 26 (1983), 154–163.

Теорема 7.4 (успешный и неуспешный поиск отнимает в среднем одно и то же время) взята из

T. N. Hibbard, *Journal of the ACM* 9 (1962), 16–17.

Интерполяционный поиск представлен в работе

C. C. Gotlieb and L. R. Gotlieb, *Data Types and Structures*, Prentice-Hall, Englewood Cliffs, N.J., pp 133–135.

# Глава 8

## Сортировка

---

В этой главе рассматриваются некоторые важные методы сортировки списков, как непрерывных, так и связанных. Одновременно мы разработаем дальнейшие средства, которые помогут нам при анализе алгоритмов.

### 8.1. Введение и обозначения

Мы живем в мире, помешавшемся на сохранении информации, и чтобы иметь возможность использовать эту информацию, мы должны хранить ее в каком-то разумном порядке. Библиотекари следят за тем, чтобы никто не перекладывал книги; налоговые службы прослеживают путь каждого заработанного нами доллара; кредитные организации следят вообще за каждым нашим шагом. Однажды я видел мультимедиа, в котором преданный делу хранения бумаг клерк, желая поразить босса, заявил ему: «Дайте я сначала разложу все эти бумаги в алфавитном порядке, а потом уже их можно будет выбросить». Если мы хотим оставаться хозяевами этого процесса, а не его рабами, мы должны научиться хранить документы в порядке и находить в них нужное.

практическая  
важность

Несколько лет назад было подсчитано, что более половины времени многих коммерческих компьютеров было потрачено на сортировку<sup>1</sup>. Возможно, сегодня это уже не так, поскольку для организации данных были разработаны сложные и остроумные методы, не требующие сохранения документов в каком-либо определенном порядке. Тем не менее, в конце концов информация поступает людям, и тогда она должна быть как-то рассортирована.

внешняя  
и внутренняя  
сортировка

В силу такой важности сортировки, для нее было разработано огромное количество алгоритмов. Фактически столько интересных идей было предложено для реализации методов сортировки, что эта тема вполне заслуживает отдельного курса. Среди различных сред,

---

<sup>1</sup> В отечественную литературу и даже в англо-русские словари по вычислительной технике прочно вошел не очень удачный перевод английского слова «sort» (сортировка, упорядочение) применительно к упорядочению списков. Говорят «пузырьковая сортировка», «быстрая сортировка», когда в действительности речь идет отнюдь не о сортировке, т. е. раскладывании объектов по нескольким (2–3) сортам, а об их упорядочении, т. е. о выстраивании всех объектов в единый ряд по порядку значения какого-либо параметра объекта. В настоящей книге, которая в значительной степени посвящена алгоритмам упорядочения, мы, скрепя сердце, используем привившийся термин. — *Прим. перев.*

требующих различных методов, наиболее важной классификацией является разделение на *внешние* и *внутренние* методы хранения, т. е. выделение задач, в которых требуется сортировать такое большое количество записей, что они могут храниться лишь на внешних носителях — дисках, лентах и других подобных устройствах, и задач, в которых записей не так много, и они могут содержаться во внутренней быстродействующей памяти. В этой главе мы будем обсуждать только внутреннюю сортировку.

ссылка

Нашей целью не является детальное изучение внутренних методов сортировки. Если это вас интересует, обратитесь к фундаментальной работе [Knuth], том 3 (см. ссылку в главе 3). В этой книге описано около 25 методов сортировки и указывается при этом, что это «только малая часть разработанных к настоящему времени алгоритмов». Мы рассмотрим подробно только несколько методов, выбранных, во-первых, ввиду их качества — каждый из них будет лучшим вариантом при определенных обстоятельствах, во-вторых, потому что они иллюстрируют значительную часть идей, используемых во всем многообразии методов сортировки, и, в-третьих, в силу их относительной простоты, облегчающей их реализацию и понимание, так что нам не придется вдаваться в слишком большое количество деталей и тонкостей. Значительное количество вариантов этих методов предлагаются далее в качестве упражнений.

обозначения

На протяжении этой главы мы используем те же обозначения, что и в предыдущих главах, так что  $L$  будет *списком* элементов, требующих упорядочения по полю *ключа*, входящего в каждый элемент списка. Объявления для списка и имени, назначенные различным типам и операциям, будут таким же, как и предыдущих главах.

Одна возможность должна будет рассматриваться нами особо тщательно. Два или несколько элементов списка могут иметь один и тот же ключ. В этом случае, т. е. при наличии дублирующих ключей, процедуры сортировки могут приводить к различному порядку этих ключей в конечном списке. Если порядок элементов с одинаковыми ключами важен для приложения, мы должны учесть это при разработке алгоритма сортировки.

базовые операции

В процессе изучения алгоритмов поиска вскоре стало ясно, что полный объем выполняемой работы тесно связан с количеством операций сравнения ключей. То же наблюдение справедливо и для алгоритмов сортировки, однако алгоритмы сортировки должны еще либо изменять указатели, либо перемещать элементы в списке, и время, потраченное на эти операции также оказывается важным, особенно в случае больших по объему элементов, хранящихся в последовательном списке. Наш анализ, очевидно, должен затронуть эти два базовых действия.

анализ

Как и раньше, интерес представляет как производительность алгоритма сортировки в худшем случае, так и его поведение в среднем. Для нахождения среднего мы рассмотрим, что произойдет, если алгоритм будет выполняться для всех возможных порядков элементов в списке (при  $n$  элементах таких вариантов будет всего  $n!$  штук) и усредним результаты.

## 8.2. Сортировка включением

### 8.2.1. Упорядоченные списки

Когда мы впервые начали обсуждать двоичный поиск в разделе 7.4, мы упомянули, что *упорядоченный список* фактически является новым абстрактным типом, который мы определили как список, в котором каждый элемент содержит ключ, и эти ключи упорядочены, т. е. если элемент  $i$  находится в списке перед элементом  $j$ , тогда ключ элемента  $i$  меньше или равен ключу элемента  $j$ . Мы предположили, что ключи можно сравнивать с помощью операций ' $<$ ' и ' $>$ ' (например, ключи являются числами), но алгоритм легко расширить на случай использования в качестве ключей слов или других символьных цепочек, которые можно расположить в алфавитном порядке.

Для упорядоченных списков мы будем часто использовать три новые операции, которым нет соответствия в других списках, поскольку они используют для нахождения элемента в списке не позиции, а ключи. Первой операцией будет *извлечение* элемента с указанным ключом из упорядоченного списка. Вторая операция заключается во *включении* нового элемента в упорядоченный список, причем ключ нового элемента используется для определения того, в какое место списка поместить новый элемент. Наконец, третья операция заключается в *сортировке* списка, который к этому моменту может быть не упорядочен, но содержит в каждом своем элементе ключ и, следовательно, из него можно образовать упорядоченный список.

Извлечение из упорядоченного списка по ключу есть в точности то же самое, что поиск. Эту проблему мы уже изучили в главе 7. Упорядоченное включение послужит нам основой нашего первого метода сортировки.

Давайте прежде всего рассмотрим непрерывный список. В этом случае для того, чтобы включить в список новый элемент, имеющиеся элементы придется перемещать, чтобы освободить место для нового. Чтобы найти место, в которое следует включить новый элемент, нам надо выполнить операцию поиска. Один из методов выполнения упорядоченного включения в непрерывный список заключается в следующем: сначала с помощью двоичного поиска определяется требуемое положение, затем элементы списка сдвигаются, после чего в список включается новый элемент. Этот метод мы оставим для упражнений. Поскольку перемещение элементов списка всегда, независимо от того, используется ли последовательный или двоичный поиск, требует много времени, часто оказывается, что использование последовательного поиска вместо двоичного не замедляет всю операцию. При этом выполнение последовательного поиска *от конца* списка позволяет скомбинировать поиск и перемещение элементов в одном цикле, тем самым снизив издержки метода.

Пример упорядоченного включения приведен на рис. 8.1. Мы начинаем с упорядоченного списка, показанного на рис. 8.1, (a), и хотим включить в список новый элемент hen (курица). В отличие от варианта InsertOrder из раздела 7.4, мы начинаем сравнивать ключи не от начала

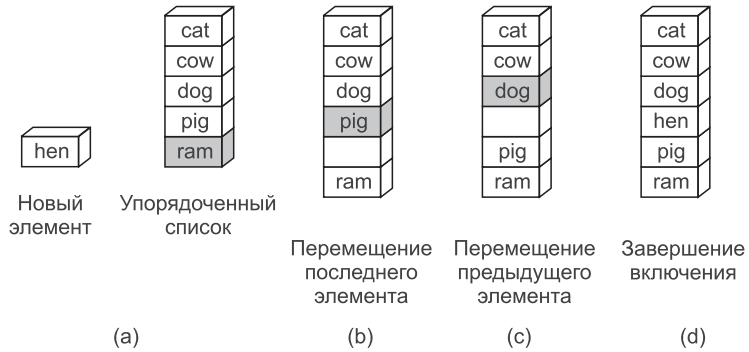


Рис. 8.1. Упорядоченное включение

списка, а от его *конца*. Мы сравниваем новый ключ *hen* с последним ключом *ram* (баран), который показан на рис. 8.1, (a) серым кубиком. Поскольку *hen* располагается перед *ram*, мы перемещаем *ram* на одну позицию вниз, оставив одну пустую позицию, как это показано на рис. 8.1, (b). Далее мы сравниваем *hen* с ключом *pig* (свинья), показанным на рис. 8.1, (b) серым кубиком. Опять *hen* располагается ранее *pig*, поэтому мы сдвигаем *pig* вниз и сравниваем *hen* с ключом *dog* (собака), показанным на рис. 8.1, (c) серым кубиком. Поскольку *hen* располагается за *dog*, мы нашли правильное место для *hen* и можем завершить операцию включения, как это показано на рис. 8.1, (d).

### 8.2.2. Сортировка методом включения

Наш первый метод сортировки списка основан на идее включения в упорядоченный список. Для сортировки неупорядоченного списка мы изымаем из него по одному элементу и затем включаем каждый из них в первоначально пустой новый список, следя за тем, чтобы элементы этого списка всегда располагались в правильном порядке в соответствии с их ключами.

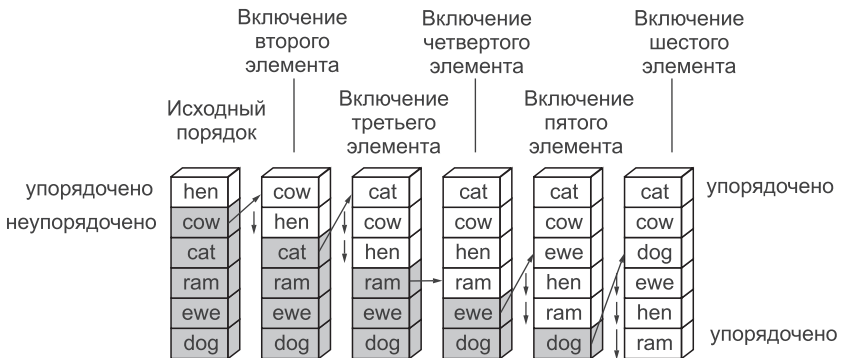


Рис. 8.2. Пример упорядочения включением

пример

Этот метод проиллюстрирован на рис. 8.2, где показаны шаги, требуемые для сортировки списка из шести слов. На каждой стадии сортировки слова, которые еще не были включены в упорядоченный список, показаны серыми кубиками, а упорядоченная часть списка — белыми. На начальной стадии первое слово не показано упорядоченным, поскольку список длиной 1 элемент всегда является упорядоченным. Все остальные слова показаны неупорядоченными (серыми). На каждой стадии процесса первое неупорядоченное слово (показанное в самом верхнем сером кубике) помещается на свое правильное место в упорядоченной части списка. Чтобы предоставить место для включения, некоторые упорядоченные слова должны перемещаться вниз по списку. Каждое такое перемещение слова показано на рис. 8.2 стрелкой. Начав с конца упорядоченной части списка, мы можем перемещать элементы одновременно с операциями сравнения ключей с целью нахождения правильных мест для новых элементов.

Основной шаг, требуемый для включения элемента, обозначенного  $i$ , в упорядоченную часть списка, показан на рис. 8.3. В приведенной ниже процедуре мы используем переменную  $L.count$  так чтобы и упорядоченный, и неупорядоченный списки занимали один и тот же массив  $L.entry$ . Теперь мы можем написать алгоритм.

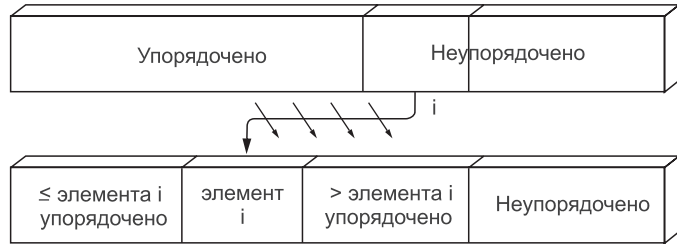


Рис. 8.3. Основной шаг сортировки включением в непрерывном списке

```

procedure InsertionSort(var L: List);           { Сортировка включением }
{ Pre:   Непрерывный список L уже создан. Каждый элемент L содержит
           ключ.
  Post:  Элементы списка L упорядочены таким образом, что ключи
           всех элементов располагаются в неубывающем порядке. }

var
  firstunsorted,           { позиция первого неупорядоченного элемента }
  place: position;         { просмотр упорядоченной части списка }
  current: listentry;      { содержит элемент, временно удаленный из списка }
  found: Boolean;

begin                                     { процедура InsertionSort }
with L do
  for firstunsorted := 2 to count do
    if entry[firstunsorted].key < entry[firstunsorted - 1].key then
      begin
        place := firstunsorted;
        current := entry[firstunsorted];
        { вытолкнем неупорядоченный элемент из списка }

```



```

repeat                                     { будем сдвигать все элементы,
                                           пока не найдем правильное место }
    place := place - 1;
    entry [place + 1] := entry [place];    { позиция place теперь пуста }
    if place = 1 then
        found := true;
    else
        found := (entry [place - 1].key <= current.key)
    until found;
    entry [place] := current
end
end;                                       { процедура InsertionSort }

```

Ход программы почти очевиден. Поскольку список с единственным элементом всегда упорядочен, цикл по `firstunsorted` начинается со второго элемента. Если этот элемент находится в правильном месте, ничего не делается. В противном случае новый элемент извлекается из списка в переменную `current`, а цикл **repeat ... until** проталкивает элементы в списке на одну позицию вниз пока не будет найдена правильная позиция, после чего `current` включается в эту позицию и программа переходит к обработке следующего неупорядоченного элемента. Случай, когда `current` должен быть помещен в первую позицию списка, должен рассматриваться особо, поскольку в этом случае отсутствует элемент с меньшим ключом, на котором должен прекращаться поиск. Мы обрабатываем этот особый случай с помощью булевой переменной `found`, значение которой устанавливается в предложении **if**.

### 8.2.3. Связный вариант

алгоритм

В случае связного варианта сортировки включением нет необходимости начинать поиск от *конца* списка, поскольку нам не надо будет перемещать данные. Мы просматриваем исходный список, извлекая каждый элемент по очереди и помещая его на правильное место в упорядоченном списке. Переменная-указатель `lastsorted` будет указывать на конец упорядоченной части списка, а `lastsorted↑.nextnode` — на первый элемент, который еще не был включен в упорядоченный подсписок. Указатель `firstunsorted` также будет указывать на этот элемент, и мы будем использовать указатель `current` для просмотра упорядоченной части списка, чтобы найти в нем, куда включить `firstunsorted↑`. Если `firstunsorted↑` должен быть помещен перед текущим головным элементом списка, мы включаем его туда. В противном случае мы перемещаем указатель `current` вниз по списку, пока не получим `current↑.entry.key >= firstunsorted↑.entry.key`, и тогда включаем `firstunsorted↑` перед `current↑`. Чтобы иметь возможность включения элемента перед `current↑`, мы следим за тем, чтобы второй указатель `trailing` был всегда на одну позицию ближе к голове списка, чем `current`.

завершение цикла

**Сигнальной меткой** называется дополнительный элемент, добавляемый к одному из концов списка, чтобы обеспечить завершение цикла без необходимости включать в алгоритм отдельную проверку. Поскольку имеет место

$$\text{lastsorted}\uparrow.\text{nextnode} = \text{firstunsorted}$$

узел  $\text{firstunsorted} \uparrow$  уже находится на своем месте и может служить сигнальной меткой при поиске, что упрощает цикл перемещения  $\text{current}$ .

Наконец стоит заметить, что список с 0 или 1 элементом уже упорядочен, поэтому мы можем проверять эти случаи отдельно, тем самым устраняя проверки на тривиальность в других местах. Детали алгоритма можно изучить по приведенному ниже тексту процедуры, а также по рис. 8.4.

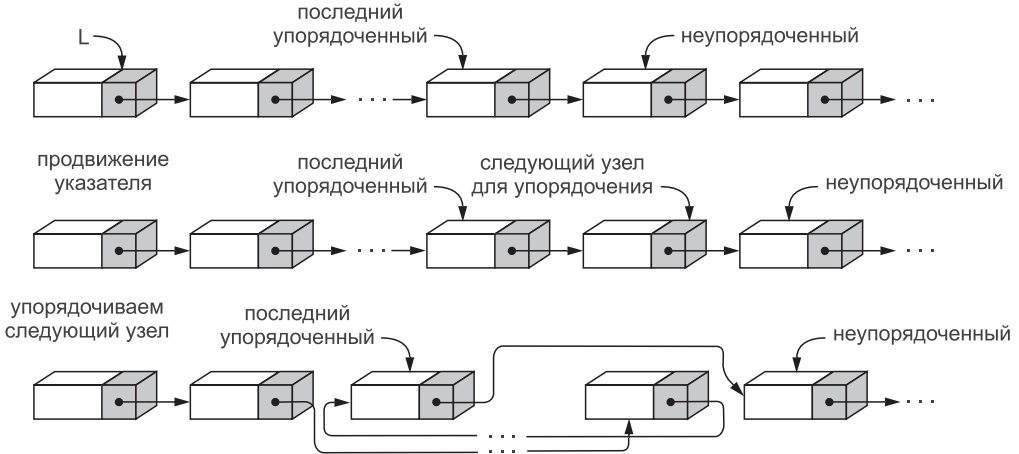


Рис. 8.4. Трассировка сортировки включением в связный список

```

procedure InsertionSort(var L: List); { Сортировка включением }
{ Pre: Связный список L уже создан. Каждый элемент L содержит ключ.
  Post: Элементы списка L были упорядочены таким образом, что
        ключи всех элементов располагаются в неубывающем порядке.}

var
  firstunsorted, { первый неупорядоченный узел, подлежащий включению }
  lastsorted,    { хвост упорядоченного подсписка }
  current,       { используется для просмотра упорядоченного подсписка }
  trailing: listpointer; { следующая позиция за current }

begin { процедура InsertionSort }
  if L.head <> nil then
    begin { пустой список всегда упорядочен }
      lastsorted := L.head; { единственный узел делает }
                           подсписок упорядоченным }
    while lastsorted $\uparrow$ .nextnode <> nil do
      begin
        firstunsorted := lastsorted $\uparrow$ .nextnode;
        if firstunsorted $\uparrow$ .entry.key < L.head $\uparrow$ .entry.key then
          begin { включаем firstunsorted $\uparrow$  в качестве головного элемента }
            упорядоченного списка}
            lastsorted $\uparrow$ .nextnode := firstunsorted $\uparrow$ .nextnode;
            { продвинем сигнальную метку }
            firstunsorted $\uparrow$ .nextnode := L.head;
            L.head := firstunsorted;
          end
        end
      end
    end
  end

```

```

else begin
    { поиск в упорядоченном подсписке
      с целью включения firstunsorted↑ }
    trailing := L.head;
    current := trailing↑.nextnode;
    while firstunsorted↑.entry.key > current↑.entry.key do
    begin
        trailing := current;
        current := trailing↑.nextnode;
    end;
    { firstunsorted↑ теперь располагается между trailing↑ и current↑ }
    if firstunsorted = current then
        lastsorted := firstunsorted;
    else begin
        lastsorted↑.nextnode := firstunsorted↑.nextnode;
        firstunsorted↑.nextnode := current;
        trailing↑.nextnode := firstunsorted
    end
end
end
end
end;
{ процедура InsertionSort }

```

Несмотря на то, что детали алгоритма для связного варианта списка сильно отличаются от непрерывного варианта, вы можете заметить, что в обоих случаях используется один и тот же базовый метод. Единственное различие заключается в том, что для непрерывного варианта поиск в упорядоченном подсписке осуществляется от его конца, в то время как для связного варианта поиск идет в порядке возрастания позиций в списке.

## 8.2.4. Анализ

Поскольку базовые идеи обоих вариантов совпадают, давайте проанализируем производительность лишь непрерывного варианта программы. Кроме того, мы ограничимся рассмотрением случая, когда список  $L$  первоначально характеризуется случайным порядком (что означает равновероятность любых расположений ключей). Если мы имеем дело с элементом  $i$ , сколь далеко назад мы должны пройти, чтобы включить его? Всего имеется  $i$  возможных позиций: вообще не перемещать элемент, переместить его на одну позицию или вплоть до  $i - 1$  позиции к началу списка. Учитывая случайный характер списка, все эти случаи равновероятны. Поэтому вероятность элементу остаться без перемещения равна  $1/i$ , причем в этом случае выполняется лишь одно сравнение ключей, и ни одного перемещения.

Оставшийся случай, когда элемент  $i$  должен быть перемещен, возникает с вероятностью  $(i - 1)/i$ . Начнем с подсчета среднего числа шагов в цикле **repeat**. Поскольку все  $i - 1$  возможных позиций равновероятны, среднее число шагов равно

$$\frac{1 + 2 + \dots + (i - 1)}{i - 1} = \frac{(i - 1)i}{2(i - 1)} = \frac{i}{2}.$$

В каждом из этих шагов выполняется одно сравнение ключей и одна операция присваивания, плюс еще вне цикла выполняется одно сравнение ключей и два присваивания элементов. Отсюда в этом втором случае элемент  $i$  требует в среднем  $\frac{1}{2}i+1$  сравнений и  $\frac{1}{2}i+2$  присваиваний.

Объединив эти два случая с учетом их относительных вероятностей, получим

$$\begin{aligned} \frac{1}{i} \times 1 + \frac{i-1}{i} \times \left( \frac{i}{2} + 1 \right) &= \frac{i+1}{2} \\ \frac{1}{i} \times 0 + \frac{i-1}{i} \times \left( \frac{i}{2} + 2 \right) &= \frac{i+3}{2} - \frac{2}{i} \end{aligned}$$

сравнений и

присваиваний.

Нам надо сложить все эти числа от  $i = 2$  до  $i = n$ , но чтобы избежать арифметических сложностей, мы прежде всего используем понятие  $O$  большого (см. раздел 7.7.2) для аппроксимации каждого из этих выражений путем подавления членов, ограниченных константой. Мы тогда получаем  $\frac{1}{2}i + O(1)$  и для числа сравнений, и для числа присваиваний элементов. Выполняя эту аппроксимацию, мы фактически ограничиваемся рассмотрением действий в главном цикле, не заботясь об операциях вне цикла или вариациях алгоритма, при которых объем работы изменяется лишь в ограниченных пределах.

Чтобы сложить  $\frac{1}{2}i + O(1)$  для значений  $i$  от  $i = 2$  до  $i = n$ , мы применим теорему А.1 (сумма целых чисел от 1 до  $n$ ), получая

$$\sum_{i=2}^n \left( \frac{1}{2}i + O(1) \right) = \frac{1}{2} \sum_{i=2}^n i + O(n) = \frac{1}{4}n^2 + O(n)$$

и для числа сравнений, и для числа присваиваний элементов.

Пока что нам не с чем сравнить это число, однако мы можем заметить, что по мере роста  $n$  вклад от члена, включающего  $n^2$ , становится значительно больше, чем от оставшихся членов, собирательно обозначенных  $O(n)$ . Отсюда по мере увеличения длины списка время, требуемое для включения, растёт как квадрат его длины.

Худший случай анализа сортировки включением мы оставим для упреждений. Мы можем сразу же заметить, что лучший случай для включения возникает, когда список уже находится в упорядоченном состоянии, и когда сортировка включением не выполнит никакой работы, за исключением  $n - 1$  сравнений ключей. Теперь мы можем показать, что не существует метода сортировки, который был бы эффективнее в своем лучшем случае.

#### Теорема 8.1

*Проверка того, что список из  $n$  элементов находится в упорядоченном состоянии, требует по меньшей мере  $n - 1$  сравнений ключей.*

#### Доказательство

Рассмотрим произвольную программу, которая проверяет, находится ли список из  $n$  элементов в упорядоченном состоянии или нет (и, возможно, упорядочивает его, если он еще не упорядочен). Программа прежде всего выполнит сравнение ключей, и это сравнение затронет некоторую пару элементов списка. Позже по меньшей мере один из этих элементов будет сравниваться с третьим, потому что в противном случае не

будет возможности определить, где эти два элемента должны находиться в списке по отношению к третьему. Таким образом, это второе сравнение затрагивает только один новый элемент, ранее не участвовавший в операциях сравнения. Продолжая рассуждения таким же образом, мы увидим, что должно выполняться еще одно сравнение, затрагивающее один из первых трех элементов и один новый элемент. Заметьте, что мы выбираем здесь сравнения не обязательно в том порядке, в котором они выполняются нашим алгоритмом. Итак, за исключением первого сравнения, каждое из выбранных нами сравнений вовлекает лишь один новый элемент из числа тех, что еще не сравнивались. Все  $n$  элементов должны рано или поздно поучаствовать в сравнении, так как нет способа определить, находится ли элемент на правильном месте, если его не сравнить по меньшей мере с одним другим элементом. В результате участие всех  $n$  элементов требует по меньшей мере  $n - 1$  сравнений, что и требовалось доказать.

конец  
доказательства

Имея эту теорему, мы обнаруживаем одно из преимуществ сортировки включением: такое упорядочение проверяет, не упорядочен ли уже список, за минимально возможное время. Далее, сортировка включением остается превосходным методом для тех случаев, когда список находится в почти упорядоченном состоянии, и лишь немногие элементы расположены далеко от своих правильных позиций.

## Упражнения 8.2

**E1.** Выполните вручную трассировку шагов при сортировке включением для каждого из приведенных ниже списков. В каждом случае подсчитайте число выполненных сравнений и число перемещений элементов.

(a) Следующие три слова надо расположить в алфавитном порядке:

triangle    square    pentagon  
[треугольник    квадрат    пятиугольник]

(b) Три слова из пункта (a) следует упорядочить соответственно числу сторон соответствующего многоугольника в возрастающем порядке

(c) Три слова из пункта (a) следует упорядочить соответственно числу сторон соответствующего многоугольника в убывающем порядке

(d) Следующие семь чисел следует расположить в возрастающем порядке:

26   33   35   29   19   12   22

(e) Те же семь чисел с другим исходным порядком следует расположить также в возрастающем порядке:

12   19   33   26   29   35   22

(f) Следующий список из 14 имен следует расположить в алфавитном порядке:

Tim Dot Eva Roy Tom Kim Guy Ami Jon Ann Jim Kay Ron Jan

**E2.** Какой исходный порядок в списке ключей даст наихудший случай для сортировки включением последовательного списка? А связанного?

- Е3.** Сколько сравнений и присваиваний элементов выполнит операция сортировки включением для последовательного списка в наихудшем случае?
- Е4.** Модифицируйте связный вариант сортировки включением так, чтобы список, уже упорядоченный или почти упорядоченный, обработался бы максимально быстро.

## **Программные проекты 8.2**

- P1.** Напишите программу, которую можно использовать для тестирования и оценки производительности алгоритма сортировки включением и, позже, других методов сортировки. Вам следует руководствоваться следующими указаниями.
- (a)** Создайте несколько файлов с целыми числами, которые будут использоваться для тестирования методов сортировки. Сделайте эти файлы разного размера, например, 20, 200 и 2000. Расположите числа в файлах по порядку, в обратном порядке, в случайном порядке и частично упорядоченными. Сохраняя все эти тестовые данные в файлах (вместо того, чтобы генерировать их в программе с помощью генератора случайных чисел при каждом запуске программы тестирования), вы получите возможность использовать одни и те же данные для тестирования различных методов, что облегчит сравнение их производительности. [Программа для создания этих тестовых файлов включена в раздел 4.2 приложения C.]
  - (b)** Напишите программу, управляемую меню, для тестирования различных методов сортировки. Один из пунктов меню позволяет прочитать файл с целыми числами и поместить его в список. Другие пункты служат для запуска различных методов сортировки списка, вывода неупорядоченного или упорядоченного списка и завершения программы. После того, как список будет упорядочен и (возможно) выведен на экран, его следует отбросить, чтобы тестирование другого метода сортировки началось с тех же самых входных данных. Для этого можно либо скопировать неупорядоченный список в другой список и начать упорядочивать эту копию, либо так организовать программу, чтобы она перед тем, как приступить к операции сортировки читала бы файл данных заново.
  - (c)** Включите в программу код, который будет вычислять и выводить (1) затраченное время процессора, (2) число сравнений ключей и (3) число присваиваний элементов в процессе упорядочения списка. Чтобы подсчитать число сравнений и присваиваний, вам придется ввести в программу глобальные переменные и включить в каждую процедуру сортировки код, инкрементирующий эти глобальные переменные при каждом сравнении ключей или присваивании элементов.
  - (d)** Используйте пакет обработки непрерывных списков, разработанный в разделе 6.2.1, добавьте в него непрерывный вариант сор-

тировки включением и соберите статистику производительности сортировки включением для непрерывных списков с целью дальнейшего сравнения с другими методами.

- (е) Используйте пакет обработки связных списков, разработанный в разделе 6.2.2, добавьте в него связный вариант сортировки включением, соберите статистику производительности сортировки включением и сравните ее с данными для упорядочения непрерывных списков. Почему число присвоений элементов оказывается для этого варианта не столь важным?

**P2.** Перепишите непрерывный вариант процедуры InsertionSort так, чтобы она для нахождения места включения следующего элемента использовала двоичный поиск. Сравните время, необходимое для упорядочения списка, с данными для первого варианта процедуры InsertionSort. Целесообразно ли использовать двоичный поиск в связанном варианте процедуры InsertionSort? Почему да или почему нет?

**P3.** Существует даже еще более простой метод сортировки, в котором вместо двух указателей для перемещения по списку используется только один. Мы можем назвать этот метод *сортировкой сканированием*. В этом методе мы начинаем с одного конца списка и продвигаемся вперед, сравнивая соседние пары ключей, пока не обнаружим пару, расположенную в неправильном порядке. В этом случае элементы пары обмениваются местами, и мы продолжаем перемещение в обратном направлении, продолжая обменивать элементы пар местами, пока не встретится пара, расположенная в правильном порядке. В этой точке мы знаем, что мы переместили один элемент назад до требуемой позиции, поэтому первая часть списка оказывается упорядоченной, однако, в отличие от сортировки включением, мы забыли, до каких пор вперед мы упорядочили список, поэтому мы просто реверсируем направление просмотра и снова выполняем сортировку вперед, ища пару, расположенную в неправильном порядке. Когда мы достигаем дальнего конца списка, сортировка завершилось.

- (а) Напишите Pascal-программу, реализующую сортировку сканированием для непрерывных списков. Ваша программа должна использовать лишь одну позиционную переменную (кроме L.count), одну переменную типа entry для обмена элементов пары и больше никаких локальных переменных.

- (б) Сравните время выполнения вашей программы с соответствующими данными для процедуры InsertionSort.

**P4.** Хорошо известный алгоритм упорядочения, называемый *пузырьковой сортировкой*, выполняется путем сканирования списка слева направо и обмена каждой пары соседних элементов, если их ключи оказываются в неправильном порядке. В этом первом проходе наибольший ключ в списке «всплывет» (как пузырек воздуха в воде) в конец списка, однако более ранние ключи все еще могут находиться в неупорядоченном состоянии. Поэтому проход сканирования неупорядоченных пар помещается в цикл, который сначала выполняет сканирующий проход по всем элементам до L.count, а в каждом следующем шаге останавли-



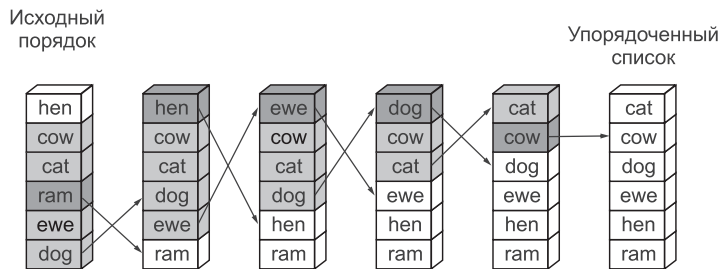
вается на одну позицию раньше. **(а)** Напишите Pascal-программу пузырьковой сортировки. **(б)** Определите производительность пузырьковой сортировки для списков разного рода и сравните результаты с данными для сортировки включением.

## 8.3. Сортировка методом выбора

Сортировка методом включения имеет один существенный недостаток. Даже после того, как большая часть элементов будет упорядочена в первой части списка, включение следующего элемента может потребовать перемещения многих из них. Все перемещения при сортировке включением выполняются только на одну позицию за раз. В результате для того, чтобы переместить элемент на 20 позиций вверх по списку, требуется 20 отдельных перемещений. Если элементы невелики по объему и состоят, возможно, только из одного ключа, или если элементы находятся в связном списке, тогда большое число перемещений не потребует много времени. Если, однако, элементы велики и представляют собой записи, содержащие сотни компонентов, как, например, личные дела сотрудников или студентов, и при этом записи должны храниться в непрерывной памяти, тогда значительно более эффективным было бы перемещение элемента непосредственно в его конечную позицию. Наш следующий метод сортировки выполняет именно такую операцию.

### 8.3.1. Алгоритм

Пример такого метода сортировки проиллюстрирован на рис. 8.5, где показаны шаги, необходимые для размещения шести слов в списке в алфавитном порядке. На первом шаге мы сканируем список, чтобы найти слово, которое по алфавиту будет последним. Это слово, *ram* [баран], показано в темно-сером кубике. Далее мы обмениваем это слово со словом *dog* [собака] в последней позиции, как показано во второй колонке рис. 8.5. Теперь мы повторяем этот процесс с более коротким списком, в котором не рассматривается последний элемент. Опять слово, которое



Темно-серые кубики обозначают наибольшие неупорядоченные ключи. Светло-серые кубики обозначают прочие неупорядоченные ключи.

**Рис. 8.5.** Пример сортировки методом выбора



должно быть последним, именно, `hen` [курица] показано в темно-сером кубике; оно обменивается с последним словом списка `ewe` [ягненок] из тех, которые еще следует рассматривать. Далее этот процесс продолжается. На рис. 8.5 слова, еще не упорядоченные, показаны в светло-серых кубиках, за исключением слова, которое по порядку должно идти самым последним; оно показано в темно-сером кубике. Когда неупорядоченный список сокращается до длины 1, процесс прекращается.

Описанный метод, преобразованный в алгоритм, называется **сортировкой методом выбора**. Основной шаг этого алгоритма проиллюстрирован на рис. 8.6. Элементы с большими ключами будут упорядочиваться и размещаться в конце списка. Элементы с меньшими ключами еще не упорядочены. Мы просматриваем неупорядоченные элементы, чтобы найти среди них элемент с наибольшим ключом, и обмениваем его с последним неупорядоченным элементом. Действуя таким образом, в каждом шаге главного цикла еще один элемент помещается в свою окончательную позицию.

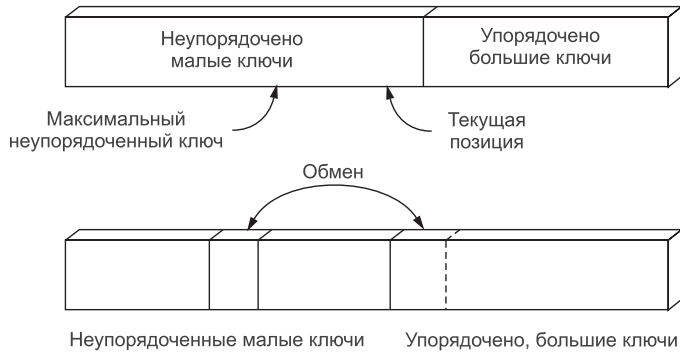


Рис. 8.6. Основной шаг при сортировке методом выбора

### 8.3.2. Непрерывная реализация

Поскольку сортировка методом выбора минимизирует перемещение данных, помещая в каждом проходе один элемент в свою окончательную позицию, алгоритм прежде всего полезен для обработки непрерывных списков с большими элементами, для которых перемещение элементов является затратной процедурой. Если элементы малы, или список является связным, так что при сортировке надо изменять только значения указателей, тогда сортировка включением обычно оказывается более быстрым методом, чем сортировка выбором. По этой причине мы приводим здесь только непрерывный вариант сортировки выбором. Алгоритм использует функцию с именем `MaxKey`, которая находит максимальный ключ в части списка `L` в соответствии со своими параметрами. Процедура `Swar` просто обменивает два элемента с данными индексами. Для удобства приводимого ниже обсуждения мы оформляем эти две процедуры как отдельные подпрограммы.



8.3.3. Анализ

порядок  
не существует

Возвращаясь к анализу алгоритмов, мы видим, что наиболее замечательным свойством последнего алгоритма является то, что оба его цикла написаны в форме **for ... do ...**. Это означает, что мы можем вычислить заранее, сколько будет шагов в каждом цикле. В смысле количества выполняемых сравнений при сортировке выбором исходный порядок элементов в списке не имеет значения. Поэтому для списка, который в начале процесса уже почти правилен, сортировка выбором будет работать значительно медленнее, чем сортировка включением. С другой стороны, сортировка выбором имеет в качестве преимущества предсказуемость: в худшем случае время выполнения будет мало отличаться от лучшего случая.

преимущество  
сортировки  
выбором

Основное преимущество метода сортировки выбором относится к перемещению данных. Если элемент находится в своей правильной конечной позиции, он никогда не будет перемещаться. Каждый раз, когда происходит обмен пары элементов, по меньшей мере один из них перемещается в свою окончательную позицию, поэтому при сортировке списка из  $n$  элементов никогда не будет выполняться более  $n - 1$  обменов. Это лучший результат для любого метода, в котором перемещение элементов целиком основано на их обмене.

число сравнений  
для сортировки  
выбором

Мы можем проанализировать производительность процедуры SelectionSort так же, как мы ее программировали. Главная процедура не делает ничего, за исключением организационных действий и вызова подпрограмм. Процедура Swap вызывается  $n - 1$  раз, и при каждом вызове выполняются 3 присваивания элементов, что всего дает  $3(n - 1)$  присваиваний. Функция MaxKey вызывается  $n - 1$  раз, причем длина подсписка изменяется от  $n$  до 2. Если  $t$  есть число элементов в той части списка, для которой вызвана эта функция, тогда MaxKey для нахождения максимального ключа выполняет в точности  $t - 1$  сравнений ключей. Таким образом, всего выполняется  $(n - 1) + (n - 2) + \dots + 1 = \frac{1}{2}n(n - 1)$  сравнений ключей, что можно, аппроксимируя, обозначить как  $\frac{1}{2}n^2 + O(n)$ .

8.3.4. Сравнения

Давайте на минуту прервемся для сравнения числа операций в методах сортировки выбором и сортировки включением. Результаты таковы:

	Выбор	Включение (в среднем)
Присваивания элементов	$3.0n + O(1)$	$0.25n^2 + O(n)$
Сравнения ключей	$0.5n^2 + O(n)$	$0.25n^2 + O(n)$

Относительные достоинства наших двух методов видны из приведенных в этой таблице данных. Когда  $n$  становится большим,  $0.25n^2$  становится значительно больше  $3n$ , и если перемещение элементов является процессом медленным, сортировка включением займет значительно больше времени, чем сортировка выбором. Однако время, затрачиваемое на сравнения, для сортировки выбором примерно в два раза меньше, чем

для сортировки включением. Если элементы списка невелики, так что их перемещение не занимает много времени, сортировка включением будет выполнена быстрее сортировки выбором.

### Упражнения 8.3

**E1.** Выполните вручную трассировку шагов при сортировке выбором для каждого из приведенных ниже списков. В каждом случае подсчитайте число выполненных сравнений и число перемещений элементов.

(a) Следующие три слова надо расположить в алфавитном порядке:

triangle      square      pentagon  
[треугольник    квадрат    пятиугольник]

(b) Три слова из пункта (a) следует упорядочить соответственно числу сторон соответствующего многоугольника в возрастающем порядке

(c) Три слова из пункта (a) следует упорядочить соответственно числу сторон соответствующего многоугольника в убывающем порядке

(d) Следующие семь чисел следует расположить в возрастающем порядке:

26   33   35   29   19   12   22

(e) Те же семь чисел с другим исходным порядком следует расположить также в возрастающем порядке:

12   19   33   26   29   35   22

(f) Следующий список из 14 имен следует расположить в алфавитном порядке:

Tim Dot Eva Roy Tom Kim Guy Ami Jon Ann Jim Kay Ron Jan

**E2.** Существует простой алгоритм, называемый *сортировкой подсчетом*, который конструирует новый упорядоченный список из списка  $L$  при условии, что все ключи в  $L$  различаются. Алгоритм сортировки подсчетом проходит по списку  $L$  один раз, и для каждого ключа  $L.entry[i].key$  просматривает  $L$  с целью подсчета, сколько ключей имеют значения, меньшие, чем  $L.entry[i].key$ . Если число ключей оказалось равным  $s$ , тогда правильная позиция данного ключа в упорядоченном списке будет  $s + 1$ . Определите, сколько сравнений ключей будет выполнено алгоритмом сортировки подсчетом. Лучший ли это алгоритм, чем сортировка включением?

сортировка  
подсчетом

### Программные проекты 8.3

**P1.** Запустите программу тестирования, написанную в проекте P1 раздела 8.2 и сравните сортировку выбором с сортировкой включением (непрерывный вариант). Используйте в обоих случаях одни и те же файлы с тестовыми данными.

**P2.** Напишите и протестируйте связный вариант алгоритма сортировки выбором.

## 8.4. Сортировка методом Шелла

Как мы уже видели, в некотором отношении алгоритмы сортировки включением и выбором ведут себя противоположно. Сортировка выбором весьма эффективно перемещает элементы, но выполняет большое количество излишних сравнений. Сортировка включением в своем лучшем случае выполняет минимальное число сравнений, но неэффективно в плане перемещения элементов каждый раз только на одну позицию. Нашей целью теперь будет разработка нового метода, в котором по возможности будут отсутствовать недостатки первых двух. Давайте начнем с сортировки включением и спросим себя, как можно было бы уменьшить число перемещений элементов.

убывающие шаги

Причина, по которой метод сортировки включением может перемещать элементы только на одну позицию, заключается в том, что он выполняет сравнение только соседних ключей. Если бы мы модифицировали метод так, чтобы он сначала сравнивал далеко отстоящие ключи, тогда он смог бы и упорядочивать далеко отстоящие элементы. После этого можно было бы упорядочивать элементы, расположенные ближе друг к другу, пока наконец различие между сравниваемыми ключами не уменьшится до 1, что будет означать полное упорядочение списка. Эта идея была реализована в 1959 г. Д. Л. Шеллом (D. L. Shell) в методе сортировки, получившем его имя. Этот метод иногда также называют сортировкой с убывающим шагом. Перед тем, как приступить к формальному описанию алгоритма, рассмотрим простой пример упорядочения имен.

пример

На рис. 8.7 показано, что произойдет, если мы сначала упорядочиваем все имена, находящиеся на расстоянии 5 шагов друг от друга (так что в каждом таком списке будет всего два или три имени), затем упорядочиваем имена с шагом 3 и, наконец, выполняем окончательное обычное упорядочение включением (с шагом 1).

Легко увидеть, что хотя мы выполняем три прохода сквозь все имена, ранние проходы перемещают имена ближе к их конечным позициям, так что в последнем проходе (в котором выполняется обычная сортировка включением) все элементы находятся весьма близко к их конечным позициям, и сортировка выполняется быстро.

выбор шага

Нет никакой таинственности в выборе в качестве величина шага чисел 5, 3 и 1. Многие другие варианты будут работать так же хорошо или даже лучше. Однако выбор степеней 2, например, 8, 4, 2 и 1, будет, возможно, малоэффективным, поскольку те же самые ключи, которые сравнивались в первом проходе, будут снова сравниваться в следующем, в то время как выбор чисел, не кратных друг другу, позволит нам получить новую информацию из большего числа сравнений. Хотя был выполнен целый ряд исследований метода Шелла, ни один не смог доказать решающего преимущества одного способа выбора числа шагов перед другим. Были, однако, выдвинуты различные рекомендации. Если шаги выбраны близко друг к другу, как это мы сделали в своем примере, придется выполнить большее число проходов, хотя каждый из них будет, скорее всего, относительно быстрым. Если величина каждого шага

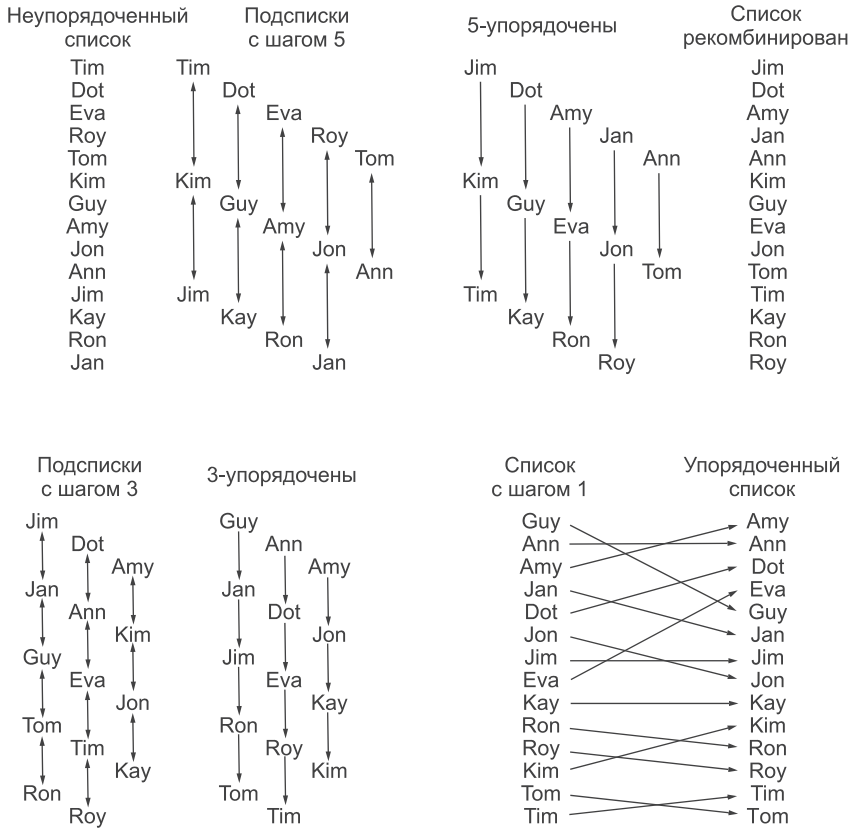


Рис. 8.7. Пример сортировки по методу Шелла

уменьшается в сильной степени, шагов будет меньше, но сами они будут длиннее. Единственным важным условием является величина последнего шага, которая должна быть равна 1, чтобы перед завершением процесса надежно упорядочился весь список. Для упрощения приведенного ниже алгоритма мы начнем с величины шага  $\text{increment} = L.\text{count}$  и в каждом следующем проходе будем уменьшать шаг следующим образом:

$\text{increment} := \text{increment} \text{ div } 3 + 1$

Теперь мы можем описать алгоритм для последовательных списков.

```

procedure ShellSort (var L: List); { Сортировка Шелла }
{ Pre: Непрерывный список L уже создан. Каждый элемент списка L
  содержит ключ.
  Post: Элементы списка L упорядочены таким образом, что ключи
  всех элементов располагаются в неубывающем порядке. }

var
  increment: integer; { расстояние между элементами в подписке }
  start: position; { начальная точка подписки }
begin { процедура ShellSort }
  increment := L.count;
  
```

```

repeat
  increment := increment div 3 + 1;
  for start := 1 to increment do
    SortInterval(start, increment, L)
    { модифицированная сортировка включением }
  until increment = 1
end;
{ процедура ShellSort }

```

Процедура SortInterval(start, increment, L) в точности совпадает с процедурой InsertionSort за исключением того, что список начинается с переменной start вместо 1 и приращение между последовательными значениями не равно 1, а задается через параметр increment. Детали модификации процедуры InsertionSort мы оставим для упражнений.

Поскольку последний проход при сортировке методом Шелла имеет шаг 1, этот метод фактически является вариантом сортировки включением, оптимизированным с помощью предварительной стадии упорядочения подписков посредством более крупных шагов. Отсюда следует, что доказательство правильности алгоритма Шелла в точности совпадает с доказательством алгоритма сортировки включением. И хотя у нас есть веские основания считать, что предварительная стадия существенно ускорит работу алгоритма, поскольку позволяет избавиться от множества перемещений элементов на одну позицию, фактически мы не доказали, что сортировка Шелла работает быстрее сортировки включением.

Анализ сортировки Шелла оказывается чрезвычайно сложной задачей, и к настоящему времени достаточно точные оценки числа сравнений и перемещений были получены только для особых условий. Было бы очень интересно узнать, как на эти оценки влияет выбор шагов, чтобы сделать этот выбор обоснованным и эффективным. Однако даже без полного математического анализа прогон на компьютере нескольких больших примеров убедит вас, что алгоритм Шелла весьма хорош. Были выполнены значительные эмпирические исследования этого алгоритма, и из них следует, что число перемещений, когда  $n$  велико, находится в диапазоне от  $n^{1.25}$  до  $1.6n^{1.25}$ . Эти оценки показывают существенное улучшение по сравнению с сортировкой включением.

анализ

#### Упражнения 8.4

- E1. Выполните упорядочение вручную 14 имен из столбца «Неупорядоченный список» на рис. 8.7 с помощью алгоритма Шелла с последовательностью шагов (a) 8, 4, 2 и 1 и (b) 7, 3 и 1. Подсчитайте число сравнений и перемещений для каждого из этих случаев.
- E2. Объясните, почему упорядочение Шелла плохо работает для связанных списков.

#### Программные проекты 8.4

- P1. Перепишите процедуру InsertionSort, чтобы она могла заменить собой процедуру SortInterval, встроенную в ShellSort.
- P2. Протестируйте ShellSort с помощью программы проекта P1 из раздела 8.2, используя те же файлы данных, что и для сортировки включением, и сравните результаты.

# 8.5. Нижние границы

Теперь, когда мы рассмотрели метод, дающий значительно лучшие результаты, чем наши первоначальные попытки, уместно задать вопрос:

*Насколько быстро вообще можно выполнить сортировку?*

Для ответа на этот вопрос мы ограничимся рассмотрением (как мы уже поступали при ответе на аналогичный вопрос, относящийся к процедуре поиска) методами сортировки, которые основаны исключительно на сравнении пар ключей.

Давайте возьмем произвольный алгоритм сортировки этого класса и рассмотрим, как он выполняет сортировку списка из  $n$  элементов. Представьте себе его дерево сравнений. На рис. 8.8 показаны деревья сравнений для сортировки включением и выбором применительно к трем числам  $a$ ,  $b$  и  $c$ . Каждому сравнению ключей соответствует внутренняя вершина (нарисованная в виде кружка). Листья дерева (квадратные узлы) показывают порядок, в котором располагаются числа после упорядочения.

дерево сравнений

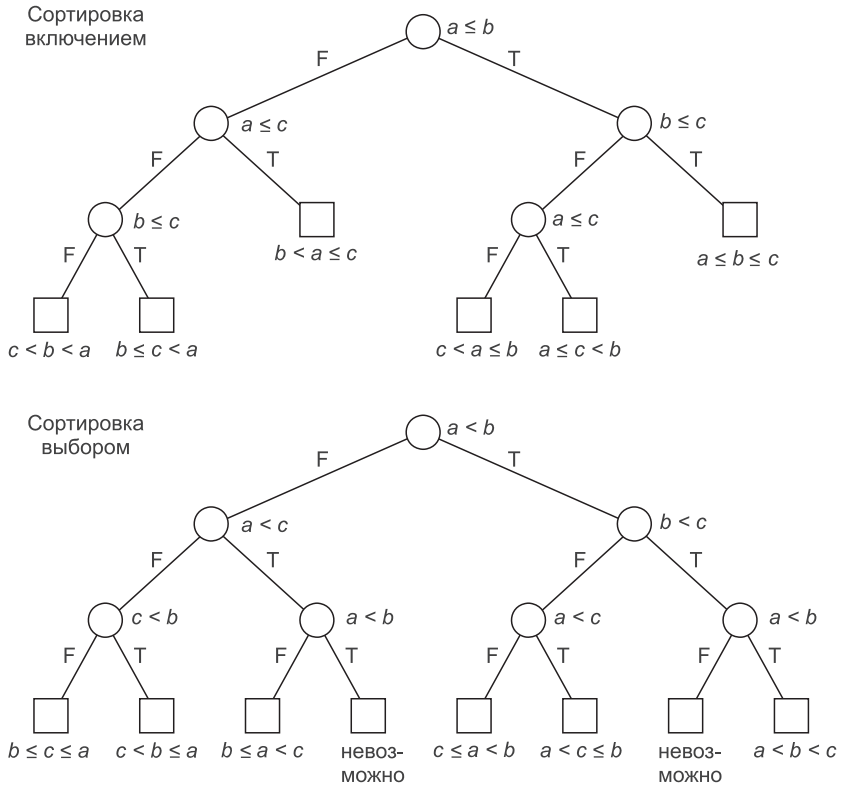


Рис. 8.8. Деревья сравнений для сортировки включением и выбором,  $n = 3$



Заметьте, что диаграммы отчетливо показывают, что в среднем при сортировке выбором выполняется большее число сравнений ключей, чем при сортировке включением. Фактически сортировка выбором сопряжена с излишними сравнениями, т. е. с повторением уже выполненных сравнений.

Дерево сравнений произвольного алгоритма сортировки позволяет наглядно представить некоторые черты алгоритма. Высота дерева равна максимальному числу выполняемых сравнений, и, таким образом, отображает поведение алгоритма в наихудшем случае. Длина внешнего пути, после деления на число листьев, дает среднее число сравнений, выполняемых алгоритмом. Дерево сравнений показывает все возможные последовательности сравнений, которые могут выполняться по всем возможным путям от корня к листьям. Поскольку эти сравнения управляют тем, как элементы списка выстраиваются в процессе сортировки, любые два различных порядка в списке должны приводить к различным решениям, и, следовательно, к различным путям по дереву, и закончиться на различных листьях. Число исходных порядков в списке из  $n$  элементов равно  $n!$  (см. раздел А.3.1), так что число листьев дерева тоже должно быть по меньшей мере  $n!$ . Лемма 7.5 утверждает, что высота дерева есть по меньшей мере  $\lceil \lg n! \rceil$ , а его длина внешнего пути по меньшей мере  $n! \lg n!$ . (Вспомним, что  $\lceil k \rceil$  обозначает наименьшее целое, не меньшее, чем  $k$ .) Преобразуя эти результаты в число сравнений, мы получаем

дерева  
сравнений: высота  
и длина пути

#### Теорема 8.2

*Любой алгоритм, который сортирует список из  $n$  шагов посредством сравнения ключей, должен в худшем случае выполнить по меньшей мере  $\lceil \lg n! \rceil$  сравнений ключей, а в среднем случае по меньшей мере  $\lg n!$  сравнений ключей.*

Формула Стирлинга (теорема А.5) дает приблизительную оценку факториала целого числа, которая, при использования логарифма по основанию 2, выглядит так:

аппроксимация  
 $\lg n!$

$$\lg n! \approx \left( n + \frac{1}{2} \right) \lg n - (\lg e)n + \lg \sqrt{2\pi} + \frac{\lg e}{12n}.$$

Константы в этом выражении имеют приблизительные значения

$$\lg e \approx 1.442695041 \quad \text{и} \quad \lg \sqrt{2\pi} \approx 1.325748069.$$

Аппроксимация Стирлинга  $\lg n!$  весьма точна, значительно точнее, чем это нам когда-либо потребуется при анализе алгоритмов. Для большинства практических случаев можно использовать следующую грубую аппроксимацию:

$$\lg n! \approx \left( n + \frac{1}{2} \right) \left( \lg n - 1 \frac{1}{2} \right) + 2,$$

а часто мы используем еще более простую аппроксимацию  $\lg n! = n \lg n + O(n)$ .

другие методы

Перед завершением этого раздела мы должны заметить, что иногда методы сортировки не используют сравнений и могут работать быстрее.

Например, если вы заранее знаете, что у вас есть 100 элементов, и что их ключи есть в точности числа от 1 до 100 в некотором порядке, и в них нет дубликатов, тогда лучшим способом сортировки будет не выполнение сравнений, а просто помещение конкретного элемента с ключом  $i$  в позицию  $i$ . При таком методе мы (по крайней мере временно) рассматриваем упорядочиваемые элементы не как список, а скорее как таблицу, и для нахождения правильного места в таблице для каждого элемента можем рассматривать ключ как индекс. Проект P1 предлагает развитие этой идеи до стадии алгоритма.

### Упражнения 8.5

- E1.** Изобразите деревья сравнений для (а) сортировки включением и (б) сортировки выбором применительно к четырем объектам.
- E2.** (а) Найдите метод сортировки для четырех ключей, оптимальный в смысле выполнения минимально возможного числа сравнений ключей в худшем случае. (б) Определите, сколько сравнений выполняет ваш алгоритм в среднем случае (применительно к четырем ключам). Модифицируйте ваш алгоритм так, чтобы он максимально приближался к нижней границе  $\lg 4! \approx 5.585$  сравнений ключей. Почему невозможно достичь этой нижней границы?
- E3.** Представьте себе, что у вас есть перемешанная колода из 52 карт, по 13 карт каждой из 4 мастей, и вы хотите упорядочить колоду так, чтобы каждая из 4 мастей была собрана вместе и внутри каждой масти все карты располагались в порядке их достоинств. Какой из указанных ниже методов будет самым быстрым?
- (а) Пройдите по колоде и отберите все трефы; затем упорядочьте их отдельно. Прделайте то же самое с бубнами, червами и пиками.
  - (б) Разделите колоду на 13 кучек согласно достоинству карт. Соберите эти 13 кучек снова вместе и разложите на четыре кучки согласно мастям. Соберите все кучки вместе.
  - (с) Выполните только один проход по всей колоде, помещая каждую карту в свою правильную позицию относительно ранее упорядоченных карт.

### Программные проекты 8.5.

Проекты сортировки в этом разделе являются специализированными методами, требующими ключей определенного типа, именно, псевдослучайных чисел между 0 и 1. Поэтому они не могут работать с программами тестирования, разработанными для других методов, и использовать данные, подготовленные для других методов этой главы.

- P1.** Сконструируйте список из  $n$  псевдослучайных чисел между 0 и 1. Подходящие значения для  $n$  — это 10 (для отладки) и 500 (для сравнения с результатами других методов). Напишите программу для упорядочения этих чисел в массив посредством метода *интерполяционной сортировки*. Прежде всего очистите массив (дайте всем элементам значение 0). Для каждого числа из старого списка умножьте его на  $n$ , возьмите целую часть и посмотрите в соответствующую позицию таб-

лицы. Если в этой позиции 0, поместите туда число. Если нет, сдвиньтесь влево или вправо (согласно относительной величине текущего числа и того, что находится на его месте), чтобы найти место для включения нового числа, перемещая при этом в случае необходимости элементы таблицы, чтобы освободить место для нового числа (как в случае сортировки включением). Покажите, что ваш алгоритм действительно упорядочит числа правильным образом. Сравните время его выполнения с временем работы других методов сортировки применительно к списку со случайными числами того же размера.

- P2.** [Предложено Б. Ли (B. Lee).] напишите программу, выполняющую сортировку дистрибутивным методом как это описано ниже. Возьмите в качестве ключей псевдослучайные числа между 0 и 1, как в предыдущем проекте. Создайте массив связанных списков и распределите ключи по связным спискам согласно их значениям. Связные списки можно либо упорядочивать по мере включения в них чисел, либо упорядочить их во втором проходе, когда все списки объединяются вместе в один упорядоченный список. Поэкспериментируйте, чтобы определить оптимальное число списков. (Похоже, что метод работает удовлетворительно, если число списков таково, что средняя длина каждого списка оказывается приблизительно 3.)

связная  
дистрибутивная  
сортировка

## 8.6. Сортировка методом разбиения

### 8.6.1. Базовые идеи

Начать с чистого листа иногда оказывается весьма полезно, и мы поступим именно так, забыв (временно) почти все, что мы узнали о сортировке. Попробуем приложить к нашим рассуждениям только один важный принцип, который уже использовался в изученных нами методах и который достаточно очевиден: значительно проще сортировать короткие списки, чем длинные. Если число упорядочиваемых элементов удваивается, выполняемая работа увеличивается более чем в два раза (для алгоритмов включения или выбора она возрастает приблизительно в четыре раза). Таким образом, если мы найдем способ разделения списка на два списка приблизительно одинакового размера, и упорядочим их по отдельности, мы сократим объем выполняемой работы. Если, например, вы работаете в библиотеке и получили тысячу библиографических карточек, которые следует расположить в алфавитном порядке, то неплохим методом будет разделение всех карточек на кучки согласно первой букве названия и сортировка затем этих кучек по отдельности.

короче значит  
проще

метод разбиения

Мы здесь опять сталкиваемся с приложением идеи разделения задачи на меньшие, но схожие подзадачи, т. е. идеи *разбиения* (этот метод, как уже отмечалось выше, иногда образно называют методом «разделяй и властвуй»).

Прежде всего заметим, что сравнения, выполняемые компьютером, обычно реализуются как двухвариантные условные переходы, поэтому

мы будем на каждой стадии процесса разделять сортируемые элементы на два списка.

Однако вы можете спросить, каким же методом сортировать сокращенный список? Поскольку мы (временно) забыли все рассмотренные нами методы, давайте просто использовать тот же самый метод разбиения, многократно подразделяя список. Но при этом мы не уйдем в бесконечность: сортировка списка, содержащего один элемент, не требует никакой работы, даже если мы не знакомы ни с какими методами сортировки.

В итоге давайте неформально опишем эскиз процедуры сортировки разбиением:

```

procedure Sort(list)                                { Сортировка }
if список имеет длину больше 1 then
begin
    Разобьем список на lowlist, highlist;
    Sort(lowlist);                                { сортируем нижнюю половину }
    Sort(hightlist);                              { сортируем верхнюю половину }
    Combine(lowlist,highlist);                    { объединяем }
end.
    
```

Мы еще должны решить, каким образом мы будем разделять список на два подсписка, а также каким образом после того, как они будут упорядочены, объединять их в единый список. Здесь можно использовать два метода, каждый из которых работает очень неплохо при определенных обстоятельствах.

сортировка  
слиянием

- *Сортировка слиянием.* В этом методе мы просто делим список на два подсписка насколько возможно равного размера и затем сортируем их по отдельности. После мы аккуратно сольем оба упорядоченных подсписка в единый упорядоченный список. Поэтому этот метод и называется методом *слияния*, или *объединения*.

быстрая  
сортировка

- *Быстрое упорядочение.* Второй метод требует больше работы на первом шаге деления списка на два подсписка, а конечный шаг слияния подсписков становится весьма простым. Этот метод был изобретен и назван *быстрой сортировкой* К. А. Р. Хоаром (С. А. R Hoare). Для деления списка мы сначала выбираем некоторый ключ из имеющихся в списке, такой, что, как мы надеемся, половина оставшихся ключей будет располагаться перед ним, а половина после. Этот выбранный ключ мы будем называть *опорным*. Далее мы сортируем два уменьшенных списка по отдельности, объединяем эти два подсписка, и весь список будет упорядочен.

опорный ключ

### 8.6.2. Пример

Перед тем, как приступить к реализации наших методов в виде детализированных процедур, давайте рассмотрим конкретный пример. Возьмем такой ряд из семи чисел:

26 33 35 29 19 12 22

## 1. Пример сортировки слиянием

Первым шагом при сортировке слиянием будет разбиение списка на два. Если (как в нашем примере) список имеет нечетную длину, примем соглашение, по которому левый подсписок будет всегда больше правого на один элемент. Итак, мы разделяем список на два

26 33 35 29 и 19 12 22

и сначала рассматриваем левый подсписок. Его опять можно разбить на две половины

26 33 и 35 29

Для каждого из этих подсписков мы опять применяем тот же метод, разбивая каждый список на подписки по одному элементу. Такие подписки длиной по одному элементу, разумеется, не нуждаются в сортировке. Теперь мы можем начать объединять подписки с целью образования упорядоченного списка. Подписки 26 и 33 объединяются с образованием упорядоченного списка 26 33, а подписки 35 и 29 при слиянии образуют 29 35. На следующем шаге мы объединяем эти два упорядоченные подписка длиной два элемента с получением упорядоченного подписка длиной четыре элемента:

26 29 33 35

Теперь, когда левая половина исходного списка упорядочена, мы выполняем те же шаги с правой половиной. Сначала мы разбиваем ее на подписки

19 12 и 22

Первый из этих подсписков разбивается на два подписка длиной один элемент, которые при слиянии дают 12 19. Второй подсписок, 22, имеет длину один элемент и не нуждается в упорядочении. При его слиянии с подписанием 12 19 мы получаем упорядоченный список

12 19 22

Наконец, упорядоченные подписки длиной четыре и три элемента объединяются с образованием

12 19 22 26 33 35

Способ объединения всех этих подсписков посредством рекурсивных вызовов показан в виде дерева рекурсий для метода слияния на рис. 8.9. Порядок, в котором выполняются рекурсивные вызовы, показан на рисунке стрелками. Содержимое каждого подписка, передаваемое рекурсивному вызову, показано слева от узлов дерева, а то же содержимое, упорядоченное после слияния, показано справа от узлов. Вызовы, для которых не требуются дальнейшие рекурсии (подписки длиной 1 элемент) являются листьями дерева и обозначены квадратиками.

## 2. Пример использования метода быстрой сортировки

Давайте еще раз пройдемся по тому же примеру, используя теперь метод быстрой сортировки и ведя тщательный учет выполняемых шагов согласно приведенному выше эскизу процедуры сортировки. Для использования метода быстрой сортировки мы прежде всего должны решить, ка-

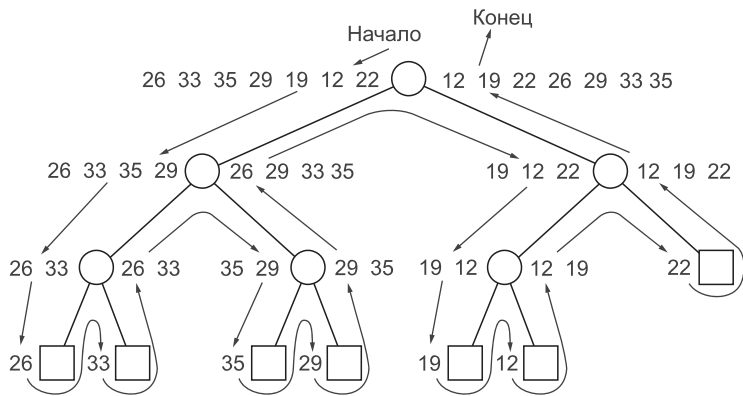


Рис. 8.9. Дерево рекурсий для сортировки слиянием списка из 7 чисел

разбиение

кой ключ выбрать в качестве опорного, чтобы относительно него разбить список на две половины. Мы можем выбрать любое число, но ради единообразия будем придерживаться определенного правила. Возможно, простейшим правилом будет выбор в качестве опорного первого числа в списке, и именно так мы и поступим в этом примере. В практических применениях, однако, другие способы выбора опорного ключа обычно дают лучшие результаты.

Итак, нашим первым опорным ключом будет 26, и список разбивается на два подсписка

19 12 22 и 33 35 29

состоящих, соответственно, из чисел, меньших опорного ключа и больших, чем опорный ключ. Мы оставили порядок элементов в подсписках неизменным по отношению к исходному списку, однако это решение тоже было произвольным. Некоторые варианты алгоритма быстрого упорядочения помещают опорный ключ в один из подсписков, но мы предпочли оставить его вне обоих подсписков.

Теперь мы подходим к следующей строке эскиза процедуры сортировки, которая требует от нас сортировки первого подсписка. Мы снова начинаем выполнение алгоритма с самого верха, но на этот раз прикладывая его к укороченному списку

19 12 22

нижняя половина

Опорным ключом этого списка оказывается число 19, которое разбивает свой список на два подсписка с одним элементом в каждом, 12 в первом и 22 во втором. Поскольку в обоих этих подсписках содержатся лишь по одному элементу, они не нуждаются в сортировке, поэтому мы приходим к последней строке эскиза, где два списка объединяются с помещением между ними опорного ключа, в результате чего образуется упорядоченный список

12 19 22

внутренние  
и внешние  
вызовы процедур

На этом вызов процедуры сортировки для этого подсписка завершается, и происходит возврат в точку вызова. Вызов этот был выполнен изнутри процедуры сортировки для полного списка из семи чисел, поэтому мы переходим к следующей строке этой процедуры.

Мы теперь использовали процедуру дважды, причем второй экземпляр процедуры выполнялся внутри первого экземпляра. Обратите особое внимание на то, что эти два экземпляра процедуры обрабатывают разные списки, при этом они отличаются друг от друга так же, как отличаются действия по выполнению одного и того же кода дважды в цикле. Наглядность этого процесса повысится, если мы будем думать, что эти экземпляры имеют разный цвет, и тогда команды во втором (внутреннем) вызове можно было бы написать полностью на месте вызова, но другого цвета, отчетливо разделяя их тем самым как отдельные экземпляры процедуры. Шаги этого процесса показаны на рис. 8.10.

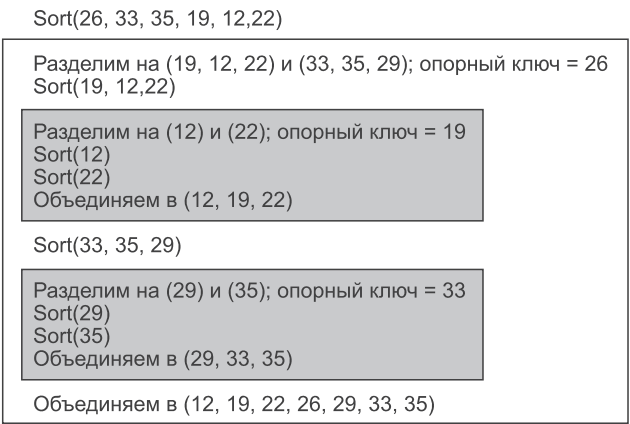


Рис. 8.10. Трассировка выполнения алгоритма быстрой сортировки

Возвращаясь к нашему примеру, мы видим, что следующая строка первого экземпляра процедуры является вторым вызовом процедуры Sort упорядочения второго списка, на этот раз с тремя числами

33 35 29

верхняя половина

Как и в предыдущем (внутреннем) вызове, опорный ключ 33 тут же подразделяет список, образуя подсписки длиной 1 элемент, которые объединяются с образованием упорядоченного списка

29 33 35

Наконец, происходит возврат из этого вызова процедуры Sort, и мы достигаем последней строки (внешнего) экземпляра процедуры, который упорядочивает весь список. К этому моменту два упорядоченных подсписка длиной три элемента объединяются с опорным ключом 26 с образованием упорядоченного списка

12 19 22 26 29 33 35

слияние

После выполнения этого шага сортировка завершилась.

Простейший способ проследить все вызовы в нашем примере быстрой сортировки заключается в изображении его дерева рекурсии, как это показано на рис. 8.11. Два вызова процедуры Sort на каждом уровне показаны в виде дочерних вершин одного родительского узла. Подписки размера 1 или 0, которые не нуждаются в сортировке, изображены в виде листьев. Для других вершин (чтобы избежать загромождения диаграммы) мы указываем только опорный ключ, используемый для вызова. Однако не составляет труда прочитать все числа в каждом подписке (хотя не обязательно в их исходном порядке). Числа в подписке при каждом рекурсивном вызове представляют собой числа у соответствующей вершины, а также у всех вершин, являющихся потомками данной.

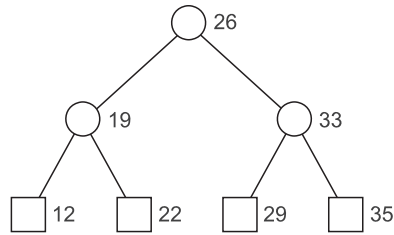


Рис. 8.11. Дерево рекурсии для быстрой сортировки списка из 7 чисел

пример

Если вы еще не вполне разобрались в выполнении рекурсии, будет очень полезно просмотреть сортировку списка из 14 имен, с которым мы уже работали в предыдущем разделе, используя оба способа: сортировку слиянием и быструю сортировку. Для проверки ваших рассуждений на рис. 8.12 приведено дерево вызовов для быстрой сортировки в той же сокращенной форме, что и в предыдущем примере. Это дерево приведено в двух вариантах, одном, в котором опорным ключом является первый ключ в каждом подписке, и втором, где в качестве опорного ключа выбран центральный ключ (или слева от центра в случае списка с четным числом элементов).

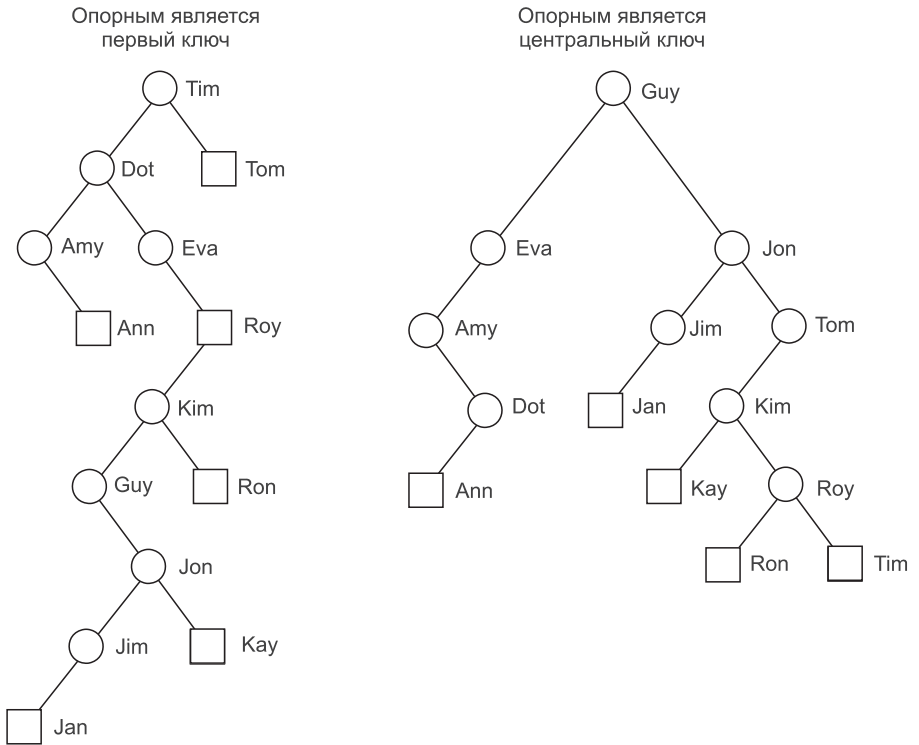
### Упражнения 8.6

- E1.** Примените быструю сортировку к списку из семи чисел, рассмотренному в настоящем разделе, выбрав в качестве опорного ключа (а) последнее число в подписке и (b) центральное (или слева от центра) число в подписке. Для каждого случая нарисуйте дерево рекурсивных вызовов.
- E2.** Примените сортировку слиянием к списку из 14 имен, использованному при рассмотрении предыдущих методов упорядочения:

Tim Dot Eva Roy Tom Kim Guy Ami Jon Ann Jim Kay Ron Jan

- E3.** Примените быструю сортировку к списку из 14 имен, упорядочив их вручную по алфавиту. В качестве опорного ключа выберите (а) первый ключ в каждом подписке и (b) центральный (или слева от центра) ключ в каждом подписке. См. рис. 8.12.





**Рис. 8.12.** Дерево рекурсии для быстрой сортировки списка из 14 имен

**Е4.** В обоих методах разбиения мы старались разделить список на два подсписка приблизительно одинаковой длины, однако базовый принцип сортировки методом разбиения остается применимым и при делении исходного списка на неравные половины. Рассмотрите случай разделения списка таким образом, что один подсписок будет иметь длину 1 элемент. Такое разбиение приводит к двум методам, в зависимости от того, выполняется ли сортировка путем отделения одного элемента от списка или путем смешивания подсписков.

- (a) Разделите список, найдя в нем элемент с наибольшим значением ключа, и составив из него подсписок с длиной 1 элемент. После сортировки оставшихся элементов полученные подсписки легко объединяются вместе путем помещения наибольшего ключа на последнее место списка.
- (b) Отделите последний элемент от списка. После сортировки оставшихся элементов включите этот элемент в список.

Покажите, что один из этих методов является в точности методом сортировки включением, а другой совпадает с сортировкой методом выбора.

```
procedure Divide (var L, secondhalf: list);           { Разделение }
{ Pre:    Связный список L уже создан.
Post:    Список L был уменьшен до его первой половины, а вторая
            половина элементов из L помещена в связный список secondhalf.
            Если в списке L было нечетное число элементов, тогда его
            первая половина будет на один элемент больше второй
            половины. }
```

**var**

current,	{ для просмотра всего списка L }
midpoint: listpointer;	{ перемещается на половинной скорости, чтобы найти среднюю точку }

```

begin
    midpoint: L.head;
    if midpoint = nil then
        { процедура Divide }
        { начнем с установки midpoint в позицию 1 }
        { если L пуста, тогда пуста }
        { и его вторая половина secondhalf }
        secondhalf.head := nil
    else begin
        current := midpoint↑.nextnode;
        { начнем с установки current }
        { во вторую позицию }
        while current <> nil do begin
            { перемещаем current дважды на одно }
            { перемещение midpoint }
            current := current↑.nextnode;
            if current <> nil then begin
                midpoint := midpoint↑.nextnode;
                current := current↑.nextnode;
            end
        end;
        secondhalf.head := midpoint↑.nextnode;
        { разорвем список после midpoint↑ }
        midpoint↑.nextnode := nil;
        { создадим конец у первой половины }
        { случай непустого списка }
    end;
end;
{ процедура Divide }

```

Вторая процедура, Merge(first, second, out), объединяет списки first и second, возвращая список-результат слияния через третий параметр out. Действие процедуры Merge проиллюстрировано на рис. 8.13.

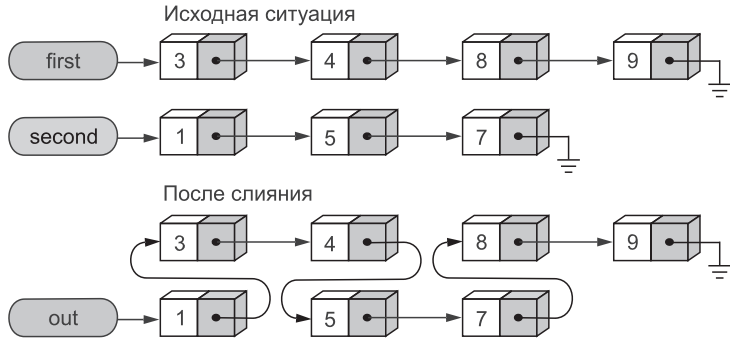


Рис. 8.13. Слияние двух упорядоченных связанных списков процедурой Merge

объединение  
(слияние) двух  
упорядоченных  
связных списков

```

procedure Merge (var L: List);
{ Слияние }
{ Pre: first и second являются упорядоченными связными списками }
{ и уже созданы. }
{ Post: out является упорядоченным связным списком, содержащим }
{ все элементы, которые были в first и second. Исходные списки }
{ first и second уничтожены. }
var
    p1, p2,
    lastsorted: listpointer;
    { используются для просмотра первого и второго списков }
    { всегда указывает на последний узел }
    { упорядоченного списка }

```

```

begin                                                                 { процедура Merge }
  if first.head = nil then
    out := second;
  else if second.head = nil then
    out := first;
  else begin { сначала найдем головной элемент объединенного списка }
    p1 := first.head;
    p2 := second.head;
    if p1↑.entry.key <= p2↑.entry.key then begin
      out := first;
      p1 := p1↑.nextnode
    end
    else begin
      out := second;
      p2 := p2↑.nextnode
    end;
    lastsorted := out.head;    { lastsorted указывает на последний элемент
                                объединенного списка }

    while (p1 <> nil) and (p2 <> nil) do
      if p1↑.entry.key <= p2↑.entry.key then
        begin                  { присоединим узел с меньшим ключом
                                к упорядоченному списку }

          lastsorted↑.nextnode := p1;
          lastsorted := p1;
          p1 := p1↑.nextnode    { передвинемся к следующему узлу,
                                еще не участвовавшему в операции слияния }

        end
        else begin
          lastsorted↑.nextnode := p2;
          lastsorted := p2;
          p2 := p2↑.nextnode
        end
      end
      if p1 = nil then          { после конца одного списка
                                присоединим остаток другого }
        lastsorted↑.nextnode := p2;
      else
        lastsorted↑.nextnode := p1;
      end
    end
  end;
end;                                                                 { процедура Merge }

```

## 8.7.2. Анализ метода сортировки слиянием

Теперь, имея рабочую процедуру для сортировки слиянием, наступило время приостановиться и проанализировать ее поведение, чтобы мы могли сравнить этот метод с другими методами сортировки. Как и в случае других алгоритмов для связанных списков, нам не нужно заботиться о времени, расходуемом на перемещение элементов. Нам больше будет интересно число сравнений ключей, выполняемое процедурой.

### 1. Подсчет числа операций сравнения

В законченной процедуре сортировки слиянием сравнение ключей выполняется лишь в одном месте. Это место находится внутри главного цикла процедуры слияния. После каждого сравнения один из двух узлов

посылается в выходной список. Отсюда число сравнений безусловно не может превысить числа смешиваемых элементов. Чтобы найти полную длину этих списков, давайте снова рассмотрим дерево рекурсий для этого алгоритма, которое для простоты мы привели на рис. 8.14 для случая, когда  $n = 2^m$ , т. е. составляет целую степень двух.

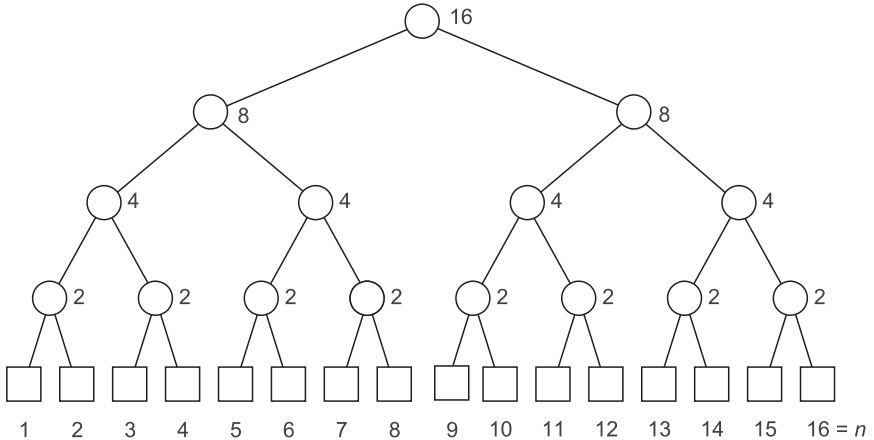


Рис. 8.14. Длины смешиваемых подсписков

Рассмотрев дерево, изображенное на рис. 8.14, легко увидеть, что полная длина списков на каждом уровне в точности равна  $n$ , полному числу элементов. Другими словами, каждый элемент объединяется на каждом уровне в точности один раз. Отсюда полное число сравнений на каждом уровне не может превысить  $n$ . Число уровней, исключая листья (для которых слияние не выполняется) составляет величину  $\lg n$ , округленную вверх до ближайшего минимального целого числа, т. е. равно «потолку»  $\lceil \lg n \rceil$ . Число сравнений ключей, выполняемых в методе слияния при сортировке списка из  $n$  элементов, следовательно, не превышает  $\lceil n \lg n \rceil$ .

## 2. Сопоставление с сортировкой включением

Вспомним (раздел 8.2.4), что при сортировке включением списка из  $n$  элементов выполняется в среднем не более  $3n^2$  операций сравнения. Как только  $n$  становится больше 16,  $\lg n$  становится меньше  $3n$ . В случае, когда  $n$  принимает практически значимый размер,  $\lg n$  становится значительно меньше  $3n$ , и поэтому число сравнений, выполняемых в методе слияния, значительно меньше, чем в методе включения. Когда, например,  $n = 1024$ ,  $\lg n = 10$ , и граница числа сравнений для метода слияния составляет 10240, в то время как среднее число для сортировки включением будет более 250000. Задача, которая при использовании метода включения потребует полчаса компьютерного времени, при решении ее методом слияния займет едва ли минуту.

$n \lg n$

Появление в приведенных выше оценках выражения  $n \lg n$  ни в коей мере не является случайным, а, наоборот, тесно связано с нижней гра-

ницей, установленной в разделе 8.5, где было доказано, что любой метод сортировки, использующий сравнения ключей, должен выполнить по меньшей мере

$$\lg n! \approx n \lg n - 1.44n + O(\lg n)$$

сравнений ключей. Когда  $n$  велико, первый член этого выражения становится более важным, чем все остальные. Таким образом, в методе слияния мы нашли алгоритм, близко подходящий к нижней границе.

### 3. Уточнение числа операций сравнения

Подойдя к делу более тщательно, мы можем получить более точную оценку числа сравнений, выполняемых при сортировке слиянием, которая покажет, что фактические характеристики этого метода даже еще ближе к наименьшему числу сравнений ключей, допускаемому нижней границей.

Прежде всего заметим, что слияние двух списков полного размера  $k$  элементов никогда не требует  $k$  сравнений, а только максимум  $k - 1$ , поскольку после того, как второй за наибольшим ключом ключ был выведен в выходной список, в списке уже нет ничего, с чем можно было бы сравнивать наибольший ключ, и он выводится в выходной список без всяких сравнений. Отсюда для каждого выполняемого слияния мы должны уменьшить полное число сравнений на 1. Полное число слияний равно

$$\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots + 1 = n - 1.$$

(Эта формула точна, если  $n$  есть целая степень 2, и является хорошим приближением в остальных случаях.) Полное число сравнений ключей, выполняемых при сортировке слиянием, следовательно, меньше чем

$$n \lg n - n + 1.$$

Далее, вполне может оказаться, что обработка одного из двух смешиваемых списков будет закончена раньше, чем другого, и тогда все элементы второго списка должны передаваться на выход без дальнейших сравнений, так что число сравнений может быть значительно меньше вычисленного нами. Например, каждый элемент одного списка может предшествовать каждому элементу второго списка, и тогда все элементы второго списка попадут в выходной список без всяких сравнений. В упражнениях выполняется доказательство того, что полное число может быть уменьшено в среднем до величины

$$n \lg n - 1.1583n + 1,$$

и правильный коэффициент для  $n$  может оказаться близок к  $-1.25$ . Таким образом, мы видим, что не только ведущий член настолько мал, насколько позволяет нижняя граница, но и второй член также близок к этому. Путем дальнейшего улучшения процедуры слияния метод может быть оптимизирован настолько, что число сравнений будет отличаться всего на несколько процентов от теоретически оптимального числа сравнений (см. ссылки).

## 4. Выводы

преимущества  
методов слияния  
двух разных  
списков

Из наших рассуждений может показаться, что сортировка слиянием является наилучшим методом сортировки, и, действительно, для связанных списков со случайным порядком ключей трудно придумать лучший метод. Однако мы не должны забывать, что необходимо принимать в расчет и другие соображения помимо сравнения ключей. Написанная нами программа тратит значительное время на нахождение центра списка, чтобы мы могли разделить его пополам. В упражнениях обсуждается один из методов экономии этого времени. Связный вариант сортировки слиянием эффективно использует память. Он не нуждается в больших вспомогательных массивах или других списках, и поскольку глубина рекурсии составляет всего  $\lg n$ , пространство памяти, требуемое для проследивания рекурсивных вызовов, оказывается весьма малым.

## 5. Сортировка слиянием для непрерывных списков

три критерия  
анализа процедур  
слияния

К сожалению, для непрерывных списков сортировка слиянием не является столь безоговорочно лучшим методом. Проблема заключается в том, что два непрерывных списка трудно слить без существенных затрат (1) памяти, (2) компьютерного времени и (3) программистских усилий. Первый и наиболее прямолинейный способ слияния двух непрерывных списков заключается в использовании вспомогательного массива, достаточно длинного, чтобы вместить в себя объединенный список и позволить копировать в массив элементы по мере слияния списков. Этот метод требует дополнительного места, объем которого пропорционален  $n$ . Во втором методе мы можем расположить смешиваемые подсписки рядом друг с другом, забыть о степени их упорядоченности и использовать метод вроде сортировки включением для упорядочения объединенного списка. Такой подход почти не использует дополнительной памяти, но компьютерное время возрастает пропорционально  $n^2$  по сравнению с прямой пропорциональностью от  $n$  для хорошего алгоритма слияния. Наконец (см. ссылки), были изобретены алгоритмы, смешивающие два непрерывных списка за время, пропорциональное  $n$ , используя при этом только небольшой фиксированный объем дополнительной памяти. Эти алгоритмы, однако, весьма сложны.

## Упражнения 8.7

- E1.** В статье, опубликованной в профессиональном журнале, говорится: «Этот рекурсивный процесс [сортировки слиянием] занимает время  $O(n \lg n)$  и поэтому выполняется в 64 раза быстрее предыдущего метода [сортировки включением] при упорядочении 256 чисел». Выступите критиком этого утверждения.
- E2.** Оценка числа сравнений ключей при слиянии обычно оказывается завышенной, поскольку она не принимает во внимание тот факт, что обработка одного из списков может быть закончена ранее другого. Например, могло случиться, что все элементы первого списка располагаются перед любыми элементами второго списка, и тогда число сравнений будет равно лишь длине первого списка. Для данного уп-

ражнения предположите, что все числа в обоих списках различаются и что все возможные расположения чисел равновероятны.

- (a) Покажите, что среднее число сравнений, выполняемых нашим алгоритмом слияния двух упорядоченных списков длиной 2 элемента есть  $8/3$ . [Подсказка: начните с упорядоченного списка 1, 2, 3, 4. Напишите шесть способов, которыми можно расположить эти числа в двух упорядоченных списках длиной 2 элемента и покажите, что четыре из этих способов дадут по 3 сравнения, а два используют по 2 сравнения.]
- (b) Покажите, что среднее число сравнений при слиянии двух упорядоченных списков длиной 3 элемента составляет 4.5.
- (c) Покажите, что среднее число сравнений при слиянии двух упорядоченных списков длиной 4 элемента составляет 6.4.
- (d) Используйте полученные ранее результаты для уточненной оценки полного количества сравнений при сортировке слиянием.
- (e) Покажите, что по мере того, как  $m$  стремится к бесконечности, среднее число сравнений, выполняемых при слиянии двух упорядоченных списков длиной  $m$  элементов, приближается к  $2m - 2$ .

слияние  
с использованием  
фиксированного  
объема памяти и  
пропорциональ-  
ного времени  
выполнения

**Е3.** [Очень сложная задача] Прямолинейный метод слияния двух непрерывных списков путем построения смешанного списка в отдельном массиве использует дополнительное пространство памяти, пропорциональное числу элементов в двух списках, но может быть реализован весьма эффективно, так что время выполнения всего лишь пропорционально числу элементов. Попробуйте разработать метод слияния для непрерывных списков, который потребует минимально возможного дополнительного пространства, но при этом все еще будет выполняться за время, линейно пропорциональное числу элементов в списках. [Имеется решение, использующее только небольшой постоянный объем дополнительной памяти. См. ссылки.]

## Программные проекты 8.7

- P1.** Реализуйте метод сортировки слиянием на вашем компьютере. Используйте те же соглашения и те же тестовые данные, что и для реализации и тестирования связанного варианта сортировки включением. Сравните производительность алгоритмов сортировки слиянием и включением для коротких и длинных списков, а также для списков, находящихся в почти упорядоченном состоянии и в случайном порядке.
- P2.** Наша программа сортировки слиянием для связанных списков тратит значительное время на поиск центра в каждом подсписке, чтобы можно было разделить этот подсписок на два. Реализуйте следующую модификацию, которая сэкономит значительную часть этого времени. Прежде всего образуйте запись для описания связанного списка, которая будет содержать не только (a) указатель на головной элемент списка, но также и (b) указатель на центральный элемент списка и (c) длину списка. В начале исходный список следует просмотреть однажды с целью определения этой информации. При ее наличии становится отно-



сительно просто разделить список на две половины и получить длины подсписков. Центр подсписка можно найти просмотром лишь половины подсписка. Перепишите процедуру сортировки слиянием, чтобы в нее предавались записи, описывающие связанные списки, в виде параметров вызова, и затем используйте их для упрощения разделения образуемых списков.

- P3.** Наша процедура сортировки слиянием не принимает во внимание тот факт, что исходный список может находиться в частично упорядоченном состоянии. В методе *естественной сортировки слиянием* список делится на подспiski в конце возрастающей последовательности ключей вместо произвольного выбора точки на полпути. Это упражнение требует реализации двух вариантов естественной сортировки слиянием.

В первом варианте исходный список просматривается только однажды, и используются только два подсписка. До тех пор, пока порядок ключей правилен, узлы помещаются в первый подсписк. Как только будет найден неупорядоченный ключ, первый подсписк закрывается и начинается второй подсписк. Когда будет найден второй неупорядоченный ключ, второй подсписк закрывается, и второй подсписк сливается с первым. Затем второй подсписк повторно создается заново и сливается с первым. Когда достигается конец исходного списка, сортировка заканчивается. Эта первый вариант легко запрограммировать, но по мере его выполнения первый подсписк становится значительно длиннее второго, и производительность процедуры будет ухудшаться, приближаясь к характеристикам сортировки включением.

Второй вариант обеспечивает приблизительное равенство длин смешиваемых подсписков и, следовательно, преимущества метода разбиения реализуются в полной мере. В этом методе используется (небольшой) вспомогательный массив, содержащий **(a)** длины и **(b)** указатели на головные элементы упорядоченных, но еще не смешанных подсписков. Элементы этого массива должны сохраняться в порядке согласно длинам подсписков. По мере того как каждый (естественным образом упорядоченный) подсписк отделяется от исходного списка, он помещается во вспомогательный массив. Если в массиве имеется другой список, длина которого лежит между половиной и удвоенной длиной нового списка, тогда оба списка сливаются, и процесс повторяется. Когда исходный список исчерпывается, все остающиеся в массиве подспiski сливаются (сначала с меньшими длинами) и сортировка заканчивается.

Нет ничего священного в степени 2 в качестве критерия слияния подсписков. Выбор этого значения просто обеспечивает, что количество элементов во вспомогательном массиве не будет превышать  $\lg n$  (докажите это!). Меньшее отношение (оно должно быть больше 1) приведет к увеличению длины вспомогательной таблицы, а большее отношение уменьшит преимущества метода разделения. Поэкспериментируйте с тестовыми данными, чтобы найти оптимальное отношение.

естественная  
сортировка  
слиянием

один  
упорядоченный  
список

**Р4.** Разработайте вариант алгоритма сортировки слиянием для непрерывных списков. Сложность здесь будет в написании процедуры для слияния двух упорядоченных списков в непрерывной памяти. Простейшее решение заключается в использовании двух массивов, каждый из которых должен быть достаточно велик, чтобы содержать все элементы двух исходных списков. Два упорядоченных подсписка занимают различные части одного и того же массива. По мере их слияния новый список строится во втором массиве. После завершения процедуры слияния новый список может быть при желании скопирован назад в первый массив. В противном случае на следующем шаге назначение двух массивов может быть сделано взаимно обратным.

## 8.8. Метод быстрой сортировки для непрерывных списков

Обратимся теперь к методу быстрой сортировки, в котором список сначала разделяется на нижний и верхний подсписки, для которых ключи, соответственно, меньше некоторого опорного ключа и больше этого ключа. Метод быстрой сортировки связных списков может быть разработан без особого труда, и этот проект мы оставим для упражнений. Наиболее важным применением быстрой сортировки, однако, являются непрерывные списки, где этот метод оказывается весьма быстрым, и где он имеет преимущества перед сортировкой слиянием, поскольку не требует выбора между использованием значительного объема дополнительной памяти для вспомогательного массива или серьезных затрат программистского труда для реализации сложного и запутанного алгоритма слияния.

### 8.8.1 Главная процедура

Наша задача разработки быстрой сортировки для непрерывных списков состоит, в сущности, в написании алгоритма для разделения элементов списка посредством опорного ключа и обмена элементов внутри списка так, чтобы сначала располагались все элементы с ключами, меньшими опорного, затем шел опорный ключ, а затем элементы с большими ключами. Мы будем считать, что переменная *pivotposition* предоставляет позицию опорного ключа в разделенном списке.

Поскольку разделенные подсписки содержатся в одном массиве в правильно упорядоченных относительно друг друга позициях, завершающий шаг комбинирования упорядоченных подсписков совершенно излишен и опущен в нашей процедуре.

Для рекурсивного приложения процедуры упорядочения к под спискам нижняя (*low*) и верхняя (*high*) границы должны выступать в качестве параметров процедуры. Другие наши процедуры сортировки, однако, имели в качестве параметра один лишь список *L*, поэтому ради единообразия обозначений мы выполним рекурсию в процедуре *RecQuickSort*, которая вызывается следующей процедурой:

главная процедура  
QuickSort

```
procedure QuickSort (var L: list); { Быстрая сортировка }
{ Pre: Непрерывный список L уже создан. Каждый элемент списка L
      содержит ключ.
  Post: Элементы списка L упорядочены таким образом, что ключи
        всех элементов располагаются в неубывающем порядке.
  Uses: Использует процедуру RecQuickSort. }
begin
  RecQuickSort(L, 1, L.count)
end;
```

Собственно процедура быстрой сортировки выглядит тогда так:

```
procedure RecQuickSort (var L: list; low, high: position);
{ Рекурсивная быстрая сортировка }
{ Pre: Непрерывный список L уже создан. Каждый элемент списка L
      содержит ключ. Параметры low и high являются допустимыми
      позициями в списке L.
  Post: Элементы списка L упорядочены таким образом, что ключи
        всех элементов располагаются в неубывающем порядке.
  Uses: Использует процедуру RecQuickSort рекурсивно и процедуру
        Partition. }
var pivotposition: position; { позиция опорного ключа
                              после разделения списка }
begin
  if low < high then begin { процедура RecQuickSort }
                              { если нет, тогда упорядочение
                              не требуется }
    Partition(L, low, high, pivotposition);
    RecQuickSort(L, low, pivotposition - 1);
    RecQuickSort(L, pivotposition + 1, high);
  end
end; { процедура RecQuickSort }
```

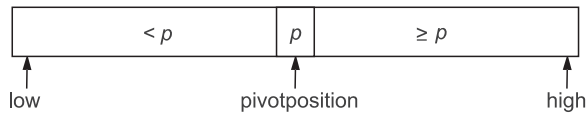
### 8.8.2. Разделение списка

Теперь мы можем сконструировать процедуру Partition. Для этого можно использовать различные методы (один из которых предлагается в качестве упражнения), иногда более быстрые, чем разрабатываемый здесь алгоритм, но более запутанные и трудные в отладке. Алгоритм, на котором мы остановились, значительно проще и доступнее для понимания, и, в то же время, отнюдь не медленный; фактически он выполняет минимально возможное число сравнений ключей для любого алгоритма разделения.

#### 1. Разработка алгоритма

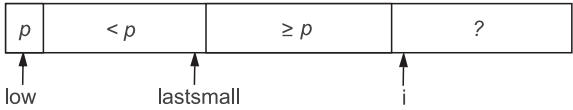
Если дано значение опорного ключа  $p$ , мы должны переставить элементы списка и вычислить  $\text{pivotposition}$  так, чтобы все элементы слева от этой позиции имели ключи меньше  $p$ , а все элементы справа — ключи больше  $p$ . Допуская возможность иметь более одного элемента с ключом, равным  $p$ , мы принимаем, что элементы слева от  $\text{pivotposition}$  имеют ключи строго меньше  $p$ , а элементы справа имеют ключи, большие или равные  $p$ , как это показано на следующем рисунке:

цель  
(постусловие)



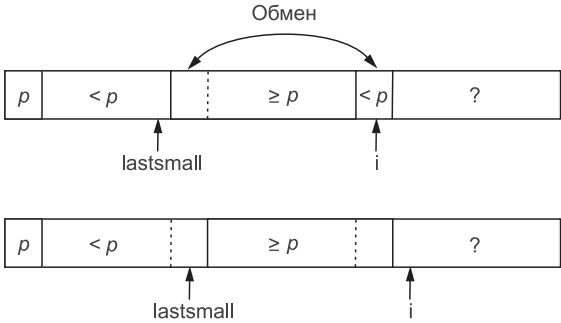
инвариант цикла

Для переупорядочения элементов таким образом мы должны сравнить каждый ключ списка с опорным ключом. Для этого мы воспользуемся циклом **for** (с переменной цикла  $i$ ). Мы также используем вторую переменную  $lastsmall$  так, чтобы все элементы в позиции  $lastsmall$  и перед ней имели ключи, меньшие  $p$ . Предположим, что начальное положение опорного ключа  $p$  соответствует первой позиции списка и временно оставим его в этом месте. Тогда в середине цикла список будет характеризоваться таким состоянием:

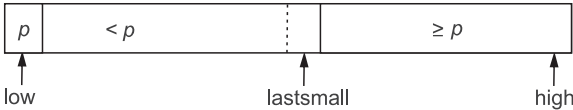


восстановление инварианта

Когда процедура анализирует элемент  $i$ , могут встретиться два случая. Если элемент больше или равен  $p$ , тогда  $i$  можно увеличить, при этом список все еще будет иметь требуемые свойства. Если элемент меньше  $p$ , тогда мы восстанавливаем структуру путем увеличения  $lastsmall$  и обменивая этот элемент (первый из тех, которые по меньшей мере равны  $p$ ) с элементом  $i$ , как это показано на следующем рисунке.



Когда цикл завершается, мы имеем такую ситуацию:



окончательное состояние

и тогда для получения требуемого состояния списка нам только нужно поменять опорный ключ в позиции  $low$  с элементом в позиции  $lastsmall$ .

## 2. Выбор опорного ключа

Мы не ограничены выбором в качестве опорного ключа первого элемента списка; мы можем выбрать любой элемент и обменять его с первым элементом перед началом цикла, который разделяет список. Фактически выбор первого элемента в качестве опорного ключа часто оказывается неудачным, поскольку, если список уже упорядочен, первый ключ не будет иметь меньших, чем он сам, ключей, и поэтому один из подсписков будет пуст. Давайте лучше выберем опорный ключ в районе центра списка, в надежде, что такой выбор разделит ключи приблизительно поровну с каждой стороны опорного ключа.

опорный ключ от центра

### 3. Кодирование

Приняв эти решения, мы получаем следующую процедуру, в которой мы используем процедуру `swar` из раздела 8.3.2. Для удобства ссылок мы также включаем в процедуру состояние, которое должно поддерживаться в течение выполнения цикла в качестве инварианта цикла.

```

procedure Partition (var L: list; low, high: position; var pivotposition: position);
                                {Разделение}
{ Pre: Непрерывный список L уже создан. Параметры low и high
  являются допустимыми позициями в списке L.
  Post: Центральный элемент списка L (или элемент слева от центра)
  выбран в качестве опорного элемента и перемещен в позицию
  pivotposition. Все элементы списка L упорядочены так, что
  элементы с ключами меньшими, чем опорный ключ,
  располагаются перед pivotposition, а все оставшиеся элементы
  располагаются после pivotposition.
  Uses: Использует процедуру Swap(i, j, L), которая обменивает
  местами элементы с индексами i и j в непрерывном списке L. }

var
  pivot: keytype;
  i,                                { используется для просмотра списка }
  lastsmall: position; { позиция последнего ключа, меньшего, чем опорный }
begin                                { процедура Partition }
  Swap(low, (low+high) div 2, L); { обмен первого и среднего элементов }
  pivot := L.entry [low].key;      { первый элемент теперь стал
                                опорным ключом }

  lastsmall := low;
  for i:= low + 1 to high do
  { В начале каждого прохода этого цикла мы имеем следующие условия:
    if low < j <= lastsmall then L.entry [j].key < pivot.
    if lastsmall < j < i then L.entry [j].key >= pivot }

    if L.entry [i].key < pivot then begin
      lastsmall := lastsmall + 1;
      Swap(lastsmall, i, L) { переместим больший элемент направо,
                            а меньший налево }

    end;
    Swap(low, lastsmall, L); { поместим опорный ключ
                            в правильную позицию }

  pivotposition := lastsmall
end;                                { процедура Partition }

```

### 8.8.3. Анализ метода быстрой сортировки

Теперь наступило время тщательно проанализировать алгоритм быстрой сортировки, чтобы решить, когда он работает хорошо, а когда не очень, и каков объем выполняемых им вычислений.

#### 1. Выбор опорного ключа

Наш выбор в качестве опорного ключа одного из центральных ключей списка вполне произволен. Такой выбор может оказаться успешным в том смысле, что список разделится точно на две половины, или нам может не повезти, и один подсписок может оказаться значительно длиннее

худший случай

другого. Некоторые другие методы выбора опорного ключа будут рассмотрены в упражнениях. Крайний случай при нашем выборе возникает для приведенного ниже списка, в котором, какой бы опорный ключ мы не выбрали, он окажется наибольшим ключом в своем подсписке:

2 4 6 7 3 1 5

Проверьте это, используя процедуру Partition из текста. Если быстрая сортировка приложена к этому списку, название метода окажется совершенно неуместным, поскольку при первой рекурсии непустой список будет иметь длину 6, при следующей 5 и т. д.

Если мы будем выбирать в качестве опорного ключа первый или последний ключ в каждом подсписке, тогда крайний случай будет возникать, если ключи находятся в естественном порядке или в порядке, обратном естественному. Такие состояния списка более вероятны, чем некоторый случайный порядок, и поэтому выбор первого или последнего ключа в качестве опорного может породить проблемы.

## 2. Подсчет числа сравнений и обменов

Давайте определим число сравнений и обменов в алгоритме быстрой сортировки для непрерывного списка. Пусть  $C(n)$  есть число сравнений ключей, выполняемых алгоритмом быстрой сортировки в случае списка длины  $n$ , и пусть  $S(n)$  есть число обменов элементов. Мы имеем  $C(1) = S(1) = 0$ . Процедура разделения сравнивает опорный ключ с каждым другим ключом в списке точно один раз, и, следовательно, процедура Partition дает нам точно  $n - 1$  сравнений. Если один из двух создаваемых ею подсписков имеет длину  $r$ , тогда другой подсписок будет иметь длину в точности равную  $n - r - 1$ . Число сравнений, выполненных в двух рекурсивных вызовах, будет тогда  $C(r)$  и  $C(n - r - 1)$ . В результате мы имеем

полное число  
сравнений

$$C(n) = n - 1 + C(r) + C(n - r - 1).$$

Для решения этого уравнения нам нужно только знать  $r$ . Фактически наша запись несколько обманчива, поскольку значение  $C(\ )$  зависит не только от длины списка, но и от точного порядка элементов в нем. Поэтому в различных случаях, в зависимости от начального порядка, мы получим разные ответы.

## 3. Число сравнений, наихудший случай

Прежде всего рассмотрим худший в плане сравнений случай. Мы уже видели, что такая ситуация возникает, когда опорный ключ вообще не разделяет список, так что один подсписок содержит  $n - 1$  элементов, а другой подсписок вообще пуст. В этом случае, поскольку  $C(0) = 0$ , мы получаем  $C(n) = n - 1 + C(n - 1)$ . Выражение такого рода носит название **рекуррентного отношения**, потому что оно выражает ответ через предыдущий результат такой же формулы. Нам необходимо разрешить эту рекуррентность, т. е. найти такое выражение для  $C(n)$ , чтобы на другой его стороне не присутствовало  $C(\ )$ . Для решения рекуррентных отношений существуют различные (иногда довольно сложные) методы, но в на-

рекуррентное  
отношение

шем случае мы можем без труда получить ответ, начав вычисления не сверху, а снизу:

$$\begin{aligned}
 C(1) &= 0 \\
 C(2) &= 1 + C(1) &= 1 \\
 C(3) &= 2 + C(2) &= 2 + 1 \\
 C(4) &= 3 + C(3) &= 3 + 2 + 1 \\
 &\dots \\
 C(n) &= n - 1 + C(n - 1) &= (n - 1) + (n - 2) + \dots + 2 + 1 \\
 & &= \frac{1}{2}(n - 1)n = \frac{1}{2}n^2 - \frac{1}{2}n.
 \end{aligned}$$

В этих вычислениях мы использовали теорему А.1 для получения суммы чисел от 1 до  $n - 1$ .

Вспомним, что при сортировке выбором выполняется приблизительно  $\frac{1}{2}n^2 - \frac{1}{2}n$  сравнений ключей, и слишком большое число сравнений было как раз слабым местом сортировки выбором (по сравнению с сортировкой включением). Отсюда в своем худшем случае быстрая сортировка работает так же плохо, как и сортировка выбором.

сортировка  
выбором

#### 4. Число обменов, наихудший случай

Далее давайте определим, сколько обменов элементов будет выполняться при быстрой сортировке, опять же в худшем случае. Процедура разделения выполняет один обмен в своем цикле для каждого ключа, меньшего, чем опорный ключ, и два обмена вне своего цикла. В худшем случае опорный ключ является наибольшим ключом в списке, и процедура сортировки выполнит  $n + 1$  обменов. Если  $S(n)$  есть полное число обменов в списке длиной  $n$ , то мы имеем для худшего случая рекуррентное отношение

$$S(n) = n + 1 + S(n - 1).$$

Процедура разделения вызывается только при  $n \geq 2$ , и в худшем случае  $S(2) = 3$ . Отсюда, как и при подсчете сравнений, мы можем разрешить рекуррентность, последовательно выполняя вычисления, и в результате для худшего случая мы имеем

$$S(n) = n + 1 + n + \dots + 3 = \frac{1}{2}(n + 1)(n + 2) - 3 = 0.5n^2 + 1.5n - 1$$

#### 5. Сравнение с методами сортировки включением и выбором

В худшем случае при сортировке включением для последовательного списка должно быть выполнено приблизительно вдвое больше сравнений и присваиваний элементов, чем в среднем случае, что дает в сумме для каждой операции  $0.5n^2 + O(n)$ . Каждый обмен при быстрой сортировке требует трех операций присваивания элементов, поэтому быстрая сортировка в своем худшем случае выполняет  $1.5n^2 + O(n)$  присваиваний, т. е. при больших  $n$  приблизительно в три раза больше, чем сортировка включением. Отсюда в своем худшем случае быстрая (так называемая!) сортировка оказывается хуже сортировки включением в худшем случае, и в отношении сравнения ключей такой же плохой, как сортировка выбором в ее худшем случае. Фактически, если рассматривать худший случай, быстрая сортировка никуда не годится, и ее название является всего лишь ложной рекламой.

незавидное  
поведение  
в худшем случае



превосходное  
поведение  
в худшем случае

Видимо, есть какая-то иная причина того, что быстрая сортировка не была давным-давно выброшена на свалку негодных программ. Действительно, причина этого заключается в том, что *среднее* поведение быстрой сортировки применительно к спискам со случайным порядком элементов оказывается одним из лучших среди всех известных на сегодня методов сортировки (использующих сравнение ключей и обрабатывающих непрерывные списки).

### 8.8.4. Анализ метода быстрой сортировки для среднего случая

Чтобы выполнить анализ среднего случая, мы предположим, что все возможные порядки элементов списка равновероятны, и для простоты будем считать, что ключи представляют собой просто целые числа от 1 до  $n$ .

#### 1. Подсчет числа обменов

Когда мы выбираем опорный ключ в процедуре Partition, он с равной вероятностью может оказаться любым из ключей. Обозначим через  $p$  ключ, выбранный в качестве опорного. Тогда после разделения ключ  $p$  будет гарантированно находиться в позиции  $p$ , поскольку все ключи в диапазоне 1, ...,  $p - 1$  будут находиться левее его, а все ключи в диапазоне  $p + 1$ , ...,  $n$  — правее.

Число обменов, которые будут выполнены в одном вызове процедуры Partition, равно  $p + 1$ , куда входит один обмен в каждом шаге цикла для каждого из  $p - 1$  ключей, меньших  $p$ , и плюс еще два обмена вне цикла. Обозначим через  $S(n)$  среднее число обменов, выполняемых при быстрой сортировке в списке длиной  $n$ , и через  $S(n, p)$  среднее число обменов в списке длиной  $n$ , если опорный ключ для первой половины равен  $p$ . Нам известно, что для  $n \geq 2$

$$S(n, p) = (p + 1) + S(p - 1) + S(n - p).$$

Теперь мы должны получить среднее для этих выражений; поскольку  $p$  случайно, мы складываем их от  $p = 1$  до  $p = n$  и делим затем на  $n$ . В этих вычислениях используется формула для суммы целых чисел (теорема A.1); результат выглядит так:

$$S(n) = \frac{n}{2} + \frac{3}{2} + \frac{2}{n} (S(0) + S(1) + \dots + S(n - 1)).$$

#### 2. Решение рекуррентного отношения

В качестве первого шага к решению этого рекуррентного отношения заметим, что при сортировке списка длиной  $n - 1$  мы получим то же самое выражение, где  $n$  заменяется на  $n - 1$ , при условии, что  $n \geq 2$ :

$$S(n - 1) = \frac{n - 1}{2} + \frac{3}{2} + \frac{2}{n - 1} (S(0) + S(1) + \dots + S(n - 2)).$$



Умножая первое выражение на  $n$ , второе на  $n - 1$ , и вычитая, получим

$$nS(n) - (n - 1)S(n - 1) = n + 1 + 2S(n - 1)$$

или

$$\frac{S(n)}{n + 1} = \frac{S(n - 1)}{n} + \frac{1}{n}.$$

Мы можем решить это рекуррентное отношение как и раньше, начав снизу. Результат будет таким:

$$\frac{S(n)}{n + 1} = \frac{S(2)}{3} + \frac{1}{3} + \dots + \frac{1}{n}.$$

Сумма обратных величин для целых чисел рассмотрена в разделе А.2.8, где показано, что

$$1 + \frac{1}{2} + \dots + \frac{1}{n} = \ln n + O(1).$$

Отличие этой суммы от той, которая нам нужна, ограничено константой, поэтому можно написать

$$S(n)/(n + 1) = \ln n + O(1)$$

или, в итоге,

$$S(n) = n \ln n + O(n).$$

Для сравнения этого результата с характеристиками других методов упорядочения мы замечаем, что

$$\ln n = (\ln 2) (\lg n)$$

а  $\ln 2 \approx 0.69$ , откуда

$$S(n) \approx 0.69 (n \lg n) + O(n)$$

### 3. Подсчет числа сравнений

Поскольку вызов процедуры разделения для списка длиной  $n$  дает в точности  $n - 1$  сравнений, рекуррентное отношение для числа сравнений, выполняемых в среднем случае, будет отличаться от такого же отношения для числа обменов только в одном: вместо  $p + 1$  обменов в процедуре разделения мы имеем  $n - 1$  сравнений. Отсюда первое рекуррентное отношение для числа  $C(n, p)$  сравнений в списке длиной  $n$  с опорным ключом  $p$  будет таким:

$$C(n, p) = n - 1 + C(p - 1) + C(n - p)$$

Усреднив эти выражения от  $p = 1$  до  $p = n$ , получим

$$C(n) = n + \frac{2}{n}(C(0) + C(1) + \dots + C(n - 1))$$

Поскольку это рекуррентное отношение для числа  $C(n)$  сравнений ключей отличается от  $S(n)$  только множителем 2 в последнем, те же шаги, использованные для решения  $S(n)$ , дадут

$$C(n) \approx 2n \lg n + O(n) \approx 1,39n \lg n + O(n)$$

### 8.8.5. Сравнение с методом слияния

сравнение ключей Только что выполненные вычисления показали, что в среднем при быстрой сортировке выполняется приблизительно на 39 процентов больше сравнений ключей, чем при сортировке слиянием. Причина, разумеется, заключается в том, что алгоритм сортировки слиянием специально разработан так, чтобы список делился на половины существенно одинакового размера, в то время как размеры подсписков при быстрой сортировке заранее непредсказуемы. Отсюда возникает вероятность существенного ухудшения производительности быстрой сортировки, однако на практике такая ситуация мало вероятна, потому усреднение времени выполнения для плохих случаев и времени выполнения для хороших дает только что полученный результат.

перемещение данных Рассматривая перемещение данных, мы не получили детальную информацию относительно сортировки слиянием, поскольку нас в основном интересовал вариант для связанных списков. Если, однако, мы рассмотрим вариант сортировки слиянием для непрерывных списков, который формирует смешанные подписки во втором массиве и при каждом проходе изменяет на обратное назначение массивов, сразу станет ясно, что на каждом уровне дерева рекурсии все  $n$  элементов должны копироваться из одного массива во второй. Число уровней в дереве рекурсии равно  $\lg n$ , и отсюда следует, что число присваиваний элементов при сортировке слиянием для непрерывных списков составляет  $n \lg n$ . С другой стороны, для быстрой сортировки мы получаем среднее число обменов около  $0,69n \lg n$ . Хорошая (написанная на машинном языке) реализация выполнит обмен элементов за две операции присваивания. Опять получается, что при быстрой сортировке выполняется приблизительно на 39 процентов больше операций присваивания, чем при сортировке слиянием. В упражнениях, однако, описывается другая процедура разделения, которая в среднем выполняет только около одной трети обменов по сравнению с процедурой, разработанной нами в настоящем разделе. При таком улучшении быстрая сортировка непрерывных списков может дать сокращение числа операций присваивания более чем в два раза по сравнению с сортировкой слиянием.

оптимизация

#### Упражнения 8.8.

- E1.** Как будет выполняться процедура быстрой сортировки (описанная в тексте), если все ключи в списке совпадают?
- E2.** [Из Knuth] (см. ссылку в главе 3). Опишите алгоритм, который упорядочит непрерывный список с ключами, представляющими собой действительные числа, так, чтобы все элементы с отрицательными ключами выстроились перед элементами с неотрицательными ключами.

Полученный список не требуется окончательно упорядочивать. Постарайтесь, чтобы ваш алгоритм выполнял минимум перемещений элементов и минимум сравнений. Не используйте вспомогательный массив.

- Е3.** [Из Hoare] Предположим, что вместо упорядочения мы хотим лишь найти  $m$ -й наименьший ключ в данном списке длиной  $n$ . Покажите, как можно применить быструю сортировку к решению этой задачи, чтобы существенно ограничить объем выполняемой работы по сравнению с полной сортировкой.
- Е4.** Пусть дан список целых чисел. Разработайте процедуру, схожую с процедурой разделения, которая изменит порядок целых чисел так, что все целые числа в четных позициях будут четными или все целые числа в нечетных позициях будут нечетными. (Ваша процедура предоставит доказательство того, что одна или другая из поставленных целей всегда может быть достигнута, хотя, возможно, обеих целей одновременно достигнуть нельзя.)
- Е5.** Еще один метод выбора опорного ключа для алгоритма быстрой сортировки заключается в получении среднего из первого, последнего и центрального ключей в списке. Опишите изменения, которые следует внести в процедуру QuickSort для реализации такого подхода. Сколько дополнительных вычислений будет выполнено в этом случае? Для  $n = 7$  найдите порядок ключей

1, 2, ..., 7

который будет соответствовать худшему случаю для этого алгоритма. Насколько этот худший случай лучше худшего случая для исходного алгоритма?

- Е6.** Иной подход к выбору опорного ключа заключается в получении среднего значения всех ключей в списке. Результирующий алгоритм носит название *сортировки по среднему значению*.

**(а)** Напишите процедуру, реализующую сортировку по среднему значению. Процедуру разделения придется изменить, поскольку среднее из ключей не обязательно образует ключ, присутствующий в списке. В первом проходе опорный ключ можно выбрать любым удобным для вас способом. Далее по мере разделения ключей можно накапливать и сохранять текущие суммы и количество ключей для обоих подсписков, и тогда средние значения (которые будут являться новыми опорными ключами) подсписков можно вычислить, не выполняя дополнительных проходов по всему списку.

**(б)** В методе сортировки по среднему значению относительные значения ключей определяют, насколько одинаковыми будут подсписки после разделения; начальный порядок ключей не имеет значения, за исключением подсчета числа выполняемых обменов. Насколько плох может быть худший случай в плане относительных размеров двух подсписков? Найдите набор из  $n$  целых чисел, который приведет к худшему случаю для сортировки по среднему значению.

**Е7.** [Требуется знание элементарной теории вероятности] Удачным способом выбора опорного ключа является использование генератора случайных чисел, с помощью которого можно выбирать позицию следующего опорного ключа в каждом вызове `RecQuickSort`. Учитывая независимость этих операций выбора, найдите вероятность того, что алгоритм быстрой сортировки столкнется со своим худшим случаем. **(а)** Решите эту задачу для  $n = 7$ . **(б)** Решите эту задачу для произвольного значения  $n$ .

**Е8.** Ценой некоторого увеличения числа сравнений ключей можно переписать процедуру разделения так, чтобы число обменов уменьшилось приблизительно в 3 раза, в среднем от  $\frac{1}{2}n$  до  $\frac{1}{6}n$ . Идея метода заключается в использовании двух индексов, перемещающихся от концов списка к середине и в выполнении обмена лишь в тех случаях, когда больший ключ найден в младшей позиции, а меньший ключ — в старшей. Это упражнение описывает разработку такой процедуры.

**(а)** Введите два индекса  $i$  и  $j$  и следите за выполнением инвариантного условия, что все ключи перед позицией  $i$  меньше опорного ключа, а все ключи после позиции  $j$  больше или равны опорному ключу. Для простоты переместите (обменом) опорный ключ в первую позицию и начните разделение со второго элемента. Напишите цикл, который будет увеличивать позицию  $i$ , пока инвариантное условие выполняется, и другой цикл, который будет уменьшать позицию  $j$ , пока инвариантное условие выполняется. Ваши циклы должны также осуществлять проверку того, не вышли ли значения индексов за допустимые пределы, например, проверяя условие  $i \leq j$ . Когда обнаруживается пара элементов, каждый из которых находится не на той стороне, для них следует выполнить обмен, после чего циклы повторяются. Каково будет условие завершения внешнего из этих циклов? В конце процедуры опорный ключ следует поместить (опять обменом) на свое правильное место.

**(б)** Используя инвариантное условие, удостоверьтесь, что ваша процедура работает правильно.

**(с)** Покажите, что каждый выполняемый в цикле обмен помещает два элемента в их окончательные позиции. После этого покажите, что процедура в худшем случае для списка длиной  $n$  выполняет максимум  $\frac{1}{2}n + O(1)$  обменов.

**(д)** Если после разделения опорный ключ находится в позиции  $p$ , тогда число обменов, выполняемых процедурой, приблизительно равно числу элементов, первоначально находившихся в одной из  $p$  позиций до (включительно) опорного ключа, но значения ключей которых больше или равны опорному ключу. Если ключи распределены случайным образом, тогда вероятность того, что конкретный ключ будет больше или равен опорному ключу, составляет  $\frac{1}{n}(n - p + 1)$ . Покажите, что среднее число таких ключей и, соответственно, среднее число обменов равно приблизительно  $\frac{p}{n}(n - p)$ . Взяв среднее из этих значений от  $p = 1$  до  $p = n$ , покажите, что число обменов приблизительно равно  $\frac{n}{6} + O(1)$ .

- (е) Необходимость проверки того, что индексы  $i$  и  $j$  находятся в допустимых границах подписка, можно устранить, если использовать опорный ключ в качестве сигнальной метки, останавливающей цикл. Реализуйте этот метод в своей процедуре. Тщательно проверьте, правильно ли работает ваша процедура во всех возможных случаях.
- (ф) [Из Wirth] Рассмотрите следующий простой и “очевидный” способ реализации циклов с использованием опорного ключа в качестве сигнальной метки:

```
repeat
  repeat i := i + 1 until L.entry [i].key >= pivot;
  repeat j := j + 1 until L.entry [i].key <= pivot;
  Swap(i, j, L)
until i >= j;
```

Найдите список ключей, для которых этот вариант будет работать неверно.

### Программные проекты 8.8

- P1.** Реализуйте алгоритм быстрой сортировки (для непрерывных списков) на своем компьютере и протестируйте его с помощью программы из проекта P1 раздела 8.2. Сравните его производительность с производительностью всех остальных изученных к настоящему моменту методов сортировки.

- P2.** Напишите вариант алгоритма быстрой сортировки для связанных списков, интегрируйте его в связный вариант тестовой программы из проекта P1 раздела 8.2 и сравните его производительность с производительностью других методов сортировки для связанных списков.

быстрая  
сортировка для  
связных списков

Используйте первый элемент подписка в качестве опорного ключа для разделения. Процедура разделения для связанных списков оказывается несколько проще, чем для непрерывных списков, поскольку в ней не приходится минимизировать перемещения данных. Для разделения подписка вам надо только просматривать его, удаляя каждый элемент по ходу движения, и затем добавлять элемент к одному или другому из двух списков в зависимости от значения его ключа относительно значения опорного ключа.

Поскольку процедура Partition теперь образует два новых списка, вам придется создать небольшую дополнительную процедуру для объединения упорядоченных подписков в единый связный список.

- P3.** Ввиду больших издержек, метод быстрой сортировки может оказаться хуже более простых методов при обработке коротких списков. Посредством экспериментирования найдите значение длины, при которой для списков со случайным порядком метод быстрой сортировки в среднем становится более эффективным, чем сортировка включением. Напишите смешанную процедуру упорядочения, которая начинает выполнять быструю сортировку, но когда подписки становятся достаточно короткими, переходит на сортировку включением. Определите, лучше ли осуществлять такой переход внутри рекурсивной процеду-

ры, или лучше завершить рекурсивные вызовы, когда подписки станут достаточно короткими, чтобы сменить метод, и затем в самом конце процесса прогнать весь список один раз через алгоритм сортировки включением.

## 8.9. Пирамиды и пирамидальная сортировка

Метод быстрой сортировки имеет тот недостаток, что хотя обычно его производительность превосходна, некоторые последовательности входных данных могут существенно ухудшить его характеристики. В этом разделе мы изучим еще один метод сортировки, который свободен от указанного недостатка. Этот алгоритм, называемый **пирамидальной сортировкой**, сортирует непрерывный список длиной  $n$ , выполняя при этом даже в своем худшем случае  $O(n \log n)$  сравнений и перемещений элементов. В результате его граница для худшего случая оказываются меньше, чем в методе быстрой сортировки, а для непрерывных списков меньше, чем для сортировки слиянием, поскольку для его реализации, помимо упорядочиваемого списка, требуется лишь небольшая и постоянная по объему дополнительная память.

корпоративная  
иерархия

Пирамидальная сортировка основана на древовидной структуре, которая отражает типичную иерархию, складывающуюся в корпорациях. Представьте себе организационную структуру корпоративного управления в виде дерева, наверху которого находится президент. Если президент уходит на пенсию, вице-президенты начинают конкурировать за занятие места наверху; один из них получает повышение и занимает вакансию. Младшие руководители всегда конкурируют между собой в борьбе за повышение, когда наверху возникает вакансия. Теперь представьте себе, что корпорация непрерывно «сокращается» (чего, разумеется, никогда не бывает в реальной жизни), отправляя на пенсию своего наиболее затратного служащего, президента. В результате наверху постоянно возникает вакансия, служащие постоянно конкурируют между собой в борьбе за повышение, и как только каждый служащий достигает вершины пирамиды, эта позиция опять освобождается. Такая процедура описывает в общих чертах идею нашего метода сортировки.

### 8.9.1. Двухвариантные деревья как списки

Давайте начнем с полного 2-дерева вроде того, что изображено на рис. 8.15, и перенумеруем его вершины, начав с корня, слева направо на каждом уровне.

Мы теперь можем поместить это 2-дерево в список, сохраняя каждый узел в позиции, соответствующей его номеру на рисунке. Мы замечаем, что:

*Левый и правый дочерние узлы для узла с позицией  $k$  займут позиции в списке  $2k$  и  $2k + 1$ , соответственно. Если эти позиции находятся за пределами списка, тогда эти дочерние узлы не существуют.*

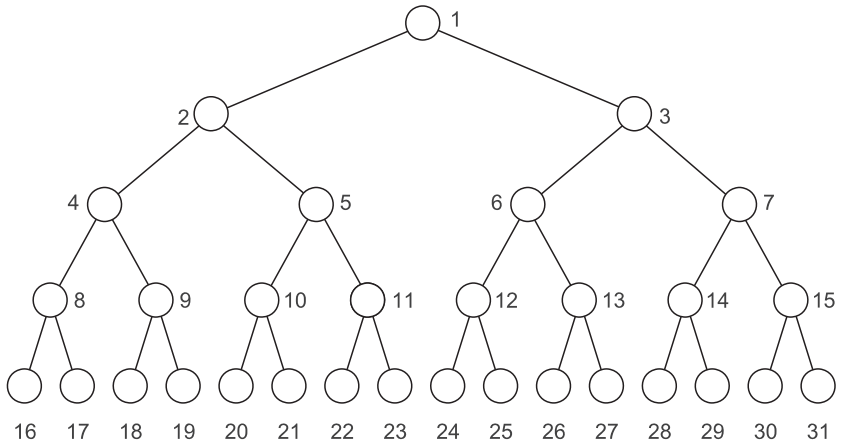


Рис. 8.15. Полное 2-дерево с 31 вершиной

Имея в виду эту идею, мы можем определить, что же мы понимаем под пирамидой.

**Определение**

***Пирамида** есть список, в котором каждый элемент имеет ключ, и для всех позиций  $k$  в списке значение ключа в позиции  $k$  по меньшей мере столь же велико, как у ключей в позициях  $2k$  и  $2k + 1$ , при условии, что эти позиции существуют в списке.*

Определенная таким образом пирамида аналогична корпоративной иерархии, в которой каждый служащий (за исключением тех, что находятся на самом нижнем уровне пирамиды) наблюдает за работой двух других служащих.

Поясняя детали использования пирамид, мы будем рисовать деревья вроде того, что изображено на рис. 8.16, чтобы показать иерархические отношения, однако алгоритмы обслуживания пирамид всегда рассматривают их как особого рода списки.

Заметьте, что пирамида безусловно *не* является упорядоченным списком. Первым элементом фактически должен быть самый большой ключ в пирамиде, в то время как в упорядоченном списке первым ключом является ключ с самым маленьким значением. В пирамиде нет обязательной упорядоченности ключей в позициях  $k$  и  $k + 1$ , если  $k > 1$ .

**Замечание**

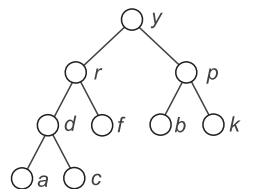


Рис. 8.16. Пирамида как дерево и как список







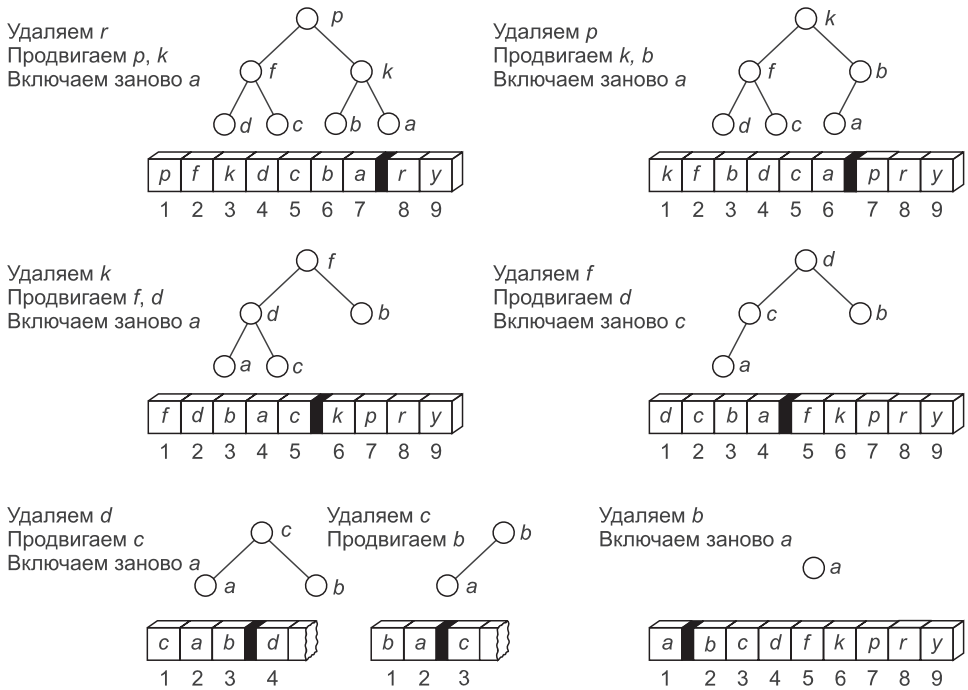


Рис. 8.18. Трассировка процедуры Heapsort

**Post:** Элемент *current* включен в *L*, и элементы *L* переупорядочены так, что элементы между индексами *low* и *high*, включительно, образуют пирамиду. }

```

var large: position { позиция дочернего от L.entry элемента
                    с большим ключом }
begin
  large := 2 * low; { large теперь левый дочерний элемент от low }
  while large <= high do begin
    if large < high then
      if L.entry[large].key < L.entry[large + 1].key then
        large := large + 1; { large теперь дочерний от low элемент
                             с большим ключом }
    if current.key >= L.entry[large].key then { current принадлежит
                                              позиции low }
      large := high + 1 { установим large как флаг для завершения цикла }
    else begin { продвинем L.entry[large] и переместимся
                ниже по дереву }
      L.entry[low] := L.entry[large];
      low := large;
      large := 2 * low
    end
  end
  L.entry[low] := current
end;

```

{ процедура InsertHeap }

## 5. Построение исходной пирамиды

инициализация

Нам осталось определить, как построить начальную пирамиду из списка, имеющего произвольный порядок. Прежде всего заметим, что 2-дерево с единственным узлом автоматически удовлетворяет условию пирамиды, и нам, следовательно, не надо заботиться о каких-либо листьях дерева, другими словами, о каких-либо элементах второй половины списка. Если мы начинаем с середины списка и продвигаемся назад к началу, мы можем использовать процедуру InsertHeap для включения каждого элемента в частичную пирамиду, состоящую из всех более поздних элементов, и тем самым построить полную пирамиду. Требуемая процедура оказывается, таким образом, совсем простой:

```

procedure BuildHeap (var L: list); { Построить пирамиду }
{ Pre: Непрерывный список L уже создан. Все элементы L содержат
  ключи.
  Post: Элементы L переупорядочены так, L становится пирамидой.
  Uses: Использует процедуру InsertHeap. }
var low: position { все элементы ниже позиции low образуют пирамиду }
begin { процедура BuildHeap }
  for low := (L.count div 2) downto 1 do
    InsertHeap(L.entry [low], low, L.count, L)
end; { процедура BuildHeap }

```

### 8.9.3 Анализ пирамидальной сортировки

Из рассмотренного выше примера трудно сделать заключение об эффективности пирамидальной сортировки, и, действительно, этот метод вряд ли стоит использовать для коротких списков. Кажется весьма странным, что мы сначала должны переместить большие ключи к началу списка с тем, чтобы в конце концов перенести их в конец. Однако, когда  $n$  велико, эти индивидуальные странности становятся несущественными, и пирамидальная сортировка доказывает свою значимость как один из очень немногих алгоритмов сортировки непрерывных списков, который гарантированно завершается за время  $O(n \log n)$  с минимальными требованиями к памяти.

включение  
в худшем случае

Прежде всего давайте решим, какой объем работы выполняет InsertHeap в худшем случае. При каждом проходе цикла позиция low по меньшей мере удваивается; отсюда число проходов не может превысить величины  $\lg(\text{high} \text{ div } \text{low})$ , которая одновременно является высотой поддерева с корнем в L.entry[low]. В каждом проходе цикла выполняется два сравнения ключей (обычно) и одно присваивание элементов. Отсюда число сравнений, выполняемых в InsertHeap, не превышает величины  $2\lg(\text{high} \text{ div } \text{low})$ , а число присваиваний — величины  $\lg(\text{high} \text{ div } \text{low})$ .

первый этап

Пусть  $m = \lfloor \frac{1}{2}n \rfloor$  (т. е. равно максимальному целому числу, не превышающему  $\frac{1}{2}n$ ). В процедуре BuildHeap мы выполняем  $m$  вызовов InsertHeap, для значений  $k = \text{low}$  в диапазоне от  $m$  и вниз до 1. Отсюда полное число сравнений приблизительно равно

$$2 \sum_{k=1}^m \lg \left( \frac{n}{k} \right) = 2(m \lg n - \lg m!) \approx 5m \approx 2.5n$$

поскольку, согласно аппроксимации Стирлинга (теорема А.5) и соотношения  $\lg m = \lg n - 1$ , мы имеем

$$\lg m! \approx m \lg m - 1.5m \approx m \lg n - 2.5m.$$

второй этап

Аналогично на этапе сортировки и включения мы имеем

$$2 \sum_{k=2}^n \lg k = 2 \lg n! \approx 2n \lg n - 3n$$

полное число  
сравнений для  
худшего случая

сравнений. Этот результат доминирует над результатом первого этапа, и мы можем заключить, что число сравнений равно  $2n \lg n + O(n)$ .

В процедуре InsertHear для каждого двух сравнений выполняется (приблизительно) одно присваивание элемента. Поэтому полное число присваиваний составляет  $n \lg n + O(n)$ .

В итоге мы получаем:

*В своем худшем случае при сортировке списка длиной  $n$  пирамидальная сортировка выполняет  $2n \lg n + O(n)$  сравнений ключей и  $n \lg n + O(n)$  операций присваивания элементов.*

В разделе 8.8.4 было показано, что соответствующая величина для быстрой сортировки в среднем случае равна  $1.39 n \lg n + O(n)$  сравнений и  $0.69 n \lg n + O(n)$  обменов, причем число обменов можно уменьшить до величины  $0.23 n \lg n + O(n)$ . Отсюда следует, что в худшем случае пирамидальная сортировка работает несколько хуже, чем быстрая сортировка в среднем случае. Худший случай для быстрой сортировки, однако, дает величину  $O(n^2)$ , что гораздо хуже, чем худший случай пирамидальной сортировки при больших  $n$ . Анализ среднего случая для пирамидальной сортировки оказывается весьма сложным, но эмпирические исследования показывают, что (как и в случае сортировки выбором) имеется относительно небольшое различие между средним и худшим случаями, и пирамидальная сортировки обычно занимает приблизительно в два раза больше времени, чем быстрая.

сравнение  
с быстрой  
сортировкой

Таким образом, пирамидальную сортировку следует рассматривать как нечто, напоминающее стратегию страхования: в среднем пирамидальная сортировка приблизительно в два раза хуже быстрой, но зато она позволяет избежать незначительной вероятности катастрофической деградации производительности.

## 8.9.4. Очереди с приоритетами

Завершая этот раздел, мы кратко упомянем другое применение пирамид.

Определение

**Очередь с приоритетами** состоит из элементов, каждый из которых содержит ключ, называемый **приоритетом** элемента. Для такой очереди характерны только две операции помимо обычных создания, очистки, получения размера и определения полноты или пустоты:

- Включить элемент.
- Удалить элемент, имеющий самый большой (или самый маленький) ключ.

Если элементы имеют совпадающие ключи, тогда первым может быть удален любой элемент с максимальным ключом.

применения

В компьютерной системе с разделением времени, например, большое число задач могут ожидать освобождения времени центрального процессора. Некоторые из этих задач имеют более высокий приоритет, чем другие. Отсюда набор задач, ожидающих времени центрального процессора, образуют очередь с приоритетами. Другие применения очередей с приоритетами включают моделирование зависящих от времени событий (вроде модели аэропорта в разделе 5.4) и решение разреженных систем линейных уравнений путем построчного сжатия.

реализации

Мы можем представить очередь с приоритетами как упорядоченный последовательный список, и в этом случае удаление элемента осуществляется совсем просто, но включение займет время, пропорциональное  $n$ , числу элементов в очереди. Можно также представить очередь как неупорядоченный список; в таком случае включение элемента осуществляется быстро, а удаление медленно.

Теперь рассмотрим структуру пирамиды. Элемент с максимальным ключом находится наверху и может быть удален немедленно. Однако, восстановление структуры пирамиды для остальных ключей займет время  $O(\log n)$ . Если, однако, другой элемент включается тут же, тогда некоторая часть этого времени может быть скомбинирована с временем  $O(\log n)$ , необходимым для включения нового элемента. В результате представление очереди с приоритетами как пирамиды имеет преимущества при больших  $n$ , поскольку пирамида эффективно реализуется в непрерывной памяти и гарантированно имеет лишь логарифмическую зависимость времени от  $n$  и для включений, и для удалений.

### Упражнения 8.9

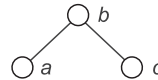
**E1.** Изобразите списки, соответствующие каждому из приведенных ниже деревьев, считая, что дочерние вершины элемента в позиции  $h$  находятся в позициях  $2k$  и  $2k+1$ . Какие из этих деревьев представляют собой пирамиды? Для деревьев, не являющихся пирамидами, укажите позицию в списке, в которой нарушается структура пирамиды.



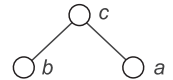
(a)



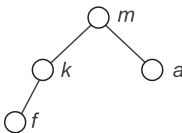
(b)



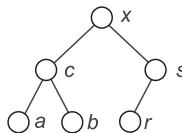
(c)



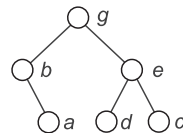
(d)



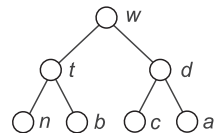
(e)



(f)



(g)



(h)

**E2.** Выполните ручную трассировку процедуры HeapSort для каждого из приведенных ниже списков. Нарисуйте исходное дерево, которому соответствует список, покажите, как его можно преобразовать в

пирамиду, и покажите результирующую пирамиду после того, как каждый элемент удален с вершины и новый элемент включен в дерево.

(a) Следующие три слова должны быть упорядочены по алфавиту:

triangle	square	pentagon
[треугольник	квадрат	пятиугольник]

(b) Три слова из пункта (a) должны быть упорядочены согласно числу сторон соответствующего многоугольника в возрастающем порядке.

(c) Три слова из пункта (a) должны быть упорядочены согласно числу сторон соответствующего многоугольника в убывающем порядке.

(d) Следующие семь чисел должны быть упорядочены по возрастанию значений:

26 33 35 29 19 22

(e) Те же семь чисел в другом начальном порядке также должны быть упорядочены по возрастанию значений:

12 19 33 26 29 35 22

(f) Следующий список из 14 имен должен быть упорядочен по алфавиту:

Tim Dot Eva Roy Tom Kim Guy Ami Jon Ann Jim Kay Ron Jan

**E3. (a)** Разработайте процедуру, которая будет включать новый элемент в пирамиду, образуя новую пирамиду. (Процедура `InsertHeap` в тексте требует, чтобы корень дерева был свободен, в то время как в этом упражнении корень уже содержит элемент с максимальным значением ключа, который должен оставаться в пирамиде. Ваша процедура увеличивает количество элементов в списке.)

(b) Проанализируйте время выполнения и требуемую память для вашей процедуры.

**E4. (a)** Разработайте процедуру, которая будет удалять элемент с максимальным значением ключа (корень) с верха пирамиды и восстанавливать структуру пирамиды для результирующего, меньшего по размеру списка.

(b) Проанализируйте время выполнения и требуемую память для вашей процедуры.

**E5. (a)** Разработайте процедуру, которая будет удалять из пирамиды элемент с индексом  $i$  и восстанавливать структуру пирамиды для результирующего, меньшего по размеру списка.

(b) Проанализируйте время выполнения и требуемую память для вашей процедуры.

**E6.** Рассмотрите пирамиду из  $n$  ключей, считая, что  $x_k$  есть ключ в позиции  $k$  (в непрерывной реализации) для  $1 \leq k \leq n$ . Докажите, что высота поддерева с корнем в  $x_k$  есть наибольшее целое число, не превышающее  $\lg(n/k)$  для всех  $k$ , удовлетворяющих условию  $1 \leq k \leq n$ .

[Подсказка: используйте «обратную» индукцию по  $k$ , начав с листьев и поднимаясь вверх к корню, т. е. к элементу  $x_1$ .]

- Е7.** Определите понятие *тройной пирамиды*, аналогичной обычной пирамиде за исключением того, что каждый узел дерева (кроме листьев) имеет три дочерних узла. Разработайте метод сортировки для тройных пирамид и проанализируйте свойства этого метода.

### Программный проект 8.9

- P1.** Реализуйте метод пирамидальной сортировки (для непрерывных списков) на своем компьютере, интегрируйте его в тестовые программы проекта P1 из раздела 8.2 и сравните его производительность с производительностью всех предыдущих алгоритмов сортировки.

## 8.10. Обзор: сравнение методов

В этой главе мы изучили и тщательно проанализировали значительное количество методов сортировки. Возможно, лучшим способом подведения итогов этой работы будет краткое рассмотрение по очереди каждого из трех важных критериев эффективности:

- использование пространства памяти;
- использование времени компьютера;
- труд программиста.

### 1. Использование пространства памяти

Большинство рассмотренных нами алгоритмов почти не используют пространство памяти за исключением того, которое занято исходным списком, который в процессе сортировки реорганизуется, но остается на своем месте. Исключениями являются быстрая сортировка и сортировка слиянием, где рекурсии требуют некоторого объема дополнительной памяти для хранения информации о подписках, которые еще не были упорядочены. Однако в тщательно написанной процедуре объем дополнительной памяти, используемой для рекурсии, составляет  $O(\log n)$  и, соответственно, ничтожен по сравнению с памятью, требуемой для других целей.

Наконец, следует помнить, что основной недостаток сортировки слиянием для непрерывных списков заключается в том, что прямолинейный вариант алгоритма требует дополнительной памяти, равной объему исходного списка.

Во многих приложениях упорядочиваемый список слишком велик, чтобы поместиться в быструю память компьютера, и в этих случаях следует использовать другие методы. Обычным подходом является деление списка на подписки, которые могут быть упорядочены в быстрой памяти компьютера, а затем слияние этих упорядоченных подписков осуществляется уже во внешней памяти. В силу этого было выполнено много исследований по разработке алгоритмов слияния, особенно для тех случаев, когда требуется объединять много подписков сразу. Мы не будем обсуждать здесь эту тему.



## 2. Компьютерное время

Вторым критерием эффективности является использование компьютерного времени, и мы тщательно анализировали этот критерий для каждого из разработанных нами методов. Вкратце можно сказать, что простые методы сортировки включением и выбором требуют времени, возрастающего при увеличении длины списка  $n$  как  $n^2$ ; сортировка Шелла работает значительно быстрее; остальные методы обычно работают еще быстрее, требуя времени порядка  $O(n \log n)$ . Однако быстрая сортировка в худшем случае тратит время, возрастающее как  $n^2$ . Пирамидальная сортировка напоминает стратегию страхования. Обычно она медленнее быстрой сортировки, но зато позволяет избежать незначительной вероятности катастрофической деградации производительности.

## 3. Труд программиста

Третий критерий эффективности часто является самым важным: эффективность и продуктивность использования времени программиста.

Если список невелик, использование сложных методов сортировки, разработанных ради экономии компьютерного времени, обычно дают отрицательный результат по сравнению с более простыми методами. Если программа будет запущена один или два раза, и вы не ограничены в машинном времени, будет довольно глупо расходовать дни и недели времени программиста на исследование сложных алгоритмов, которые в конечном счете сэкономят несколько секунд машинного времени.

При программировании на языках вроде большинства диалектов Fortran, Cobol или Basic, не поддерживающих рекурсий, реализация сортировки слиянием или быстрая сортировка становится весьма сложным делом, хотя и может быть достигнута путем использования стеков для хранения значений переменных, как мы уже отмечали в главе 3.

Сортировка Шелла не многим хуже сортировки слиянием или быстрой сортировки по производительности, не требует применения рекурсии и не представляет особого труда при программировании. Поэтому никогда не следует окончательно отказываться от сортировки Шелла.

Экономия времени программиста оказывается достаточным побуждением для использования простых алгоритмов, даже если они не очень эффективны, однако два предупреждения будут здесь не лишними. Во-первых, экономия времени на программирование никогда не должна служить оправданием написания неправильной программы, такой, что она, как правило, работает, но иногда проявляет странности поведения. Закон Мерфи неминуемо рано или поздно сработает. Во-вторых, простые программы, первоначально написанные, чтобы выполнить их всего несколько раз и затем отбросить, часто «пролезают» в приложения, о которых в момент написания этих программ совсем даже и не думали. Неаккуратность и невнимательность на ранних стадиях работы в дальнейшем часто оборачиваются существенными потерями.

Для многих приложений лучшим алгоритмом следует признать сортировку включением. Этот алгоритм нетрудно реализовать и поддержи-



вать, и он вполне эффективен для небольших списков. Даже со списками большой длины, если они уже более или менее упорядочены, сортировка включением может оказаться весьма эффективной. Если список полностью упорядочен, сортировка включением лишь проверяет его состояние максимально быстро.

#### 4. Статистический анализ

Окончательный выбор алгоритма зависит не только от длины списка, размера записей и их представления в памяти, но и также в очень большой степени от того, в каком порядке ожидается исходное расположение записей перед сортировкой. Анализ алгоритмов с точки зрения вероятности и статистики имеет огромное значение. Для большинства алгоритмов мы смогли получить результаты поведения в среднем, но обсуждение быстрой сортировки показало, что разброс производительности в зависимости от исходного порядка элементов в списке является также существенным фактором.

Статистической мерой изменчивости является *стандартное отклонение*. Быстрая сортировка является превосходным средством по средней производительности, и стандартное отклонение невелико, что означает незначительный разброс в производительности при использовании в разных условиях. Для таких алгоритмов, как сортировка выбором или слиянием и для пирамидальной сортировки, лучший случай отличается от худшего незначительно, т. е. стандартное отклонение для этих методов почти равно 0. Другие алгоритмы, вроде сортировки включением, имеют значительный разброс в производительности. Поэтому конкретное распределение начального порядка упорядочиваемых элементов имеет существенное значение при выборе метода сортировки. Для принятия взвешенного решения профессиональный компьютерный исследователь должен быть знаком с важными аспектами математической статистики в их приложении к анализу алгоритмов.

#### 5. Практическое тестирование

Наконец, принимая все эти решения, мы должны удачно сочетать теоретический анализ с практическим тестированием. При этом различные компьютеры и компиляторы дадут различающиеся результаты. Поэтому экспериментальные исследования того, как различные алгоритмы ведут себя в различных условиях, могут оказаться весьма поучительными.

#### Упражнения 8.10

- E1.** Проведите классификацию рассмотренных нами методов сортировки, отнеся их в одну из следующих категорий: (а) метод не требует доступа к элементам в одном конце списка, пока не будут упорядочены элементы с другого конца списка; (б) метод не требует доступа к уже упорядоченным элементам; (с) метод требует доступа к всем элементам списка на протяжении всего процесса сортировки.
- E2.** Некоторые из рассмотренных нами методов сортировки не годятся для их использования со связными списками. Какие и почему?

поведение  
в среднем

стандартное  
отклонение

- Е3.** Проранжируйте рассмотренные нами методы сортировки (и для связанных, и для непрерывных списков) согласно объему дополнительной памяти, которая требуется для хранения индексов или указателей, выполнения рекурсии и хранения копий упорядоченных элементов.
- Е4.** Какие из рассмотренных нами методов сортировки следует выбрать для каждого из описанных ниже приложений? Почему? Если представление списка в непрерывной или связанной памяти имеет значение для вашего выбора, укажите, почему.
- (a) Вы хотите написать программу сортировки общего назначения, которая будет использоваться многими людьми и в разных ситуациях.
  - (b) Вы хотите упорядочить однажды 1000 чисел. После завершения этой работы вы выбросите программу сортировки.
  - (c) Вы хотите упорядочить однажды 50 чисел. После завершения этой работы вы выбросите программу сортировки.
  - (d) Вы хотите упорядочить 5 элементов в середине длинной программы. Процедура сортировки будет вызываться по ходу выполнения длинной программы сотни раз.
  - (e) У вас есть список из 1000 ключей, хранящийся в оперативной памяти компьютера, и сравнение ключей выполняется быстро, но каждый раз, когда перемещается ключ, вместе с ним перемещается файл на диске, имеющий размер 500 блоков; перемещение файла занимает значительное время.
  - (f) Имеется полка длиной 4 метра, заполненная книгами по вычислительной технике, снабженными номерами и хранящимися в порядке этих номеров. Некоторые из этих книг были поставлены читателями на неправильные места, но ошибка в месте нахождения книги редко превышает 30 см.
  - (g) У вас имеется стопка из 500 библиографических карточек, разложенных в случайном порядке, и вам надо упорядочить их по алфавиту.
  - (h) У вас имеется список из 5000 слов, упорядоченных по алфавиту, но вы хотите удостовериться в этом и упорядочить те слова, которые могут находиться на неправильных местах.
- Е5.** Обсудите преимущества и недостатки разработки процедуры сортировки общего назначения, объединяющей в себе алгоритмы быстрой сортировки и сортировки Шелла. По какому критерию вы будете осуществлять переключение с одного алгоритма на другой? Для какого рода списков следует выбрать один или другой из этих алгоритмов?
- Е6.** Подытожьте результаты тестовых прогонов на своем компьютере методов сортировки из этой главы. Включите также все варианты методов, разработанные вами в качестве упражнений. Изобразите графики для сравнения следующих данных:
- (a) число операций сравнения ключей;
  - (b) число операций присваивания элементов;
  - (c) полное время выполнения;
  - (d) требования к объему памяти;

- (е) длина программы;
- (ф) время, требуемое для написания и отладки программы.
- Е7.** Напишите одностраничную инструкцию в помощь пользователю вашей компьютерной системы, чтобы облегчить ему выбор алгоритма сортировки согласно требованиям и условиям конкретного приложения.
- Е8.** Процедура сортировки называется **стабильной**, если при наличии в списке двух одинаковых ключей, соответствующие им элементы после сортировки останутся в списке в том же порядке, в котором они были до сортировки. Стабильность важна в тех случаях, когда список уже был упорядочен по одному ключу, и теперь упорядочивается по другому, и при этом желательно сохранить результаты исходного упорядочения, насколько это допускается новым упорядочением. Определите, какие из описанных в настоящей главе методов сортировки являются стабильными, и какие нет. Для нестабильных методов представьте список (возможно более короткий), содержащий некоторые из элементов с равными ключами, порядок которых не сохранился. В дополнение посмотрите, не можете ли вы предложить простые модификации алгоритма, которые сделают его стабильным.

стабильные  
методы  
сортировки

## Подсказки и ловушки

1. Многие компьютерные системы имеют встроенную утилиту сортировки общего назначения. Если у вас есть к ней доступ и вы находите ее адекватной вашим потребностям, используйте ее и не тратьте время на разработку с нуля новой программы сортировки.
2. При выборе метода сортировки принимайте в расчет такие вопросы, как типичный исходный порядок ключей, подлежащих сортировке, размер приложения, доступный объем машинного времени, необходимость экономии компьютерного времени и пространства памяти, способ реализации структур данных, стоимость перемещения данных и стоимость сравнения ключей.
3. Наиболее широко используемым и наиболее мощным методом разработки алгоритмов является метод разбиения (метод «разделяй и властвуй»). Столкнувшись с некоторой программистской проблемой, посмотрите, нельзя ли получить требуемый результат путем решения сначала двух (или большего числа) проблем того же общего характера, но меньшего размера. Если это так, вы, возможно, сможете сформулировать алгоритм, использующий метод разбиения и рекурсивный подход.
4. Сортировка слиянием, быстрая сортировка и пирамидальная сортировка являются мощными методами, которые, при относительной сложности программирования в сравнении с более простыми методами, показывают значительно более высокую эффективность при работе с большими списками. Внимательно продумайте требования вашего приложения и решите, оправданы ли в вашем случае затраты труда на реализацию одного из этих сложных алгоритмов.

5. Очереди с приоритетами важны для многих приложений, а пирамидальные структуры предоставляют превосходную реализацию таких очередей.
6. Пирамидальная сортировка напоминает стратегию страхования. Обычно она медленнее быстрой сортировки, но зато гарантирует, что сортировка будет выполнена за  $O(n \log n)$  операций сравнения ключей, что не достигается методом быстрой сортировки.

## Обзорные вопросы

- 8.2
  1. Сколько операций сравнения ключей требуется для проверки того, что список из  $n$  элементов уже упорядочен?
  2. Объясните с помощью не более 20 слов, как работает сортировка включением.
- 8.3
  3. Объясните с помощью не более 20 слов, как работает сортировка выбором.
  4. На сколько в среднем больше операций сравнения ключей потребует алгоритм сортировки выбором по сравнению с алгоритмом сортировки включением при обработке списка из 20 элементов?
  5. Каковы преимущества сортировки выбором по сравнению со всеми остальными рассмотренными выше методами?
- 8.4
  6. Каковы недостатки метода сортировки включением, которые преодолеваются в методе Шелла?
- 8.5
  7. Какова нижняя граница числа операций сравнения ключей, выполняемых любым методом сортировки при размещении в порядке  $n$  ключей, если этот метод использует для упорядочения сравнения ключей? Приведите значения для среднего и наихудшего случаев.
  8. Какова нижняя граница, если опустить требование использования для принятия решений сравнение ключей?
- 8.6
  9. Определите термин *разработка методом разбиения*.
  10. Объясните с помощью не более 20 слов, как работает сортировка слиянием.
  11. Объясните с помощью не более 20 слов, как работает быстрая сортировка.
- 8.7
  12. Объясните, почему сортировка слиянием оказывается эффективнее для связанных списков, чем для списков непрерывных.
- 8.8
  13. Почему в методе быстрой сортировки мы выбираем опорный ключ в центре списка, а не на одном из его концов?
  14. Насколько в среднем алгоритм быстрой сортировки выполняет больше сравнений ключей, чем оптимальное число сравнений? Приблизительно сколько сравнений выполняется этим алгоритмом в худшем случае?
- 8.9
  15. Что такое пирамида?
  16. Как работает пирамидальная сортировка?

8.10

17. Сравните производительность пирамидальной сортировки в худшем случае с аналогичным показателем для быстрой сортировки, а также с производительностью быстрой сортировки для среднего случая.
18. Когда простые алгоритмы сортировки оказываются лучше сложных и совершенных алгоритмов?

## Литература для дальнейшего изучения

Основным пособием для настоящей главы является книга [Knuth], том 3 (см. ссылку в главе 3). Методы внутренней сортировки описаны в томе 3, стр. 73–180. Д. Кнут выполнил анализ алгоритмов значительно более подробно, чем это сделали мы, разрабатывая свои алгоритмы на псевдо-ассемблере и подсчитывая при этом число операций. Он описал все методы, приведенные в настоящей книге, несколько дополнительных методов и много различных вариантов.

Исходные ссылки на метод сортировки Шелла и на метод быстрой сортировки, соответственно, см.

D. L. Shell, «A high-speed sorting procedure», *Communication of the ACM* 2 (1959), 30–32.

C. A. R. Hoare, «Quicksort», *Computer Journal* 5 (1962), 10–15.

Последовательное развитие методов сортировки слиянием и быстрой сортировки, которое может быть также применено к методам сортировки включением и выбором, основано на работе

John Darlington, «A synthesis of several sorting algorithms», *Acta Informatica* 11 (1978), 1–30.

Метод сортировки слиянием может быть улучшен с приближением его производительности к оптимальной нижней границе. Пример такого улучшения с достижением производительности всего на 6 процентов ниже наилучшего возможного значения приведен в работе

R. Michael Tanner, «Minimean merging and sorting: An algorithm», *SIAM J. Computing* 7 (1978), 18–38.

Относительной простой алгоритм сортировки слиянием для непрерывных списков, затраты времени которого возрастают линейно с увеличением числа элементов, и которому требуется небольшой дополнительный объем памяти постоянной величины, описан в статье

Bing-Chao Huang and Michael A. Langston, «Practical In-Place Merging», *Communications of the ACM* 31 (1988), 348–352.

Алгоритм разделения списка для метода быстрой сортировки был предложен Нико Ломуто (Nico Lomuto) и опубликован в работе

Jon L. Bentley, «Programming pearls: How to sort», *Communications of the ACM* 27 (1984), 287–291.

Колонка «Programming pearls» («Жемчужины программирования») содержит много элегантных алгоритмов и полезных соображений для программирования, которые были собраны в следующих двух книгах:

Jon L. Bentley, *Programming Pearls*, Addison-Wesley, Reading, Mass., 1986, 195 pages.

Jon L. Bentley, *More Programming Pearls*, Addison-Wesley, Reading, Mass., 1988, 224 pages.

Имеются русские переводы:

Бентли Д. Жемчужины творчества программистов. — М.: Радио и связь, 1990. — 225 с.

Бентли Д. Жемчужины программирования. — 2-е изд. СПб.: Питер, 2002. — 272 с.

Подробный анализ алгоритма быстрого упорядочения дан в статье Robert Sedgewick, «The analysis of quicksort programs», *Acta Informatica* 7 (1976–77), 327–355.

Упражнение на метод сортировки по среднему значению (когда в качестве опорного ключа берется ключ со средним значением) взято из работы

Dalia Motzkin, «MEANSORT», *Communications of the ACM* 26 (1983), 250–251; 27 (1984), 719–722.

Пирамидальная сортировка предложена и названа так в работе J. W. J. Williams, *Communications of the ACM* 7 (1964), 347–348.

Простые, но полные программы алгоритмов пирамидальной сортировки и очередей с приоритетами даны в статье

Jon L. Bentley, «Programming pearls: Thanks, heaps», *Communications of the ACM* 28 (1985), 245–250.

Разумеется, имеется богатая литература по теории вероятностей и математической статистике с возможными приложениями к вычислительной технике. Классической работой такого рода является книга

W. Feller, *An Introduction to Probability Theory and Its Applications*, Vol 1, second edition, Wiley-InterScience, New York, N.Y., 1957.

Имеется русский перевод: В. Феллер. Введение в теорию вероятностей и ее приложения. — М.: Мир, 1983.

# Таблицы и извлечение информации

В этой главе продолжается изучение вопросов извлечения информации, начатое в главе 7, но теперь вместо списков мы рассмотрим таблицы. Мы начнем с простых прямоугольных массивов, затем рассмотрим массивы другого рода и, наконец, в процессе обобщения перейдем к хеш-таблицам. Одной из наших важнейших целей, как и раньше, будет анализ и сравнение различных алгоритмов, что даст нам возможность грамотно выбирать алгоритмы для тех или иных условий. Прикладные программы, рассматриваемые в этой главе, включают метод сортировки, основанный на таблицах, и вариант игры «Жизнь», использующий хеш-таблицы.

## 9.1. Введение: переход через барьер $Ig\ n$

В главе 7 было показано, что при использовании лишь сравнения ключей, невозможно завершить поиск  $n$  элементов за меньшее (в среднем) число сравнений, чем  $Ig\ n$ . Но этот вывод относится только к сравнению ключей. Если мы сможем использовать какой-нибудь другой метод, тогда, возможно, нам удастся организовать наши данные так, что мы сможем обнаруживать нужный элемент более быстро.

табличный поиск

Фактически мы обычно так и делаем. Если у нас есть 500 различающихся записей, каждой из которых назначен индекс от 1 до 500, нам и в голову не придет при поиске элемента использовать последовательный или двоичный поиск. Мы просто сохраним наши записи в массиве размером 500 элементов и используем индекс  $n$  для определения местонахождения записи элемента  $n$  простым просмотром таблицы.

функции для  
извлечения  
информации

И табличный просмотр, и поиск с тем или иным перебором вариантов имеют в сущности общую цель, именно, *извлечение информации*. Мы начинаем с ключа (который может быть сколь угодно сложным или просто представлять собой индекс) и ставим перед собой задачу поиска элемента (если он существует) с этим ключом. Другими словами, и табличный просмотр, и наши алгоритмы поиска перебором вариантов предоставляют функции перехода от набора ключей к ячейкам списка или массива. Функции эти фактически образуют отношение один к одному от набора возможных ключей к набору возможных ячеек, поскольку мы полагаем, что каждый элемент имеет только один ключ и с каждым ключом имеется только один элемент.

В этой главе мы изучим способы реализации таблиц и доступа к ним в непрерывной памяти, начав с обычных прямоугольных массивов, после чего рассмотрим таблицы с ограниченным местоположением ненулевых элементов, например, треугольные таблицы. После этого мы перейдем к более общим проблемам, с целью ввести понятия и научиться использовать для извлечения информации сначала таблицы доступа, а затем и хеш-таблицы.

Мы увидим, что, в зависимости от формы таблицы, для извлечения информации может потребоваться несколько шагов, но даже при этом требуемое время остается порядка  $O(1)$ , т. е. оно ограничено константой, не зависящей от размера таблицы — и по этой причине табличный поиск может быть более эффективен, чем поиск любым из рассмотренных методов.

## 9.2. Прямоугольные массивы

В силу важности прямоугольных массивов, почти все языки высокого уровня предоставляют удобные и эффективные средства их хранения и доступа к ним, поэтому обычно программист не должен беспокоиться о деталях реализации. Однако, несмотря на то, что компьютерная память существенно организована в виде непрерывной последовательности (другими словами, в виде прямой линии, на которой следующий элемент примыкает к предыдущему), при каждом обращении к прямоугольному массиву машина должна выполнить некоторую работу по преобразованию позиции в прямоугольном массиве в позицию вдоль этой линии. Давайте рассмотрим этот процесс чуть более подробно.

### 1. Развертывание по строкам и по столбцам

Возможно, наиболее естественным способом чтения прямоугольного массива является чтение элементов первой строки слева направо, затем элементов второй строки и т. д., пока не будет прочитана последняя строка. Это, кстати, именно тот порядок, в котором большинство компиляторов сохраняют в памяти прямоугольные массивы, и этот порядок называется *развертыванием по строкам*. Например, если строки массива пронумерованы от 1 до 2, а столбцы пронумерованы от 1 до 3, тогда порядок индексов, при котором элементы массива хранятся упорядоченными по строкам, будет таким:

[1, 1]   [1, 2]   [1, 3]   [2, 1]   [2, 2]   [2, 3]

Fortran

Однако стандартный вариант языка Fortran использует *развертывание по столбцам*, при котором сначала располагаются элементы первого столбца, затем второго и т. д. Наш пример при упорядочении по столбцам будет выглядеть так:

[1, 1]   [2, 1]   [1, 2]   [2, 2]   [1, 3]   [2, 3]

На рис. 9.1 представлено упорядочение по строкам и столбцам для массива с тремя строками и четырьмя столбцами.



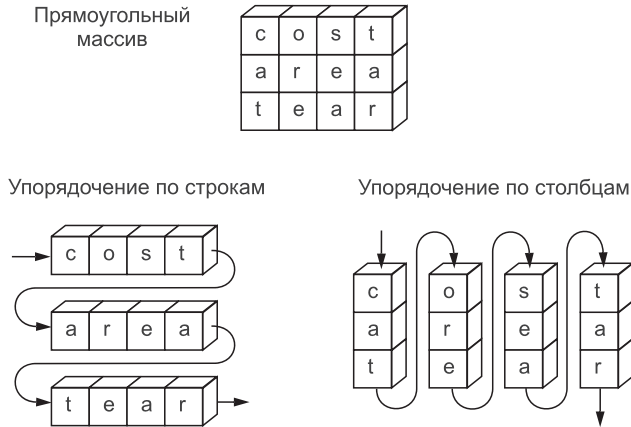


Рис. 9.1. Последовательное представление прямоугольного массива

## 2. Индексация прямоугольных массивов

В общем случае компилятор должен иметь возможность, начав с индекса  $[i, j]$ , вычислить, где в памяти располагается соответствующий элемент массива. Мы выведем формулу для такого рода вычислений. Для простоты будем использовать только упорядочение по строкам и будем также полагать, что строки пронумерованы от 0 до  $m - 1$ , а столбцы — от 0 до  $n - 1$ . Общий случай мы оставим для упражнений. Всего массив будет иметь  $mn$  элементов, так же, как и его последовательное представление. Пронумеруем элементы массива от 0 до  $mn - 1$ . Чтобы получить формулу для вычисления позиции элемента с индексом  $[i, j]$ , мы сначала рассмотрим некоторые специальные случаи. Очевидно, что индекс  $[0, 0]$  располагается в позиции 0, и фактически решить нашу задачу для всей первой строки не представляет труда: индекс  $[0, j]$  располагается в позиции  $j$ . Первый элемент следующей строки,  $[1, 0]$ , идет вслед за  $[0, n - 1]$ , и, таким образом, располагается в позиции  $n$ . Продолжая, мы видим, что элемент  $[1, j]$  располагается в позиции  $n + j$ . Элементы следующей строки будут иметь перед собой две полных строки, т. е.  $2n$  элементов. Отсюда элемент  $[2, j]$  будет располагаться в позиции  $2n + j$ . В общем случае элементы строки  $i$  предваряются  $ni$  предыдущими элементами, поэтому искомая формула выглядит так:

*Элемент  $[i, j]$  располагается в позиции  $ni + j$*

Формула такого рода, которая позволяет получить последовательное положение элемента массива, носит название **индексной функции**.

## 3. Вариант: таблица доступа

Индексную функцию для прямоугольного массива получить, разумеется, нетрудно, и компиляторы большинства языков высокого уровня будут попросту переводить на машинный язык все шаги, необходимые для вычисления местоположения ячейки, каждый раз, когда в программе встре-

таблица доступа,  
прямоугольный  
массив

чается ссылка на элемент прямоугольного массива. Однако на малых машинах умножение может выполняться весьма медленно, и для устранения операций умножения можно предложить несколько иной метод.

Метод заключается в хранении вспомогательного массива, части таблицы умножения на  $n$ . Этот массив будет содержать значения

$$0, \quad n, \quad 2n, \quad 3n, \quad \dots, \quad (m-1)n$$

Заметьте, что этот массив значительно меньше (как правило), чем прямоугольный массив, поэтому его можно постоянно держать в памяти, не думая о занимаемом им месте. Его элементы можно вычислить только один раз (и заметьте, что для их вычисления потребуются лишь операции сложения). При всех последующих ссылках на прямоугольный массив компилятор может найти позицию для  $[i, j]$ , взяв элемент в позиции  $i$  вспомогательной таблицы, прибавив  $j$  и перейдя на результирующую позицию.

Эта вспомогательная таблица дает нам первый пример *таблицы доступа*, или *ссылочной таблицы* (см. рис. 9.2). В общем случае таблица доступа представляет собой вспомогательный массив, используемый для нахождения данных, хранящихся где-то в другом месте. Таблицу доступа иногда называют *вектором доступа*.

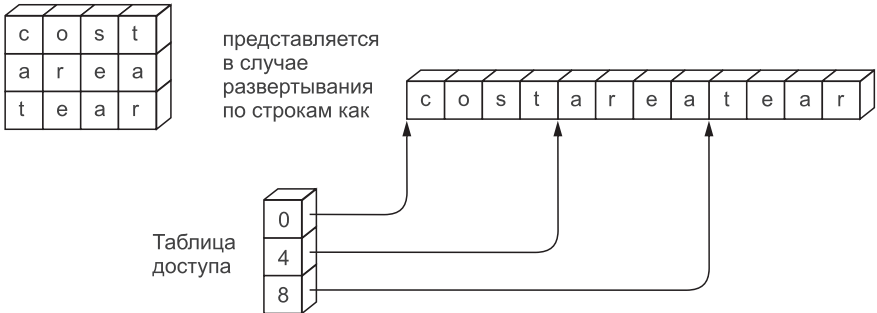


Рис. 9.2. Таблица доступа для прямоугольного массива.

## Упражнения 9.2

**Е1.** Какова будет индексная функция для двумерного прямоугольного массива с границами

$$\text{array} [0..m-1, 0..n-1]$$

в случае развертывания по столбцам?

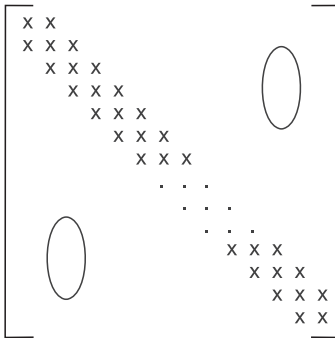
**Е2.** Приведите индексную функцию при развертывании по строкам двумерного массива с произвольными границами

$$\text{array} [r..s, t..u]$$

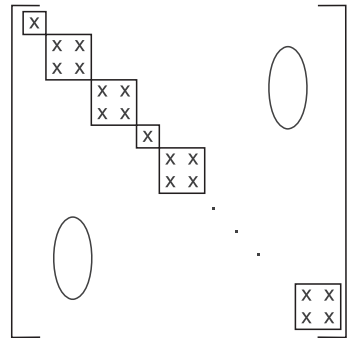
**Е3.** Найдите общий вид индексной функции при развертывании по строкам массива с  $d$  измерениями и произвольными границами для каждого измерения.

## 9.3. Таблицы различных форм

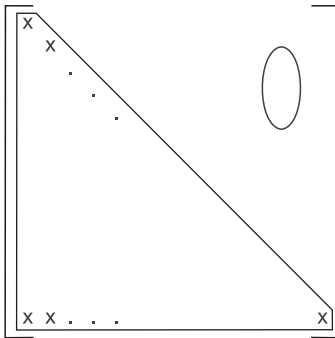
Информация, содержащаяся в прямоугольном массиве, не обязательно должна при своем представлении занимать все позиции прямоугольника. Если мы определим **матрицу** как массив чисел, тогда довольно часто некоторые из позиций в матрице будут содержать 0. Несколько примеров такого рода изображено на рис. 9.3. Даже когда элементы таблицы не являются числами, фактически используемые позиции могут не занимать всю площадь прямоугольника, и для таких случаев можно предусмотреть лучшую реализацию, чем использовать прямоугольный массив и оставлять ряд его позиций пустыми. В этом разделе мы рассмотрим способы реализации таблиц различных форм, способов, не требующих отведения в прямоугольном массиве неиспользуемого места.



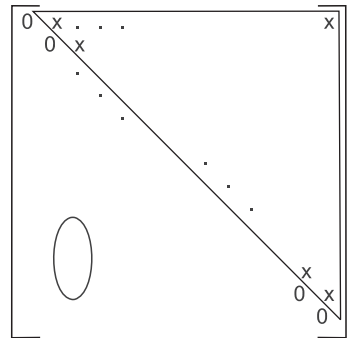
Трехдиагональная матрица



Блочно-диагональная матрица



Нижняя треугольная матрица



Строго верхняя треугольная матрица

Рис. 9.3. Матрицы различных форм

### 9.3.1. Треугольные таблицы

Давайте рассмотрим представление нижней треугольной таблицы, изображенной на рис. 9.3. Такую таблицу можно формально определить как таблицу, в которой все индексы  $[i, j]$  должны удовлетворять условию  $i \geq j$ .

Мы можем реализовать треугольную таблицу в непрерывном массиве, смещая каждую строку к концу предыдущей строки, как это показано на рис. 9.4.

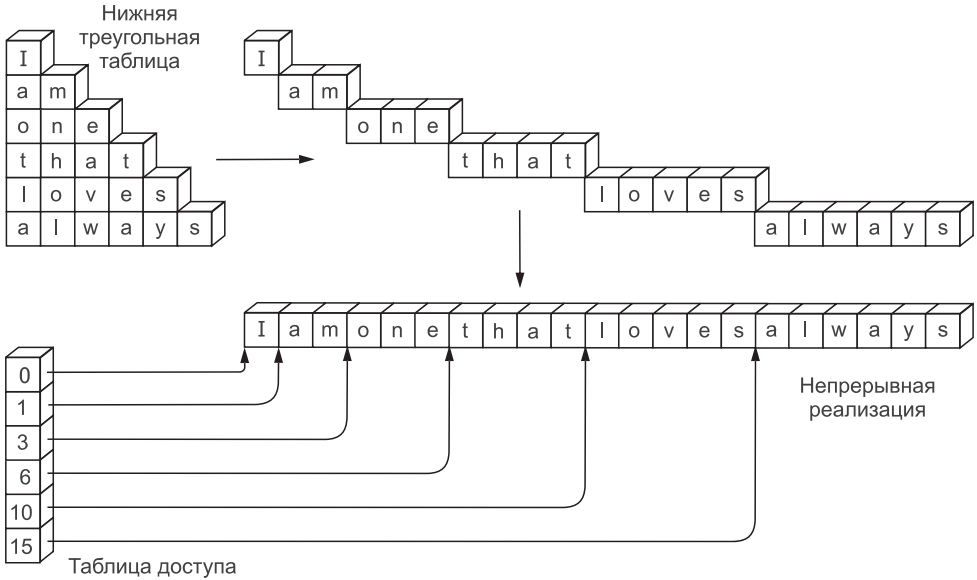


Рис. 9.4. Непрерывная реализация треугольной таблицы

Для построения индексной функции, описывающей такое отображение элементов в памяти, мы снова слегка упрощаем задачу, принимая, что строки и столбцы нумеруются от 0. Для определения позиции элемента  $[i, j]$  мы теперь должны найти, где начинается строка  $i$ , после чего для определения положения столбца  $j$  мы должны лишь добавить  $j$  к начальной позиции строки  $i$ . Если элементы непрерывного массива также пронумерованы от 0, тогда индекс начальной позиции будет совпадать с числом элементов, предшествующих строке  $i$ . Очевидно, что перед строкой 0 имеется 0 элементов, и только один элемент строки 0 предшествует строке 1. Для строки 2 число предшествующих элементов составит  $1 + 2 = 3$ , и в общем случае мы видим, что перед строкой  $i$  всегда находятся

индексная  
функция,  
треугольная  
таблица

$$1 + 2 + \dots + i = \frac{1}{2}i(i + 1)$$

элементов. Отсюда требуемая функция должна для элемента  $[i, j]$  треугольной таблицы давать нам соответствующий элемент

$$\frac{1}{2}i(i + 1) + j$$

непрерывного массива.

таблица доступа,  
треугольная  
таблица

Как это было и в случае прямоугольных массивов, мы можем устранить необходимость использовать операции умножения и деления, организовав таблицу доступа, элементы которой соответствуют индексам треугольной таблицы. Позиция  $i$  таблицы доступа будет всегда содержать значение  $\frac{1}{2}i(i + 1)$ . Таблицу доступа можно вычислить один раз в начале программы, а затем использовать повторно при каждом обращении к треугольной таблице. Заметьте, что даже начальные вычисления этой таблицы доступа не требуют операций умножения или деления, так как для вычисления ее элементов достаточно выполнить операции сложения:

$$0, \quad 1, \quad 1 + 2, \quad (1 + 2) + 3, \quad \dots$$

### 9.3.2. Рваные таблицы

В обоих предшествующих примерах мы рассматривали прямоугольный массив как состоящий из его строк. В обычных прямоугольных массивах все строки имеют одинаковую длину; в треугольных таблицах длина каждой строки может быть найдена с помощью простой формулы. Теперь мы рассмотрим случай «рваных» таблиц, вроде той, что представлена на рис. 9.5, где между позицией строки и ее длиной нет предсказуемого отношения.

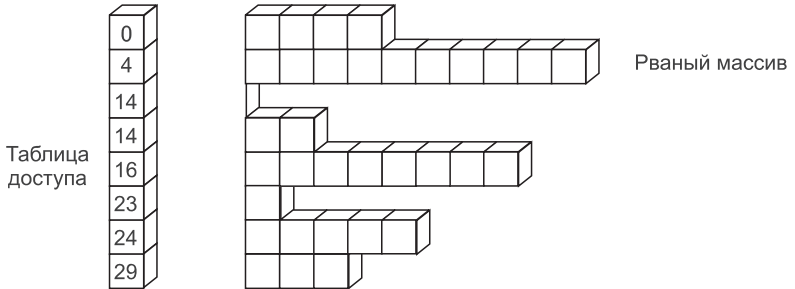


Рис. 9.5. Таблица доступа для рванной таблицы

Из этого рисунка ясно, что хотя мы не можем составить априорную функцию, которая отображала бы рваную таблицу на непрерывную память, использование таблицы доступа оказывается столь же простым, как и в предыдущих примерах, и к элементам рванной таблицы можно обращаться так же быстро, как и раньше. Чтобы создать таблицу доступа, мы должны расположить равную таблицу в ее естественном порядке, начиная с первой строки. Элемент 0 таблицы доступа, как и раньше, указывает на начало непрерывного массива. После расположения в памяти каждой строки рванной таблицы, индекс первой неиспользуемой позиции непрерывной памяти следует ввести в следующий элемент таблицы доступа и использовать его для конструирования следующей строки равной таблицы.

### 9.3.3. Инвертированные таблицы

Теперь давайте рассмотрим пример, иллюстрирующий использование таблиц множественного доступа, с помощью которых мы сможем обращаться к единой таблице записей посредством нескольких различных ключей.

Рассмотрим проблему, с которой сталкивается телефонная компания при обращении к записям о своих клиентах. При публикации телефонной книги записи следует упорядочить по алфавиту абонентов телефонных номеров, но для обработки междугородних переговоров счета должны быть упорядочены по телефонным номерам. Для выполнения планового обслуживания компания нуждается также в списках своих абонентов, упорядоченных по их адресам, чтобы ремонтный мастер мог обслужить несколько вызовов за одну поездку. В этом случае компания может, конечно, хранить три (или больше) наборов записей, один упорядоченный по именам, другой по номерам и третий по адресам. Это, однако, не только приведет к напрасному расходованию машинной памяти, но будет служить источником постоянных неприятностей, если один набор данных был обновлен, а другой нет, и в этом случае компания может получить ошибочную или вообще непредсказуемую информацию.

Используя таблицы доступа, мы можем устранить множественные наборы записей, и при этом мы сможем по-прежнему отыскивать требуемые записи по любому из трех ключей почти так же быстро, как если бы записи были полностью упорядочены по этому ключу. Для имен абонентов мы организуем одну таблицу доступа. Первый элемент в такой таблице представляет собой позицию записи об абоненте, имя которого идет первым по алфавиту, второй элемент дает позицию записи следующего (в алфавитном порядке) абонента и т. д. Во второй таблице доступа первый элемент указывает на расположение записи об абоненте, телефонный номер которого является наименьшим в численном порядке. В третьей таблице доступа элементы предоставляют доступ к записям, упорядоченным лексикографически по адресам.

Заметьте, что в этом методе все поля, рассматриваемые как ключи, обрабатываются единым образом. Нет никакой особой нужды упорядочивать сами записи об абонентах именно по одному, а не по другому ключу, да и вообще нет никакой необходимости их как-то упорядочивать. Сами записи могут храниться в произвольном порядке, скажем, в том порядке, в котором они были впервые введены в систему. Не имеет также особого значения, хранятся ли записи в массиве, и тогда элементы таблиц доступа представляют собой индексы этого массива, или записи хранятся в динамической памяти, и тогда в таблицах доступа содержатся указатели на отдельные записи. В любом случае для извлечения информации используются таблицы доступа, и, как и в случае непрерывных массивов, эти таблицы можно использовать для непосредственного просмотра, или для двоичного поиска, или с любой другой целью, допустимой для непрерывной реализации.

Пример такой схемы для небольшого числа абонентов показан на рис. 9.6.

множественные  
наборы записей

таблицы  
множественного  
доступа

неупорядоченные  
записи для  
упорядоченных  
таблиц доступа

<i>Индекс</i>	<i>Имя</i>	<i>Адрес</i>	<i>Телефон</i>
1	Hill, Thomas M.	High Towers #317	2829578
2	Baker, John S.	17 King Street	2884285
3	Roberts, L. B.	53 Ash Street	4372296
4	King, Barbara	High Towers #802	2863386
5	Hill, Thomas M.	39 King Street	2495723
6	Byers, Carolyn	118 Maplr Street	4394231
7	Moody, C. L.	High Towers #210	2822214

Таблицы доступа

<i>Имя</i>	<i>Адрес</i>	<i>Телефон</i>
2	3	5
6	7	7
1	1	1
5	4	4
4	2	2
7	5	3
3	6	6

Рис. 9.6. Таблицы доступа по многим ключам: инвертированная таблица

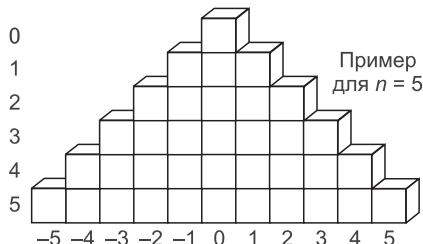
### Упражнения 9.3

- Е1. Главная диагональ** квадратной матрицы состоит из элементов, для которых индексы строки и столбца совпадают. **Диагональной матрицей** называется квадратная матрица, в которой все элементы, не находящиеся на главной диагонали, равны 0. Опишите способ хранения диагональной матрицы, не требующий использования пространства памяти для элементов, равных 0, и приведите соответствующую индексную функцию.
- Е2. Трехдиагональная матрица** представляет собой квадратную матрицу, у которой все элементы равны 0 за исключением, возможно, тех, что расположены на главной диагонали, а также на диагоналях непосредственно выше и ниже главной (см. рис. 9.3). Другими словами,  $T$  есть трехдиагональная матрица, если  $T[i, j] = 0$  за исключением тех случаев, когда  $|i - j| \leq 1$ .
- (а) Разработайте эффективную с точки зрения расходования памяти схему хранения трехдиагональных матриц и приведите соответствующую индексную функцию.
- (б) **Транспонированная матрица** образуется путем обмена ее строк с соответствующими столбцами. Другими словами, матрица  $B$  является транспонированной от матрицы  $A$ , если  $B[j, i] = A[i, j]$  для

всех индексов  $i$  и  $j$ , соответствующих позициям в матрице. Предложите алгоритм, который транспонирует трехдиагональную матрицу, использующую схему хранения в памяти, разработанную в предыдущей части этого упражнения.

**Е3. Верхней треугольной матрицей** называется квадратный массив, в котором все элементы ниже главной диагонали равны 0 (см. рис. 9.3). Опишите, каким образом можно использовать метод таблиц отклика для хранения в памяти верхней треугольной матрицы.

**Е4.** Рассмотрите таблицу треугольной формы, изображенную на рис. 9.7, где столбцы индексируются от  $-n$  до  $n$ , а строки от 0 до  $n$ .



**Рис. 9.7.** Таблица, симметрично треугольная относительно 0

- Разработайте индексную функцию, которая отобразит таблицу такой формы на последовательный массив.
- Напишите процедуру, которая будет генерировать таблицу доступа, позволяющую находить в непрерывном массиве первый элемент каждой строки таблицы такой формы.
- Напишите процедуру, которая будет отражать таблицу слева направо. Элементы в столбце 0 (центральный столбец) остаются без изменений, столбцы с номерами  $-1$  и  $1$  обмениваются местами и т. д.

### Программные проекты 9.3

Реализуйте метод, описанный в тексте и использующий таблицу доступа для хранения нижней треугольной таблицы. Примените этот метод к следующим проектам.

- Р1.** Напишите процедуру, которая будет считывать элементы нижней треугольной таблицы с клавиатуры.
- Р2.** Напишите процедуру, которая будет выводить элементы нижней треугольной таблицы на экран.
- Р3.** Будем считать, что нижняя треугольная таблица является таблицей расстояний между городами, как это часто изображается на дорожных картах. Напишите процедуру, которая будет проверять правило треугольника: расстояние между городами  $A$  и  $C$  никогда не превышает расстояния между  $A$  и  $B$ , плюс расстояние от  $B$  до  $C$ .
- Р4.** Встройте процедуру из предыдущего проекта в законченную программу для демонстрации нижних треугольных таблиц.



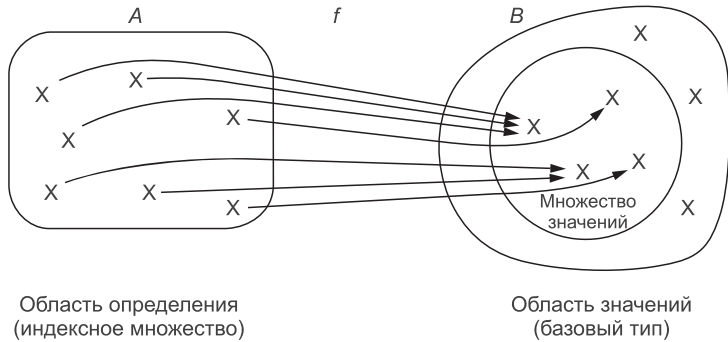
# 9.4. Таблицы: новый абстрактный тип данных

В начале этой главы мы изучили несколько *индексных функций*, используемых для поиска элементов в таблицах, и затем мы обратились к *таблицам доступа*, которые представляют собой массивы, используемые с той же целью, что и индексные функции. Аналогия между функциями и табличным просмотром оказывается весьма тесной: имея функцию, мы задаем аргумент и вычисляем соответствующее значение; имея таблицу, мы задаем индекс и ищем соответствующее значение в таблице. Давайте теперь используем эту аналогию для получения формального определения термина *таблица*, определения, которое, в свою очередь, будет служить мотивацией для формулирования новых идей, которые будут реализованы в следующем разделе.

## 1. Функции

В математике *функция* определяется как два набора элементов, для которых задано соотношение между элементами первого набора и элементами второго набора. Если  $f$  есть функция набора  $A$ , образующая набор  $B$ , тогда  $f$  назначает каждому элементу  $A$  уникальный элемент  $B$ . Набор  $A$  называют *областью определения*  $f$ , а набор  $B$  называют *областью значений* функции  $f$ . Поднабор  $B$ , содержащий только те элементы, которые являются значениями  $f$ , называют *множеством значений* функции  $f$ . Эти определения проиллюстрированы на рис. 9.8.

область  
определения,  
область значений  
и множество  
значений



**Рис. 9.8.** Область определения, область значений и множество значений функции

При табличном доступе мы имеем индекс и используем таблицу чтобы найти соответствующее значение. Поэтому для таблицы мы называем область определения *индексным множеством*, а область значений — *базовым типом* или *типом значения*. (Вспомним, что в разделе 5.7 *тип* был определен как набор значений.) Если, например, мы имеем объявление массива

набор индексов,  
тип значения

`array [m..n] of real`

тогда индексное множество представляет собой набор целых чисел между  $m$  и  $n$ , а базовый тип есть набор всех действительных чисел. В качест-

ве второго примера рассмотрим треугольную таблицу с  $m$  строками, элементы которой имеют тип `item`. Тогда базовый тип есть просто тип `item`, а индексное множество есть набор упорядоченных пар целых чисел

$$\{(i, j) \mid 1 \leq j \leq i \leq m\}$$

## 2. Абстрактный тип данных

Теперь мы уже приблизились к определению *таблицы* как нового абстрактного типа данных, однако из раздела 5.7 можно вспомнить, что для получения полного определения мы должны также задать допустимые операции. Перед тем, как это сделать, подытожим все, что нам уже известно.

### Определение

**Таблица** с индексным множеством  $I$  и базовым типом  $T$  есть функция множества  $I$ , образующая  $T$ , вместе со следующими операциями:

1. *Табличный доступ*: оценка значения функции при любом индексе из  $I$ .
2. *Табличное присваивание*: модификация функции, изменяющее ее значение при заданном индексе из  $I$  на новое значение, заданное в операции присваивания.

Этими двумя операциями ограничиваются возможности языка Pascal и некоторых других языков, но это не причина, чтобы не допустить возможность других операций. Если мы вспомним определение списка, то мы увидим, что кроме доступа и присваивания мы допускали еще включение и удаление. То же самое мы можем сделать и с таблицами.

3. *Создание*: образование новой функции.
4. *Очистка*: удаление всех элементов из индексного множества  $I$ , так что область определения перестает существовать.
5. *Включение*: присоединение нового элемента  $x$  к индексному множеству  $I$  и определение соответствующего значения функции от  $x$ .
6. *Удаление*: удаление элемента  $x$  из индексного множества  $I$  и ограничение области определения функции результирующей меньшей областью.

Хотя эти последние операции в языке Pascal непосредственно недоступны, они остаются весьма полезными для многих приложений, и мы изучим их подробнее в следующем разделе. В некоторых других языках, например, APL и SNOBOL, таблицы, изменяющие свой размер в процессе выполнения программы, являются важным средством языка. В любом случае мы никогда не должны допускать, чтобы ограничения конкретного языка ограничивали наши идеи разработки программ.

## 3. Реализация

Только что данное определение является определением абстрактного типа и само по себе ничего не говорит ни о реализации данных, ни об индексных функциях или таблицах доступа, рассмотренных ранее. Индексные функции и таблицы доступа являются, в сущности, методами

метод разбиения

реализации таблиц более общего вида. Индексная функция или таблица доступа получают обобщенное индексное множество какой-либо конкретной формы и производят в качестве результата индекс в некоторой области значений (поддиапазоне) индексов, например, в поддиапазоне целых чисел. Этот диапазон может быть затем использован непосредственно как набор индексов для массивов, обеспечиваемых языком программирования. При таком подходе реализация таблицы разделяется на две меньшие задачи: нахождение таблицы доступа или индексной функции и программирование массива. Следует заметить, что и то, и другое есть специальные случаи таблиц, и, следовательно, мы здесь имеем пример решения задачи путем разбиения ее на две меньшие задачи той же природы. Этот процесс проиллюстрирован на рис. 9.9.

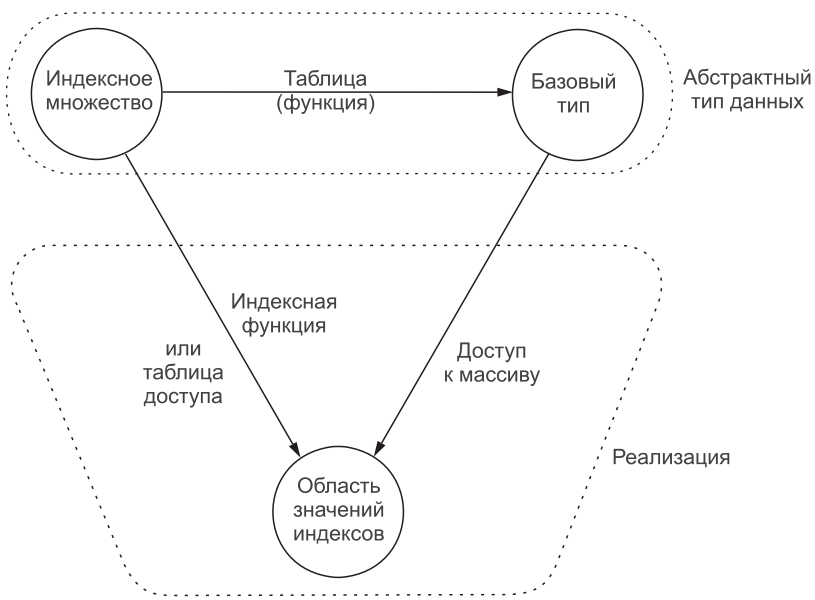


Рис. 9.9. Реализация таблицы

4. Сравнения

списки и таблицы

извлечение

Давайте сравним абстрактные типы данных *список* и *таблицу*. Математическая конструкция, лежащая в основе списка, есть последовательность, что же касается таблицы, то она представляет собой множество и функцию. Последовательности имеют явный порядок: первый элемент, затем второй и т. д., однако множества и функции не обладают таким порядком. (Если множество индексов имеет некоторый естественный порядок, то иногда этот порядок отражается и в таблице, но это условие не является обязательным.) Отсюда извлечение информации из списка естественным образом вовлекает операции поиска вроде тех, что изучались в предыдущей главе, однако извлечение информации из таблицы требует иных методов, методов доступа, позволяющих непосредственно обра-

таться к требуемому элементу. Время, требуемое для поиска в списке, обычно зависит от числа  $n$  элементов списка и по меньшей мере составляет  $\lg n$ , однако время доступа к таблице чаще всего не зависит от числа элементов в ней; другими словами, оно имеет порядок  $O(1)$ . По этой причине во многих приложениях табличный доступ выполняется существенно быстрее, чем поиск в списке.

просмотр

С другой стороны, просмотр является естественной операцией для списков, но не для таблиц. В общем случае не составляет особого труда перемещение по списку с выполнением некоторой операции над каждым элементом списка. Однако выполнение операции над каждым элементом таблицы в общем случае может оказаться не таким простым делом, особенно, если для элементов заранее задан некоторый определенный порядок.

таблицы  
и массивы

Наконец, следует прояснить различие между терминами *таблица* и *массив*. В общем случае мы будем использовать термин *таблица*, как мы определили его в этом разделе, а под термином *массив* будем ограниченно понимать программное средство, имеющееся в языке Pascal и большинстве других языков высокого уровня и используемое для реализации и таблиц, и связанных списков.

## 9.5. Приложение: поразрядная сортировка

Формальный алгоритм сортировки, предшествующий эпохе компьютеров, был впервые разработан для использования с перфокартами, однако его можно преобразовать в весьма эффективный метод сортировки для связанных списков, использующий таблицы и очереди.

### 9.5.1. Идея метода

Идея метода состоит в рассмотрении ключа по одному символу за раз и разделению элементов не на два подсписка, а на такое количество подсписков, сколько имеется возможностей для данного символа, взятого из ключа. Если наши ключи, например, представляют собой слова или другие алфавитно-цифровые строки, тогда мы разделяем список на этом этапе на 26 подсписков. Другими словами, мы создаем *таблицу* из 26 списков и распределяем элементы по спискам согласно каждому символу ключа.

Старомодные перфокарты содержали 12 строк; отсюда механические сортировщики карт разрабатывались так, что обрабатывали только один столбец за раз, и разделяли при этом карты на 12 стопок.

Человек, сортирующий слова этим методом, мог бы сначала распределить слова по 26 спискам согласно первой букве, затем разделить каждый из этих подсписков на дальнейшие подписки согласно второй букве и т. д. Следующая идея устраняет множество списков: подразделите элементы, образовав таблицу подсписков сначала по *наименее* значимой позиции, не по наиболее значимой. После этого первого деления подписки из таблицы складываются назад в единый список в порядке, задаваемом символом в наименее значимой позиции. Затем список разделя-

метод

ется с образованием таблицы подписков согласно второй в порядке повышения значимости позиции и опять собирается в единый список. Когда, после повторения этих шагов, список будет разделен в последний раз по наиболее значимой позиции и снова собран, он будет полностью упорядочен.

Этот процесс проиллюстрирован на примере сортировки списка из девяти трехбуквенных слов на рис. 9.10. Слова в исходном порядке показаны в левом столбце. Сначала они разделяются на три списка согласно их третьей букве, как это показано во втором столбце, где рамками обведены результирующие подписки. Порядок слов в каждом подписке остается тем же, каким он был до разделения. Далее подписки складываются вместе в том же порядке, в каком они образованы (второй столбец рисунка), после чего разделяются на два подписка согласно второй букве. Результат показан в рамках в третьем столбце. Наконец, эти подписки собираются вместе и распределяются на четыре подписка согласно первой букве. После их окончательного объединения весь список оказывается упорядоченным.

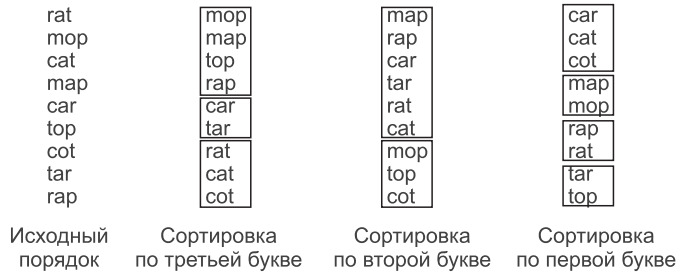


Рис. 9.10. Трассировка поразрядной сортировки

## 9.5.2. Реализация

Теперь мы реализуем этот метод на языке Pascal с помощью связанных списков, ключами которых будут строки символов фиксированной длины. Каждый раз после разделения элементов на подписки в таблице, подписки должны быть собраны в единый список, чтобы его можно было опять разделить согласно следующей по возрастанию значимости позиции в ключе. Мы будем работать с подписками как с очередями, поскольку элементы всегда включаются в конец подписки, а при объединении подписков удаляются от начала.

Для большей ясности представления мы используем для реализации наших действий обобщенный пакет обработки списков и очередей. Однако в этом случае возникают необоснованные перемещения данных, в то время как нам нужно всего лишь реализовать очереди в связной памяти. Для объединения всех связанных очередей в общий список нам нужно только подсоединить хвост каждой очереди к голове следующей. Этот процесс проиллюстрирован на рис. на рис. 9.11 для того же уже использованного нами ранее списка из девяти слов. На каждом этапе прямыми

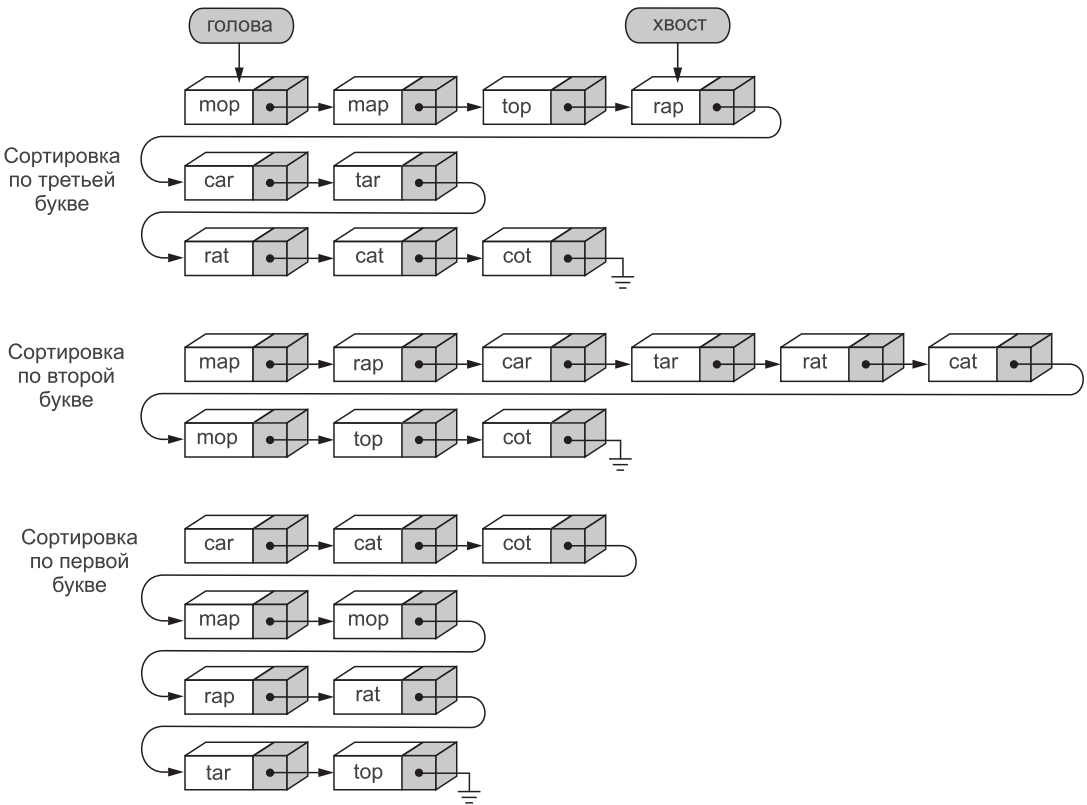


Рис. 9.11. Связная поразрядная сортировка

линиями со стрелками обозначены связи внутри очередей, а длинными изогнутыми линиями показаны связи, добавленные в процессе объединения очередей в единый список. Программирование этого оптимизированного варианта поразрядной сортировки оставлено для проекта.

Мы создадим массив из 28 связанных очередей. Позиция 0 соответствует символу пробела, позиции с 1 по 26 — буквам латинского алфавита (при этом строчные и прописные буквы считаются одинаковыми), а позиция 27 соответствует любому другому символу, который может встретиться в ключе. Внутри цикла, выполняемого в порядке от наименее значимой к наиболее значимой позиции в ключе, мы будем просматривать связный список и добавлять каждый элемент к концу соответствующей очереди. После того как список таким образом разделен, мы собираем очереди в единый список. В конце главного цикла по позициям в ключе список будет полностью упорядочен.

Что касается объявлений и обозначений, давайте использовать стандартные объявления и операции для списка (из главы 6) и для очереди (из главы 5). Мы принимаем объявление

```
keytype = array [1..keysize] of char
```

где константа `keysize` объявлена соответствующим образом.

## 1. Главная процедура

Процедура сортировки принимает такую форму:

```

procedure RadixSort (var L: list);                                { Поразрядная сортировка }
{ Pre: Связный список L уже создан. Все элементы L содержат ключи,
  которые представляют собой буквенные массивы размера
  keysize (константа).
  Post: Элементы L упорядочены так, что все их ключи располагаются
  в алфавитном порядке.
  Uses: Использует пакет обработки связанных очередей, подпрограммы
  Position и Rethread. }
type keyposition = 1..keysizes;
var i: keyposition;                                                { переменная управления циклом выбора
                                                                    позиции в ключе }

    Q: queuearray;
    j: queuecount;
    x: entry;
begin                                                                { процедура RadixSort }
  for j := 0 to maxqueuearray do
    CreateQueue(Q[j]);
  for i := keysize downto 1 do begin    { выполнить от младшего места
                                          до старшего }
    while not ListEmpty(L) do begin
      DeleteList(1, x, L);
      Append(x, Q[QueuePosition(x.key[i])])    { операция над очередью }
    end;
    Rethread(L, Q)                                { собрать список заново }
  end
end;                                                                { процедура RadixSort }

```

Эта процедура использует две вспомогательные подпрограммы: QueuePosition определяет, в какую конкретно очередь должен быть направлен данный ключ, а Rethread собирает очереди в единый список.

## 2. Определение позиции

Функция QueuePosition проверяет, является ли символ буквой, и назначает ему соответствующую позицию, а также назначает всем неалфавитным символам, отличным от пробела, позицию 27. Пробелам назначается позиция 0. Функция выполнена так, что она не делает различия между строчными и прописными буквами. Как и большинство других процедур обработки символов, QueuePosition является системно-зависимой; она будет нормально работать с кодировкой ASCII, но может потребовать модификаций при использовании других систем кодирования.

```

function QueuePosition (x: char): queuecount;                    { Позиция очереди }
{ Pre: Предусловия отсутствуют.
  Post: Если x является буквой алфавита (строчной или прописной),
    тогда функция возвращает позицию x в алфавите (от 1 для 'a'
    или 'A' до 26 для 'z' или 'Z'). В противном случае если x
    является пробелом, тогда возвращается 0; в остальных
    случаях функция возвращает 27.

```

**Uses:** Буквам должен соответствовать последовательный ряд кодов (что справедливо для кодировки ASCII, но не выполняется для кодировки EBCDIC). }

```

begin                                     { функция QueuePosition }
  if x = '' then                         QueuePosition := 0
  else if x in ['A'..'Z'] then           QueuePosition := ord(x) - ord('A') + 1
  else if x in ['a'..'z'] then           QueuePosition := ord(x) - ord('a') + 1
  else                                   QueuePosition := 27
end;                                     { функция QueuePosition }

```

### 3. Объединение очередей

Процедура Rethread объединяет 28 очередей в единый список. Проект требует переписывания этой процедуры в виде, зависящем от реализации, в результате чего процедура будет выполняться значительно быстрее приводимого здесь варианта.

```

procedure Rethread (var L: list; var Q: queuearray);           { Объединение }
{ Pre:   Все 28 связанных очередей в массиве Q уже созданы; список L уже
          создан и пуст.
  Post:  Все очереди объединены с образованием одного списка L; все
          очереди пусты. }

var
  i: queuecount;
  x: entry;
begin                                     { процедура Rethread }
  for i := 0 to maxqueuearray do
    while not QueueEmpty(Q[i]) do
      begin
        Serve(x, Q[i]);
        InsertList(ListSize(L) + 1, x, L)
      end;
  end;                                     { процедура Rethread }

```

### 9.5.3. Анализ

Заметьте, что время, требуемое для поразрядной сортировки, пропорционально  $nk$ , где  $n$  есть число сортируемых элементов, а  $k$  — число символов в ключе. Время, затрачиваемое всеми остальными методами сортировки, зависит от  $n$ , но длина ключа на него непосредственно не влияет. Наилучшее время, определяемое как  $\lg n + O(n)$ , давала сортировка слиянием. Относительная производительность методов будет, таким образом, как-то зависеть от относительной величины  $nk$  и  $n \lg n$ . Если ключи имеют большую длину, но их мало, тогда  $k$  велико, но  $n$  относительно мало, и другие методы (например, сортировка слиянием) дадут лучшую производительность, чем поразрядная сортировка; но если  $k$  мало, а ключей много, тогда поразрядная сортировка окажется быстрее любых изученных нами методов.



## Упражнения 9.5

**E1.** Проследите действия поразрядной сортировки, примененной к списку из 14 имен, которым мы уже пользовались для трассировки других методов упорядочения:

Tim Dot Eva Roy Tom Kim Guy Ami Jon Ann Jim Kay Ron Jan

**E2.** Проследите действия поразрядной сортировки, примененной к следующему списку из семи чисел, рассматриваемых как двухразрядные целые:

26 33 35 29 19 12 22

**E3.** Проследите действия поразрядной сортировки, примененной к приведенному выше списку из семи чисел, рассматриваемых как шестизначные двоичные числа.

## Программные проекты 9.5

**P1.** Разработайте, запрограммируйте и оттестируйте вариант реализационно-независимой поразрядной сортировки с алфавитными ключами.

**P2.** Программа поразрядной сортировки, представленная в этой книге, весьма неэффективна, поскольку ее реализационная независимость приводит к большому объему перемещения данных. Разработайте реализационно-зависимый проект, в котором устранена необходимость перемещения данных. В процедуре Rethread вам нужно только связать хвост одной очереди с головой следующей. Сравните производительность этого варианта с производительностью других методов сортировки для связанных списков.

## 9.6. Хеширование

### 9.6.1. Разреженные таблицы

#### 1. Индексные функции

Мы можем по-прежнему использовать табличный просмотр даже в ситуациях, когда ключ уже не является индексом, которым можно непосредственно пользоваться для индексирования массива. Нам просто надо установить однозначное соответствие между ключами, с помощью которых мы хотим извлекать информацию, и индексами, которыми мы пользуемся для доступа к массиву. Созданная нами индексная функция будет несколько сложнее тех, что были рассмотрены в предыдущих разделах, поскольку ей понадобится преобразовывать ключ из, скажем, алфавитной формы в целочисленную, но в принципе это вполне можно сделать.

Единственная сложность возникает, когда количество возможных ключей превышает объем памяти, который мы можем отвести под нашу таблицу. Если, например, ключи являются восьмибуквенными словами, тогда всего имеется  $26^8 \approx 2 \times 10^{11}$  возможных ключей, что, скорее всего,

индексная  
функция  
с неоднозначным  
соответствием

превышает возможное число ячеек в быстрой памяти компьютера. Фактически, однако, в программу поступит небольшая доля этих ключей. Другими словами, таблица является *разреженной* (sparse). Концептуально мы можем рассматривать такую таблицу как индексируемую очень большим индексным множеством, но с относительно небольшим числом фактически занятых позиций. В языке Pascal, например, мы можем предусмотреть такую форму концептуального объявления:

```
type ... = sparse table [keytype] of entry
```

Хотя такое объявление невозможно реализовать непосредственно, оно может оказаться полезным на первом этапе решения задачи, когда мы рассуждаем на уровне концепций, и лишь постепенно доходим до деталей практической реализации разрабатываемого метода.

2. Хеш-таблицы

Идея *хеш-таблиц* (вроде той, что изображена на рис. 9.12) заключается в том, чтобы позволить многим различным ключам, которые могут поступить реально в программу, отображаться на одни и те же ячейки массива в результате действия индексной функции. В этом случае возникает вероятность того, что две записи затребуют одно и то же место, но если число фактически имеющихся записей относительно мало по сравнению с размером массива, тогда такая ситуация приведет лишь к небольшой потере времени. Даже когда большая часть ячеек массива оказывается занята, хеш-методы могут оставаться эффективным средством извлечения информации.

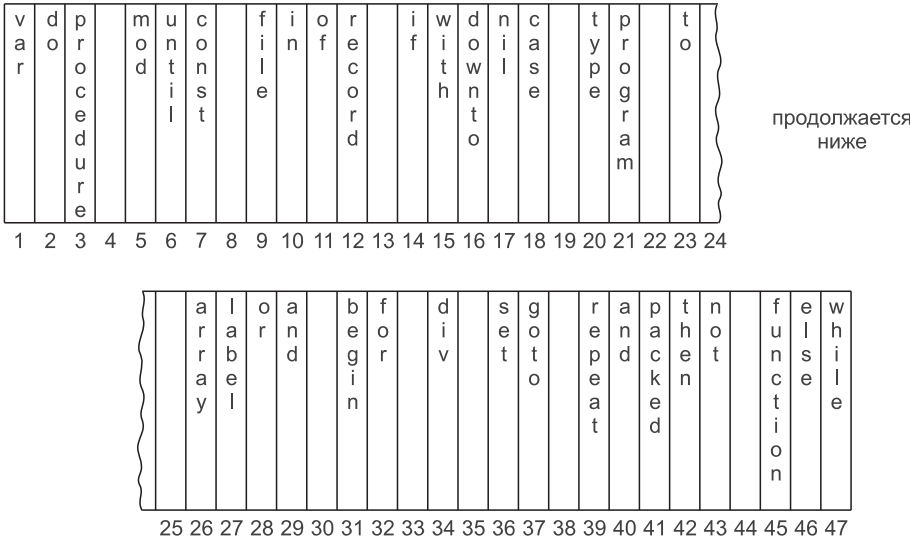


Рис. 9.12. Хеш-таблица

хеш-функция

Рассмотрим *хеш-функцию*, которая принимает ключ и отображает его на некоторый индекс в массиве. Такая функция будет в общем случае отображать несколько разных ключей на один и тот же индекс. Если тре-

буемая запись находится в ячейке, на которую указывает индекс, тогда наша проблема решена; в противном случае мы должны использовать какой-то метод разрешения **столкновений** или **конфликтов**, которые происходят, когда две записи хотят поступить в одну и ту же ячейку. Таким образом, при использовании хеширования мы должны решить две проблемы. Во-первых, мы должны найти удобную хеш-функцию, и, во-вторых, мы должны определить, как разрешать (т. е. ликвидировать) столкновения.

столкновение

Перед тем, как приступить к решению этих проблем, сделаем паузу и очертим неформально шаги, необходимые для реализации хеширования.

### 3. Схема алгоритма

Прежде всего следует объявить массив, который будет содержать хеш-таблицу. В обычных массивах в качестве ключей, с помощью которых мы обращаемся к элементам массива, используются индексы, поэтому не возникает необходимости хранить сами ключи в массиве, но в случае хеш-таблицы одному и тому же индексу могут соответствовать несколько ключей, поэтому одно поле в каждой записи массива должно быть зарезервировано под сам ключ.

ключи в таблице

инициализация

Далее, все ячейки в массиве должны быть инициализированы, чтобы показать, что они пусты. Как это сделать — зависит от приложения. Часто принимают такую схему: во все поля ключей записывают некоторое значение, которое гарантированно не может возникнуть в качестве фактического ключа. Если, например, используются алфавитно-цифровые ключи, пустую позицию может представлять ключ, состоящий из пробелов.

включение

Для включения в хеш-таблицу записи сначала вычисляется значение хеш-функции для данного ключа. Если соответствующая ячейка свободна, запись может быть включена в массив; если ключи оказались равными, включение новой записи не будет разрешено; в оставшихся случаях (в данной ячейке уже есть запись с другим ключом) возникает необходимость разрешения столкновения.

извлечение

Процедура извлечения записи с данным ключом практически совпадает с уже описанной. Прежде всего вычисляется хеш-функция для данного ключа. Если требуемая запись находится в соответствующей ячейке, извлечение прошло успешно; в противном случае, если ячейка не пуста, и еще не все ячейки были просмотрены, вступает в дело алгоритм разрешения столкновений. Если найдена пустая ячейка или просмотрены все ячейки, тогда в таблице нет записи с данным ключом, и поиск закончился неуспешно.

## 9.6.2. Выбор хеш-функции

При выборе хеш-функции мы должны принимать в расчет два принципиальных критерия: желательно, чтобы функция была простой и быстрой в выполнении и при этом она должна давать равномерное распределение реально поступающих ключей по области индексов. Если мы заранее знаем, какие именно будут поступать ключи, тогда возможно разрабо-

метод

тять весьма эффективную хеш-функцию, однако в общем случае мы не можем предугадать состав поступающих ключей. Поэтому обычно хеш-функция берет ключ, разбивает его, смешивает его отдельные части различными способами и получает таким образом индекс, который (схоже с псевдослучайными числами, генерируемыми компьютером) будет распределен равномерно по области индексов.

Заметьте, однако, что в поведении хеш-функции нет ничего случайного. Если функция вызывается более одного раза для одного и того же ключа, она каждый раз даст тот же результат, поэтому ключ всегда извлекается без ошибок.

Именно этот процесс явился причиной такого названия функции<sup>1</sup>, поскольку функция преобразует ключ во что-то, несколько его не напоминающее. В то же время можно надеяться, что любые упорядоченности или регулярности в поступающих ключах будут уничтожены, так что результаты будут всегда распределены равномерно.

Хотя термины «хеш-функция» и «хеширование» имеют столь описательный характер, в некоторых книгах используются вместо них более технические термины *рассеянная память* или *преобразование ключей*.

Мы рассмотрим три метода, которые, при комбинировании их различным образом, дают возможность получить хеш-функцию.

### 1. Усечение

Игнорируйте часть ключа и используйте оставшуюся часть непосредственно в качестве индекса (рассматривая вместо нечисловых ключей их числовые коды). Если, например, ключи представляют собой восьмиразрядные целые, а хеш-таблица имеет 1000 ячеек, тогда первый, второй и пятый разряды справа могут образовать хеш-функцию, в результате действия которой, к примеру, число 62538194 отображается на число 394. Усечение является весьма быстрым методом, но часто приводит к неравномерному распределению ключей в таблице.

### 2. Свертывание

Разделите ключ на несколько частей и для образования индекса скомбинируйте части удобным способом (часто с использованием сложения или умножения). Например, восьмиразрядное целое число может быть разделено на группы по три, три и два разряда, группы сложены вместе и результат в случае необходимости усечен для приведения его в соответствие с областью возможных индексов. В этом случае 62538194 отображается на  $625 + 381 + 94 = 1100$ , что усекается до 100. Поскольку при таком методе вся информация в ключе влияет на значение функции, свертывание часто дает лучшее распределение индексов, чем простое усечение.

### 3. Операции по модулю

Преобразуйте ключ в целое число (используя при желании описанные методы), разделите его на размер области индексов и используйте в качестве результата остаток. Это приводит к использованию оператора

<sup>1</sup> В переводе с английского hash — мелко рубить, крошить. — Прим. перев.

простые делители

языка Pascal **mod**. Распределение индексов, получаемое путем использования остатка, очень сильно зависит от делителя (в данном случае от размера хеш-массива). Если делитель представляет собой степень небольшого числа вроде 2 или 10, тогда многие ключи имеют тенденцию отбрасываться на один и тот же индекс, в то время как другие индексы остаются неиспользованными. Часто (хотя не всегда) наилучшие результаты дает выбор в качестве делителя простого числа, деление на которое обычно приводит к высокой равномерности распределения ключей. (Позже мы увидим, что простые делители также улучшают важный метод разрешения столкновений.) Отсюда следует, что вместо того чтобы выбрать для хеш-таблицы размер 1000, часто лучше остановиться на размере 997 или 1009; выбор же  $2^{10} = 1024$  скорее всего даст плохие результаты. Использование остатка обычно является наилучшим способом завершения вычисления хеш-функции, так как этот способ обеспечивает равномерное распределение и одновременно гарантирует попадание результата в допустимую область.

#### 4. Пример на языке Pascal

В качестве простого примера давайте напишем хеш-функцию на языке Pascal для преобразования ключа, состоящего из восьми алфавитно-цифровых символов, в целое число в диапазоне

0..hashsize – 1

Для этого начнем с образования типа

**type** keytype = **array** [1..8] **of** char;

Теперь мы можем написать простую хеш-функцию:

простая  
хеш-функция

```

function Hash (x: keytype): integer;                                { Хеш-функция }
{ Pre:  x является допустимым значением типа keytype.
  Post: Параметр x был хеширован с возвратом значения между 0 и
        hashsize - 1. }
var
  i: 1..8;
  h: integer;
begin                                                                { функция Hash }
  h := 0;
  for i := 1 to 8 do
    h := 4 * h + ord(x[i]);
  Hash := h mod hashsize
end;                                                                { функция Hash }

```

Мы просто прибавляем целочисленные коды, соответствующие каждому из восьми символов, умножая результат каждый раз на 4. Однако нет оснований считать, что этот метод будет лучше (или хуже), чем любой другой. Мы могли бы, например, вычитать некоторые коды, перемножать их пары, или игнорировать каждый следующий символ. Иногда на практике выясняется, что одна хеш-функция оказывается лучше другой; часто нахождение лучшей функции требует проведения экспериментов.

## 9.6.3. Разрешение конфликтов с помощью открытой адресации

### 1. Линейное зондирование

Простейший метод разрешения столкновений заключается в следующем. Начните с хеш-адреса (ячейки, где произошло столкновение) и выполняйте последовательный поиск по таблице, пока не найдете требуемый ключ или пустую ячейку. При таком методе поиск (зондирование ячеек) осуществляется прямолинейно, отчего он и называется *линейным зондированием*. Таблицу в этом случае следует рассматривать как кольцевую, так что когда достигается последняя ячейка, поиск продолжается с первой ячейки таблицы.

### 2. Кластеризация

Основной недостаток линейного зондирования заключается в том, что когда приблизительно половина таблицы оказывается заполненной, в ней возникает тенденция к *кластеризации*; другими словами, записи начинают заполнять длинные последовательности прилегающих ячеек с промежутками между этими последовательностями. В результате последовательный поиск, требующийся для обнаружения пустой позиции, длится все дольше и дольше. Рассмотрим пример на рис. 9.13, где занятые позиции выделены серым цветом. Предположим, что в массиве имеются  $n$  ячеек и что хеш-функция выбирает любую из них с равной вероятностью  $1/n$ . Начнем с относительно равномерного распределения, показанного в верхней части рис. 9.13. Если при включении новая запись отображается посредством хеширования на ячейку  $b$ , она туда и попадет, но если она отображается на ячейку  $a$  (которая занята), она также попадет в  $b$  (средняя часть рис. 9.13). В результате вероятность записи попасть в ячейку  $b$  удваивается и становится равной  $2/n$ . На следующем шаге попытка включить новую запись в любую из ячеек  $a, b, c$  или  $d$  закончится на ячейке  $d$  (нижняя часть рис. 9.13), и вероятность ее заполнения возрастет до  $4/n$ . После этого ячейка  $e$  будет иметь вероятность заполнения, равную  $5/n$ , и далее по мере последующих включений наиболее вероятным становится удлинение кластера занятых ячеек, начинающегося с ячейки  $a$ . В результате производительность хеш-таблицы начинает ухудшаться, приближаясь к последовательному поиску.

пример  
кластеризации

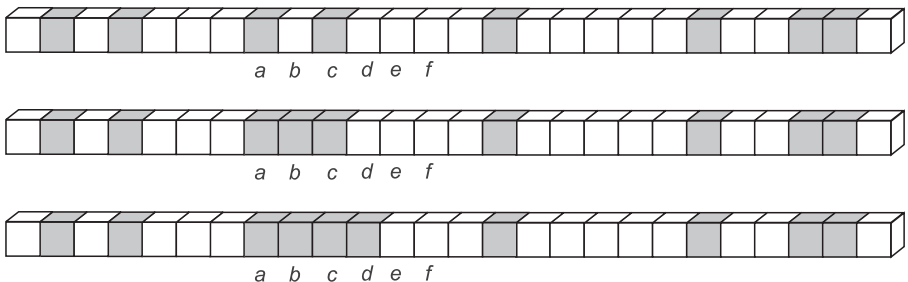


Рис. 9.13. Кластеры в хеш-таблице

нестабильность

Проблема кластеризации фактически сводится к нестабильности: если несколько ключей случайно оказываются близко друг к другу, возникает все большая вероятность того, что к ним будут также присоединяться другие ключи, и распределение будет становиться все более несбалансированным.

### 3. Инкрементные функции

решивание

Если мы хотим избавиться от проблемы кластеризации, мы должны использовать какой-то более изощренный метод просмотра последовательности ячеек с целью нахождения свободной ячейки после возникновения столкновения. Таких методов существует большое количество. Один из них, называемый **рехешированием**, использует вторую хеш-функцию для получения следующей рассматриваемой позиции. Если эта позиция заполнена, тогда для поиска третьей позиции требуется какой-то другой метод и т. д. Если, однако, от первой хеш-функции мы получаем достаточно равномерное распределение, тогда в использовании второй независимой хеш-функции мало выгоды. Будет лучше, если мы найдем какой-то более тонкий метод определения расстояния перехода от первой хеш-позиции и будем применять этот метод независимо от местоположения первой хеш-позиции. Фактически мы хотим разработать инкрементную функцию, которая будет учитывать ключ и число уже выполненных проб и позволит избежать кластеризации.

### 4. Квадратичное зондирование

Если по адресу  $h$  зафиксировано столкновение, в этом методе выполняется зондирование ячеек  $h + 1, h + 4, h + 9, \dots$  т. е. ячеек с адресами  $h + i^2 \pmod{\text{hashsize}}$  для  $i = 1, 2, \dots$ . Другими словами, инкрементная функция представляет собой  $i^2$ .

Квадратичное зондирование существенно снижает кластеризацию, однако совсем не очевидно, что при этом методе будут зондироваться все ячейки в таблице, и в действительности так и происходит. Для некоторых значений  $\text{hashsize}$ , например, степеней 2, функция будет зондировать лишь относительно немногие ячейки в таблице. Если, однако,  $\text{hashsize}$  является простым числом, квадратичное зондирование позволяет использовать половину ячеек в массиве.

доказательство

Для доказательства этого утверждения предположим, что  $\text{hashsize}$  является простым числом. Предположим также, что мы достигаем той же ячейки при попытке  $i$  и при некоторой более поздней попытке, которую можно обозначить как  $i + j$  для некоторого целого  $j > 0$ . Кроме того, будем считать, что  $j$  является наименьшим из таких целых чисел. Тогда значения, вычисленные функцией в попытках  $i$  и  $i + j$ , будут различаться на величину, кратную  $\text{hashsize}$ . Другими словами,

$$h + i^2 \equiv h + (i + j)^2 \pmod{\text{hashsize}}$$

Если это выражение упростить, мы получим:

$$j^2 + 2ij = j(j + 2i) \equiv 0 \pmod{\text{hashsize}}$$

Это последнее выражение означает, что на  $\text{hashsize}$  делится (без остатка) произведение  $j(j + 2i)$ . Единственный способ, которым произведение можно разделить без остатка на простое число — это разделить один



из его сомножителей. Отсюда ли  $j$ , либо  $j + 2i$  без остатка делится на  $\text{hashsize}$ . Если имеет место первый случай, тогда перед дублированием пробы  $i$  будет выполнено  $\text{hashsize}$  проб. (Вспомним, что  $j$  есть такое наименьшее положительное число, что проба  $i + j$  дублирует пробу  $i$ .) Во втором случае, однако, дублирование произойдет раньше, когда  $j = \text{hashsize} - 2i$ , или, если это выражение отрицательно, в точке, соответствующей этому выражению, увеличенному на  $\text{hashsize}$ . Отсюда полное число несовпадающих позиций, которые будут участвовать в пробах, равно в точности

$$(\text{hashsize} + 1) \text{ div } 2$$

Обычно считают, что таблица полна, когда зондировано такое количество позиций в таблице и результаты оказались вполне удовлетворительными.

вычисление

Заметьте, что квадратичное зондирование может быть реализовано без использования умножения. Действительно, после первой пробы в позиции  $x$  инкремент устанавливается равным 1. В каждой последующей пробе инкремент возрастает на 2 и прибавляется к предыдущему значению инкремента. Поскольку

$$1 + 3 + 5 + \dots + (2i - 1) = i^2$$

для всех  $i \geq 1$  (вы можете сами доказать этот факт посредством математической индукции) проба  $i$  обратится к позиции  $x + 1 + 3 + \dots + (2i - 1) = x + i^2$ , что нам и требовалось.

## 5. Инкремент, зависящий от ключа

Вместо того, чтобы определять инкремент в зависимости от числа уже выполненных проб, мы можем вычислять его как какую-либо простую функцию от самого ключа. Например, мы можем усечь ключ до одного символа и использовать этот код в качестве инкремента. На языке Pascal это можно выразить так:

```
increment := ord(k [1])
```

Удачным подходом, когда остаток от деления используется в качестве хеш-функции, является определение инкремента в зависимости от частного, получаемого в той же операции деления. Оптимизирующий компилятор выполнит деление лишь один раз, поэтому вычисления будут быстрыми, а результаты, как правило, вполне удовлетворительными.

В этом методе инкремент, однажды определенный, остается постоянным. Если  $\text{hashsize}$  есть простое число, зондирование просмотрит все элементы массива перед тем, как начнется повторение. Отсюда переполнение не будет возникать до полного заполнения массива.

## 6. Случайное зондирование

Последний метод заключается в использовании для получения инкремента генератора псевдослучайных чисел. Используемый генератор должен всегда образовывать одну и ту же последовательность при условии, что она начинается с одной заправки. (См. приложение В.) Заправка в этом случае может определяться как некоторая функция ключа. Этот метод дает превосходные результаты в плане устранения кластеризации, но скорее всего окажется медленнее других.



## 7. Алгоритм на языке Pascal

Завершая обсуждение открытой адресации, мы продолжим изучение уже начатого нами примера на языке Pascal, в котором используются алфавитно-цифровые ключи типа

**type** keytype = **array** [1..8] **of** char;

Мы организуем хеш-таблицу посредством объявления

объявления

```
const hashsize = 997;           { простое число приемлемой величины }
      hashmax = 996             { должно быть на 1 меньше hashsize }
type hashaddress = 0..hashmax;
      hashtable = array [hashaddress] of tableentry;
var   H: hashtable;
```

инициализация

Хеш-таблица должна быть создана путем определения специального ключа, названного nullkey, который состоит из восьми пробелов, и инициализации полей всех элементов в H, предназначенный для ключей, значением nullkey. Эта задача возлагается на процедуру CreateTable со следующей спецификацией :

<pre><b>procedure</b> CreateTable(<b>var</b> H: hashtable);           { Создать таблицу } <i>предусловие:</i> Отсутствует. <i>постусловие:</i> Хеш-таблица H создана и инициализирована пустыми значениями</pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Должна быть также предусмотрена процедура ClearTable, которая переводит уже созданную ранее таблицу в пустое состояние. Ниже следует спецификация этой функции, но ее код (для случая хеш-таблицы, использующей открытую адресацию) идентичен коду процедуры CreateTable.

<pre><b>procedure</b> ClearTable(<b>var</b> H: hashtable);           { Очистить таблицу } <i>предусловие:</i> Хеш-таблица H уже создана. <i>постусловие:</i> Хеш-таблица H очищена и пуста.</pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Хотя мы и приступили к спецификации операций над хеш-таблицами, мы не будем сейчас разрабатывать полный и обобщенный пакет. Поскольку выбор удачной хеш-функции сильно зависит от рода используемых ключей, операции с хеш-таблицами обычно слишком специфичны для конкретного приложения, чтобы их можно было оформить в виде обобщенного пакета или модуля.

Чтобы показать, как может выглядеть код для остальных процедур, мы продолжим рассмотрение примера уже написанной в разделе 9.6.2 хеш-функции, причем для разрешения столкновений будем использовать квадратичное зондирование. Мы уже показали, что максимальное число проб, выполняемых таким способом, равно  $(\text{hashsize} + 1) \text{ div } 2$ , и для проверки на эту верхнюю границу мы используем счетчик probecount.

При описанных условиях давайте напишем процедуру включения нового элемента `newentry` в хеш-таблицу `H`.

```

procedure InsertTable (var H: hashtable; newentry: tableentry);
    { Включить в таблицу }
{ Pre: Хеш-таблица H уже создана и не полна. H не содержит
    текущего элемента с ключом, равным newentry.
  Post: Элемент newentry включен в таблицу H. }
var probecount,      { счетчик, гарантирующий, что таблица не полна }
    increment: integer;      { инкремент, используемый
                                при квадратичном зондировании }
    probe: hashaddress;      { зондируемая в настоящий момент
                                позиция в H }
begin
    probe := Hash(newentry.key);
    probecount := 0;
    increment := 1;
    while (H[probe].key <> nullkey)      { место пусто? }
    and (H[probe].key <> newentry.key)  { имеется ключ-дубликат? }
    and (probecount <= hashsize div 2) do begin
        { произошло переполнение? }
        probecount := probecount + 1;
        probe := (probe + increment) mod hashsize;
        increment := increment + 2
            { подготовим increment
              для следующего шага цикла }
    end;
    if H[probe].key = nullkey then
        H[probe] := newentry      { включение нового элемента }
    else if H[probe].key = newentry.key then
        Error('Один и тот же ключ не может появляться в хеш-таблице дважды.')
    else
        Error('Хеш-таблица полна; включение невозможно.')
end;
    { процедура InsertTable }

```

Процедура для извлечения записи (при ее наличии) по данному ключу будет иметь такую же форму, и мы ее оставим для упражнений. Процедура извлечения должна возвращать полный состав элемента, отвечающего ключу-мишени. Ее спецификация выглядит следующим образом:

```
procedure RetrieveTable (target: keytype; var H: hashtable;  
                        var found: Boolean; var targetentry: tableentry);  
                        { Извлечь из таблицы }
```

*предусловие:* Хеш-таблица  $H$  уже создана.

*постусловие:* Если элемент в `H` имеет ключ, равный `target`, тогда `found` становится равным `true`, и параметр `targetentry` принимает значение этого элемента. В противном случае `found` становится равным `false`, а `targetentry` не определен.

## 8. Удаления

До сих пор мы ничего не говорили об удалении элементов из хеш-таблицы. На первый взгляд это может показаться простой задачей, требующей всего лишь маркировки удаленного элемента специальным ключом, го-

ворящим о том, что эта ячейка пуста. Однако такой метод работать не будет. Причина здесь в том, что пустая ячейка используется в качестве сигнала для остановки поиска ключа-мишени. Предположим, что до удаления имело место одно или несколько столкновений, и что некоторый элемент, хеш-адрес которого соответствует только что удаленной ячейке, фактически сохранен в каком-то другом месте таблицы. Если мы теперь попытаемся извлечь этот элемент, только что опустошенная ячейка прекратит поиск, и найти элемент будет невозможно, даже если он все еще находится в таблице.

специальный ключ

Одним из методов устранения этой трудности является применение другого специального ключа, который должен быть помещен в удаленную ячейку. Этот специальный ключ будет указывать, что данная ячейка свободна для очередного включения элемента, но поиск любого другого элемента таблицы не должен останавливаться на этой ячейке. С другой стороны, использование этого второго специального ключа в какой-то степени усложнит алгоритм и замедлит его работу. Для тех методов организации хеш-таблиц, которые мы до сих пор исследовали, удаления элементов представляют определенные сложности, и их следует избегать, насколько это возможно.

#### 9.6.4. Разрешение столкновений посредством связанных цепочек

связная память

До сих мы молчаливо предполагали, что при работе с хеш-таблицами мы используем только непрерывную память. Вообще-то выбор именно непрерывной памяти для хранения хеш-таблиц представляется естественным, поскольку нам желательно как можно быстрее обращаться к случайным позициям в таблице, а связанная память не приспособлена для произвольного доступа. Однако отсюда не следует, что связную память нельзя использовать для хранения самих записей. Хеш-таблица может служить в качестве массива указателей на записи, другими словами, в качестве массива связанных списков. Пример такой организации хеш-таблицы приведен на рис. 9.14.

связные цепочки

Традиционно связанные списки, берущие свое начало от хеш-таблицы, называют **цепочками** (или **связными цепочками**), и про сам метод разрешения столкновений говорят, что столкновения разрешаются с помощью цепочек.

##### 1. Преимущества связанных цепочек

экономия пространства памяти

Можно отметить несколько преимуществ такого метода. Первое и наиболее важное в случае записей большого размера заключается в значительной экономии пространства памяти. Поскольку хеш-таблица представляет собой последовательный массив, для того, чтобы не было переполнения, под этот массив еще на этапе компиляции следует выделить достаточный объем памяти. Если сами записи находятся в хеш-таблице, и в таблице много пустых позиций (а их желательно иметь, чтобы избежать накладных расходов на разрешение столкновений), они займут значительный объем памяти, которую можно было бы использовать более эффективно для других целей. Если, с другой стороны, хеш-таблица со-

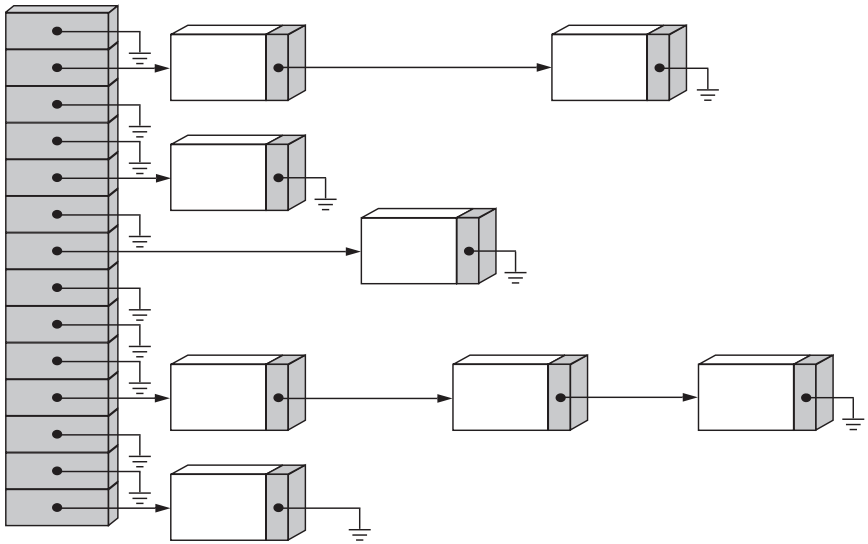


Рис. 9.14. Хеш-таблица со связными цепочками

держит только указатели на записи, указатели размером всего лишь в одно слово каждый, тогда размер хеш-таблицы уменьшается существенным образом (практически в число раз, равное размеру записей) и мы освободим память под сами записи или для других нужд.

разрешение  
столкновений

Другое важное преимущество хранения в хеш-таблице только указателей заключается в том, что в этом случае обработка столкновений становится более простой и эффективной. Нам нужно только добавить к каждой записи поле связи, и организовать все записи с одним хеш-адресом в связный список. При наличии эффективной хеш-функции тот же хеш-адрес будет даваться небольшому числу ключей, поэтому связные списки будут короткими, а их просмотр — быстрым. Кластеризации в этом случае не будет возникать совсем, потому что ключи с различающимися хеш-адресами всегда будут относиться к разным спискам.

переполнения

Третье преимущество определяется тем, что теперь размер хеш-таблицы уже не должен превышать число записей. Если число записей превысит число элементов в таблице, это будет лишь означать, что некоторые из связных списков теперь точно будут содержать более одной записи. Даже если число записей окажется в несколько раз больше длины таблицы, средняя длина связных списков останется небольшой, и последовательный поиск в этих списках будет достаточно эффективен.

удаление

Наконец, удаление записей в хеш-таблице со связными цепочками становится простым и быстрым. Удаление осуществляется в точности так же, как и в случае простых связных списков.

## 2. Недостатки связных цепочек

Недостатки связных цепочек поистине велики. Если вам вдруг покажется, что хеш-таблицы со связными цепочками всегда лучше открытой адресации, подумайте только об одном существенном недостатке: все свя-

использование  
памяти

зи требуют места в памяти. Если записи велики, тогда это место будет незначительным по сравнению с пространством, потребляемым самими записями; если же записи короткие, то будет как раз наоборот.

короткие записи

Предположим, например, что каждая связь требует одного слова, и что сами записи тоже помещаются в одно слово (представляющее собой ключ). Такие приложения встречаются весьма часто; в них хеш-таблица используется для ответов типа «да-нет» на запросы по ключам. Предположим теперь, что мы используем связные цепочки и делаем саму хеш-таблицу совсем маленькой, с числом элементов  $n$ , совпадающим с числом возможных ключей. Тогда вся наша конструкция займет  $3n$  слов:  $n$  для хеш-таблицы,  $n$  для ключей и  $n$  для связей, позволяющих найти следующий узел в каждой цепочке (если он там будет). Поскольку хеш-таблица будет почти заполнена, в ней будет возникать много столкновений, и некоторые цепочки будут содержать по несколько элементов. В результате поиск сильно замедлится. Предположим, с другой стороны, что мы используем открытую адресацию. Если мы выделим под хеш-таблицу те же  $3n$  слов памяти, то при  $n$  элементах она будет заполнена всего лишь на треть, и, следовательно, число столкновений будет относительно невелико, а поиск любого требуемого элемента займет меньше времени.

### 3. Алгоритм на языке Pascal

Хеш-таблица со связными цепочками требует на языке Pascal простого объявления:

```
type hashtable = array [0..hashmax] of list
```

где list относится к связной реализации списков, описанной в главе 6. Для большей согласованности с предыдущими процедурами для хеш-таблиц мы также используем объявление типа listentry = tableentry.

инициализация

Код, требуемый для создания хеш-таблицы, выглядит так:

```
for i := 0 to hashmax do CreateList(H[i]);
```

Для очистки хеш-таблицы со связными цепочками, которая была создана ранее, приходится применять отдельную процедуру. Вспомним, что в случае открытой адресации процедура очистки совпадала с процедурой создания таблицы. Для очистки таблицы мы должны очистить связные списки в каждой позиции таблицы. Эта задача может быть выполнена с помощью процедуры ClearList для связных списков.

Мы даже можем использовать процедуры из пакета обработки списков для доступа к хеш-таблице. Сама хеш-функция не отличается от той, что мы использовали в случае открытой адресации; для извлечения данных мы просто используем связный вариант процедуры SequentialSearch. Существо процедуры извлечения RetrieveTable заключается в предложении

```
SequentialSearch(target, H[Hash(target)], found, position)
```

Детали преобразования этого объявления в полную процедуру мы оставим для упражнений.

Аналогично, существо включения заключается в одном предложении

```
InsertList (newentry, 1, H[newentry.key])
```

Здесь мы выбрали способ включения нового элемента в качестве первого узла списка, поскольку это самый простой путь. Легко заметить, что и включение, и извлечение оказываются проще, чем для варианта открытой адресации, поскольку разрешение столкновений уже не является проблемой и мы можем воспользоваться выполненной ранее работой для связанных списков.

удаление

Удаление из хеш-таблицы со связными цепочками также реализуется значительно проще, чем в случае открытой адресации. Для удаления элемента с данным ключом нам нужно только с помощью последовательного поиска найти местоположение этого элемента в его цепочке в хеш-таблице, после чего мы можем удалить его из связного списка. Спецификация этой процедуры выглядит следующим образом:

```
procedure DeleteTable (target: keytype; var x: tableentry; var H: hashtable);
                                { Удалить из таблицы }
```

*предусловие:* Хеш-таблица со связными цепочками Н уже создана и содержит элемент с ключом, совпадающим с мишенью target.

*постусловие:* Элемент с ключом, совпадающим с мишенью, был удален из  $H$  и возвращен в качестве  $x$ .

Разработку соответствующей процедуры мы оставим для упражнений.

## Упражнения 9.6

- E1.** Докажите методом математической индукции, что  $1 + 3 + 5 + \dots + (2i - 1) = i^2$  для всех целых чисел  $i > 0$ .
- E2.** Напишите на языке Pascal процедуру включения элемента в хеш-таблицу с открытой адресацией с использованием линейного зондирования.
- E3.** Напишите на языке Pascal процедуру извлечения элемента из хеш-таблицы с открытой адресацией с использованием: **(а)** линейного зондирования; **(б)** квадратичного зондирования.
- E4.** В студенческом проекте, где ключами были целые числа, один студент решил, что он может эффективно перемешать ключи, используя тригонометрическую функцию, преобразовав ее в целочисленный индекс. Соответственно, он определил свою хеш-функцию как  $\text{trunc}(\sin(n))$

Что здесь неправильно? Затем этот студент решил заменить  $\sin(n)$  на  $\exp(n)$ . Критически обсудите его решение.

- E5.** Разработайте простую хеш-функцию для отображения трехбуквенных слов на целые числа между 0 и  $n - 1$ , включительно. Найдите значения вашей хеш-функции для слов

PAL LAP PAM MAP PAT PET SET SAT TAT BAT

для  $n = 11, 13, 17, 19$ . Постарайтесь, насколько это возможно, избежать столкновений.

**Е6.** Предположим, что хеш-таблица содержит  $\text{hashsize} = 13$  элементов, индексруемых от 0 до 12, и что на таблицу отображаются следующие ключи:

10 100 32 45 58 126 3 29 200 400 0

- (a) Определите хеш-адреса и найдите, сколько произойдет столкновений, если эти ключи сократить посредством **mod**  $\text{hashsize}$ .
- (b) Определите хеш-адреса и найдите, сколько произойдет столкновений, если эти ключи сначала свернуты путем сложения вместе их цифр (в обычном десятичном представлении), а затем сокращены посредством **mod**  $\text{hashsize}$ .
- (c) Найдите хеш-функцию, которая для этих ключей не приведет к столкновениям. (Хеш-функция, которая не приводит к столкновениям для фиксированного набора ключей, называется *совершенной*.)
- (d) Повторите предыдущие пункты этого упражнения для  $\text{hashsize} = 11$ . (Хеш-функция, которая не приводит к столкновениям для фиксированного набора ключей, полностью заполняющих хеш-таблицу, называется *минимально совершенной*.)

**Е7.** Другой метод разрешения столкновений в случае открытой адресации заключается в создании отдельного массива, называемого *таблицей переполнений*, в которую помещаются все элементы, которые сталкиваются с занятыми уже ячейками. Эти элементы могут включаться либо посредством другой хеш-функции, либо просто помещаться в порядке возникновения, и тогда для их извлечения потребуется последовательный поиск. Обсудите преимущества и недостатки такого метода.

**Е8.** Напишите следующие процедуры для обработки хеш-таблицы со связными цепочками, используя для реализации этих операций процедуры обработки списков из раздела 6.1:

- (a) CreateTable
- (b) ClearTable
- (c) InsertTable
- (d) RetrieveTable
- (e) DeleteTable

**Е9.** Напишите следующие процедуры для обработки хеш-таблицы со связными цепочками. Не используйте пакет обработки списков; впишите инструкции обработки указателей, описывающих связанные цепочки, непосредственно в хеш-таблицу.

- (a) CreateTable
- (b) ClearTable
- (c) InsertTable
- (d) RetrieveTable
- (e) DeleteTable

**Е10.** Напишите алгоритм удаления для хеш-таблицы с открытой адресацией с использованием линейного зондирования посредством второго специального ключа, указывающего на удаленный элемент (см. пункт 8 раздела 9.6.3). Соответственно измените и алгоритмы извлечения и включения.



**E11.** Используя линейное зондирование, возможно удалять элементы, не прибегая ко второму специальному ключу, следующим образом. Отметьте удаленный элемент как пустой. Организуйте поиск следующей пустой позиции. Если в процессе поиска найдется ключ, хеш-адрес которого попадает на первую пустую позицию или до нее, переместите его назад, отметьте его предыдущую позицию как пустую и продолжайте от новой пустой позиции. Напишите алгоритм, реализующий этот метод. Потребуется ли изменение алгоритмов извлечения и включения?

## Программные проекты 9.6

гардероб

**P1.** С помощью хеш-таблицы со связными цепочками реализуйте операции с гардеробом, описанные в разделе 7.3. Вам придется запрограммировать одну дополнительную операцию (не входящую в число стандартных операций для хеш-таблиц): просмотр хеш-таблицы со связными цепочками. Ваша хеш-функция должна будет воспользоваться тем преимуществом, что ключи (т. е. жетоны) уже являются случайными целыми числами. Выбирая размер хеш-таблицы, заметьте, что большая таблица приведет к замедлению операций создания, очистки и просмотра, в то время как таблица небольшого размера увеличит вероятность столкновений и, следовательно, замедлит операции включения, извлечения и удаления. Протестируйте свой пакет с демонстрационной программой гардероба, а затем прогоните программу тестирования времени выполнения с пакетом обработки хеш-таблиц, и сравните результаты с соответствующими данными для последовательного и двоичного поиска.

**P2.** Рассмотрите 35 зарезервированных слов языка Pascal, перечисленных в разделе D.5.1. Рассматривайте эти слова как строки из девяти символов, в которых слова длиной менее девяти символов заполнены с правой стороны пробелами.

(a) Разработайте целочисленную функцию, которая будет возвращать различные значения для всех этих 35 зарезервированных слов. [Вам может оказать помощь написание специальной короткой программы, которая читает слова из файла, применяет к ним разработанную вами функцию и определяет наличие столкновений.]

(b) Найдите наименьшее целое значение `hashsize`, при котором, когда значения вашей функции сокращены посредством `mod hashsize`, все 35 значений не перекрываются.

(c) Модифицируйте должным образом вашу функцию, пока вы не сможете достичь результата `hashsize = 35` для предыдущего пункта. (В этом случае вы найдете *минимальную совершенную* хеш-функцию для 35 зарезервированных слов языка Pascal.)

**P3.** Напишите программу, которая будет читать молекулярную формулу вроде  $\text{H}_2\text{SO}_4$  и выводить молекулярный вес представляемого ею вещества. Ваша программа должна правильно обрабатывать кислотные остатки, помещаемые в круглые скобки, как в случае  $\text{Al}_2(\text{SO}_4)_3$ . [Подсказка: для нахождения молекулярного веса кислотного остатка в

молекулярный вес



скобках используйте рекурсию. *Упрощение:* Вам может помочь заключение в скобки ( ... ) всей формулы. Вам понадобится создать хеш-таблицу атомных весов элементов, индексируемую по их сокращенным обозначениям. Для упрощения можете ограничить таблицу наиболее распространенными элементами. Некоторые элементы имеют однобуквенные сокращения, некоторые — двухбуквенные. Для единообразия можете добавить пробелы к однобуквенным сокращениям.]

## 9.7. Анализ хеширования

### 1. Удивительный результат задачи про дни рождения

Вероятность столкновений при хешировании сводится к хорошо известной математической задаче с игровым содержанием: сколько случайно выбранных людей должно быть в комнате, чтобы возникла некоторая вероятность того, что два человека будут иметь совпадающую дату рождения (месяц и день)? Поскольку (за исключением високосных лет) в году имеется 365 возможных дат рождения, большинство людей полагают, что ответ должен быть порядка сотен, хотя в действительности ответ равен всего лишь 24.

Мы можем определить вероятности для этой задачи, найдя ответ на противоположный вопрос: если в комнате находятся  $m$  случайно выбранных людей, какова вероятность того, что среди них не окажется ни одной пары с одинаковой датой рождения? Начните с любого человека и вычеркните из календаря его дату рождения. Вероятность того, что второй человек будет иметь другую дату рождения, равна  $364/365$ . Вычеркните ее. Вероятность третьему человеку иметь другую дату рождения теперь будет равна  $363/365$ . Продолжая тем же образом, мы видим, что если первые  $m - 1$  людей имеют различающиеся даты рождения, тогда вероятность того, что  $m$ -й человек будет иметь другую дату рождения, равна  $(365 - m + 1)/365$ . Поскольку даты рождения всех людей независимы, вероятности умножаются, и мы получаем для вероятности того, что все люди будут иметь различающиеся даты рождения, выражение

вероятность

$$\frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \times \dots \times \frac{365 - m + 1}{365}$$

Это выражение становится меньше 0.5, как только  $m \geq 24$ .

столкновения  
вероятны

Если теперь вернуться к хешированию, полученный нами удивительный результат говорит нам, что при любой задаче разумного размера мы почти всегда будем иметь какое-то количество столкновений. Наш подход, следовательно, должен заключаться не только в попытке минимизации числа столкновений, но и в обработке неминуемо возникающих столкновений так быстро, как это только возможно.

### 2. Подсчет числа проб

Как и в случае других методов извлечения информации, нам хотелось бы знать, сколько сравнений ключей будет выполняться в среднем в процессе как успешного, так и неуспешного поиска данного ключа-мишени.

Мы будем использовать слово **проба** для операции нахождения одного элемента и сравнения его ключа с мишенью.

коэффициент  
заполнения

Очевидно, что число проб, которые нам потребуются, зависит от того, насколько заполнена таблица. Поэтому (как и в случае методов поиска) мы будем считать  $n$  числом элементов в таблице, а  $t$  (которое совпадает с  $\text{hashsize}$ ) числом позиций в массиве. **Коэффициент заполнения** таблицы составит величину  $\lambda = n/t$ . Таким образом,  $\lambda = 0$  обозначает пустую таблицу; при  $\lambda = 0.5$  таблица заполнена наполовину. В случае открытой адресации  $\lambda$  никогда не может превысить 1, но в случае связанных цепочек на величину  $\lambda$  не накладывается никаких ограничений. Мы рассмотрим цепочки и открытую адресацию отдельно.

### 3. Анализ хеш-таблиц со связными цепочками

В случае хеш-таблицы со связными цепочками, мы перед выполнением проб непосредственно переходим к одному из связанных списков. Предположим, что цепочка, содержащая мишень (если таковая имеется), имеет  $k$  элементов.

неуспешное  
извлечение

В случае неуспешного поиска мишень будет сравниваться со всеми  $k$  ключами. Поскольку элементы распределены равномерно по всем  $t$  спискам (с равной вероятностью обнаружения в любом списке), ожидаемое число элементов в том списке, в котором осуществляется поиск, равно  $\lambda = n/t$ . Отсюда среднее число проб для неуспешного поиска равно  $\lambda$ .

успешное  
извлечение

Теперь рассмотрим успешный поиск. Из анализа последовательного поиска мы знаем, что среднее число сравнений равно  $\frac{1}{2}(k + 1)$ , где  $k$  есть длина цепочки, содержащей мишень. Однако средняя длина этой цепочки уже не равна  $\lambda$ , поскольку мы знаем заранее, что она должна содержать по меньшей мере один узел (мишень). Остальные  $n - 1$  узлов, отличающихся от мишени, распределены равномерно по всем  $t$  цепочкам; отсюда среднее число элементов в цепочке с мишенью равно  $1 + (n - 1)/t$ . Если не рассматривать таблицы тривиально малого размера, то мы можем аппроксимировать  $(n - 1)/t$  выражением  $n/t = \lambda$ . Отсюда среднее число проб для успешного поиска весьма близко к

$$\frac{1}{2}(k + 1) \approx \frac{1}{2}(1 + \lambda + 1) = 1 + \frac{1}{2}\lambda.$$

В итоге:

*Извлечение из хеш-таблицы со связными цепочками с коэффициентом заполнения  $\lambda$  требует приблизительно  $1 + \frac{1}{2}\lambda$  проб в успешном случае и  $\lambda$  проб в неуспешном случае.*

### 4. Анализ открытой адресации

случайные пробы

Для нашего анализа числа проб, выполняемых при использовании открытой адресации, давайте сначала проигнорируем проблему кластеризации, предположив, что не только начальные пробы распределены равномерно, но и после столкновения последующие пробы тоже распределены равномерно по всем позициям в таблице. Фактически мы предположим, что таблица так велика, что все пробы можно рассматривать как независимые события.

Сначала изучим неуспешный поиск. Вероятность того, что первая проба попадет на занятую ячейку, равна коэффициенту заполнения  $\lambda$ . Вероятность того, что проба попадет на пустую ячейку, равна  $1 - \lambda$ . Вероятность того, что неуспешный поиск завершится в точности за две пробы, равна, таким образом,  $\lambda(1 - \lambda)$ , и, аналогично, вероятность того, что в точности  $k$  проб будут выполнены в неуспешном поиске, равна  $\lambda^{k-1}(1 - \lambda)$ . Отсюда ожидаемое число числа проб  $U(\lambda)$  при неуспешном поиске равно

$$U(\lambda) = \sum_{k=1}^{\infty} k \lambda^{k-1} (1 - \lambda).$$

Эта сумма оценена в разделе A.1; мы, таким образом, получаем

$$U(\lambda) = \frac{1}{(1 - \lambda)^2} (1 - \lambda) = \frac{1}{1 - \lambda}.$$

Для того чтобы подсчитать число проб, необходимых для успешного поиска, мы замечаем, что это число будет в точности на 1 больше числа проб, выполненных в неуспешном поиске перед включением элемента. Теперь давайте рассмотрим таблицу начиная с того момента, когда она еще пуста, и в нее последовательно друг за другом включаются элементы. По мере включения этих элементов коэффициент заполнения медленно растет от 0 до своего конечного значения,  $\lambda$ . Представляется разумным аппроксимировать этот пошаговый процесс роста непрерывным ростом и заменить сумму интегралом. Мы заключаем, что среднее число проб в успешном поиске приблизительно равно

$$S(\lambda) = \frac{1}{\lambda} \int_0^{\lambda} U(\mu) d\mu = \frac{1}{\lambda} \ln \frac{1}{1 - \lambda}.$$

В итоге:

*Извлечение из хеш-таблицы с открытой адресацией, случайным зондированием и коэффициентом заполнения 1 требует приблизительно*

$$\frac{1}{\lambda} \ln \frac{1}{1 - \lambda}$$

*проб в успешном случае и  $1/(1 - \lambda)$  проб в неуспешном случае.*

Аналогичные выкладки могут быть сделаны и для случая открытой адресации с линейным зондированием, где мы уже не можем считать, что последовательные пробы являются независимыми. Детали этих выкладок, однако, оказываются значительно более сложными, поэтому мы представляем только результаты. Полный вывод можно найти в ссылках, приведенных в конце этой главы.

*Извлечение из хеш-таблицы с открытой адресацией, линейным зондированием и коэффициентом заполнения 1 требует приблизительно*

$$\frac{1}{2} \left( 1 + \frac{1}{1 - \lambda} \right)$$

*проб в успешном случае и*

$$\frac{1}{2} \left( 1 + \frac{1}{(1 - \lambda)^2} \right)$$

*проб в неуспешном случае.*

неуспешное  
извлечение

успешное  
извлечение

линейное  
зондирование

### 5. Теоретические сравнения

На рис. 9.15 приведены значения полученных нами выражений для различных значений коэффициента заполнения.

<i>Коэффициент заполнения</i>	0.10	0.50	0.80	0.90	0.99	2.00
<i>Успешный поиск, ожидаемое число проб:</i>						
<i>Связные цепочки</i>	1.05	1.25	1.40	1.45	1.50	2.00
<i>Открытая адресация, случайное зондирование</i>	1.05	1.4	2.0	2.6	4.6	—
<i>Открытая адресация, линейное зондирование</i>	1.06	1.5	3.0	5.5	50.5	—
<i>Неуспешный поиск, ожидаемое число проб:</i>						
<i>Связные цепочки</i>	0.10	0.50	0.80	0.90	0.99	2.00
<i>Открытая адресация, случайное зондирование</i>	1.1	2.0	5.0	10.0	100	—
<i>Открытая адресация, линейное зондирование</i>	1.12	2.5	13	50	5000	—

**Рис. 9.15.** Теоретическое сравнение методов хеширования

Мы можем сделать из этой таблицы несколько выводов. Во-первых, ясно, что связные цепочки требуют меньше проб, чем открытая адресация. С другой стороны, просмотр связанных списков обычно осуществляется медленнее доступа к массиву, что может уменьшить это преимущество, особенно, если сравнение ключей можно сделать быстрым. Связные цепочки полностью проявляют свои качества, когда записи имеют большой размер, а сравнение ключей занимает значительное время. Преимущества связанных цепочек также особенно ярко проявляются, когда поиск часто оказывается неуспешным, поскольку при использовании связанных цепочек может быть найден пустой или очень короткий список, и поэтому во многих случаях для установления факта неуспешного поиска вообще не приходится выполнять сравнений ключей.

Для успешного поиска в таблице с открытой адресацией простой метод линейного зондирования лишь незначительно медленнее гораздо более сложного метода разрешения столкновений, по крайней мере, пока таблица не заполнится почти до конца. Однако, для неуспешного поиска кластеризация быстро ухудшает характеристики линейного зондирования, превращая его в длительный последовательный поиск. Мы, таким образом, можем заключить, что если поиск с большой вероятностью будет успешным, а коэффициент заполнения невелик, тогда линейное зондирование дает вполне удовлетворительные результаты, но при других обстоятельствах следует использовать другой метод (например, квадратичное зондирование).

### 6. Эмпирические сравнения

Важно помнить, что результаты, приведенные на рис. 9.15, только приближительны, а также что на практике ничто не является полностью случайным, поэтому мы можем ожидать некоторого расхождения между теоретическими и практическими результатами. Ради сравнения на рис. 9.16 показаны результаты одного эмпирического исследования с 900 ключами, которые являлись псевдослучайными числами в диапазоне от 0 до 1.

Коэффициент заполнения	0.1	0.5	0.8	0.9	0.99	2.0
<i>Успешный поиск, среднее число проб:</i>						
<i>Связные цепочки</i>	1.04	1.2	1.4	1.4	1.5	2.0
<i>Открытая адресация, случайное зондирование</i>	1.04	1.5	2.1	2.7	5.2	—
<i>Открытая адресация, линейное зондирование</i>	1.05	1.6	3.4	6.2	21.3	—
<i>Неуспешный поиск, среднее число проб:</i>						
<i>Связные цепочки</i>	0.10	0.50	0.80	0.90	0.99	2.00
<i>Открытая адресация, случайное зондирование</i>	1.13	2.2	5.2	11.9	126	—
<i>Открытая адресация, линейное зондирование</i>	1.13	2.7	15.4	59.8	430	—

**Рис. 9.16.** Эмпирическое сравнение методов хеширования

выводы

Сравнивая приведенные выше результаты с другими методами извлечения информации, важно отметить, что все эти числа зависят только от коэффициента заполнения, но не от абсолютного числа элементов в таблице. Извлечение из хеш-таблицы с 20000 элементов при 40000 возможных позиций в среднем не требует больше времени, чем извлечение из таблицы с 20 элементами при 40 возможных позициях. При использовании последовательного поиска список в 1000 раз более длинный потребует для поиска в 1000 раз больше времени. В случае двоичного поиска это отношение уменьшается до 10 (более точно до  $\lg 1000$ ), но в любом случае время поиска возрастает с увеличением размеров списка, чего нет в случае хеширования.

Наконец, следует подчеркнуть важность разработки эффективной хеш-функции, которая выполняется быстро и максимально равномерно распределяет ключи. Если хеш-функция выбрана неудачно, производительность хеширования оказывается хуже, чем в случае последовательного поиска.

## Упражнения 9.7

- E1.** Предположим, что каждый элемент в хеш-таблице занимает  $s$  слов памяти (не считая поля для указателя, требуемого в случае использования связанных цепочек), и что в хеш-таблице имеется  $n$  элементов, а сама хеш-таблица содержит в общей сложности  $t$  возможных позиций ( $t$  есть то же самое, что и  $\text{hashsize}$ ).
- (a) Если коэффициент заполнения есть 1, и используется открытая адресация, определите, сколько слов памяти потребуется для хеш-таблицы.
  - (b) При использовании связанных цепочек каждый узел требует  $s + 1$  слов, включая поле для указателя. Сколько всего слов будет использовано для  $n$  узлов?
  - (c) Если коэффициент заполнения есть  $\lambda$ , и используются связанные цепочки, сколько слов памяти потребуется для самой хеш-таблицы? (Вспомните, что в случае связанных цепочек сама хеш-таблица содержит только указатели, каждый из которых требует одного слова.)
  - (d) Добавьте ваши ответы к двум первым пунктам, чтобы найти суммарную потребность в памяти для коэффициента заполнения  $\lambda$  и случая использования связанных цепочек.
  - (e) Если  $s$  мало, тогда открытая адресация требует меньше суммарной памяти для данного  $\lambda$ , но при больших  $s$  связанные цепочки потребляют больше памяти. Найдите для  $s$  значение, при котором оба метода используют один и тот же объем памяти. Ваш ответ будет зависеть от величины  $\lambda$ .
- E2.** Одна из причин, по которой ответ на задачу про даты рождения оказывается неожиданным, заключается в том, что он отличается от ответа на очевидно схожие вопросы. При ответе на следующие вопросы предположите, что в комнате находятся  $n$  людей и не принимайте в расчет високосные годы.
- (a) Какова вероятность того, что кто-то в комнате будет иметь дату рождения в случайный день, полученный путем вытягивания даты из шляпы?
  - (b) Какова вероятность того, что по меньшей мере два человека в комнате будут иметь один и тот же случайно выбранный день?
  - (c) Если мы выбираем одного человека и узнаем его дату рождения, какова вероятность того, что кто-то еще в комнате будет иметь такую же дату рождения?
- E3.** Для хеш-таблицы со связными цепочками сделайте предположение, что разумно говорить о порядке ключей, и что узлы в каждой цепочке упорядочены по ключам. В этом случае поиск может быть завершен, как только он проходит то место, где должен быть расположен ключ (при его наличии). Насколько меньше проб будет выполнено в среднем в неуспешном поиске? А в успешном? Сколько проб требуется в среднем, чтобы включить новый узел на предназначенное ему место? Сравните свои ответы с соответствующими числами, извлеченными из текста для случая неупорядоченных цепочек.

**Е4.** В нашем обсуждении связанных цепочек сама хеш-таблица состояла из одних лишь указателей на списки, образующие цепочки. Один из вариантов этого метода заключается в помещении первого фактического элемента в саму хеш-таблицу. (Пустая позиция описывается недопустимым ключом, как и в случае открытой адресации.) Для данного коэффициента заполнения вычислите экономию памяти при использовании этого метода как функцию от числа слов (за исключением связей) в каждом элементе. (Связь требует одного слова.)

### Программный проект 9.7

**P1.** Получите для своего компьютера таблицу наподобие Рис. 9.16, написав и прогнав тестовые программы с реализацией различных хеш-таблиц при разных значениях коэффициента заполнения.

## 9.8. Заключение: сравнение методов

В этой и предыдущей главах мы рассмотрели четыре совершенно разных метода извлечения информации: последовательный поиск, двоичный поиск, просмотр таблицы и хеширование. Если задать себе вопрос, какой из них лучше, мы сначала должны выбрать критерии сравнения, и эти критерии будут включать в себя как требования, налагаемые применением, так и другие соображения, влияющие на выбор используемых структур данных, поскольку первые два метода применимы только к спискам, а два других — только к таблицам. Во многих приложениях, однако, мы можем сами выбрать способ представления наших структур данных — в виде списков или таблиц.

Что касается скорости и удобства, обычный просмотр последовательных таблиц безусловно лучше других методов, но для многих приложений он неприменим, например, когда желательно иметь список, или набор ключей весьма разрежен. Такой метод также неудобен, если приходится часто выполнять включения и удаления, поскольку эти действия в последовательной памяти могут потребовать перемещения значительных объемов данных.

Какой из оставшихся трех методов лучше, зависит от ваших критериев, например, от формы данных.

Последовательный поиск несомненно самый гибкий из наших методов. Данные могут сохраняться в любом порядке, как в непрерывной, так и в связанной реализации. Двоичный поиск значительно более требователен. Ключи должны быть упорядочены, а данные должны быть организованы так, чтобы допускать произвольный доступ (непрерывная память). Хеширование требует даже большего, своеобразного упорядочения ключей, удачно приспособленного для извлечения из хеш-таблицы, но обычно бесполезного в других случаях. Если данные должны немедленно представляться для обозрения, обычно необходим определенный их порядок, и тогда хеш-таблицы оказываются неудобным инструментом.

Наконец, имеется вопрос неуспешного поиска. Последовательный поиск и хеширование сами по себе ничего не говорят, кроме того, что

выбор структур  
данных

табличный поиск

другие методы



поиск оказался неуспешным. Двоичный поиск позволяет определить, какие данные имеют ключи, ближайшие к мишени, и, таким образом, может предоставить дополнительную полезную информацию.

## 9.9. Приложение: снова игра «Жизнь»

В конце главы 2 мы заметили, что границы, которые мы использовали для массивов в игре Конвея (J. H. Conway) «Жизнь», были весьма жесткими и искусственными. Ячейки жизни должны представлять собой неограниченную решетку. Другими словами, мы в действительности хотели бы иметь объявление языка Pascal

```
type grid = array [integer, integer] of cell;
```

что, разумеется, незаконно. Поскольку в действительности в любой момент времени будет занято лишь определенное количество этих ячеек, мы будем рассматривать решетку для игры «Жизнь» как разреженную таблицу, а в этом случае наиболее привлекательным способом представления решетки является хеш-таблица.

разреженная  
таблица

### 9.9.1. Выбор алгоритма

Перед тем, как приступить к спецификации наших структур данных более конкретно, рассмотрим базовый алгоритм, который мы сможем использовать. Мы уже имели два варианта модели жизни, и вряд ли стоит разрабатывать третий, если мы не будем уверены, что он обеспечит существенное улучшение модели. В первом варианте мы в каждом поколении сканировали всю решетку, выполняя действие, непригодное для разреженных массивов (где оно будет недопустимо медленным). Второй вариант, с другой стороны, был разработан главным образом с целью трактовки решетки как разреженного массива. В этом варианте не происходит явного сканирования всех мертвых ячеек, а вместо для локализации всех ячеек, требующих внимания, используются четыре списка. Вместо того, чтобы начать писать всю программу заново с нуля, давайте посмотрим, до какой степени мы сможем воспользоваться второй программой Life2, если разреженный массив будет представлен в виде хеш-таблицы.

модификация  
программы Life2

### 9.9.2. Спецификация структур данных

Мы уже решили представить наш разреженный массив ячеек в виде хеш-таблицы, однако еще не выбрали способ адресации: открытую адресацию или связные цепочки. Для каждой ячейки мы должны хранить ее состояние (мертвая или живая), число живых соседей, а также (поскольку при использовании хеш-таблицы в ней должен храниться сам ключ) номера строки и столбца ячейки. Поскольку содержимое каждой записи вполне определено — она должна содержать эти четыре элемента, рассуждения об использовании пространства памяти упрощаются. При использовании связных цепочек размер каждой записи придется увеличить

использование  
памяти



на 25 процентов, чтобы хранить необходимый в этом случае указатель, но сама хеш-таблица будет меньше, а коэффициент заполнения можно будет сделать выше, чем при открытой адресации. В случае же выбора открытой адресации записи будут меньше, но в хеш-таблице придется предусмотреть больше свободного места для устранения длинных цепочек поиска и возможного переполнения.

гибкость

Рассмотрев использование памяти, мы должны задать второй вопрос, касающийся гибкости. Нужно ли нам будет удалять ячейки, и, если так, то когда? Мы можем хранить все ячейки до заполнения памяти, после чего удалить все более ненужные ячейки. Но такой подход потребует рехеширования всего массива, что отнимет много времени. Используя связанные цепочки, мы можем легко избавляться от более ненужных нам ячеек, и, таким образом, максимально снизить число ячеек в хеш-таблице.

доступ

Наконец, нам следует рассмотреть требования к доступу к ячейкам. В процессе работы с данной ячейкой нам потребуется локализовать ее восемь соседей, и для этого мы можем использовать хеш-таблицу. Некоторые из этих соседей могут быть включены в четыре списка *maylive*, *maydie*, *newlive* и *newdie*. Когда позже мы будем извлекать ячейку из одного из этих списков, мы опять сможем использовать для нахождения ячейки хеш-таблицу, но это будет повторением уже выполненной работы. Если мы используем связанные цепочки, тогда мы можем добавить ячейку к списку включением либо самой ячейки, либо указателя на нее, вместо того, чтобы, как это мы делали раньше, включать ее координаты. Таким способом мы сможем локализовать ячейку непосредственно без всякого поиска. В то же время, связанные списки помогут нам избежать проблем с возможным переполнением.

спецификация

Таким образом, ради достижения гибкости и повышения скорости работы остановимся на использовании динамического выделения памяти, хеш-таблицы со связными цепочками и связных списков.

реализация списка

Наиболее очевидным способом реализации четырех списков является соединение ячеек в одном списке с помощью полей указателей. В этом случае в запись ячейки потребуется ввести второе поле указателя, поскольку первый указатель используется для цепочек из хеш-таблицы. Однако здесь возникает тонкая проблема. Мы не можем гарантировать того, что одна и та же ячейка не будет одновременно входить в два списка, и при наличии в записи одного поля для указателя элементы двух списков перепутаются. Очевидным способом разрешения этой проблемы является включение в каждую запись ячейки пяти полей указателей, одного для цепочки хеш-таблицы, и четырех для возможного их использования в четырех списках. Это решение приводит к растрате пространства памяти, поскольку многие ячейки не будут входить ни в один из четырех списков, и лишь очень немногие (если вообще таковые найдутся) будут входить хотя бы в два списка.

список  
с косвенными  
связями

Значительно лучшим решением будет поместить в четыре списка указатели на ячейки, а не сами ячейки. Результат такого подхода проиллюстрирован на рис. 9.17. Каждый узел списка теперь содержит два указателя: один на ячейку, и другой на следующий узел этого списка.

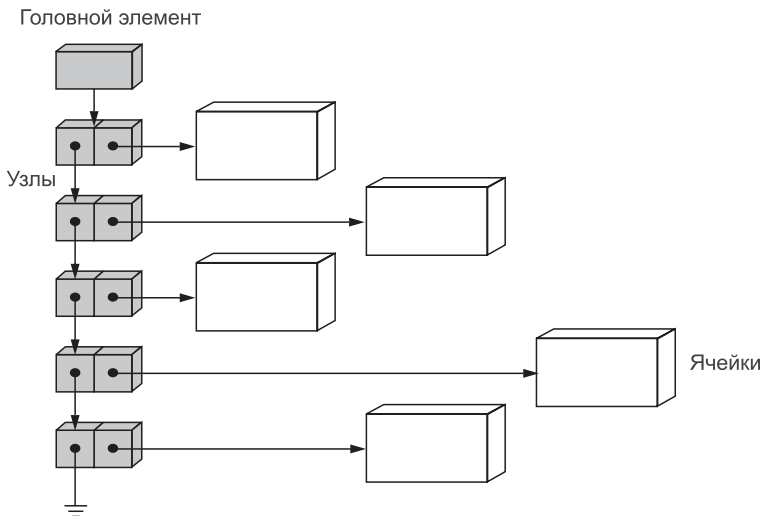


Рис. 9.17. Список с косвенными связями

### 9.9.3. Главная программа

Приняв все эти решения, мы можем теперь связать воедино представление и обозначения наших структур данных, написав главную программу. Поскольку мы следуем методу программы Life2, часть программы, описывающая действия, почти идентична программе Life2.

```

program Life3 (input, output);           { Модель игры Конвея «Жизнь» }
{ Pre: Пользователь предоставляет начальную конфигурацию живых
      ячеек.
  Post: Программа выводит последовательность решеток,
          показывающих изменения в конфигурации живых ячеек согласно
          правилам игры Life.
  Uses: Использует пакеты утилит и обслуживания простых списков;
          процедуры ReadCommand, DoCommand, Introduction, Initialize,
          WriteMap. }

uses Utility, Crt;                       { экранный модуль Turbo Pascal }
const hashsize = 997                      { выберем удобное простое число }
      hashmax = 996;                      { = hashsize - 1 }
      blank = ' ';
      maxlist = 2000;

type state = (alive, dead);               { состояние ячейки }
      count = 0..9;                       { число живых соседей ячейки }
      cellpointer = ↑cell;                 { для образования цепочек из хеш-таблицы }
      cell = record
        row col: integer;                 { номера строки и столбца ячейки }
        status: state;                   { цепочка жива или мертва? }
        neighborcount: count;            { сколько живых соседей имеет
                                          эта ячейка? }
        nextonchain: cellpointer; { цепочки для разрешения столкновений }
{ $I simplist.seg }

```

```

type hashaddress = 0 .. hashmax;
      hashtable = array [hashaddress] of cellpointer; { для хеш-таблицы map }
var map: hashtable;
      newlive, newdie, { ячейки, которые только что ожили или погибли }
      maylive, { кандидаты на возможное оживление в следующем поколении }
      maydie: list; { кандидаты на возможную гибель в следующем поколении }
begin
  Introduction;
  Initialize(newlive, newdie, maylive, maydie, map);
  GoToXY(1,21); { запросы будут внизу экрана }
  write("Продолжить демонстрацию?");
  while UserSaysYes do begin
    TraverseList(maylive, Vivify);
    ClearList(maylive);
    TraverseList(maydie, Kill);
    ClearList(maydie);
    WriteMap(newlive, newdie);
    TraverseList(newlive, AddNeighbors);
    ClearList(newlive);
    TraverseList(newdie, SubtractNeighbors);
    ClearList(newdie);
    writeln; GoToXY(31,21)
  end
end. { главная программа Life3 }

```

## 9.9.4. Процедуры

Давайте теперь напишем несколько процедур, чтобы показать, как выполняется обработка ячеек и списков. Оставшиеся процедуры и функции будут оставлены для упражнений.

### 1. Процедура Vivify

Задача процедуры Vivify заключается в том, чтобы внутри просмотра списка maylive она определяла для каждой ячейки, удовлетворяет ли она условиям стать живой, и если так, то оживляла бы ее и добавляла в список newlive.

```

procedure Vivify (var current: listentry); { Оживить }
{ Pre: Ячейка, на которую указывает current, является кандидатом
      на оживление.
  Post: Проверяет, что ячейка в хеш-таблице map, на которую
          указывает current, удовлетворяет всем условиям на оживление.
          Если это не так, не выполняется никаких изменений. Если это
          так, указатель на ячейку добавляется в список newlive.
          Состояние ячейки устанавливается в значение alive.
  Uses: Использует процедуру AddList, а также список newlive в
          качестве глобальной переменной (побочный эффект). }
begin { процедура Vivify }
  with current↑ do
    if (status = dead) and (neighborcount = 3) then
      begin
        status = alive;
        AddList(current, newlive)
      end
end; { процедура Vivify }

```

Если вы сравните эту процедуру с соответствующей процедурой программы Life2, вы увидите, что единственным изменением является снятие ссылки с `current` в предложении **with** (поскольку списки теперь косвенные) и удалении ссылок на переменную `hedge`, которая (ограда) уже не существует в Life3.

## 2. Процедура AddNeighbors

Задачей этой процедуры является увеличение для каждой ячейки из списка `newlive` счетчика числа соседей на 1 и в добавлении по мере необходимости ячеек к списку `maylive` и `maydie`.

Нахождение соседей для данной ячейки потребует использования хеш-таблицы; мы отложим решение этой задачи ссылкой на процедуру

**procedure** GetCell(row, col: integer; **var** p: cellpointer);

которая вернет указатель на искомую ячейку, создав ячейку, если ее ранее не было в хеш-таблице.

извлечение  
из хеш-таблицы

```

procedure AddNeighbors(var current: listentry);           { Добавить соседей }
{ Pre:  current является указателем на только что ожившую ячейку.
  Post:  Для всех ячеек, соседних с current, увеличены отсчеты числа
          соседей. Если увеличенный отсчет числа соседей делает
          ячейку кандидатом на оживление [соответственно на гибель],
          тогда ячейка добавляется к списку maylive [соответственно
          maydie].
  Uses:  Использует процедуру AddList; изменяет списки maylive
          и maydie как глобальные переменные (побочные эффекты). }
var nbro, nbrcol: integer;                                { индекс цикла для циклов по строкам соседей }
    neighbor: cellpointer;                                  { индекс цикла по столбцам }
begin                                                       { форма записи соседа }
  with current↑ do                                           { процедура AddNeighbors }
    for nbro := row - 1 to row + 1 do
      for nbrcol := col - 1 to col + 1 do
        if (nbro <> row) or (nbrcol <> col) then begin
          { пропустим саму ячейку current }
          GetCell(nbro, nbrcol, neighbor);
          with neighbor↑ do begin
            neighborcount := neighborcount + 1;
            case neighborcount of
              0: writeln('Недопустимый случай в AddNeighbors. ');
              1, 2: ; { никаких действий не требуется }
              3: if status = dead then
                  AddList(neighbor, maylive);
              4: if status = alive then
                  AddList(neighbor, maydie);
              5, 6, 7, 8: ; { изменений нет }
            end
          end
        end
      end
    end
  end;
  { обработка одного соседа }
  { процедура AddNeighbors }

```

И здесь отличия этой процедуры от соответствующей процедуры программы Life2 минимальны, состоя главным образом в использовании процедуры GetCell вместо непосредственного обращения к массиву.

### 3. Обработка хеш-таблицы

Теперь мы можем обратиться к первому принципиальному отличию от Life2, процедуре GetCell, которая явным образом обращается к хеш-таблице. Задача процедуры

GetCell(row, col: integer; var p: cellpointer);

задача

заключается в следующем. Прежде всего она ищет в хеш-таблице ячейку с данными координатами. Если такая ячейка нашлась, процедура возвращает указатель на нее; в противном случае она должна создать новую ячейку, назначить ей данные координаты, инициализировать остальные поля этой ячейки значениями по умолчанию и поместить ее в хеш-таблицу, вернув указатель на новую ячейку.

Этот алгоритм преобразуется в следующую Pascal-процедуру:

```
procedure GetCell (row, col: integer; var p: cellpointer); { Получить ячейку }
{ Pre: Хеш-таблица map уже инициализирована.
  Post: Процедура возвращает указатель p на ячейку [row, col]
        из хеш-таблицы map, если она там присутствует;
        в противном случае ячейка создается, инициализируется
        и помещается в хеш-таблицу map, и возвращается указатель
        на нее. }
var location: hashaddress;           { позиция, возвращенная хеш-функцией }
    found: Boolean;                   { находится ли уже ячейка в хеш-таблице? }
begin                                { процедура GetCell }
  found := false;
  location := Hash(row, col);
  p := map [location];                { p начинает цепочку, содержащую ячейку
                                       с местоположением [row, map] }
  while (not found) and (p <> nil) do { поиск в цепочке требуемой ячейки }
    if (p↑.row = row) and (p↑.col = col) then
      found := true
    else
      p := p↑.nextonchain;
  if not found then begin              { создаем ячейку и включаем ее
                                       в хеш-таблицу map }
    new(p);
    p↑.row := row;
    p↑.col := col;
    p↑.status := dead;
    p↑.neighborcount := 0;
    p↑.nextonchain := map [location]; { протолкнуть в цепочку
                                       в хеш-таблице }
    map [location] := p
  end                                { создание новой ячейки }
end;                                { процедура GetCell }
```

операции  
над указателями

Вы можете заметить, что эта процедура при работе со связными цепочками хеш-таблицы не использует пакет обработки связных списков, а вместо этого непосредственно выполняет операции, пользуясь указателя-

ми. Фундаментальная причина этого заключается в отсутствии в языке Pascal родовых средств: пакет обработки списков уже используется при обработке четырех базовых списков, и, следовательно, не может быть применен к связным цепочкам хеш-таблицы. Проще включить операции с применением указателей в процедуру, чем писать целый новый пакет, имея в виду его использование в единственном приложении.

#### 4. Хеш-функция

Наша хеш-функция будет слегка отличаться от описанных ранее в этой главе в том отношении, что ее аргументы всегда состоят из двух частей (строки и столбцы), поэтому свертывание того или иного рода осуществляется без труда. Перед выбором конкретного варианта давайте рассмотрим специальный случай небольшого массива, функция для которого работает по принципу один к одному и является в точности индексной функцией. Если в каждой строке имеется в точности `maxrow` элементов, то индекс `i`, `j`, отображаясь на

$$i + \text{maxrow} * j$$

помещает прямоугольный массив в непрерывную память, строку за строкой.

Представляется эффективным использовать аналогичное отображение для нашей хеш-функции, только мы заменяем `maxrow` на некоторое удобное число (вроде простого числа), которое максимально увеличит разброс и снизит вероятность столкновений. В результате мы получаем

```
function Hash (row, col: integer): hashaddress;           { Хеш-функция }
{ Pre:   Пара [row, col] определяет местоположение ячейки.
  Post:  Функция возвращает адрес цепочки в хеш-таблице map,
          соответствующей данной ячейке. }
const factor = 101;                                     { выберем удобное простое число }
begin
  Hash := abs(row + factor * col) mod hashsize
end;                                                     { функция Hash }
```

#### 5. Другие подпрограммы

Оставшиеся подпрограммы очень похожи либо на одну из приведенных выше процедур, либо на соответствующие процедуры в программе Life2, и их вполне можно оставить для проектов.

### Программные проекты 9.9

**P1.** Напишите процедуру Kill.

**P2.** Напишите процедуру SubtractNeighbors. Если ячейка мертва и имеет количество соседей, равное 0, она должна быть удалена из хеш-таблицы. Однако вполне может оказаться так, что на эту ячейку указывают несколько указателей, поскольку `maylive` и `maydie` могут содержать избыточные элементы. В этом случае удаление должно быть отложено до полной обработки `maylive` и `maydie`, поскольку в противном случае в них могут остаться указатели на удаленные ячейки. Напишите для этого проекта процедуру SubtractNeighbors, которая будет оставлять такие ячейки в таблице.

**P3.** Модифицируйте программу таким образом, чтобы мертвые ячейки с числом соседей, равным 0, удалялись бы из хеш-таблицы. Для этого создайте связный стек свободного пространства. Модифицируйте `SubtractNeighbors` так, чтобы она удаляла из хеш-таблицы мертвые ячейки с числом соседей, равным 0, и проталкивала бы такие ячейки в стек. Модифицируйте `GetCell` таким образом, чтобы перед созданием новой ячейки она снимала бы ячейку со стека, если, разумеется, стек не пуст.

## Подсказки и ловушки

1. Используйте для своих структур данных принцип нисходящего проектирования точно так же, как вы это делаете применительно к своим алгоритмам. Сначала определите логическую структуру данных, затем постепенно специфицируйте все больше деталей и отложите конкретные детали реализации на возможно больший срок.
2. Перед тем, как приступить к детализации процедур, решите, какие вам потребуются операции над данными и на основании этой информации выберите способ представления данных – *список* или *таблицу*. Необходимость просмотра структуры данных или доступа ко всем данным в предопределенном порядке обычно приводит к выбору списка. Доступ к любому элементу за время  $O(1)$ , как правило, требует выбора таблицы.
3. Принципы разработки и программирования списков описаны в главе 6.
4. При решении вопроса о том, какого рода таблицу использовать, обычный массив, таблицу специальной формы, систему инвертированных таблиц или хеш-таблицу, опирайтесь на логическую структуру данных. Выбирайте простейшую структуру, обеспечивающую требуемые операции и удовлетворяющую требованиям памяти для вашей задачи. Не пишите сложные процедуры, экономящие пространство памяти, если эта память останется неиспользуемой.
5. Структура ваших данных должна помочь вам в выборе способа доступа к таблице с данными: посредством индексной функции или табличного доступа. Везде, где это только возможно, используйте средства, встроенные в ваш язык программирования.
6. При использовании хеш-таблицы выбирайте способ адресации — открытую адресацию или связные цепочки — исходя из природы данных и требуемых операций. Связные цепочки обычно удобнее, если придется удалять данные, если записи относительно велики, или если вы можете столкнуться с проблемой переполнения. Открытая адресация лучше, когда индивидуальные записи невелики и нет опасности переполнения хеш-таблицы.
7. Хеш-функции обычно выбираются индивидуально, исходя из свойств ключей, используемых для доступа к хеш-таблице. Разрабатывая хеш-функцию, старайтесь использовать максимально простые вычисления, обеспечив в то же относительно равномерный разброс ключей



по хеш-таблице. Нет необходимости в использовании для вычислений все части ключа. Разрабатывая ответственные приложения, проведите на своем компьютере эксперименты с несколькими вариантами хеш-функций, оценивая скорость вычислений и равномерность распределения ключей.

8. Вспомните из анализа хеширования, что во всех случаях неминуемо возникает какое-то количество столкновений, поэтому предусматривайте средства разрешения столкновений, даже если ключи распределены по таблице почти равномерно.
9. В случае открытой адресации кластеризация вряд ли превратится в серьезную проблему, пока хеш-таблица не заполнится более чем наполовину. Если таблицу можно сделать в несколько раз больше, чем требуемое для записей пространство, линейное зондирование даст удовлетворительные результаты; в противном случае придется применить более изощренные методы разрешения столкновений. С другой стороны, если таблица оказывается во много раз длиннее того, что нужно в действительности, тогда инициализация всех неиспользуемых ячеек может потребовать значительного времени.

## Обзорные вопросы

- |     |                                                                                                                                      |
|-----|--------------------------------------------------------------------------------------------------------------------------------------|
| 9.1 | 1. С помощью понятия <i>O</i> большого сравните время, требуемое для табличного просмотра и для поиска в списке.                     |
| 9.2 | 2. Что такое <i>развертывание по строкам</i> и <i>развертывание по столбцам</i> ?                                                    |
| 9.3 | 3. Почему <i>рванные таблицы</i> требуют использования таблиц доступа, а не индексных функций?                                       |
|     | 4. С какой целью используются <i>инвертированные таблицы</i> ?                                                                       |
|     | 5. Чем различаются (и различаются ли) в плане конечной цели <i>индексная функция</i> и <i>таблица доступа</i> ?                      |
| 9.4 | 6. Какие операции допустимы для абстрактных таблиц?                                                                                  |
|     | 7. Какие операции обычно оказываются более простыми для списков, чем для таблиц?                                                     |
| 9.5 | 8. Опишите с помощью не более 20 слов, как работает поразрядная сортировка.                                                          |
|     | 9. Почему при использовании поразрядной сортировки ключи обычно разделяются по наименее значимой позиции, а не по наиболее значимой? |
|     | 10. Чем различаются (и различаются ли) в плане конечной цели <i>индексная функция</i> и <i>хеш-функция</i> ?                         |
| 9.6 | 11. Какие цели надлежит преследовать при разработке хеш-функции?                                                                     |
|     | 12. Назовите три метода, часто реализуемые в хеш-функциях.                                                                           |
|     | 13. Что представляет собой <i>кластеризация</i> в хеш-таблице?                                                                       |
|     | 14. Опишите два метода минимизации кластеризации.                                                                                    |



9.7

15. Назовите четыре преимущества хеш-таблиц со связными цепочками по сравнению с открытой адресацией.
16. Назовите одно преимущество открытой адресации по сравнению со связными цепочками.
17. Если хеш-функция назначает 30 ключей случайным позициям в хеш-таблице размером 300 ячеек, насколько вероятно возникновение столкновений?

## Литература для дальнейшего изучения

Основным пособием для настоящей главы является книга [Knuth], том 3 (см. ссылку в главе 3). Хеширование описывается в томе 3, стр. 506-549. Кнут выполнил анализ алгоритмов значительно более подробно, чем это сделали мы, разрабатывая свои алгоритмы на псевдоассемблере и подсчитывая при этом число операций. Он описал все методы, приведенные в настоящей книге, несколько дополнительных методов и много различных вариантов.

Другие подходы, включающие в себя тщательный анализ алгоритмов поиска и хеширования, вместе с рассмотрением других вопросов см. книгу

Lydia I. Kronsjo, *Algorithms: Their Complexity and Efficiency*, John Wiley, New York, 1979.

В следующей книге (стр. 156-185) весьма детально рассматриваются массивы различных видов, индексные функции и таблицы доступа:

C. C. Gotlieb and L. R. Gotlieb, *Data Types and Structures*, Prentice-Hall, Englewood Cliffs, N.J., 1978.

Интересное исследование хеш-функций и выбора используемых констант можно найти в статье

B. J. McKenzie, R. Harries, and T. C. Bell, "Selecting a hashing algorithm", *Software Practice and Experience* 20 (1990), 209-224.

Расширения задачи о датах рождения рассмотрены в статье

M. S. Klamkin and D. J. Newman, *Journal of Combinatorial Theory* 3 (1967), 279-282.

# Двоичные деревья

Связные списки обеспечивают существенное повышение гибкости по сравнению с непрерывным представлением структур данных, но имеют одно слабое место: они являются последовательными списками; другими словами, они организованы таким образом, что перемещение по ним возможно только по одной позиции за раз. В настоящей главе мы преодолеем этот недостаток, рассмотрев представление структур данных в виде деревьев и используя для их реализации указатели и связные списки. Структуры данных, организованные в виде деревьев, окажутся незаменимыми для широкого круга приложений, особенно, связанных с извлечением информации.

## 10.1. Двоичные деревья

Мы уже использовали изображения деревьев для иллюстрации поведения алгоритмов. Мы рисовали деревья сравнений, показывающие операции сравнения ключей в алгоритмах поиска и сортировки; мы рисовали деревья вызовов подпрограмм; и мы рисовали деревья рекурсий. Если, к примеру, мы собираемся приложить двоичный поиск к приведенному ниже списку имен, то последовательность выполняемых операций сравнения будет такой, как изображена на рис. 10.1.

Amy Ann Dot Eva Guy Jan Jim Jon Kay Kim Ron Roy Tim Tom

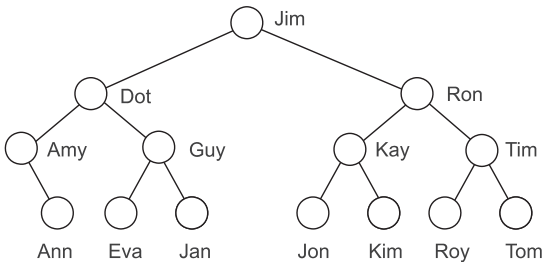


Рис. 10.1. Дерево сравнений для двоичного поиска

### 10.1.1. Определения

При двоичном поиске, выполнив сравнение с ключом, мы сдвигаемся влево или вправо в зависимости от результата сравнения. Таким образом, для возникающей структуры понятия *влево* и *вправо* оказываются существенными. Возможно также, что часть дерева с одной стороны некоторо-

го узла или с обеих сторон ниже него окажется пустой. Так, в примере на рис. 10.1, имя Апу имеет пустое левое поддереву.

Теперь мы можем дать формальное определение новой структуре данных.

#### Определение

**Двоичное дерево** либо пусто, либо состоит из узла, называемого **корнем**, вместе с двумя двоичными деревьями, называемыми **левым поддеревом** и **правым поддеревом** корня.

АТД

Заметьте, что это определение является определением математической структуры. Для спецификации двоичных деревьев как абстрактного типа данных мы должны задать, какие операции будут выполняться над двоичными деревьями. Мы не будем перечислять здесь эти операции; они будут вводиться и разрабатываться по ходу этой главы.

Заметьте, что приведенное определение ничего не говорит о способе реализации двоичных деревьев в памяти. Как мы вскоре увидим, связанная реализация весьма естественна и проста в использовании, но возможны и другие методы. Заметьте, наконец, что в определении нет никаких указаний на ключи или на способ их упорядочения. Двоичные деревья используются с очень многими целями, не только в операциях поиска, поэтому определение дано в общем виде.

Перед тем как приступить к рассмотрению общих свойств двоичных деревьев, давайте вернемся к общему определению и посмотрим, как его рекурсивная природа проявляется при конструировании небольших двоичных деревьев.

небольшие  
двоичные деревья

Нашим первым случаем, не требующим рекурсии, будет пустое двоичное дерево. В случае деревьев другого рода нам вряд ли надо допускать возможность существования пустого дерева, но для двоичных деревьев это оказывается удобным не только при определении понятия дерева, но также и при рассмотрении алгоритмов. Пустое дерево будет у нас служить базовым случаем для рекурсивных алгоритмов и будет определять условие остановки алгоритма.

Единственный способ конструирования двоичного дерева с одним узлом заключается в том, чтобы сделать этот узел корнем дерева, а оба поддерева, и левое, и правое, сделать пустыми. Таким образом, одиночный узел без ветвей представляет собой одно и только одно двоичное дерево с одним узлом.

Если в дереве имеются в точности два узла, один из них будет корнем, а другой — поддеревом. В этом случае либо левое, либо правое поддерево должно быть пустым, а другое будет содержать в точности один узел. Таким образом, имеются два различных дерева с двумя узлами.

Здесь вы можете заметить, что концепция двоичного дерева отличается от нескольких примеров деревьев, с которыми мы встречались раньше, в том отношении, что левая и правая стороны важны для двоичных деревьев. Два двоичных дерева с двумя узлами могут быть нарисованы как



лево и право

причем эти деревья различаются. Мы никогда не будем рисовать какую бы то ни было часть двоичного дерева в виде



поскольку в этом случае нет возможности определить, является ли нижний узел левым или правым дочерним узлом своего родителя.

деревья сравнений

Следует далее отметить, что двоичные деревья не относятся к тому же классу деревьев, что и 2-деревья, с которыми мы познакомились при анализе алгоритмов в главах 7 и 8. Каждый узел в 2-дереве имеет либо 0, либо 2 дочерних узла, и никогда не 1, как это вполне может случиться для двоичного дерева. Понятия лево и право не являются фундаментальными при исследовании свойств деревьев сравнений, в то время как для двоичных деревьев они критически важны.

Теперь рассмотрим случай двоичного дерева с тремя узлами, один из которых будет корнем, а другие распределятся между левым и правым поддеревьями одним из следующих способов:

двоичные деревья  
с тремя узлами

$$2 + 0 \quad 1 + 1 \quad 0 + 2$$

Поскольку существуют лишь два двоичных дерева с двумя узлами и только одно пустое дерево, первый вариант дает нам два двоичных дерева. Третий вариант, аналогично, дает еще два двоичных дерева. Во втором варианте левое и правое поддеревья оба имеют по одному узлу, и, поскольку может быть лишь одно двоичное дерево с одним узлом, второй вариант дает нам одно двоичное дерево. Всего, таким образом, имеются пять двоичных деревьев с тремя узлами; все они изображены на рис. 10.2.

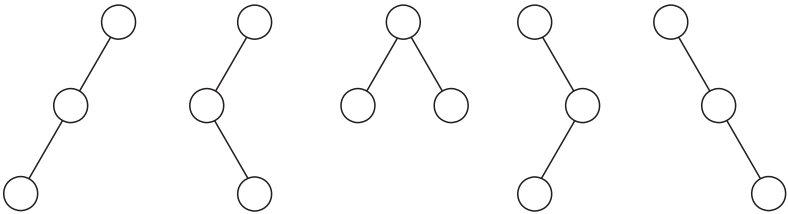


Рис. 10.2. Двоичные деревья с тремя узлами

Шаги, через которые мы прошли, конструируя эти двоичные деревья, типичны и для деревьев большего размера. Мы начинаем с корня и полагаем, что оставшиеся узлы распределены между левым и правым поддеревом. Левое и правое поддеревья тогда оказываются меньшими случаями, для которых мы уже имеем результат из предыдущего анализа.

Перед тем, как перейти к изучению следующего материала, вы должны сделать паузу и сконструировать все четырнадцать двоичных деревьев с четырьмя узлами. Это упражнение поможет вам освоить идею, стоящую за определением двоичных деревьев.

## 10.1.2. Просмотр двоичных деревьев

Одной из наиболее важных операций над двоичными деревьями является их *обход*, или *просмотр*, т. е. продвижение по узлам двоичного дерева с посещением каждого из них по очереди по одному разу. Как и в случае просмотра других структур данных, действие, которое мы будем принимать при *посещении* каждого узла, определяется приложением.

В случае списков узлы располагаются в естественном порядке от первого до последнего, и просмотр осуществляется в том же порядке. В случае деревьев, однако, мы можем обойти все узлы многими различными путями. Когда мы пишем алгоритм просмотра двоичного дерева, нам почти всегда будет желательно иметь одинаковые правила движения для каждого узла, и поэтому мы должны выработать общие правила просмотра.

Для каждого данного узла у нас имеются три задачи, которые мы должны выполнить в некотором порядке: посещение самого узла; просмотр его левого поддерева; просмотр его правого поддерева. Ключевое различие в порядке просмотра заключается в решении, будем ли мы посещать сам узел перед просмотром двух его поддеревьев, между просмотром поддеревьев или после просмотра обоих поддеревьев.

Если мы назовем задачу посещения узла  $V$  (от visit, посещение), задачу просмотра левого поддерева  $L$  (от left, левый) и задачу просмотра правого поддерева  $R$  (от right, правый), то организовать эти действия можно шестью различными способами:

$VLR \quad LVR \quad LRV \quad VRL \quad RVL \quad RLV$

### 1. Стандартный порядок просмотра

По принятому соглашению эти шесть способов просмотра сокращаются до трех путем запрещения просмотра правого поддерева перед левым. Три зеркальных отражения очевидно идентичны. Три способа просмотра, когда левое поддерево просматривается перед правым, имеют специальные имена, которыми мы и будем пользоваться впредь:

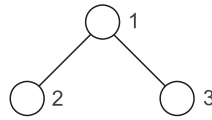
$VLR$	$LVR$	$LRV$
Прямой просмотр	Симметричный просмотр	Обратный просмотр

Эти три имени выбраны в соответствии с положением шага, на котором выполняется посещение узла. При *прямом просмотре* узел посещается до просмотра поддеревьев; при *симметричном просмотре* он посещается между ними; при *обратном просмотре* корень посещается после полного просмотра обоих поддеревьев.

Прямой просмотр еще называют *просмотром в ширину*, а обратный просмотр — *просмотром в глубину*.

### 2. Простые примеры

В качестве первого примера рассмотрим такое двоичное дерево:



прямой просмотр

В случае прямого просмотра прежде всего посещается корень, помеченный на рисунке цифрой 1. Затем просмотр перемещается к левому поддереву. Левое поддерево содержит только один узел, помеченный цифрой 2, и он посещается вторым. Далее прямой просмотр перемещается к правому поддереву корня, где посещается узел, помеченный цифрой 3. В результате при прямом просмотре узлы посещаются в порядке 1, 2, 3.

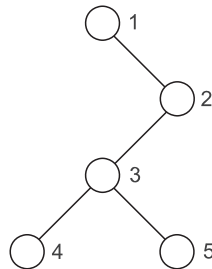
симметричный  
просмотр

В случае симметричного просмотра мы перед посещением корневого узла должны сначала просмотреть левое поддерево. Следовательно, узел, помеченный цифрой 2, посещается первым. Этот узел является единственным в левом поддереве корня, поэтому просмотр перемещается сначала к корню, помеченному цифрой 1, и, наконец, к правому поддереву. Таким образом, при симметричном просмотре узлы посещаются в порядке 2, 1, 3.

обратный  
просмотр

В случае обратного просмотра мы перед посещением узла должны просмотреть и левое, и правое поддерева. Сначала мы идем к левому поддереву, которое содержит единственный узел, помеченный цифрой 2, и посещаем его первым. Затем мы переходим к правому поддереву, посещаем узел 3 и, наконец, посещаем корневой узел, помеченный цифрой 1. Таким образом, при обратном просмотре узлы посещаются в порядке 2, 3, 1.

В качестве второго, слегка более сложного примера, рассмотрим такое двоичное дерево:



прямой просмотр

Сначала рассмотрим прямой просмотр. Корень, помеченный цифрой 1, посещается первым. Далее мы переходим к левому поддереву. Но это поддерево пусто, и просмотр не приводит ни к каким действиям. Далее мы должны просмотреть правое поддерево корня. Это поддерево содержит вершины, помеченные цифрами 2, 3, 4 и 5. Мы должны, следовательно, просмотреть это поддерево, опять используя метод прямого просмотра. Прежде всего мы посещаем корень этого поддерева, помеченный цифрой 2, и затем просматриваем левое поддерево узла 2. На более позднем шаге мы должны будем просмотреть правое поддерево узла 2, которое пусто, так что никаких действий выполняться не будет. Но сначала мы просматриваем левое поддерево с корнем в узле 3. Прямой просмотр

поддерева с корнем 3 требует посещения его узлов в порядке 3, 4, 5. Наконец, мы обрабатываем пустое правое поддерево узла 2. В результате при полном прямом просмотре такого дерева его узлы посещаются в порядке 1, 2, 3, 4, 5.

симметричный  
просмотр

В случае симметричного просмотра мы должны начать с левого поддерева корня, которое пусто. В результате корень, помеченный цифрой 1, будет первым посещенным узлом, а затем мы просматриваем его правое поддерево с корнем в узле 2. Перед посещением узла 2 мы должны просмотреть его левое поддерево, которое имеет корень в узле 3. Симметричный просмотр этого поддерева приведет к посещению его узлов в порядке 4, 3, 5. Наконец, мы посещаем узел 2 и просматриваем его правое поддерево, в котором не выполняется никаких действий, поскольку оно пусто. В результате при симметричном просмотре нашего дерева его узлы посещаются в порядке 1, 4, 3, 5, 2.

обратный  
просмотр

В случае обратного просмотра мы перед посещением самого узла должны просмотреть оба его поддерева, и правое, и левое. Таким образом, мы сначала просматриваем левое пустое поддерево корня 1, затем правое поддерево. При обратном просмотре корень двоичного дерева всегда является последним посещенным узлом. Перед посещением узла 2 мы просматриваем его левое и правое (пустое) поддерева. Обратный просмотр поддерева с корнем в узле 3 определяет порядок посещения его узлов 4, 5, 3. В результате полный обратный просмотр нашего дерева приводит к порядку посещения его узлов 4, 5, 3, 2, 1.

### 3. Деревья выражений

Выбор названий *прямой*, *симметричный* и *обратный* порядок для трех наиболее важных методов просмотра не случаен, но тесно связан с весьма интересным мотивационным примером, именно, примером дерева выражения.

дерево выражения

**Дерево выражения** конструируется из простых операндов и операторов (арифметического или логического) выражения путем размещения простых операндов в качестве листьев двоичного дерева, а операторов в качестве внутренних узлов. Для каждого двуместного (бинарного) оператора левое поддерево содержит все простые операнды и операторы, входящие в левый операнд данного оператора, а правое поддерево содержит весь состав правого операнда.

операторы

Для одноместного (унарного) оператора одно из двух поддеревьев будет пустым. Мы традиционно пишем некоторые из одноместных операторов слева от их операндов; это справедливо для знака '-' (одноместного отрицания) или для стандартных функций вроде  $\log( )$  и  $\cos( )$ . Другие одноместные операторы пишутся справа, например функция факториала  $( )!$  или функция, возводящая число в квадрат  $( )^2$ . Иногда допустимы оба обозначения, как в случае производной, которую можно обозначить  $d/dx$  (т. е. с левой стороны) или  $( )'$  (т. е. с правой стороны) или оператора инкремента  $++$  в языке C (где, впрочем, действия при использовании этого оператора слева и справа будут различными). Если оператор пишется слева, тогда в дереве выражения левое поддерево будет пустым, так что операнды располагаются в дереве с правой стороны опера-

тора, в точности так же, как это выглядит в выражении. Если же оператор пишется справа, тогда его правое поддереве будет пустым, и операнды будут располагаться в левом поддереве оператора.

Деревья выражений для нескольких простых выражений изображены на рис. 10.3. На рис. 10.4 показан слегка более сложный пример квадратичной формулы, где возведение в степень мы обозначили знаком  $\uparrow$ .

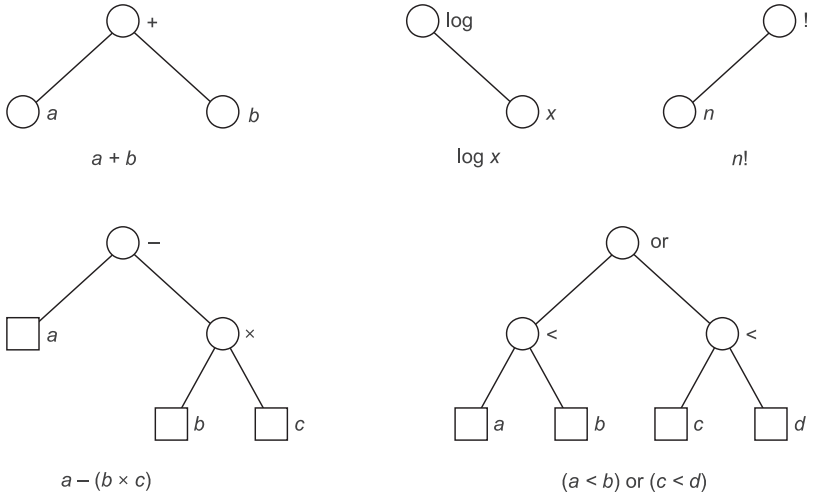


Рис. 10.3. Деревья выражений

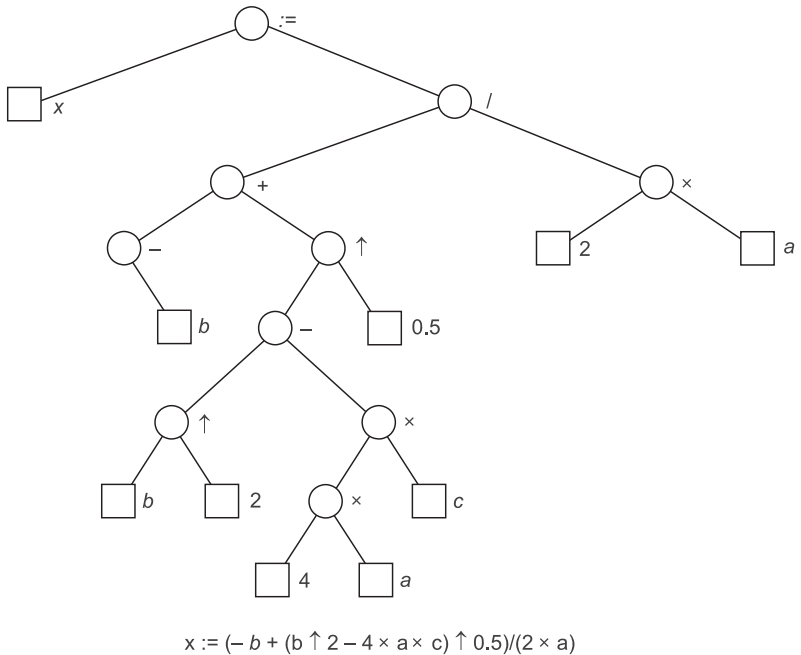


Рис. 10.4. Дерево выражения для квадратичной формулы



Вам следует потратить несколько минут для просмотра каждого из этих деревьев выражений во всех трех порядках: прямом, симметричном и обратном. Чтобы помочь вам в этой работе, результаты каждого такого просмотра приведены на рис. 10.5.

Выражение:	$a + b$	$\log x$	$n!$	$a - (b \times c)$	$(a < b) \text{ or } (c < d)$
Прямой просмотр:	$+ a b$	$\log x$	$! n$	$- a \times b c$	$\text{or} < a b < c d$
Симметричный просмотр:	$a + b$	$\log x$	$n !$	$a - b \times c$	$a < b \text{ or } c < d$
Обратный просмотр:	$a b +$	$x \log$	$n !$	$a b c \times -$	$a b < c d < \text{or}$

Рис. 10.5. Порядок просмотра для деревьев выражений

Названия методов просмотра соответствуют *польским формам* в выражениях: просмотр дерева выражения в прямом порядке отвечает *префиксной форме*, в которой каждый оператор пишется перед его операндом (или операндами); симметричный просмотр отвечает *инфиксной форме* (обычному способу написания выражений); обратный порядок просмотра соответствует *постфиксной форме*, в которой все операторы пишутся после своих операндов. Простое рассуждение убедит вас в правильности сказанного: левое и правое поддеревья каждого узла являются его операндами, а относительная позиция оператора по отношению к его операндам в трех польских формах та же самая, что и относительный порядок посещения компонентов в каждом из трех методов просмотра. Польская нотация является основной темой главы 13.

4. Дерево сравнений

В качестве дальнейшего примера давайте возьмем дерево из 14 имен, изображенное на рис. 10.1 (дерево сравнений для двоичного поиска), и напомним эти имена в порядке, определяемом каждым из методов просмотра.

Прямой просмотр:

Jim Dot Amy Ann Guy Eva Jan Ron Kay Jon Kim Tim Roy Tom

Симметричный просмотр:

Amy Ann Dot Eva Guy Jan Jim Jon Kay Kim Ron Roy Tim Tom

Обратный просмотр:

Ann Amy Eva Jan Guy Dot Jon Kim Kay Roy Tom Tim Ron Jim

Не случайно симметричный просмотр выстраивает имена в алфавитном порядке. Когда мы конструировали дерево сравнений на рис. 10.1, мы перемещались влево, если ключ-мишень предшествовал ключу в рассматриваемом узле, и вправо в обратном случае. В результате двоичное дерево строится так, что все его узлы в левом поддереве данного узла располагаются перед ним в порядке алфавита, а все узлы в его правом поддереве располагаются после него также в порядке алфавита. Поэтому при симметричном просмотре сначала посещаются все узлы перед данным, затем данный узел и, наконец, все более поздние узлы.

В следующем разделе мы сначала рассмотрим двоичные деревья с таким именно свойством. Они называются *двоичными деревьями поиска*, поскольку они весьма полезны и эффективны в задачах, включающих в себя операции поиска.

### 10.1.3. Связная реализация двоичных деревьев

Для двоичного дерева естественна реализация в виде связной памяти. Как и всегда для связных структур, мы будем выделять память для каждого узла динамически, поэтому нам понадобится отдельная переменная-указатель для нахождения дерева. Обычное имя для этой переменной-указателя будет **root** (корень), поскольку она будет указывать на корень дерева. При наличии этой переменной легко распознать пустое двоичное дерево по выполнению условия

root = nil

Для создания нового пустого двоичного дерева достаточно назначить его корневому указателю значение **nil**.

#### 1. Объявления

Каждый узел двоичного дерева (будучи корнем некоторого поддеревья) имеет два поддерева, левое и правое, к которым мы можем обратиться с помощью указателей, объявив

```
type
  treepointer = treenode;
  treenode = record
    entry: treeentry;
    right,
    left: treepointer
  end;
```

Тип **treeentry** зависит от приложения. Эти объявления преобразуют дерево сравнений для 14 имен, изображенное на рис. 10.1, в связанное двоичное дерево, которое показано на рис. 10.6. Легко видеть, что единственным различием дерева сравнений и связанного двоичного дерева является то, что в последнем мы явным образом показали связи **nil**, хотя обычно, рисуя деревья, опускают все пустые поддеревья и идущие к ним ветви.

```
procedure CreateTree (var root: treepointer);           { Создать дерево }
{ Pre:  Предусловия отсутствуют.
  Post: Пустое двоичное дерево поиска создано и на него указывает root. }
begin
  root := nil
end;                                                     { процедура CreateTree }

function TreeEmpty(root: treepointer): Boolean;         { Дерево пусто? }
{ Pre:  Дерево, на которое указывает root, уже создано.
  Post: Функция возвращает значение true или false в зависимости
        от того, пусто ли дерево или нет. }
begin
  TreeEmpty := (root = nil)
end;                                                     { процедура TreeEmpty }
```

корень

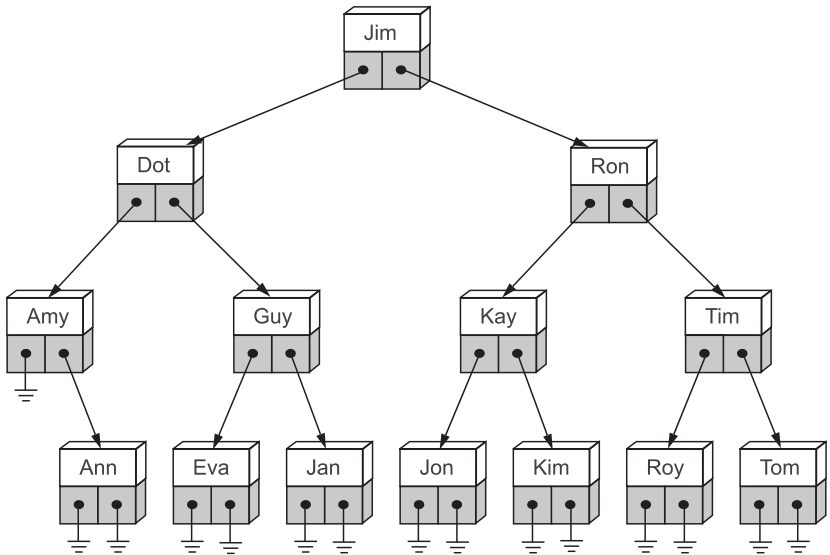


Рис. 10.6. Связное двоичное дерево

## 2. Просмотр

Рекурсия существенно облегчает нам задачу преобразования определений в формальные процедуры, осуществляющие просмотр связанного двоичного дерева каждым из трех рассмотренных нами способов. Как обычно, мы считаем, что переменная *root* является указателем на корень дерева; мы также полагаем, что существует процедура *Visit*, которая выполняет для каждого узла требуемые действия. Как и в случае процедур просмотра, определенных для других структур данных, мы сделаем *Visit* формальным процедурным параметром для процедур просмотра.

```

procedure Preorder (root: treepointer; { Прямой просмотр }
                    procedure Visit(var x: treeentry));
{ Pre: Двоичное дерево, на которое указывает root, уже создано. }
{ Post: Процедура Visit была выполнена над каждым элементом }
в двоичном дереве в прямом порядке. }
begin { процедура Preorder }
    if root <> nil then
        begin
            Visit(root↑.entry);
            Preorder(root↑.left, Visit);
            Preorder(root↑.right, Visit);
        end
    end; { процедура Preorder }

procedure Inorder (root: treepointer; procedure Visit(var x: treeentry));
{ Симметричный просмотр }
{ Pre: Двоичное дерево, на которое указывает root, уже создано. }
{ Post: Процедура Visit была выполнена над каждым элементом }
в двоичном дереве в симметричном порядке. }

```

```

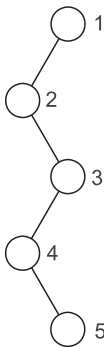
begin                                     { процедура Inorder }
  if root <> nil then
    begin
      Inorder(root↑.left, Visit);
      Visit(root↑.entry);
      Inorder(root↑.right, Visit);
    end
  end;                                   { процедура Inorder }

procedure Postorder (root: treepointer; { Обратный просмотр }
                    procedure Visit(var x: treeentry));
{ Pre: Двоичное дерево, на которое указывает root, уже создано.
  Post: Процедура Visit была выполнена над каждым элементом
        в двоичном дереве в обратном порядке. }
begin                                     { процедура Postorder }
  if root <> nil then
    begin
      Postorder(root↑.left, Visit);
      Postorder(root↑.right, Visit);
      Visit(root↑.entry);
    end
  end;                                   { процедура Postorder }

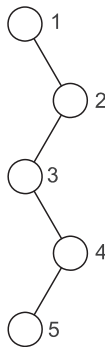
```

### Упражнения 10.1

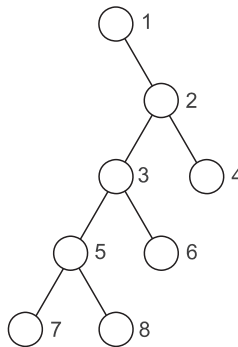
- Е1.** Сконструируйте 15 двоичных деревьев для четырех узлов.  
**Е2.** Определите порядок, в котором будут посещаться узлы изображенных далее двоичных деревьев в случае (1) прямого просмотра, (2) симметричного просмотра, (3) обратного просмотра.



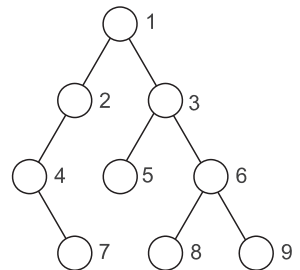
(a)



(b)



(c)



(d)

- Е3.** Нарисуйте дерево выражения для каждого из приведенных ниже выражений и покажите порядок посещения узлов в случае (1) прямого просмотра, (2) симметричного просмотра, (3) обратного просмотра.
- $\log n!$
  - $(a - b) - c$
  - $a - (b - c)$
  - $(a < b) \text{ and } (b < c) \text{ and } (c < d)$

- TreeSize      **E4.** Напишите на языке Pascal функцию TreeSize(root: treepointer): integer, которая пересчитает все узлы в связном двоичном дереве.
- TreeHeight      **E5.** Напишите функцию, которая пересчитает листья (т. е. узлы, у которых оба поддерева пустые) в связном двоичном дереве.
- ClearTree      **E6.** Напишите на языке Pascal функцию TreeHeight(root: treepointer): integer, которая определит высоту связного двоичного дерева в предположении, что высота пустого дерева считается равной 0, а дерево с единственным узлом имеет высоту 1.
- CopyTree      **E7.** Напишите на языке Pascal процедуру ClearTree(var root: treepointer), которая просмотрит двоичное дерево (в любом удобном для вас порядке) и удалит все его узлы.
- двойной просмотр      **E8.** Напишите на языке Pascal процедуру CopyTree(root: treepointer; var newroot: treepointer), которая создаст копию связного двоичного дерева. Процедура должна получить необходимые новые узлы от системы и скопировать поля записей из узлов старого дерева в новое.
- двойной просмотр      **E9.** Напишите процедуру, выполняющую *двойной просмотр* двоичного дерева, понимая под этим следующие действия: для каждого узла процедура посещает этот узел, затем просматривает его левое поддерево (в режиме двойного просмотра), затем посещает узел снова, после чего просматривает его правое поддерево (в режиме двойного просмотра).
- двойной просмотр      **E10.** Для каждого из двоичных деревьев из упражнения E2 определите порядок посещения узлов при использовании смешанного порядка просмотра, определяемого вызываемой процедурой A:
- ```

procedure A(root: treepointer;
procedure Visit(var x: treeentry));
begin
  if root <> nil then begin
    Visit(root↑.entry);
    B((root↑.left, Visit);
    B((root↑.right, Visit);
  end
end;

```

```

procedure B(root: treepointer;
procedure Visit(var x: treeentry));
begin
  if root <> nil then begin
    A((root↑.left, Visit);
    Visit(root↑.entry);
    A((root↑.right, Visit);
  end
end;

```
- вывод двоичного дерева на экран      **E11. (a)** Напишите процедуру, которая выведет на экран ключи из двоичного дерева в *скобочной форме* (key: LT, RT), где key есть ключ корневого узла, LT обозначает левое поддерево корневого узла, выводимое в скобочной форме, а RT обозначает правое поддерево корневого узла, выводимое в скобочной форме.
- вывод двоичного дерева на экран      **(b)** Модифицируйте эту процедуру так, чтобы она для пустого дерева вместо (: .) не выводила бы ничего, а для дерева, состоящего из единственного узла с ключом x выводила бы x вместо (x: .)
- поуровневый просмотр      **E12.** Напишите процедуру, которая обменивает местами все левые и правые поддерева в связном двоичном дереве. Пример этого см. на рис. 10.7.
- поуровневый просмотр      **E13.** Напишите процедуру, которая просматривает двоичное дерево уровень за уровнем. Другими словами, сначала посещается корень, затем ближайшие дочерние от корня узлы, затем «внуки» корня и т. д. [Подсказка: для хранения порядка следования дочерних узлов каждого узла до тех пор, пока не придет пора их посещения, используйте очередь.]

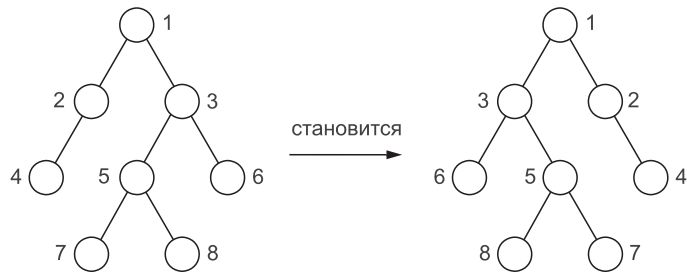


Рис. 10.7. Реверсирование двоичного дерева

ширина

**Е14.** Напишите функцию, которая возвращает ширину связного двоичного дерева, т. е. максимальное число узлов на одном уровне.

дважды связный  
список

**Е15.** Напишите процедуру, которая преобразует двоичное дерево в дважды связный список, в котором узлы имеют порядок симметричного просмотра дерева. При завершении процедуры указатель `root` должен указывать на самый левый узел дважды связного списка; для перемещения по списку следует использовать связи `right` и `left`, которые для двух концов списка должны быть равны `nil`.

последовательности  
просмотр

Для последующих упражнений считайте, что ключи, сохраняемые в узлах двоичных деревьев, не повторяются, однако не предполагается, что деревья являются двоичными деревьями поиска. Другими словами, между порядком ключей и их расположением на дереве нет определенной связи. Если дерево просматривается в определенном порядке, и каждый ключ выводится на экран после посещения его узла, результирующая последовательность носит название последовательности, соответствующей просмотру.

**Е16.** Пусть вам даны две последовательности, которые предположительно соответствуют прямому и симметричному просмотру двоичного дерева. Докажите, что из них можно однозначно сконструировать двоичное дерево.

**Е17.** Докажите (найдя противоположный пример) справедливость или несправедливость аналогичного результата для пары симметричного и обратного просмотра.

**Е18.** Докажите справедливость или несправедливость аналогичного результата для пары прямого и обратного просмотра.

**Е19.** Найдите пару (коротких) последовательностей ключей, которые могли бы соответствовать прямому и симметричному просмотру одного и того же двоичного дерева.

## 10.2. Двоичные деревья поиска

Рассмотрим задачу поиска в связном списке некоторого ключа-мишени. По списку нельзя двигаться иначе чем по одному узлу за раз, поэтому поиск по списку всегда должен осуществляться последовательным образом. Как вы уже знаете, последовательный поиск обычно заметно медленнее двоичного. Таким образом, предположив, что мы можем хранить

дилемма

ключи в некотором порядке, поиск существенно ускорится, если мы будем использовать последовательный список и двоичный поиск. Предположим так же, что нам приходится часто вносить в список изменения, включая новые элементы или удаляя имеющиеся. Тогда работа с последовательным списком будет осуществляться заметно медленнее, чем со связным, потому что при включении элементов в последовательный список и удалении их из него каждый раз требуется перемещать много элементов, в то время как те же операции со связным списком потребуют только перенастройки нескольких указателей.

Ключевая проблема, рассматриваемая в настоящем разделе, состоит в следующем.

*Можем ли мы найти реализацию для упорядоченных списков, в которой мы можем организовать быстрый поиск (как при двоичном поиске в последовательных списках) и при этом иметь возможность быстрых включений и удалений (как в случае связных списков)?*

Двоичные деревья предоставляют превосходное решение этой проблемы. Преобразовав элементы упорядоченного списка в узлы двоичного дерева, мы обнаружим, что найти ключ-мишень можно за  $O(\log n)$  шагов, как и в случае двоичного поиска; при этом мы получим алгоритмы включения и исключения элементов, выполняющие эти операции также за время  $O(\log n)$ .

деревья сравнений

Изучая двоичный поиск, мы рисовали деревья сравнений, показывающие ход двоичного поиска путем перемещения либо влево (если ключ-мишень оказывался меньше ключа текущего узла дерева), либо вправо (если ключ-мишень был больше текущего узла). Пример такого дерева сравнений был изображен на рис. 10.1 и еще раз на рис. 10.6, где он показан в виде связного двоичного дерева. Из этих рисунков должно быть ясно, что для сохранения преимуществ связной памяти и получения скорости двоичного поиска мы должны сохранять узлы в виде двоичного дерева со структурой самого дерева сравнений, в котором связи используются для описания отношений в дереве.

Существенной чертой дерева сравнений является порядок ключей: когда мы перемещаемся к левому поддереву, мы переходим к меньшим ключам, а когда мы перемещаемся к правому поддереву, мы переходим к большим ключам. Это особое условие, накладываемое на ключи узлов двоичного дерева, является существенной частью приведенного ниже важного определения.

#### Определение

**Двоичное дерево поиска** есть двоичное дерево, которое либо пусто, либо содержит в каждом своем узле ключ, удовлетворяющий следующим условиям:

1. Ключ в левом дочернем узле (если он существует) некоторого узла меньше ключа в родительском узле.
2. Ключ в правом дочернем узле (если он существует) некоторого узла больше ключа в родительском узле.
3. Левое и правое поддеревья корневого узла также являются двоичными деревьями поиска.

Два первых свойства описывают порядок ключей по отношению к ключу корневого узла, а третье свойство расширяет этот порядок на все узлы данного дерева; таким образом, мы можем продолжать использовать рекурсивную структуру двоичного дерева. После проверки корневого узла мы перемещаемся либо к его левому, либо к правому поддереву, и эти поддерева опять представляют собой двоичные деревья поиска. Таким образом, мы можем использовать тот же метод снова и снова применительно к этим меньшим деревьям.

Мы написали это определение так, чтобы обеспечить невозможность двум элементам двоичного дерева поиска иметь одинаковые ключи, поскольку ключи левого поддерева должны быть строго меньше ключа корневого узла, а ключи правого поддерева — строго больше. Определение можно изменить, чтобы допустить возможность совпадающих ключей, но в этом случае алгоритм получится несколько более сложным. Итак, мы всегда полагаем:

*Никакие два элемента в двоичном дереве поиска не могут иметь равные ключи.*

Дерево, изображенное на рис. 10.1 и 10.6, автоматически подпадает под определение двоичного дерева поиска, поскольку решение о перемещении влево или вправо основывается на таком же сравнении ключей, которое использовалось в определении дерева поиска.

### 10.2.1. Упорядоченные списки и реализации

три точки зрения

Когда наступает момент формулирования Pascal-процедур для манипулирования двоичными деревьями поиска, мы можем принять по меньшей мере три точки зрения:

- мы можем рассматривать двоичное дерево поиска как новый абстрактный тип данных со своим собственным определением и своими собственными операциями;
- поскольку двоичные деревья поиска являются особого рода двоичными деревьями, мы можем рассматривать их операции как особый вид операций над двоичными деревьями;
- поскольку элементы двоичных деревьев поиска содержат ключи, и поскольку к ним применимы те же правила извлечения информации, что и к упорядоченным спискам, мы можем рассматривать двоичные деревья поиска как новую реализацию абстрактного типа данных *упорядоченный список*.

На практике программисты часто используют все три точки зрения, и мы поступим также. Позже в этом разделе мы покажем, что ключи двоичного дерева поиска можно считать уже упорядоченными, и поэтому удобно рассматривать двоичное дерево поиска как упорядоченный список. Для него основной операцией является просмотр, и мы используем алгоритм просмотра, применимый вообще к двоичным деревьям. Для многих приложений, однако, проще всего рассматривать двоичные деревья поиска как отдельный абстрактный тип данных, и такой подход мы также будем использовать.



Поскольку связная реализация является без всяких сомнений наиболее естественным способом программирования двоичных деревьев поиска, мы часто будем изменять спецификации и параметры таким образом, чтобы получить возможность использовать указатели на узлы вместо непосредственно самих узлов, что характерно для списков и таблиц.

## 1. Объявления

Мы уже ввели Pascal-объявления, позволяющие нам манипулировать узлами двоичного дерева, и мы можем продолжать использовать те же объявления для двоичных деревьев поиска. Элементы информации в любом двоичном дереве объявляются так, чтобы они имели тип `treeentry`, связи имеют тип `treepointer`, а каждый узел, в дополнении к элементу `entry`, содержит левую и правую связь.

Вспомним, что для упорядоченных списков мы использовали `listentry` в качестве имени для записей в списке, причем каждая из записей содержала поле ключа. Если мы хотим, чтобы наши новые объявления были совместимы с теми, что мы давали для упорядоченных списков в предыдущих главах, нам достаточно объявить

`type treeentry = listentry`

и элементы нашего двоичного дерева поиска становятся совместимыми с элементами упорядоченного списка.

Вспомним, что каждый элемент в упорядоченном списке содержит ключ типа `keytype`. Это условие особенно необходимо для элементов типа `treeentry` в двоичном дереве поиска. Мы оставляем тип `keytype` необъявленным, полагая, что `keytype` может быть любого типа, например, числом или символьной строкой, для которого допустима операция сравнения двух ключей с целью определения, какой из них должен быть первым.

Поскольку двоичные деревья поиска являются двоичными деревьями особого рода, мы можем без труда приложить операции, уже заданные для двоичных деревьев вообще, к двоичным деревьям поиска. Эти операции включают в себя `CreateTree`, `ClearTree`, `TreeEmpty`, `TreeSize`, а также процедуры просмотра `Preorder`, `Inorder` и `Postorder`.

## 10.2.2. Поиск по дереву

Первой важной новой операцией для двоичных деревьев поиска является то, от которой происходит их имя: процедура поиска в связном двоичном дереве с целью обнаружения элемента с ключом, совпадающим с заданным ключом-мишенью.

### 1. Метод

Для поиска мишени мы сначала сравниваем ее с ключом в корневом узле дерева. Если ключи совпадают, поиск завершается. Если это не так, мы перемещаемся к левому поддереву или к правому поддереву и повторяем поиск в этом поддереве.

Давайте для примера попробуем найти имя `Kim` в двоичном дереве поиска, изображенном на рис. 10.1 и 10.6. Сначала мы сравниваем `Kim` с ключом `Jim` в корневом узле. Поскольку `Kim` располагается по алфавиту

после Jim, мы перемещаемся вправо и следующей операцией сравниваем Kim с Ron. Поскольку Kim располагается до Ron, мы перемещаемся влево и сравниваем Kim с Kay. Теперь Kim оказывается больше Kay, поэтому мы перемещаемся вправо и находим требуемый ключ.

Какое событие будет служить условием завершения поиска? Очевидно, если мы нашли ключ, процедура должна завершиться успешно. Если же ключ не найден, поиск должен продолжаться, пока мы не дойдем до пустого поддерева. Используя указатель `position` для перемещения по дереву, мы можем также воспользоваться этим указателем для передачи результата поиска назад в вызывающую программу. Итак, мы имеем:

## спецификации

```
procedure TreeSearch (root: treepointer; target: keytype; { Поиск в дереве }  
                      var position: treepointer);
```

*предусловие:* Двоичное дерево поиска, на которое указывает root, уже создано.

*постусловие:* Если какой-либо элемент двоичного дерева поиска имеет ключ, равный ключу `target`, тогда указатель `position` указывает на соответствующий узел; в противном случае `position` становится равным **nil**.

## 2. Рекурсивный вариант

Возможно, проще всего написать процедуру поиска с использованием рекурсии:

## рекурсивный поиск в дереве

```
procedure TreeSearch(root: treepointer; target: keytype;      { Поиск в дереве }
                    var position: treepointer);
```

```
begin                                     { процедура TreeSearch }  
    if root = nil then  
        position := nil                  { мишень находится в позиции root }  
    else if root↑.entry.key = target then  
        position := root;  
    else if target < root↑.entry.key then  
        TreeSearch(root↑.left, target, position)  
    else  
        TreeSearch(root↑.right, target, position)  
end;                                   { процедура TreeSearch }
```

### 3. Отказ от рекурсии

## хвостовая рекурсия

Рекурсия, использованная в этой процедуре, является *хвостовой рекурсией*, так как она образуется выполнением последнего предложения процедуры. Хвостовую рекурсию всегда можно преобразовать в итерацию с помощью цикла. В этой процедуре мы можем написать цикл на месте вложенных предложений **if**, а для продвижения по дереву мы используем переменную **position**. Тогда тело процедуры будет по существу состоять из одного предложения

```
while(position <> nil) and (target <> position↑.entry.key) do  
  if target < position↑.entry.key then position := position↑.left  
  else position := position↑.right;
```

Однако при использовании стандартного языка Pascal это предложение должно быть переписано ради устранения ошибки времени выполнения в случае неуспешного поиска, поскольку в таком варианте процедура будет пытаться искать позицию  $\text{position}\uparrow$ , даже если  $\text{position} = \text{nil}$ . Один из способов избавиться от этой ошибки заключается в использовании булевой переменной, как это реализовано в следующей процедуре.

нерекурсивный  
поиск в дереве

```

procedure TreeSearch(root: treepointer; target: keytype;    { Поиск в дереве }
                    var position: treepointer);
begin                                                         { процедура TreeSearch }
    position := root;
    repeat
        if position = nil then
            finished := true;
        else if position $\uparrow$ .entry.key = target then
            finished := true;
        else begin
            finished := false;
            if target < position $\uparrow$ .entry.key then
                position := position $\uparrow$ .left
            else
                position := position $\uparrow$ .right
            end
        until finished
    end;                                                         { процедура TreeSearch }

```

#### 4. Поведение алгоритма

Вспомним, что действия процедуры TreeSearch основаны на двоичном поиске. Если мы используем двоичный поиск применительно к упорядоченному списку и рисуем его дерево сравнений, мы видим, что при двоичном поиске выполняются в точности те же сравнения, что и в процедуре TreeSearch, если ее приложить к тому же дереву. Из раздела 7.5 мы уже знаем, что при двоичном поиске выполняется  $O(\log n)$  сравнений в списке длиной  $n$ . Это превосходная производительность в сравнении с другими методами, поскольку по мере увеличения  $n$  величина  $\log n$  растет очень медленно.

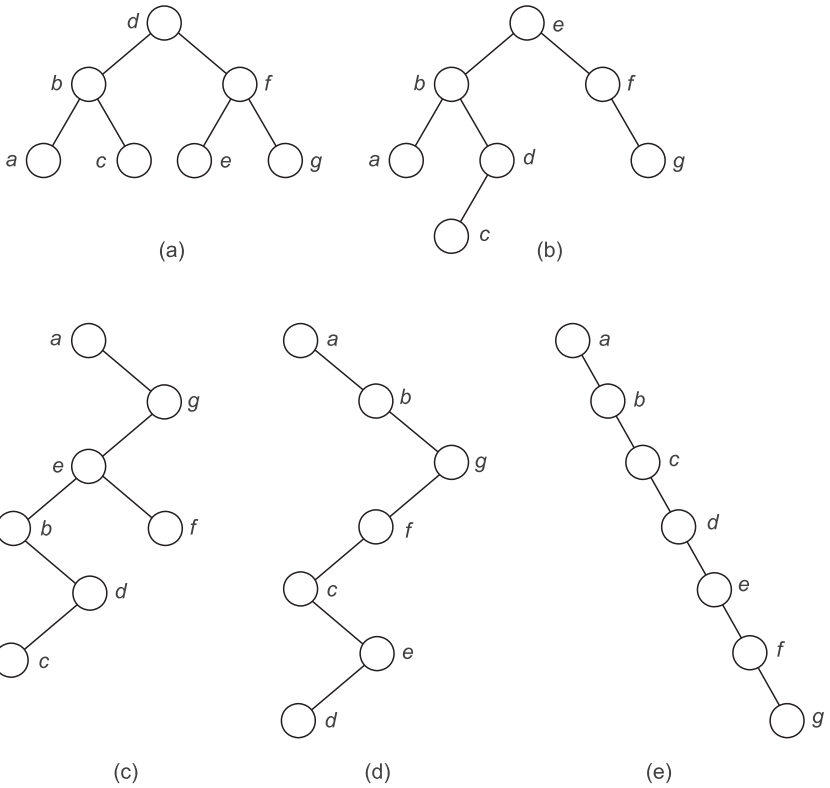
пример

Предположим, например, что мы прикладываем двоичный поиск к списку из семи букв  $a, b, c, d, e, f$  и  $g$ . Результирующее дерево изображено на рис. 10.8, (a). Если к этому дереву применить процедуру TreeSearch, она выполнит такое же число сравнений, как и операция двоичного поиска.

Однако вполне возможно, что те же буквы образуют двоичное дерево поиска совсем другой формы, например, одной из форм, изображенных рис. 10.8, (b)–(c).

оптимальное  
дерево

Дерево, изображенное на рис. 10.8, (a), является наилучшим с точки зрения поиска в нем. Оно максимально «кустисто», т. е. имеет максимально возможное число вершин для его высоты. Число вершин между корневым узлом и мишенью включительно определяет число сравнений, которые должны быть выполнены для того, чтобы найти мишень. Поэтому чем кустистее дерево, тем меньше число требуемых сравнений.



**Рис. 10.8.** Несколько деревьев двоичного поиска с теми же ключами

типичное дерево

плоское дерево

цепочки

Не всегда возможно заранее предсказать, какой формы у нас будет двоичное дерево поиска, и дерево, изображенное на рис. 10.8, (b), оказывается более типичным, чем то, что показано рис. 10.8, (a). При поиске мишени *c* в дереве рис. 10.8, (b) потребуется четыре сравнения, а для той же операции с деревом рис. 10.8, (a) — только три. Дерево рис. 10.8, (b), однако, остается еще достаточно кустистым и его производительность лишь незначительно меньше, чем у дерева рис. 10.8, (a).

На рис. 10.8, (c) дерево стало заметно хуже, и поиск мишени *c* потребует шести сравнений. На рис. 10.8, (d) и (e) дерево вырождается просто в цепочку. Процедура `TreeSearch`, будучи применена к такого рода цепочкам, не может действовать иначе, чем просто перемещаться по списку элемент за элементом. Другими словами, действия процедуры `TreeSearch` при обработке таких цепочек вырождаются в последовательный поиск. В своем худшем случае для дерева с *n* узлами процедуре `TreeSearch` может потребоваться для нахождения мишени вплоть до *n* сравнений.

На практике, если ключи располагаются в двоичном дереве поиска в случайном порядке, то крайне мало вероятно, что дерево выродится во что-то, напоминающее рис. 10.8, (d) и (e). Значительно более вероятно



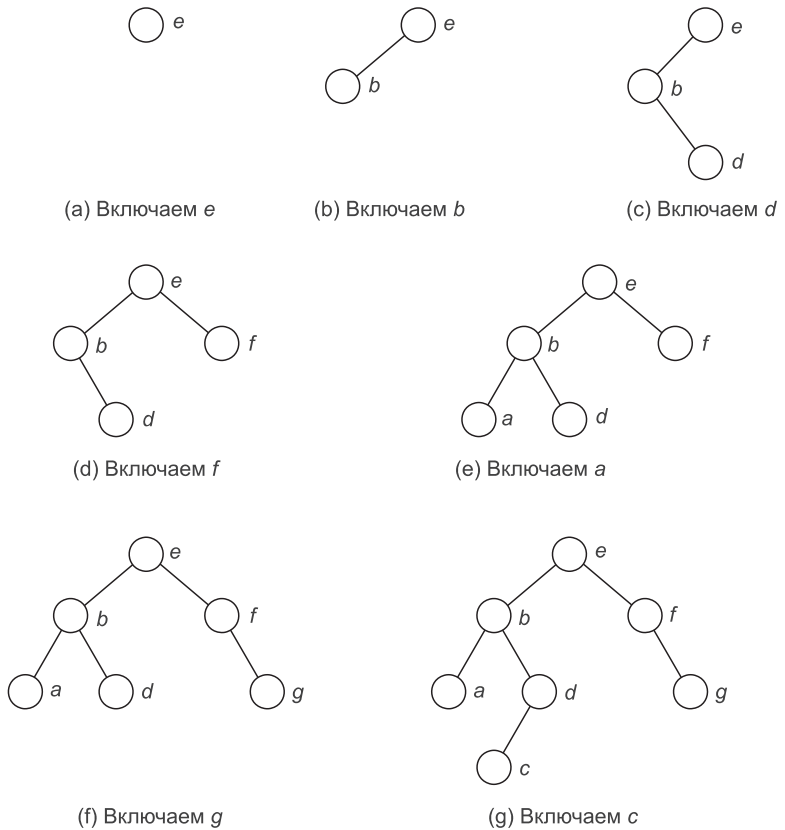


Рис. 10.9. Последовательные включения в двоичное дерево поиска

другой порядок  
включения,  
то же дерево

рис. 10.9, может быть получено включением ключей в любой из приведенных ниже последовательностей

$e, f, g, b, a, d, c$  или  $e, b, d, c, a, f, g$

а также и нескольких других.

естественный  
порядок

Один случай имеет особое значение. Предположим, что ключи включаются в первоначально пустое дерево в их естественном порядке  $a, b, \dots, g$ . Тогда  $a$  будет содержаться в корневом узле,  $b$  войдет в его правый дочерний узел,  $c$  поместится справа от  $a$  и  $b$  и т. д. Включения образуют цепочку для двоичного дерева поиска, как это показано на рис. 10.8, (е). Такая цепочка, как мы уже видели, крайне неэффективна для поиска. Отсюда мы заключаем:

*Если ключи, подлежащие включению в пустое двоичное дерево поиска, расположены в их естественном порядке, тогда процедура InsertTree образует дерево, вырождающееся в неэффективную цепочку. InsertTree никогда не следует использовать для включения уже упорядоченных ключей.*

Эта рекомендация сохраняет свое значение и для случая ключей, упорядоченных в обратном порядке, а также для ключей, не вполне, но почти упорядоченных.

### 3. Метод

От рассмотренного нами примера до общего метода включения новых узлов в двоичное дерево поиска всего один шаг.

Первый случай, включение узла в пустое дерево, не представляет сложностей. Мы просто должны сделать так, чтобы переменная `root` указывала на новый узел. Если дерево не пусто, мы должны сравнить ключ с тем, который входит в корневой узел. Если наш ключ окажется меньше корневого, новый узел должен быть включен в левое поддереву; если он окажется больше — в правое. Если ключи совпадают, мы принимаем приглашение о включении дублирующего ключа в правое поддереву.

Заметьте, что мы описали включение с помощью рекурсии. После сравнения нового ключа с тем, который находится в корневом узле, мы используем в точности тот же метод включения либо в левое, либо в правое поддереву, который ранее мы использовали применительно к корневому узлу.

### 4. Рекурсивная процедура

Рассмотрев эти основные принципы, мы можем написать нашу процедуру, воспользовавшись объявлениями, приведенными в начале этого раздела.

рекурсивное  
включение

```
procedure InsertTree (var root: treepointer;           { Включить в дерево }
                      newnode: treepointer);
{ Pre:   Двоичное дерево поиска, на которое указывает root, уже
          создано. Переменная newnode указывает на созданный узел,
          который содержит в своем элементе ключ.
  Post:  Узел newnode↑ включен в дерево таким образом, что свойства
          двоичного дерева поиска сохранились. }
begin                                           { процедура InsertTree }
  if root = nil then                             { включение в пустое дерево }
    begin
      root := newnode;
      root↑.left := nil;
      root↑.right := nil;
    end
  else if newnode↑.entry.key < root↑.entry.key then
    InsertTree(root↑.left, newnode)
  else
    InsertTree(root↑.right, newnode)
  end;                                           { процедура InsertTree }
```

Обратите внимание на то, как эта процедура обрабатывает дублирующие ключи: она включает новый ключ, дублирующий уже имеющийся, с правой стороны старого элемента. Используя такой прием, мы обеспечиваем, что при последующем просмотре дерева (а мы просматриваем левые поддеревья перед правыми) элементы с дублирующими ключами будут посещаться в том же порядке, в каком они были включены в дерево. Поскольку сравнения ключей выполняются в том же порядке, что и в

TreeSearch, процедура поиска всегда найдет сначала тот элемент с данным ключом, который был включен в дерево первым.

Использование рекурсии в процедуре InsertTree не является существенным требованием, поскольку это хвостовая рекурсия. Мы оставляем преобразование процедуры InsertTree в нерекурсивную форму для упражнений.

Что касается производительности, InsertTree выполняет те же сравнения ключей, что и TreeSearch при поиске того жеключаемого ключа. InsertTree также изменяет несколько указателей, но не перемещает элементы и не выполняет каких-либо операций, требующих значительного времени или пространства памяти. Поэтому производительность InsertTree будет практически такой же, как и у TreeSearch:

*Процедура InsertTree обычно включает новый узел в двоичное дерево поиска со случайным порядком ключей и  $n$  узлами за  $O(\log n)$  шагов. Возможно, хотя и крайне мало вероятно, что дерево со случайным порядком может вырождаться так, что включение потребует до  $n$  шагов. Если, однако, упорядоченные заранее ключи включаются в пустое дерево, будет иметь место такое вырождение.*

## 10.2.4. Древоидная сортировка

Вспомним из нашего обсуждения процесса просмотра двоичных деревьев, что когда мы просматриваем двоичное дерево поиска методом симметричного просмотра, ключи будут встречаться упорядоченным образом. Причина этого кроется в том, что ключи слева от данного ключа всегда предшествуют ему (или равны), а все ключи справа следуют за ним (или равны). По рекурсии это обстоятельство повторяется снова и снова, пока мы не дойдем до поддерева с одним единственным ключом. В результате симметричный просмотр всегда выдает ключи упорядоченным образом.

### 1. Метод

Высказанное наблюдение лежит в основе интересного метода сортировки, носящего название **древовидной сортировки**. Мы попросту берем элементы, подлежащие упорядочению, используем процедуру InsertTree для включения их в двоичное дерево поиска и для их упорядочения применяем симметричный просмотр.

### 2. Сравнение с алгоритмом быстрой сортировки

Рассмотрим кратко, какие сравнения ключей выполняются алгоритмом древовидной сортировки. Первый узел помещается в корень двоичного дерева поиска без всякого сравнения ключей. При поступлении нового узла его ключ сначала сравнивается с ключом корневого узла, а затем мы переходим либо к левому, либо к правому поддереву. Обратите внимание на сходство с алгоритмом быстрой сортировки, в котором на первом шаге каждый ключ сравнивается с первым опорным ключом, а затем помещается либо в левый, либо в правый подсписок. При древовидной сортировке, однако, каждый приходящий узел поступает в свою окончательную позицию в связанной структуре. Второй узел становится корнем



либо левого, либо правого поддеревя (в зависимости от результата сравнения его ключа с корневым ключом). Далее все ключи, поступающие в то же поддерево, сравниваются с этим вторым ключом. Аналогично в алгоритме быстрой сортировки все ключи одного подподписки сравниваются со вторым опорным ключом, тем, который принадлежит этому подподписку. Продолжая этот процесс, мы можем сформулировать следующее наблюдение.

#### Теорема 10.1

*Древовидная сортировка выполняет в точности то же число сравнений ключей, что и алгоритм быстрой сортировки, если в качестве опорного ключа каждого подподписки выбирается его первый ключ.*

преимущества

Как мы знаем, быстрая сортировка обычно оказывается превосходным методом. В среднем только сортировка слиянием среди прочих изученных нами методов выполняет меньшее число сравнений ключей. Таким образом, мы можем ожидать, что в среднем древовидная сортировка будет превосходным методом сортировки в плане числа сравнения ключей. Фактически, вспомнив раздел 8.8.4, мы можем заключить:

#### Следствие 10.2

*В среднем случае для случайно упорядоченного списка длиной  $n$  древовидная сортировка выполняет*

$$2n \ln n + O(n) \approx 1.39n \lg n + O(n)$$

*сравнений ключей.*

Древовидная сортировка имеет одно преимущество перед быстрой сортировкой. Для быстрой сортировки требуется доступ ко всем упорядочиваемым элементам по ходу процесса сортировки. При древовидной сортировке в начале процесса нет необходимости иметь доступ ко всем узлам; узлы включаются в дерево по одному по мере их поступления. В результате древовидная сортировка оказывается предпочтительной для тех приложений, в которых узлы поступают последовательно во времени. Основное преимущество древовидной сортировки заключается в том, что ее дерево поиска остается доступным для последующих включений и удалений, и, следовательно, последующий просмотр дерева требует логарифмического времени, в то время как все предыдущие методы сортировки либо требовали непрерывных списков, для которых включения и удаления затруднены, либо образовывали связные списки, для которых возможен только последовательный просмотр.

недостатки

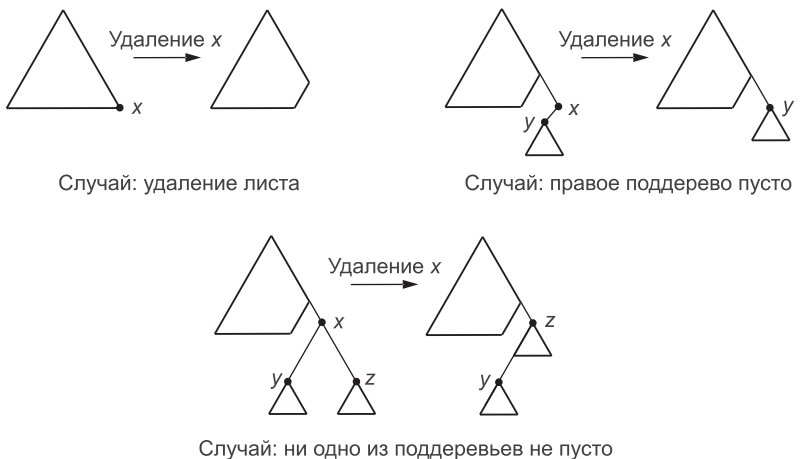
Основной недостаток древовидной сортировки следует из теоремы 10.1. Быстрая сортировка имеет очень плохую производительность в своем худшем случае, и, хотя при аккуратном выборе опорных ключей этот случай оказывается крайне маловероятным, выбор в качестве опорного ключа первого ключа в каждом подподписке приводит к худшему случаю, если ключи уже упорядочены. Если ключи, поставляемые древовидной сортировке, уже упорядочены, тогда древовидная сортировка будет просто катастрофой — образуемое ею дерево поиска вырождается в цепочку. Древовидную сортировку никогда не следует использовать, если ключи уже полностью или хотя бы почти упорядочены.

Имеются и другие ограничения древовидной сортировки, которые не относятся в равной степени ко всем связным структурам. Для небольших задач с небольшими элементами последовательная память обычно дает лучшие результаты, но для больших задач с объемными записями связная память проявляет все свои преимущества.

### 10.2.5. Удаление из двоичного дерева поиска

Обсуждая древовидную сортировку, мы упомянули возможность внесения изменений в двоичное дерево поиска в качестве преимущества этого метода. Мы уже получили алгоритм, который включает новый узел в двоичное дерево поиска, и этот алгоритм можно использовать для обновления дерева так же просто, как и для его формирования с самого начала. Однако мы еще не обсуждали, каким образом можно удалять узлы из дерева. Если удаляемый узел представляет собой лист, процесс удаления прост: мы заменяем связь с удаляемым узлом на `nil`. Процесс остается простым, если удаляемый узел имеет только одно поддереву: мы настраиваем связь от родительского по отношению к удаляемому узлу так, чтобы она указывала на его поддереву.

Однако, если удаляемый узел имеет и левое, и правое непустые поддерева, проблема становится более сложной. На какое из поддеревьев теперь должна указывать связь родительского по отношению к удаляемому узлу? И что делать со вторым поддеревом? Эта проблема иллюстрируется на рис. 10.10 вместе с одним из возможных решений. (В упражнениях очерчивается другое, иногда лучшее, решение.) Операция обновления дерева заключается в следующем: мы присоединяем правое поддерево на место удаленного узла, а затем подвешиваем левое поддерево к соответствующему узлу правого поддерева.



**Рис. 10.10.** Удаление узла из двоичного дерева поиска

К какому узлу правого поддерева следует подсоединить то, что ранее было левым поддеревом? Поскольку любой ключ в левом поддереве предшествует любому ключу в правом поддереве, оно должна располагаться в максимально возможной левой позиции; эту точку можно найти, анализируя левые ветви до тех пор, пока не будет найдено пустое левое поддерево.

Теперь мы можем разработать процедуру, реализующую этот план. В качестве параметра вызова эта процедура будет использовать указатель pointer на удаляемый узел. Поскольку нашей целью является обновление двоичного дерева поиска, мы должны предположить, что соответствующий фактический параметр представляет собой одну из связей дерева, а не просто копию, поскольку в случае изменения копии структура самого дерева после удаления не изменится. Другими словами, если удаляется узел слева от  $x\uparrow$ , вызов процедуры должен быть таким:

DeleteNodeTree( $x\uparrow$ .left)

если же удаляется корень, вызов будет таким:

DeleteNodeTree(root)

С другой стороны, следующий вызов не приведет к желаемому результату:

$y := x\uparrow$ .left; DeleteNodeTree(y)

удаление

```

procedure DeleteNodeTree (var position: treepointer);
    { Удалить узел дерева }
{ Pre: Параметр position является действительной связью (не копией)
    в двоичном дереве поиска, причем position не имеет значения nil.
Post: Узел position $\uparrow$  удален из двоичного дерева поиска, и
    результирующее меньшее по размеру дерево сохранило
    свойства, требуемые для двоичного дерева поиска. }
var temp: treepointer;
    { используется для поиска места,
    куда можно привесить меньшее дерево }
begin
    { процедура DeleteNodeTree }
    if position = nil then
        Error('Попытка удалить несуществующий узел из двоичного дерева поиска')
    else if position $\uparrow$ .right = nil then
        begin
            { переместить левое поддерево на место position $\uparrow$  }
            temp := position;
            position := position $\uparrow$ .left;
            dispose(temp)
        end
    else if position $\uparrow$ .left = nil then
        begin
            { переместить правое поддерево на место position $\uparrow$  }
            temp := position;
            position := position $\uparrow$ .right;
            dispose(temp)
        end
    else begin
        { ни одно из поддеревьев не пусто }
        temp := position $\uparrow$ .right;
        { сдвинемся вправо, а затем
        насколько возможно влево }
    end

```

одно поддерево  
пусто

переместим  
поддерево

```

while temp↑.left <> nil do
    temp := temp↑.left;
    temp↑.left := position↑.left;
    temp := position;
    position := position↑.right;
    dispose(temp)
end
end;

```

{ процедура DeleteNodeTree }

Вам следует выполнить трассировку этой процедуры, чтобы удостовериться в том, что все указатели обновляются должным образом, особенно в случае, когда ни одно из поддеревьев не пусто. Обратите внимание на шаги, требуемые для того, чтобы цикл остановился на узле с пустым левым поддеревом, но не на самом пустом поддереве.

Приведенная процедура далека от оптимальной в том отношении, что она может существенно увеличить высоту дерева. Два таких примера показаны на рис. 10.11. Когда из этих двух деревьев удаляются корни, у верхнего дерева уменьшается высота, однако у нижнего высота увеличивается. Это может существенно увеличить время, требуемое для последующего поиска, даже если полный размер дерева уменьшается. Более того, включения и удаления часто имеют тенденцию совершаться упорядоченным образом, что будет еще более удлинять двоичное дерево поиска. Поэтому, с целью более оптимального использования двоичных деревьев поиска, мы должны разработать методы, которые приведут к большей сбалансированности левого и правого поддеревьев. Позже в этой главе мы рассмотрим такие методы.

сбалансирован-  
ность

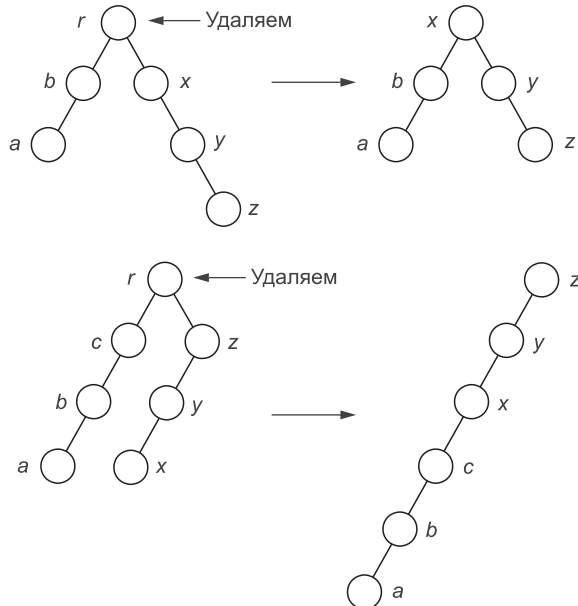


Рис. 10.11. Удаление из двух двоичных деревьев поиска

Для многих приложений нам не дается указатель на удаляемый узел; вместо этого мы получаем *ключ* того узла, который требуется удалить. Для выполнения такой операции мы комбинируем поиск по дереву с описанной ранее процедурой удаления. Результат выглядит следующим образом.

```

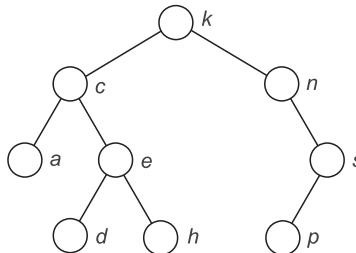
procedure DeleteKeyTree (target: key;           { Удалить узел по ключу }
                        var keyposition, root: treepointer);
{ Pre:  Параметр root является корнем двоичного дерева поиска
        с узлом, содержащим ключ, совпадающий с параметром target.
  Post: Узел с ключом, совпадающим с параметром target, удален из
        дерева и возвращен как keyposition↑. Результирующее дерево
        имеет свойства двоичного дерева поиска.
  Uses: Использует процедуру DeleteKey рекурсивно; процедуру
        DeleteNodeTree. }

begin                                           { процедура DeleteKeyTree }
  if root = nil then
    Error('Попытка удалить узел, не существующий в двоичном дереве поиска')
  else if root↑.entry.key = target then
    begin
      keyposition := root;
      DeleteNodeTree (root)
    end
  else if target < root↑.entry.key then
    DeleteKeyTree(target, keyposition, root↑.left)
  else
    DeleteKeyTree(target, keyposition, root↑.right)
end;                                           { процедура DeleteKeyTree }

```

## Упражнения 10.2

Первые несколько упражнений основываются на приведенном ниже двоичном дереве поиска. Ответьте на каждую часть каждого упражнения независимо, используя исходное дерево в качестве основы для каждой из этих частей.



**E1.** Покажите ключи, с которыми будут сравниваться каждая из следующих мишеней в процессе поиска в изображенном выше двоичном дереве поиска.

- |       |       |       |
|-------|-------|-------|
| (a) c | (d) a | (g) f |
| (b) s | (e) d | (h) b |
| (c) k | (f) m | (i) t |

- Е2.** Включите каждый из указанных ниже ключей в то же двоичное дерево поиска. Покажите, какие сравнения ключей будут выполняться в каждом случае. Выполните каждую часть независимо, включая ключ в исходное дерево.
- |              |              |              |
|--------------|--------------|--------------|
| (a) <i>m</i> | (c) <i>b</i> | (e) <i>c</i> |
| (b) <i>f</i> | (d) <i>t</i> | (f) <i>s</i> |
- Е3.** Удалите каждый из указанных ниже ключей из того же двоичного дерева поиска, используя алгоритм, разработанный в настоящем разделе. Выполните каждую часть независимо, удаляя ключи из исходного дерева.
- |              |              |              |
|--------------|--------------|--------------|
| (a) <i>a</i> | (c) <i>n</i> | (e) <i>e</i> |
| (b) <i>p</i> | (d) <i>s</i> | (f) <i>k</i> |
- Е4.** Нарисуйте двоичные деревья поиска, которые построит процедура InsertTree для списка из 14 имен, порядок которых указан в приведенных ниже вариантах, если эти имена будут включаться в предварительно пустое двоичное дерево поиска.
- |                                                             |
|-------------------------------------------------------------|
| (a) Jan Guy Jon Ann Jim Eva Amy Tim Ron Kim Tom Roy Kay Dot |
| (b) Amy Tom Tim Ann Roy Dot Eva Ron Kim Kay Guy Jon Jan Jim |
| (c) Jan Jon Tim Ron Guy Ann Jim Tom Amy Eva Roy Kim Dot Kay |
| (d) Jon Roy Tom Eva Tim Kim Ann Ron Jan Amy Dot Guy Jim Kay |
- Е5.** Все части этого упражнения относятся к двоичным деревьям поиска, изображенным на рис. 10.8 и рассматривают различные варианты порядка включения ключей *a*, *b*, ..., *g* в первоначально пустое двоичное дерево поиска.
- (a) Приведите четыре различных порядка включения ключей, каждый из которых приведет к двоичному дереву поиска, изображенному на рис. 10.8, (a).
  - (b) Приведите четыре различных порядка включения ключей, каждый из которых приведет к двоичному дереву поиска, изображенному на рис. 10.8, (b).
  - (c) Приведите четыре различных порядка включения ключей, каждый из которых приведет к двоичному дереву поиска, изображенному на рис. 10.8, (c).
  - (d) Объясните, почему имеется только один порядок включения ключей, который приведет к двоичному дереву поиска, вырожденному до цепочек, показанных на рис. 10.8, (d) и (e).
- Е6.** Использование рекурсии в процедуре InsertTree не является существенным условием, поскольку это хвостовая рекурсия. Перепишите процедуру InsertTree в нерекурсивной форме. [Вам для продвижения по дереву понадобится локальная переменная-указатель; кроме того, полезно иметь булеву переменную для индикации, когда включение действительно имело место.]
- Е7.** Напишите процедуру, которая будет удалять узел из связного двоичного дерева с помощью описанного ниже метода в случае, когда удаляемый узел имеет оба непустых поддерева. Прежде всего найдите ближайшего предка удаляемого узла по ходу симметричного просмотра, путем перемещения к его левому дочернему узлу, а затем вправо насколько возможно. (Можно с таким же успехом использо-

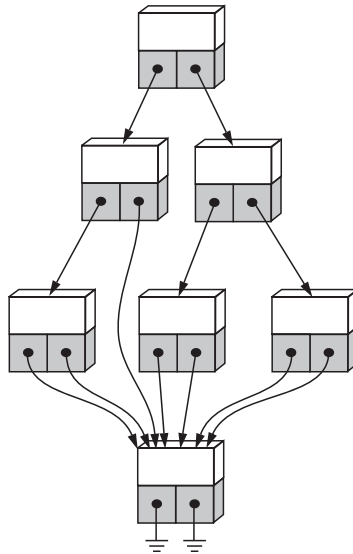
вать ближайшего потомка.) Ближайший предок гарантированно имеет не более одного дочернего узла (почему?), поэтому его можно удалить из текущей позиции без особых сложностей. Затем он может быть помещен в дерево в позицию, ранее занимаемую удаляемым узлом, после чего дерево сохранит свои свойства двоичного дерева поиска (почему?).

## Программные проекты 10.2

- P1.** Подготовьте пакет, содержащий объявления для двоичного дерева поиска и процедуры, разработанные в настоящем разделе. Пакет должен обеспечивать включение (либо в качестве модуля в Turbo Pascal, либо с помощью директивы `include`, если ее можно использовать на вашем компьютере) в любую прикладную программу.
- P2.** Подготовьте демонстрационную программу, управляемую меню, для иллюстрации использования двоичных деревьев поиска. Элементы могут состоять из одних лишь ключей, и ключами должны быть одиночные символы. Минимальные демонстрируемые возможности должны включать создание (инициализацию) дерева, включение и удаление элемента с данным ключом, поиск ключа-мишени и просмотр дерева тремя стандартными методами. Проект можно усилить включением в качестве дополнительных возможностей результатов упражнений, выполненных в этом и предыдущем разделах. Сюда относятся определение размеров дерева, вывод на экран всех элементов, расположенных так, чтобы отобразить форму дерева, и просмотр дерева различными способами.

Старайтесь, что подпрограммы вашего проекта обладали максимальной возможной модульностью, что даст вам возможность в дальнейшем заменить пакет операций над двоичным деревом поиска функционально эквивалентным пакетом для дерева другого рода.

- P3.** Напишите процедуру для древовидной сортировки, которую можно добавить в проект P1 из раздела 8.2. Определите, необходимо ли, чтобы структура списка была непрерывной или связной. Сравните полученные результаты с данными для других методов сортировки из главы 8.
- P4.** Напишите процедуру для поиска, используя двоичное дерево поиска с сигнальной меткой. Предусмотрите новый узел сигнальной метки, а также указатель на него с именем `null` (см. рис. 10.10). Замените все связи `nil` внутри двоичного дерева поиска связями `null` (к сигнальной метке). Далее при каждом поиске перед его началом сохраняйте ключ-мишень в узле сигнальной метки. Удалите из `TreeSearch` проверку на неуспешный поиск, поскольку теперь такая ситуация возникнуть не может; поиск, который находит сигнальную метку `null`, фактически и является неуспешным поиском. Запустите эту процедуру с тестовыми данными предыдущего проекта, чтобы сравнить производительность такого варианта с производительностью исходной процедуры `TreeSearch`.



**Рис. 10.12.** Двоичное дерево поиска с сигнальной меткой

- Р5.** Используя двоичное дерево поиска, реализуйте операции с гардеробом из раздела 7.3. Протестируйте ваш пакет совместно с демонстрационной программой гардероба, а затем запустите программу анализа процессорного времени совместно с пакетом двоичных деревьев поиска, проведя сравнение результатов для разных реализаций.
- Р6.** Различные авторы имеют тенденцию использовать различный словарный состав и применять те или иные слова с различными частотами. Любопытно, взяв некоторый рассказ или другой текст, определить, какие слова в нем используются и сколько раз было использовано каждое. Цель настоящего проекта заключается в сравнении нескольких различных видов двоичных деревьев поиска, которые можно было бы использовать для решения поставленной выше задачи извлечения информации. Настоящая, первая часть проекта посвящена созданию программы-драйвера и пакета извлечения информации с использованием обычных двоичных деревьев поиска. Ниже приведена последовательность операций главной программы драйвера.
1. Создайте структуру данных (двоичное дерево поиска).
  2. Запросите у пользователя имя текстового файла и откройте его для чтения.
  3. Прочитайте файл, разбейте его на отдельные слова и включите слова в структуру данных. Для каждого слова будет храниться счетчик частоты его использования (сколько раз данное слово появилось во входном потоке), и при каждом дублировании того же слова счетчик частоты будет инкрементироваться. При этом то же слово не будет дважды включаться в дерево.
  4. Выведите число выполненных сравнений и время процессора, требуемое для выполнения действий пункта 3.



5. Если пользователь пожелает, выведите все слова из структуры данных в алфавитном порядке вместе со значениями частоты их использования.
6. Поместите действия пунктов 2–5 в цикл **repeat**, который будет повторяться столько раз, сколько пожелает пользователь. Таким образом пользователь сможет построить структуры деревьев более чем для одного файла. Читая один и тот же файл дважды, пользователь сможет сравнить время извлечения информации с временем, требуемым для первоначального включения.

Ниже следуют дальнейшие спецификации для программы драйвера.

- Входом для драйвера будет файл типа **text**. Программа будет выполняться с несколькими различными файлами; имя используемого файла должно запрашиваться у пользователя в процессе выполнения программы. См. приложение С, где описан пакет ввода-вывода.
- Слово определяется как последовательность букв вместе с апострофами (') и дефисами (-), при условии, что апострофы и дефисы либо непосредственно следуют за буквой, либо непосредственно предшествуют ей. Прописные и строчные буквы следует рассматривать, как одинаковые (путем преобразования всех букв либо в строчные, либо в прописные по вашему выбору). Слово усекается до своих первых 20 символов (т. е. в структуре данных сохраняются только первые 20 символов каждого слова), хотя в тексте могут появляться слова длиной более 20 символов. В текстовом файле могут встречаться неалфавитные символы (цифры, пробелы, знаки препинания, управляющие символы). Появление любого из них завершает слово, а следующее слово появляется только когда встречается буква.
- Напишите свой драйвер таким образом, чтобы его нисколько не надо было изменять, если в дальнейшем изменится реализация структур данных.

Ниже приведены спецификации подпрограмм для обработки двоичных деревьев поиска, которые следует реализовать в первую очередь путем объявления типа **type structure = treepointer** и использования подходящих операций над двоичными деревьями поиска.

|                                                                                                                                                                                                     |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>procedure</b> Create( <b>var</b> S: structure); <span style="float: right;">{ Создание }</span><br><i>предусловие:</i> Отсутствует<br><i>постусловие:</i> Структура S создана и является пустой. |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>procedure</b> Update( <b>w</b> : word; <b>var</b> S: structure; <b>var</b> ncomp: integer); <span style="float: right;">{ Обновление }</span><br><i>предусловие:</i> S уже создана; переменной w было назначено слово.<br><i>постусловие:</i> Если значение w еще не присутствовало в S, тогда w включено в S и ее счетчик частоты приравнен 1. Если значение w уже присутствовало в S, ее счетчик частоты увеличен на 1. Переменная ncomp показывает выполненное число сравнений слов. |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**procedure** Print(**var** S: structure); { Вывод на экран }  
*предусловие:* Структура S уже создана.  
*постусловие:* Все слова в S выведены на терминал в алфавитном порядке вместе с их счетчиками частот.

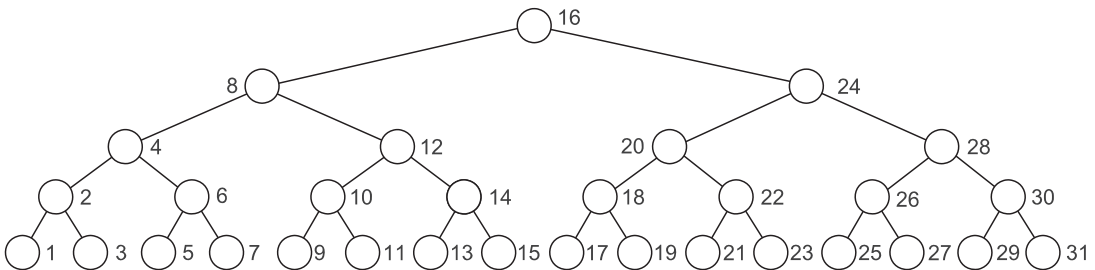
**procedure** WriteMethod; { Вывод метода }  
*предусловие:* Отсутствует  
*постусловие:* Процедура вывела короткую строку, идентифицирующую абстрактный тип данных, используемый для структуры.

## 10.3. Построение двоичного дерева поиска

Предположим, что мы имеем список уже упорядоченных узлов, или, возможно, файл с записями, в котором ключи уже упорядочены по алфавиту. Если мы хотим использовать эти узлы для поиска информации, добавления новых узлов или внесения других изменений, тогда нам следует взять этот список или этот файл с узлами и преобразовать их в двоичное дерево поиска.

Мы можем, разумеется, начать с пустого двоичного дерева поиска и с помощью алгоритма включения в дерево включать в него один узел за другим. Однако узлы у нас уже упорядочены, поэтому результирующее дерево примет форму одной длинной цепочки, работа с которой будет осуществляться очень медленно, со скоростью последовательного, а не двоичного поиска. Поэтому нам хотелось бы взять эти узлы и построить из них дерево с максимально возможной кустистостью, чтобы уменьшить и время построения дерева, и в дальнейшем время поиска в нем. Если, например, число узлов  $n$  равно 31, мы хотели бы построить дерево, изображенное на рис. 10.13.

цель



**Рис. 10.13.** Полное двоичное дерево с 31 узлом

На рис. 10.13 все узлы пронумерованы в соответствии с их естественным порядком, т. е. в последовательности симметричного просмотра, которая совпадает с порядком, в котором они были получены и включены в дерево. Если вы внимательно посмотрите на этот рисунок, вы мо-

жете обнаружить важное свойство обозначений узлов. Все обозначения листьев представляют собой нечетные числа, т. е. числа, которые не делятся на 2. Обозначениями узлов на один уровень выше являются числа 2, 6, 10, 14, 18, 22, 26 и 30. Все эти числа являются удвоенными нечетными числами, т. е. они четные, но не делятся на 4. На следующем уровне по дороге вверх обозначения составляют ряд 4, 12, 20 и 28, т. е. числа, делящиеся на 4, но не на 8. Наконец, узлы непосредственно под корнем имеют обозначения 8 и 24, а сам корень — обозначение 16. Таким образом, мы можем сформулировать свойство ключей:

свойство ключей

*Если узлы полного двоичного дерева обозначены в последовательности симметричного просмотра, тогда каждый узел находится в точности на столько уровней выше листьев, какова высшая степень 2, на которую можно разделить его обозначение.*

Это утверждение имеет также то преимущество, что мы не должны беспокоиться о ситуации, когда число узлов не оказывается в точности на единицу меньше некоторой степени 2, в результате чего результирующее дерево будет неполным и не столь симметричным, как дерево на рис. 10.13. Мы будем разрабатывать наш алгоритм исходя из того, что дерево полностью симметрично, а после получения всех узлов подумаем, как его подправить.

### 10.3.1. Начинаем

Наши действия после поступления узла номер 1 совершенно очевидны. Он будет листом и, следовательно, его левый и правый указатели оба должны принять значение **nil**. Узел номер 2 размещается над узлом 1, как это показано на рис. 10.14. Поскольку узел 2 будет связан с узлом 1, мы, очевидно, должны иметь какую-то возможность помнить, где расположен узел 1. Узел 3 опять является листом, однако он располагается в правом поддереве узла 2, и мы должны запомнить указатель на узел 2.

Значит ли все это, что мы должны хранить список указателей на все уже обработанные узлы, чтобы определить, как подсоединить очередной узел? Ответ будет «нет», так как когда поступает узел 3, все связи для узла 1 уже созданы. Про узел 2 следует помнить до получения узла 4, чтобы установить левую связь от узла 4, но к этому моменту указатель на узел 2 уже не нужен. Точно так же про узел 4 следует помнить, пока не будет обработан узел 8. На рис. 10.14 округлыми стрелками обозначены узлы, о которых следует помнить по мере роста дерева.

Теперь должно быть ясно, что для установки последующих связей мы должны запоминать лишь указатели на один узел на каждом уровне, именно, на последний обработанный на этом уровне узел. Мы будем хранить эти указатели в массиве с именем **lastnode**, и этот массив будет относительно невелик. Так, дерево с 20 уровнями может вместить  $2^{20} - 1 > 1000000$  узлов.

Когда поступает новый узел, он, очевидно, является последним по порядку узлом, и мы можем установить его правый указатель в **nil** (по крайней мере, временно). Левый указатель нового узла есть **nil**, если он

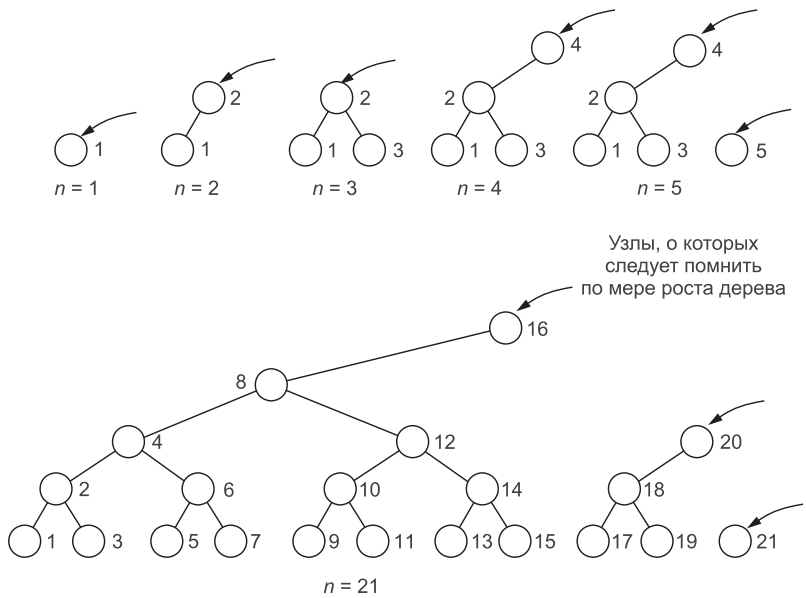


Рис. 10.14. Встраивание в дерево первых узлов

является листом. В противном случае этим указателем является элемент массива `lastnode`, расположенный на один уровень ниже, чем новый узел. Для того, чтобы обрабатывать листья так же, как и остальные узлы, мы считаем, что листья находятся на уровне 0, индексируем массив `lastnode` от  $-1$  до максимальной допустимой высоты, и обеспечиваем условие `lastnode[-1] = nil`.

### 10.3.2. Объявления и главная процедура

Теперь мы можем написать объявления переменных, необходимых для решения нашей задачи, и, занимаясь этими объявлениями, заодно набросать контуры главной процедуры. Первым шагом будет получение всех узлов и включение их в дерево. Мы полагаем, что для получения каждого нового узла у нас имеется вспомогательная процедура с приведенными ниже спецификациями.

```
procedure GetNode (var newnode: treepointer; var S: structure);
                                { Получить узел }
```

*предусловие:* S есть структура данных, содержащая элементы типа `treeentry`, упорядоченная согласно значениям ключей, которые будут включаться в двоичное дерево поиска.

*постусловие:* Если структура S пуста, `newnode` становится `nil`; в противном случае первый элемент (в порядке ключей) структуры S удален из S, помещен в узел дерева, а `newnode` указывает на этот узел.

Мы оставляем тип данных `structure` неопределенным: он может представлять собой файл, из которого читаются элементы, или упорядоченный список, или двоичное дерево поиска, подлежащее балансированию.

После того, как все узлы, полученный с помощью процедуры `GetNode`, включены в новое двоичное дерево поиска, мы должны найти корень этого дерева и затем подсоединить все правые поддеревья, которые могут остаться «висящими» (см. узлы 5 и 21 на рис. 10.14).

Главная функция выглядит таким образом:

главная процедура

```
procedure BuildTree (var root: treepointer);           { Построить дерево }
{ Pre: Уже создано двоичное дерево поиска, на которое указывает
  параметр root.
  Post: Дерево было реорганизовано в сбалансированное дерево.
  Uses: Использует процедуры GetNode, Insert, ConnectSubtrees, Findroot. }
{ Замечание: процедура GetNode используется для получения списка
  элементов в правильном порядке ключей. }
const maxheight = 20;
type treelevel = - 1..maxheight;                       { число шагов над листьями }
var
  lastnode: array [treelevel] of treepointer;
           { содержит указатель на последний обработанный узел
             на каждом уровне }
  counter: integer;                                     { число прочитанных к данному моменту узлов }
  newnode: treepointer;                                { newnode является текущим входным узлом }
  level: treelevel;                                    { уровень newnode }
begin
  for level := - 1 to maxheight do
    lastnode [level] := nil;
  counter := 0;
  GetNode(newnode);
  while newnode <> nil do
    begin
      counter := counter + 1;
      Insert(newnode);
      GetNode(newnode);                                { получение и обработка входных данных }
    end;
  FindRoot;
  ConnectSubtrees
end;                                                    { процедура BuildTree }
```

### 10.3.3. Включение узла

В предыдущем разделе мы выяснили, как правильно устанавливать левую связь каждого узла, однако для некоторых узлов значение правой связи не должно навсегда оставаться равным `nil`. Когда поступает новый узел, он еще не может иметь правильное правое дерево, поскольку он всегда является последним (в порядке значений ключей) из поступивших к этому моменту узлов. Этот узел, следовательно, может быть правым дочерним узлом какого-то предыдущего узла. С другой стороны, он может оказаться и левым дочерним узлом, и в этом случае его родительский узел еще не прибыл. Мы можем определить, какой из этих случаев

имеет место, проанализировав массив `lastnode`. Если `level` обозначает уровень нового узла, тогда его родитель имеет уровень `level + 1`. Мы смотрим на узел `lastnode[level + 1]`. Если его правая связь все еще равна `nil`, тогда его правым дочерним узлом должен быть новый узел; если же нет, тогда его правый дочерний узел уже поступил, и новый узел должен быть левым дочерним узлом некоторого будущего узла.

Теперь мы можем формально описать, как включить новый узел в дерево.

включение

```
procedure Insert (newnode: treepointer);                                { Включение }
{ Pre:  newnode является допустимым указателем элемента,
        включаемого в двоичное дерево поиска.
  Post: newnode↑ был включен в качестве самого правого узла
        в частичное двоичное дерево поиска.
  Uses: Использует функцию Power2. }
var level: treelevel;                                                { уровень узла }
begin                                                                { процедура Insert }
    level := Power2(counter);
    newnode↑.right := nil;
    newnode↑.left := lastnode[level - 1];
    lastnode[level] := newnode;
    if lastnode[level + 1] <> nil then
      with lastnode[level + 1]↑ do
        if right = nil then right := newnode
    end;                                                            { процедура Insert }
```

Эта процедура использует функцию, которая определяет уровень `newnode↑`:

```
function Power2 (x: integer): treelevel;                                { Степень двух }
{ Pre:  Параметр x есть допустимое целое число.
  Post: Функция находит наибольшую степень двух, на которую
        делится параметр x; требуется, чтобы x ≠ 0. }
var level: treelevel;
begin                                                                { функция Power2 }
    level := 0;
    while not odd(x) do begin
      x := x div 2;
      level := level + 1
    end;
    Power2 := level
end;                                                                { функция Power2 }
```

### 10.3.4. Завершение задачи

Нахождение корня дерева не составляет труда: корень является самым верхним узлом в дереве; его указатель есть самый верхний элемент в массиве `lastnode`, не равный `nil`. Таким образом, мы имеем:

поиск корня

```

procedure FindRoot;                                     { Поиск корня }
{ Pre: Массив lastnode содержит указатели на занятые уровни
      двоичного дерева поиска.
  Post: Переменная root указывает на корень заново созданного
      двоичного дерева поиска.
var level: treelevel;
begin                                                     { процедура FindRoot }
  if counter = 0 then
    root := nil                                           { дерево пусто }
  else begin                                             { непустое дерево }
    level := maxheight;                                  { найдем высший занятый уровень;
                                                         это и есть корень }

    while lastnode[level] = nil do
      level := level - 1;
      root := lastnode[level]
    end
  end;                                                  { процедура FindRoot }

```

Наконец, мы должны определить, как привязать к дереву все поддеревья, которые после получения всех узлов оказываются еще не подсоединенными правильным образом. Трудность здесь заключается в том, что некоторые узлы в верхней части дерева все еще могут иметь свои правые связи установленными в **nil**, даже несмотря на то, что уже могли поступить узлы, принадлежащие к их правым поддеревьям.

Любой узел, для которого правый дочерний узел все еще **nil**, будет одним из узлов в массиве lastnode. В качестве его правого дочернего узла должен быть установлен самый верхний узел в lastnode, который еще не входит в его левое поддерево. В результате мы приходим к следующему алгоритму.

связывание  
поддеревьев  
вместе

```

procedure ConnectSubtrees;                               { Подсоединить поддеревья }
{ Pre: Почти законченное двоичное дерево поиска уже
      инициализировано. Массив lastnode уже инициализирован
      и содержит информацию, требуемую для завершения
      двоичного дерева поиска.
  Post: Двоичное дерево поиска полностью создано.}
var
  p: treepointer;
  level: treelevel;
  s: treelevel;
begin                                                     { процедура ConnectSubtrees }
  level := maxheight;
  while (lastnode[level] = nil) and (level > 1) do
    level := level - 1;                                  { найдем наивысший узел: корень }
  while level > 1 do                                     { узлы на уровнях 1 и 0 уже OK }
    with lastnode[level] do
      if right <> nil then
        level := level - 1                                { просмотр вниз в поисках наивысшего
                                                         висящего узла }
      else begin                                         { случай: правое поддерево не определено }
        p := left;                                       { найдем наивысший элемент в последнем узле,
                                                         который не входит в левое поддерево }
        s := level - 1;

```

```

repeat
  p = p↑.right;
  s := s - 1
until (p = nil) or ( p <> lastnode [s]);
right := lastnode [s];
level := s           { узлы на уровнях между level и s находятся
                      с левой стороны }
end                  { подсоединяем висящие поддеревья }
end;                 { процедура ConnectSubtrees }

```

### 10.3.5. Оценка

Алгоритм, описанный в настоящем разделе, образует двоичное дерево поиска, которое не во всех случаях будет полностью сбалансировано. Если, например, поступают 32 узла, тогда узел 32 станет корнем дерева, а все оставшиеся узлы (31 узел) расположатся в его левом поддереве. В результате листья окажутся удаленными от корня на пять шагов. Если бы корень дерева выбрать оптимальным образом, тогда большинство листьев будут отстоять от него на четыре шага, и только один окажется на пятом шаге. Без такой оптимизации обычно будет выполняться на одно сравнение больше, чем необходимо.

Одно лишнее сравнение при двоичном поиске не такая уж высокая цена, причем легко видеть, что дерево, образованное нашим методом, никогда не будет отличаться от оптимального более, чем на один уровень. Существуют изощренные методы построения двоичных деревьев поиска, сбалансированных наилучшим образом, однако вполне допустимо рекомендовать более простой метод, при котором мы не должны заранее знать, сколько узлов будет в дереве.

В упражнениях рассматриваются способы использования нашего алгоритма, когда берется произвольное двоичное дерево поиска и его узлы переставляются с целью получить более сбалансированное дерево и, следовательно, уменьшить время поиска. Опять же существуют более изощренные методы (которые, впрочем, скорее всего будут работать медленнее) для балансирования дерева. В разделе 10.4 мы изучим AVL-деревья, в которых включения и удаления осуществляются таким образом, что дерево всегда остается почти сбалансированным. Однако для многих практических применений описанные выше простые алгоритмы работают достаточно удовлетворительно.

### 10.3.6. Случайные деревья поиска и оптимизация

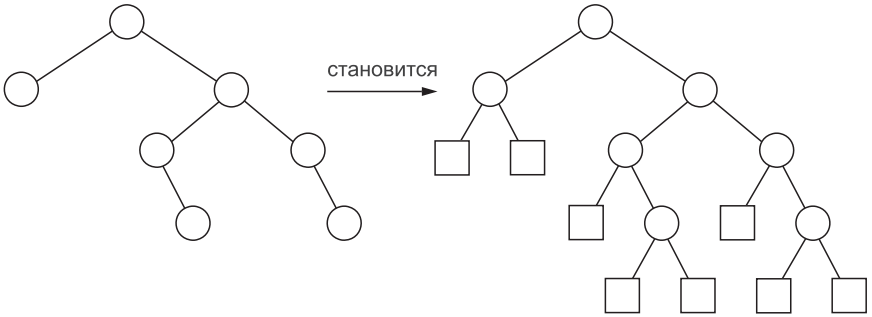
Завершая этот раздел, зададим себе вопрос, стоит ли в среднем заниматься балансированием дерева? Если мы полагаем, что ключи поступают в случайном порядке, тогда в среднем насколько больше сравнений будет выполняться в полученном дереве, чем в полностью сбалансированном дереве?

Чтобы ответить на этот вопрос, мы прежде всего преобразуем двоичное дерево поиска в 2-дерево следующим образом. Будем изображать все вершины двоичного дерева в виде кружков, и добавим к рисунку новые



расширенное  
двоичное дерево

вершины, изображаемые квадратами, которые заменят все пустые под-деревья (**nil**-связи). Этот процесс проиллюстрирован на рис. 10.15. Все вершины исходного двоичного дерева стали внутренними вершинами 2-дерева, а все новые вершины (листья) являются внешними. Успешный поиск завершается на внутренней вершине 2-дерева, а неуспешный поиск — на листе. В результате длина внутреннего пути дает нам число сравнений при успешном поиске, а длина внешнего пути дает число сравнений при неуспешном поиске. Поскольку для каждого внутреннего узла выполняется два сравнения, число сравнений, выполняемых при однократном поиске каждого ключа в дереве составляет удвоенную длину внутреннего пути.



**Рис. 10.15.** Расширение двоичного дерева с образованием 2-дерева

подсчет числа  
сравнений

Мы будем полагать, что при построении дерева все  $n!$  возможных порядков ключей равновероятны. Если в дереве  $n$  узлов, мы обозначим через  $S(n)$  число сравнений, выполняемых в среднем успешном поиске и через  $U(n)$  число сравнений в среднем неуспешном поиске.

Число сравнений, требуемых для нахождения любого ключа в дереве в точности на 1 больше числа сравнений, которые потребовались для включения этого ключа в дерево, а включение его требовало такое же число сравнений, как и неуспешный поиск, который показал, что этого ключа еще нет в дереве. Отсюда мы имеем соотношение

$$S(n) = 1 + \frac{U(0) + U(1) + \dots + U(n-1)}{n}$$

Соотношение между длинами внутреннего и внешнего путей, как было показано в теореме 7.4, составляет

$$S(n) = \left(1 + \frac{1}{n}\right)U(n) - 3$$

рекуррентное  
отношение

Объединив два последние равенства, получаем

$$(n+1)U(n) = 4n + U(0) + U(1) + \dots + U(n-1).$$

Мы можем разрешить это рекуррентное отношение, написав равенство для  $n-1$  вместо  $n$ :

$$nU(n-1) = 4(n-1) + U(0) + U(1) + \dots + U(n-2).$$

Выполнив вычитание, получим:

$$U(n) = U(n-1) + \frac{4}{n+1}.$$

Сумма

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

гармонический  
ряд

носит название *гармонического ряда* порядка  $n$ , а в теореме А.4 показано, что это число приблизительно равно натуральному логарифму  $\ln n$ . Поскольку  $U(0) = 0$ , мы можем оценить  $U(n)$  начав со дна и добавляя:

$$U(n) = 4 \left[ \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n+1} \right] = 4H_{n+1} - 4 \approx 4 \ln n.$$

По теореме 7.4 число сравнений при успешном поиске также приблизительно равно  $4 \ln n$ . Поскольку поиск в любом двоичном дереве поиска требует двух сравнений на узел, а оптимальная высота составляет  $\lg n$ , оптимальное число сравнений составляет  $2 \lg n$ . Однако (см. раздел А.2)

$$\ln n = (\ln 2)(\lg n).$$

Преобразовав натуральные логарифмы в логарифмы по основанию 2, в итоге получим

**Теорема 10.3**

*Среднее число узлов, посещенных при поиске в среднем двоичном дереве поиска с  $n$  узлами, приблизительно равно  $2 \ln n = (2 \ln 2)(\lg n) \approx 1.39 \lg n$ , а число сравнений ключей приблизительно составляет  $4 \ln n = (4 \ln 2)(\lg n) \approx 2.67 \lg n$ .*

**Следствие 10.4**

*Среднее двоичное дерево поиска требует приблизительно в  $2 \ln 2 \approx 1.39$  раз больше операций сравнения, чем полностью сбалансированное дерево.*

цена  
недостаточного  
балансирования

Другими словами, средняя цена недостаточного балансирования двоичного дерева поиска составляет приблизительно 39 процентов дополнительных операций сравнения. В приложениях, где оптимальность действительно важна, эту цену следует сравнить с дополнительными издержками балансирования дерева, или с издержками поддержания его в сбалансированном состоянии. Заметьте, что эти последние задачи требуют не только компьютерного времени, но и дополнительного труда программистов.

### Упражнения 10.3.

- Е1.** Изобразите на бумаге частичные двоичные деревья поиска (как на рис. 10.14), которые будут построены методом, рассмотренным в настоящем разделе, для  $n = 6$ ,  $n = 7$  и  $n = 8$ .
- Е2.** Напишите процедуру `GetNode` для случая, когда структура данных представляет собой упорядоченный список. Процедура попросту удалит первый элемент списка, поместит его в дерево в качестве узла и вернет указатель на этот узел.

- Е3.** Напишите процедуру `GetNode` для случая, когда входная структура представляет собой двоичное дерево поиска, поэтому тип `structure = treepointer`, а `S` указывает на корень дерева. Процедура должна будет найти и удалить самый левый узел двоичного дерева поиска и вернуть указатель на него как `newnode`. [Этот вариант вместе с `BuildTree` образует процедуру для балансирования двоичного дерева поиска.]
- Е4.** Напишите вариант процедуры `GetNode`, которая будет читать ключи из текстового файла, по одному ключу за раз, создавать узел дерева, содержащий этот ключ и возвращать указатель на узел. При достижении конца файла процедура должна вернуть значение `nil`. [Этот вариант вместе с `BuildTree` образует процедуру для чтения двоичного дерева поиска из упорядоченного файла.]
- Е5.** Для трех ключей имеются  $6 = 3!$  возможных порядков следования, но только 5 различных двоичных деревьев с тремя узлами. Поэтому эти двоичные деревья с неодинаковой вероятностью соответствуют деревьям поиска. Найдите, какие из этих пяти двоичных деревьев поиска соответствуют каждому из шести возможных порядков ключей. Тем самым вы найдете вероятность построения каждого из двоичных деревьев поиска из случайно упорядоченных входных данных.
- Е6.** Для четырех ключей имеются  $24 = 4!$  возможных порядков следования, но только 14 различных двоичных деревьев с четырьмя узлами. Поэтому эти двоичные деревья с неодинаковой вероятностью соответствуют деревьям поиска. Найдите, какие из этих 14 двоичных деревьев поиска соответствуют каждому из 26 возможных порядков ключей. Найдите вероятность построения каждого из двоичных деревьев поиска из случайно упорядоченных входных данных.

## 10.4. Балансирование по высоте: AVL-деревья

Алгоритм, рассмотренный в разделе 10.3, может использоваться для построения почти сбалансированных двоичных деревьев поиска, или для восстановления сбалансированности, если оказывается необходимым полностью реконструировать дерево. Однако во многих приложениях операции включения и удаления должны выполняться непрерывно и в непредсказуемом порядке. В некоторых приложениях такого рода важно оптимизировать время поиска путем сохранения почти полной сбалансированности дерева в любой момент. Метод достижения этой цели, рассматриваемый в настоящем разделе, был предложен в 1962 г. двумя русскими математиками, Г. М. Адельсон-Вельским и Е. М. Ландисом; образуемые двоичные деревья поиска были в их честь названы AVL-деревьями.

AVL-деревья дают возможность осуществлять поиск, включение и удаление в дереве с  $n$  узлами за время порядка  $O(\log n)$  даже в худшем случае. Высота AVL-дерева с  $n$  узлами, как будет показано ниже, никогда не может превысить  $1.44 \lg n$ , в результате чего даже в худшем случае поведение AVL-дерева не может быть заметно хуже случайного двоичного дерева поиска. Почти во всех случаях, однако, фактическая длина

поиска весьма близка к  $\lg n$ , отчего поведение AVL-деревьев приближается к поведению идеального полностью сбалансированного двоичного дерева поиска.

### 10.4.1. Определение

В полностью сбалансированном дереве левое и правое поддеревья любого узла должны иметь одинаковую высоту. Хотя мы не можем по всех случаях достигнуть этой цели, однако аккуратно строя дерево поиска, мы всегда можем обеспечить, чтобы высоты левых и правых поддеревьев не различались больше, чем на 1. Соответственно мы можем дать следующее определение:

#### Определение

**AVL-дерево** представляет собой двоичное дерево поиска, в котором высоты левых и правых поддеревьев отличаются не более чем на 1, и в котором левые и правые поддеревья также являются AVL-деревьями.

С каждым узлом AVL-дерева ассоциируется **фактор сбалансированности**, который может принимать значения *левовысокий*, *равновысокий* или *правовысокий*, соответствующие ситуациям, когда левое поддерево имеет высоту соответственно большую, равную или меньшую, чем правое поддерево.

Рисуя графические представления деревьев, мы будем обозначать левовысокий узел знаком '/', равновысокий узел знаком '=', а правовысокий узел знаком '\'. На рис. 10.16 изображены несколько AVL-деревьев, а также несколько двоичных деревьев, которые не удовлетворяют приведенному выше определению.

Заметьте, что наше определение не требует, чтобы все листья были на одном или прилегающих уровнях. На рис. 10.17 показаны несколько существенно перекошенных AVL-деревьев, у которых правые поддеревья имеют значительно большую высоту, чем левые.

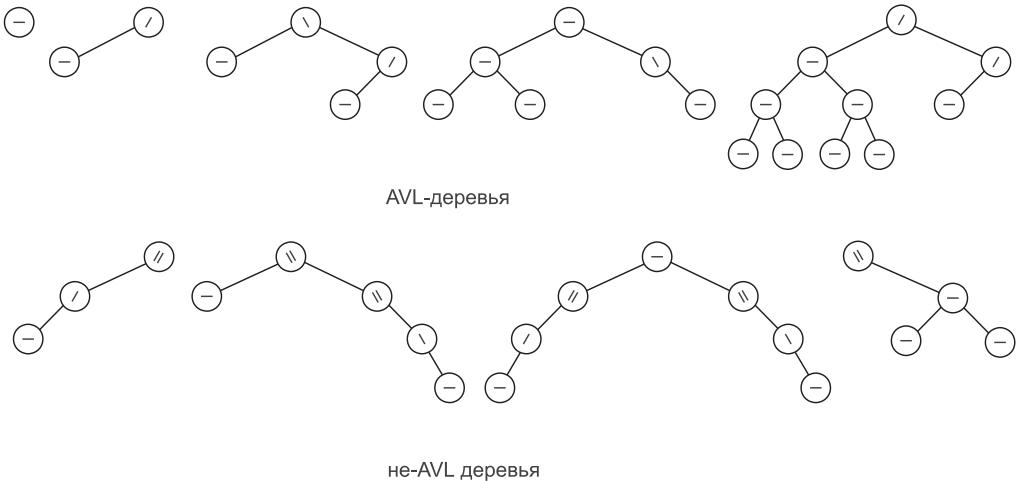


Рис. 10.16. Примеры AVL-деревьев и других двоичных деревьев

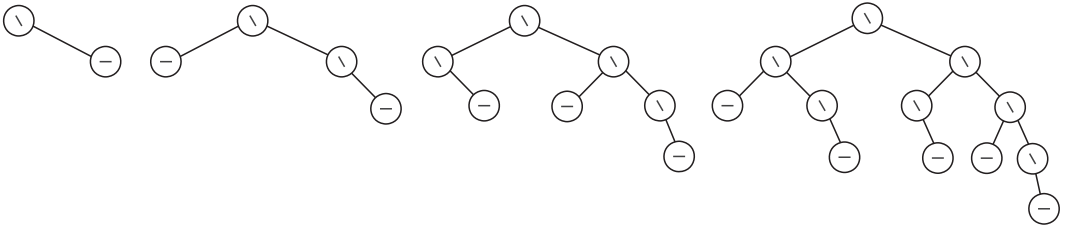


Рис. 10.17. AVL-деревья, скошенные вправо

## 10.4.2. Включение узла

### 1. Введение

Мы можем включить новый узел в AVL-дерево, воспользовавшись прежде всего алгоритмом включения для обычного двоичного дерева, сравнивая ключ нового узла с ключом корня и включая затем новый узел в левое или правое поддерево. Часто оказывается, что новый узел может быть включен в поддерево без изменения его высоты, и тогда ни высота, ни сбалансированность корня не изменяются. Даже в тех случаях, когда высота поддерева увеличивается, может оказаться, что удлинилось более короткое поддерево, в результате чего всего лишь изменится значение фактора сбалансированности. Единственный случай, который может привести к неприятностям, возникает, когда новый узел добавляется в корневое поддерево (т. е. поддерево, идущее от корневого узла), которое уже выше, чем второе поддерево. В этом случае это поддерево будет иметь высоту на 2 больше второго, в то время как для AVL-деревьев высоты поддеревьев не могут различаться более чем на 1. Перед тем, как мы приступим к более тщательному изучению этой ситуации, посмотрим на рис. 10.18, где показан рост AVL-дерева по ходу нескольких включений, а затем попробуем собрать вместе наши идеи, сформулировав алгоритм на языке Pascal.

неприятности

### 2. Соглашения языка Pascal

Базовая структура нашего алгоритма будет той же самой, что и у обычного рекурсивного алгоритма включения в двоичное дерево, который был рассмотрен в разделе 10.2.3, но с некоторыми добавлениями, отвечающими условиям, накладываемым на AVL-деревья. Прежде всего, каждая запись, соответствующая узлу, будет иметь дополнительное поле (вместе с информационным полем, а также левым и правым указателями), определяемое как

bf: balancefactor;

где мы используем перечислимый тип данных

**type** balancefactor = (LH, EQ, RH);

в котором символические константы LH, EQ, RH обозначают левовысокий, равновысокий или правовысокий фактор сбалансированности.

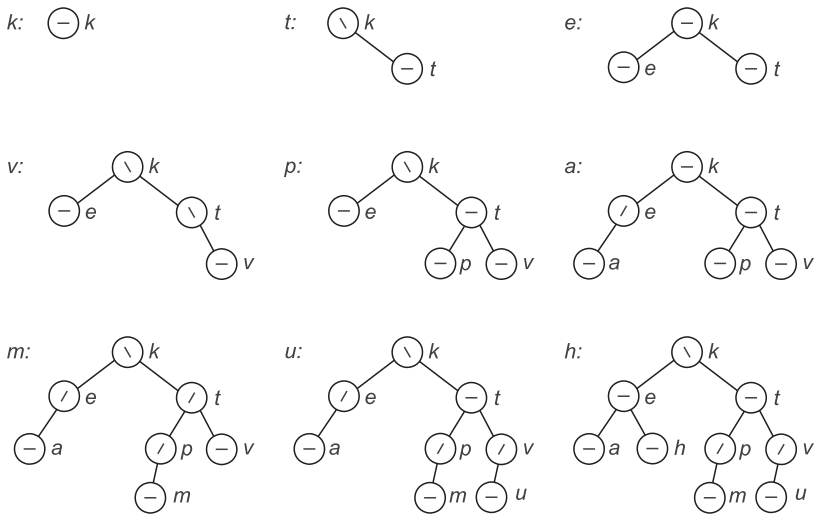


Рис. 10.18. Простое включение узлов в AVL-дерево

Далее, мы должны сохранять информацию о том, произошло ли в результате включения увеличение высоты, или нет, чтобы можно было соответствующим образом изменить значение фактора сбалансированности. Мы достигаем этой цели путем введения дополнительного параметра вызова с именем *taller* (выше) булева типа. Задача восстановления сбалансированности (в случае необходимости) будет решаться вспомогательными процедурами *LeftBalance* и *RightBalance*.

Имея эти определения, мы теперь можем написать процедуру включения нового узла в AVL-дерево.

```

procedure InsertAVL (var root: treepointer;      { Включение в AVL-дерево }
                     newnode: treepointer; var taller: Boolean);
{ Pre:   На корень AVL-дерева указывает указатель root, а newnode
          есть новый узел, который требуется включить в дерево.
  Post:  Узел newnode включен в AVL-дерево, причем переменная taller
          равна true, если высота дерева увеличилась, и false
          в противном случае.
  Uses:  Использует процедуру InsertAVL рекурсивно; процедуры
          RightBalance, LeftBalance. }

var
  tallersubtree: Boolean;      { Увеличилась ли высота поддерева? }
begin
  if root = nil then begin
    root := newnode;
    root↑.left := nil;
    root↑.right := nil;
    root↑.bf := EQ;
    taller := true
  end

```

```

else with root↑ do
  if newnode↑.entry.key = entry.key then
    Error('Ключи-дубликаты недопустимы в AVL-деревьях.')
  else if newnode↑.entry.key < entry.key then begin
    { включаем в левое поддерево }
    InsertAVL(left, newnode, tallersubtree);
    if tallersubtree then { изменим факторы балансирования }
      case bf of
        LH: LeftBalance;
        EQ: begin bf := LH; taller := true end;
        RH: begin bf := EQ; taller := false end
      end
    else taller := false
  end
else begin { включаем в правое поддерево }
  InsertAVL(right, newnode, tallersubtree);
  if tallersubtree then
    case bf of
      LH: begin bf := EQ; taller := false end;
      EQ: begin bf := RH; taller := true end;
      RH: RightBalance
    end
  else taller := false
end
end; { процедура InsertAVL }

```

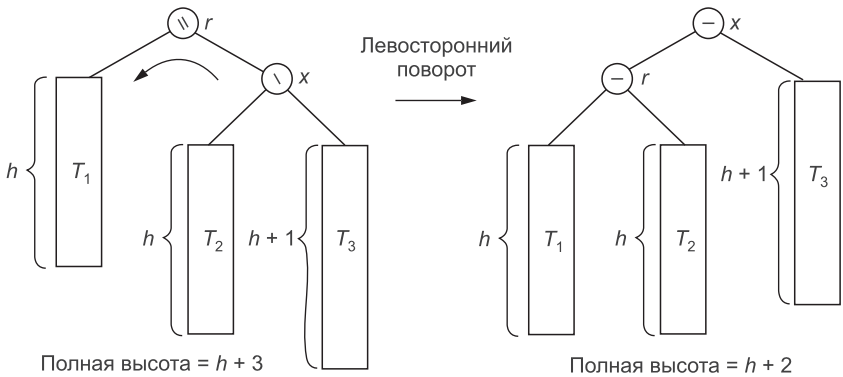
### 3. Повороты

Теперь рассмотрим случай, когда новый узел был включен в более высокое корневое поддерево, высота которого еще увеличилась, так что теперь одно поддерево имеет высоту на 2 больше, чем другое, в результате чего дерево уже не удовлетворяет условиям AVL. Мы должны перестроить часть дерева, чтобы восстановить его сбалансированность. Для определенности давайте предположим, что мы включили новый узел в правое поддерево, его высота увеличилась, а исходное дерево было правовысоким. Другими словами, мы рассматриваем случай, обрабатываемый процедурой RightBalance. Пусть  $r$  есть корень дерева, а  $x$  — корень его правого поддерева.

Нам придется рассмотреть три случая, в зависимости от фактора балансирования корня  $x$ .

#### 4. Случай 1: фактор балансирования правовысокий

Первый случай, когда фактор балансирования  $x$  является правовысоким, проиллюстрирован на рис. 10.19. Требуемое в этом случае действие называется *левосторонним поворотом*; мы поворачиваем узел  $x$  вверх к корню, сбрасывая  $r$  вниз в левое поддерево  $x$ ; поддерево  $T_2$  узлов с ключами, расположенными между ключами  $r$  и  $x$ , теперь становится правым поддеревом  $r$ , а не левым поддеревом  $x$ . Левосторонний поворот реализован в приведенной ниже Pascal-процедуре. Обратите особое внимание на то, что при выполнении шагов левостороннего поворота в соответствующем порядке, эти шаги осуществляют повороты трех переменных-указателей. Заметьте также, что после завершения поворотов высота повернутого дерева уменьшилась на 1, и, поскольку включение увеличи-



**Рис. 10.19.** Первый случай: восстановление сбалансированности путем левостороннего поворота

ло его на 1, повороты привели высоту дерева к первоначальному значению.

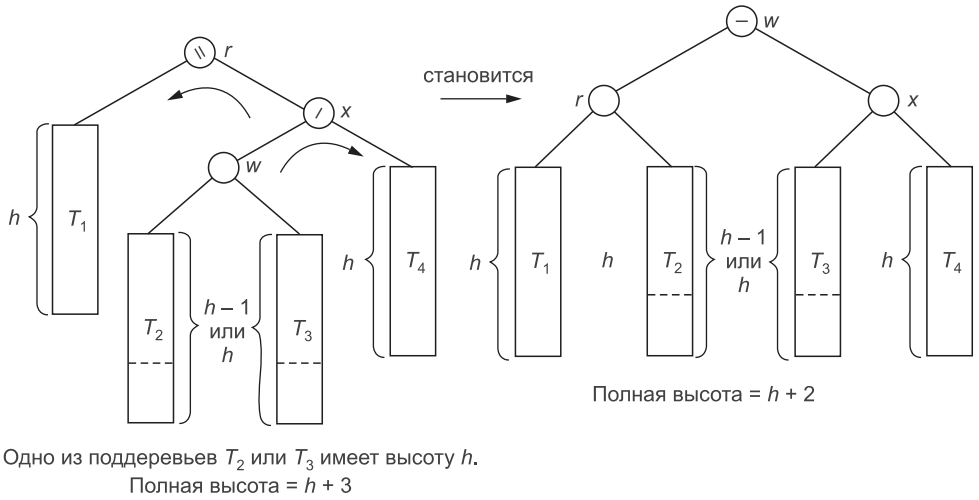
```

procedure RotateLeft (var p: treepointer;);           { Поворот влево }
{ Pre: Параметр p является корнем непустого AVL-дерева, которое
  подвергается повороту, причем его правое дочернее поддерево
  не пусто.
  Post: Правый дочерний узел корня p становится новым корнем p.
  Старый корень p становится левым дочерним узлом нового
  корня p.
var rightchild: treepointer
begin                                                     { процедура RotateLeft }
  if p = nil then
    Error('Невозможно в процедуре RotateLeft поворачивать пустое дерево.')
  else if p↑.right = nil then
    Error('Невозможно в процедуре RotateLeft сделать корнем пустое поддерево.')
  else begin
    rightchild := p↑.right;
    p↑.right := rightchild↑.left;                          { перемещаем правое поддерево
                                                             в промежуточный узел }
    rightchild↑.left := p;                                { перемещаем p в левое поддерево
                                                             промежуточного узла }
    p := rightchild                                       { назначаем p корнем нового повернутого дерева }
  end
end;                                                     { процедура RotateLeft }
  
```

## 5. Случай 2: фактор балансирования левовысокий

Второй случай, когда фактор сбалансированности  $x$  является левовысоким, несколько более сложен. Здесь для того, чтобы найти новый корень, необходимо переместить два уровня, вплоть до узла  $w$ , который является корнем левого поддерева узла  $x$ . Этот процесс показан на рис. 10.20 и называется **двойным поворотом**, поскольку преобразование выполняется за два шага, сначала путем поворота поддерева с корнем  $x$  вправо (так что  $w$  становится его новым корнем), а затем поворотом дерева с корнем  $r$  влево (перемещая узел  $w$  так, что он становится новым корнем).





**Рис. 10.20.** Второй случай: восстановление сбалансированности путем двойного поворота

В этом втором случае новые значения фактора сбалансированности для  $r$  и  $x$  зависят от прежнего значения этого фактора для  $w$ . На рис. 10.20 поддеревья  $w$  имеют одинаковую высоту, но вполне возможно, что  $w$  является либо левовысоким, либо правовысоким. Результирующие факторы сбалансированности выглядят так:

| старое $w$ | новое $r$ | новое $x$ |
|------------|-----------|-----------|
| —          | —         | —         |
| /          | —         | \         |
| \          | /         | —         |

### 6. Случай 3: фактор балансирования равновысокий

Представляется, что мы должны рассмотреть и третий случай, когда два поддерева узла  $x$  имеют равные высоты, однако этот случай фактически никогда не будет иметь места. Чтобы понять, почему это так, вспомним, что мы только что включили новый узел в поддерево с корневым узлом  $x$ , и это поддерево теперь имеет высоту на 2 больше, чем левое поддерево того же корня. Новый узел помещается либо в левое, либо в правое поддерево  $x$ . Поэтому включение увеличило высоту только одного поддерева  $x$ . Если эти поддеревья имели равные высоты после включения, тогда высота полного поддерева с корнем  $x$  не изменилась от включения, что противоречит тому, что мы уже знаем.

### 7. Pascal-процедура для балансирования

Теперь нетрудно реализовать описанные выше преобразования в Pascal-процедуре. Очевидно, что процедуры `RotateRight` и `LeftBalance` схожи с процедурами `RotateLeft` и `RightBalance`, и мы их оставим для упражнений.

```

procedure RightBalance;                                { Правое балансирование }
{ Pre: Узел AVL-дерева стал дважды несбалансирован вправо.
  Post: Свойства AVL-дерева восстановлены.
  Uses: Использует процедуры RotateRight, RotateLeft, переменные
           из InsertAVL глобально. }

var
  x,                                { указатель на правое поддереву корня root }
  w: treepointer;                    { левое поддереву узла x }
{ Также используются переменные root и taller из процедуры InsertAVL. }
begin                                { процедура RightBalance }
  x := root↑.right;
  case x↑.bf of
    RH: begin                                { одиночный поворот влево }
      root↑.bf := EQ;
      x↑.bf := EQ;
      RotateLeft(root);
      taller := false
    end
    EQ: Error('Невозможный случай в RightBalance');
    LH: begin                                { двойной поворот влево }
      w := x↑.left;
      case w↑.bf of
        EQ: begin
          root↑.bf := EQ; x↑.bf := EQ
        end
        LH: begin
          root↑.bf := EQ; x↑.bf := RH
        end
        RH: begin
          root↑.bf := LH; x↑.bf := EQ
        end
      end;
      w↑.bf := EQ;
      RotateRight(x);
      root↑.right := x;
      RotateLeft(root);
      taller := false
    end
  end
end;                                { процедура RightBalance }

```

Примеры включений, требующих одиночного или двойного поворота приведены на рис. 10.21.

## 8. Поведение алгоритма

Процедура InsertAVL может в наихудшем случае рекурсивно вызывать сама себя с целью включения нового узла столько раз, какова высота дерева. На первый взгляд может показаться, что каждый из этих вызовов может потребовать либо одиночного, либо двойного поворота соответствующего поддерева, однако в действительности будет выполнен максимум один (одиночный или двойной) поворот. Чтобы убедиться, что это так, вспомним, что повороты выполняются только в процедурах RightBalance или LeftBalance, и что эти процедуры вызываются лишь если

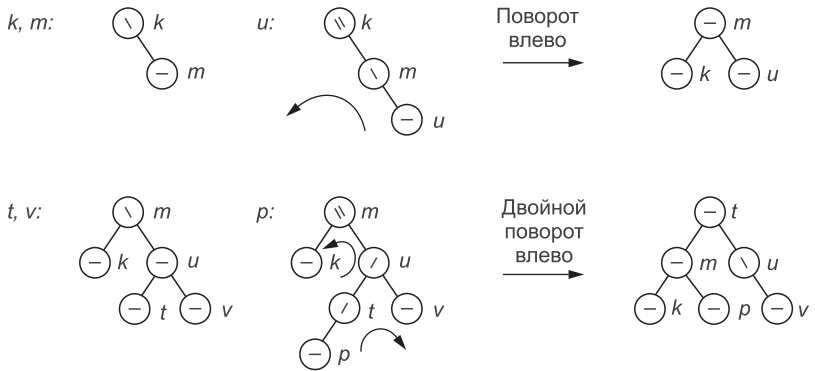


Рис. 10.21. AVL-включения, требующие поворота узлов

высота дерева увеличилась. Однако после выхода из этих процедур выполненные в них повороты ликвидировали увеличение высоты, поэтому для оставшихся (внешних) рекурсивных вызовов высота дерева не увеличивается, следовательно, нет необходимости в дополнительных поворотах или изменениях фактора балансирования.

Большая часть включений в AVL-дерево не приведет к поворотам. Даже если повороты оказываются необходимыми, они обычно осуществляются около листа, которой был только что включен. Несмотря на то, что алгоритм включения в AVL-дерево достаточно сложен, разумно ожидать, что время его выполнения будет мало отличаться от времени, требуемого для обычного дерева поиска той же высоты. Позже мы увидим, что высота AVL-деревьев значительно меньше, чем у случайных деревьев поиска, и поэтому операции включения и извлечения для AVL-деревьев оказываются значительно более эффективными, чем для случайных деревьев двоичного поиска.

### 10.4.3. Удаление узла

Удаление узла  $x$  из AVL-дерева требует применения тех же базовых операций, в том числе одиночных и двойных поворотов, которые мы использовали для включения. Мы здесь только приведем шаги в неформальном описании метода, оставив разработку полных алгоритмов для программных проектов.

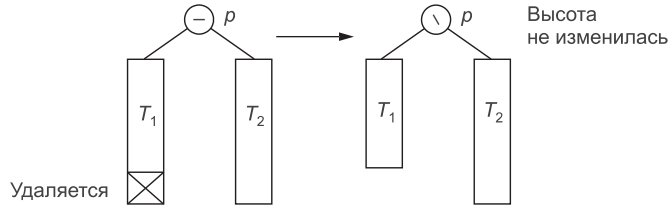
1. Сократим задачу до случая, когда удаляемый узел  $x$  имеет максимум один дочерний узел. Сначала предположим, что  $x$  имеет два дочерних узла. Найдите непосредственного предка  $x$  (пусть это будет узел  $y$ ) по ходу симметричного просмотра (с таким же успехом можно найти непосредственного потомка), для чего сначала найдите левый дочерний узел от  $x$ , а затем перемещайтесь вправо настолько это возможно до получения  $y$ . Узел  $y$  гарантированно не имеет правого дочернего узла в силу способа его поиска. Поместите  $y$  (или копию  $y$ ) в позицию дерева, занимаемую узлом  $x$  (с тем же родителем, левым и правым дочерними узлами и фактором сбалансированности, которыми характеризовался

узел  $x$ ). Теперь удалите  $y$  из его предыдущей позиции, выполняя последующие шаги и используя на каждом шаге  $y$  вместо  $x$ .

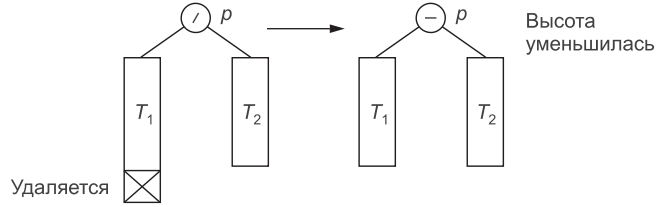
2. Удалите из дерева узел  $x$ . Поскольку мы знаем (из шага 1), что  $x$  имеет не более одного дочернего узла, мы удаляем  $x$ , просто связывая родителя  $x$  с единственным дочерним узлом  $x$  (или с `nil`, если  $x$  вообще не имеет дочерних узлов). Высота поддерева, у которого ранее был корень  $x$ , уменьшается на 1, и мы теперь должны проследить влияние этого изменения на высоту, проходя по всем узлам на пути от  $x$  назад к корню дерева. Для того чтобы зафиксировать факт уменьшения высоты поддерева, мы используем булеву переменную `shorter` (короче). Действие, выполняемое в каждом узле, зависит от значения `shorter`, от фактора сбалансированности узла и иногда от фактора сбалансированности дочернего узла.
3. Булева переменная `shorter` первоначально имеет значение `true`. Последующие шаги должны выполняться для каждого узла  $p$  на пути от родителя  $x$  к корню дерева при условии, что значение `shorter` остается равным `true`. Когда значение `shorter` становится `false`, дальнейшие изменения не нужны, и алгоритм завершается.
4. *Случай 1:* текущий узел  $p$  имеет равновысокий фактор сбалансированности. Фактор сбалансированности  $p$  изменяется соответствующим образом при укорачивании левого или правого поддерева узла  $p$ , а `shorter` принимает значение `false`.
5. *Случай 2:* Фактор сбалансированности  $p$  не является равновысоким, и более высокое поддерево укоротилось. Измените фактора сбалансированности  $p$  на равновысокий и оставьте значение `shorter true`.
6. *Случай 3:* Фактор сбалансированности  $p$  не является равновысоким, и укоротилось более короткое поддерево. Теперь условие высоты для AVL-дерева нарушается в узле  $p$ , поэтому для восстановления сбалансированности мы используем поворот, как это описано ниже. Пусть  $q$  есть корень более высокого из поддеревьев узла  $p$  (того, которое не укоротилось). Мы имеем три случая в зависимости от значения фактора сбалансированности  $q$ .
7. *Случай 3a:* Фактор сбалансированности  $q$  является равновысоким. Одиночный поворот (с изменением факторов сбалансированности  $p$  и  $q$ ) восстанавливает сбалансированность, и `shorter` становится равным `false`.
8. *Случай 3b:* Факторы сбалансированности  $q$  и  $p$  одинаковы. Используем одиночный поворот, устанавливаем значения факторов сбалансированности  $p$  и  $q$  равновысокими и оставляем значение `shorter true`.
9. *Случай 3c:* Факторы сбалансированности  $q$  и  $p$  противоположны. Используем двойной поворот (сначала вокруг  $q$ , затем вокруг  $p$ ), устанавливаем значение фактора сбалансированности для нового корня равновысоким и оставляем значение `shorter true`.

В случаях 3a, b, с направление поворота зависит от того, какое из двух поддеревьев — правое или левое — было укорочено. Некоторые возможности проиллюстрированы на рис. 10.22, а пример удаления узла показан на рис. 10.23.

повороты  
не требуются

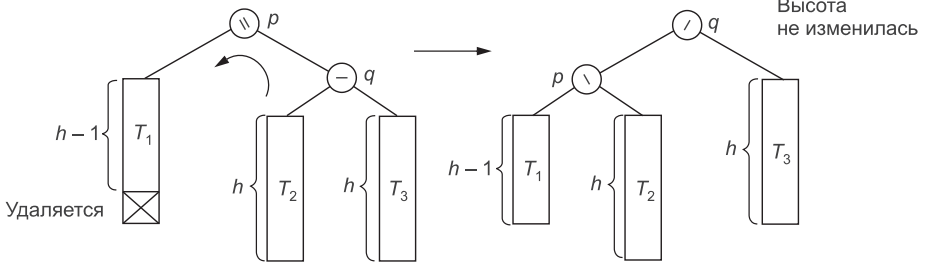


Случай 1

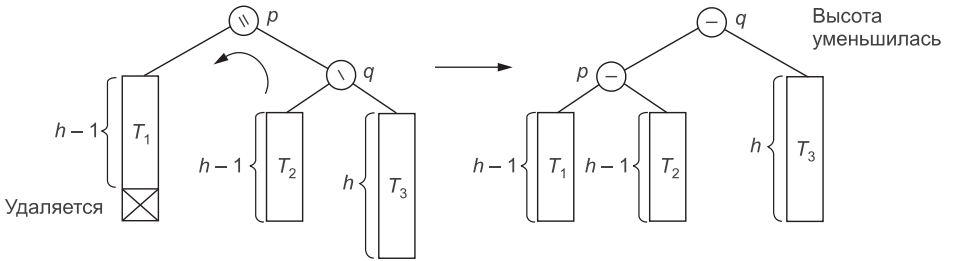


Случай 2

одиночный  
поворот  
влево

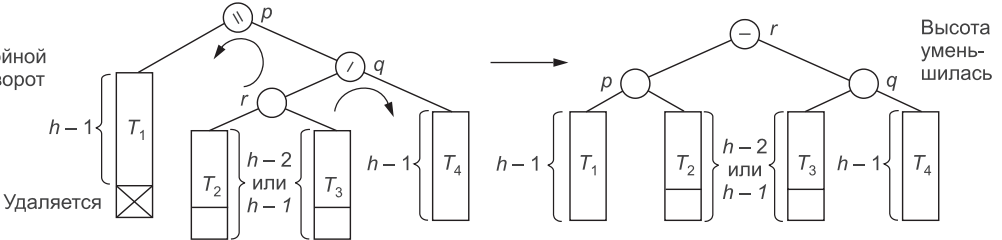


Случай 3а



Случай 3б

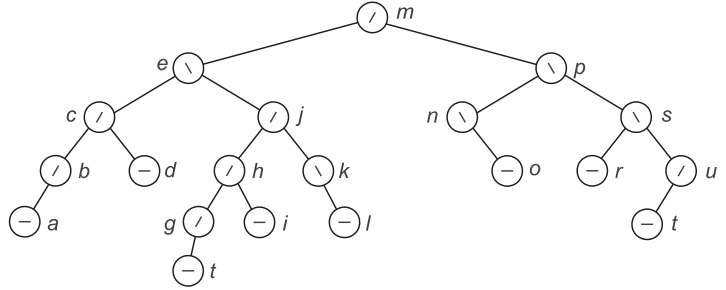
двойной  
поворот



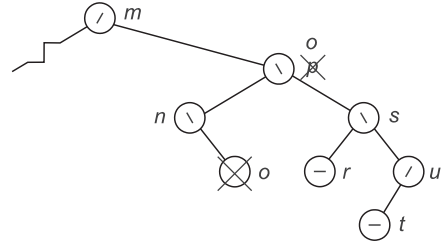
Случай 3с

Рис. 10.22 Некоторые случаи удаления из AVL-дерева

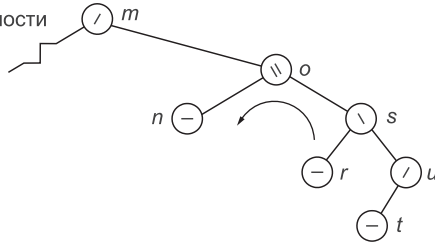
Начальное состояние



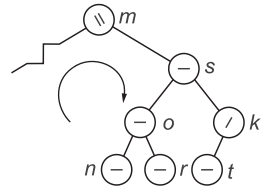
Удаляем p:



Корректируем факторы сбалансированности



Поворот влево:



Двойной поворот вправо вокруг m:

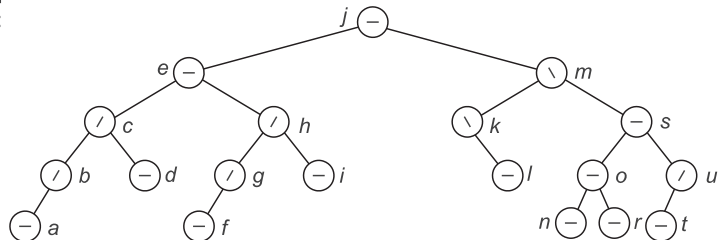


Рис. 10.23. Пример удаления из AVL-дерева

### 10.4.4. Высота AVL-дерева

анализ худшего случая

Определение высоты среднего AVL-дерева представляет собой весьма трудную задачу, и, соответственно, трудно определить, сколько в среднем шагов требуется для выполнения алгоритмов этого раздела. Однако найти, что происходит в худшем случае, оказывается значительно проще, и результаты исследований показывают, что поведение AVL-деревьев в худшем случае не хуже, чем поведение случайных деревьев. Эмпирические же наблюдения показывают, что среднее поведение AVL-деревьев оказывается значительно лучшим, чем у случайных деревьев, почти таким же хорошим, как можно ожидать от идеально сбалансированного дерева.

Чтобы подойти к определению максимальной высоты, которую может иметь AVL-дерево с  $n$  узлами, мы можем задать другой вопрос — каково минимальное число узлов, которое может иметь AVL-дерево высоты  $h$ . Если  $F_h$  есть такое дерево, и левое и правое поддеревья его корня есть  $F_l$  и  $F_r$ , тогда одно из поддеревьев  $F_l$  или  $F_r$  должно иметь высоту  $h - 1$  (пусть это будет высота  $F_l$ ), а другое будет иметь высоту  $h - 1$  или  $h - 2$ . Поскольку  $F_h$  имеет минимальное число узлов среди AVL-деревьев высоты  $h$ , отсюда следует, что  $F_l$  должно иметь минимальное число узлов среди AVL-деревьев высоты  $h - 1$  (т. е.  $F_l$  имеет форму  $F_{h-1}$ ), а  $F_r$  должно иметь высоту  $h - 2$  с минимумом узлов (так что  $F_r$  имеет форму  $F_{h-2}$ ).

деревья Фибоначчи

Деревья, построенные согласно приведенному выше правилу, и которые, следовательно, максимально разрежены среди всех AVL-деревьев, называются **деревьями Фибоначчи**. Несколько первых деревьев Фибоначчи представлены на рис. 10.24.

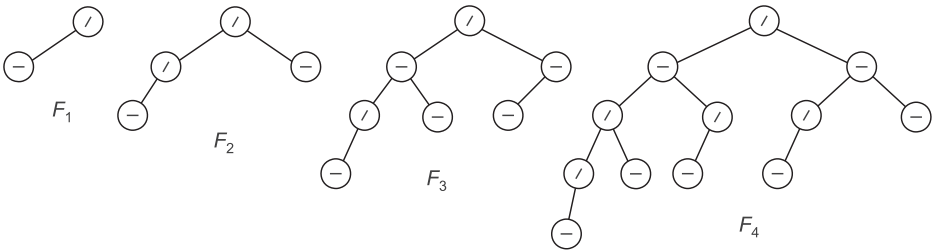


Рис. 10.24. Деревья Фибоначчи

Если мы обозначим через  $|T|$  число узлов в дереве  $T$ , то нетрудно написать (считая корень и поддеревья) рекуррентное отношение

$$|F_h| = |F_{h-1}| + |F_{h-2}| + 1,$$

где  $|F_0| = 1$ , а  $|F_1| = 2$ . Прибавив 1 к обеим сторонам, получим, что числа  $|F_h| + 1$  удовлетворяют определению чисел Фибоначчи (см. раздел А.4) с индексами, измененными на 3. Воспользовавшись оценкой чисел Фибоначчи, данной в разделе А.4, получим, что

$$|F_h| + 1 \approx \frac{1}{\sqrt{5}} \left[ \frac{1 + \sqrt{5}}{2} \right]^{h+2}$$

высота деревьев  
Фибоначчи

Далее, взяв логарифмы от обеих сторон и отбросив все, кроме самых больших членов, мы решаем это уравнение относительно  $h$ . Приблизительный результат таков:

$$h = 1.44 \lg |F_h|.$$

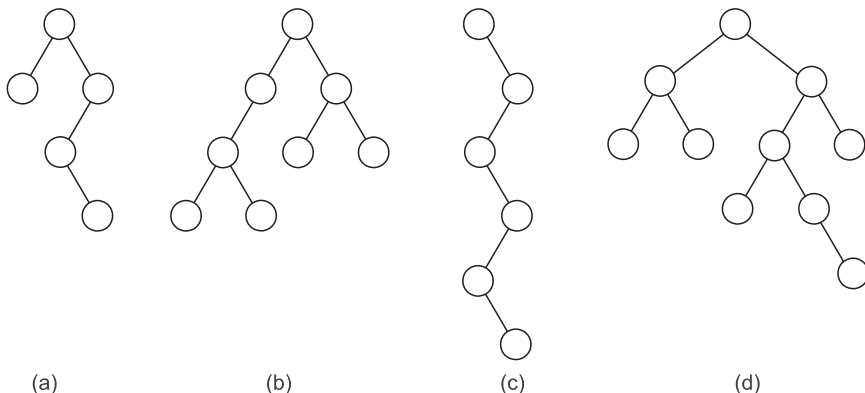
границы худшего  
случая

Это означает, что максимально разреженное из всех AVL-деревьев дерево с  $n$  узлами имеет высоту, приблизительно равную  $1.44 \lg n$ . Идеально сбалансированное двоичное дерево с  $n$  узлами имеет высоту около  $\lg n$ , а вырожденное дерево может иметь высоту вплоть до  $n$ . Таким образом, для алгоритмов обработки AVL-деревьев можно гарантировать, что они будут выполняться за время, не более чем на 44 процента больше, чем оптимальное время. На практике AVL-деревья имеют заметно лучшую производительность. Можно показать, что даже для деревьев Фибоначчи, которые представляют собой худший случай AVL-деревьев, среднее время поиска только на 4 процента больше оптимального. Большинство AVL-деревьев далеко не так разрежены, как деревья Фибоначчи, и разумно ожидать, что среднее время поиска для среднего AVL-дерева весьма близко к оптимальному. Действительно, эмпирические исследования показывают, что среднее число сравнений при больших  $n$  приблизительно равно  $\lg n + 0.25$ .

наблюдения для  
среднего случая

### Упражнения 10.4

**Е1.** Определите, какие из изображенных ниже деревьев являются AVL-деревьями. Для тех, которые таковыми не являются, найдите все узлы, в которых нарушаются требования к AVL-деревьям.

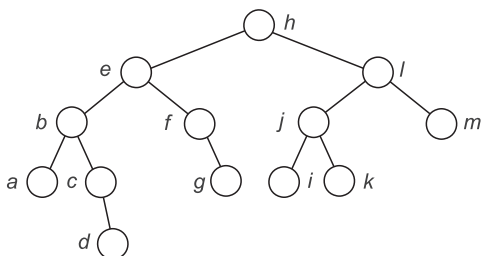


**Е2.** Для каждой из приведенных ниже последовательностей ключей включите ключи в указанном порядке, чтобы построить из них AVL-дерево.

- (a) A, Z, B, Y, C, X.
- (b) A, B, C, D, E, F.
- (c) M, T, E, A, Z, G, P.
- (d) A, Z, B, Y, C, X, D, W, E, V, F.
- (e) A, B, C, D, E, F, G, H, I, J, K, L.
- (f) A, V, L, T, R, E, I, S, O, K.



- Е3.** Удалите из AVL-дерева каждый из ключей, включенных в него в упражнении Е2 в порядке LIFO (ключ, включенный последним, удаляется первым).
- Е4.** Удалите из AVL-дерева каждый из ключей, включенных в него в упражнении Е2 в порядке FIFO (ключ, включенный первым, удаляется первым).
- Е5.** Начните с изображенного ниже AVL-дерева и удалите каждый из указанных ниже ключей. Каждое удаление выполните независимо, каждый раз начиная с исходного дерева.
- |       |       |       |
|-------|-------|-------|
| (a) k | (d) a | (f) m |
| (b) c | (e) g | (g) h |
| (c) j |       |       |



- Е6.** Напишите функцию, которая возвращает высоту AVL-дерева путем трассировки только одного пути к листу, без исследования всех узлов дерева.
- Е7.** Напишите процедуру, возвращающую указатель на самый левый лист, ближайший к корню в непустом AVL-дереве.
- Е8.** Докажите, что число (одинокных или двойных) поворотов, выполняемых при удалении ключа из AVL-дерева, не может превысить половину высоты дерева.

### Программные проекты 10.4

- P1.** Напишите Pascal-процедуру для удаления узла из AVL-дерева, следуя описанному в тексте порядку шагов.
- P2.** Подставьте процедуры для AVL-включения и удаления в управляемую с помощью меню демонстрационную программу для двоичных деревьев поиска (раздел 10.2), получив таким образом демонстрационную программу для AVL-деревьев.
- P3.** Воспользуйтесь AVL-деревьями для реализации операций над гардеробом из раздела 7.3. Протестируйте ваш пакет с помощью демонстрационной программы гардероба, а затем запустите программу определения процессорного времени с пакетом для AVL-деревьев, сравнив полученные результаты с данными для других реализаций.
- P4.** Подставьте процедуру для AVL-включения в проект P6 из раздела 10.2 (программа извлечения информации). Сравните производительность AVL-деревьев с обычными двоичными деревьями поиска для различных комбинаций входных текстовых файлов.

## 10.5. Скошенные деревья: самонастраивающиеся структуры данных

### 10.5.1. Введение

медицинские  
карты пациентов  
госпиталя

Рассмотрим проблему хранения и поддержки медицинских карт пациентов в госпитале. Медицинские карты пациентов, находящихся в госпитале, исключительно активны, поскольку непрерывно используются для получения из них необходимой врачам и медсестрам информации и обновления этой информации. Когда пациент покидает госпиталь, его медицинская карта становится менее активной, но все еще время от времени требуется домашнему врачу пациента и другим лицам. Если в дальнейшем пациент снова поступает в госпиталь, тогда его медицинская карта сразу же опять становится исключительно активной. Поскольку такое повторное поступление может произойти в результате несчастного случая, то даже неактивная карта должна быть доступна незамедлительно, а не храниться в виде архивных файлов с большим временем доступа.

время доступа

Если мы для хранения медицинских карт пациентов используем обычное дерево двоичного поиска, или даже AVL-дерево, тогда записи о вновь поступивших пациентах будут помещены в позиции листьев, далеко от корня, и, следовательно, время доступа к ним будет большим. Мы же, наоборот, хотели бы хранить дальше от корня, в листьях или около них, неактивные записи, а записи о вновь поступивших больных или другие часто требуемые записи держать поближе к корню. Но мы не можем выключить систему записей госпиталя даже на час, чтобы перестроить дерево, придав ему желательную форму. Чтобы решить эту проблему, нам надо преобразовать дерево в самонастраивающуюся структуру, которая будет *автоматически* изменять свою форму и переносить записи, к которым осуществляется более частый доступ, ближе к корню, в то время как неактивные записи будут постепенно перемещаться ближе к листьям.

самонастраиваю-  
щиеся деревья

**Скошенные деревья** представляют собой деревья двоичного поиска, которые удовлетворяют нашим требованиям, обладая способностью самонастройки весьма замечательным образом: каждый раз, когда мы обращаемся к узлу дерева, либо ради включения, либо ради извлечения, мы осуществляем над деревом радикальную хирургическую операцию, поднимая узел, к которому только что произошло обращение, на самый верх, так что он становится корнем модифицированного дерева. Другие узлы отталкиваются с дороги, чтобы дать место для этого нового корня. Узлы, к которым осуществляется частый доступ, будут часто подниматься наверх и становиться по очереди корнем дерева, поэтому они никогда не сместятся слишком далеко от верхней позиции. Неактивные узлы, наоборот, будут постепенно проталкиваться все дальше и дальше от корня.

Вполне возможно, что скошенное дерево станет сильно несбалансированным, в результате чего доступ к узлам дерева станет весьма затратным. Ниже мы покажем, однако, что при длительном использовании скошенных деревьев не теряют своей эффективности, и гарантированно не требуют заметно больше операций, чем AVL-деревья. Примененное в этом доказательстве аналитическое средство носит название *амортизационного анализа алгоритмов*, поскольку, как и при расчете страховых взносов, несколько затратных случаев усредняются с большим количеством дешевых случаев, в результате чего на протяжении большого числа операций достигается превосходная производительность.

Мы выполняем радикальные хирургические операции над скошенными деревьями посредством поворотов приблизительно так же, как мы это делали для AVL-деревьев, но только с большим числом операций поворотов для каждого включения или извлечения из дерева. В сущности, повороты выполняются по всему пути от корня до последнего использованного узла. Давайте теперь обсудим, как выполняется вся эта работа.

## 10.5.2. Шаги скашивания дерева

Когда в дереве двоичного поиска выполняется одиночный поворот вроде того, что было показано на рис. 10.19, некоторые узлы перемещаются выше по дереву, а некоторые — ниже. При левостороннем повороте родительский узел перемещается вниз, а его правый дочерний узел поднимается на один уровень выше. Двойной поворот, как это видно из рис. 10.20, состоит из двух одиночных поворотов, причем один узел перемещается на два уровня выше, в то время как остальные либо остаются на своих уровнях, либо смещаются вниз. Начав с узла, к которому только что произошло обращение, и двигаясь вверх к корню, мы можем на каждом шаге выполнять операции поворотов, тем самым перемещая наш узел вверх до самого корня. Такой метод приведет к желаемому результату, сделав узел с последним обращением корневым, но анализ показывает, что производительность дерева, усредненная по многим операциям доступа, может оказаться невысокой.

Ключевая идея скашивания заключается в перемещении узла с последним обращением на *два* уровня вверх в каждом шаге. Прежде всего введем простую терминологию. Рассмотрим путь, идущий от корня *вниз* к узлу с последним обращением. Каждый раз, когда мы, идя вниз по этому пути, перемещаемся влево, будем говорить, что мы выполняем *«зиг»*, а каждый раз, когда мы перемещаемся вправо — *«заг»*. Перемещение на два шага влево (при продвижении вниз) тогда будет называться *«зиг-зиг»*; перемещение на два шага вправо — *«заг-заг»*; влево, а затем вправо — *«зиг-заг»*; а вправо и затем влево — *«заг-зиг»*. Этими четырьмя случаями исчерпываются все возможности переместиться вниз на два шага. Если, однако, длина пути является нечетным числом, в конце потребуется еще один шаг, либо *«зиг»* (перемещение влево), либо *«заг»* (перемещение вправо).

Операции поворотов, выполняемые в процессе скашивания дерева для каждого из перемещений «зиг-зиг», «зиг-заг» и «зиг», показаны на рис. 10.25. Случай «зиг-заг» идентичен двойному повороту в AVL-дереве, а случай «зиг» идентичен одиночному повороту. Случай «зиг-зиг», однако, не совпадает с подъемом узла с помощью двух одиночных поворотов, как это показано на рис. 10.26.

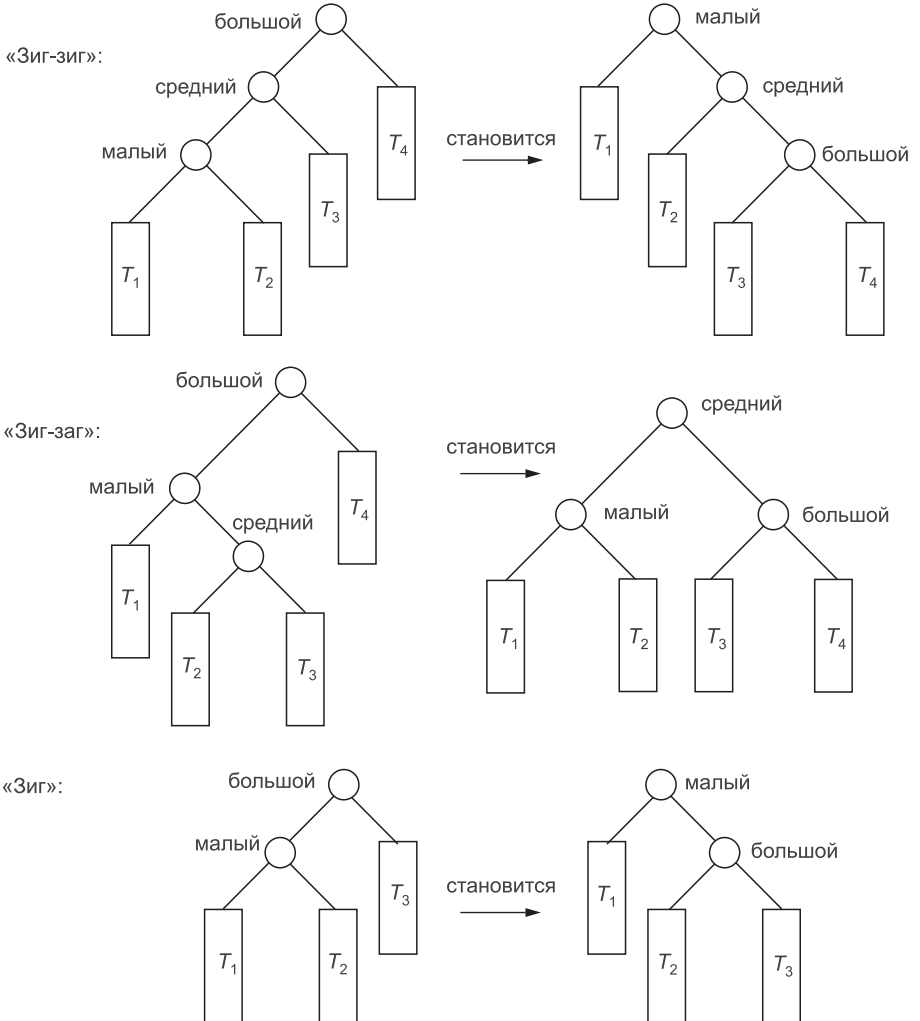


Рис. 10.25. Повороты при скашивании дерева

Будем держать все эти рассуждения в памяти, рассматривая пример, приведенный на рис. 10.27.

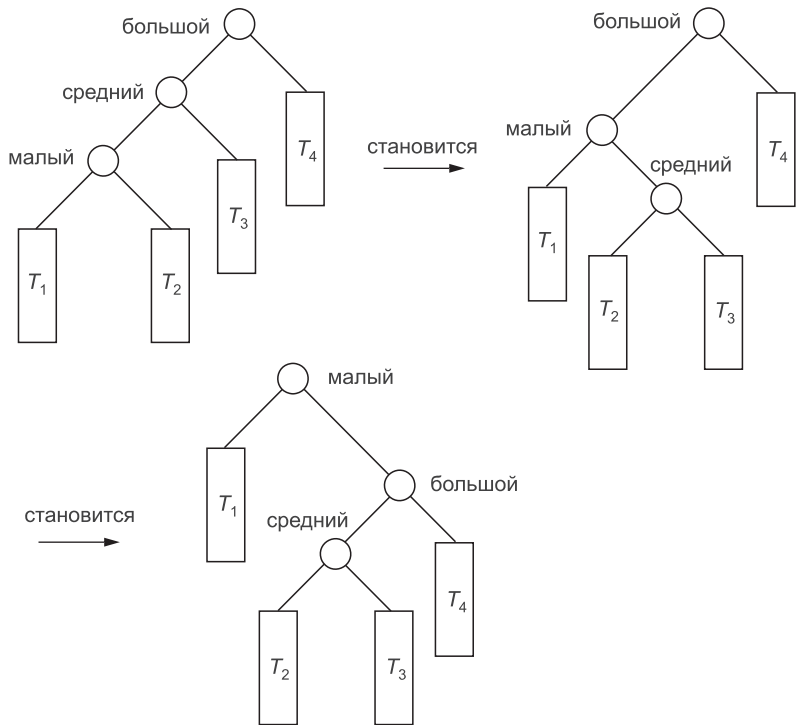


Рис. 10.26. «Зиг-зиг» заменен одиночными поворотами

Скашивание  
в узле c:

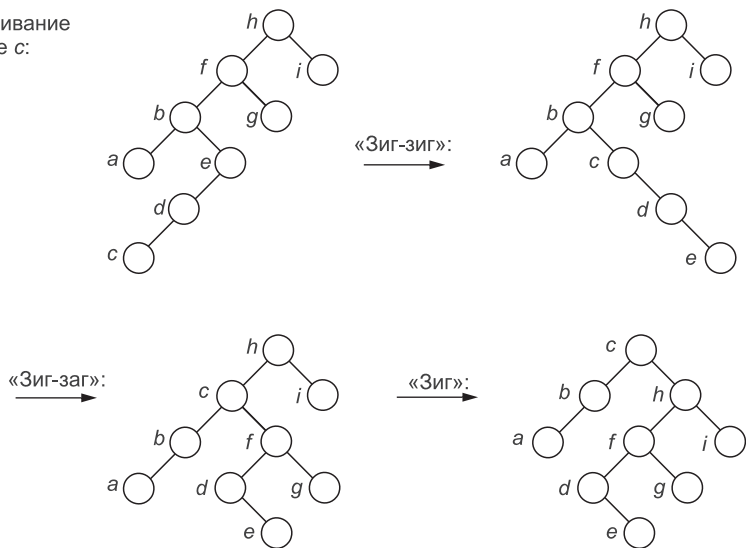


Рис. 10.27. Пример скашивания

Начнем с самого левого дерева и скашивания в узле  $c$ . Путь от корня к  $c$  проходит через  $h, f, b, e, d, c$ . Путь от  $e$  до  $d$  и далее до  $c$  представляется как «зиг-зиг» (влево-влево), поэтому мы выполняем поворот «зиг-зиг», получая второе дерево, изображенное на рис. 10.27. Путь от  $f$  через  $b$  к  $c$  выглядит как «зиг-заг» (влево-вправо), и результирующий поворот «зиг-заг» дает нам третье дерево. В этом дереве  $c$  отстоит от корня всего на один шаг, находясь с левой стороны, и поворот «зиг» приводит нас к конечному дереву, изображенному на рис. 10.27.

скашивание  
снизу вверх

В этом примере мы выполняли скашивание *снизу вверх*, начав с узла  $c$  с последним обращением и двигаясь вверх по цепочке к корню. При проработке этих примеров вручную такой метод кажется естественным, поскольку, выполнив поиск сверху вниз, хочется повернуть назад и начать скашивать дерево снизу вверх до его вершины. В результате последний одиночный «зиг» или «заг», выполняемый в случае необходимости, придется на вершину дерева. Однако, при написании компьютерной программы, оказывается проще скашивать дерево сверху вниз, в процессе поиска требуемого узла. В этом случае одиночный «зиг» или «заг» выполняется в конце процесса скашивания, у дна дерева. В результате, если вы запустите для пробных деревьев процедуру скашивания, которую мы вскоре разработаем, она не всегда даст те же результаты, что и ваше «ручное» скашивание, выполняемое снизу вверх.

скашивание  
сверху вниз

### 10.5.3. Алгоритм скашивания

Мы разработаем только одну процедуру скашивания, которую можно будет использовать и для извлечения, и для включения. Получив ключ-мишень, процедура осуществит поиск ключа по дереву, выполняя по ходу поиска скашивание. Если ключ обнаруживается, соответствующая запись извлекается; если нет, процедура включает новую запись. В любом случае узел с ключом-мишенью оказывается в корне дерева.

обычное двоичное  
дерево поиска

Объявления для двоичного дерева поиска со скашиванием идентичны объявлениям для обычного двоичного дерева. Скошенное дерево *не* учитывает свою высоту и *не* использует фактор сбалансированности, как это делают AVL-деревья.

узел-сигнальная  
метка null↑

Вспомним, что *узел-сигнальная метка* для двоичного дерева поиска представляет собой дополнительный узел, который мы будем называть null↑, и который мы помещаем на дне дерева, после чего все связи *null* мы заменяем на связи null (см. рис. 10.12). Узел-сигнальная метка часто позволяет избежать проверки пустых поддеревьев. Для процедуры скашивания это обстоятельство оказывается весьма ценным. Нам не надо беспокоиться о пустых поддеревьях; поддерево, которое в другом случае было бы пустым, всегда содержит узел null↑. Поэтому мы напишем нашу процедуру только для двоичных деревьев поиска с сигнальной меткой.

расщепление  
дерева на три  
части

По ходу процесса скашивания сверху вниз мы расщепляем дерево на три части. Одна часть содержит узлы, между которыми находится мишень, если, конечно, она имеется. Эта *центральная* часть дерева первоначально занимает все дерево. Поиск заканчивается тем, что корень центрального дерева оказывается узлом, содержащим мишень

(при ее наличии), или pull-узлом, если мишень в дереве отсутствует. Слева от этого центрального дерева пока еще не просмотренных узлов находится вторая часть, поддереву, ключи которого, как мы знаем, меньше мишени; мы будем называть его поддеревом *меньших ключей*. Справа от центрального дерева лежит третья часть, поддереву *больших ключей*, т. е. узлов, ключи которого, как мы знаем, больше мишени. Первоначально оба поддерева, и меньших ключей, и больших ключей, включают лишь сигнальные метки null↑.

переменные

Наша процедура использует, помимо своих параметров (корень дерева и ключ-мишень), четыре переменные. Переменная current будет указывать на корень центрального поддерева еще не просмотренных узлов. Узлы отщепляются таким образом, что *каждый* ключ в поддереве меньших ключей должен быть меньше *каждого* ключа в центральном поддереве; чтобы отщеплять следующие ключи от центрального дерева, мы должны иметь указатель lastsmall на наибольший (т. е. самый правый) узел в поддереве меньших ключей. Аналогично нам нужен и указатель firstlarge на наименьший (т. е. самый левый) узел в поддереве больших ключей.

По мере выполнения поиска мы сравниваем мишень с ключом в корне current↑ центрального поддерева. Предположим, что мишень оказалась больше этого ключа. Тогда поиск переместится вправо, но нам надо взять current↑ и его левое поддерево и присоединить их к поддереву меньших ключей. Эта операция выполняется следующей процедурой.

```

procedure LinkLeft;                                { Связь влево }
{ Pre:  target > current↑.entry.key
  Post: Процедура перемещает current и его левое поддерево в дерево
           ключей, про которые известно, что они меньше target; в силу
           этого переназначает lastsmall на старый current, и перемещает
           current к его правому дочернему узлу.
  Uses: Изменяет lastsmall, его правый дочерний узел и current как
           глобальные переменные; объявления должны быть сделаны
           только внутри TreeSplay. }

begin                                                { процедура LinkLeft }
  lastsmall↑.right := current;
  lastsmall := current;
  current := current↑.right
end;                                                { процедура LinkLeft }

```

Поскольку эта процедура изменяет правый дочерний для current узел, нам нужна четвертая переменная, child, которая будет говорить нам, с какого узла продолжить поиск мишени.

Аналогичная процедура LinkRight подсоединяет current↑ и его правое поддерево к поддереву больших ключей; в этом случае поиск и указатель current перемещаются к его левому дочернему узлу.

шаги скашивания

Теперь мы подошли к самым шагам скашивания, которые выполняются на удивление просто. «Зиг» перемещает current к его левому дочернему узлу, а старый current↑ вместе с его правым поддеревом перемещает в поддерево больших ключей (правое); это в точности задача процедуры

LinkRight. Для перемещения «зиг-зиг» мы должны сначала выполнить поворот вправо, чтобы левый дочерний узел узла `current↑` стал новым `current↑`, а затем оба узла связываются с поддеревом больших ключей. Другими словами, мы должны выполнить

RotateRight; LinkRight

где процедура RotateRight осуществляет обычный поворот вправо:

```

procedure RotateRight;                                { Поворот вправо }
{ Pre:   current указывает на узел с сигнальной меткой в двоичном
           дереве поиска
  Post:   Поворачивает вправо край, прилегающий к current, и его левый
           дочерний узел. }
var leftchild: treepointer;
begin                                                { процедура RotateRight }
  leftchild := current↑.left
  current↑.left := leftchild↑.right
  leftchild↑.right := current;
  current := leftchild
end;                                                { процедура RotateRight }

```

Для перемещения «зиг-заг» мы должны переместить `current` вместе с его правым поддеревом в поддерево больших ключей, а его левый дочерний узел вместе с его левым поддеревом переместить в поддерево меньших ключей. Это достигается выполнением

LinkRight; LinkLeft.

Три симметричных случая, «заг», «заг-заг» и «заг-зиг», аналогичны.

завершение

Когда поиск заканчивается, корнем центрального поддерева становится либо узел-мишень, либо узел-сигнальная метка. Если мишень найдена, она становится корневым узлом всего дерева, однако перед этим его левое и правое поддерева, про которые мы знаем, что они принадлежат поддеревьям меньших и больших ключей, соответственно, должны быть перемещены туда. Если же найдена сигнальная метка, создается новый корневой узел.

Теперь левое и правое поддерева нового корня должны быть поддеревьями меньших и больших ключей. Как нам найти корни этих поддеревьев, ведь мы храним только указатели на их самый правый и самый левый узлы, соответственно? Мы должны вспомнить, что происходило в начале поиска. Первоначально оба эти поддерева содержали только узлы `null↑`. Когда узел (и поддерево) объединяются с поддеревом меньших ключей, они подсоединяются к нему справа путем изменения `lastsmall↑.right`. Поскольку `lastsmall` был первоначально `null`, мы можем теперь, в конце поиска, найти первый узел, включенный в поддерево меньших ключей, а отсюда и его корень, просто как `null↑.right`. Аналогично, `null↑.left` указывает на корень поддерева больших ключей.

После всей этой предварительной работы мы можем наконец написать главную процедуру для извлечения и включения узлов в двоичное дерево поиска со скашиванием.



```

procedure TreeSplay (var root: treepointer; target: keytype);
                                { Скашивание дерева }
{ Pre:  Уже создано скошенное дерево, на которое указывает
        параметр root.
  Post: Дерево было скошено вокруг ключа-мишени target. Если узел
        с ключом target был в дереве, теперь он стал корнем. Если
        нет, тогда в качестве корня был создан новый узел с ключом
        target. }

var
  current,                                { текущая позиция в дереве }
  child,                                { один из дочерних от current узлов }
  lastsmall,                            { наибольший из ключей, меньших, чем target }
  firstlarge: treepointer;              { наименьший из ключей, больших, чем target }
{ Включите объявления процедур LinkLeft, LinkRight, RotateLeft и
  RotateRight. }
begin                                { процедура TreeSplay }
  null↑.entry.key := target;            { установим сигнальную метку для поиска }
  current := root;
  lastsmall := null;
  firstlarge := null;
  while current↑.entry.key <> target do
    if current↑.entry.key < target then
      begin
        child := current↑.right;
        if target = child↑.entry.key then
          LinkLeft                                { перемещение "заг" }
        else if target > child↑.entry.key then
          begin RotateLeft; LinkLeft end            { перемещение "заг-заг" }
        else
          begin LinkLeft; LinkRight end            { перемещение "заг-зиг" }
        end
      else begin
        child := current↑.left;
        if target = child↑.entry.key then
          LinkRight                                { перемещение "зиг" }
        else if target < child↑.entry.key then
          begin RotateRight; LinkRight end          { перемещение "зиг-зиг" }
        else
          begin LinkRight; LinkLeft end            { перемещение "зиг-заг" }
        end;
      if current = null then begin { поиск неуспешен: создадим новый корень }
        writeln('Мишень была включена в качестве корня дерева. ');
        new(root);
        root↑.entry.key := target;
        root↑.left := null;
        root↑.right := null;
        current := root
      end
      else
        writeln('Мишень была найдена; теперь она стала корнем дерева. ');
        lastsmall↑.right := current↑.left;        { переместим оставшиеся
                                                    центральные узлы }
        firstlarge↑.left := current↑.right;
        current↑.right := null↑.left;              { корень поддеревы больших ключей }
        current↑.left := null↑.right;              { корень поддеревы меньших ключей }
      end

```

```

root := current;                                     { определим новый корень }
null↑.entry.key := chr(0);                           { удалим target из сигнальной метки }
null↑.left := null;                                  { установим заново стандартное
                                                    использование сигнальной метки }

null↑.right := null
end;                                                  { процедура TreeSplay }

```

С учетом всех наших рассуждений и решений, приведенный алгоритм оказывается весьма тонким и изощренным в плане манипуляции указателями и экономного использования ресурсов.

## 10.5.4. Амортизационный анализ алгоритмов: введение

Теперь нам хотелось бы исследовать поведение процедур скашивания при усреднении по большому числу операций, но перед тем, как приступить к этой работе, мы рассмотрим амортизационный анализ алгоритмов на более простых примерах.

### 1. Введение

определение

До сих пор мы рассматривали два рода анализа алгоритмов: анализ *худшего случая* и анализ *среднего случая*. И для того, и для другого анализа мы брали одиночное событие или одиночную ситуацию и пытались определить, какой объем работы выполняет алгоритм при обработке этого события или этой ситуации. Амортизационный анализ отличается от этих двух видов анализа тем, что он рассматривает не одиночное изолированное событие, а *последовательность* событий. Амортизационный анализ дает оценку затрат для *худшего случая* длинной последовательности событий.

множественные  
выталкивания

Вполне может получиться, что одиночное событие в последовательности событий влияет на затраты на обработку последующих событий. Некоторая задача может быть сложной и затратной при ее выполнении, но она может оставить структуру данных в таком состоянии, что последующие операции по обработке этих данных станут существенно проще. Рассмотрим, например, стек, куда можно сразу протолкнуть любое количество элементов, как и вытолкнуть сразу любое количество элементов. Если в стеке содержатся  $n$  элементов, тогда затратами на множественную операцию выталкивания в наихудшем случае будет, очевидно,  $n$ , поскольку сразу, возможно, выталкиваются все элементы. Если, однако, почти все элементы уже вытолкнуты из стека (в одной затратной операции выталкивания), тогда последующие операции выталкивания не могут быть затратными, поскольку в стеке осталось лишь несколько элементов. Пусть затраты на выталкивание 0 элементов составляют 0. Тогда если мы начинаем с  $n$  элементов в стеке и выполняем последовательно из  $n$  множественных выталкиваний, амортизационные затраты в пересчете на одно выталкивание будут равны всего лишь 1, даже несмотря на то, что затраты в худшем случае составляют  $n$ . Это происходит потому, что  $n$  множественных выталкиваний могут вместе удалить из стека только  $n$  элементов, поэтому их общие затраты не могут превысить  $n$ .

амортизация

В мире финансов *амортизация* означает распределение больших затрат на значительный период времени, как это происходит, когда ипотека распределяет стоимость дома (с процентами) на много помесечных платежей. Бухгалтеры амортизируют большие капитальные затраты, учитывая приносящую доход деятельность, которая стала возможной благодаря этим капитальным затратам. Статистики страховых компаний амортизируют случаи больших выплат, учитывая общее число застрахованных клиентов.

## 2. Анализ среднего случая и амортизационный анализ

Амортизационный анализ не есть то же самое, что анализ среднего случая, поскольку первый рассматривает последовательность *связанных* ситуаций, а второй — все возможные *независимые* ситуации. Для методов сортировки мы выполняли анализ среднего случая, рассматривая все возможные случаи. Не имеет смысла говорить о сортировке одного и того же списка дважды, и поэтому амортизационный анализ обычно неприменим к сортировке.

Мы, однако, можем привести пример, когда это утверждение оказывается несправедливым. Рассмотрим список, который был сначала упорядочен, а затем, после использования этого списка в течение определенного времени, в некоторую случайную позицию списка был включен новый элемент. После определенного периода использования список снова сортируется. Позже другой элемент включается в случайном месте и т. д. Какой метод сортировки мы используем в этом случае? Если основываться на анализе среднего случая, мы выберем алгоритм быстрой сортировки. Если мы предпочитаем основываться на худшем случае, мы выберем сортировку слиянием или пирамидальную сортировку с гарантированной производительностью  $O(n \lg n)$ . Амортизационный анализ, однако, приведет нас к сортировке включением: после того, как список был упорядочен и в его случайную позицию включается новый элемент, сортировка включением поместит этот элемент в его правильную позицию с производительностью  $O(n)$ . Поскольку список почти упорядочен, алгоритм быстрой сортировки (с наилучшей производительностью для среднего случая) скорее всего приведет к наихудшей реальной производительности, поскольку неудачный выбор опорного ключа может свести сортировку почти к худшему случаю.

## 3. Просмотр дерева

В качестве другого примера рассмотрим симметричный просмотр двоичного дерева, когда мы измеряем затраты посещения одной вершины числом ветвей, просмотренных до этой вершины от последней посещенной. В лучшем случае затраты составляют 1, если мы перемещаемся от вершины к одной из ее дочерних вершин или, наоборот, к родительской. Затраты худшего случая для дерева с  $n$  вершинами составят  $n - 1$ , как это может быть для дерева, представляющего собой одну длинную левую цепочку; в этом случае, чтобы дойти до первой (самой левой) вершины, потребуется пройти  $n - 1$  ветвей. В такой цепочке, однако, все оставшиеся вершины достигаются всего лишь за один шаг, по мере того, как про-

смотр перемещается от каждой вершины к ее родителю. В полностью сбалансированном двоичном дереве размера  $n$  некоторые вершины требуют до  $\lg n$  шагов, а другие всего 1 шаг.

Если, однако, мы амортизируем затраты по всему двоичному дереву, тогда затраты на перемещение от одной вершины к следующей будут меньше 2. Чтобы убедиться, что это так, заметьте прежде всего, что каждое двоичное дерево с  $n$  вершинами имеет в точности  $n - 1$  ветвей, поскольку каждая вершина за исключением корня имеет лишь одну спускающуюся к ней ветвь. Полный просмотр дерева проходит по каждой ветви дважды, одни раз вниз и другой — вверх. (Сюда мы включили перемещение вверх к корню после посещения последней, самой правой, вершины.) Таким образом, суммарное число шагов при полном просмотре составляет  $2(n - 1)$ , а амортизированное число шагов от одной вершины к следующей равно  $2(n - 1)/n < 2$ .

#### 4. Кредитовая функция: усреднение затрат

Представьте себе, что вы работаете над домашним бюджетом. Если вы служащий, тогда (как вы надеетесь) ваш доход обычно достаточно стабилен от месяца к месяцу. Однако этого нельзя сказать о ваших затратах. В некоторые месяцы вам надо много платить за страховку, или за обучение, или за крупные покупки. В другие месяцы таких экстраординарных затрат не предвидится. Тогда, чтобы ваш банковский счет всегда оставался положительным, вам придется в течение месяцев с низкими затратами накапливать деньги, чтобы иметь возможность оплатить все эти крупные счета, когда до них дойдет очередь. В начале месяца, в котором ожидаются крупные выплаты, ваш счет в банке велик. После уплаты по счетам остаток вклада существенно уменьшится, но это не портит вам настроения, поскольку вы должны теперь меньше денег.

кредитовая  
функция

Мы хотим приложить эту идею к анализу алгоритмов. Для этого мы вводим функцию, назвав ее *кредитовой функцией* (или *кредитовым остатком*), которая ведет себя аналогично остатку банковского вклада в семье со взвешенной бюджетной политикой. Кредитовая функция выбирается таким образом, чтобы она имела большое значение, если очередная операция имеет затратный характер, и меньшее значение, если следующая операция может быть выполнена быстро, т. е. с малыми затратами. Кредитовая функция помогает нам справиться с потерями от затратных операций, поскольку при операциях с малыми затратами мы откладываем больше, чем фактически тратим, используя образующийся избыток для увеличения значения кредитовой функции с целью использования накопленного позже.

Опишем нашу идею более точно. Пусть у нас имеется последовательность из  $m$  операций над структурой данных и пусть величина  $t_i$  составляет *фактические* затраты операции при  $1 \leq i \leq m$ . Будем считать, что значения нашей кредитовой функции равны  $c_0, c_1, \dots, c_m$ , где  $c_0$  есть кредитовый остаток перед первой операцией, а  $c_i$  есть кредитовый остаток после операции  $i$  при  $1 \leq i \leq m$ . При таких обозначениях мы можем дать следующее фундаментальное определение:

**Определение**

*Амортизированные затраты*  $a_i$  каждой операции определяются как

$$a_i = t_i + c_i - c_{i-1}$$

для  $i = 1, 2, \dots, m$ , где  $t_i$  и  $c_i$  были определены выше.

Из приведенного выражения следует, что амортизированные затраты представляют собой фактические затраты плюс величина, на которую наш кредитовый остаток изменился в процессе данной операции.

Не следует забывать, что наш кредитовый остаток представляет собой всего лишь средство учета. Мы можем использовать по желанию любую кредитовую функцию, хотя некоторые из них лучше чем другие. Наша цель — помочь в определении кредитовой политики, и ее можно определить таким образом:

цель

*Выбирайте кредитовую функцию  $c_i$  таким образом, чтобы амортизированные затраты были по возможности одинаковы, независимо от того, как изменяются фактические затраты  $t_i$ .*

Теперь используем амортизированные затраты для вычисления фактических затрат последовательности  $m$  операций. Фундаментальное определение можно преобразовать как  $t_i = a_i + c_{i-1} - c_i$ , и тогда суммарные фактические затраты составят

$$\sum_{i=1}^m t_i = \sum_{i=1}^m (a_i + c_{i-1} - c_i) = \left( \sum_{i=1}^m a_i \right) + c_0 - c_m.$$

За исключением первого и последнего членов, все кредитовые остатки взаимно уничтожаются и не входят в окончательную формулу. Для последующих ссылок мы определим полученный результат в виде леммы:

**Лемма 10.5**

*Суммарные фактические затраты и суммарные амортизированные затраты последовательности  $m$  операций над структурой данных соотносятся как*

$$\sum_{i=1}^m t_i = \left( \sum_{i=1}^m a_i \right) + c_0 - c_m$$

Нашей целью является такой выбор кредитовой функции, чтобы  $a_i$  были приблизительно одинаковы; тогда будет нетрудно вычислить правую часть этого уравнения и, следовательно, фактические затраты последовательности  $m$  операций.

Суммы, подобные приведенной выше, где члены имеют попеременно знаки плюс и минус, из-за чего они при сложении уничтожают друг друга, называются **телескопическими суммами**, поскольку они напоминают нам игрушечный (или портативный) телескоп, сделанный из нескольких коротких трубок, которые входят друг в друга, но могут быть раздвинуты для образования одной длинной трубы телескопа.

телескопические  
суммы

**5. Инкремент целых двоичных чисел**

Давайте подытожим сказанное, рассмотрев еще один простой пример, после чего будет уместно приложить высказанные идеи к доказательству фундаментальной и удивительной теоремы, касающейся скошенных деревьев.

Рассматриваемый здесь пример представляет собой алгоритм непрерывного возрастания двоичных (по основанию 2) чисел на 1. Мы выполняем изменение двоичного числа с правой стороны; если текущий бит равен 1, мы изменяем его на 0 и перемещаемся влево до тех пор, пока мы не достигнем самого левого бита или не встретим бит 0, который мы изменяем на 1 и прекращаем перемещение. Затраты на выполнение этого алгоритма определяются как число битов (двоичных разрядов), которые подверглись изменению. Результат приложения этого алгоритма 16 раз к целому двоичному числу, состоящему из четырех разрядов, показан в трех левых столбцах рис. 10.28.

| <i>Шаг i</i> | Число   | $t_i$ | $c_i$ | $a_i$ |
|--------------|---------|-------|-------|-------|
|              | 0 0 0 0 |       | 0     |       |
| 1            | 0 0 0 1 | 1     | 1     | 2     |
| 2            | 0 0 1 0 | 2     | 1     | 2     |
| 3            | 0 0 1 1 | 1     | 2     | 2     |
| 4            | 0 1 0 0 | 3     | 1     | 2     |
| 5            | 0 1 0 1 | 1     | 2     | 2     |
| 6            | 0 1 1 0 | 2     | 2     | 2     |
| 7            | 0 1 1 1 | 1     | 3     | 2     |
| 8            | 1 0 0 0 | 4     | 1     | 2     |
| 9            | 1 0 0 1 | 1     | 2     | 2     |
| 10           | 1 0 1 0 | 2     | 2     | 2     |
| 11           | 1 0 1 1 | 1     | 3     | 2     |
| 12           | 1 1 0 0 | 3     | 2     | 2     |
| 13           | 1 1 0 1 | 1     | 3     | 2     |
| 14           | 1 1 1 0 | 2     | 3     | 2     |
| 15           | 1 1 1 1 | 1     | 4     | 2     |
| 16           | 0 0 0 0 | 4     | 0     | 0     |

**Рис. 10.28.** Затраты и амортизированные затраты последовательного увеличения двоичных чисел

Кредитовая функция в данном примере образуется как полное число единиц в двоичном числе. Очевидно, что число разрядов, подлежащих изменению, в точности на 1 больше числа единиц в правой части числа, так что чем больше там единиц, тем больше разрядов придется изменить. Теперь мы можем подсчитать с помощью нашей фундаментальной формулы амортизированные затраты каждого шага. Результаты показаны в последнем столбце рис. 10.28, причем оказывается, что затраты составляют 2 для каждого шага, кроме последнего, для которого они равны 0. Отсюда мы заключаем, что мы можем выполнять последовательный инкремент четырех двоичных разрядов с амортизированными затратами на изменение двух разрядов, хотя фактические затраты колеблются от одного до четырех.

### 10.5.5. Амортизационный анализ скашивания

После всех эти предварительных рассуждений и выкладок мы можем наконец использовать процедуру амортизационного анализа алгоритмов для определения объема работы, выполняемой нашим алгоритмом скошенного дерева в длинной последовательности извлечений и включений. В качестве меры фактической сложности мы примем глубину, на которой расположен в дереве искомый узел до скашивания. Все действия алгоритма — сравнение ключей и повороты — тесно связаны с этой глубиной. Так, например, число итераций главного цикла, выполняемого процедурой, составляет приблизительно половину этой глубины.

Прежде всего введем несколько простых обозначений. Пусть  $T$  представляет собой двоичное дерево поиска, над которым мы выполняем операции включения и извлечения с использованием скашивания. Пусть  $T_i$  определяет дерево  $T$ , как оно выглядит после шага  $i$  процесса скашивания, причем  $T_0 = T$ . Если  $x$  является произвольным узлом  $T_i$ , тогда под  $T_i(x)$  мы понимаем поддерево  $T_i$  с корнем в  $x$ , а под  $|T_i(x)|$  — число узлов в этом поддереве.

Мы предполагаем, что скашивание производится в узле  $x$ , при этом используется восходящее скашивание, поэтому первоначальная позиция  $x$  находится где-то в дереве  $T$ , но после  $m$  шагов скашивания  $x$  становится корнем  $T$ .

Для каждого шага  $i$  процесса скашивания и каждой вершины  $x$  в  $T$  мы определяем **ранг** узла  $x$  на шаге  $i$  как

$$r_i(x) = \lg |T_i(x)|.$$

Эта функция ранжирования ведет себя схоже с идеализированной высотой: она зависит от размеров поддерева с корнем  $x$ , а не от его высоты, но она указывает, какой была бы высота поддерева в случае его полной сбалансированности.

Если  $x$  является листом, тогда  $|T_i(x)| = 1$ , поэтому  $r_i(x) = 0$ . Узлы, близкие к краям дерева, имеют небольшие значения ранга; корень дерева обладает максимальным рангом.

Объем работы, выполняемой алгоритмом при включении или извлечении из поддерева, очевидно связан с высотой поддерева и, как мы можем надеяться, с рангом корня поддерева. Нам хотелось бы определить кредитовый остаток таким образом, чтобы большие и высокие деревья имели большое значение кредитового остатка, а короткие или небольшие деревья — меньшее значение, поскольку объем работы по скашиванию дерева растет с увеличением высоты дерева. Для достижения нашей цели мы используем функцию ранжирования. Фактически мы распределим кредитовый остаток дерева между всеми его вершинами, потребовав, чтобы всегда выполнялось следующее отношение:

#### Кредитовый инвариант

*Для каждого узла  $x$  дерева  $T$  и после каждого шага  $i$  скашивания узел  $x$  имеет кредитовый остаток, равный его рангу  $r_i(x)$ .*

кредитовый  
инвариант



Полный кредитовый остаток для дерева тогда определяется просто как сумма индивидуальных остатков для всех узлов дерева:

$$c_i = \sum_{x \in T_i} r_i(x).$$

Если дерево пусто или содержит лишь один узел, тогда его кредитовый остаток равен 0. По мере роста дерева его кредитовый остаток увеличивается, причем его величина должна отражать объем работы, которую необходимо выполнить для построения дерева. Вклад в кредитовый остаток дерева выполняется двумя способами.

- Мы вкладываем фактическую работу, выполняемую в конкретной операции. Мы уже решили, что будем оценивать эту работу, как одну единицу для каждого уровня, на который поднимается узел в процессе скашивания. В результате каждый шаг скашивания дает нам две единицы, за исключением шагов «зиг» или «заг», которые прибавляют по единице.
- Поскольку форма дерева в процессе скашивания изменяется, мы должны либо увеличивать, либо уменьшать кредитовый остаток, вложенный в дерево, чтобы для всех моментов времени сохранялся кредитовый инвариант. (Как мы уже отмечали в предыдущем разделе, кредитовый остаток является исключительно учетным средством, служащим для выравнивания затрат различных шагов.)

Все эти вклады суммируются в уравнении, определяющем амортизационную сложность  $a_i$  шага  $i$ :

$$a_i = t_i + c_i - c_{i-1},$$

где  $t_i$  есть реальная работа, а  $c_i - c_{i-1}$  описывает изменение кредитового остатка.

Теперь, опираясь на данные ранее определения и особенности метода скашивания, мы должны определить границы  $a_i$ , которые, в свою очередь, позволят нам найти затраты на весь процесс скашивания, амортизированные по последовательности извлечений и включений.

Прежде всего, нам потребуются некоторые предварительные математические выкладки.

#### Лемма 10.6

*Если  $\alpha$ ,  $\beta$  и  $\gamma$  суть положительные действительные числа, причем  $\alpha + \beta \leq \gamma$ , тогда*

$$\lg \alpha + \lg \beta \leq 2 \lg \gamma - 2.$$

#### Доказательство

Поскольку квадрат любого действительного числа неотрицателен, мы имеем  $(\sqrt{\alpha} - \sqrt{\beta})^2 \geq 0$ . Это выражение можно развернуть и упростить:

$$\sqrt{\alpha\beta} \leq \frac{\alpha + \beta}{2}.$$

(Это неравенство называют неравенством арифметико-геометрических средних.) Поскольку  $\alpha + \beta \leq \gamma$ , мы получаем  $\sqrt{\alpha\beta} \leq \gamma/2$ . Возведя в квадрат обе стороны и взяв от них логарифмы по основанию 2, получим приведенную выше лемму.



Далее мы должны проанализировать различные виды шагов процесса скашивания.

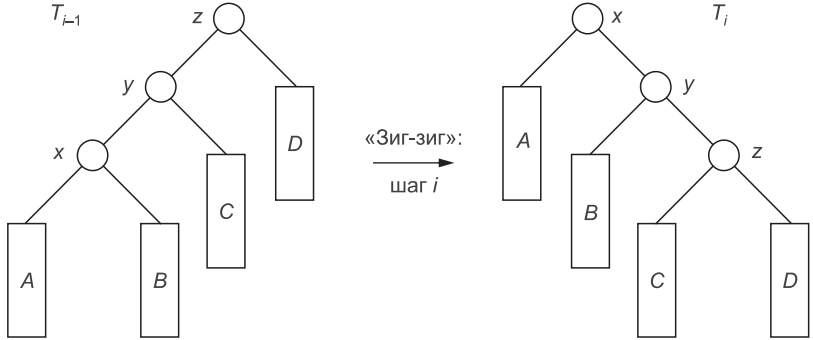
**Лемма 10.7**

Если  $i$ -й шаг скашивания есть «зиг-зиг» или «заг-заг» в узле  $x$ , тогда амортизационная сложность  $a_i$  удовлетворяет соотношению

$$a_i < 3(r_i(x) - r_{i-1}(x)).$$

**Доказательство**

Этот случай иллюстрируется следующим рисунком:



Фактическая сложность  $t_i$  шагов «зиг-зиг» или «заг-заг» составляет 2 единицы, причем на этих шагах изменяются только размеры поддеревьев с корнями  $x$ ,  $y$  и  $z$ . Поэтому все члены в сумме, определяющей  $c_i$ , взаимно сокращаются с членами  $c_{i-1}$ , за исключением показанных в приведенном ниже уравнении:

$$\begin{aligned} a_i &= t_i + c_i - c_{i-1} \\ &= 2 + r_i(x) + r_i(y) + r_i(z) - r_{i-1}(x) - r_{i-1}(y) - r_{i-1}(z) \\ &= 2 + r_i(y) + r_i(z) - r_{i-1}(x) - r_{i-1}(y). \end{aligned}$$

Чтобы получить последнюю строчку, мы берем логарифм от равенства  $|T_i(x)| = |T_{i-1}(z)|$ , которое означает, что поддерево с корнем  $z$  перед шагом скашивания имеет тот же размер, что и поддерево с корнем  $x$  после шага скашивания.

Теперь, приложив лемму 10.6 к этому выражению, можно избавиться от члена 2 (фактическая сложность). Пусть  $\alpha = |T_{i-1}(x)|$ ,  $\beta = |T_i(z)|$  и  $\gamma = |T_i(x)|$ . Из диаграммы, отображающей этот случай, мы видим, что  $T_{i-1}(x)$  содержит компоненты  $x$ ,  $A$  и  $B$ ;  $T_i(z)$  содержит компоненты  $z$ ,  $C$  и  $D$ ; а  $T_i(x)$  содержит все эти компоненты (и, кроме того,  $y$ ). Отсюда  $\alpha + \beta < \gamma$ , и по лемме 10.6 получаем  $r_{i-1}(x) + r_i(z) \leq 2r_i(x) - 2$  или  $2r_i(x) - r_{i-1}(x) - r_i(z) - 2 \geq 0$ . Прибавив эту неотрицательную величину к правой стороне последнего уравнения для  $a_i$ , получим

$$a_i \leq 2r_i(x) - 2r_{i-1}(x) + r_i(y) - r_{i-1}(y).$$

Перед шагом  $i$  узел  $y$  является родительским узлом для  $x$ , поэтому  $|T_{i-1}(y)| > |T_{i-1}(x)|$ . После шага  $i$   $x$  является родительским узлом для  $y$ , поэтому  $|T_i(x)| > |T_i(y)|$ . Взяв логарифмы, мы имеем  $r_{i-1}(y) > r_{i-1}(x)$  и  $r_i(x) > r_i(y)$ . Отсюда, наконец, получаем

$$a_i < 3r_i(x) - 3r_{i-1}(x),$$

что и представляет собой доказательство леммы 10.7.

**Лемма 10.8**

Если  $i$ -й шаг скашивания есть «зиг-заг» или «заг-зиг» в узле  $x$ , тогда амортизационная сложность  $a_i$  удовлетворяет соотношению

$$a_i < 2(r_i(x) - r_{i-1}(x)).$$

**Лемма 10.9**

Если  $i$ -й шаг скашивания есть «заг» в узле  $x$ , тогда амортизационная сложность  $a_i$  удовлетворяет соотношению

$$a_i < 1 + (r_i(x) - r_{i-1}(x)).$$

Доказательство леммы 10.8 схоже с доказательством леммы 10.7 (не смотря на то, что результат оказывается сильнее), а доказательство леммы 10.9 вообще элементарно; оба эти доказательства оставлены для упоминаний.

Наконец, нам надо найти полные амортизированные затраты на извлечение или включение. Для этого мы должны сложить стоимость всех шагов скашивания, выполняемых в процессе извлечения или включения. Если число таких шагов  $m$ , тогда максимум один (последний) шаг может быть шагом «зиг» или «заг», к которому приложима лемма 10.9, а остальные будут удовлетворять границам лемм 10.7 и 10.8, причем коэффициент 3 в лемме 10.7 дает более слабую границу. Отсюда можно получить полные амортизированные затраты:

$$\begin{aligned} \sum_{i=1}^m a_i &= \sum_{i=1}^{m-1} a_i + a_m \\ &\leq \sum_{i=1}^{m-1} (3r_i(x) - 3r_{i-1}(x)) + (1 + 3r_m(x) - 3r_{m-1}(x)) \\ &= 1 + 3r_m(x) - 3r_0(x) \\ &\leq 1 + 3r_m(x) \\ &= 1 + 3\lg n. \end{aligned}$$

В этом выводе мы использовали свойство телескопических сумм, и в итоге остались только первый ранг  $r_0(x)$  и последний ранг  $r_m(x)$ . Поскольку  $r_0(x) \geq 0$ , опущение этого члена только усиливает правую сторону, и поскольку в конце процесса скашивания  $x$  становится корнем дерева, мы имеем  $r_m(x) = \lg n$ , где  $n$  есть число узлов в дереве.

В результате мы завершили доказательство основного результата этого раздела:

**Теорема 10.10**

Амортизированные затраты включения или извлечения со скашиванием в двоичном дереве поиска с  $n$  узлами не превосходит

$$1 + 3\lg n$$

перемещений вверх узла-мишени в дереве.

Наконец, мы можем соотнести эти амортизированные затраты с фактическими затратами каждой операции включения или извлечения со скашиванием в длинной последовательности таких операций. Для этого мы используем лемму 10.5, замечая, что суммирование теперь производится по последовательности извлечений или включений, а не по шагам отдельной операции извлечения или включения. Мы видим, что полные фактические затраты последовательности из  $m$  скашиваний отличаются

от полных амортизированных затрат лишь на  $c_0 - c_m$ , где  $c_0$  и  $c_m$  есть кредитовые остатки начального и конечного дерева соответственно. Если объем дерева никогда не превышает  $n$  узлов, тогда эти остатки лежат где-то в пределах между 0 и  $\lg n$ , поэтому стоимость по теореме 10.10 нам не надо увеличивать более чем на  $\lg n$ , чтобы получить:

**Следствие 10.11**

*Полная сложность последовательности из  $t$  включений или извлечений со скашиванием в двоичном дереве поиска, объем которого никогда не превышает  $n$  узлов, не превосходит*

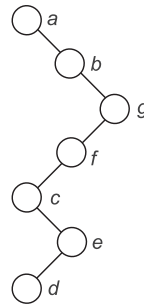
$$m(1 + 3 \lg n) + \lg n$$

*перемещений вверх узла-мишени в дереве.*

В этом результате каждый шаг скашивания включает два перемещения вверх, за исключением шагов «зиг» или «заг», которые включают по одному перемещению каждый.

### Упражнения 10.5.

**Е1.** Рассмотрим такое дерево двоичного поиска:



Выполните скашивание этого дерева по очереди для следующих ключей:

*d b g f a d b d*

Каждая часть этого упражнения выполняется на основе предыдущей части, т. е. результирующее дерево каждой части используется в качестве исходного дерева для следующей части. [Проверьте: после выполнения последней части, как и после выполнения предпоследней, дерево должно оказаться полностью сбалансированным.]

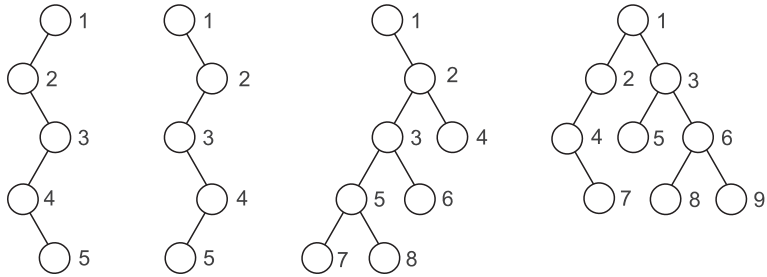
**Е2.** Определите функцию ранжирования  $r(x)$  для узлов любого двоичного дерева следующим образом:

Если  $x$  есть корень, тогда  $r(x) = 0$ .

Если  $x$  есть левый дочерний узел узла  $y$ , тогда  $r(x) = r(y) - 1$ .

Если  $x$  есть правый дочерний узел узла  $y$ , тогда  $r(x) = r(y) + 1$ .

Мы определяем *кредитовый остаток* дерева по ходу просмотра как ранг посещаемого узла. Для каждого из приведенных ниже двоичных деревьев определите ранги каждого узла и подготовьте таблицу, подобную таблице на рис. 10.28, показывающую фактические затраты, кредитовый остаток и амортизированные затраты каждого шага симметричного просмотра.



- Е3.** Обобщите описание процедуры амортизационного анализа, данное в тексте для инкремента четырех разрядов двоичного числа, на случай  $n$ -разрядных двоичных чисел.
- Е4.** Докажите лемму 10.8. Метод доказательства схож с доказательством леммы 10.7.
- Е5.** Докажите лемму 10.9. Это доказательство не требует использования леммы 10.6 или каких-либо сложных вычислений.

### Программные проекты 10.5

- Р1.** Подставьте процедуру извлечения и включения со скашиванием дерева в управляемую меню демонстрационную программу для двоичных деревьев поиска, созданную в проекте Р2 раздела 10.2 (используя вариант с сигнальной меткой *null*), получив таким образом демонстрационную программу для скошенных деревьев.
- Р2.** Реализуйте операции гардероба из раздела 7.3 с помощью скошенных деревьев. Оттестируйте вашу программу посредством демонстрационной программы для гардероба, а затем запустите программу для анализа процессорного времени с пакетом для скошенных деревьев и сравните результаты с теми, что были получены для других реализаций. Почему вы скорее всего получите обескураживающие результаты?
- Р3.** Подставьте процедуру извлечения и включения со скашиванием дерева в проект Р6 из раздела 10.2. Сравните производительность скошенных деревьев с обычными двоичными деревьями поиска для различных входных текстовых файлов.

## Подсказки и ловушки

1. Рассматривайте двоичные деревья поиска как альтернативу упорядоченным спискам (разумеется, в плане реализации абстрактного типа данных *список*). Ценой дополнительного поля для указателя в каждом узле двоичные деревья поиска обеспечивают произвольный доступ (с  $O(\log n)$  сравнений ключей) ко всем узлам, сохраняя гибкость связанных списков в отношении операций включения, удаления и переупорядочения.
2. Рассматривайте двоичные деревья поиска как альтернативу таблицам (разумеется, в плане реализации абстрактного типа данных *таблица*).

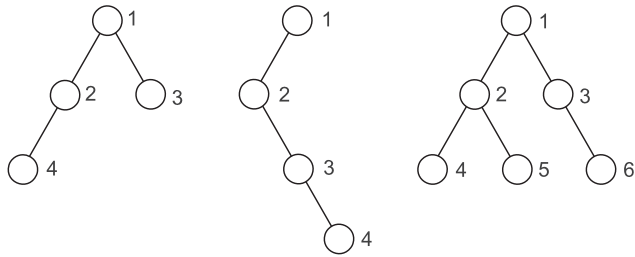
Ценой времени доступа, которое имеет порядок  $O(\log n)$  вместо  $O(1)$ , двоичные деревья поиска допускают просмотр структуры данных в порядке, определяемым ключами, сохраняя преимущества произвольного доступа, предоставляемого таблицами.

3. Выбирая структуры данных для решения своих задач, всегда тщательно анализируйте состав требуемых операций. Двоичные деревья особенно удобны, когда требуется такое сочетание возможностей, как произвольный доступ, просмотр в предопределенном порядке и гибкость в выполнении включений и удалений.
4. Выбирая структуры данных и алгоритмы, не забывайте о вероятности образования крайне несбалансированных двоичных деревьев поиска. Если предполагается, что входные данные будут поступать в случайном порядке, тогда обычное двоичное дерево поиска даст превосходные результаты. Если же данные могут поступать в упорядоченном или почти упорядоченном виде, это придется учесть при разработке алгоритмов. При этом, если вероятность серьезного разбалансирования дерева невелика, ею можно пренебречь. В случаях, когда в большом проекте возникает заметная вероятность серьезного разбалансирования, в программах, возможно, удастся найти удобные места для проверки деревьев на разбалансирование и переупорядочение их. Для приложений, в которых существенно обеспечивать логарифмическое время доступа в любой момент, AVL-деревья обеспечивают почти идеальное балансирование ценой незначительных затрат компьютерного времени и пространства памяти, но при значительных издержках на программирование. Если дерево должно динамически адаптироваться к изменениям в частоте поступающих данных, тогда наилучшим вариантом будет скошенное дерево.
5. Двоичные деревья определяются рекурсивно; рекурсивные алгоритмы для манипулирования двоичными деревьями также дают наилучшие результаты. При программировании двоичных деревьев не забывайте о проблемах, обычно сопутствующих рекурсивным алгоритмам. Обеспечьте корректное завершение алгоритмов при любых условиях, а также корректную обработку тривиального случая пустого дерева.
6. Хотя двоичные деревья обычно реализуются как связанные структуры, держите в уме возможность и других методов реализации. Программируя связанные двоичные деревья, не забывайте о типичных ловушках, сопровождающих любое программирование связанных списков.

## Обзорные вопросы

10.1

1. Определите термин *двоичное дерево*.
2. В чем различие между двоичным деревом и обычным деревом, в котором каждая вершина имеет максимум две ветви?
3. Определите порядок посещения вершин в каждом из приведенных ниже двоичных деревьев при: **(а)** прямом, **(б)** симметричном **(с)** обратном просмотре.



4. Изобразите дерево выражения для каждого из приведенных ниже выражений и покажите результат просмотр дерева при: **(а)** прямом, **(б)** симметричном **(с)** обратном просмотре.

- (а)**  $a - b$   
**(б)**  $n/m!$   
**(с)**  $\log m!$   
**(д)**  $(\log x) + (\log y)$   
**(е)**  $x \times y \leq x + y$   
**(ф)**  $a > b$  или  $b \geq a$

- 10.2 5. Определите термин *двоичное дерево поиска*.

6. Если двоичное дерево поиска с  $n$  узлами хорошо сбалансировано, сколько приблизительно надо выполнить сравнений ключей, чтобы найти мишень? Каково это число, если дерево выродилось в цепочку?
7. Объясните с помощью не более двадцати слов, как работает древовидная сортировка.
8. Какова связь между древовидной и быстрой сортировкой?
9. Почему удаление из дерева поиска осуществляется с большим трудом, чем включение в то же дерево?

- 10.3 10. В каких случаях оказывается полезен алгоритм построения двоичного дерева поиска, разработанный в разделе 10.3, и почему он предпочтительнее простого использования процедуры включения элементов в дерево поиска по мере их поступления из входного потока?

11. Насколько медленнее в среднем происходит поиск в случайном двоичном дереве поиска, чем в полностью сбалансированном двоичном дереве поиска?

- 10.4 12. Каково назначение AVL-деревьев?

13. Какое условие выделяет AVL-дерево среди всех двоичных деревьев поиска?

14. Нарисуйте схему, поясняющую, каким образом восстанавливается сбалансированность, когда включение в AVL-дерево переводит узел в разбалансированное состояние.

15. Какова производительность AVL-дерева в худшем случае в сравнении с производительностью (также в худшем случае) случайного двоичного дерева поиска? А в сравнении с производительностью в среднем случае? Как соотносятся производительность AVL-дерева в среднем случае и такая же производительность случайного двоичного дерева поиска?

10.5

16. Объясните с помощью не более двадцати слов, в чем суть *скашивания*.
17. Какова цель скашивания?
18. Что собой представляет *амортизационный* анализ алгоритмов?
19. Что собой представляет *кредитовая функция*, и как она используется?
20. Каковы затраты на скашивание, амортизированные по последовательности извлечений и включений, выраженные с помощью понятия *O* большого? Почему эти затраты представляются неожиданными?

## Литература для дальнейшего изучения

Наиболее исчерпывающим источником информации по двоичным деревьям можно найти в серии книг Кнута ([Knuth], см. ссылку в главе 3). Свойства двоичных деревьев, другие классы деревьев, просмотр, длина пути и история вопроса описаны в томе 1 (стр. 305–405). В томе 3 (стр. 422–480) обсуждаются деревья двоичного поиска и AVL-деревья, а также связанные с ними темы. Доказательство теоремы 10.2 взято из тома 3, стр. 427.

Еще одно исследование двоичных деревьев с подробными программами на языке Pascal можно найти в книге

N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, N. J., 1976.

Имеется русский перевод: Вирт Н. Алгоритмы и структуры данных. СПб. [б.и.]: 2001. — 351 с.

Эта книга также содержит (на стр. 189–264) описание с алгоритмами на языке Pascal двоичных деревьев, методов балансирования и обобщения, включая (на стр. 215–256) алгоритмов включения и удаления для AVL-деревьев.

Математический анализ поведения AVL-деревьев содержится в книге E. M. Reingold, J. Nievergelt, N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, N. J., 1977.

Первая публикация об AVL-деревьях содержится в статье

Г.М. Адельсон-Вельский и Е.М. Ландис, *Доклады Академии Наук СССР* 146 (1962), 263–266.

Несколько алгоритмов конструирования сбалансированных деревьев двоичного поиска обсуждаются в статье

Hsi Chang and S. S. Iyengar, «Efficient algorithms to globally balance a binary search tree», *Communications of the ACM* 27 (1984), 695–702.

Понятия скошенных деревьев и амортизационного анализа алгоритмов даны в статье

D. D. Sleator and R. E. Tarjan, «Self-adjusting binary search trees», *Journal of the ACM* 32 (1985), 652–686.

Из этой же статьи взяты описываемые нами алгоритмы.

Хорошим источником для более детального изучения тем, представленных в настоящей главе, являются книги

Harry R. Lewis and Larry Denenberg, *Data Structures & Their Algorithms*, Harper Collins, 1991, 509 pages.

Derick Wood, *Data Structures, Algorithms, and Performance*, Addison-Wesley, 1993, 594 pages.

Другой интересный метод подгонки формы двоичного дерева поиска, получивший название деревьев *со взвешенной длиной путей* и основанный на частотах, с которыми осуществляется обращение к узлам, приведен в следующей статье, легкой для чтения и содержащей обзор результатов анализа:

G. Argo, «Weighting without waiting: the weighted path length tree», *Computer Journal*, 34 (1991), 444–449.



## Глава 11

# Многовариантные деревья

---

В этой главе продолжается обсуждение деревьев как структур данных, с акцентом на деревья, которые могут иметь в каждом узле более двух ветвей. Мы начнем с установления связи с двоичными деревьями. Затем мы рассмотрим класс лексикографических деревьев поиска, называемых трай-деревьями, свойства которых напоминают свойства табличного поиска. Далее мы исследуем В-деревья, которые оказываются неоценимыми для решения задач извлечения информации из внешних источников. Каждый из этих разделов независим от остальных. Наконец, мы приложим идеи В-деревьев для получения еще одного класса двоичных деревьев поиска, называемых красно-черными деревьями.

### 11.1. Сады, деревья и двоичные деревья

Двоичные деревья, как мы видели, являются мощной и элегантной формой структуры данных. Однако ограничение на количество дочерних узлов — не более двух для каждого узла — представляется весьма жестким, учитывая, что имеется много приложений, где хотелось бы в качестве структур данных использовать деревья с произвольным числом дочерних узлов. В этом разделе мы покажем, что двоичные деревья позволяют представить структуры данных, которые на первый взгляд кажутся существенно более широким классом деревьев.

#### 11.1.1. Классификация видов

Поскольку мы уже столкнулись с несколькими видами деревьев в рассматриваемых нами приложениях, нам следует перед тем как приступить к дальнейшему изучению, попытаться дать несколько определений. В математике термин *дерево* имеет очень широкое значение; под деревом понимается любой набор точек (называемых вершинами) вместе с любым набором различающихся линий между парами вершин (эти линии называются ребрами или ветвями), в котором: (1) от любой вершины к любой другой можно найти последовательность ребер (путь) и (2) отсутствуют замкнутые контуры, т. е. нет путей, начинающихся в некоторой вершине и возвращающихся в ту же вершину.

В компьютерных приложениях нас обычно не интересуют исследования деревьев в таком общем плане; в тех случаях, когда приходится этим заниматься, такие деревья мы для определенности называем *свободными*

*деревьями*. Наши деревья почти всегда ограничены наличием одного выделенного узла, являющегося **корнем** дерева, и такие деревья мы для определенности называем **корневыми деревьями**.

корневое дерево

Корневое дерево можно изобразить, как это мы всегда и делали, взяв его за корень и встряхнув так, чтобы все его ветви и вершины свесились вниз, а листья оказались в самом низу. Однако в этом случае корневое дерево еще не полностью отвечает тем структурам, которые мы обычно используем. В корневом дереве нельзя отличить левую сторону от правой или сказать, в случае, когда некоторая вершина имеет несколько дочерних вершин, какая из них первая, какая вторая и т. д. Если даже не касаться других причин, ограничения последовательного выполнения команд (не говоря уж об последовательной организации памяти) обычно накладывают некоторый порядок на дочерние узлы каждой вершины. Поэтому мы называем **упорядоченным деревом** корневое дерево, в котором каждая вершина образуется в определенном порядке.

упорядоченное  
дерево

Заметьте, что упорядоченные деревья, в которых каждая вершина имеет не более двух дочерних вершин, все еще не относятся к классу двоичных деревьев. Если вершина двоичного дерева имеет только одну дочернюю вершину, то эта дочерняя вершина может находиться как с левой, так и с правой стороны от родительской вершины, причем эти два двоичных дерева различаются, хотя оба они относятся к классу упорядоченных деревьев.

2-дерево

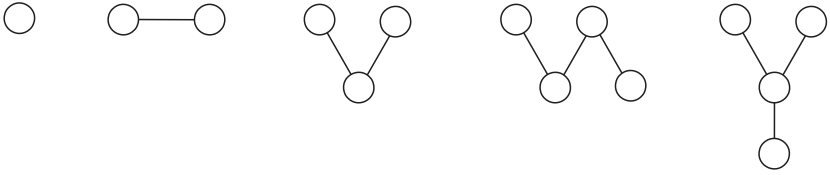
В качестве последнего замечания, относящегося к определениям, следует указать, что 2-деревья, которые мы изучали в связи с анализом алгоритмов, являются корневыми деревьями (но не обязательно упорядоченными деревьями) с той особенностью, что каждая вершина имеет либо 0, либо 2 дочерних вершины. Поэтому 2-деревья не совпадают ни с одним из рассмотренных нами классов.

На рис. 11.1. показано, что происходит с деревьями различных типов с небольшим числом вершин. Заметим, что каждый класс деревьев, начиная со второго, может быть получен из деревьев предыдущего класса после включения в него вариантов, соответствующих новым критериям. Сравните пять упорядоченных деревьев с четырьмя вершинами с набором из четырнадцати двоичных деревьев с четырьмя вершинами, сконструированными в результате решения упражнения E1 раздела 10.1. Вы обнаружите, что и здесь двоичные деревья могут быть получены из соответствующих упорядоченных деревьев путем добавления варианта, в котором левая ветвь отличается от правой ветви.

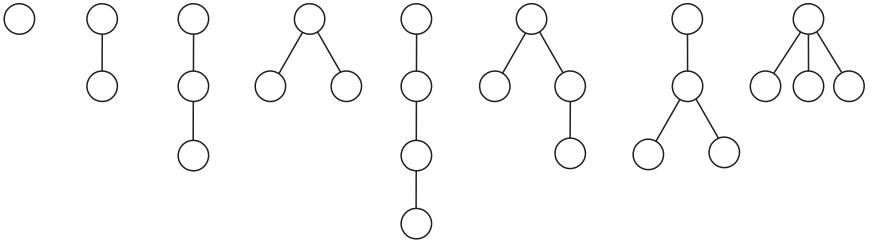
## 11.1.2. Упорядоченные деревья

### 1. Компьютерная реализация

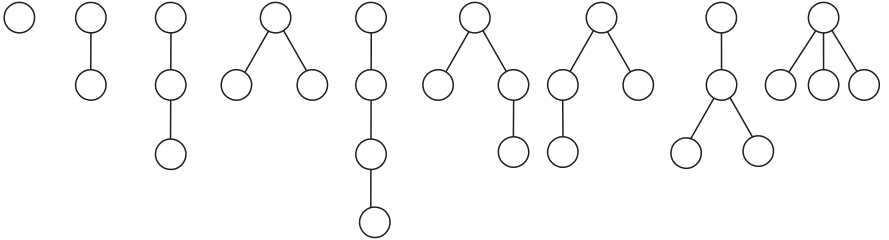
Если мы решили использовать упорядоченное дерево в качестве структуры данных, очевидным способом реализации этой структуры в памяти компьютера является расширение стандартного способа реализации двоичного дерева, когда в каждом узле вместо двух связей, необходимых для двоичных деревьев, предусматривается столько полей, сколько может быть поддеревьев. В результате дерево, в котором некоторые узлы



Свободные деревья с 4 или меньшим количеством вершин  
(взаимное расположение вершин не имеет значения)



Корневые деревья с 4 или меньшим количеством вершин  
(корень находится на верху дерева)



Упорядоченные деревья с 4 или меньшим количеством вершин

**Рис. 11.1.** Различные виды деревьев

могут иметь до десяти поддеревьев, будет содержать в каждом узле десять полей связи. Однако в этом случае очень многие поля связи будут иметь значение **nil**. Мы, кстати, можем определить, сколько именно. Если дерево имеет  $n$  узлов и каждый узел имеет  $k$  полей связи, тогда всего в дереве будет  $n \times k$  связей. На каждый из  $n - 1$  узлов, отличных от корня, указывает в точности одна связь, поэтому пропорция **nil**-связей должна быть

$$\frac{(n \times k) - (n - 1)}{n \times k} > 1 - \frac{1}{k}.$$

Отсюда, если вершина может иметь до десяти поддеревьев, то более девяноста процентов связей будут иметь значение **nil**. Очевидно, что такой метод представления упорядоченных деревьев является весьма затратным в плане используемой памяти. Причина здесь в том, что для каждого узла мы предусматриваем непрерывный список связей ко всем его дочерним узлам, и этот список резервирует большой объем неиспользуе-

мой памяти. Ниже мы рассмотрим способ, позволяющий заменить непрерывные списки связными, и весьма элегантно приводящий к аналогии с двоичными деревьями.

## 2. Связная реализация

Чтобы хранить дочерние узлы каждого родительского узла в связном списке, нам потребуются два рода связей. Первая связь будет представлять собой заголовок каждого списка; это будет связь от каждого узла к его самому левому дочернему узлу, который мы будем называть *firstchild* (первый ребенок). Далее, каждый узел, за исключением корня, будет находиться в одном из этих списков, и, таким образом, ему потребуется связь со следующим узлом в списке, т. е. со следующим дочерним узлом этого родителя. Эту вторую связь мы можем назвать *nextchild* (следующий ребенок). Предлагаемая реализация проиллюстрирована на рис. 11.2.

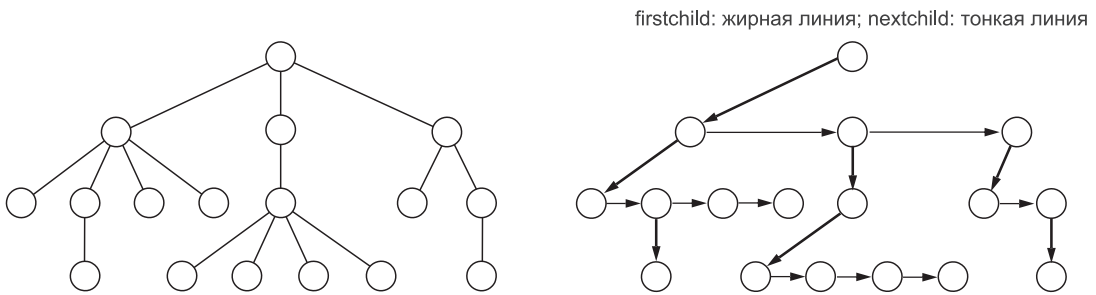


Рис. 11.2. Связная реализация упорядоченного дерева

## 3. Естественное соответствие

Для каждого узла в упорядоченном дереве мы определили две связи (которые при отсутствии заданных для них значений будут иметь значение *nil*), *firstchild* и *nextchild*. С помощью этих двух связей мы теперь имеем структуру двоичного дерева; другими словами, связная реализация упорядоченного дерева представляет собой связное двоичное дерево. При желании мы можем даже сформировать лучшее изображение двоичного дерева, взяв связную реализацию упорядоченного дерева и повернув ее на несколько градусов по часовой стрелке, чтобы идущие вниз связи (*firstchild*) указывали влево, а горизонтальные связи (*nextchild*) указывали вниз и вправо.

## 4. Обратное соответствие

Предположим, что мы обратили шаги описанного процесса, начав с двоичного дерева и пытаясь восстановить упорядоченное дерево. Прежде всего мы заметим, что с помощью такого процесса не каждое двоичное дерево можно получить из корневого дерева посредством описанного выше процесса. Поскольку связь *nextchild* корневого узла всегда равна *nil*, корень соответствующего двоичного дерева всегда будет иметь пустое правое поддерево. Для более тщательного изучения обратного соответствия мы должны рассмотреть еще один класс структур данных.

### 11.1.3. Леса и сады

Работая с двоичными деревьями, мы с большим успехом использовали рекурсию, так что есть смысл пользоваться этим методом и при изучении других классов деревьев. Использование рекурсии сводится к сокращению задачи до аналогичной задачи меньшего объема. Таким образом, мы должны посмотреть, что произойдет, если мы возьмем корневое дерево или упорядоченное дерево иотрежем от него корень. В этом случае от него останется (если этот остаток не пуст) набор корневых деревьев или упорядоченный набор упорядоченных деревьев, соответственно.

лес

Стандартным обозначением произвольного набора деревьев является термин *лес*, но при использовании этого термина мы обычно подразумеваем, что деревья имеют корни. Для упорядоченного набора упорядоченных деревьев иногда используется фраза *упорядоченный лес*, но мы для такого класса деревьев будем пользоваться более содержательным (и более колоритным) термином *сад*.

сад

Заметьте, что мы не только можем образовать лес или сад, удалив корень корневого дерева или упорядоченного дерева, соответственно, но мы можем также построить корневое или упорядоченное дерево, начав с леса или сада, создав новую вершину на самом верху и добавив ветви от этой новой вершины (которая станет корнем) к корням всех деревьев леса или сада.

рекурсивное  
определение

С помощью этого наблюдения мы дадим новое рекурсивное определение упорядоченным деревьям и садам, определение, которое приведет к формальному доказательству аналогии с двоичными деревьями. Прежде всего, подумаем, как начать. Вспомним, что двоичное дерево может быть пустым; другими словами, оно может не иметь ветвей. Также возможно, что лес или сад будет пуст; другими словами, в том или в другом нет деревьев. Однако невозможно, чтобы корневое или упорядоченное дерево было пусто, поскольку оно гарантированно имеет по меньшей мере корень. Если мы приступаем к построению деревьев и леса, мы можем заметить, что дерево с единственной вершиной образуется путем подсоединения нового корня к пустому лесу. Получив это дерево, мы можем создать лес, состоящий из любого требуемого количества одновершинных деревьев, и, подсоединив к ним новый корень, образовать корневые деревья с высотой 1. Таким же образом мы можем продолжать конструировать по очереди все корневые деревья в соответствии со следующими взаимно рекурсивными определениями.

Определение

**Корневое дерево** состоит из единственной вершины  $v$ , называемой **корнем** дерева, вместе с лесом  $F$ , деревья которого называются **поддеревьями** корня.

**Лес**  $F$  представляет собой (возможно пустой) набор корневых деревьев.

Аналогичные определения можно сформулировать для упорядоченных деревьев и садов.

**Определение**

*Упорядоченное дерево*  $T$  состоит из единственной вершины  $v$ , называемой *корнем* дерева, вместе с садом  $O$ , деревья которого называются *поддеревьями* корня  $v$ . Мы можем отождествить упорядоченное дерево с упорядоченной парой

$$T = \{v, O\}.$$

Сад  $O$  представляет собой либо пустой набор  $\emptyset$ , либо набор, состоящий из упорядоченного дерева  $T$ , называемого первым деревом сада, вместе с другим садом  $O'$  (который содержит оставшиеся деревья сада). Мы можем отождествить упорядоченное дерево с упорядоченной парой

$$O = (T, O').$$

Обратите внимание на явное указание порядка деревьев в определении сада. Непустой сад содержит первое дерево, а также оставшиеся деревья от другого сада, который опять имеет первое дерево, являющееся вторым деревом исходного сада. Продолжение анализа оставшегося сада дает нам третье дерево и т. д., пока оставшийся сад не окажется пустым садом.

### 11.1.4. Формальное соответствие

Теперь мы можем получить принципиальные для этого раздела результаты.

**Теорема 11.1**

*Пусть  $S$  есть конечный набор вершин. Имеется взаимно однозначное соответствие  $f$  между набором садов, набором вершин которых является  $S$ , и набором двоичных деревьев, набором вершин которых является  $S$ .*

**Доказательство**

Для доказательства теоремы воспользуемся обозначениями, введенными нами в определениях. Прежде всего нам потребуется аналогичное обозначение для двоичных деревьев. Двоичное дерево  $B$  есть либо пустой набор  $\emptyset$ , либо набор, состоящий из корневой вершины  $v$  с двумя двоичными деревьями  $B_1$  и  $B_2$ . Таким образом, мы можем отождествить двоичное дерево с упорядоченной тройкой

$$B = [v, B_1, B_2]$$

Мы докажем теорему методом математической индукции, изменяя число вершин в  $S$ . Первый случай, требующий рассмотрения, — это пустой сад  $\emptyset$ , который будет соответствовать пустому двоичному дереву:

$$f(\emptyset) = \emptyset.$$

Если сад  $\emptyset$  не пуст, он обозначается упорядоченной парой

$$O = (T, O_2),$$

где  $T$  есть упорядоченное дерево, а  $O_2$  — другой сад. Упорядоченное дерево обозначается парой

$$T = \{v, O_1\},$$

где  $v$  есть вершина, а  $O_1$  — другой сад. Мы подставляем это выражение для  $T$  в первое выражение, получая

$$O = (\{v, O_1\}, O_2).$$

Согласно индуктивному предположению,  $f$  предоставляет однозначное соответствие между садами с числом вершин, меньшим, чем в  $S$ , и двоичными деревьями, а сады  $O_1$  и  $O_2$  меньше сада  $O$ , поэтому двоичные деревья  $f(O_1)$  и  $f(O_2)$  определяются по индуктивному предположению. Мы определяем соответствие  $f$  между садом и двоичным деревом равенством

$$f(\{v, O_1\}, O_2) = [v_1, f(O_1), f(O_2)]$$

Теперь становится очевидным, что функция  $f$  представляет собой однозначное соответствие между садами и двоичными деревьями с теми же вершинами. Каким бы способом мы не наполняли значением символы  $v$ ,  $O_1$  и  $O_2$  с левой стороны равенства, имеется в точности единственный способ наполнения значением тех же символов с правой стороны, и наоборот.

### 11.1.5. Повороты

Мы можем также использовать введенное выше обозначение соответствия для пояснения преобразования от сада к двоичному дереву. В двоичном дереве  $[v_1, f(O_1), f(O_2)]$  левая связь от  $v$  указывает на корень двоичного дерева  $f(O_1)$ , который фактически является первым дочерним узлом вершины  $v$  в упорядоченном дереве  $\{v, O_1\}$ . Правая связь от  $v$  указывает на вершину, которая первоначально была корнем следующего упорядоченного дерева с правой стороны. Другими словами, «левая связь» в двоичном дереве соответствует «первому дочернему узлу» в упорядоченном дереве, а «правая связь» соответствует «следующему дочернему узлу того же уровня (брату или сестре первого ребенка)». При графическом отображении преобразование сводится к следующим правилам:

1. Нарисуйте сад так, чтобы первый дочерний узел каждой вершины помещался непосредственно под этой вершиной, вместо того, чтобы изображать дочерние узлы симметрично под родительской вершиной.
2. Нарисуйте вертикальную связь от каждой вершины к ее первому дочернему узлу, после чего нарисуйте горизонтальную линию от каждой вершины к следующему дочернему узлу того же уровня.
3. Удалите оставшиеся первоначальные связи.
4. Поверните диаграмму на 45 градусов по часовой стрелке, чтобы вертикальные линии отображались как левые связи, а горизонтальные – как правые связи.

Этот процесс проиллюстрирован на рис. 11.3.

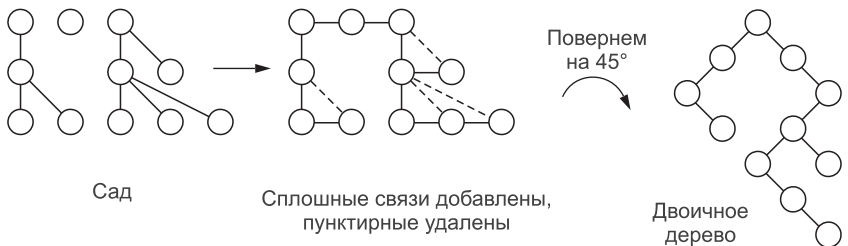


Рис. 11.3. Преобразование сада в двоичное дерево

### 11.1.6. Резюме

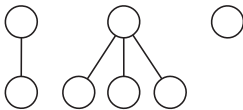
Мы обнаружили три способа описания соответствия между садами и двоичными деревьями:

- связи `firstchild` и `nextchild`;
- повороты диаграмм;
- формальные выражения равенства.

Большинство людей находят второй способ, поворота диаграмм, наиболее простым и наглядным. Однако при написании компьютерных программ требуется как раз первый способ, в котором соответствие описывается установкой связей. Третий способ, формальное соответствие, оказывается наиболее полезным при конструировании доказательств различных свойств двоичных деревьев и садов.

### Упражнения 11.1

**Е1.** Преобразуйте каждый из приведенных ниже садов в двоичное дерево.



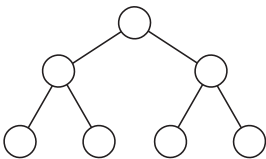
(a)



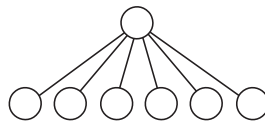
(b)



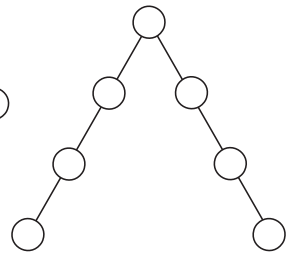
(c)



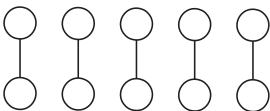
(d)



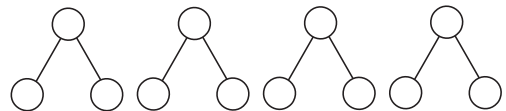
(e)



(f)



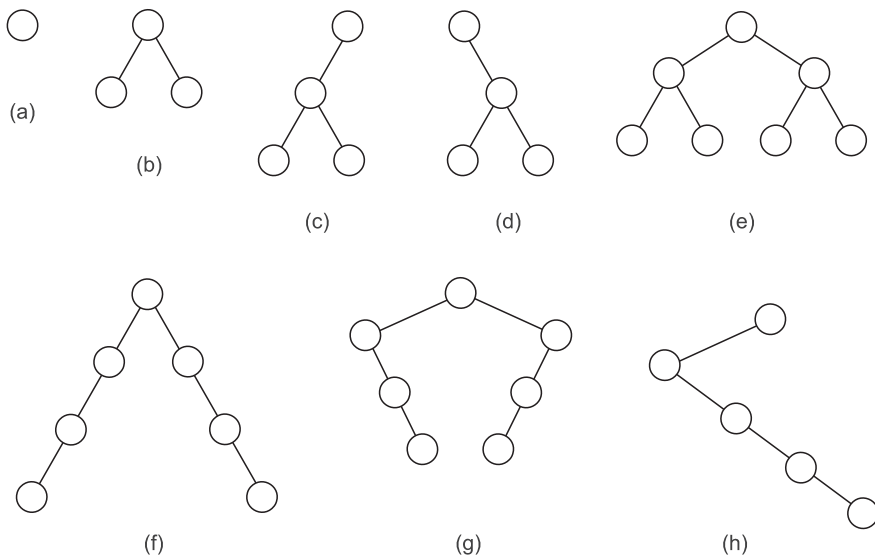
(g)



(h)



**Е2.** Преобразуйте каждое из приведенных ниже двоичных деревьев в сад.



**Е3.** Нарисуйте все возможные:

- (a) свободные деревья,
  - (b) корневые деревья,
  - (c) упорядоченные деревья
- с пятью вершинами.

**Е4.** Мы можем определить *прямой просмотр* сада следующим образом. Если сад пуст, не делайте ничего. В противном случае, сначала посетите корень первого дерева, затем просматривайте сад поддеревьев первого дерева в порядке прямого просмотра, а далее просматривайте сад оставшихся деревьев в том же порядке прямого просмотра. Докажите, что при прямом просмотре сада и прямом просмотре соответствующего двоичного дерева вершины будут посещаться в том же порядке.

**Е5.** Мы можем определить *симметричный просмотр* сада следующим образом. Если сад пуст, не делайте ничего. В противном случае, сначала посетите сад поддеревьев корня первого дерева в симметричном порядке, затем посетите корень первого дерева, а далее просматривайте сад оставшихся деревьев в том же симметричном порядке. Докажите, что при симметричном просмотре сада и симметричном просмотре соответствующего двоичного дерева вершины будут посещаться в том же порядке.

**Е6.** Опишите способ просмотра сада, при котором вершины будут посещаться в том же порядке, что и при обратном просмотре соответствующего двоичного дерева. Докажите, что ваш метод просмотра посетит все вершины в правильном порядке.

## 11.2. Деревья лексикографического поиска: трай-деревья

многовариантные  
переходы

На протяжении предыдущих глав мы несколько раз сопоставляли поиск в списке с поиском элемента в таблице. Мы можем приложить идею табличного поиска к извлечению информации из дерева, используя ключ или часть ключа, чтобы организовать многовариантные переходы.

Вместо того, чтобы выполнять сравнение на равенство всего ключа, мы можем рассматривать ключ как последовательность символов (например, букв или цифр), и использовать эти символы для задания на каждом шаге множественных переходов. Если наши ключи представляют собой имена, состоящие из букв, мы выполняем 26-вариантный переход согласно первой букве имени, затем выполняем такой же переход согласно второй букве имени и т. д. Многовариантный переход напоминает указатель на обрезе справочника с выемками для букв алфавита. Такой указатель, однако, обычно используется лишь для поиска слова с данной начальной буквой; для продолжения поиска используется какой-либо другой метод. В компьютерных программах мы можем реализовать два или три уровня многовариантных переходов, но в этом случае дерево получится слишком большим, и для продолжения поиска нам также придется поискать какой-то другой метод.

### 11.2.1. Трай-деревья

Один из методов заключается в обрезке с дерева всех ветвей, которые не ведут к каким-либо ключам. В английском языке, например, нет слов, которые начинались бы с сочетаний 'bb', 'bc', 'bf', 'bg', ..., однако есть слова, начинающиеся с 'ba', 'bd' или 'be'. Таким образом, все ветви и узлы для несуществующих слов могут быть удалены с дерева. Результирующее дерево носит название *трай-дерева*. (Этот термин образован удалением букв из слова *retrieval* [извлечение], однако он обычно произносится «трай», как и английское слово «try» [попробовать].)

Трай-дерево порядка  $m$  может быть формально определено как либо пустое, либо содержащее упорядоченную последовательность в точности  $m$  трай-деревьев порядка  $m$ .

### 11.2.2. Поиск ключа

Трай-дерево, описывающее слова (как они перечислены в Оксфордском толковом словаре), составленные только из букв  $a$ ,  $b$  и  $c$ , показано на рис. 11.4. Каждый узел, вместе с ветвями, идущими к низшим уровням дерева, содержит указатель на запись с информацией о ключе (если таковой имеется), который был найден при достижении данного узла. Поиск конкретного ключа начинается с корня. Первая буква ключа используется в качестве индекса для определения ветви, на которую следует перейти. Пустая ветвь означает, что искомый ключ не содержится в дереве. В противном случае мы используем вторую букву ключа для определения ветви следующего уровня, и продолжаем далее таким же образом. При



Предполагается, что константа `maxlength`, определяющая максимальную длину ключа, а также тип `entrypointer` [указатель на элемент], указывающий для каждого ключа на требуемую информацию, определены где-то в другом месте. Мы используем косвенную адресацию этой информации, чтобы уменьшить объем памяти, занимаемый деревом. Мы будем предполагать, что все ключи составлены только из строчных букв и пробелов, и что ключ завершается первым встретившимся в нем пробелом. Наш метод поиска реализуется следующей процедурой.

```

procedure TrieSearch (root: triepointer; target: keytype;           { Трай-поиск }
                      var location: triepointer);
{ Pre:  root указывает на корень трай-деревя.
  Post: Если поиск завершился успешно, location↑.ref является
          элементом трай-деревя, который указывает
          на информационную запись, содержащую мишень target;
          в противном случае location имеет значение nil. }
var keyposition: integer;           { счетчик символов для ключа target }
begin                               { процедура TrieSearch }
  location := root;
  keyposition := 1;
  while (keyposition <= maxlength) and (location <> nil) do
    if target[keyposition] = ' ' then
      keyposition := maxlength + 1 { завершает поиск по пробелу в target }
      { location остается указывающей на узел с информацией для target. }
    else begin
      location := location↑.branch[target[keyposition] ];
      { Перемещаем вниз соответствующую ветвь трай-деревя }
      keyposition := keyposition + 1 { перемещаемся к следующему
                                     символу мишени }
    end;
    if location <> nil then
      if location↑.ref = nil then
        location := nil { поиск неуспешен }
  end;                               { процедура TrieSearch }

```

Условие завершения цикла сделано более сложным, чтобы избежать ошибок индексации после шага цикла с `keyposition = maxlength`. При завершении поиска переменная `location` (если она не равно `nil`) указывает на узел в дереве, соответствующий мишени. Фактическую информацию для этого элемента можно затем найти посредством `location↑.ref↑`.

#### 11.2.4. Включение в трай-дерево

Процедура добавления в трай-дерево нового ключа совершенно аналогична процедуре поиска ключа. Мы должны пройти трай-дерево вниз до соответствующей точки и установить указатель `info` на информационную запись для нового ключа. Если, по ходу просмотра, мы сталкиваемся с `nil`-ветвью дерева, мы не должны завершать поиск, а вместо этого должны создать новые узлы и поместить их в трай-дерево, чтобы заполнить путь, соответствующий новому ключу. Все это приводит к следующей программе.

```

procedure InsertTrie (var root: triepointer;      { Включить в трай-дерево }
                     newkey: keytype; newentry: entriepointer);
{ Pre: Параметр root является корнем трай-дерева, newkey является
  ключом, пока отсутствующим в дереве, а newentry указывает
  на соответствующую ему информацию.
Post: Этот новый элемент включен в трай-дерево. }
var keyposition: integer; { проходит по newkey, по одному символу за раз }
    location: triepointer; { проходит по трай-дереву с keyposition }
    ch: letter;             { переменная управления циклом }
begin                                { процедура InsertTrie }
  if root = nil then begin           { создадим новое трай-дерево
                                     со всеми пустыми поддеревьями }

    new(root);
    for ch := 'a' to 'z' do
      root↑.branch[ch] := nil;
    root↑.ref := nil
  end;
  location := root;                  { начнем поиск вниз по трай-дереву }
  keyposition := 1;
  while keyposition <= maxlength do
    if newkey[keyposition] = ' ' then
      keyposition := maxlength + 1    { завершим поиск: конец newkey }
    else begin
      if location↑.branch[newkey[keyposition]] <> nil then
        location := location↑.branch[newkey[keyposition]]
      else begin                      { создадим новый узел в трай-дерево }
        new(location↑.branch[newkey[keyposition]]);
        location := location↑.branch[newkey[keyposition]];
        for ch := 'a' to 'z' do
          location↑.branch[ch] := nil;
        location↑.ref := nil
      end;
      keyposition := keyposition + 1
    end;
  { В этой точке мы проверили все символы newkey, отличные от
    пробелов. }
  if location↑.ref = nil then
    Error('Сделана попытка включить в трай-дерево ключ, уже
    присутствующий в нем.')
  else location↑.ref := newentry
end;                                { процедура InsertTrie }

```

### 11.2.5. Удаление из трай-дерева

Тот же общий план, использованный нами для организации поиска и включения, может быть применен и для удаления из трай-дерева. Мы движемся по пути, соответствующему ключу, который должен быть удален, и, достигнув требуемого узла, устанавливаем соответствующее поле ref в nil. Теперь, однако, если все поля этого узла имеют значение nil (все ветви и поле ref), мы должны избавиться от этого узла. Чтобы иметь возможность удаления узла, мы можем установить стек указателей на узлы по пути от корня к последнему достигнутому нами узлу. Либо мы можем использовать рекурсию в алгоритме удаления и тем самым избавиться от необходимости иметь явный стек. В любом случае мы оставим программирование этих операций для упражнений.

## 11.2.6. Оценка трай-деревьев

сравнение  
с двоичным  
поиском

Число шагов, необходимых для поиска в трай-дереве (или для включения в него) пропорционально числу символов, составляющих ключ, а не логарифму от числа ключей, как это было в случае операций поиска в других деревьях. Если длина ключа невелика по сравнению с логарифмом по основанию 2 от числа ключей, тогда трай-дерево может оказаться лучше двоичного дерева. Если, например, ключи состоят из любых возможных последовательностей из пяти букв, тогда трай-дерево затратит на поиск любого из  $n = 26^5 = 11881376$  ключей 5 шагов, в то время двоичному поиску для выполнения той же операции потребуется минимум  $\lg n \approx 23,5$  сравнений ключей.

Во многих приложениях, однако, число символов в ключе оказывается больше, а набор фактически используемых ключей весьма разрежен по отношению к набору всех возможных ключей. В таких приложениях число шагов, требуемое для поиска в трай-дереве, может легко превзойти число сравнений ключей, необходимое для двоичного поиска.

Наконец, наилучшим решением может оказаться комбинация методов. Трай-алгоритм может быть использован для первых нескольких символов ключа, а затем для остатка ключа может быть применен другой метод. Если мы вернемся к примеру индексного указателя на обрезе справочника с выемками для букв алфавита, мы увидим, что фактически мы используем многовариантный переход (дерево со множественным ветвлением) для обнаружения первой буквы слова, но затем для поиска требуемого слова среди всех слов, начинающихся на ту же букву, мы используем другой метод поиска.

### Упражнения 11.2

**E1.** Нарисуйте трай-деревья, сконструированные из приведенных ниже последовательностей ключей.

- (a) Все трехразрядные целые числа, содержащие только цифры 1, 2 и 3 (в десятичном представлении).
- (b) Все трехбуквенные последовательности, построенные из букв a, b, c и d, в которых первой буквой является буква a.
- (c) Все четырехразрядные двоичные целые числа (состоящие только из 0 и 1).
- (d) Слова

*a ear re rare area are ere era rarer trear err*

построенные из букв a, e и r.

- (e) Слова

*gig i inn gin in inning gigging ginning*

построенные из букв g, i и n.

- (f) Слова

*pal lap a papa al papal all ball lab*

построенные из букв a, b, l и p.

- Е2.** Напишите процедуру, которая будет просматривать трай-дерево и выводить на экран все его слова в алфавитном порядке.
- Е3.** Напишите процедуру, которая будет просматривать трай-дерево и выводить на экран все его слова в порядке, определяемом, во-первых, длиной слова, начиная с более коротких слов, и, во-вторых, алфавитным порядком слов одинаковой длины.
- Е4.** Напишите процедуру, которая будет удалять слово из трай-дерева.

### Программный проекты 11.2

---

- Р1.** Напишите управляемую меню демонстрационную программу для трай-деревьев. Ключами должны быть слова длиной до 8 символов и состоящие из 26 строчных букв латинского алфавита. Информацией, которую следует хранить в записи `trieentry` (вместе с ключом), будет всего лишь порядковый номер, указывающий, когда было включено данное слово.

## 11.3. Внешний поиск: В-деревья

По ходу этой книги мы предполагали, что все наши структуры данных хранятся в быстрой оперативной памяти компьютера; другими словами, мы рассматривали извлечение только *внутренней* информации. Для многих приложений такое предположение вполне разумно, но для многих других столь же важных приложений это не так. Давайте теперь кратко рассмотрим проблему извлечения *внешней* информации, когда нам требуется найти и извлечь записи, сохраняемые в дисковом файле.

### 11.3.1. Время доступа

Время, требуемое для обращения к ячейке оперативной памяти и извлечения из нее слова составляет максимум несколько микросекунд. Время же, требуемое для локализации конкретной записи на диске измеряется миллисекундами, а для гибких дисков может превысить секунду. Отсюда следует, что время единичного доступа при внешнем извлечении будет в тысячи раз больше, чем при внутреннем. С другой стороны, если записи хранятся на диске, обычной практикой является не чтение их слово за словом, а считывание в память больших *страниц* или *блоков* информации. Типичный размер таких блоков варьируется от 256 до 1024 символов или слов.

Нашей целью при рассмотрении внешнего поиска будет минимизация числа операций доступа к диску, поскольку каждая операция доступа забирает такое большое, по сравнению с внутренними вычислениями, время. Однако при каждом обращении к диску мы получаем блок, в котором может быть достаточно места для нескольких записей. Используя эти записи, мы можем принять многовариантное решение о том, какой блок считывать следующим. Таким образом, для внешнего поиска особенно подходят многовариантные деревья.

### 11.3.2. Многовариантные деревья поиска

Двоичные деревья поиска можно непосредственно обобщить на многовариантные деревья поиска, в которых для некоторого целого  $m$ , называемого *порядком* дерева, каждый узел имеет максимум  $m$  дочерних узлов (детей). Если  $k \leq m$  есть число детей, тогда узел содержит в точности  $k - 1$  ключей, что разделяет все ключи в поддеревьях на  $k$  поднаборов. Если некоторые из этих поднаборов пусты, тогда соответствующие дети в дереве пусты. На рис. 11.5 показано 5-вариантное дерево, в котором в некоторых узлах некоторые из детей пусты.

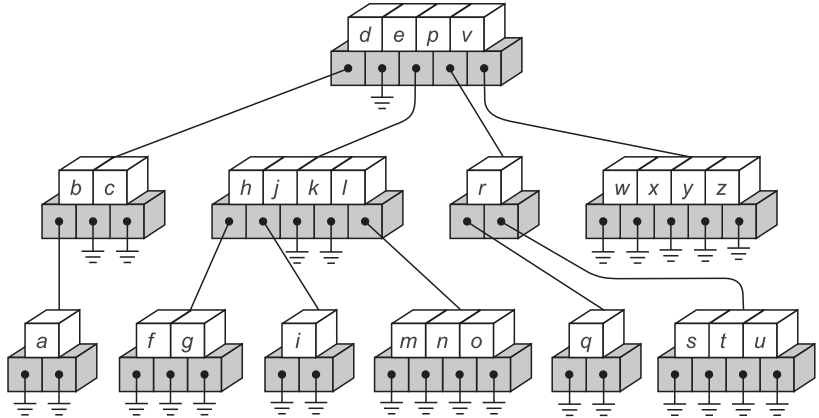


Рис. 11.5. 5-вариантное дерево поиска

### 11.3.3. Сбалансированные многовариантные деревья

Нашей целью является разработка многовариантного дерева поиска, которое минимизирует доступ к файлу; следовательно, нам хотелось бы, чтобы высота дерева была как можно меньше. Для этого мы устанавливаем, что, во-первых, выше листьев не должно быть пустых поддеревьев (в этом случае разделение ключей на поднаборы будет максимально эффективным); во-вторых, все листья должны быть на одном уровне; в третьих, каждый узел (кроме листьев) должен иметь по меньшей мере некоторое минимальное число детей. Мы потребуем, чтобы число детей каждого узла (кроме листьев) составляло по меньшей мере половину от максимального возможного числа детей. Эти условия приводят к следующему формальному определению.

**Определение**

**В-дерево порядка  $m$**  есть  $m$ -вариантное дерево, в котором:

1. Все листья находятся на одном уровне.
2. Все внутренние узлы за исключением корня имеют не более  $m$  непустых детей, и по меньшей мере  $\lceil m/2 \rceil$  непустых детей.
3. Число ключей в каждом внутреннем узле на 1 меньше числа непустых детей, и эти ключи разделяют ключи детей так же, как это делается в дереве поиска.



4. Корень имеет максимум  $t$  детей и минимум 2, если это не лист; корень может также не иметь детей совсем, если дерево состоит из одного только корня.

Дерево на рис. 11.5 не является В-деревом, поскольку у некоторых узлов имеются пустые дети, и не все листья находятся на одном уровне. На рис. 11.6 показано В-дерево порядка 5, ключами которого являются 26 букв латинского алфавита.

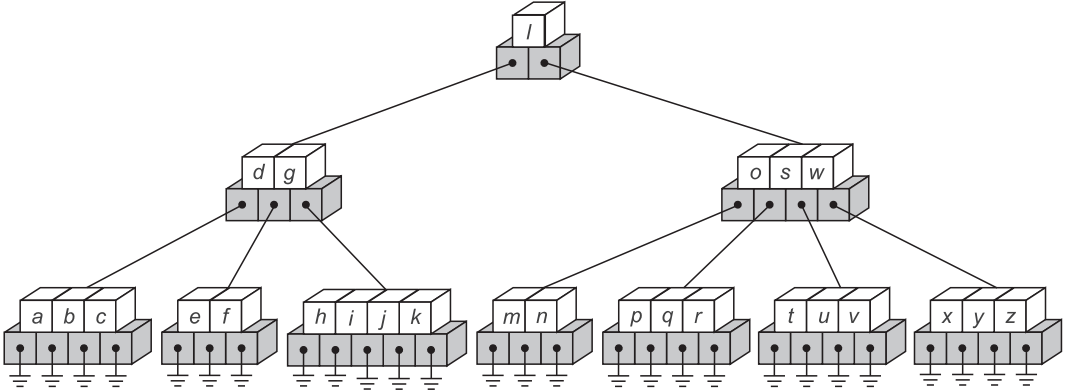


Рис. 11.6. В-дерево порядка 5

### 11.3.4. Включение в В-дерево

метод

Условие нахождения всех листьев на одном уровне определяет характерное поведение В-деревьев: в отличие от двоичных деревьев поиска, В-дереву не разрешается расти у своих листьев; рост В-дерева может происходить только у корня. Общий метод включения заключается в следующем. Прежде всего выполняется поиск, чтобы установить, не имеется ли уже новый ключ в дереве. Этот поиск (если ключ действительно новый) неуспешно завершится на листе. Затем новый ключ добавляется к узлу листа. Если узел не был предварительно заполнен, включение завершается.

Если ключ добавляется к заполненному узлу, тогда узел расщепляется на два узла на том же уровне, при этом срединный ключ не помещается ни в один из новых узлов, а посылается вверх по дереву, чтобы быть включенным в родительский узел. Когда позже будет выполняться просмотр дерева, сравнение со срединным ключом направит поиск в правильное поддерево. Когда ключ добавляется к заполненному корню, корень расщепляется на два, а срединный ключ посылается вверх и становится новым корнем.

Этот процесс приобретет большую ясность, если рассмотреть пример роста В-дерева порядка 5, показанного на рис. 11.6. Мы хотим включить ключи

*a g f b k d h m j e s i r x c l n t u p*

в первоначально пустое дерево в указанном выше порядке.

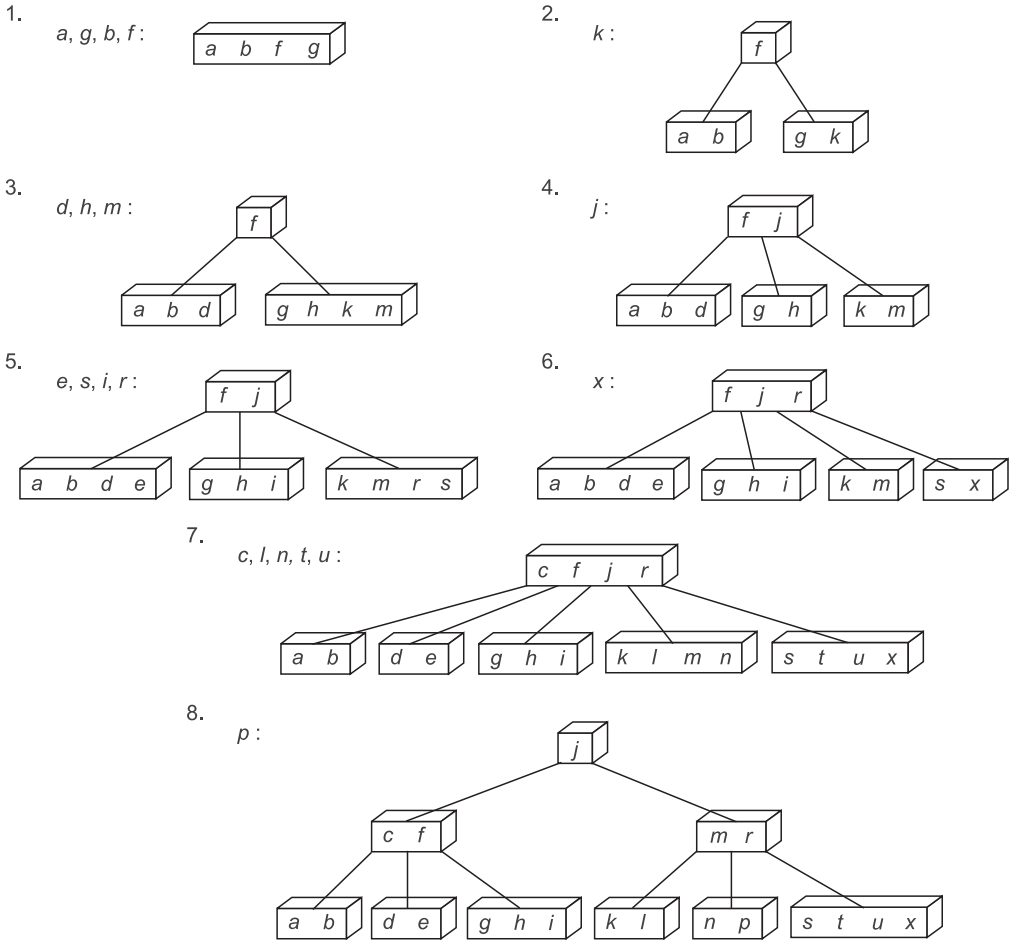


Рис. 11.7. Рост В-дерева

Первые четыре ключа будут включены в один узел, как это показано на диаграмме 1 рис. 11.7. По мере включения они упорядочиваются по алфавиту. Однако для пятого ключа,  $k$ , места в узле нет, и его включение заставляет узел расщепиться на два, а срединный ключ,  $f$ , перемещается вверх с образованием нового узла, который теперь становится новым корнем. Поскольку расщепленные узлы теперь заполнены только наполовину, следующие три ключа могут быть включены в них без особых сложностей. Заметьте, однако, что эти простые включения могут потребовать перестановок ключей в узле. Следующее включение,  $j$ , опять расщепляет узел, причем сам ключ  $j$  оказывается срединным ключом, и, следовательно, он перемещается вверх и присоединяется к  $f$  в корне.

Следующие несколько включений происходят таким же образом. Окончательно включение, именно, включение ключа  $p$ , более интересно. Это включение сначала расщепляет узел, ранее содержащий ключи  $k l m$

расщепление узла

распространение  
вверх

$n$ , и посылает срединный ключ  $m$  вверх в узел, содержащий  $c f j r$ , который, однако, уже заполнен. В результате этот узел в свою очередь расщепляется, и создается новый корень, содержащий  $j$ .

улучшение  
сбалансированности

Здесь следует сделать два замечания относительно роста В-дерева. Во-первых, когда узел расщепляется, он производит два узла, которые теперь заполнены только наполовину. В результате последующие включения могут с некоторой вероятностью выполняться без расщепления узлов. Таким образом, одно расщепление подготавливает условия для нескольких простых включений. Во-вторых, вверх по дереву всегда посылается срединный ключ, и совсем не обязательно тот, который включается. Отсюда повторные включения имеют тенденцию повышать сбалансированность дерева, независимо от того, в каком порядке поступают новые ключи.

### 11.3.5. Алгоритмы на языке Pascal: поиск и включение

Разработку алгоритмов на языке Pascal для поиска и включения в В-дерево начнем с объявлений, требуемых для создания В-дерева. Ради упрощения мы сконструируем наше В-дерево целиком в оперативной памяти, используя для описания его структуры указатели языка Pascal. В большинстве приложений эти указатели будут заменены адресами различных блоков или страниц на диске, и ссылки на указатель фактически будут преобразовываться в обращения к диску.

#### 1. Объявления

Мы полагаем, что `keytype` и `treeentry` уже определены. Каждый элемент помимо той или иной информации содержит ключ. Внутри каждого узла находится список элементов и список указателей на дочерние узлы. Поскольку эти списки имеют небольшую длину, мы для простоты используем непрерывные массивы и отдельную переменную `count` для их представления.

реализация  
В-дерева

```

const
  max = 4;           { максимальное число элементов в узле; max = m - 1 }
  min = 2;           { минимальное число элементов в узле; min = ⌈ $\frac{1}{2}m$ ⌉ - 1 }
type
  treepointer = ↑treeentry;
  index = 0..max;
  treenode = record
    count: index;      { за исключением корня нижняя граница есть min }
    entry: array [1..max] of treeentry;
    branch: array [index] of treepointer
  end;
```

Способ организации индексов предполагает, что в одном из наших алгоритмов нам понадобится отдельно проанализировать `branch[0]`, а затем рассматривать каждый ключ, связанный с ветвью, расположенной с правой стороны.

## 2. Поиск

В качестве простого примера напишем процедуру поиска в В-дереве ключа-мишени. Входными параметрами для процедуры будут ключ-мишень и указатель на корень В-дерева. Первым выходным параметром будет булева переменная *found*, которая указывает, найдена ли мишень в дереве. Если мишень найдена, тогда второй выходной параметр указывает на узел с ключом-мишенью, а третий параметр — на позицию мишени в этом узле.

В целом метод поиска путем просмотра дерева сверху вниз аналогичен поиску в двоичном дереве поиска. В многовариантном дереве, однако, мы должны исследовать каждый узел более тщательно с целью определения ветви, на которую надо перейти в следующем шаге. Это исследование выполняется вспомогательной процедурой *SearchNode*, которая возвращает выходные параметры *found* и *targetpos*, последний из которых задает позицию мишени, если она найдена, а в противном случае — номер ветви, на которой следует продолжить поиск.

```

procedure SearchTree (target: keytype; root: treepointer;  { Поиск в дереве }
                      var found: Boolean;
                      var targetnode: treepointer; var targetpos: index);
{ Pre: В-дерево, на которое указывает root, уже создано.
  Post: Если ключ target присутствует в В-дереве, тогда found
        становится равным true, а targetnode указывает на узел,
        содержащий target в позиции targetpos. В противном случае
        found становится равным false, а targetnode и targetpos
        не определены.
  Uses: Использует процедуру SearchTree рекурсивно; процедуру
        SearchNode. }

begin                                     { процедура SearchTree }
  if root = nil then
    found := false
  else
    begin
      SearchNode(target, root, found, targetpos);
      if found then
        targetnode := root
      else
        SearchTree (target, root↑.branch [targetpos], found, targetnode, targetpos)
    end
  end;                                     { procedure SearchTree }

```

Эта процедура написана рекурсивным образом, чтобы подчеркнуть сходство ее структуры с процедурой включения, к которой мы сейчас перейдем. Поскольку здесь используется хвостовая рекурсия, ее легко заменить при желании итеративным алгоритмом.

## 3. Поиск в узле

Эта процедура определяет, входит ли мишень в текущий узел и, если нет, находит, какая из *count + 1* ветвей будет содержать ключ-мишень. Для удобства возможность перехода на ветвь 0 рассматривается отдельно, а затем выполняется последовательный поиск по оставшимся вариантам, начиная с конца, чтобы ключ в индексе 1 мог служить сигнальной меткой для останова поиска.

```

procedure SearchNode (target: keytype; current: treepointer; { Поиск в узле }
                      var found: Boolean; var position: index);
{ Pre: Параметр target является ключом, а параметр current
  указывает на узел В-дерева.
  Post: Просматривает ключи в узле current↑ в поисках ключа target;
  через параметр position возвращает позицию мишени или
  переходит к ветви, где будет продолжен поиск. }
begin
  with current↑ do
    if target < entry[1].key then
      begin
        found := false;
        position := 0
      end
    else
      begin
        { начнем последовательный поиск по ключам }
        position := count;
        while (target < entry[position].key) and (position > 1) do
          position := position - 1;
          found := (target = entry[position].key)
        end
      end
    end;
  { процедура SearchNode }

```

Для В-деревьев большого порядка эту процедуру следует модифицировать и использовать в ней не последовательный, а двоичный поиск. В некоторых приложениях в узлах В-дерева вместе с ключом хранится значительный объем информации, поэтому порядок В-дерева оказывается относительно небольшим, и последовательный поиск внутри узла вполне допустим. Однако в других приложениях в узлах содержатся только ключи, поэтому порядок дерева оказывается значительно больше, и для определения позиции ключа внутри узла следует использовать двоичный поиск.

Есть, однако, еще одна возможность: вместо последовательного массива элементов в каждом узле использовать связанное двоичное дерево поиска; этот вариант будет подробно рассмотрен позже в этой же главе.

#### 4. Включение: главная процедура

Включение в В-дерево наиболее естественно сформулировать как рекурсивную процедуру, поскольку после того, как включение в поддереву завершится, может остаться (срединный) ключ, который должен быть заново включен выше по дереву. Рекурсия позволяет нам сохранять последовательность пройденных позиций в дереве и дает возможность возвращаться назад, не требуя при этом создания явного вспомогательного стека.

Мы будем предполагать, что включаемого ключа в дереве еще нет. В этом случае процедуре включения требуются всего два параметра: newentry, включаемый ключ, и root, корень В-дерева. Однако для организации рекурсии нам понадобятся три дополнительных выходных параметра. Первый — это булева переменная pushup, которая указывает, был ли расщеплен на два узла корень поддерева; в этом случае образуется (срединный) ключ, который надо будет заново включить выше по дере-

ву. При наличии такой ситуации мы примем соглашение, по которому старый корневой узел будет содержать левую половину элементов, а новый узел — правую половину элементов. Когда выполняется расщепление, второй выходной параметр medentry представляет собой срединный ключ, а третий параметр medright является указателем на новый узел, правую половину первоначального корня  $p \uparrow$  данного поддерева.

Для правильной обработки всех этих параметров мы будем выполнять рекурсию в процедуре с именем PushDown. Эта ситуация показана на рис. 11.8, где мы обозначили срединный ключ через  $x$ , а указатель на правую часть через  $xr$ .

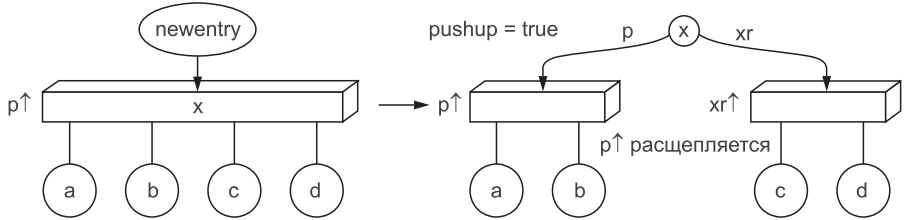


Рис. 11.8. Действие процедуры PushDown

Рекурсия запускается главной процедурой InsertTree. Если при выходе из самого внешнего вызова процедуры PushDown переменная pushup приняла значение true, значит, имеется ключ для помещения в новый корень, и высота всего дерева должна увеличиться. Главная процедура выглядит следующим образом.

```

procedure InsertTree (newentry: treeentry; var root: treepointer);
    { Включить в дерево }
{ Pre: В-дерево, на которое указывает параметр root, уже создано;
    ни в одном элементе В-дерева ключ не совпадает с ключом
    newentry.key.
Post: Элемент newentry включен в В-дерево.
Uses: Использует процедуру PushDown. }
var pushup: Boolean; { увеличилась ли высота дерева? }
    medentry: treeentry; { элемент, который следует включить
    заново в качестве нового корня }
    medright, { указатель на поддерево с правой стороны от medentry }
    newroot: treepointer; { используется только в случае
    увеличения высоты дерева }
begin
    PushDown(newentry, root, pushup, medentry, medright);
    if pushup then begin { все дерево увеличивается по высоте }
        new(newroot); { сделаем для всего В-дерева совершенно новый корень }
        with newroot $\uparrow$  do begin
            count := 1;
            entry[1] := medentry;
            branch[0] := root;
            branch[1] := medright;
            root := newroot
        end
    end
end; { процедура InsertTree }

```

## 5. Рекурсивное включение в поддереву

Теперь обратимся к рекурсивной процедуре PushDown, которая использует параметр current для указания на корень просматриваемого поддерева. В В-дереве новый ключ сначала включается в лист. Поэтому для завершения рекурсии мы можем использовать условие `current = nil`; другими словами, мы будем продолжать двигаться вниз по дереву в поисках newentry до тех пор, пока нам не встретится пустое поддерево. Поскольку В-дерево не растет посредством добавления новых листьев, мы не включаем newentry немедленно, но устанавливаем переменную pushup в состояние true и посылаем ключ (который теперь называется medentry) вверх для включения.

Когда осуществляется возврат из рекурсивного вызова, и pushup равно true, мы пытаемся включить medentry в текущий узел. Если там есть место, процедура завершается. В противном случае текущий узел current расщепляется на узлы current↑ и medright↑, а (возможно другой) срединный ключ medentry посылается вверх по дереву. Процедура использует три вспомогательные процедуры: SearchNode (такую же, что и для поиска); PushIn помещает ключ medentry в узел current↑ (при условии, что там есть место); Split разбивает заполненный узел current↑ на два узла.

```

procedure PushDown (newentry: treeentry; current: treepointer;
                     var pushup: Boolean; var medentry: treeentry;
                     var medright: treepointer);      { Протолкнуть вниз }
{ Pre: Элемент newentry принадлежит поддереву, на которое
  указывает параметр current.
  Post: Элемент newentry включен в поддерево, на которое указывает
  current; если переменная pushup приняла значение true, тогда
  высота поддерева увеличилась, а элемент medentry требуется
  включить заново выше по дереву, с поддеревом medright
  с правой стороны от него.
  Uses: Использует процедуру PushDown рекурсивно; процедуры
  SearchNode, Split, PushIn. }
var position: index;      { ветвь, с которой следует продолжить поиск }
    found: Boolean;        { элемент newentry уже есть в дереве (ошибка)? }
begin                                { процедура PushDown }
  if current = nil then begin          { включение в пустое дерево невозможно;
                                         рекурсия завершается }
    pushup := true;
    medentry := newentry;
    medright := nil
  end
  else begin                          { поиск в текущем узле }
    SearchNode(newentry.key, current, found, position);
    if found then
      Error('Включение ключа-дубликата в В-дереву');
    Push Down (newentry, current↑.branch [position], pushup, medentry,
              medright);
  if pushup then                      { включить заново срединный элемент medentry }
    with current↑ do
      if count < max then begin
        pushup := false;
        PushIn(medentry, medright, current, position)
      end
  end

```

```

    else begin
        pushup := true;
        Split(medentry, medright, current, position, medentry, medright)
    end
end
end;
{ процедура PushDown }

```

## 6. Включение ключа в узел

Следующая процедура помещает ключ `medentry` и его правый указатель `medright` в узел `current↑` при условии, что там есть место для этого включения.

```

procedure PushIn (medentry: treeentry; medright, { Протолкнуть внутрь }
                  current: treepointer; position: index);
{ Pre: Ключ medentry принадлежит индексу position в узле current↑,
  в котором есть для него место.
  Post: Включает ключ medentry и указатель medright в current↑
  по индексу position. }
var i: index; { индекс перемещения ключей
               для освобождения места для medentry }
begin
  with current↑ do begin
    for i := count downto position + 1 do
      begin { Сдвинем все ключи и ветви вправо }
        entry [i + 1] := entry [i];
        branch [i + 1] := branch [i]
      end;
    entry [position + 1] := medentry;
    branch [position + 1] := medright;
    count := count + 1
  end
end;
{ процедура Pushin }

```

## 7. Расщепление заполненного узла

Следующая процедура включает ключ `medentry` вместе с указателем `medright` на поддерево в заполненный узел `current↑`, расщепляет этот узел с образованием из его правой части нового узла `newright↑`, и посылает срединный ключ `newmedian` вверх для будущего включения его заново. Разумеется, не представляется возможным включение ключа `medentry` непосредственно в заполненный узел: вместо этого мы должны сначала определить, должен ли `medentry` попасть в левую или правую половину, разделить узел (в позиции `median`) соответствующим образом, и затем включить `medentry` в подходящую половину. Пока выполняются эти действия, мы оставляем срединный ключ `newmedian` в левой половине.

```

procedure Split (medentry: treeentry; medright, current: treepointer;
                 position: index; var newmedian: treeentry;
                 var newright: treepointer); { Расщепление }
{ Pre: Ключ medentry принадлежит индексу position в узле current↑,
  который заполнен.

```



**Post:** *Расщепляет узел  $\text{current}\uparrow$  с элементом  $\text{medentry}$  и указателем  $\text{medright}$  по индексу  $\text{position}$  на узлы  $\text{current}\uparrow$  and  $\text{newright}\uparrow$  со срединным элементом  $\text{newmedian}$ .*

**Uses:** *Использует процедуру  $\text{PushIn}$ . }*

```

var
i,                                { индекс цикла, используемый для копирования
                                   из  $\text{current}\uparrow$  в новый узел }
median: index;                    { средняя позиция в комбинированном
                                   переполненном узле }
begin
  if position <= min then         { Элемент  $\text{medentry}$  идет в левую половину }
    median := min
  else
    median := min + 1;
  new(newright);                  { Получим новый узел и поместим его справа }
  with  $\text{current}\uparrow$  do
  begin
    for i := median + 1 to max do
    begin
      newright $\uparrow$ .entry [i - median] := entry [i];
      newright $\uparrow$ .branch [i - median] := branch [i]
    end;
    newright $\uparrow$ .count := max - median;
    count := median;
    if position <= min then
      PushIn(medentry, medright, current, position)
    else
      PushIn(medentry, medright, newright, position - median);
    newmedian := entry [count];
    newright $\uparrow$ .branch [0] := branch [count];
    count := count - 1
  end
end;                               { процедура Split }

```

найдем точку  
расщепления

переместим ключи  
в правый узел

включим новый  
элемент

## 11.3.6. Удаление из В-дерева

### 1. Метод

В процессе включения новый элемент всегда сначала помещается в лист. При удалении мы также должны извлечь элемента из листа. Если удаляемый элемент не находится в листе, тогда его непосредственный предшественник (или последователь) в естественном порядке ключей гарантированно находится в листе (докажите это!). Таким образом, мы можем переместить непосредственного предшественника (или последователя) в позицию, занимаемую удаляемым элементом, и удалить элемент из листа.

Если лист содержит более минимального числа элементов, тогда один из них может быть удален без каких-либо дополнительных действий. Если лист содержит минимальное число элементов, тогда мы сначала смотрим на два листа (или, в случае внешнего узла, на один лист), которые непосредственно примыкают друг к другу и являются дочерними от одного узла. Если один из этих листьев содержит более минимального числа элементов, тогда один из них может быть перемещен в роди-

тельский узел, а элемент из родительского узла перемещен в лист, где осуществляется удаление. Если, наконец, примыкающий лист содержит точно минимальное число элементов, тогда два листа и срединный элемент из родительского узла могут быть объединены в один новый лист, который будет содержать не более максимально допустимого числа элементов. Если этот шаг оставляет родительский узел со слишком малым числом элементов, тогда процесс продвигается вверх. В граничном случае последний элемент удаляется из корня, и тогда высота дерева уменьшается.

## 2. Пример

Процесс удаления из рассмотренного ранее В-дерева порядка 5 показан на рис. 11.9. Первое удаление,  $h$ , выполняется из листа, в котором имеется более минимального числа элементов, и не вызывает никаких проблем. Второе удаление,  $r$ , выполняется не из листа, поэтому непосредственный последователь  $r$ , именно,  $s$ , продвигается в позицию  $r$ , после чего  $s$  удаляется из своего листа. Третье удаление,  $p$ , оставляет свой узел со слишком малым числом элементов. Поэтому ключ  $s$  из родительского узла перемещается вниз и замещается ключом  $t$ .

Удаление  $d$  приводит к более далеко идущим последствиям. Это удаление оставляет узел со слишком малым числом элементов, при этом ни один из его «братских» узлов, т. е. узлов, находящихся на том же уровне и с тем же подчинением, не может обойтись без своих элементов. Поэтому узел объединяется с одним из этих «братских» узлов и со срединным элементом из родительского узла, как это показано пунктирной линией на первой диаграмме и в виде объединенного узла  $a b c e$  на второй диаграмме. Этот процесс, однако, оставляет родительский узел лишь с одним ключом  $f$ . Следовательно, верхние три узла дерева должны быть объединены, что приводит к конфигурации, показанной на последней диаграмме рис. 11.9.

### 3. Pascal-процедура

Мы можем написать алгоритм удаления, общая структура которого близка к той, что использовалась для включения. Опять мы используем рекурсию и отдельную главную процедуру, чтобы эту рекурсию активизировать. Вместо того, чтобы пытаться в ходе внутреннего рекурсивного вызова переместить элемент из родительского узла вниз, мы позволим рекурсивной процедуре осуществить возврат, даже если в корневом узле оказалось слишком мало элементов. Далее внешний вызов обнаружит это несоответствие и переместит элементы должным образом.

Главная процедура выглядит следующим образом.

{ Удалить из дерева }

**{ Pre:** Параметр *target* является ключом некоторого элемента в B-дереве, на которое указывает *root*.

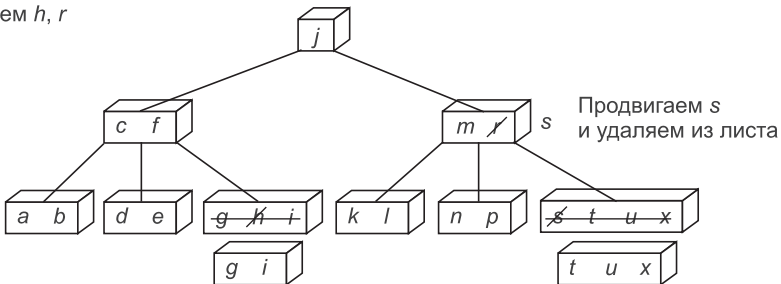
**Post:** Этот элемент был удален из B-дерева.

**Uses:** *Использует процедуру RecDeleteTree. }*

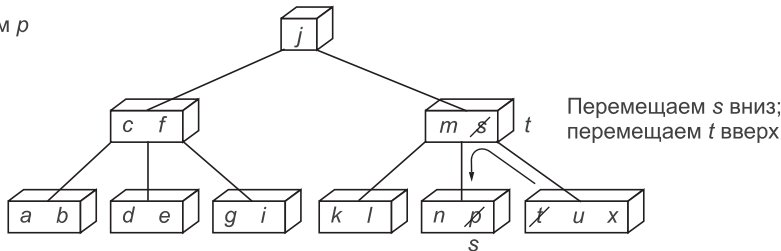
```
var oldroot: treepointer;
```

{ используется для того  
чтобы избавиться от пустого корня }

1. Удаляем  $h, r$



2. Удаляем  $p$



3. Удаляем  $d$

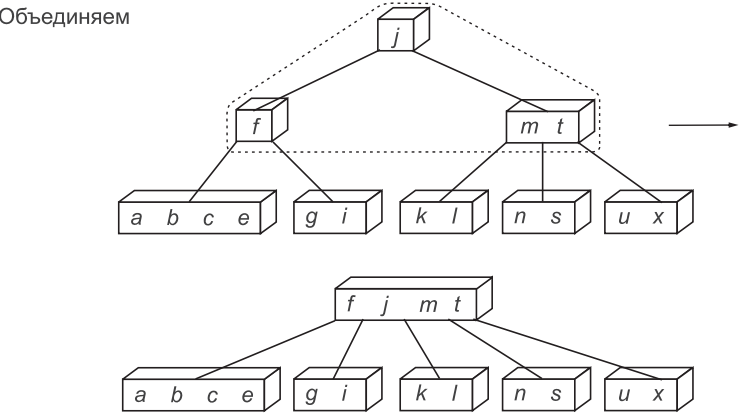
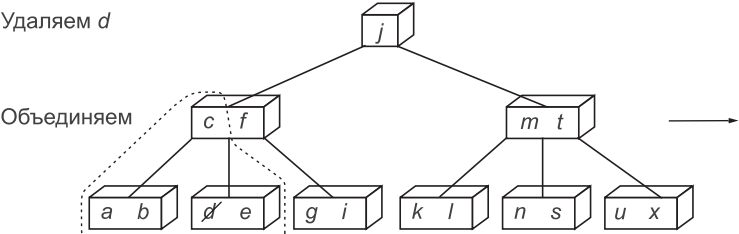


Рис. 11.9. Удаление из B-дерева

```
begin                                     { процедура DeleteTree }
RecDeleteTree(target, root);
if root↑.count = 0 then begin           { root теперь пуст }
    oldroot := root;
    root := root↑.branch[0];
    dispose(oldroot)
end
end;                                    { процедура DeleteTree }
```

#### 4. Рекурсивное удаление

Большая часть работы выполняется в рекурсивной процедуре. В ней прежде всего осуществляется поиск мишени в текущем узле. Если ключ найден, и узел не является листом, тогда ищется непосредственный последователь ключа, помещается в текущий узел, а из своей прежней позиции удаляется. Удаление из листа осуществляется без каких-либо ухищрений, в противном же случае процесс рекурсивно продолжается. Когда происходит возврат из рекурсивного вызова, процедура проверяет, осталось ли в соответствующем узле минимально допустимое число элементов, и если нет, то перемещает элементы должным образом. В некоторых из этих шагов используются вспомогательные процедуры.

```

procedure RecDeleteTree (target: keytype; current: treepointer);
    { Рекурсивное удаление из дерева }
{ Pre: Мишень target является ключом некоторого элемента
  в поддереве B-дерева, на который указывает current.
Post: Этот элемент удален из B-дерева.
Uses: Использует RecDeleteTree рекурсивно; использует процедуры
  SearchNode, Successor, Remove, Restore. }
var position: index;
    { местоположение target или ветви,
      в которой следует осуществлять поиск }
    found: Boolean;
    { находится ли target в текущем узле? }
begin
    { процедура RecDeleteTree }
    if current = nil then
        Error('Мишень target не найдена: попытка удаления из пустого поддерева')
    else with current↑ do begin
        SearchNode(target, current, found, position);
        if found then
            { мишень находится в текущем узле }
            if branch [position - 1] = nil then { случай: current↑ является листом }
                Remove(current, position)
                { удаляет key по индексу position
                  узла current↑ }
            else begin
                { случай: узел current↑ не является листом }
                Successor(current, position);
                { заменяет элемент entry[position]
                  его последователем }
                RecDeleteTree(entry [position].key, branch [position]);
            end
        else
            { мишень не найдена в текущем узле }
            RecDeleteTree(target, branch [position]);
        if branch [position] <> nil then
            if branch [position]↑.count < min then Restore(current, position)
    end
end;
    { процедура RecDeleteTree }

```

## 5. Вспомогательные процедуры

Теперь мы можем завершить процесс удаления из В-дерева, написав несколько вспомогательных процедур, выполняющих ряд требуемых действий. Процедура Remove просто удаляет элемент из узла В-дерева и переходит к его правой ветви. Эта процедура активизируется лишь в тех случаях, когда удаляемый элемент находится в листе дерева.

```

procedure Remove (p: treepointer; position: index);           { Удалить }
{ Pre:   Указатель current указывает на узел в В-дерева с элементом
           по индексу position.
  Post:  Этот элемент и ветвь справа от него удалены из current↑. }
var
  i: index;                                     { индекс цикла для перемещения элементов }
begin                                           { процедура Remove }
  with current↑ do
    begin
      for i := position + 1 to count do
        begin
          entry [i - 1] := entry [i];
          branch [i - 1] := branch [i]
        end;
      count := count - 1
    end
end;                                           { процедура Remove }

```

Процедура Successor активизируется, когда удаляемый элемент находится в узле, не являющемся листом. В этом случае поиск непосредственного последователя (в порядке расположения ключей) осуществляется путем перемещения к ветви, находящейся справа от элемента, а затем продвижения по самым левым ветвям вплоть до листа. Самый левый элемент в этом листе затем замещает удаляемый элемент.

```

procedure Successor (current: treepointer; position: index); { Последователь }
{ Pre:   Указатель current указывает на узел в В-дерева с элементом
           по индексу position.
  Post:  Этот элемент заменяется своим непосредственным
           последователем в порядке следования ключей. }
var
  leaf: treepointer;                             { используется для перемещения по дереву
                                                    вниз до листа }
begin                                           { процедура Successor }
  leaf := current↑.branch [position];           { сначала пойдем вправо
                                                    от текущего узла }
  while leaf↑.branch [0] <> nil do              { сдвинемся влево
                                                    насколько это возможно }
    leaf := leaf↑.branch [0];
    current↑.entry [position] := leaf↑.entry [1]
end;                                           { процедура Successor }

```

Наконец, мы должны показать, как восстановить в узле current↑.branch[position] минимально необходимое число элементов, если в результате рекурсивного вызова число элементов стало ниже допустимого минимума. Наша процедура, можно сказать, левоориентирована; она





```

procedure Combine (current: treepointer; position: index);      { Объединение }
{ Pre:    Указатель current указывает на узел в B-дереве с элементами
           в ветвях position и position - 1, со слишком малым числом
           элементов для их перемещения.
  Post:    Узлы в ветвях position - 1 и position были объединены в один
           узел, который также включает элемент, ранее находившийся
           в узле current↑ по индексу position. }

var
  i: index;                      { индекс цикла для перемещения элементов }
  rightbranch: treepointer;

begin                                { процедура Combine }
  rightbranch := current↑.branch [position];
  with current↑.branch [position - 1]↑ do begin          { левая ветвь }
    count := count + 1;
    entry [count] := current↑.entry [position];
    branch [count] := rightbranch↑.branch [0];
    for i := 1 to rightbranch↑.count do
      begin
        count := count + 1;
        entry [count] := rightbranch↑.entry [i];
        branch [count] := rightbranch↑.branch [i]
      end
    end;
  with current↑ do begin
    for i := position to count - 1 do
      begin
        entry [i] := entry [i + 1];
        branch [i] := branch [i + 1]
      end;
    count := count - 1
  end;
  dispose(rightbranch)
end;                                { процедура Combine }

```

### Упражнения 11.3

**Е1.** Включите шесть оставшихся букв алфавита в порядке

z, v, o, q, w, y

в последнее В-дерево на рис. 11.7.

**E2.** Включите перечисленные ниже элементы в указанном порядке в первоначально пустое В-дерево порядка **(a)** 3, **(b)** 4, **(c)** 7.

*a g f b k d h n j e s i r x c l n t u p*

**Е3.** Каково минимальное число элементов, которые, будучи включены в соответствующем порядке, приведут к тому, что В-дерево порядка 5 будет иметь высоту 2 (т. е. 3 уровня)?

**Е4.** Нарисуйте все В-деревья порядка 5 (с числом ключей на узел от 2 до 4), которые можно сконструировать из ключей 1, 2, 3, 4, 5, 6, 7 и 8?

**Е5.** Докажите, что если ключ в В-дереве не является листом, то его непосредственный предшественник и непосредственный последователь (в естественном порядке) есть листья.



- Е6.** Предположим, что аппаратура дисковой системы позволяет нам произвольно выбирать размер дисковых записей, но время, требуемое для чтения записи с диска, составляет  $a + bd$ , где  $a$  и  $b$  являются константами, а  $d$  есть порядок В-дерева. (В каждой записи на диске хранится один узел В-дерева.) Пусть  $n$  есть число элементов в В-дереве. Предположим для простоты, что все узлы В-дерева заполнены (каждый содержит  $d - 1$  элементов).
- (a) Объясните, почему время, требуемое для выполнения типичной операции над В-деревом (например, поиска или включения) приблизительно составляет  $(a + bd)\log_d n$ .
  - (b) Покажите, что требуемое время минимизируется, когда значение  $d$  удовлетворяет условию  $d(\ln d - 1) = a/b$ . (Обратите внимание на то, что это выражение не зависит от числа  $n$  элементов в В-дереве.) [Подсказка: Для фиксированных значений  $a$ ,  $b$  и  $n$  требуемое время есть функция от  $d$ :  $f(d) = (a + bd)\log_d n$ . Заметьте, что  $\log_d n = (\ln n)/(\ln d)$ . Для нахождения минимума необходимо вычислить производную  $f'(d)$  и приравнять ее 0.]
  - (c) Пусть  $a = 20$  миллисекунд, а  $b = 0.1$  миллисекунд. (Записи очень короткие.) Найдите значение  $d$  (приблизительное), при котором требуемое время минимально.
  - (d) Пусть  $a = 20$  миллисекунд, а  $b = 10$  миллисекунд. (Записи длинные.) Найдите значение  $d$  (приблизительное), при котором требуемое время минимально.
- Е7.** Напишите процедуру, которая просматривает В-дерево, посещая все его элементы в порядке ключей (сначала меньшие ключи, затем большие).
- Е8.** Определите *прямой просмотр* В-дерева рекурсивным образом, имея в виду сначала посещение всех элементов корневого узла, затем просмотр всех поддеревьев слева направо в прямом порядке. Напишите процедуру, которая будет просматривать В-дерево в прямом порядке.
- Е9.** Определите *обратный просмотр* В-дерева рекурсивным образом, имея в виду сначала просмотр всех поддеревьев корневого узла слева направо в обратном порядке, затем посещение всех элементов корневого узла. Напишите процедуру, которая будет просматривать В-дерево в обратном порядке.
- Е10.** Удалите хвостовую рекурсию из процедуры Search.
- Е11.** Перепишите процедуру SearchBinary так, чтобы она использовала двоичный поиск.
- Е12.** *В\*-дерево* есть В-дерево, в котором каждый узел, за исключением, возможно, корня, заполнен по меньшей мере на две трети, а не наполовину. Включение в В\*-дерево перемещает по мере необходимости элементы между «братскими» узлами (как это делалось при удалении), тем самым откладывая расщепление узла до тех пор, пока оба «братских» узла не будут заполнены целиком. Эти два узла могут тогда

быть расщеплены на три, каждый из которых будет заполнен по меньшей мере на две трети.

- (a) Укажите, какие изменения необходимо внести в алгоритм включения, чтобы он поддерживал свойства В\*-деревя.
- (b) Укажите, какие изменения необходимо внести в алгоритм удаления, чтобы он поддерживал свойства В\*-деревя.
- (c) Обсудите относительные преимущества и недостатки В\*-деревьев сравнительно с обычными В-деревьями.

### Программные проекты 11.3

- P1.** Объедините все процедуры, описанные в этом разделе, в управляемую меню демонстрационную программу для В-деревьев. Если вы разработали демонстрационную программу для деревьев двоичного поиска (раздел 10.2, проект P2) достаточно тщательно, мы сможем непосредственно заменить один пакет операций другим.
- P2.** С помощью В-дерева порядка 5 реализуйте операции гардероба из раздела 7.3. Протестируйте ваш пакет с демонстрационной программой гардероба, а затем запустите программу анализа процессорного времени с пакетом для В-деревьев, сравнив получаемые результаты с результатами при других реализациях.
- P3.** Подставьте процедуры извлечения и включения для В-деревьев в проект извлечения информации из секции 10.2 (проект P6). Сравните производительность В-деревьев с производительностью двоичных деревьев поиска для различных комбинаций данных во входных тестовых файлах и различных порядков В-деревьев.

гардероб

## 11.4. Красно-черные деревья

### 11.4.1. Введение

узлы В-дерева

В последнем разделе мы для сохранения элементов в каждом узле В-дерева использовали непрерывные списки. Это было достаточно удобно, потому что число элементов в одном узле обычно невелико; кроме того, мы эмулировали методы, применимые к внешним файлам на магнитных дисках, когда использование динамической памяти затруднено.

представление  
двоичного дерева

В общем случае, однако, для хранения элементов в каждом узле В-дерева мы можем использовать любую упорядоченную структуру. Прекрасным средством в этом случае являются небольшие двоичные деревья поиска. Нам только следует проявлять осторожность и различать связи внутри конкретного узла В-дерева и связи между одним В-деревом и другим. Чтобы подчеркнуть это различие, будем рисовать связи внутри одного узла волнистыми линиями (и будем считать, что они для большей наглядности изображены красным цветом), а связи между В-деревьями — обычными прямыми черными линиями. На рис. 11.11 показано В-дерево порядка 4 с такими обозначениями.

черные и красные  
связи

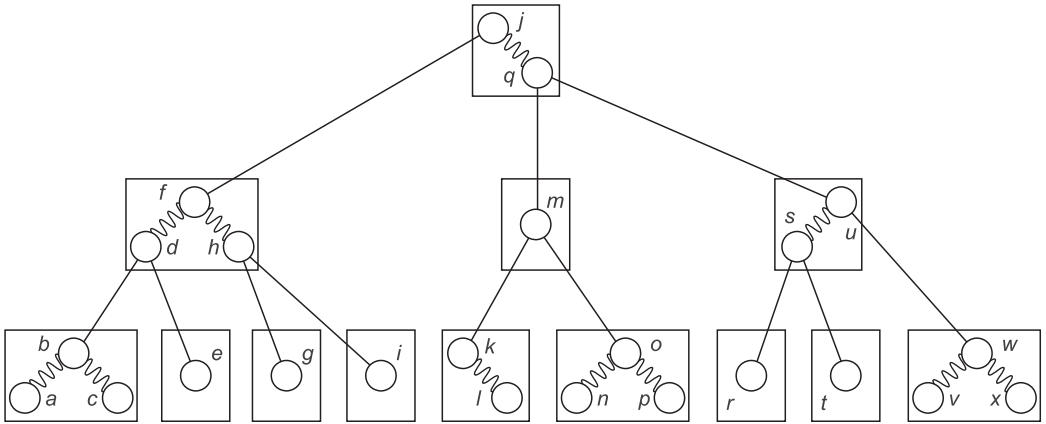


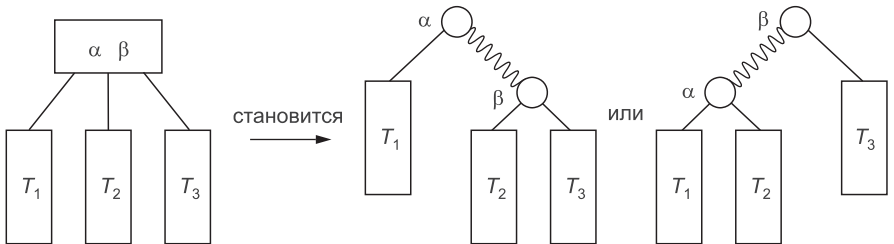
Рис. 11.11. В-дерево порядка 4, представленное в виде двоичного дерева поиска

### 11.4.2. Определения и анализ

Наша конструкция особенно полезна для В-деревьев порядка 4 (как на рис. 11.11), где каждый узел дерева содержит один, два или три элемента. Узел с одним элементом одинаков и в В-дереве, и в дереве двоичного поиска; узел с тремя элементами преобразуется следующим образом:



Узел с двумя элементами имеет два возможных представления:



При желании мы можем всегда использовать только одну из приведенных на рисунке форм, но особых оснований к этому нет; позже мы увидим, что наши алгоритмы естественным образом создают обе воз-

можные формы, поэтому будем считать, что они в равной степени представляют В-дерево с двумя элементами.

первое  
определение

Отсюда мы получаем фундаментальное определение этого раздела: красно-черное дерево представляет собой дерево двоичного поиска, у которого связи покрашены в красный или черный цвет, и которое получено из В-дерева порядка 4 только что описанным способом. После того, как мы преобразовали В-дерево в красно-черное дерево, мы можем использовать его как любое другое двоичное дерево поиска. В особенности, поиск и просмотр красно-черного дерева выполняются в точности так же, как и в случае обычного двоичного дерева поиска; мы просто игнорируем цвет связей. Однако, включение и удаление требуют большего внимания, если мы хотим сохранить структуру базового В-дерева. Давайте поэтому преобразуем требования к В-дереву в соответствующие требования к красно-черным деревьям.

цветные узлы

Сначала, однако, примем некоторые дополнительные обозначения. Мы будем рассматривать каждый узел красно-черного дерева как покрашенный в цвет связи, расположенной *над ним*; тогда мы сможем говорить о красных узлах и черных узлах вместо красных связей и черных связей. В этом случае для указания цвета каждого узла нам потребуется хранить лишь один дополнительный бит.

цвет корня

Поскольку корень не имеет над собой никаких связей, он не получает цвета описанным выше способом. Ради упрощения некоторых алгоритмов мы примем соглашение, что корень покрашен черным цветом. Аналогично будем считать, что все пустые поддеревья (соответствующие *nil*-связям) покрашены черным.

черное условие

Первое условие определения В-дерева, именно, что все его пустые поддеревья должны быть на одном уровне, означает, что каждый простой путь от корня к пустому поддереву (*nil*) проходит через одно и то же число узлов В-дерева. Соответствующее красно-черное дерево имеет один черный узел (и, возможно, один или два красных узла) для каждого узла В-дерева. Отсюда мы получаем **черное условие**:

*Каждый простой путь от корня к пустому поддереву проходит через одно и то же число черных узлов.*

Утверждение, что В-дерево удовлетворяет свойствам дерева поиска, автоматически удовлетворяется для красно-черного дерева, и, для порядка 4, оставшиеся части определения сводятся к заявлению, что каждый узел содержит один, два или три элемента. Для красно-черных деревьев нам требуется условие, гарантирующее, что не более трех узлов определяются вместе (красными связями) как один узел В-дерева, и что узлы с тремя связями находятся в сбалансированной форме. Эта гарантия возникает из **красного условия**:

красное условие

*У красного узла родительский узел всегда черный.*

(Поскольку мы условились, что корень всегда черный, родительский узел красного узла всегда существует.)

Мы можем подытожить это обсуждение представлением формального определения, которое уже вообще не ссылается на В-деревья:

**Определение**

**Красно-черное дерево** есть дерево двоичного поиска, в котором каждый узел имеет **красный** или **черный** цвет, и которое удовлетворяет следующим требованиям:

1. Каждый простой путь от корня к пустому поддереву (**nil**-связь) проходит через одно и то же число черных узлов.
2. Если узел красный, у него существует родительский узел, и этот узел черный.

Из этого определения следует, что никакой путь от корня к пустому поддереву не может превышать любой другой путь по длине более, чем в два раза, поскольку, в соответствии с красным условием, не более чем половина узлов на таком пути могут быть красными, а в соответствии с черным условием на каждом таком пути имеется одно и то же число черных узлов. В результате мы получаем:

**Теорема 11.2**

*Высота красно-черного дерева, содержащего  $n$  узлов, не может превышать  $2\lg n$ .*

производительность поиска

Отсюда следует, что время поиска в красно-черном дереве с  $n$  узлами составляет  $O(\lg n)$  в любом случае. Позже мы покажем, что время включения также составляет  $O(\lg n)$ , но сначала мы должны разработать соответствующий алгоритм.

### 11.4.3. Включение

общая схема

Давайте начнем со стандартного рекурсивного алгоритма для включения в двоичное дерево поиска. Мы сравниваем новый ключ-мишень с ключом корня (если дерево не пусто) и затем рекурсивно включаем новый элемент в левое или правое поддерево корня. Этот процесс завершается, когда мы сталкиваемся с пустым поддеревом, после чего мы создаем новый узел и присоединяем его к дереву вместо пустого поддерева.

новый узел

Должен ли этот новый узел быть красным или черным? Если мы сделаем его черным, мы увеличим число черных узлов на одном пути (и только на одном пути), тем самым нарушая черное условие. Следовательно, новый узел должен быть красным. (Вспомним также, что включение нового узла в В-дерево сначала происходит в существующем узле, процесс, соответствующий присоединению нового красного узла к красно-черному дереву.) Если родительский узел нового красного узла черный, включение на этом заканчивается, но если родительский узел красный, тогда мы сталкиваемся с нарушением красного условия. Основная часть работы алгоритма включения заключается в ликвидации этого нарушения, и мы обнаружим несколько различных случаев, которые придется обрабатывать по отдельности.

откладывание работы

Мы, однако, можем существенно упростить наш алгоритм, если не будем рассматривать эти случаи сейчас, а отложим это рассмотрение на возможно более дальний срок. Тогда, если мы создаем новый красный узел, мы не пытаемся немедленно восстанавливать дерево, а вместо этого просто возвращаемся из рекурсивного вызова с установленным индикатором состояния, который говорит, что только что обработанный узел был красным.

переменная состояния

родительский  
узел: нарушение  
красного условия

После этого возврата мы снова обрабатываем родительский узел. Если он черный, тогда условия красно-черного дерева удовлетворяются и процесс завершается. Если же он красный, тогда снова мы не пытаемся немедленно восстановить дерево, а вместо этого устанавливаем переменную состояния, указывающую, что у нас есть два прилегающих красных узла, и затем просто возвращаемся из рекурсивного вызова. В этом случае оказывается полезным использовать переменную состояния для указания на отношение двух красных узлов — как левый или как правый дочерний узел.

После возврата из второго рекурсивного вызова мы обрабатываем родительский узел родительского узла (узел-«дед»). Здесь нам помогает принятое нами соглашение, что корень всегда должен быть черным: поскольку родительский узел красный, он не может быть корнем, и, следовательно, узел-«дед» существует. Более того, этот узел-«дед» гарантированно черный, поскольку дочерний от него узел (т. е. наш родительский узел) красный, и единственное нарушение красного условия находится ниже по дереву.

узел-«дед»:  
восстановление

Наконец, на рекурсивном уровне узла-«деда» мы можем преобразовать дерево, чтобы восстановить красно-черные условия. Мы будем рассматривать только случаи, в которых родительский узел является левым дочерним узлом узла-«деда»; обратные случаи являются симметричными. Мы должны различать два случая в зависимости от цвета другого (правого) дочернего узла от узла-«деда», т. е. «тети» или «дяди» исходного узла.

черная «тетя»

Сначала предположим, что узел-«тетя» черный. Сюда также входит возможность того, что этот узел вообще не существует. (Вспомним, что пустое поддерево считается черным.) В этом случае красно-черные свойства восстанавливаются одиночным или двойным поворотом вправо, как это показано в первых двух частях рис. 11.12. Вам следует удостовериться в том, что на обеих этих диаграммах поворот (и связанные с ним изменения цвета) устраняет нарушение красного условия и сохраняет черное условие, не изменяя число черных узлов на любом пути вниз по дереву.

красная «тетя»

Теперь предположим, что узел-«тетя» красный, как это показано в последних двух частях рис. 11.12. В этом случае преобразование упрощается: повороты здесь не нужны, однако происходят изменения цветов. Родительский узел и узел-«тетя» становятся черными, а узел-«дед» — красным. Опять следует удостовериться в том, что число черных узлов на любом пути вниз по дереву остается неизменным. Поскольку, однако, узел-«дед» стал красным, вполне возможно, что красное условие все еще остается нарушенным: узел-«прадед» мог оказаться красным. В этом случае процесс не должен завершиться. Нам потребуется установить индикатор состояния, чтобы показать, что у нас образовался новый красный узел, а затем продолжать выходы из рекурсии. Однако любое нарушение красного состояния приводит к перемещению по дереву на два уровня выше, и, поскольку узел черный, процесс в конце концов завершится. Возможно также, что этот процесс изменил цвет корня с черного на красный; поэтому в самом внешнем вызове мы должны при необходимости восстановить черный цвет корня.



### 11.4.4. Включение на языке Pascal

Давайте теперь преобразуем рассмотренный метод включения в программу на языке Pascal. Как обычно, почти всю работу мы выполним внутри рекурсивной процедуры, поэтому главная процедура включения выполняет лишь начальные установки и проверку на ошибки. Поскольку нам нужно, чтобы единственная переменная, являющаяся индикатором состояния, непрерывно изменяла свое значение по ходу включения, мы объявляем эту переменную *status* в главной процедуре. Единственное, что должна еще сделать главная процедура включения — это покрасить корень в черный цвет и выполнить некоторую проверку на ошибки. Программа выглядит следующим образом.

```

procedure InsertTree (var root: treepointer; target: keytype);
                                { Включить в дерево }
{ Pre:   Параметр root указывает на корень красно-черного дерева
        поиска; параметр target является ключом, который
        отсутствует в дереве.
  Post:  Узел, содержащий ключ target, включен в дерево, причем
        свойства красно-черного дерева восстановлены. }

type
outcome = (OK, rednode, rightred, leftred); { Эти значения,
        возвращаемые процедурой рекурсивного включения, описывают
        следующие результаты вызова:
  OK:      Цвет текущего корня (поддерева) не изменился; дерево
        удовлетворяет условиям красно-черного дерева.
  rednode:  Цвет текущего корня изменился с черного на красный;
        для восстановления красно-черных свойств модификации
        могут потребоваться, но могут и не потребоваться.
  rightred; Текущий корень и его правый "ребенок" теперь стали
        красными; требуется смена цвета или поворот.
  leftred:  Текущий корень и его левый "ребенок" теперь стали
        красными; требуется смена цвета или поворот.
        }

var status: outcome;
begin                                { процедура InsertTree }
  ReclInsertTree (root, target, status);
  if status = rednode then
    root↑.red := false                { Всегда расщепляйте корневой узел, чтобы
        он оставался черным. Это предохранит вас от появления двух
        красных узлов на верху дерева и от результирующей попытки
        поворота или изменения цветов с использованием
        несуществующего родительского узла. }
  else if status <> OK then
    Error('Включение с недопустимым красно-черным состоянием в корне' )
end;                                { процедура InsertTree }

```

Рекурсивная процедура *ReclInsertTree* выполняет собственно включение, просматривая дерево обычным образом до тех пор, пока она не натолкнется на пустое поддерево; само включение осуществляется процедурой *MakeNewNode*. Далее, по мере того, как процедура осуществляет последовательные возвраты из рекурсивных вызовов, она использует



процедуры `ModifyLeft` или `ModifyRight` для выполнения поворотов и смены цветов, как это требуется в ситуациях, изображенных на рис. 11.12.

```

procedure ReInsertTree (var current: treepointer;
                        target: keytype; var status: outcome);
                        { Рекурсивное включение в дерево }
{ Pre: Указатель current является фактической связью (не копией)
      в красно-черном дереве; параметр target является ключом,
      не встречающимся в дереве.
Post: Создан и включен в дерево новый узел с ключом target;
      свойства красно-черного дерева восстановлены,
      за исключением, возможно, мест у корня current и одного
      из его дочерних узлов, состояние которого задается
      выходным параметром status.
Uses: Использует процедуру ReInsertTree рекурсивно, процедуры
      MakeNewNode, ModifyLeft, ModifyRight. }
begin
  if current = nil then
    MakeNewNode(current, target, status)
  else with current do
    if target < entry.key then
      begin
        ReInsertTree(left, target, status);
        ModifyLeft(current, status)
      end
    else if target > entry.key then
      begin
        ReInsertTree (right, target, status);
        ModifyRight(current, status)
      end
    else
      { target совпадает с ключом в текущем узле }
      Error('В дерево включен ключ-дубликат.')
  end;
                        { процедура ReInsertTree }

```

Процедура `ModifyLeft` обновляет переменную состояния и обнаруживает ситуации, показанные на рис. 11.12 и требующие поворотов или смены цветов. Именно в этой процедуре мы выполняем отложенные ранее действия по восстановлению красно-черных свойств. Когда активируется `ModifyLeft`, мы знаем, что включение было выполнено в левом поддереве текущего узла; мы знаем его цвет; и, по значению переменной состояния мы определяем состояние поддерева, в котором было сделано включение. Используя всю эту информацию, мы можем теперь определить, какие именно действия требуется (и требуются ли вообще) для восстановления красно-черных свойств.

```

procedure ModifyLeft (var current: treepointer;
                      var status: outcome);
                      { Левая модификация }
{ Pre: В левом поддереве узла current было выполнено включение,
      которое вернуло значение переменной состояния status
      для этого поддерева.
Post: Любое требуемое для дерева с корнем в current изменение
      цвета или поворот выполнено, и состояние обновлено.
Uses: Использует процедуры RotateRight, FlipColor, DRotate Right. }

```

```

begin
with current↑ do
case status of
    OK: ; { никаких действий не требуется, так как дерево уже ОК. }
    rednode: if red then
        status := leftred { и текущий узел current,
                           и левый "ребенок" красные }
    else
        status := OK; { current черный, а левый "ребенок"
                       красный, поэтому все ОК }
    leftred: if right = nil then { пустое поддерево ведет себя
                                  как черное }
        RotateRight(current, status)
    else if right↑.red then { оба дочерних узла красные }
        FlipColor(current, status)
    else { правое дочерний узел черный, левый красный }
        RotateRight(current, status);
    rightred: if right = nil then { пустое поддерево ведет себя как черное }
        DRotateRight(current, status)
    else if right↑.red then { оба дочерних узла красные }
        FlipColor(current, status)
    else DRotateRight(current, status); { правый дочерний узел
                                          черный, левый красный }
end
end; { процедура ModifyLeft }

```

Процедура `ModifyRight` аналогична; она обрабатывает ситуации, зеркально отображенные от тех, что показаны на рис. 11.12. Действия процедур поворота и смены цвета показаны на рис. 11.12, и их вполне можно оставить для упражнений. При разработке процедур поворота можно в качестве основы воспользоваться аналогичными процедурами для AVL-деревьев, однако для красно-черных деревьев важно правильно устанавливать цвета и индикатор состояния.

Единственная оставшаяся процедура — `MakeNewNode`, ее текст достаточно очевиден.

```

procedure MakeNewNode (var current: treepointer; { Создать новый узел }
                        target: keytype; var status: outcome);
{ Pre: Мишень target является ключом, который будет включен
  в новый узел.
  Post: Создан новый красный узел current, содержащий ключ target;
  переменная состояния status принимает значение rednode. }
begin
    new(current);
    with current↑ do begin { Включим target в новый узел по адресу current }
        entry.key := target;
        red := true;
        left := nil;
        right := nil
    end;
    status := rednode
end;
{ процедура MakeNewNode }

```

### Упражнения 11.4

- E1.** Включите ключи  $c, o, r, n, f, l, a, k, e, s$  в первоначально пустое красно-черное дерево.
- E2.** Включите ключи  $a, b, c, d, e, f, g, h, i, j, k$  в первоначально пустое красно-черное дерево.
- E3.** Найдите двоичное дерево поиска, для которого невозможно покрасить узлы, чтобы превратить его в красно-черное дерево.
- E4.** Найдите красно-черное дерево, которое не является AVL-деревом.
- E5.** Докажите, что у любого AVL-дерева можно покрасить его узлы, превратив его в красно-черное дерево. Вам будет проще доказать более сильное утверждение: AVL-дерево высоты  $h$  допускает покраску его узлов с преобразованием в красно-черное дерево, причем на каждом пути к пустому поддереву будет в точности  $\lceil h/2 \rceil$  черных узлов и, если  $h$  нечетно, тогда оба дочерних от корня узла будут черными.

### Программные проекты 11.4

- P1.** Завершите набор процедур красно-черного включения, написав отсутствующие процедуры `ModifyRight`, `FlipColor`, `RotateLeft`, `RotateRight`, `DRotateLeft` и `DRotateRight`. Удостоверьтесь в том, что в конце каждой процедуры цвета затронутых узлов установлены должным образом, а переменная состояния правильно отражает текущее состояние. Включив расширенную проверку на возникновение недопустимых ситуаций, вы можете упростить процесс отладки.
- P2.** Подставьте процедуру для красно-черного включения в управляемую меню демонстрационную программу для деревьев двоичного поиска из раздела 10.2 (проект P2), получив тем самым демонстрационную программу для черно-красных деревьев. Удаление можете оставить нереализованным.
- P3.** Подставьте процедуру для красно-черного включения в проект извлечения информации из раздела 10.2 (проект P6). Сравните производительность красно-черных деревьев с производительностью других деревьев поиска при различных комбинациях входных текстовых данных.

## Подсказки и ловушки

1. Деревья представляют собой гибкие и мощные структуры для задач моделирования и для организации данных. Решив использовать деревья для решения задач или разработки алгоритмов, прежде всего выберите требуемый вид дерева (упорядоченное, корневое, свободное или двоичное), и лишь после этого рассматривайте детали реализации.
2. Большинство деревьев можно легко описать с помощью рекурсии; рекурсивная формулировка соответствующих алгоритмов также часто оказывается наилучшим методом.
3. При решении задач извлечения информации, выбирая структуры данных, рассмотрите размер, число и расположение записей, а также тип

и структуру элементов. Для небольших записей или небольшого числа элементов будет использоваться внутренняя быстрая память, и деревья двоичного поиска скорее всего будут подходящим вариантом. При извлечении информации из дисковых файлов обычно лучшие результаты дают методы многовариантного ветвления с использованием трай-деревьев, В-деревьев и хеш-таблиц. Трай-деревья особенно подходят для приложений, в которых ключи структурируются как последовательности символов и где наборы ключей представляют собой относительно плотные поднаборы множества всех возможных ключей. Для других приложений часто более удобны методы, рассматривающие ключ как единый элемент. В-деревья, а также их различные обобщения и расширения могут быть с большой пользой приложены к решению задач, связанных с извлечением внешней информации.

## Обзорные вопросы

- 11.1
  1. Определите термины: **(а)** свободное дерево, **(б)** корневое дерево и **(с)** упорядоченное дерево.
  2. Изобразите графически различные **(а)** свободные деревья, **(б)** корневые деревья и **(с)** упорядоченные деревья.
  3. Назовите три способа описания соответствия между садами и двоичными деревьями, и укажите основное назначение каждого из этих способов.
- 11.2
  4. Что такое трай-дерево?
  5. Как трай-дерево с шестью уровнями и пятивариантным ветвлением в каждом узле может отличаться от корневого дерева с шестью уровнями и пятью дочерними узлами для каждого узла за исключением листьев? Какое из этих деревьев будет иметь меньшее число узлов и почему?
  6. Обсудите относительные преимущества в скорости извлечения трай-деревьев и деревьев двоичного поиска.
- 11.3
  7. Чем многовариантное дерево поиска отличается от трай-дерева?
  8. Что такое В-дерево?
  9. Что происходит при попытке включить новый элемент в заполненный узел В-дерева?
  10. Растет ли В-дерево со стороны своих листьев или со стороны своего корня? Почему?
  11. В каких случаях при удалении элемента из В-дерева приходится объединять его узлы?
  12. Где особенно удобно использовать В-деревья?
- 11.4
  13. В чем заключается взаимосвязь красно-черных деревьев и В-деревьев?
  14. Определите красное и черное условия.
  15. Как высота красно-черного дерева связана с его размером?

## Литература для дальнейшего изучения

Наиболее исчерпывающее исследование деревьев можно найти в серии книг Кнута [Knuth], (см. ссылку в главе 3). Соответствие упорядоченных деревьев двоичным деревьям описано в томе 1, стр. 332–347. В томе 3 на стр. 471–505 обсуждаются многовариантные деревья, В-деревья и трай-деревья. Первое изучение трай-деревьев было проведено в статье

Edward Fredkin, «Trie memory», *Communications of the ACM* 3 (1960), 490–499.

Первое упоминание о В-деревьях содержится в статье

R. Bayer and E. McCreight, «Organization and maintenance of large ordered indexes», *Acta Informatica* 1 (1972), 173–189.

Интересный обзор приложений и вариантов В-деревьев содержится в статье

D. Comer, «The ubiquitous B-tree», *Computing Surveys* 11 (1979), 121–137.

В следующей книге содержится (на стр. 242–264) описание В-деревьев с алгоритмами включения и удаления на языке Pascal.

N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, N.J., 1976, p. 170.

Имеется русский перевод: Вирт Н. Алгоритмы и структуры данных. СПб. [б.и.]: 2001. — 351 с.

Различные особенности красно-черных деревьев, включая алгоритм удаления, описаны в следующей книге, где рассмотрено большое количество разного рода алгоритмов.

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, M.I.T. Press, Cambridge, Mass., and McGraw Hill, New York, 1990, 1028 pages.

Имеется русский перевод:

Т. Х. Кормен, Ч. Э. Лейзерсон, Р. Л. Райвест. Введение в алгоритмы. — М.: МЦНМО, 2004.

Более обстоятельный математический анализ красно-черных деревьев можно найти в книге

Derick Wood, *Data Structures, Algorithms, and Performance*, Addison-Wesley, 1993, 594 pages.

В этой главе вводятся важные математические структуры, называемые графами и находящие свое применение в таких далеких друг от друга областях науки и техники, как социология, химия, география и электротехника. Мы изучим методы представления графов в доступных нам структурах данных и разработаем несколько важных алгоритмов для обработки графов. Наконец, мы рассмотрим возможности использования самих графов в качестве структур данных.

### 12.1. Математические основы

#### 12.1.1. Определения и примеры

графы и  
ориентированные  
графы

*Граф*  $G$  состоит из множества  $V$ , члены которого называются **вершинами** графа  $G$ , вместе с множеством  $E$  пар отдельных вершин, входящих в  $V$ . Эти пары носят название **ребер** графа  $G$ . Если  $e = (v, w)$  есть ребро с вершинами  $v$  и  $w$ , тогда про  $v$  и  $w$  говорят, что они *лежат на  $e$* , а про  $e$  говорят, что это ребро *инцидентно* вершинам  $v$  и  $w$ . Если пары неупорядочены, тогда  $G$  называют неориентированным графом; если пары упорядочены, тогда  $G$  называют ориентированным графом. Термин *ориентированный граф* (directed graph) часто сокращают до **орграф** или **диграф**, а неуточненный термин *граф* обычно означает *неориентированный граф*.

Естественным способом изображения графа является представление его вершин как точек или кружков, а ребер — как линейных сегментов или дуг, соединяющих вершины. Если граф ориентированный, тогда линейные сегменты или дуги имеют стрелки, указывающие направление. На рис. 12.1 приведено несколько примеров графов.

Населенные пункты в первой части рис. 12.1 являются вершинами графа, а воздушные пути, их соединяющие — ребрами. Во второй части рисунка атомы водорода и углерода (обозначенные буквами H и C) являются вершинами, а химические связи — ребрами. В третьей части рисунка показан ориентированный граф, в котором узлы сети (A, B, ... F) представляют собой вершины, а ребра от одной вершины к другой имеют направление, показанное стрелками.

Графы чрезвычайно важны как модельное представление многих видов процессов или структур. Графы образуются, например, городами и соединяющими их шоссе, а также компонентами печатной платы вместе

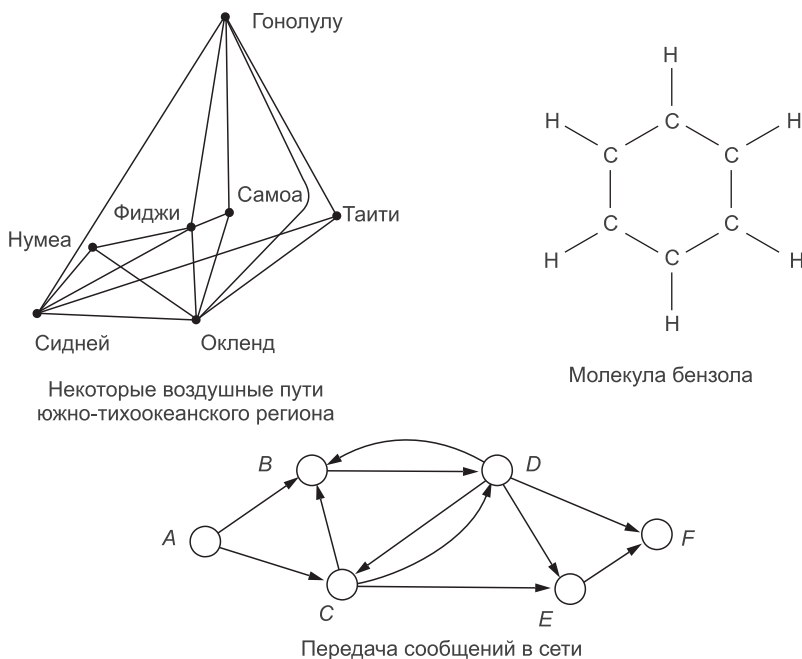


Рис. 12.1. Примеры графов

с межсоединениями. Органическое сложное вещество можно рассматривать как граф, в котором молекулы представляют собой вершины, а химические связи между ними — ребра. Люди, живущие в городе, могут рассматриваться как узлы графа, а их отношения «знаком с» ассоциироваться с ребрами. Люди, работающие в некоторой корпорации, образуют ориентированный граф, в котором отношения «руководит» связываются с ребрами. Тех же самых людей можно рассматривать и как неориентированный граф, в котором различные ребра описывают отношения «работает с».

### 12.1.2. Неориентированные графы

Несколько видов неориентированных графов представлено на рис. 12.2. Две вершины в неориентированном графе называются **смежными**, если между первой и второй имеется ребро. Таким образом, в неориентированном графе рис. 12.2, (а) вершины 1 и 2 смежные, как и вершины 3 и 4, но вершины 1 и 4 не являются смежными. Последовательность различных вершин, каждая из которых является смежной по отношению к другой, называется **маршрутом** или **цепью**. Такой маршрут показан на 12.2, (b). **Циклом** называют цепь, содержащую по меньшей мере три вершины, в которой последняя вершина цепи смежна с первой. Цикл показан на 12.2, (c). Граф называют **связным**, если каждая пара его вершин соединена маршрутом; на 12.2, (a), (b) и (c) представлены связные графы, а на 12.2, (d) — несвязный граф.

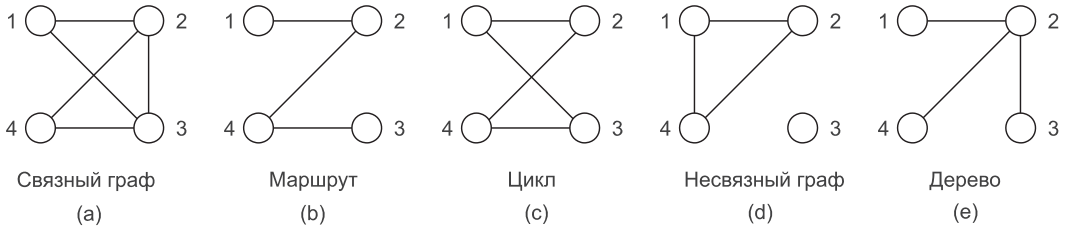


Рис. 12.2. Различные виды неориентированных графов

На рис. 12.2, (e) изображен связный граф без циклов. Вы можете заметить, что такой граф фактически представляет собой дерево, и мы используем это свойство в качестве определения: *свободное дерево* определяется как связный неориентированный граф без циклов.

12.1.3. Ориентированные графы

Мы можем дать схожие определения для ориентированных графов. Мы можем потребовать, чтобы все ребра в маршруте или цикле имели одно направление, так что продвижение по маршруту или циклу означает движение в направлении, указываемом стрелками. Такой маршрут (цикл) называется *направленным* маршрутом (циклом). Ориентированный граф называют *сильносвязным*, если от любой вершины к любой другой вершине имеется направленный маршрут. Если при удалении направления ребер получившийся неориентированный граф оказался связным, то исходный ориентированный граф называется *слабосвязным*. На рис. 12.3 показаны примеры направленного цикла, сильносвязного ориентированного графа и слабосвязного ориентированного графа.

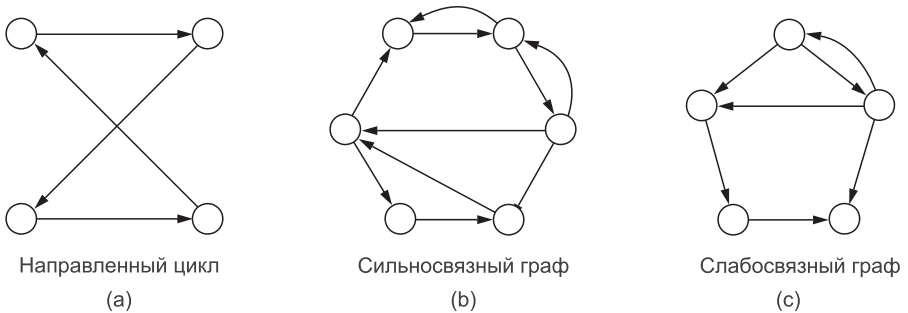


Рис. 12.3. Примеры ориентированных графов

Ориентированные графы на рис. 12.3, (b) и (c) имеют пары вершин с направленными ребрами, соединяющими эти вершины в обоих направлениях. Поскольку направленные ребра являются упорядоченными парами, а упорядоченные пары  $(v, w)$  и  $(w, v)$  различаются, если  $v \neq w$ , такие пары ребер допустимы в ориентированных графах. Поскольку, однако, соответствующие неупорядоченные пары неразличимы, в неориентирован-



ном графе может быть максимум одно ребро, соединяющее пару вершин. Аналогично, поскольку вершины ребра должны быть различимы, не может быть ребра, соединяющего вершину с той же самой вершиной. Следует, однако, заметить, что (хотя мы этого делать не будем) иногда эти требования ослабляют, и допускают наличие множества ребер, соединяющих пару вершин, а также замкнутые петли, соединяющие вершину с той же самой вершиной.

замкнутые петли

## 12.2. Компьютерное представление

Если мы пишем программы для решения задач, связанных с графами, мы прежде всего должны найти способ представить математическую структуру графа в виде некоторой структуры данных. Для этого широко используется целый ряд методов, принципиально различающиеся выбором абстрактного типа данных, представляющего графы; кроме этого, реализация каждого абстрактного типа данных может иметь различные варианты. Другими словами, мы начинаем с одной математической системы (*граф*), затем изучаем, как ее можно описать с помощью абстрактных типов данных (с этой целью могут быть использованы, например, *множества, таблицы и списки*), и, наконец, выбираем реализацию абстрактного типа данных, на котором мы остановились.

### 1. Представление в виде множества

Графы описываются с помощью понятия множества, и естественно сначала обратиться к множествам и рассмотреть возможности их представления в качестве данных. Во-первых, мы имеем множество вершин и, во-вторых — ребра, определяемые как множество пар вершин. Вместо того, чтобы пытаться представить это множество пар непосредственно, мы разделяем его на части, рассматривая множество ребер, подсоединенных к каждой вершине по отдельности. Другими словами, мы можем проследить все ребра в графе, сохраняя для всех вершин  $v$  в графе множество  $E_v$  ребер, содержащих  $v$ , или, эквивалентно, множество  $A_v$  всех вершин, смежных с  $v$ . Фактически мы можем воспользоваться этой идеей для образования нового эквивалентного определения графа:

Определение

**Граф**  $G$  состоит из множества  $V$ , называемого *вершинами* графа  $G$ , а также, для всех  $v \in V$ , подмножества  $A_v$  множества  $V$ , называемого вершинами, *смежными* с  $v$ .

Из подмножества  $A_v$  мы можем реконструировать ребра как упорядоченные пары согласно правилу: пара  $(v, w)$  является ребром только и если только  $w \in A_v$ . Однако проще работать с множеством вершин, чем с парами. Более того, это новое определение одинаково применимо и к ориентированным, и к неориентированным графам. Если граф не ориентирован, это означает, что он удовлетворяет следующему свойству симметрии:  $w \in A_v$  подразумевает, что  $v \in A_w$  для всех  $v, w \in V$ . Это свойство может быть сформулировано менее формально: ненаправленное ребро между  $v$  и  $w$  может рассматриваться как составленное из двух направленных ребер, одного от  $v$  к  $w$ , а другого от  $w$  к  $v$ .

## 2. Реализация множеств

У нас есть два общих способа реализации множества вершин с помощью структур данных и алгоритмов. Один способ заключается в представлении множества как *списка* его элементов; этот способ мы вскоре рассмотрим. Другая реализация, часто называемая *битовой строкой*, хранит булево значение (т. е. один бит) для каждого возможного члена множества, чтобы показать, имеется ли он реально в множестве или нет. Последний метод используется в конструкторе типа **set** языка Pascal, который мы вскоре рассмотрим. Для реализации множеств языка Pascal, мы должны начать с порядкового типа, значения которого соответствуют возможным вершинам. Для простоты мы будем считать, что все эти вершины индексируются с помощью целых чисел от 1 до size, где size обозначает число вершин. Поскольку желательно, чтобы величина size была переменной, мы также введем константу maxvertex, ограничивающую число вершин, после чего мы можем полностью специфицировать первое представление графа:

множества как  
битовые строки

первая реализация  
множества

```

type
  vertex = 1..maxvertex;           { идентифицируем вершины их индексами }
  counter = 0..maxvertex;          { счетчик вершин }
  adjacencysset = set of vertex;
  graph = record
    size: counter;                  { число вершин в графе }
    A: array[vertex] of adjacencysset
end;
```

В этой реализации массив A [v] есть множество всех вершин, смежных с вершиной v.

## 3. Таблицы смежности

Имеется одна практическая проблема, связанная с рассматриваемой ниже реализацией: конструктор типа **set** доступен не во всех языках программирования, а в тех случаях, когда он имеется, на максимальный размер множества часто накладываются такие ограничения, что делают этот тип практически бесполезным. Мы, однако, можем преодолеть это затруднение и в то же время получить лучшее представление графов.

Мы уже видели, что тип языка Pascal **set of vertex** может быть реализован как **array [vertex] of Boolean**, где каждый элемент массива показывает, является ли соответствующая вершина членом множества. Если мы заменим этим массивом множество вершин adjacencysset, мы обнаружим, что массив A в объявлении типа graph может быть заменен массивом массивов, т. е. двумерным массивом следующим образом:

ограничения

множество  
массивов

вторая  
реализация:  
таблица  
смежности

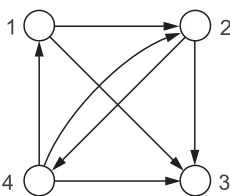
```

type
  vertex = 1..maxvertex;           { идентифицируем вершины их индексами }
  counter = 0..maxvertex;          { счетчик вершин }
  adjacencytable = array [vertex, vertex] of Boolean;
  graph = record
    size: counter;                  { число вершин в графе }
    A: array [vertex] of array [vertex] of Boolean;
end;
```

интерпретация

Таблица смежности  $A$  имеет естественную интерпретацию: элемент  $A[v, w]$  истинен, если и только если вершина  $v$  смежна с вершиной  $w$ . Если граф ориентирован, мы интерпретируем  $A[v, w]$  как индикатор наличия в графе ребра от  $v$  к  $w$ . Если граф не ориентирован, тогда таблица смежности симметрична, т. е.  $A[v, w] = A[w, v]$  для всех  $v$  и  $w$ . Представление графа с помощью множества смежности и таблицы смежности приведено на рис. 12.4.

Ориентированный граф



| Множества смежности |             |
|---------------------|-------------|
| вершина             | Множество   |
| 1                   | {2, 3}      |
| 2                   | {3, 4}      |
| 3                   | $\emptyset$ |
| 4                   | {1, 2, 3}   |

| Таблица смежности |         |
|-------------------|---------|
|                   | 1 2 3 4 |
| 1                 | F T T F |
| 2                 | F F T F |
| 3                 | F F F F |
| 4                 | T T T F |

Рис. 12.4. Множество смежности и таблица смежности

4. Списки смежности

Другим способом представления множества является *список* его элементов. Для представления графа мы должны иметь два списка: список вершин и, для каждой вершины, список смежных вершин. Мы рассмотрим реализацию графов с использованием как непрерывных списков, так и простых связных списков. Для более сложных приложений, однако, часто оказывается полезным более изощренные реализации списков в виде двоичных или многовариантных деревьев или в виде пирамид.

Заметьте, что различая вершины по их индексам в предыдущих реализациях, мы фактически реализовывали множество вершин в виде непрерывного списка, но теперь мы можем сделать осознанный выбор в пользу непрерывных или связных списков.

5. Связная реализация

При использовании связных списков и для вершин, и для списков смежности, достигается максимальная гибкость. Такая реализация проиллюстрирована на рис. 12.5 (a); ее объявление представлено ниже.

третья  
реализация:  
связные списки

```
type
  pointvertex = ↑vertex;
  pointedge = ↑edge;
  vertex = record
    firstedge: pointedge;
    nextvertex: pointvertex
  end;
  edge = record
    endpoint: pointvertex;
    nextedge: pointedge;
  end;
  graph = pointvertex;
```

{ начало списка связности }

{ следующая вершина в связном списке }

{ вершина, на которую указывает ребро }

{ следующее ребро в списке связности }

{ заголовок списка вершин }

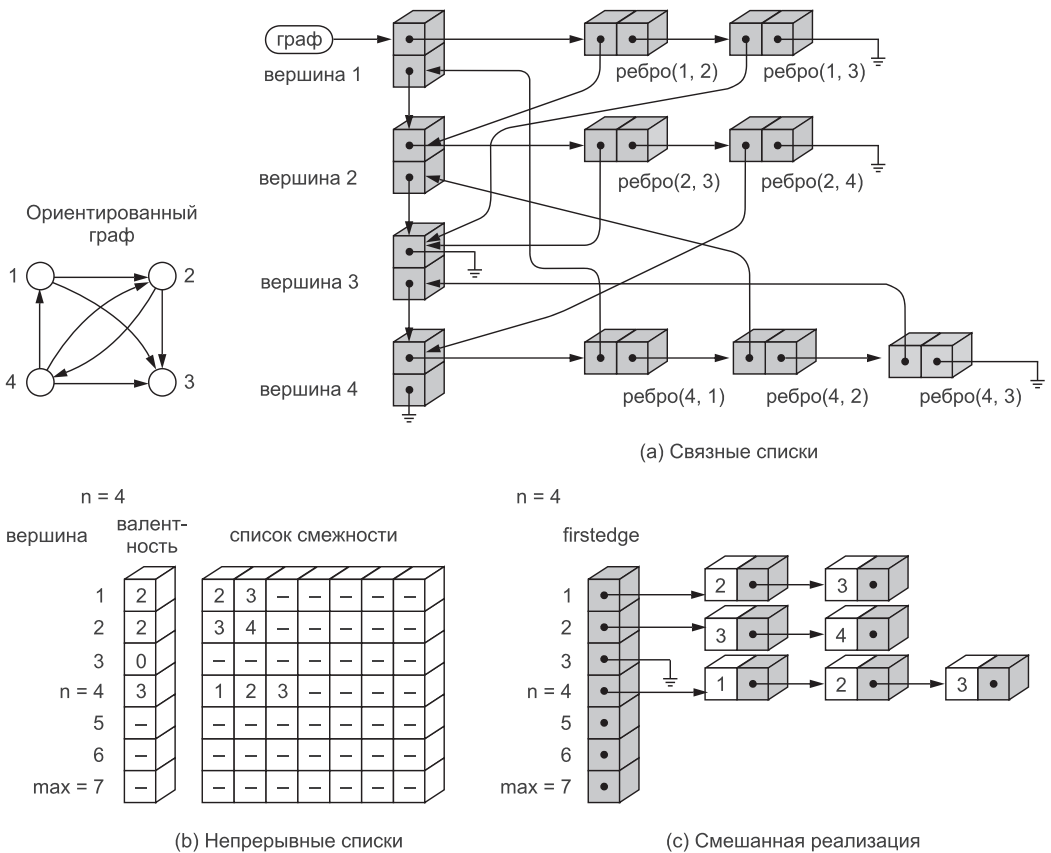


Рис. 12.5. Реализации графа списками

## 6. Непрерывная реализация

Хотя эта связная реализация отличается высокой гибкостью, иногда поиск в связных списках оказывается весьма утомительным; в то же время, многие алгоритмы требуют произвольного доступа к вершинам. Потому предлагаемая ниже непрерывная реализация часто дает лучшие результаты. Для непрерывного списка смежности мы должны хранить счетчик, и для него мы используем стандартное обозначение из теории графов: **валентность** вершины определяется как число ребер, на которых она лежит, или, другими словами, это число смежных с ней вершин. Такая непрерывная реализация представлена на рис. 12.5 (б).

четвертая  
реализация:  
непрерывные  
списки

**type**

```
vertex = 1..maxvertex;      { идентифицируем вершины по их индексам }
counter = 0..maxvertex;      { счетчик вершин }
adjacencylist = array[vertex] of vertex;
graph = record
  size: counter;
  valence: array[vertex] of counter;
  A: array[vertex] of adjacencylist
end;
```

{ число вершин в графе }

## 7. Смешанная реализация

Последняя реализация использует непрерывный список для вершин и связную память для списков смежности. Такая смешанная реализация показан на рис. 12.5 (с).

пятая реализация:  
смешанные списки

```

type
  vertex = 1..maxvertex;      { идентифицируем вершины по их индексам }
  counter = 0..maxvertex;      { счетчик вершин }
  pointedge = ↑edge;
  edge = record
    endpoint: vertex;
    nextedge: pointedge
  end;
  graph = record
    size: counter;              { число вершин в графе }
    firstedge: array [vertex] of pointedge
  end;
```

## 8. Информационные поля

Многие приложения графов требуют не только информации о связности, задаваемой в различных реализациях, но также и дополнительной информации, касающейся каждой вершины или каждого ребра. В связных реализациях такая информация может быть включена в виде дополнительных полей внутри соответствующих записей, а в непрерывных реализациях ее можно включить посредством преобразования элементов массива в записи.

сети, веса

Особо важным примером является организация *сети*, которая определяется как граф, в котором каждому ребру соответствует числовой *вес*. Для многих сетевых алгоритмов наилучшей реализацией является таблица смежности, в которой в качестве элементов используются не булевы значения, а веса. Позже в этой главе мы еще вернемся к этому вопросу.

## 12.3. Просмотр графа

### 12.3.1. Методы

При решении многих задач нам требуется исследовать все вершины графа в том или ином систематическом порядке, как было и в случае двоичных деревьев, для которых мы разработали несколько методов систематического просмотра. При просмотре дерева у нас есть корневой узел, от которого мы обычно и начинали просмотр; в графе часто нет вершины, выделенной среди остальных, и поэтому просмотр<sup>1</sup> может начаться с любой произвольно взятой вершины. Хотя имеется много возможных способов посещения вершин графа, два метода имеют особое значение.

<sup>1</sup> Применительно к графам обычно используется термин *обход графа*, но мы ради преемственности будем использовать прежний термин. — *Прим. перев.*

просмотр  
в глубину

*Метод просмотра графа в глубину* имеет грубую аналогию с прямым просмотром упорядоченного дерева. Предположим, что по ходу просмотра только что была посещена вершина  $v$ , и пусть  $w_1, w_2, \dots, w_k$  есть вершины, смежные с  $v$ . Тогда на следующем шаге мы посетим  $w_1$ , а  $w_2, \dots, w_k$  будут ждать. После посещения  $w_1$  мы просматриваем все вершины, смежные с  $w_1$ , и только после этого возвращаемся к просмотру  $w_2, \dots, w_k$ .

просмотр  
в ширину

*Метод просмотра графа в ширину* имеет грубую аналогию с поуровневым просмотром упорядоченного дерева. Если по ходу просмотра только что была посещена вершина  $v$ , тогда следующим шагом будет посещение *всех* вершин, смежных с  $v$ , а вершины, смежные с этими, будут помещены в список ожидающих и просмотрены после просмотра всех вершин, смежных с  $v$ . На рис. 12.6. показан порядок посещения вершин графа обоими методами.

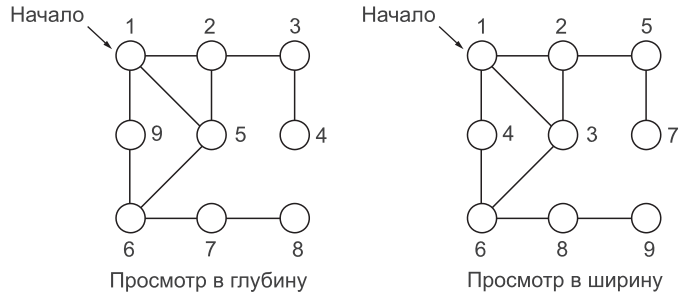


Рис. 12.6. Просмотр графа

### 12.3.2. Алгоритм просмотра в глубину

Просмотр в глубину естественным образом формулируется в виде рекурсивного алгоритма. Действие этого алгоритма при достижении вершины  $v$  выглядят следующим образом:

```
Visit(v);
for все вершины w, смежные to v do
    Traverse(w);
```

осложнения

Однако при просмотре графа возникают два затруднения, которых не могло быть при просмотре дерева. Во-первых, граф может содержать циклы, поэтому алгоритм просмотра может достичь той же вершины второй раз. Для предотвращения бесконечной рекурсии мы должны предусмотреть булев массив `visited`, устанавливая `visited[v]` в значение `true` непосредственно перед посещением  $v$ , и проверяя значение `visited[w]` перед обработкой  $w$ . Во-вторых, граф может быть несвязным, и в этом случае алгоритм просмотра, начав свою работу с некоторой фиксированной стартовой точки, может и не достигнуть всех вершин. Поэтому мы помещаем действие алгоритма в цикл, который проходит по всем вершинам графа.

При этих уточнениях мы получаем следующий эскиз просмотра в глубину. Дальнейшие детали зависят от выбора реализации графа, и мы отложим их рассмотрение до разработки прикладных программ.

главная процедура

```
procedure DepthFirst (var G: graph; procedure Visit(var v: vertex));
    { Просмотр в глубину }
{ Pre:  Граф G уже создан.
Post:  Процедура Visit была выполнена над каждой вершиной графа G
        в порядке просмотра в глубину.
Uses:  Использует процедуру Traverse, которая выполняет
        рекурсивный просмотр в глубину. }
var
    visited: array [vertex] of Boolean;
    v: vertex;
begin                                     { процедура DepthFirst }
    for всех v in G do
        visited [v] := false;
        for всех v in G do
            if not visited [v] then
                Traverse(v)
end;                                     { процедура DepthFirst }
```

Рекурсия выполняется в следующей процедуре, которая должна быть объявлена внутри предыдущей.

рекурсивный  
просмотр

```
procedure Traverse (var v: vertex; procedure Visit(var v: vertex));
    { Просмотр }
{ Pre:  v является вершиной графа G.
Post:  Просмотр в глубину посредством процедуры Visit был завершен
        для v и всех вершин, смежных с v.
Uses:  Использует процедуру Traverse рекурсивно. }
var w: vertex;
begin                                     { процедура Traverse }
    visited [v] := true;
    Visit(v);
    for всех w смежных to v do
        if not visited [w] then
            Traverse(w)
end;                                     { процедура Traverse }
```

### 12.3.3. Алгоритм просмотра в ширину

Поскольку использование рекурсии и программирование с участием стека в принципе эквивалентны, мы можем сформулировать алгоритм просмотра в глубину с помощью стека, когда все непосещенные вершины, смежные с посещаемой в настоящий момент, проталкиваются в стек, а при выталкивании из стека представляют следующую вершину для очередного посещения. Алгоритм просмотра в ширину практически совпадает с алгоритмом просмотра в глубину, за исключением того, что вместо стека требуется использование очереди. Эскиз этого алгоритма приведен ниже.

стеки и очереди

просмотр  
в ширину

```

procedure BreadthFirst (var G: graph; procedure Visit(var v: vertex));
                                     { Просмотр в ширину }
{ Pre:  Граф G уже создан.
  Post: Процедура Visit выполнена для каждой вершины графа G,
        причем вершины выбирались в порядке просмотра в ширину.
  Uses: Использует пакет операций с очередью. }
type
  queueentry = vertex;
var
  Q: queueentry;
  visited: array [vertex] of Boolean;
  v,
  w: vertex;
begin                                     { процедура BreadthFirst }
  for всеx v in G do
    visited [v] := false;
    CreateQueue(Q);
    for всеx v in G do
      if not visited [v] then
        begin
          Append(v, Q);
          repeat
            Serve(v, Q);
            if not visited [v] then
              begin
                visited [v] := true;
                Visit(v)
              end;
            for всеx w смежных to v do
              if not visited [w] then
                Append(w, Q)
            until QueueEmpty(Q)
          end
        end;
      end
    end;                                     { процедура BreadthFirst }

```

## 12.4. Топологическая сортировка

### 12.4.1. Постановка задачи

топологический  
порядок

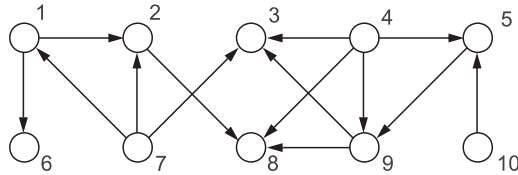
Если  $G$  является ориентированным графом без направленных циклов, тогда **топологическим порядком** графа  $G$  называется такой последовательный список всех вершин  $G$ , что для всех вершин  $v, w \in G$ , если имеется ребро от  $v$  к  $w$ , тогда  $v$  предшествует  $w$  в последовательном списке. Повсюду в этом разделе мы будем рассматривать только ориентированные графы, не имеющие направленных циклов. Для обозначения графа без циклов часто используется термин **ациклический**.

приложения

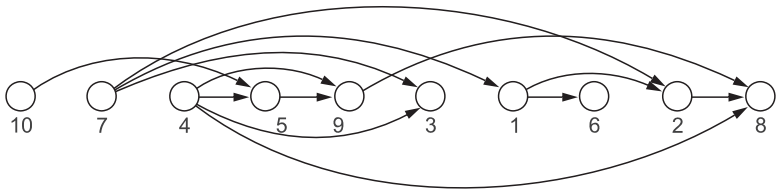
Такие графы приводят к ряду проблем. В качестве первого приложения топологического порядка будем считать, что университетские учебные курсы представляют собой вершины ориентированного графа, в котором протянуты ребра от одного курса к другому, если первый курс являет-



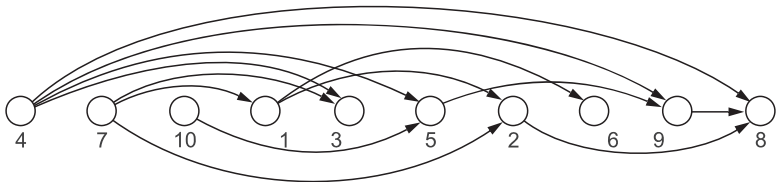
ся необходимой предпосылкой для второго. В этом случае топологический порядок представляет собой список всех курсов, в котором все предпосылочные курсы указаны перед тем курсом, для которого они необходимы. В качестве второго примера можно взять словарь технических терминов, упорядоченный так, что никакой термин не используется в определениях, если он сам еще не был определен. Два топологических порядка ориентированного графа показаны на рис. 12.7.



Ориентированный граф без направленных циклов;



Упорядочение в глубину



Упорядочение в ширину

**Рис. 12.7.** Топологический порядок ориентированного графа

В качестве примера алгоритмов для просмотра графа мы разработаем процедуры, которые выполняют топологическое упорядочение вершин ориентированного графа, не имеющего циклов. Мы разработаем две процедуры: первую для просмотра в глубину, а вторую — для просмотра в ширину. Обе процедуры будут оперировать с графом  $G$ , заданным в смешанной реализации (с непрерывным списком вершин и связными списками смежности), и обе процедуры образуют на выходе массив типа

`toporder = array [vertex] of vertex;`

определяющий порядок, в котором следует перечислить вершины, чтобы получить топологический порядок просмотра.

## 12.4.2. Алгоритм упорядочения в глубину

метод

При топологическом порядке каждая вершина должна появиться перед всеми вершинами, которые следуют за ней в ориентированном графе. В случае топологического упорядочения в глубину мы начинаем с поиска вершины, у которой нет последующих, и помещаем ее в список последней по порядку. После того, как мы посредством рекурсии поместили все вершины, следующие за некоторой вершиной, в топологическом порядке, мы можем поместить саму вершину в позицию перед всеми следующими за ней. Переменная *place* указывает позицию в топологическом порядке, куда будет помещена следующая упорядочиваемая вершина. Поскольку мы начинаем с последних вершин, в начале процесса переменная *place* принимает значение, равное числу вершин графа. Главная процедура является непосредственной реализацией общего алгоритма, разработанного в предыдущем разделе.

```

procedure DepthTopSort (var G: graph; var T: toporder);
    { Топологическое упорядочение в глубину }
{ Pre:   G является направленным графом без циклов, реализованным
        с непрерывным списком вершин и связными списками
        смежности.
Post:   Процедура выполняет просмотр графа G в глубину и
        генерирует результирующий топологический порядок
        в массиве T.
Uses:   Использует: процедура RecDepthSort выполняет рекурсивный
        просмотр в глубину. }

var
    visited: array [vertex] of Boolean;           { проверяет, что G
                                                    не содержит циклов }
    v: vertex;                                     { очередная вершина, следующие за которой
                                                    должны быть упорядочены }
    place: counter;                               { следующая заполняемая позиция
                                                    в топологическом порядке }
                                                    { процедура DepthTopSort }
begin
    for v := 1 to G.size do
        visited [v] := false;
        place := G.size;
        for v := 1 to G.size do
            if not visited [v] then
                RecDepthSort(v, place, T);
    end;                                           { процедура DepthTopSort }

```

Процедура RecDepthSort, выполняющая рекурсию и основанная на эскизе общей процедуры Traverse, сначала размещает все вершины, следующие за *v*, в их позиции в топологическом порядке, а затем помещает в свою позицию вершину *v*.

```

procedure RecDepthSort (v: vertex; var place: counter; var T: toporder);
    { Рекурсивное упорядочение в глубину }
{ Pre:   Параметр v является вершиной графа G, place является
        следующим местом в топологическом порядке T, которое
        следует определить (начиная с конца окончательно
        упорядоченного списка).

```

**Post:** Процедура размещает все вершины, следующие за  $v$ , и, наконец, саму  $v$ , в топологическом порядке  $T$ , упорядочивая в глубину.

**Uses:** Использует: глобальный массив `visited` и глобальный граф  $G$ ; процедуру `RecDepthSort` рекурсивно. }

```

var
  curvertex: vertex;           { вершина, смежная с v }
  curedge: pointedge;         { просматривает список вершин, смежных с v }
begin
  visited[v] := true;
  curedge := G.firstedge[v];   { найдем первую вершину, следующую за v }
  while curedge nil do
    begin
      curvertex := curedge↑.endpoint; { curvertex есть вершина, смежная с v }
      if not visited[curvertex] then
        RecDepthSort(curvertex, place, T);
        { упорядочим вершины, следующие за curvertex }
      curedge := curedge↑.nextedge
        { перейти к следующей
          непосредственно за вершиной v }
    end;
    T[place] := v;              { разместим саму v в топологическом порядке }
    place := place - 1
  end;                          { процедура RecDepthSort }

```

производитель-  
ность

Поскольку этот алгоритм посещает все узлы графа в точности по одному разу и проходит по каждому ребру тоже один раз, не выполняя при этом поиск, время выполнения процедуры составляет  $O(n + e)$ , где  $n$  есть число вершин, а  $e$  есть число ребер графа.

### 12.4.3. Алгоритм упорядочения в ширину

метод

При топологическом упорядочении ориентированного графа без циклов методом просмотра в ширину, мы начинаем с поиска вершины, которая должна быть первой в топологическом порядке, а затем используем тот факт, что каждая вершина должна быть помещена перед всеми идущими за ней в топологическом порядке вершинами. Вершины, помещаемые раньше, это те, которые не расположены после каких-либо других вершин. Для того, чтобы найти такие вершины, мы предусматриваем массив `predecessorcount`, в котором элемент с индексом  $v$  содержит число непосредственных предшественников вершины  $v$ . Вершины, которые не следуют за кем-либо, не имеют предшественников. Мы, таким образом, инициализируем просмотр в ширину, поместив эти вершины в очередь вершин, подлежащих посещению. По мере того, как посещается каждая вершина, она извлекается из очереди, помещается в следующую доступную позицию в топологическом порядке (от начала списка), а затем изымается из дальнейшего рассмотрения посредством уменьшения на единицу отсчета числа предшественников для каждой из непосредственно следующих за ней вершин. Когда один из этих отсчетов достигнет нуля, все предшественники соответствующей вершины были посещены, и саму эту вершину теперь можно обработать, поэтому она добавляется в очередь. В результате получается следующая процедура.

```

procedure BreadthTopSort (var G: graph; var T: toporder);
    { Топологическое упорядочение в ширину }
{ Pre:  G является направленным графом без циклов, реализованным
      с непрерывным списком вершин и связными списками
      смежности.
Post: Процедура выполняет просмотр графа G в ширину
      и генерирует результирующий топологический порядок
      в массиве T.
Uses: Использует пакет для обработки очередей. }
var
  predecessorcount: array [vertex] of integer; { число непосредственных
                                                    предшественников каждой вершины }
  Q: queue; { вершины, готовые к размещению в порядке }
  v, { вершина, посещаемая в настоящий момент }
  succ: vertex; { одна из непосредственно следующих за v вершин }
  curedge: pointedge; { просматривает список смежности для v }
  place: integer; { следующая позиция в топологическом порядке }
begin { процедура BreadthTopSort }
  for v := 1 to G.size do { инициализируем все счетчики
                            предшественников нулем }
    predecessorcount[v] := 0;
    for v := 1 to G.size do
      begin { увеличим отсчет предшественников
            для последующих вершин }
        curedge := G.firstedge[v];
        while curedge <> nil do begin
          predecessorcount[curedge↑.endpoint] :=
            predecessorcount[curedge↑.endpoint] + 1;
          curedge := curedge↑.nextedge
        end
      end;
    CreateQueue(Q);
    for v := 1 to G.size do { поместим вершины, не имеющие
                              предшественников, в очередь }
      if predecessorcount[v] = 0 then
        Append(v, Q);
      place := 0; { начнем просмотр в ширину }
      while not QueueEmpty(Q) do begin
        Serve(v, Q); { посетим вершину v, разместив ее
                      в топологическом порядке }
        place := place + 1;
        T[place] := v;
        curedge := G.firstedge[v]; { просмотр следующих за v вершин }
        while curedge <> nil do begin { уменьшим отсчеты предшественников
                                         для каждой следующей вершины }
          succ := curedge↑.endpoint;
          predecessorcount[succ] := predecessorcount[succ] - 1;
          if predecessorcount[succ] = 0 then
            { вершина succ не имеет следующих за ней, поэтому она готова
              к обработке }
            Append(succ, Q);
          curedge := curedge↑.nextedge
        end
      end
    end;
  { процедура BreadthTopSort }

```

производитель-  
ность

Этот алгоритм требует наличия одного из пакетов обработки очередей. Элементы в очереди являются вершинами, и очередь может быть реализована любым из способов, описанных в главе 5.

Как и в случае просмотра в глубину, время, требуемое для процедуры просмотра в ширину, составляет  $O(n + e)$ , где  $n$  — число вершин, а  $e$  — число ребер в ориентированном графе.

## 12.5. Алгоритм экономного продвижения: кратчайшие маршруты

### 1. Постановка задачи

кратчайший  
маршрут

В качестве заключительного примера приложения графов, примера, требующего несколько более изощренных рассуждений, мы рассмотрим следующую задачу. Нам дан ориентированный граф  $G$ , в котором каждое ребро имеет связанный с ним неотрицательный *вес*, и нашей задачей является нахождение такого маршрута от одной вершины  $v$  к другой  $w$ , для которого сумма весов по ходу маршрута принимает минимальное значение. Мы называем такой маршрут **кратчайшим маршрутом**, несмотря на то, что веса могут представлять собой стоимость, время или другую величину, отличную от расстояния.

Мы можем представить себе граф  $G$ , например, как карту авиационных маршрутов, а вес каждого ребра как стоимость полета из одного города в другой. В этом случае наша задача заключается в поиске такого маршрута из города  $v$  в город  $w$ , для которого суммарная стоимость перелета была бы минимальна. Рассмотрим ориентированный граф, изображенный на рис. 12.8. Кратчайший маршрут из вершины 1 в вершину 2 проходит через вершину 3 и имеет суммарную стоимость 4 в сравнении со стоимостью 5 для ребра, непосредственно соединяющего вершины 1 и 2 и стоимостью 8 для маршрута через вершину 5.

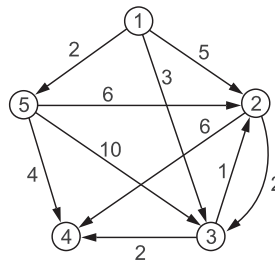


Рис. 12.8. Ориентированный граф с весами

источник

Оказывается, столь же легко решить более общую задачу поиска кратчайших маршрутов от произвольной вершины, называемой **источником**, до всех остальных вершин, как и частную задачу для одной целевой вершины. Для простоты выберем в качестве источника вершину 1; тогда наша задача состоит в нахождении кратчайшего маршрута от вершины 1 к каждой вершине графа. Будем считать, что все веса неотрицательны.

## 2. Метод

Алгоритм предусматривает хранение множества  $S$  тех вершин, для которых уже известно кратчайшее расстояние от 1. Первоначально единственной вершиной в  $S$  является 1. На каждом шаге мы добавляем к  $S$  оставшуюся вершину, для которой определен кратчайший маршрут от 1. Задача заключается в определении, какую вершину добавлять к  $S$  на каждом шаге. Представим себе, что вершины, уже находящиеся в  $S$ , помечены каким-либо цветом, и так же покрашены ребра, образующие кратчайший маршрут от источника 1 к этим вершинам.

Предусмотрим таблицу  $D$ , в которой хранится для каждой вершины  $v$  расстояние от 1 до  $v$  по маршруту, в котором все ребра покрашены, за исключением, возможно, последнего ребра. Другими словами, если  $v$  содержится в  $S$ , тогда  $D[v]$  содержит кратчайшее расстояние до  $v$ , и все ребра вдоль соответствующего маршрута покрашены. Если  $v$  не содержится в  $S$ , тогда  $D[v]$  содержит длину маршрута от 1 к какой-то другой вершине  $w$  в  $S$  плюс вес ребра от  $w$  к  $v$ , и все ребра этого маршрута, за исключением последнего, покрашены. Таблица  $D$  инициализируется путем помещения в  $D[v]$  веса ребра от 1 к  $v$ , если такое ребро существует, и бесконечности в противном случае.

Для определения того, какую вершину добавлять к  $S$  на каждом шаге, мы для выбора вершины  $v$  с минимальным расстоянием, записанным в таблице  $D$ , и еще не входящим в  $S$ , используем критерий *экономного продвижения*.

Мы должны доказать, что для этой вершины  $v$  расстояние, записанное в  $D$ , в действительности есть кратчайший маршрут от 1 к  $v$ . Предположим, что имеется более короткий маршрут от 1 к  $v$ , как это показано на рис. 12.9. Идя по этому маршруту, мы сначала покидаем  $S$ , чтобы прийти к некоторой вершине  $x$ , затем продолжаем продвигаться к  $v$  (возможно даже снова входя в  $S$  по дороге к  $v$ ). Но если этот маршрут короче покрашенного маршрута к  $v$ , тогда его начальный сегмент от 1 до  $x$  также должен быть короче, и критерий экономного продвижения выбрал бы  $x$ , а не  $v$  в качестве следующей вершины, добавляемой к  $S$ , поскольку мы имели бы  $D[x] < D[v]$ .

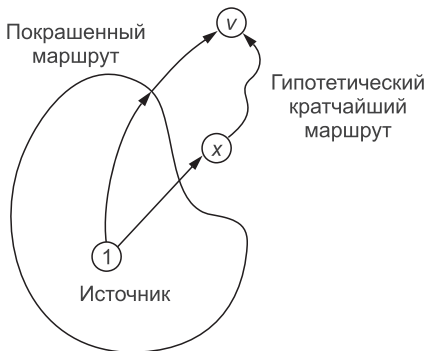


Рис. 12.9. Определение кратчайшего пути

таблица  
расстояний

алгоритм  
экономного  
продвижения

доказательство

конец  
доказательства

сохранение  
инварианта

Когда мы добавляем  $v$  к  $S$ , мы рассматриваем  $v$  как уже покрашенную вершину, и так же красим кратчайший маршрут от 1 к  $v$  (каждое ребро которого, за исключением последнего, фактически уже покрашено). Далее, мы должны обновить элементы в  $D$  посредством проверки для каждой вершины  $w$ , не входящей в  $S$ , действительно ли маршрут через  $v$  и затем непосредственно к  $w$  короче, чем предварительно записанное расстояние до  $w$ . Другими словами, мы заменяем  $D[w]$  на  $D[v]$  плюс вес ребра от  $v$  к  $w$ , если последняя величина оказалась меньше.

### 3. Пример

Перед тем, как приступить к написанию формальной процедуры, использующей этот метод, рассмотрим пример, изображенный на рис. 12.10.

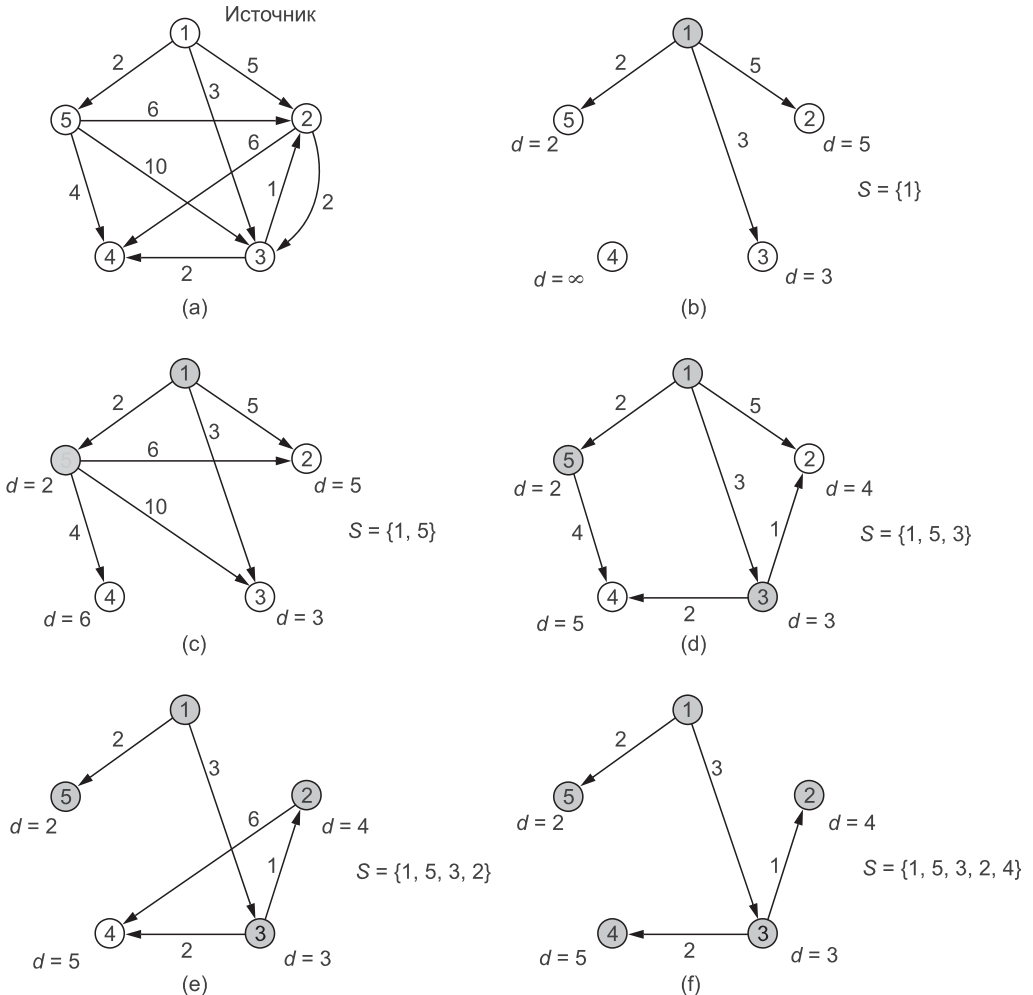


Рис. 12.10. Пример кратчайших путей

представление  
графа

Приступая к написанию процедуры, которая определяет кратчайшие расстояния с помощью алгоритма экономного продвижения, мы должны выбрать реализацию для ориентированного графа. Использование в реализации таблицы смежности обеспечивает произвольный доступ ко всем вершинам графа, как это требуется для решения нашей задачи. Более того, записывая веса в таблицу, мы можем использовать таблицу для предоставления данных не только о смежности, но и о весах. Мы будем помещать специальное большое значение  $\infty$  в любую позицию таблицы, для которой соответствующее ребро не существует. Все эти решения отражены в следующих объявлениях на языке Pascal, которые следует включить в вызывающую программу.

 $\{ \text{кратчайшие расстояния от вершины 1} \}$ 

Разрабатываемая нами процедура будет принимать таблицу смежности и число вершин в графе в качестве своих входных параметров, и образовывать в качестве своего выходного параметра таблицу минимальных расстояний.



Процедура Distance выглядит следующим образом:

```

procedure Distance (size: counter; var cost: adjacencytable; { Расстояние }
                        var D: distancetable);
{ Pre: Дан направленный граф, имеющий size вершин с весами ребер,
    данными в таблице cost.
  Post: Процедура находит кратчайший путь от вершины 1
    к каждой вершине графа и возвращает найденный ею путь
    в массиве D. }
var
  final: array [vertex] of Boolean; { Найдено ли окончательное расстояние
                                     от 1 до v? Значение final [v] равно true,
                                     если и только если v принадлежит множеству S. }
  i, { счетчик повторений для главного цикла }
  { В каждом шаге цикла заканчивается обработка одного расстояния. }
  w, { вершина, еще не добавленная в множество S }
  v: vertex; { вершина с минимальным пробным расстоянием в D [ ] }
  min: weight; { расстояние для v, равно D [v] }
begin { процедура Distance }
  final [1] := true; { инициализируем с единственной вершиной 1
                     в множестве S }

  D[1] := 0;
  for v := 2 to size do
    begin
      final [v] := false;
      D[v] := cost [1, v]
    end;
  for i := 2 to size do
    begin { начнем главный цикл; на каждом шаге
          добавляем одну вершину v к S }
      min := infinity; { найдем вершину, ближайшую к вершине 1 }
      for w := 2 to size do
        if not final [w] then
          if D[w] < min then
            begin
              v := w;
              min := D[w]
            end;
      final [v] := true; { добавим v к множеству S }
      for w := 2 to size do { обновим оставшиеся расстояния в D }
        if not final [w] then
          if min + cost [v, w] < D[w] then
            D[w] := min + cost [v, w]
        end
    end;
end; { процедура Distance }

```

Для оценки времени выполнения этой процедуры мы замечаем, что главный цикл выполняется  $n - 1$  раз, где  $n = \text{size}$  есть число вершин, а внутри главного цикла имеются еще два цикла, каждый из которых выполняется  $n - 1$  раз, поэтому все эти циклы дают вклад из  $(n - 1)^2$  операций. Предложения, выполняемые вне циклов, добавляют всего лишь  $O(n)$ , и в результате время выполнения алгоритма имеет порядок  $O(n^2)$ .

процедура  
кратчайшего  
расстояния

производитель-  
ность

## 12.6. Графы как структуры данных

В этой главе мы рассмотрели некоторые приложения графов, но при этом мы едва коснулись широкого и глубокого предмета алгоритмов, построенных на основе графов. Во многих таких алгоритмах графы выступают, как и в этой главе, в качестве математических структур, отражающих принципиальные характеристики решаемой задачи, а не как вычислительные средства для ее решения.

математические  
структуры и  
структуры данных

Заметьте, что в этой главе мы ввели графы как математические структуры, а не как структуры данных, потому что мы использовали графы для математической формулировки задачи, а для того, чтобы написать алгоритмы, мы реализовали графы с помощью таких структур данных, как таблицы и списки. Графы, однако, безусловно могут и сами рассматриваться как структуры данных, в которых соотношения между данными оказываются более сложными, чем в списках или деревьях.

гибкость и  
мощность

В силу своей общности и гибкости графы могут представлять собой мощные структуры данных, весьма ценные для таких сложных приложений, как, например, разработка систем управления базами данных. Такие мощные средства предназначены, разумеется, для использования всюду, где они необходимы, однако к их использованию следует подходить с большой осторожностью, чтобы их мощь не привела к путанице и, в конечном счете, к разрушению ваших планов. Возможно, лучшей гарантией правильного использования таких мощных средств является соблюдение аккуратности и системности, т. е. применение этих средств только в тщательно разработанных и глубоко осознанных алгоритмах. Ввиду общности понятия графа проведение этого принципа не всегда оказывается простым делом.

В нашем далеком от идеала мире, однако, ошибки несистемного подхода имеют тенденцию вкрадываться в нашу работу, как бы тщательно мы не старались от них избавиться. Они являются бичом системных аналитиков и программистов, которые, стараясь поддерживать целостность базового системного проекта, должны бороться с проявлениями несистемностей. Нарушения системности могут проявляться даже в самих концепциях, которые мы используем в качестве моделей для разрабатываемых нами структур данных, например, в генеалогическом древе, терминологией которого («тетя», «дед») мы пользовались при рассмотрении деревьев. Прекрасной иллюстрацией к сказанному является классическая история, цитируемая Н. Виртом (N. Wirth)<sup>2</sup> из цюрихской газеты за июль 1922 г.

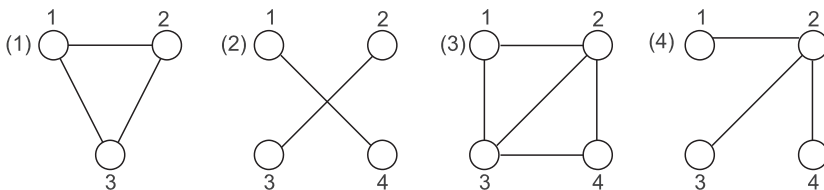
Я женился на вдове, у которой была взрослая дочь. Мой отец, часто посещавший нас, влюбился в мою падчерицу и женился на ней. Таким образом, мой отец стал моим зятем, а моя падчерица стала моей матерью. Через несколько месяцев моя жена родила мне сына, который стал шурином (братом жены) моего отца, а также и моим дядей. Жена моего отца, т. е. моя падчерица, также родила сына. Таким образом, я приобрел бра-

<sup>2</sup> *Algorithms + Data Structures = Programs*, Prentice Hall, Englewood Cliffs, N.J., 1976, page 170.

та и, в то же время, внука. Моя жена стала моей бабушкой, поскольку она является матерью моей матери. Поэтому я являюсь мужем своей жены и, в то же время, ее сводным внуком; другими словами, я оказался собственным дедушкой.

### Упражнения 12.6

**Е1. (а)** Найдите все циклы в каждом из приведенных ниже графов. **(б)** Какие из этих графов являются связными? **(с)** Какие из этих графов представляют собой свободные деревья?



**Е2.** Для каждого из графов упражнения Е1 дайте реализацию графа как **(а)** таблицы смежности, **(б)** связного списка вершин со связными списками смежности, **(с)** непрерывного списка вершин с непрерывными списками смежности.

**Е3.** Граф называется *регулярным*, если все его вершины имеют одну и ту же валентность (т. е. каждая из них смежна с одним и тем же числом других вершин). Для регулярного графа удобной реализацией является связный список вершин и непрерывные списки смежности. Длина всех списков смежности называется *степенью* графа. Напишите на языке Pascal объявление этой реализации регулярных графов.

**Е4.** Процедуры топологической сортировки, представленные в тексте книги, несовершенны в том отношении, что не содержат проверку на ошибки. Модифицируйте процедуры **(а)** просмотра в глубину и **(б)** просмотра в ширину так, чтобы они обнаруживали любые (направленные) циклы в графе и указывали, какие вершины не могут быть расположены в топологическом порядке, поскольку они лежат на циклах.

### Программные проекты 12.6

**P1.** Напишите Pascal-процедуру с именем ReadGraph, которая читает с терминала число вершин в неориентированном графе и списки смежных вершин. Не забудьте включить в процедуру проверку на ошибки. Граф следует реализовать как:

- (а)** таблицу смежности;
- (б)** связный список вершин со связными списками смежности;
- (с)** непрерывный список вершин со связными списками смежности.

**P2.** Напишите Pascal-процедуру с именем WriteGraph, которая выводит на терминал всю необходимую для определения графа информацию. Граф следует реализовать как:

- (а)** таблицу смежности;
- (б)** связный список вершин со связными списками смежности;
- (с)** непрерывный список вершин со связными списками смежности.

- P3.** Используйте процедуры `ReadGraph` и `WriteGraph` для реализации и тестирования процедур топологической сортировки, разработанных в этом разделе, рассмотрев варианты:
- (a) упорядочения по глубине;
  - (b) упорядочения по ширине.
- P4.** Реализуйте и протестируйте процедуру определения кратчайших расстояний в ориентированном графе с весами.

## Подсказки и ловушки

1. Графы предоставляют превосходный способ описания существенных черт многих приложений, тем самым облегчая спецификацию базовых задач и формулировку алгоритмов для их решения. Иногда графы используются как структуры данных, но чаще они рассматриваются как математические абстракции, полезные при решении задач.
2. Графы можно реализовывать многими различными способами путем использования различного рода структур данных. Откладывайте выбор способа реализации до тех пор, пока вы не добьетесь отчетливого понимания особенностей применения графов на этапах решения задачи и разработки алгоритма.
3. Многие приложения требуют реализации просмотра графа. Выбор метода просмотра (в глубину, в ширину или в каком-то другом порядке) зависит от приложения. Просмотр в глубину естественным образом требует рекурсии (или стека). Просмотр в ширину обычно использует очередь.
4. Алгоритм экономного продвижения представляет собой лишь один пример многочисленных средств разработки алгоритмов с графами. При желании познакомиться с другими методами и примерами, обратитесь к ссылкам.

## Обзорные вопросы

- |      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 12.1 | <ol style="list-style-type: none"><li>1. Что представляет собой <i>граф</i> в плане материала этой главы? Что представляют собой <i>ребра</i> и <i>вершины</i>?</li><li>2. Какова разница между <i>неориентированным</i> и <i>ориентированным</i> графом?</li><li>3. Определите термины <i>смежный</i>, <i>маршрут</i>, <i>цикл</i> и <i>связный</i>.</li><li>4. Что обозначает термин <i>сильносвязный</i> применительно к ориентированному графу? А <i>слабосвязный</i>?</li></ol> |
| 12.2 | <ol style="list-style-type: none"><li>5. Опишите три способа реализации графов в компьютерной памяти.</li><li>6. Объясните разницу в просмотре графа в ширину и в глубину.</li></ol>                                                                                                                                                                                                                                                                                                 |
| 12.3 | <ol style="list-style-type: none"><li>7. Какие структуры данных необходимы для прослеживания ожидающих посещения вершин в процессе (a) просмотра в глубину и (b) просмотра в ширину?</li></ol>                                                                                                                                                                                                                                                                                       |

- 12.4                    8. Для какого вида графов определен термин *топологическая сортировка*?
9. Что представляет собой *топологический порядок* для такого графа?
- 12.5                   10. Почему алгоритм нахождения кратчайших расстояний носит название *алгоритма экономного продвижения*?

## Литература для дальнейшего изучения

Исследование графов и алгоритмов их обработки представляет собой большой предмет, включающий в себя достижения и математики, и вычислительной техники. Для более глубокого ознакомления с этим предметом рекомендуем три книги, каждая из которых содержит много интересных алгоритмов.

R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, 1983, 131 pages.

Shimon Even, *Graph Algorithms*, Computer Science Press, Rockville, Md., 1979, 249 pages.

E. M. Reingold, J. Nievergelt, N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, N. J., 1977, 433 pages.

Имеется русский перевод:

Рейнгольд Э., Нивергельт Ю., Део Н. Комбинаторные алгоритмы. Теория и практика. — М.: Мир, 1980.

Первая публикация об алгоритме экономного продвижения содержится в статье

E. W. Dijkstra, «A note on two problems in connexion with graphs», *Numerische Mathematik* 1 (1959), 269–271.

## Глава 13

# Конкретный пример: польская нотация

---

В этой главе рассматривается использование польской нотации для арифметических и логических выражений, сначала в качестве инструмента решения задач, а затем в приложении к программе, которая интерактивно принимает выражение, компилирует его и затем оценивает. Эта глава иллюстрирует использование рекурсии, стеков, таблиц и деревьев, а также их взаимодействие при решении задач и разработке алгоритмов.

### 13.1. Постановка задачи

Одним из важнейших достижений ранних разработчиков компьютерных языков было предоставление возможности программистам писать арифметические выражения в форме, близкой к той, которую они использовали на бумаге. Явилось полным триумфом создание компилятора, понимавшего выражения вроде

$$\begin{aligned} &(x + y) * \exp(x - z) - 4.0 \\ &a * b + c/d - c * (x + y) \\ &\text{not } (p \text{ and } q) \text{ or } (x \leq 7.0) \end{aligned}$$

и преобразовывавшего их в программу на машинном языке. Фактически имя языка Fortran было произведено от названия транслятора формул

ЭТИМОЛОГИЯ:  
FORTRAN

FORmula TRANslator

в честь этого замечательного достижения. Часто всего лишь одна простая идея, ясно осознанная, предоставляет ключ к элегантному решению сложной проблемы, в данном случае — трансляции математических выражений в последовательность команд на машинном языке.

Суть метода, которому посвящена эта глава, заключается в том, что в противоположность привычному подходу, для расшифровки выражения не требуется его многократный просмотр и расшифровка, и после первого преобразования уже не возникает необходимости принимать во внимание ни скобки, ни приоритеты операторов, так что оценка выражения может быть выполнена с большой эффективностью.

#### 13.1.1. Формула корней квадратного уравнения

Перед тем как приступить к обсуждению этой идеи, остановимся кратко на проблемах, с которыми могли столкнуться ранние разработчики ком-

пиляторов при трансляции относительно сложных выражений. Даже формула для вычисления корней квадратного уравнения приводит к проблемам:

$$x := (-b + (b \uparrow 2 - (4 \times a) \times c) \uparrow \frac{1}{2}) / (2 \times a)$$

обозначения

Здесь и далее в этой главе мы обозначаем возведение в степень символом ' $\uparrow$ '. Извлекая квадратные корни, мы будем ограничиваться только положительными корнями.

Какие операции следует выполнять перед другими? Каково влияние скобок? Когда их можно опускать? Отвечая на эти вопросы применительно к приведенному примеру, вы, скорее всего, будете просматривать выражение назад и вперед несколько раз.

Раздумывая, каким образом транслировать такого рода выражения, разработчики компиляторов вскоре остановились на хорошо известных теперь соглашениях. Операции обычно выполняются слева направо с учетом приоритетов, присвоенных операторам, которые располагаются в таком порядке: максимальным приоритетом обладает операция возведения в степень, затем идут умножение и деление, далее сложение и вычитание. Этот порядок может быть изменен с помощью скобок. Для формулы корней квадратного уравнения порядок операторов выглядит следующим образом:

---


$$x := (-b + (b \uparrow 2 - (4 \times a) \times c) \uparrow \frac{1}{2}) / (2 \times a)$$

|            |            |            |            |            |            |            |            |            |
|------------|------------|------------|------------|------------|------------|------------|------------|------------|
| $\uparrow$ | $\uparrow$ | $\uparrow$ | $\uparrow$ | $\uparrow$ | $\uparrow$ | $\uparrow$ | $\uparrow$ | $\uparrow$ |
| 10         | 1          | 7          | 2          | 5          | 3          | 4          | 6          | 9          |

---

Заметьте, что присваивание в действительности является оператором, который берет значение своего правого операнда и назначает его своему левому операнду. Приоритет оператора  $:=$  оказывается самым низким из всех операторов, поскольку присваивание может быть выполнено только после того, как выражение будет полностью вычислено.

## 1. Одноместные операторы и приоритеты

С одним исключением все операторы в формуле корней квадратного уравнения являются *двуместными*, т. е. имеющими два операнда. Единственное исключение — лидирующий знак минуса в выражении  $-b$ . Этот оператор является *одноместным*, и одноместные операторы приводят к некоторым сложностям при определении приоритетов. Обычно мы интерпретируем  $-2^2$  как  $-4$ , что означает выполнение отрицания после возведения в степень, однако  $2^{-2}$  мы интерпретируем как  $\frac{1}{4}$ , а не как  $-4$ , т. е. отрицание выполняется в первую очередь. Разумно назначить одноместным операторам тот же приоритет, что и возведению в степень, и, с целью сохранения обычных алгебраических соглашений, оценивать операторы этого приоритета справа налево. Соблюдение этого правила позволяет рассматривать выражение  $2 \uparrow 3 \uparrow 2$  как

$$2^{(3^2)} = 512, \text{ а не как } (2^3)^2 = 64.$$

список  
приоритетов

приоритеты  
языка Pascal

Кроме отрицания, есть и другие одноместные операторы. Сюда входят такие операции, как факториал  $x$ , обозначаемый  $x!$ , производная от функции  $f$ , обозначаемая  $f'$ , а также все функции от одной переменной: тригонометрические, экспоненциальные и логарифмические. Имеется также и булев оператор **not**, который производит отрицание булевой переменной.

Некоторые двуместные операторы также производят булев результат: операторы **and** и **or**, а также операторы сравнения  $=$ ,  $\neq$ ,  $<$ ,  $>$ ,  $\leq$  и  $\geq$ . Эти сравнения обычно выполняются после арифметических операторов, но перед **and**, **not** и оператором присваивания.

Итак, мы получаем следующий список приоритетов, отражающий привычные нам правила оценивания операторов:

| Операторы                              | Приоритет |
|----------------------------------------|-----------|
| $\uparrow$ , все одноместные операторы | 6         |
| $\times$ / <b>div mod</b>              | 5         |
| $+$ $-$ (двуместный)                   | 4         |
| $= \neq > < \leq \geq$                 | 3         |
| <b>not</b>                             | 2         |
| <b>and or</b>                          | 1         |
| $:=$                                   | 0         |

Заметьте, что приведенные в этой таблице приоритеты не совпадают с приоритетами, используемыми в языке Pascal. Согласно синтаксическим правилам языка Pascal, **and** имеет приоритет 5, а **or** имеет приоритет 4. Это означает, что в Pascal-выражениях часто приходится использовать скобки, хотя при назначении операторам **and** и **or** более низких приоритетов выражение читается вполне недвусмысленно. Например, выражение

$x < 10$  **and**  $y < 12$

будет неправильно интерпретировано языком Pascal как

$x < (10$  **and**  $y) < 12$

что является очевидной бессмыслицей. Если, однако, мы разрабатываем свою собственную систему, мы можем устанавливать собственные соглашения по своему желанию. Используя приоритеты, приведенные в таблице, мы интерпретируем выражения более привычным для нас образом, а не так, как это делает язык Pascal.

## 13.2. Идея

### 13.2.1. Дерево выражения

Графическое представление часто является превосходным способом более глубокого проникновения в проблему. Применительно к рассматриваемой нами задаче таким удобным графическим представлением является *дерево выражения*, с которым мы уже познакомились в разделе



10.1.2. Вспомним, что дерево выражения — это двоичное дерево, в котором листья представляют простые операнды, а внутренние вершины — операторы. Если оператор двуместный, тогда он имеет два непустых поддерева, соответствующие его левому и правому операндам (которые могут быть простыми операндами или подвыражениями). Если оператор одноместный, тогда только одно из его поддеревьев не пусто, левое или правое, в зависимости от того, пишется ли этот оператор справа или слева от своего операнда. Вам следует снова взглянуть на рис. 10.3, где приведено несколько простых примеров деревьев выражения, а также на рис. 10.4 с деревом выражения для формулы вычисления корней квадратного уравнения.

Давайте решим, как мы будем оценивать дерево выражения, например, то, что изображено на рис. 13.1, (а). Ясно, что мы должны начать с одного из листьев, поскольку, начиная оценку, мы знаем значения только простых операндов. Ради единообразия начнем с самого левого листа, значение которого равно 2.9. Поскольку в нашем примере оператор, расположенный непосредственно над этим листом, является одноместным отрицанием, мы можем тут же применить его к операнду и заменить и оператор, и его операнд полученным результатом, именно, числом  $-2.9$ . Этот шаг отражен ромбовидным узлом на рис. 13.1, (b).

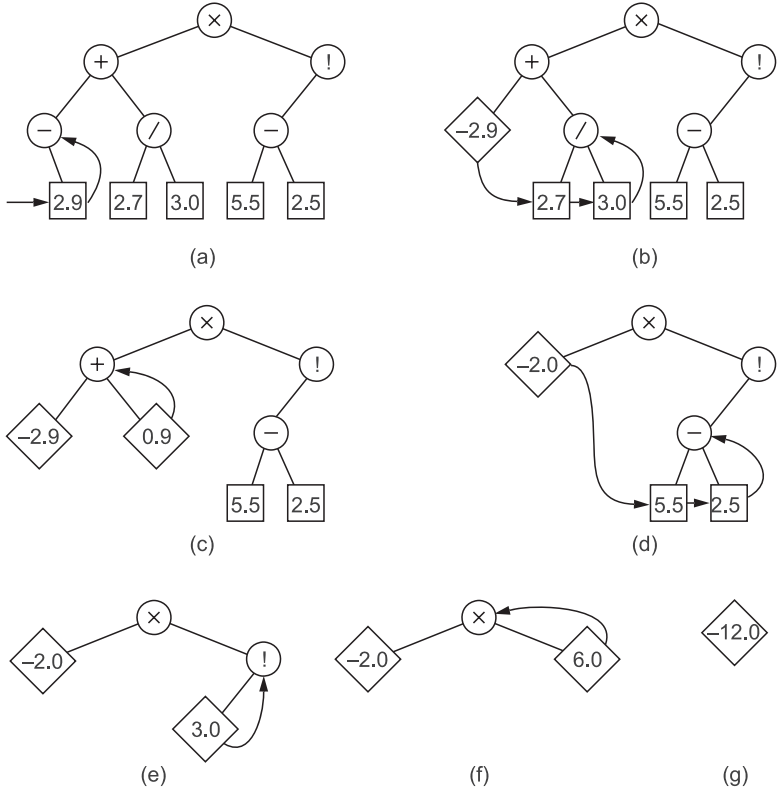


Рис. 13.1. Оценка дерева выражения

Родителем ромбовидного узла на рис. 13.1, (b) является двуместный оператор, и его второй операнд еще не вычислен. Мы, следовательно, еще не можем применить этот оператор, а должны вместо этого рассмотреть два следующих листа, к которым ведет изогнутая стрелка. После прохождения по этим двум листьям маршрут (линии со стрелками) подходит к родительскому для них оператору, который теперь можно вычислить, и поместить результат во второй ромбовидный узел, как это показано на рис. 13.1, (c).

На этой стадии оба операнда для операции сложения уже имеются, поэтому мы можем выполнить эту операцию, получив упрощенное дерево рис. 13.1, (d). Таким же образом мы продолжаем движение по дереву до тех пор, пока дерево не сократится до единственного узла, который и даст нам конечный результат. В целом мы обрабатывали узлы дерева в таком порядке:

$$2.9 - 2.7 \quad 3.0 / + \quad 5.5 \quad 2.5 - ! \times$$

Таким образом, мы видим, что обработка поддерева, корнем которого является любой данный оператор, должна выполняться в следующем порядке:

*Оцениваем левое поддерево; оцениваем правое поддерево; выполняем оператор.*

(Если оператор одноместный, тогда один из этих шагов будет пустым.) Этот порядок в точности совпадает с обратным просмотром дерева выражения. Мы уже отмечали в разделе 10.1.2, что обратный просмотр дерева выражения соответствует постфиксной форме выражения, в которой каждый оператор пишется после своих операндов, а не между ними.

Эта простая идея является ключом к эффективному вычислению выражений компьютером.

Собственно говоря, наш привычный способ написания арифметических или логических выражений, когда операторы указываются между своими операндами, слегка нелогичен. Инструкция

*Возьмите число 12 и умножьте его на ...*

будет незаконченной до тех пор, пока не будет указан второй сомножитель. Тем временем мы вынуждены запоминать и число, и оператор. С точки зрения установки унифицированных правил более разумно писать либо

*Возьмите числа 12 и 3, затем умножайте их.*

либо

*Выполните умножение. Числа есть 12 и 3.*

### 13.2.2. Польская нотация

Такой метод указания операторов либо перед своими операндами, либо после них носит название *польской нотации*, или *польской формы* в честь ее создателя, польского математика Яна Лукасевича. Если операторы указываются перед своими операндами, запись называется **префиксной формой**. Если операторы указываются после своих операндов, мы

имеем постфиксную форму, которая иногда называется также *обратной польской формой* или *суффиксной формой*. Наконец, в этом контексте для обозначения обычной формы указания двуместных операторов между своими операндами, используется название *инфиксная форма*.

Выражение  $a \times b$  превращается в  $\times a b$  в префиксной форме и в  $a b \times$  постфиксной форме. В выражении  $a + b \times c$  первым выполняется умножение, поэтому мы преобразуем его, получая сначала  $a + (b \times c)$ , а затем  $a b c \times +$  в постфиксной форме. Префиксной формой этого выражения будет  $+ a \times b c$ . Заметьте, что префиксная и постфиксная формы не образуются путем зеркального отражения или других столь же простых преобразований. Заметьте также, что в польских формах опускаются все скобки. Об этом мы поговорим позже.

В качестве более сложного примера, мы можем написать префиксную и постфиксную формы формулы вычисления корней квадратного уравнения, начав с ее дерева выражения, изображенного на рис. 10.4.

прямой просмотр

Прежде всего выполним прямой просмотр дерева. Оператор в корне есть присвоение  $:=$ , после которого мы перемещаемся к левому поддереву, в котором лишь один операнд  $x$ . Правое поддерево начинается с деления  $/$ , после которого с левой стороны следует  $+$ , а далее одноместное отрицание  $'$ .

специальный символ

Теперь мы столкнулись с неопределенностью, которая будет преследовать нас и позже, если мы ее не исправим. Первый  $'$  (минус) в выражении является одноместным отрицанием, а второй — двуместным вычитанием. В польской форме неясно, какой из них что означает. Когда мы переходим к оценке префиксной строки, мы не знаем, использовать ли для операции  $'$  один операнд или два, а результаты в этих случаях будут совершенно различными. Чтобы устранить эту неопределенность, мы в этой главе зарезервируем  $'$  для обозначения двуместного вычитания, и будем использовать специальный символ  $\sim$  для одноместного отрицания. (Эта терминология безусловно нестандартна. Для решения нашей проблемы имеются и другие способы.)

Прямой просмотр дерева на рис. 10.4 к этому моменту дал

$$:= x / + \sim b$$

а следующий шаг заключается в просмотре правого поддерева оператора  $+$ . В результате получится предложение

$$\uparrow - \uparrow b 2 \times \times 4 a c \frac{1}{2}$$

Наконец, мы просматриваем правое поддерево оператора деления  $/$ , получая

$$\times 2 a$$

Отсюда законченная префиксная форма формулы корней квадратного уравнения выглядит так:

$$:= x / + \sim b \uparrow - \uparrow b 2 \times \times 4 a c \frac{1}{2} \times 2 a$$

Вам следует самостоятельно убедиться в правильности постфиксной формы :

$$x b \sim b 2 \uparrow 4 a \times c \times - \frac{1}{2} \uparrow + 2 a \times / :=$$

### 13.2.3. Метод для языка Pascal

Перед тем, как завершить этот раздел, следует заметить, что большинство Pascal-компиляторов при трансляции выражений в машинный язык не используют польские формы (хотя другие языки это делают). Большинство Pascal-компиляторов используют метод рекурсивного спуска, который был описан в разделе 4.3. В этом методе каждый приоритет оператора требует отдельной процедуры для его компиляции (и отдельной синтаксической диаграммы для его определения). Это требование частично объясняет, почему Pascal использует усеченный список приоритетов в сравнении с используемым в этой главе.

синтаксический  
анализ

Трансляция выражения методом рекурсивного спуска носит название *нисходящего синтаксического анализа*, в то время как описанный в этой главе метод трансляции выражения посредством просмотра каждого из его компонентов по очереди есть пример *восходящего синтаксического анализа*.

#### Упражнения 13.2

(a) Изобразите графически дерево выражения для каждого из приведенных ниже выражений в (b) префиксной и (c) постфиксной формах. Используйте при этом таблицу приоритетов, приведенную в этом разделе, не таблицу языка Pascal.

E1.  $a + b < c$

E2.  $a < b + c$

E3.  $a - b < c - d \text{ or } e < f$

E4.  $n \text{ div } (k! \times (n - k)!)$  (Формула для биномиальных коэффициентов.)

E5.  $s := (n/2) \times (2 \times a + (n - 1) \times d)$  (Это сумма первых  $n$  членов арифметической прогрессии.)

E6.  $g := a \times (1 - r^n)/(1 - r)$  (Сумма первых  $n$  членов геометрической прогрессии.)

E7.  $a = 1 \text{ or } b \times c = 2 \text{ or } (a > 1 \text{ and not } b < 3)$

## 13.3. Оценка выражений в польской нотации

Мы сначала ввели постфиксную форму, как более естественный порядок просмотра дерева выражения, с целью оценки соответствующего выражения. Позже в этом разделе мы сформулируем алгоритм для оценки выражения непосредственно из постфиксной формы, но сначала (поскольку это даже проще) рассмотрим префиксную форму.

### 13.3.1. Оценка выражений в префиксной форме

Прямой просмотр двоичного дерева выполняется сверху вниз. Сначала посещается корень, и оставшаяся часть просмотра разделяется на две части. Естественным способом организации процесса является рекурсивный алгоритм разбиения. Та же ситуация возникает при оценке выражения в префиксной форме. Первым символом (если символов более одного) является оператор (который фактически будет выполняться в последнюю очередь), а остаток выражения состоит из операнда (или операндов)

этого оператора (одного для одноместного оператора и двух для двуместного). Наша процедура оценки префиксной формы должна, таким образом, начать с этого первого символа. Если это одноместный оператор, тогда процедура должна вызвать саму себя рекурсивно для определения значения операнда. Если первым символом является двуместный оператор, тогда процедура должна выполнить два рекурсивных вызова для двух операндов этого оператора. Рекурсия завершается в оставшемся случае: когда первый символ является простым операндом, он является собственной префиксной формой и процедура просто должна вернуть его значение.

Следующий эскиз подытоживает оценку выражения в префиксной форме.

```

procedure Evaluate(expression, result);                                { Оценить }
begin
  Пусть t является первым символом в выражении и мы смещаемся
  на одну позицию по выражению;
  if t есть одноместный оператор then
    begin
      Evaluate(expression, x);
      Присвоим результату значение оператора t, приложенного к x
    end
  else if t есть двуместный оператор then
    begin
      Evaluate (expression, x);
      Evaluate(expression, y);
      Присвоим результату значение оператора t, приложенного к x и y
    end
  else                                { t является простым операндом }
    Присвоим результату значение оператора t
  end;

```

### 13.3.2. Соглашения языка Pascal

С целью дальнейшей детализации приведенного эскиза, давайте установим некоторые соглашения и перепишем алгоритм на языке Pascal. Операторы и операнды в нашем выражении вполне могут иметь имена, имеющие длину более одного символа; поэтому мы не должны просматривать выражение по одному символу за раз. Будем называть *лексемой* (token) один оператор или операнд в выражении. Для подчеркивания того факта, что процедура просматривает выражение только один раз, мы используем вспомогательную процедуру

GetToken(**var** t: token; **var** E: expression)

которая будет перемещаться по выражению E и возвращать каждый раз по одной лексеме t. Нам нужно знать, является ли лексема операндом, одноместным оператором или двуместным оператором, поэтому мы предполагаем наличие функции Kind( ), которая возвращает одну из трех констант перечислимого типа:

|          |                      |                      |
|----------|----------------------|----------------------|
| operand  | unaryop              | binaryop             |
| [операнд | одноместный оператор | двуместный оператор] |

Для простоты мы будем предполагать, что все операнды и результаты оценки операторов относятся к одному и тому же типу, который мы пока оставим неопределенным, и будем называть его *value*. Во многих приложениях этот тип будет целочисленным, действительным или булевым.

Наконец, мы должны предположить наличие трех вспомогательных функций, возвращающих результат типа *value*. Первые две из них

```
DoUnary(t: token, x: value): value
DoBinary(t: token, x, y: value): value
```

фактически выполняют данную операцию над своими операндами (операндом). Эти функции распознают символы, используемые для операции *t* и операндов *x* и *y*, и активизируют соответствующие команды машинного языка. Аналогично

```
GetValue (t: token): value
```

возвращает фактическое значение простого операнда *t* и может понадобиться, например, для преобразования константы из десятичной формы в двоичную или для определения значения некоторой переменной. Реальный текст этих функций будет решающим образом зависеть от приложения. Мы не можем решить все эти вопросы здесь, а хотели бы лишь рассмотреть проектирование одного важного элемента компилятора или программы оценивания выражений.

### 13.3.3. Pascal-процедура для префиксной оценки

Высказав эти предварительные соображения, мы можем теперь начать детализацию нашего эскиза, преобразуя его в Pascal-процедуру для оценки префиксных выражений.

```
procedure EvaluatePrefix (var E: expression; var result: value);
                                { Оценить префиксное выражение }
{ Pre:   E является выражением в префиксной форме.
  Post:  Выражение E было оценено (и, возможно, поглощено в процессе
          оценки) с получением результата в виде значения. }
var
  t: token;
  x, y: value;
begin                                { процедура Evaluate Prefix }
  GetToken(t, E);
  case Kind(t) of
    unaryop:
      begin
        EvaluatePrefix(E, x);
        result := DoUnary(t, x)
      end;
    binaryop:
      begin
        EvaluatePrefix(E, x);
        EvaluatePrefix(E, y);
        result := DoBinary(t, x, y)
      end;
```

```

operand:
    result := GetValue(t)
end
end;                                     { процедура EvaluatePrefix }

```

### 13.3.4. Оценка постфиксных выражений

Представляется почти неизбежным, что префиксная форма естественным образом приводит к рекурсивной функции оценки выражений, поскольку префиксная форма есть, по существу, «нисходящая» формулировка алгебраического выражения: внешние, глобальные действия указываются в первую очередь, а затем в выражении описываются компонентные части. С другой стороны, в постфиксной форме сначала указываются операнды, а все выражение постепенно строится начиная от простых операндов и внутренних операторов в «восходящем» стиле. Поэтому для постфиксной формы представляются более естественными итеративные программы, использующие стеки. (Разумеется, вполне возможно писать и для той, и для другой форм как рекурсивные, так и нерекурсивные программы. Мы здесь обсуждаем только мотивацию, или наиболее естественный выбор.)

Чтобы оценить выражение в постфиксной форме, необходимо помнить операнды до тех пор, пока позже не будет обнаружен их оператор. Естественным способом запоминания операндов является их проталкивание в стек. Когда будет обнаружен первый выполняемый оператор, он найдет свои операнды на вершине стека. Если результат операции также отправляется назад в стек, тогда этот результат окажется в правильном месте в качестве операнда для следующего оператора. После полного завершения оценки окончательный результат будет единственным значением на стеке. В результате мы получаем процедуру оценки постфиксного выражения.

Теперь следует отметить существенное различие между постфиксными и префиксными выражениями. В префиксной процедуре не было необходимости явным образом проверять достижение конца выражения, поскольку все выражение автоматически поставляло операнды (операнд) для первого оператора. Однако, читая постфиксное выражение слева направо, мы можем встретиться с подвыражениями, которые сами по себе являются допустимыми постфиксными выражениями. Например, если мы прекратим чтение выражения

$$b \ 2 \ \uparrow \ 4 \ a \ \times \ c \ \times \ -$$

после знака  $\uparrow$ , мы найдем, что прочитанное является допустимым постфиксным выражением. Для устранения этой проблемы мы введем булеву функцию

```
function EndExpression(var E: expression): Boolean;
```

которая принимает значение true после того, как GetToken вернет последнюю лексему выражения E.

Оставшиеся типы языка Pascal, а также другие вспомогательные процедуры, остаются такими же, как и в случае оценки префиксных выражений.

```

procedure EvaluatePostfix (var E: expression; var result: value);
                                { Оценить постфиксное выражение }
{ Pre:   E является выражением в постфиксной форме.
  Post:  Выражение E было оценено (и, возможно, поглощено
          в процессе оценки) с получением результата в виде значения. }
var
  S: stack;           { хранит значение до момента распознания оператора }
  t: token;           { текущий оператор или операнд }
  x, y: value;        { операнды текущего оператора }
begin                                { процедура EvaluatePostfix }
  CreateStack(S);
  repeat
    GetToken(t, E);
    case Kind(t) of
      operand:   Push(GetValue(t), S);
      unaryop:   begin
                    Pop(x, S);
                    Push(DoUnary(t, x), S)
                  end;
      binaryop:  begin
                    Pop(y);
                    Pop(x);
                    Push(DoBinary(t, x, y), S)
                  end
    end
  until EndExpression(E);
  if StackSize(S) = 1 then
    Pop(result, S)
  else
    Error('В конце постфиксной оценки должен быть в точности один элемент.')
  end;
                                { процедура EvaluatePostfix }

```

### 13.3.5. Доказательство правильности программы: подсчет элементов в стеке

До сих пор мы привели лишь неформальную мотивацию для разработки предыдущей программы, и остается неясным, будет ли она давать во всех случаях правильные результаты. К счастью, нетрудно дать формальное обоснование программе и, в то же время, установить полезный критерий проверки того, правильно ли написано выражение в постфиксной форме.

Метод, которым мы воспользуемся, заключается в прослеживании числа элементов в стеке. Как только получается очередной операнд, он сразу же проталкивается в стек. Одноместный оператор сначала вытаскивает операнд из стека, а затем проталкивает результат в стек, не изменяя в результате число элементов в стеке. Двуместный оператор вытаскивает из стека два операнда, а проталкивает только один, давая в ре-





В любом случае длины  $F$  и  $G$  меньше чем у  $E$ , поэтому по индуктивному предположению оба эти выражения удовлетворяют условию текущей суммы, и процедура оценит каждое из них отдельно и получит правильный результат.

одноместный  
оператор

Сначала возьмем случай, когда  $op$  есть одноместный оператор. Поскольку  $F$  удовлетворяют условию текущей суммы, эта сумма в конце  $F$  будет в точности равна  $+1$ . Будучи одноместным оператором,  $op$  вносит в сумму  $0$ , поэтому полное выражение  $E$  удовлетворяет условию текущей суммы. Аналогично, когда процедура достигает конца  $F$ , она по индуктивному предположению оценит  $F$  правильно и оставит его значение в качестве единственного элемента стека. Одноместный оператор  $op$  будет в конце концов приложен к этому значению, что вытолкнет его из стека в качестве окончательного результата.

двуместный  
оператор

Наконец рассмотрим случай, когда  $op$  есть двуместный оператор, и  $E$  имеет форму  $F G op$ , причем  $F$  и  $G$  есть постфиксные выражения. Когда процедура достигает последней лексемы  $F$ , значение  $F$  будет единственным элементом в стеке. Текущая сумма будет равна  $1$ . На следующей лексеме программа начинает оценивать  $G$ . По индуктивному предположению оценка  $G$  также будет выполняться правильно, и его текущая сумма не упадет ниже  $1$ , а в конце будет в точности равна  $1$ . Поскольку текущая сумма в конце  $F$  есть  $1$ , общая текущая сумма не упадет ниже  $2$ , а в конце  $G$  будет в точности равна  $2$ . Таким образом, оценка  $G$  будет продолжаться и ни при каких обстоятельствах не затронет один элемент на дне стека, который является результатом  $F$ . Когда оценка достигнет последнего двуместного оператора  $op$ , текущая сумма правильно снизится с  $2$  до  $1$ , и оператор найдет в точности свои два операнда в стеке, где после оценки он оставит единственный результат. Этим завершается доказательство теорем 13.1 и 13.2.

Такая проверка ошибок путем прослеживания текущей суммы числа элементов в стеке предоставляет удобный способ убедиться в том, что последовательность лексем есть в действительности правильно сформированное постфиксное выражение, и особенно полезна ввиду того, что обратное также справедливо:

#### Теорема 13.3

*Если  $E$  есть любая последовательность операндов и операторов, удовлетворяющая условию текущей суммы, тогда  $E$  есть правильно сформированное выражение в постфиксной форме.*

#### Доказательство

Мы снова используем для доказательства теоремы 13.3 метод математической индукции. Начальной точкой будет выражение, содержащее только одну лексему. Поскольку текущая сумма (как и конечная сумма) для последовательности длины  $1$  равна  $1$ , эта лексема должна быть простым операндом. Единственный простой операнд действительно является синтаксически правильным выражением.

метод индукции

Теперь для выполнения шага индукции предположим, что теорема была проверена для всех выражений, строго меньших, чем  $E$ , и  $E$  имеет длину больше  $1$ . Если бы последняя лексема выражения  $E$  была операндом, она вложила бы в сумму  $+1$ , и поскольку конечная сумма равна  $1$ , текущая сумма в точке за один шаг до конца была бы равна  $0$ , что проти-

случай: операнд

воречит предположению, что условие текущей суммы удовлетворяется. Таким образом, последняя лексема выражения  $E$  должна быть оператором.

случай:  
одноместный  
оператор

Если оператор одноместный, тогда он может быть опущен, и оставшаяся последовательность все еще удовлетворяет условию текущей суммы. Поэтому, согласно индуктивному предположению, выражение является синтаксически правильным, и таковым же является все выражение  $E$ .

случай:  
двуместный  
оператор

Наконец, предположим, что последняя лексема есть двуместный оператор  $op$ . Чтобы доказать, что выражение  $E$  синтаксически правильно, мы должны с помощью текущей суммы найти, где в этой последовательности заканчивается первый операнд оператора  $op$  и начинается второй. Поскольку оператор  $op$  дает в сумму вклад  $-1$ , она за один шаг до конца была равна 2. Это 2 означает, что в стеке были два элемента, первый и второй операнды оператора  $op$ . Если мы отступаем назад по последовательности  $E$ , в конце концов мы достигаем места, где в стеке находится только один элемент (текущая сумма равна 1), и этот один элемент будет первым операндом  $op$ . Таким образом, место разрыва последовательности находится в последней позиции перед концом, где текущая сумма в точности равна 1. Такая позиция должна существовать, поскольку на дальнем левом конце  $E$  (если не раньше) мы найдем значение текущей суммы, равное 1. Когда мы разрываем выражение  $E$  на его последней 1, оно принимает форму  $F G op$ . Подпоследовательность  $F$  удовлетворяет условию текущих сумм и заканчивается с суммой, равной 1, поэтому по индуктивному предположению она является правильно сформированным постфиксным выражением. Поскольку текущие суммы по ходу  $G$  из  $F G op$  опять никогда не снижаются до 1, и перед самым  $op$  сумма равна 2, мы можем вычесть 1 из каждого значения суммы и заключить, что текущие суммы для одного только  $G$  удовлетворяют условию. Поэтому по индуктивному предположению  $G$  также является правильно сформированным постфиксным выражением. Итак, оба выражения, и  $F$ , и  $G$ , являются правильными выражениями, и мы можем скомбинировать их посредством двуместного оператора  $op$  в правильное выражение  $E$ . Доказательство теоремы получено.

Мы можем включить в доказательство еще один шаг и доказать, что последняя позиция, где сумма принимает значение 1, является единственным местом, где предположение  $E$  может быть разорвано на синтаксически правильные подпоследовательности  $F$  и  $G$ . Для доказательства предположим, что расщепление было произведено в каком-то другом месте. Если в конце  $F$  текущая сумма не равна 1, тогда  $F$  не является синтаксически правильным выражением. Если текущая сумма равна 1 в конце  $F$ , но снова достигает 1 по ходу части  $G$  из  $F G op$ , тогда суммы для одного только  $G$  в этой точке достигли бы 0, и  $G$  было бы неверным. Мы теперь показали, что имеется лишь один способ извлечь из выражения два операнда двуместного оператора. Очевидно, имеется лишь один способ извлечь единственный оператор одноместного оператора. Таким образом, мы можем из постфиксной формы выражения получить его инфиксную форму вместе с порядком, в котором выпол-

няются операции, и в инфиксной форме мы можем обозначить этот порядок, помещая в дополнительные скобки результат каждой операции.

Мы, таким образом, доказали:

**Теорема 13.4**

*Выражение в постфиксной форме, которое удовлетворяет условию текущей суммы, соответствует в точности одному выражению в инфиксной форме, содержащему все необходимые скобки. Отсюда следует, что для единственного представления выражения в постфиксной форме нет необходимости прибегать к скобкам.*

Аналогичные теоремы справедливы для префиксной формы; их доказательство мы оставим для упражнений. Доказанные теоремы предоставляют как теоретическое основание использования польской нотации, так и удобный способ проверки выражения на правильность его синтаксиса.

### 13.3.6. Рекурсивная оценка постфиксных выражений

Большинство людей находят, что рекурсивную процедуру для оценки префиксных выражений легче понять, чем основанную на использовании стека нерекурсивную процедуру для оценки постфиксных выражений. В этом (факультативном) разделе мы покажем, как можно отказаться от стека и воспользоваться рекурсией при оценке постфиксного выражения.

Сначала, однако, посмотрим, почему естественный подход приводит нас к рекурсивной процедуре в случае префиксной оценки, но не в случае постфиксной. Мы можем изобразить и префиксное, и постфиксное выражения с помощью синтаксических диаграмм, приведенных на рис. 13.3. В обоих случаях имеются три возможности: выражение состоит из только одного операнда; самый внешний оператор является одноместным; самый внешний оператор является двуместным.

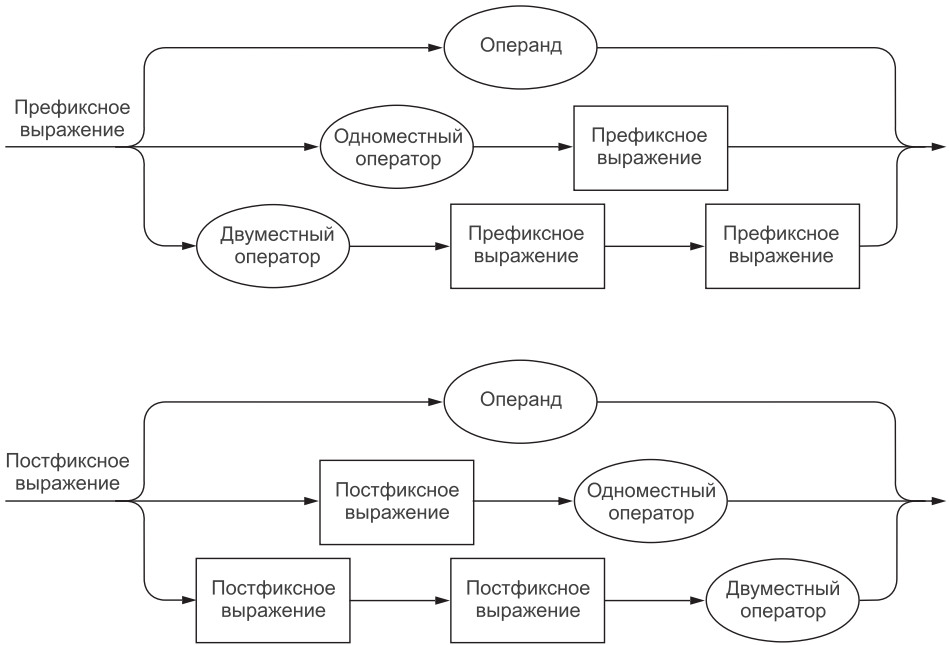
префиксная  
оценка

Продвигаясь по диаграмме префиксной формы, первая лексема, которую мы встречаем в выражении, определяет, какую из трех ветвей мы должны выбрать, после чего уже не потребуются делать выбор (за исключением выбора внутри рекурсивных вызовов, но их мы пока не будем рассматривать). В результате структура рекурсивной процедуры для префиксной оценки весьма схожа с синтаксической диаграммой.

постфиксная  
оценка

Постфиксная диаграмма, однако, не дает нам способа определить по первой лексеме (которая всегда будет операндом), какую из трех ветвей мы должны выбрать. Этот факт, впрочем, ведет к одному простому рекурсивному решению: прочитайте выражение справа налево, обратите все стрелки на синтаксической диаграмме и используйте ту же процедуру, что и для префиксной оценки!

Если, однако, мы хотим читать выражение обычным образом, слева направо, тогда нам придется выполнить больший объем работы. Давайте рассмотрим по отдельности каждый из трех видов лексем в постфиксной форме. Мы уже заметили, что первая лексема в выражении должна быть операндом; это непосредственно следует из того факта, что текущая сумма после первой лексемы равна (по меньшей мере) 1. Поскольку одноместные операторы не изменяют текущую сумму, такой оператор можно вставить куда угодно после начального операнда. Таким образом, рас-



**Рис. 13.3.** Синтаксические диаграммы польских выражений

смотрение именно третьего случая, двуместных операторов, ведет к решению.

текущая сумма

Рассмотрим последовательность текущих сумм и место (или места) в последовательности, где сумма уменьшается от 2 до 1. Поскольку двуместные операторы вносят в сумму  $-1$ , такие места должны существовать, если постфиксное выражение содержит хотя бы один двуместный оператор, причем они должны соответствовать тем местам в выражении, где два операнда двуместного оператора составляют целое выражение с левой стороны. Такие ситуации можно обнаружить в кадрах стека на рис. 13.2. Элемент на дне стека есть первый операнд; последовательность позиций с высотой не меньше 2, начинающаяся и кончающаяся высотой точно 2, составляет вычисление второго операнда, и, взятая в отдельности, эта последовательность сама по себе является правильно сформированным постфиксным выражением. Падение высоты от 2 к 1 отмечает один из двуместных операторов, которые нас сейчас интересуют.

После двуместного оператора могут встретиться одноместные операторы, а затем процесс может повторяться (если текущая сумма снова возрастает), образуя новые последовательности, являющиеся законченными постфиксными выражениями, за которыми опять могут следовать двуместные и одноместные операторы. В целом мы показали, что постфиксные выражения описываются синтаксической диаграммой рис. 13.4, которая легко преобразуется в рекурсивную процедуру, приведенную ниже. Соглашения языка Pascal здесь такие же, как и в предыдущих процедурах.

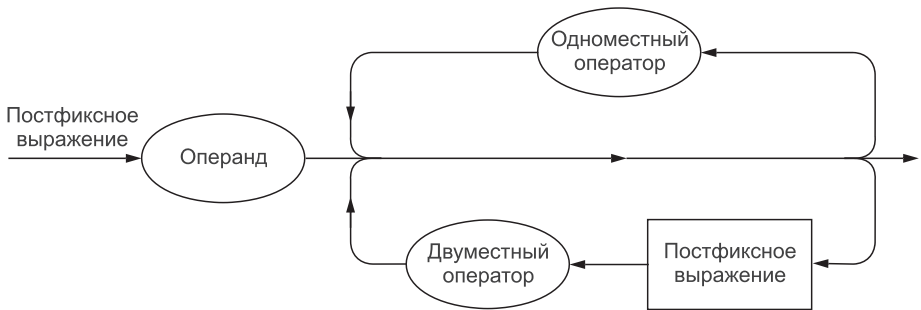


Рис. 13.4. Альтернативная синтаксическая диаграмма, постфиксное выражение

левая рекурсия

Ситуация, показанная на постфиксной диаграмме рис. 13.3, называется *левой рекурсией*, а шаги, предпринимаемые нами для преобразования ее в диаграмму на рис. 13.4, типичны для случаев снятия левой рекурсии.

Сначала приведем текст процедуры, иницирующей рекурсию.

```

procedure EvaluatePostfix (var E: expression; var result: value);
    { Оценить постфиксное выражение }
{ Pre: E является выражением в постфиксной форме.
Post: Выражение E было оценено (и, возможно, поглощено в процессе
    оценки) с получением результата в виде значения. }
var t: token;
begin                                     { процедура EvaluatePostfix }
    GetToken(t, E);
    if Kind(t) <> operand then
        Error('Постфиксное выражение начинается с операнда.')
    else RecEvaluate(t, E, result);
    if not EndExpression(E) then
        Error('Постфиксное выражение закончилось неправильно.')
    end;                                   { процедура EvaluatePostfix }
  
```

Фактическая рекурсия использует первую лексему в выражении отдельно от остальных.

```

procedure RecEvaluate (t: token; var E: expression; var result: value);
    { Рекурсивно оценить }
{ Рекурсивно оценивает постфиксное выражение E начиная с лексемы t,
которая гарантированно является операндом при активизации
RecEvaluate. При завершении процедуры t является первой лексемой
за оцениваемым выражением и при этом является двуместным
оператором, если только не достигнут конец выражения E. }
var x: value;
begin                                     { процедура RecEvaluate }
    result := GetValue(t);                 { постфиксное выражение всегда
                                           начинается с операнда }

    while not EndExpression(E) do
        begin
            GetToken(t, E);
  
```

```

if Kind(t) = unaryop then
  result := DoUnary(t, result)
else begin
  RecEvaluate(t, E, x);
  if Kind(t) = binaryop then
    result := DoBinary(t, result, x)
  else
    Error("За внутренним постфиксным выражением не следует
      двуместный оператор")
end
end
end;

```

{ *t* должно быть операндом }

{ процедура RecEvaluate }

### Упражнения 13.3

**E1.** Проследите действия над каждым из приведенных ниже выражений, выполняемые процедурой EvaluatePostfix в **(a)** нерекурсивном и **(b)** рекурсивном вариантах. Для рекурсивной процедуры нарисуйте дерево рекурсивных вызовов, указав у каждого узла, какая лексема обрабатывается в этом месте. Для нерекурсивной процедуры нарисуйте последовательность кадров стека, показывающую, какие лексемы обрабатываются на каждом шаге.

**(a)**  $a \ b \ + \ c \ \times$

**(b)**  $a \ b \ c \ + \ \times$

**(c)**  $a \ ! \ b \ ! \ / \ c \ d \ - \ a \ ! \ - \ \times$

**(d)**  $a \ b \ < \ \text{not} \ c \ d \ \times \ < \ e \ \text{or}$

**E2.** Проследите действия над каждым из приведенных ниже выражений, выполняемые процедурой EvaluatePrefix, нарисовав дерево рекурсивных вызовов с указанием, какая лексема обрабатывается на каждом шаге.

**(a)**  $/ \ + \ x \ y \ ! \ n$

**(b)**  $/ \ + \ ! \ x \ y \ n$

**(c)**  $\text{and} \ < \ x \ y \ \text{or not} \ = \ + \ x \ y \ z \ > \ x \ 0$

**E3.** Какие из приведенных ниже выражений являются синтаксически правильными? Отметьте ошибки в каждом неправильном выражении. Преобразуйте все правильные выражения в инфиксную форму, используя при необходимости скобки для устранения неоднозначности.

**(a)**  $a \ b \ c \ + \ \times \ a \ / \ c \ b \ + \ d \ / \ -$

**(b)**  $a \ b \ + \ c \ a \ \times \ b \ c \ / \ d \ -$

**(c)**  $a \ b \ + \ c \ a \ \times \ - \ c \ \times \ + \ b \ c \ -$

**(d)**  $a \ \sim \ b \ \times$

**(e)**  $a \ \times \ b \ \sim$

**(f)**  $a \ b \ \times \ \sim$

**(g)**  $a \ b \ \sim \ \times$

**E4.** Преобразуйте каждое из приведенных ниже выражений из префиксной в постфиксную форму.

**(a)**  $/ \ + \ x \ y \ ! \ n$

**(b)**  $/ \ + \ ! \ x \ y \ n$

**(c)**  $\text{and} \ < \ x \ y \ \text{or not} \ = \ + \ x \ y \ z \ > \ x \ 0$



Е5. Преобразуйте каждое из приведенных ниже выражений из постфиксной в префиксную форму.

(a)  $a \quad b \quad + \quad c \quad \times$

(b)  $a \quad b \quad c \quad + \quad \times$

(c)  $a \quad ! \quad b \quad ! \quad / \quad c \quad d \quad - \quad a \quad ! \quad - \quad \times$

(d)  $a \quad b \quad < \quad \text{not} \quad c \quad d \quad \times \quad < \quad e \quad \text{or}$

Е6. Сформулируйте и докажите теоремы, аналогичные теоремам (a) 13.1,

(b) 13.3 и (c) 13.4 для префиксной формы выражений.

## 13.4. Преобразование из инфиксной формы в польскую

Редкие программисты пишут алгебраические или математические выражения в польской форме, даже несмотря на то, что такой способ может быть более последовательным и логичным, чем привычная нам инфиксная запись. Поэтому, чтобы можно было с удобством использовать разработанные нами алгоритмы для оценки польских выражений, мы должны придумать эффективный метод преобразования произвольных выражений из инфиксной формы в польскую.

В качестве первого упрощения мы рассмотрим лишь алгоритм преобразования инфиксных выражений в постфиксную форму. Далее, мы не будем рассматривать одноместные операторы, которые помещаются справа от своих операндов. Такие операторы не должны привести к концептуальным трудностям при разработке алгоритма, но результирующая процедура окажется более запутанной.

Возможным методом для разработки нашего алгоритма могло бы стать построение дерева выражения на основе инфиксной формы, а затем просмотр этого дерева для получения постфиксной формы. Однако построение дерева из инфиксной формы фактически оказывается более сложным, чем непосредственное конструирование постфиксной записи.

Поскольку в постфиксной форме все операторы пишутся за своими операндами, задача преобразования инфиксной записи в постфиксную сводится к такому перемещению операторов, чтобы они размещались после своих операндов, а не между ними. Другими словами,

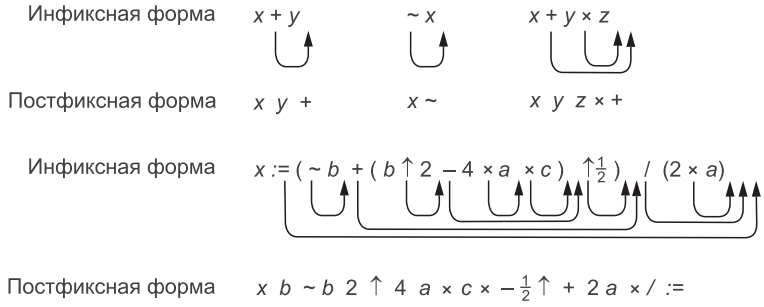
*Отложите действие каждого оператора до тех пор, пока не будет оттранслирован его правый операнд. Каждый простой оператор передавайте на выход без задержки.*

Это действие проиллюстрировано на рис. 13.5.

Основная проблема, которую мы должны разрешить, заключается в определении лексемы, которая завершает правый операнд данного оператора и, следовательно, отмечает то место, где должен быть размещен этот оператор. Чтобы выполнить этот анализ, мы должны принять во внимание и скобки, и приоритеты операторов.

Первая задача относительно проста. Если в операнд входит левая скобка, тогда все, находящееся в выражении вплоть до правой скобки (включая саму скобку), также должно входить в этот операнд. Для реше-





**Рис. 13.5.** Откладывание действия операторов в постфиксной форме

ния второй задачи, учета приоритетов операторов, мы должны будем рассматривать двуместные операторы отдельно от операторов приоритета 6, именно, одноместных операторов и оператора возведения в степень. Причина такого подхода заключается в том, что операторы приоритета 6 оцениваются справа налево, в то время как двуместные операторы с более низким приоритетом оцениваются слева направо.

поиск конца  
правого операнда

Пусть  $op_1$  есть двуместный оператор приоритета, оцениваемого слева направо, и пусть  $op_2$  есть первый не входящий в скобки оператор справа от  $op_1$ . Если приоритет  $op_2$  меньше или равен приоритету  $op_1$ , тогда  $op_2$  не будет частью правого операнда  $op_1$ , и его появление завершит правый операнд  $op_1$ . Если приоритет  $op_2$  больше приоритета  $op_1$ , тогда  $op_2$  является частью правого операнда  $op_1$ , и мы продолжаем просмотр выражения, пока не найдем оператор с приоритетом меньшим или равным, чем у  $op_1$ ; этот оператор и завершит правый операнд  $op_1$ .

оценка  
справа налево

Далее, предположим, что  $op_1$  имеет приоритет 6 (т. е. он принадлежит к одноместным операторам или представляет собой возведение в степень), и вспомним, что операторы этого приоритета оцениваются справа налево. Если первый операнд оператора  $op_2$  справа от оператора  $op_1$  имеет тот же приоритет, он, следовательно, будет частью правого операнда  $op_1$ , и правый операнд завершается только оператором строго меньшего приоритета.

Имеются еще два способа завершения правостороннего операнда данного оператора. Выражение может подойти к концу, или данный оператор может сам быть внутри скобок, заключающих в себе подвыражение, и в этом случае его правый операнд завершится, когда встретится несогласованная правая скобка ')'. В итоге мы имеем такие правила:

*Если  $op$  есть оператор инфиксного выражения, тогда его правосторонний операнд содержит все лексемы справа от себя, пока не встретится следующее:*

1. конец выражения;
2. несогласованная правая скобка ')';
3. оператор с приоритетом, меньшим или равным приоритету  $op$ , и при этом не внутри заключенного в скобки подвыражения, если  $op$  имеет приоритет меньше 6; или
4. оператор с приоритетом, строго меньшим чем у  $op$ , и не внутри заключенного в скобки подвыражения, если  $op$  имеет приоритет 6.

стек операторов

Из этих правил можно увидеть, что для запоминания отложенных операторов их удобнее всего хранить в стеке. Если оператор  $op_2$  встречается справа от оператора  $op_1$ , но имеет более высокий приоритет, тогда  $op_2$  должен быть выведен перед оператором  $op_1$ . Таким образом, операторы выводятся в порядке последним вошел, первым вышел.

Ключевым моментом при написании алгоритма для преобразования является незначительное изменение нашей точки зрения, заключающееся в выяснении, по мере поступления на вход каждой лексемы, для какого из предварительно отложенных операторов (т. е. операторов, хранящихся в стеке) правый операнд завершился в силу появления новой лексемы, что требует передачи его на выход. Приведенные выше условия теперь изменяются:

1. В конце выражения все операторы поступили на выход.
2. Правая скобка требует передачи на выход всех операторов, найденных после соответствующей левой скобки.
3. Оператор с приоритетом, отличным от 6, требует передачи на выход всех остальных операторов большего или равного приоритета.
4. Оператор с приоритетом 6 требует передачи на выход всех остальных операторов строго большего приоритета.

Для реализации второго правила мы будем помещать встреченные нами левые скобки в стек. Тогда, как только встретится парная правая скобка, и операторы будут вытолкнуты из стека, левую скобку также можно удалить.

Теперь мы можем реализовать эти правила в процедуре. Мы будем использовать то же самые вспомогательные типы и процедуры, что и в предыдущем разделе, за исключением того, что функция `Kind(...)` теперь возвращает два дополнительных значения

`leftparen`                      `rightparen`

которые обозначают соответственно левую и правую скобку. Стек теперь будет содержать лексемы (операторы), а не значения. В дополнение к процедуре `GetToken(...)`, получающей следующую лексему из входного потока (инфиксного выражения), мы используем вспомогательную процедуру

`PutToken(t: var R: expression)`

которая помещает данную лексему в постфиксное выражение. Эти две процедуры могут читать и записывать файлы или просто работать с уже созданными ранее списками, как это требуется конкретному приложению. Наконец, мы используем функцию `Priority(op)`, которая возвращает значение приоритета оператора `op`.

С учетом всех этих соглашений мы можем написать процедуру, которая преобразует выражение из инфиксной формы в постфиксную.

**procedure** InfixtoPostfix (**var** infix, postfix: expression); { Инфикс в постфикс }  
 { **Pre:** Параметр infix является допустимым выражением в инфиксной форме; параметр postfix пуст.  
**Post:** Выражение infix преобразовано в постфиксную форму и поступило в параметр postfix. }

**var**

```

S: stack;      { сохраняет лексемы, которые будут обработаны позже }
t,             { обрабатываемая в настоящий момент лексема }
top,
x: token;      { оператор вытолкнут из стека }
endright: Boolean; { достигнут конец правого операнда? }
begin
  CreateStack(S);
  while not ListEmpty(infix.exp) do begin
    GetToken(t, infix);
    case Kind(t) of
      operand: PutToken(t, postfix);
      leftparen: Push(t, S);
      rightparen:
        begin
          Pop(t, S);
          while Kind(t) <> leftparen do begin
            PutToken{ t, postfix};
            Pop(t, S) { отбросим левую скобку }
          end
        end;
    unaryop, binaryop: { Обработываем оба вида операторов одинаково }
    repeat
      if StackEmpty(S) then
        endright := true
      else begin
        StackTop(top, S);
        if Kind(top) = leftparen then
          endright := true
        else if Priority(top) < Priority(t) then
          endright := true
        else if (Priority(top) = Priority(t)) and (Priority(t) = 6) then
          endright := true
        else begin
          endright := false;
          Pop(x, S);
          PutToken(x, postfix)
        end
      end
    until endright;
    Push(t, S)
  end
end
end;
while not StackEmpty(S) do begin
  Pop(t, S);
  PutToken(t, postfix);
end
end
end; { процедура InfixtoPostfix }
```

На рис. 13.6 показаны, в качестве иллюстрации этого алгоритма, шаги, выполняемые в процессе преобразования формулы вычисления корней квадратного уравнения

$$x := (\sim b + b^2 - 4 \times a \times c)^{\frac{1}{2}} / (2 \times a)$$

в постфиксную форму. (Вспомним, что мы используем знак '~' для обозначения одноместного отрицания.)

| <i>Входная лексема</i> | <i>Содержимое стека (самая правая лексема на вершине)</i> | <i>Выходная лексема (лексемы)</i> |
|------------------------|-----------------------------------------------------------|-----------------------------------|
| <i>x</i>               |                                                           | <i>x</i>                          |
| <i>:=</i>              | <i>:=</i>                                                 |                                   |
| <i>(</i>               | <i>:= (</i>                                               |                                   |
| <i>~</i>               | <i>:= ( ~</i>                                             |                                   |
| <i>b</i>               | <i>:= ( ~</i>                                             | <i>b</i>                          |
| <i>+</i>               | <i>:= ( +</i>                                             | <i>~</i>                          |
| <i>(</i>               | <i>:= ( + (</i>                                           |                                   |
| <i>b</i>               | <i>:= ( + (</i>                                           | <i>b</i>                          |
| <i>↑</i>               | <i>:= ( + ( ↑</i>                                         |                                   |
| <i>2</i>               | <i>:= ( + ( ↑</i>                                         | <i>2</i>                          |
| <i>–</i>               | <i>:= ( + ( –</i>                                         | <i>↑</i>                          |
| <i>4</i>               | <i>:= ( + ( –</i>                                         | <i>4</i>                          |
| <i>×</i>               | <i>:= ( + ( – ×</i>                                       |                                   |
| <i>a</i>               | <i>:= ( + ( – ×</i>                                       | <i>a</i>                          |
| <i>×</i>               | <i>:= ( + ( – ×</i>                                       | <i>×</i>                          |
| <i>c</i>               | <i>:= ( + ( – ×</i>                                       | <i>c</i>                          |
| <i>)</i>               | <i>:= ( +</i>                                             | <i>×</i>                          |
| <i>↑</i>               | <i>:= ( + ( ↑</i>                                         | <i>–</i>                          |
| $\frac{1}{2}$          | <i>:= ( + ( ↑</i>                                         | $\frac{1}{2}$                     |
| <i>)</i>               | <i>:=</i>                                                 | <i>↑ +</i>                        |
| <i>/</i>               | <i>:= /</i>                                               |                                   |
| <i>(</i>               | <i>:= / (</i>                                             |                                   |
| <i>2</i>               | <i>:= / (</i>                                             | <i>2</i>                          |
| <i>×</i>               | <i>:= / ( ×</i>                                           |                                   |
| <i>a</i>               | <i>:= / ( ×</i>                                           | <i>a</i>                          |
| <i>)</i>               | <i>:= /</i>                                               | <i>×</i>                          |
| <i>Конец выражения</i> |                                                           | <i>/ :=</i>                       |

**Рис. 13.6.** Преобразование формулы вычисления корней квадратного уравнения в постфиксную форму

На этом можно закончить обсуждение вопроса преобразования в постфиксную форму. Процедура преобразования в префиксную форму будет, очевидно, во многом похожа, однако здесь возникают некоторые сложности, связанные с казалось бы, не имеющим отношения к делу фактом, что в европейских языках мы читаем слева направо. Если преобразовывать выражение в префиксную форму, продвигаясь по нему слева направо, то не только придется переставлять операторы, но и откладывать операнды до вывода их операторов. Однако относительный порядок операндов преобразование не изменяет, поэтому удобной структурой данных для хранения операндов будет не стек, а очередь. Поскольку стеки здесь не годятся, также не годятся и рекурсивные программы без явно выделенной вспомогательной памяти, поскольку эти вычислительные объекты выполняют эквивалентные задачи. Таким образом, преобразование слева направо в префиксную форму требует иного подхода. Подход этот заключается в том, чтобы выполнять преобразование в префиксную форму, продвигаясь по выражению справа налево, используя при этом методы, полностью схожие с преобразованием слева направо в постфиксную форму, которое мы уже рассмотрели. Детали этого преобразования мы оставим для упражнений.

### Упражнения 13.4

---

- E1.** Разработайте алгоритм преобразования выражения из префиксной формы в постфиксную. Используйте при этом соглашения языка Pascal, описанные в этой главе.
- E2.** Напишите на языке Pascal алгоритм преобразования выражения из постфиксной формы в префиксную. Используйте при этом соглашения языка Pascal, описанные в этой главе.
- E3. *Выражение в скобках*** имеет одну из следующих форм:
1. простой операнд;
  2.  $(op E)$ , где  $op$  есть одноместный оператор, а  $E$  — выражение в скобках;
  3.  $(E op F)$ , где  $op$  есть двуместный оператор, а  $E$  и  $F$  представляют собой выражения в скобках.

Таким образом, в выражении в скобках результат любой операции заключается в скобки. Примеры выражений в скобках:  $((a + b) - c)$ ,  $(-a)$ ,  $(a + b)$ ,  $(a + (b + c))$ . Напишите алгоритм на языке Pascal, который будет преобразовывать выражения из (a) префиксной и (b) постфиксной форм в форму со скобками.

- E4.** Напишите процедуру `InfixtoPostfix` в виде рекурсивной процедуры, которая не использует стек или другой массив.

### Программный проект 13.4

---

- P1.** Разработайте управляемую меню демонстрационную программу для польских выражений. На вход программы должны поступать выражения в инфиксной, префиксной или постфиксной формах. Программа затем по указанию пользователя преобразует входное выражение в любую из рассмотренных нами форм (инфиксную со скобками, пре-

фиксную или постфиксную) и выводит результат. В качестве операндов рассматривайте только одиночные буквы или цифры. Допустимые операторы:

|                             |                                                     |
|-----------------------------|-----------------------------------------------------|
| двуместные:                 | $+$ $-$ $*$ $/$ $\uparrow$ $:$ $<$ $>$ $\&$ $ $ $=$ |
| левосторонние одноместные:  | $\#$ $\sim$ $\$$                                    |
| правосторонние одноместные: | $!$ $'$ $"$ $\%$                                    |

В дополнение программа должна допускать использование скобок  $\grave{\text{y}}$  ( $\grave{\text{y}}$  и  $\grave{\text{y}}$ ) только в инфиксных выражениях. Значение некоторых из специальных символов, используемых в этом проекте, таковы:

|                                 |                                 |
|---------------------------------|---------------------------------|
| $\&$ Булева операция <b>and</b> | $ $ Булева операция <b>or</b>   |
| $:$ Присваивание (как в $:=$ )  | $!$ Факториал (справа)          |
| $'$ Производная (справа)        | $"$ Вторая производная (справа) |
| $\%$ Процент (справа)           | $\sim$ Одноместное отрицание    |
| $\#$ Булева операция <b>not</b> | $\$$ Игнорируется программой    |

## 13.5. Интерактивная программа оценки выражений

цель

Можно найти много приложений для программы, которая оценивает функцию, интерактивно вводимую с клавиатуры в процессе выполнения программы. Одним из таких приложений будет программа, рисующая график математической функции. Будем считать, что вы пишете такую программу, которую будут использовать студенты первого года обучения для графического отображения различных функций. Большинство этих студентов еще не знают, как писать или компилировать программы, поэтому вы хотите включить в свою программу средства, позволяющие пользователю вводить выражение для функции вроде

$$x * \log(x) - x \uparrow 1.25$$

в процессе выполнения программы; эта функция затем будет отображаться программой на экране в виде графика для соответствующих значений  $x$ .

Целью этого раздела является описание такой программы и, в частности, завершение разработки двух подпрограмм, необходимых для выполнения нашей задачи. Первая подпрограмма будет принимать в качестве своих входных данных выражение с константами, переменными, арифметическими операторами и стандартными функциями, заключаемыми при необходимости в скобки, по мере того как эти данные вводятся пользователем с клавиатуры терминала. Затем программа преобразует выражение в постфиксную форму и сохранит его в виде списка лексем. Вторая подпрограмма будет вычислять это постфиксное выражение для значений переменных, переданных ей в качестве параметров, и возвращать результат, который затем можно отобразить графически.

причины

Мы занимаемся этим проектом по нескольким причинам. Он показывает нам, как можно воспользоваться рассмотренными уже принципами работы с польскими записями для построения на базе этих принципов законченной, конкретной и правильно функционирующей программы. В этом плане проект иллюстрирует перенос известного метода решения за-



Процедура `ReadExpression` требует от пользователя ввода отображаемого выражения, разделяет его на лексемы и определяет его синтаксическую правильность. Далее выражение преобразуется в постфиксную форму уже рассмотренным нами методом.

Для графического отображения выражения постфиксная форма оценивается для многих различных значений  $x$ , каждое из которых отличается от предыдущего на небольшую величину `increment`, и результаты отображаются в виде последовательности точек на экране. Как это делается — зависит от системы; версия на языке `Turbo Pascal`, приводимая ниже в этой главе, лишь иллюстрирует принципы вывода.

Процедура `ReadNewGraphLimits` позволяет пользователю выбрать диапазон значений  $x$ , для которых требуется вывести график, значение инкремента, а также диапазон, в котором будут отображаться результаты вычисления введенного выражения. Эти значения хранятся в записи `plotdata`.

В дополнение к (независимой) переменной  $x$ , используемой для графического отображения, выражение может содержать другие переменные, которые мы назовем *параметрами* графика. Например, в выражении

$$a * \cos(x) + b * \sin(x)$$

$a$  и  $b$  являются параметрами. Для каждого отображаемого графика параметры остаются неизменными, но их значения могут изменяться при отображении следующего графика, не приводя при этом к каким-либо иным изменениям выражения. Процедура `ReadNewParameters` получает значения всех параметров, которые входят в выражение.

Инициализация в главной программе устанавливает определения для предопределенных лексем (среди прочих, операторов `+`, `-` и `*`, операнда  $x$ , который будет использоваться для отображения, и, возможно, каких-либо констант). Детали инициализации будут зависеть от выбора вами структур данных.

## 13.5.2. Представление данных

Выбор структур данных для нашей программы определяется тем, как мы собираемся хранить и извлекать используемые в польских выражения лексемы и результаты их вычислений. Для каждой отдельной лексемы мы должны хранить:

- имя лексемы (в виде строки символов), чтобы мы могли распознать ее во входном выражении;
- вид лексемы, именно, операнд (`operand`), одноместный оператор (`unaryop`), двуместный оператор (`binaryop`), правосторонний одноместный оператор (вроде факториала `!`), левая скобка (`leftparen`) или правая скобка (`rightparen`);
- для операторов — их приоритет, для операндов — их значение.

Каждую лексему естественно представить в виде записи, хранящей всю эту информацию. Здесь возникает одна небольшая трудность: одна и



та же лексема может встретиться в выражении несколько раз. Если она является операндом, мы должны быть уверены, что каждый раз она принимает одно и то же значение. Если мы помещаем сами записи в выражения, тогда при присвоении значения операнду мы должны быть уверены, что оно обновится во всех записях, соответствующих этому операнду.

словарь

Мы можем избавиться от хранения цепочек ссылок на данное значение, присвоив каждой лексеме целочисленный код и помещая в выражение этот код, а не всю запись. Мы тогда приходим к идее *словаря* для лексем, который будет представлять собой массив, индексируемый целочисленным кодом лексемы, в котором будут храниться полные записи для всех лексем. В этом случае, если *k* есть код переменной, тогда каждое вхождение *k* в выражение заставит нас посмотреть в позицию *k* словаря, где находится соответствующее значение, и мы автоматически получаем каждый раз одно и то же значение.

Помещение в выражение вместо записей целочисленных кодов лексем имеет также преимущество некоторой экономии пространства памяти, однако объем памяти, выделяемый под лексемы, вряд ли будет важным для этого проекта. Более важным может оказаться время, требуемое для оценки постфиксного выражения при различных значениях аргумента *x*.

По мере расшифровки входного выражения программа может находить константы и новые переменные (параметры), которые она должна дополнительно включить в словарь. Все они будут представлять собой операнды, однако вспомните, что в случае параметров пользователь получит запрос на ввод их значений еще перед тем, как выражение будет оценено. Отсюда возникает необходимость хранить еще один список, список кодов лексем, соответствующих параметрам.

Все эти структуры данных показаны на рис. 13.7, вместе с некоторыми структурами данных, используемыми в принципиально важных процедурах.

Объявления включают несколько дополнительных свойств, которые будут детально разъяснены позже. Ниже приведены краткие описания некоторых из них.

- Выражения объявляются как *списки* кодов лексем.
- Преобразование инфиксной формы в постфиксную использует *стек* кодов лексем.
- Вычисление выражений требует наличия *стека* действительных чисел. Чтобы получить возможность в программе на языке Pascal временно иметь два рода стеков, стек вычисленных выражений задокументирован как модуль Turbo Pascal.
- Получив *код* лексемы, словарь предоставляет информацию о лексеме. Однако, при синтаксическом анализе входного выражения мы должны начинать с лексемы *имя* (строки символов) и найти по нему код. Эта задача решается с помощью хеш-таблицы с именем *index*.
- Ошибки различной степени тяжести обрабатываются отдельно.
- Для определения выбора пользователем пунктов меню используется набор символов.

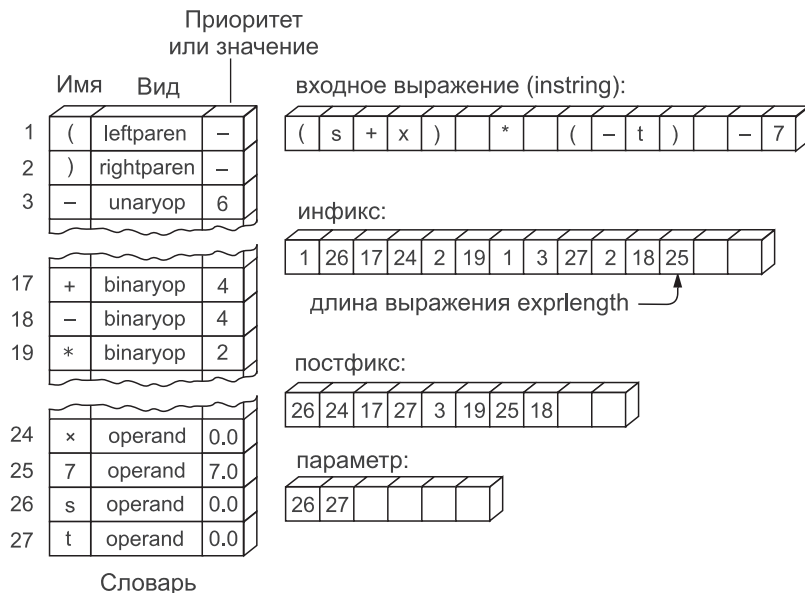


Рис. 13.7. Структуры данных для программы GraphExpression

Ниже приводятся все объявления констант, типов и переменных, необходимые для работы программы, некоторые из которых потребуют дальнейших разъяснений.

```

program GraphExpression (input, output); { Графический вывод выражения }
{ Pre: Предусловия отсутствуют.
  Post: Считывает выражение в инфиксной форме с клавиатуры;
    преобразует его в постфиксную форму; оценивает
    постфиксное выражение и выводит его на экран в виде
    графика в заданном диапазоне. }

uses
  Utility, { откомпилированный модуль, специфичный для Turbo Pascal }
  RealStac,
  Graph; { стандартный графический модуль Turbo Pascal }

const
  firstunary = 3; { где среди лексем находится
                  первый одноместный оператор? }
  lastunary = 16; { последний код одноместного оператора }
  firstbinary = 17; { каков код первого двуместного оператора? }
  lastbinary = 23; { каков код последнего двуместного оператора? }
  firstoperand = 24; { где начинаются операнды? }
  { по договоренности первый операнд представляет собой код для x. }
  lastoperand = 26; { последний предопределенный операнд; пользователь
                    может включить в выражение и другие операнды (параметры) }
  maxlist = 100; { максимальное число лексем в выражении }
  maxstack = 100;
  namelength = 7; { число знаков в имени идентификатора лексемы }
  maxpriority = 6; { максимальный приоритет любого оператора }
  maxtoken = maxlist; { максимальное число различающихся типов лексем }
  maxstring = 200; { максимальный размер входной строки }

```

```

hashsize = 101;           { максимальное число различающихся лексем }
hashmax = 100;

type
tokencode = 0..maxtoken; { код лексемы для ее поиска в таблице словаря }
tokencount = tokencode;
hashaddress = 0..hashsize; { для индекса перехода от лексем к кодам }
hashtable = array [hashaddress] of tokencode;
instring = array [1..maxstring] of char; { выражение, как оно прочитано }
message = string [80];
errortype = (warning, badvalue, badexpr, fatal);
                                   { степень серьезности ошибки }

priorityrange = 1..maxpriority;
tokenname = array [1..namelength] of char;
tokenkind = (operand, unaryop, rightunaryop, binaryop, leftparen, rightparen);
tokendefinition = record
  name: tokenname;
  case kind: tokenkind of
    operand: (value: real);
    unaryop, rightunaryop,
    binaryop: (priority: priorityrange);
    leftparen,
    rightparen: () { пусто }
end;
dictionary = record
  entry : array [tokencode] of tokendefinition;
  count: tokencount;
end;
listentry = tokencode;
stackentry = tokencode;
{ $! list.seg }           { специфично для Turbo Pascal; для списков лексем }
{ $! stack.seg }         { специфично для Turbo Pascal; для трансляции }
type
expression = list;
plotparameter = record
  paramlist: list; { список параметров }
  x, y, { y есть значение выражения, оцененное для аргумента x }
  xlow, xhigh, { границы графика по x }
  ylow, yhigh, { границы графика по y }
  increment: real; { приращение для графика }
end;
commandset = set of char; { для команд меню пользователя }
var
infix, postfix: expression;
plotdata: plotparameter; { параметры для вывода графиков }
lexicon: dictionary; { глобальная просмотревая таблица для всех лексем }

index: hashtable; { задает код для каждой лексемы в словаре }
ch: char; { символ команды }
validset: commandset; { множество допустимых команд }
digit, { множество цифр от 0 до 9 }
upper, { множество символов верхнего регистра клавиатуры }
lower, { множество символов нижнего регистра клавиатуры }
alphabet: set of char; { множество символов верхнего и нижнего регистров клавиатуры }

```

### 13.5.3. Инициализация и вспомогательные задачи.

Задачи инициализации глобальных переменных собраны в одну процедуру, большая часть которой потребует дальнейших разъяснений.

```

procedure Initialize (var infix, postfix: expression; var index: hashtable;
                      var plotdata: plotparameter);
                                                    { Инициализация }

{ Pre:  Предусловия отсутствуют.
  Post: Все параметры инициализированы, как и все глобальные
         переменные.
  Uses: Использует процедуры SetupLexicon, CreateList, CreateHashTable,
         FillHashTable. }

begin                                                    { процедура Initialize }
  validset := ['g', 'h', 'l', n, 'p', 'q', r, 'w'];
  lower := ['a' .. 'z'];
  upper := ['A' .. 'Z'];
  alphabet := lower + upper;
  digit := ['0' .. '9'];
  SetupLexicon;
  CreateList(infix);
  Create List(postfix);
  with plotdata do begin                                { инициализируем параметры
                                                         отображения графика plotdata }

    CreateList(paramlist);
    x := 0;
    y := 0;
    xlow := -10;                                         { значения по умолчанию; могут быть
                                                         изменены пользователем }

    xhigh := 10;
    ylow := -1000;
    yhigh := 1000;
    increment := 0.01;

  end;
  CreateHashTable(index); { инициализируем индекс хеш-таблицы index }
  FillHashTable(index)

end;                                                    { процедура Initialize }

```

#### 1. Предопределенные лексемы

Единственной задачей процедуры SetupLexicon является размещение в словаре всех имен, видов, приоритетов (для операторов) и значений (для операндов) всех предопределенных лексем. Процедура состоит всего лишь из длинной последовательности предложений присваивания, которые мы здесь опустим. Полный список предопределенных лексем, используемых в этой реализации, приведен на рис. 13.8.

Заметьте, что мы можем добавить операторы, которые не являются стандартным средством компьютерного языка (например, логарифм по основанию 2  $\lg$ ), а также константы, например,  $e$  и  $\pi$ . Выражения, которыми мы будем заниматься в этом разделе, всегда имеют в качестве результата действительные числа. Поэтому мы не включили в таблицу лексем какие-либо операции с булевыми значениями или с множествами. Включено, однако, несколько операций над целыми числами, поскольку целые числа вполне могут рассматриваться как частный случай чисел действительных.

| Лексема | Имя    | Вид         | Приоритет/значение        |
|---------|--------|-------------|---------------------------|
| 1       | (      | leftparen   |                           |
| 2       | )      | rightparen  |                           |
| 3       | ~      | unaryop     | 6 отрицание               |
| 4       | abs    | unaryop     | 6                         |
| 5       | sqr    | unaryop     | 6                         |
| 6       | sqrt   | unaryop     | 6                         |
| 7       | exp    | unaryop     | 6                         |
| 8       | ln     | unaryop     | 6 натуральный логарифм    |
| 9       | lg     | unaryop     | 6 логарифм по основанию 2 |
| 10      | sin    | unaryop     | 6                         |
| 11      | cos    | unaryop     | 6                         |
| 12      | arctan | unaryop     | 6                         |
| 13      | round  | unaryop     | 6                         |
| 14      | trunc  | unaryop     | 6                         |
| 15      | !      | right unary | 6 факториал               |
| 16      | %      | right unary | 6 процент                 |
| 17      | +      | binaryop    | 4                         |
| 18      | —      | binaryop    | 4                         |
| 19      | *      | binaryop    | 5                         |
| 20      | /      | binaryop    | 5                         |
| 21      | div    | binaryop    | 5                         |
| 22      | mod    | binaryop    | 5                         |
| 23      | ↑      | binaryop    | 6                         |
| 24      | x      | operand     | 0.00000                   |
| 25      | pi     | operand     | 3.14159                   |
| 26      | e      | operand     | 2.71828                   |

Рис. 13.8. Предопределенные лексемы для GraphExpression

2. Обработка хеш-таблицы

Далее нам нужно определить хеш-таблицу для индексирования. Вспомним, что хотя словарь принимает индекс и возвращает имя лексемы и информацию о ней, хеш-таблица принимает имя лексемы и возвращает ее индекс.

Для занесения индексов в хеш-таблицу, мы сначала должны инициализировать таблицу, сделав ее пустой:

```

procedure CreateHashTable(var index: hashtable); { Создать хеш-таблицу }
{ Pre:  Предусловия отсутствуют.
  Post: Индекс хеш-таблицы создан и инициализирован пустым
         значением. }

var
  a: hashaddress;
begin                                     { процедура CreateHashTable }
  for a := 1 to hashmax do
    index [a] := 0;                       { установим все индексы
                                           вне просмотровой таблицы словаря }
end;                                     { процедура CreateHashTable }

```

Наша следующая задача заключается в переборе всех лексем, определенных в словаре, и во включении каждой из них в хеш-таблицу. Эта задача решается в следующей простой процедуре:

```

procedure FillHashTable (var index: hashtable); { Заполнить хеш-таблицу }
{ Pre:  Уже создана хеш-таблица index.
  Post: Все коды лексем из словаря lexicon помещены в таблицу index.
  Uses: Использует функцию Hash. }

var
  a: hashaddress;
  t: tokencode;
begin                                     { процедура FillHashTable }
  for t := 1 to lastoperand do
    index [Hash(index, lexicon.entry [t].name)] := t
end;                                     { процедура FillHashTable }

```

Часто оказывается, что производительность хеш-таблицы может быть повышена, если принять во внимание особенности приложения, для которого она строится. В нашей графической программе многие лексемы представляют собой отдельные символы (иногда буквы, иногда односимвольные операторы). Поэтому создаваемая нами хеш-таблица придаст особое значение тому факту, что многие лексемы являются одиночными символами:

```

function Hash (var index: hashtable; x: tokenname): hashaddress);
                                                    { Хеш-функция }
{ Pre:  Хеш-таблица уже создана, а x является именем лексемы.
  Post: Функция возвращает местоположение x в таблице index,
        если эта лексема там имеется; в противном случае
        возвращается пустая ячейка, в которую может быть
        включена лексема x. }

var
  a: integer;
  ch: char;
  found: Boolean;
begin                                     { функция Hash }
  ch := x[1] ;
  a := abs(ord(ch)) mod hashsize;
  repeat
    if index [a] = 0 then                   { пустая ячейка }
      found := true

```

```

else if lexicon.entry [index [a]]. name = x then
    found := true                                { лексема x найдена в словаре }
else begin                                     { начнем разрешение конфликтов
                                                до нахождения допустимого значения }

    ch:=x[2];
    a := a + abs(ord(ch));
    if a > hashsize then
        a := a mod hashsize;
        found := false
    end
until found;
    Hash := a
end;                                           { функция Hash }

```

### 3. Обработка ошибок

Чтобы предоставить пользователю более детальную информацию и больше возможностей, стандартную процедуру Error мы дополняем следующим образом:

```

procedure ErrorInform (severity: errortype; m: message);
                                                { Диагностика ошибок }
{ Pre:   Тип severity является допустимым типом кодов ошибок;
          m является сообщением.
Post:   Сообщение m выведено на экран вместе с сообщением
          об уровне ошибки.
Uses:   Использует процедуру Error. Вызов halt является специфичным
          для Turbo Pascal; он останавливает выполнение программы. }
begin                                           { процедура ErrorInform }
    Error(m);
    case severity of
        warning:  writeln('Программа продолжает свою работу. ');
        badvalue: writeln('Недопустимое значение. Укажите новые');
                  write(' пределы для графика. ');
        badexpr:  writeln('Недопустимое выражение. ');
                  write('Введите новое выражение. ');
        fatal:    begin
                    writeln('Фатальная ошибка. Выполнение остановлено. ');
                    halt
                end
    end
end;                                           { procedure ErrorInform }

```

### 4. Выражения: списки лексем

Мы уже решили, что будем рассматривать выражения (и инфиксные, и постфиксные) как списки (целочисленных) кодов лексем. В этом случае мы можем использовать стандартные операции со списками. Процедура GetToken по ходу своего выполнения удаляет коды лексем из своего входного списка.

```

procedure GetToken (var t: tokencode; var E: expression);
                                { Получить лексему }
{ Pre:  Параметр E является допустимым выражением.
  Post: Первый элемент выражения E удален и возвращен
        как параметр t.
  Uses: Использует процедуры DeleteList, ListEmpty, ErrorInform. }
begin                                { процедура GetToken }
  if ListEmpty(E) then
    ErrorInform (badexpr, 'Удаляем код лексемы из пустого выражения')
  else
    DeleteList(1,t, E)
end;                                { процедура GetToken }

```

Аналогично, процедура PutToken включает код лексемы в конец своего списка, а процедура EndExpression проверяет, не пуст ли список.

### 5. Свойства лексем

Ради согласованности с процедурами, разработанными ранее в этой главе, мы будем использовать функцию Kind для определения вида лексемы, соответствующей каждому коду лексемы; эта функция для получения результата всего лишь смотрит в словарь.

Аналогично мы используем функцию Priority для извлечения из словаря приоритета оператора. Обе функции, и Kind, и Priority, мы оставляем для упрощений.

## 13.5.4. Преобразование выражения

В следующей важной секции программы мы должны прочитать выражение в обычной (инфиксной) форме, проверить его на синтаксическую правильность, разделить на лексемы и найти их коды. Будучи представлено в виде последовательности лексем, выражение может быть затем преобразовано в постфиксную форму процедурой, написанной в разделе 13.4, лишь с небольшими изменениями, отображающими объявление настоящей программы.

### 1. Поиск определений лексем

Как только очередная лексема выделяется из входной строки, мы должны найти ее код. Эта задача является выражением проблемы извлечения информации: имя лексемы образует ее ключ, по которому мы должны извлечь целочисленный код. В выражении скорее всего будет не больше нескольких десятков лексем, и в этом случае наилучшим способом извлечения кода является последовательный поиск в словаре. Такой поиск легко запрограммировать, он не требует дополнительных структур данных, а потери времени в сравнении с другими, более изощренными методами, будут пренебрежимо малы.

С другой стороны, одной из целей этого проекта является иллюстрация больших приложений, где затраты на последовательный поиск могут оказаться значительными. В компиляторе, например, может быть много сотен различных символических обозначений, подлежащих распознаванию, и в этом случае следует использовать более сложные таблицы лексем. Неплохим методом является использование хеш-таблиц для преоб-



разования имен лексем в целочисленные коды. Мы назовем хеш-таблицу `index`, поскольку она служит индексным указателем в словаре. В хеш-таблице мы будем сохранять только коды, и использовать их для поиска в словаре всей информации о лексеме с данным именем.

## 2. Расшифровка выражения

На рис. 13.8 можно найти лексемы, имена которых состоят из одного специального символа, и лексемы, представляющие собой слова, начинающиеся с буквы. Эти последние могут принадлежать к одноместным или двуместным операторам или операндам. Возможно также встретиться с лексемами, которые представляют собой числа, т. е. составлены из цифр (возможно с десятичной точкой). Поэтому при разделении входного выражения на лексемы, мы будем рассматривать три случая, используя в главном цикле три процедуры

```
ExtractWord  ExtractNumber  ExtractSymbol
[Извлечь слово  Извлечь число  Извлечь символ]
```

которые будут определять имя лексемы и помещать ее код в выходное выражение (все еще в инфиксной форме на этой стадии).

входной формат

Теперь мы должны установить соглашения, касающиеся входного формата. Будем считать, что входное выражение вводится в виде одной строки, так что, дойдя до символа конца строки, мы также достигнем и конца входного выражения. Мы будем также использовать соглашения языка Pascal относительно использования пробелов: пробелы между лексемами игнорируются, но появление пробела завершает лексему. Если лексема является словом, тогда она начинается с буквы, за которой могут следовать как буквы, так и цифры. Давайте преобразуем все строчные буквы в прописные, и будем использовать в операциях сравнения только прописные (большие) буквы. (Позаботьтесь о том, чтобы предопределенные лексемы попали в хеш таблицу только в виде прописных символов.) Далее, усечем имена до `namelength` символов, где константа `namelength` определена в главной программе.

проверка  
на ошибки

Для того, чтобы сделать нашу программу максимально робастной, следует с особой тщательностью выполнять проверку на ошибки именно на этапе ввода входного выражения, чтобы убедиться, что выражение имеет правильный синтаксис. Проверка на ошибки будет выполняться во вспомогательной процедуре.

Со всеми этими требованиями на ввод выражения мы получаем следующую процедуру.

```
procedure ReadExpression (var infix; expression; var index: hashtable;
                        var plotdata: plotparameter);
                                { Прочитать выражение }
{ Pre:   Хеш-таблица index уже создана и инициализирована
        предопределенными лексемами, найденными в словаре.
Post:  Выражение infix прочитано, и лексемы, не найденные в словаре
        и в хеш-таблице index, включены в хеш-таблицу.
Uses:  Использует процедуры и функции CreateList, InsertList, ListFull,
        ListSize, ExtractWord, ExtractNumber, ExtractSymbol, PutToken,
        Leading, DeleteList, Hash, ValidInfix, ClearList. }
```

```

var
  valid: Boolean;
  inputstring: instring;
  stringsize: integer;
  i: integer;
  ch: char;
  t: tokencode;
  symbol: Boolean;
  x: tokenname;
begin
  { процедура ReadExpression }
  repeat
    CreateList(infix);
    writeln('Введите выражение одной строкой, затем нажмите [RETURN].');
    stringsize := 0;
    while (stringsize < maxstring) and (not eoln) do begin
      stringsize := stringsize + 1;
      read(inputstring [stringsize]);
    end;
    symbol := false;
    valid := eoln;
    readln;
    i := 1;
    { просмотр строки и извлечение лексем }
    while (i <= stringsize) and (inputstring [i] = ' ') do i := i + 1;
    while (i <= stringsize) and (not ListFull(infix)) and valid do begin
      if inputstring [i] in alphabet then
        ExtractWord(inputstring, stringsize, i, valid, t, index, plotdata)
      else if inputstring [i] in (digit + ['.']) then
        ExtractNumber(inputstring, stringsize, i, valid, t, index)
      else begin
        ExtractSymbol(inputstring, stringsize, i, valid, t, index);
        symbol := true
      end;
      if valid then PutToken(t, infix);
      if valid and symbol then begin { проверка на одноместные + или - }
        symbol := false;
        if Leading(infix, ListSize(infix)) then
          if (lexicon.entry [t].name = '+' or '-') then
            Deletelst(ListSize(infix), t, infix)
          else if (lexicon.entry [t].name = '-' or '+') then begin
            DeleteList(Listsize(infix), t, infix);
            x := lexicon.entry [t].name;
            x[1] := '~';
            t := index[Hash(index, x)];
            PutToken(t, infix)
          end
        end;
        while (i <= stringsize) and (inputstring [i] = ' ') do i := i + 1
      end;
      if valid then
        valid := ValidInfix(infix);
        { проверка правильности выражения }
      if not valid then begin
        writeln('Выражение недопустимо или слишком длинное. ');
        write(' Попробуйте снова. ');
        ClearList(infix)
      end
    until valid
  end;
  { процедура ReadExpression }
end;

```

### 3. Ведущие и ведомые позиции

Наиболее важным аспектом проверки на ошибки, с которым мы впервые сталкиваемся в этой процедуре, является функция `Leading`, возвращающая булево значение. Чтобы объяснить роль этой процедуры, рассмотрим сначала частный случай. Пусть наше выражение состоит только из простых операндов и двуместных операторов, т. е. в нем нет ни скобок, ни одноместных операторов. Тогда единственно синтаксически правильной формой выражения будет такая:

*operand binaryop operand binaryop ... operand*

где первая и последняя лексемы представляют собой операнды, и при этом два допустимых вида лексем чередуются. Понятно, что два операнда не могут идти друг за другом, как и два оператора. В начальной позиции должен быть операнд, и операнд должен указываться после каждого оператора, так что мы можем рассматривать эти позиции как “ведущие”, поскольку предшествующий оператор должен приводить к наличию операнда.

Теперь предположим, что в наше предложение должны быть включены одноместные операторы. Перед любым операндом может быть помещено любое число левосторонних операторов, однако недопустимо размещать левосторонние одноместные операторы непосредственно перед двусторонним оператором. Другими словами, одноместные операторы, указываемые слева от операндов, могут размещаться в точности в тех местах, где допустимы операнды, т. е. в ведущих позициях, и нигде больше. С другой стороны, появление левостороннего одноместного оператора по-прежнему оставляет эту позицию «ведущей», так как в ней должен обязательно появиться операнд, и лишь после него будет допустим двуместный оператор.

ведущие позиции

Правосторонние одноместные операторы, наоборот, могут размещаться в любой ведомой позиции, т.е. после операндов или других правосторонних одноместных операторов, и появление правостороннего одноместного оператора оставляет позицию ведомой, поскольку в ней должен обязательно появиться двусторонний оператор, и лишь после него будет допустим операнд.

Теперь, наконец, допустим использование скобок в нашем выражении. Подвыражение, заключенное в скобки, рассматривается как операнд и, следовательно, может появиться в точности там, где допустимы операнды. Отсюда левая скобка может появиться в точности в ведущей позиции, причем она оставляет эту позицию ведущей, а правая скобка может появиться только в ведомой позиции, так же оставляя эту позицию ведомой.

Все рассмотренные возможности подытожены на рис. 13.9.

Рассмотренные требования учтены в приводимой ниже функции, которая будет использоваться во вспомогательных процедурах для проверки синтаксиса входных данных.

| <i>Предыдущая лексема<br/>любого вида из следующих:</i> | <i>Допустимая лексема<br/>любого вида из следующих:</i> |
|---------------------------------------------------------|---------------------------------------------------------|
| <i>Ведущая позиция</i>                                  |                                                         |
| начало выражения                                        | операнд                                                 |
| двуместный оператор                                     | одноместный оператор                                    |
| одноместный оператор                                    | левая скобка                                            |
| левая скобка                                            |                                                         |
| <i>Ведомая позиция</i>                                  |                                                         |
| операнд                                                 | двуместный оператор                                     |
| правосторонний<br>одноместный оператор                  | правосторонний<br>одноместный оператор                  |
| правая скобка                                           | правая скобка                                           |
|                                                         | конец выражения                                         |

**Рис. 13.9.** Лексемы, допустимые в ведущих и ведомых позициях

```

function Leading (infix: expression; current: integer): Boolean;
                                                    { Ведущая позиция }
{ Pre:  Выражение infix уже инициализировано.
  Post: Функция возвращает true, если лексема перед текущей
          позицией current выражения infix делает эту позицию ведущей,
          и false в противном случае.
  Uses: Использует процедуры Kind, RetrieveList. }
var
  t: tokencode;
begin
  if current = 1 then
    Leading := true
  else
    begin
      RetrieveList(current - 1, t, infix);
      Leading := (Kind(t) in [leftparen, unaryop, binaryop])
    end
  end;
                                                    { функция Leading }

```

#### 4. Одноместные и двуместные операторы

Процедура ReadExpression содержит секцию кода, которая требует пояснений. Речь идет о двух символах, '+' и '-', которые могут выступать в качестве как одноместных, так и двуместных операторов. К счастью, функция Leading сообщит нам, какой из этих двух случаев имеет место, поскольку только одноместный оператор допустим в ведущей позиции. Для одноместного оператора '+' мы не предусматриваем каких-либо действий, так как он не оказывает никакого эффекта, а одноместный '-' мы заменяем нашим обозначением '~'. Заметьте, однако, что это изменение годится только для нашей программы. Пользователь не обязан, более того, не имеет права использовать символ '~' для обозначения одноместного отрицания.

## 5. Проверка на синтаксические ошибки

Задачей следующей функции является проверка того, что в ведущих и ведомых позициях находятся только допустимые лексемы, а скобки должным образом сбалансированы.

```

function ValidInfix (infix: expression): Boolean;
    { Допустимое инфиксное выражение }
{ Pre:  Выражение infix уже инициализировано.
  Post: Возвращает true, если выражение имеет допустимую
        инфиксную форму, и false в противном случае.
  Uses: Использует процедуры и функции Kind, Leading, ListSize. }
var
    t: tokencode;
    k: tokenkind;
    OK: Boolean;
    i,
    parencount: integer;
    { вид обрабатываемой лексемы }
    { до сих пор выражение OK? }
    { переменная управления циклом }
    { подсчет скобок для проверки
      их сбалансированности }
    { функция ValidInfix }
begin
    OK := true;
    parencount := 0;
    i:=1;
    while (i <= Listsize(infix) ) and OK do
    begin
        RetrieveList(i, t, infix);
        k := Kind(t);
        if Leading(infix, i) then
            OK := k in [operand, unaryop, leftparen]
        else
            OK := k in [binaryop, rightparen, rightunaryop];
        if OK then
            if k = leftparen then
                parencount := parencount + 1
            else if k = rightparen then
                begin
                    parencount := parencount - 1;
                    OK := parencount >= 0
                end;
                i := i + 1
            end;
        ValidInfix := OK
    end;
end;
    { функция ValidInfix }

```

## 6. Случай: лексема представляет собой слово

Теперь мы обратимся к трем вспомогательным процедурам для обработки слов, чисел и специальных символов. Первая из них должна реализовывать решения относительно структуры слов, принятые нами ранее в этом разделе. Лексема-слово может быть операндом, одноместным оператором или двуместным оператором. Проверку на ошибки следует скорректировать соответственно этому условию. Однако может быть и так, что лексема-слово еще не вошла в словарь; в этом случае она представляет

событием первое появление нового параметра и, соответственно, должна быть включена в словарь и в список параметров.

Все эти требования отражаются в двух следующих процедурах. Первая из них извлекает целое слово из входной строки.

```

procedure GetWord (var inputstring: instring; size: integer; { Получить слово
}
                                var i: integer; var word: tokenname);
{ Pre : Строка inputstring уже инициализирована и size есть ее длина.
  Параметр i указывает на начало извлекаемого слова.
  Post : Слово word извлечено, а i указывает на следующий символ в
  string непосредственно за извлеченным словом.
  Uses: Использует процедуру Uppercase. }
var
  wordsize: integer;
  j: integer;
begin                                     { процедура GetWord }
  wordsize := 0;
  while (inputstring [i] in (alphabet + digit)) and
    (wordsize < namelength) and (i <= size) do
    begin
      wordsize := wordsize + 1;
      word [wordsize] := inputstring [i];
      Uppercase(word [wordsize]);
      i := i + 1
    end;
  if i <= size then
    if inputstring [i] in (alphabet + digit) then
      begin
        write('Предупреждение: имя ');
        for j := 1 to namelength do
          write(word [j]);
        while (i <= size) and (inputstring [i] in (alphabet + digit)) do
          begin
            write(inputstring [i]);
            i := i + 1;
          end;
        writeln(' было усечено до ', namelength: 2, ' символов')
      end;
    while wordsize < namelength do
      begin                                     { заполняем пробелами }
        wordsize := wordsize + 1;
        word [wordsize] := ' '
      end
    end
  end;                                     { процедура GetWord }

```

Вторая процедура включает слово в словарь и в хеш-таблицу.

```

procedure ExtractWord (var inputstring: instring; { Извлечь слово }
  stringsize: integer; var i: integer;
  var valid: Boolean; var t: tokencode;
  var index: hashtable; var plotdata: plotparameter);
(Pre: Строка inputstring уже инициализирована, и stringsize есть ее длина.
  i есть позиция в строке, начинающая обрабатываемое слово.

```

**Post:** Слово прочитано из строки `inputstring`, а `i` указывает на следующий символ в `inputstring` непосредственно за прочитанным словом. Слово включено в словарь, а в хеш-таблице `index` содержится местоположение лексемы в словаре.

**Uses:** Использует процедуры и функции `GetWord`, `Hash`, `ErrorInform`, `ListFull`, `InsertList`. }

```
var word: tokenname;
begin
    GetWord (inputstring, stringsize, i, word);
    t := index [Hash(index, word)];
    if t = 0 then { нет в словаре: определим новую лексему }
    if lexicon.count >= maxtoken then
        ErrorInform(badexpr, 'Слишком много отдельных переменных и констант.')
    else with lexicon do begin
        count := count + 1;
        index[Hash(index, word)] := count;
        t := count;
        entry [count].name := word;
        entry [count].kind := operand;
        entry [count].value := 0.0;
        if ListFull(plotdata.paramlist) then
            ErrorInform(badexpr, 'Выражение содержит слишком много параметров.')
        else
            InsertList(Listsize(plotdata.paramlist) + 1, t, plotdata.paramlist);
        end
    end;
end; { процедура ExtractWord }
```

## 7. Случай: лексема представляет собой число

Обработка чисел в общем выполняется так же, как и обработка переменных, однако имеются два отличия. Одно из них заключается в том, что мы должны преобразовать число в двоичную форму, чтобы его значение можно было получить из словаря и использовать непосредственно, а не вводить его в список параметров. Другое отличие возникает из-за того, что для числа не обязательно существует уникальное имя. Если значение `namelength` достаточно велико, чтобы строка могла вместить в себя все цифры машинного диапазона, тогда числу может быть назначено уникальное имя, но если это не так, два числа могут получить одно и то же имя. Чтобы защититься от такой ситуации, мы будем рассматривать каждое вхождение числа как заново определенную лексему, и действовать затем соответствующим образом. Мы в действительности назначаем числу имя, но исключительно в целях отладки, и даже не включаем это имя в хеш-таблицу.

Преобразуя число в двоичную форму, мы для простоты отказываемся от стандартного обозначения с десятичной точкой и рассматриваем целую и дробную части отдельно. Для преобразования строки цифр в целое (запоминаемое как действительное число), мы используем вспомогательную функцию. Эта функция, приведенная здесь после процедуры, требует, как это и положено в языке Pascal, чтобы числовая строка начиналась с цифры.

```

procedure ExtractNumber (var inputstring: instring; { Извлечь число }
    stringsize: integer; var i: integer;
    var valid: Boolean; var t: tokencode;
    var index: hashtable);
{ Pre: Строка inputstring уже инициализирована, и stringsize есть
ее длина. i есть позиция в строке, начинающая
обрабатываемое число.
Post: Число прочитано из строки inputstring, а i указывает
на следующий символ в inputstring непосредственно за концом
прочитанного числа. Число включено в словарь,
а в хеш-таблице index содержится местоположение лексемы
в словаре.
Uses: Использует процедуры ErrorInform, ConvertToReal. }
var newnumber: instring;
    size: integer;
begin { процедура ExtractNumber }
  if lexicon.count >= maxtoken then begin
    ErrorInform(badexpr, 'Слишком много констант и переменных в словаре');
    valid := false; size:=1; newnumber[1] := '0';
  end
  else begin
    size := 0;
    while (inputstring[i] in digit) and (i <= stringsize) do begin
      size := size + 1; { получим цифры перед десятичной точкой }
      newnumber[size] := inputstring[i];
      i := i + 1
    end;
    if inputstring[i] = '.' then begin
      size :=size + 1; { извлечем десятичную точку }
      newnumber[size] := inputstring[i];
      i := i + 1;
      while (inputstring[i] in digit) and (i <= stringsize) do begin
        size := size + 1;
        newnumber[size] := inputstring[i];
        i := i + 1 ;
      end
    end
    else if inputstring[i] in ['E', 'e'] then begin
      ErrorInform(badexpr, 'Извините, обозначения E и e не допустимы');
      valid := false
    end;
    with lexicon do begin { включим константу в словарь }
      count := count + 1;
      entry[count].name := 'const ';
      entry[count].kind := operand;
      entry[count].value := ConvertToReal(newnumber, size);
      t := lexicon.count;
    end
  end { допустимый случай }
end; { процедура ExtractNumber }

```



```

function ConvertToReal (var number: instring; size: integer): real;
    { Преобразовать в действительное число }
{ Pre:   Символьная строка number содержит символы, представляющие
          действительное число, а параметр size содержит длину строки
          number. Число должно начинаться с цифры и может содержать
          или не содержать десятичную точку.
Post:   Символьная строка number преобразована в действительное
          число.
Uses:   Использует процедуры CharToDigit, ErrorInform. }
var
    i, j; integer;
    leftsum, { сумма цифровых компонентов слева от десятичной точки }
    rightsum: real; { сумма цифровых компонентов справа
                     от десятичной точки }
begin { функция ConvertToReal }
    if not (number[1] in digit) then begin
        ErrorInform(badexpr, 'Первый символ не может быть десятичной точкой. ');
        ConvertToReal := 0.0;
    end
    else
    begin
        leftsum := 0.0;
        i:=1;
        while (i <= size) and (number[i] in digit) do
            begin
                leftsum := (leftsum * 10.0) + CharToDigit(number[i]);
                i := i + 1
            end;
        rightsum := 0.0;
        if (i <= size) and (number[i] = '.') then
            begin
                i := i + 1;
                j := size;
                while (j >= i) and (number[j] in digit) do
                    begin
                        rightsum := (rightsum + CharToDigit(number[j])) * 0.1;
                        j := j - 1
                    end
                end
            end
            else if (i <= size) and (number[i] <> '.') then
                ErrorInform(fatal, 'При преобразовании действительного числа
                                   встретилаcь не цифра');
                ConvertToReal := rightsum + leftsum
            end
        end;
    end; { функция ConvertToReal }

```

## 8. Случай: лексема представляет собой специальный символ

Третья вспомогательная процедура обрабатывает специальные символы. Большая часть выполняемой ею работы проще, чем в предыдущих случаях; так как ей не требуется создавать новые лексемы. Если процедура не может распознать символ, возникает ошибка. Все специальные символы имеют длину 1, что облегчает отсчет позиций в instring.

```

procedure ExtractSymbol (inputstring: instring;           { Извлечь символ }
                        stringsize: integer; var i: integer;
                        var valid: Boolean; var t: tokencode;
                        var index: hashtable);
{ Pre:  Параметр inputstring является допустимой символьной строкой,
        a stringsize есть ее длина.
  Post: Параметр i указывает на следующий символ за только что
        обработанным символом; t является кодом лексемы этого
        символа.
  Uses: Использует процедуру ErrorInform, функцию Hash. }
var
  j: integer; x: tokename;
begin                                     { процедура Extract Symbol }
  if i > stringsize then begin
    ErrorInform(fatal, 'Индекс i вышел за допустимые пределы в ExtractSymbol. ');
    valid := false;
  end
  else begin
    x[1] := inputstring [i];
    i := i + 1;
    for j := 2 to namelength do           { заполним символами пробела }
      x[j] := ' ';
    t := index[Hash(index, x)];
    if t = 0 then begin
      ErrorInform(badexpr, 'Нераспознанный символ в выражении');
      valid := false;
    end
  end
end;                                     { procedure ExtractSymbol }

```

## 9. Преобразование в постфиксную форму

После завершения процедуры ReadExpression входное выражение оказывается преобразованным в инфиксную последовательность лексем, т. е. в точности в ту форму, которая требуется для процедуры InfixtoPostfix из раздела 13.4. Фактически мы подошли к ключевому шагу нашего алгоритма и можем теперь воспользоваться предыдущей работой без существенных изменений; единственная модификация процедуры InfixtoPostfix заключается в согласовании объявлений с теми, что даны в этом разделе.

После завершения процедуры InfixtoPostfix входное выражение представляет собой последовательность лексем в постфиксной форме, и, следовательно, может быть эффективно оценено на следующем шаге. Эффективность здесь имеет значение, так как она позволяет выводить график без заметных задержек, несмотря на то, что оценка выражения выполняется для большого числа различных значений.

### 13.5.5. Оценка выражения

#### 1. Чтение параметров

Первым шагом в оценке выражения является установка значений параметров, если таковые имеются. Это действие выполняется для каждого графика только однажды, при условии, что пользователь не желает использовать для всех параметров значения по умолчанию 0.0. Процедуру ReadNewParameters мы оставим для упражнений.



Заметьте, что мы можем с легкостью использовать структуру этой функции для улучшения проверки на ошибки, выполняемой операционной системой. Сообщение, выводимое в результате деления на 0, будет скорее всего более содержательным, чем стандартное сообщение вроде

Floating point error  
[Ошибка вычислений с плавающей точкой]

которым ограничивается информация, предоставляемая обычно операционной системой.

Поскольку в языке Pascal отсутствует оператор возведения в степень, мы можем также воспользоваться возможностью написать соответствующую функцию. В ней мы реализуем для двух случаев входных данных два различных метода: если степень является целым числом (или очень близка к целому), мы для вычисления степени используем цикл и умножение или деление; в противном случае мы прибегаем к логарифмам.

```
function Exponent (x, y: real): real;                                { Степень }
{ Pre:  Параметры x и y являются допустимыми операндами-значениями.
  Post: Параметр x возведен в степень y.
  Uses: Использует процедуру ErrorInform. }
const epsilon = 0.000001;
var i: integer;
    p: real;
begin
    if abs(y - round(y)) < epsilon then begin                        { функция Exponent }
                                                                    { будем рассматривать y
                                                                    как целое число }
        p := 1.0;
        if y > 0.0 then
            for i := 1 to round(y) do
                p := p * x;
        else if x = 0.0 then
            ErrorInform(badvalue, 'Отрицательная степень числа 0.0')
        else
            for i := -1 downto round(y) do
                p := p/x;
        Exponent := p;
    end
    else if x > 0.0 then                                             { используем логарифм и экспоненту }
        Exponent := exp(y * ln(x))
    else if abs(x) < epsilon then
        Exponent := 0.0
    else
        ErrorInform(badvalue, 'Отрицательное число возводится в нецелую степень')
    end;                                                            { функция Exponent }
```

### 13.5.6. Графическое отображение выражения

Теперь, наконец, мы подошли к цели всей нашей программы, именно, к выводу выражения на экран компьютера в графической форме. Графика, к сожалению, целиком основывается на системных средствах, поэтому программа, работающая на одной машине, совсем не обязательно будет работать на другой. В этом разделе мы обсудим требования системы Turbo Pascal.

При необходимости вывести на экран график следует использовать модуль Graph системы Turbo Pascal. Программа затем должна решить, какие графические средства имеются в наличии, инициализировать их в графическом режиме и определить размер экрана в пикселях. Эти действия выполняются в Turbo Pascal следующим образом:

```
procedure InitializeScreen;
    { Инициализировать экран; специфично для Turbo Pascal }
{ Pre: Предусловия отсутствуют.
Post: Экран переведен в графический режим, максимальные размеры
экрана занесены в переменные maxX и maxY.
Uses: Использует процедуры модуля Graph GetMaxX, GetMaxY,
InitGraph, GraphErrorMsg. }
begin
    GraphDriver := Detect;
    InitGraph(GraphDriver, GraphMode, ' ')
    ErrorCode := GraphResult;
    if ErrorCode = grOK then begin
        writeln('Возникла ошибка графики: ', GraphErrorMsg(ErrorCode));
        halt
    end;
    maxX := GetMaxX;
    maxY := GetMaxY;
end;
    { процедура InitializeScreen }
```

После того, как график выведен на экран, следует восстановить текстовый режим:

```
procedure RestoreScreen;
    { Восстановить экран; специфично для Turbo Pascal }
{ Pre: Экран находится в графическом режиме.
Post: Восстановлен текстовый режим экрана.
Uses: Использует процедуру модуля Graph RestoreCrtMode. }
begin
    RestoreCrtMode
    { Очистить экран и перевести курсор в низ экрана }
end;
    { процедура RestoreScreen }
```

Используя размер экрана в пикселях, мы можем преобразовать каждую точку со значениями  $(x, y)$  в указанном пользователем диапазоне в координаты пикселя и вывести ее на экран:

```
procedure GraphPoint (x, y: real;
    var plotdata: plotparameter);
    { Вывести точку }
{ Pre: Значения maxX и maxY уже вычислены. Параметры x и y
являются допустимыми действительными числами и находятся
в пределах диапазона, заданного пользователем.
Post: Пиксель в точке x, y был отображен на экране относительно
значений maxX и maxY. }
var ix, iy: integer;
begin
    with plotdata do begin
        ix := round((x - xlow) * maxX/(xhigh - xlow));
        iy := maxY - round((y - ylow) * maxY/(yhigh - ylow));
        PutPixel(ix, iy, 1)
    end
end;
    { процедура GraphPoint }
```

Эта процедура активизируется другой процедурой, которая рисует график, повторно выполняя оценку выражения для последовательных значений  $x$ , отличающихся друг от друга на величину `increment`, и для каждой оценки выводя на экран единственную точку:

```

procedure DrawGraph (var postfix: expression;           { Вывести график }
                     var plotdata: plotparameter);
{ Pre:  Выражение postfix уже инициализировано. Параметры
        отображения графика уже инициализированы.
  Post: Выражение postfix выведено на экран в графической форме
        с использованием параметров plotdata.
  Uses: Использует процедуру EvaluatePostFix; модуль Graph }
var x: real;
begin                                     { процедура DrawGraph }
  InitializeScreen;
  with plotdata do begin
    x := xlow;
    repeat
      lexicon.entry [firstoperand].value := x;
    { По соглашению  $x$  всегда является первым в списке операндов }
      EvaluatePostfix(postfix, y);
      if (x >= xlow) and (x <= xhigh) and
        (y >= ylow) and (y <= yhigh) then
        GraphPoint(x, y, plotdata);
      x := x + increment
    until x > xhigh;
  end;
  readln;                                { пауза; подождем, пока пользователь нажмет Return,
                                          чтобы стереть график }

  Restore Screen
end;                                    { процедура DrawGraph }

```

Мы, наконец, полностью описали весь проект. Неразработанными остались несколько подпрограмм, но все они достаточно просты и могут быть оставлены для упражнений.

### Упражнения 13.5

- E1.** Укажите детально, какие изменения следует внести в программу, чтобы добавить в качестве дополнительного одноместного оператора `log( )` вычисление логарифмической функции по основанию 10.
- E2.** Простодушные пользователи этой программы могут (если графически отображаются какие-то денежные расчеты) написать внутри выражения знак доллара '\$'. Что в этом случае отобразит программа? Какие изменения следует внести, чтобы программа игнорировала знак '\$'?
- E3.** Программисты на языках C и Pascal могут случайно ввести ';' в конце строки. Какие изменения следует внести в программу, чтобы точка с запятой в конце выражения игнорировалась, но в любом месте внутри выражения приводила к ошибке?
- E4.** Объясните, какие изменения следует внести в программу, чтобы она принимала, наряду с круглыми скобками ( ... ), еще и квадратные [ ... ], и фигурные { ... }? Гнездование должно выполняться с помощью скобок одного вида, т. е. выражение в форме ( ... [ ... ] ... ) недопустимо, но в форме вроде [ ...( ...) ... { ... } ... ] вполне возможно.

### Программный проект 13.5

P1. Составьте недостающие подпрограммы и реализуйте программу GraphExpression на своем компьютере.

## Литература для дальнейшего изучения

Польская нотация столь естественна и полезна, что вы могли бы ожидать ее изобретения сотни лет назад. Удивительно, что она была предложена только в прошлом столетии:

Jan Lukasiewicz, *Elementy Logiki Matematycznej*, Warsaw, 1929; English translation: *Elements of Mathematical Logic*, Pergamon Press, 1963.

Рассмотрение итеративных алгоритмов для формирования и оценки польских выражений (обычно в постфиксной форме) можно найти в нескольких книгах по структурам данных, а также и в более серьезных книгах по теории компиляторов. Итеративный алгоритм для преобразования выражений из инфиксной формы в постфиксную был независимо предложен Е. У. Дейкстра и К. Л. Хамлином и опубликован в статьях

E. W. Dijkstra, «Making a Translator for ALGOL 60», *Automatic Programming Information* number 7 (May 1961); reprinted in *Annual Review of Automatic Programming* 3 (1963), 347–356.

C. L. Hamblin, «Translation to and from Polish notation», *Computer Journal* 5 (1962), 210–213.

Рекурсивный алгоритм для оценки постфиксных выражений был предложен, хотя и с несколько иной точки зрения и только для двуместных операторов, в статье

Edward M. Reingold, «A comment on the evaluation of Polish postfix expressions», *Computer Journal* 24 (1981), 288.

# Приложение А

## Математические методы

В первой части этого приложения представлено несколько математических результатов, используемых при анализе алгоритмов. Последние два раздела (числа Фибоначчи и Каталана) составляют факультативный материал, предназначенный для читателей с математическим уклоном.

### А.1. Суммы степеней целых чисел

Следующие две формулы полезны при подсчете числа шагов, выполняемых алгоритмом.

Теорема А.1

$$1 + 2 + \dots + n = \frac{n(n+1)}{2}.$$
$$1^2 + 2^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}.$$

Доказательство

Первое равенство имеет простое и элегантное доказательство. Мы полагаем, что  $S$  равна сумме с левой стороны, записываем ее дважды (в прямом и в обратном направлении) и складываем вертикально:

$$\begin{array}{cccccccccccc} 1 & + & 2 & + & 3 & + & \dots & + & n+1 & + & n & = & S \\ \frac{n}{n+1} & + & \frac{n-1}{n+1} & + & \frac{n-2}{n+1} & + & \dots & + & \frac{2}{n+1} & + & \frac{1}{n+1} & = & \frac{S}{2S} \end{array}$$

С левой стороны имеется  $n$  столбцов; отсюда  $n(n+1) = 2S$ , и наша формула справедлива.

Первое равенство можно также доказать графически, как это показано на рис. А.1.

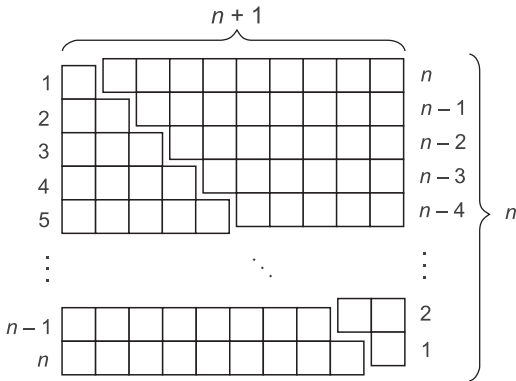


Рис. А.1. Геометрическое доказательство формулы для суммы целых чисел



доказательство  
по индукции

Для доказательства второго равенства мы воспользуемся методом *математической индукции*. Этот метод требует, чтобы мы начали с определения начального случая, называемого *индуктивной базой*, которым для нашей формулы является случай  $n = 1$ . В этом случае формула превращается в следующую:

$$1^2 = \frac{1(1+1)(2+1)}{6}.$$

Это равенство справедливо, т. е. мы установили индуктивную базу. Далее, используя формулу для случая  $n - 1$ , мы должны установить ее справедливость для случая  $n$ . Для случая  $n - 1$  мы имеем

$$1^2 + 2^2 + \dots + (n-1)^2 = \frac{(n-1)n(2(n-1)+1)}{6}.$$

Отсюда следует, что

$$\begin{aligned} 1^2 + 2^2 + \dots + (n-1)^2 + n^2 &= \frac{(n-1)n(2(n-1)+1)}{6} + n^2 \\ &= \frac{2n^3 - 3n^2 + n + 6n^2}{6} \\ &= \frac{n(n+1)(2n+1)}{6} \end{aligned}$$

что и является требуемым результатом; таким образом, доказательство по индукции завершено.

Удобным кратким обозначением суммы того вида, что фигурирует в приведенных равенствах, является прописная греческая буква сигма

$\Sigma$

указываемая перед слагаемым; начальное значение индекса, управляющего суммированием, пишется под знаком суммы, а конечное значение над ним. Тогда наши равенства можно записать таким образом:

$$\sum_{k=1}^n k = \frac{n(n+1)}{2}$$

и

$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}.$$

Полезны также две другие формулы, особенно при исследовании деревьев.

#### Теорема А.2

$$\begin{aligned} 1 + 2 + 4 + \dots + 2^{m-1} &= 2^m - 1. \\ 1 \times 1 + 2 \times 2 + 3 \times 4 + \dots + m \times 2^{m-1} &= (m-1) \times 2^m + 1. \end{aligned}$$

При использовании обозначения суммы эти равенства запишутся таким образом:

$$\sum_{k=0}^{m-1} 2^k = 2^m - 1.$$

$$\sum_{k=0}^m k \times 2^{k-1} = (m-1) \times 2^m + 1.$$

обозначение  
суммы

### Доказательство

Первую формулу мы докажем в более общей форме. Мы начнем со следующего равенства, которое для любого значения  $x \neq 1$  может быть проверено просто умножением обеих сторон на  $x - 1$ :

$$\frac{x^m - 1}{x - 1} = 1 + x + x^2 + \dots + x^{m-1}$$

для любого  $x \neq 1$ . При  $x = 2$  это выражение становится нашим первым равенством.

Для доказательства второй формулы мы возьмем то же выражение для случая  $m + 1$  вместо  $m$ :

$$\frac{x^{m+1} - 1}{x - 1} = 1 + x + x^2 + \dots + x^m$$

для любого  $x \neq 1$ , и продифференцируем его по  $x$ :

$$\frac{(x - 1)(m + 1)x^m - (x^{m+1} - 1)}{(x - 1)^2} = 1 + 2x + 3x^2 + \dots + mx^{m-1}$$

конец  
доказательства

для любого  $x \neq 1$ . При  $x = 2$  мы получим вторую формулу.

Предположим, что в предыдущих формулах  $|x| < 1$ . По мере увеличения  $m$   $x^m$  становится все меньше, так что

$$\lim_{m \rightarrow \infty} x^m = 0.$$

Перейдя к пределу при  $m \rightarrow \infty$ , получим следующие варианты наших равенств:

### Теорема А.3

Если  $|x| < 1$ ,

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}.$$

$$\sum_{k=1}^{\infty} kx^{k-1} = \frac{1}{(1 - x)^2}.$$

## А.2. Логарифмы

Основная цель использования логарифмов заключается в переходе от умножения и деления к сложению и вычитанию, а от возведения в степень к умножению. Перед изобретением карманных калькуляторов логарифмы были незаменимым средством ручных вычислений; вспомните большие таблицы логарифмов и в свое время вездесущие логарифмические линейки. Хотя сегодня мы имеем другие средства для числовых расчетов, фундаментальные свойства логарифмов придают им значение, далеко выходящее за рамки их использования в качестве средства, облегчающего вычисления.

Во-первых, поведение многих явлений обнаруживает внутреннюю логарифмическую структуру; таким образом, используя логарифмы, мы обнаруживаем важные взаимосвязи, которые в противном случае остаются незамеченными. Например, измерение громкости звука выполняется логарифмически: если один звук на 10 дБ (децибел) громче другого, тогда его фактическая звуковая энергия больше в 10 раз. Если звуковой

физические  
измерения

уровень в одной комнате составляет 40 дБ, а в другой — 60 дБ, тогда по восприятию человеком вторая комната в полтора раза шумнее первой, но фактически звуковой энергии во второй комнате в 100 раз больше.

Этот феномен объясняет, почему отдельная скрипка слышна на фоне целого оркестра (если она играет собственную мелодию) и, тем не менее, в оркестре должно быть такое большое количество скрипок, чтобы звук был должным образом сбалансирован.

Интенсивность землетрясений также измеряется логарифмически: увеличению интенсивности на 1 по шкале Рихтера (С. R. Richter) соответствует десятикратное увеличение высвобожденной при землетрясении энергии.

большие числа

Во-вторых, логарифмы предоставляют удобный способ работы с очень большими числами. Используемая в научных приложениях нотация, когда число записывается в виде очень небольшого действительного числа (часто в диапазоне от 1 до 10), умноженного на степень 10, фактически основана на логарифмах, поскольку степень 10 есть логарифм этого числа. Ученые, которым приходится работать с очень большими числами (астрономы, физики-ядерщики и геологи) часто говорят о порядке величины, и тем самым концентрируют внимание на логарифме числа.

В третьих, логарифмический график является весьма полезным средством для выражения свойств функции на значительно более широком диапазоне, чем линейный график. Используя логарифмический график, мы можем отобразить детальную информацию о функции при очень маленьких значениях ее аргумента и, в то же время, получить представление о ее поведении при значительно больших значениях аргумента. Логарифмические графики особенно удобны, когда мы хотим показать относительные изменения функции.

## А.2.1. Определение логарифмов

основание

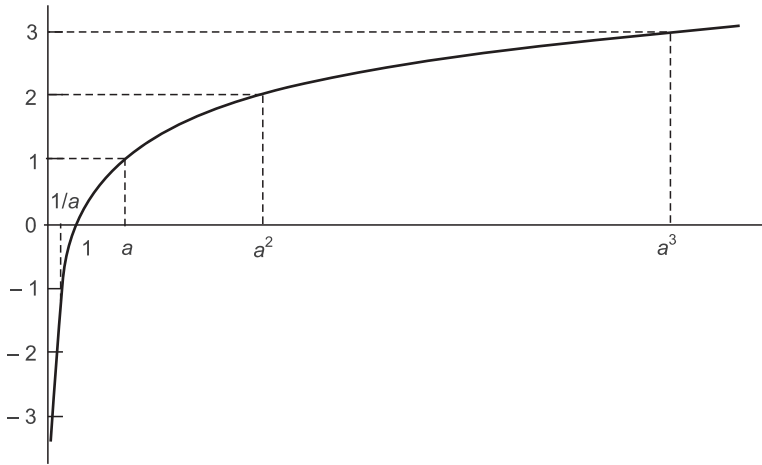
Логарифмы определяются с помощью действительного числа  $a > 1$ , которое называется *основанием* логарифма. (Можно также определить логарифмы с основанием  $0 < a < 1$ , но такое определение приведет к ненужным усложнениям наших рассуждений.) Для любого числа  $x > 0$  мы определяем  $\log_a x = y$ , где  $y$  есть такое действительное число, что  $a^y = x$ . Логарифм отрицательного числа, как и логарифм 0, не определены.

## А.2.2. Простые свойства

Из определения и из свойств показателей мы получаем

$$\begin{aligned} \log_a 1 &= 0, \\ \log_a a &= 1, \\ \log_a x &< 0 \text{ для всех } x, \text{ удовлетворяющих условию } 0 < x < 1, \\ 0 < \log_a x &< 1 \text{ для всех } x, \text{ удовлетворяющих условию } 1 < x < a, \\ \log_a x &> 1 \text{ для всех } x, \text{ удовлетворяющих условию } a < x. \end{aligned}$$

Логарифмическая функция имеет вид, изображенный на рис. А.2.



**Рис. А.2.** График логарифмической функции

Мы также получаем равенства

равенства

$$\begin{aligned}\log_a(xy) &= (\log_a x) + \log_a y \\ \log_a(x/y) &= (\log_a x) - \log_a y \\ \log_a x^z &= z \log_a x \\ \log_a a^z &= z \\ a^{\log_a x} &= x\end{aligned}$$

которые справедливы для любых положительных целых чисел  $x$  и  $y$  и для любого действительного числа  $z$ .

Из графика на рис. А.2 вы можете заметить, что по мере увеличения  $x$  логарифм растет все медленнее и медленнее. Графики положительных степеней  $x$ , меньших 1, например, квадратного корня из  $x$  или кубического корня из  $x$ , также растут все более медленно, но никогда не становятся столь плоскими, как логарифм. Фактически

*Для любых  $c > 0$   $\log x$  по мере роста  $x$  растет более медленно, чем  $x^c$ .*

### А.2.3. Выбор основания

обычные  
логарифмы

В качестве основания логарифмов можно выбрать любое действительное число  $a > 1$ , однако некоторые варианты выбора используются значительно чаще других. Для вычислений и графиков часто используется  $a = 10$ , и логарифмы по основанию 10 называют **десятичными** (или **обычными**) **логарифмами**. При изучении компьютерных алгоритмов, однако, основание 10 встречается нечасто, и мы редко пользуемся десятичными логарифмами. Значительно более часто приходится обращаться к логарифмам по основанию 2, и для них зарезервировано специальное обозначение

$\lg x$ .

## А.2.4. Натуральные логарифмы

При исследовании математических свойств логарифмов и во многих задачах, где логарифмы входят составной частью в результат, в качестве основания используется число

$$e = 2.718281828549... .$$

натуральные  
логарифмы

Логарифмы по основанию  $e$  называются **натуральными логарифмами**. В этой книге мы повсюду обозначает натуральный логарифм  $x$

$$\ln x.$$

Во многих математических книгах, однако, другие основания, кроме  $e$ , почти не встречаются, и тогда для натурального логарифма используется неуточненное обозначение  $\log x$ . На рис. А.3 показаны графики логарифмов по трем разным основаниям: 2,  $e$  и 10.

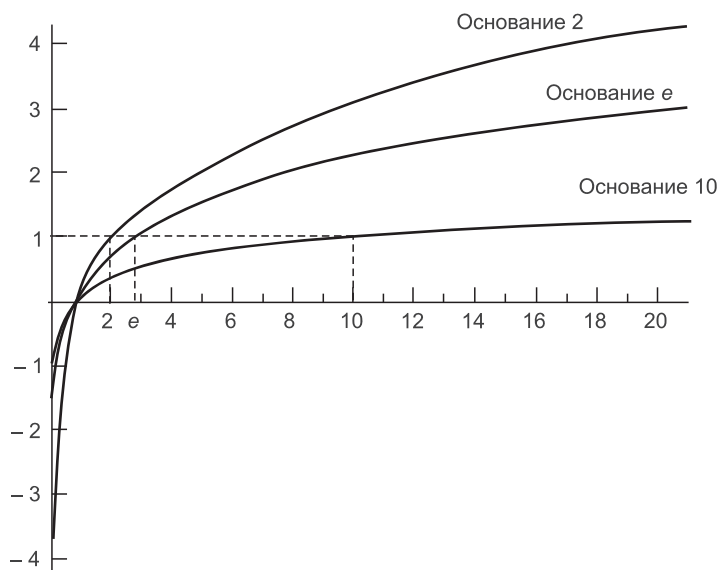


Рис. А.3. Логарифмы по трем разным основаниям

Свойства логарифмов, которые обосновывают выбор  $e$  в качестве основания, тщательно изучены и разработаны, но мы только упомянем без доказательства два таких свойства. Первое, график  $\ln x$  имеет то свойство, что его наклон в каждой точке  $x$  равен  $1/x$ ; другими словами, производная от  $\ln x$  равна  $1/x$  для всех действительных чисел  $x > 0$ . Второе, натуральный логарифм удовлетворяет бесконечному ряду

$$\ln(x+1) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots$$

для  $-1 < x < 1$ , но для получения удовлетворительного приближения этот ряд требует много членов и поэтому непосредственно в вычислениях не

экспоненциальная  
функция

используется. Значительно удобнее вместо этого ряда рассматривать экспоненциальную функцию, которая удовлетворяет ряду

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

для всех действительных чисел  $x$ . Экспоненциальная функция обладает также тем важным свойством, что производная от нее равна ей самой.

## А.2.5. Обозначения

Приведенные выше обозначения логарифмов по различным основаниям являются стандартными. Подытожим сказанное.

### Соглашения

*Если не указано обратное, все логарифмы берутся по основанию 2.*

*Обозначение  $\lg$  относится к логарифму по основанию 2, а обозначение  $\ln$  относится к натуральному логарифму.*

*Если основание логарифмов не указано или не имеет значения, используется обозначение  $\log$ .*

## А.2.6. Изменение основания логарифмов

Логарифмы по одному основанию тесно связаны с логарифмами по любому другому основанию. Чтобы найти эту взаимосвязь, мы начнем со следующего отношения, которое в сущности является определением и справедливо при любых  $x > 0$ .

$$x = a^{\log_a x}.$$

Из него следует, что

$$\log_b x = \log_b a^{\log_a x} = (\log_a x)(\log_b a).$$

Коэффициент  $\log_b a$  не зависит от  $x$ , а лишь от двух разных оснований. Следовательно:

*Для преобразования логарифма по одному основанию в логарифм по другому его следует умножить на постоянный коэффициент, равный логарифму первого основания по второму основанию.*

коэффициенты  
преобразования

В этом отношении наиболее полезными для нас являются следующие коэффициенты:

$$\begin{aligned} \lg e &\approx 1.442695041, \\ \ln 2 &\approx 0.693147181, \\ \ln 10 &\approx 2.302585093, \\ \lg 1000 &\approx 10. \end{aligned}$$

Последнее значение является следствием важного приближения  $2^{10} = 1024 \approx 10^3 = 1000$ .

## А.2.7. Логарифмические графики

На логарифмической шкале числа выстроены как на логарифмической линейке, т. е. большие числа располагаются ближе друг к другу, чем маленькие. В этом случае равные расстояния на шкале отвечают равным *отношениями*, а не равным *разностям*, как это имеет место для обычной линейной шкалы. Логарифмическую шкалу следует использовать в тех случаях, когда важным является относительное изменение величины, или когда восприятие величины носит логарифмический характер. Например, человеческое восприятие времени иногда на коротких интервалах времени кажется почти линейным — то, что произошло два дня назад, вдвое дальше от того, что случилось сегодня — но часто на больших интервалах представляется ближе к логарифмическому: мы видим меньше разницы между одним миллионом лет назад и двумя миллионами лет назад, чем между моментами времени десять лет назад и сто лет назад.

логарифмические  
графики

Графики, в которых и вертикальная, и горизонтальная оси координат являются логарифмическими, так и называются **логарифмическими графиками**. В дополнение к явлениям, в которых обе шкалы оказываются логарифмическими по естественным причинам, логарифмические графики полезны для отображения поведения функции в широком диапазоне значений. Для небольших значений график отображает детальное поведение функции, а для больших значений на том же графике отображается поведение функции в более широком плане. Для алгоритмов поиска и сортировки нам интересно сравнить методы и для задач небольшого размера, и для крупных проектов; в таких случаях удобны как раз логарифмические графики (см. рис. А.4).

Одно наблюдение стоит того, чтобы его отметить: графики степеней  $x$  в логарифмическом масштабе представляют собой прямые линии. Для доказательства этого утверждения начнем с произвольной степенной функции  $y = x^n$ , и возьмем логарифмы от обеих сторон, получив

$$\log y = n \log x.$$

Логарифмический график зависимости  $y$  от  $x$  становится линейным в осях  $u = \log x$  и  $v = \log y$ , а уравнение относительно  $u$  и  $v$  превращается в  $v = nu$ , что безусловно отображает прямолинейную зависимость.

## А.2.8. Гармонические числа

В качестве заключительного приложения логарифмов получим аппроксимацию суммы ряда, с которым мы часто встречаемся при анализе алгоритмов, особенно в методах сортировки. **Гармоническое число** степени  $n$  определяется суммой

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

обратных величин целых чисел от 1 до  $n$ .

Для оценки  $H_n$  рассмотрим функцию  $1/x$  и взаимосвязь, графически изображенную на рис. А.5. Площадь под ступенчатой функцией, очевидно, равна  $H_n$ , поскольку ширина каждой ступени равна 1, а высота  $k$  есть

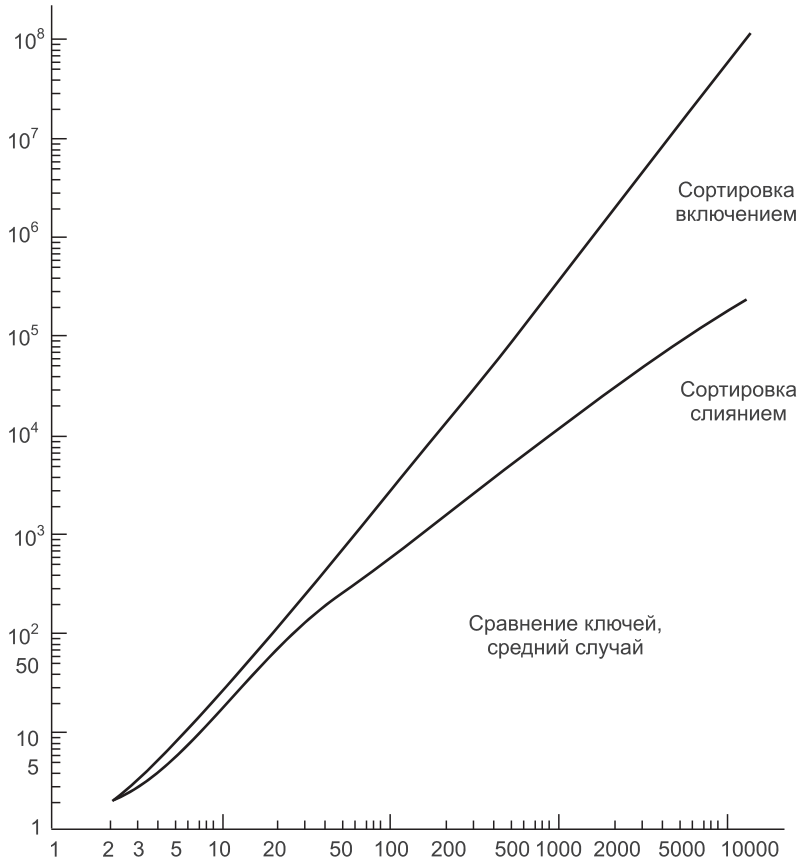


Рис. А.4. Логарифмический график, сравнение методов сортировки

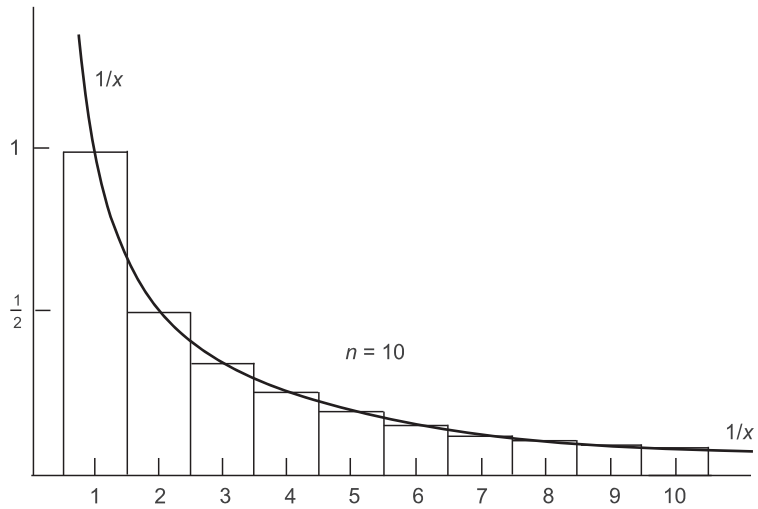


Рис. А.5. Аппроксимация выражения  $\int_{\frac{1}{2}}^{n+\frac{1}{2}} \frac{1}{x} dx$



$1/k$  для всех  $k$  от 1 до  $n$ . Эта площадь приблизительно равна площади под кривой  $1/x$  от  $\frac{1}{2}$  до  $n + \frac{1}{2}$ . Площадь под кривой составляет

$$\int_{\frac{1}{2}}^{n+\frac{1}{2}} \frac{1}{x} dx = \ln(n + \frac{1}{2}) - \ln \frac{1}{2} \approx \ln n + 0.7.$$

Когда  $n$  велико, дробный член 0.7 не имеет значения, и мы получаем, что  $\ln n$  есть хорошая аппроксимация  $H_n$ .

Улучшив этот метод аппроксимации использованием интеграла, можно получить значительно более точную аппроксимацию  $H_n$ , если это требуется. Конкретно,

**Теорема А.4**

*Гармоническое число  $H_n$  при  $n \geq 1$  удовлетворяет выражению*

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \varepsilon,$$

где  $0 < \varepsilon < 1/(252n^6)$ , а  $\gamma \approx 0.577215665$ , известна как **постоянная Эйлера**.

## А.3. Перестановки, сочетания, факториалы

### А.3.1. Перестановки

Перестановка объектов представляет собой их упорядочение или реорганизацию объектов в ряду. Если мы начинаем с  $n$  различных объектов, мы можем выбрать в качестве первого любой из  $n$  объектов. Для второго объекта тогда остается только  $n - 1$  вариантов выбора, и поскольку эти варианты могут комбинироваться во всех возможных сочетаниях, число вариантов перемножается. Отсюда два первых объекта могут быть выбраны  $n(n - 1)$  способами. У нас остается  $n - 2$  объектов, любой из которых может быть выбран в качестве третьего объекта нашего ряда. Продолжая рассуждать таким же образом, мы видим, что число перестановок из  $n$  различных объектов составляет

$$n! = n \times (n - 1) \times (n - 2) \times \dots \times 2 \times 1$$

Все возможные перестановки из 4 объектов показаны на рис. А.6.

число  
перестановок

| Перестанавливаемые объекты: a b c d |         |         |         |         |         |         |  |
|-------------------------------------|---------|---------|---------|---------|---------|---------|--|
| Выберем сначала a:                  | a b c d | a b d c | a c b d | a c d b | a d b c | a d c b |  |
| Выберем сначала b:                  | b a c d | b a d c | b c a d | b c d a | b d a c | b d c a |  |
| Выберем сначала c:                  | c a b d | c a d b | c b a d | c b d a | c d a b | c d b a |  |
| Выберем сначала d:                  | d a b c | d a c b | d b a c | d b c a | d c a b | d c b a |  |

**Рис. А.6.** Конструирование перестановок

Заметьте, что мы предположили, что все объекты различаются, другими словами, мы можем отличить один объект от другого. Часто оказывается проще подсчитать конфигурации различающихся объектов, чем рассматривать случай, когда некоторые объекты неразличимы. Послед-

ную задачу иногда можно решить путем назначения объектам временных меток, затем подсчетом числа конфигураций, и, наконец, делением на число способов назначения меток. Частный случай такого рода оказывается особенно важным и будет рассмотрен в следующем разделе.

### А.3.2. Сочетания

Сочетание из  $n$  объектов по  $k$  есть отбор  $k$  объектов из общего их числа  $n$  безотносительно к порядку объектов в выбранной конфигурации. Число таких комбинаций (сочетаний) обозначается либо как

$$C(n, k), \text{ либо как } \binom{n}{k}.$$

Мы можем вычислить  $C(n, k)$ , начав с  $n!$  перестановок из  $n$  объектов и образуя сочетания просто путем выбора первых  $k$  объектов в перестановке. Однако порядок, в котором эти  $k$  объектов появляются в сочетании, игнорируется, поэтому мы должны разделить это число на число  $k!$  способов упорядочения  $k$  выбранных объектов. Порядок  $n - k$  невыбранных объектов также не имеет значения, поэтому мы должны еще разделить результат на  $(n - k)!$ . Отсюда получаем:

$$C(n, k) = \frac{n!}{k!(n - k)!}.$$

На рис. А.7 показаны все сочетания из 6 объектов по 3.

| Объекты, из которых мы выбираем сочетания: a b c d e f |       |       |       |       |
|--------------------------------------------------------|-------|-------|-------|-------|
| a b c                                                  | a c d | a d f | b c f | c d e |
| a b d                                                  | a c e | a e f | b d e | c d f |
| a b e                                                  | a c f | b c d | b d f | c e f |
| a b f                                                  | a d e | b c e | b e f | d e f |

Рис. А.7. Сочетания из 6 объектов по 3

биномиальные  
коэффициенты

Число сочетаний  $C(n, k)$  называется **биномиальным коэффициентом**, поскольку оно выступает в качестве коэффициента члена  $x^k y^{n-k}$  в разложении  $(x + y)^n$  в ряд. Можно насчитать сотни различных отношений и равенств, описывающих различные суммы и произведения биномиальных коэффициентов. Наиболее важные из них можно найти в учебниках по элементарной алгебре и комбинаторике.

### А.3.3. Факториалы

При анализе алгоритмов нам часто приходится использовать перестановки и сочетания, и для такого рода приложений мы должны оценивать величину  $n!$  для различных значений  $n$ . Превосходное приближение к  $n!$  было найдено Джеймсом Стирлингом (James Stirling) в 18-м столетии:

$$n! \approx \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \left[ 1 + \frac{1}{12n} + O\left( \frac{1}{n^2} \right) \right].$$

Теорема А.5

аппроксимация  
Стирлинга

Обычно мы используем это приближение в логарифмической форме:

**Следствие А.6**

$$\ln n! \approx (n + \frac{1}{2}) \ln n - n + \frac{1}{2} \ln(2\pi) + \frac{1}{12n} + O\left(\frac{1}{n^2}\right).$$

Заметьте, что по мере роста  $n$  приближение к логарифму становится все более точным; другими словами, разность между ними стремится к 0. Разность между приближением к факториалу и самим факториалом  $n!$  не обязательно становится малой (другими словами, разность не стремится к 0), но относительная ошибка становится произвольно малой (отношение стремится к 1). Кнут [Knuth] в томе 1, стр. 111 (см. ссылку в главе 3), приводит улучшенное приближение Стирлинга, которое еще ближе к аппроксимируемой величине.

Полное доказательство приближения Стирлинга требует сложных математических приемов и завело бы нас слишком далеко. Мы можем, однако, использовать некоторые элементарные рассуждения для иллюстрации первого шага в приближении Стирлинга. Прежде всего мы берем натуральный логарифм от факториала, учитывая, что логарифм произведения равен сумме логарифмов:

$$\begin{aligned} \ln n! &= \ln(n \times (n-1) \times \dots \times 1) \\ &= \ln n + \ln(n-1) + \dots + \ln 1 \\ &= \sum_{x=1}^n \ln x. \end{aligned}$$

Далее мы аппроксимируем сумму интегралом, как это показано на рис. А.8.

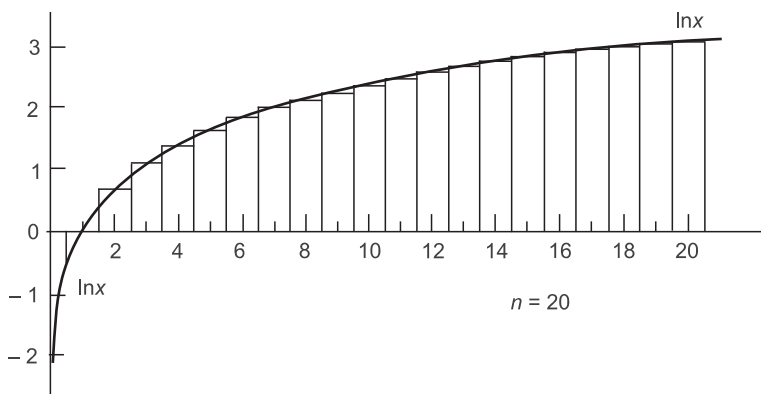


Рис. А.8. Аппроксимация  $\ln n!$  посредством  $\int_{\frac{1}{2}}^{n+\frac{1}{2}} \ln x dx$

Из рисунка ясно, что площадь под ступенчатой функцией, равная в точности  $\ln n!$ , приблизительно совпадает с площадью под плавной кривой, которая равна

$$\int_{\frac{1}{2}}^{n+\frac{1}{2}} \ln x dx = (x \ln x - x) \Big|_{\frac{1}{2}}^{n+\frac{1}{2}} = (n + \frac{1}{2}) \ln(n + \frac{1}{2}) - n + \frac{1}{2} \ln 2.$$

Для больших значений  $n$  разность между  $\ln n$  и  $\ln(n + \frac{1}{2})$  оказывается незначительной, и это приближение отличается от приближения Стирлинга только на постоянную величину, лежащую между  $\frac{1}{2} \ln 2$  (около 0.35) и  $\frac{1}{2} \ln(2\pi)$  (около 0.919).

## А.4. Числа Фибоначчи

Числа Фибоначчи ведут свою историю от упражнения по арифметике, предложенного Леонардо Фибоначчи (Leonardo Fibonacci) в 1202 г.

*Сколько пар кроликов могут родиться от одной пары за год? Мы начинаем с только что родившейся пары; кролики взрослеют в течение месяца, после чего они начинают приносить по новой паре ежемесячно, причем кролики никогда не умирают.*

В месяце 1 мы имеем только одну пару. В месяце 2 мы все еще имеем только одну пару, но теперь они уже взрослые. В месяце 3 у них родилось потомство, так что теперь у нас есть две пары. И далее этот процесс продолжается. Число  $F_n$  пар кроликов, имеющих в наличии в месяце  $n$ , удовлетворяет следующим отношениям

$$F_0 = 0, \quad F_1 = 1 \quad \text{и} \quad F_n = F_{n-1} + F_{n-2} \quad \text{для} \quad n \geq 2.$$

Такая же последовательность чисел, называемая **последовательностью Фибоначчи**, возникает при решении многих задач. В разделе 10.4, например,  $F_n$  определяет минимальное число узлов в AVL-дереве высоты  $n$ . Целью этого раздела является нахождение формулы для  $F_n$ .

Мы используем здесь метод **производящих функций**, важный и для многих других приложений. Производящая функция представляет собой формальный бесконечный ряд по некоторому  $x$ , в котором коэффициентами являются числа Фибоначчи:

$$F(x) = F_0 + F_1 x + F_2 x^2 + \dots + F_n x^n + \dots$$

Нас не интересует, сходится ли этот ряд, или каково может быть значение  $x$ , поскольку мы не собираемся присваивать  $x$  какое бы то ни было конкретное значение. Мы только выполним формальные алгебраические манипуляции над производящей функцией.

Далее, мы умножаем обе стороны равенства на степени  $x$ :

$$\begin{aligned} F(x) &= F_0 + F_1 x + F_2 x^2 + \dots + F_n x^n + \dots \\ xF(x) &= F_0 x + F_1 x^2 + \dots + F_{n-1} x^n + \dots \\ x^2 F(x) &= F_0 x^2 + \dots + F_{n-2} x^n + \dots \end{aligned}$$

и вычитаем вторые два уравнения из первого:

$$(1 - x - x^2)F(x) = F_0 + (F_1 - F_0)x = x,$$

поскольку  $F_0 = 0$ ,  $F_1 = 1$  и  $F_n = F_{n-1} + F_{n-2}$  для всех  $n \geq 2$ . В результате мы получаем:

$$F(x) = \frac{1}{1 - x - x^2}.$$

рекуррентное  
отношение

аналитическая  
форма

Корнями  $1 - x - x^2$  будут  $\frac{1}{2}(-1 \pm \sqrt{5})$ . С помощью метода разложения на элементарные дроби мы можем преобразовать формулу для  $F(x)$  следующим образом:

$$F(x) = \frac{1}{\sqrt{5}} \left( \frac{1}{1 - \varphi x} - \frac{1}{1 - \psi x} \right),$$

где

$$\varphi = \frac{1}{2}(1 + \sqrt{5}) \quad \text{и} \quad \psi = 1 - \varphi = \frac{1}{2}(1 - \sqrt{5}).$$

(Проверьте это уравнение для  $F(x)$ , приведя обе дроби с правой стороны к общему знаменателю.)

Следующим шагом будет разложение дробей с правой стороны в бесконечные ряды:

$$F(x) = \frac{1}{\sqrt{5}} (1 + \varphi x + \varphi^2 x^2 + \dots - 1 - \psi x - \psi^2 x^2 - \dots).$$

Наконец, воспользовавшись тем, что коэффициенты разложения  $F(x)$  есть числа Фибоначчи, мы можем приравнять коэффициенты при каждой степени  $x$  в двух выражениях для  $F(x)$ , получив

$$F(n) = \frac{1}{\sqrt{5}} (\varphi^n - \psi^n).$$

Приблизительные значения для  $\varphi$  и  $\psi$  составляют

$$\varphi \approx 1.618034 \quad \text{и} \quad \psi \approx -0.618034.$$

Этот удивительно простое выражение для чисел Фибоначчи интересно в нескольких отношениях. Во-первых, не очевидно, что правая сторона всегда будет целым числом. Во-вторых,  $\psi$  представляет собой отрицательное число с небольшим абсолютным значением и поэтому мы всегда имеем  $F_n = \varphi^n / \sqrt{5}$ , округленное до ближайшего целого. В-третьих, число  $\varphi$  интересно само по себе. Оно изучалось еще со времен древних греков — его называют *золотым сечением* — и отношение  $\varphi$  к 1 считается наиболее приятной для глаза формой прямоугольника. Парфенон и многие другие здания Древней Греции имели стороны с таким отношением.

золотое сечение

## А.5. Числа Каталана

Целью настоящего раздела является определение числа двоичных деревьев с  $n$  вершинами. Мы получим этот результат, несколько отклонившись от прямого пути и обнаружив по дороге несколько других задач, имеющих тот же ответ. Полученные нами числа, называемые *числами Каталана*, представляют собой значительный интерес, поскольку они встречаются во многих не связанных друг с другом задачах.

### А.5.1. Основной результат

**Определение** Для  $n \geq 0$   $n$ -е *число Каталана* определяется таким образом:

$$\text{Cat}(n) = \frac{C(2n, n)}{n+1} = \frac{(2n)!}{(n+1)!n!}.$$

## Теорема А.7

*Число различающихся двоичных деревьев с  $n$  вершинами при  $n \geq 0$  есть  $n$ -е число Каталана  $Cat(n)$ .*

## А.5.2. Доказательство посредством однозначного соответствия

### 1. Сады

Для начала вспомним полученное из теоремы 11.1 однозначное соответствие между двоичными деревьями с  $n$  вершинами и садами с  $n$  вершинами. Из него следует, что, желая определить число двоичных деревьев, мы можем с таким же успехом определить число садов.

### 2. Корректная последовательность скобок

Далее, рассмотрим множество корректных последовательностей из  $n$  левых скобок '(' и  $n$  правых скобок ')'. Последовательность называется *корректной*, если при просмотре ее слева направо число встреченных правых скобок никогда не превышает число левых скобок. Так, последовательности '((( )))' и '()()()' являются корректными, но последовательность '())(((' таковой не является, как и последовательность '(()', поскольку полное число левых и правых скобок в выражении должно быть одинаковым.

## Лемма А.8

*Имеется взаимно однозначное соответствие между садами с  $n$  вершинами и корректными последовательностями из  $n$  левых скобок и  $n$  правых скобок при  $n \geq 0$ .*

Для определения соответствия мы сначала вспомним, что сад может быть либо пуст, либо содержать упорядоченную последовательность упорядоченных деревьев. Мы определяем *скобочную форму* сада как последовательность скобочных форм его деревьев, написанных друг за другом в том же порядке, как и деревья сада. Скобочная форма пустого сада пуста. Вспомним также, что упорядоченное дерево определяется как состоящее из его корневых вершин вместе с садом поддеревьев. Мы таким образом определяем *скобочную форму* упорядоченного дерева как состоящую из левой скобки '(', за которой следует корень (имя корня), далее скобочная форма сада поддеревьев и, наконец, правая скобка ')'.  
Скобочные формы нескольких упорядоченных деревьев и садов показаны на рис. А.9. Легко сообразить, что данные нами взаимно рекурсивные определения образуют уникальную скобочную форму для любого сада, и что результирующие последовательности скобок являются корректными. Если, с другой стороны, мы начинаем рассуждение с корректной последовательности скобок, тогда самая внешняя пара (или пары) скобок соответствует дереву (деревьям) сада, а внутри такой пары скобок находится определение соответствующего дерева с помощью понятий его корня и его сада поддеревьев. Рассуждая таким образом, мы получаем однозначное соответствие между садами с  $n$  вершинами и корректными последовательностями из  $n$  левых и  $n$  правых скобок.

Определяя число садов, мы не будем обращать внимание на ярлыки, прикрепленные к вершинам, и, поскольку мы опускаем ярлыки, мы опре-

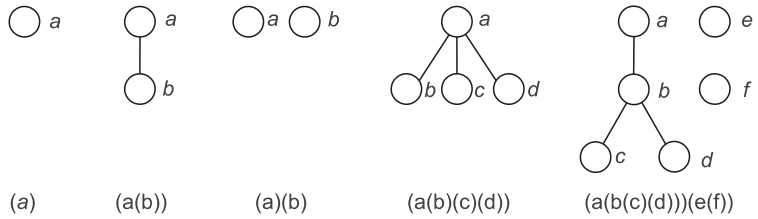


Рис. А.9. Скобочные формы садов

делим число корректных последовательностей из  $n$  левых и  $n$  правых скобок, в которых внутри скобок нет ничего кроме (возможно) скобок.

### 3. Перестановки в стеке

Заметим, что при замене каждой левой скобки обозначением  $+1$ , а каждой правой скобки обозначением  $-1$ , образуется соответствующая корректной последовательности скобок последовательность чисел  $+1$  и  $-1$ , частичные суммы которой слева направо всегда неотрицательны, а полная сумма равна  $0$ . Если мы представим себе каждую  $+1$  как проталкивание данного в стек, а каждую  $-1$  как выталкивание из стека, тогда частичные суммы отражают число элементов в стеке в данный момент времени. Отсюда можно показать, что число перестановок из  $n$  объектов в стеке (см. упражнение ЕЗ в разделе 3.1) оказывается еще одной задачей, ответ на которую дают числа Каталана. Далее, если мы начинаем с сада и выполняем полный просмотр (проходя по каждой ветви и вершине сада, как если бы он был декоративной стенкой), отсчитывая  $+1$  при каждом спуске по ветви и  $-1$  при каждом подъеме по ветви (и, соответственно,  $+1 -1$  для каждого листа), мы опять получим соответствие с корректными последовательностями.

### 4. Произвольные последовательности скобок

Нашим последним шагом должно стать определение числа корректных последовательностей скобок, однако вместо этого мы определим число *некорректных* последовательностей и вычтем их из полного числа всех возможных последовательностей. Нам потребуется окончательное однозначное соответствие:

**Лемма А.9**

*Последовательности из  $n$  левых и  $n$  правых скобок, которые не являются корректными, соответствуют в точности всем последовательностям из  $n - 1$  левых скобок и  $n + 1$  правых скобок (во всех возможных порядках).*

Для доказательства этого соответствия давайте начнем с последовательности из  $n$  левых и  $n$  правых скобок, которая не является корректной. Пусть  $k$  есть первая позиция, начиная с которой последовательность оказывается неправильной, т. е. элемент в позиции  $k$  есть правая скобка, и в результате число правых скобок на 1 превышает число левых скобок до этой позиции. Тогда в точности справа от позиции  $k$  имеется на 1 меньше правых скобок, чем левых. Теперь строго справа от позиции  $k$

заменяем все левые скобки на правые, а правые на левые. В результирующей последовательности будет всего  $n - 1$  левых скобок и  $n + 1$  правых скобок.

Наоборот, начнем с последовательности из  $n - 1$  левых скобок и  $n + 1$  правых скобок и пусть  $k$  есть первая позиция, в которой число правых скобок превышает число левых скобок (такая позиция должна существовать, поскольку в последовательности больше правых скобок, чем левых). Опять заменим в оставшейся части последовательности (после позиции  $k$ ) левые скобки на правые, а правые на левые. Мы получим некорректную последовательность из  $n$  левых и  $n$  правых скобок, и мы, таким образом, доказали требуемое однозначное соответствие.

## 5. Конец доказательства

Принимая во внимание все эти предварительные соответствия, наша задача определения числа последовательностей сводится к простым сочетаниям. Число последовательностей из  $n - 1$  левых и  $n + 1$  правых скобок есть число способов, которыми можно выбрать  $n - 1$  позиций, занятых левыми скобками, из  $2n$  позиций в последовательности; это число составляет  $C(2n, n - 1)$ . По лемме А.9 это число есть также число некорректных последовательностей из  $n$  левых и  $n$  правых скобок. Число всех последовательностей из  $n$  левых и  $n$  правых скобок равно  $C(2n, n)$ , откуда число корректных последовательностей составляет

$$C(2n, n) - C(2n, n - 1),$$

что в точности совпадает с  $n$ -м числом Каталана.

В силу всех этих однозначных соответствий мы также имеем:

### Следствие А.10

*Число корректных последовательностей из  $n$  левых и  $n$  правых скобок, число перестановок из  $n$  объектов, находящихся в стеке, число садов из  $n$  вершин и число двоичных деревьев с  $n$  вершинами равны  $n$ -му числу Каталана  $\text{Cat}(n)$ .*

## А.5.3. История вопроса

Любопытно, что числа Каталана были впервые получены не для решения одной из поставленных выше задач, а при рассмотрении геометрических проблем. Конкретно,  $\text{Cat}(n)$  определяет число способов, которыми можно разделить выпуклый многоугольник с  $n + 2$  сторонами на треугольники путем проведения  $n - 1$  непересекающихся диагоналей (см. рис. А.10). Эта задача была, видимо, впервые поставлена Л. Эйлером (L. Euler) и решена Сегнером (J. A. v. Segner) в 1759 г. Она снова была решена Е. Каталаном (E. Catalan) в 1838 г. По этой причине получаемые числа иногда называют *числами Сегнера*, хотя чаще о них говорят, как о *числах Каталана*.

## А.5.4. Численные результаты

Мы завершаем этот раздел замечаниями о значениях чисел Каталана. Первые двадцать значений приведены на рис. А.11.



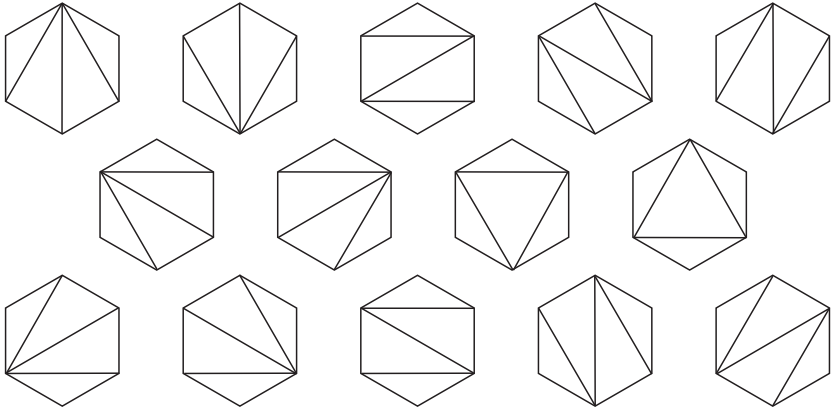


Рис. А.10. Триангуляции шестиугольника с помощью диагоналей

| $n$ | $\text{Cat}(n)$ | $n$ | $\text{Cat}(n)$ |
|-----|-----------------|-----|-----------------|
| 0   | 1               | 10  | 16796           |
| 1   | 1               | 11  | 58786           |
| 2   | 2               | 12  | 208012          |
| 3   | 5               | 13  | 742900          |
| 4   | 14              | 14  | 2674440         |
| 5   | 42              | 15  | 9694845         |
| 6   | 132             | 16  | 35357670        |
| 7   | 429             | 17  | 129644790       |
| 8   | 1430            | 18  | 477638700       |
| 9   | 4862            | 19  | 1767263190      |

Рис. А.11. Первые 20 чисел Каталана

Для больших  $n$  мы можем получить оценку значений чисел Каталана, используя формулу Стирлинга. Если приложить ее к каждому из трех факториалов и упростить результат, мы получим

$$\text{Cat}(n) \approx \frac{4^n}{(n+1)\sqrt{\pi n}}.$$

Если сравнить получаемые с помощью этой формулы числа с точными значениями на рис. А.11, мы получим хорошее представление о точности приближения Стирлинга. Если, например,  $n = 10$ , оценочное значение числа Каталана составляет 17007, в то время как точное значение — 16796.

## Литература для дальнейшего изучения

Более развернутые обсуждения доказательств по индукции, обозначения операций суммирования, сумм степеней целых чисел и логарифмов можно найти во многих учебниках по алгебре. В этих книгах также содержатся примеры и упражнения на затрагиваемые темы. Превосходное обсуждение важности логарифмов и тонкого искусства оценочных вычислений можно найти в статье

логарифмы N. David Mermin, «Logarithms!», *American Mathematical Monthly* 87 (1980), 1–7.

Несколько интересных примеров оценки больших чисел и представления их с помощью логарифмов обсуждаются в работе

Douglas R. Hofstadter, «Metamagical themas», *Scientific American* 246, no. 5 (May 1982), 20–34.

гармонические  
числа

Несколько удивительных и занимательных применений гармонических чисел описаны в статье нетехнического характера

Ralph Boas, «Snowfalls and elephants, pop bottles and  $\pi$ », *Two-Year College Mathematics Journal* 11 (1980), 82–89.

Точные оценки гармонических чисел и факториалов (приближение Стирлинга) взяты из [Knuth], том 1, стр. 108–111, где можно также найти подробные доказательства. Книга [Knuth] (см. ссылку в главе 3), том 1 также является великолепным источником дополнительной информации о перестановках, сочетаниях и смежных вопросах.

факториалы

Первое упоминание о приближении Стирлинга дано в работе James Stirling, *Methodus Differential* (1730), p. 137.

комбинаторика

Область математики, занимающаяся перечислением различных множеств или классов объектов называется **комбинаторикой**. Эта наука может быть представлена на крайне простом уровне или изучена во всей своей тонкости. Ниже приведены два элементарных учебных пособия, содержащие дальнейшую разработку представленных здесь идей:

Gerald Berman and K. D. Fryer, *Introduction to Combinatorics*, Academic Press, 1972.

Alan Tucker, *Applied Combinatorics*, John Wiley, New York, 1980.

числа Фибоначчи

Рассмотрение чисел Фибоначчи можно найти почти в любой книге по комбинаторике, а также в [Knuth], том 1, стр. 78–86, куда включены несколько интересных сведений из истории, а также много упражнений по этой теме. Проявление чисел Фибоначчи в природе проиллюстрировано в книге

Peter Stevens, *Patterns in Nature*, Little, Brown, Boston, 1974.

Уже найдены и продолжают обнаруживаться многие сотни разнообразных свойств чисел Фибоначчи; часто результаты этих исследований публикуются в журнале *Fibonacci Quarterly*.

числа Каталана

Рассмотрение чисел Каталана (в приложении к триангуляции выпуклых многоугольников) можно найти в первой из цитируемых книг по комбинаторике (Berman and Fryer, pp 230–232). В книге [Knuth], том 1, стр. 385–406, перечисляется несколько классов деревьев, в том числе го-

ворится о приложении к двоичным деревьям чисел Каталана. Список из 46 ссылок по истории и приложениям чисел Каталана можно найти в статье

W. G. Brown, «*Historical note on a recurrent combinatorial problem*», *American Mathematical Monthly* 72 (1965), 973–977.

В следующей статье рассматривается большое количество приложений чисел Каталана:

Martin Gardner, «*Mathematical games*» (regular column), *Scientific American* 234, no. 6 (June, 1976), 120–125.

Первые упоминания о получении чисел Каталана содержатся в работах

J. A. v. Segner, «*Enumeratio modorum, quibus figuræ planæ rectilinæ per diagonales dividuntur in triangula*», *Novi Commentarii Academiæ Scientiarum Imperialis Petropolitane* 7 (1758–1759), 203–209.

E. Catalan, «*Solution nouvelle de cette question: un polygone étant donné, de combien de manieres peut-on le partager en triangles au moyen de diagonales?*», *Journal de Mathématiques Pures et Appliquées* 4 (1839), 91–94.

# Приложение В

## Случайные числа

---

Случайные числа представляют собой ценное средство для проверки работы компьютерных программ при различных исходных данных. В этом приложении кратко рассматриваются вопросы генерации случайных чисел и описываются случайные числа разных видов. Приводятся Pascal-процедуры генерации разного рода случайных чисел.

### В.1. Введение

Неопределенности украшают нашу жизнь. Компьютеры, с другой стороны, действуют абсолютно предсказуемо и поэтому скучны. Случайные числа предоставляют способ включить в компьютерную программу элементы непредсказуемости, отчего в некоторых случаях программы лучше имитируют внешние события. При использовании в компьютерных играх, графическом представлении информации или программах моделирования, случайные числа могут сделать эти программы значительно более интересными, а при многократных прогонах программ случайные числа позволяют показать разнообразие их поведения, схожее с поведением реальных систем, которые эти программы имитируют.

Многие компьютерные системы включают в себя генераторы случайных чисел, и если один из таких генераторов имеется на вашей системе, его можно использовать вместо тех, что разрабатываются далее в этом разделе. Поскольку, однако, стандартный Pascal не включает процедур генерации случайных чисел, мы кратко обсуждаем здесь вопросы их разработки. Системные генераторы случайных чисел иногда оказываются не очень качественными, так что стоит посмотреть, как конструируются генераторы с лучшими характеристиками.

В любом случае мы будем рассматривать в этом разделе генератор случайных чисел и все относящиеся к нему подпрограммы как *пакет* для создания случайных чисел. Мы оформим его как модуль в системе Turbo Pascal. Для других Pascal-компиляторов придется внести соответствующие изменения. Разработав этот пакет, мы сможем использовать его в программах моделирования или любых других приложениях, при этом нам не потребуется заглядывать внутрь пакета, чтобы узнать, как он работает. Таким образом, все детали, обсуждаемые в этом разделе (часто носящие математический характер) должны рассматриваться как часть

реализации, о которой не надо заботиться при использовании случайных чисел в приложении.

## В.2. Метод

затравка для  
псевдослучайных  
чисел

Общая идея, которой мы будем руководствоваться при генерации случайного числа, заключается в том, что мы начинаем с одного числа и выполняем последовательность арифметических операций, которая приводит к образованию другого числа, видимым образом не связанного с первым. Таким образом, образуемые нами числа не являются истинно случайными, так как каждое из них определенным образом зависит от предыдущего, и более строго следует говорить о *псевдослучайных* числах. Число, которое мы используем (и одновременно изменяем) носит название *начального числа* или *числа-затравки*.

Если при каждом запуске программы используется одна и та же затравка, тогда и вся последовательность псевдослучайных чисел будет в точности той же самой, поэтому мы обычно начинаем с того, что придаем затравке некоторое случайное значение. Для инициализации затравки мы будем использовать время суток, поскольку, скорее всего, последовательные запуски программы будут осуществляться в разное время суток.

Число-затравка будет использоваться и изменяться генератором случайных чисел, но оно не должно использоваться программой и лучше, чтобы программа вообще не имела к нему доступа. Другими словами, пользователь пакета генерации случайных чисел может и не иметь представления о существовании затравки. Таким образом, затравка не должна быть параметром программы генератора случайных чисел. Также не следует объявлять ее как глобальную переменную в пользовательской программе, и уж тем более инициализировать пользователем<sup>1</sup>.

В стандартном языке Pascal описанные нами проблемы не имеют удобного решения, но модули Turbo Pascal предоставляют превосходные средства генерации случайных чисел. Мы объявим число-затравку глобальной переменной, но только внутри *реализации* модуля. В результате затравка будет доступна все подпрограммам модуля, но к ней нельзя будет обратиться извне модуля. Более того, модуль предоставляет специальную секцию инициализации, которая выполняется автоматически при каждом запуске вызывающей программы, отчего пользователю не приходится затрачивать специальных усилий для инициализации затравки.

<sup>1</sup> Во многих случаях, особенно при отладке программ моделирования, использующих генераторы случайных чисел, псевдослучайные последовательности *должны быть* одинаковыми при многократных запусках программы (оставаясь псевдослучайными). Поэтому в наборах функций для генерации случайных чисел, включаемых в современные системы программирования, обычно имеются как средства задания числа-затравки (и тогда создается возможность генерации многих одинаковых псевдослучайных последовательностей, а также создания предсказуемых последовательностей то с одной затравкой, то с другой), так и средства «перемешивания» псевдослучайных последовательностей выбором для числа-затравки случайного значения. — Прим. перев.

## В.3. Разработка программы

Для реализации поставленных задач мы напишем главную часть модуля следующим образом. Все приведенные ниже объявления не нуждаются в пояснениях, за исключением объявления трех видов генераторов случайных чисел, которые мы в дальнейшем разработаем.

```

unit RandomGE;
interface
function RandomInt(low, high: integer): integer;
function Random: real;
function PoissonRandom(expectedvalue: real): integer;
implementation
uses
    Dos,                                     { для процедуры GetTime }
    Utility;                               { для процедуры Error }
var seed,
    hours, minutes, secs, hsec: word;
{ Сюда следует включить объявления всех процедур и функций. }
begin { Здесь начинается код инициализации, основная часть модуля. }
    GetTime(hours, minutes, secs, hsec);    { используем время
                                           для инициализации заправки }

    seed := hsec + 100 * secs + 6000 * minutes
end.                                     { конец кода инициализации и модуля }

```

Пример функции, выдающей одно псевдослучайное число на основе предыдущего вызова этой функции:

```

function NewRandom: integer;               { Новое случайное число }
{ Pre: Заправка seed содержит произвольное значение.
  Post: Значение функции представляет собой псевдослучайное число.
  Uses: Зависит от значения заправки и изменяет это значение.
        [Реализация функции доступна только внутри этого модуля] }
{ Эта функция закрыта в секции реализации этого модуля. }
const
a = 2743;
c = 5923;
begin                                     { функция NewRandom }
    seed := seed * a + c;
    NewRandom := seed
end;                                     { функция NewRandom }

```

Эта функция получает требуемый результат, выполняя умножение и сложение и затем отбрасывая старшую часть результата и оставляя только младшие, более случайные цифр. Константы *a* и *c* в этой функции не следует выбирать случайным образом; наоборот, их следует выбирать так, чтобы результаты удовлетворяли разнообразным тестам на случайность. Приведенные константы работают достаточно хорошо на 16-рядных компьютерах, но для других машин следует выбирать другие числа.

Мы не позволим программе непосредственно использовать функцию *NewRandom*; вместо этого мы будем преобразовывать получаемый из нее результат в одну из трех более практически полезных форм.

равномерное  
распределение

## 1. Действительные значения

Первая из функций имитирует генератор случайных чисел большинства компьютерных систем, возвращая результат в виде действительного числа, равномерно распределенного между 0 и 1. Под **равномерным распределением** мы понимаем, что если мы возьмем два интервала одинаковой длины, не выходящих из диапазона генерируемых функцией значений, тогда результат с равной вероятностью попадет как в один, так и в другой из выбранных интервалов. Поскольку диапазон нашей функции лежит в пределах от 0 до 1, это определение означает, что вероятность того, что результат попадет в любой подинтервал, равна длине этого подинтервала.

На диапазон генерируемых чисел обычно накладывается еще одно ограничение: 0 может появляться в качестве результата, но 1 не может. В математических терминах область случайных чисел представляет собой *полуоткрытый* интервал  $[0,1)$ .

Ниже приведена результирующая функция, которая просто преобразует результат функции NewRandom так, чтобы он лежал в требуемом диапазоне.

```
function Random: real;                                     { Случайное число }
{ Pre:  Затравка seed содержит произвольное значение.
  Post:  Функция возвращает псевдослучайное действительное число v,
         удовлетворяющее условию  $0 < v < 1$ .
  Uses:  Использует функцию NewRandom. }
const m = maxint + 1.0;                                   { константа принудительно делается
                                                         действительным числом }

var temp: real;
begin                                                     { функция Random }
  temp := NewRandom;
  if temp < 0 then
    temp := temp + m;
  Random := temp/m;
end;                                                     { функция Random }
```

## 2. Целочисленные значения

Второй часто используемой формой случайных чисел являются целые числа. Мы, однако, не можем с полным правом говорить о случайных целых числах, поскольку количество целых чисел бесконечно, а в компьютере можно образовать лишь определенное их количество. Отсюда вероятность того, что истинно случайное целое окажется таким, что его можно получить в компьютере, равно 0. Поэтому мы рассматриваем только целые числа в диапазоне между двумя целыми числами low и high, включительно. Для вычисления такого целого числа мы начинаем со случайного действительного числа из диапазона  $[0,1)$ , умножаем его на  $low - high + 1$ , поскольку именно столько чисел содержится в требуемом диапазоне, усекаем результат до целого числа и добавляем low, чтобы поместить результат в требуемый диапазон целых чисел.

```

function RandomInt (low, high: integer): integer;           { Случайное целое }
{ Pre:   Затравка seed содержит произвольное значение.
  Post:  Значение функции представляет собой псевдослучайное число,
          равномерно распределенное в диапазоне от low до high,
          включительно.
  Uses:  Использует функцию Random. }
begin                                                         { функция RandomInt }
  if low > high then
    Error('Функция RandomInt была вызвана с нижним значением,
          превышающим верхнее')
  else
    RandomInt := trunc((high - low + 1) * Random) + low
end;                                                         { функция RandomInt }

```

### 3. Значения, распределенные по закону Пуассона

Третья, более изощренная форма случайных чисел, в частности, нужна для моделирования аэропорта из раздела 5.4. Такие числа носят название Пуассоновского распределения случайных целых чисел. Мы начинаем с положительного действительного числа, называемого **математическим ожиданием**  $\nu$  в последовательности случайных чисел. Тогда если мы говорим, что последовательность неотрицательных целых чисел подчиняется распределению Пуассона с математическим ожиданием  $\nu$ , это означает, что на достаточно длинных подпоследовательностях среднее значение целых чисел в последовательности стремится к  $\nu$ .

Если, например, мы начинаем с математического ожидания 1.5, мы могли бы получить последовательность 1, 0, 2, 2, 1, 1, 3, 0, 1, 2, 0, 0, 2, 1, 3, 4, 2, 3, 1, 1, ... . Если вы будете вычислять средние значения для подпоследовательностей этой последовательности, вы обнаружите, что иногда среднее будет меньше 1.5, иногда больше, но постепенно оно будет все больше приближаться к 1.5.

Приводимая ниже функция генерирует псевдослучайные целые числа, подчиняющиеся распределению Пуассона. Разработка этого метода и доказательства правильности его работы требуют привлечения средств вычислительной математики и математической статистики, далеко выходящих за рамки этой книги, но из этого не следует, что мы не можем приложить теорию для получения нужных нам чисел. Результирующая программа выглядит следующим образом.

```

function PoissonRandom (expectedvalue: real): integer;       { Случайное
                                                                пуассоновское число }
{ Pre:   Предусловия отсутствуют.
  Post:  Генерирует случайное неотрицательное число, распределенное
          по закону Пуассона с математическим ожиданием, задаваемым
          в качестве параметра.
  Uses:  Использует функцию Random. }
var limit: real;      {  $e^{-\nu}$ , где  $\nu$  есть математическое ожидание }
    count: integer;    { число шагов }
    product: real;     { псевдослучайное число,  $0 < \text{product} < 1$  }

```



```

begin
    limit := exp(- expectedvalue);
    product := Random;
    count := 0;
    while product > limit do begin
        count := count + 1;
        product := product * Random
    end;
    PoissonRandom := count
end;

```

{ функция PoissonRandom }

{ функция PoissonRandom }

### Программные проекты В.3

- P1.** Напишите программу-драйвер для тестирования трех функций генерации случайных чисел, представленных в этом приложении. Для каждой функции вычислите среднее значение возвращаемых ею чисел на заданном числе вызовов (число вызовов должно задаваться пользователем). Для функции Random среднее значение должны быть около 0.5; для RandmInt(low, high) среднее значение должно быть приблизительно  $(low + high)/2$ , причем значения low и high задаются пользователем; для PoissonRandom среднее значение должно приблизительно равняться математическому ожиданию, задаваемому пользователем.
- P2.** Один из тестов на равномерность распределения случайных целых чисел заключается в проверке того, что все возможные значения создаются приблизительно с одной вероятностью. Создайте массив целочисленных счетчиков, получите от пользователя число используемых позиций и число испытаний, а затем генерируйте повторно случайные числа в заданном диапазоне и инкрементируйте соответствующую ячейку массива. В конце испытаний числа, находящиеся во всех ячейках массива, должны быть приблизительно одинаковыми.
- P3.** Обобщите тест предыдущего проекта, воспользовавшись прямоугольным массивом и парами случайных чисел для выбора инкрементируемой ячейки.
- P4.** В одной из детских игр два игрока одновременно выбрасывают руку, обозначая условно пальцами камень, ножницы или бумагу. Правила игры таковы, что камень побеждает ножницы (поскольку камень тупит ножницы), ножницы побеждают бумагу (поскольку ножницы режут бумагу), а бумага побеждает камень (поскольку камень можно завернуть в бумагу). Напишите программу, моделирующую такую игру, предполагая, что компьютер играет с человеком, вводящим при каждом ходе буквы К, Н или Б.
- P5.** В игре «Хамураби» вы представляете императора небольшого королевства. Вы начинаете жизнь со 100 подданными, 100 бушелями зерна и 100 акрами земли. Каждый год вы должны принимать решение о распределении этих ресурсов. Вы можете отдать подданными столько зерна для еды, сколько пожелаете, вы можете посеять столько зерна, сколько пожелаете, или вы можете купить или продать землю за зерно. Цена земли (в бушелях зерна за акр) определяется случайным образом в диапазоне от 6 до 13. Крысы будут поедать случайную долю



- (b) Модифицируйте исходную ситуацию так, что если пьяный, блуждая случайным образом, снова приходит к трактиру, он заходит в трактир, и на этом сеанс блуждания заканчивается. Определите (в зависимости от размера и формы сетки), какова вероятность для пьяного попасть домой.
- (c) Модифицируйте исходную ситуацию так, чтобы придать пьяному в помощь некоторую память следующим образом. Если пьяный подходит к перекрестку, на котором он уже побывал в этом сеансе, он вспоминает, какие улицы он уже выбирал, и пробует одну из неиспытанных. Если он уже попробовал все улицы, идущие от данного перекрестка, он выбирает дальнейший маршрут случайным образом. Насколько быстрее он попадет домой?

## Литература для дальнейшего изучения

[Knuth], том 2, стр. 1–177, (см. ссылку в главе 3) содержит одно из наиболее детальных обсуждений генераторов псевдослучайных чисел, методов их тестирования и других смежных вопросов.

# Приложение С

## Модули, включаемые файлы и утилиты

---

В этом приложении обсуждается синтаксис и использование модулей Turbo Pascal и включаемых файлов, объясняются шаги, необходимые для преобразования модулей во включаемые файлы, описываются процедуры и функции нескольких модулей, а также перечисляется состав всех модулей и других утилит, используемых по ходу этой книги.

### С.1. Модули Turbo Pascal

#### С.1.1. Введение

В Turbo Pascal **модулем** называется собрание констант, типов, переменных, процедур и функций. По внешнему виду модуль напоминает собой законченную программу со своими собственными объявлениями и своим телом, причем эта программа должна компилироваться отдельно тем же самым способом, что и наша программа, однако она не выполняется как независимый объект. Наоборот, она *используется* нашей программой (или другим модулем) для выполнения некоторой специфической задачи.

Обычно, хотя не всегда, модуль состоит из объявлений для набора тесно связанных задач. Например, в этой книге мы используем один модуль для анализа потребностей в процессорном времени при выполнении программы, другой модуль для открытия и закрытия файлов, и еще один модуль для вычисления псевдослучайных чисел. Один из модулей, называемый Utility, содержит в себе набор часто используемых процедур, многие из которых не связаны друг с другом.

Объявления в модуле делятся на две секции. В первой описывается **интерфейс**; эта секция еще называется **открытой**. После нее идет секция **реализации**; она называется **закрытой**. В интерфейсной части устанавливается, *что* делает модуль и эта часть используется вместе с любыми программами, обращающимися к модулю, в то время как реализационная часть описывает, *как* модуль выполняет возложенные на него задачи; эта часть недоступна никаким другим программам. Открытые, т. е. доступные программам процедуры и функции могут включаться в обе секции; интерфейсная часть содержит только строки заголовка (имя и параметры) для этих процедур и функций; секция реализации содержит их полные объявления. В модуле могут быть еще процедуры и функции, используемые только внутри самого модуля; они могут входить только в секцию реализации.

функции

интерфейс  
и реализации

сокрытие  
информации

смена реализации

Используя отдельные секции интерфейса и реализации, модуль эффективно навязывает принципы сокрытия информации. Модули особенно удобны для обработки разнообразных структур данных. В разделе 2.2.1, например, даются спецификации операций для простого списка, но ничего не говорится об их реализации. Интерфейсная секция модуля для простых списков содержит только эти самые спецификации. Секция же реализации будет содержать все детали каждой из этих операций. Эти детали могут быть взяты или из непрерывной реализации простых списков (раздел 2.3.1), или из связанной реализации, рассматриваемой в приложении D. Поскольку интерфейсная секция при этом никак не изменяется, программе, использующей модуль простых списков, не требуется знать, какая из возможных реализаций используется. Фактически, если реализации запрограммированы должным образом, программа вообще никаким образом не может узнать, какая реализация содержится в модуле (разве что только путем измерения скорости выполнения различных операций).

### С.1.2. Синтаксис модулей

Программа, использующая один или несколько модулей, включает предложение в такой форме:

```
uses unit1, unit2, и т. д.;
```

где unit1, unit2, и т. д. представляет собой перечисление имен модулей. Предложение должно включаться в программу непосредственно после заголовка программы. При наличии этого предложения весь состав интерфейсной секции данного модуля (константы, типы, переменные, процедуры, функции) становится доступен программе, как если бы все это было объявлено в самой программе.

Предложение **uses** должно быть в *главной* программе, не в процедуре или функции. Процедуры и функции, тем не менее, имеют доступ к модулям в точности так же, как они имеют доступ ко всему, объявленному глобально в программе. Модули могут использовать другие модули путем включения предложений **uses** двумя способами. Если модуль перечислен в интерфейсной части, тогда он может использоваться в объявлениях интерфейса. Если он перечислен только в реализационной части, он может быть использован там, но его использование будет скрыто от любых программ, использующих первый модуль.

Синтаксис модуля весьма похож на синтаксис любой программы, за исключением второго набора объявлений. Скелетная схема модуля выглядит следующим образом:

```
unit unitname;
interface    { этим начинается открытая секция; обязательная часть }
uses unitlist; { необязательный список модулей, разделяемых запятыми }
const        { каждая из этих секций объявлений необязательна }
    { открытые константы объявляются здесь };
type
    { открытые типы объявляются здесь };
var
    { открытые переменные объявляются здесь };
```

```

procedure A(параметры);
    { заголовки только для открытых процедур и функций]
function B(параметры): тип_функции;
procedure C(параметры);
implementation                                { этим начинается закрытая секция;
                                                обязательная часть }
{ Все, объявленное в интерфейсной части, видимо здесь и не должно
  объявляться повторно}
uses unitlist;                                { необязательный список из одного или нескольких
                                                модулей, разделяемых запятыми]
const                                { каждая из этих секций объявлений необязательна }
    { закрытые константы объявляются здесь }
type
    { закрытые типы объявляются здесь };
var
    { закрытые переменные объявляются здесь };
procedure A;
{ Полные объявления всех процедур и функций из интерфейсной части
  должны быть даны здесь. Список параметров может быть повторен,
  но это не обязательно; если он повторен, он должен быть идентичен
  тому, что дан в интерфейсной части. }
function B(параметры): тип_функции;
    { полное объявление };
procedure X(параметры);                    { закрытые процедуры и функции]
{ Полные объявления дополнительных процедур и функций могут быть
  размещены здесь вместе с открытыми процедурами и функциями. }
procedure C(параметры);
    { полное объявление };
begin                                { не обязательно: включайте begin только
                                    если требуется инициализация }
{ Предложения инициализации размещаются здесь }
end.                                { требуется как указание на конец модуля }

```

инициализиру-  
ющий код

Часть инициализации (с ключевым словом **begin**) не является обязательной. Если она включена в модуль, то эта часть будет выполнена только однажды, активизируясь автоматически, когда использующая модуль программа начинает свое выполнение. Инициализирующая часть удобна в тех случаях, когда модуль содержит закрытые переменные, нуждающиеся в присвоении им конкретных значений перед тем, как вызывающая программа начнет работать. Так, модуль случайных чисел из приложения В включает закрытую переменную с именем *seed*, которая должна быть инициализирована.

С другой стороны, для модуля обслуживания простых списков было бы неудачным решением заменить процедуру *CreateList* кодом инициализации в модуле. Напротив, место под один или несколько списков объявляется в *использующей* программе и должно быть инициализировано именно там путем вызова *CreateList*. Использующая программа может объявить несколько списков и инициализировать их. При наличии инициализирующего кода в модуле придется заранее определить, сколько списков требуется использовать, в результате чего будет потеряна значительная часть гибкости модуля.

имена модулей

Ради простоты использования имя файла, содержащего модуль, должно быть тем же самым, что и имя, присвоенное модулю. В этом случае компилятору Turbo Pascal, компилирующему программу, будет легче найти модуль. По этой причине модули, предназначенные для работы в DOS, должны иметь имена не длиннее восьми символов.

## С.2. Включаемые файлы

Другим способом упаковки объявлений, процедур и функций, чтобы их могли использовать различные программы, является директива компилятора **включения** файлов `include (I)`. В Turbo Pascal ее синтаксис таков:

```
{ $I filename }
```

Большинство компиляторов языка Pascal предоставляют аналогичное средство, хотя синтаксис, как правило, для разных компиляторов различается. Эта директива включения заставляет компилятор скопировать содержимого файла с именем `filename` в программу, в то ее место, где указана эта директива. Включаемый файл может содержать любые предложения, которые компилятор может воспринять в данном месте.

Заметьте, прежде всего, что включаемые файлы и модули совсем не являются одним и тем же. Модуль компилируется отдельно, должен иметь определенную структуру, удовлетворять правилам синтаксиса Turbo Pascal и полностью объявить все идентификаторы перед их использованием. Модуль удобен для навязывания принципов сокрытия информации и абстракции данных, а также и для реализации других целей (например, разделения очень большой программы на части разумного размера). Включаемый файл ни имеет ни этих ограничений, ни достоинств: это просто фрагмент кода, который можно скопировать куда угодно. Тем не менее, включаемые файлы имеют важные области использования и иногда определенные преимущества перед модулями.

### С.2.1. Замена модулей включаемыми файлами

Почти все Pascal-компиляторы предоставляют те или иные средства, эквивалентные директиве *include*, в то время как модули уникальны для Turbo Pascal и недоступны в других системах. Если вам приходится программировать с другим Pascal-компилятором, вы должны все модули преобразовать во включаемые файлы. Общий метод выполнения этой операции заключается в следующем.

1. Убедитесь, что нигде в вашей программе или в используемых ею модулях нет повторяющихся имен.
2. Возьмите код инициализации из модуля и поместите его в начало вашей главной программы (или в процедуру, вызываемую при запуске вашей главной программы).
3. Оставшуюся часть модуля, после редактирования специфичных для Turbo Pascal ключевых слов, и объединения интерфейсной и реализационной секций, поместите во включаемый файл.
4. Включите в программу директиву *include* для копирования этого файла в глобальные объявления вашей программы.



Разумеется, после трансляции включаемый файл больше не скрывает какую-либо информацию от программы пользователя; он содержит дополнительные глобальные объявления, что затруднит отладку и поддержку программы по сравнению с программой, использующей модули. Однако программа становится переносимой на компиляторы, отличные от Turbo Pascal.

### С.2.2. Родовые средства

Одна прикладная программа может использовать список, элементами которого являются целые числа; другая программа может использовать список с символьными элементами; третьей программе могут быть нужны элементы с действительными числами или данными какого-то другого типа. Было бы крайне желательно иметь возможность компилировать *родовой* модуль обработки списка, который включает все операции над списками, но оставляет тип `listentry` не специфицированным. Тогда прикладная программа могла бы сама специфицировать тип `listentry` нужным ей образом и использовать *родовой* модуль списков. Это, к сожалению, невозможно. Фундаментальным требованием языка Pascal является обязательное объявление любого идентификатора перед его использованием. Таким образом, нет способа, даже в Turbo Pascal, откомпилировать модуль для операций со списками перед тем, как тип `listentry` полностью специфицирован. Поэтому для решения нашей проблемы нам придется прибегнуть к другому, менее удовлетворительному методу.

Более современные языки, вроде Ada и C++, *поддерживают* раздельную компиляцию родовых пакетов, в остальном сходных с модулями. В этих языках, следовательно, проблемы абстракции данных, сокрытия информации и родовых конструкций могут быть решены более элегантно, чем это можно сделать в любом диалекте языка Pascal.

Метод (несколько неудовлетворительный) решения этих проблем заключается в комбинировании использования модулей и включаемых файлов. Мы помещаем в модуль только минимальную информацию, необходимую для спецификации типа элементов и (для непрерывных структур данных) максимальный размер. Все остальные объявления помещаются в отдельный файл, который мы включаем в модуль для компиляции.

Например, модуль для обработки (непрерывного) простого списка целых чисел выглядит так:

```
unit IntSL;           { модуль Turbo Pascal для простых непрерывных  
                      списков целых чисел }  
interface  
const maxiist = 20;   { зависит от приложения }  
type listontry = integer;  
($I simplist.inc )  
end.                 { конец модуля: ключевое слово implementation  
                      указано во включаемом файле }
```



Этот модуль определяет константу `maxlist` и тип `listentry`, но все остальное включается в файл `simplist.inc`, для которого расширение `inc` означает включаемый файл. Этот последний файл (за исключением объявления процедур и функций) выглядит следующим образом:

```
{ Упрощенный пакет обработки непрерывных списков. }
{ Этот файл завершает объявление упрощенного пакета обработки
  списков частью, общей для всех типов элементов списков. Файл
  должен быть включен в такие модули Turbo Pascal как CharSL.pas,
  RealSL.pas, IntSL.pas или в модуль, определенный пользователем.
  Константа maxlist и тип listentry должны быть объявлены в этом
  модуле. }
type
  listindex = 1..maxlist;
  listcount = 0..maxlist;
  list = record
    count: listcount;
    entry: array [listindex] of listentry;
  end;
processlistentry = procedure(var x: listentry); { специфично для Turbo Pascal}
procedure CreateList(var L: list);
procedure ClearList(var L: list);
function ListEmpty(var L: list): Boolean;
function ListFull(var L: list): Boolean;
function ListSize( var L: list): listcount;
procedure AddList(x: listentry; var L: list);
procedure TraverseList(var L: list; Visit: processlistentry);
implementation
uses Utility;
{ Полные объявления перечисленных выше процедур и функций должны
  размещаться здесь. }
```

Такой подход, хотя и требует от нас вести счет нескольким файлам, дает ту гибкость, которая необходима для изменения типов данных и размеров структур. Кроме того, в программу вносится модульность и сокрытие информации, что позволяет нам без труда изменять реализацию типов данных. Например, для того, чтобы заменить целочисленные элементы списка на символы, вместо `intSL` мы используем следующее:

```
unit CharSL; { модуль Turbo Pascal для простых непрерывных
               списков символов }

interface
const maxlist = 20; { зависит от приложения }
type listentry = char;
{ $I simplist.inc }
end.
```

Если мы захотим использовать вместо непрерывной реализации связную, мы просто изменяем модуль следующим образом:

```
unit IntSLL; { модуль Turbo Pascal для простых связных
              списков целых чисел }

interface
type listentry = integer;
{ $I simplis.inc }
end.
```

(Заметьте, что этот модуль стал даже короче, так как теперь нет необходимости объявлять `maxlist`.)

## С.3. Модули, используемые в тексте

### С.3.1. Структуры данных

Большинство модулей и включаемых файлов, разработанных в этой книге, предназначены для различных изучаемых нами структур данных. Эти модули совершенствуются и модифицируются (часто в виде упражнений и проектов) в различных разделах книги по мере ввода в рассмотрение новых структур данных. На рис. С.1 перечислены включаемые файлы с рекомендуемыми именами, а также ссылки на разделы этой книги. Скомпоновать соответствующие модули, как это описано в предыдущем разделе, обычно не составляет труда.

|                            |                              |                                     |
|----------------------------|------------------------------|-------------------------------------|
| <code>SimplList.inc</code> | Раздел 2.3.1                 | Простой непрерывный список          |
| <code>SimplLLis.inc</code> | Раздел D.3.4                 | Простой связный список              |
| <code>Stack.inc</code>     | Раздел 3.1.5                 | Непрерывный стек                    |
| <code>LinkedSt.inc</code>  | Раздел 3.1.6                 | Связный стек                        |
| <code>LookAhea.seg</code>  | Раздел 4.2                   | LookAhead для игр                   |
| <code>Queue.inc</code>     | Раздел 5.2                   | Непрерывная очередь                 |
| <code>LinkedQu.inc</code>  | Раздел 5.5                   | Связная очередь                     |
| <code>List.inc</code>      | Раздел 6.2.1                 | Обобщенный список, непрерывный      |
| <code>LinkedLi.inc</code>  | Раздел 6.2.2                 | Обобщенный связный список           |
| <code>DLL.inc</code>       | Раздел 6.2.4                 | Дважды связный список без заголовка |
| <code>DLLhead.inc</code>   | Упражнение E5(a), раздел 6.2 | Дважды связный список с заголовком  |
| <code>ArrayLL.inc</code>   | Раздел 6.5                   | Простой связный список в массиве    |
| <code>ArrayDLL.inc</code>  | Упражнение E5, раздел 6.5    | Дважды связный список в массиве     |
| <code>BST.inc</code>       | Раздел 10.2                  | Дерево двоичного поиска             |
| <code>AVL.inc</code>       | Раздел 10.4                  | AVL-дерево                          |
| <code>Splay.inc</code>     | Раздел 10.5                  | Скошенное дерево                    |
| <code>Trie.inc</code>      | Раздел 11.2                  | Трай-дерево                         |
| <code>BTree.inc</code>     | Раздел 11.3                  | В-дерево                            |
| <code>RedBlack.inc</code>  | Раздел 11.4                  | Красно-черное дерево                |

**Рис. С.1.** Включаемые файлы для структур данных



```
function IsLowerCase (ch; char): Boolean;  
{ Pre: Предусловия отсутствуют.  
Post: Возвращает true, если ch является строчной буквой,  
в противном случае возвращает false. }  
begin  
IsLowerCase := (Ord(ch) >= Ord('a')) and (Ord(ch) <= Ord('z'))  
end; { функция IsLowerCase }  
  
procedure UpperCase (var ch: char);  
{ Pre: Параметр ch содержит символ.  
Post: Если ch является строчной буквой, тогда ch преобразуется  
в прописную букву, в противном случае ch остается  
без изменений.  
Uses: Использует функцию IsLowerCase. }  
begin  
if IsLowerCase(ch) then ch := chr(ord(ch) – ord('a') + ord('A'))  
end; { процедура Uppercase }  
  
procedure LowerCase (var ch: char);  
{ Pre: Параметр ch содержит символ.  
Post: Если ch является прописной буквой, тогда ch преобразуется  
в строчную букву, в противном случае ch остается  
без изменений.  
Uses: Использует функцию IsUpperCase. }  
begin  
if IsUpperCase(ch) then ch := chr(ord(ch) – ord('A') + ord('a'))  
end; { процедура LowerCase }  
  
function CharToDigit (ch: char): integer;  
{ Pre: Параметр ch содержит символ, являющийся одной  
из десятичных цифр.  
Post: Возвращает целочисленное значение, представляемое  
символом ch.  
Uses: Использует процедуру Error }  
begin { функция CharToDigit }  
if ch in ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'] then  
CharToDigit := Ord(ch) – Ord('0')  
else  
Error('Не-цифра не может быть преобразована в целое число')  
end; { функция CharToDigit }  
  
function DigitToChar (num: integer): char;  
{ Pre: Параметр num содержит целое число от 0 and 9  
включительно.  
Post: Параметр num преобразуется из целого числа в символ.  
Uses: Использует процедуру Error }  
begin { функция DigitToChar }  
if (num > 9) or (num < 0) then  
Error('Число выходит из диапазона, в котором возможно  
преобразование в символ')  
else  
DigitToChar := Chr(num + Ord('0'));  
end; { function DigitToChar }
```

Заметьте, что процедуры для преобразования символов написаны так, чтобы они были максимально более переносимыми с одной компьютерной системы на другую. Тот же код будет работать правильно и на ASCII, и на EBCDIC-компьютерах (см. приложение D).

### С.3.3. Модуль анализа процессорного времени

SetTimer

TotalTime

ElapsedTime

При сравнении различных алгоритмов и структур данных часто оказывается полезным сравнить, сколько времени компьютер тратит на одну программу или одну фазу программы в сравнении с другой. Для этой цели мы разработали этот модуль. Процедура SetTimer начинает отсчет времени; для активизации процедуры SetTimer мы используем инициализирующую часть модуля, поэтому, если вызывающая программа сама не вызывает SetTimer, отсчет времени начнется вполне правильно с момента, когда программа начинает выполнение. Функция TotalTime возвращает время процессора, использованное с момента начала выполнения или последнего вызова процедуры SetTimer, в зависимости от того, что произошло позже. Функция ElapsedTime возвращает время, протекшее от последнего вызова ElapsedTime, или то же, что и TotalTime, если предшествующих вызовов ElapsedTime не было. В результате действие функции ElapsedTime напоминает работу обычного секундомера.

Определение времени не является частью стандартного языка Pascal; поэтому, хотя многие Pascal-системы представляют такие средства, способ их использования отличается для разных систем. Разработанный нами модуль специфичен для Turbo Pascal, но, собрав все фрагменты кода в одно место, его можно при необходимости модифицировать для использования на другой системе.

Полный текст модуля CPUtimer приведен ниже. Этот модуль стоит изучения, чтобы посмотреть, как в секции реализации объявляются глобальные переменные и функция Clock, в результате чего они становятся доступны всюду внутри модуля, но скрыты от прикладной программы.

*{ Этот модуль содержит функции и процедуры, используемые для определения времени центрального процессора, используемого частями программы. Таймер имеет границу временного диапазона 24 часа.*

|             |                                                                                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------|
| SetTimer    | Устанавливает начальное значение времени; это делается при запуске программы.                                                            |
| ElapsedTime | Определяет разницу во времени от момента установки начального времени или момента последнего вызова функции ElapsedTime.                 |
| TotalTime   | Определяет разницу во времени от момента установки начального времени или момента последнего вызова функции SetTimer и текущим временем. |

```

}
unit CPUtimer;
interface
procedure SetTimer;
function ElapsedTime: real;
function TotalTime: real;
```

**implementation**

**uses** Dos; { предоставляет специфичную для Turbo Pascal  
процедуру GetTime }

**const**

MaxTime = 8640000; { число сотых долей секунды в 24 часах }

**var**

InitialClockTime: real;  
LastCheckClockTime: real;  
TotalClockTime: real;

{ Замечание: Эти переменные глобальны для модуля процессорного времени, однако они должны быть недоступны для остальной части программы. }

**function** Clock: real; { Часы; функция, закрытая в модуле }

{ **Pre:** Предусловия отсутствуют. }

**Post:** Возвращает значение времени в виде 32-битового целого числа.

**Uses:** Использует процедуру GetTime }

**var** hour, minute, second, sec100: word;

**begin** { функция Clock }

GetTime(hour, minute, second, sec100);  
Clock := (hour \* 360000.0) + (minute \* 6000.0) + (second \* 100.0) +  
sec100;

**end;** { функция Clock }

**procedure** SetTimer; { Установить таймер }

{ **Pre:** Предусловия отсутствуют. }

**Post:** Время центрального процессора при вызове сохраняется в глобальных переменных ради возможности дальнейших ссылок. }

**begin** { процедура SetTimer }

InitialClockTime := Clock;  
LastCheckClockTime := InitialClockTime;  
TotalClockTime := InitialClockTime

**end;** { процедура SetTimer }

**function** ElapsedTime: real; { Прошедшее время }

{ **Pre:** Предусловия отсутствуют. }

**Post:** Возвращает время центрального процессора, истекшее с момента последнего вызова функции. }

**var** temp: real; { вспомогательная переменная для интервала времени }

**begin** { функция ElapsedTime }

temp := Clock – LastCheckClockTime; { преобразуем в секунды }

LastCheckClockTime := Clock;

**if** temp < 0.0 **then**  
EapsedTime := (MaxTime + temp)/100.0 { от 23:55:59 к 00:00:00 }

**else**  
ElapsedTime := temp/100.0 { истинное время не возвращается к 0 в полночь }

**end;** { функция ElapsedTime }

```

function TotalTime: real;                                { Полное время }
{ Pre:  Предусловия отсутствуют.
  Post: Возвращает истекшее время центрального процессора после
          первого запуска программы или после повторного вызова
          функции SetTimer. }
var temp: real;    { вспомогательная переменная для интервала времени }
begin
    temp := Clock – TotalClockTime;                        { функция TotalTime }
    if temp < 0.0 then                                     { Convert to seconds. }
        TotalTime := (MaxTime + temp)/100.0                { от 23:59:59 к 00:00:00 }
    else
        TotalTime := temp/100.0;                          { истинное время не возвращается к 0
                                                             в полночь }
    end;                                                    { функция TotalTime }

begin                                                      { начинается инициализация главного модуля
                                                             при запуске программы }
    SetTimer                                                { позволяет установить таймер
                                                             в процессе запуска программы }
end.

```

### С.3.4. Модуль для обслуживания файлов

Методы доступа к файлам (кроме стандартного ввода и вывода посредством терминала), к сожалению, различаются в разных Pascal-системах. В модуле обработки файлов мы собрали несколько процедур, полезных для ввода-вывода файлов. Если программа должна быть перемещена на другую систему, будет проще изменить процедуры обработки файлов, если они собраны в одном месте.

```

{ Этот модуль содержит различные функции и процедуры,
  используемые для файлового ввода-вывода:
  OpenInfile      Открывает существующий текстовый файл
                   и назначает имя файла переменной infile.
  OpenOutfile     Открывает текстовый файл, который может
                   существовать или не существовать. Если файл
                   существует, процедура запрашивает
                   разрешение на перезапись файла, в противном
                   случае файл создается. Существующий теперь
                   файл открывается, и его имя присваивается
                   переменной outfile.
  CloseInfile     Входной файл закрывается.
  CloseOutfile    Выходной файл закрывается.
  OpenExtensionInfile  Открывается существующий входной файл
                   с расширением по умолчанию.
  OpenExtensionOutfile  Открывается выходной файл с расширением
                   по умолчанию. Если файл существует,
                   процедура запрашивает разрешение на его
                   перезапись.

unit FileIO;
interface
procedure OpenInfile(var infile: text);
procedure OpenOutfile(var outfile: text);
procedure OpenExtensionInfile(var infile: text; extension: string);

```

```

procedure OpenExtensionOutfile(var outfile: text; extension: string);
procedure CloseInfile(var infile: text);
procedure CloseOutfile(var outfile: text);
implementation
uses Utility;

```

```

procedure OpenInfile (var infile: text);           { Открыть входной файл }
{ Pre:  Предусловия отсутствуют.
  Post:  Открывает существующий текстовый файл и назначает имя
         файла переменной infile.
  Uses:  Использует процедуры Error, Assign, Reset, IOresult. }
var infilename: string;
      OK: Boolean;                                { файл открылся нормально? }
begin                                           { процедура OpenInfile }
  repeat    { повторять попытки, пока файл не откроется нормально }
    write('Имя входного файла? ');
    readln(infilename);
    Assign(infile, infilename);                  { специфично для Turbo Pascal }
    { $I - }                                     { отменить обработку ошибок в Turbo Pascal }
    Reset(infile);
    { $I + }                                     { восстановить обработку ошибок в Turbo Pascal }
    OK := (IOresult = 0);
    if not OK then Error('Входной файл не может быть открыт.')
```

```

  until OK;

```

```

end;                                           { процедура OpenInfile }

```

```

procedure OpenOutfile (var outfile: text);       { Открыть выходной файл }
{ Pre:  Предусловия отсутствуют.
  Post:  Открывает текстовый файл, который может существовать или
         не существовать. Если файл существует, процедура
         запрашивает разрешение на перезапись файла. Если файл
         не существует, процедура создает файл. Существующий
         теперь файл открывается, и его имя присваивается
         переменной outfile.
  Uses:  Использует процедуры Error, Assign, Reset, IOresult, Rewrite,
         UserSaysYes. }

```

```

var
  outfilename: string;
  OK: Boolean;                                { файл открылся нормально? }
begin                                           { процедура OpenOutfile }
  repeat    { повторять попытки, пока файл не откроется нормально }
    write('Имя выходного файла? ');
    readln(outfilename);
    Assign(outfile, outfilename);                { специфично для Turbo Pascal }
    { $I - }                                     { отменить обработку ошибок в Turbo Pascal }
    Reset(outfile);
    { $I + }                                     { восстановить обработку ошибок в Turbo Pascal }
    OK := (IOresult = 0);
    if OK then begin
      write('Выходной файл уже существует. Хотите его перезаписать? ');
      OK := UserSaysYes;
      if OK then Rewrite(outfile);
    end

```



```

    else begin
        { $I - }           { отменить обработку ошибок в Turbo Pascal }
        Rewrite(outfile);
        { $I + }           { восстановить обработку ошибок в Turbo Pascal }
        OK := (IOresult = 0);
        if not OK then
            writeln('Не могу открыть выходной файл с этим именем.')
        end
    until OK
end;                                     { процедура OpenOutfile }

procedure CloseInfile (var Nile: text);           { Закрывает входной файл }
{ Pre: Файл infile уже открыт.
  Post: Закрывает существующий текстовый файл. }
begin
    Close(infile);
end;                                     { процедура CloseInfile }

procedure CloseOutfile(var outfile: text);       { Закрывает выходной файл }
{ Pre: Файл outfile уже открыт.
  Post: Закрывает существующий текстовый файл. }
begin
    Close(outfile);
end;                                     { процедура CloseOutfile }

procedure OpenExtensionInfile (var infile: text; extension: string);
                                     { Открыть входной файл с расширением }
{ Pre: Параметр extension содержит строку с расширением
    по умолчанию, используемым для открытия файла.
  Post: Файл открыт, и ему присвоено имя infile. }
var infilename: string;
    dotplace: integer;
    OK: Boolean;
begin                                     (процедура OpenExtensionInfile )
    repeat                               { повторять попытки, пока файл не откроется }
        write('Введите имя открываемого файла, ');
        write('(расширение ', extension, ' по умолчанию): ');
        readln(infilename);
        dotplace := Pos('.', infilename);
        if dotplace = 0 then infilename := infilename + extension;
        Assign(infile, infilename);
        { $I - } Reset(infile); { $I + }
        OK:=(IOresult = 0);
        if not OK then Error('Входной файл не может быть открыт. ');
    until OK;
end;                                     { процедура OpenExtensionInfile }

procedure OpenExtensionOutfile (var outfile: text; extension: string);
                                     { Открыть выходной файл с расширением }
{ Pre: Параметр extension содержит строку с расширением
    по умолчанию, используемым для открытия файла.
  Post: Файл открыт, и ему присвоено имя outfile. }

```

```

var outfilename: string;
OK: Boolean;                                { файл открылся нормально? }
dotplace: integer;
begin                                       { процедура OpenExtensionOutfile }
  repeat { повторять попытки, пока файл не откроется нормально }
    write ('Введите имя выходного файла, ');
    write('расширение ', extension, ' по умолчанию:');
    readln(outfilename);
    dotplace := Pos('.', outfilename);
    if dotplace = 0 then outfilename := outfilename + extension;
    Assign(outfile, outfilename);          { специфично для Turbo Pascal }
    { $I - } Reset(outfile); { $I + }
    OK := (IOresult = 0);
    if OK then begin
      write('Выходной файл уже существует. Хотите перезаписать его? ');
      OK := UserSaysYes;
      if OK then Rewrite (outfile);
    end
    else begin
      { $I - } Rewrite(outfile); { $I + }
      OK := (IOresult = 0)
    end;
  until OK
end;                                       { процедура OpenExtensionOutfile }
end.                                     { Модуль FileIO }

```

### С.3.5. Модуль случайных чисел

Этот модуль полностью описан в приложении В, и все его функции перечислены там же. Модуль случайных чисел в особенности достоин рассмотрения, поскольку он, как и модуль анализа процессорного времени, включает инициализирующий код, глобальную переменную и закрытую функцию.

## С.4. Программы поиска и сортировки

Ниже описываются две программы общего назначения, весьма полезные для поиска, сортировки и извлечения информации.

### С.4.1. Демонстрационная программа

Программа S&SDemo является демонстрационной программой, иллюстрирующей многие методы поиска и сортировки, изучаемые в этой книге. Демонстрация осуществляется на очень маленьких наборах данных, но по ходу ее выполнения подробно поясняется каждый шаг процесса с привлечением графического отображения получаемых результатов. Программу можно выполнять с остановкой на каждом шаге или запустить в непрерывном режиме, когда она выполняется от начала до конца, показывая при этом перемещения данных в соответствии с демонстрируемым алгоритмом. В качестве входных данных используются псевдослучайные числа, так что последовательные запуски приводят к различающимся результатам.

Программа демонстрирует следующие методы:

- Последовательные поиск
- Двоичный поиск, вариант с забыванием
- Двоичный поиск, второй вариант
- Сортировку включением
- Сортировку выбором
- Пузырьковую сортировку
- Сортировку слиянием
- Быструю сортировку
- Пирамидальную сортировку

*Благодарность:* программа S&SDemo была написана Кори П. Йитманом (Corey P. Yeatman) из Крайстчерч, Новая Зеландия, который любезно согласился на включение своей программы в эту книгу.

### С.4.2. Создание файлов данных для тестирования программ

Вторая программа, названная DataMake, создает файлы с целыми числами в форме, удобной для тестирования и сравнения различных программ поиска, сортировки и извлечения информации. Для тестирования и сравнения этих программ с помощью программы DataMake удобно создать небольшой набор файлов различного размера и с различным порядком данных, а затем с помощью модуля анализа процессорного времени сравнивать различные программы, запуская их с одним и тем же файлом входных данных.

Программа DataMake создает файлы любого размера вплоть до 15000 положительных целых чисел, упорядоченных различным образом. Порядок данных в файлах может быть одним из следующих:

- Случайный порядок, имеются дублирующие ключи
- Случайный порядок без дублирующих ключей
- Возрастающий порядок, имеются дублирующие ключи
- Возрастающий порядок, без дублирующих ключей
- Убывающий порядок, имеются дублирующие ключи
- Убывающий порядок, без дублирующих ключей
- Почти, но не полностью упорядоченные данные, имеются дублирующие ключи
- Почти, но не полностью упорядоченные данные, без дублирующих ключей

Почти упорядоченные данные могут быть изменены посредством ввода пользователем своих данных.

Эта программа при создании файлов с неупорядоченными данными использует псевдослучайные числа. Исходный текст программы стоит изучить в качестве примера реальной программы среднего размера, использующей несколько модулей, несколько подпрограмм, управляемых с помощью меню, а также несколько фрагментов кода общего назначения,

которые можно использовать в других программах. Ниже приведен полный исходный текст программы DataMake.

```

program DataMake (input, output, outfile);           { Создать файл данных }
{ Pre:  Предусловия отсутствуют.
  Post: Создает файл, состоящий из целых чисел от 1 до заданной
          границы. Могут быть созданы три различных вида файлов
          с тестовыми данными, как с дублирующими ключами, так и без
          дублирования ключей: случайные числа с дубликатами,
          случайные числа без дубликатов, числа в возрастающем
          порядке с дубликатами, возрастающий порядок без дубликатов,
          числа в убывающем порядке с дубликатами, убывающий порядок
          без дубликатов, почти упорядоченные числа с дубликатами,
          почти упорядоченные числа без дубликатов. Почти
          упорядоченные последовательности могут быть изменены
          путем ввода пользователем конкретных данных.
  Uses: Использует модули Utility, RandomGe, FileIO; процедуры
          WriteRandom, WriteDulpicateRandom, WritelnIncreasing,
          WriteDuplicateIncreasing, WriteDecreasing, WriteDuplicateDecreasing,
          WriteNearRandom, WriteDuplicateNearRandom. }

uses
  Utility,      { специфичный для Turbo Pascal откомпилированный модуль }
  RandomGe,     { пакет генерации случайных чисел }
  FileIO;       { пакет файлового ввода-вывода }

const
  maxarray = 15000;      { максимальный размер массива целых чисел }

type
  filetype = text;
  datarun = array [1..maxarray] of integer; { массив, содержащий данные }

var
  outfile: filetype;      { файл, в который записываются данные }
  size: integer;          { количество целых чисел, помещаемых в файл }
  order: char; duplicate: char;
  validcommands: set of char; { символы, допустимые в качестве команд
                                (константы) }
  Buffer: datarun;        { глобальный буфер для манипуляций с данными }

begin                                     { главная программа DataMake }
  Introduction;
  repeat
    write('Пожалуйста, введите длину выходного файла: ');
    readln(size);
    OpenOutFile(outfile);
    repeat
      writeln('Пожалуйста, введите способ упорядочения файла:
      writeln(' [R] случайный, [I] возрастающий');
      write(' или [D] убывающий, [N] Почти упорядоченный :');
      readln(order);
      LowerCase(order);
      write{ 'Хотите иметь в файле ключи-дубликаты? (у или n): '};
      readln(duplicate);
      LowerCase(duplicate);
    until order in validcommands;

```

```

case order of
  'i': if duplicate = 'y' then
    WriteDuplicateIncreasing(size)
  else
    WriteIncreasing(size);
  'd': if duplicate = 'y' then
    WriteDuplicateDecreasing(size)
  else
    WriteDecreasing(size);
  'r': if duplicate = 'y' then
    WriteDuplicateRandom(size)
  else
    WriteRandom(size);
  'n': if duplicate = 'y' then
    WriteDuplicateNearRandom(size)
  else
    WriteNearRandom(size);
end;
WriteOutfile(size, outfile);
CloseOutfile(outfile);
write('Хотите создать другой файл данных ?')
until not UserSaysYes
end.                                     { главная программа DataMake }

procedure Introduction;                                     { Введение }
{ Pre: Предусловия отсутствуют.
  Post: На экран выводится текст введения к подпрограмме,
    и инициализируется глобальная переменная validcommands. }
begin                                     { процедура Introduction }
  writeln('          Программа создания файла с данными ');
  writeln(' Эта программа создает файлы, содержащие целые числа');
  writeln(' в качестве данных. Эти файлы данных используются');
  writeln(' в различных приложениях, например, для организации списка,');
  writeln(' который будет затем упорядочиваться посредством различных');
  writeln(' методов упорядочения. Другая область использования');
  writeln(' программы – различные проекты извлечения данных,');
  writeln(' в которых необходимо выполнять разнообразные операции');
  writeln(' поиска, включения и удаления в деревьях разного рода,');
  writeln(' списках и других абстрактных типах даных. ');
  writeln(' Файлы с данными, будучи созданы, позволяют тестировать ');
  writeln(' все приложения с одними и теми же наборами данных,');
  writeln(' получая тем самым сравнимые результаты. ');
  writeln(' Эта программа может создавать файлы, содержащие ');
  writeln(' до', maxarray: 1, ' элементов' );
  validcommands := ['i', 'r', 'd', 'n'];
end;                                     { процедура Introduction }

function MaxKey(low, high: integer): integer;               { Максимальный ключ }
{ Pre: Параметры low и high являются допустимыми позициями
  элементов в глобальном массиве Buffer.
  Post: Сканирует глобальный массив Buffer в поисках максимального
    ключа. }
var m, j: integer;                                         { m есть индекс максимального
    к настоящему моменту ключа }

```

```

begin
    m := low;
    for j := low + 1 to high do
        if Buffer[m] < Buffer[j] then m := j;
    MaxKey := m
end;
{ функция MaxKey }

procedure Swap(x, y: integer);
{ Обмен }
{ Pre: Параметры x и y являются допустимыми позициями элементов
      в глобальном массиве Buffer.
  Post: Обменивает местами элементы, на которые указывают
        индексы x и y. }
var t: integer;
begin
    t := Buffer[x];
    Buffer[x] := Buffer[y];
    Buffer[y] := t
end;
{ процедура Swap }

procedure SelectSort(size: integer);
{ Сортировка выбором }
{ Pre: Глобальный массив Buffer содержит допустимые элементы,
      а параметр size содержит число элементов в глобальном
      массиве Buffer.
  Post: Упорядочивает глобальный массив buffer в порядке возрастания.
  Uses: Использует функцию MaxKey, процедуру Swap. }
var
    i, m: integer;
begin
    for i := size downto 2 do begin
        m := MaxKey(1, i);
        Swap(m, i)
    end
end;
{ процедура SelectSort }

procedure WriteOutFile(size: integer;
                        var outfile: filetype);
{ Записать выходной файл }
{ Pre: Глобальный массив Buffer содержит правильным образом
      организованные данные, как это было затребовано
      пользователем. Текстовый файл outfile инициализирован.
  Post: Выводит элементы глобального массива Buffer в выходной
        файл outfile. }
var i: integer;
begin
    for i := 1 to size do
        writeln(outfile, buffer[i]);
end;
{ процедура WriteOutFile }

procedure Initialize(var size: integer);
{ Инициализировать }
{ Pre: Предусловия отсутствуют.
  Post: Заполняет буфер Buffer целыми числами от 1 до size. }
var i: integer;
begin
    for i := 1 to size do
        buffer[i] := i;
end;
{ процедура Initialize }

```

```

procedure WriteIncreasing(size: integer);           { Записать в возрастающем
                                                    порядке }
{ Pre: Параметр size содержит число элементов, которые следует
    поместить в выходной файл outfile.
  Post: Инициализирует глобальный массив Buffer так, что он
    содержит значения от 1 до size в возрастающем порядке
    без дубликатов. }

var
  i: integer;
begin                                           { процедура WriteIncreasing }
  for i := 1 to size do
    buffer[i] := i;
end;                                           { процедура WriteIncreasing }

procedure WriteDuplicateIncreasing(size: integer);
{ Pre: Параметр size содержит число элементов, которые следует
    поместить в выходной файл outfile.
  Post: Инициализирует глобальный массив Buffer так, что он
    содержит значения от 1 до size в возрастающем порядке
    с возможными дубликатами.
  Uses: Использует процедуру SelectSort. }

var
  i: integer;
begin                                           { процедура WriteDuplicateIncreasing }
  for i := 1 to size do
    Buffer[i] := RandomInt(1, size);
    SelectSort(size);
end;                                           { процедура WriteDuplicateIncreasing }

procedure WriteDecreasing(size: integer); { Записать в убывающем порядке }
{ Pre: Параметр size содержит число элементов, которые следует
    поместить в выходной файл outfile.
  Post: Инициализирует глобальный массив Buffer так, что он
    содержит значения от 1 до size в убывающем порядке
    без дубликатов. }

var
  i: integer;
  t: integer;
begin                                           { процедура WriteDecreasing }
  t := 1;
  for i := size downto 1 do
    begin
      buffer[i] := t;
      t := t + 1;
    end;
end;                                           { процедура WriteDecreasing }

procedure WriteDuplicateDecreasing(size: integer); { Записать в убывающем
    порядке с дубликатами }
{ Pre: Параметр size содержит число элементов, которые следует
    поместить в выходной файл outfile.
  Post: Инициализирует глобальный массив Buffer так, что он
    содержит значения от 1 до size в убывающем порядке
    с возможными дубликатами.
  Uses Использует процедуру WriteDuplicateIncreasing. }

```

```

var
i, j: integer;
begin
    WriteDuplicateIncreasing(size);
    i:=1;
    j := size;
    while (i < j) do
        begin
            Swap(i, j);
            i := i + 1;
            j := j - 1;
        end;
end;
    { процедура WriteDuplicateDecreasing }

procedure WriteRandom(size: integer);      { Записать случайные числа }
{ Pre: Параметр size содержит число элементов, которые следует
        поместить в выходной файл outfile.
  Post: Инициализирует глобальный массив Buffer так, что он
        содержит значения от 1 до size в случайном порядке
        без дубликатов.
  Uses: Использует процедуры Swap, RandomInt. }
var
i, j: integer;
begin
    Initialize(size);
    for j := 1 to (size) do
        Swap(j, RandomInt(j, size));
end;
    { процедура WriteRandom }

procedure WriteDuplicateRandom(size: integer);      { Записать случайные
                                                    числа с дубликатами }
{ Pre: Параметр size содержит число элементов, которые следует
        поместить в выходной файл outfile.
  Post: Инициализирует глобальный массив Buffer так, что он
        содержит значения от 1 до size в случайном порядке
        с возможными дубликатами.
  Uses: Использует процедуру RandomInt. }
var
i: integer;
begin
    { процедура WriteDuplicateRandom }
    for i := 1 to size do
        buffer[i] := RandomInt(1, size);
end;
    { процедура WriteDuplicateRandom }

procedure GetSpecs(var degree, maxmove: real);
    { Получить спецификации }
{ Pre: Предусловия отсутствуют.
  Post: Возвращает degree и maxmove с допустимыми значениями
        между 0 и 100. }
begin
    { процедура GetSpecs }
    writeln(' Каково максимальное расстояние от точки перемещения ');
    write(' элемента [0 до 100] 0 = перемещения нет , 100 = максимальное');
    write(' перемещение:');
    readln (maxmove);

```



```
writeln(' Каков процент упорядоченных элементов в группах ');
write(' [0 до 100] 0 = все в одной группе, 100 = все группы');
write(' по 1 элементу: ');
readln(degree);
end;                                     { процедура GetSpecs }
```

```
procedure WriteNearRandom(size: integer);
                                { Записать почти случайные числа }
{ Pre:   Параметр size содержит число элементов, которые следует
        поместить в выходной файл outfile.
  Post:   Инициализирует глобальный массив Buffer так, что он
        содержит значения от 1 до size в почти случайном порядке
        (в зависимости от спецификаций пользователя)
        без дубликатов.
  Uses:  Использует процедуры Random, RandomInt. }
var
  i: integer;
  degree,
  maxmove: real;
  absmove: integer;
  absmin, absmax: integer;
begin                                     { процедура WriteNearRandom }
  GetSpecs (degree, maxmove);
  absmove := round(maxmove/100 * size);
  WritelnIncreasing(size);
  for i := 1 to size do
    if ((Random * 100) < degree) then
      begin
        if ((i - absmove) <= 0) then
          absmin := 1
        else
          absmin := i - absmove;
        if ((i + absmove) > size) then
          absmax := size
        else
          absmax := i + absmove;
        Swap(i, RandomInt(absmin, absmax));
      end;
  end;                                     { процедура WriteNearRandom }
```

```
procedure WriteDuplicateNearRandom(size: integer);
                                { Записать почти случайные числа с дубликатами }
{ Pre:   Параметр size содержит число элементов, которые следует
        поместить в выходной файл outfile.
  Post:   Инициализирует глобальный массив Buffer так, что он
        содержит значения от 1 до size в почти случайном порядке
        (в зависимости от спецификаций пользователя) с возможными
        дубликатами.
  Uses:  Использует процедуры Random, RandomInt. }
var
  i: integer; degree,
  maxmove: real;
  absmove: integer;
  absmin, absmax: integer;
```

```

begin
  GetSpecs(degree, maxmove);
  absmove := round(maxmove/100 * size);
  WriteDuplicateIncreasing(size);
  for i := 1 to size do
    if ((Random * 100) < degree) then
      begin
        if ((i - absmove) <= 0) then
          absmin := 1
        else
          absmin := i - absmove;
        if ((i + absmove) > size) then
          absmax := size
        else
          absmax := i + absmove;
        Swap(i, Randomint(absmin, absmax));
      end;
    end;
  end;
  { процедура WriteDuplicateNearRandom }

```

# Приложение D

## Свойства языка Pascal

---

В этом приложении дается обзор некоторых средств языка Pascal, которые не всегда изучаются достаточно детально во вводных курсах программирования. К этим средствам относятся записи, процедурные параметры и упреждающие объявления, а также типы указателей и динамическое выделение памяти в приложении к связным спискам. Приложение завершается набором синтаксических диаграмм, таблиц и списков информации, который может помочь вам при написании Pascal-программ.

### D.1. Записи в языке Pascal

Многие реализации структур данных требуют, чтобы в структуру данных были включены несколько переменных различных типов. Например, простой непрерывный список составляется из *массива* элементов и *счетчика*, задающего число элементов. Для демонстрации логических связей между различными данными в языке Pascal существует категория типов, носящая название *записи*.

#### 1. Определение и примеры

Определение типа

```
record ... end
```

в языке Pascal устанавливает тип, состоящий из нескольких *полей* (называемых также *компонентами*), каждое из которых само по себе принадлежит к (произвольно заданному) типу. Например, *комплексное число* состоит из *действительной* части и *мнимой* части, каждая из которых представляет собой действительное число. Комплексное число может быть представлено в виде записи

```
type complex = record realpart, imagpart: real end.
```

Простой последовательный список можно объявить как

```
type list = record
    count: listcount;
    entry: array[position] of listentry
end;
```

#### 2. Обращение к записям

обращение  
к полям

К индивидуальным частям записи языка Pascal следует обращаться, сначала указав имя переменной, а затем точку (.), после которой указывает-

ся имя требуемой части, как оно объявлено в предложении определения типа записи. Так, если L1 и L2 есть списки, определенные выше, тогда их счетчики обозначаются L1.count и L2.count соответственно.

Конкретный k-й элемент списка L1 обозначается L1.entry[k].

### 3. Иерархические записи: абстракция данных

иерархические  
записи

Представим себе, что элементами списка являются комплексные числа. Мы можем определить такой список, указав тип listentry = complex. Заметьте, что мы имеем здесь пример записей (complex), содержащихся в качестве элементов массивов, которые являются полями другой записи (list). Такие записи называются *иерархическими*. Помещая записи внутрь записей, записи в массивы и массивы в записи, мы можем сконструировать сложные структуры данных, которые с максимальной точностью описывают взаимосвязи между данными, обрабатываемыми программой.

нисходящее  
проектирование  
для структур  
данных

В то же время, работая с записями, мы никогда не должны думать об их внутренней структуре. Наоборот, мы можем использовать нисходящее проектирование для структур данных точно так же, как и для алгоритмов. Когда мы обрабатываем большие внешние записи, мы не должны принимать в расчет детальную структуру каждого компонента внутри записи. Когда мы пишем алгоритмы манипулирования внутренними компонентами, мы можем рассматривать их только с учетом их собственной простой структуры и не думать о том, что позже эти компоненты будут включены в записи большего размера или в массивы. Используя записи таким образом, мы достигаем *сокрытия информации*, когда мы получаем возможность разрабатывать верхние уровни как алгоритмов, так и структур данных, не заботясь о деталях, которые будут определены позже на более низких уровнях детализации.

### 4. Предложение with

Для обращения к индивидуальным полям внутри иерархической записи мы вынуждены проходить весь путь от самого верхнего уровня до самого нижнего, используя точку (.) каждый раз, когда мы обращаемся к полю внутри записи, и квадратные скобки ([ ]) каждый раз, когда мы извлекаем элемент массива. Например, чтобы получить мнимую часть элемента k списка L1, мы должны написать L1.entry[k].imagpart.

Хотя, как видно из приведенного примера, такое обозначение вполне логично, его использование может оказаться крайне неудобным, если друг в друга вложены несколько записей и массивов. Язык Pascal предоставляет специальное предложение

**with recordvariable do ...**

обращение  
к полям

В блоке, контролируемым предложением **with**, специфицированные переменные-записи имеют особый статус, так что к их различным полям можно обращаться просто по их именам, без необходимости повторять каждый раз имя переменной-записи.

Продолжая наш пример со списками, состоящими из записей с комплексными числами, мы замечаем, что список L1 может быть обработан с помощью предложений такого рода:

```

with L1 do
  for k := 1 to count do
    with entry [k] do
      begin realpart := 1.0; imagpart := 0.0 end;

```

## 5. Вариантные записи

Приведенной выше информации о записях вполне достаточно для выполнения большинства приложений из этой книги, но в более сложных программах часто требуется использовать дополнительные свойства записей. Подробное описание этих свойств вместе с примерами можно найти в учебниках по программированию на языке Pascal.

При хранении в записи разнородной информации некоторые ее поля могут иногда оказаться неиспользуемыми. Для данных одного рода в записи может использоваться одно из полей, однако для данных другого рода будет использоваться другое поле. Если фигура представляет собой круг, нам будет нужно поле для *радиуса* круга. Если фигура — прямоугольник, нужны поля для его *высоты* и *ширины*, а также поле, где будет указано, является ли прямоугольник *квадратом*, или нет. Если фигура — треугольник, нужны поля для трех *сторон* и для сведений о том, является ли треугольник *равносторонним*, *равнобедренным* или *неравносторонним*. Для любой фигуры требуются поля для *площади* и *длины образующей*. Одним из способов определения типа записи для всех этих геометрических фигур будет выделение отдельного поля для каждого из требуемых атрибутов фигуры, но тогда, если конкретная фигура представляет собой прямоугольник, поля с радиусом и типом треугольника окажутся бессмысленными. Для круга или треугольника неопределенными будут другие поля.

Для преодоления этой трудности Pascal предоставляет **вариантные записи**, в которых некоторые поля определяются только если запись используется для хранения информации определенного вида. Вариантная запись содержит две части: **фиксированную** часть, в которой все поля остаются теми же самыми, независимо от вида сохраняемой в записи информации, и **вариантную** часть, в которой поля изменяются в зависимости от вида информации. Вид информации, помещенной в запись (и, соответственно, вариант использования записи) зависит от значения специального поля, называемого **полем тега**. Типом этого поля может быть любой перечислимый тип, а вариантная часть, предвояемая словом **case**, отдаленно схожа с предложением, в котором вместо селектора используется поле тега. Различные варианты выбираются константами порядкового типа, а поля в каждом варианте помещаются в круглые скобки ( ... ).

Все это станет более ясным, если мы вернемся к нашему примеру геометрической фигуры. Перечислимый тип, задающий вид информации в записи, определяется следующим образом:

```

type figuretype = (circle, rectangle, triangle);      { круг, прямоугольник,
                                                         треугольник }

```

пример

фиксированная и  
вариантная части

поле тега

Сама запись будет объявлена так:

геометрический  
пример

```
type figure = record
{ Это фиксированная часть записи }
    area,                                     { площадь }
    circumference: real;                     { длина образующей }
    case shape: figuretype of               { это поле тега }
        circle:                             { первый вариант (круг) }
            (radius: real);
        rectangle:                           { второй вариант (прямоугольник) }
            (height,                          { высота }
             width: real;                     { ширина }
            square: Boolean);                { квадрат? }
        triangle:                           { третий вариант (треугольник) }
            (side1,                          { сторона 1 }
             side2,                          { сторона 2 }
             side3: real;                    { сторона 3 }
            kind: (equilateral, isosceles, scalene)) { вид: равносторонний,
                                                    равнобедренный,
                                                    неравносторонний }
    end;                                     { конец объявления записи }
```

преимущества  
вариантных  
записей

Первым преимуществом вариантных записей является то, что они делают более наглядной логику программы, показывая в каждом случае, какая именно требуется информация. Второе преимущество заключается в экономии памяти при компиляции программы. Поскольку для конкретной записи используются поля только одного варианта, компилятор может выделять полям различных вариантов одну и ту же память, и в результате полный объем памяти, отводимый под запись, определяется суммарным размером полей наибольшего варианта. Эта ситуация проиллюстрирована на рис. D.1 применительно к примеру записей, описывающих геометрические фигуры.

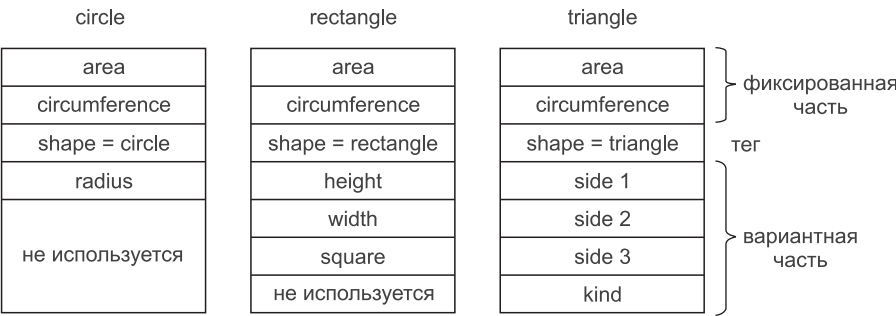


Рис. D.1. Память, выделяемая под варианты записи

правила и  
рекомендации  
для записей

Приведем несколько правил и рекомендаций относительно использования вариантных записей в программах на языке Pascal.

1. Фиксированная часть записи должна идти первой, затем размещается поле тега, а после него — вариантная часть (если она существует).
2. Запись может содержать лишь одну вариантную часть, хотя в качестве любых полей внутри записи могут выступать другие записи с собственными вариантными частями.

3. Все идентификаторы полей в записи должны различаться, даже если они относятся к различным вариантам.
4. Идентификатор поля тега может быть опущен, хотя это не рекомендуется, а *тип* тега должен указываться обязательно.
5. Варианты выбираются с помощью констант типа тега. Несколько констант, разделенных запятыми, могут задавать один и тот же вариант.
6. Список полей в варианте заключается в круглые скобки ( ... ). Список полей для варианта может быть пустым; такая ситуация обозначается пустыми скобками ( ).
7. Вариант, используемый в конкретной записи, определяется во время выполнения путем присвоения значений полю тега и полям соответствующего варианта. Вариант может быть изменен в любой момент изменением поля тега и вариантных полей. Если вариант изменяется, вариантные поля, принадлежащие предыдущему варианту, теряются.

## 6. Приведение типов

Тот факт, что различные варианты в записи занимают одно и то же место, позволяют нам при желании обмануть Pascal-компилятор и тем самым нарушить строгие правила языка Pascal относительно совместимости типов. Такие действия нельзя рекомендовать, поскольку, если только вы не понимаете в точности, как данные располагаются в памяти, они приведут к совершенно непредсказуемым результатам.

Сначала пример из естественного языка. Английское слово *car* (автомобиль) обозначает экипаж определенного рода, но на французском языке то же самое слово *car* обозначает *потому что*. *Gift* (подарок) по-английски порождает приятные чувства, но по-немецки слово *Gift* означает *яд*. В точности те же сочетания букв на различных языках имеют абсолютно разное значение.

Так же и в компьютерной памяти те же самые последовательности 0 и 1 представляют абсолютно разную информацию в зависимости от закрепленного за ними типа. Вот короткая программа, использующая вариантную запись, позволяющую использовать одну и ту же область памяти для хранения целых чисел, действительного числа или символов. Эта программа написана применительно к системе Turbo Pascal, где целое число обычно занимает два байта, действительное — шесть байт, а символы — по одному байту каждый.

```

program TypeCast (input, output);                                { Приведение типа }
{ Эта программа иллюстрирует приведение типов для целых чисел,
  действительных чисел и символов. Специфично для Turbo Pascal }
type threecases = (first, second, third);                        { используется только
                                                                    для установки трех случаев }

var convert: record case threecases of
    first: (i, j, k: integer);                                     { три поля по два байта каждое }
    second: (r: real);                                           { одно поле из шести байтов }
    third: (a, b, c, d, e, f: char)                               { шесть полей по одному байту
                                                                    каждое }

end;
```

```

begin                                     { главная программа TypeCast }
  with convert do begin
    write('Введите шесть символов: ');
    readln(a, b, c, d, e, f);
    writeln('Приведение к трем целым числам: ', i, ' и ', j, ' и ', k, '.');
    writeln('Приведение к одному действительному числу: ', r, '.');
  end
end.                                     { главная программа TypeCast }

```

приложение:  
запись значений  
указателей

Приведение типов имеет полезное применение в случае включения в программу временных фрагментов, служащих для отладки программы. Pascal в обычном применении не позволяет выводить на экран значения переменных-указателей, хотя такое действие может иногда помочь при проверке работы программы. Приводимая ниже функция приводит указатель к типу целого числа, которое может быть выведено на экран. Эта функция написана согласно спецификациям Turbo Pascal, где указатели занимают четыре байта, а тип `longint` определяет целое, занимающее также четыре байта.

```

function PtrToInt (p: pointer): longint;           { Указатель к целому }
{ Pre: Специфично для Turbo Pascal: параметр p является
        указателем; мы предполагаем, что указатели и длинные
        целые числа требуют одинаково по четыре байта.
  Post: Функция возвращает значение p, приведенное к типу
        длинного целого. }
var typecast: record case Boolean of
  false: (ptr: pointer);
  true:  (int: longint);
end;
begin                                             { функция PtrToInt }
  typecast.ptr := p;                             { сохраним указатель в записи }
  PtrToInt := typecast.int                       { извлечем из того же места длинное целое }
end;                                             { функция PtrToInt }

```

## Упражнения D.1

- E1. В чем главная причина использования записей, а не отдельных переменных?
- E2. Что такое иерархические записи, и как их надо обрабатывать?
- E3. Приведите два полезных свойства вариантных записей.
- E4. Определите типы записей (без вариантных полей) для каждого из следующих применений:
  - (a) Строка символов, состоящая из целочисленного счетчика вместе с массивом символов.
  - (b) Адрес, состоящий из улицы, города и государства или страны, причем каждый элемент адреса представляет собой строку (определенную в пункте (a)).
  - (c) Тип `personaldata`, состоящий из имени, являющегося строкой (определенной в пункте (a)), адреса (определенного в пункте (b)), пола индивидуума (мужчина или женщина), семейного положения (холостяк, женат, вдов или разведен) и числа иждивенцев от 0 до 20.



**E5.** Определите тип записи с вариантными полями для следующего упрощенного банковского вклада. Запись включает *имя*, являющееся строкой (как это определено в упражнении 4(a)), *адреса* (как определено в 4(b)), целочисленного *номера счета*, *даты открытия* счета (строка) и *остатка средств* на счете (действительное число). Имеются три вида вкладов: *накопительный*, *чековый* и *срочный*. Для накопительного вклада предусмотрен *депозитный список*, *расходный список* и *текущий остаток* (действительное число). Для чекового вклада предусматривается список *транзакций* (включая депозиты, кредиты, чеки и долги в виде положительных или отрицательных чисел) и *текущая плата за обслуживание* (действительное число). Все эти списки организуются как простые списки действительных чисел. Для срочного вклада предусмотрен *начальный вклад* (действительное число), *процентная ставка* (действительное число), *накопленная сумма процентов* (действительное число) и *срок вклада* (строка).

## D.2. Процедуры

### D.2.1. Процедуры в качестве параметров

Процедура или функция может быть передана другой процедуре или функции в качестве параметра, и затем вызвана, как и любая другая процедура или функция, из второй процедуры или функции. Цель этого действия такая же, как и цель передачи любого другого параметра: один и тот же формальный параметр может быть заменен любым из многих различных параметров. Пусть, например, *P* есть формальный процедурный параметр (т. е. параметр-процедура) для процедуры *A*. В этом случае внутри тела процедуры *A* могут быть вызовы процедуры *P*. Когда *A* используется в программе, формальный параметр *P* заменяется именем фактической процедуры. При использовании *A* в разных местах программы, параметр *P* может заменяться именами различных фактических процедур.

просмотр

В этой книге процедурные параметры используются главным образом при просмотре структур данных, т. е. при перемещении по структуре данных и выполнении при каждом обращении к структуре определенной работы. Какой бы ни была эта работа, мы используем для ее обозначения имя *Visit*. Фактически *Visit* является формальным процедурным параметром разрабатываемой нами процедуры *Traverse*. Сама по себе процедура *Visit* фактически *не существует*. Когда выполняется *Traverse*, этот формальный параметр заменяется именем некоторой требуемой процедуры. Одно приложение может вызывать процедуру вывода на экран элементов структуры данных; другое приложение может модифицировать данные; третье приложение может передавать данные в какую-либо другую процедуру. При активизации *Traverse* на месте *Visit* используются имена этих фактических процедур.

ограничения

Формальный процедурный параметр *P* сам может иметь параметры. Однако для них действует ограничение: фактическая процедура, которая замещает *P*, должна иметь в точности то же число параметров, как и *P*,

тех же типов и расположенных в том же порядке. (Поскольку эти параметры сами являются всего лишь формальными параметрами, их имена могут быть и другими.) Еще одно ограничение, накладываемое на фактический процедурный параметр заключается в том, что он должен представлять собой пользовательскую процедуру или функцию; он не может быть одной из встроженных в Pascal процедур или функций.

синтаксис,  
стандартный  
Pascal

В отношении синтаксиса между стандартным языком Pascal и Turbo Pascal имеются серьезные различия. В стандартном языке Pascal строка заголовка для объявления процедуры имеет форму

```
procedure A ( ...; procedure P( ...); ...);
```

Здесь первое и третье многоточие обозначают другие параметры процедуры A, а среднее многоточие стоит на месте параметров процедуры P. Все эти параметры могут быть любыми **var**-параметрами, параметрами-значениями или процедурными параметрами.

Поскольку процедурные параметры в стандартном языке Pascal требуют включения одного набора параметров (для P) внутрь другого набора параметров (для A), неудивительно, что программист иной раз может запутаться во всех этих параметрах.

синтаксис,  
Turbo Pascal

В языке Turbo Pascal используется совсем другой подход, сводящий эти трудности к минимуму. В секции **type** объявления типов среди прочих типов мы включаем типы **procedure** и **function**. Эти объявления принимают следующую форму:

```
type proctype = procedure(a: type1; var b, c:type2, ...);
```

или

```
type fnctype = function (x, y: type3; var z: type4, ...): type 5;
```

Параметры, указываемые с правой стороны, выглядят в точности так же, как и любой список формальных параметров процедуры или функции. Однако имена этих параметров (a, b, c, x, y, z) полностью игнорируются; они используются лишь как пустые обозначения, показывающие количество параметров каждого рода и их типы.

После того как процедурные или функциональные типы объявлены, процедурные параметры могут использоваться в точности так же, как и любые другие параметры. Объявление процедуры P становится таким:

```
procedure A ( ...; P: proctype; ...);
```

В этом объявлении процедурный параметр выглядит так же, как обычный параметр-значение.

far-объявления

Turbo Pascal предъявляет еще одно требование. Ради оптимизации процедуры Turbo Pascal могут иметь два варианта, называемые *дальним* (*far*) и *ближним* (*near*). Хотя любую процедуру можно сделать дальней, поместив в конце строки заголовка директиву **far**, проще отменить оптимизацию, заставив компилятор считать все процедуры и функции дальними. Эта операция выполняется помещением в начале программы, перед объявлениями каких-либо процедур и функций, директивы компилятора

```
{ $F + }
```

просмотр

Итак, чтобы применить все эти требования к нашему примеру просмотра структуры данных, мы должны, используя стандартный язык Pascal, включить в программу следующую строку, начинающую объявление процедуры просмотра:

```
procedure Traverse(var S: structure; procedure Visit(var x: entry));
```

Работая в системе Turbo Pascal, мы разделяем это объявление на два. Первое из них является объявлением типа:

```
type processentry = procedure(var x: entry);
```

Затем идет строка-заголовок для объявления процедуры просмотра:

```
procedure Traverse(var S: structure; Visit: processentry);
```

Эти два набора объявлений выполняют одно и то же, хотя, разумеется, они взаимно не совместимы.

## D.2.2. Упреждающие объявления

В рекурсивной программе процедура или функция обычно вызывает саму себя. Такой вызов носит название *прямой* (или *непосредственной*) рекурсии. Однако возможно, чтобы одна процедура или функция вызывала другую, которая, в свою очередь, может вызвать третью, и т. д. по цепочке, последняя процедура или функция которой вызывает первую, замыкая всю эту конструкцию в кольцо. Такой вызов называется *косвенной* рекурсией.

косвенная  
рекурсия

Косвенная рекурсия вызывает проблемы в языке Pascal, так как фундаментальное требование языка Pascal заключается в том, чтобы любые элементы объявлялись *перед* их использованием. Если у нас имеется длинная цепочка процедур, каждая из которых вызывает следующую, тогда, чтобы удовлетворить указанное фундаментальное требование, мы должны написать объявления всех процедур, входящих в цепочку, в *обратном* порядке, чтобы каждая из них объявлялась перед своим использованием. Если, однако, цепочка замыкается в кольцо, в котором последний член вызывает первый, становится невозможным объявить каждую процедуру перед ее использованием.

*Упреждающие объявления* предоставляют решение этой проблемы. Мы разрываем цепочку, удаляя одну из ее процедур или функций. Пусть она носит имя P. Мы затем помещаем строку-заголовок для P в более раннее место программы, перед объявлением тех процедур или функций, которые непосредственно используют P. В это более раннее место мы помещаем только строку-заголовок, в которой указывается имя P и ее параметры. Вместо полного объявления P мы помещаем только одно ключевое слово forward. Такое сокращенное объявление выглядит следующим образом:

```
procedure P(все параметры процедуры P); forward;
```

Далее идут полные объявления всех функций и процедур, непосредственно использующих P. Позже в программе, после того, как были объявлены все процедуры, используемые P, мы помещаем полное объявление

модули  
Turbo Pascal

для Р. Это объявление отличается от обычного объявления процедуры или функции только в том отношении, что параметры для Р не требуют повторения, поскольку они уже были указаны ранее. Некоторые компиляторы (включая Turbo Pascal) допускают, хотя и не требуют повторения списка параметров; другие (включая стандартный Pascal) запрещают повторение списка параметров. В последнем случае полезно повторить список параметров, поместив его в знаки комментария, поскольку между формальным объявлением параметров для Р и второй частью ее объявления может располагаться значительный фрагмент программы.

В модулях Turbo Pascal используется другой подход. Упреждающие объявления и не требуются, и вообще запрещены. Вместо них строка-заголовков для процедуры помещается в интерфейсную секцию **interface** модуля. В этом случае, когда тело процедуры встречается в реализационной секции **implementation**, имена и параметры всех используемых ею процедур и функций уже известны. В результате объявления процедур и функций в секции **implementation** могут появляться почти в любом порядке без всяких проблем, при условии, что они были указаны в секции **interface**.

Modula-2, более современный язык, построенный на базе языка Pascal, использует тот же подход, что и модули Turbo Pascal, тем самым устраняя возможную путаницу, присущую упреждающим объявлениям.

## D.3. Указатели и связанные списки

### D.3.1. Введение и обзор

#### 1. Проблема переполнения

Если мы реализуем структуру данных в непрерывной памяти, тогда все данные, входящие в структуру, размещаются в массивах, которые должны быть объявлены с указанием их размеров еще при написании программы, и, следовательно, размеры этих массивов не могут изменяться при выполнении программы. Составляя программу, мы должны определить максимальный объем памяти, который может потребоваться для наших массивов, и указать этот объем в объявлениях. Если мы затем выполняем прогон программы с небольшим набором входных данных, значительная часть зарезервированного объема не будет использоваться. Если же мы выполняем прогон программы с большим набором, мы можем превысить зарезервированный объем памяти, в результате чего возникнет ситуация переполнения, даже если память самого компьютера используется далеко не полностью, просто потому, что установленные заранее границы массива оказались недостаточными.

Даже если мы, имея в виду эту проблему, объявляем наши массивы достаточно большими и используем всю память компьютера, мы все еще можем столкнуться с переполнением, если один из массивов окажется заполненным до конца, в то время как в других остается еще много неиспользуемого места. Поскольку при различных прогонах одной и той же программы могут расти или сокращаться то одни, то другие структуры

неравномерное  
использование  
памяти

данных, сказать заранее, в каких структурах произойдет переполнение, невозможно.

Мы, однако, можем предложить способ хранения структур в памяти без использования массивов, который позволяет обойти эти трудности.

## 2. Указатели

Суть идеи заключается в использовании указателей. *Указатель*, также называемый *связью* или *ссылкой*, определяется как переменная, которая хранит адрес некоторой другой переменной, обычно записи, содержащей нужные нам данные. Если для обращения ко всем нашим записям мы используем указатели, нам не надо думать о том, где хранятся сами записи, поскольку при использовании указателя местонахождение требуемой записи определяет сама компьютерная система.

## 3. Соглашения относительно диаграмм

На рис. D.2 показаны указатели на несколько записей. Обычно указатели обозначаются в виде стрелок, а записи — в виде прямоугольных рамок. В рисунках этой книги переменные, содержащие указатели, обычно изображаются в виде рамок, кружков или кубиков серого цвета. Таким образом, на диаграмме *r* представляет собой указатель на запись "Lynn", а *v* — указатель на запись "Jack". Как можно заметить, использование указателей характеризуется гибкостью: два указателя могут относиться к одной и той же записи, как, например, *t* и *u* на рис. D.2, но указатель может и ни на что не указывать. Эту последнюю ситуацию мы обозначаем на диаграмме в виде символа *электрического заземления*, как это сделано для указателя *s*. При использовании указателей следует соблюдать осторожность, чтобы при перемещении указателей не потерять какую-либо запись. Так, на рисунке D.2 запись "Dave" потеряна, так как ни один указатель на нее не указывает, и, следовательно, нет никакого способа к ней обратиться.

указатели,  
ни на что  
не указывающие

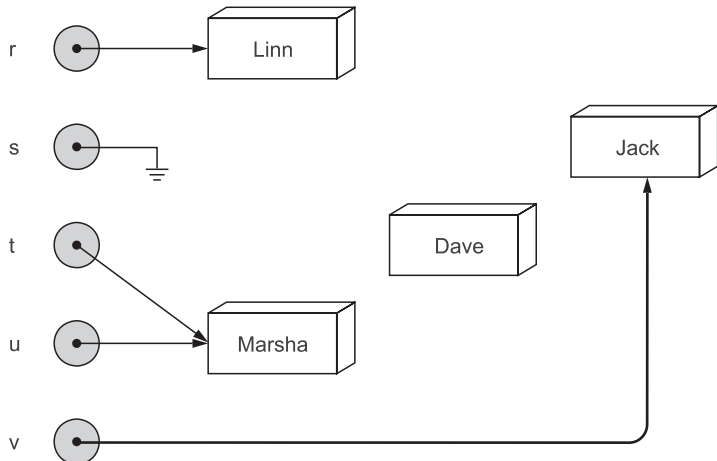


Рис. D.2. Указатели на записи

#### 4. Связные списки

связные списки

Идея *связного списка* заключается в том, что в каждую запись в списке помещается указатель на следующую запись в том же списке. Эта идея проиллюстрирована на рис. D.3.

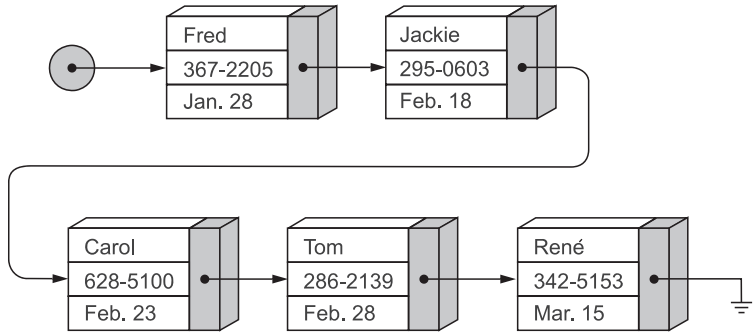


Рис. D.3. Связный список

аналогии

Как видно из этого рисунка, принцип построения связного списка прост. Он использует ту же идею, что и в детской игре поиска спрятанного сокровища, где каждая найденная подсказка указывает, где найти следующую. Или можно представить себе ситуацию с приятелями, передающими друг другу популярную кассету. Она находится у Фреда, и он обещал дать ее Джеки. Кэрол просит ее на время у Джеки и передает ее Тому и т. д. Связный список можно уподобить последовательности команд, каждая из которых выдается только после завершения выполнения предыдущей. В этом случае на число выполняемых действий не накладывается никаких ограничений, поскольку каждое действие может содержать следующую команду, и заранее нельзя сказать, сколько всего будет команд. С другой стороны, реализация списка, описанная в разделе 2.2.1, аналогична перечню команд, написанному на листке бумаги. В этом случае все команды можно увидеть заранее, и при этом на листке бумаги может уместиться лишь ограниченное число команд.

Приобретя некоторую практику, вы увидите, что со связными списками работать так же легко, как и со списками, реализованными в виде массивов. Однако эти методы различаются существенно, и для их освоения требуются некоторые программистские усилия. Перед тем, как мы обратимся к этой работе, рассмотрим еще несколько общих соображений.

#### 5. Непрерывные и связанные списки

определения

Слово *непрерывный* означает *находящийся в контакте, примыкающий*. Элементы в массиве непрерывны, и поэтому мы говорим о списке, хранящемся в массиве, как о *непрерывном списке*. Таким образом, термины *связный список* и *непрерывный список* однозначно указывают на их организацию, а более общий термин *список* включает в себя обе разновидности. Те же соглашения применимы к стекам, очередям и другим структурам данных.

## 6. Указатели для непрерывных списков

Указатель представляет собой переменную, в которой хранится адрес некоего другого элемента, и для непрерывных списков мы фактически все время неформальным образом использовали указатели. Переменная *top* (вершина) является указателем на элемент, находящийся на вершине стека, а переменные *front* (голова) и *rear* (хвост) определяют местоположение головного и хвостового элементов очереди. Для того, однако, чтобы избежать возможной путаницы между связными и непрерывными списками, мы, как правило, будем использовать слово *указатель* применительно к связным спискам, а для определения положения ячейки в массиве будем пользоваться словом *индекс*.

## 7. Динамическое выделение памяти

многозадачность  
и разделение  
времени

Указатели, помимо предотвращения проблем переполнения, связанных с выходом за пределы границ массивов, создают еще определенные преимущества при работе в многозадачных средах с разделением времени. Если мы используем массивы и резервируем заранее максимальный объем памяти, который может потребоваться для нашей задачи, тогда эта память выделяется именно этой задаче, и будет недоступна для других задач<sup>1</sup>. Если возникает необходимость выгрузить нашу задачу из памяти на диск, этот процесс может потребовать значительного времени на копирование на диск или с диска фактически неиспользуемой памяти. Однако, вместо того, чтобы использовать для хранения всех наших данных массивы, мы можем начать с очень маленькой по размеру программы, в которой зарезервировано место только для программных команд и простых переменных, а при необходимости сохранения новых данных мы будем обращаться к системе с запросом на выделение требуемой памяти. Когда какой-то элемент нам больше не нужен, мы можем вернуть занимаемую им память системе, которая использует ее для других задач. При такой организации программа может начать свое выполнение, имея совсем небольшой размер, и увеличивать его лишь при необходимости размещения новых элементов. Небольшой начальный размер программы повышает эффективность ее выполнения, и, в то же время, программа по мере необходимости может увеличивать свои размеры до пределов, диктуемых компьютерной системой.

Даже если в системе в каждый момент выполняется лишь одна задача, динамическое управление памятью может оказаться полезным. По ходу выполнения некоторого фрагмента программы ей может потребоваться значительный объем памяти для хранения определенных данных, но этот объем памяти может быть освобожден и затем снова выделен для других целей, возможно, для хранения данных другого типа и назначения.

<sup>1</sup>

В современных многозадачных средах, например, в среде операционной системы Windows, эта проблема не возникает, так как каждой задаче выделяется свое виртуальное адресное пространство объемом, например, до 2 Гбайт, которое каждая задача может свободно использовать, и увеличение размера одной задачи никак не влияет на возможности использования памяти другими задачами. Поэтому число задач, выполняемых в системе, никак не связано с объемами памяти, и освобождение памяти (виртуальной) некоторой задачей (см. ниже по тексту) не увеличивает возможности других задач, хотя, разумеется, уменьшение размеров программы повышает эффективность ее выполнения и сокращает время, требуемое для ее копирования на диск и с диска в процессе переключения задач. — Прим. перев.



## D.3.2. Указатели и динамическая память в языке Pascal

Большинство современных языков программирования, включая Pascal, предоставляют мощные средства для обработки указателей, а также стандартные процедуры для выделения и освобождения дополнительной памяти в процессе выполнения программы.

### 1. Статические и динамические переменные

статические  
переменные

динамические  
переменные

Переменные, используемые при выполнении Pascal-программ, могут быть двух типов. *Статическими переменными* называются переменные, которые объявляются и именуется при написании программы. Память, выделяемая под эти переменные, как правило, существует все время, пока выполняется программа, в которой они объявлены. *Динамические переменные* создаются (и, возможно, уничтожаются) в процессе выполнения программы. Поскольку динамические переменные не существуют во время компиляции программы, но только когда она выполняется, им нельзя назначить имена при написании программы.

Единственным способом обращения к динамическим переменным является использование указателей. Однако после своего создания динамическая переменная должна содержать данные и иметь тип, как и любая другая переменная. Поэтому мы можем говорить о создании новой динамической переменной типа  $x$  и установки указателя на эту переменную, или о перемещении указателя с одной динамической переменной типа  $x$  на другую, или о возвращении динамической переменной типа  $x$  в системе.

Статические переменные, с другой стороны, не могут создаваться или уничтожаться в процессе выполнения программы, в которой они объявлены, а переменные-указатели не могут использоваться со статическими переменными. Обращение к статическим переменным может осуществляться только по их именам — как мы, собственно говоря, всегда и делали — и если нам требуется указать на позицию внутри массива, мы, как и раньше, используем индексные переменные того же типа, что и индексы массива.

### 2. Обозначения языка Pascal

привязки  
указательного  
типа

В языке Pascal для обозначения указателей используются знаки стрелка вверх  $\uparrow$ , каре  $\wedge$  или диакритический знак (циркумфлекс)  $\hat{\cdot}$ . (Эти три символа представляют собой разные варианты изображения на экране одного и того же компьютерного символа.) Если эти символы недоступны, используется знак  $@$ . Если  $node$  обозначает тип интересующего нас элемента, тогда мы объявляем тип указателя, *привязанного* к типу  $node$ , следующим образом:

**type** pointer =  $\uparrow node$

Тип  $node$ , на который ссылается указатель, может быть каким угодно, но в большинстве приложений это тип записи. Заметьте, что слово  $pointer$  (указатель) не является зарезервированным словом в языке Pascal; мы



объявили его аналогично любому другому идентификатору, используемому для обозначения вновь определенного типа. Слова `link` (связь) и `reference` (ссылка) также часто используются для обозначения указательных типов. Как и в случае любого другого типа мы теперь можем объявлять переменные, которые будут иметь тип `pointer`, и эти переменные будут указывать на динамические переменные типа `node`.

Если используются более одного типа указателей на динамические переменные, эти типы можно назвать так, чтобы они отражали существо динамических переменных, например в таких объявлениях:

```
type pointitem = ↑item
type pointnode = ↑node
```

Здесь указатель на динамическую переменную типа `item` объявлен как переменная типа `pointitem`, а указатель на динамическую переменную типа `node` объявлен как переменная типа `pointnode`.

### 3. Привязка типов

Язык Pascal накладывает строгие правила на использование указателей. Каждый указатель *привязан* к типу переменной, на которую он указывает, и тот же самый указатель никогда не может использоваться для указания (в различные моменты времени) на переменные различных типов. Переменные двух различных указательных типов не могут смешиваться друг с другом; Pascal допускает операции взаимного присваивания двух переменных-указателей одного типа, но запрещает выполнение таких операций над переменными-указателями различных типов. Если у нас имеются объявления

```
var a, b: pointitem;
    x, y: pointnode;
```

тогда операции присваивания `x := y` и `a := b` вполне законны, но операция `x := a` недопустима.

### 4. Создание и уничтожение динамических переменных

Создание и уничтожение динамических переменных в языке Pascal выполняется с помощью стандартных процедур этого языка. Если переменная `p` была объявлена как указатель на тип `node`, тогда процедура

```
new (p)
```

создаст новую динамическую переменную типа `node` и присвоит ее адрес указателю `p`. Процедура

```
dispose (p)
```

вернет память, занятую переменной типа `node`, системе. (*Предупреждение:* Некоторые системы программирования на языке Pascal утрачивают эту память и больше ее уже не используют.) После вызова процедуры `dispose(p)`, переменная-указатель `p` не определена, и, следовательно, не может использоваться до тех пор, пока ей не будет присвоено новое значение. Эти действия показаны на рис. D.4.

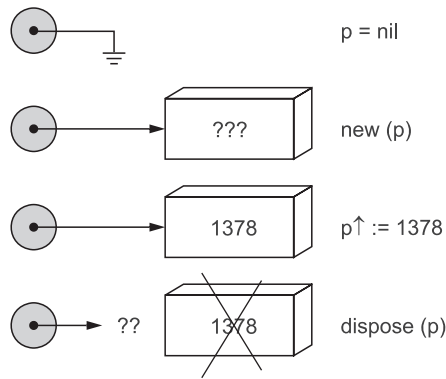


Рис. D.4. Создание и уничтожение динамических переменных

## 5. Указатели nil

Иногда переменная-указатель  $p$  оказывается не привязанной к конкретной динамической переменной. Эта ситуация устанавливается присваиванием

$$p := \text{nil}$$

и может быть проанализирована предложением вроде

$$\text{if } p \neq \text{nil} \text{ then } \dots$$

На диаграммах мы используем для обозначения **nil**-указателей символ электрического заземления



Слово **nil** является в языке Pascal зарезервированным словом, не идентификатором, значение которого можно при желании изменить. Оно служит для переменных-указателей предопределенной константой и имеет родовой характер в том отношении, что может быть присвоено переменной любого указательного типа.

Обратите внимание на разницу между переменной-указателем с неопределенным значением и переменной-указателем со значением **nil**. Операция присваивания  $p = \text{nil}$  означает, что указатель  $p$  в настоящий момент не указывает ни на какую динамическую переменную. Если же значение  $p$  не определено, тогда  $p$  может указывать на любое случайное место в памяти. Как и в случае переменных, когда программа начинает выполнение, значения переменных-указателей не определены. Перед тем, как переменную  $p$  можно будет использовать, требуется либо вызвать  $\text{new}(p)$ , либо выполнить присваивание, например,  $p := q$  или  $p := \text{nil}$ . После вызова  $\text{dispose}(p)$  значение  $p$  становится неопределенным, поэтому разумно сразу же выполнить предложение  $p := \text{nil}$ , чтобы обезопасить себя от использования переменной  $p$  с неопределенным значением.

## 6. Идем по стрелке

Стрелка вверх ( $\uparrow$ ), обозначающая *указатель*, может использоваться не только в объявлениях, но также и в активной части Pascal-программы. В активной части стрелка указывается не слева от типа, а справа от переменной-указателя. Так,  $p\uparrow$  обозначает переменную, на которую указывает  $p$ . Поначалу эти обозначения могут показаться путанными, но их логика становится понятной, если вы вспомните, что  $\uparrow$  означает *указывает*. Так, объявление

$p: \uparrow \text{item}$

снятие ссылки  
с указателя

читается « $p$  указывает на  $\text{item}$ », а  $p\uparrow$  читается «то, на что указывает  $p$ ». Как уже отмечалось, в этом же контексте часто используются слова *связь* и *ссылка*. Действие по получению  $p\uparrow$  иногда называют «снятием ссылки с указателя  $p$ ».

файловое окно

Обратите внимание на сходство этих обозначений с обозначениями языка Pascal для *файлового окна*. Если  $F$  есть файл, тогда  $F\uparrow$  обозначает один элемент этого файла, в настоящий момент доступный программе. Другими словами,  $F\uparrow$  обозначает позицию в файле, на которую указывает указатель позиции в файле, точно так же, как  $p\uparrow$  есть позиция в памяти, на которую в настоящий момент указывает указатель  $p$ .

## 7. Ограничения, накладываемые на переменные-указатели

операции  
с указателями

Единственным назначением переменных типа  $\uparrow \text{item}$  является хранение адресов переменных типа  $\text{item}$ . Переменные-указатели могут участвовать в предложениях присваивания, проверки на равенство, а также могут (в качестве параметров) присутствовать в вызовах подпрограмм, но нигде больше. Программисту не разрешается выполнять над указателями арифметические операции, так как они являются не числами с внутренним значением, а адресами ячеек памяти. Также не допускается чтение указателей и запись в них значений, поскольку они являются адресами, назначаемыми в процессе выполнения программы, поскольку их значения могут отличаться от одного прогона программы к следующему, и поскольку их значения (как адреса компьютерной памяти) определяются деталями реализации, которыми программист не должен заниматься непосредственно. (Некоторые Pascal-системы позволяют выводить значения переменных-указателей в отладочных целях с тем, чтобы программист мог проконтролировать проверку на равенство и правильность выполнения операций присваивания для указателей.)

Заметьте, что эти ограничения в использовании указателей не относятся к динамическим переменным, на которые указывают указатели. Если  $p$  является указателем, тогда  $p\uparrow$  обычно не является указателем (хотя иметь указатели, указывающие на указатели, вполне допустимо), но переменной некоторого другого типа  $\text{node}$ , и, следовательно,  $p\uparrow$  может использоваться любым способом, допустимым для типа  $\text{node}$ .

операции  
присваивания

В отношении предложений присваивания важно понимать разницу между двумя вполне допустимыми (при условии, что  $p$  и  $q$  привязаны к одному и тому же типу) выражениями,  $p := q$  и  $p\uparrow := q\uparrow$ , которые приводят к совершенно различным последствиям. После выполнения первого

предложения указатель  $p$  будет указывать на тот же объект, на который указывает указатель  $q$ , причем это действие не изменяет ни этот объект, ни тот, который ранее был  $p\uparrow$ . Этот последний объект будет потерян, если только нет какого-то другого указателя, который все еще указывает на этот объект. Второе же предложение,  $p\uparrow := q\uparrow$ , копирует значение объекта  $q\uparrow$  в объект  $p\uparrow$ , так что теперь мы имеем два объекта с одним и тем же значением, причем  $p$  и  $q$  указывают на две различные копии этого объекта. Наконец, предложения  $p := q\uparrow$  и  $p\uparrow := q$  содержат переменные разных типов и являются недопустимыми (за исключением тех редких случаев, когда и  $p$ , и  $q$  указывают на указатели их собственного типа!). Рис. D.5 иллюстрирует операции присваивания указателей.

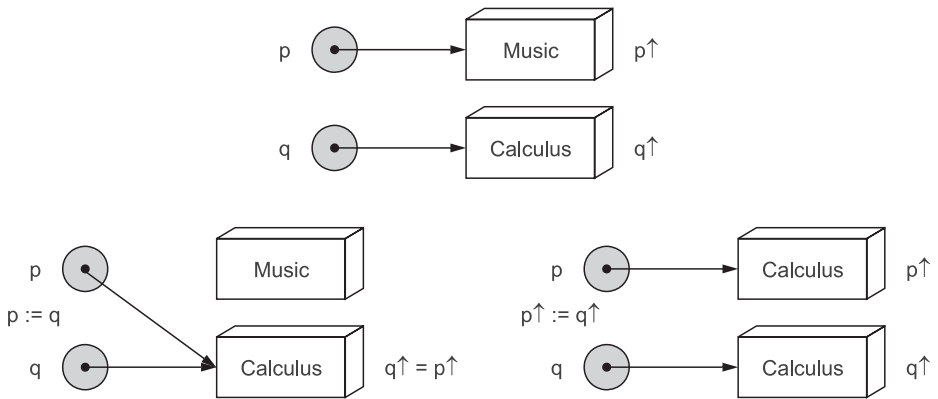


Рис. D.5. Присваивание значений переменным-указателям

### D.3.3. Основы связанных списков

Познакомившись с указателями и указательными типами, мы можем приступить к рассмотрению вопросов реализации связанных списков на языке Pascal. Начнем с объявлений, требуемых для организации связанного списка.

#### 1. Узлы и объявления типов

Вспомним (см. рис. D.3), что связанный список состоит из записей, каждая из которых содержит как информацию, сохраняемую в качестве элемента списка, так и указатель, позволяющий найти в списке следующую запись. Мы будем называть эти записи, составляющие связанный список, **узлами** списка, а указатели часто называются **связями**. Поскольку связь в каждом узле позволяет найти следующий узел (next node) списка, мы для обозначения связи будем использовать переменную *nextnode*. Запись этих определений на языке Pascal выглядит следующим образом:

```

type
  listpointer = ↑listnode;
  listnode = record
    entry:    listentry;    { информационные поля, помещаемые в список }
    nextnode: listpointer;  { связь со следующим узлом в списке }
  end;
```

использование  
перед  
объявлением

Обратите внимание на то, что этому объявлению присуща проблема «порочного круга». Поле `nextnode` типа `listpointer` является частью записи типа `node`, и Pascal требует, чтобы тип `listpointer` был объявлен перед `listnode`, как мы и сделали. С другой стороны, тип `listpointer` объявлен как `↑listnode` и, следовательно, использует тип `listnode` внутри своего объявления. Отсюда следует, что тип `listnode` должен быть объявлен перед `listpointer`. Для устранения этой проблемы «порочного круга» Pascal для указательных типов (и только для указательных типов) ослабляет фундаментальное правило, гласящее, что каждый идентификатор должен быть объявлен перед его использованием. В результате конструкция

`↑ sometype { некоторый тип }`

вполне допустима в объявлении типа в любой точке программы, даже если тип `sometype` не был еще объявлен. (Однако тип `sometype` должен быть объявлен где-то в пределах программы, иначе компилятор сообщит об ошибке.)

объем памяти  
для указателей:  
слово

Причина, по которой Pascal может ослабить это правило и, тем не менее, выполнять эффективную компиляцию, заключается в том, что все указатели занимают один и тот же объем памяти, часто тот же самый объем, что и целочисленные переменные (этот объем часто называют *словом*), независимо от того, к каким переменным они относятся. Поэтому, встретившись с объявлением указательного типа, компилятор может выделить правильный объем памяти и отложить задачу проверки соответствия всех объявлений и использования переменных правилам языка.

## 2. Начало списка

В нашем связном списке мы используем поле указателя `nextnode` для перемещения от любого узла связного списка к его следующему узлу и, таким образом, начав просмотр, мы можем пройти по всему списку. Однако остается маленькая проблема, с которой мы никогда не встречаемся при использовании непрерывных списков или других статических переменных и массивов: как нам найти начало списка?

Возможно, нам поможет аналогия с чтением журнальной статьи. Если мы находимся в середине статьи, то часто внизу страницы мы можем увидеть указание «Продолжение на странице ...». Следуя этим указаниям, мы можем продолжить чтение, пока не дойдем до конца статьи. Но как нам найти ее начало? Для этого мы просматриваем содержание, которое обычно располагается в фиксированном месте, где-то в начале журнала.

головной элемент  
списка

При работе со связными списками мы также для того, чтобы найти начало списка, обращаемся к некоторой *фиксированной* ячейке; другими словами, чтобы получить первый узел списка, мы используем статическую переменную. **Головным указателем** (head) списка является переменная-указатель, в которой хранится адрес начала списка. Обычно эта переменная является статической, и, используя ее значение, мы получаем первый (динамический) узел списка. Головной указатель иногда называется также *базой* или *якорем* списка. Эти термины хорошо описывают существо переменной, которая привязана к началу списка, но ввиду их

малой распространенности мы будем использовать более распространенный термин *головной указатель* или просто *голова списка*. Тот же указатель часто называют **заголовком** (header) списка.

Короткий связный список с головным указателем показан на рис. D.6, а. На этом рисунке также показано разделение каждого узла списка на содержательную часть с элементом entry (типа listentry) и связывающую часть link, указывающую на следующий узел. Переменная связи, составляющая содержимое части link, в дальнейших программных фрагментах будет называться *nextnode* (и принадлежать типу listpointer). На рис. D.6, так же, как и на других рисунках этой книги, связи и другие указательные переменные выделены серым цветом.

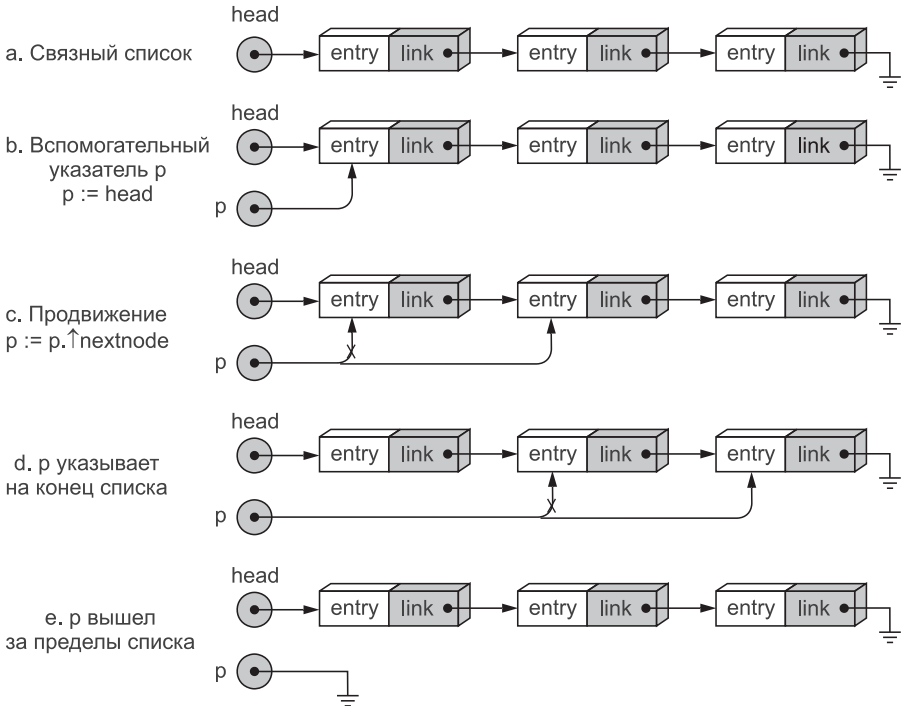


Рис. D.6. Указатели и связный список

инициализация

При запуске программы мы инициализируем список так, что он оказывается пустым; при наличии головного указателя это сделать очень просто. Головной указатель является статической переменной, существующей с самого начала программы. Другими словами, переменная *head* объявляется и именуется как любая другая статическая переменная. Чтобы установить для нее значение, соответствующее пустому списку, достаточно выполнить предложение присваивания:

`head := nil;`

### 3. Конец списка

Найти конец списка проще, чем начало. Поскольку каждый узел содержит указатель на следующий, полю связи последнего узла указывать не на что, и мы придаем ему специальное значение **nil**. Такая ситуация также отображена на рис. D.6. Таким образом, если у используемого нами узла связь со следующим узлом имеет значение **nil**, этот узел является последним узлом списка.

### 4. Продвижение по списку

В большинстве случаев, работая со списком, нам требуется обработать не только первый элемент списка, на который указывает **head**, но и другие его элементы. Для продвижения по списку нам понадобятся дополнительные переменные-указатели. На рис. D.6 показана одна из таких переменных, названная **p** (от **pointer**, указатель). Эта статическая переменная, и, следовательно, у нее есть имя, и ее следует объявить так же, как и другие статические переменные.

Чтобы установить **p** на начало списка, мы используем предложение присваивания  $p := \text{head}$ . В этом предложении копирование значения головного указателя **head** в переменную **p** приводит к тому, что **p** будет указывать на тот же узел, на который указывает **head**, т. е. на первый узел списка.

Далее нам нужно продвинуться по списку на одну позицию, как это показано на рис. D.6, с. Если **p** указывает на некоторый узел списка, этот узел обозначается  $p \uparrow$ , и поле **nextnode** узла  $p \uparrow$  указывает на следующий узел списка. Отсюда следует, что именно это значение нам нужно присвоить переменной **p**, чтобы переместиться по списку на одну позицию. Другими словами,

*Присваивание  $p := p \uparrow.\text{nextnode}$  продвигает переменную-указатель **p** на одну позицию по связному списку.*

Это одна из наиболее фундаментальных операций, используемых при обработке связанных списков.

Обратите внимание на аналогию между присвоением значения указателю  $p := p \uparrow.\text{nextnode}$  и присвоением значения индексу  $i := i + 1$ , которое продвигает индекс **i** на одну позицию в массиве или непрерывном списке.

Повторяя присваивания  $p := p \uparrow.\text{nextnode}$ , мы можем продвигать **p** по списку на одну позицию за раз, пока **p** не достигнет последнего узла, как это показано на рис. D.6, d. Если мы в такой ситуации продвинем **p** еще один раз, этот указатель выйдет за пределы списка. Если это происходит, **p** присваивается значение поля **nextnode** последнего узла, а это значение равно **nil**. Таким образом, ситуация, когда **p** выходит за пределы списка, может быть распознана по состоянию  $p = \text{nil}$ , как это показано на рис. D.6, e.

Работая со связными списками, мы должны постоянно выполнять проверку вроде "if  $p <> \text{nil}$  then ..." или "while  $p <> \text{nil}$  do ...", чтобы удостовериться, что мы обращаемся к существующему в списке узлу. На-

начало списка

продвижение  
на одну позицию

выход за пределы  
списка

пример, цикл просмотра списка с заходом в каждый узел часто принимает такой вид:

```
p := head;           { начнем с того, что p указывает на голову списка }
while p <> nil do      { будем продолжать, пока не дойдем до конца }
begin
  { Предложения в этом блоке обрабатывают узел p↑ }
  p := p↑.nextnode    { переместим p на одну позицию по списку }
end;
```

обратное  
перемещение

Наконец следует заметить, что хотя по связанному списку легко перемещаться в прямом направлении, продвижение назад попросту невозможно, поскольку связи в списке допускают переходы только в одном направлении. Единственным очевидным способом переместиться на один узел назад является возврат к головному элементу и продвижение по списку в прямом направлении до требуемого элемента.

Отсюда следует, что мы, привязав головной указатель к началу списка, никогда не должны изменять его значение. Мы, разумеется, можем выполнить операцию присваивания `head := head↑.nextnode`, но после нее мы невосстановимо потеряем первый узел списка (если, конечно, у нас нет какой-нибудь второй переменной, указывающей на этот узел). Несмотря на то, что все данные первого узла останутся в неприкосновенности, и пространство памяти, выделенное под этот узел, не сможет быть использовано каким-либо иным образом, у нас не будет никакой возможности найти эти данные и получить к ним доступ, или даже уничтожить этот узел.

### D.3.4. Связная реализация простых списков

Имея в виду все рассмотренные средства, давайте разработаем связанную реализацию простых списков в соответствии со спецификацией, данной в разделе 2.2.1. Вспомним, что для простого списка специфицированы операции *create* (создать), *empty* (пуст), *full* (полон), *size* (размер), *add* (добавить в конец) и *traverse* (просмотр). Большую часть этих операций нетрудно запрограммировать с помощью рассмотренных выше приемов; часть этой работы мы здесь выполним, часть оставим для упражнений. Две операции, *size* и *add*, можно упростить, если мы выполним тщательное объявление типа для списка.

#### 1. Объявления типов

счетчик

Одним из способов определения размера связанного списка является продвижение по нему, узел за узлом, с одновременным инкрементом счетчика узлов. Однако значительно проще зарезервировать одно дополнительное поле в объявлении записи списка, которое будет служить счетчиком элементов списка. Изменение длины списка может быть следствием всего двух операций — добавления нового элемента и очистки списка, поэтому обновление счетчика будет выполняться только в этих двух операциях, и определение размера списка становится тривиальной задачей.

Добавление в конец списка нового элемента, если у нас есть только указатель на головной элемент списка, так же требует каждый раз про-





### 3. Добавление нового узла

Теперь, когда мы поместили добавляемый элемент в новый узел, мы должны добавить этот новый узел к концу списка.

Здесь нам придется рассмотреть два случая. При наличии любого связанного списка (с головным указателем) включение узла в первую позицию списка должно всегда рассматриваться отдельно от включения в любую более позднюю позицию, поскольку включение в первую позицию требует изменения переменной *head*, в то время как включение в другие позиции переменную *head* не изменяют. Для наших простых связанных списков мы также поддерживаем поле хвостового указателя *tail*, наличие которого упрощает включение в конец списка. Если список перед этим был пуст, тогда его головным узлом становится новый узел. Если список не был пуст, тогда старый хвостовой элемент списка будет указывать на новый узел. В любом случае хвостовым узлом списка становится новый узел. В результате мы имеем:

```
procedure AddNode (newnode: listpointer; var L: list);      { Добавить узел }
{ Pre:   Список L уже создан, и параметр newnode указывает на узел,
        которого еще нет в списке L.
  Post:  Новый узел newnode↑ добавлен в простой список L. }
begin                                                    { процедура AddNode }
  if newnode = nil then
    Error('Попытка присоединить к списку несуществующий узел.')
  else begin
    L.count := L.count + 1;
    newnode.nextnode := nil;  { новый узел будет хвостовым элементом,
                              { поэтому за ним ничего не следует }
    if L.head = nil then      { случай предварительно пустого списка }
      L.head := newnode { новый узел становится головным элементом }
    else                     { случай предварительно непустого списка }
      L.tail↑.nextnode := newnode; { старый хвост связывается
                                   { с новым узлом }
    L.tail := newnode        { новый узел становится
                              { хвостовым элементом списка }

  end
end;                                                    { процедура AddNode }
```

Процедура *AddNode* не перечислена в спецификациях для простого списка, поскольку для непрерывных списков не существует ничего схожего с ней. Мы стремимся к обеспечению полной совместимости различных реализаций простых списков, поэтому нам не следует предоставлять прикладной программе непосредственный доступ к процедуре *AddNode*. В модуле *Turbo Pascal* мы можем легко добиться такого положения, поместив *AddNode* в секцию **implementation** (разумеется), но не включая *AddNode* в секцию **interface** модуля.

### 4. Просмотр списка

Используя для продвижения по связанному списку на одну позицию фундаментальную операцию *current := current↑.nextnode*, нетрудно написать процедуру, которая выполняет просмотр списка, посещая при этом каждый узел. В этой процедуре обратите особое внимание на использование

процедурного параметра и на различающиеся способы объявлений для Turbo Pascal и стандартного Pascal.

```
{ для Turbo Pascal: }
procedure TraverseList (var L: list; Visit: processlistentry);      { Просмотр }
{ для стандартного Pascal:
procedure Traverse List(var L: list; procedure Visit(var x: listentry)); }
{ Pre: Список L уже создан.
  Post: Действие, определяемое процедурой Visit, было выполнено
        над каждым элементом списка L, начиная с его первого
        элемента и по очереди над всеми остальными. }
var current: listpointer;
begin                                     { процедура TraverseList }
    current := L.head;
    while current <> nil do
        begin
            Visit(current↑.entry);
            current := current↑.nextnode
        end
    end;                                { процедура Traverselist }
```

## 5. Остальные операции

Программирование оставшихся операций для простых связанных списков выполняется очевидным образом, и мы оставим его для упражнений. Чтобы очистить список, его необходимо просмотреть и уничтожить каждый узел. Программу просмотра придется написать, а не использовать TraverseList, поскольку по ходу просмотра связи должны уничтожаться. В этой процедуре просмотра-уничтожения вам понадобятся два указателя, один на уничтожаемый узел, а другой — на следующий за ним. В противном случае, уничтожая узел, вы потеряете связь, которой вы должны следовать для перехода к узлу, следующему за уничтожаемым.

### D.3.5. Советы для программистов

Завершая этот раздел, мы хотели бы дать несколько советов по программированию связанных списков и предостеречь от распространенных ошибок.

подсказки  
и ловушки

1. Рисуите диаграммы соответствующих частей связанного списка, отображающие состояние списка до и после операции. Показывайте на этих диаграммах указатели, имеющие отношение к операции, а также то, каким образом они изменяются.
2. При решении вопроса, в каком порядке следует размещать значения в полях указателей при выполнении тех или иных изменений, учтите, что обычно удобно сначала задать значения предварительно инициализированным указателям, далее тем, значение которых равно **nil**, и, наконец, оставшимся указателям. После копирования одной переменной-указателя в другую, первая переменная освобождается и может принять новое значение.

- неопределенные связи
- крайние случаи
- множественное снятие ссылки
- переменные-псевдонимы
3. Внимательно следите за тем, чтобы к концу вашей процедуры не оставалось неопределенных связей, в частности, неинициализированных связей в новых узлах или «висящих» связей в старых узлах, т. е. связей с уже отсутствующими узлами.
  4. Обязательно проверяйте, правильно ли обслуживает ваша процедура пустой список, а также список с единственным элементом.
  5. Никогда не используйте выражения вроде  $p1 \uparrow .nextnode \uparrow .nextnode$ , даже несмотря на то, что они могут быть синтаксически правильными. Одиночная переменная должна использовать в своем определении только один указатель. Конструкции с повторяющимися ссылками обычно указывают, что процедуру можно улучшить, переосмыслив использование объявленных в ней указателей. Иногда оказывается удобным включить в алгоритм дополнительные переменные-указатели, чтобы в результате ни в одной переменной не использовалось более одного знака указателя ( $\uparrow$ ).
  6. Вполне допустимо, чтобы два (или несколько) различных указателей указывали на один и тот же узел. Поскольку к такому узлу можно обратиться посредством нескольких различных имен, его называют переменной-псевдонимом. Узел можно изменить посредством одного имени, а позже использовать под другим именем, возможно не осознавая при этом, что узел был изменен. Один из указателей мог быть перемещен на другой узел, и второй остаться «висящим». Переменные-псевдонимы являются источником неприятностей, и, насколько это возможно, их следует избегать. Во всех случаях, когда у вас есть два указателя, указывающих на один и тот же узел, удостоверьтесь в том, что вы понимаете, зачем это нужно, и не забывайте, что изменение одной ссылки требует такого же изменения другой.

### Упражнения D.3

Предлагаемые ниже упражнения основаны на таких объявлениях:

```

type pointer =  $\uparrow$ node;
var p, q, r:   pointer;
    x, y, z:    node;

```

В этих объявлениях предполагается, что тип `node` был объявлен ранее по ходу программы.

**E1.** Для каждого из приведенных ниже предложений либо опишите его результат, либо объясните, почему оно недопустимо.

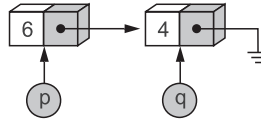
- |                                              |                                                  |                                                                   |
|----------------------------------------------|--------------------------------------------------|-------------------------------------------------------------------|
| (a) <code>new(p)</code>                      | (f) <code>r := nil</code>                        | (k) <code>dispose(r)</code>                                       |
| (b) <code>new(q<math>\uparrow</math>)</code> | (g) <code>z := p<math>\uparrow</math></code>     | (l) <code>x := new (p)</code>                                     |
| (c) <code>new(x)</code>                      | (h) <code>p := <math>\uparrow</math>x</code>     | (m) <code>q<math>\uparrow</math> := nil</code>                    |
| (d) <code>p := r</code>                      | (i) <code>dispose(y)</code>                      | (n) <code>p<math>\uparrow</math> := x<math>\uparrow</math></code> |
| (e) <code>q := y</code>                      | (j) <code>dispose(p<math>\uparrow</math>)</code> | (o) <code>z := nil</code>                                         |

**E2.** Напишите Pascal-процедуру для обмена значениями указателей `p` и `q`. После выполнения этой процедуры указатель `p` должен указывать на узел, на который первоначально указывал `q`, и наоборот.

- Е3.** Напишите Pascal-процедуру для обмена значениями динамических переменных, на которые указывают  $p$  и  $q$ . После выполнения этой процедуры переменная  $p\uparrow$  должна иметь значение, которое ранее имела переменная  $q\uparrow$ , и наоборот.
- Е4.** Напишите Pascal-процедуру, в результате выполнения которой  $p$  указывает на тот же узел что и  $q$ , а узел, на который первоначально указывал указатель  $p$ , отбрасывается.
- Е5.** Напишите Pascal-процедуру, создающую новую переменную, на которую указывает указатель  $p$ , и содержимое которой будет тем же самым, что и у узла, на который указывает  $q$ .

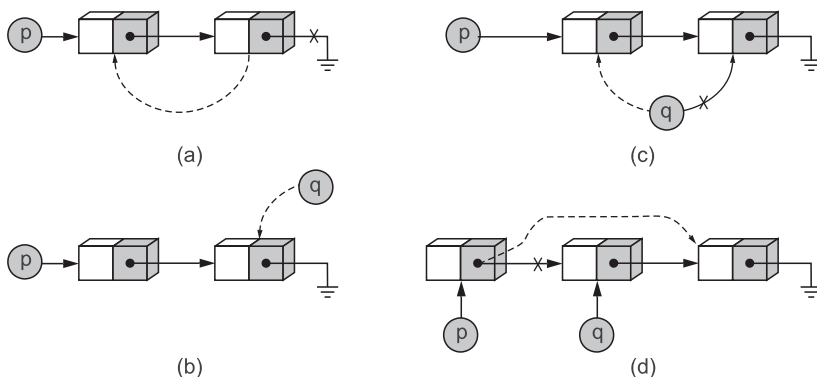
Все последующие упражнения используют объявления для простого связного списка, приведенные в настоящем разделе.

- Е6.** Предположим, что имеются две переменных-указателя и два узла, поля `nextnode` которых выглядят так, как показано на приведенном ниже рисунке. Поле `entry` каждого узла содержит целое число.



Определите, какое из приведенных ниже выражений истинно, какое ложно, и какое не имеет смысла. Если выражение бессмысленно, укажите, почему.

- |                                                       |                                     |
|-------------------------------------------------------|-------------------------------------|
| (a) $p\uparrow.\text{nextnode} = q$                   | (g) $p\uparrow = 6$                 |
| (b) $q\uparrow.\text{nextnode} = \text{nil}$          | (h) $q = 4$                         |
| (c) $q\uparrow = 4$                                   | (i) $q\uparrow.\text{entry} = 4$    |
| (d) $p < q$                                           | (j) $p\uparrow.\text{nextnode} = 4$ |
| (e) $p\uparrow.\text{entry} < q\uparrow.\text{entry}$ | (k) $q\uparrow = \text{nil}$        |
| (f) $p\uparrow < q\uparrow$                           | (l) $p\uparrow = 6$                 |
- Е7.** Для каждого из приведенных ниже предложений нарисуйте диаграмму, показывающую результат его выполнения. В качестве исходных условий воспользуйтесь рисунком из упражнения Е6. Некоторые из указанных ниже предложений приведут к ошибке; объясните, почему.
- |                                                              |                                                              |
|--------------------------------------------------------------|--------------------------------------------------------------|
| (a) $p := p\uparrow.\text{nextnode}$                         | (g) $p := q$                                                 |
| (b) $q\uparrow.\text{nextnode} := p$                         | (h) $p\uparrow.\text{nextnode} := p$                         |
| (c) $p\uparrow.\text{entry} := 5$                            | (i) $p := \text{nil}$                                        |
| (d) $p\uparrow.\text{entry} := q\uparrow.\text{entry}$       | (j) $p\uparrow.\text{nextnode} := \text{nil}$                |
| (e) $p\uparrow.\text{nextnode} := q\uparrow.\text{nextnode}$ | (k) $p\uparrow.\text{nextnode}\uparrow.\text{nextnode} := p$ |
| (f) $p\uparrow := q\uparrow$                                 | (l) $q\uparrow.\text{nextnode}\uparrow.\text{nextnode} := q$ |
- Е8.** Напишите одно Pascal-предложение, приводящее к изменению, обозначенному пунктирной линией на каждой из изображенных ниже диаграмм.



**Е9.** Напишите процедуры и функции для оставшихся операций со связными простыми списками в соответствии с реализациями, приведенными в этом разделе:

- (a) CreateList
- (b) ClearList
- (c) ListEmpty
- (d) ListFull
- (e) ListSize

**Е10.** Учитывая тривиальный характер функции ListFull для связной реализации, объясните важность ее включения в пакет или модуль.

**Е11.** Предположите, что в объявлении типа для связного простого списка опущено поле count. Напишите новый вариант ListSize, который определяет размер списка путем подсчета его узлов. Какие изменения придется внести в этом случае в другие процедуры и функции?

### Программный проект D.3

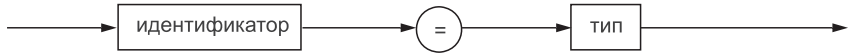
**P1.** Соберите вместе все процедуры и функции для связных простых списков в include-пакет или в модуль Turbo Pascal. Замените им пакет или модуль Turbo Pascal для непрерывных списков в демонстрационной программе раздела 2.3.3. Если все программы написаны правильно, демонстрационная программа будет работать правильно без каких-либо изменений, и в точности так же, как и для непрерывных списков.

## D.4. Синтаксические диаграммы

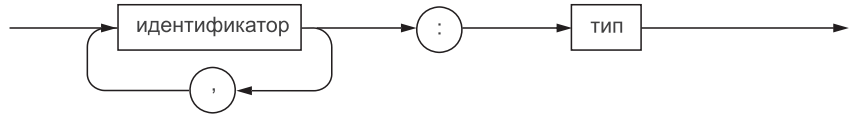
Синтаксис языка Pascal определяется прохождением по диаграммам в направлении, указанном стрелками. Символические обозначения или слова в кружках или овалах должны включаться в точности так, как они указаны на рисунках. Все эти ключевые слова языка Pascal написаны прописными буквами. Прямоугольные рамки обозначают другие синтаксические диаграммы. Все производные классы вроде «идентификатор функции», «идентификатор переменной» или «идентификатор типа» имеют тот же синтаксис, что и «идентификатор». Уточняющие слова в фигурных скобках приведены лишь для разъяснения смысла диаграмм.



определение типа

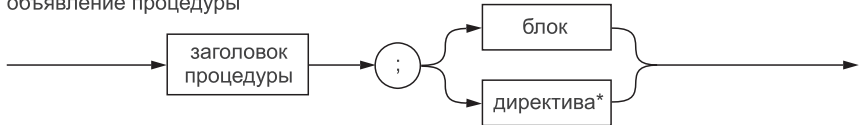


объявление переменной

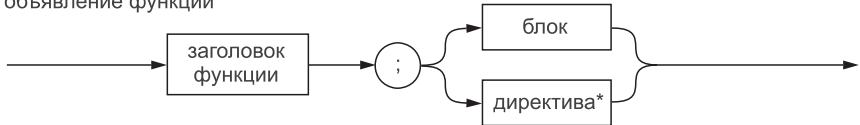


подпрограммы

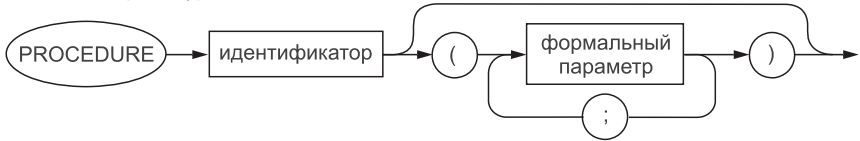
объявление процедуры



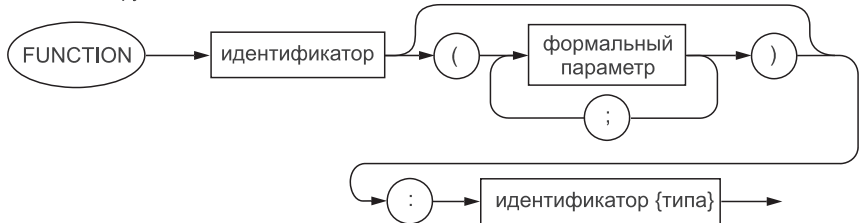
объявление функции



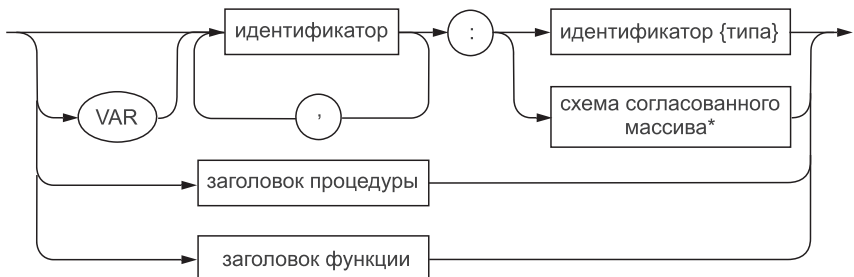
заголовок процедуры



заголовок функции

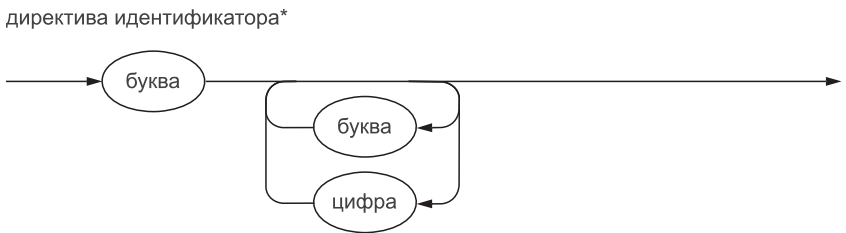
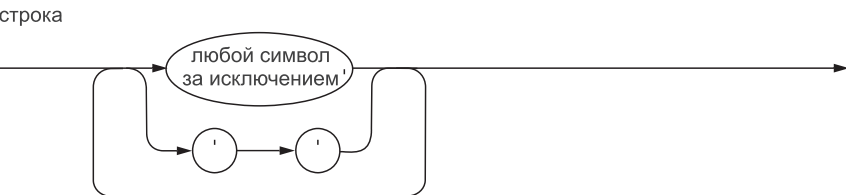
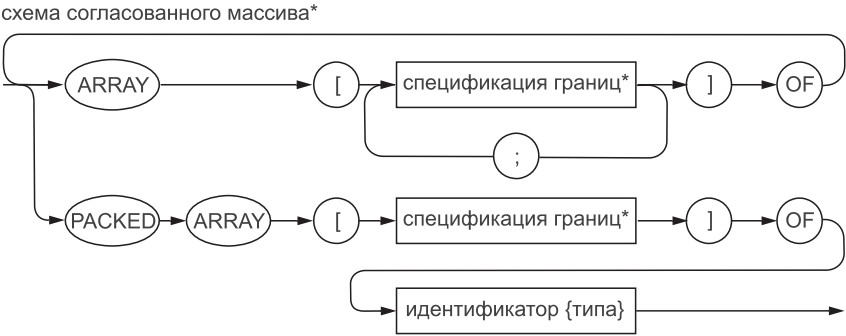


формальный параметр

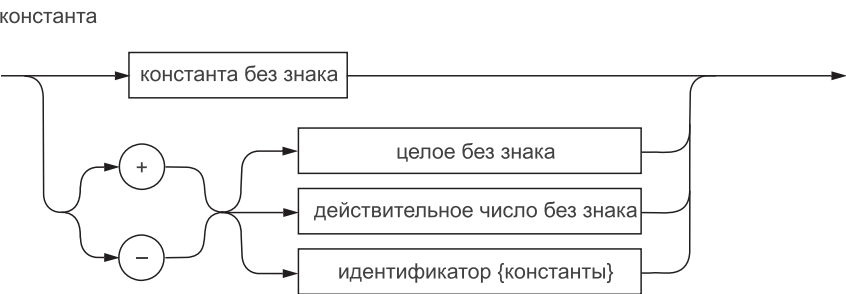




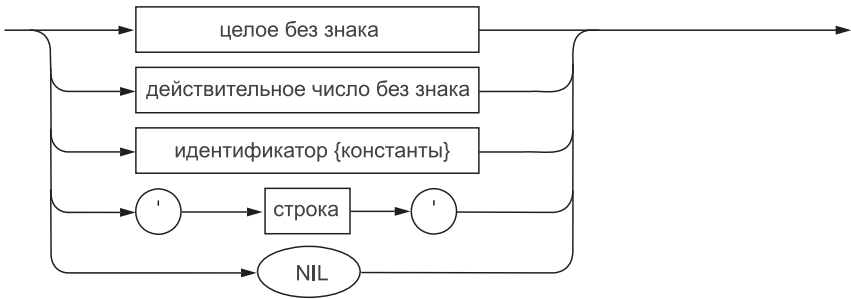
реализуется  
не всегда



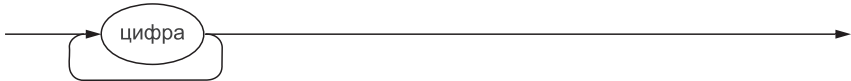
числа



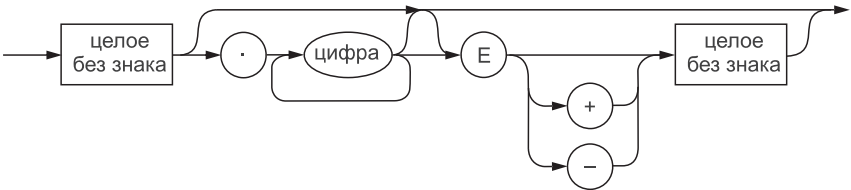
константа без знака



целое без знака

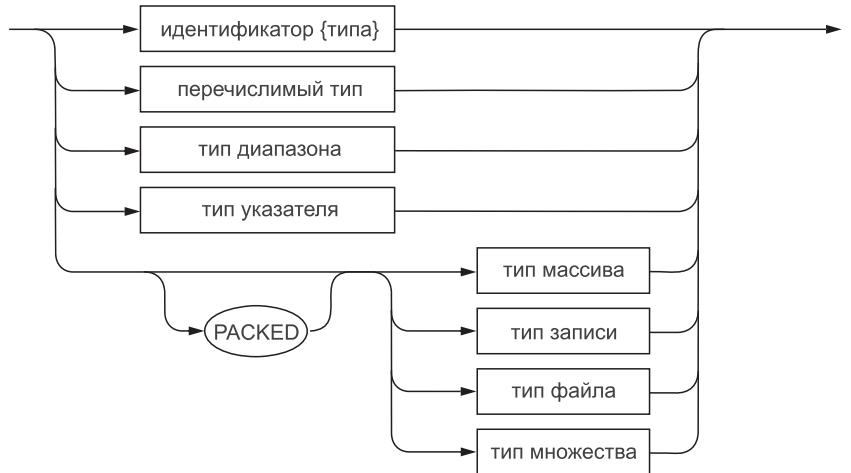


действительное число без знака

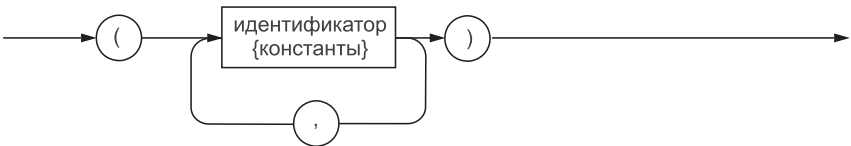


типы

тип



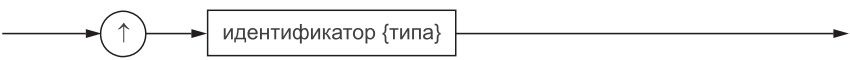
перечислимый тип



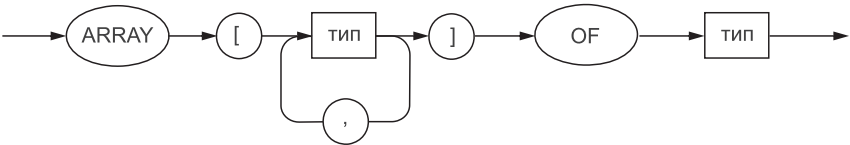
тип диапазона



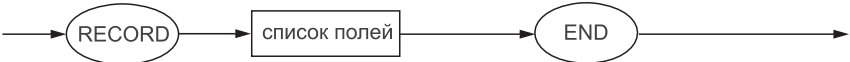
тип указателя



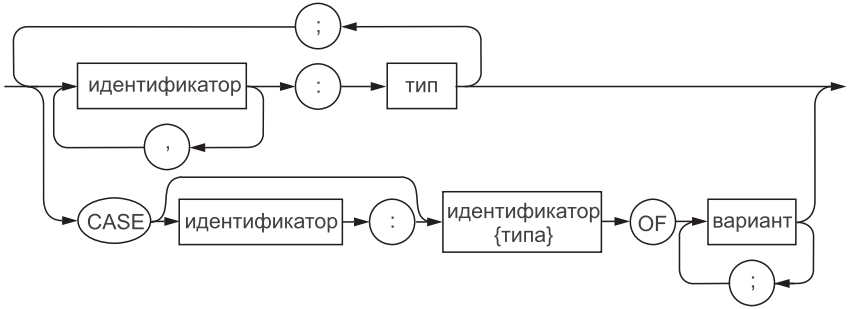
тип массива



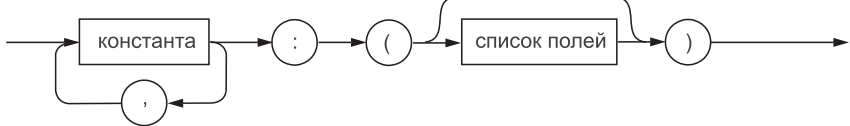
тип записи



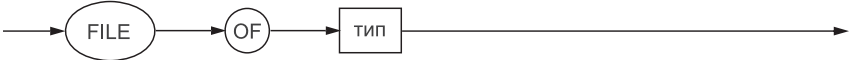
список полей



вариант



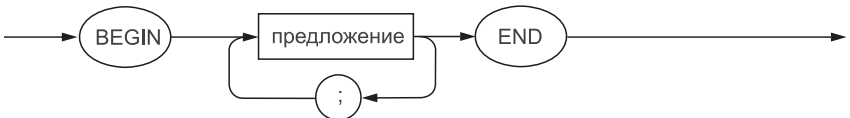
тип файла



тип множества

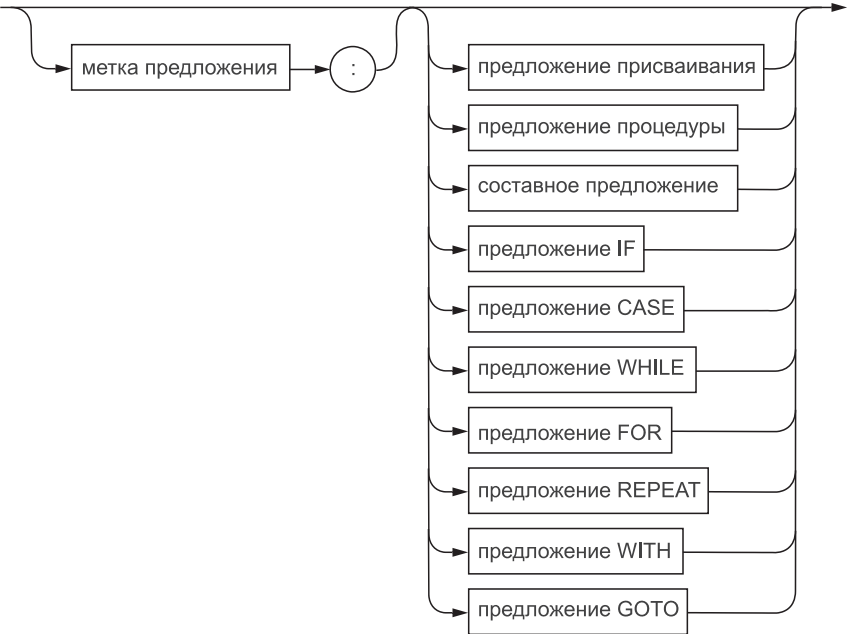


составное предложение

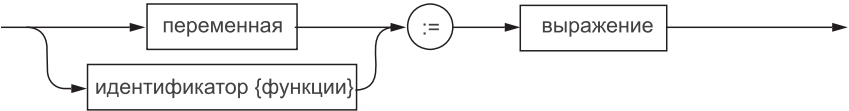


предложения

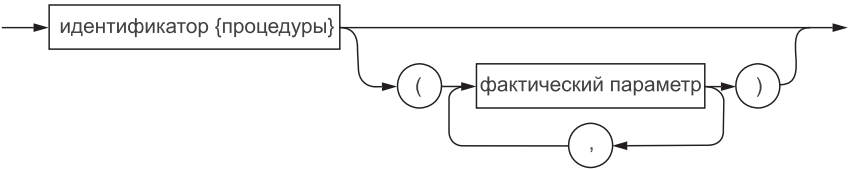
предложение



предложение присваивания

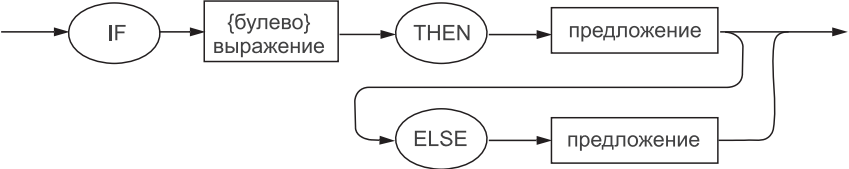


предложение процедуры

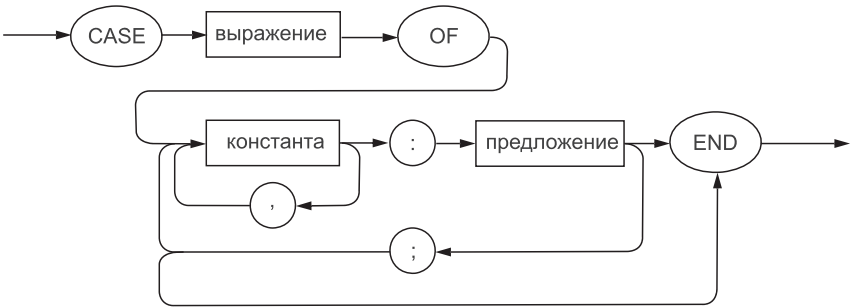


предложения  
выбора

предложение IF



предложение CASE

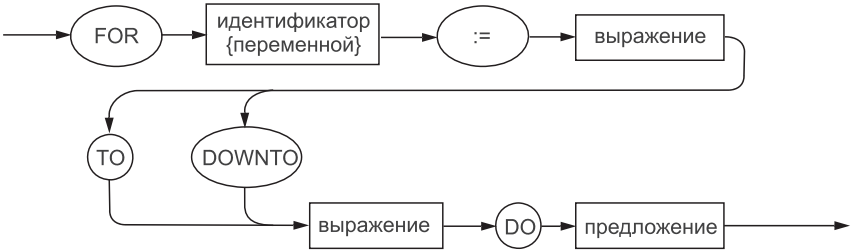


предложения  
циклов

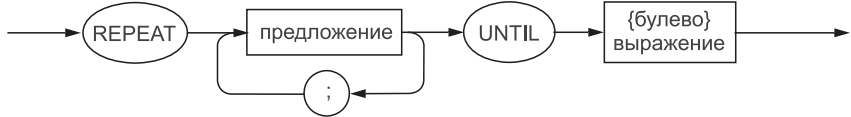
предложение WHILE



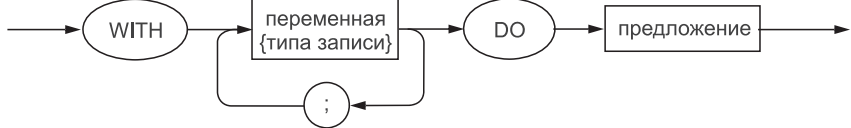
предложение FOR



предложение REPEAT



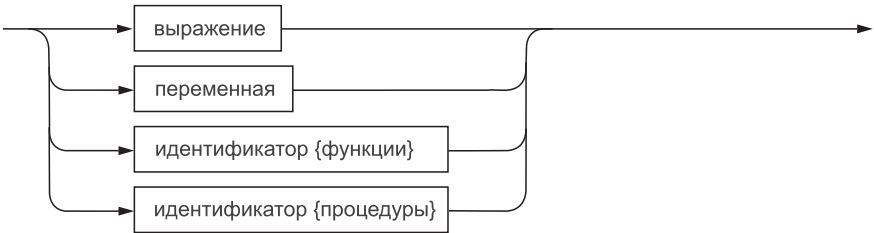
предложение WITH



предложение GOTO

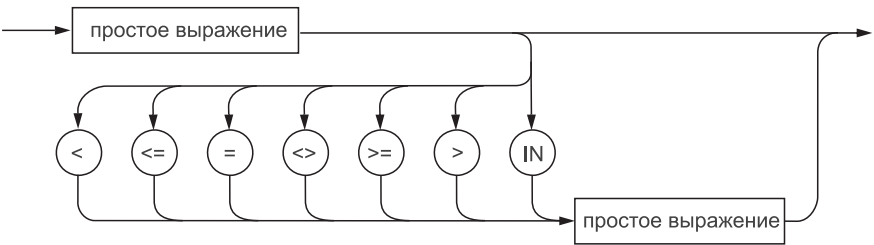


фактический параметр

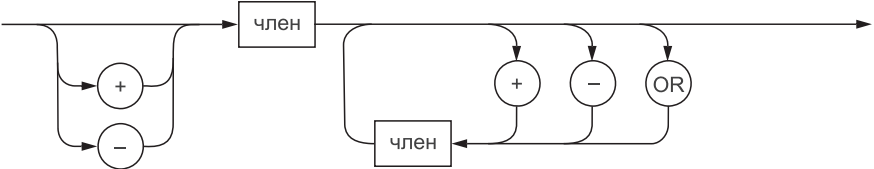


выражения

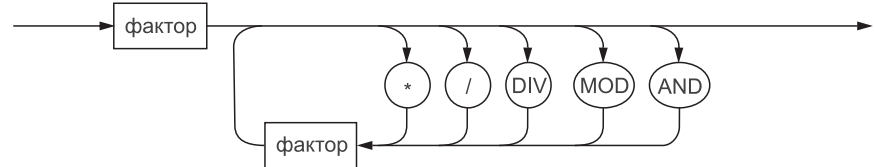
выражение



простое выражение



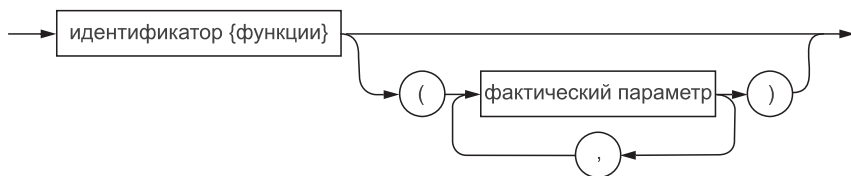
член



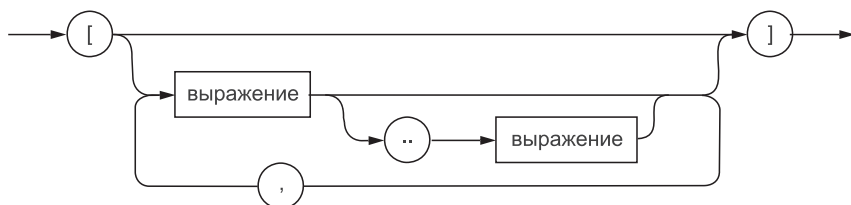
фактор



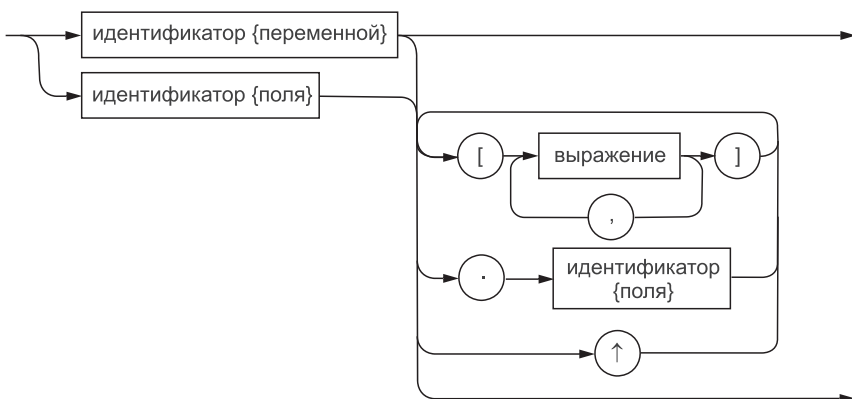
обозначение функции



значение множества



переменная



## D.5. Общие правила

### D.5.1. Идентификаторы

Идентификатор может иметь любую длину, подчиняясь следующим правилам.

1. Внутри идентификатора недопустимы пробелы.
2. Идентификатор нельзя разделить на две строки текста. (Таким образом, максимальная длина идентификатора равна максимальной длине строки экрана, часто 80 или 132 символа.)
3. Стандартный Pascal требует, чтобы идентификаторы различались в своих первых восьми символах. (Многие компиляторы с языка Pascal обходят это правило и рассматривают идентификаторы как разные, если они различаются в любой своей позиции.)
4. Строчные и прописные символы могут использоваться в равном успехом по желанию программиста (если оба набора имеются на компью-

тере). При этом большинство (но не все) компиляторов не различают строчные и прописные буквы, рассматривая их как одно и то же. Однако ради переносимости одни и те же идентификаторы всегда следует писать в точности одинаково.

зарезервированные  
слова

5. Перечисленные ниже *зарезервированные слова* не могут использоваться в качестве идентификаторов, или вообще с любыми целями, отличными от тех, для которых они предназначены, и которые определены в соответствующих синтаксических диаграммах.

|              |                 |              |                  |              |
|--------------|-----------------|--------------|------------------|--------------|
| <b>and</b>   | <b>downto</b>   | <b>if</b>    | <b>or</b>        | <b>then</b>  |
| <b>array</b> | <b>else</b>     | <b>in</b>    | <b>packed</b>    | <b>to</b>    |
| <b>begin</b> | <b>end</b>      | <b>label</b> | <b>procedure</b> | <b>type</b>  |
| <b>case</b>  | <b>file</b>     | <b>mod</b>   | <b>program</b>   | <b>until</b> |
| <b>const</b> | <b>for</b>      | <b>nil</b>   | <b>record</b>    | <b>var</b>   |
| <b>div</b>   | <b>function</b> | <b>not</b>   | <b>repeat</b>    | <b>while</b> |
| <b>do</b>    | <b>goto</b>     | <b>of</b>    | <b>set</b>       | <b>with</b>  |

символ  
подчеркивания

6. В Turbo Pascal также зарезервированы слова **asm**, **constructor**, **destructor**, **implementation**, **inline**, **interface**, **object**, **shl**, **shr**, **string**, **unit**, **uses** и **xor**.
7. Хотя эта возможность и не предусмотрена в стандартном Pascal, многие компиляторы допускают использование внутри идентификаторов символа подчеркивания '\_'. Как этот символ рассматривается — зависит от компилятора. Он может игнорироваться, рассматриваться как буква или как цифра.

стандартные  
идентификаторы

8. Перечисленные ниже *стандартные идентификаторы* входят в язык Pascal как предопределенные слова. Программист может использовать их для целей, отличных от их стандартного назначения, хотя обычно это не следует делать, поскольку при объявлении для этих слов иного назначения стандартное теряется. Вы, например, можете объявить слово **write** переменной, но если вы так поступите, мы уже не сможем использовать стандартные процедуры, которые выводят информацию в файл или на терминал. Полный список стандартных идентификаторов выглядит следующим образом.

|                |                |               |                |                |
|----------------|----------------|---------------|----------------|----------------|
| <b>abs</b>     | <b>eoln</b>    | <b>new</b>    | <b>read</b>    | <b>sqrt</b>    |
| <b>arctan</b>  | <b>exp</b>     | <b>odd</b>    | <b>readln</b>  | <b>succ</b>    |
| <b>Boolean</b> | <b>false</b>   | <b>ord</b>    | <b>real</b>    | <b>text</b>    |
| <b>char</b>    | <b>get</b>     | <b>output</b> | <b>reset</b>   | <b>true</b>    |
| <b>chr</b>     | <b>input</b>   | <b>pack</b>   | <b>rewrite</b> | <b>trunc</b>   |
| <b>cos</b>     | <b>integer</b> | <b>page</b>   | <b>round</b>   | <b>unpack</b>  |
| <b>dispose</b> | <b>in</b>      | <b>pred</b>   | <b>sin</b>     | <b>write</b>   |
| <b>eof</b>     | <b>maxint</b>  | <b>put</b>    | <b>sqr</b>     | <b>writeln</b> |

9. По поводу выбора идентификаторов см. раздел 1.3.

## D.5.2. Правила использования пробелов

1. Пробелы не допускаются внутри идентификаторов, зарезервированных слов, чисел и специальных символических обозначений, составленных из нескольких символов (например, **<=**, **<>**, **>=**, **:=**).



2. Для отделения друг от друга двух идентификаторов, зарезервированных слов или чисел, требуется включение по меньшей мере одного пробела. (Например,

ForX := AtoBdo

является синтаксически правильным предложением присваивания (если были объявлены соответствующие переменные), но отнюдь не началом цикла **for**.)

3. Внутри апострофов ' ... ' (т. е. в константной строке) пробелы рассматриваются как обычные символы.
4. За исключением отмеченных выше случаев все пробелы игнорируются и могут включаться в предложения в любом месте.
5. Конец строки рассматривается в точности так же, как и пробел, и, следовательно, предложение может переходить на следующую строку и продолжаться на ней в любом месте, где допустим пробел.
6. Комментарий рассматривается в точности так же, как и пробел: его можно включать в предложение в любом месте, где допустим пробел, при этом комментарий может продолжаться на следующих строках.

### D.5.3. Указания по формату программы

Приводимые ниже советы по форматированию программных строк используются в большей части программ этой книги.

1. Слова **const**, **type**, **var** и **record** пишутся на отдельной строке, если только не оказывается, что все объявление уместится на одной строке.
2. На каждой строке объявляется только один элемент, за исключением тех случаев, когда логически связанные элементы объявляются в точности одинаково.
3. Строки, содержащие объявляемые элементы, слегка сдвигаются вправо.
4. Слово **end** пишется на отдельной строке и выравнивается со строкой, содержащей соответствующее слово **begin**, **case** или **record**.
5. Предложения или объявления между одним из этих слов или фраз и соответствующим словом **end** слегка сдвигаются вправо.
6. Одиночное предложение, следующее непосредственно за одним из слов **then**, **do**, **repeat** или **else**, слегка сдвигается вправо, но слово **begin** после одного из этих слов не сдвигается.
7. Как правило, на каждой строке пишется только одно предложение присваивания.
8. Каждая альтернатива предложения **case** пишется на новой строке и слегка двигается вправо относительно слова **case**.
9. В длинных программах для разделения логических завершенных фрагментов программы включаются пустые строки.
10. В тексте книги объявления функций и процедур обычно отделяются от главной программы. Однако, когда объявления включаются в предназначенные для них места, они окружаются достаточными пустыми промежутками, чтобы ясно видно, где заканчивается одна подпрограмма и начинается следующая.

## D.5.4. Пунктуация

Синтаксические диаграммы точно описывают правила пунктуации в программах на языке Pascal, и если у вас возникают затруднения, вы в процессе поиска ошибок должны сверяться с синтаксическими диаграммами. Однако для начинающих программистов типичны некоторые общие ошибки, и приводимые ниже советы могут помочь вам в их устранении.

1. Элементы программы, рассматриваемые языком Pascal единообразно, обычно разделяются запятыми, а элементы, рассматриваемые различным образом, разделяются символами точки с запятой. Так, переменные, объявляемые в точности одинаково, разделяются запятыми, как и индексы в (многомерном) массиве, фактические параметры подпрограмм и альтернативы, приводящие к одному и тому же действию в предложении **case**. С другой стороны, символы точки с запятой разделяют объявления переменных (возможно) различающихся типов, отдельные различающиеся предложения, отдельные формальные параметры (которые могут объявляться различным образом) и различные альтернативы в предложении **case**.
2. Символы точки с запятой используются для разделения элементов или предложений, но не для завершения предложений.
3. Недопустимое включение или исключение точки с запятой обычно не приводит к появлению диагностического сообщения об ошибке в строке, где имеется эта ошибка, но, как правило, порождает странное и мало вразумительное диагностическое сообщение, относящее к строке, находящейся *вслед* за ошибочной.
4. Точка с запятой во всех случаях недопустима перед словом **else**.
5. Точка с запятой почти никогда не требуется непосредственно после слова **begin** или непосредственно перед словами **end** или **until**, но избыточное включение ее не является ошибкой.

запятые и символы  
точки с запятой

## D.5.5. Альтернативные знаки

В системах, где недоступны некоторые стандартные символы, допустимы следующие замены:

Вместо ↑ или ^ можно писать @.

Вместо { и } можно писать (\* и \*).

Вместо [ и ] можно писать ( и ).

## D.6. Стандартные объявления

### D.6.1. Константы

Имеются предопределенные константы **false**, **true** и **maxint**.

#### 1. Обычные значения для **maxint**

Для большинства 8- и 16-разрядных машин **maxint** = 32767

Для большинства 32-разрядных машин **maxint** = 2147483647

Для большинства 60-разрядных машин **maxint** = 281474976710655

## 2. Коды ASCII для символов с их числовыми значениями

|       |        |       |      |      |      |       |         |
|-------|--------|-------|------|------|------|-------|---------|
| 0 NUL | 16 DLE | 32 SP | 48 0 | 64 @ | 80 P | 96 `  | 112 p   |
| 1 SOH | 17 DC1 | 33 !  | 49 1 | 65 A | 81 Q | 97 a  | 113 q   |
| 2 STX | 18 DC2 | 34 «  | 50 2 | 66 B | 82 R | 98 b  | 114 r   |
| 3 ETX | 19 DC3 | 35 #  | 51 3 | 67 C | 83 S | 99 c  | 115 s   |
| 4 EOT | 20 DC4 | 36 \$ | 52 4 | 68 D | 84 T | 100 d | 116 t   |
| 5 ENQ | 21 NAK | 37 %  | 53 5 | 69 E | 85 U | 101 e | 117 u   |
| 6 ACK | 22 SYN | 38 &  | 54 6 | 70 F | 86 V | 102 f | 118 v   |
| 7 BEL | 23 ETB | 39 '  | 55 7 | 71 G | 87 W | 103 g | 119 w   |
| 8 BS  | 24 CAN | 40 (  | 56 8 | 72 H | 88 X | 104 h | 120 x   |
| 9 HT  | 25 EM  | 41 )  | 57 9 | 73 I | 89 Y | 105 i | 121 y   |
| 10 LF | 26 SUB | 42 *  | 58 : | 74 J | 90 Z | 106 j | 122 z   |
| 11 VT | 27 ESC | 43 +  | 59 ; | 75 K | 91 [ | 107 k | 123 {   |
| 12 FF | 28 FS  | 44 ,  | 60 < | 76 L | 92 \ | 108 l | 124     |
| 13 CR | 29 GS  | 45 -  | 61 = | 77 M | 93 ] | 109 m | 125 }   |
| 14 SO | 30 RS  | 46 .  | 62 > | 78 N | 94 ^ | 110 n | 126 ~   |
| 15 SI | 31 US  | 47 /  | 63 ? | 79 O | 95 _ | 111 o | 127 DEL |

Альтернативные символы:

Код 39 может печататься как '

Код 94 может печататься как ^ или ↑

Код 39 может печататься как ¬

## 3. Коды EBCDIC для символов с их числовыми значениями

|        |        |        |       |       |       |       |       |
|--------|--------|--------|-------|-------|-------|-------|-------|
| 0 NUL  | 21 NL  | 43 CU2 | 79    | 124 @ | 150 o | 195 C | 227 T |
| 1 SOH  | 22 BS  | 45 ENQ | 80 &  | 125 ' | 151 p | 196 D | 228 U |
| 2 STX  | 23 IL  | 46 ACK | 90 !  | 126 = | 152 q | 197 E | 229 V |
| 3 ETX  | 24 CAN | 47 BEL | 91 \$ | 127 " | 153 r | 198 F | 230 W |
| 4 PF   | 25 EM  | 50 SYN | 92 *  | 129 a | 155 } | 199 G | 231 X |
| 5 HT   | 26 CC  | 52 PN  | 93 )  | 130 b | 161 ~ | 200 H | 232 Y |
| 6 LC   | 27 CU1 | 53 RS  | 94 ;  | 131 c | 162 s | 201 I | 233 Z |
| 7 DEL  | 28 IFS | 54 UC  | 95 ¬  | 132 d | 163 t | 208 } | 240 0 |
| 10 SMM | 29 IGS | 55 EOT | 96 -  | 133 e | 164 u | 209 J | 241 1 |
| 11 VT  | 30 IRS | 59 CU3 | 97 /  | 134 f | 165 v | 210 K | 242 2 |
| 12 FF  | 31 IUS | 60 DC4 | 106   | 135 g | 166 w | 211 L | 243 3 |
| 13 CR  | 32 DS  | 61 NAK | 107 , | 136 h | 167 x | 212 M | 244 4 |
| 14 SO  | 33 SOS | 63 SUB | 108 % | 137 i | 168 y | 213 N | 245 5 |
| 15 SI  | 34 FS  | 64 SP  | 109 _ | 139 { | 169 z | 214 O | 246 6 |
| 16 DLE | 36 BYP | 74 Ÿ   | 110 > | 145 i | 173 [ | 215 P | 247 7 |
| 17 DC1 | 37 LF  | 75 .   | 111 ? | 146 k | 189 ] | 216 Q | 248 8 |
| 18 DC2 | 38 ETB | 76 <   | 121 ' | 147 l | 192 { | 217 R | 249 9 |
| 19 DC3 | 39 ESC | 77 (   | 122 : | 148 m | 193 A | 224 \ | 250   |
| 20 RES | 42 SM  | 78 +   | 123 # | 149 n | 194 B | 226 S |       |

4. Смысл некоторых управляющих кодов

Управляющими являются коды ASCII со значениями до 32, а также код 127. В таблице кодов EBCDIC управляющими являются коды со значениями до 64. Эти коды при выводе на устройство не отображаются в виде символов, а вызывают некоторые специальные действия. Ниже приведено назначение некоторых управляющих кодов.

|      |                        |    |                             |     |                     |
|------|------------------------|----|-----------------------------|-----|---------------------|
| NULL | ноль<br>(игнорируется) | HT | горизонтальная<br>табуляция | CR  | возврат<br>каретки  |
| ETX  | конец текста           | LF | перевод строки              | ESC | выход               |
| BEL  | звуковой сигнал        | VT | вертикальная<br>табуляция   | SP  | пробел              |
| BS   | шаг назад              | FF | перевод страницы            | DEL | удаление<br>символа |

D.6.2. Типы

Предобъявленными типами являются следующие:

integer;  
real;  
Boolean = (false, true);  
char;  
text = **packed file of char**;

Термины, используемые для описания различных категорий типов, приведены на рис. D.7.

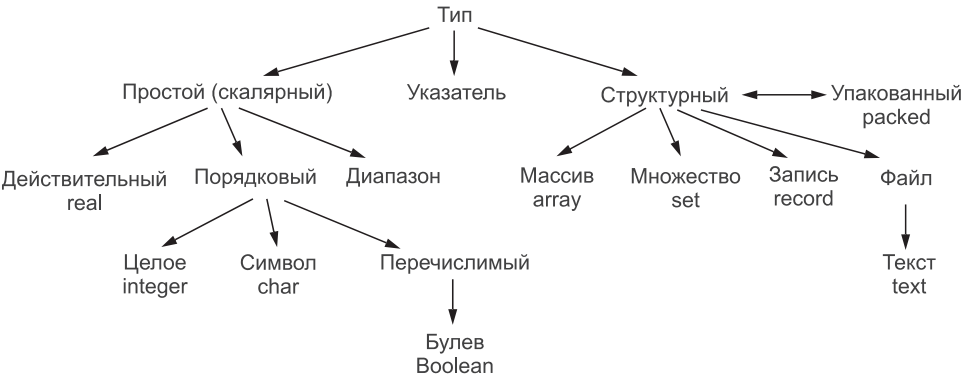


Рис. D.7. Категории типов

D.6.3. Переменные

Предобъявленными переменными являются только файлы с типом text:  
input output

## D.6.4. Процедуры

Имеются следующие предобъявленные процедуры.

Для ввода и вывода:

reset, rewrite, get, put, read, write, readln, writeln, page.

Для указательных типов:

new, dispose.

Для упакованных данных:

pack, unpack.

## D.6.5. Функции

| <i>Имя</i>                    | <i>Аргумент</i>       | <i>Результат</i>               | <i>Действие</i>                        |
|-------------------------------|-----------------------|--------------------------------|----------------------------------------|
| <i>Арифметические функции</i> |                       |                                |                                        |
| abs                           | integer, real         | <i>тот же, что у аргумента</i> | <i>Абсолютное значение</i>             |
| sqr                           | integer, real         | <i>тот же, что у аргумента</i> | <i>Квадрат аргумента</i>               |
| sqrt                          | integer, real         | real                           | <i>Квадратный корень из аргумента</i>  |
| exp                           | integer, real         | real                           | <i>Экспонента</i>                      |
| ln                            | integer, real         | real                           | <i>Натуральный логарифм</i>            |
| sin                           | integer, real         | real                           | <i>Синус</i>                           |
| cos                           | integer, real         | real                           | <i>Косинус</i>                         |
| arctan                        | integer, real         | real                           | <i>Арктангенс</i>                      |
| <i>Преобразование типа</i>    |                       |                                |                                        |
| chr                           | integer               | char                           | <i>Символ с заданным кодом</i>         |
| ord                           | <i>порядковый тип</i> | integer                        | <i>Порядковый код аргумента</i>        |
| round                         | real                  | integer                        | <i>Округление до ближайшего целого</i> |
| trunc                         | real                  | integer                        | <i>Усечение до целой части</i>         |
| <i>Обработка файлов</i>       |                       |                                |                                        |
| eof                           | <i>файловый тип</i>   | Boolean                        | <i>Проверяет конец файла</i>           |
| eoln                          | text                  | Boolean                        | <i>Проверяет конец строки</i>          |
| <i>Различные</i>              |                       |                                |                                        |
| odd                           | integer               | Boolean                        | <i>Целое нечетно?</i>                  |
| succ                          | <i>порядковый тип</i> | <i>тот же тип</i>              | <i>Следующее по порядку значение</i>   |
| pred                          | <i>порядковый тип</i> | <i>тот же тип</i>              | <i>Предыдущее по порядку значение</i>  |

# D.7. Операторы

| Оператор                      | Типы операндов                                           | Тип<br>результата | Действие                            |
|-------------------------------|----------------------------------------------------------|-------------------|-------------------------------------|
| <i>Присваивание</i>           |                                                          |                   |                                     |
| <b>:=</b>                     | любой тип кроме файла                                    | тот же тип        | копирует правый операнд в левый     |
| <i>Арифметические</i>         |                                                          |                   |                                     |
| <b>+</b>                      | integer, real                                            | тот же тип        | сложение или одноместный плюс       |
| <b>–</b>                      | integer, real                                            | тот же тип        | вычитание или одноместный минус     |
| <b>*</b>                      | integer, real                                            | тот же тип        | умножение                           |
| <b>div</b>                    | integer                                                  | integer           | деление с усечением                 |
| <b>mod</b>                    | integer                                                  | integer           | остаток после <b>div</b>            |
| <b>/</b>                      | integer, real                                            | real              | деление                             |
| <i>Сравнения</i>              |                                                          |                   |                                     |
| <b>=</b>                      | любой тип кроме файла                                    | Boolean           | равенство                           |
| <b>&lt;&gt;</b>               | любой тип кроме файла                                    | Boolean           | неравенство                         |
| <b>&lt;</b>                   | любой простой тип                                        | Boolean           | меньше чем, предшествует            |
| <b>&gt;</b>                   | любой простой тип                                        | Boolean           | больше чем, следует                 |
| <b>&lt;=</b>                  | любой простой тип                                        | Boolean           | меньше чем или равно                |
| <b>&gt;=</b>                  | любой простой тип                                        | Boolean           | больше чем или равно                |
| <i>Логические</i>             |                                                          |                   |                                     |
| <b>not</b>                    | Boolean                                                  | Boolean           | логическое отрицание                |
| <b>and</b>                    | Boolean                                                  | Boolean           | конъюнкция                          |
| <b>or</b>                     | Boolean                                                  | Boolean           | дизъюнкция                          |
| <i>Операции с множествами</i> |                                                          |                   |                                     |
| <b>+</b>                      | любой тип множества                                      | тот же тип        | объединение множеств                |
| <b>*</b>                      | любой тип множества                                      | тот же тип        | пересечение множеств                |
| <b>–</b>                      | любой тип множества                                      | тот же тип        | разность множеств                   |
| <b>&lt;=</b>                  | любой тип множества                                      | Boolean           | вхождение левого множества в правое |
| <b>&gt;=</b>                  | любой тип множества                                      | Boolean           | вхождение правого множества в левое |
| <b>in</b>                     | левый: порядковый тип                                    | Boolean           | принадлежность множеству            |
| <b>[ ]</b>                    | правый: множество типа левого операнда<br>порядковый тип | set of тип        | конструирование множества операндов |

*Приоритет двуместных операторов*

|        |                                                                                         |
|--------|-----------------------------------------------------------------------------------------|
| Высший | 1. <b>*</b> <b>/</b> <b>div</b> <b>mod</b> <b>and</b>                                   |
|        | 2. <b>+</b> <b>–</b> <b>or</b>                                                          |
|        | 3. <b>=</b> <b>&lt;</b> <b>&lt;=</b> <b>&gt;</b> <b>&gt;=</b> <b>&lt;&gt;</b> <b>in</b> |
| Низший | 4. <b>:=</b>                                                                            |

# Предметный указатель

---

2-дерево 306, 525

взаимосвязь с двоичным деревом 483

длина внешнего пути 314

теорема о длине пути 308

число вершин на уровне 306

Ada, язык программирования 64, 675

Add, калькулятор для полиномов 218

AddList, простой список

связная реализация 716

спецификация 66

AddNeighbors, игра «Жизнь» 86, 439

AddNode, связный простой список 717

AddQueen, задача о восьми ферзях 146, 151

ADT, см. **Абстрактный тип данных** 222

Aho, Alfred V. 181

Airport, программа моделирования аэропорта 198

Alagic, Suad 100

APL, язык программирования 405

Append, очередь

реализация 191

спецификация 183

AppendNode, связная очередь, реализация 206

AppendTerm, калькулятор для полиномов 216

Arbib, Michael A. 100

Argo, G. 523

ASCII коды 734

AVL-дерево 486

алгоритмы на языке Pascal 488

анализ 493, 498

в модели гардероба (проект) 500

включение узла 488

высота 498

двойной поворот 491

демонстрационная программа (проект) 500

и дерево Фибоначчи 498

извлечение информации (проект) 500

определение 487

повороты 490

процедуры:

InsertAVL 489

RightBalance 492

RotateLeft 491

разреженное 498

связь с красно-черным деревом 566

удаление узла 494

фактор сбалансированности 487

Basic, язык программирования 387

отсутствие указателей 261

Bayer, R. 568

Bell, T. C. 444

Bentley, Jon L. 326, 392

Bergin, Joseph 326

Berman, Gerald 661

Binary1Search, двоичный поиск 299

анализ процедуры 307

оптимальность процедуры 316

Binary2Search, двоичный поиск 301

анализ процедуры 308

Bird, R. S. 181

Boas, Ralph 661

BreadthFirst, просмотр графа 578

BreadthTopSort, упорядочение графа 582

Brooks, Frederick P., Jr. 100

Brown, S. 144

Brown, William. G. 662

Budden, F. J. 278

BuildHeap, пирамидальная сортировка 382

BuildTree, дерево двоичного поиска 480

Bustard, David 230

B-дерево 538

B\*-дерево (упражнение) 556

алгоритмы поиска и включения на языке

Pascal 542

включение ключа в узел 540, 544

объявления 542

определение 539

поиск ключа-мишени 543

- процедуры
  - Combine 555
  - DeleteTree 549
  - InsertTree 545
  - MoveLeft 554
  - MoveRight 554
  - PushDown 546
  - PushIn 547
  - RecDeleteTree 551
  - Remove 552
  - Restore 553
  - SearchNode 544
  - SearchTree 543
  - Split 547
  - Successor 552
- удаление элемента 548
- C++, язык программирования 64, 675
- case, предложение языка Pascal 29
- Catalan, E. 659, 662
- Chang, Hsi 522
- ChangeString, текстовый редактор 260
- CharToDigit, модуль Utility 679
- ClearList, простой связный список, реализация (упражнение) 244
- ClearList, простой список 66
- ClearQueue, очередь
  - реализация (упражнение) 192
  - спецификация 184
- ClearStack, стек
  - реализация (упражнение) 113
  - связная реализация (упражнение) 114
  - спецификация 106
- ClearTable, хеш-таблица
  - реализация (упражнение) 426
  - реализация со связными цепочками (упражнение) 426
  - спецификация 420
- Clock, модуль CPUTimer 681
- CloseInfile, модуль FileIO 684
- Cobol, язык программирования 132
  - недопустимость рекурсий 132
- Combine, B-дерево 555
- Comer, D. 568
- Concatenate, цепочки символов
  - реализация (упражнение) 251
  - спецификация 248
- Conclude, моделирование работы аэропорта 202
- ConnectSubtrees, дерево двоичного поиска 482
- ConvertToReal, программа оценки выражений 636
- Conway, J. H. 24, 56, 435
- Cooper, Doug 55
- CopyList, список
  - реализация (упражнение) 233
  - спецификация (упражнение) 84
- CopyString, цепочки символов
  - реализация (упражнение) 251
  - спецификация 247
- Cormen, Thomas H. 568
- CPUTimer, модуль 680
- CreateHashTable, программа оценки выражений 625
- CreateList, простой список 65
- CreateQueue, очередь
  - непрерывная очередь, реализация 191
  - связная очередь, реализация 206
  - спецификация 183
- CreateStack, стек
  - реализация 108
  - связная реализация (упражнение) 114
  - спецификация 105
- CreateString, цепочки символов
  - реализация (упражнение) 250
  - спецификация 247
- CreateTable, хеш-таблица
  - реализация (упражнение) 426
  - реализация со связными цепочками (упражнение) 426
  - спецификация 420
- CreateTree, двоичное дерево 453
- Darlington, John 392
- DataMake, генератор тестовых файлов 686
- DeleteFirst, список (упражнение) 233
- DeleteKeyTree, дерево двоичного поиска 472
- DeleteLast, список (упражнение) 233
- DeleteList, простой список
  - непрерывная реализация (упражнение) 244
  - связная реализация (упражнение) 244
  - спецификация 232
- DeleteNodeTree, дерево двоичного поиска 470
- DeleteTable, хеш-таблица
  - реализация (упражнение) 426
  - реализация со связными цепочками 425
  - упражнение со связными цепочками 426
- DeleteTree, B-дерево 549
- Denenberg, Larry 523
- Deo, N. 522, 592
- DepthFirst, просмотр графа 578
- DepthToSort, упорядочение графа 581
- DigitToChar, модуль Utility 679
- Dijkstra, E. W. 55, 592, 642



- dispose, процедура языка Pascal 708
- Distance, граф 588
- DoBinary, программа оценки выражений 638
- DoCase, компиляция методом рекурсивного спуска 177
- DoCommand, программа оценки выражений 618  
калькулятор для полиномов 211  
текстовый редактор 256
- Dolf, компиляция методом рекурсивного спуска 165
- DoModule, компиляция методом рекурсивного спуска 162
- DoOther, компиляция методом рекурсивного спуска 177
- DoProgram, компиляция методом рекурсивного спуска 178
- DoStatement, компиляция методом рекурсивного спуска 176
- DoWhile, компиляция методом рекурсивного спуска 176
- DrawGraph, программа оценки выражений 641
- DriveNeighborCount, игра «Жизнь», драйвер 46
- EBCDIC коды 734
- ElapsedTime, модуль CPUTimer 681
- Elder, John 230
- eof, стандартная функция языка Pascal 43
- Error, процедура-утилита 81
- ErrorInform, программа оценки выражений 626
- Euler, L. 659
- EvaluatePostfix, оценка выражений 638  
нерекурсивный вариант 603  
рекурсивный вариант 609
- EvaluatePrefix, оценка выражений 601
- Even, Shimon 592
- Exponent, программа оценки выражений 639
- ExtractNumber, программа оценки выражений 634
- ExtractSymbol, программа оценки выражений 637
- ExtractWord, программа оценки выражений 633
- Feller, W. 393
- Fibonacci, Leonardo 655
- FIFO, дисциплина обслуживания (см. также Очередь) 182
- FileIO, модуль 682
- FillHashTable, программа оценки выражений 625
- FindRoot, дерево двоичного поиска 481
- FindSize, игра «Жизнь» (проект) 88
- FindString, текстовый редактор 258
- Fly, моделирование работы аэропорта 201
- Fortran, язык программирования 132  
индексация таблицы 395  
недопустимость рекурсий 132  
связный список 261  
синтаксический анализ 148  
этимология 593
- Fredkin, Edward 568
- Fryer, K. D. 661
- Gardner, Martin 26, 56, 662
- Gauss, C. F. 145
- GetCell, игра «Жизнь» 440
- GetCommand, текстовый редактор 255
- GetNextChar, компиляция методом рекурсивного спуска 172
- GetNode, двоичное дерево поиска 479
- GetToken, компиляция методом рекурсивного спуска 171
- GetToken, программа оценки выражений 626
- GetValue, программа оценки выражений 638
- GetWord, программа оценки выражений 633
- Gotlieb, C. C. 326, 444
- Gotlieb, L. R. 326, 444
- GraphExpression, программа оценки выражений 621
- GraphPoint, программа оценки выражений 640
- Gries, David 100, 144
- Hamblin, C. L. 642
- Harries, R. 444
- Hash, программа оценки выражений 625
- HeapSort, пирамидальная сортировка 379
- Hibbard, T. N. 326
- Hoare, C. A. R. 100, 351, 392  
упражнение 374
- Hofstadter, Douglas R. 661
- Horowitz, E. 181
- Huang, Bing-Chao 392
- Hueras, John F. 55
- Idle, моделирование работы аэропорта 201
- InfixtoPostfix, оценка выражений 613, 637
- Initialize, игра «Жизнь» 88
- Initialize, программа оценки выражений 623
- InsertTree, В-дерево 545
- InsertTree, красно-черное дерево 563
- InitializeScreen, программа оценки выражений 640
- Inorder, двоичное дерево 454
- Insert, дерево двоичного поиска 481
- InsertAVL, AVL-дерево 489

- InsertFirst, список (упражнение) 233
- InsertHeap, пирамидальная сортировка 380
- InsertLast, список (упражнение) 233
- InsertLine, текстовый редактор 257
- InsertList, список
  - дважды связный, реализация 242
  - непрерывный, реализация 234
  - связный в массиве 268
  - связный, реализация 237
  - спецификация 232
- InsertOrder, двоичный поиск 295
- InsertTable, хеш-таблица 421
  - реализация (упражнение) 426
  - реализация со связными цепочками (упражнение) 426
- InsertTree, дерево двоичного поиска 466
  - нерекурсивный вариант (упражнение) 473
- InsertTrie, трай-дерево 535
- InterchageList, список (упражнение) 233
- Iyengar, S. S. 522
- Jensen, K. 55
- JoinList, список (упражнение) 233
- Kernighan, Brian 55
- Kill, игра «Жизнь»
  - проект 88, 441
  - упражнение 78
- Kind, программа оценки выражений 627
- Klamkin, M. S. 444
- Knuth, Donald E. 144, 230, 278, 326, 328, 392, 522, 568, 654, 661, 670
  - упражнение 373
- Kronsjo, Lydia I. 444
- Land, моделирование работы аэропорта 201
- Langston, Michael A. 392
- Leading, программа оценки выражений 631
- Ledgard, Henry F. 55
- Lee, Bernie (проект) 350
- Leiserson, Charles E. 568
- LengthOfString, цепочки символов
  - реализация (упражнение) 250
  - спецификация 247
- Lesuisse, R. 326
- Lewis, Harry R. 523
- lg (логарифм по основанию 2) 307, 647
- Life, игра 24
  - сигнальная метка 41
  - DriveNeighborCount, драйвер для программы Life 46
  - SubtractNeighbors (проект) 88
  - альтернативные подходы 59
  - верификация 75
  - второй вариант 63, 68
  - конфигурации 53
  - ограда 40
  - описание 24
  - первый вариант 27
  - правила 24
  - примеры 25
  - примеры (упражнение) 29
  - сравнение программ 89
  - тестирование программы 52
- Life1, игра
  - анализ 58
  - обзор 58
  - подсчет числа предложений 59
  - программа 27
  - процедуры
    - Initialize 43
    - NeighborCount 41
    - UserSaysYes 44
    - WriteMap 44
- Life2, игра
  - Kill (проект) 88
  - SubtractNeighbors (проект) 88
  - анализ 89
  - инвариант цикла 77
  - подсчет числа предложений 89
  - программа 68
  - реализация 85, 88
  - реализацияts 86
  - спецификация 73, 75, 77
- Life3, игра
  - программа 437
  - процедуры
    - AddNeighbors 439
    - GetCell 440
    - Vivify 438
  - функция GetCell 441
- LinkLeft, скошенное дерево 506
- ListEmpty, простой список 66
- listentry, объявление типа для поиска и сортировки 280
- ListFull, простой список 66
- listnode, объявление типа для связного списка 712
- ListSize, простой список 66
- ln (натуральный логарифм) 307, 648
- Lomuto, Nico 392
- LookAhead, дерево игры 159
- LowerCase, утилита 255
  - модуль Utility 679

- IsLowerCase, модуль Utility 679
- IsUpperCase, модуль Utility 678
- Lukasiewicz, Jan 642
- Mailhot, Paul 20
- MakeNewNode, красно-черное дерево 565
- maxint, значения, Pascal 733
- MaxKey, сортировка выбором 341
- McCreight, E. 568
- McKenzie, B. J. 444
- MedianList, список (упражнение) 233
- Mermin, N. David 661
- Miller, Jonathan K. 100
- ModifyLeft, красно-черное дерево 564
- Modula-2, язык программирования 703
- Motzkin, Dalia 393
- Move, Башни Ханоя 121, 135
  - реализация 123
  - спецификация 122
- MoveLeft, B-дерево 554
- MoveRight, B-дерево 554
- MoveTerm, калькулятор для полиномов 219
- Nagin, Paul A. 55
- NeighborCount, игра «Жизнь» 41
- nex, процедура языка Pascal 708
- Newman, D. J. 444
- SingleNode, связный список 266
- NewPlane, моделирование работы аэропорта 200
- NewRandom, генератор случайных чисел 665
- Nievergelt, J. 522, 592
- nil-указатель, Pascal 709
- OpenExtensionInfile, модуль FileIO 684
- OpenExtensionOutfile, модуль FileIO 684
- OpenInfile, модуль FileIO 683
- OpenOutfile, модуль FileIO 683
- Out, компиляция методом рекурсивного спуска 175
- Pascal
  - dispose, стандартная процедура 708
  - nex, стандартная процедура 708
  - nil-указатель 709
  - альтернативные символы для обозначения указателя 733
  - вариантные записи 696, 726
  - ввод 42
  - включаемые файлы 45
  - глобальные и локальные переменные 151
  - директива forward 702
  - директива включения файлов 674
  - записи 694
  - зарезервированные слова 731
  - значения maxint 733
  - идентификаторы 730–731
  - компиляция методом рекурсивного спуска 161
  - «ленивый» ввод 43
  - локальные и глобальные переменные 151
  - модуль (Turbo Pascal) 45, 65
  - неопределенный указатель 709
  - область действия объявления 151
  - объявление указателя 710
  - объявления 733, 735
  - ограничения, накладываемые на указатели 710
  - отсроченный ввод 43
  - параметр-процедура 67
  - параметры процедур 151
  - предобъявленные переменные 735
  - предобъявленные процедуры 736
  - предобъявленные типы 735
  - приведение типов 698
  - приоритеты операторов 595, 737
  - присваивание указателей 710
  - пробелы 731
  - процедурные параметры 70, 700
  - пунктуация 733
  - разбор предложений языка 595
  - рекомендации по использованию вариантов записей 697
  - рекурсия 702
  - символьные цепочки 246
  - синтаксис
    - блока 722
    - выражения 729
    - действительного числа 725
    - заголовка функции 723
    - задания границ 724
    - записи 726
    - значения множества 730
    - идентификатора 724, 730
    - ключевого слова
      - program 722
      - type 725
    - константы 722, 724
      - без знака 725
    - метки 724
    - множества 726
    - объявления
      - массива 726
      - метки 722
      - переменной 723

- процедуры 723
- определения типа 723
- параметра 723, 729
- перечислимого типа 725
- поддиапазона 726
- предложения 727
  - case 29, 728
  - for 728
  - goto 728
  - if 727
  - repeat 728
  - while 728
  - with 695, 728
- процедуры 727
- присваивания 727
- простого выражения 729
- процедуры 723
- секция объявлений 722
- указателя 726
- файла 726
- фактора 729
- функции 723, 730
- целого числа 725
- члена 729
- синтаксические диаграммы 721, 723, 725, 727, 729
- синтаксический анализ предложений 599
- согласованный массив 724
- составное предложение 726
- список полей 726
- стандарт ISO 722
- стандартные функции 736
- стрелка вверх 707, 710
- типы
  - данных 96
  - связей 707
  - ссылок 707
  - указателей 707
- файловое окно 710
- файлы 42
- формат программы 732
- функция возведения в степень 639
- характерные черты 23
- цепочки символов 246, 724
- PascalCompiler, компиляция методом рекурсивного спуска 162
- Perry, William E. 100
- Plauger, P. J. 55
- PoissonRandom, генератор случайных чисел 667
- Pop, стек
  - реализация 108
  - связная реализация (упражнение) 114
  - спецификация 105
- PopNode, стек, связная реализация (упражнение) 114
- PositionString, цепочки символов
  - реализация 250
  - спецификация 248
- Postorder, двоичное дерево 455
- Power2, двоичное дерево 481
- Preorder, двоичное дерево 454
- prettyprinter 33
- Priority, программа оценки выражений 627
- PtrToInt, приведение типа 699
- Push, стек
  - реализация 107, 109
  - спецификация 105
- PushDown, B-дерево 546
- PushIn, B-дерево 547
- PushNode, стек, связная реализация 111
- PutToken, программа оценки выражений 627
- Pólya, George 100
- Queen, задача о восьми ферзях 150
- QueueEmpty, очередь
  - реализация 192
  - спецификация 183
- QueueFront, очередь
  - реализация (упражнение) 192
  - спецификация 184
- QueueFull, очередь
  - реализация 192
  - спецификация 183
- QueuePosition, поразрядная сортировка 410
- QueueSize, очередь
  - реализация 192
  - спецификация 184
- QuickSort, быстрая сортировка 365
- RadixSort, поразрядная сортировка 410
- Random, генератор случайных чисел 666
- RandomInt, генератор случайных чисел 666
- ReadCommand, калькулятор для полиномов 212
- ReadExpression, программа оценки выражений 628
- ReadFile, текстовый редактор 257
- ReadMap, игра «Жизнь»
  - проект 88
  - упражнение 78
- ReadNewParameters, программа оценки выражений 637
- ReadPolynomial, калькулятор для полиномов 217

- ReadString, цепочки символов
  - реализация (упражнение) 250
  - спецификация 247
- RecBinary1, двоичный поиск 298
- RecBinary1Search, двоичный поиск 298
- RecBinary2, двоичный поиск 300
- RecBinary2Search, двоичный поиск 300
- RecDeleteTree, B-дерево 551
- RecDepthSort, упорядочение графа 581
- RecEvaluate, рекурсивная постфиксная оценка 609
- RecInsertTree, красно-черное дерево 564
- RecQuickSort, быстрая сортировка 366
- Refuse, моделирование работы аэропорта 201
- Reingold, Edward M. 522, 592, 642
- Remove, B-дерево 552
- ReplaceList, список
  - непрерывная реализация (упражнение) 244
  - связная реализация (упражнение) 244
  - спецификация 232
- Restore, B-дерево 553
- RestoreScreen, программа оценки выражений 640
- Rethread, порярядная сортировка 411
- RetrieveList, список
  - непрерывная реализация (упражнение) 244
  - связная реализация (упражнение) 244
  - спецификация 232
- RetrieveTable, хеш-таблица, реализация 421
  - со связными цепочками (упражнение) 426
  - упражнение 426
- ReverseList, список (упражнение) 233
- ReverseRead, пример стека 102
- ReverseTraverseList, список (упражнение) 233
- Richter, Charles R. 646
- RightBalance, AVL-дерево 492
- Rivest, Ronald L. 568
- Roberts, Eric S. 144
- RotateLeft, AVL-дерево 491
- RotateRight, скошенное дерево 507
- S&SDemo, демонстрационная программа 685
- Sahni, S. 181
- SearchNode, B-дерево 544
- SearchTree, B-дерево 543
- Sedgewick, Robert 278, 393
- Segner, J. A. v. 659, 662
- SelectionSort, сортировка выбором 340
- SequentialSearch, процедура последовательного поиска 281
- Serve, очередь
  - реализация 192
  - спецификация 183
- ServeNode, связная очередь 207
- Sethi, Ravi 181
- SetPosition, список
  - дважды связный 241
  - связный 236, 239
- SetTimer, модуль CPUTimer 681
- SetupLexicon, программа оценки выражений 623
- Shell, D. L. 344, 392
- SkipExpression, компиляция методом рекурсивного спуска 174, 175
- Skippast, компиляция методом рекурсивного спуска 173
- Sleator, D. D. 522
- SNOBOL, язык программирования 405
- Sommerville, Ian 100
- Split, B-дерево 547
- SplitList, список (упражнение) 233
- StackEmpty, стек
  - реализация 108
  - связная реализация (упражнение) 114
  - спецификация 105
- StackFull, стек
  - реализация 108
  - связная реализация (упражнение) 114
  - спецификация 105
- StackSize, стек
  - реализация (упражнение) 113
  - спецификация 106
- StackTop, стек
  - реализация (упражнение) 114
  - связная реализация (упражнение) 114
  - спецификация 106
- Start, моделирование работы аэропорта 199
- Stevens, Peter 661
- Stirling, James 653, 661
- StopsStatement, компиляция методом рекурсивного спуска 174
- Stubbs, Daniel F. 230
- Substring, цепочки символов
  - реализация (упражнение) 251
  - спецификация 248
- SubtractNeighbors, игра «Жизнь»
  - проект 88, 441
  - упражнение 78
- Successor, B-дерево 552
- Swap, сортировка выбором 341
- Szymanski, T. 144
- Tanner, R. Michael 392
- Tarjan, Robert Endre 522, 592
- Topor, R. W. 278

- TotalTime, модуль CPUTimer 682
- Traverse, просмотр графа 578
- TraverseList, простой список  
реализация 80  
спецификация 67
- TraverseList, связный список 718  
в массиве 267
- TraverseQueue, очередь  
реализация (упражнение) 192  
спецификация 184
- TraverseStack, стек  
реализация (упражнение) 114  
связная реализация (упражнение) 114  
спецификация 106
- TreeSearch, двоичное дерево поиска  
нерекурсивный вариант 462  
рекурсивный вариант 461
- TreeSplay, скошенное дерево 507
- TrieSearch, трай-дерево 535
- Tucker, Alan 661
- Turbo Pascal  
модуль 45, 671, 703  
процедурный параметр 701  
синтаксис модуля 672
- TypeCast 698
- Ullman, Jeffrey D. 181
- unit, Turbo Pascal 45
- UpperCase, модуль Utility 679
- UserSaysYes, интерактивный ввод с терминала  
29, 257, 292, 438, 678  
функция-утилита 45
- uses, Turbo Pascal 672
- Utility, модуль Turbo Pascal 678
- ValidInfix, программа оценки выражений 632
- Van Tassel, Dennie 55
- Visit, процедура посещения  
просмотр двоичного дерева 454  
формальный процедурный параметр 67, 70, 700
- Vivify, игра «Жизнь» 85, 438
- Webre, Neil W. 230
- Welsh, Jim 230
- Wickelgren, Wayne A. 100
- Williams, J. W. J. 393
- Wirth, Niklaus 55, 180, 522, 568, 589  
упражнение 376
- Wood, Derick 144, 523, 568
- WriteFile, текстовый редактор 257
- WriteMap, игра «Жизнь» 44  
упражнение 78
- WritePolynomial, калькулятор для полиномов 216
- WriteString, цепочки символов  
реализация (упражнение) 251  
спецификация 247
- WriteTerm, калькулятор для полиномов 216
- Yeatman, Corey P. 686
- Yourdon, Edward 99
- Абстрактный тип данных 222, 226, 404  
двоичное дерево поиска 459  
двоичные деревья 445, 447, 449, 451, 453, 455  
детализация 227  
определение 226  
очередь 226  
простой список 225  
стек 226  
таблица 404  
упорядоченный список 459
- Абстракция данных, уровни 227
- Адам 29
- Адельсон-Вельский, Г. М. 486, 522
- Алгоритм  
деления (упражнение) 140  
детализация 34  
кодирование 39  
определение 148  
разработка 22  
рекурсиям 146  
с отходом 145
- Альтернативные символы для обозначения  
указателя, Pascal 733
- Альтернативы, игра «Жизнь» 59
- Альфа-бета отсечение 161
- Амортизационный анализ 502, 509  
двоичные числа 512  
затрат 511  
кредитовая функция 511  
кредитовый остаток 511  
определение 509  
при просмотре двоичного дерева 510  
скашивания 514  
фактические и амортизированные затраты 512
- Анализ  
AVL-дерево 493, 498  
Binary1Search 307  
Binary2Search 308  
алгоритма с отходом 152  
алгоритмов 23

- амортизационный, см. [Амортизационный анализ](#) 502
- асимптотика 318–319, 321, 323
- быстрой сортировки 368, 467
- двоичного поиска 303
- двоичного дерева поиска 483
- древовидной сортировки 467
- задачи о восьми ферзях 152
- красно-черного дерева 560
- методов хеширования 428
- пирамидальной сортировки 382
- поиска, нижние границы 313
- поразрядной сортировки 411
- последовательного поиска 282
- программ
  - Hanoi 125
  - Life1 58
  - Life2 89
- размера задачи 318
- рекурсии 125, 139
- сравнения ключей 282
- статистический закон 283
- трай-дерева 537
- сортировки 328, 346
- Аналогии при разработке приложений 223
- Аппроксимация
  - Ньютона, кубический корень 38
  - Стирлинга, факториалы 348, 361, 383
- Арифметика, операции по модулю 188
- Асимптотика 318
  - критерии порядка функций 323
- Атомарный тип 224
- База, связный список 712
- Базовый тип 404
- Башни Ханоя
  - анализ 125
  - второй рекурсивный вариант 135
  - дерево рекурсии 125
  - описание 121
  - правила 121
  - программа Hanoi 122
  - процедура Move 122, 135
- Безопасное программирование 48
- Бесконечные суммы 645
- Бесскобочная форма 607
- Библиография 55, 99, 143, 180, 230, 278, 326, 392, 444, 522, 568, 592, 642, 661, 670
- Биномиальные коэффициенты 653
  - рекурсивное вычисление (упражнение) 141
  - треугольник Паскаля (упражнение) 141
- Блок, внешний файл 538
- Бомба замедленного действия 51
- Быстрая сортировка 351, 365
  - анализ 368, 467
  - дерева рекурсии 355
  - для непрерывных списков 365
  - по среднему значению 374
  - подсчет числа сравнений 372
  - пример 352
  - процедуры
    - Partition 368
    - QuickSort 365
    - RecQuickSort 366
  - сравнение
    - с древовидной 467
    - с опорным ключом 351
    - с пирамидальной 383
    - с сортировкой включением 370
    - с сортировкой выбором 370
  - связного списка (проект) 376
  - среднего случая 371
  - худшего случая 369
  - числа сравнений и обменов 369
- Вариант с забыванием, двоичный поиск 297
- Вариантная часть, запись 696
- Вариантные записи, Pascal 696
- Ввод с терминала
  - Pascal 43
  - функция UserSaysYes 44
- Ведущая позиция, инфиксное выражение 630
- Вектор (см. также [Таблица](#)) 63
- Вектор доступа (см. также [Таблица доступа](#)) 397
- Верификация
  - алгоритма 75, 78, 326
  - и математическая индукция 76
  - и разработка 75
  - начального случая 76
  - программы 23
- Верхняя треугольная матрица (упражнение) 403
- Вершина
  - графа 569
  - дерева 302
- Вершина-источник, граф 584
- Вес, граф 576
- Ветви дерева 302
- Включаемые файлы, Pascal 45, 674
  - и структуры данных 677
- Включение
  - в AVL-дерево 488



- в В-дерево 540
- в связный список 237
- в хеш-таблицу 421
- Внешний поиск 280, 538
- Внешняя вершина дерева 302
- Внешняя память, блок 538
- Внешняя сортировка 328, 357, 386
- Внутренний поиск 280
- Внутренняя вершина дерева 302
- Внутренняя сортировка 328
- Возведение в степень, Pascal 639
- «Восемь», игра 155
- Восходящий синтаксический анализ 163, 599
- Временные затраты, рекурсия 133
- Временные фрагменты для отладки 699
- Время доступа 538
- Всевышний 29
- Вставка в программу 47
- Входные параметры 35
- Вывод графика (упражнение) 38
- Вывод на экран двоичного дерева (упражнение) 456
- Выделение памяти под данные 703
  - в языке Pascal 707
- Выражение в скобках (упражнение) 616
- Высота
  - AVL-дерева 486, 498
  - дерева 129, 302, 314, 348, 471, 539, 545
  - дерева Фибоначчи 499
  - красно-черного дерева 560
- Выталикивание из стека 102
  - непрерывная реализация 107
  - связная реализация 109
- Выходные параметры 35
- Гардероб, моделирование 287
  - AVL-дерево 500
  - В-дерево (проект) 557
  - дерево двоичного поиска (проект) 475
  - операции обслуживания 289
  - спецификации 288
  - хеширование (проект) 427
- Гармоническое число 485, 650
- Генеалогическое дерево 589
- Генератор случайных чисел 663
  - затравка 664
  - модуль 665
  - тест 668
  - функции
    - NewRandom 665
    - PoissonRandom 667
    - Random 666
    - RandomInt 666
- Генерирование перестановок 271, 273, 275
- «Гибридный» поиск (проект) 312
- Главная диагональ квадратной матрицы (упражнение) 402
- Главная программа
  - Airport 196
  - DataMake 687
  - Editor 254
  - GraphExpression 618
  - Hanoi 122
  - Life1 27
  - Life2 68
  - Life3 437
  - Parse 169
  - PascalCompiler 162
  - PermutationGeneration 275
  - PolynomialCalculator
    - реализация 212
    - эскиз 211
  - Queen 150
  - SimpleListDriver 81
- Глобальные и локальные переменные 151
- Глобальные переменные 36, 68, 264
- Голова
  - очереди 183
  - списка 713
- Головной указатель, связный список 712
- Головной элемент, очередь 183
- Граница для сравнения ключей, поиск 313
- Граничные условия, кольцевая очередь 188
- Граф 569
  - вершина 569
  - вершина-источник 584
  - вес 576
  - инцидентность 569
  - кратчайший маршрут 584
  - критерий экономного продвижения 585
  - маршрут 570
  - множественность ребер 572
  - неориентированный 569, 571
  - непрерывная реализация 575
  - обход 576
  - определение 569, 572
  - ориентированный 569
  - представление в виде множества 572
  - приложения 569
  - просмотр 576
    - в глубину 577
    - в ширину 577
  - процедуры
    - BreadthFirst 578
    - BreadthTopSort 582



- DepthFirst 578
- DepthTopSort 581
- Distance 588
- RecDepthSort 581
- Traverse 578
- реализация 573
- ребро 569
- регулярный (упражнение) 590
- свободное дерево 571
- связная реализация 574
- связный 570
- сильносвязный 571
- слабосвязный 571
- смежность 570
- смешанная реализация 576
- список смежности 574
- степень (упражнение) 590
- таблица расстояний 585
- таблица смежности 573
- цепь 570
- цикл 570
- Графики логарифмические 650
- Групповое обсуждение 47
- Групповой проект, калькулятор для полиномов 220
- Данные для тестирования программы 48, 686
- дБ (децибел, сокращение) 645
- Дважды связный список 240, 245
  - InsertList 242
  - SetPosition 239, 241
- Двоичное дерево 446
  - абстрактный тип данных 446
  - амортизационный анализ 511
  - вывод на экран (упражнение) 456
  - двойной просмотр (упражнение) 456
  - дерево выражения 450
  - длина пути 484
  - обратный просмотр 448
  - повороты 530
  - порядок посещения узлов (упражнение) 456
  - преобразование в 2-дерево 483
  - преобразование в список (упражнение) 457
  - примеры 446
  - просмотр 448
    - в глубину 448
    - в ширину 448
    - прямой 448
  - процедуры
    - CreateTree 453
    - Inorder 454
    - Postorder 455
    - Preorder 454
    - расширение до 2-дерева 483
    - реверсирование (упражнение) 456
    - симметричный 448
    - скобочная форма (упражнение) 456
    - соответствие саду 529
    - уровень за уровнем (упражнение) 456
    - уровень и индекс 478
    - функция Power2 481
- Двоичное дерево поиска
  - абстрактный тип данных 459
  - анализ 467, 483
  - древовидная сортировка 467
  - извлечение информации 475
  - подсчет сравнений 484
  - построение 477
  - процедуры
    - BuildTree 480
    - ConnectSubtrees 482
    - DeleteKeyTree 472
    - DeleteNodeTree 470
    - FindRoot 481
    - GetNode 479
    - InsertTree 466
    - Insert 481
    - TreeSearch, нерекурсивный вариант 462
    - TreeSearch, рекурсивный вариант 461
  - самонастраивающееся (см. также Скошенное дерево) 501
  - сбалансированность 483
  - сигнальная метка 474
  - скошенное (см. также Скошенное дерево) 501
  - сравнение с трай-деревом 537
  - удаление 469
  - Фибоначчи 498
- Двоичные деревья, перечисление 657
- Двоичные числа, амортизационный анализ 512
- Двоичный поиск 295, 457
  - вариант с забыванием 297
  - выбор вариантов 309
  - дерево сравнений 303
  - инвариант цикла 297
  - оптимальность 316
  - процедуры
    - Binary1Search 299
    - Binary2search 301
    - InsertOrder 295
    - RecBinary1 298
    - RecBinary1Search 298

- RecBinary2 300
- RecBinary2search 300
- распознавание равенства ключей 299
- сравнение вариантов 301
- сравнение с трай-поиском 537
- типичные ошибки 296
- Двойной просмотр, двоичное дерево (упражнение) 456
- Двуместный оператор 450, 594
- Двусторонняя очередь 193
- Дек
  - непрерывный (упражнение) 193
  - список с кольцевой связью 209
- Деление по модулю, хеш-функция 415
- Демонстрационная программа, управляемая меню 81, 85, 194, 245, 252, 294, 313, 474, 500, 519, 538, 557, 566, 616
- Дерево
  - 2-дерево (см. 2-дерево) 306
  - AVL-дерево (см. AVL-дерево) 486
  - В\*-дерево (упражнение) 556
  - В-дерево (см. В-дерево) 538
  - вершина 302
  - ветвь 302
  - внешнего и 305
  - внешняя вершина 302
  - внутреннего 305
  - внутренняя вершина 302
  - вызовов подпрограмм 117
  - выражения 595, 611
  - высота 302
  - двоичное (см. Двоичное дерево) 446
  - дочерние вершины 303
  - квадратичная формула 451
  - корень 302
  - корневое 525, 528
  - красно-черное (см. Красно-черное дерево) 557–565
  - лексикографическое 533
  - лес деревьев 528
  - линия в дереве 302
  - лист 302
  - многовариантное (см. также В-дерево) 533, 539
  - определение 524
  - определение как графа 571
  - просмотр с использованием стека 118, 133
  - расширенное двоичное (см. также 2-дерево) 306
  - реализация 525
  - рекурсии 118, 125
  - родительские вершины 303
  - сад деревьев 528
  - свободное 524
  - степень вершины 302
  - строго двоичное (см. также 2-дерево) 306
  - трай (см. также Трай-дерево) 533
  - узел 302
  - упорядоченное 525, 529
  - уровень вершины 302
  - число вершин на уровне 306
- Дерево выражения 450
- Дерево игры 154
  - игра «Восемь» 155
  - игра «Ним» 160
  - процедура LookAhead 159
- Дерево поиска 302
- Дерево рекурсии
  - анализ 125
  - Башни Ханоя 126
  - быстрая сортировка 355
  - генерирование перестановок 271
  - задача о восьми ферзях 153
  - определение 118
  - факториалы 137
  - числа Фибоначчи 138
- Дерево решений 302
- Дерево сравнений
  - двоичный поиск 303
  - длина внешнего пути 314, 348
  - последовательный поиск 302
  - упорядочение 347
- Десятичные логарифмы 647
- Детализация
  - алгоритмов 34
  - нисходящая 34
  - спецификации данных 227
  - типов данных 227
- Диагональная матрица (упражнение) 402
- Диграф 569
- Динамическая переменная 707
- Динамическая структура данных 64
- Динамическое выделение памяти 706
- Pascal 707
- Директива forward, Pascal 702
- Директива включения файлов, Pascal 674
- Диск, время доступа 538
- Дисперсия последовательности чисел (упражнение) 38
- Длина пути
  - в двоичном дереве 484
  - внешнего в 2-дерево 314
  - внешнего в дереве 305, 314
  - внешнего в дереве сравнений 348

- внутреннего в дереве 305
- теорема 308
- Длина списка, n элементов 282
- Доказательство
  - критерия экономного продвижения 585
  - посредством математической индукции 76
  - соответствия садов и двоичного дерева 529
  - теоремы о длине пути 308
- Документация
  - рекомендации 32
  - стиль 32
- Дочерние вершины дерева 303
- Драйвер 46
  - генератор случайных чисел (проект) 668
  - программа SimpleListDriver 81
  - простого списка 81
- Древовидная сортировка 467
  - анализ 468
  - преимущества и недостатки 468
  - сравнение с быстрой сортировкой 467
- Дюрер, Альбрехт (проект) 96
- Естественная сортировка слиянием 364
- Живучесть 618
- Жизненный цикл 94
- Журнальная статья 712
- Заглушка 40
  - вместо случайных чисел (проект) 205
  - калькулятор для полиномов 213
- Задача о восьми ферзях 146
  - анализ 152
  - дерево рекурсии 153
  - программа Queen 150
  - процедура AddQueen 146, 151
  - структуры данных 148
- Задача про дни рождения (хеширование) 428
- Задача, спецификация 22
- Запись
  - Pascal 694
  - вариантная часть 696
  - геометрический пример 697
  - иерархическая 695
  - используемая память 697
  - компонент 694
  - наложение полей вариантной записи 697
  - поле 694
  - поле тега 696
  - правила и рекомендации 697
  - приведение типа 698
  - фиксированная часть 696
- Запятые, Pascal 733
- Затравка, генератор случайных чисел 664
- Затраты, амортизационный анализ 511
- Защитное программирование 48
- Зарезервированные слова, Pascal 731
- Значение, определение термина 224
- Золотое сечение 656
- Зондирование, хеш-таблица 417
- Игра
  - Башни Ханоя 121
  - «Восемь» 155
  - «Жизнь», одномерная 97
  - «Жизнь», см. Life, игра 24
  - «Крестики и нолики» 160
  - лабиринт (проект) 154
  - «Ним» (упражнение) 160
  - ферзи (см. также Задача о восьми ферзях) 145
- Игра «Жизнь», третий вариант 435
  - процедуры
    - AddNeighbors 439
    - GetCell 440
  - разреженная таблица 435
  - структуры данных 435
  - хеш-таблица 440
  - хеш-функция Hash 441
- Идентификатор, синтаксис, Pascal 730
- Идентификаторы, Pascal 730
- Иерархическая запись, Pascal 695
- Извлечение
  - генератор тестовых файлов DataMake 686
  - данных (см. также Поиск) 279, 394
  - двоичное дерево поиска 474
  - из AVL-дерева 500
  - из очереди 184
  - информации 279, 394
- Изменение основания, логарифмы 649
- Измерение громкости звука 645
- Имена
  - рекомендации к выбору 31
  - типов и операций со списком 328
- Инвариант цикла 77, 367
  - двоичный поиск 297
- Инвертированная таблица 401
- Индексная функция 396, 413
  - треугольная матрица 399
- Индексное множество 404
- Индексы
  - в дважды связанных списках 268
  - в двумерном массиве 150
  - в кольцевом массиве 188

- в массивах, сравнение 189
- в очереди 186, 190
- в связном списке 263
- в списке 232
- в таблице 394
- Индуктивная база 644
- Инициализация 43, 77, 87, 199, 619, 623
- Инкрементная функция, хеш-таблица 418
- Интеграл 654
- Интегрирование, анализ хеш-таблицы 430
- Интенсивность землетрясений 646
- Интерактивная графическая программа 617
- Интерактивный ввод
  - ограничения языка Pascal 42
  - функция UserSaysYes 44
- Интерполяционная сортировка (проект) 349
- Интерполяционный поиск 317
- Интерфейс, Turbo Pascal 672
- Инфиксная форма 16, 452, 598, 611
  - определение 598
  - преобразование в постфиксную 611
  - процедура InfixtoPostfix 613
- Инцидентность, граф 569
- Использование памяти 16, 35
  - записи 697
  - и время выполнения 90
  - список 79
  - стек 109
  - хеш-таблица 423
- Календарь (проект) 97
- Калькулятор для полиномов 210
  - групповой проект 220
  - программа PolynomialCalculator 212
  - процедуры
    - Add 218
    - AppendTerm 216
    - DoCommand 211
    - MoveTerm 219
    - ReadCommand 212
    - ReadPolynomial 217
    - WritePolynomial 216
    - WriteTerm 216
  - связные очереди 214
  - сложение полиномов 218
  - структуры данных 214
  - тип polynomial 213
  - эскиз 211
- Калькулятор с обратной польской нотацией 210
- Камень-ножницы-бумага, игра (проект) 668
- Кампанологии 278
- Карте (обозначение указателя на языке Pascal) 707
- Каталана числа 656
- Кафе, пример стека 101
- Квадратичная зависимость времени выполнения 321
- Квадратичная формула 594
  - дерево выражения 451
  - постфиксная форма 598
  - преобразование в постфиксную форму 615
  - префиксная форма 598
- Квадратичное зондирование, хеш-таблица 418
- Квадраты целых чисел, сумма 644
- Кластеризация при хешировании 423
- Кластеризация, хеш-таблица 417
- Ключ
  - в дереве 458
  - в таблице 394
  - записи 279
  - мишень 286
  - поле элемента списка 328
  - при поиске 279
  - упорядочение 329
- Книга Бытия 29
- Кодирование 39
- Коды ASCII 734
- Коды EBCDIC 734
- Колокольный звон (проект) 276
- Кольцевая реализация очереди 187
- Команды, текстовый редактор 253
- Комбинаторика 661
- Компиляция методом рекурсивного спуска
  - объявления типов 163
  - программы
    - Parse 169
    - PascalCompiler 162
  - процедуры
    - DoCase 177
    - Dof 165
    - DoModule 162
    - DoOther 177
    - DoProgram 178
    - DoStatement 176
    - DoWhile 176
    - GetNextChar 172
    - GetToken 171
    - Out 175
    - SkipDeclarations 175
    - SkipExpression 174
    - Skippast 173
  - функция StopsStatement 174

Компиляция, рекурсивный спуск 161  
 Компонент записи, Pascal 694  
 Компромисс между памятью и временем 362, 386  
 Конечная вершина дерева 302  
 Конкретный пример  
   моделирование работы аэропорта 195  
   полиномиальный калькулятор 211  
   польская нотация 593  
   текстовый редактор 253, 255, 257, 259  
 Конфликт, хеш-таблица 414  
 Корень дерева 302  
 Корневое дерево 525  
   определение 528  
 Корректная последовательность скобок 657  
 Косвенная рекурсия 702  
 Красно-черное дерево 557, 559, 561, 563, 565  
   анализ 559  
   включение 560  
   красное условие 559  
   определение 559  
   процедуры  
     InsertTree 563  
     MakeNewNode 565  
     ModifyLeft 564  
     ReclInsertTree 564  
   связь с AVL-деревом 566  
   черное условие 559  
 Кратчайший маршрут, граф 584  
 Кредитовая функция 511  
   выбор 512  
   пример 513  
 Кредитовый инвариант 514  
   анализ скошенного дерева 514  
 Кредитовый остаток 511  
   выбор 512  
   вычисление (упражнение) 518  
 «Крестики и нолики», игра (упражнение) 160  
 Критерии  
   при разработке программ 93  
   эффективности сортировки 386  
 Критерий экономного продвижения, граф 585  
   доказательство 585  
 Кролики 655  
 Кубическая зависимость времени выполнения 321  
 Кубический корень, аппроксимация Ньютона 38  
 Лабиринт, решение с помощью алгоритма отхода (проект) 154  
 Ландис, Е. М. 486, 522

Левая рекурсия 609  
 Лексема  
   определение 162  
   поиск лексемы 170  
   программа оценки выражений 623  
   соглашения языка Pascal 600  
 Лексикографический порядок (упражнение) 251  
 Лексикографическое дерево 533  
 Лемма  
   10.5 (фактические и амортизированные затраты) 512  
   10.6 (сумма логарифмов) 515  
   10.7, 10.8, 10.9 (затраты на шаг скашивания) 516  
   7.1, 7.2 (число вершин 2-дерева) 306  
   7.5 (минимальная длина внешнего пути) 314  
   A.8 (сады и последовательности скобок) 657  
   A.9 (последовательность скобок) 658  
 «Ленивый» ввод 43  
 Лес 528  
 Линейная зависимость времени выполнения 321  
 Линейная реализация, очередь 186  
 Линейное зондирование, хеш-таблица 417  
 Линии в дереве 302  
 Лист дерева 302  
 Литература для дальнейшего изучения 55, 99, 143, 180, 230, 278, 326, 392, 444, 522, 568, 592, 642, 661, 670  
 Логарифмическая зависимость времени 321  
 Логарифмы 645, 647, 649, 651  
   десятичные 647  
   изменение основания 649  
   натуральные 648  
   обозначения 307, 649  
   обычные 647  
   определение 646  
   основание 646  
 Логарифмы по основанию 2 307  
 Локальные и глобальные переменные 151  
 Локальные переменные 35  
 Лукасевич, Ян 597  
 Магический квадрат (проект) 96  
 Маршрут, граф 571  
 Массив  
   определение 407  
   отличие от таблицы 407  
   прямоугольный 27, 395  
   реализация связного списка 261

- Массив (см. также [Таблица](#)) [63](#), [395](#)  
Математическая индукция [76](#), [308](#), [604](#), [644](#)  
Математическое ожидание [198](#), [667](#)  
Матрица  
    верхняя треугольная (упражнение) [403](#)  
    диагональная (упражнение) [402](#)  
    определение [398](#)  
    транспонированная (упражнение) [402](#)  
    тредиагональная (упражнение) [402](#)  
Матрица (см. также [Таблица](#)) [398](#)  
Мгновенный снимок программы [47](#)  
Медиана, поиск (упражнение) [302](#)  
Медицинские карты пациентов [501](#)  
Меланхолия, гравюра Дюрера [96](#)  
Метка конца файла, Pascal [43](#)  
Метод  
    «разделяй и властвуй» [137](#)  
    минимакса [156](#)  
    разбиения [122](#), [350](#), [406](#)  
    рекурсивного спуска [599](#)  
    стеклянного ящика, тестирование программы [50](#)  
    тикающего ящика, тестирование программы [51](#)  
    черного ящика, тестирование программы [49](#)  
Минимакса метод [156](#)  
Минимаксная оценка дерева игры [157](#)  
Минимально совершенная хеш-функция (упражнение) [426](#)  
Мишень, поиск [281](#)  
Многовариантное дерево поиска (см. также [В-дерево](#)) [539](#)  
Многовариантный переход [533](#)  
Многозадачность [706](#)  
Множество  
    значений функции [404](#)  
    индексов [404](#)  
    как абстрактный тип данных [224](#)  
    реализация множеств [573](#)  
Моделирование игры Хамураби (проект) [668](#)  
Моделирование работы аэропорта [195](#)  
    во времени [196](#)  
    главная программа [198](#)  
    инициализация [198](#)  
    объявления [198](#)  
    правила [195](#)  
    процедуры  
        Conclude [202](#)  
        Fly [201](#)  
        Idle [201](#)  
        Land [201](#)  
        NewPlane [200](#)  
        Refuse [201](#)  
        Start [199](#)  
        спецификации [195](#)  
        статистики моделирования [198](#)  
Моделирование, аэропорт (см. также [Airport](#)) [195](#)  
Модуль [671](#)  
    Utility [678](#)  
    CPUtimer [680](#)  
    FileIO [682](#)  
    анализа процессорного времени [680](#)  
    в системе Turbo Pascal [671](#), [703](#)  
    генератора случайных чисел [665](#)  
    для простых списков целых чисел [675](#)  
    закрывающая секция [671](#)  
    обслуживания файлов [682](#)  
    определение [671](#)  
    открытая секция [671](#)  
    родовой [675](#)  
    секция интерфейса [671](#)  
    секция реализации [671](#)  
    синтаксис [672](#)  
    случайных чисел [685](#)  
    структур данных [677](#)  
Модульное тестирование [51](#)  
Молекулярный вес, рекурсивные вычисления (проект) [427](#)  
  
Наибольший общий делитель (упражнение) [140](#)  
Наименьшее общее кратное, упражнение [52](#)  
Наложение полей вариантной записи [697](#)  
Направленный маршрут [571](#)  
Натуральные логарифмы [648](#)  
Неатакующие ферзи (см. также [Задача о восьми ферзях](#)) [146](#)  
Неопределенный указатель, Pascal [709](#)  
Неориентированный граф [569](#), [571](#)  
Непрерывная память, преимущества [244](#)  
Непрерывная реализация  
    графа [575](#)  
    списка [234](#)  
    стека [107](#)  
    таблицы [399](#)  
Непрерывные и связанные списки, сравнение [243](#)  
Непрерывный и последовательный, термины, сопоставление [225](#)  
Непрерывный список [705](#)  
Нижняя граница  
    сортировка сравнением [347](#), [349](#)  
    сравнение ключей при поиске [313](#)

- «Ним», игра (упражнение) 160
- тализация 34
- Нисходящее проектирование 695
- Нисходящий синтаксический анализ 163, 599
- О большое 319
- Область
  - видимости, переменные 35
  - действия объявления, Pascal 151
  - значений функции 404
  - определения функции 404
- Обозначение
  - «пола» 307
  - «потолка» 307
  - суммы 644
- Обработка ошибок 80
- Обратная польская нотация, см. Обратная польская форма 210
- Обратная польская форма (см. также Постфиксная форма) 598
- Обратный просмотр, двоичное дерево 448
- Обход графа 576
- Объявление
  - модуля 45
  - перечислимого типа 158
  - процедурного параметра 80
  - процедуры 67
  - связного стека (упражнение) 114
  - структуры узла 109
  - типа 107
  - узлов 235
  - указателей 235
- Объявления
  - в программе оценки выражений 619
  - область действия 151
  - переменных 30
  - программные 178
  - синтаксис языка Pascal 721
  - типов 64, 163
  - упреждающие 179
- Обычные логарифмы 647
- Ограда, игра «Жизнь» 40
- Одномерная игра «Жизнь» 97
- Одноместное отрицание, обозначение 598
- Одноместный оператор 450, 594
- Ожидаемое время
  - поиска 282
  - сортировки 328
- Окно в файл, Pascal 171, 710
- Операторы
  - двуместный 594
  - одноместный 594
  - приоритет при вычислениях 594
  - приоритеты, Pascal 737
  - языка Pascal 737
- Операции
  - абстрактные типы данных 225
  - гардероба 289
  - для очереди 183
  - по модулю
    - арифметические 188
    - хеш-функция 415
  - цепочек символов 247
- Опорный ключ
  - быстрая сортировка 351
  - выбор (упражнение) 374
- Определение логарифмов 646
- Оптимальность двоичного поиска 316
- Орграф 569
- Ориентированный граф 569
- Основание логарифма 646
- Открытая адресация, хеш-таблица 417
- Отладка 23, 39, 46
- Отношение, рекуррентное 369
- Отсечение, альфа-бета (проект) 161
- Отсроченный ввод 43
- Отход, при поиске в цепочке символов 249
- Оценка
  - выражений 599
  - методом минимакса 156
  - постфиксных выражений 602
  - префиксных выражений 599
- Очереди
  - сравнение реализаций 208
- Очередь 182, 230
  - абстрактный тип данных 226
  - абстракция данных 228
  - включение (см. Append) 184
  - голова и хвост 183
  - граничные условия 188
  - извлечение данных 184
  - линейная реализация 186
  - методы реализации, обзор 189
  - обзор реализаций 189
  - операции 183
  - определение 226
  - постановка в очередь (см. Append) 184
  - процедуры
    - AppendNode 206
    - Append 191
    - CreateQueue 191, 206
    - QueueEmpty 192
    - QueueFull 192
    - QueueSize 192



- ServeNode 207
  - Serve 192
  - реализация в виде кольцевого массива 186
  - с двусторонним доступом 193
  - с приоритетами 383
  - связная реализация 205
  - спецификации 183
  - удаление из очереди 184
  - уровни детализации 228
  - физическая реализация 185
- Пакет утилит 45
- обработка ошибок 81
- Палиндром (упражнение) 251
- Память для подпрограмм 131
- Память и время выполнения, соотношение 90
- Параллельные вычисления 130
- Параллельные процессы 130
- Параметр-процедура, Pascal 67
- Параметры 35
- var-параметры 35
  - в рекурсивной процедуре 151
  - входа-выхода 35
  - входные 35
  - выходные 35
  - и эффективность программы 66
  - процедурные 700
- Параметры-процедуры 700
- Парфенон 656
- Первым вышел, первым вышел, FIFO (см. также Очередь) 182
- Переменная
- глобальная 36, 68, 131, 151, 264
  - динамическая 707
  - локальная 31, 35, 116, 131, 151
  - псевдоним 719
  - статическая 707
- Переменная-псевдоним 719
- Переменные
- в рекурсивной процедуре 151
  - область существования 35
  - предобъявленные в языке Pascal 735
- Переполнение памяти 703
- Перестановки 652
- генерирование 271
  - дерево рекурсии 271
  - колокольный звон 276
  - процедура Permute 274
  - с помощью дека 194
  - с помощью стека (упражнение) 113
  - структуры данных для них 274
- Перестановочная железнодорожная ветка (упражнение) 113
- Перечисление
- двоичных деревьев 657
  - перестановок в стеке 658
  - садов 657
  - триангуляция многоугольников 659
- Перфокарты 42
- сортировка 407
- Пирамида
- определение 378
  - тройная (упражнение) 386
- Пирамидальная сортировка 377
- анализ 382
  - на языке Pascal 379
  - процедуры
    - BuildHeap 382
    - HeapSort 379
    - InsertHeap 380
- Побочные эффекты 36, 68, 131, 264
- упражнения 221
- Поворот
- двойной, AVL-дерево 491
  - левосторонний, AVL-дерево 490
  - правосторонний, AVL-дерево 491
- Повороты
- AVL-дерево 490
  - двоичное дерево и сад 530
  - скошенное дерево 503
- Поддержка программ 23, 57, 59, 61
- Подносы, стек в кафе 101
- Подпрограммы
- дерево вызовов 117
  - драйверы 46
  - заглушки 40
  - стек вызовов 116
  - тестирование 48
  - хранение данных 131
- Подсказки и ловушки
- анализ алгоритмов 324
  - графы 591
  - двоичные деревья 519
  - деревья 566
  - детализация данных 442
  - извлечение данных 442
  - пирамиды 390
  - поиск 324
  - программная инженерия 99
  - разработка программы 52–53
  - реализация списков 228–229, 276
  - рекурсия 142, 179
  - связные списки 718



- сортировка 390
- списки 276
- структуры данных 228–229, 276
- таблицы 442
- хеширование 442
- Подсчет числа предложений, Life1 59
- Поиск 279
  - S&SDemo, демонстрационная программа 685
  - асимптотический анализ 322
  - в цепочке символов 248
  - внешний 280, 538
  - внутренний 280
  - время неуспешного поиска 310
  - время успешного поиска 310
  - генератор тестовых файлов DataMake 686
  - «гибридный» (проект) 312
  - двоичный (см. также Двоичный поиск) 295
  - интерполяция 317
  - ключ 279
  - мишень 281
  - нижняя граница при сравнении ключей 313
  - обозначения для записей и ключей 280
  - ожидаемое время 282
  - последовательный 281
  - распределение ключей 317
  - сигнальная метка 286
  - спрятанного сокровища 705
  - сравнение с табличным просмотром 394
  - среднее время 283
  - тернарный (проект) 312
  - трай-дерево 533
  - Фибоначчи 313
  - цепочки символов 258
- Покупка акций (упражнение) 185
- «Пол» 307
- Поле
  - записи, Pascal 694
  - тега в записи 696
- Полиномиальный калькулятор (см. Калькулятор для полиномов) 210
- Полиномы
  - арифметические операции 214
  - определение 214
  - реализация 215
- Полуоткрытый интервал 666
- Польская нотация 593, 597
- Польская форма 452
  - определение 597
  - преобразование в постфиксную форму 611
  - программа оценки выражений 617
  - синтаксические диаграммы 607
- Поразрядная сортировка 407
  - анализ 411
  - процедуры
    - RadixSort 410
    - Rethread 411
  - реализация 408
  - функция QueuePosition 410
- Порядок
  - многовариантного дерева 539
  - функции 320
- Посещение
  - списка 67
  - узла дерева 448
- Последним вошел, первым вышел, LIFO 102
- Последовательность
  - конечная 225
  - определение 225
  - символов 246
- Последовательный и непрерывный, термины, сопоставление 225
- Последовательный поиск 281
  - анализ 282
  - дерево сравнений 303
  - упорядоченные ключи (упражнение) 286
- Последовательный список 225
- Постановка в очередь 184
- Постоянная Эйлера 652
- Постусловия 28
- Постфиксная оценка, проверка 604
- Постфиксная форма 452
  - откладывание операторов 611
  - преобразование в постфиксную форму 611
  - приоритеты операторов 611
  - программа оценки выражений 617
  - процедуры
    - EvaluatePostfix 603
    - InfixtoPostfix 613
    - RecEvaluate 609
  - рекурсивная оценка 607
  - синтаксическая диаграмма 607
- Постфиксное выражение, оценка 603
- «Потолок» 307
- Правила
  - игры «Жизнь» 24
  - выбора имен 31
  - использования указателей 710
  - моделирования работы аэропорта 195
  - удаления узла из AVL-дерева 494
- Правило треугольника, расстояние между городами 403
- Предложение
  - case, Pascal 728

- for, Pascal 728
- goto, Pascal 728
- if, Pascal 727
- repeat, Pascal 728
- while, Pascal 728
- with, Pascal 695, 728
- процедуры, Pascal 727
- синтаксический анализ 165
- Предобъявленные процедуры, Pascal 736
- Предусловия 28
- Преобразование ключей (см. Хеш-таблица) 415
- Префиксная форма 452
  - определение 597
  - оценка выражений 599
  - процедура EvaluatePrefix 601
  - синтаксическая диаграмма 607
- Приближение Стирлинга
  - точность 660
  - факториалы 660
- Приведение типов 698
- Привязка типа указателя 708
- Принцип голубятни 149
- Принципы
  - «разделяй и властвуй» 350
  - ввода-вывода 42
  - восходящего синтаксического анализа 599
  - нисходящего кодирования 79
  - нисходящего программирования 36
  - нисходящего проектирования 140, 442
  - нисходящего синтаксического анализа 163, 599
  - нисходящей детализации 34
  - нисходящей детализации алгоритмов 227
  - нисходящей разработки структур данных 104
  - нисходящей спецификации структур данных 227
  - разделения 350
  - разработки подпрограмм 34
- Приоритеты операторов 595, 611
- Присваивание указателей 710
- Приращение 188
- Пробелы, Pascal 731
- Прогнозирование в играх 154
- Программа Editor 254
- Программа оценки выражений 617
  - ведущая позиция 630
  - лексемы-специальные символы 636
  - объявления 619
  - проверка на ошибки 630
- процедуры
  - ConvertToReal 636
  - CreateHashTable 625
  - DoBinary 638
  - DoCommand 618
  - DrawGraph 641
  - ErrorInform 626
  - EvaluatePostfix 638
  - Exponent 639
  - ExtractNumber 634
  - ExtractSymbol 637
  - ExtractWord 633
  - FillHashTable 625
  - GetToken 626
  - GetValue 638
  - GetWord 633
  - GraphExpression 621
  - GraphPoint 640
  - Hash 625
  - InfixtoPostfix 637
  - InitializeScreen 640
  - Kind 627
  - Leading 631
  - Initialize 623
  - Priority 627
  - ReadExpression 628
  - ReadNewParameters 637
  - RestoreScreen 640
  - SetupLexicon 623
  - ValidInfix 632
- структуры данных 619
- хеш-таблица 624
- числовые лексемы 634
- Программа, разработка 22
- Программистский принцип
  - ввод и вывод 42
  - вторая попытка решения задачи 91
  - глобальные переменные 36, 68
  - детализация 34
  - документация 32
  - кодирование 79
  - латание старой программы 92
  - модульность 34
  - откладывание задачи 73
  - отладка и тестирование 48
  - побочные эффекты 36, 68
  - пред- и постусловия 28
  - простота 90
  - прототипы 93
  - сокрытие информации 190
  - соотношение памяти и времени выполнения 90

- спецификация задачи 92
- спецификация программы 79
- структур данных 228
- структурированная разработка 228
- тестовые данные 48
- торопливость 92
- чтение программы 33
- эффективность 59
- Программная инженерия 57
  - групповой проект 220
  - определение 94
- Программы
  - анализ 58
  - поддержка 23, 57, 59, 61
- Производящая функция 655
- Произвольный доступ 243
- Просматриваемые очереди 226
- Просматриваемые стеки 226
- Просмотр
  - амортизационный анализ 510
  - в глубину 577
  - в глубину дерева 448
  - в ширину 577
  - в ширину дерева 448
  - графа 576
  - двоичного дерева 448
  - процедурный параметр 700
  - роль параметра Visit 700
  - сада 532
  - списка 67
  - таблицы 407
  - уровень за уровнем (упражнение) 456
  - цепочки символов 248
- Простая хеш-функция 416
- Простой список 64
  - связный 715
  - целых чисел, модуль для их обработки 675
- Простые сомножители (проект) 116
- Проталкивание в стек 101
  - непрерывная реализация 107
  - связная реализация 109
- Прототип 93
- Прототипирование 93
- Процедура разделения, быстрая сортировка 374
- Процедурные параметры, Pascal 70, 700
- Прямая рекурсия 702
- Прямой просмотр
  - двоичного дерева 448
  - сада 532
- Псевдослучайные числа 664
  - генерация 664
  - затравка 664
- Псевдослучайные числа (см. также Генератор случайных чисел) 197
- Пуассона распределение 198
- Пунктуация, Pascal 733
- Пустая цепочка символов 246
- Пустой связный список 713
- Равномерное распределение 666
- Разбор программы, структурированный 47
- Развертывание массива
  - по строкам 395
- Развертывание массива
  - по столбцам 395
- Разделение времени 706
- Разделение списка, быстрая сортировка 366, 368
- Разработка
  - алгоритма 63, 75, 157, 296, 366
  - компилятора 148
  - критерии 93
  - подпрограммы 34
  - программы 22, 93
  - структур данных 22
- Разреженная таблица 413
  - игра «Жизнь» 435
- Разрешение столкновений
  - задача про дни рождения 428
  - связные цепочки 422
  - хеш-таблица 417
- Распознавание равенства ключей, двоичный поиск 299
- Распределение
  - ключей, поиск 317
  - математическое ожидание 667
  - Пуассона 198, 667
  - равномерное 666
- Рассеянная память (см. Хеш-таблица) 415
- Расширенное двоичное дерево (см. также 2-дерево) 306, 483
- Рваная таблица 400
- Реализация
  - полиномов 214
  - рекурсии 129
  - связных списков в массивах 262
  - секция модуля Turbo Pascal 672
  - списков 64, 234
  - стека 107
  - упорядоченного дерева 525
  - цепочек символов 248
- Ребро графа 569

- Реверсирование
  - входных данных 102
  - двоичного дерева (упражнение) 457
  - строки 102
- Регулярный граф (упражнение) 590
- Реентерабельные программы 132
- Рекомендации
  - выбор имен 31
  - дерево рекурсии 140
  - детализация 34
  - документация программы 32
  - записи 697
  - разработка программы 93
  - связные списки 718
- Рекуррентное отношение 369, 484
- Рекурсивный спуск 161, 163
- Рекурсия 116, 117, 119, 121, 123, 125, 127
  - Pascal 702
  - алгоритмы с отходом 146
  - анализ 139
  - временные затраты 133
  - итерация вместо рекурсии 136
  - косвенная 702
  - левая 609
  - локальные и глобальные переменные 151
  - неудачное использование 136
  - параллельные процессы 130
  - постфиксная оценка 607
  - принципы 128, 129, 131, 133, 135, 137, 139, 141
  - прямая 702
  - разработка алгоритмов 128
  - реализация 129
  - рекомендации к использованию 140
  - синтаксический анализ 163
  - стековая реализация 134
  - структуры данных 132
  - требования к памяти 131
  - хвостовая рекурсия 134, 467, 473, 543
- Решение задачи 33
- Решений дерево 302
- Рихтера шкала 646
- Робастность 618
- Родительская вершина дерева 303
- Родовые конструкции 64
- Родовые средства 441, 675
- С (язык программирования) 450
- Сад
  - определение 528
  - повороты 530
  - преобразование в двоичное дерево 530
  - просмотр 532
- Сады, перечисление 657
- Самонастраивающееся дерево поиска, см. Скошенное дерево 501
- Сбалансированность, дерево поиска 483
- Свертывание ключа, хеш-функция 415
- Свободное дерево 525
  - определение 571
- Связная и непрерывная память, сравнение 243
- Связная очередь 205, 207, 209
  - приложение для работы с полиномами 210
- Связная память 215, 243, 273, 469
  - недостатки 422
  - преимущества 205, 244
- Связные списки 703, 705, 707, 709, 711, 713, 715, 717, 719
  - рекомендации для программиста 718
  - упорядочение слиянием 357
- Связные стеки 109
- Связные цепочки, хеш-таблица 422
- Связный граф 570
- Связный список 705
  - Basic 261
  - Cobol 261
  - Fortran 261
  - база 712
  - включение элемента 237, 267
  - голова списка 713
  - головной указатель 712
  - дважды связный 240, 245
  - индекс 263
  - кольцевой 209
  - косвенные связи 436
  - множественные связи 264
  - недостатки 243
  - объявление типа listnode 712
  - переменная-псевдоним 719
  - преимущества 243
  - просмотр 267
  - процедура NewNode 266
  - пустой 713
  - реализация в массиве 261, 273
  - узел 711
  - якорь 712
- Связный список процедуры DisposeNode 267
- Связь
  - Pascal 708
  - определение 704
- Священник 188
- Секция объявлений, Pascal 722
- Сеть 576

- Сигнальная метка 41, 332, 505
  - поиск (упражнение) 286
  - скошенное дерево 505
- Сильносвязный граф 571
- Символ электрического заземления 704, 709
- Символьная цепочка, см. [Цепочка символов](#) 246
- Символьные коды, ASCII и EBCDIC 734
- Симметричный просмотр
  - двоичного дерева 448
  - сада 532
- Синтаксис
  - вариантной записи, Pascal 726
  - выражения, Pascal 729
  - действительного числа, Pascal 725
  - заголовка функции, Pascal 723
  - записи, Pascal 726
  - значения множества, Pascal 730
  - идентификатора, Pascal 724
  - инфиксного выражения 630
  - константы без знака, Pascal 725
  - константы, Pascal 722, 724
  - массива, Pascal 726
  - множества, Pascal 726
  - модуля 672
  - объявления
    - метки, Pascal 722
    - переменной, Pascal 723
    - процедуры, Pascal 723
  - определения типа, Pascal 723
  - параметра, Pascal 723, 729
  - перечислимого типа, Pascal 725
  - предложения метки, Pascal 724
  - предложения присваивания, Pascal 727
  - предложения, Pascal 727
  - простого выражения, Pascal 729
  - процедуры, Pascal 723
  - символьной цепочки, Pascal 724
  - составного предложения, Pascal 726
  - терма, Pascal 729
  - типа, Pascal 725
  - файла, Pascal 726
  - фактора, Pascal 729
  - функции, Pascal 723, 730
  - целого числа, Pascal 725
- Синтаксис блока, Pascal 722
- Синтаксис объявления программы, Pascal 722
- Синтаксические диаграммы
  - Pascal 721, 723, 725, 727, 729
  - польские формы 608
- Синтаксический анализ 148, 163
  - восходящий и нисходящий 599
- Синтаксический разбор, см. [Синтаксический анализ](#) 163
- Скобки, корректная последовательность 657
- Скошенное дерево 501
  - амортизационный анализ 514
  - кредитовый инвариант 514
  - повороты 503
  - процедуры
    - LinkLeft 506
    - RotateRight 507
    - TreeSplay 507
  - сигнальная метка 505
  - шаги «зиг» и «заг» 502
- Слабосвязный граф 571
- Следствие
  - 10.11 (полная стоимость скашивания) 518
  - 10.2 (средняя производительность древо-видной сортировки) 468
  - 10.4 (затраты на балансирование, дерево поиска) 485
  - 7.7 (оптимальность Binary1Search) 316
  - A.10 (перечисления и числа Каталана) 659
  - A.6 (логарифмическое приближение Стирлинга) 654
- Слияние
  - внешнее 386
  - для связанных списков 357
  - метод сортировки 357
- Словарь лексем 620
- Слово памяти 712
- Сложение полиномов 218
- Случайное блуждание 669
- Случайное зондирование, хеш-таблица 419
- Случайные числа 663
- Смежность, граф 570
- Снятие ссылки с указателя 710
- Совершенная хеш-функция 426
- Согласованные массивы, Pascal 724
- Создание мира 29, 121, 321
- Соккрытие информации 70, 104, 695
- Соотношение памяти и времени выполнения 90
- Сортировка 327–392
  - S&SDemo, демонстрационная программа 685
  - анализ 328
  - быстрая 351
  - внешняя 328, 357, 386
  - внутренняя 328
  - выбором, процедура SelectionSort 340
  - генератор тестовых файлов DataMake 686

- дистрибутивная, связанных списков (проект) 350
- древовидная 467
- интерполяционная (проект) 349
- критерии эффективности 386
- методом разбиения 350
- методом Шелла 344
- нижняя граница сравнений 347, 349
- объединением 351
- ожидаемое время 328
- перфокарт 407
- пирамидальная (см. **Пирамидальная сортировка**) 377
- по среднему значению 374
- подсчетом (упражнение) 343
- поразрядная 407
- проверка упорядоченности списка 336
- пузырьковая (проект) 338
- рекомендации к тестированию программы 337
- с разделением на подписки 351
- сканированием (проект) 338
- смешанная (проект) 376
- сравнение методов 386
- стабильность (упражнение) 390
- стандартное отклонение 388
- тестирование 388
- число сравнений ключей 328
- Сортировка включением 329
  - анализ 334
  - и выбором, сравнение 342
  - непрерывный список 331
  - связный список 332
- Сортировка выбором
  - анализ 342
  - и включением, сравнение 342
  - процедура Swap 341
  - функция MaxKey 341
- Сортировка методом Шелла 344
  - анализ 346
  - процедура ShellSort 345
- Сортировка слиянием 351
  - анализ 359
  - для непрерывных списков 362
  - для связанных списков 357
  - естественная 364
  - пример 352
  - процедуры
    - Divide 357
    - MergeSort 357
    - Merge 358
  - сравнение структур данных 362
  - упорядоченные связанные списки 358
- Сочетания 653
- Спецификации 78
  - гардероба 288
  - для стека 105
  - моделирование работы аэропорта 195
  - очереди 183
  - списков 231
  - текстовый редактор 253
- Спецификация
  - границ, Pascal 724
  - задачи 22, 92, 96
  - подпрограммы 34, 71
  - программы 28
  - требований 95
  - цепочки символов 247
- Список 63, 231
  - дважды связный 240, 245
  - длина  $n$  элементов 282
  - кольцевой связный 209
  - непрерывная реализация 64, 234
  - непрерывный 705
    - процедуры
      - ClearList (упражнение) 84
      - CreateList (упражнение) 84
      - InsertList 234, 242
      - ListEmpty (упражнение) 84
      - ListFull (упражнение) 84
      - ListSize (упражнение) 84
      - SetPosition 236, 239, 241
  - операции обслуживания 232
  - определение 225
  - первым вошел, первым вышел, FIFO (см. также **Очередь**) 182
  - полей, Pascal 726
  - посещение 67
  - последовательный 225
  - просмотр 67
  - простой 65
  - процедуры
    - AddList 79
    - TraverseList 80, 267
  - с кольцевой связью 209
  - с косвенными связями 437
  - связный 705
    - процедуры
      - InsertList 238, 268
  - связный простой 715
  - сигнальная метка 332
  - смежности, граф 574
  - спецификации 231
  - сравнение с таблицей 406

- упорядоченный 329
- упражнение метка 286
- Сравнение
  - вариантов двоичного поиска 301, 309
  - вариантов программы Life 89
  - двоичного поиска с трай-поиском 537
  - затрат на балансирование 483
  - итерации и рекурсии 139
  - методов извлечения информации 434
  - непрерывной и связной памяти 243
  - пирамидального и быстрого 383
  - префиксных и постфиксных выражений 602
  - реализации очередей 208
  - рекурсии и итерации 139
  - сортировка 386
  - списка и таблицы 406
  - таблицы и списка 406
  - успешного и неуспешного поиска 310
- Сравнение ключей
  - нижние границы для поиска 316
  - сортировка 328
- Сравнений дерево 302
- Среднее время
  - в AVL-деревьях 499
  - в двоичных деревьях 310
  - в деревьях Фибоначчи 499
  - ожидания взлета, моделирование аэропорта 202
  - ожидания посадки, моделирование аэро-порта 202
  - последовательного 283
- Среднее значение (упражнение) 38
- Ссылка
  - Pascal 708
  - определение 704
- Ссылочная таблица 397
- Стабильная процедура сортировки 390
- Стабильные методы сортировки (упражнение) 390
- Стандарт ISO 722
- Стандартное отклонение
  - в методах сортировки 388
  - определение 38
  - последовательности чисел (упражнение) 38
- Стандартные идентификаторы, Pascal 731
- Стандартные объявления, Pascal 733, 735
- Стандартные функции языка Pascal 736
- Статистика, анализ алгоритмов 283
- Статистики 388
- Статистический анализ 388
- Статическая переменная 707
- Статическая структура данных 64
- Статический анализатор 48
- Стек 101, 133
  - абстрактный тип данных 226
  - вершина 101
  - для перестановок вагонов (упражнение) 113
  - использование в дереве вызовов подпро-грамм 133
  - использование перед реализацией 104
  - использование при просмотре дерева 118
  - калькулятор для полиномов 214
  - непрерывная реализация 107
  - области памяти для подпрограмм 116
  - операции push и pop 102
  - определение 226
  - пара стеков (упражнение) 114
  - пара стеков, организация памяти (упражне-ние) 114
  - перечисление перестановок 658
  - постфиксная оценка 602
  - постфиксное преобразование 613
  - просматриваемый 106
  - процедуры
    - CreateStack 108
    - Pop 108
    - PushNode 111
    - Push 107, 109
    - ReverseRead (пример) 102
    - StackEmpty 108
    - StackFull 108
  - пружинная аналогия 101
  - реализация 107
  - рекурсия 134
  - спецификации 105
- Степень
  - вершины дерева 302
  - графа (упражнение) 590
  - роста функций 321
- Стиль программирования 22
- Столбец, в прямоугольном массиве 26
- Столкновение, хеш-таблица 414
- Стохастическое моделирование 197
- Страница, внешний файл 538
- Стрелка вверх, обозначение указателя на язы-ке Pascal 707, 710
- Строго двоичное дерево (см. также 2-дерево) 306
- Строка, в прямоугольном массиве 26
- Структурированный разбор программы 47
- Структурное программирование 34
- Структурный тип 224



## Структуры данных

- анализ 22
- генерирование перестановок 273
- графы 589
- задача о восьми ферзях 148
- игра «Жизнь» 435
- извлечение информации 394
- калькулятор для полиномов 214
- обобщающая таблица 677
- определение 224
- подсказки и ловушки 142, 228, 276, 324, 442, 519, 566
- программа оценки выражений 619
- рекурсия 132
- хеш-таблица 422

## Сумма

- квадратов целых чисел 643–644
- степеней целых чисел 643
- телескопическая 512
- целых чисел 335, 342, 370, 399, 419

## Суффиксная форма, определение 598

## Счет проб, хеширование 429

## Таблица

- Fortran 395
- абстрактный тип данных 405
- время извлечения информации 406
- доступа 396
- инвертированная 401
- индексация 394
- определение 405
- отличие от массива 407
- переполнений, хеширование (упражнение) 426
- просмотр 407
- просмотр в сравнении с поиском 394
- прямоугольный массив 397
- разреженная 413
- расстояний между городами 403
- расстояний, граф 585
- рваная 400
- реализация 395, 405
- смежности, граф 573
- сравнение со списком 406
- треугольная 398, 400

## Таблицы доступа

- множественного 401
- рваная 400

## Текстовый редактор 253

- главная программа 254
- команды 253
- поиск цепочки символов 258

## процедуры

- ChangeString 260
- DoCommand 256
- FindString 258
- GetCommand 255
- InsertLine 257
- ReadFile 257
- WriteFile 257

## спецификации 253

## Телескопическая сумма 512

## Теорема

- 10.1 (древовидная и быстрая сортировка) 468
- 10.3 (сравнение ключей, дерево поиска) 485
- 11.1 (сады и двоичные деревья) 529
- 11.2 (высота красно-черного дерева) 560
- 13.1 (синтаксис постфиксной формы) 604
- 13.2 (постфиксная оценка) 604
- 13.4 (бесскобочная форма) 607
- 3.1 (стеки и деревья) 118
- 7.3 (длина пути в 2-дереве) 308
- 7.4 (сравнение процедур поиска) 311
- 7.6 (нижняя граница, сравнение ключей при поиске) 316
- 8.1 (проверка упорядоченности списка) 335
- 8.2 (нижняя граница, упорядочение сравнением ключей) 348
- A.1 (сумма степеней целых чисел) 643
- A.2 (сумма степеней 2) 644
- A.3 (бесконечные суммы) 645
- A.4 (гармонические числа) 652
- A.5 (приближение Стирлинга) 653
- A.7 (перечисление двоичных деревьев) 657

## Тернарный поиск (проект) 312

## Тест, генератор случайных чисел 668

## Тестирование 23, 40, 46

- калькулятора для полиномов 213
  - методов сортировки 388
  - принципы 48
  - программ 48, 81
  - различных методов сортировки (проект) 337
  - стеклянного ящика 50
  - тикающего ящика 51
  - управление с помощью меню 81
  - черного ящика 49
- Тестовые данные 48, 686
- Тип
- атомарный 224
  - базовый 404
  - данных, определение 224



- диапазона, Pascal 726
- значение 404
- определение 224
- привязка 708
- структурированный 224
- указателя, Pascal 726
- Типичные ошибки, двоичный поиск 296
- Типы
  - данных, Pascal 96
  - построение 225
  - предобъявленные в языке Pascal 735
- Трай-дерево 533
  - анализ 537
  - включение 535
  - процедуры
    - InsertTrie 535
    - TrieSearch 535
  - реализация на языке Pascal 534
  - удаление 536
- Транспонированная матрица (упражнение) 402
- Трассировка
  - быстрой сортировки 354
  - поразрядной сортировки 408
  - программы 47, 74
  - рекурсивной программы 123
  - сортировки 333
- Требования к памяти, рекурсия 131
- Треугольная таблица 399
  - индексная функция 399
  - таблица доступа 400
- Треугольник Паскаля, биномиальные коэффициенты (упражнение) 141
- Трехдиагональная матрица (упражнение) 402
- Триангуляция многоугольников 659
- Тройная пирамида (упражнение) 386
- Удаление
  - из дерева двоичного поиска 469
  - из очереди 184
  - узла AVL-дерева 494
  - элемента из В-дерева 548
  - элементов из хеш-таблицы 421
- Узел
  - дерева 302
  - связного списка 711
  - связного стека 109
- Указатель
  - вывод на экран значения 699
  - объявление 710
  - ограничения 710
  - определение 704
  - преобразование в целое число 699
  - применительно к спискам 706
  - снятие ссылки 710
  - требования к памяти 712
- Упорядоченное дерево 525
  - определение 529
  - реализация 525
- Упорядоченный лес 528
- Упорядоченный список 295, 329
  - абстрактный тип данных 459
- Управляющие коды, мнемонические обозначения 735
- Уровень вершины дерева 302
- Усечение ключа, хеш-функция 415
- Утверждения 77
- Утилита 45
- Ученик мага 125
- Фазы жизненного цикла 94
- Файл
  - Pascal 42
  - страница или блок 538
- Файловое окно, Pascal 171, 710
- Фактор сбалансированности, AVL-дерево 487
- Факториал
  - аппроксимация Стирлинга 348
  - вычисление 119, 136
  - посредством приближения Стирлинга 653
  - с рекурсией и без рекурсии 136
- Ферзи, шахматная доска 145
- Фибоначчи
  - дерево 498
  - нерекурсивное вычисление 138
  - поиск 313
  - рекурсивное вычисление 137
  - числа 655
- Физическая реализация, очередь 185
- Фиксированная часть, запись 696
- Финансовые операции, LIFO и FIFO (упражнение) 185
- Формат программы, Pascal 33, 732
- Функция 404
  - индексная 396
  - множество значений 404
  - область значений 404
  - область определения 404
  - определение 404
  - степень роста 321
- Функция Аккермана (упражнение) 141
- Функция-утилита 45
- Хвост, очередь 183

Хвостовая рекурсия 134, 467, 543

упражнение 473

Хвостовой элемент, очередь 183

Хеширование 413

анализ 428

игра «Жизнь» 441

кластеризация 423

конфликт 414

минимально совершенная хеш-функция  
(упражнение) 426

совершенная хеш-функция 426

сравнение методов 431, 432

столкновение 414

Хеш-таблица 413

включение нового элемента 421

задача про дни рождения 428

зондирование, зависящее от ключа 419

игра «Жизнь» 440

идея 413

инкрементная функция 418

использование памяти 423

квадратичное зондирование 418

кластеризация 417

конфликт 414

коэффициент заполнения 429

линейное зондирование 417

операции по модулю 415

открытая адресация 417

переполнение (упражнение) 426

пример на языке Pascal 420, 424

программа оценки выражений 624

процедуры

ClearTable 420

CreateTable 420

DeleteTable 425

InsertTable 421

RetrieveTable 421

разреженная 413

разрешения столкновений 417

решивание 418

свертывание ключа 415

связные цепочки 422

случайное зондирование 419

сравнение ключей 428

столкновение 414

структуры данных 422

счет проб 429

удаление элементов 421

усечение ключа 415

функция Hash 416

функция деления по модулю 416

Хеш-функция 413

игра «Жизнь» 441

пример на языке Pascal 416

совершенная (упражнение) 426

Ход коня (проект) 154

Цепочка символов 246

операции 247

определение 246

процедуры

Concatenate 247

CopyString 247

CreateString 247

LengthOfString 247

Pascal 246

ReadString 247

Substring 248

WriteString 247

пустая 246

реализация 248

спецификации 247

функция PositionString 248

Церковь 188

Цикл, граф 570, 571

Циклы, Pascal 28

**Числа**

геометрические задачи 659

Каталана 656

Сегнера 659

случайные 663

Фибоначчи 137, 655

**Число  $e$**  (основание натуральных логарифмов)  
648

Шаги «зиг» и «заг», скошенное дерево 502

Шахматные задачи

восемь ферзей 145

ходы коня (проект) 154

Шкала Рихтера 646

Экспоненциальная зависимость времени вы-  
полнения 321

Экспоненциальная функция 649

Элементарные дроби 656

Эффективность, параметры 66

Якорь, связный список 712

*Учебное электронное издание*

Серия: «Программисту»

**Круз Роберт Л.**

## **СТРУКТУРЫ ДАННЫХ И ПРОЕКТИРОВАНИЕ ПРОГРАММ**

Ведущий редактор *И. А. Маховая*

Художественный редактор *Н. А. Новак*

Технический редактор *Е. В. Денюкова*

Корректор *Е. Н. Клитина*

Компьютерная верстка: *В. Ф. Носенко*

Подписано 11.04.14. Формат 70 × 100/16.

Усл. печ. л. 62,16.

Издательство «БИНОМ. Лаборатория знаний»

125167, Москва, проезд Аэропорта, д. 3

Телефон: (499) 157-5272

e-mail: [binom@Lbz.ru](mailto:binom@Lbz.ru), <http://www.Lbz.ru>

Минимальные системные требования определяются соответствующими требованиями программы Adobe Reader версии не ниже 10-й для операционных систем Windows, Android, iOS, Windows Phone и BlackBerry