国外经典计算机科学教材

Data Structures & Algorithms in Java

Java数据结构和算法

（第二版）

[美] Robert Lafore 著
计晓云 赵研 曾希 狄小菡 译

SAMS

中国电力出版社
www.infopower.com.cn

国外经典计算机科学教材

Data Structures & Algorithms in Java

Java数据结构和算法

（第二版）

[美]Robert Lafore 著
计晓云 赵研 曾希·狄小菡 译

中国电力出版社
www.infopower.com.cn

Data Structures & Algorithms in Java, second edition (ISBN 0-672-32453-9)
Robert Lafore

图书在版编目（CIP）数据
Java 数据结构和算法（第二版）/（美）拉佛著；计晓云等译. —北京：中国电力出版社，2003
ISBN 7-5083-1911-7
（国外经典计算机科学教材）
Ⅰ.J... Ⅱ.①拉...②计... Ⅲ.JAVA 语言－程序设计 Ⅳ.TP312
中国版本图书馆 CIP 数据核字（2003）第 111164 号

丛 书 名：国外经典计算机科学教材
书　　名：Java数据结构和算法（第二版）
编　　著：（美）Robert Lafore
翻　　译：计晓云　赵研　曾希　狄小菡
责任编辑：陈维宁
出版发行：中国电力出版社
　　地址：北京市三里河路6号　　邮政编码：100044
　　电话：（010）88515918　　传　　真：（010）88518169
印　　刷：汇鑫印务有限公司
开　　本：787×1092　1/16　　印　　张：36　　字　　数：925 千字
书　　号：ISBN 7-5083-1911-7
版　　次：2004 年 2 月北京第 1 版　　2004 年 2 月第 1 次印刷
定　　价：55.00 元

出 版 说 明

新世纪的朝阳刚刚露出丝抹微红，如火如荼的全球信息化浪潮便汹涌而至，让人无时无刻不感受到新一轮产业革命的气息。如何在这场变革中占尽先机，既是对民族信息业的挑战，也是机遇。从而，作为民族信息产业发展基石的高等教育事业就被赋予了比以往更重的责任，对培养和造就我国 21 世纪的一代新人提出了更高的要求。但在计算机科学突飞猛进的同时，专业教材的发展却严重滞后，越来越成为人才培养的瓶颈。同时，以美国为代表的西方国家计算机科学教育经历了充分的发展，产生了一批有着巨大影响力的经典教材，因此，以批判、借鉴的态度有选择地引进这些国外经典计算机教材，将促进国内教学体系和国外接轨，大大推动我国计算机教育事业的发展。

中国电力出版社进入计算机图书市场已有近 6 个年头，通过坚持“高端、精品、经典”战略，致力于与国外著名出版机构合作，出版了大批博得计算机业界和教育界赞誉的作品。通过与信息技术教育界人士的广泛沟通，同时依托丰富的出版资源，中国电力出版社适时推出了“国外经典计算机科学教材”的出版计划。本次教材出版计划是和美国最大的计算机教育出版机构——Pearson 教育集团（Addison-Wesley、Prentice-Hall 等皆为其下属子公司）合作，依托其数十年积累的大批经典教材资源，确保了教材选题的权威经典。

为保证这套教材的含金量，并做到有的放矢，我们在国内组织了由中国科学院、北京大学等一流院校教师组成的专家指导委员会，对高校课程教学体系做了系统、详细的调查，听取了众多教育专家、行业专家的意见，对教育部的教育规划进行了认真研究，并深入了解国外大学实际教学选用的教材状况，对国外教材做了理性的分析，确立了依托国家教育计划、传播先进教学理念、为培养符合社会需要的高素质创新型人才服务，来作为本次“国外经典计算机科学教材”出版计划的宗旨。

我们从 2002 年的下半年开始着手这套教材的策划工作，并多次组织了专家研讨会、座谈会等，分析现有教材的优点与不足，采其精华，并力争体现本套教材的质量和特色。

1. 深入理解国内的教学体系结构，并比照国外相同专业的课程设置，既具有现实的适用性，又立足发展眼光，具备一定的前瞻性。

2. 以计算机专业的核心课程为基础，同时配合专业教学计划，争取覆盖专业选修课程和专业任选课程。

3. 选取国外的最新教材版本，同时对照国内同专业课程的学时要求，对不适用的版本进行剔除，充分满足国内教学要求。

4. 根据专业对口和必须具备同课程教学经验的要求，严格挑选译者，并严把质量关，确保教材翻译的高质量。

5. 通过从原出版社网站下载勘误表及与原书作者进行沟通的方式，对原书中的错误一一做了修改。

6. 对教材出版的后期工作，如审校、编辑、排版、印刷进行了严格的质量把关。

经过专家指导委员会的集体讨论，并广泛听取广大高等院校师生的意见，反复比较，从数百种国外教材中遴选出数十种，列入第一阶段的出版计划。这些教材的作者无一不是学富五车的大师，

如 Stallings，Date，Ullman，Aho，Bryant，Sedgewick 等，他们的作品均是一版再版，并被众多国外一流大学如 Stanford University，MIT，UC Bekerley，Carnegie Mellon Univeristy，University of Michigan 等采用为教材。拟订的第一阶段出版计划包括 30 种图书，内容覆盖程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等计算机专业核心基础课程，基本满足国内计算机专业的教学要求。

此外，为了帮助广大任课教师加深对本系列教材的理解，减轻他们的备课难度，我们从国外出版机构引进了大批的课程教学辅助资料，并积极延请国内优秀教师，根据其使用该系列教材中的教学经验，着手编写更加适合国内应用状况的教辅材料。

由于我们对国内高校计算机教育存在认识深度上的不足，在选题、翻译、编辑加工出版等方面的工作中还有许多有待提高之处，恳请广大师生和读者提出批评和建议,并期待有更多的人加入到我们的工作中来。我们的联系方式是：

电子邮件：csbook@cepp.com.cn

联系电话：010-88515918-300

联系地址：北京市西城区三里河路 6 号中国电力出版社

邮政编码：100044

献　词

本书献给我忠实的读者，他们不仅年复一年地购买我的书，还总是给予我有益的建议和溢美之词。谢谢你们。

第一版致谢

在这份短短的谢辞中，无论如何也不能完全表达我对以下这些人（和其他许多人）的感激之情。Mitch Waite 总是能赶在其他人之前将 Java 的问题解决，他允许我毫无顾虑地同他就各种关于 applet（Java 小应用部件）的问题进行探讨，最终使它们都能够成功运行，同时还让我从思考的迷潭中提炼出本书的整体构架。我的编辑 Kurt Stephan 为本书找到了许多大师级的评论家，使大家同心同德，把事情进展下去，并温和而又坚定地使我相信我正在做一件应该做的事。Harry Henderson 为初稿作了内行的评价，并提出了许多有益的建议。Richard S.Wright 作为技术编辑，通过他那敏锐的双眼纠正了大量具体细节问题。新奥尔良大学的 Jaime Niño 博士多次尝试将我从困境中拯救出来，确实也产生了很好的效果，当然他不需要对书中的代码细节和我所采用的方法负任何责任。Susan Walton 帮助我把这个项目的实质传达给那些非技术人员，是我的一位坚定的支持者，我对她真是感激不尽。Carmela Carvajal 在我们与学术界的沟通中起了无价的作用。Dan Scherf 不仅将 CD-ROM 整合进来，并且不知疲倦地督促我跟上了迅速发展的软件更新的脚步。最后，我要感谢 Cecile Kaufman，他卓有成效地引领本书渡过了从编辑到出版的一系列过程。

第二版致谢

在这里我要对 Sams 出版社的如下这些人为本书第二版的出版所付出的精力、努力和耐心表示感谢。策划编辑(Acquisitions Editor)Carol Ackerman 和责任编辑(Development Editor)Songlin Qiu 出色地领导并管理了这次复杂的出版过程。项目编辑 Matt Purcell 对书中那些几乎无穷尽的语法错误进行了更改，并保证了所有的内容都有意义。技术编辑 Mike Kopak 复查了程序并指正了若干错误。我要感谢的最后但同样重要的人是一位原先的老朋友，Dan Scherf，他为我在 Sams Web 网站中的代码和 applets 提供了专业的管理。

简　介

在简介中简要说明：

- 第二版的新颖之处
- 本书相关内容
- 为什么它与众不同
- 本书是为哪些读者而作
- 在开始读书之前需要知道的知识
- 所需的软件和设备
- 本书是如何组织的

第二版的新颖之处

本书的第二版在第一版的基础上有所扩充，使之能更好地适应教师的计算机科学课堂教学，并使读者在阅读上更加容易。除去新增的章节，还添加了章末问题、实验和编程作业。

新增章节

在本版中加入了许多有趣的新内容。其中许多为编制上机作业提供了基础。这些新主题有：

- 深度优先搜索和游戏模拟
- 约瑟夫（Josephus）问题
- 数据压缩中的赫夫曼编码
- 旅行售货员问题
- 汉密尔顿（Hamilton）回路
- 骑士旅行问题
- 弗洛伊德（Floyd）算法
- 沃赛尔（Warshall）算法
- 2-3 树
- 背包问题
- 从 n 个事物中取 k 个的组合方案
- 哈希函数的数字折叠法
- 基数排序

章末问题

每章结束后都会有几个简短的问题，它们涵盖了该章的所有重点。在附录 C“问题答案”中可以找到相应的答案。这些问题是用来给读者自测的，以确保他们已经基本理解该章的内容。

实验

本书中收入了一些希望读者去做的活动。这些实验经常包括使用专题 applet 或通过示例程序来检查某算法的一些特性，当然有些实验只需要使用纸和笔，或只需要“思维实验”。

编程作业

最重要的是，在每章的结尾收入了一些（经常是五道）富有挑战性的编程题目。它们的难度各不相同，简单的题只是示例程序的变形，最有挑战性的是那些在该章中讨论过的但并没有样例的程序问题。本书没有附编程作业的答案和解法，但可以看下面的注释。

> **注释**
>
> 编程作业是为教师在留作业时提供参考用的。为此，那些有资格证明的教师可以得到编程作业题的参考解法的源程序和可执行文件。请访问 Sams Web 网站获取相关信息。

本书相关内容

本书是一本有关计算机编程中所应用的数据结构和算法的书。数据结构是指数据在计算机存储空间中（或磁盘中）的安排方式。算法是指软件程序用来操作这些结构中的数据的过程。

几乎所有的计算机程序都使用数据结构和算法，即使最简单的程序也不例外。比如设想一个打印地址标签的程序，这个程序使用一个数组来存储地址，并且使用一个简单的 for 循环来遍历数组，打印每一个地址。

在上面例子中的数组就是一个数据结构，用 for 循环来顺序访问该数组，这就构造了一个简单的算法。对于一个仅有少量数据的简单程序来说，上述的这种方法已经足够了。但是如果用程序来处理中等规模以上的数据或解决那些不太平常的问题时，就需要用一些更加复杂的技术来应付它们。仅仅知道诸如 Java 或 C++等计算机语言的语法是远远不够的。

本书提供了学完一门编程语言后进一步需要知道的知识。本书所涵盖的内容通常作为大学或学院中计算机系二年级的课程，在学生掌握了编程的基础后才开始本书的学习。

本书的不同之处

有关数据结构和算法的书很多，本书的不同之处有如下三条：

- 我们在写书过程中的主要目标是使本书所涉及到的知识尽可能地容易理解。
- 书中称作专题 applet（Workshop applet）的演示程序可以将知识生动化，它可以一步一步地通过“活动的图像”来展示数据结构和算法是如何工作的。
- 示例程序是用 Java 编写的，它比那些传统的用来演示计算机问题的语言，如 C、C++或 Pascal 更好理解。

让我们来更加详细地讨论上述特性。

容易理解

传统的计算机课本中充满了理论、数学公式和抽象的代码示例。而本书将精力集中在那些可以

解决现实问题的技术和方法的解释上，竭力避免那些复杂的证明和笨重的数学公式。本书中还有许多图表，它们可以增加文字所不能表达的效果。

许多数据结构和算法的书都包括了大量的软件工程内容。软件工程是一门有关设计和实现大型复杂的软件项目的学科。

然而我们认为数据结构和算法除去这些附加的规则后也是相当的复杂，所以我们在本书中故意不强调软件工程的知识。（我们会在第 1 章中讨论数据结构和算法与软件工程的关系。）

当然，我们在本书中使用了面向对象的方法，随着讨论的深入，会涉及到面向对象设计的各个方面，本书在第 1 章中还包含一个关于 OOP 的小教程。但是，我们的重点还是在数据结构和算法本身。

专题 applet

Sams Web 网站上可以下载由涉及本书中所讨论主题的 Java 应用小程序（applet）组成的演示程序。这些小程序（我们称之为"专题 applet"）可以在绝大多数 Web 浏览器中运行（请参阅附录 A）。专题 applet 通过“慢动作”的图像演示使读者能更好地理解一个算法的运行过程。

例如，在一个专题 applet 中，每按一下按钮，柱状图会按照由小到大的顺序进行一步排列。图中显示了排序算法中变量的值，图中的文字表明了正在进行的步骤，这样使读者能够清楚地看到在算法执行时计算机代码是如何工作的。

另一个小程序模拟了一棵二叉树。通过观察箭头的上下移动，可以跟踪插入或删除树中节点的每一步。本书中有 20 多个专题 applet，每一章至少会有一个。

专题 applet 可以将数据结构和算法的真实一面清晰地表达出来，这比文字描述要好得多。当然，我们在本书中同样会给出文字描述。通过专题 applet、文字和图例的结合，可使读者更容易理解书中的知识。

这些专题 applet 是可独立运行的图形化程序，可以把它们当作学习工具来补充书中的材料。请注意这些程序与我们下面要讨论的示例代码是不太一样的。

注释

关于专题 applet，在 Sams 网站（http://www.samspublishing.com/）中有 Java.class 形式的文件。请在输入框中输入本书的 ISBN 号（没有连字符），然后点击搜索。当本书的名称显示出来后，请点击可以下载小程序那页的链接。

Java 示例代码

Java 语言比 C 和 C++等语言都简单易懂且易写。最大的原因就在于它没有指针。有些人很惊讶为什么在建造复杂数据结构和算法时不需要指针。事实上，取消指针不仅意味着代码更加易写易懂，还为程序提供了更高的安全性和更少的出错可能性。

Java 是一门现代的面向对象的语言，这意味着可以使用面向对象的方法来进行编程。这一点非常重要，因为面向对象编程(OOP)不仅比过去的面向过程的方法拥有更多毋庸置疑的好处和优点，而且在正式的程序开发中正在迅速地取代面向过程的方法。读者如果对面向对象编程(OOP)不太熟悉的话，请不必惊慌，因为它并不那么难懂，尤其是通过象 Java 这种没有指针的语言环境来学习就更方便了。第 1 章将讲述面向对象编程的基础。

注释

与专题 applet 一样，示例程序（源代码和可执行文件）可以在 Sams 网站中下载。

本书的读者对象

本书可以用来当作数据结构和算法课程的课本，它通常是计算机科学课程表中二年级的课程。当然，它还适用于专业程序员和那些虽然仅有一些编程语言基础但是还想更上一层楼的人。本书还可以作为其他正式教科书的补充教材。

在阅读前所需的知识

在开始阅读本书之前，只需要知道一些编程语言的知识。

尽管示例代码是用 Java 写的，但是并不是只有懂得 Java 才能明白我们在解释什么。Java 也不难懂，况且我们尽可能使用了普通的语法，避开形式复杂的或 Java 专用的结构。

当然，如果对 Java 已经很熟悉，那是最好。因为 Java 的语法规则与 C++十分接近，所以如果会用 C++，对阅读本书也同样有帮助。对于示例程序来说，Java 和 C++的区别不大（只是对指针的接受程度不同），第 1 章中会讨论这两种语言。

阅读本书所需的软件

运行专题 applet 需要诸如 Microsoft Internet Explorer 或 Netscape Communicator 等 Web 浏览器，还可以使用 applet 察看工具，在许多的 Java 开发系统中都可以找到这些工具，包括在附录 A 中讨论的 Sun 公司的免费系统。

可以使用 Microsoft Windows 中的 MS-DOS 环境（使用 MS-DOS 命令）或类似的文本环境来运行示例程序。

如果希望对示例程序进行修改或编写自己的程序，还需要一个 Java 开发系统。这些系统有的是收费的，当然还可以从 Sun 公司下载一个优秀的免费系统，请参阅附录 A。

本书的组织结构

本节是为教师和那些希望快速了解本书内容的人而准备的。下面的内容假定读者对数据结构和算法中的问题和术语已经相当熟悉。

前两章试图使读者尽可能轻松地进入数据结构和算法的世界。

第 1 章“综述”，给读者一个各主题的总体印象并介绍少量后面要用到的术语。对于那些对面向对象编程不太熟悉的读者，本章总结了一些相关的知识。对于那些知道 C++而不熟悉 Java 的程序员，本章对这两种语言的主要差别进行了描述。

第 2 章“数组”，集中讨论数组。这里面包含有两层意思：如何使用类来对数据存储结构进行封装和类的接口。其中包括数组和有序数组的查找、插入、删除、线性查找和二分查找。专题 applet

通过对无序和有序的数组进行操作来解释上述算法。

第 3 章“简单排序”介绍三种简单的（但是慢速的）排序方法：冒泡排序、选择排序和插入排序。每一种排序都有一个相应的专题 applet。

第 4 章“栈和队列”涉及到三种可以被认为是抽象数据类型（ADT）的数据结构：栈、队和优先级队列。这些结构在本书中大量重复出现，是许多算法的基础。每一种结构都有一个相应的专题 applet。ADT 的概念也会在本章中讨论。

第 5 章“链表”介绍了链表中的双向链表和双端链表。本章还解释了 Java 中被称作“无痛指针”的使用，并用一个专题 applet 演示了链表的插入、查找和删除是如何进行的。

第 6 章“递归”探索了递归的知识，这是书中仅有的非数据结构的几章之一。本章给出了大量的递归例子，包括汉诺塔问题和归并排序，它们都有相应的专题 applet。

第 7 章“高级排序”研究了几种高级的排序方法：希尔排序和快速排序。专题 applet 演示了希尔排序，快速排序的基础——划分（partitioning）和两种形式的快速排序。

第 8 章“二叉树”开始了对树的探索。本章中介绍了最简单最通用的树型结构：不平衡的二叉搜索树。一个专题 applet 演示了此类树的插入、删除和遍历是如何进行的。

第 9 章“红-黑树”解释了红-黑树，它是最有效的平衡树之一。专题 applet 演示了平衡这种树所需的旋转和颜色转换。

第 10 章“2-3-4 树和外部存储”将 2-3-4 树作为多叉树的一个例子进行了讲解。专题 applet 会演示它们是如何工作的。我们还将讨论 2-3 树和 2-3-4 树与 B 树的关系，这些知识对于存储外部（磁盘）的文件十分有用。

第 11 章“哈希表”转到哈希表这个新的讨论领域。专题 applet 演示了几种方法：线性、二次探测和再哈希及链地址法。本章中还讨论了哈希表方法在组织外部文件方面的应用。

第 12 章“堆”讨论了一种特殊的树——堆，用它作为优先队列的一种有效的实现手段。

第 13 章“图”和第 14 章“带权图”处理图的相关问题，前者处理未加权图和简单的查找算法，后者处理带权图和更加复杂的算法，如最小生成树和最短路径。

第 15 章“应用场合”总结了前几章描述过的各种数据结构，还着重讨论了如何在给定情况下应用合适的数据结构的问题。

附录 A“运行专题 applet 和示例程序”提供了如何使用这两种软件的细节。此部分同时讲解了如何使用来自 Sun 公司的软件开发工具集，可以用它来修改示例程序或开发自己的程序，还可以通过它来运行专题 applet 和示例程序。

附录 B“进一步学习”介绍了一些关于数据结构和相关内容的进阶书籍。

附录 C“问题答案”包括了书中章末问题的解答。

请好好享受吧！

希望我们能使学习的过程尽可能地没有痛苦，依我们的理想，学习的过程甚至应该是快乐的。如果我们实现了这个理想，请让我们一起分享你所得到的快乐；如果没有，请告诉我们何处应该改进。

目　录

第1章 综 述

本章重点

- 数据结构和算法能起到什么作用？
- 数据结构的概述
- 算法的概述
- 一些定义
- 面向对象编程
- 软件工程
- 对于C++程序员的Java
- Java数据结构的类库

当你开始阅读这本书时，心中可能会产生许多问题：

- 什么是数据结构和算法？
- 学习它们后会有什么好处？
- 为什么不能只使用数组和for循环来处理数据？
- 何时何地使用在本书学到的知识才算有意义？

本章旨在解答上述问题。在这一章中先介绍了一些必备知识和那些为更深层次的章节所需的预备术语。

然后，在本章中还简要解释了必须具备的面向对象设计的思想，使那些尚未接触过面向对象的读者能够理解本书的内容。最后，我们为那些不清楚Java的C++程序员指出了这两种语言的不同之处。

数据结构和算法能起到什么作用？

本书的主题是数据结构和算法。数据结构是对在计算机内存中（有时在磁盘中）的数据的一种安排。数据结构包括数组、链表、栈、二叉树、哈希表等等。算法对这些结构中的数据进行各种处理，例如，查找一条特殊的数据项或对数据进行排序。

掌握这些知识以后可以解决哪些问题呢？粗略地估计一下，上述知识可以用于下面三类情况：

- 现实世界数据存储
- 程序员的工具
- 建模

这些并不是必须遵循的分类，但它们可以体现出本书内容的实用性和重要性。下面更加具体地讨论一下这些问题。

现实世界数据存储

在这里讨论的许多数据结构和技术与如何处理现实世界数据存储问题紧密相连。现实世界数据指的是那些描述处于计算机外部的物理实体的数据。看几个例子：一条人事档案记录描述了一位真实人的信息，一条存货记录描述了一个真实存在的汽车部件或杂货店里的一种商品，一条财务交易记录描述了一笔支付电费实际填写的支票。

举一个非计算机的现实世界数据存储的例子，有一叠3×5的索引卡片。这些卡片可以被用在不同的场合。如果每张卡上写有某人的姓名、地址和电话号码，那么这叠卡片一定是一本地址簿。如果每一张卡片上写有家庭拥有物的名称、位置和价值，那么这一定是一本家庭财产清单。

当然，索引卡片并不能代表现在的科技发展水平。几乎所有以前用索引卡片处理的事务现在都可以用计算机来处理。如果想将旧式的索引卡片系统更新为计算机程序，便有可能发现会被如下问题所困扰：

- 如何在计算机内存中安放数据？
- 所用方法适用于100张卡片吗？那1000张呢？1000000张呢？
- 所用方法能够快速地插入新卡片和删除老卡片吗？
- 它能快速地查找一张特定的卡片吗？
- 若想将卡片按照字母的顺序排列，又应该如何去排呢？

本书将会讨论类似于一叠索引卡片这种形式的数据结构。

然而，大多数程序比索引卡片要复杂得多。想像一下机动车管理局（不管它在你所处国家中叫什么）的数据库，这个库被用来记录驾驶员的执照的情况；或者看一个航班预订系统，这个系统存储了旅客和航班的各种信息。这些系统由许多数据结构组成，设计这些复杂的系统还需要应用软件工程的技术，这一点在本章的结尾将会谈到。

程序员的工具

并不是所有的存储结构都用来存储现实世界的数据。通常情况下，现实世界的数据或多或少会由程序的用户直接存取。但是有些数据存储结构并不打算让用户接触，它们仅被程序本身所使用。程序员经常将诸如栈、队列和优先级队列等结构当作工具来简化另一些操作。随着讨论的深入，我们将要见到这样的例子。

现实世界的建模

有些数据结构能直接对真实世界的情况构造模型。其中最重要的数据结构是图。图可以用来表示城市之间的航线，电路中的连接，或是某一工程中的任务安排关系。第13章（“图”）和第14章（“带权图”）中将详细介绍图的问题。其他诸如栈和队列等数据结构也会应用在事件模拟中。例如，一个队列可以模拟顾客在银行中排队等待的模型，还可以模拟汽车在收费站前等待交费的模型。

数据结构的概述

还可以通过另一个方面来看数据结构，那就是从它们的强项和弱项来看。在这一节中用表格的方式来概述本书中的主要数据结构。在深入到各章细节中之前，下面的表1.1显示了本书中不同的数据结构的优缺点，大致给出它们的特性，就象从天空中鸟瞰地面的风景一样。所以如果对其中的术语不太清楚的话，请不要太着急。

表1.1 数据结构的特性

数据结构	优点	缺点
数组	插入快，如果知道下标，可以非常快地存取	查找慢，删除慢，大小固定
有序数组	比无序的数组查找快	删除和插入慢，大小固定
栈	提供后进先出方式的存取	存取其他项很慢
队列	提供先进先出方式的存取	存取其他项很慢
链表	插入快，删除快	查找慢
二叉树	查找、插入、删除都快（如果树保持平衡）	删除算法复杂

续表

数据结构	优点	缺点
红-黑树	查找、插入、删除都快。树总是平衡的	算法复杂
2-3-4 树	查找、插入、删除都快。树总是平衡的。类似的树对磁盘存储有用	算法复杂
哈希表	如果关键字已知则存取极快。插入快	删除慢，如果不知道关键字则存取很慢，对存储空间使用不充分
堆	插入、删除快，对最大数据项的存取很快	对其他数据项存取慢
图	对现实世界建模	有些算法慢且复杂

表 1.1 中的数据结构除了数组之外都可以被认为是抽象数据结构（ADT）。第 5 章“链表”将讨论这个概念。

算法的概述

许多将要讨论到的算法直接适用于某些特殊的数据结构。对于大多数数据结构来说，都需要知道如何

- 插入一条新的数据项。
- 寻找某一特定的数据项。
- 删除某一特定的数据项。

还需要知道如何迭代地访问某一数据结构中的各数据项，以便进行显示或其他操作。

另一种重要的算法范畴是排序，排序有许多种算法，第 3 章“简单排序”和第 7 章“高级排序”都是讨论这些算法的。

递归的概念在设计某些算法时十分重要。递归意味着一个方法调用它自身。在第 6 章“递归”中将会仔细地讨论它（方法是 Java 中的术语，在其他语言中，它被称为函数、过程或子例程）。

一些定义

下面看一些在全书中都会用到的术语。

数据库（database）

我们将会使用数据库这个术语来表示在某一特定情况下所有要查阅的数据，数据库中的每一条数据都被认为是同样格式的。例如，如果使用索引卡片来做一本地址簿，其中所有的卡片便构成了一个数据库。文件这个术语有时也代表同样的意思。

记录（record）

记录是指数据库中划分成的单元。它们为存储信息提供了一个结构格式。在索引卡片的模拟系统中，每一张卡片就代表一条记录。当有许多类似的实体时，一条记录包含了某一个实体的所有信息。一条记录可能对应于人事档案中的某一个人，汽车供应存货目录中的某一个零部件，或是烹调书中的某一道菜谱。

字段（field）

一条记录经常被划分为几个字段。一个字段保存某一种特定的数据。在地址簿中的一张索引卡片上，一个人的名字、地址或电话号码都是一个独立的字段。

更复杂的数据库程序使用带有更多字段的记录。图 1.1 显示了一条记录，其中每一行代表了一个不同的字段。

在 Java 语言（和其他面向对象语言）中，记录经常被表示为一个相应类的对象。一个实例中各个变量表示不同的数据字段（data field）。在 Java 语言中一个类对象的字段被称为字段（但在其他语言中叫做成员，如 C++）。

雇员号码：
社会保险号码：
姓：
名：
地址：
城市：
省：
邮编：
电话：
生日：
开始工作时间：
工资：

图 1.1 有多个字段的记录

关键字

在数据库中查找一条记录，需要指定记录的某一个字段为关键字（或查找关键字）。通过这个特定的关键字来进行查找。例如，在一个地址簿的程序中，可以在每条记录的姓名字段中查找关键字“Brown”。当找到具有该关键字的记录时，便可以访问它的所有字段，而不仅仅是关键字了，可以说是关键字释放了整个记录。还可以通过电话号码字段或地址字段在整个文件中再次查找。在图 1.1 中的任何字段都可以被用作查找关键字。

面向对象编程

本节是为那些还没有接触过面向对象编程的读者准备的。但是，请读者自己注意，在短短的几页中要评判所有面向对象编程的富有创造精神的新思想是不可能的，我们的目标仅仅是使读者能够读懂课文中的示例程序。

如果在读完这部分并看过后面几章的示例程序之后，仍对整个面向对象的这件事陌生得如同对量子物理一样，那么恐怕就需要对面向对象编程进行一次更加彻底的探求了。请参阅附录 B“更进一步的阅读”中的阅读列表。

过程性语言的问题

面向对象编程语言的产生是由于发现过程性语言（诸如 C、Pascal 和早期版本的 Basic）在处理大型的复杂问题时有些力不从心。为什么会这样呢？

有两类问题：一是程序与现实世界缺乏对应关系，二是程序内部的结构出现了问题。

对现实世界建模的无能为力

使用过程语言对现实世界问题进行抽象及概念化十分困难：方法执行任务，而数据存储信息，但是现实世界中的事物是对二者同时进行操作的。例如，炉子上的自动调温器执行任务（炉子的开/关）但同时也存储信息（现在的温度和所希望的温度）。

如果用过程语言来写一个自动调温器控制程序，可能会以两个方法 furnace_on()和 furnace_off()，即炉开和炉闭来完成，但是还会有两个全局变量 currentTemp 和 desiredTemp，即现在的温度（由调温器提供）和希望的温度（由用户设置）。然而这些方法和变量并没有形成任何编程对象，在程序中不会出现任何可以称之为自动调温器的单元。这个单元的惟一概念仅存在于程序员的脑海中。

对于大型的程序，有可能包括上百个类似调温器的实体，过程语言面对这种情况会将程序搞得极为混乱，错误频繁出现，有时还完全不可能实现。因此需要一种可以更好地将程序中的事物与现实世界中的事物相匹配的语言。

粗糙的组织结构

解决程序的内部组织结构是一个更微妙而且事关重大的问题。面向过程的程序被划分为一个一个的方法。这种基于方法组织形式的一个巨大问题是它仅仅考虑了方法，而没有重视数据。当不得不面对数据时，它没有太多的选择。简言之，数据可以是一个特定的方法的局部量，也可以是所有方法都可以存取的全局量，就是无法（至少没有灵活的方法）规定一个变量只允许某些方法存取而不允许另一些方法存取。

当几个方法都要存取同一个数据时，这种不灵活性会产生问题。如果一个变量要想被一个以上的方法存取到，它必须是全局的变量。但是全局变量会在不经意的情况下被程序中的任何一个方法存取，这就导致了频繁的编程错误。因此需要一种可以精调数据的可访问性的办法；使数据对应该存取它的方法是可用的，而对其他方法来说是隐藏的。

对象简述

对象的概念在编程社团中渐渐传播开来，它被当作解决过程语言所面临问题的途径之一。

对象

面向对象编程思想的关键性突破就是：一个对象同时包括方法和变量。例如，一个自动调温器对象不仅包括 furnace_on() 和 furnace_off()两个方法，还包括 currentTemp 和 desiredTemp 两个变量。在 Java 中，这些变量被称为字段。

这个新的实体——对象，同时解决了许多问题。它不仅将计算机中的事物与现实世界中的事物联系的更加紧密，而且解决了在过程语言中由全局变量造成的麻烦。furnace_on()和 furnace_off()两个方法可以访问 currentTemp 和 desiredTemp，这两个变量对那些不属于自动调温器的方法是隐藏的，以防止它们被一些不可靠的方法所修改。

类

有些人认为对于一次编程技术的革命来说，一个全新的对象的概念就已经足够了，但实际上这并不足够。人们认识到在编程中有可能希望得到属于同一类型的好几个对象。例如，为整幢大楼编写一个温度控制器的程序，在程序中必然会出现许多控制器对象。如果为每一个都指定一段程序的话会很麻烦。类的概念便由此而生。

类是针对一个或多个对象的说明（或蓝图）。例如，下面是一个温度控制器类在 Java 中的形式：

```
class thermostat
   {
   private float currentTemp();
   private float desiredTemp();

   public void furnace_on()
      {
      // method body goes here
      }

   public void furnace_off()
```

```
    {
    // method body goes here
    }
} // end class thermostat
```

Java 语言中的关键字 class 引出了整个类的说明，随后是为这个类而起的名字，在这个例中，用 thermostat(温度控制器)作为类名。大括号中是组成这个类的字段和方法。我们在这个例子中略去了方法体（方法的内容），通常每个方法都有若干行代码。

C 语言的程序员会认为上面的语法规则同一个结构类似，而 C++程序员会意识到除了在最后没有分号之外，它同 C++中的类十分一致。（既然如此，我们究竟为什么要在 C++的类似情况下使用分号呢?）

创建对象

类的声明并没有创建这个类的任何对象（同样的，在 C 语言中声明一个结构体并没有创建任何变量），要想在 Java 中真正创建对象，必须使用关键字 new 。在创建对象的同时，需要将一个引用存储到一个具有合适的类型的变量中。

什么是引用？我们会在后面的部分更加详细地介绍。目前，先将引用认为是一个对象的名字。（实际上，它是一个对象的地址，但现在并不需要知道它真正是什么。）

下面的例子说明了我们如何创建两个温度控制器类的引用，创建两个新的控制器对象，并将对它们的引用存储在如下两个变量中：

```
thermostat therm1, therm2;  // create two references

therm1 = new thermostat();  // create two objects and
therm2 = new thermostat();  // store references to them
```

顺便提一下，创建对象也称作实例化对象，常把对象叫作类的实例。

访问对象方法

声明一个类并创建了几个对象后，程序就有可能需要让这些对象相互作用。这是如何做的呢？

一般来说，程序的其他部分通过调用方法与这些对象相互作用，而不是通过它的数据（字段）。例如，若想使 therm2（二号控制器）打开炉子，会写如下语句：

```
therm2.furnace_on();
```

点运算符（.）将一个对象同它的某一个方法（或有时同它的某个字段）连接起来。

至此我们（相当简单地）讨论了面向对象的几个最重要的特性。总结如下：

- 对象同时包括方法和字段（数据）。
- 类是任意数目的对象的说明。
- 创建一个对象，要将关键字 new 和类的名称连用。
- 调用一个对象的方法，要使用点运算符。

这些概念都很深奥难懂，第一次见到它们的时候更是不可能完全掌握，所以如果你现在感到有一些头晕的话，请不要着急。等见过更多的类和明白它们的功能之后，眼前的迷雾便会慢慢散去。

一个能运行的面向对象的程序

让我们来看一个能运行并真正有输出的面向对象程序。它描述了一个被称为 BankAccount 的类，该类模拟了银行中的账户操作。程序创建了一个开户金额、显示余额、存款、取款并显示新的余额。程序清单 1.1 显示了 bank.java。

清单 1.1 bank.java 程序

```
// bank.java
// demonstrates basic OOP syntax
// to run this program: C>java BankApp
////////////////////////////////////////////////////////////////
class BankAccount
   {
   private double balance;                  // account balance

   public BankAccount(double openingBalance) // constructor
      {
      balance = openingBalance;
      }

   public void deposit(double amount)       // makes deposit
      {
      balance = balance + amount;
      }

   public void withdraw(double amount)      // makes withdrawal
      {
      balance = balance - amount;
      }

   public void display()                    // displays balance
      {
      System.out.println("balance=" + balance);
      }
   } // end class BankAccount
////////////////////////////////////////////////////////////////
class BankApp
   {
   public static void main(String[] args)
      {
      BankAccount ba1 = new BankAccount(100.00); // create acct

      System.out.print("Before transactions, ");
      ba1.display();                        // display balance

      ba1.deposit(74.35);                   // make deposit
```

```
        ba1.withdraw(20.00);                    // make withdrawal

        System.out.print("After transactions, ");
        ba1.display();                          // display balance
        }  // end main()
    }  // end class BankApp
```

下面是程序的输出：

```
Before transactions, balance=100
After transactions, balance=154.35
```

在 bank.java 中有两个类。第一个是 BankAccount，它包括了银行账户的字段和方法，随后我们会详细察看它的功能。第二个是 BankApp，它在程序中扮演了一个特殊的角色。

BankApp 类

若要从 MS-DOS 命令行运行清单 1.1 中的程序，请在 C:提示符后键入 java BankApp:

```
C:\>java BankApp
```

这条命令会使 Java 解释器在 BankApp 类中查找名为 main()的方法。每一个 Java 执行程序必须有一个 main()方法，程序从 main()的开头开始执行，这正如同从程序清单 1.1 中看到的。（请不要在意 main()中的 String[] args 参数。）

main()方法通过下面这条语句创建了 BankAccount 类的一个对象，初始化开户余额为 100.0：

```
BankAccount ba1 = new BankAccount(100.00); // create acct
```

System.out.print()方法将用作其参数的字符串 Before transactions，显示出来，账户通过下面的语句显示它的余额：

```
ba1.display();
```

随后程序从账户中存了一笔款，又取出一笔钱：

```
ba1.deposit(74.35);
ba1.withdraw(20.00);
```

最后，程序将新的账户余额显示在屏幕上并退出。

BankAccount 类

在 BankAccount 类中，惟一的数据字段是账户中的存钱余额（balance）。它有三个方法，deposit()方法将余额加上一个数，withdrawal()将余额减去一个数，display()显示余额。

构造函数

BankAccount 类还拥有一个构造函数，这是一个特殊的方法，在每个对象创建时都会被自动调用。构造函数总是与类的名称相同，在本例中它便叫做 BankAccount()。这个构造函数有一个参数，它被用来在创建对象时对开户金额进行设置。

构造函数可以很方便地对一个新的对象进行初始化。如果程序中没有构造函数的话，则只能在创建时增加对 deposit()的调用，以便将开户余额注入到账户里。

公有和私有

请注意在 BankAccount 类中的关键字 public（公有）和 private（私有）。这些关键字是访问修饰符，它们决定了某些方法是否能够访问另一个类的字段或方法。balance 字段前面有 private。当一

个字段或方法为私有（private）时，它仅能被同一个类中的方法访问。因为 main()不是 BankAccount 类中的方法，所以 balance 不能被 main()中的语句访问。

在 BankAccount 类中每个方法都有访问修饰符 public，因此它们可以被外部的类所访问。这就是为什么 main()函数中的语句可以调用 deposit()、withdrawal()和 display()的原因。

一个类中的数据字段经常被设为私有的，而方法经常是公有的。这样可以保护数据，使之不会被其他类的方法所修改。所有外界的实体要想访问一个类中的数据，必须使用那个类自己的方法。数据就像是蜂后，它被藏在蜂巢里面，被工蜂（就是方法）精心地喂养、照料。

继承与多态

下面简要地提一下面向对象编程的其他两个重要特征：继承与多态。

继承是指由基类扩展或派生形成一个新类。这个扩展类拥有基类的所有属性，并加上了几种其他属性。例如，秘书类可能是从一个更加一般化的雇员类派生而来，它也许会包括一个雇员类所缺少的字段：打字速度。

在 Java 语言中，继承又称为子类化（subclassing）。基类被称为父类，扩展类被称为子类。

继承可以方便地向一个现有的类中增加属性，这对于设计一个有许多相关类的系统而言，是十分重要的。由于用途发生微小变化而需要重用类，而继承使这个工作变得十分简单，这是 OOP 的一个重要的优点。

多态指的是以相同的办法处理来自不同类的对象。为了使多态能够正常运行，这些不同的类必须从同一个基类中派生出来。实际上，多态经常指的是通过一种方法的调用，而实质是对不同的类的对象执行不同的方法。

例如，调用一个 secretary 对象中的 display()方法会激活 secretary 类中的相应方法，当对一个 manager 对象进行同样调用时，便会激活 manager 类中的不同的显示方法。多态可以简化程序设计和编写，并使之更加清晰。

对那些不熟悉继承和多态的人来说，可能会带来多余的麻烦。为了将注意力集中在数据结构和算法上，示例程序会尽量避免使用这些特性。继承和多态是 OOP 的重要且强大的特性，但它们对于数据结构和算法的解释并不是必需的。

软件工程

最近几年，在数据结构和算法的书中时兴将以软件工程为内容的一章作为开场白。本书不打算采用这种方式，但是还会简要地讨论一下软件工程，并简述它同本书中的内容是如何相关的。

软件工程研究的是由许多程序员参与的大型复杂的计算机程序的创建方法。它强调的是程序的整体设计和如何依照最终用户需求而进行设计的问题。软件工程关系着一个软件项目的整个生命周期，包括分析、设计、验证、编码、测试、生产和维护各阶段。

将软件工程与数据结构和算法的知识融合在一起对学生学习这两个不同的主题是否有帮助，现在还不得而知。除非参加一个大型的项目，否则软件工程相当抽象，很难被领悟。但是另一方面，数据结构和算法却是有关编程和数据存储的细致入微的基本规则。

因此，我们将精力集中在数据结构和算法的精华所在。它们是如何工作的？在特定的情况下，哪种结构或算法是最好的？在 Java 程序中，它们是如何实现的？正如上文所述，本书尝试将这些问

题尽可能地简单化。如果希望更进一步的学习，请参阅附录B中有关软件工程的书。

对于C++程序员的Java

如果你是一位从未遇到过Java的C++程序员，请阅读本节。下面会提到Java与C++的几处不同。

本书不打算把这一节作为Java的初级教程，甚至都没有将Java与C++所有的不同点都包括进来。在这里只列出了那些可以帮助C++程序员看懂示例程序的Java特性。

没有指针

Java和C++的最大不同就在于Java没有指针。对于一个C++程序员来说，没有指针在最初看起来会很令人惊讶，程序离了指针怎么能运行呢？

在整本书中，无指针的代码编写了复杂的数据结构，这种方法不仅可能，而且事实上比使用C++指针更简单。

其实，Java只是摆脱了显式表露的指针，指针依旧以存储地址的形式埋藏在程序的深处。有时甚至可以说，在Java中，所有东西都是指针。这句话虽不是百分之一百的正确，但也差不多。下面来更仔细地讨论这个问题。

引用（reference）

Java对基本数据类型（如int、float和double）的处理与对象的处理不同。请看下面两条语句：

```
int intVar;       // an int variable called intVar
BankAccount bc1;  // reference to a BankAccount object
```

在第一行语句中，一个被称为intVar的存储地址保存了一个数值127（假设这个值已经被放在那里）。然而，bc1这个存储地址并没有保存BankAccount对象的数据。与第一句不同的是，它存储了一个BankAcount对象的地址，而这个对象实际上被存储在内存空间的其他某个地方。这个名称bc1是对象的一个引用，它并不是对象本身。

实际上，如果bc1没有被预先赋值为程序中某个指针预先指向的对象的话，它就不会成为引用。在赋值为某个对象之前，它保存一个被称做null的特殊对象的引用。同样，intVar在它被赋指之前也不会保存数值。如果尝试使用一个没有被赋值的变量，编译器便会报错。

在C++中的语句：

```
BankAccount bc1;
```

实际上创建了一个对象；它留出了所有这个对象的数据的空间。在Java中，这条语句只创建了一个放置某一对象的存储地址的空间。可以将引用认为是普通变量语法中的指针。（C++也有引用变量，但它们必须用&符号显式说明。）

赋值

Java中的赋值操作符与C++中的不一样。在C++中，这条语句：

```
bc2 = bc1;
```

将一个名为bc1的对象的所有数据都拷贝到另一个名为bc2的对象中。

这条语句执行后，程序中有两个含有相同数据的对象。然而在Java中，这条相同的赋值语句只向bc2中拷贝了bc1指向的存储地址，现在bc1和bc2实际上指的是同一个对象，它们都是这个对

象的引用。

如果对赋值操作符理解不深的话，上面的语句会让人感到困惑。在那条语句之后是下面两句：

```
bc1.withdraw(21.00);
```

和

```
bc2.withdraw(21.00);
```

这两次都是从同一个银行账户对象中取款的。

如果想从一个对象中复制数据到另一个对象中，必须保证在一开始就有两个不同的对象，然后分别拷贝每一个字段，等号是不起复制作用的。

new 操作符

在 Java 中任何创建对象的工作都必须使用 new。但是 new 在 Java 中返回一个引用，而不像 C++中返回一个指针。因此，指针并不需要使用 new。下面是创建对象的一种方法：

```
BankAccount ba1;
ba1 = new BankAccount();
```

取消指针意味着一个更安全的系统。程序员不可能找到 ba1 的真实地址，也就不可能意外地破坏它。除非故意想做一些使系统混乱的事情，否则的话是不需要知道真实地址的。

用 new 向系统申请空间后，如何去释放那些不再使用的空间？在 C++中，可以使用 delete。在 Java 中，则根本不需要对释放空间担心。Java 每隔一段时间就会查看每一块由 new 开辟的内存，看指向它的有效引用是否依旧存在。如果这个引用不存在，系统会自动将这块空间归入空闲内存区。这个过程被称为垃圾收集。

几乎所有的程序员在使用 C++过程中都会有忘记删除存储空间的经历，这会导致“存储泄漏”（memory leak）从而消耗系统资源，使程序处于不稳定的状态，甚至会导致系统崩溃。存储泄漏在 Java 中不可能出现（或至少几乎没有）。

参数

在 C++中，指针经常被用来在函数之间传递对象，从而避免拷贝一个大的对象的系统开销。在 Java 中，对象经常以引用的形式传递。这种方法同样避免了对象的拷贝：

```
void method1()
   {
   BankAccount ba1 = new BankAccount(350.00);
   method2(ba1);
   }

void method2(BankAccount acct)
   {
   }
```

在上面的代码中，引用 ba1 和 acct 都指向同一个对象。而在 C++中，acct 则是从 ba1 拷贝而来的另一个对象。

然而，简单数据类型总是通过它的值来传递。即由方法创建一个新的变量，并将参数的值复制到这个新的变量中去。

相等与同一

在 Java 中，对基本数据类型，可以通过相等操作符(= =)来判断两个变量是否含有相同的值：

```
int intVar1 = 27;
int intVar2 = intVar1;
if(intVar1 == intVar2)
   System.out.println("They're equal");
```

这同 C 和 C++中的语法是一致的。但是在 Java 中，由于关系操作符使用引用，所以它们在涉及到对象的判断时有些不同。当使用相等操作符(= =)判断类时，实际上判断的是类的引用是否一致，即它们是否指的是同一个对象：

```
carPart cp1 = new carPart("fender");
carPart cp2 = cp1;
if(cp1 == cp2)
   System.out.println("They're Identical");
```

在 C++中，这个操作符会判断出两个对象是否含有相同的数据。如果在 Java 中要判断两个对象中是否含有相同的数据，则要使用 Object 类中的 equals()方法：

```
carPart cp1 = new carPart("fender");
carPart cp2 = cp1;
if( cp1.equals(cp2) )
   System.out.println("They're equal");
```

能够使用这个方法是因为 Java 中所有类都是从 Object 类中派生而来的。

重载操作符

Java 中没有重载操作符。在 C++中，可以重新定义+、*、=及大多数其他操作符，以便使它们在特定的类中达到不同的效果。在 Java 中，任何类似的重新定义都是不可能的，而可以使用命名的方法，例如 add()或其他名字。

基本变量类型

表 1.2 是在 Java 中的基本或内置的变量类型。

表 1.2 基本数据类型

名称	大小（以位计）	取值范围
boolean	1	true 或 false
byte	8	-128～+127
char	16	'\u0000'～'\uFFFF'
short	16	-32768～+32767
int	32	-2147483648～+2147483647
long	64	-9223372036854775808～ +9223372036854775807
float	32	约 10^{-38}～10^{+38}; 7 位有效数字
double	64	约 10^{-308} 或 10^{+308}; 15 位有效数字

与 C 和 C++用整数类型来表示真/假值不同的是，Java 中的 boolean 型是一个独立的类型。

char(字符)型是无符号的，采用两个字节的空间来表示 Unicode 字符集，这个字符集可以处理国际通用的字符。

在 C 和 C++中 int 型的大小可能不同，这取决于它们运行的计算机环境；在 Java 中，一个 int 型的变量永远是 32 位。

float 型变量的字面表达用 F 做后缀（例如，3.14159F）；double 型变量的字面表达不需要后缀。long 型变量的字面表达用 L 做后缀（例如，45L）；其他整型的字面表达不需要后缀。

Java 与 C 和 C++相比，属于强类型语言。在其他语言中可以由系统自动进行的转换，在 Java 中却需要显式的转换（explicit cast）。

所有不包括在表 1.2 中的类型，例如 String，都是类。

输入/输出

随着 Java 的进化，输入输出也在跟着发生变化。对于类似书中即将出现的控制台模式的示例程序，输入输出可以用一些看起来比较笨拙但却很有效的方法。它们同 C++中广泛应用的 cout 和 cin 以及 C 中的 printf()和 scanf()截然不同。

若是用 Java 软件开发环境工具集（SDK）的较早版本实现输入输出例程(routine)，那么在源程序的最开始需要有一行：

```
import java.io.*;
```

现在只有当有输入时，才需要这一行语句。

输出

可以通过下面的语句输出任何基本类型数据（数字和字符），String 类也可以：

```
System.out.print(var);    // displays var, no linefeed
System.out.println(var);  // displays var, then starts new line
```

print()方法将光标停在同一行上；而 println()方法将光标移至下一行的开始。

在 SDK 的老版本中，System.out.print()实际上并没有向屏幕上写任何东西，它的后面必须紧接一句 System.out.println()或 System.out.flush()才能将缓冲区中的所有东西显示出来。但是在现在的版本中，它会直接显示出来。

参数可以用好几个变量，中间用加号连接即可。假设下句中的 ans 值为 33：

```
System.out.println("The answer is " + ans);
```

输出为：

```
The answer is 33
```

输入字符串

输入比输出更棘手。通常希望程序读入的所有值都被当作一个 String 类。如果输入的实际上是其他的东西，如一个字母或数字，就还得需要将 String 类转换为希望的类型。

正如我们所注意到的，有输入过程的程序的开始必须包括下条语句：

```
import java.io.*;
```

如果没有的话，编译器便不能识别如 IOException 和 InputStreamReader 这样的实体。

字符串输入很复杂。下面的方法返回用户输入的字符串：

```
public static String getString() throws IOException
   {
   InputStreamReader isr = new InputStreamReader(System.in);
   BufferedReader br = new BufferedReader(isr);
   String s = br.readLine();
   return s;
   }
```

这个方法返回一个 String 类对象，它由从键盘输入的字符组成，通过回车键终止。现在并不需要对 InputStreamReader 和 BufferedReader 类的细节知道得太多。

除了引入 java.io.*，还需要像上面的代码一样，为所有的输入方法加上 throws IOException。事实上，所有方法，譬如 main()，凡是要调用任何输入方法的，就需要在其中加上 throws IOException。

输入字符

当程序用户输入一个字符时（输入指键入一些字母并按下回车键），他有可能输入一个或输入多于一个（错误）的字符，因此最安全的读入一个字符的方法是，先读入一个字符串，再通过 charAt() 方法摘取它的第一个字符：

```
public static char getChar() throws IOException
   {
   String s = getString();
   return s.charAt(0);
   }
```

String 类的 charAt()方法返回一个 String 类对象中某个特定位置的字符，在上面的例子中，是第一个字符，号码是 0。这个方法避免了无关的字符留在输入的缓存中，正是无关的字符导致后续的输入总是出现错误。

输入整数

读入数字需要得到一个 String 类对象并用转换方法将其转成所需的类型。下面这个 getInt()方法就是将 String 类对象转化成 int 型并返回：

```
public int getInt() throws IOException
   {
   String s = getString();
   return Integer.parseInt(s);
   }
```

Integer 类的 parseInt()方法将字符串转成 int 型。还有一个类似的例程——parseLong()，可以转成 long 型。

在 SDK 的老版本中，任何使用 parseInt()方法的程序开头都必须包括下条语句：

```
import java.lang.Integer;
```

但现在已不是必需的了。

为了简单起见，示例程序中的输入例程没有加进任何纠错过程。用户可能正确输入，但也会有异常出现。在上面的代码中，异常会导致程序的终止。在正式的程序中应该在转换之前先分析输入的字符串，捕获所有异常并正确地处理它们。

输入浮点型数

输入浮点数和双精度数的处理方法和处理整数的方法几乎一样，但转换的过程更复杂。下面的代码可以读入一个双精度型数：

```
public int getDouble() throws IOException
   {
   String s = getString();
   Double aDub = Double.valueOf(s);
   return aDub.doubleValue();
   }
```

字符串先转换成一个 Double 型的对象（D 大写），它是 double 类型的“封装”类。然后 Double 类的 doubleValue()方法将这个对象转化成 double 型。

对于 float 型来说，同样有一个 Float 类，其中同样也有 valueOf()和 floatValue()这两个方法。

Java 数据结构的类库

Java.util 包中包含有诸如向量（一个可扩充的数组）、栈、库（dictionary）和哈希表等类型的数据结构。本书中经常故意忽视这些内置的类。我们将精力集中在教授基础知识，而不是某个特殊操作的细节。当然，有时这些结构还是很有用的。

如果想使用这些类，必须先加入

```
import java.util.*;
```

尽管重点不是介绍这些类库，不论是 Java 自带的还是第三方开发人员提供的类库，但这些类都是通用的，并经过了调试，它们为我们提供了丰富的资源。本书会使你明白何种数据结构是所需的，它是如何运行的。只有真正明白数据结构的基础，才能决定是应该写自己的类还是用别人的类。

小 结

- 数据结构是指数据在计算机内存空间中或磁盘中的组织形式。
- 正确选择数据结构会使程序的效率大大提高。
- 数据结构的例子有数组、栈和链表。
- 算法是完成特定任务的过程。
- 在 Java 中，算法经常通过类的方法实现。
- 本书中介绍的大部分数据结构和算法经常被用来建造数据库。
- 一些数据结构的用途是作为程序员的工具：它们帮助执行算法。
- 其他数据结构可以模拟现实世界中的情况，例如城市之间的电话线网。
- 数据库是指由许多类似的记录组成的数据存储的集合。
- 一条记录经常表示现实世界中的一个事物，例如一名雇员或一个汽车零件。
- 一条记录被分成字段。每个字段都存储了由这个记录所描述事物的一条特性。
- 一个关键字是一条记录中的一个字段，通过它可以对数据执行许多操作。例如，人事记录可以通过 LastName 字段进行排序。

- 可以搜索数据库以便找到关键字字段有定值的所有记录，这个值被称为查找关键字。

问 题

下列问题作为读者的自测题。答案可见附录 C。

1. 对于许多数据结构来说，可以_____一条记录，可以_____它，还可以_____它。
2. 按照某种顺序对一个数据结构的内容进行重新排列被称为_____。
3. 在数据库中，一个字段是
   a. 一条特殊的数据项。
   b. 一个特殊的对象。
   c. 一条记录的一部分。
   d. 一个算法的一部分。
4. 当查找一个特定的记录时，所使用的那个字段被称为_____。
5. 在面向对象程序设计中，一个对象
   a. 是一个类。
   b. 可能包含有数据和方法。
   c. 是一个程序。
   d. 可能含有类。
6. 一个类
   a. 是许多对象的蓝图（blueprint）。
   b. 表示了一个特定的现实世界中的事物。
   c. 在它的字段中可以保存特殊的值。
   d. 规定了一个方法的类型。
7. 在 Java 中，声明（specify）一个类
   a. 创建了一个对象。
   b. 需要关键字 new。
   c. 创建了一个引用。
   d. 以上均不是。
8. 当一个对象想做一些事情时，它使用一个_____。
9. 在 Java 中，访问一个类的方法要用_____操作符。
10. 在 Java 中，boolean 和 byte 是_____。

（第 1 章中没有实验和编程作业。）

第 2 章
数　组

本章重点

- Java 中数组的基础知识
- 将程序划分成类
- 类接口
- 有序数组的 Java 代码
- 对数
- 存储对象
- 大 O 表示法
- 为什么不用数组表示一切?

数组是应用最广泛的数据存储结构。它被植入到大部分编程语言中。由于数组十分易懂，所以它被用来作为介绍数据结构的起步点，并展示面向对象编程和数据结构之间的相互关系。本章中将会介绍 Java 中的数组并展示自制的数组类。

本章还将分析一种特殊的数组：有序数组，其中的数据是按关键字升序（或降序）排列的。这种排列使快速查找数据项成为可能：即可以使用二分查找。

本章以一个能展示数组插入、查找、删除的 Java 专题 applet 开始，随后给出实现相同操作的 Java 代码。

后面将分析有序数组，同样以专题 applet 开始，这个 applet 会演示二分查找的过程。最后介绍大 O 表示法，它是应用最广泛的评价算法效率的方法。

Array 专题 Applet

假设你正在训练少儿棒球联盟的队伍，并希望知道队中哪些运动员出现在训练场上，就需要在笔记本电脑里存有一个出勤记录程序，它可以维护参加训练的运动员的数据库。需要用一个简单的数据结构对这些数据进行保存，还可能要执行以下这些操作：

- 当运动员到场时，向数据结构中插入一条记录。
- 通过在结构中查找运动员的号码来查看某个运动员是否出席。
- 当运动员退场时，从数据结构中删除一条记录。

这三个操作：插入、查找和删除，是本书所介绍的大部分数据结构的最基本的三个操作。

本书将频繁地通过演示与某个数据结构有关的专题 applet 来开始讨论。用这种方法可以在开始详细解释和给出示例代码之前，使你对这种结构和算法的作用有一个感性认识。由被称作 Array 的专题 applet 来演示数组的插入、查找和删除是如何进行的。

现在打开 Array 专题 applet，正如附录 A 中所介绍的，键入

```
C:\>appletviewer Array.html
```

图 2.1 显示了有 20 项的数组，其中含有 10 条数据项。可以将这些数据项认为是棒球队的球员。假设球队中每位队员的背心后面都有一个号码。为了使模拟更加真实有趣，背心有不同的颜色。每位队员的号码和背心颜色都显示在数组中。

这个 applet 演示了前面提到的三个基本操作：

- Ins 按钮插入一个新的数据项。

- Find 按钮查找特定的数据项。
- Del 按钮删除特定的数据项。

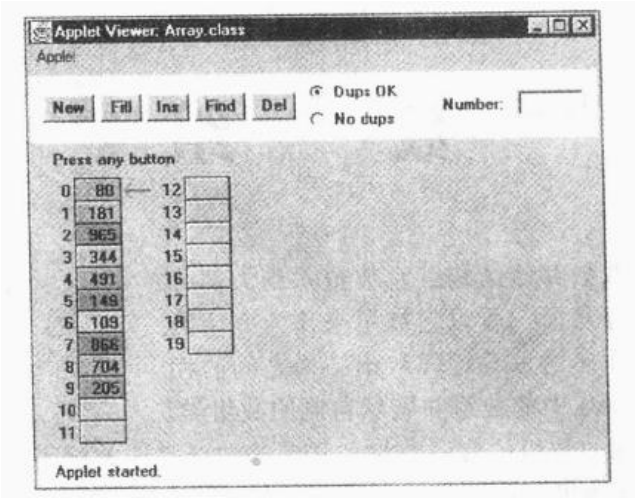


图 2.1 Array 专题 applet

使用 New 按钮，可以创建一个指定大小的新数组。可以通过 Fill 按钮向数组中填入尽可能多的数据。Fill 创建了一组数据项并分配给它们各自的号码和颜色。号码的范围是 0~999。数组大小的限制是最多只能创建 60 个数据项，同时也不能向数组中填入超过数组大小的数据。

当选择创建新数组时，需要决定数组中的数据项是否可以相同，过一会我们将回到这个话题上来。缺省值为不允许重复出现相同值，所以单选按钮 No Dups 最初被选中以表明这个设定。

插入

以缺省的设置开始程序，其中有一个大小为 20 的数组和 10 个数据项，No Dups 按钮被选中。当球员到达训练场时，需要向数组中插入一个棒球队员的号码。按一下 Ins 按钮来插入一条新数据项。程序将提示输入数据项的值：

```
Enter key of item to insert
```

在 applet 右上角的文本框中输入一个数，比如 678。（当然，在子节点背心上写三个数字是比较难的。）再按 Ins 按钮，随后 applet 会确认输入：

```
Will insert item with key 678
```

最后再按一下此按钮，数组中第一个空着的单元将会出现这个值和随机的颜色数据。屏幕上将会出现类似下面的提示：

```
Inserted item with key 678 at index 10
```

在专题 applet 中每次点击按钮都对应算法执行的一步。算法越长，步数越多。在 Array 专题 applet 中插入过程是很快的，它只需要一步即可完成。这是由于新的数据项总是插在数组中第一个空位上，并且由于数组中已有数据项个数已知，所以算法知道这个空位的具体位置。新的数据项只是简单地插入到下一个可用的空间中。然而查找和删除却没有这么快。

在不允许出现相同值（No Dups）的模式下，请自觉地不要插入一个与数组中已有数据项关键字（key）相同的数据项。如果这样做的话，applet 会显示出错误信息，但它不会阻止这样的输入，

因为它假设用户不会犯诸如此类的错误。

查找

点击 Find 按钮开始查找。程序会提示输入查找队员的号码。从数组中部挑出一条记录的号码。输入这个号码并再次按 Find 按钮。每按一下，算法就会执行一步。红箭头会从 0 单元开始依次向下移动，每一次会判断一个新的数据项。按步执行时，消息中的下标值会改变，如：

```
Checking next cell, index = 2
```

当找到该数据项后，屏幕上会出现如下信息：

```
Have found item with key 505
```

说明已找到输入的关键字值。假设数据项值不允许重复，查找过程会在找到该关键字值后立即结束。

如果在数组中未出现输入的关键字，则 applet 在检查所有的非空单元后会告知未找到该数据项。

注意查找算法（同样是假设不允许重复数据项值）必须平均搜索一半的数据项来查找特定数据值。找数组头部的数据项快，找数组尾部的数据项慢。设数据项个数为 N，则一个数据项的平均查找长度为 N/2。在最坏的情况下，待查的数据项在数组的最后，这需要 N 步才能找到。

正如上文所说过的一样，执行算法的时间长度与执行步数成正比，所以查找算法的时间（N/2 步）要比插入算法（一步）长很多。

删除

只有在找到某一数据项后才能删除它。输入待删除数据项的值，重复点击按钮使箭头一步步下移，直至找到该数据项。再点击一次按钮，该数据项在数组中的单元变空。（严格说来，这一步并不必要，因为无论如何该数据项的位置上将会再拷入另一个数据项，但删除这个数据项是为了清楚地表明算法执行的情况。）

删除算法中暗含着一个假设，即数组中不允许有洞。洞指的是一个或几个空单元，它们后面还有非空数据项（在更高的下标处还有数据项）。如果删除算法中允许有洞，那么所有其他的算法都将变得更加复杂，因为在查看某一单元的数据项之前还需判断它是否为空。同样，算法会由于需要找到非空数据项而变得效率很低。因此，非空数据项必须连续存储，不能有洞。

所以当找到特定数据项并将其删除后，applet 必须将随后的数据项都向前移一步来填补这个洞。图 2.2 演示了一个例子。

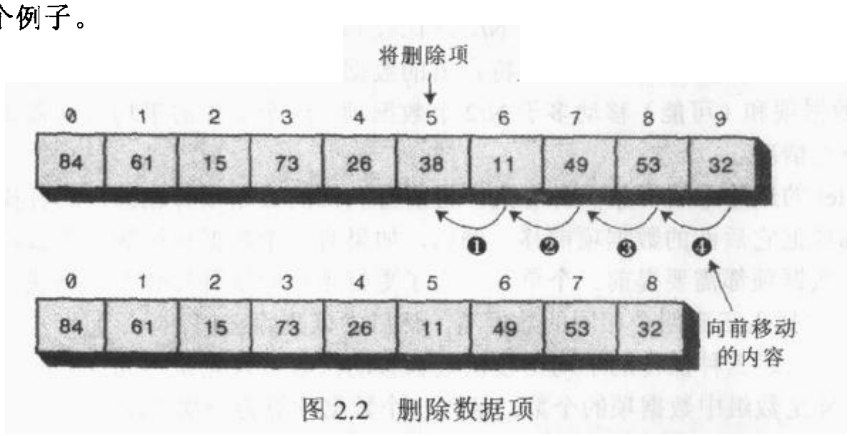


图 2.2 删除数据项

如果第5号单元中的数据项（在图2.2中是38）被删除，则6号数据项前移到5号，7号移到6号，依此类推至最后一个数据项。在删除过程中，当数据项位置被确定后，applet会在持续按Del键的同时将更高序号的数据项前移。

删除需要（假设不允许数据项重复）查找平均N/2个数据项并平均移动剩下的N/2个数据项来填充删除而带来的洞。总共是N步。

重复值问题

设计数据存储结构时需要决定数据项的关键字是否可以重复。如果处理的是人事档案，则关键字雇员号码不能是重复值，因为不可能把相同的号码分配给两个雇员。然而，如果关键字的值是每个人的姓，就很有可能几个雇员有相同的关键字值，这种情况下应该允许重复。

当然，对于棒球队员来说，重复的号码是不行的。如果有人穿着相同号码球衣的话，跟踪并了解这些球员在场上的情况就会变得很困难。

Array专题applet允许对此进行选择。当使用New按钮来创建一个数组时，程序会提示输入它的大小和确定是否允许重复值，使用单选按钮Dups OK或No Dups来作出选择。

编写一个不允许重复的数据存储程序时需要提防人为错误，比如在插入算法中检查数组中所有的数据以便保证要插入的数据项关键字的值与已输入的数据不相同。然而这种检查效率不高，而且还会增加插入算法的步数，因此本书中的applet没有进行这种检查。

允许重复值条件下的查找算法

正如上文所说，允许重复将会使查找算法复杂化：即使匹配上了一个，它还得继续寻找可能的匹配，直到最后一个数据项。这只是一种方法而已，当然可以直接在第一次匹配成功后就停止查找。如何实现这个程序就好比是选择“请找到所有蓝眼睛的人”和“请找到一个蓝眼睛的人”。

当选中Dups OK按钮后，applet执行第一种方法，找到所有与查找关键字匹配的数据项。由于该算法需要一直执行至数组的最后，所以它通常需要执行N步。

允许重复值条件下的插入算法

这与数据项不可重复的插入算法完全一致：插入新数据项只需一步。但请注意，即使不允许重复，用户也有可能输入相同的关键字，因此在插入之前需要先检查已有的数据项。

允许重复值条件下的删除算法

允许重复会使删除算法更加复杂，这取决于“删除”是如何定义的。如果它意味着仅删除第一个含有特定值的数据项，那么平均只需要N/2次比较和N/2次移动。这与不允许重复时是一样的。

但是如果删除意味着删掉每一个含有特定值的数据项的话，那么同样的操作可能需要多次。这需要检查N个数据项和（可能）移动多于N/2个数据项。这个操作的平均时间依赖于重复数据项在整个数组中的分布情况。

本书的applet的删除采用了第二种含义，将所有相同的数据项都删掉。这种操作很复杂，每删除一个数据项都要把它后面的数据项前移。例如，如果有三个数据项被删，那么在最后删除那个数据项后面的所有数据项都需要提前三个单元。为了更好地理解这种操作是如何进行的，请在applet中选择Dups OK并插入三到四个相同的数据项，然后尝试删除它们。

表2.1给出了上述三种操作的平均比较次数和移动次数，首先是不允许重复的情况，然后是允许重复的情况。N是数组中数据项的个数。插入一个数据项算为一次移动。

表 2.1 允许重复与不允许重复的比较

	不允许重复	允许重复
查找	N/2 次比较	N 次比较
插入	无比较，一次移动	无比较，一次移动
删除	N/2 次比较，N/2 次移动	N 次比较，多于 N/2 次移动

可以通过 Array 专题 applet 来查验这些可能出现的情况。

通常认为 N 和 N/2 之间的差异不很重要，除非正在对程序进行微调以达到最优。正如我们会在本章结尾处讨论的一样，更重要的是操作是否需要执行一步、N 步、log(N)步或 N^2 步。

不要太匆忙

当使用 Array applet 时需要注意的一件重要事情是：算法的缓慢但有条不紊的本质特性。除了插入之外，其他算法都涉及到了一些或所有数组中的数据项。另一些数据结构有更快(但更复杂)的算法。本章中将介绍一种更快的算法，即有序数组中的二分查找，当然书中还有其他一些快而复杂的算法。

Java 中数组的基础知识

上一节以图形化的方式演示了数组中使用的主要算法。下面将介绍如何编写执行这些算法的程序，但首先谈一些 Java 中数组的基础知识。

如果你是一位 Java 专家，可以直接进入下一节，但即使 C 和 C++程序员都必须将本节读完。Java 中的数组的语法与 C 和 C++一样(与其他语言相差也并不大)，但在 Java 的方法中仍然有一些独特的方面。

创建数组

正如第 1 章“综述”中提到的，Java 中有两种数据类型：基本类型（如 int 和 double）和对象类型。在许多编程语言中（甚至有些面向对象语言，如 C++），数组也是基本类型，但在 Java 中把它们当作对象来对待，因此在创建数组时必须使用 new 操作符：

```
int[] intArray;              // defines a reference to an array
intArray = new int[100];     // creates the array, and
                             // sets intArray to refer to it
```

或使用等价的单语句声明的方法：

```
int[] intArray = new int[100];
```

[]操作符对于编译器来说是一个标志，它说明正在命名的是数组对象而不是普通的变量。当然还可以通过另一种语法来使用这个操作符，将它放在变量名的后面，而不是类型后面：

```
int intArray[] = new int[100];  // alternative syntax
```

但是将[]放在 int 后面会清楚地说明[]是数据类型的一部分，而不是变量名的一部分。

由于数组是一个对象，所以它的名字（前面程序中的 intArray）是数组的一个引用；它并不是数组本身。数组存储在内存中的其他地址中，而 intArray 仅仅保存着这个地址。

数组有一个 length 字段，通过它可以得知当前数组大小（数据项的个数）：

```
int arrayLength = intArray.length;   // find array size
```

正如大多数编程语言一样，一旦创建数组，数组大小便不可改变。

访问数组数据项

数组数据项通过使用方括号中的下标数来访问。这与其他语言类似：

```
temp = intArray[3];  // get contents of fourth element of array
intArray[7] = 66;    // insert 66 into the eighth cell
```

请注意无论是在 C、C++，还是 Java 中，第一个数据项的下标都是 0，所以一个有 10 个数据项的数组下标是从 0 至 9。

如果访问小于 0 或比数组大小大的数据项，程序会出现 Array Index Out of Bounds（数组下标越界)的运行时错误。

初始化

当创建整型数组之后，如果不另行指定，那么整型数组会自动初始化为空。与 C++不同的是，即使通过方法（函数）来定义数组也是这样的。创建一个对象数组如下：

```
autoData[] carArray = new autoData[4000];
```

除非将特定的值赋给数组的数据项，否则它们一直是特殊的 null 对象。如果尝试访问一个含有 null 的数组数据项，程序会出现 Null Pointer Assignment（空指针赋值）的运行时错误。这主要是为了保证在读取某个数据项之前要先对其赋值。

使用下面的语法可以对一个基本类型的数组初始化，赋入非空值：

```
int[] intArray = { 0, 3, 6, 9, 12, 15, 18, 21, 24, 27 };
```

上面的语句可能简单得令人惊讶，它同时取代了引用声明和使用 new 来创建数组。在大括号中的数据被称为初始化列表。数组大小由列表中数据项的个数决定。

数组例子

下面是一些演示数组应用的示例程序。先介绍一个老式的面向过程的版本，然后再介绍一个可以达到同样效果的面向对象的方法。清单 2.1 是老式版本，叫做 array.java。

清单 2.1 array.java 程序

```
// array.java
// demonstrates Java arrays
// to run this program: C>java arrayApp
////////////////////////////////////////////////////////////////
class ArrayApp
   {
   public static void main(String[] args)
      {
      long[] arr;                  // reference to array
      arr = new long[100];         // make array
      int nElems = 0;              // number of items
```

```
      int j;                        // loop counter
      long searchKey;               // key of item to search for
//--------------------------------------------------------------
      arr[0] = 77;                  // insert 10 items
      arr[1] = 99;
      arr[2] = 44;
      arr[3] = 55;
      arr[4] = 22;
      arr[5] = 88;
      arr[6] = 11;
      arr[7] = 00;
      arr[8] = 66;
      arr[9] = 33;
      nElems = 10;                  // now 10 items in array
//--------------------------------------------------------------
      for(j=0; j<nElems; j++)       // display items
         System.out.print(arr[j] + " ");
      System.out.println("");
//--------------------------------------------------------------
      searchKey = 66;               // find item with key 66
      for(j=0; j<nElems; j++)           // for each element,
         if(arr[j] == searchKey)        // found item?
            break;                      // yes, exit before end
      if(j == nElems)                   // at the end?
         System.out.println("Can't find " + searchKey); // yes
      else
         System.out.println("Found " + searchKey);      // no
//--------------------------------------------------------------
      searchKey = 55;               // delete item with key 55
      for(j=0; j<nElems; j++)           // look for it
      if(arr[j] == searchKey)
         break;
      for(int k=j; k<nElems; k++)       // move higher ones down
         arr[k] = arr[k+1];
      nElems--;                         // decrement size
//--------------------------------------------------------------
      for(j=0; j<nElems; j++)       // display items
         System.out.print( arr[j] + " ");
      System.out.println("");
      } // end main()
   } // end class ArrayApp
```

在这个程序中创建了一个名为 arr 的数组，并赋进 10 个数据（运动员的号码），在所有数据项中查找 66（Louisa，游击手）并在屏幕上显示所有的数据项，删除 55（Freddy，他去看牙），随后显示了剩下的 9 个数据项。程序的输出如下：

```
77 99 44 55 22 88 11 0 66 33
Found 66
77 99 44 22 88 11 0 66 33
```

在这个数组中存储的数据类型是 long 型。使用 long 为的是表明这是数据，而 int 型被用来表示下标。为简化编码我们选择简单类型。通常数据结构中存储的数据项包含有好几个字段，所以应该由对象而不是简单类型来代表它们。本章的结尾处有一个这样的例子。

插入

向数组中插入一个数据项很容易，这里使用了常规的数组语法：

```
arr[0] = 77;
```

用变量 nElems 记录已经插入数据项的个数。

查找

变量 searchKey 保存了待查找的值。在查找过程中，用 searchKey 一个一个地与数组中的数据项比较。如果循环变量 j 变化到最后一个数据项，但是仍旧没有匹配上，这个值就不在数组中。屏幕上会显示相关消息：Found 66 或 Can't find 27。

删除

删除从查找特定的数据项开始。为了简化问题，可以假设（有可能太草率了）)该数据项在数组中。找到该数据项后，向前移动所有下标比它大的数据项来填补删除后留下的“洞”并将 nElems 减一。在一个实际的程序中，如果没有找到要删除的数据项的话，还要采取一些相关的操作。

显示

将所有的数据项显示出来是很简单的：逐步读取 arr[j]数组的每个数据项，然后将它显示出来。

程序结构

array.java 的结构中有些部分还是需要改进的。程序中只有一个类 ArrayApp，这个类只有一个方法 main()。array.java 实际上是一个老式的面向过程的程序。使之更加对象化可以让程序更加易懂。

下面将通过两个步骤逐步地介绍面向对象的方法。首先将数据存储结构（数组）从程序中分离出来。程序的其他部分成为使用这个结构的用户。第二步是改进存储结构和用户之间的通信。

将程序划分成类

清单 2.1 中的 array.java 程序实际上包含了一个大的方法。通过将程序划分成类以后，可以得到许多好处。分成什么类？数据存储结构本身就是类，程序中使用这个结构的部分也是类。通过将程序划分成两个类，可以使程序的功能清晰，使之更加容易设计和理解（在实际的程序中更加容易修改和维护）。

在 array.java 中使用了一个数组作为数据存储结构，但只把它简单地看成是一个语言成分。现在把这个数组封装在一个类中，称作 LowArray。在这个类中还会封装一些使其他类（在这里是 LowArrayApp）可以访问这个数组的方法。这些方法可以提供 LowArray 和 LowArrayApp 之间的联系。

第一个 LowArray 类的设计也许不会完全成功，但它显示出对更好方法的需要。清单 2.2 中的 lowArray.java 给出了这个类。

清单 2.2 lowArray.java 程序

```
// lowArray.java
// demonstrates array class with low-level interface
// to run this program: C>java LowArrayApp
////////////////////////////////////////////////////////////////
class LowArray
   {
   private long[] a;                 // ref to array a
//--------------------------------------------------------------
   public LowArray(int size)         // constructor
      { a = new long[size]; }        // create array
//--------------------------------------------------------------
public void setElem(int index, long value)     // set value
      { a[index] = value; }
//--------------------------------------------------------------
   public long getElem(int index)              // get value
      { return a[index]; }
//--------------------------------------------------------------
   } // end class LowArray
////////////////////////////////////////////////////////////////
class LowArrayApp
   {
   public static void main(String[] args)
      {
      LowArray arr;                // reference
      arr = new LowArray(100);     // create LowArray object
      int nElems = 0;              // number of items in array
      int j;                       // loop variable

      arr.setElem(0, 77);          // insert 10 items
      arr.setElem(1, 99);
      arr.setElem(2, 44);
      arr.setElem(3, 55);
      arr.setElem(4, 22);
      arr.setElem(5, 88);
      arr.setElem(6, 11);
      arr.setElem(7, 00);
      arr.setElem(8, 66);
      arr.setElem(9, 33);
      nElems = 10;                 // now 10 items in array

      for(j=0; j<nElems; j++)      // display items
         System.out.print(arr.getElem(j) + " ");
      System.out.println("");
```

```
      int searchKey = 26;          // search for data item
      for(j=0; j<nElems; j++)            // for each element,
         if(arr.getElem(j) == searchKey) // found item?
            break;
      if(j == nElems)                    // no
         System.out.println("Can't find " + searchKey);
      else                               // yes
         System.out.println("Found " + searchKey);

                                 // delete value 55
      for(j=0; j<nElems; j++)            // look for it
      if(arr.getElem(j) == 55)
         break;
      for(int k=j; k<nElems; k++)        // higher ones down
         arr.setElem(k, arr.getElem(k+1) );
      nElems--;                          // decrement size

      for(j=0; j<nElems; j++)      // display items
         System.out.print( arr.getElem(j) + " ");
      System.out.println("");
      } // end main()
   } // end class LowArrayApp
////////////////////////////////////////////////////////////////
```

lowArray.java 程序的输出与 array.java 类似，不同之处是它在删除关键字值 55 之前尝试查找一个不存在的关键字值(26)：

```
77 99 44 55 22 88 11 0 66 33
Can't find 26
77 99 44 22 88 11 0 66 33
```

LowArray 类和 LowArrayApp 类

lowArray.java 中实际上是将一个普通的 Java 数组封装进 LowArray 类中。类中的数组隐藏了起来，它是私有的，所以只有 LowArray 类中的方法才可以访问它。LowArray 中有三个方法：setElem()和 getElem()，分别用来插入和检索一个数据项；另外是一个构造函数，用来创建一个特定大小的空数组。

另一个类 LowArrayApp 创建一个 LowArray 类的对象并用它存储和操作数据。可以将 LowArray 类想成是工具，LowArrayApp 类是工具的使用者。现在程序被划分成两个各自扮演不同角色的类。这对于编写一个面向对象的程序来说是关键的第一步。

用来存储数据对象的类有时被称作容器类(container class)，例如在 lowArray.java 中的 LowArray 类。通常容器类不仅存储数据，并且提供访问数据的方法和其他诸如排序等复杂的操作。

类接口

上文已经介绍了如何将一个程序划分成若干不同的类。这些类如何相互通信呢？类之间的责任

分配是面向对象编程的重要方面。

当一个类有许多不同的用户时这一点尤其重要。通常一个类可以被不同的用户(或相同的用户)一次又一次地用于不同的目的。例如有人可能在其他程序中使用 LowArray 类存储旅行支票的序列号。这个类可以处理这种任务，正如同它能存储棒球队员号码一样。

如果一个类被许多程序员使用，那么这个类必须设计得容易使用才行。类的用户通过类接口的方式与类相连。由于类的字段经常是私有的，所以当我们讨论接口时，经常是指类的方法，它们是用来干什么的和它们的参数是什么。通过调用这些方法，类用户与类对象进行交互。面向对象编程最重要的优点之一是类的接口可以设计得尽可能方便且高效。图 2.3 是对 LowArray 接口的一个形象的解释。

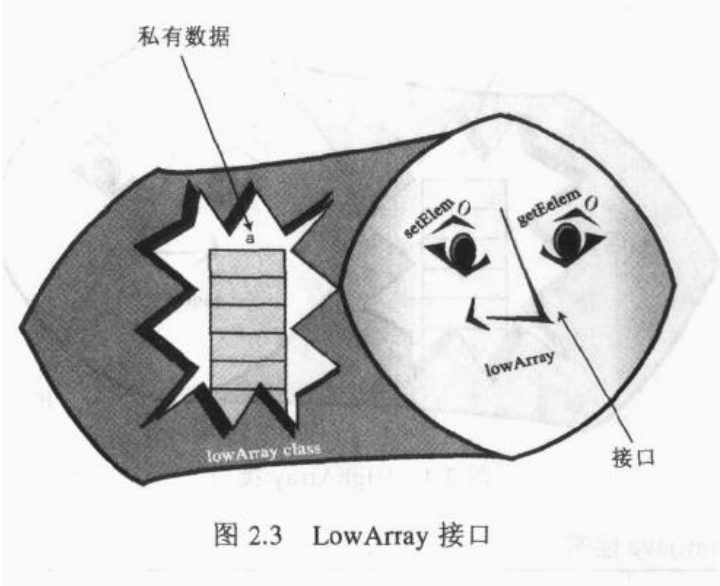


图 2.3 LowArray 接口

不那么方便

lowArray.java 中的 LowArray 类的接口并不是特别方便。setElem()和 getElem()方法还是在低层次构思，它们与普通 Java 数组中的[]操作符没什么区别。LowArrayApp 类中的 main()方法所代表的类用户执行与 array.java 中那个过程性版本中的数组同样低级的操作。惟一的不同是它使用 setElem()和 getElem 而不是[]操作符。这种方法看起来也没有改进多少。

显示数组的内容同样没有简便的方法。LowArrayApp 类只是简单地使用了 for 循环和 getElem()方法来达到这个目的。这本可以通过为 LowArrayApp 写一个单独的显示数组内容的方法来避免重复，但真的就应该由 LowArrayApp 类来承担这个责任吗？

所以 lowArray.java 只演示了如何将一个程序划分成类，但它并没有带给我们太多的实际价值。下面要做的是重新分配类之间的责任，从而可以获得更多的 OOP 的好处。

各自应当担负的责任

在 lowArray.java 程序中，LowArrayApp 类中的 main()函数即数据存储结构的使用者必须知道数组的下标。但有些数组的用户只需要随机访问数组数据项，不太需要知道数组的下标，这种思路可能更有道理。例如，在下一章介绍的数组的排序就可以有效地使用这种直接现成的方法。

然而在一个普通的程序中，数据结构的用户不会意识到访问数组下标是有用或有关系的。

highArray.java 程序示例

下面一个名为 HighArray 的程序给出了数据结构类的一个改进的接口。类用户（HighArrayApp 类）使用这个接口就不再需要考虑下标了。它取消了 setElem()和 getElem()方法，取而代之的是 insert()、find()和 delete()。因为由类来负责处理下标问题，所以这些方法不再需要将下标当作参数。类（HighArrayApp）用户可以集中精力于做什么而不是怎么做：什么要被插入、删除和访问，而不是如何执行这些操作。

图 2.4 给出了 HighArray 接口，清单 2.3 给出了 highArray.java 程序。

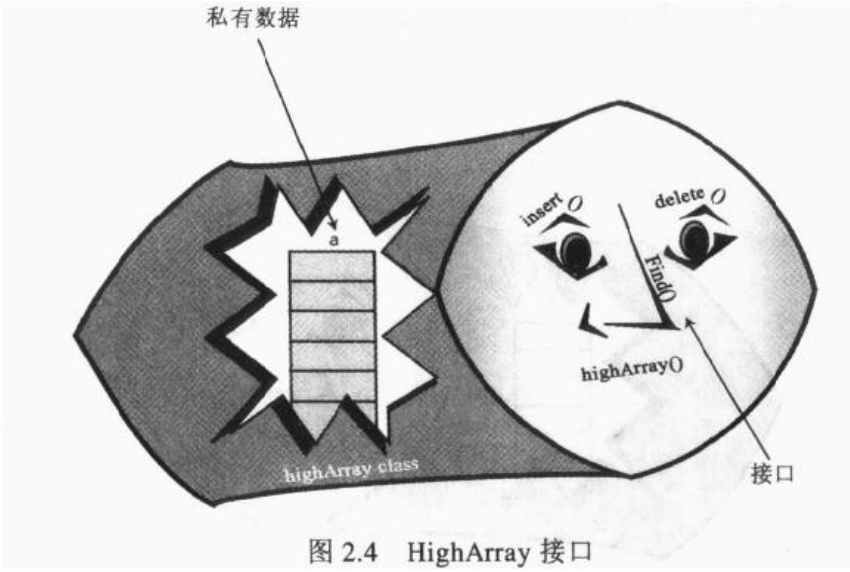


图 2.4 HighArray 接口

清单 2.3 highArray.java 程序

```
// highArray.java
// demonstrates array class with high-level interface
// to run this program: C>java HighArrayApp
////////////////////////////////////////////////////////////////
class HighArray
   {
   private long[] a;                 // ref to array a
   private int nElems;               // number of data items
   //-----------------------------------------------------------
   public HighArray(int max)         // constructor
      {
      a = new long[max];                 // create the array
      nElems = 0;                        // no items yet
      }
   //-----------------------------------------------------------
   public boolean find(long searchKey)
      {                              // find specified value
      int j;
```

```
      for(j=0; j<nElems; j++)          // for each element,
         if(a[j] == searchKey)         // found item?
            break;                     // exit loop before end
      if(j == nElems)                  // gone to end?
         return false;                 // yes, can't find it
      else
         return true;                  // no, found it
      } // end find()
   //-------------------------------------------------------
   public void insert(long value)    // put element into array
      {
      a[nElems] = value;             // insert it
      nElems++;                      // increment size
      }
   //-------------------------------------------------------
   public boolean delete(long value)
      {
      int j;
      for(j=0; j<nElems; j++)        // look for it
         if( value == a[j] )
            break;
      if(j==nElems)                  // can't find it
         return false;
      else                           // found it
         {
         for(int k=j; k<nElems; k++) // move higher ones down
            a[k] = a[k+1];
         nElems--;                   // decrement size
         return true;
         }
      } // end delete()
   //-------------------------------------------------------
   public void display()             // displays array contents
      {
      for(int j=0; j<nElems; j++)       // for each element,
         System.out.print(a[j] + " ");  // display it
      System.out.println("");
      }
   //-------------------------------------------------------
   } // end class HighArray
////////////////////////////////////////////////////////////////
class HighArrayApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;             // array size
```

```
    HighArray arr;                 // reference to array
    arr = new HighArray(maxSize); // create the array

    arr.insert(77);                // insert 10 items
    arr.insert(99);
    arr.insert(44);
    arr.insert(55);
    arr.insert(22);
    arr.insert(88);
    arr.insert(11);
    arr.insert(00);
    arr.insert(66);
    arr.insert(33);

    arr.display();                 // display items

    int searchKey = 35;            // search for item
    if( arr.find(searchKey) )
       System.out.println("Found " + searchKey);
    else
       System.out.println("Can't find " + searchKey);

    arr.delete(00);                // delete 3 items
    arr.delete(55);
    arr.delete(99);

    arr.display();                 // display items again
    }  // end main()
  }  // end class HighArrayApp
////////////////////////////////////////////////////////////////
```

数组现在被封装在 HighArray 类中。main()中创建了一个数组，并执行了与 lowArray.java 中几乎完全相同的操作：插入 10 个数据项，查找一个不存在的数据项，显示数组的内容。由于删除操作十分容易，程序删除了三个数据项(0、55 和 99)而不是一个，最后再次显示数组内容。下面是程序的输出：

```
77 99 44 55 22 88 11 0 66 33
Can't find 35
77 44 22 88 11 66 33
```

请留意 main()是多么的短小简单。在 lowArray.java 中必须由 main()处理的细节现在被 HighArray 类中的方法解决了。

HighArray 类中的 find()方法用数据项的值作为参数传递，依次查找数组中的每个数据项。它的返回值是 true 或 false，取决于是否找到该数据项。

insert()方法向数组下一个空位置上放置一个新的数据项。一个名为 nElem 的字段跟踪记录着数

组中实际已有的数据项个数。main()方法不再需要为数组中还有多少数据项而担心了。

根据以参数形式传入的关键字，delete()方法查找相应的数据项，当它找到该数据项后，便将所有后面的数据项前移，从而覆盖了待删除的数据项，然后 nElem 减一。

程序中还有一个 display()方法，用来显示数组中所有数据项的值。

更加轻松的用户

在 lowArray.java(清单 2.2)中，main()中查找一个数据项的代码需要八行，而在 highArray.java 中只需要一行。类用户 HighArrayApp 类不需要关心数组的下标或其他细节。令人惊讶的是，类用户甚至不必知道 HighArray 类中使用*何种数据结构*来存储数据。结构被隐藏在接口之后。实际上在下一节将要介绍对于不同的数据结构如何使用这个相同的接口。

抽象

从 what(什么)中将 how(如何)分离出来的过程，即类中的操作如何进行，相对什么是类用户可见的，被称为抽象。抽象是软件工程中重要的方面。把类的功能抽象出来后，可以使程序设计更简单，因为不需要在设计的初期就考虑操作的细节。

Ordered 专题 applet

假设一个数组，其中的数据项按关键字升序排列，即最小值在下标为 0 的单元上，每一个单元都比前一个单元的值大。这种数组被称为有序数组。

当向这种数组中插入数据项时，需要为插入操作找到正确的位置：刚好在稍小值的后面，稍大值的前面。然后将所有比待插数据项大的值向后移以便腾出空间。

为什么要将数据按顺序排列？好处之一就是可以通过二分查找显著地提高查找速度。

根据第 1 章介绍的步骤，启动 Ordered 专题 applet。屏幕上会出现一个与 Array 专题 applet 中类似的数组，但其中的数据是有序的。图 2.5 显示了这个 applet。

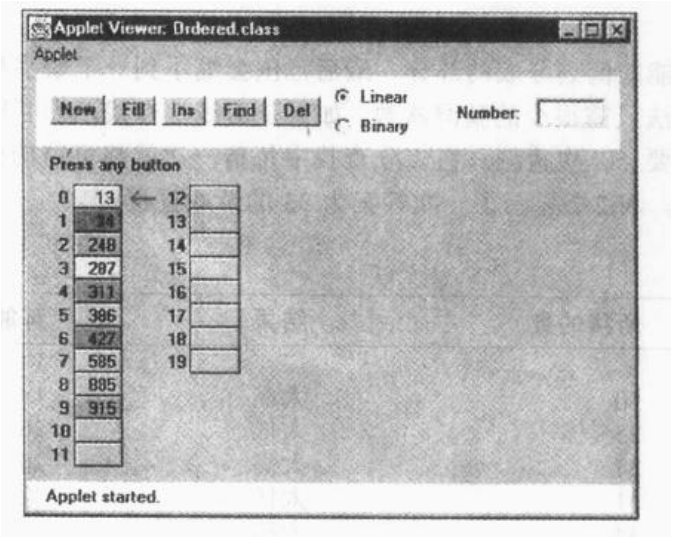


图 2.5 Ordered 专题 applet

在有序数组中预先设定了不允许重复。正如前面所得出的结论，这种选择提高了查找的速度，但降低了插入操作的速度。

线性查找

Ordered 专题 applet 中可以选择两种查找算法：线性查找和二分查找。缺省情况下是线性查找。它同 Arrayapplet 中未经排序的数组的查找操作很相似：红箭头依次向后，寻找匹配。在有序数组中有些不同：当找到一个比待查数据大的数时就退出查找。

试一下线性查找：选中 Linear 单选按钮。使用 Find 按钮查找一个不存在或在数组中部的值。在图 2.5 中可以选择 400 这个数。当箭头到达第一个大于 400 的数据项时，查找结束，即停在图中的 427。由算法可知继续寻找是没有意义的。

同样试一试 Ins 和 Del 按钮。使用 Ins 按钮插入一个能处于数组中部的数据项值，这需要将大于这个数据项的所有数据项都向后移动。

使用 Del 按钮删除一个数组中部的数据项。删除与 Arrayapplet 中的操作一样，向前移动所有比删除数据项下标大的数据项。但在有序数组中如果没有找到待删除的数据项时，删除算法可以在中途退出，就像查找算法一样。

二分查找

当使用二分查找时就体现出有序数组的好处。这种查找比线性查找快很多，尤其是对大数组来说更为显著。

猜数游戏

二分查找使用的方法与我们在孩童时期常玩的猜数游戏中所用的方法一样。在这个游戏里，一个朋友会让你猜她正想的一个 1 至 100 之间的数。当你猜了一个数后，她会告诉你三种选择中的一个：你猜的比她想的大，或小，或猜中了。

为了能用最少的次数猜中，必须从 50 开始猜。如果她说你猜得太小，则推出那个数在 51 至 100 之间，所以下一次猜 75(51 至 100 的一半)。但如果她说有些大，则推出那个数在 1 至 49 之间，所以下一次猜 25。

每猜一次就会将可能的值划分成两部分。最后范围会缩小到一个数字那么大，那就是答案。

请注意运用这种方法只需很少的猜测次数。如果是用线性查找，从 1 开始，然后是 2，3，等等，找到这个数字平均会需要 50 次猜测。在二分查找中每猜一次就将可能的值划分成两部分，因此猜测的次数就会大大减少。表 2.2 显示了一次答案为 33 的游戏过程。

表 2.2 猜数

步数	所猜的数	结果	可能值的范围
0			1～100
1	50	太高	1～49
2	25	太低	26～49
3	37	太高	26～36
4	31	太低	32～36
5	34	太高	32～33
6	32	太低	33～33
7	33	正确	

只需七次就可以猜出正确的答案。这是最大值。如果幸运的话，不需要到最后就能猜出那个数来。例如当答案为 50 或 34 时。

Ordered 专题 applet 的二分查找

若要演示 Ordered 专题 applet 中的二分查找，应先使用 New 按钮创建一个新数组。按完第一下后，程序会提示输入数组大小(最大 60)和查找的方式：线性查找或二分查找。点击 Binary 单选按钮来选择二分查找。当创建完毕后，使用 Fill 按钮来填充数据。请按照提示输入数据(不能超过数组大小)。再按几下 Fill 继续填充。

当数组填好后，选择数组中的一个值，然后使用 Find 按钮来观察如何找到它的位置。开始要按几下，屏幕上会显示一个红色的箭头来指示算法当前查找的值，在单元的旁边会有垂直的蓝条来显示当前的范围。图 2.6 描述了当范围是整个数组时的情景。

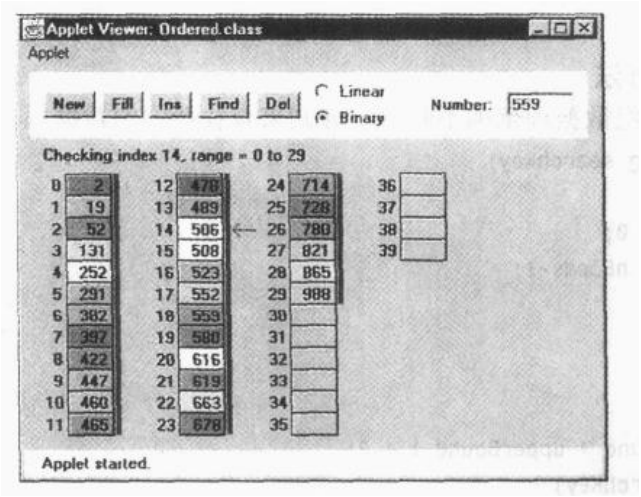


图 2.6 二分查找的初始范围

每按一下 Find 按钮，范围会减小一半，并且选择这个范围的中点处作为新的猜测值。图 2.7 显示了查找过程中的下一步。

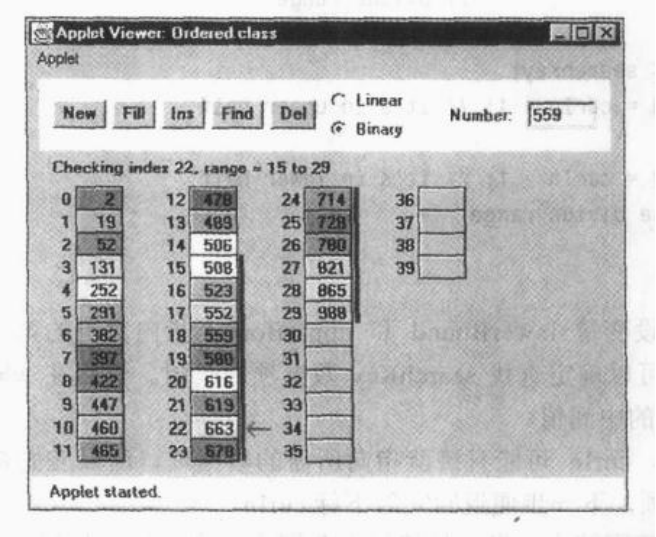


图 2.7 二分查找第二步的范围

即使是数组的最大容量 60 个数据项的范围，最多也只需按六次就可以找到任何数据项。

尝试对不同大小数组使用二分查找。在 applet 运行之前你能够算出所需的步数来吗？在本章的最后一节会回答这个问题。

可以看到插入和删除操作中也可使用二分查找(当它被选中时)，由二分查找来确定数据项插入或删除的位置。在这个 applet 中数据项不允许有重复的值出现。

有序数组的 Java 代码

下面讨论一下有序数组的 Java 代码，它使用 OrdArray 类来封装数组和它的算法。类的核心是 find()方法，通过它可以用二分查找来定位一个特定的数据项。在给出整个程序之前先详细研究一下这个方法。

二分查找中的 find()方法

find()方法通过将数组数据项范围不断对半分割来查找特定的数据项。这个方法如下所示：

```
public int find(long searchKey)
   {
   int lowerBound = 0;
   int upperBound = nElems-1;
int curIn;

while(true)
   {
   curIn = (lowerBound + upperBound ) / 2;
   if(a[curIn]==searchKey)
      return curIn;              // found it
   else if(lowerBound > upperBound)
      return nElems;             // can't find it
   else                          // divide range
      {
      if(a[curIn] < searchKey)
         lowerBound = curIn + 1; // it's in upper half
      else
         upperBound = curIn - 1; // it's in lower half
      } // end else divide range
   } // end while
} // end find()
```

这个方法在一开始设变量 lowerBound 和 upperBound 指向数组的第一个和最后一个非空数据项。通过设置这些变量可以确定查找 searchKey 数据项的范围。然后在 while 循环中，当前的下标 curIn 被设置为这个范围的中间值。

如果足够幸运的话，curIn 可能直接就指向所需的数据项，所以应先查看是否相等。如果是，则意味着找到了该数据项，下一步便返回它的下标 curIn。

循环中的每一步将范围缩小一半。最终这个范围会小到无法再分割。在下一条语句中会判断：

如果 lowerBound 比 upperBound 大，则范围已经不存在了。（当 lowerBound 等于 upperBound，范围是一个数据项所以还需要再一次循环。）当范围不再有效时查找停止，但没有找到所需的数据项，所以返回数据项总数 nElems。由于数组的最后一个非空数据项的下标为 nElem-1，所以下标 nElems 是无效的。类用户把这个值解释为没有找到特定数据项。

如果 curIn 没有指向所需的数据项，但范围仍足够大，此时需要将范围缩小一半。比较当前下标所指的 a[curIn]，即范围中部值与待查数据的值 searchKey。

如果 searchKey 较大，则应该将范围设为当前范围的后半部。因此将 lowerBound 移到 curIn。实际上是将 lowerBound 移到 curIn 后的一个单元，这是由于在循环的开始已经检查了 curIn。

如果 searchKey 比 a[curIn]小，则应将范围设为当前范围的前半部。因此将 upperBound 移到 curIn 的前一个数据项。图 2.8 显示了这两种情况下范围的变化。

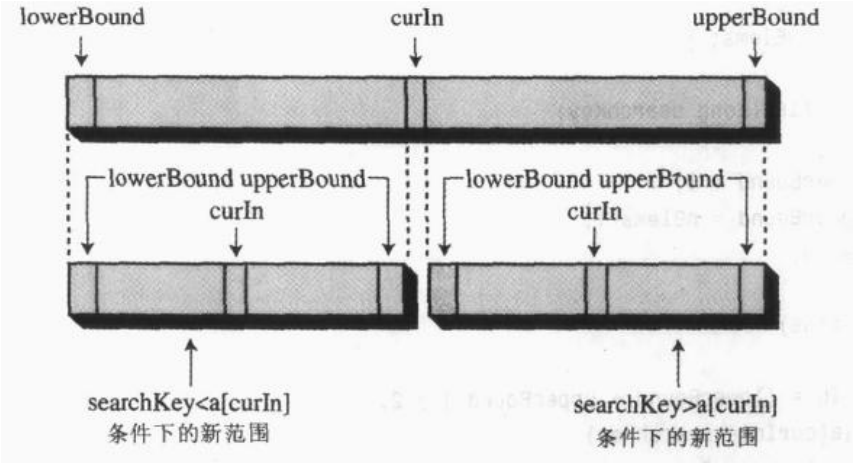


图 2.8 二分查找中的范围割分

OrdArray 类

总之 orderedArray.java 程序同 highArray.java（清单 2.3）类似。但主要差异在于 find()使用了二分查找。

定位新数据项插入的位置也可以使用二分查找。这需要在 find()函数中做一些变化，但为了简单起见，程序保持了 insert()中的线性查找。速度上有些缺陷不重要，这是由于在插入过程中平均一半的数据项都要移动，所以即使用二分查找也不会明显提高 insert()的速度。当然如果追求速度的话，可以将 insert()开始的部分变为二分查找（与 Ordered 专题 applet 相同）。同样，在 delete()方法中也可以调用 find()来找出待删除数据项的位置。

OrdArray 类中还有一个新的 size()方法，它返回数组当前数据项的个数。这对类用户 main()在调用 find()时十分有用。如果 find()返回 nElems，同时 main()通过 size()也得到相同的 nElems，这意味着查找不成功。清单 2.4 给出了 orderArray.java 程序的全部代码。

清单 2.4 orderArray.java 程序

```
// orderedArray.java
// demonstrates ordered array class
// to run this program: C>java OrderedApp
```

```
////////////////////////////////////////////////////////////////
class OrdArray
   {
   private long[] a;                 // ref to array a
   private int nElems;               // number of data items
   //-----------------------------------------------------------
   public OrdArray(int max)          // constructor
      {
      a = new long[max];             // create array
      nElems = 0;
      }
   //-----------------------------------------------------------
   public int size()
      { return nElems; }
   //-----------------------------------------------------------
   public int find(long searchKey)
      {
      int lowerBound = 0;
      int upperBound = nElems-1;
      int curIn;

      while(true)
         {
         curIn = (lowerBound + upperBound ) / 2;
         if(a[curIn]==searchKey)
            return curIn;              // found it
         else if(lowerBound > upperBound)
            return nElems;             // can't find it
         else                          // divide range
            {
            if(a[curIn] < searchKey)
               lowerBound = curIn + 1; // it's in upper half
            else
               upperBound = curIn - 1; // it's in lower half
            } // end else divide range
         } // end while
      } // end find()
   //-----------------------------------------------------------
   public void insert(long value)    // put element into array
      {
      int j;
      for(j=0; j<nElems; j++)        // find where it goes
         if(a[j] > value)            // (linear search)
            break;
      for(int k=nElems; k>j; k--)    // move bigger ones up
```

```
        a[k] = a[k-1];
      a[j] = value;                  // insert it
      nElems++;                      // increment size
      }  // end insert()
   //--------------------------------------------------------
   public boolean delete(long value)
      {
      int j = find(value);
      if(j==nElems)                  // can't find it
         return false;
      else                           // found it
         {
         for(int k=j; k<nElems; k++) // move bigger ones down
            a[k] = a[k+1];
         nElems--;                   // decrement size
         return true;
         }
      }  // end delete()
   //--------------------------------------------------------
   public void display()             // displays array contents
      {
      for(int j=0; j<nElems; j++)       // for each element,
         System.out.print(a[j] + " ");  // display it
      System.out.println("");
      }
   //--------------------------------------------------------
   }  // end class OrdArray
////////////////////////////////////////////////////////////////
class OrderedApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;             // array size
      OrdArray arr;                  // reference to array
      arr = new OrdArray(maxSize);   // create the array

      arr.insert(77);                // insert 10 items
      arr.insert(99);
      arr.insert(44);
      arr.insert(55);
      arr.insert(22);
      arr.insert(88);
      arr.insert(11);
      arr.insert(00);
      arr.insert(66);
```

```
      arr.insert(33);

      int searchKey = 55;              // search for item
      if( arr.find(searchKey) != arr.size() )
         System.out.println("Found " + searchKey);
      else
         System.out.println("Can't find " + searchKey);

      arr.display();                   // display items

      arr.delete(00);                  // delete 3 items
      arr.delete(55);
      arr.delete(99);

      arr.display();                   // display items again
      }  // end main()
   }  // end class OrderedApp
////////////////////////////////////////////////////////////////
```

有序数组的优点

使用有序数组会给我们带来什么好处？最主要的好处是查找的速度比无序数组快多了。不好的方面是在插入操作中由于所有靠后的数据都需移动以腾开空间，所以速度较慢。有序数组和无序数组中的删除操作都很慢，这是因为数据项必须向前移动来填补已删除数据项的洞。

有序数组在查找频繁的情况下十分有用，但若是插入与删除较为频繁时，则无法高效工作。例如，有序数组适合于公司雇员的数据库。雇用和解雇雇员同读取一个已存在雇员的有关信息或更改薪水、住址等信息相比，前两者是不经常发生的。

另一方面，零售商店的存货清单不适合用有序数组来实现，这是由于与频繁的进货和出货相应的插入与删除操作都会执行得很慢。

对 数

在本部分中会介绍用对数来计算二分查找所需的步骤。数学专业的读者可以跳过此小节。如果数学使你感到恼火，请花些时间好好看看表 2.3，然后再跳过这里吧。

我们已经见识到二分查找同线性查找相比在速度上的显著提高。在一个范围是 0~100 的猜数游戏中，二分查找最多只需七步就可以确定范围中的任意一个数；同样在一个有 100 条记录的数组中，通过特定的关键字来查找一条记录只需要七次比较。那么换个范围怎么样？表 2.3 显示了一些有代表性的范围和用二分查找所需的比较次数。

请注意二分查找次数与线性查找次数之间的区别。对于小数目的记录来说，它们之间的差别并不明显。线性查找搜索 10 条记录平均需要五次比较（N/2），二分查找最多需要四次比较。但是记录越多，差异越明显。对于 100 条记录来说，线性查找需要 50 次比较，而二分查找只需要 7 次。对于 1000 条来说，结果是 500 比 10，而 1000000 条记录则是 500000 比 20。由此我们可以总结出，

除了对于特别小的数组以外，二分查找的表现都是极为优秀的。

表 2.3 二分查找所需的比较次数

范围	所需比较次数
10	4
100	7
1000	10
10000	14
100000	17
1000000	20
10000000	24
100000000	27
1000000000	30

方程

可以通过不断地将范围(第一列中的)对分直至小到不可再分为止，由此来验证表 2.3 中的结果。这个过程所需分割的次数就是表中第二列所示的比较次数。

通过不断地将范围除以 2 来求出比较次数是一种算术的方法。有人也会想到用一个简单的方程来求得答案。当然确实有这样一个方程，并且由于在学习数据结构的过程中它会不时地出现，因此研究一下这个方程也是十分有用的。这个公式与对数有关。(稍安勿躁。)

表 2.3 中的数字省掉了一些有趣的数据。这些数字不能回答某些问题，例如，在五步之内可以查找的最大范围是多少？为了解决这个问题，首先需要建立一个类似的表，它的范围由 1 开始，然后通过每次乘以 2 逐步建立起来。表 2.4 里显示了前 7 步的情况。

表 2.4 2 的幂

第 s 步，等同于 $\log_2(r)$	范围 r	由 2 的幂表示的范围（2^s）
0	1	2^0
1	2	2^1
2	4	2^2
3	8	2^3
4	16	2^4
5	32	2^5
6	64	2^6
7	128	2^7
8	256	2^8
9	512	2^9
10	1024	2^{10}

对于我们最初提到的 100 这个范围，通过上表可以看出 6 步不能提供足够大的范围（64），但 7

步就可以轻易地涵盖这个范围（128）。因此表 2.3 中所示 100 个数据项需 7 步是正确的，同样可以验证范围 1000 需要 10 步。

每次对范围加倍可以创建出一个数列，它是 2 的幂，与表 2.4 中的第三列所示一样。设 s 表示步数（乘 2 的次数，即 2 的幂），r 表示范围，则有下面的方程

$r = 2^s$

如果已知步数 s，通过这个方程可以得出范围 r。例如，如果 s 是 6，则范围便是 2^6，即 64。

2 的指数函数的反函数

我们最初的问题同上面所提到的正好相反：已知范围，求出完成搜索的步数，即已知的是 r，希望有一个方程可以求出 s。

某数的指数函数的反函数被称作对数。下面是我们所需的式子，由对数来表示：

$s = \log_2(r)$

这个方程说明步数（比较次数）等于以 2 为底数的范围的对数。什么是对数？以 2 为底 r 的对数是为了得到 r 重复乘 2 的次数。在表 2.4 中，第一列显示的数字 s 便等于 $\log_2(r)$。

如何得到一个数的对数而不用做多次的除法？计算器和大多数计算机语言都有 log 功能。它们经常是以 10 为底来求对数，但是通过将结果乘以 3.322 可以轻松地转换成以 2 为底的对数。例如，$\log_{10}(100) = 2$，从而 $\log_2(100) = 2$ 乘以 3.322，即 6.644。四舍五入至 7，这正是表 2.3 中 100 右面一列所示的数字。

无论如何，关键一点并不是在于计算对数。理解一个数与它的对数之间的关系更重要。重新看一下表 2.3，其中比较了查找特定数据项所需的步数和总数据项的个数。每次对数据项个数（范围）乘 10，对于找到其中某个数据项所需的步数只增加了三到四步（实际上是 3.322，在四舍五入之前）。这是由于当一个数变大时，它的对数增长得很慢。当本章讨论到大 O 表示法时，我们会将这个对数的增长速率与其他数学函数进行比较。

存储对象

到现在为止本书的 Java 示例中，数据结构中只存储 long 类型的简单变量。存储这些变量简化了程序，但它对如何在现实世界中使用数据存储结构来说并没有代表性。通常我们存储的数据记录是许多字段的结合。例如一条职员记录需要存储姓、名、年龄、社会保险号等等。对于收藏一套邮票来说，需要存储邮票的发行国家、编目号、状态、现在的价值等等。

在下面的 Java 示例中将给出对象是如何存储的，而不再是简单类型的变量了。

Person 类

在 Java 中，一条数据记录经常由一个类对象来表示。让我们来看一下如何用一个典型的类来存储职员数据。下面是 Person 类的源代码：

```
class Person
   {
   private String lastName;
   private String firstName;
   private int age;
```

```
//--------------------------------------------------------
public Person(String last, String first, int a)
   {                                // constructor
   lastName = last;
   firstName = first;
   age = a;
   }
//--------------------------------------------------------
public void displayPerson()
   {
   System.out.print("   Last name: " + lastName);
   System.out.print(", First name: " + firstName);
   System.out.println(", Age: " + age);
   }
//--------------------------------------------------------
public String getLast()          // get last name
   { return lastName; }
} // end class Person
```

在这个类中只有三个变量，一个人的姓、名和年龄。当然大多数应用程序会包括许多另外的字段。

构造函数创建一个新的 Person 对象并将它的各个字段初始化。displayPerson()方法显示了一个 Person 对象的数据，getLast()方法返回 Person 的姓；这是用于搜索所需的关键字字段。

classDataArray.java 程序

使用 Person 类的程序与存储 long 类型数据项的 highArray.java 程序（清单 2.3）类似。只需要几个变化就可以适应处理 Person 对象。下面是一些比较大的变化：

- 数组类型改为 Person。
- 关键字(姓)现在是一个 String 类对象，因此作比较时需要使用 equals()方法而不是==运算符。Person 中的 getLast()方法可以得到一个 Person 对象的姓，equals()通过下面代码进行比较：

  ```
  if( a[j].getLast().equals(searchName) )  // found item?
  ```

- insert()方法创建一个新的 Person 对象，并把它插入到数组中，而不再是插入一个 long 类型的值。

main()方法有少许改动，大多数是为处理增加的输出。下面：插入 10 条记录，显示它们，查找一条记录，删除 3 条记录，然后再全部显示出来。清单 2.5 给出了完整的 classDataArray.java 程序。

清单 2.5 classDataArray.java 程序

```
// classDataArray.java
// data items as class objects
// to run this program: C>java ClassDataApp
////////////////////////////////////////////////////////////////
class Person
```

```
   {
   private String lastName;
   private String firstName;
   private int age;
//--------------------------------------------------------------
   public Person(String last, String first, int a)
      {                                 // constructor
      lastName = last;
      firstName = first;
      age = a;
      }
//--------------------------------------------------------------
   public void displayPerson()
      {
      System.out.print("   Last name: " + lastName);
      System.out.print(", First name: " + firstName);
      System.out.println(", Age: " + age);
      }
//--------------------------------------------------------------
   public String getLast()          // get last name
      { return lastName; }
   } // end class Person
////////////////////////////////////////////////////////////////
class ClassDataArray
   {
   private Person[] a;              // reference to array
   private int nElems;              // number of data items

   public ClassDataArray(int max)   // constructor
      {
      a = new Person[max];              // create the array
      nElems = 0;                       // no items yet
      }
//--------------------------------------------------------------
   public Person find(String searchName)
      {                                 // find specified value
      int j;
      for(j=0; j<nElems; j++)           // for each element,
         if( a[j].getLast().equals(searchName) )  // found item?
            break;                      // exit loop before end
      if(j == nElems)                   // gone to end?
         return null;                   // yes, can't find it
      else
         return a[j];                   // no, found it
      } // end find()
//--------------------------------------------------------------
```

```
   // put person into array
   public void insert(String last, String first, int age)
      {
      a[nElems] = new Person(last, first, age);
      nElems++;                          // increment size
      }
//--------------------------------------------------------------
   public boolean delete(String searchName)
      {                                 // delete person from array
      int j;
      for(j=0; j<nElems; j++)           // look for it
         if( a[j].getLast().equals(searchName) )
            break;
      if(j==nElems)                     // can't find it
         return false;
      else                              // found it
         {
         for(int k=j; k<nElems; k++)    // shift down
            a[k] = a[k+1];
         nElems--;                      // decrement size
         return true;
         }
      } // end delete()
//--------------------------------------------------------------
   public void displayA()            // displays array contents
      {
      for(int j=0; j<nElems; j++)       // for each element,
         a[j].displayPerson();          // display it
      }
//--------------------------------------------------------------
   } // end class ClassDataArray
////////////////////////////////////////////////////////////////
class ClassDataApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;             // array size
      ClassDataArray arr;            // reference to array
      arr = new ClassDataArray(maxSize);  // create the array
                                     // insert 10 items
      arr.insert("Evans", "Patty", 24);
      arr.insert("Smith", "Lorraine", 37);
      arr.insert("Yee", "Tom", 43);
      arr.insert("Adams", "Henry", 63);
      arr.insert("Hashimoto", "Sato", 21);
```

```
      arr.insert("Stimson", "Henry", 29);
      arr.insert("Velasquez", "Jose", 72);
      arr.insert("Lamarque", "Henry", 54);
      arr.insert("Vang", "Minh", 22);
      arr.insert("Creswell", "Lucinda", 18);

      arr.displayA();                 // display items

      String searchKey = "Stimson";  // search for item
      Person found;
      found=arr.find(searchKey);
      if(found != null)
         {
         System.out.print("Found ");
         found.displayPerson();
         }
      else
         System.out.println("Can't find " + searchKey);

      System.out.println("Deleting Smith, Yee, and Creswell");
      arr.delete("Smith");            // delete 3 items
      arr.delete("Yee");
      arr.delete("Creswell");

      arr.displayA();                 // display items again
      }  // end main()
   }  // end class ClassDataApp
////////////////////////////////////////////////////////////////
```

Here's the output of this program:

```
   Last name: Evans, First name: Patty, Age: 24
   Last name: Smith, First name: Lorraine, Age: 37
   Last name: Yee, First name: Tom, Age: 43
   Last name: Adams, First name: Henry, Age: 63
   Last name: Hashimoto, First name: Sato, Age: 21
   Last name: Stimson, First name: Henry, Age: 29
   Last name: Velasquez, First name: Jose, Age: 72
   Last name: Lamarque, First name: Henry, Age: 54
   Last name: Vang, First name: Minh, Age: 22
   Last name: Creswell, First name: Lucinda, Age: 18
Found    Last name: Stimson, First name: Henry, Age: 29
Deleting Smith, Yee, and Creswell
   Last name: Evans, First name: Patty, Age: 24
   Last name: Adams, First name: Henry, Age: 63
   Last name: Hashimoto, First name: Sato, Age: 21
```

```
Last name: Stimson, First name: Henry, Age: 29
Last name: Velasquez, First name: Jose, Age: 72
Last name: Lamarque, First name: Henry, Age: 54
Last name: Vang, First name: Minh, Age: 22
```

classDataArray.java 程序显示了数据存储结构可以通过与简单类型同样的方法来处理类对象。（请注意在一个正式的程序中使用姓作为关键字时必须处理重姓的问题，正如前面所讨论的一样，这样程序会更加复杂。）

大 O 表示法

汽车按尺寸被分为若干类：微型，小型，中型等等。在不提及具体尺寸的情况下，这些分类可以为我们所涉及到车的大小提供一个大致概念。我们同样也需要一种快捷的方法来评价计算机算法的效率。在计算机科学中，这种粗略的度量方法被称作“大 O”表示法。

在比较算法时似乎应该说一些类似“算法 A 比算法 B 快两倍”之类的话，但实际上这类陈述并没有多大意义。为什么？这是由于当数据项个数变化时，对应的比例也会发生根本改变。有可能数据项增加了 50%，则 A 就比 B 快了三倍。或有可能只有一半的数据项，但现在 A 和 B 的速度是相同的。我们所需的是一个可以描述算法的速度是如何与数据项的个数相联系的比较。下面是我们目前所见过的算法的大 O 表示。

无序数组的插入：常数

无序数组的插入是我们到现在为止所见过的算法中惟一一个与数组中的数据项个数无关的算法。新数据项总是被放在下一个有空的地方，a[nElems]，然后 nElems 增加。无论数组中的数据项个数 N 有多大，一次插入总是用相同的时间。我们可以说向一个无序数组中插入一个数据项的时间 T 是一个常数 K：

T = K

在现实情况中，插入所需的实际时间（不管是微秒还是其他单位）与以下这些因素有关：微处理器，编译程序生成程序代码的效率，等等。上面等式中的常数 K 包含了所有这些因素。在现实情况中要得到 K 的值，需要测量一次插入所花费的时间。（软件就是为了这个目的而存在的。）K 就等于这个时间。

线性查找：与 N 成正比

在数组数据项的线性查找中，我们已经发现寻找特定数据项所需的比较次数平均为数据项总数的一半。因此设 N 为数据项总数，搜索时间 T 与 N 的一半成正比：

T = K*N/2

同插入一样，若要得到方程中 K 的值，首先需要对某个 N 值（有可能很大）的查找进行计时，然后用 T 来计算 K。当得到 K 后便可对任意 N 的值来计算 T。

将 2 并入 K 可以得到一个更方便的公式。新 K 值等于原先的 K 除以 2。新公式为：

T = K*N

这个方程说明平均线性查找时间与数组的大小成正比。即如果一个数组增大两倍，则所花费的

查找时间也会相应地增长两倍。

二分查找：与 log(N)成正比

同样，我们可以为二分查找制定出一个与 T 和 N 有关的公式：

$T = K*\log_2(N)$

正如前面所提到的，时间 T 与以 2 为底 N 的对数成正比。实际上，由于所有的对数都和其他对数成比例（从底数为 2 转换到底数为 10 需乘以 3.322），我们可以将这个为常数的底数也并入 K。由此不必指定底数：

$T = K*\log(N)$

不要常数

大 O 表示法同上面的公式比较类似，但它省去了常数 K。当比较算法时，并不在乎具体的微处理器芯片或编译器；真正需要比较的是对应不同的 N 值，T 是如何变化的，而不是具体的数字。因此不需要常数。

大 O 表示法使用大写字母 O，可以认为其含义是“order of”(大约是)。我们可以使用大 O 表示法来描述线性查找使用了 O(N)级时间，二分查找使用了 O(log N)级时间。向一个无序数组中的插入使用了 O(1)，或常数级时间。(小括号中是数字 1。)

表 2.5 总结了我们到目前为止讨论过的算法的运行时间。

表 2.5 用大 O 表示法表示运行时间

算法	大 O 表示法表示的运行时间
线性查找	O(N)
二分查找	O(logN)
无序数组的插入	O(1)
有序数组的插入	O(N)
无序数组的删除	O(N)
有序数组的删除	O(N)

图 2.9 显示了一些时间与数据项个数的大 O 关系。通过它我们可以比较不同的大 O 值（非常主观）：O(1)是优秀，O(logN)是良好，O(N)是还可以，$O(N^2)$则差一些了。$O(N^2)$会在本书后面的冒泡排序和其他某些算法中出现。

大 O 表示法的实质并不是对运行时间给出实际值，而是表达了运行时间是如何受数据项个数所影响的。除了实际安装后真正去测量一次算法的运行时间之外，这可能是对算法进行比较的最有意义的方法了。

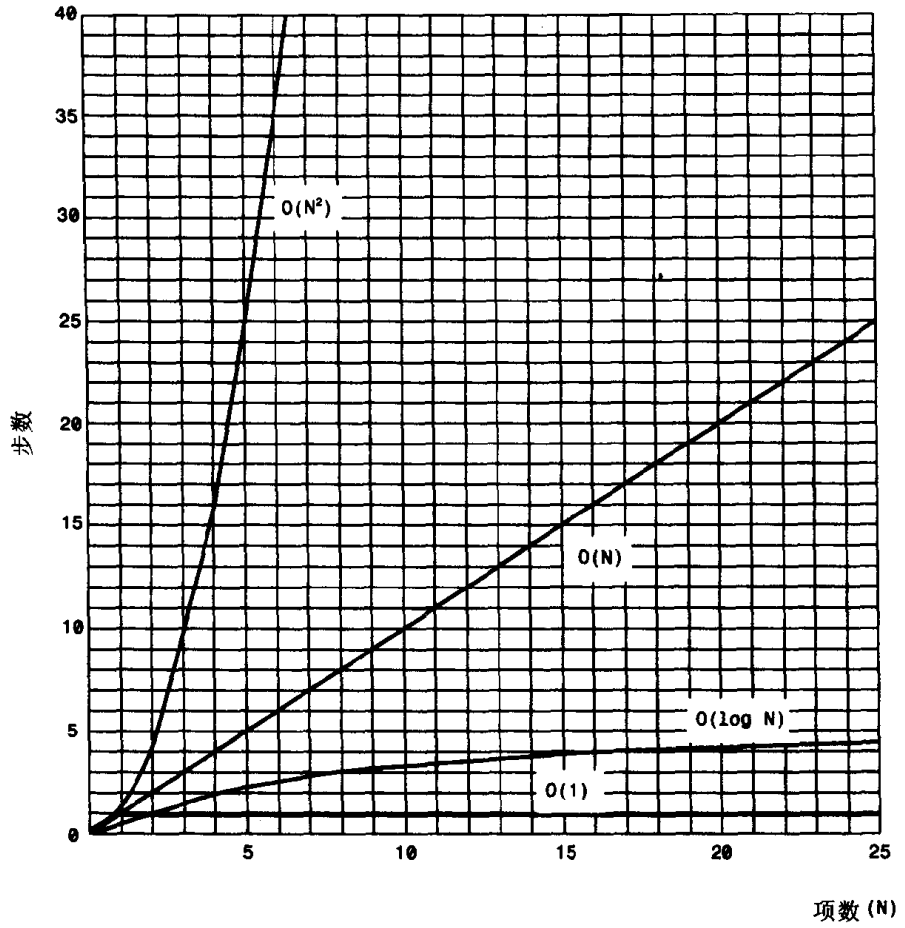


图 2.9 大 O 时间的图

为什么不用数组表示一切？

仅使用数组似乎就可以完成所有工作，为什么不用它们来进行所有的数据存储呢？我们已经见到了许多关于数组的缺点。在一个无序数组中可以很快进行插入（O(1)时间），但是查找却要花费较慢的 O(N)时间。在一个有序数组中可以查找得很快，用 O(logN)的时间，但插入却花费了 O(N)时间。对于这两种数组而言，由于平均半数的数据项为了填补“空洞”必须移动，所以删除操作平均需要 O(N)时间。

如果有一种数据结构进行任何如插入、删除和查找的操作都很快（理想的 O(1)或是 O(logN)时间），那就好了。在后面章节中可以见到我们离这种理想是多么近，但是这是以程序的复杂度做为代价的。

数组的另一个问题便是它们被 new 创建后，大小尺寸就被固定住了。但通常在开始设计程序时并不知道会有多少数据项将会被放入数组中，所以需要猜它的大小。如果猜的数过大，会使数组中的某些单元永远不会被填充而浪费空间。如果猜得过小，会发生数组的溢出，最好的情况下会向程

序的用户发出警告消息，最坏的情况则会导致程序崩溃。

有些更加灵活的数据结构可以随插入数据项而扩展大小。第5章中讨论的链表就是这样的结构。

Java中有一个被称作Vector的类，使用起来很像数组，但是它可以扩展。这些附加的功能是以效率作为代价的。

你可能想尝试创建自己的向量(vector)类。当类用户使用类中的内部数组将要溢出时，插入算法创建一个大一点的数组，把旧数组中的内容复制到新数组中，然后再插入新数据项。整个过程对于类用户来说是不可见的。

小 结

- Java中的数组是对象，由new操作符创建。
- 无序数组可以提供快速的插入，但查找和删除较慢。
- 将数组封装到类中可以保护数组不被随意更改。
- 类的接口由类用户可访问的方法（有时还有字段）组成。
- 类的接口被设计成使类用户的操作更加简单。
- 有序数组可以使用二分查找。
- 以B为底A的对数（大概）是在结果小于1之前用B除A的次数。
- 线性查找需要的时间与数组中数据项的个数成正比。
- 二分查找需要的时间与数组中数据项的个数的对数成正比。
- 大O表示法为比较算法的速度提供了一种方便的方法。
- O(1)级时间的算法是最好的，O(logN)次之，O(N)为一般，$O(N^2)$最差。

问 题

下列问题作为读者的自测题。答案可见附录C。

1. 向一个无序数组中插入一个数据项
   a. 费时与数组的大小成正比。
   b. 需要多次比较。
   c. 需要移动其他数据项来提供空间。
   d. 不管已有多少数据项都花费同样的时间。
2. 判断题：当从无序数组中删除数据项时，大多数情况下需要移动其他数据项来填洞。
3. 在无序数组中，允许重复会导致
   a. 所有操作时间都会增加。
   b. 在某些情况下查找时间的增加。
   c. 总会增加插入时间。
   d. 有时会减少插入时间。
4. 判断题：在无序数组中，查找一个不在数组中的数据项通常要比找到一个数据项快。

5. Java 中创建一个数组需要使用关键字_____。

6. 如果类 A 要使用类 B，那么

a. 类 A 中的方法应该很好理解。

b. 如果类 B 与程序用户交互是更加可取的。

c. 比较复杂的操作都应放在类 A 中。

d. 类 B 能做的操作越多越好。

7. 当类 A 使用类 B 时，可以被类 A 访问的类 B 中的方法和字段被称作是类 B 的_____。

8. 与无序数组相比，有序数组

a. 删除更快。

b. 插入更快。

c. 创建更快。

d. 查找更快。

9. 对数是_____的反函数。

10. 以 10 为底 1000 的对数是_____。

11. 在一个含有 200 个数据项的数组中完成二分查找所需检查的最大数据项个数是

a. 200。

b. 8。

c. 1。

d. 13。

12. 以 2 为底 64 的对数是_____。

13. 判断题：以 2 为底 100 的对数是 2。

14. 大 O 表示法表示了

a. 算法的速度是如何与数据项的个数相关的。

b. 含有给定大小的数据结构的算法的运行时间。

c. 含有给定数据项数目的算法的运行时间。

d. 数据结构的大小是如何与数据项的个数相关的。

15. O(1)意味着一个操作执行了_____的时间。

16. 简单类型变量和_____都可以存入数组中。

实 验

完成这些实验可以帮助深入理解本章的主题。不需要编程实现。

1. 使用 Array 专题 applet 进行插入、查找和删除数据项的操作。确保在点击前能预知执行的过程。在允许重复值和不允许重复值的情况下都做一遍。

2. 请事先想好 Ordered 专题 applet 每步选择的范围。

3. 在一个含有偶数个数据项的数组中是没有中间数据项的。二分查找算法会先检查哪个数据项呢？使用 Ordered 专题 applet 来验证你的想法。

编程作业

编程作业有助于巩固对本章内容的理解，并展示如何应用本章的概念。（在“简介”中提到过，有资格证明的教师可以从出版者的网站上得到编程作业的完整答案。）

2.1 向 highArray.java 程序（清单 2.3）的 HighArray 类添加一个名为 getMax()的方法，它返回数组中最大关键字的值，当数组为空时返回-1。向 main()中添加一些代码来使用这个方法。可以假设所有关键字都是正数。

2.2 修改编程作业 2.1 中的方法，使之不仅返回最大的关键字，而且还将该关键字从数组中删除。将这个方法命名为 removeMax()。

2.3 编程作业 2.2 中的 removeMax()方法提供了一种通过关键字值进行数组排序的方法。实现一个排序方案，要求不修改 HighArray 类，只需对 main()中的代码进行修改。这个方法需要第二个数组，在排序结束时数组数据项是逆序排列的。（这个方法是第 3 章“简单排序”中选择排序的一个变体。）

2.4 修改 orderedArray.java 程序（清单 2.4）使 insert()、delete()与 find()方法一样都使用二分查找，正如书中所建议的那样。

2.5 向 orderedArray.java 程序（清单 2.4）的 OrdArray 类加入一个 merge()方法，使之可以将两个有序的源数组合并成一个有序的目的数组。在 main()中添加代码，向两个源数组中插入随机数，调用 merge()方法，并将结果目的数组显示出来。两个源数组的数据项个数可能不同。在算法中需要先比较源数组中的关键字，从中选出最小的一个数据项复制到目的数组。同时还要考虑如何解决当一个源数组的数据项已经取完而另一个还剩一些数据项的情况。

2.6 向 highArray.java 程序（清单 2.3）的 HighArray 类中加入一个 noDup()的方法，使之可以将数组中的所有重复数据项删除。即如果数组中有三个数据项的关键字为 17，noDup()方法会删除其中的两个。不必考虑保持数据项的顺序。一种方法是先用每一个数据项同其他数据项比较，并用 null（或是一个不会用在真正的关键字中的特殊值）将重复的数据项覆盖掉。然后将所有的 null 删除，当然还要缩小数组的大小。

第3章 简单排序

本章重点

- 如何排序?
- 冒泡排序
- 选择排序
- 插入排序
- 对象排序
- 简单排序的比较

一旦建立了一个重要的数据库后，就可能根据某些需求对数据进行不同方式的排序。比如对姓名按字母序排序，对学生按年级排序，对顾客按邮政编码排序，对国内销售品按价格排序，对城市按人口增长率排序，对国家按国民生产总值排序，以及对恒星按大小排序等等。

对数据进行排序有可能是检索的一个初始步骤。正如在第二章“数组”中看到的那样，二分查找法比线性查找法要快得多，然而它只能应用于有序的数据。

由于排序非常重要而且可能非常耗时，所以它已经成为一个计算机科学中广泛研究的课题，而且人们的确已经研究出一些非常成熟的方法。在这一章中，能看到三个比较简单的算法：冒泡排序、选择排序和插入排序。每一个算法都通过自己的一个专题 applet 来演示说明。在第 7 章“高级排序”中，将介绍一些更为复杂的排序方法：希尔排序和快速排序。

本章描述的算法虽然比较简单，执行速度也相对慢一些，但是仍然值得学习。因为这些简单排序算法除了比较容易理解之外，在某些情况下比那些复杂的算法实际上还要好一些。比如，对于小规模的文件以及基本有序的文件，插入排序算法能比快速排序算法更为有效。实际上，插入排序通常也作为快速排序算法实现的一部分。

这一章中的示例程序建立在前面章节中所开发的各数组类的基础上。在类似的数组类中，排序算法作为类的方法来实现。

请读者一定要试运行这一章中的专题 applet。这些 applet 比那些文字和图片更能有效地说明排序算法是如何工作的。

如何排序?

假设让棒球队队员（第 1 章“综述”中提到过）在运动场上排列成一队，如图 3.1 所示。九个正式的队员加一个替补已经站立好，准备练习。现在需要按身高从低到高为队员们排队（最矮的站在左边），给他们照一张集体照。应该怎么排队呢？

在排序这件事上，人与计算机程序相比有以下优势：我们可以同时看到所有的队员，并且可以立刻找出最高的一个，而不用费力地测量和比较每个人的身高。而且，队员们不一定要固守特定的空间，他们可以相互推推搡搡就腾出了位置，还能互相前后站立。经过一些具体的调整，就可以毫不费力地给队员们排好队，如图 3.2 所示。

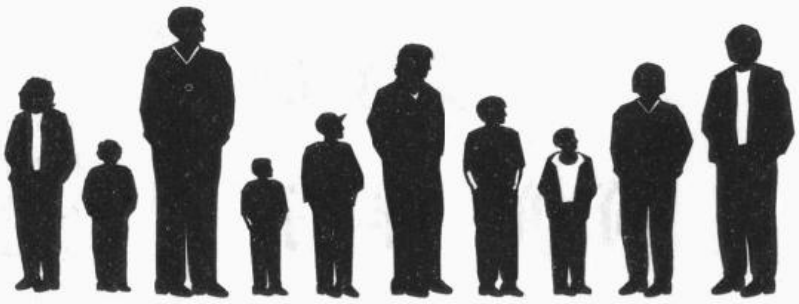


图 3.1 未排序的棒球队

图 3.2 排序后的棒球队

计算机程序却不能像人这样通览所有的数据。它只能根据计算机的“比较”操作原理，在同一时间内对两个队员进行比较。算法的这种“管视”将是一个反复出现的问题。在人类看来很简单的事情，计算机的算法却不能看到全景，因此它只能一步一步地解决具体问题和遵循一些简单的规则。

这一章中的三个算法都包括如下的两个步骤，这两步循环执行，直到全部数据有序为止：

1. 比较两个数据项。
2. 交换两个数据项，或复制其中一项。

但是，每种算法具体实现的细节有所不同。

冒泡排序

冒泡排序算法运行起来非常慢，但在概念上它是排序算法中最简单的，因此冒泡排序算法在刚开始研究排序技术时是一个非常好的算法。

使用冒泡排序算法对棒球队队员排序

如果人近视得像计算机程序一样，以至于只能看到站在他面前的两个相邻的棒球队员。在这种困难的情况下，如何为队员们排队呢？假设有 N 个队员，并且根据所站的位置从左到右分别给每一个队员编号，从 0 到 N-1。

冒泡排序例程执行如下：从队列的最左边开始，比较 0 号位置和 1 号位置的队员。如果左边的队员（0 号）高，就让两个队员交换。如果右边的队员高，就什么也不做。然后右移一个位置，比较 1 号位置和 2 号位置的队员。和刚才一样，如果左边的队员高，则两个队员交换位置。这个排序过程如图 3.3 所示。

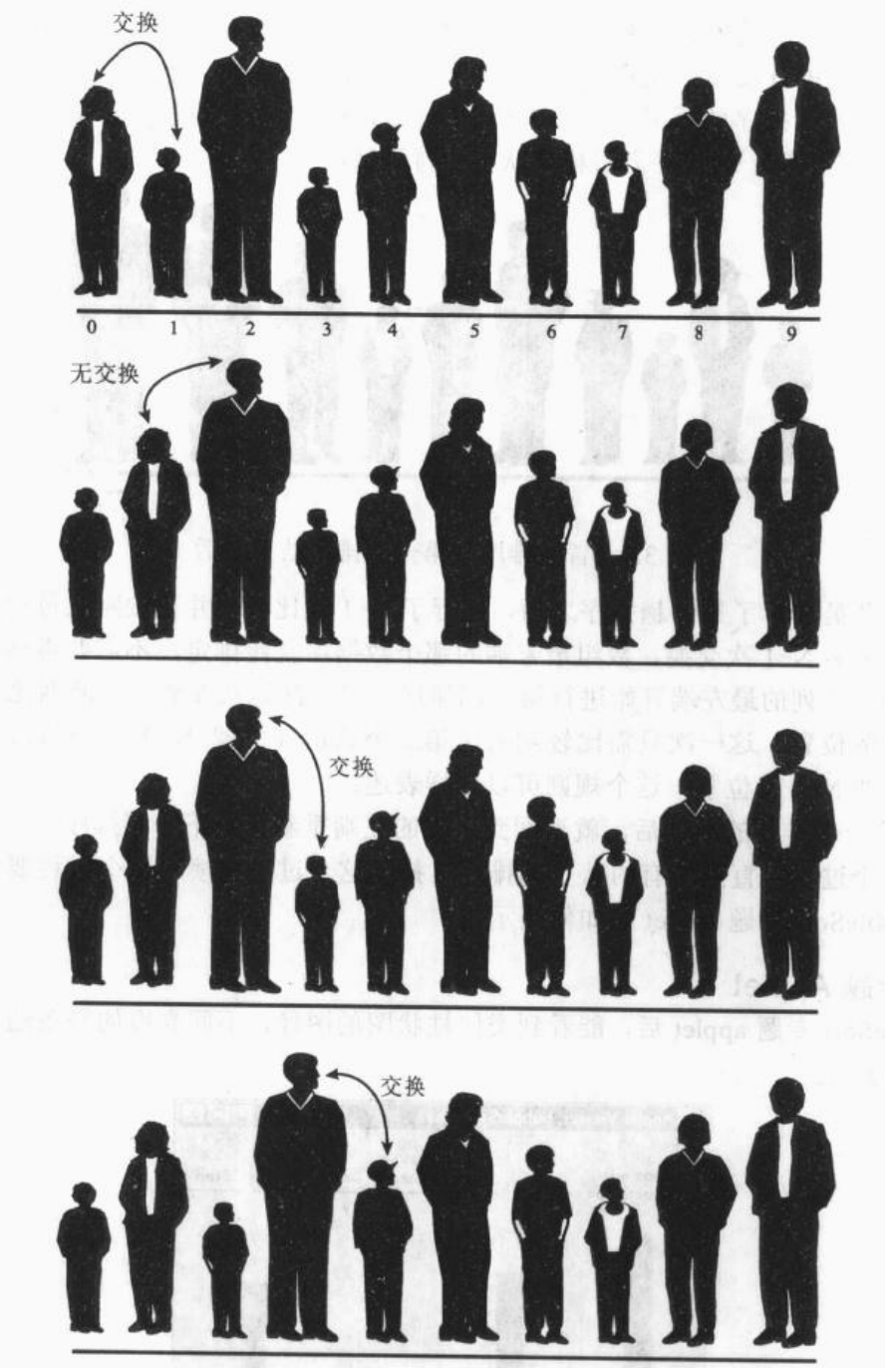


图 3.3 冒泡排序：第一趟排序开始的几个步骤

以下是要遵循的规则：

1. 比较两个队员。
2. 如果左边的队员高，则两队员交换位置。
3. 向右移动一个位置，比较下面两个队员。

沿着这个队列照刚才那样比较下去，一直比较到队列的最右端。虽然还没有完全把所有队员都

排好序，但是最高的队员确实已经被排在最右边了。这是可以确定的，因为在每次比较两个队员的时候，只要遇到最高的球员就会交换他（她）的位置，直到最后他（她）到达队列的最右边。这也是这个算法被称为冒泡排序的原因：因为在算法执行的时候，最大的数据项总是“冒泡”到数组的顶端。图 3.4 显示了在第一趟排序之后棒球队员的排列情况。

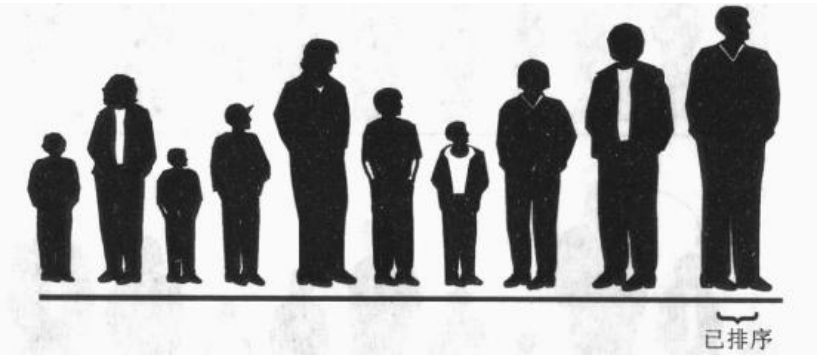


图 3.4 冒泡排序：第一趟排序结束之后

在对所有的队员进行了第一趟排序之后，进行了 N-1 次比较，并且按照队员们的初始位置，进行了最少 0 次，最多 N-1 次交换。数组最末端的那个数据项就此排定，不需要再移动了。

现在重新回到队列的最左端开始进行第二趟排序。再一次地从左到右，两两比较，并且在适当的时候交换队员的位置。这一次只需比较到右边第二个队员（位置 N-2），因为最高的队员已经占据了最后位置，即 N-1 号位置。这个规则可以这样表述：

4. 当碰到第一个排定的队员后，就返回到队列的左端重新开始下一趟排序。

不断执行这个过程，直到所有的队员都排定。描述这个过程比演示这个过程要困难得多，所以还是来看看 BubbleSort 专题 applet 是如何演示的。

BubbleSort 专题 Applet

运行 BubbleSort 专题 applet 后，能看到类似柱状图的图像，不同高度的竖条随机排列在这个画面上，如图 3.5 所示。

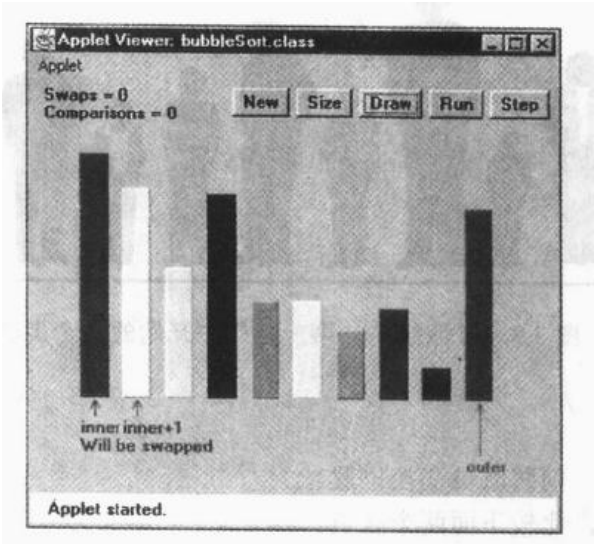


图 3.5 BubbleSort 专题 applet

Run 按钮

这个专题 applet 可以用两种方式运行：自动运行和单步运行。点击 Run 按钮则自动运行这个 applet，可以快速了解冒泡排序算法的全过程。这个算法用冒泡排序的原理对竖条排序。大约 10 秒钟左右，排序完成，竖条按其高度有序排成一排，如图 3.6 所示。

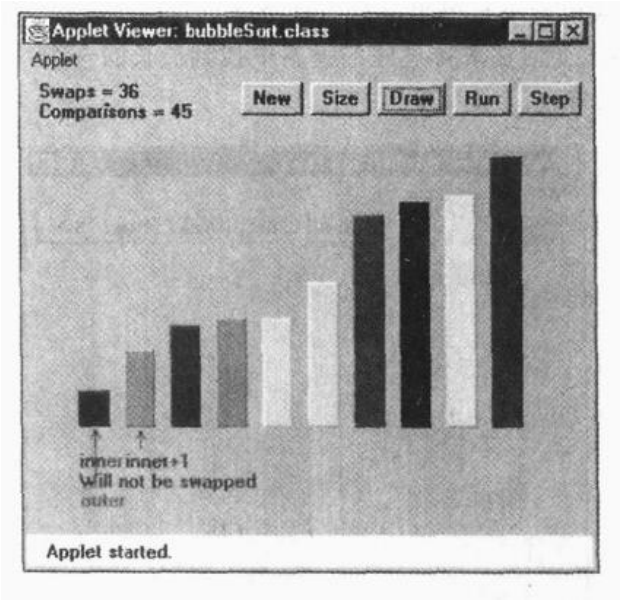


图 3.6 冒泡排序之后

New 按钮

点击 New 按钮开始一次新的排序过程。applet 创建一组新的竖条，并初始化排序例程。反复点击 New 按钮将重复显示竖条的两种初始排列顺序：一种是如图 3.5 所示的随机排列顺序，另一种是逆序的排列顺序，即竖条从高到低排列。对许多排序算法来说逆序的情况是富有挑战性的。

Step 按钮

当单步运行排序算法时，BubbleSort 专题 applet 的好处就真正体现出来了，你可以清楚地看到算法每一步的执行情况。

点击 New 按钮后就可以新建一组随机排列竖条的图，另外还有三个箭头，它们指向不同的竖条。其中有两个相邻的箭头排在左边，一个标为 inner，一个标为 inner+1。另外还有一个箭头，标为 outer，初始时排在最右边。（箭头的命名对应于算法嵌套的循环中使用的内部循环变量和外部循环变量。）

点击一次 Step 按钮，箭头 inner 和 inner+1 一起右移一位，如果符合条件就交换两个竖条。如棒球队的排序问题所述，这两个箭头同步指着两个正在被比较的队员，有可能交换他们的位置。

箭头下面的信息表明箭头 inner 和 inner+1 所指的竖条是否需要交换，信息说明两个正在比较的竖条：如果长的在左边，它们就需要交换位置。图顶部显示的信息说明，到目前为止已经执行过的交换和比较的次数。（10 个竖条的完整排序过程需要 45 次比较和大概平均 22 次的交换。）

继续点击 Step。每一次箭头 inner 和 inner+1 都从 0 开始，逐渐移动到 outer 为止，然后 outer

指针左移一位。在整个排序的过程中，outer 右边的所有竖条都已经排定，而 outer 左边的（包括 outer）竖条则未排过序。

Size 按钮

Size 按钮可以选择所排序的竖条的数量，10 或 100 个。图 3.7 显示了 100 个竖条的情况。除非有特别的耐心，否则一般人可能不想单步执行 100 个竖条的排序过程。那么就点击 Run 按钮，观察蓝色的指针 inner 和 inner+1 如何在未排过序的竖条中找到最长的一个，并把它右移，准确地放到以前排好序的竖条的左边一位。

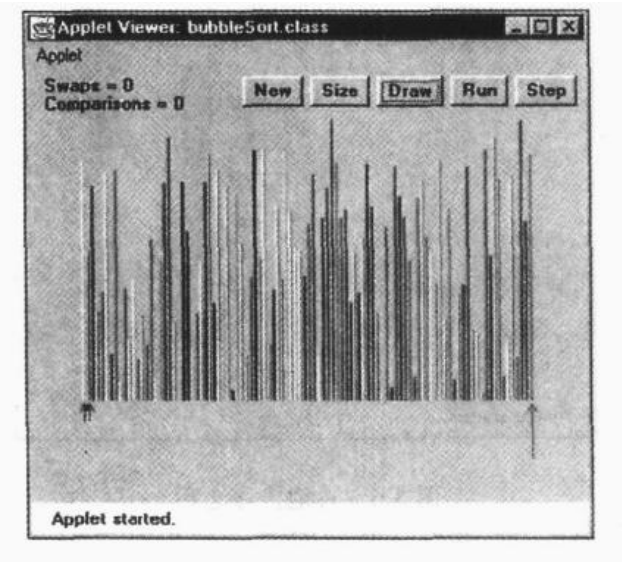


图 3.7 未排序的 100 个竖条

图 3.8 表示了排序过程当中某时刻的情况。红色（最长）箭头右边的竖条都是有序的。左边的竖条看着大致有序，但离真正排好序还有很多工作要做。

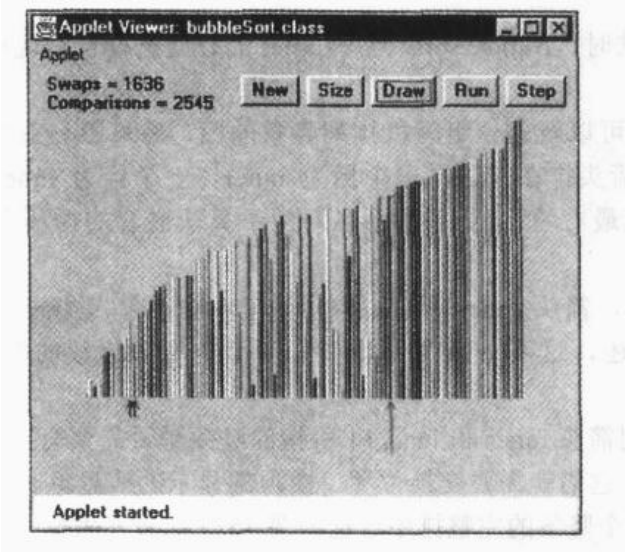


图 3.8 局部有序的 100 个竖条

如果用 Run 按钮启动一个排序，箭头会不停地移来移去，可以在任何时候点击 Step 按钮来暂停这个过程。可以接着单步运行来看操作的细节，或者再次点击 Run 按钮，回到高速运行的模式。

Draw 按钮

计算机全速运行排序算法时，可能要抽出时间来执行一些其他的工作。这就可能会导致有些竖条没有被显示出来。当这种情况发生的时候，可以点击 Draw 按钮来重画所有的竖条。这么做会导致程序暂停运行，所以需要再次点击 Run 按钮来使程序继续运行。

在显示出现小问题的时候，可以随时点击 Draw 按钮。

冒泡排序的 Java 代码

清单 3.1 列出了 bubbleSort.java 的程序，其中在 ArrayBub 类里封装了一个 long 类型的数组 a[]。

在更复杂的程序中，数据很可能由对象组成，这里为了简单起见，使用的是基本数据类型。（在清单 3.4 的 objectSort.java 程序中会看到如何将对象排序。）另外，尽管 ArrayBub 类的 find()和 delete()方法是这个类的一部分，为了减少书中程序代码的长度，就不把它们列出来了。

清单 3.1 bubbleSort.java 程序

```
// bubbleSort.java
// demonstrates bubble sort
// to run this program: C>java BubbleSortApp
////////////////////////////////////////////////////////////////
class ArrayBub
   {
   private long[] a;                 // ref to array a
   private int nElems;               // number of data items
//--------------------------------------------------------------
   public ArrayBub(int max)          // constructor
      {
      a = new long[max];                 // create the array
      nElems = 0;                        // no items yet
      }
//--------------------------------------------------------------
   public void insert(long value)    // put element into array
      {
      a[nElems] = value;             // insert it
      nElems++;                      // increment size
      }
//--------------------------------------------------------------
   public void display()             // displays array contents
      {
      for(int j=0; j<nElems; j++)       // for each element,
         System.out.print(a[j] + " ");  // display it
      System.out.println("");
      }
//--------------------------------------------------------------
   public void bubbleSort()
```

```
      {
      int out, in;

      for(out=nElems-1; out>1; out--)   // outer loop (backward)
         for(in=0; in<out; in++)        // inner loop (forward)
            if( a[in] > a[in+1] )       // out of order?
               swap(in, in+1);          // swap them
      } // end bubbleSort()
//--------------------------------------------------------------
   private void swap(int one, int two)
      {
      long temp = a[one];
      a[one] = a[two];
      a[two] = temp;
      }
//--------------------------------------------------------------
   } // end class ArrayBub
////////////////////////////////////////////////////////////////
class BubbleSortApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;            // array size
      ArrayBub arr;                 // reference to array
      arr = new ArrayBub(maxSize);  // create the array

      arr.insert(77);               // insert 10 items
      arr.insert(99);
      arr.insert(44);
      arr.insert(55);
      arr.insert(22);
      arr.insert(88);
      arr.insert(11);
      arr.insert(00);
      arr.insert(66);
      arr.insert(33);

      arr.display();                // display items

      arr.bubbleSort();             // bubble sort them
      arr.display();                // display them again
      } // end main()
   } // end class BubbleSortApp
////////////////////////////////////////////////////////////////
```

这个类的构造函数，以及 insert()和 display()方法与见过的那些方法是类似的，但是这里多了一

个新方法：bubbleSort( )。当 main()调用这个方法时，数组中的数据就会排列成有序。

在 main()中，往数组中随机插入 10 个数据项，显示数组，然后调用 bubbleSort()来对其排序，接着再一次显示数组。下面是输出结果：

```
77 99 44 55 22 88 11 0 66 33
0 11 22 33 44 55 66 77 88 99
```

bubbleSort( )方法只有 4 行。从清单中抽取出来，如下所示：

```
public void bubbleSort()
   {
   int out, in;

   for(out=nElems-1; out>1; out--)    // outer loop (backward)
      for(in=0; in<out; in++)         // inner loop (forward)
         if( a[in] > a[in+1] )        // out of order?
            swap(in, in+1);           // swap them
   } // end bubbleSort()
```

这个算法的思路是要将最小的数据项放在数组的最开始（数组下标为 0），并将最大的数据项放在数组的最后（数组下标为 nElems-1）。外层 for 循环的计数器 out 从数组的最后开始，即 out 等于 nElems-1，每经过一次循环 out 减一。下标大于 out 的数据项都已经是排好序的了。变量 out 在每完成一次内部循环（计数器为 in）后就左移一位，因此算法就不再处理那些已经排好序的数据了。

内层 for 循环计数器 in 从数组的最开始算起，即 in=0，每完成一次内部循环体加一，当它等于 out 时结束一次循环。在内层 for 循环体中，数组下标为 in 和 in+1 的两个数据项进行比较，如果下标为 in 的数据项大于下标为 in+1 的数据项，则交换两个数据项。

为了清晰起见，使用了一个独立的 swap()方法来执行交换操作。它只是交换数组中的两个数据项的值，使用一个临时变量来存储第一个数据项的值，然后把第二项的值赋给第一项，之后让第二项的值等于临时变量的值。实际上，使用一个独立的 swap()方法不一定好，因为方法调用会增加一些额外的消耗。如果写自己使用的排序程序，最好将交换操作这段代码直接放到程序中，这样可以提高一些速度。

不变性

在许多算法中，有些条件在算法执行时是不变的。这些条件被称为不变性。认识不变性对理解算法是有用的。在一定的情况下它们对调试也有用；可以反复地检查不变性是否为真，如果不是的话就标记出错。

在 bubbleSort.java 程序中，不变性是指 out 右边的所有数据项为有序。在算法的整个运行过程中这个条件始终为真。（在第一趟排序开始前，尚未排序，因为 out 开始时在数据项的最右边，没有数据项在 out 的右边。）

冒泡排序的效率

通过考察 10 个竖条的 BubbleSort 专题 applet 可以看到，箭头 inner 和 inner+1 在第一趟排序时进行了 9 次比较，第二趟排序进行了 8 次比较，如此类推，直到最后一趟进行了一次比较。对于 10 个数据项，就是

9+8+7+6+5+4+3+2+1=45

一般来说，数组中有 N 个数据项，则第一趟排序中有 N-1 次比较，第二趟中有 N-2 次，如此类推。这种序列的求和公式如下：

(N-1)+(N-2)+(N-3)+...+1=N*(N-1)/2

当 N 为 10 时，N*(N-1)/2 等于 45(10*9/2)。

这样，算法作了约 $N^2/2$ 次比较（忽略减 1，不会有很大差别，特别是当 N 很大时）。

因为两个竖条只有在需要时才交换，所以交换的次数少于比较的次数。如果数据是随机的，那么大概有一半数据需要交换，则交换的次数为 $N^2/4$。（不过在最坏的情况下，即初始数据逆序时，每次比较都需要交换。）

交换和比较操作次数都和 N^2 成正比。由于常数不算在大 O 表示法中，可以忽略 2 和 4，并且认为冒泡排序运行需要 $O(N^2)$时间级别。运行 100 个竖条的 BubbleSort 专题 applet 可以证实，这种排序算法的速度是很慢的。

无论何时，只要看到一个循环嵌套在另一个循环里，例如在冒泡排序和本章中的其他排序算法中，就可以怀疑这个算法的运行时间为 $O(N^2)$级。外层循环执行 N 次，内部循环对于每一次外层循环都执行 N 次（或者几分之 N 次）。这就意味着将大约需要执行 N*N 或者 N^2 次某个基本操作。

选择排序

选择排序改进了冒泡排序，将必要的交换次数从 $O(N^2)$减少到 O(N)。不幸的是比较次数仍保持为 $O(N^2)$。然而，选择排序仍然为大记录量的排序提出了一个非常重要的改进，因为这些大量的记录需要在内存中移动，这就使交换的时间和比较的时间相比起来，交换的时间更为重要。（一般来说，在 Java 语言中不是这种情况，Java 中只是改变了引用位置，而实际对象的位置并没有发生改变。）

用选择排序算法对棒球队员排序

让我们再来考虑棒球队排队的问题。在选择排序中，不再只比较两个相邻的球员。因此，需要记录下某一个指定队员的高度；可以使用记事本写下指定队员的身高，同时还需要准备好一条紫红色的毛巾。

简述

进行选择排序就是把所有的队员扫描一趟，从中挑出（或者说选择，这正是这个排序名字的由来）最矮的一个队员。最矮的这个队员和站在队列最左端的队员交换位置，即站到 0 号位置。现在最左端的队员是有序的了，不需要再交换位置了。注意，在这个算法中有序的队员都排列在队列的左边（较小的下标值），而在冒泡排序中则是排列在队列右边的。

再次扫描球队队列时，就从 1 号位置开始，还是寻找最矮的，然后和 1 号位置的队员交换。这个过程一直持续到所有的队员都排定。

更详细的描述

排序从球员队列的最左边开始。在记录本上记录下最左端球员的身高，并且把紫红色的毛巾放在这个队员的前面。于是开始用下一个球员的身高和记录本上记录的值相比较。如果这个队员更矮，则划掉第一个球员的身高，记录下第二个队员的身高；同时移动毛巾，把它放在这个新的“最矮的”（当前的）队员前面。继续沿着队列走下去，每一个队员都和记录本上的最小值进行比较。当发现

更矮的队员时，就更新记录本上的最小值并且移动毛巾。做完这些事情后，毛巾就会落在最矮的队员前面。

这个最矮的队员和队列最左边的队员交换位置。现在已经对一个队员排好了序，这期间做了N-1次比较，但是只进行了一次交换。

在下一趟排序中，所做的事情是一模一样的，只是要完全忽略最左边队员的存在，因为他已经排定了。因此算法的第二趟排序从位置1而不是从位置0开始。每进行完一趟排序，就多一个队员有序，并被安排在左边，下次再找新的最小值时就可以少考虑一个队员。图3.9显示了这种排序的前三趟过程。

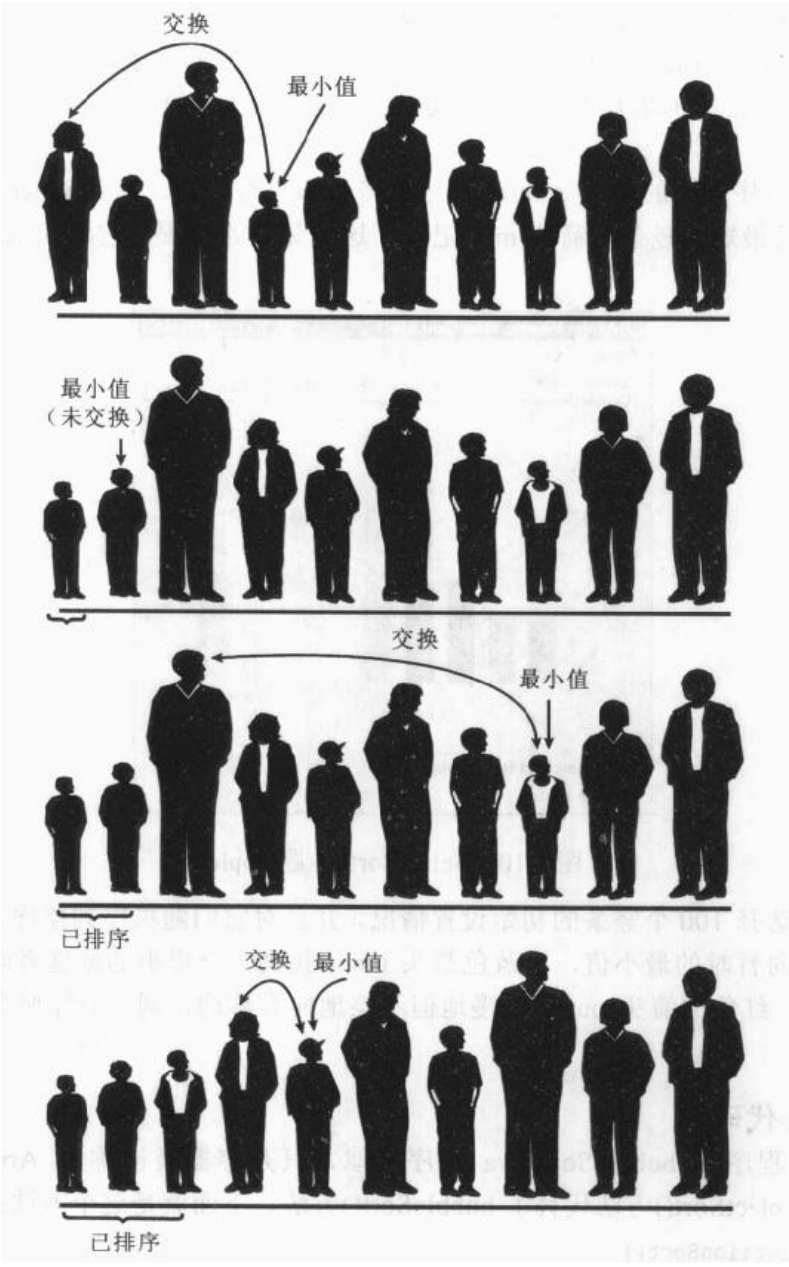


图3.9 棒球队员的选择排序

SelectSort 专题 Applet

为了考察选择排序算法在实际中是如何工作的，可以用 SelectSort 专题 applet。按钮操作和 BubbleSort applet 中是一样。使用 New 按钮创建一个有 10 个随机排列竖条的数组。红色箭头 outer 从左边开始，指向最左边的未排序的竖条。渐渐地，随着越来越多的竖条加入到左端已有序的竖条队列中，红色箭头 outer 不断地右移。

紫红色的箭头 min 开始也指向最左端的竖条，它会移到当前找到的最短的竖条上。（紫红色的箭头 min 对应于对棒球队排队时使用的紫红色毛巾。）蓝色的箭头 inner 标记当前正在和最小值比较的竖条。

随着反复地按 Step 按钮，inner 从左向右移动，依次检查每一个竖条，并且和 min 所指的竖条进行比较。如果 inner 所指的条更短，min 则指向这个新的更短的竖条。当 inner 到达图中队的最右端时，min 就指向了未排序的最短的竖条。于是这个竖条和 outer 所指的竖条，即最左边的无序竖条交换位置。

图 3.10 表示了排序中途的情况。outer 左边的竖条都是有序的，箭头 inner 从 outer 所指的位置向右进行扫描，寻找最短的竖条。箭头 min 记录了这个竖条的位置，它将和 outer 所指的竖条交换位置。

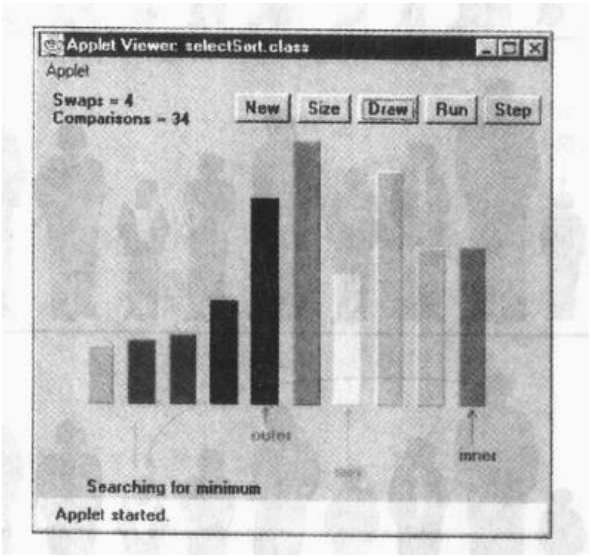


图 3.10 SelectSort 专题 applet

使用 Size 按钮选择 100 个竖条的初始设置情况，并且对它们随机序列排序。可以看到紫红色的箭头 min 是怎样指向暂时的最小值，当蓝色箭头 inner 找到一个更小的候选者时它又是如何跳转指向这个新的竖条的。红色的箭头 outer 缓慢地但不停地向右移动，同时有序竖条在它的左边逐渐积聚增多。

选择排序的 Java 代码

selectSort.java 程序和 bubbleSort.java 程序类似，只是容器类被称为 ArraySel，而不是叫作 ArrayBub，以及用 selectSort()方法代替了 bubbleSort()方法。下面就是这个方法：

```
public void selectionSort()
   {
```

```
   int out, in, min;

   for(out=0; out<nElems-1; out++)   // outer loop
      {
      min = out;                     // minimum
      for(in=out+1; in<nElems; in++) // inner loop
         if(a[in] < a[min] )         // if min greater,
             min = in;               // we have a new min
      swap(out, min);                // swap them
      } // end for(out)
   } // end selectionSort()
```

外层循环用循环变量 out，从数组开头开始（数组下标为 0）向高位增长。内层循环用循环变量 in，从 out 所指位置开始，同样是向右移位。

在每一个 in 的新位置，数据项 a[in]和 a[min]进行比较。如果 a[in]更小，则 min 被赋值为 in 的值。在内层循环的最后，min 指向最小的数据项，然后交换 out 和 min 指向的数组数据项。清单 3.2 显示了完整的 selectSort.java 程序。

清单 3.2 selectSort.java 程序

```
// selectSort.java
// demonstrates selection sort
// to run this program: C>java SelectSortApp
////////////////////////////////////////////////////////////////
class ArraySel
   {
   private long[] a;                 // ref to array a
   private int nElems;               // number of data items
//--------------------------------------------------------------
   public ArraySel(int max)          // constructor
      {
      a = new long[max];                 // create the array
      nElems = 0;                        // no items yet
      }
//--------------------------------------------------------------
   public void insert(long value)    // put element into array
      {
      a[nElems] = value;             // insert it
      nElems++;                      // increment size
      }
//--------------------------------------------------------------
   public void display()             // displays array contents
      {
      for(int j=0; j<nElems; j++)       // for each element,
         System.out.print(a[j] + " ");  // display it
      System.out.println("");
```

```
    }
//--------------------------------------------------------------
  public void selectionSort()
    {
    int out, in, min;

    for(out=0; out<nElems-1; out++)   // outer loop
       {
       min = out;                     // minimum
       for(in=out+1; in<nElems; in++) // inner loop
          if(a[in] < a[min] )         // if min greater,
              min = in;               // we have a new min
        swap(out, min);                // swap them
        } // end for(out)
     } // end selectionSort()
//--------------------------------------------------------------
  private void swap(int one, int two)
     {
     long temp = a[one];
     a[one] = a[two];
     a[two] = temp;
     }
//--------------------------------------------------------------
  } // end class ArraySel
////////////////////////////////////////////////////////////////
class SelectSortApp
  {
  public static void main(String[] args)
     {
     int maxSize = 100;            // array size
     ArraySel arr;                 // reference to array
     arr = new ArraySel(maxSize);  // create the array

     arr.insert(77);               // insert 10 items
     arr.insert(99);
     arr.insert(44);
     arr.insert(55);
     arr.insert(22);
     arr.insert(88);
     arr.insert(11);
     arr.insert(00);
     arr.insert(66);
     arr.insert(33);

     arr.display();                // display items
```

```
        arr.selectionSort();          // selection-sort them

        arr.display();                // display them again
        }  // end main()
    }  // end class SelectSortApp
////////////////////////////////////////////////////////////////
```

selectSort.java 的输出结果和 BubbleSort.java 的输出结果一样，是：

```
77 99 44 55 22 88 11 0 66 33
0 11 22 33 44 55 66 77 88 99
```

不变性

在 selectSort.java 程序中，下标小于或等于 out 的位置的数据项总是有序的。

选择排序的效率

选择排序和冒泡排序执行了相同次数的比较：N*(N-1)/2。对于 10 个数据项，需要 45 次比较。然而，10 个数据项只需要少于 10 次交换。对于 100 个数据项，需要 4950 次比较，但只进行了不到 100 次的交换。N 值很大时，比较的次数是主要的，所以结论是选择排序和冒泡排序一样运行了 O（N^2）时间。但是，选择排序无疑更快，因为它进行的交换少得多。当 N 值较小时，特别是如果交换的时间级比比较的时间级大得多时，选择排序实际上是相当快的。

插入排序

在大多数情况下，插入排序算法是本章描述的基本的排序算法中最好的一种。虽然插入排序算法仍然需要 $O(N^2)$的时间，但是在一般情况下，它要比冒泡排序快一倍，比选择排序还要快一点。尽管它比冒泡排序算法和选择排序算法都更麻烦一些，但它也并不很复杂。它经常被用在较复杂的排序算法的最后阶段，例如快速排序。

用插入排序为棒球队员排序

开始插入排序之前，把棒球队员按随机顺序排成一行。（他们可能急着想打球，但是现在显然没有时间让他们去打）。从排序过程的中间开始，可以更好地理解插入排序，这时队列已经排好了一半。

局部有序

此时，在队伍的中间有一个作为标记的队员。（可以把一件红色运动衫扔到这个队员前面。）在这个作为标记的队员左边的所有队员已经是局部有序的了。这意味着这一部分人之间是按顺序排列的；每个人比他/她左边的人都高。然而这些队员在队列中最终的位置还没有确定，因为当没有被排过序的队员要插入到他们中间的时候，他们的位置还要发生变动。

注意，局部有序在冒泡排序和选择排序中是不会出现的。在这两个算法中，一组数据项在某个时刻是完全有序的；在插入排序中，一组数据仅仅是局部有序的。

被标记的队员

作为标记的队员，称为“被标记”的队员，他和他右边所有的队员都是未排过序的。如图 3.11.a

所示。

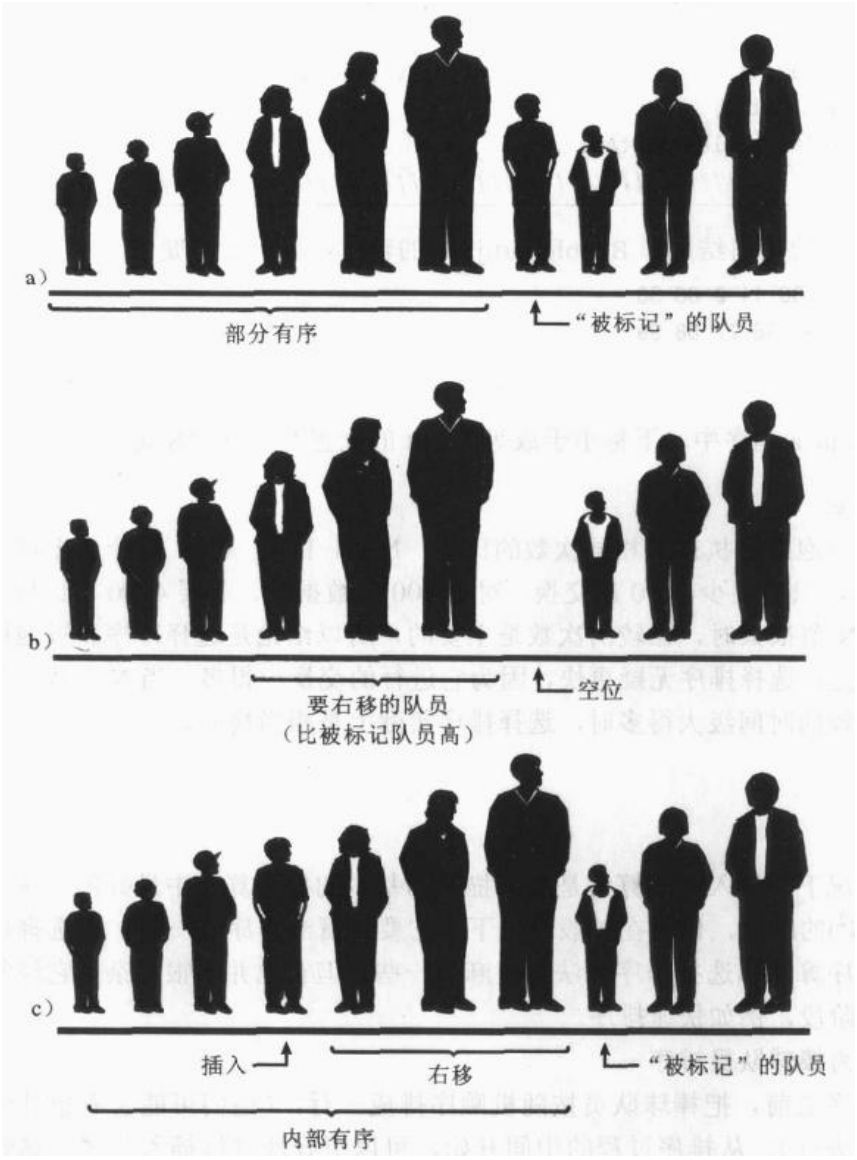


图 3.11 用插入排序为棒球队员排序

下面将要做的是在（局部）有序组中的适当位置插入被标记的队员。然而要做到这一点，需要把部分已排序的队员右移以腾出空间。为了提供移动所需的空间，就先让被标记的队员出列。（在程序中，这个数据项被存储在一个临时变量中。）这个步骤如图 3.11.b 所示。

现在移动已经排过序的队员来腾出空间。将局部有序中最高的队员移动到原来被标记队员所在位置，次高的队员移动到原来最高的队员所在的位置，依此类推。

这个移动过程什么时候结束呢？假设你和被标记的队员一起向队列的左端移动。在每个位置上，队员都向右移动一位，同时被标记的队员和下一个要移动的队员比较身高。当把最后一个比被标记的队员还高的队员移位之后，这个移动的过程就停止了。最后一次移位空出的位置，就是被标记队员应该插入的位置。这个步骤如图 3.11.c 所示。

现在，局部有序的部分里多了一个队员，而未排序的部分里少了一个队员。作为标记的运动衫向右移动一个位置，所以它仍然放在未排序部分的最左边的队员面前。重复这个过程，直到所有未排序的队员都被插入（插入排序由此得名）到局部有序队列中合适的位置。

InsertSort 专题 Applet

使用 InsertSort 专题 applet 来演示插入排序。插入排序 applet 不像其他排序的 applet，它对 100 个随机排列的竖条做插入排序比对 10 个竖条做插入排序可能更有意义。

100 个竖条的排序

按 Size 按钮选成 100 竖条的情况，然后点击 Run 按钮仔细观察这些竖条的自动排序。将会看到一个短小的红色箭头 outer 标记了一条分割线。线的左边是局部有序的竖条，右边是未排序的竖条。蓝色的箭头 inner 从 outer 所指的位置开始，向左迅速移动，寻找要插入的被标记竖条的正确位置。图 3.12 显示了当大约一半的竖条已经成为局部有序时这个过程的情况。

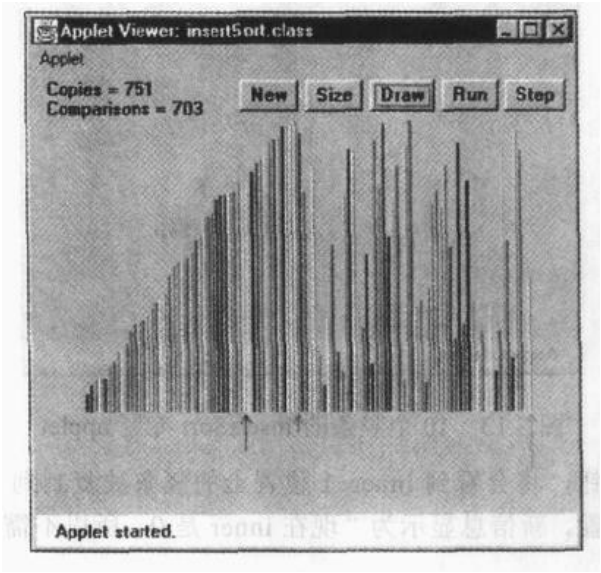


图 3.12　100 个竖条的 InsertSort 专题 applet

被标记的竖条存储在图中最右端，由紫红色箭头所指的临时变量中，但是这个临时变量的内容变化得如此频繁，以至于几乎看不出来其中的值是什么（除非采用单步执行模式把运行速度降下来）。

10 个竖条的排序

为了看到细节，按 Size 按钮转换成 10 竖条的情况。（如果需要，用 New 按钮生成一个新的序列以确保竖条是随机排列的。）

刚开始，inner 和 outer 指向从左边数的第二个竖条（数组下标为 1），第一条信息是“将把 outer 所指的竖条复制到 temp 处”。这将为移位腾出空间。（并没有箭头 inner-1，但显然它总是指示 inner 左边一个竖条。）

点击 Step 按钮。outer 所指的竖条将被复制到 temp 的位置。复制是指把数据项从源地址复制到目的地址。当执行一次复制时，applet 把竖条从原来的位置移走，留下一个空位。这里可能会有点误导，因为在真正的 Java 程序中，对源地址数据的引用不会删除数据。但是界面上显示为空白是为

了更容易看到正在发生的事情。

下一步做什么取决于头两个竖条是否已经是有序的（较短的在左边）。如果是有序的，可以看到信息“已经比较了 inner-1 和 temp 位置的竖条，没有必要复制”。

如果头两个竖条不是有序的，显示信息是“已经比较了 inner-1 和 temp 位置的竖条，将要把 inner-1 位置的竖条复制到 inner 位置”。这是一个必要的移动，它为在 temp 位置的竖条的重新插入腾出位置。在第一趟排序的过程中，只有一次这样的移动；在后续的过程中，将会进行更多的移动。现在的状态如图 3.13 显示。

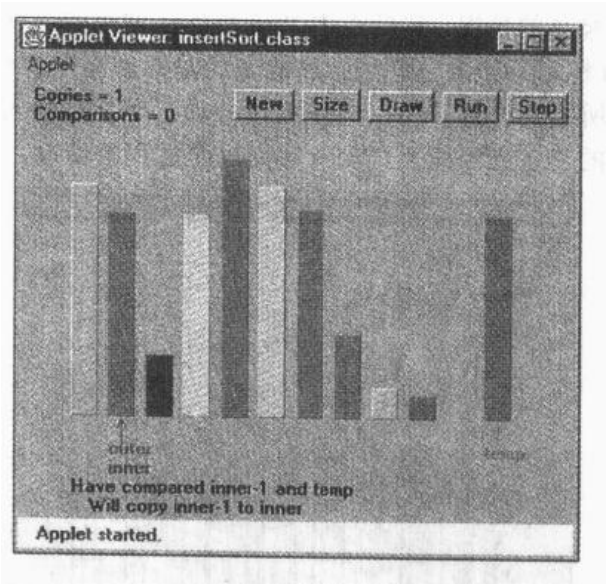


图 3.13 10 个竖条的 InsertSort 专题 applet

接着再次点击 Step 按钮，将会看到 inner-1 位置上的竖条被复制到 inner 位置上。同时，箭头 inner 也向左移动了一个位置。新信息显示为“现在 inner 是 0，所以不需要复制”。整个移动的过程就结束了。

不论头两个条哪个更短，再次点击 Step 按钮会显示信息“把 temp 位置的竖条复制到 inner 位置上”。复制将会发生，但是如果头两个竖条本来就是有序的，可能会怀疑是否发生了一次复制，因为 temp 和 inner 位置是同样长的竖条。把数据复制到相同数据的位置好像是无意义的，但是如果免除这种可能性的检查，算法就能运行得更快。相对来说发生这种情况可能性不会太大。

现在，头两个竖条是局部有序的（两个竖条之间彼此有序），箭头 outer 向右移动一个位置，指向第三个竖条（下标 2）。信息显示为“将把 outer 所指的竖条复制到 temp 处”，并重复前面的过程。在这一趟排序通过已排序的数据时，可能没有移动，可能有一次移动，可能有两次移动，这取决于第三个竖条应插入头两个竖条中的什么位置。

继续单步执行排序，过程执行到一定的程度后，有更多的竖条已经在左边成为局部有序的，就会更容易了解排序中所发生的事情。有序的竖条不断地移动，为 temp 位置的竖条重新插入到正确的位置腾出了空间。

插入排序的 Java 代码

下面是一个执行插入排序的方法，取自 insertSort.java 程序：

```
public void insertionSort()
   {
   int in, out;

   for(out=1; out<nElems; out++)     // out is dividing line
      {
      long temp = a[out];         // remove marked item
      in = out;                      // start shifts at out
      while(in>0 && a[in-1] >= temp) // until one is smaller,
         {
         a[in] = a[in-1];            // shift item right,
      --in;                       // go left one position
      }
   a[in] = temp;                  // insert marked item
   } // end for
} // end insertionSort()
```

在外层的 for 循环中，out 变量从 1 开始，向右移动。它标记了未排序部分的最左端的数据。而在内层的 while 循环中，in 变量从 out 变量开始，向左移动，直到 temp 变量小于 in 所指的数组数据项，或者它已经不能再往左移动为止。while 循环的每一趟都向右移动了一个已排序的数据项。

也许很难看出 InsertSort 专题 applet 中的步骤和代码有什么关系，所以图 3.14 显示了一个 insertionSort()方法的活动图，并且带有在 applet 中显示的相应的文本信息。清单 3.3 显示了完整的 insertSort.java 程序。

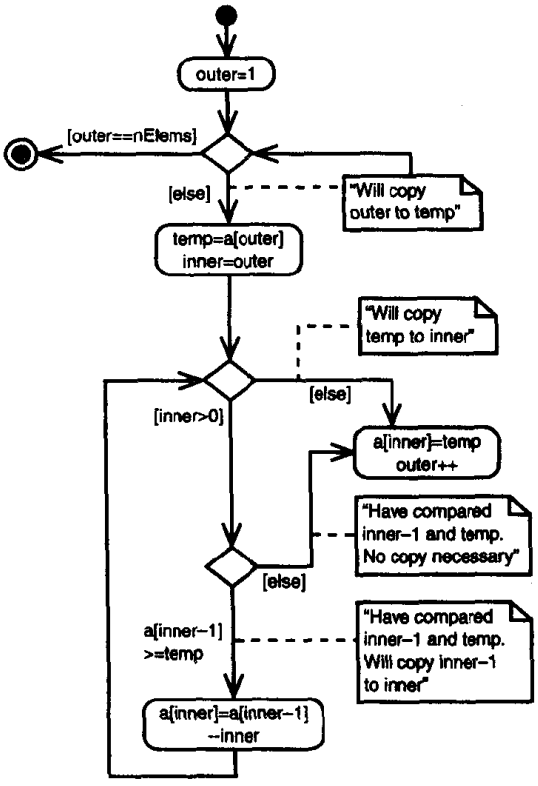


图 3.14 insertSort()的活动图

清单 3.3 insertSort.java 程序

```
// insertSort.java
// demonstrates insertion sort
// to run this program: C>java InsertSortApp
//--------------------------------------------------------------
class ArrayIns
   {
   private long[] a;                 // ref to array a
   private int nElems;               // number of data items
//--------------------------------------------------------------
   public ArrayIns(int max)          // constructor
      {
      a = new long[max];                 // create the array
      nElems = 0;                        // no items yet
      }
//--------------------------------------------------------------
   public void insert(long value)    // put element into array
      {
      a[nElems] = value;             // insert it
      nElems++;                      // increment size
      }
//--------------------------------------------------------------
   public void display()             // displays array contents
      {
      for(int j=0; j<nElems; j++)       // for each element,
         System.out.print(a[j] + " ");  // display it
      System.out.println("");
      }
//--------------------------------------------------------------
   public void insertionSort()
      {
      int in, out;

      for(out=1; out<nElems; out++)     // out is dividing line
         {
         long temp = a[out];            // remove marked item
         in = out;                      // start shifts at out
         while(in>0 && a[in-1] >= temp) // until one is smaller,
            {
            a[in] = a[in-1];            // shift item to right
            --in;                       // go left one position
            }
         a[in] = temp;                  // insert marked item
         }  // end for
      }  // end insertionSort()
```

```
//--------------------------------------------------------------
   } // end class ArrayIns
////////////////////////////////////////////////////////////////
class InsertSortApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;            // array size
      ArrayIns arr;                 // reference to array
      arr = new ArrayIns(maxSize);  // create the array

      arr.insert(77);               // insert 10 items
      arr.insert(99);
      arr.insert(44);
      arr.insert(55);
      arr.insert(22);
      arr.insert(88);
      arr.insert(11);
      arr.insert(00);
      arr.insert(66);
      arr.insert(33);

      arr.display();                // display items

      arr.insertionSort();          // insertion-sort them

      arr.display();                // display them again
      } // end main()
   } // end class InsertSortApp
////////////////////////////////////////////////////////////////
```

这是 insertSort.java 程序的输出结果；它和本章中其他程序的输出结果是相同的：

```
77 99 44 55 22 88 11 0 66 33
0 11 22 33 44 55 66 77 88 99
```

插入排序中的不变性

在每趟结束时，在将 temp 位置的项插入后，比 outer 变量下标号小的数据项都是局部有序的。

插入排序的效率

这个算法需要多少次比较和复制呢？在第一趟排序中，它最多比较一次，第二趟最多比较两次，依此类推。最后一趟最多，比较 N-1 次。因此有

$1 + 2 + 3 + \cdots + N-1 = N*(N-1)/2$

然而，因为在每一趟排序发现插入点之前，平均只有全体数据项的一半真的进行了比较，我们除以 2 得到

$N * (N-1)/4$

复制的次数大致等于比较的次数。然而，一次复制与一次交换的时间耗费不同，所以相对于随机数据，这个算法比冒泡排序快一倍，比选择排序略快。

在任意情况下，和本章其他的排序例程一样，对于随机顺序的数据进行插入排序也需要 $O(N^2)$ 的时间级。

对于已经有序或基本有序的数据来说，插入排序要好得多。当数据有序的时候，while 循环的条件总是假，所以它变成了外层循环中的一个简单语句，执行 N-1 次。在这种情况下，算法运行只需要 O(N)的时间。如果数据基本有序，插入排序几乎只需要 O(N)的时间，这对把一个基本有序的文件进行排序是一个简单而有效的方法。

然而，对于逆序排列的数据，每次比较和移动都会执行，所以插入排序不比冒泡排序快。你可以在 InsertSort 专题 applet 中使用逆序数据选项（点击 New 按钮来选择）来验证这一点。

对象排序

为简单起见，前面应用排序算法时使用的是基本数据类型：long（长整型）。但是，排序例程却更多地应用于对象的排序，而不是对基本数据类型的排序。因此，清单 3.4 显示的 Java 程序，objectSort.java，是对一组 Person 对象进行排序的代码（参考第 2 章中的 classDataArray.java 程序）。

对象排序的 Java 代码

下面程序中的算法是前面提到的插入排序算法。Person 对象以 lastname（姓）为关键字来排序；它就是关键字段。objectSort.java 程序如下所示。

清单 3.4 objectSort.java 程序

```
// objectSort.java
// demonstrates sorting objects (uses insertion sort)
// to run this program: C>java ObjectSortApp
////////////////////////////////////////////////////////////////
class Person
   {
   private String lastName;
   private String firstName;
   private int age;
   //-----------------------------------------------------------
   public Person(String last, String first, int a)
      {                                 // constructor
      lastName = last;
      firstName = first;
      age = a;
      }
   //-----------------------------------------------------------
   public void displayPerson()
      {
      System.out.print("   Last name: " + lastName);
```

```
      System.out.print(", First name: " + firstName);
      System.out.println(", Age: " + age);
      }
   //-------------------------------------------------------
   public String getLast()          // get last name
      { return lastName; }
   } // end class Person
////////////////////////////////////////////////////////////////
class ArrayInOb
   {
   private Person[] a;              // ref to array a
   private int nElems;              // number of data items
//-------------------------------------------------------------
   public ArrayInOb(int max)        // constructor
      {
      a = new Person[max];             // create the array
      nElems = 0;                      // no items yet
      }
//-------------------------------------------------------------
                                   // put person into array
   public void insert(String last, String first, int age)
      {
      a[nElems] = new Person(last, first, age);
      nElems++;                        // increment size
      }
//-------------------------------------------------------------
   public void display()            // displays array contents
      {
      for(int j=0; j<nElems; j++)      // for each element,
         a[j].displayPerson();         // display it
      System.out.println("");
      }
//-------------------------------------------------------------
   public void insertionSort()
      {
      int in, out;

      for(out=1; out<nElems; out++)  // out is dividing line
         {
         Person temp = a[out];       // remove marked person
         in = out;                   // start shifting at out

         while(in>0 &&               // until smaller one found,
               a[in-1].getLast().compareTo(temp.getLast())>0)
            {
```

```
            a[in] = a[in-1];          // shift item to the right
            --in;                     // go left one position
            }
         a[in] = temp;                // insert marked item
         } // end for
      } // end insertionSort()
//--------------------------------------------------------------
   } // end class ArrayInOb
////////////////////////////////////////////////////////////////
class ObjectSortApp
{
   public static void main(String[] args)
      {
      int maxSize = 100;             // array size
      ArrayInOb arr;                 // reference to array
      arr = new ArrayInOb(maxSize);  // create the array

      arr.insert("Evans", "Patty", 24);
      arr.insert("Smith", "Doc", 59);
      arr.insert("Smith", "Lorraine", 37);
      arr.insert("Smith", "Paul", 37);
      arr.insert("Yee", "Tom", 43);
      arr.insert("Hashimoto", "Sato", 21);
      arr.insert("Stimson", "Henry", 29);
      arr.insert("Velasquez", "Jose", 72);
      arr.insert("Vang", "Minh", 22);
      arr.insert("Creswell", "Lucinda", 18);

      System.out.println("Before sorting:");
      arr.display();                 // display items

      arr.insertionSort();           // insertion-sort them

      System.out.println("After sorting:");
      arr.display();                 // display them again
      } // end main()
   } // end class ObjectSortApp
////////////////////////////////////////////////////////////////
```

这个程序的输出结果是：

```
Before sorting:
   Last name: Evans, First name: Patty, Age: 24
   Last name: Smith, First name: Doc, Age: 59
   Last name: Smith, First name: Lorraine, Age: 37
   Last name: Smith, First name: Paul, Age: 37
   Last name: Yee, First name: Tom, Age: 43
```

```
   Last name: Hashimoto, First name: Sato, Age: 21
   Last name: Stimson, First name: Henry, Age: 29
   Last name: Velasquez, First name: Jose, Age: 72
   Last name: Vang, First name: Minh, Age: 22
   Last name: Creswell, First name: Lucinda, Age: 18
After sorting:
   Last name: Creswell, First name: Lucinda, Age: 18
   Last name: Evans, First name: Patty, Age: 24
   Last name: Hashimoto, First name: Sato, Age: 21
   Last name: Smith, First name: Doc, Age: 59
   Last name: Smith, First name: Lorraine, Age: 37
   Last name: Smith, First name: Paul, Age: 37
   Last name: Stimson, First name: Henry, Age: 29
   Last name: Vang, First name: Minh, Age: 22
   Last name: Velasquez, First name: Jose, Age: 72
   Last name: Yee, First name: Tom, Age: 43
```

单词排序

objectSort.java 程序中的 insertSort()方法和 insertSort.java 中的方法类似，只是不再比较基本数据类型，而是比较记录中的 lastname（姓）属性。

用 String 类中的 compareTo()方法执行 insertSort()方法中的比较工作。下面是一个含有这个方法的表达式：

```
a[in-1].getLast().compareTo(temp.getLast()) > 0
```

compareTo()方法根据两个 String 的字典顺序（字母序）返回给调用者不同的整数值，这两个 String 一个是方法的调用者，一个是这个方法的参数，如表 3.1 所示。

表 3.1 compareTo()方法的操作

s2.compareTo(s1)	返回值
s1<s2	<0
s1=s2	0
s1>s2	>0

例如，s1 是“cat”，s2 是“dog”，方法返回值小于 0。objectSort.java 程序中，这个方法用于比较 a[n-1]的 lastname（姓）和 temp 的 lastname（姓）。

稳定性

有些时候，排序要考虑数据项拥有相同关键字的情况。例如，雇员数据按雇员的姓的字典序排序（排序以姓为关键字），现在又想按邮政编码排序，并希望具有相同邮政编码的数据仍然按姓排序。这种情况下，则只需要算法对需要排序的数据进行排序，让不需要排序的数据保持原来的顺序。某些算法满足这样的要求，它们就可以称为稳定的算法。

本章中所有的算法都是稳定的。例如，请注意 objectSort.java 程序（清单 3.4）的输出。有三个人的姓叫做 Smith。开始，顺序为 Doc Smith、Lorraine Smith 和 Paul Smith。排序后，这个顺序还依

然存在，但不同的 Smith 对象已经移动到了各自的新位置上。

几种简单排序之间的比较

除非手边没有算法书可参考，一般情况几乎不太使用冒泡排序算法。它过于简单了，以至于可以毫不费力地写出来。然而当数据量很小的时候它会有些应用的价值。（第 15 章“应用场合”中将讨论“很小”的含义。）

选择排序虽然把交换次数降到了最低，但比较的次数仍然很大。当数据量很小，并且交换数据相对于比较数据更加耗时的情况下，可以应用选择排序。

在大多数情况下，假设当数据量比较小或基本上有序时，插入排序算法是三种简单排序算法中最好的选择。对于更大数据量的排序来说，快速排序通常是最快的方法；在第 7 章中将介绍快速排序。

除了在速度方面比较排序算法外，还有一种对各种算法的衡量标准是算法需要的内存空间有多大。本章中的三种算法都可以“就地”完成排序，即除了初始的数组外几乎不需要其他内存空间。所有排序算法都需要一个额外的变量来暂时存储交换时的数据项。

重新编译实例程序，如 bubbleSort.java，用它给数量更大的数据排序，通过记录大数据量排序的时间，就可以明白不同排序算法的区别，以及在自己特定的系统中为不同数量级的数据排序所需要的时间。

小 结

- 本章提到的排序算法都假定了数组作为数据存储结构。
- 排序包括比较数组中数据项的关键字和移动相应的数据项（实际上，是数据项的引用），直到它们排好序为止。
- 本章所有算法的时间复杂度都是 $O(n^2)$。不过，某些情况下某个算法可以比其他算法快很多。
- 不变性是指在算法运行时保持不变的条件。
- 冒泡排序算法是效率最差的算法，但它最简单。
- 插入排序算法是本章介绍的 $O(n^2)$排序算法中应用最多的。
- 如果具有相同关键字的数据项，经过排序它们的顺序保持不变，这样的排序就是稳定的。
- 本章介绍的所有排序算法除了需要初始数组之外，都只需要一个临时变量。

问 题

下列问题是读者的自测题，答案见附录 C。

1. 计算机排序算法与人类排序相比较，它的局限性是：
   a. 人类擅长发明新算法。

b. 计算机只能处理数量固定的数据。

c. 人类知道什么需要排序，而计算机不知道。

d. 计算机一次只能比较两件东西。

2. 简单排序算法中的两个基本操作是_____数据项和_____数据项（或_____）。

3. 判断题：冒泡排序算法总是在所有数据项两两比较完成后停止。

4. 冒泡排序算法在哪两者之间交替进行：

a. 比较和交换

b. 移动和复制

c. 移动和比较

d. 复制和比较

5. 判断题：有 N 个数据项，冒泡排序算法精确操作了 N*N 次比较。

6. 选择排序中

a. 最大的关键字聚集到左边（较小的下标）。

b. 最小的关键字被重复的发现。

c. 为了将每个数据项插入到正确排序的位置，很多数据项将被移动。

d. 有序的数据项聚集到右边。

7. 判断题：在某个特定的排序情况中，如果交换与比较相比费时得多，那么选择排序将比冒泡排序快大约一倍。

8. 复制是交换的_____倍。

9. 选择排序中不变性是什么？

10. 插入排序中，文中描述的“被标记的队员”对应于 insertSort.java 中的哪个变量？

a. in

b. out

c. temp

d. a[out]

11. 在插入排序中，“局部有序”是指：

a. 一些数据项已经排好序了，但它们可能需要被移动。

b. 大部分数据项已在它们最终排序的位置了，但仍有一些需要排序。

c. 只有一些数据项有序。

d. 组内的数据项已经排好序，而组外面的数据项需要插入到组中来。

12. 向左或向右移动一组数据项需要重复地_____。

13. 在插入排序中，一个数据项被插入到局部有序的组合后，它将：

a. 永远不会再移动。

b. 永远不会向左边移动。

c. 经常被移出这个组。

d. 发现这组的数据项不断减少。

14. 插入排序中的不变性是_____。

15. 稳定性是指：

a. 在排序中排除有次关键字的项。
b. 在对州进行排序时，每个州的城市还要求按人口递增有序。
c. 让相同的名配相同的姓。
d. 数据项按照主关键字有序，不考虑次关键字。

实 验

执行下列实验有助于深入理解本章主题的内容。不需要编程。

1. 修改 bubbleSort.java（清单 3.1）中的 main()方法，新建一个大数组并给数组赋值。可以用下面的代码产生随机数：

```
for(int j=0; j<maxSize; j++)          // fill array with
   {                                  // random numbers
   long n = (long)( java.lang.Math.random()*(maxSize-1) );
   arr.insert(n);
   }
```

试插入 10000 个数据项。在排序前和排序后分别显示这些数据。滚屏显示将花费很长的时间。把 display()方法注释掉就可以看出排序本身花了多少时间。在不同的机器上时间是不同的。为 100000 个数据排序可能不到 30 秒钟。选择一个数组的容量，排序并计时。然后用同样的数组容量对 selectSort.java（清单 3.1）和 insertSort.java（清单 3.3）计时。观察这三种排序算法的速度的不同。

2. 在 bubbleSort.java 中设计代码使得输入的数据呈逆序（99999，99998，99997，.....）。用和实验 1 中同样数量的数据，观察排序运行的快慢，用 selectSort.java 和 insertSort.java 重复这个实验。

3. 在 bubbleSort.java 中设计代码使得输入的数据为有序（0，1，2，.....）。用实验 1 和实验 2 相比，观察排序运行的快慢，用 selectSort.java 和 insertSort.java 重复这个实验。

编程作业

做编程作业有助于巩固对本章内容的理解，并说明如何应用本章的概念。（在简介中已提到，资深教师可以从出版社的网站上得到完整的编程作业的答案。）

3.1 bubbleSort.java 程序（清单 3.1）和 BubbleSort 专题 applet 中，in 索引变量都是从左到右移动的，直到找到最大数据项并把它移动到右边的 out 变量处。修改 bubbleSort()方法，使它成为双向移动的。这样，in 索引先像以前一样，将最大的数据项从左移到右，当它到达 out 变量位置时，它掉头并把最小的数据项从右移到左。需要两个外部索引变量，一个在右边（以前的 out 变量），另一个在左边。

3.2 在 insertSort.java 程序（清单 3.3）中给 ArrayIns 类加一个 median()方法。这个方法将返回数组的中间值。（回忆一下，数组中一半数据项比中间值大，一半数据项比中间小值。）

3.3 在 insertSort.java 程序（清单 3.3）中增加一个名为 noDups()的方法，这个方法从已经有序的数组中删掉重复的数据项而不破坏有序性。（可以用 insertionSort()方法对数据排序，或者也可以简单地用 main()方法将数据有序地插入到表中。）一种解决方法是每发现一个重复的数据，就从这个位置开始到数组结尾都向前移动一个位置，但这样就导致消耗很长的 $O(N^2)$的时间级，起码在有

很多重复数据项的情况下是这样的。在设计的算法中，不论有多少重复数据，要确保数据项最多只能移动一次。这样算法就只消耗 O（N）数量级的时间。

3.4 还有一种简单排序算法是奇偶排序。它的思路是在数组中重复两趟扫描。第一趟扫描选择所有的数据项对，a[j]和 a[j+1]，j 是奇数（j=1，3，5，……）。如果它们的关键字的值次序颠倒，就交换它们。第二趟扫描对所有的偶数数据项进行同样的操作（j=2，4，6，……）。重复进行这样两趟的排序直到数组全部有序。用 oddEvenSort()方法替换 bubbleSort.java 程序（清单 3.1）中的 bubbleSort()方法。确保它可以在不同数据量的排序中运行，还需要算出两趟扫描的次数。

奇偶排序实际上在多处理器环境中很有用，处理器可以分别同时处理每一个奇数对，然后又同时处理偶数对。因为奇数对是彼此独立的，每一对都可以用不同的处理器比较和交换。这样可以非常快速地排序。

3.5 修改 insertSort.java 程序（清单 3.3）中的 insertionSort()方法，使它可以计算排序过程中复制和比较的次数并显示出总数。为计算比较的次数，要把内层 while 循环的两个条件分开。用这个程序测量各种数量的逆序数据排序的复制和比较次数。结果满足 $O(n^2)$吗？与已经基本有序的数据（仅有很少的数据无序）的情况一样吗？从对基本有序数据排序的表现中可得出关于这个算法效率的什么结论？

3.6 有一个有趣的方法用来删除数组中相同的数据项。插入排序算法中用一个循环嵌套算法，将数组中的每一个数据项与其他数据项一一比较。如果要删除相同的数据项，可以这样做（参见第 2 章第 2.6 小节）。修改 insertSort.java 中的 insertionSort()方法，使它可以在排序过程中删除相同的数据项。方法如下：当找到一个重复数据项的时候，通常用一个小于任何值的关键值来改写这个相同数据项（如果所有值都是正数，则可取-1）。于是，一般的插入排序算法就会像处理其他数据项一样，来处理这个修改了关键值的数据项，把它移到下标为 0 的位置。从现在开始，算法可以忽略这个数据项。下一个相同的数据项将被移到下标为 1 的位置，依此类推。排序完成后，所有相同的数据项（现在关键值为-1）都在数组的开头部分。可以改变数组的容量并把需要的数据前移到数组下标为 0 的位置。

第4章 栈和队列

本章重点
- 不同的结构类型
- 栈
- 队列
- 优先级队列
- 解析算术表达式

本章涉及到三种数据存储类型：栈、队列和优先级队列。首先讨论这几种结构与数组的区别；然后依次介绍每种结构。最后一节将观察栈在解析算术表达式问题中的重要应用。

不同的结构类型

本章所讲的数据结构和算法与前面章节提到的有很大不同。在开始详细讲解新的结构之前，先来看看其中的三个区别。

程序员的工具

数组是前面已经介绍过的数据存储结构，和本书后面将遇到的其他结构（链表、树等等）一样，都适用于数据库应用中作数据记录。它常用于记录那些对应于现实世界的对象和活动的数据，如职员档案、目录和商务数据等等。这些结构便于数据的访问：它们易于进行插入、删除和查找特定数据项的操作。

然而，本章要讲解的数据结构和算法更多的是作为程序员的工具来运用。它们主要作为构思算法的辅助工具，而不是完全的数据存储工具。这些数据结构的生命周期比那些数据库类型的结构要短得多。在程序操作执行期间它们才被创建，通常用它们去执行某项特殊的任务；当完成任务之后，它们就被销毁。

受限访问

在数组中，若知道数据项的下标，便可以立即访问该数据项；或者通过顺序搜索数据项，访问到数组中的各数据项。而在本章的数据结构中，访问是受限制的，即在特定时刻只有一个数据项可以被读取或者被删除（除非“作弊”）。

这些结构接口的设计增强了这种受限访问。访问其他数据项（理论上）是不允许的。

更加抽象

栈、队列和优先级队列是比数组和其他数据存储结构更为抽象的结构。主要通过接口对栈、队列和优先级队列进行定义，这些接口表明通过它们可以完成的操作，而它们的主要实现机制对用户来说是不可见的。

例如，栈的主要机制可以用数组来实现，本章的示例就是这样处理的；但它也可以用链表来实现。优先级队列的内部实现可以用数组或一种特别的树——堆来实现。在第5章“链表”中讨论抽象数据类型（ADT）的时候，将再次讨论用一种数据结构实现另一种数据结构的问题。

栈

栈只允许访问一个数据项：即最后插入的数据项。移除这个数据项后才能访问倒数第二个插入的数据项，依此类推。这种机制在不少编程环境中都很有用。本节中将看到如何利用栈来检验源程序中的小括号、中括号和大括号是否匹配的问题。本章的最后会讲到栈在解析算术表达式时起到的极为重要的作用，比如解析 3*（4+5）。

栈也是那些应用了相当复杂的数据结构算法的便利工具。比如在第 8 章“二叉树”中，用栈来辅助遍历树的节点；第 13 章“图”中，利用栈来辅助查找图的顶点（一种可以用来解决迷宫问题的技术）。

大部分微处理器运用基于栈的体系结构。当调用一个方法时，把它的返回地址和参数压入栈，当方法结束返回时，那些数据出栈。栈操作就嵌入在微处理器中。

一些比较老的便携式计算器也用基于栈的体系结构。它们不是输入带括号的算术表达式，而是把中间结果先存入栈中。本章最后一节讨论解析算术表达式问题时将更详细地讲述这种方法。

邮政模拟例

为了理解栈的思想，本章从美国邮政服务的一个模拟例子入手。许多人在工作时收到信后，会随手将它放在大厅桌子上的信堆上，或者把它投入一个“专门的”筐里。等他们有空闲时间的时候，就会从上到下处理这些堆积的邮件。他们首先打开堆叠在最上面的信，然后做相应的处理——付账单、扔掉邮件或其他的什么事情。第一封信处理完之后，接下来会处理下一封信，此时它正处于信堆的最上面。按照此方法处理下去，直到最后轮到处理信堆最下面的一封（此时它在信堆的最上面）。图 4.1 显示了一叠信件组成的栈。

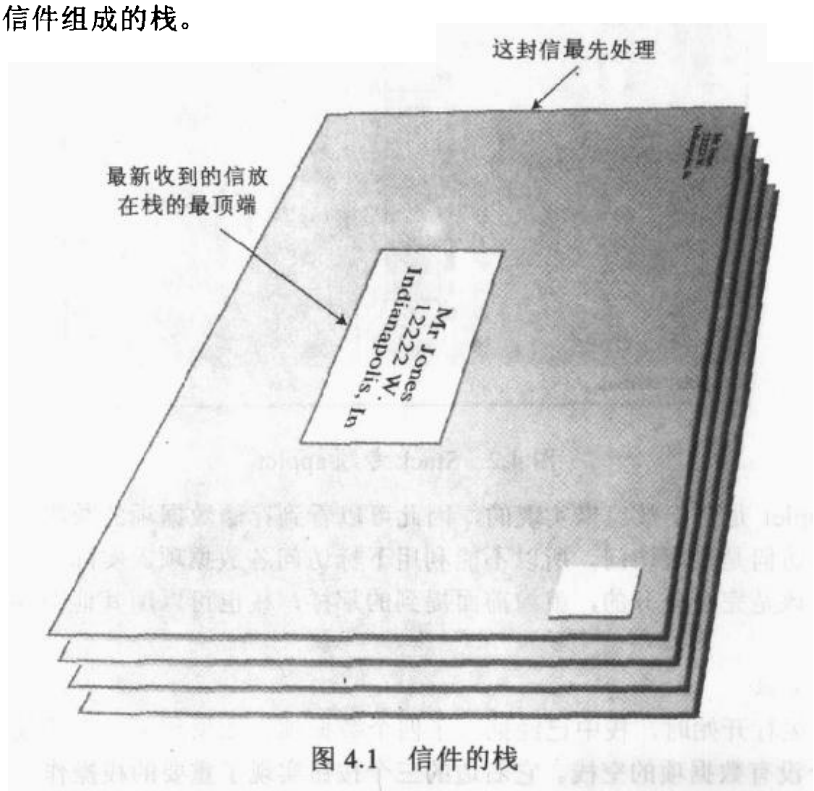


图 4.1 信件的栈

只有能够在合理的时间内从容处理完所有的信件，这种“先处理最上面的邮件”的工作方法才不会产生麻烦。否则，会发生接连几个月也处理不到放在栈底信件的危险，而信件里的账单就会逾期未付。

当然，许多人并不生硬地遵循这种从上到下的顺序。比如，他们可能先拿栈底的邮件，从最早收到的信看起。或者他们还可以在处理信件之前，改变信件的顺序，将紧急的信放在上面。在这些情况下，这些人的邮件系统就不再是计算机科学含义中的栈了。如果从栈底拿信处理，就是队列的结构；如果区分优先次序，那就是优先级队列结构。稍后会讨论这些结构。

另一个栈的模拟例子是在一个普通工作日发生的情景：某人正忙于做一个长期的项目（A），但是被同事打断叫去临时帮忙做另外一个项目（B）。当他做项目 B 的时候，负责财务的同事要他去开关于旅行花费的会议（C），在开会的过程中，他接到了销售人员的紧急电话，花了几分钟解决了一个庞大商品的纠纷（D）。挂上电话，处理完事情 D，接着开会 C；开完会 C，接着做项目 B，做完 B 后，才可以回去继续做项目 A。优先级低的项目被压到栈底，等待回来处理。

Stack 专题 applet

下面是一个 Stack 的专题 applet，用它来显示栈是怎样工作的。启动 applet，首先看到四个按钮：它们是 New（新建）、Push（入栈）、Pop（出栈）和 Peek（察看），如图 4.2 所示。

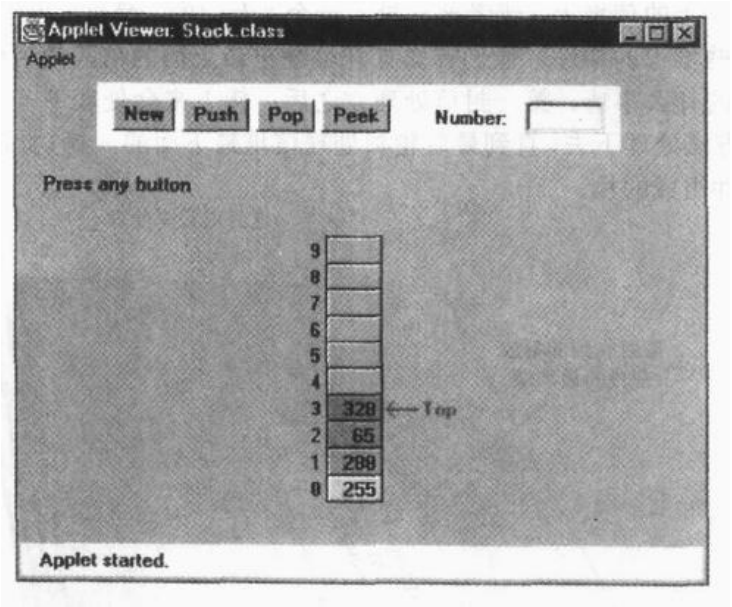


图 4.2 Stack 专题 applet

Stack 专题 applet 是基于数组来实现的，因此可以看到存储数据项的数组。虽然它是基于数组的，但是因为栈的访问是受限访问，所以不能利用下标访问各数据项。实际上，栈的概念和实现它的内部数据结构应该是完全分开的，就像前面提到的那样，栈也可以用其他的存储结构，比如链表来实现。

New（新建）按钮

专题 applet 中运行开始时，栈中已经插入了四个数据项。如果想从一个空栈开始运行，就点击 New 按钮建立一个没有数据项的空栈。它右边的三个按钮实现了重要的栈操作。

Push（入栈）按钮

点击标记为“Push”的按钮可以向栈中插入一个数据项。第一次点击这个按钮，applet 提示输入要入栈的数据项的值。在文本框中输入数值后，再点击几次就将把它压入了栈顶。

红色箭头永远指向栈顶，就是指向最后插入的数据项。注意，在插入过程中，第一步是点击一次按钮，把顶端的箭头 Top 增加一（即上移了一个单元），下一步再点击，才是真正将数据项插入到数组存储单元中。如果颠倒顺序，就会将原存在栈顶的数据项覆盖。当编写代码实现栈时，记住这两个步骤的执行顺序是非常重要的。

如果栈满了，还想继续插入数据项，会看到“Can't insert: stack is full”（栈满，不能插入）这样的消息。理论上说，ADT 定义的栈是不会满的，但数组实现的栈会满。

Pop（出栈）按钮

用 Pop 按钮移除栈顶的数据项。出栈的数据值显示在数据文本框中；它对应于 pop()例程的返回值。

同样的，注意有两步动作：首先，移除由栈顶箭头 Top 所指着的数据项；然后，栈顶箭头减一，指向新的栈顶元素。在这里的操作顺序与入栈的操作顺序相反。

出栈操作表明真的从数组中删除了一个数据项。这时此数据单元的颜色变灰，表示数据项被移除了。这有一点误导，实际上被删除的数据还继续留在数组里，直到它被新的数据项覆盖为止。但是，由于顶端标记箭头 Top 移到那些数据的位置下面，那么就不能再访问它们了。因此，像 applet 上显示的那样，在概念上它们已经不存在了。

把栈中最后一个数据项移除以后，Top 箭头指向－1，在最底端的数组单元下面。这个位置表示栈空。栈空时，还试图弹出一个数据项，就会看到“Can't pop: stack is empty”（栈空，不能出栈）的消息。

Peek（查看）按钮

入栈和出栈是栈的两个最主要的操作。但是，有时只需要读取栈顶元素的值，而不移除它。查看操作就是实现这样的功能。点击 Peek 按钮几次，就会看到栈顶元素的值复制到数字的文本框中，这个数据项没有出栈，而且保持不变。

注意，只能查看栈顶元素。通过设计，用户看不到任何其他的数据项。

栈的容量

栈通常很小，是临时的数据结构，所以 applet 中的栈只有 10 个数据单元。当然，实际程序中的栈的容量会更大一些。但奇怪的是栈容量的需求是那么小，例如，解析一个很长的算术表达式只需要一个十几个单元的栈就够了。

栈的 Java 代码

下面来看一个程序，stack.java，它用 StackX 类实现了栈。清单 4.1 包括这个类和一个很短的 main()方法来测试它。

清单 4.1 stack.java 程序

```
// stack.java
// demonstrates stacks
// to run this program: C>java StackApp
```

```
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;        // size of stack array
   private long[] stackArray;
   private int top;            // top of stack
//--------------------------------------------------------------
   public StackX(int s)         // constructor
      {
      maxSize = s;             // set array size
      stackArray = new long[maxSize];  // create array
      top = -1;                // no items yet
      }
//--------------------------------------------------------------
   public void push(long j)    // put item on top of stack
      {
      stackArray[++top] = j;     // increment top, insert item
      }
//--------------------------------------------------------------
   public long pop()           // take item from top of stack
      {
      return stackArray[top--];  // access item, decrement top
      }
//--------------------------------------------------------------
   public long peek()          // peek at top of stack
      {
      return stackArray[top];
      }
//--------------------------------------------------------------
   public boolean isEmpty()    // true if stack is empty
      {
      return (top == -1);
      }
//--------------------------------------------------------------
   public boolean isFull()     // true if stack is full
      {
      return (top == maxSize-1);
      }
//--------------------------------------------------------------
   } // end class StackX
////////////////////////////////////////////////////////////////
class StackApp
   {
   public static void main(String[] args)
      {
      StackX theStack = new StackX(10);  // make new stack
```

```
      theStack.push(20);              // push items onto stack
      theStack.push(40);
      theStack.push(60);
      theStack.push(80);

      while( !theStack.isEmpty() )    // until it's empty,
         {                            // delete item from stack
         long value = theStack.pop();
         System.out.print(value);     // display it
         System.out.print(" ");
         } // end while
      System.out.println("");
      } // end main()
   } // end class StackApp
////////////////////////////////////////////////////////////////
```

StackApp 类中的 main()方法创建一个可以容纳 10 个数据项的栈，将 4 个数据项压入栈，接着通过出栈将所有的数据项显示出来，直到栈空。下面是输出结果：

```
80 60 40 20
```

请注意显示数据的顺序和输入的顺序正好相反。这是因为最后入栈的数据项最先弹出，所以输出结果中 80 显示在最前面。

在这个版本中，StackX 类里面数据项的类型为 long 型。如第 3 章“简单排序”中提到过的一样，这个类型可以改用任何其他的类型，包括对象类型。

StackX 类方法

构造方法根据参数规定的容量创建一个新栈。栈的域包括表示最大容量的变量（即数组的大小）、数组本身以及变量 top，它存储栈顶元素的下标。（注意，由于栈是由数组实现的，需要先规定栈的大小。但是，如果使用链表来实现栈，就不需要先规定栈的容量。）

push()方法中将 top 值增加一，使它指向原顶端数据项上面的一个位置，并在这个位置上存储一个数据项。再次提醒，top 是在插入数据项之前递增的。

pop()方法返回 top 标识的数据项值，然后 top 减一。这个方法有效地从栈中移除了数据项；虽然数据项仍然存在数组中（直到有新的数据项压入栈中覆盖这个数据项），但不能再访问它了。

peek()方法仅返回 top 所指的数据项的值，不对栈做任何改动。

isEmpty()和 isFull()方法分别在栈空和栈满时返回 true。栈空时 top 变量为-1，栈满时 top 变量为 maxSize-1。

图 4.3 展示了 StackX 类的方法是如何操作的。

出错处理

有不同的方法来处理栈的错误。当向已经满了的栈中再添加一个数据项，或要从空栈中弹出一个数据项时会发生什么情况？

可以把处理这些错误的工作推给类用户。用户在插入数据项前必须要确定栈不满：

```
if( !theStack.isFull() )
   insert(item);
```

```
else
   System.out.print("Can't insert, stack is full");
```

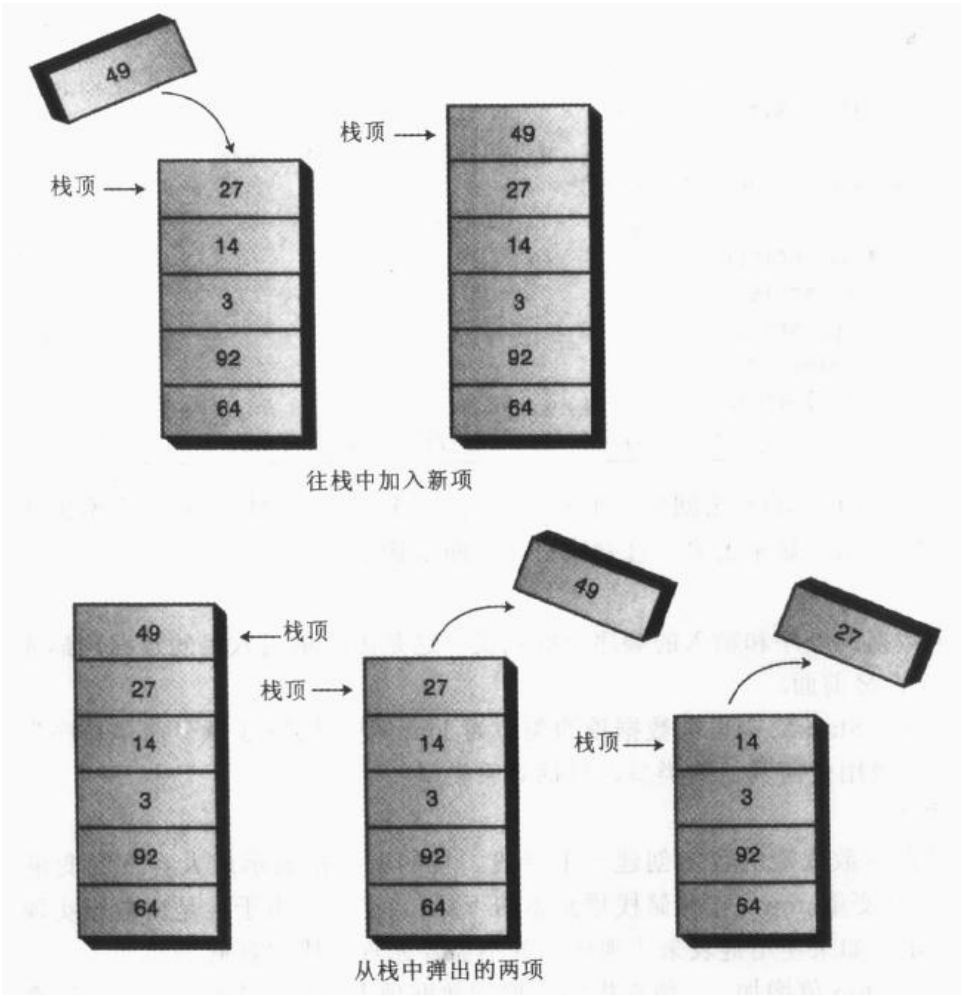


图 4.3 StackX 类中方法的操作

为了简单起见，我们没有在 main()方法中添加这些代码（而且在这个简单的程序中，因为栈刚刚初始化，它是不会满的）。当 main()中调用 pop()方法时确实判定了栈是否为空。

许多栈的类在 push()和 pop()方法内部检查这些错误，那是更好的方法。Java 语言中，对栈来说发现这些错误一个好的方法是抛出异常，异常可以被用户捕获并处理。

栈实例 1：单词逆序

栈的第一个例子是做一件非常简单的事情：单词逆序。运行程序，提示输入一个单词，点击“Enter”按钮后，屏幕上便显示字母顺序倒置后的词。

用栈进行单词逆序：首先，字母从输入的字符串中一个接一个地提取出来并且压入栈中。接着它们依次弹出栈，并显示出来。因为栈的后进先出的特性，所以字母的顺序就颠倒过来了。清单 4.2 显示 reverse.java 程序的代码。

清单 4.2 reverse.java 程序

```
// reverse.java
// stack used to reverse a string
// to run this program: C>java ReverseApp
import java.io.*;                  // for I/O
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;
   private char[] stackArray;
   private int top;
//--------------------------------------------------------------
   public StackX(int max)      // constructor
      {
      maxSize = max;
      stackArray = new char[maxSize];
      top = -1;
      }
//--------------------------------------------------------------
   public void push(char j)  // put item on top of stack
      {
      stackArray[++top] = j;
      }
//--------------------------------------------------------------
   public char pop()         // take item from top of stack
      {
      return stackArray[top--];
      }
//--------------------------------------------------------------
   public char peek()        // peek at top of stack
      {
      return stackArray[top];
      }
//--------------------------------------------------------------
   public boolean isEmpty()  // true if stack is empty
      {
      return (top == -1);
      }
//--------------------------------------------------------------
   }  // end class StackX
////////////////////////////////////////////////////////////////
class Reverser
   {
   private String input;               // input string
```

```
   private String output;              // output string
//--------------------------------------------------------------
   public Reverser(String in)          // constructor
      { input = in; }
//--------------------------------------------------------------
   public String doRev()               // reverse the string
      {
      int stackSize = input.length();  // get max stack size
      StackX theStack = new StackX(stackSize);  // make stack

      for(int j=0; j<input.length(); j++)
         {
         char ch = input.charAt(j);    // get a char from input
         theStack.push(ch);            // push it
         }
      output = "";
      while( !theStack.isEmpty() )
         {
         char ch = theStack.pop();     // pop a char,
         output = output + ch;         // append to output
         }
      return output;
      }  // end doRev()
//--------------------------------------------------------------
   }  // end class Reverser
////////////////////////////////////////////////////////////////
class ReverseApp
   {
   public static void main(String[] args) throws IOException
      {
      String input, output;
      while(true)
         {
         System.out.print("Enter a string: ");
         System.out.flush();
         input = getString();          // read a string from kbd
         if( input.equals("") )        // quit if [Enter]
            break;
                                       // make a Reverser
         Reverser theReverser = new Reverser(input);
         output = theReverser.doRev(); // use it
         System.out.println("Reversed: " + output);
         }  // end while
      }  // end main()
//--------------------------------------------------------------
```

```
    public static String getString() throws IOException
       {
       InputStreamReader isr = new InputStreamReader(System.in);
       BufferedReader br = new BufferedReader(isr);
       String s = br.readLine();
       return s;
       }
//--------------------------------------------------------------
   } // end class ReverseApp
////////////////////////////////////////////////////////////////
```

建立 Reverse 类来处理输入字符串的逆序工作。该类的关键组成部分是 doRev()方法，该方法利用栈实现逆置操作。在 doRev()中创建了一个栈，它根据输入字符串的长度确定栈的大小。

main()方法中由用户输入一个字符串，创建 Reverser 对象，字符串作为一个参数传给构造方法，接着调用这个对象的 doRev()方法，并显示返回值，这个返回值是逆序的字符串。下面是程序的一个输入和输出结果的例子：

```
Enter a string: part
Reversed: trap
Enter a string:
```

栈实例 2：分隔符匹配

栈通常用于解析某种类型的文本串。通常，文本串是用计算机语言写的代码行，而解析它们的程序就是编译器。

为了解释清楚，下面看一个检查用户输入的一行文本中分隔符的程序。文本不一定是实际的 Java 代码（也可以是），但它需要使用和 Java 同样的分隔符。分隔符包括大括号‘{’和‘}’，中括号‘[’和‘]’，和小括号‘(’和‘)’。每个左分隔符需要和右分隔符匹配；这就是说，每个‘{’后面要有‘}’来匹配，依次类推。同时，在字符串中后出现的左分隔符应该比早出现的左分隔符先完成匹配。下面是一些例子：

```
c[d]        // correct
a{b[c]d}e   // correct
a{b(c]d}e   // not correct; ] doesn't match (
a{b{c}d]e}  // not correct; nothing matches final }
a{b(c)      // not correct; nothing matches opening {
```

栈中的左分隔符

分隔符匹配程序从字符串中不断地读取字符，每次读取一个字符。若发现它是左分隔符，将它压入栈中。当从输入中读到一个右分隔符时，弹出栈顶的左分隔符，并且查看它是否和右分隔符相匹配。如果它们不匹配（比如，一个左大括号和一个右小括号），则程序报错。如果栈中没有左分隔符和右分隔符匹配，或者一直存在着没有被匹配的分隔符，程序也报错。分隔符没有被匹配，表现为把所有的字符读入之后，栈中仍留有分隔符。

看看对下面这个正确的字符串，栈的变化过程：

```
a{b(c[d]e)f}
```

表 4.1 显示了每次从字符串中读取一个字符后栈的状况。表中第二列显示的是栈中数据项，左边是栈底，右边是栈顶。

随着字符串的读取，读到左分隔符，就被压入栈。然而每读到一个右分隔符，就和从栈顶弹出的左分隔符进行匹配。如果它们匹配成功，则一切正常进行。非分隔符的字符不插入栈中，只需略过它们。

表 4.1 分隔符匹配中栈的数据项

所读字符	栈中内容
a	
{	{
b	{
(	{(
c	{(
[	{([
d	{([
]	{(
e	{(
)	{
f	{
}	

这个方法的可行性在于，最后出现的左边分隔符需要最先匹配。这个规律符合栈的后进先出的特点。

brackets.java 的 Java 代码

清单 4.3 是解析程序的代码，bracket.java。用 BracketChecker 类中的 check()方法完成解析。

清单 4.3 bracket.java 程序

```
// brackets.java
// stacks used to check matching brackets
// to run this program: C>java BracketsApp
import java.io.*;                    // for I/O
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;
   private char[] stackArray;
   private int top;
//--------------------------------------------------------------
   public StackX(int s)       // constructor
      {
      maxSize = s;
      stackArray = new char[maxSize];
```

```
      top = -1;
      }
//--------------------------------------------------------------
   public void push(char j)  // put item on top of stack
      {
      stackArray[++top] = j;
      }
//--------------------------------------------------------------
   public char pop()         // take item from top of stack
      {
      return stackArray[top--];
      }
//--------------------------------------------------------------
   public char peek()        // peek at top of stack
      {
      return stackArray[top];
      }
//--------------------------------------------------------------
   public boolean isEmpty()    // true if stack is empty
      {
      return (top == -1);
      }
//--------------------------------------------------------------
   } // end class StackX
////////////////////////////////////////////////////////////////
class BracketChecker
   {
   private String input;                  // input string
//--------------------------------------------------------------
   public BracketChecker(String in)       // constructor
      { input = in; }
//--------------------------------------------------------------
   public void check()
      {
      int stackSize = input.length();      // get max stack size
      StackX theStack = new StackX(stackSize);  // make stack

      for(int j=0; j<input.length(); j++)  // get chars in turn
         {
         char ch = input.charAt(j);        // get char
         switch(ch)
            {
            case '{':                      // opening symbols
            case '[':
            case '(':
               theStack.push(ch);          // push them
```

```
            break;

         case '}':                    // closing symbols
         case ']':
         case ')':
            if( !theStack.isEmpty() )   // if stack not empty,
               {
               char chx = theStack.pop();  // pop and check
               if( (ch=='}' && chx!='{') ||
                   (ch==']' && chx!='[') ||
                   (ch==')' && chx!='(') )
                  System.out.println("Error: "+ch+" at "+j);
               }
            else                        // prematurely empty
               System.out.println("Error: "+ch+" at "+j);
            break;
         default:    // no action on other characters
            break;
         } // end switch
      } // end for
   // at this point, all characters have been processed
   if( !theStack.isEmpty() )
      System.out.println("Error: missing right delimiter");
   } // end check()
//--------------------------------------------------------------
   } // end class BracketChecker
////////////////////////////////////////////////////////////////
class BracketsApp
   {
   public static void main(String[] args) throws IOException
      {
      String input;
      while(true)
         {
         System.out.print(
                     "Enter string containing delimiters: ");
         System.out.flush();
         input = getString();      // read a string from kbd
         if( input.equals("") )    // quit if [Enter]
            break;
                                   // make a BracketChecker
         BracketChecker theChecker = new BracketChecker(input);
         theChecker.check();       // check brackets
         } // end while
      } // end main()
```

```
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   } // end class BracketsApp
////////////////////////////////////////////////////////////////
```

check()例程从 reverse.java 程序（清单 4.2）中调用了 StackX 类。请注意重用这个类非常容易，只需要在一个地方添加代码就可以。这是面向对象编程的优点之一。

BracketsApp 类中的 main()例程从用户那里不间断地读取文本行，把文本字符串作为参数创建一个 BracketChecker 对象，然后调用这个 BracketChecker 对象的 check()方法。如果出现任何错误，check()方法将显示出来，否则分隔符的语法就是正确的。

check()方法能报告发现不匹配分隔符的位置，（最左边字符从 0 开始计算）。例如，输入字符串：

```
a{b(c]d}e
```

check()方法将会输出：

```
Error: ] at 5
```

栈是一个概念上的辅助工具

由上可见，在 brackets.java 程序中使用栈是多么方便。同样也可以利用普通数组来完成栈的操作，但是那样就不得不自己老惦记着最后添加的字符的下标值，和某些记账的问题一样。栈在抽象概念上更便于使用。栈通过提供限定性的访问方法 push()和 pop()，使程序易读且不易出错。（拿木匠的话来说，用正确的工具干活更安全。）

栈的效率

StackX 类中实现的栈，数据项入栈和出栈的时间复杂度都为常数 O(1)。这也就是说，栈操作所耗的时间不依赖于栈中数据项的个数，因此操作时间很短。栈不需要比较和移动操作。

队　列

“队列”（queue）这个单词是英国人说的“排”（line）（一种等待服务的方式）。在英国，“排队”的意思就是站到一排当中去。计算机科学中，队列是一种数据结构，有点类似栈，只是在队列中第一个插入的数据项也会最先被移除（先进先出，FIFO），而在栈中，最后插入的数据项最先移除（LIFO）。队列的作用就像电影院前的人们站成的排一样：第一个进入队尾的人将最先到达队头买票。最后排队的人最后才能买到票（或许，售票处挂出售空的牌子，他就买不到票了）。图 4.4 展示了一个队列。

图 4.4 一队人

队列和栈一样也被用作程序员的工具。第 13 章中，将看到一个利用队列在图中查找的应用例子。它也可以用于模拟真实世界的环境，例如模拟人们在银行里排队等待，飞机等待起飞，或者因特网络上数据包等待传送。

在计算机（或网络）操作系统里，有各种队列在安静地工作着。打印作业在打印队列中等待打印。当在键盘上敲击时，也有一个存储键入内容的队列。同样，如果使用文字处理程序敲击一个键，而计算机又暂时要做其他的事，敲击的内容不会丢失，它会排在队列中等待，直到文字处理程序有时间来读取它。利用队列保证了键入内容在处理时其顺序不会改变。

Queue 专题 applet

下面用 Queue 专题 applet 展示队列是如何工作的。启动 applet，可以看到一个预装了四个数据项的队列，如图 4.5 所示。

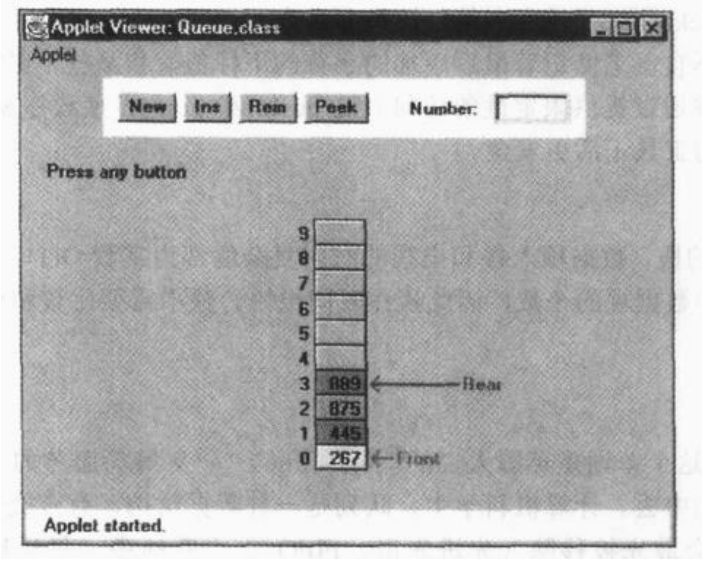


图 4.5 Queue 专题 applet

这个 applet 演示了基于数组的队列。这是一个常用的方法，同时链表也常用来实现队列。

队列的两个基本操作是 inserting（插入）一个数据项，即把一个数据项放入队尾，另一个是

removing（移除）一个数据项，即移除队头的数据项。这类似于电影爱好者排队买票时先排到队尾，然后到达队头买票后离开队列。

栈中的插入和移除数据项方法的命名是很标准，称为 push 和 pop。队列的方法至今没有标准化的命名。“插入”可以称作 put、add 或 enque，而“删除”可以叫 delete、get 或 deque。插入数据项的队尾，也可以叫作 back、tail 或 end。而移除数据项的队头，也可以叫 head。本书将使用 insert、remove、front 和 rear。

Insert（插入）按钮

在 Queue 专题 applet 上重复按 Ins（插入）按钮，可以插入新数据项。第一次点击后，出现在数值文本框处输入新数据项的关键字值的提示；可以是 0 到 999 之间任意的数字。接下来再点击按钮会将带有输入值插入队尾，同时队尾箭头增加一，指向新的数据项。

Remove（移除）按钮

类似地，可以点击 Rem 按钮删除队头数据项。数据项被移除后，数据项的值会存储到数据的文本框（Number）中（对应 remove()方法的返回值），同时队头指针（Front）增加一。在 applet 中，存放被删除数据项的单元会变灰，表示数据项已经被删除了。通常实现队列时，删除的数据项还会保存在内存中，只是它不能被访问了，因为队头指针（Front）已经移到它的下一个位置了。图 4.6 显示了插入和移除操作的过程。

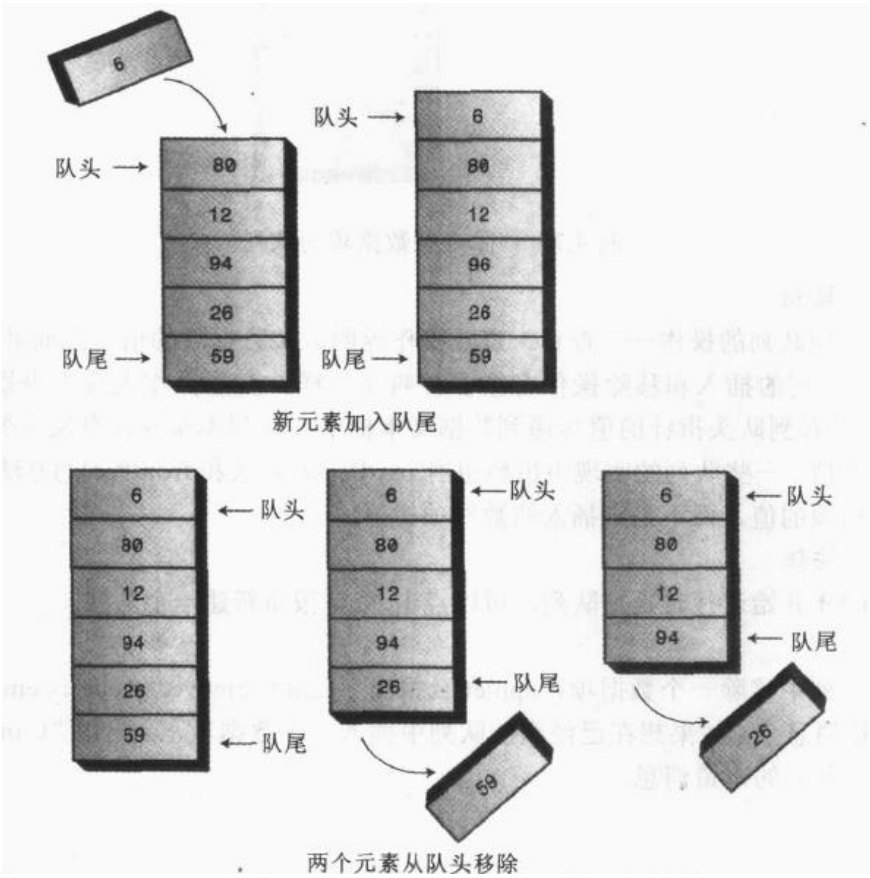


图 4.6 Queue 类方法的操作

和栈中的情况不同，队列中的数据项不总是从数组的 0 下标处开始。移除了一些数据项后，队头指针会指向一个较高的下标位置，如图 4.7 所示。

注意在图 4.7 中，数组中队头指针（Front）在队尾指针（Rear）的下面；这就是说，队头指针有比较小的下标。但是马上可以看到，情况并不总是这样的。

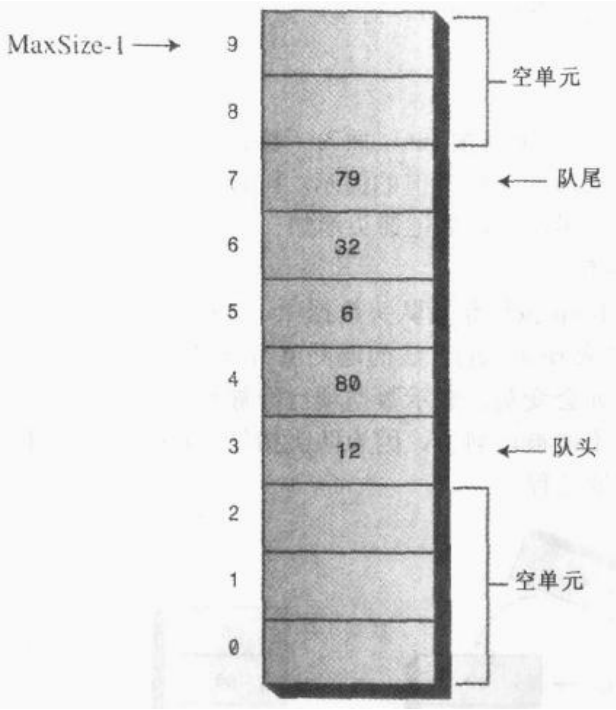


图 4.7 移除一些数据项的队列

Peek（查看）按钮

下面演示另一种队列的操作——查看。查看操作返回队头数据项的值，然而并不从队中删除这个数据项。（就像队列的插入和移除操作有很多种叫法一样，查看操作也有不少别的名字。）点击 Peek 按钮，就可以看到队头指针的值传递到数据文本框中。队列本身并没有发生变化。peek()方法返回队头数据项的值。一些队列的实现中虽然也有 rearPeek()方法和 frontPeek()方法，但通常只希望获得要移除的数据项的值，而不是刚插入的数据项的值。

New（新建）按钮

如果希望 applet 开始运行时是空队列，可以点击 New 按钮新建一个队列。

队空和队满

要是想从空队列中移除一个数据项，applet 会报出“Can't remove, queue is empty error”（队空不能移除）的出错消息来。如果想在已经满的队列中插入一个数据项，会出现“Can't insert, queue is full”（队满不能插入）的出错消息。

循环队列

当在 Queue 专题 applet 的队列中插入一个新数据项，队头的 Rear 箭头向上移动，移向数组下标大的位置。移除数据项时，队尾 Front 指针也会向上移动。在专题 applet 中试试这些操作，可以

看到情况确实如此。这种设计可能和人们直观感觉相反，因为人们在买电影票排队时，队伍总是向前移动的，当前面的人买完票离开队伍后，其他人都向前移动。计算机中在队列里删除一个数据项后，也可以将其他数据项都向前移动，但这样做的效率很差。相反，我们通过队列中队头和队尾指针的移动保持所有数据项的位置不变。

这样设计的问题是队尾指针很快就会移到数组的末端（数组下标最大）。虽然在数组的开始部分有空的数据项单元，这是点击 Rem（移除）按钮后移除的数据项的位置，但是由于因为队尾指针不能再向后移动了，因此也不能再插入新的数据项。这能怎么办？图 4.8 显示的就是这样的情况。

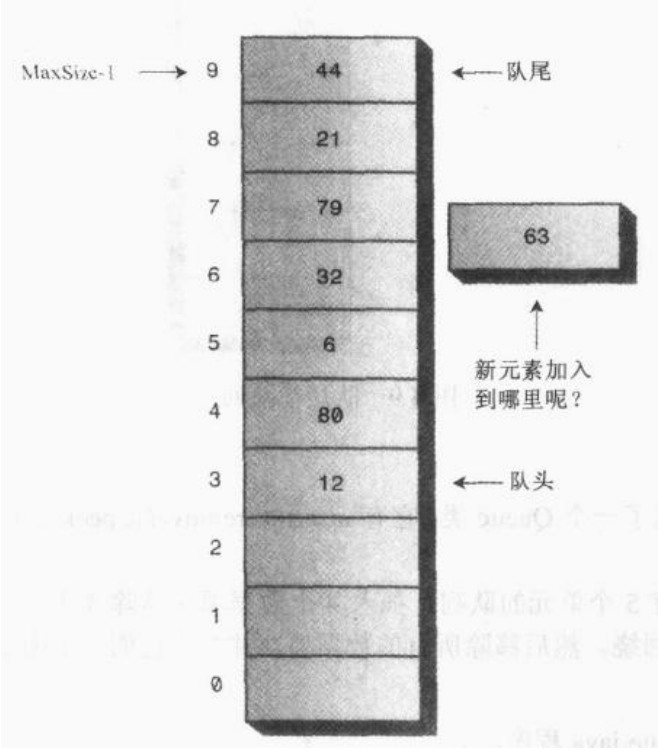


图 4.8 队尾箭头在数组末端的情况

环绕式处理

为了避免队列不满却不能插入新数据项的问题，可以让队头队尾指针绕回到数组开始的位置。这就是循环队列（有时也称为“缓冲环”）。

专题 applet 显示了指针回绕的过程：在队列中插入足够多的数据项，使队尾指针指向数组的高端（下标为 9）。再删除几个数组前端的数据项。现在插入一个新的数据项。就会看到队尾指针从下标 9 回绕到下标 0 的位置；新的数据项将插入这个位置。图 4.9 显示了这种情况。

插入更多的数据项。队尾指针如预计的那样向上移动。注意在队尾指针回绕之后，它现在处在队头指针的下面，这就颠倒了初始的位置。这可以称为“折断的序列”：队列中的数据项存在数组两个不同的序列中。

删除足够多的数据项后，队头指针也回绕。这时队列的指针回到了初始运行时的位置状态，队头指针在队尾指针的下面。数据项也恢复为单一的连续的序列。

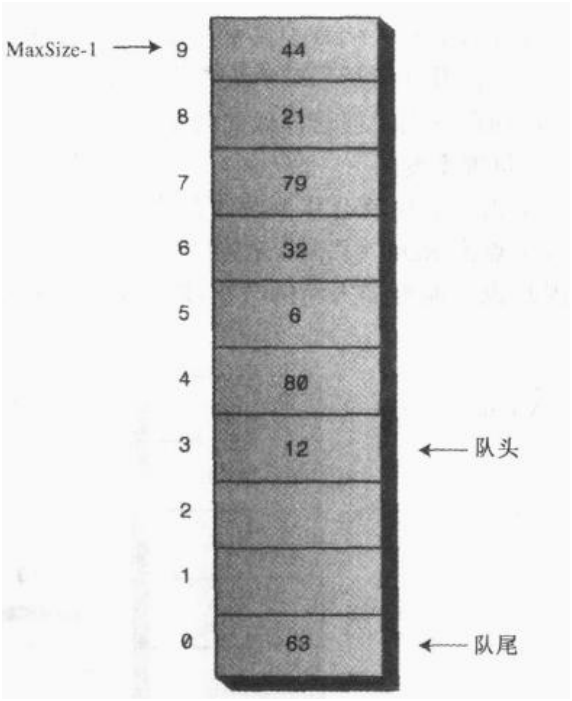


图 4.9 队尾箭头回绕

队列的 Java 代码

queue.java 程序创建了一个 Queue 类，它有 insert()、remove()、peek()、isFull()、isEmpty()和 size()方法。

main()程序创建了有 5 个单元的队列，插入 4 个数据项，移除 3 个，然后再插入 4 个数据项。第六次插入导致了指针回绕。然后移除所有的数据项，并显示它们。下面是该程序的输出：

```
40 50 60 70 80
```

清单 4.4 显示了 queue.java 程序。

清单 4.4 queue.java 程序

```
// queue.java
// demonstrates queue
// to run this program: C>java QueueApp
////////////////////////////////////////////////////////////////
class Queue
   {
   private int maxSize;
   private long[] queArray;
   private int front;
   private int rear;
   private int nItems;
//--------------------------------------------------------------
   public Queue(int s)          // constructor
```

```
      {
      maxSize = s;
      queArray = new long[maxSize];
      front = 0;
      rear = -1;
      nItems = 0;
      }
//--------------------------------------------------------------
   public void insert(long j)   // put item at rear of queue
      {
      if(rear == maxSize-1)         // deal with wraparound
         rear = -1;
      queArray[++rear] = j;         // increment rear and insert
      nItems++;                     // one more item
      }
//--------------------------------------------------------------
   public long remove()         // take item from front of queue
      {
      long temp = queArray[front++]; // get value and incr front
      if(front == maxSize)           // deal with wraparound
         front = 0;
      nItems--;                     // one less item
      return temp;
      }
//--------------------------------------------------------------
   public long peekFront()      // peek at front of queue
      {
      return queArray[front];
      }
//--------------------------------------------------------------
   public boolean isEmpty()    // true if queue is empty
      {
      return (nItems==0);
      }
//--------------------------------------------------------------
   public boolean isFull()     // true if queue is full
      {
      return (nItems==maxSize);
      }
//--------------------------------------------------------------
   public int size()           // number of items in queue
      {
      return nItems;
      }
//--------------------------------------------------------------
```

```
    } // end class Queue
////////////////////////////////////////////////////////////////
class QueueApp
   {
   public static void main(String[] args)
      {
      Queue theQueue = new Queue(5);  // queue holds 5 items

      theQueue.insert(10);            // insert 4 items
      theQueue.insert(20);
      theQueue.insert(30);
      theQueue.insert(40);

      theQueue.remove();              // remove 3 items
      theQueue.remove();              //    (10, 20, 30)
      theQueue.remove();

      theQueue.insert(50);            // insert 4 more items
      theQueue.insert(60);            //    (wraps around)
      theQueue.insert(70);
      theQueue.insert(80);

      while( !theQueue.isEmpty() )    // remove and display
         {                            //    all items
         long n = theQueue.remove();  // (40, 50, 60, 70, 80)
         System.out.print(n);
         System.out.print(" ");
         }
      System.out.println("");
      } // end main()
   } // end class QueueApp
```

程序实现的 Queue 类中不但有 front（队头）和 rear（队尾）字段，还有队列中当前数据项的个数：nItems。有些队列实现中没有这个字段，后面会显示后一种实现方法。

insert()方法

insert()方法运行的前提条件是队列不满。在 main()中没有显示这个方法，不过通常应该先调用 isFull()方法并且返回 false 后，才调用 insert()方法。（更通用的做法是在 insert()方法中加入检查队列是否满的判定，如果出现向已满队列里插入数据项的情况就抛出异常。）

一般情况，插入操作是 rear（队尾指针）加一后，在队尾指针所指的位置处插入新的数据。但是，当 rear 指针指向数组的顶端，即 maxSize-1 位置的时候，在插入数据项之前，它必须回绕到数组的底端。回绕操作把 rear 设置为-1，因此当 rear 加 1 后，它等于 0，是数组底端的下标值。最后，nItem 加一。

remove()方法

remove()方法运行的前提条件是队列不空，在调用这个方法之前应该调用 isEmpty()方法确保队

列不空，或者在 remove()方法里加入这种出错检查的机制。

移除（remove）操作总是由 front 指针得到队头数据项的值，然后将 front 加一。但是，如果这样做使 front 的值超过数组的顶端，front 就必须绕回到数组下标为 0 的位置上。作这种检验的同时，先将返回值临时存储起来。最后 nItem 减一。

peek()方法

peek()方法简单易懂：它返回 front 指针所指数据项的值。有些队列的实现也允许查看队列队尾数据项的值；比如这些方法可称为 peekFront()、peekRear()，或者只是 front()和 rear()。

isFull()、isEmpty()和 size()方法

isEmpty()、isFull()和 size()方法的实现都依赖于 nItems 字段，它们分别返回 nItems 是否等于 0，是否等于 maxSize，或者返回它本身值。

没有数据项计数字段的队列实现

在 Queue 类中包含数据项计数字段 nItems 会使 insert()和 remove()方法增加一点额外的操作，因为 insert()和 remove()方法必须分别递增和递减这个变量值。这可能算不上额外的开销，但是如果处理大量的插入和移除操作，这就可能会影响性能了。

因此，一些队列的实现不使用数据项计数的字段，而是通过 front 和 rear 来计算出队列是否空或者满以及数据项的个数。如果这样做，isEmpty()、isFull()和 size()例程会相当复杂，因为就像前面讲过的那样，数据项的序列或者被折成两段，或者是连续的一段。

而且，一个奇怪的问题出现了。当队列满的时候，front 和 rear 指针取一定的位置，但是当队列为空时，也可能呈现相同的位置关系。于是在同一时间，队列似乎可能是满的，也可能是空的。

这个问题可以这样解决，让数组容量比队列数据项个数的最大值还要大一。清单 4.5 显示了 Queue 类。这个类使用了无数据项计数实现。

清单 4.5 没有 nItem 属性的 Queue 类

```
class Queue
   {
   private int maxSize;
   private long[] queArray;
   private int front;
   private int rear;
//--------------------------------------------------------------
   public Queue(int s)          // constructor
      {
      maxSize = s+1;                // array is 1 cell larger
      queArray = new long[maxSize];  // than requested
      front = 0;
      rear = -1;
      }
//--------------------------------------------------------------
   public void insert(long j)  // put item at rear of queue
      {
      if(rear == maxSize-1)
```

```
        rear = -1;
      queArray[++rear] = j;
      }
//--------------------------------------------------------------
   public long remove()         // take item from front of queue
      {
      long temp = queArray[front++];
      if(front == maxSize)
         front = 0;
      return temp;
      }
//--------------------------------------------------------------
   public long peek()           // peek at front of queue
      {
      return queArray[front];
      }
//--------------------------------------------------------------
   public boolean isEmpty()     // true if queue is empty
      {
      return ( rear+1==front || (front+maxSize-1==rear) );
      }
//--------------------------------------------------------------
   public boolean isFull()      // true if queue is full
      {
      return ( rear+2==front || (front+maxSize-2==rear) );
      }
//--------------------------------------------------------------
   public int size()            // (assumes queue not empty)
      {
      if(rear >= front)             // contiguous sequence
         return rear-front+1;
      else                          // broken sequence
         return (maxSize-front) + (rear+1);
      }
//--------------------------------------------------------------
   } // end class Queue
```

注意这个类中的 isFull()、isEmpty()和 size()方法的复杂性。实际上很少用这样的方法实现队列，所以在这里就不详细地讨论它了。

队列的效率

和栈一样，队列中插入数据项和移除数据项的时间复杂度均为 O(1)。

双端队列

双端队列就是一个两端都是结尾的队列。队列的每一端都可以插入数据项和移除数据项。这些

方法可以叫作 insertLeft() 和 insertRight()，以及 removeLeft()和 removeRight()。

如果严格禁止调用 insertLeft()和 removeLeft()方法（或禁用右端的操作），双端队列功能就和栈一样。禁止调用 insertLeft()和 removeRight()方法（或相反的另一对方法），它的功能就和队列一样了。

双端队列与栈或队列相比，是一种多用途的数据结构，在容器类库中有时会用双端队列来提供栈和队列的两种功能。但是，双端队列不像栈和队列那么常用，因此这里就不再深入研究它了。

优先级队列

优先级队列是比栈和队列更专用的数据结构。但它在很多的情况下都很有用。像普通队列一样，优先级队列有一个队头和一个队尾，并且也是从队头移除数据项。不过在优先级队列中，数据项按关键字的值有序，这样关键字最小的数据项（或者在某些实现中是关键字最大的数据项）总是在队头。数据项插入的时候会按照顺序插入到合适的位置以确保队列的顺序。

下面来看应用于优先级队列的如何对信件排序的例子。每拿到一封信，都根据信的优先级别把它插到没看过的邮件堆里。如果它必须马上回复（电话公司要中断你的网线），就把它放在最上面，但是如果可以等到空闲的时候再回复它（一封 Mabel 姨妈的来信），就可以把它放在最底下。中等优先级别的信件就放在中间；级别越高的放的位置就越高。邮件堆的顶端对应于优先级队列的队头。

当有时间看信的时候，就从邮件堆顶端那封信（队头）看起，这样就保证了总是最先回复最重要的信。图 4.10 中显示了这种情况。

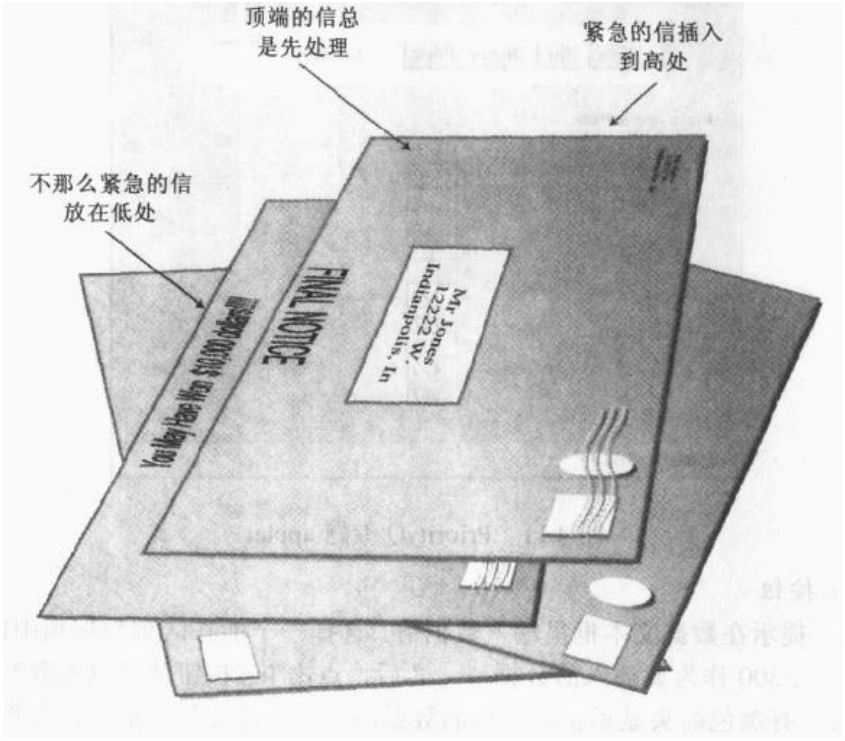


图 4.10 优先级队列中的信件

像栈和队列一样，优先级队列也经常用作程序员编程的工具。第 14 章中，可以看到在图的最小生成树算法中应用优先级队列。

像普通的队列一样，优先级队列在某些计算机系统中也有很多应用。例如，在抢先式多任务操作系统中，程序排列在优先级队列中，这样优先级最高的程序就会先得到时间片并得以运行。

很多情况下需要访问具有最小关键字值的数据项（比如要寻找最便宜的方法或最短的路径去做某件事）。因此，具有最小关键字值的数据项具有最高的优先级。本书略微有点专断，假定所有情况都是这样的，尽管有很多情况都是最大关键字具有最高的优先级。

除了可以快速访问最小关键值的数据项，优先级队列还应该可以实现相当快的插入数据项。因此，正如前面提到过的，优先级队列通常使用一种称为堆的数据结构来实现。第 12 章将会介绍堆。本章使用简单的数组实现优先级队列。这种实现方法插入比较慢，但是它比较简单，适用于数据量比较小并且不是特别注重插入速度的情况。

PriorityQ 专题 applet

PriorityQ 专题 applet 用数组实现优先级队列，并且数组中的数据项有序。这是一个升序优先级队列，具有最小关键字值的数据项具有最高优先级，它可以用 remove()方法访问。（如果访问的是最大关键字值的数据项，则是降序优先级队列。）

最小关键字值的数据项总是在数组的高端（最高下标值处），而且最大的数据项总是在下标值为 0 的位置上。图 4.11 显示了启动 applet 后队列的排列情况。初始时，队列中有 5 个数据项。

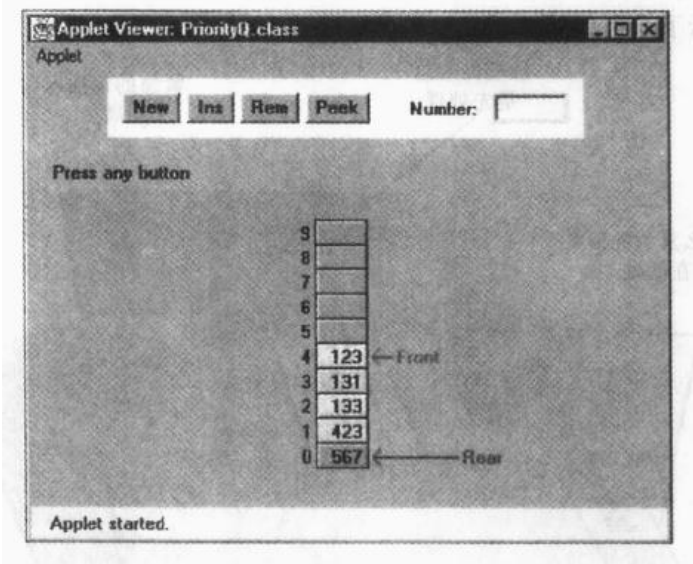


图 4.11 PriorityQ 专题 applet

Insert（插入）按钮

点击 Ins 按钮，提示在数据文本框里输入数据值。选择一个能在队列数据项中间插入的值输入。如图 4.11 所示，填写 300 作为要插入的数据值。之后，点击 Ins 按钮就可以看到数值比较小的数据项在上移腾出空间。有黑色箭头显示正在上移的数据项。找到合适位置后，新数据项就可以插到新腾出来的空间里。

注意在优先级队列的实现中没有使用指针回绕。因为要找到合适的位置，所以优先级队列的插

入肯定是很慢的，不过删除倒很快。实现指针回绕不能改善这种情况。还要注意队尾指针从不移动，它总指着数组底端下标为 0 的单元。

Delete（删除）按钮

待出队的数据项总是在数组的高端，所以删除操作又快又容易。删除数据项后，队头指针下移指向队列新的高端，不需要移动和比较。

在 PriorityQ 专题 applet 中，显示出 Front 和 Rear 箭头是为了和普通队列做比较，实际上并不需要它们。从算法中就可以知道队列的头总是数组顶端下标为 nItems-1 的位置，数据项按序插入，而不是插入到队尾处。图 4.12 展示了 PriorityQ 类方法的操作过程。

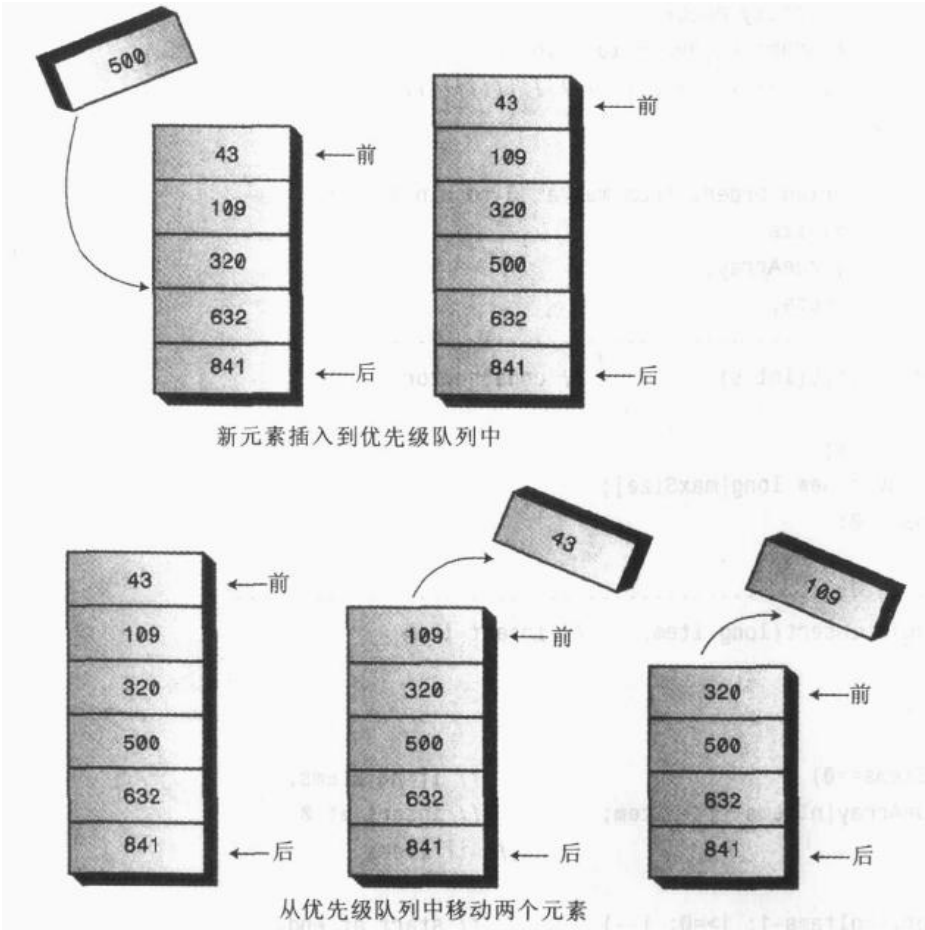


图 4.12 PriorityQ 类方法的操作

Peek（查看）和 New（创建）按钮

可以点击 Peek 按钮查看最小的数据项（查看它的值而不删除它），也可以点击 New 按钮来创建一个新的，空的优先级队列。

其他实现的可能性

PriorityQ 专题 applet 中的实现插入方法平均需要移动一半的数据项，它的效率不高。

另一种方法也用数组，但不按顺序排列数据项。新数据项就简单地插入到数组的顶部。这样插入很快，但很不幸删除却很慢，因为要找最小值。这就要在所有数据项中查找并平均移动一半数据

项，来填补删除空出的洞。因此很多情况下还是采用专题 applet 中快速删除的方法。

在数据项个数比较少，或不太关心速度的情况下，用数组实现优先级队列还可以满足要求。如果数据项很多，或速度很重要时，采用堆是更好的选择。

优先级队列的 Java 代码

清单 4.6 是一个简单的基于数组优先级队列的 Java 代码。

清单 4.6 priorityQ.java Java 代码

```
// priorityQ.java
// demonstrates priority queue
// to run this program: C>java PriorityQApp
////////////////////////////////////////////////////////////////
class PriorityQ
   {
   // array in sorted order, from max at 0 to min at size-1
   private int maxSize;
   private long[] queArray;
   private int nItems;
//--------------------------------------------------------------
   public PriorityQ(int s)          // constructor
      {
      maxSize = s;
      queArray = new long[maxSize];
      nItems = 0;
      }
//--------------------------------------------------------------
   public void insert(long item)    // insert item
      {
      int j;

      if(nItems==0)                          // if no items,
         queArray[nItems++] = item;          // insert at 0
      else                                   // if items,
         {
         for(j=nItems-1; j>=0; j--)          // start at end,
            {
            if( item > queArray[j] )         // if new item larger,
               queArray[j+1] = queArray[j]; // shift upward
            else                             // if smaller,
               break;                        // done shifting
            } // end for
         queArray[j+1] = item;               // insert it
         nItems++;
         } // end else (nItems > 0)
```

```
      } // end insert()
//--------------------------------------------------------------
   public long remove()            // remove minimum item
      { return queArray[--nItems]; }
//--------------------------------------------------------------
   public long peekMin()           // peek at minimum item
      { return queArray[nItems-1]; }
//--------------------------------------------------------------
   public boolean isEmpty()        // true if queue is empty
      { return (nItems==0); }
//--------------------------------------------------------------
   public boolean isFull()         // true if queue is full
      { return (nItems == maxSize); }
//--------------------------------------------------------------
   } // end class PriorityQ
////////////////////////////////////////////////////////////////
class PriorityQApp
   {
   public static void main(String[] args) throws IOException
      {
      PriorityQ thePQ = new PriorityQ(5);
      thePQ.insert(30);
      thePQ.insert(50);
      thePQ.insert(10);
      thePQ.insert(40);
      thePQ.insert(20);

      while( !thePQ.isEmpty() )
         {
         long item = thePQ.remove();
         System.out.print(item + " ");  // 10, 20, 30, 40, 50
         } // end while
      System.out.println("");
      } // end main()
//--------------------------------------------------------------
   } // end class PriorityQApp
```

在 main()方法中随机插入 5 个数据项，随后移除并显示它们。最小的数据项总是最先移除，所以输出是：

```
10, 20, 30, 40, 50
```

insert()方法先检查队列中是否有数据项；如果没有，就插入到下标为 0 的单元里。否则，从数组顶部开始上移存在的数据项，直到找到新数据项应当插入的位置。然后，插入新数据项，并把 nItems 加 1。注意优先级队列可能会出现满的情况，应当在用 insert()方法之前先用 isFull()判断这种可能性。

优先级队列不像 Queue 类那样必须有 front 和 rear 字段，它的 front 总是 nItems-1，rear 总是等于 0。

remove()方法本身很简单：把 nItems 减一并返回数组顶部的数据项。peekMin()方法类似，只是不把 nItems 减一。isEmpty()和 isFull()方法分别检查 nItems 等于 0 还是等于 maxSize。

优先级队列的效率

在本章实现的优先级队列中，插入操作需要 O(N)的时间，而删除操作则需要 O(1)的时间。第 12 章将介绍如何通过堆来改进插入操作的时间。

解析算术表达式

到现在为止，本章已经介绍了三种不同的数据存储结构。现在，转换一下，把注意力集中到一个实际应用问题上来。这个应用是解析（也可以说是分析）算术表达式，例如 2+3，或者 2*（3+4），或者（（2+4）*7）+3*（9-5）等。它用到的存储结构是栈。brackets.java 程序（清单 4.3）已经演示了怎样应用栈来检查分隔符是否正确匹配的问题。这和应用栈解析算数表达式的的方法类似，尽管后者更为复杂。

从某种意义上来说本节内容应作为选学内容。它不是本书后面章节的必修内容，并且除非是专门编写编译器或者是要设计便携式计算器的程序员，人们不会每天编写代码来解析算术表达式。另外，此应用的详细代码也比前面的程序复杂得多。但是这个重要的应用对于学习栈很有意义，而且这个应用本身引起的问题也非常有意思。

事实上，至少对计算机的算法来说如果要直接求算术表达式的值，还是相当困难的。因此分两步实现算法会更容易：

1. 将算术表达式转换成另一种形式：后缀表达式。
2. 计算后缀表达式的值。

第一个步骤比较难，但是第二步很简单。但是不管怎么说，这种分两步的算法比直接解析算术表达式的方法容易。当然，对人类来说，还是解析普通的算术表达式容易。稍后将会再讨论人类和计算机的方法的差别。

在研究步骤 1 和步骤 2 的具体实现之前，先来介绍后缀表达式。

后缀表达法

日常算术表达式是将操作符（operator）（+，-，*，/）放在两个操作数（operands）（数字，或代表数字的字母）之间的。因为操作符写在操作数的中间，所以把这种写法称为中缀表达法。于是，日常我们写的算术表达式就形如 2+2，4/7，或者用字母代替数字，如 A+B 和 A/B 等。

在后缀表达式中[也称作波兰逆序表达式（Reverse Polish Notation），或者 RPN，它是由一位波兰的数学家发明的]，操作符跟在两个操作数的后面。这样，A+B 就成为 AB+，A/B 成为 AB/。更复杂的中缀表达式同样可以转换成后缀表达式，如表 4.2 所示。稍后会解释后缀表达式是如何产生的。

表 4.2 中缀表达式和后缀表达式

中缀表达式	后缀表达式
A+B-C	AB+C-
A*B/C	AB*C/
A+B*C	ABC*+

续表

中缀表达式	后缀表达式
A*B+C	AB*C+
A*(B+C)	ABC+*
A*B+C*D	AB*CD*+
(A+B)*(C-D)	AB+CD-*
((A+B)*C)-D	AB+C*D-
A+B*(C-D/(E+F))	ABCDEF+/-*+

有些计算机语言也用一个操作符表示乘方（通常，用‘^’符号），但在这里暂不讨论这种表示。

除了中缀和后缀表达法，还有一种前缀表达法，操作符写在两个操作数的前面：用+AB 代替 AB+。这种表达法与后缀表达法功能类似，但是很少使用。

把中缀表达式转换成后缀表达式

下面的内容解释怎样把中缀表达的算术表达式转换成后缀表达式。这个算法是相当难的，因此刚开始要是没有完全明白也不必担心。如果很难理解这一部分，可以先跳到“求后缀表达式的值”这一节。为了理解怎样创建后缀表达式，先看怎么计算后缀表达式的值会有所帮助；例如，2*（3+4）的后缀表达式 234+*是怎样求出结果 14 的。（注意为了写起来容易，本章限制表达式中的操作数都是一位数的数字，尽管这些表达式也可以求多位数字操作数的值。）

人如何计算中缀表达式的值

怎么把中缀表达式转换成后缀表达式？首先来看一个比较容易的问题：人类是如何计算普通的中缀表达式的值的呢？正如我们前面提到的，虽然对计算机来说求值很难，但是由于 Mr. Klemmer 的长期的数学教育，我们人类做起这个问题来却相当容易。对人类来说计算出 3+4+5 或者 3*（4+5）的值不难。通过分析人类如何计算这些表达式的值，可以得到一些把表达式转换成后缀表达式的启示。

粗略地说，“解”算术表达式的时候，应该遵循下列几条规则：

1. 从左到右读取算式。（至少，假设这样。有时候人们会跳过算式前面的部分，但为了便于讨论，需要假定从左边开始系统地读表达式。）

2. 已经读到了可以计算值的两个操作数和一个操作符时，就可以计算，并用计算结果代替那两个操作数和那个操作符。（可能需要处理左边的一些还没有解决的操作，稍后可以看到。）

3. 继续这个过程——从左到右，能算就算——直到表达式的结尾。

表 4.3、4.4 和 4.5 显示了三个非常简单的中缀表达式求值的例子。在后面的表 4.6、4.7 和 4.8 中，可以看到这些求值过程与中缀到后缀的转换是多么的相似。

为了得出 3+4-5 的结果，需要执行表 4.3 中的步骤。

表 4.3 求值：3+4-5

读取元素	解析后的表达式	备注
3	3	
+	3+	
4	3+4	

续表

读取元素	解析后的表达式	备注
-	7	看到-后，可计算 3+4
	7-	
5	7-5	
End	2	到达表达式末端后，可计算 7-5

直到看到 4 后面的操作符才可以计算 3+4，如果后面的操作符是‘*’或‘/’，就要在乘除运算完成之后，才能做加法操作。

但是在这个例子中 4 后面的操作符是‘-’，它和加法运算的优先级相同，所以看到‘-’后，就可以计算 3+4 了，得到 7。7 就替代了 3+4。到表达式的末端就可以计算 7-5 了。

图 4.13 更为详细地显示了计算的过程。注意如何从左到右读取输入的数据项，然后，在得到足够的信息后，从右到左，调出前面已经查验过的输入，计算每个操作数－操作符－操作数组合体的结果。

因为优先级的问题，计算 3+4*5 更复杂一点，表 4.4 显示了它的求值过程。

表 4.4 求值：3+4*5

读取元素	解析后的表达式	备注
3	3	
+	3+	
4	3+4	
*	3+4*	不能计算 3+4，因为*比+优先级更高。 看到 5 时，可以计算 4*5
5	3+4*5	
	3+20	
End	23	到达表达式末端时，可计算 3+20

这次等到算出 4*5 的结果后才能做加 3 的运算。为什么呢？因为乘法运算比加法运算优先级别高。‘*’和‘/’运算的优先级都比‘+’和‘-’高，所有的乘法和除法运算都应该在加法和减法运算之前执行（除非用括号改变运算顺序，请看下面一个例子）。

前一个例子通常可以从左到右地求表达式的值。但是，需要确保遇到操作数－操作符－操作数的组合体时，例如 A+B，B 右边的操作符的优先级低于“+”操作符。如果右边确实是一个优先级更高的操作符，像这个例子中一样，就不能先做加法运算。当读取 5 以后，执行乘法运算，因为乘法的优先级高；无论 5 后面有‘*’或者‘/’都不会有什么影响。但是，再取过 5 的后面的数据项之前，还是不能执行加法运算。当发现 5 的后面什么也没有了，它就是表达式的结尾了，才可以执行加法运算。图 4.14 展示了这个过程。

括号用于覆盖操作符原有的优先级。表 4.5 展示了如何计算 3*（4+5）。如果没有括号，会先做乘法；而有括号，就要先做加法了。

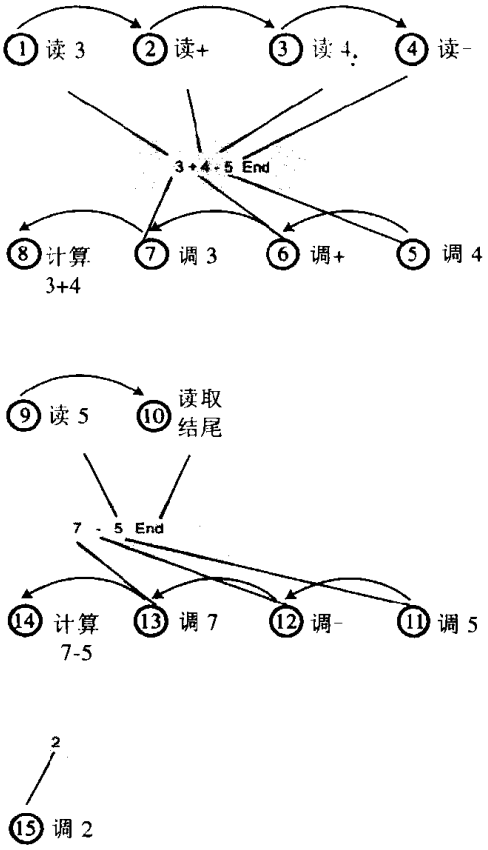


图 4.13　3+4-5 求值的详细过程

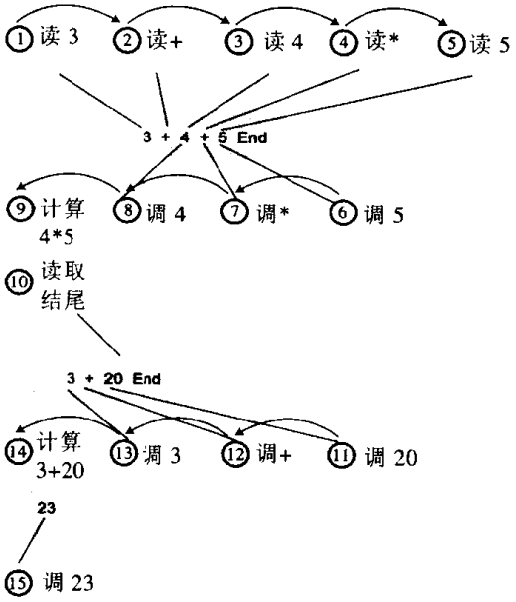


图 4.14　3+4*5 求值的详细过程

表 4.5 求值：3*（4+5）

读取元素	解析后的表达式	备注
3	3	
*	3*	
(	3*(	
4	3*(4	由于有括号，所以不能计算 3*4
+	3*(4+	
5	3*(4+5	仍不能计算 4+5
)	3*(4+5)	看到)后，就可以计算 4+5 了
	3*9	算完 4+5 后，可计算 3*9
	27	
End		完毕

这个例子里一直要读到右括号才可以计算。乘法与其他操作符相比有较高或相同的权限，因此通常情况下读到 4 就可以计算 3*4 了。但是，括号比*和/有更高的优先级。因此，必须先计算括号中的值，然后用其结果参与其他运算。遇到右括号时则可以做加法运算了，求出 4+5 等于 9，然后就可以计算 3*9 得到 27。读到表达式结尾，标志着没有计算可做了，所以这个计算过程嘎然而之。图 4.15 展示了这个过程。

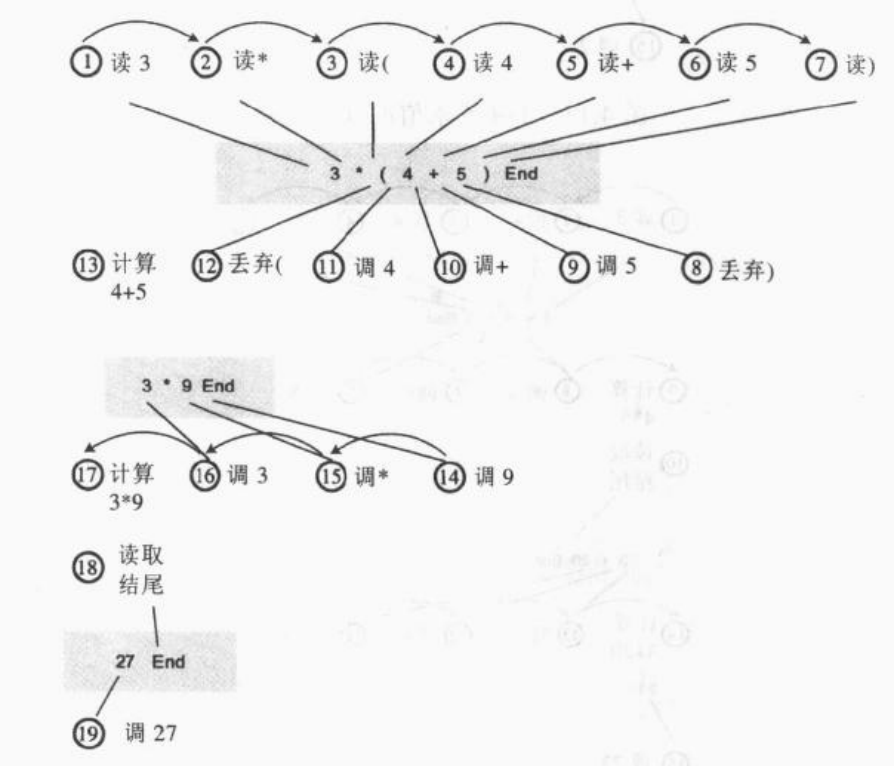


图 4.15 3*（4+5）求值的详细过程

正如看到的那样，计算中缀表达式时，既要向前，又要向后读表达式。向前（从左到右）读操作数和操作符。等到读到足够的信息来执行一个运算时，向后去找出两个操作数和一个操作符，执行运算。

有时如果后面是更高级别的操作符或者括号时，就必须推迟此运算。在这种情况下，必须先执行后面的级别高的运算；然后再回头（向左）来执行前面的运算。

我们可以直接编写这样求值的算法。但是，先把表达式转换成后缀表达法计算会更容易。

人类如何将中缀表达式转换成后缀表达式

将中缀表达式转换成后缀表达式的规则和计算中缀表达式值的规则类似。但是，还是有点变化的。将中缀表达式转换成后缀表达式不用做算术运算。它不求中缀表达式的值，只是把操作数和操作符重新排列成另一种形式：后缀表示法。然后再求后缀表达式的结果。

和以前一样，从左到右地读取中缀表达式，顺序地查看每一个字符。在此过程中，将这些操作数和操作符复制到后缀表达式输出的字符串中。关键是要知道每个字符该何时输出。

如果中缀字符串中的字符是操作数，则立即把它复制到后缀字符串中。这就是说，如果看到中缀字符串中的 A 时，就立即把 A 写到后缀字符串里。一定不要延迟：读到操作数就复制它们，而不去管多久后才能复制和它们关联的操作符。

决定何时复制操作符更复杂一些，但它的规则和计算中缀表达式时一样。一旦可以利用操作符求中缀表达式的某部分的值，就把该运算符复制到后缀字符串中。

表 4.6 将 A+B-C 转换成后缀表达式

从中缀表达式中读取的字符	分解中缀表达式过程	求后缀表达式过程	注释
A	A	A	
+	A+	A	
B	A+B	AB	
-	A+B-	AB+	读到-，可以把+复制到后缀字符串中
C	A+B-C	AB+C	
End	A+B-C	AB+C-	当读到表达式结尾处，可以复制-

注意这张表和表 4.3 的相似性，表 4.3 展示如何求中缀表达式 3+4-5 的值。前一个表中，每当计算求值的时候，在这里就只需要把操作符写到后缀表达式的输出中。

表 4.7 展示了把 A+B*C 转换成后缀表达式的过程。它和计算 3+4*5 的值的表 4.4 类似。

表 4.7 将 A+B*C 转换成后缀表达式

从中缀表达式中读取的字符	分解中缀表达式过程	求后缀表达式过程	注释
A	A	A	
+	A+	A	
B	A+B	AB	
*	A+B*	AB	读到-，可以把+复制到后缀字符串中

续表

从中缀表达式中读取的字符	分解中缀表达式过程	求后缀表达式过程	注释
C	A+B*C	ABC	看到C后，可复制*
	A+B*C	ABC*	
End	A+B*C	ABC*+	看到表达式末端时，可复制+

最后一个例子，表4.8展示了怎样将A*（B+C）转换成后缀表达式。这个过程和计算3*（4+5）的表4.5类似。直到读到右括号才能向后缀表达式输出操作符。

表4.8 将A*（B+C）转换成后缀表达式

从中缀表达式中读取的字符	分解中缀表达式过程	求后缀表达式过程	注释
A	A	A	
*	A*	A	
(	A*(	A	
B	A*(B	AB	因为有括号，所以不能复制*
	A*(B+	AB	
+	A*(B+C	ABC	仍不能复制+
C	A*(B+C)	ABC+	看到)时，可复制+
)	A*(B+C)	ABC+*	复制完+后，可复制*
End	A*(B+C)	ABC+*	完毕

在数字求值的过程中，需要向前和向后两个方向来扫描中缀表达式，以完成后缀表达的转换。当某个操作符后面的操作符优先级更高或者为左括号时，就不能把这个操作符输出到后缀表达字符串中。如果真是如上这种情况，高优先级别的操作符或括号中的操作符必须要比低优先级的操作符更早写到后缀表达字符串中。

在栈中保存操作符

在表4.7和表4.8中，可以看到从中缀到后缀的转换过程中，操作符的顺序是颠倒的。因为第一个操作符必须等第二个操作符输出后才能输出，所以操作符在后缀字符串中的顺序和中缀字符串中的顺序是相反的。一个较长的表达式可以更清楚地表明这一点。表4.9展示了将中缀表达式A+B*（C-D）转换成后缀的过程。这里添加了一列栈的内容，稍后会解释它。

表4.9 将A+B*（C-D）转换成后缀表达式

从中缀表达式中读取的字符	分解中缀表达式过程	求后缀表达式过程	栈中内容
A	A	A	
+	A+	B	+
B	A+B	AB	+

续表

从中缀表达式中读取的字符	分解中缀表达式过程	求后缀表达式过程	栈中内容
*	A+B*	AB	+*
(	A+B*(	AB	+*(
C	A+B*(C	ABC	+*(
-	A+B*(C-	ABC	+*(-
D	A+B*(C-D	ABCD	+*(-
)	A+B*(C-D)	ABCD_	+*(
	A+B*(C-D)	ABCD_	+*(
	A+B*(C-D)	ABCD_	+*
	A+B*(C-D	ABCD_*	+
	A+B*(C-D)	ABCD-*+	

从中可以看出，操作符的初始顺序在中缀表达式中是+*-，但是在后缀表达式中的却是-*+。这是因为‘*’比‘+’的优先级别高，而‘-’在括号中所以优先级比‘*’高。

这种颠倒的顺序暗示用栈来存储操作符是一个很好的方法。表 4.9 的最后一列的内容显示了转换过程中的各个阶段栈中的内容。

从某方面来说，由栈中弹出数据项实际上能向后（从右向左）扫描输入字符串。我们并没有扫描整个输入字符串，而只是查验操作符和括号。它们在读输入串的时候已经被压入栈中，所以现在可以通过出栈逆序重调用它们。

操作数（A、B 等等）在中缀和后缀表达法中出现的顺序是相同的，因此可以读到操作数的时候就输出它们；它们不需要存储在栈里。

转换规则

下面把中缀到后缀表达法的规则更明确地表示出来。从中缀表达式输入字符串中读取数据项，然后按照表 4.10 显示的操作进行处理。这些操作在伪代码中做了解释，伪代码是一种 Java 和英语的混合语言。

在这个表中，‘<’和‘>=’符号表示操作符之间的优先级关系，而不是数值的比较关系。opThis 是刚刚从中序表达式中读到的操作符，而 opTop 是刚刚出栈的操作符。

表 4.10 中缀表达式转换成后缀表达式的转换规则

从输入（中缀表达式）中读取的字符	动作
操作数	写至输出(postfix)
左括号(	推其入栈
右括号)	栈非空时，重复以下步骤 弹出一项， 若项不为(，则写至输出 项为(则退出循环

续表

从输入（中缀表达式）中读取的字符）	动作
Operator(opThis)	若栈为空， 推 opThis 否则， 栈非空时，重复 弹出一项 若项为(，推其入栈，或 若项为 operator(opTop)，且 若 opTop <opThis，推入 opTop，或 若 opTop >= opThis，输出 opTop 若 opTop < opThis 则退出循环，或项为(推入 opThis
没有更多项	当栈非空时， 弹出项目，将其输出

这些规则是相当有用的，表 4.11、4.12 和 4.13 展示了这些规则如何应用于三个中缀表达式的例子。这些表类似于表 4.6，4.7 和 4.8，不过这些表中增加了每一步相关的规则。试利用这些规则，将其他一些简单的中缀表达式转换为后缀表达式，并写出一个类似的表格。

表 4.11 转换规则应用于 A+B-C

从中缀表达式中读取的字符	分解中缀表达式的过程	求后缀表达式的过程	栈的内容	规则
A	A	A		将操作数写至输出
+	A+	A	+	若栈为空，推入 opThis
B	A+B	AB	+	将操作数写至输出
-	A+B-	AB		栈非空，所以弹出项
	A+B-	AB+		opThis 为-，opTop 为+， opTop>=opThis，所以输出 opTop
	A+B-	AB+	-	然后推 opThis
C	A+B-C	AB+C	-	将操作数写至输出
End	A+B-C	AB+C-		弹出剩余项，将其输出

表 4.12 转换规则应用于 A+B*C

从中缀表达式中读取的字符	分解中缀表达式的过程	求后缀表达式的过程	栈的内容	规则
A	A	A		将操作数写至后缀
+	A+	A	+	若栈为空，推入 opThis
B	A+B	AB	+	将操作数写至输出
*	A+B*	AB		栈非空，所以弹出 opTop

续表

从中缀表达式中读取的字符	分解中缀表达式的过程	求后缀表达式的过程	栈的内容	规则
	A+B*	AB	+	opThis 为*, opTop 为+, opTop<opThis, 所以推 opTop
	A+B*	AB	+*	然后推 opThis
C	A+B*C	ABC	+*	将操作数写至输出
End	A+B*C	ABC*	+	弹出剩余项，将其输出
	A+B*C	ABC*+		弹出剩余项，将其输出

表 4.13 转换规则应用于 A*（B+C）

从中缀表达式中读取的字符	分解中缀表达式的过程	求后缀表达式的过程	栈的内容	规则
A	A	A		将操作数写至后缀
*	A*	A	*	若栈为空，推 opThis
(	A*(	A	*(	将(推入栈
B	A*(B	AB	*(	将操作数写至后缀
+	A*(B+	AB	*	栈非空，故弹出项
	A*(B+	AB	*(	有(，所以推
	A*(B+	AB	*(+	然后推 opThis
C	A*(B+C	ABC	*(+	将操作数写至后缀
)	A*(B+C)	ABC+	*(	弹出项，写至输出
	A*(B+C)	ABC+	*	若有(则退出
End	A*(B+C)	ABC+*		弹出剩余项，将其输出

中缀表达式转换成后缀表达式的 Java 代码

清单 4.7 展示 infix.java 程序，它应用了表 4.10 的规则将中缀表达式转换成后缀表达式。

清单 4.7 infix.java 程序

```
// infix.java
// converts infix arithmetic expressions to postfix
// to run this program: C>java InfixApp
import java.io.*;               // for I/O
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;
   private char[] stackArray;
   private int top;
//--------------------------------------------------------------
   public StackX(int s)        // constructor
      {
```

```
      maxSize = s;
      stackArray = new char[maxSize];
      top = -1;
      }
//--------------------------------------------------------------
   public void push(char j)  // put item on top of stack
      { stackArray[++top] = j; }
//--------------------------------------------------------------
   public char pop()         // take item from top of stack
      { return stackArray[top--]; }
//--------------------------------------------------------------
   public char peek()        // peek at top of stack
      { return stackArray[top]; }
//--------------------------------------------------------------
   public boolean isEmpty()  // true if stack is empty
      { return (top == -1); }
//--------------------------------------------------------------
   public int size()         // return size
      { return top+1; }
//--------------------------------------------------------------
   public char peekN(int n)  // return item at index n
      { return stackArray[n]; }
//--------------------------------------------------------------
   public void displayStack(String s)
      {
      System.out.print(s);
      System.out.print("Stack (bottom-->top): ");
      for(int j=0; j<size(); j++)
         {
         System.out.print( peekN(j) );
         System.out.print(' ');
         }
      System.out.println("");
      }
//--------------------------------------------------------------
   } // end class StackX
////////////////////////////////////////////////////////////////
class InToPost                // infix to postfix conversion
   {
   private StackX theStack;
   private String input;
   private String output = "";
//--------------------------------------------------------------
   public InToPost(String in)   // constructor
      {
```

```
      input = in;
      int stackSize = input.length();
      theStack = new StackX(stackSize);
      }
//-------------------------------------------------------------
   public String doTrans()      // do translation to postfix
      {
      for(int j=0; j<input.length(); j++)
         {
         char ch = input.charAt(j);
         theStack.displayStack("For "+ch+" "); // *diagnostic*
         switch(ch)
            {
            case '+':               // it's + or -
            case '-':
               gotOper(ch, 1);      // go pop operators
               break;               //   (precedence 1)
            case '*':               // it's * or /
            case '/':
               gotOper(ch, 2);      // go pop operators
               break;               //   (precedence 2)
            case '(':               // it's a left paren
               theStack.push(ch);   // push it
               break;
            case ')':               // it's a right paren
               gotParen(ch);        // go pop operators
               break;
            default:                // must be an operand
               output = output + ch; // write it to output
               break;
            } // end switch
         } // end for
      while( !theStack.isEmpty() )     // pop remaining opers
         {
         theStack.displayStack("While ");  // *diagnostic*
         output = output + theStack.pop(); // write to output
         }
      theStack.displayStack("End   ");     // *diagnostic*
      return output;                   // return postfix
      } // end doTrans()
//-------------------------------------------------------------
   public  void gotOper(char opThis, int prec1)
      {                                // got operator from input
      while( !theStack.isEmpty() )
         {
         char opTop = theStack.pop();
```

```
         if( opTop == '(' )            // if it's a '('
            {
            theStack.push(opTop);      // restore '('
            break;
            }
         else                          // it's an operator
            {
            int prec2;                 // precedence of new op

            if(opTop=='+' || opTop=='-') // find new op prec
               prec2 = 1;
            else
               prec2 = 2;
            if(prec2 < prec1)          // if prec of new op less
               {                       //    than prec of old
               theStack.push(opTop);   // save newly-popped op
               break;
               }
            else                       // prec of new not less
               output = output + opTop;  // than prec of old
            } // end else (it's an operator)
         } //end while
      theStack.push(opThis);           // push new operator
      } // end gotOp()
//--------------------------------------------------------------
   public  void gotParen(char ch)
      {                               // got right paren from input
      while( !theStack.isEmpty() )
         {
         char chx = theStack.pop();
         if( chx == '(' )             // if popped '('
            break;                    // we're done
         else                         // if popped operator
            output = output + chx;    // output it
         } // end while
      } // end popOps()
//--------------------------------------------------------------
   } // end class InToPost
////////////////////////////////////////////////////////////////
class InfixApp
   {
   public static void main(String[] args) throws IOException
      {
      String input, output;
      while(true)
```

```
        {
        System.out.print("Enter infix: ");
        System.out.flush();
        input = getString();          // read a string from kbd
        if( input.equals("") )        // quit if [Enter]
           break;
                                      // make a translator
        InToPost theTrans = new InToPost(input);
        output = theTrans.doTrans(); // do the translation
        System.out.println("Postfix is " + output + '\n');
        } // end while
     } // end main()
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   } // end class InfixApp
////////////////////////////////////////////////////////////////
```

InfixApp 类中的 main()例程让用户输入一个中缀表达式。通过方法 readString()来完成此输入。程序创建一个 InToPost 对象，初始化参数是输入的字符串。接着调用这个对象的 doTrans()方法执行转换。这个方法返回后缀表达式字符串，并显示返回结果。

doTrans()方法用 switch 语句来处理表 4.10 中的各种转换规则。当读到操作符时，它调用 gotOper()方法，而读到右括号“)”时，调用 gotParen()方法。这些方法实现了表中第二列的两个规则，它们比其他的规则复杂得多。

StackX 类有一个 displayStack()方法，它显示栈中的所有数据项。理论上说，这样做是不合规则的：原则上只能访问栈顶元素。但是，作为一个诊断错误的辅助工具，当需要查看转换过程中每一步栈的情况时，这个例程还是很有用的。下面是一些运行时与 infix.java 程序交互的样例：

```
Enter infix: A*(B+C)-D/(E+F)
For A Stack (bottom-->top):
For * Stack (bottom-->top):
For ( Stack (bottom-->top): *
For B Stack (bottom-->top): * (
For + Stack (bottom-->top): * (
For C Stack (bottom-->top): * ( +
For ) Stack (bottom-->top): * ( +
For - Stack (bottom-->top): *
For D Stack (bottom-->top): -
For / Stack (bottom-->top): -
```

```
For ( Stack (bottom-->top): - /
For E Stack (bottom-->top): - / (
For + Stack (bottom-->top): - / (
For F Stack (bottom-->top): - / ( +
For ) Stack (bottom-->top): - / ( +
While Stack (bottom-->top): - /
While Stack (bottom-->top): -
End   Stack (bottom-->top):
Postfix is ABC+*DEF+/-
```

输出结果表明 displayStack()方法调用的位置（在 for 循环，while 循环，或者程序的结尾处）；在 for 循环中，字符就是刚刚从输入字符串中读取的。

可以用一位数，如 3 和 7 来代替类似的符号 A 和 B。对程序来说它们都是字符。例如：

```
Enter infix: 2+3*4
For 2 Stack (bottom-->top):
For + Stack (bottom-->top):
For 3 Stack (bottom-->top): +
For * Stack (bottom-->top): +
For 4 Stack (bottom-->top): + *
While Stack (bottom-->top): + *
While Stack (bottom-->top): +
End   Stack (bottom-->top):
Postfix is 234*+
```

当然，输出的缀表达式中 234 表示分隔开的数字 2、3 和 4。infix.java 程序没有检查输入的错误。如果输入了一个错误的中缀表达式，程序可能会出错、崩溃或死掉。

运行这个程序。从一些简单的中缀表达式开始，看看是不是能自己写出后缀表达式。然后运行程序检验结果。马上，你就会成为广受欢迎的后缀表达式大师。

后缀表达法求值

正如看到的一样，把中缀表达式转换成后缀表达式不是没有用。所有的这些麻烦事都是真的必要吗？是的，在计算后缀表达式的值的时候，就知道它的好处了。在讲解求值算法是多么简单之前，先来看看人类是如何对后缀表达式求值的。

人类如何用后缀表达式求值

图 4.16 展示了人类是如何通过观察和铅笔在纸上计算后缀表达式的。

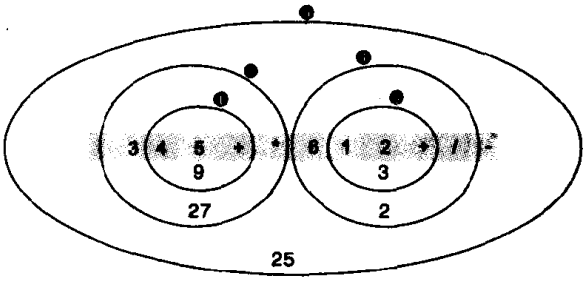


图 4.16 后缀表达式 345+*612+/-求值的形象的方法

从左边第一个操作符开始，把它和它左边相邻的两个操作数画在一个圆里。然后应用操作符运算两个操作数（执行实际的算术运算）并把结果写在圆里。如图，运算 4+5 得到 9。

现在向右到下一个操作符，在刚才画的那个圆以及它左边的操作数外面再画一个圆。应用操作符运算前一个圆的结果以及那个新的操作数，然后把结果记在新的圆里。这里得到 3*9 等于 27。继续这个过程，直到所有的操作符都运用过了：1+2 等于 3，6/3 等于 2。最大的圆里算出来的值就是这个表达式最后的结果：27-2 等于 25。

后缀表达式求值的规则

怎么才能写一个程序来重复刚才的求值过程呢？正如看到的一样，每遇到一个操作符，就用它来运算在这之前最后看到的两个操作数。这就表明可以利用栈来存储操作数。（和中缀转换后缀的算法相反，那是把操作符存储在栈里。）利用表 4.14 中的规则来求后缀表达式的值。

表 4.14 后缀表达式求值

从后缀表达式中读取的元素	执行的动作
操作数	入栈
操作符	从栈中提出两个操作数，用操作符将其执行运算。结果入栈

全部做完之后，退栈就可以得到答案。后缀表达式求值就结束了。这个过程是图 4.16 中人类画圆方法在计算机中的实现。

后缀表达式求值的 Java 代码

在中缀表达式到后缀表达式的转换过程中，使用符号（A、B 等等）代替数字。这是因为在转换过程中不需要进行算术运算，而只是将它们写成另一种形式。

现在要求后缀表达式的值了，这就需要执行算术运算并求得答案。这样，输入就必须是真实的数字了。为了简化代码，这里限制只能输入一位数的数字。

程序计算一个后缀表达式的值并输出结果。记住数字限于一位数的数字。下面是一些简单的输入输出结果：

```
Enter postfix: 57+
5 Stack (bottom-->top):
7 Stack (bottom-->top): 5
+ Stack (bottom-->top): 5 7
Evaluates to 12
```

输入数字和操作符，不要输入空格。程序可以识别出数字。虽然输入被限制为一位数的数字，但是输出结果并没有限制；如果算出大于 9 的数值是没有关系的。和 infix.java 程序一样，用 displayStack()方法来输出每一步中栈的内容。清单 4.8 展示了 psotfix.java 程序。

清单 4.8 postfix.java 程序

```
// postfix.java
// parses postfix arithmetic expressions
// to run this program: C>java PostfixApp
import java.io.*;               // for I/O
////////////////////////////////////////////////////////////////
```

```
class StackX
   {
   private int maxSize;
   private int[] stackArray;
   private int top;
//--------------------------------------------------------------
   public StackX(int size)      // constructor
      {
      maxSize = size;
      stackArray = new int[maxSize];
      top = -1;
      }
//--------------------------------------------------------------
   public void push(int j)      // put item on top of stack
      { stackArray[++top] = j; }
//--------------------------------------------------------------
   public int pop()             // take item from top of stack
      { return stackArray[top--]; }
//--------------------------------------------------------------
   public int peek()            // peek at top of stack
      { return stackArray[top]; }
//--------------------------------------------------------------
   public boolean isEmpty()     // true if stack is empty
      { return (top == -1); }
//--------------------------------------------------------------
   public boolean isFull()      // true if stack is full
      { return (top == maxSize-1); }
//--------------------------------------------------------------
   public int size()            // return size
      { return top+1; }
//--------------------------------------------------------------
   public int peekN(int n)      // peek at index n
      { return stackArray[n]; }
//--------------------------------------------------------------
   public void displayStack(String s)
      {
      System.out.print(s);
      System.out.print("Stack (bottom-->top): ");
      for(int j=0; j<size(); j++)
         {
         System.out.print( peekN(j) );
         System.out.print(' ');
         }
      System.out.println("");
      }
```

```
//--------------------------------------------------------------
   } // end class StackX
////////////////////////////////////////////////////////////////
class ParsePost
   {
   private StackX theStack;
   private String input;
//--------------------------------------------------------------
   public ParsePost(String s)
      { input = s; }
//--------------------------------------------------------------
   public int doParse()
      {
      theStack = new StackX(20);              // make new stack
      char ch;
      int j;
      int num1, num2, interAns;

      for(j=0; j<input.length(); j++)         // for each char,
         {
         ch = input.charAt(j);                // read from input
         theStack.displayStack(""+ch+" ");    // *diagnostic*
         if(ch >= '0' && ch <= '9')           // if it's a number
            theStack.push( (int)(ch-'0') ); //   push it
         else                                 // it's an operator
            {
            num2 = theStack.pop();            // pop operands
            num1 = theStack.pop();
            switch(ch)                        // do arithmetic
               {
               case '+':
                  interAns = num1 + num2;
                  break;
               case '-':
                  interAns = num1 - num2;
                  break;
               case '*':
                  interAns = num1 * num2;
                  break;
               case '/':
                  interAns = num1 / num2;
                  break;
               default:
                  interAns = 0;
               } // end switch
```

```
            theStack.push(interAns);        // push result
            } // end else
         } // end for
      interAns = theStack.pop();             // get answer
      return interAns;
      } // end doParse()
   } // end class ParsePost
////////////////////////////////////////////////////////////////
class PostfixApp
   {
   public static void main(String[] args) throws IOException
      {
      String input;
      int output;

      while(true)
         {
         System.out.print("Enter postfix: ");
         System.out.flush();
         input = getString();          // read a string from kbd
         if( input.equals("") )        // quit if [Enter]
            break;
                                       // make a parser
         ParsePost aParser = new ParsePost(input);
         output = aParser.doParse();  // do the evaluation
         System.out.println("Evaluates to " + output);
         } // end while
      } // end main()
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   } // end class PostfixApp
////////////////////////////////////////////////////////////////
```

PostfixApp 类中的 main()方法先从用户那里得到后缀表达式的字符串，然后它创建一个 ParsePost 对象，用输入的字符串初始化这个对象。然后它调用 ParsePost 类的 doParse()方法来求值。

doParse()方法从输入字符串中按字符读取。如果读到的字符是数字，就把它压入栈中。

如果读到的字符是操作符，就立即用它对栈中最上面的两个数据项执行运算。（这些操作数一

定已经在栈里了，因为输入的字符串是后缀表达式。）

算术操作的结果被压入到栈中。在最后一个字符（一定是操作符）被读取并执行计算以后，栈里只剩下一个数据项，它就是整个表达式的运算结果。

下面是一个输入更复杂表达式后的运行结果：后缀表达式是 345+*612+/-，这是图 4.16 中显示的人类求值时的表达式。这个表达式对应的中缀表达式是：3*（4+5）-6/（1+2）。（前面的章节中讲过用字母代替数字显示了一个相同的转换：中缀表示的 A*（B+C）-D/（E+F）的后缀表达式是：ABC+*DEF+/-。）下面是这个后缀表达式通过 postfix.java 程序求值的运行结果：

```
Enter postfix: 345+*612+/-
3 Stack (bottom-->top):
4 Stack (bottom-->top): 3
5 Stack (bottom-->top): 3 4
+ Stack (bottom-->top): 3 4 5
* Stack (bottom-->top): 3 9
6 Stack (bottom-->top): 27
1 Stack (bottom-->top): 27 6
2 Stack (bottom-->top): 27 6 1
+ Stack (bottom-->top): 27 6 1 2
/ Stack (bottom-->top): 27 6 3
- Stack (bottom-->top): 27 2
Evaluates to 25
```

像 infix.java 程序一样（清单 4.7），postfix.java 没有检查输入错误。如果输入了一个不正确的后缀表达式，结果将难以预测。

亲自执行一下这个程序。输入不同的后缀表达式，并观察如何求值，这样能比读程序更快地理解这个过程。

小 结

- 栈、队列和优先级队列是经常用于简化某些程序操作的数据结构。
- 在这些数据结构中，只有一个数据项可以被访问。
- 栈允许访问最后一个插入的数据项。
- 栈中重要的操作是在栈顶插入（压入）一个数据项，以及从栈顶移除（弹出）一个数据项。
- 队列只允许访问第一个插入的数据项。
- 队列的重要操作是在队尾插入数据项和在队头移除数据项。
- 队列可以实现为循环队列，它基于数组，数组下标可以从数组末端回绕到数组的开始位置。
- 优先级队列允许访问最小（或者有时是最大）的数据项。
- 优先级队列的重要操作是有序地插入新数据项和移除关键字最小的数据项。
- 这些数据结构可以用数组实现，也可以用其他机制（例如链表）来实现。
- 普通算术表达式是用中缀表达法表示的，这种命名的原因是操作符写在两个操作数的中间。

- 在后缀表达法中，操作符跟在两个操作数的后面。
- 算术表达式求值通常都是先转换成后缀表达式，然后再求后缀表达式的值。
- 在中缀表达式转换到后缀表达式以及求后缀表达式的值的过程里，栈都是很有用的工具。

问 题

下列问题作为读者的自测题。答案见附录C。

1. 假设将10、20、30和40插入到栈里，再弹出三个数据项，哪一个数据项还留在栈里？
2. 下列哪一个是正确的？
   a. 栈的出栈操作比队列的出队操作简单得多。
   b. 队列中的内容可以回绕，栈则不行。
   c. 栈顶对应于队头。
   d. 栈和队列中，序列中移除的数据项都是从数组不断增加的最大下标值处被移除的。
3. LIFO和FIFO各表示什么含义？
4. 判断题：栈和队列通常作为ADT（抽象数据类型）数组的主要实现机制。
5. 假设一个数组最左边下标值为0。一个队列代表一列排队买电影票的人，最先到的人编号为1，售票窗口在右边。那么：
   a. 数组下标值和等待买票的人的编号之间没有数字对应关系。
   b. 数组下标值和等待买票的人的编号增加的方向相反。
   c. 数组下标值与和等待买票的人的位置编号在数字上是对应的。
   d. 等待买票的人和数组数据项都向相同的方向移动。
6. 当插入或移除一个数据项时，队列中某个特定的数据项是从低下标值向高下标值移动呢，还是从高到低移动呢？
7. 假定向队列中插入15、25、35和45。然后移除三个数据项，队列中剩下哪个数据项？
8. 判断题：栈的入栈和出栈操作与队列的插入和移除操作的时间复杂度都是O(N)。
9. 队列可以用来保存：
   a. 插入排序中要排序的数据项。
   b. 对星船公司（美科幻连续剧中的名称）各种紧迫攻击的报告。
   c. 计算机用户写信时键入的内容。
   d. 正用于计算的数学表达式的符号。
10. 在典型的优先级队列中插入一个数据项的时间复杂度是多少？
11. 优先级队列中优先级（priority）的含义是：
    a. 最先插入优先级别最高的数据项。
    b. 程序员必须为实现优先级队列的数组划分访问优先级别。
    c. 按数据项的优先级别排序。
    d. 最低优先级别的数据项最先被移除。
12. 判断题：priorityQ.java程序（清单4.6）中至少有一个方法使用了线性查找。

13. 优先级队列和有序数组的一个区别是：

a. 最低优先级别的数据项不能从数组中轻易的提取出来，而在优先级队列中可以。

b. 数组必须是有序的，而优先级队列不需要。

c. 最高优先级的数据项可以很容易地从优先级队列中提取出来，而有序数组不行。

d. 以上都是。

14. 假设优先级队列基于第 2 章“数组”中，orderedArray.java 程序（清单 2.4）中的 OrdArray 类。它可以提供二分查找的能力。如果希望优先级队列有最高的性能，需要修改 OrdArray 类吗？

15. 优先级队列可以用于保存：

a. 一辆出租车需载的分布在城市不同地点的乘客。

b. 计算机键盘的键入内容。

c. 游戏程序中棋盘上的方格。

d. 模拟太阳系系统中的行星。

实　验

做这些实验题可以帮助理解本章的主题。不需要编程实现。

1. 从 Queue 专题 applet 的初始设置开始。交替地移除和插入数据项。（这样，你可以重复地用删掉的关键字值，而不必键入新数据项的值了。）注意这四个数据项是如何缓慢地上升到队列的顶端，然后出现在队列底端又接着往上移动的。

2. 使用 PriorityQ 专题 applet，计算优先级队列满和空的时候 Front 箭头和 Rear 箭头的位置。为什么优先级队列不能像普通队列那样回绕呢？

3. 想想人是怎么记住生活中的点点滴滴的？在人的脑海里，那些记忆中的往事被记住的时间是存储成栈，队列还是优先级队列的样子呢？

编程作业

做编程作业有助于巩固理解本章内容，并演示本章的概念如何应用。（在“介绍”中提到过，资深教师可以从出版者的网站上得到上机作业的完整答案。）

4.1　为 queue.java 程序（清单 4.4）的 Queue 类中写一个方法，显示队列的内容。注意这并不是要简单的显示出数组的内容。它要求按数据项插入的队列的顺序，从第一个插入的数据项到最后一个插入的数据项显示出来，不要输出因为在数组末端回绕而折成两半的样子。注意无论 front 和 rear 在什么位置上，都要正确显示出一个数据项和没有数据项的情况。

4.2　根据本章里对双端队列的讨论编写一个 Deque 类。它应该包括 insertLeft()、insertRight()、removeLeft()、removeRight()、isEmpty()和 isFull()方法。要求像队列那样支持在数据末端的回绕。

4.3　编写一个基于上机作业 4.2 的 Deque 类的栈类。这个栈类应该与 stack.java 程序（清单 4.1）中的 StackX 类具有相同的方法和功能。

4.4　清单 4.6 中展示的优先级队列能够快速地删除最高优先级的数据项，但是插入新数据项较

慢。编写程序修改 PriorityQ 类，使它插入的时间复杂度为 O(1)，但是删除最高优先级的数据项时间较慢。还要包括一个显示优先级队列内容的方法，要求和上机作业 4.1 中一样。

4.5 队列通常用于模拟人、汽车、飞机、业务等等的流动情况。应用 queue.java 程序（清单 4.4）的 Queue 类，编写一个程序模拟超市的收款队列。可以用上机作业 4.1 的 display()方法，显示出顾客的几条队列。可以通过敲击一个键插入一个新的顾客。为顾客选择排在哪一个队列上。收银员为每个顾客服务的时间是随机的（可假定为按照顾客买了多少东西而定）。一旦结完账，就从队列中删除该顾客。为了简单起见，通过敲击键模拟时间的流逝。可能每点击一下键表示时间过去了 1 分钟。（当然，Java 有更复杂的方式来处理时间。）

第 5 章 链 表

本章重点

- 链结点
- 单链表
- 查找和删除指定链结点
- 双端链结点
- 链表的效率
- 抽象数据类型
- 有序链表
- 双向链表
- 迭代器

在第 2 章“数组”中，我们看到数组作为数据存储结构有一定的缺陷。在无序数组中，搜索是低效的；而在有序数组中，插入效率又很低；不管在哪一种数组中删除效率都很低。况且一个数组创建后，它的大小是不可改变的。

在本章中，我们将看到一种新的数据存储结构，它可以解决上面的一些问题。这种数据存储结构就是链表。链表可能是继数组之后第二种使用得最广泛的通用存储结构。

链表的机制灵活，用途广泛，它适用于许多通用的数据库。它也可以取代数组，作为其他存储结构的基础，例如栈，队列。除非需要频繁通过下标随机访问各个数据，否则在很多使用数组的地方都可以用链表代替。

链表虽不能解决数据存储中的所有问题，但是它确实用途广泛，在概念上也比其他的常用结构（例如树）简单。随着问题的深入，我们将探讨链表的优点和缺点。

在本章中我们将学习单链表、双端链表、有序链表、双向链表和有迭代器的链表（迭代器是用来随机访问链表元素的一种方法）。还会实践一下抽象数据类型（ADT）的思想：如何用 ADT 描述栈和队列，以及如何用链表代替数组来实现栈和队列。

链结点（Link）

在链表中，每个数据项都被包含在“链结点”（Link）中。一个链结点是某个类的对象，这个类可以叫做 Link。因为一个链表中有许多类似的链结点，所以有必要用一个不同于链表的类来表达链结点。每个 Link 对象中都包含一个对下一个链结点引用的字段（通常叫做 next）。但是链表本身的对象中有一个字段指向对第一个链结点的引用。图 5.1 显示了这个关系。

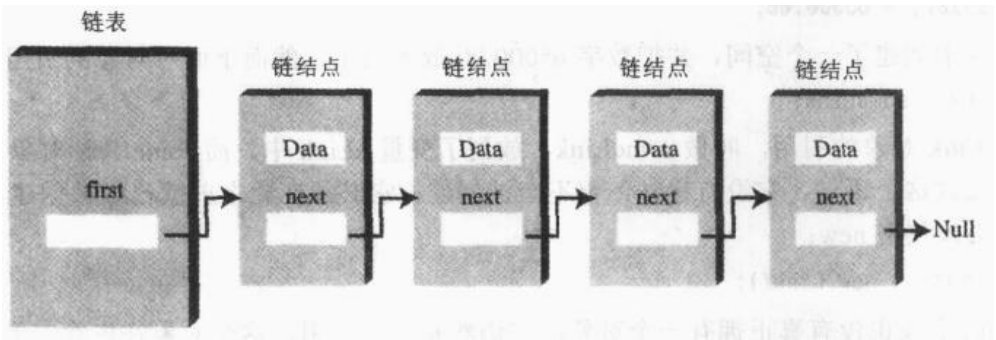


图 5.1　链表中的链结点

下面是一个 Link 类定义的一部分。它包含了一些数据和对下一个链结点的引用：

```
class Link
   {
   public int iData;      // data
   public double dData;   // data
   public Link next;      // reference to next link
   }
```

这种类定义有时叫做“自引用”式，因为它包含了一个和自己类型相同的字段（本例中叫做 next）。

链结点中仅包含两个数据项：一个 int 类型的数据，一个 double 类型的数据。在一个真正的应用程序中，可能包含更多的数据项。例如，一条个人记录可能有名字，地址，社会保险号，头衔，工资和其他许多字段。通常，用一个包含这些数据的类的对象来代替这些数据项：

```
class Link
   {
   public inventoryItem iI;  // object holding data
   public Link next;         // reference to next link
   }
```

引用和基本类型

在链表的环境中，很容易对“引用”产生混淆，所以让我们回顾一下引用如何运作。

在 Link 的类定义中定义一个 Link 类型的域，这看起来很奇怪。编译器怎样才能不混淆呢？编译器在不知道一个 Link 对象占多大空间的情况下，如何能知道一个包含了相同对象的 Link 对象占用多大空间呢？

在 Java 语言中，这个问题的答案是 Link 对象并没有真正包含另外一个 Link 对象，尽管看起来好像包含了。类型为 Link 的 next 字段仅仅是对另外的一个 Link 对象的“引用”，而不是一个对象。

一个引用是一个对某个对象的参照数值，它是一个计算机内存中的对象地址，然而不需要知道它的具体值；只要把它当成一个奇妙的数，它会告诉你对象在哪里。在给定的计算机/操作系统中，所有的引用，不管它指向谁，大小都是一样的。因此，对编译器来说，知道这个字段的大小并由此构造出整个 Link 对象，是没有任何问题的。

注意，在 Java 语言中，例如 int 和 double 等简单类型的存储与对象的存储是完全不同的。含有简单类型的字段不是引用，而是实实在在的数值，例如 7 或者 3.14159。像下面这个变量的定义

```
double salary = 65000.00;
```

它在内存中创建了一个空间，并把数字 65000.00 放入其中。然而下面对对象的引用

```
Link aLink = someLink;
```

它把对 Link 对象的引用，叫做 someLink，放到了变量 aLink 中。而 someLink 对象本身是在其他地方的。通过这个语句，它没有移动，更不会被创建；它必须在此之前就已经被创建。为了创建一个对象，必须使用 new：

```
Link someLink = new Link();
```

someLink 字段也没有真正拥有一个对象；它仍然是一个引用。这个对象在内存中的某个地方，如图 5.2 所示。

其他的程序设计语言，例如 C++，操控对象的方法与 Java 有很大不同。在 C++中，一个字段例如

```
Link next;
```

确实包含了一个 Link 类型的对象。你不能在 C++中写一个自引用的类定义（尽管你可以在类 Link 中使用一个指向 Link 对象的指针；指针的作用和引用是类似的）。C++程序员应该记住 Java 语言是如何操控对象的；这种用法和直观感觉是不一样的。

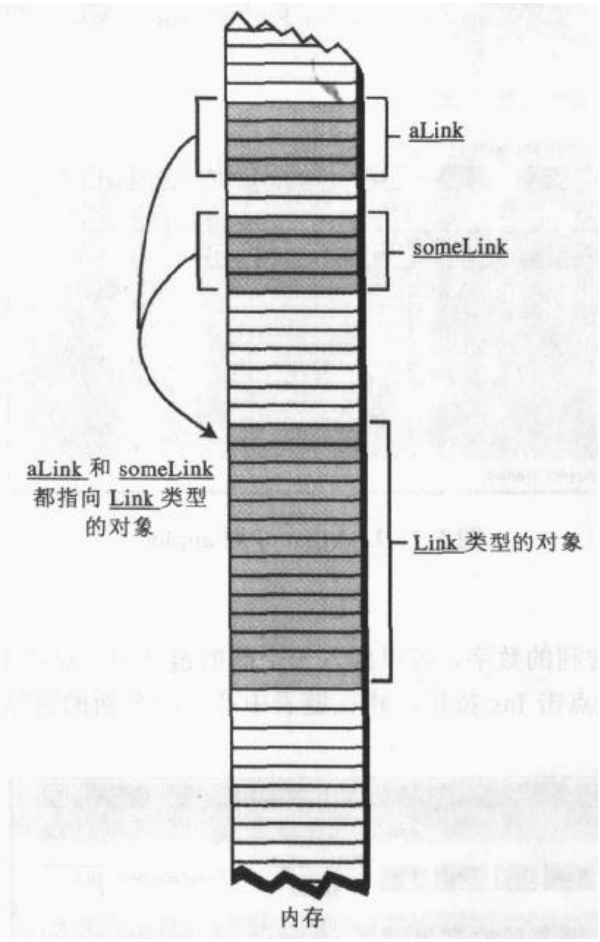


图 5.2 内存中的对象和引用

关系，而不是位置

让我们来举例说明链表不同于数组的主要特点之一。在一个数组中，每一项占用一个特定的位置。这个位置可以用一个下标号直接访问。它就像一排房子，你可以凭地址找到其中特定的一间。

在链表中，寻找一个特定元素的惟一方法就是沿着这个元素的链一直向下寻找。它很像人们之间的关系。可能你问 Harry，Bob 在哪。Harry 不知道，但是他想 Jane 可能知道，所以你又去问 Jane。Jane 看到 Bob 和 Sally 一起离开了办公室，所以你打 Sally 的手机，她说她在 Peter 的办公室和 Bob 分开了，所以……但是总有线索。不能直接访问到数据项；必须使用数据之间的关系来定位它。你从第一项开始，到第二个，然后到第三个，直到发现要找的那个数据项。

LinkList 专题 Applet

LinkList 专题 applet 提供了三种链表操作：插入一个新的数据项，按某一关键值搜索链表，删除含有特定关键值的数据项。这些操作与我们在第 2 章中讨论的 Array 专题 applet 中的操作是类似的；它们适合于通用数据库的应用。

图 5.3 显示了 LinkList 专题 applet 启动时的界面。最开始链表中有 13 个链结点。

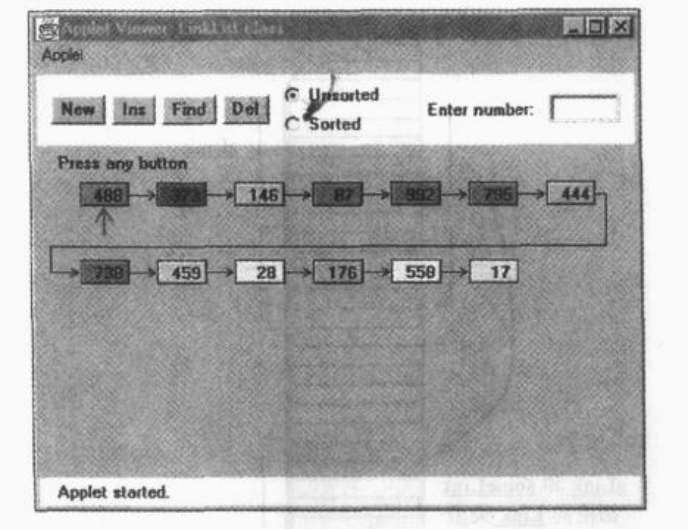


图 5.3 LinkList 专题 applet

Ins 按钮

如果你认为 13 是个不吉利的数字，可以插入一个新的链结点。点击 Ins 按钮，系统提示你输入一个 0 到 999 的数字。再次点击 Ins 按钮，将在链表中产生一个新的链结点，它的值就是刚才输入的数据，如图 5.4 所示。

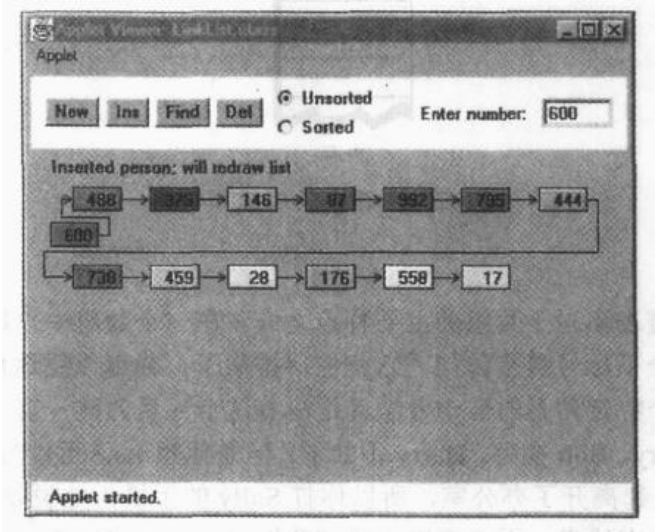


图 5.4 插入一个新的链结点

在专题 applet 这种链表的实现里，新的链结点总是插在链表的表头。这是最简单的办法，后面将会讨论把链结点插到链表的任何位置的这种情况。

最后再点击 Ins 按钮，系统将会重画链表，让新插入的链结点与其他的链结点形成一条直线。这个重画的过程不会在程序内部发生任何变化，它只是使显示更清晰。

Find 按钮

Find 按钮允许按某个指定的关键值搜索整个链表。在提示框中输入某个链结点包含的关键值，最好选在链表中部的某个值。继续按这个按钮时，将看到一个红色的箭头沿着链表移动，它正在寻找那个链结点。当箭头找到了要找的链结点，系统产生一条通知信息。如果键入了一个并不存在的关键值，箭头会沿着整个链表走到头，然后报告没有找到这个数据项。

Del 按钮

也可以根据一个特定的关键值删除链结点。键入一个存在的关键值，不断点击 Del 按钮。箭头再次沿着链表移动，查找链结点。当箭头发现链结点，它将删除这个链结点，然后把前一个链结点的箭头跨接到后一个链结点上。链结点是这样被删除的：前面链结点的引用指向了后一个链结点。

最后点击 Del 按钮，系统重画了整个界面，但是这次重画也是出于美观的考虑把存在的链结点均匀分布在界面上；箭头长度的变化并没有修改程序。

注意

LinkList 专题 applet 能创建有序或无序的链表。无序表是缺省的。在本章后面讨论有序表时，我们会看到如何在有序表的情况下使用这个 applet。

单链表

第一个示例程序，linkList.java，显示了一个单链表。这个链表仅有的操作是：

- 在链表头插入一个数据项
- 在链表头删除一个数据项
- 遍历链表显示它的内容

这些操作都相当容易实现，所以我们将从它们开始介绍。（看到后面的内容你会发现，用链表实现栈的时候，这些操作都是必须实现的。）

在着手完成 linkList.java 程序前，先看一下 Link 类和 LinkList 类中比较重要的部分。

Link 类

前面已经见过了 Link 类的数据部分。下面是完整的类定义：

```
class Link
   {
   public int iData;              // data item
   public double dData;           // data item
   public Link next;              // next link in list
// -------------------------------------------------------------
   public Link(int id, double dd) // constructor
```

```
        {
        iData = id;                 // initialize data
        dData = dd;                 // ('next' is automatically
        }                           //  set to null)
    // -------------------------------------------------------------
      public void displayLink()     // display ourself
        {
        System.out.print("{" + iData + ", " + dData + "} ");
        }
      } // end class Link
```

除了数据域外，还有构造函数和方法 displayLink()，这个方法显示链结点的数据值，格式为{22, 33.9}。一些狂热追求面向对象的程序员可能会反对把这个方法命名为 displayLink()，他们认为应该简单地叫作 display()，使用更短的名字遵循多态的思想。但是当你在代码中看到这样一个语句

```
current.display();
```

时，可能会感到有些难理解。因为你已经忘了 current 是否是一个 Link 对象，或者是 LinkList 对象还是什么其他的对象。

构造函数初始化数据，但是这里不需要初始化 next 字段，因为当它被创建时自动赋成 null 值。（然而，为了清晰起见，也可以明确的把它赋成 null 值。）null 值意味着这个字段不指向任何结点，除非该链结点后来被连接到其他的链结点才改变。

Link 各个字段的存取权限设为 public。如果它们是 private，必须提供公开的方法来访问它们，这需要额外的代码，会使清单变长且难于阅读。理想情况下，为了安全，可能需要把对 Link 对象的访问都约束在 LinkList 类的方法中。然而，如果在这些类中没有继承的关系，这样做并不太方便。我们可以用缺省访问指示符（没有保留字）使数据具有“包访问权限”（访问限制在相同目录的类中）。这样做对这些示例程序倒是没有任何影响的，因为它们都在同一个目录下。Public 指示符至少说明一个问题：这些数据不是私有的。在一个更加正式的程序中，可能需要使 Link 类中所有的数据字段都设成私有（private）的。

LinkList 类

LinkList 类只包含一个数据项：即对链表中第一个链结点的引用，叫作 first。它是惟一的链表需要维护的永久信息，用以定位所有其他的链结点。从 first 出发，沿着链表通过每个链结点的 next 字段，就可以找到其他的链结点：

```
class LinkList
   {
   private Link first;              // ref to first link on list

// -------------------------------------------------------------
   public void LinkList()          // constructor
      {
      first = null;                // no items on list yet
      }
// -------------------------------------------------------------
```

```
    public boolean isEmpty()       // true if list is empty
       {
       return (first==null);
       }
// ------------------------------------------------------------
//   ...   other methods go here
   }
```

LinkList 的构造函数把 first 赋成 null 值。实际上这不是必须的，正如上面提到的，引用类型在创建之初会自动赋成 null 值。然而，构造函数明确地说明了赋初值，从而强调 first 是怎么开始运作的。

当 first 的值为 null 时，就表明链表中没有数据项。如果有，first 字段中应该存有对第一个链结点的引用值。isEmpty()方法用这种方法来判断链表是否为空。

insertFirst()方法

LinkList 类中的 insertFirst()方法的作用是在表头插入一个新链结点。这是最容易插入一个链结点的地方，因为 first 已经指向了第一个链结点。为了插入新链结点，只需要使新创建的链结点的 next 字段等于原来的 first 的值，然后改变 first 的值，使它指向新创建的链结点。图 5.5 显示了这个过程。

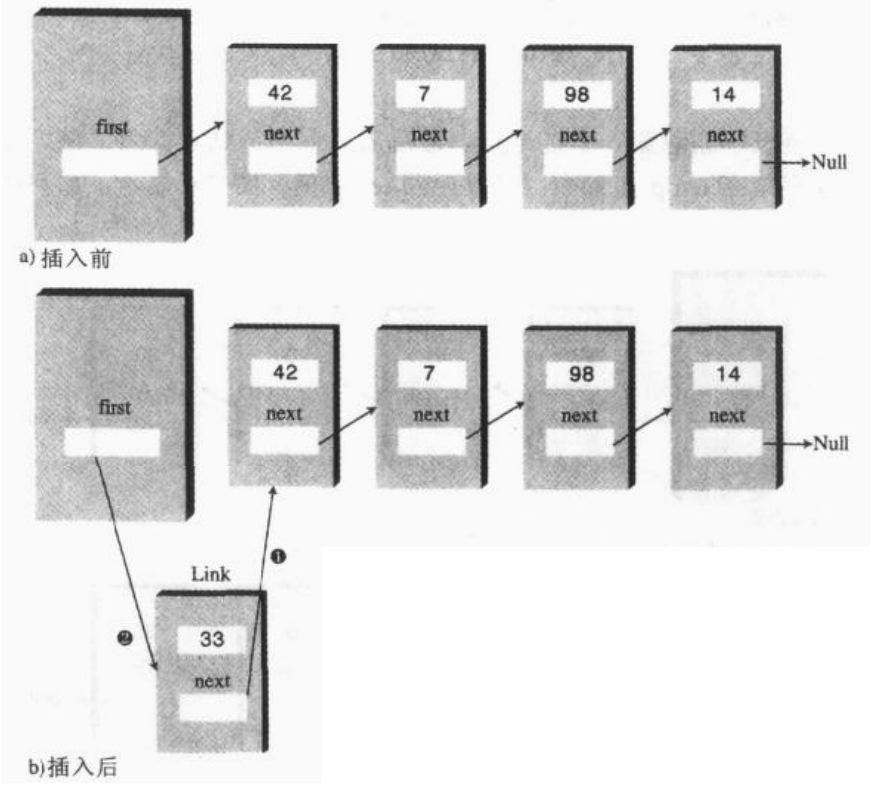


图 5.5 插入一个新的链结点

在 insertFirst()方法中，首先创建一个新链结点，把数据作为参数传入，然后改变链结点的引用，正如刚才谈到的：

```
                                  // insert at start of list
public void insertFirst(int id, double dd)
```

```
{                           // make new link
Link newLink = new Link(id, dd);
newLink.next = first;       // newLink --> old first
first = newLink;            // first --> newLink
}
```

最后两行语句的注释中有右箭头“-->”的标记，它的意思是一个链结点（或者是 first 字段）连接了下一个（下游的）链结点。（在双向链表中，将会看到下游结点连接上游结点，那时会用左箭头“<--”的标记）。比较图 5.5 和这两行语句。确保你理解了这些语句如何导致了链结点的变化，正如图中所显示的那样。这种对引用的操纵是链表算法的核心。

deleteFirst()方法

deleteFirst()方法是 insertFirst()方法的逆操作。它通过把 first 重新指向第二个链结点，断开了和第一个链结点的连接。通过查看第一个链结点的 next 字段可以找到第二个链结点：

```
public Link deleteFirst()       // delete first item
   {                            // (assumes list not empty)
   Link temp = first;           // save reference to link
   first = first.next;          // delete it: first-->old next
   return temp;                 // return deleted link
   }
```

第二行语句是从链表中删除第一个链结点。最后需要返回链结点，为了链表使用者的方便，我们在删除它之前把它存储在 temp 变量中，并且返回 temp 值。图 5.6 显示了如何改变 first，从而删除对象。

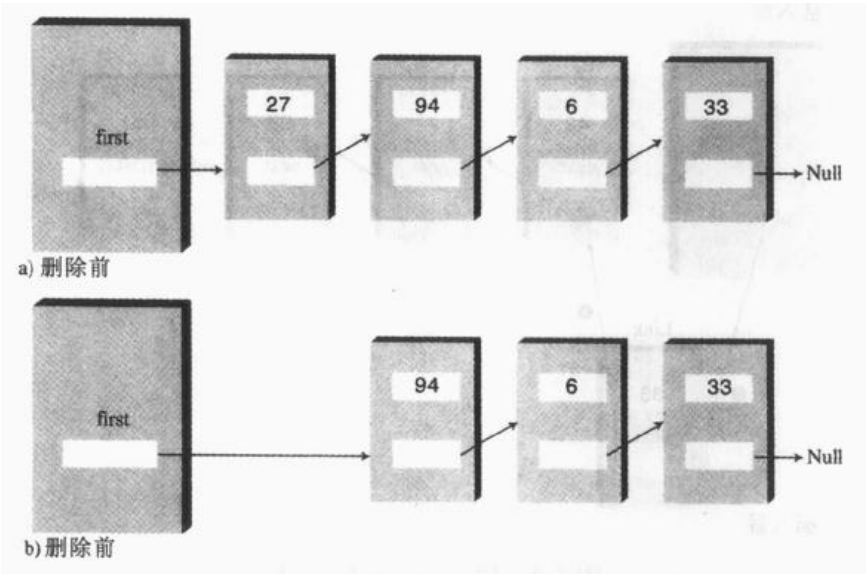


图 5.6 删除一个链结点

在 C++和类似的语言中，在从链表中取下一个链结点后，需要考虑如何删除这个链结点。它仍在内存中的某个地方，但是现在没有任何东西指向它。将如何处理它呢？在 Java 语言中，垃圾收集进程将在未来的某个时刻销毁它，现在这不是程序员操心的工作。

注意，deleteFirst()方法假定链表是不空的。调用它之前，程序应该首先调用 isEmpty()方法核实这一点。

displayList()方法

为了显示链表，从 first 开始，沿着引用链从一个链结点到下一个链结点。变量 current 按顺序指向（用术语说叫做引用）每一个链结点。current 首先指向 first，那里拥有对第一个链结点的引用。这个语句

```
current = current.next;
```

改变了 current，使它指向下一个链结点，因为那是每个链结点的 next 字段的内容。下面是完整的 displayList()方法：

```
public void displayList()
   {
   System.out.print("List (first-->last): ");
   Link current = first;       // start at beginning of list
   while(current != null)      // until end of list,
      {
      current.displayLink();   // print data
      current = current.next;  // move to next link
      }
   System.out.println("");
   }
```

链表的尾端是最后一个链结点，它的 next 字段为 null 值，而不是其他的链结点。这个字段怎么会变成 null 呢？因为在链结点被创建时这个字段就是 null，而该链结点总是停留在链表的尾端，后来再也没有改变过。当执行到链表的尾端的时候，while 循环使用这个条件来终止自己。图 5.7 显示了 current 如何沿着链表向前步进。

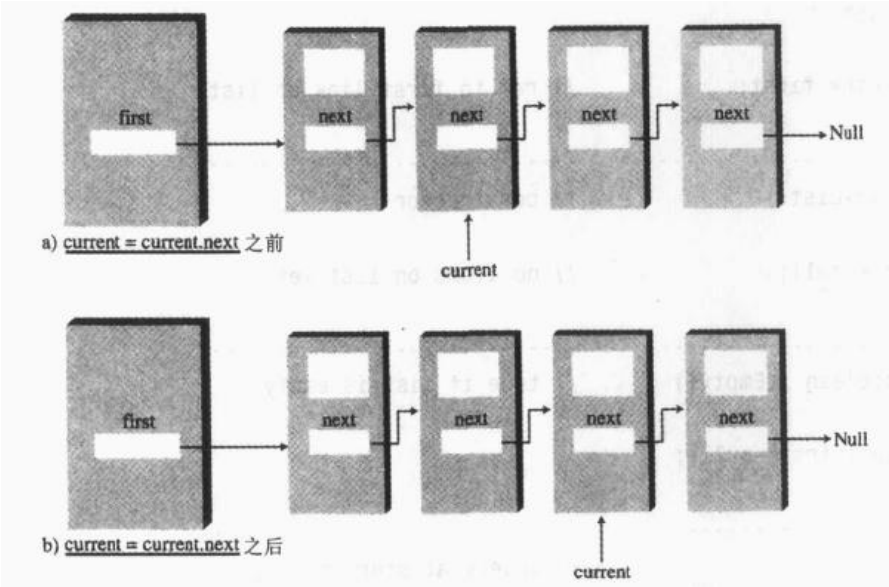


图 5.7 沿着链表步进

在每个链结点，displayList()方法都调用 displayLink()方法显示链结点中的数据。

linkList.java 程序

清单 5.1 给出了整个 linkList.java 程序。前面已经见到了除 main()例程以外的所有部分。

清单 5.1 linkList.java 程序

```
// linkList.java
// demonstrates linked list
// to run this program: C>java LinkListApp
////////////////////////////////////////////////////////////////
class Link
   {
   public int iData;              // data item (key)
   public double dData;           // data item
   public Link next;              // next link in list
// -------------------------------------------------------------
   public Link(int id, double dd) // constructor
      {
      iData = id;                 // initialize data
      dData = dd;                 // ('next' is automatically
      }                           //  set to null)
// -------------------------------------------------------------
   public void displayLink()      // display ourself
      {
      System.out.print("{" + iData + ", " + dData + "} ");
      }
   }  // end class Link
////////////////////////////////////////////////////////////////
class LinkList
   {
   private Link first;            // ref to first link on list

// -------------------------------------------------------------
   public LinkList()              // constructor
      {
      first = null;               // no items on list yet
      }
// -------------------------------------------------------------
   public boolean isEmpty()       // true if list is empty
      {
      return (first==null);
      }
// -------------------------------------------------------------
                                  // insert at start of list
   public void insertFirst(int id, double dd)
```

```
      {                           // make new link
      Link newLink = new Link(id, dd);
      newLink.next = first;       // newLink --> old first
      first = newLink;            // first --> newLink
      }
// ------------------------------------------------------------
   public Link deleteFirst()      // delete first item
      {                           // (assumes list not empty)
      Link temp = first;          // save reference to link
      first = first.next;         // delete it: first-->old next
      return temp;                // return deleted link
      }
// ------------------------------------------------------------
   public void displayList()
      {
      System.out.print("List (first-->last): ");
      Link current = first;       // start at beginning of list
      while(current != null)      // until end of list,
         {
         current.displayLink();   // print data
         current = current.next;  // move to next link
         }
      System.out.println("");
      }
// ------------------------------------------------------------
   } // end class LinkList
////////////////////////////////////////////////////////////////
class LinkListApp
   {
   public static void main(String[] args)
      {
      LinkList theList = new LinkList();  // make new list

      theList.insertFirst(22, 2.99);      // insert four items
      theList.insertFirst(44, 4.99);
      theList.insertFirst(66, 6.99);
      theList.insertFirst(88, 8.99);

      theList.displayList();              // display list

      while( !theList.isEmpty() )         // until it's empty,
         {
         Link aLink = theList.deleteFirst();   // delete link
         System.out.print("Deleted ");         // display it
         aLink.displayLink();
```

```
        System.out.println("");
        }
     theList.displayList();              // display list
     } // end main()
  } // end class LinkListApp
////////////////////////////////////////////////////////////////
```

在 main()方法中，创建了一个新链表，用 insertFirst()方法插入四个新的链结点，并且显示这个链表。然后，在 while 循环中，用 deleteFirst()方法把链结点一个一个地删除，直到链表为空。最后显示空链表。下面是 linkList.java 的输出：

```
List (first-->last): {88, 8.99} {66, 6.99} {44, 4.99} {22, 2.99}
Deleted {88, 8.99}
Deleted {66, 6.99}
Deleted {44, 4.99}
Deleted {22, 2.99}
List (first-->last):
```

查找和删除指定链结点

下一个示例程序增加了两个方法，一个是在链表中查找包含指定关键字的链结点，另一个是删除包含指定关键字的链结点。这些操作和在表头插入一样，都是在 LinkList 专题 applet 中实现的操作。清单 5.2 显示了完整的 linkList2.java 程序。

清单 5.2 linkList2.java 程序

```
// linkList2.java
// demonstrates linked list
// to run this program: C>java LinkList2App
////////////////////////////////////////////////////////////////
class Link
   {
   public int iData;              // data item (key)
   public double dData;           // data item
   public Link next;              // next link in list
// -------------------------------------------------------------
   public Link(int id, double dd) // constructor
      {
      iData = id;
      dData = dd;
      }
// -------------------------------------------------------------
   public void displayLink()      // display ourself
      {
      System.out.print("{" + iData + ", " + dData + "} ");
```

```
      }
   } // end class Link
////////////////////////////////////////////////////////////////
class LinkList
   {
   private Link first;            // ref to first link on list
// -------------------------------------------------------------
   public LinkList()              // constructor
      {
      first = null;               // no links on list yet
      }
// -------------------------------------------------------------
   public void insertFirst(int id, double dd)
      {                           // make new link
      Link newLink = new Link(id, dd);
      newLink.next = first;       // it points to old first link
      first = newLink;            // now first points to this
      }
// -------------------------------------------------------------
   public Link find(int key)      // find link with given key
      {                           // (assumes non-empty list)
      Link current = first;              // start at 'first'
      while(current.iData != key)        // while no match,
         {
         if(current.next == null)        // if end of list,
            return null;                 // didn't find it
         else                            // not end of list,
            current = current.next;      // go to next link
         }
      return current;                    // found it
      }
// -------------------------------------------------------------
   public Link delete(int key)    // delete link with given key
      {                           // (assumes non-empty list)
      Link current = first;              // search for link
      Link previous = first;
      while(current.iData != key)
         {
         if(current.next == null)
            return null;                 // didn't find it
         else
            {
            previous = current;          // go to next link
            current = current.next;
            }
```

```
            }                              // found it
         if(current == first)              // if first link,
            first = first.next;            //    change first
         else                              // otherwise,
            previous.next = current.next;  //    bypass it
         return current;
         }
   // ---------------------------------------------------------
      public void displayList()      // display the list
         {
         System.out.print("List (first-->last): ");
         Link current = first;       // start at beginning of list
         while(current != null)      // until end of list,
            {
            current.displayLink();   // print data
            current = current.next;  // move to next link
            }
         System.out.println("");
         }
   // ---------------------------------------------------------
      }  // end class LinkList
   ////////////////////////////////////////////////////////////////
   class LinkList2App
      {
      public static void main(String[] args)
         {
         LinkList theList = new LinkList();  // make list

         theList.insertFirst(22, 2.99);      // insert 4 items
         theList.insertFirst(44, 4.99);
         theList.insertFirst(66, 6.99);
         theList.insertFirst(88, 8.99);

         theList.displayList();              // display list

         Link f = theList.find(44);          // find item
         if( f != null)
            System.out.println("Found link with key " + f.iData);
         else
            System.out.println("Can't find link");

         Link d = theList.delete(66);        // delete item
         if( d != null )
            System.out.println("Deleted link with key " + d.iData);
         else
```

```
        System.out.println("Can't delete link");

      theList.displayList();              // display list
      } // end main()
   } // end class LinkList2App
////////////////////////////////////////////////////////////////
```

main()例程创建一个链表，插入四个链结点，并显示插入后的链表。然后搜索关键值为 44 的链结点，删除关键值为 66 的链结点，再次显示链表。下面是输出：

```
List (first-->last): {88, 8.99} {66, 6.99} {44, 4.99} {22, 2.99}
Found link with key 44
Deleted link with key 66
List (first-->last): {88, 8.99} {44, 4.99} {22, 2.99}
```

find()方法

find()方法很像 linkList.java 程序中的 displayList()方法。被称为 current 的变量开始时指向 first，然后通过不断地把自己赋值为 current.next，沿着链表向前移动。在每个链结点处，find()检查链结点的关键值是否与它寻找的相等。如果找到了，它返回对该链结点的引用。如果 find()到达链表的尾端，但没有发现要找的链结点，则返回 null。

delete()方法

delete()方法和 find()方法类似，它先搜索要删除的链结点。然而它需要掌握的不仅是指向当前链结点（current）的引用，还有指向当前链结点的前一个（previous）链结点的引用。这是因为，如果要删除当前链结点，必须把前一个链结点和后一个链结点连在一起，如图 5.8 显示。知道前一个链结点位置的惟一方法是拥有一个对它的引用。

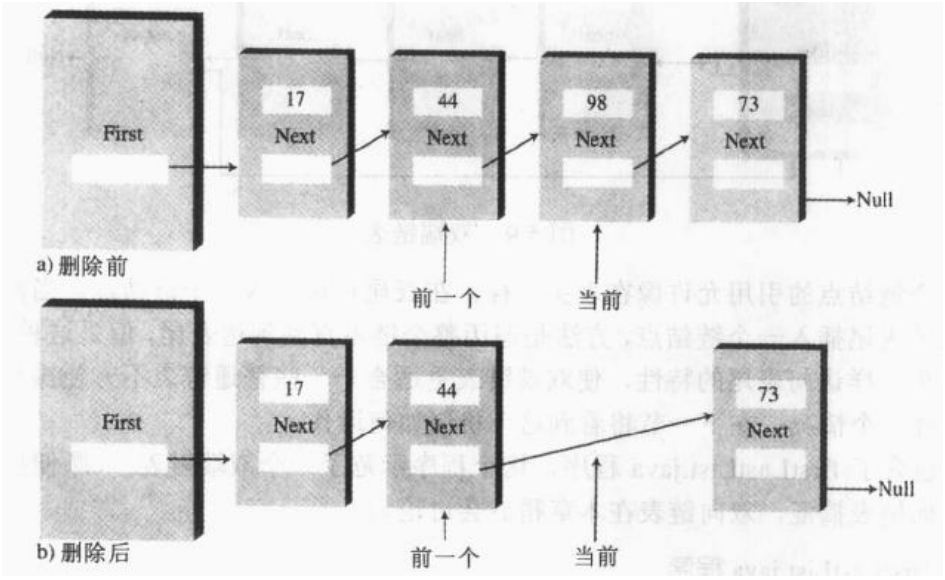


图 5.8 删除一个指定链结点

在 while 语句的每一次轮循环中，每当 current 变量赋值为 current.next 之前，先把 previous 变

量赋值为 current。这保证了它总是指向 current 所指链结点的前一个链结点。

一旦发现当前链结点是要被删除的链结点，就把前一个链结点的 next 字段赋值为当前链结点的下一个链结点。如果当前链结点是第一个链结点，这是一种特殊情况，因为这是由 LinkList 对象的 first 域指向的链结点，而不是别的链结点的 next 字段所指的。在这种情况下，使 first 字段指向 first.next，就可以删除第一个链结点，正如在 linkList.java 程序的 deleteFirst()方法中看到的那样。下面是涵盖了两种可能性的代码：

```
                                    // found it
if(current == first)                // if first link,
   first = first.next;              //    change first
else                                // otherwise,
   previous.next = current.next;    //    bypass link
```

其他方法

前面已经看到在表头插入和删除链结点的方法，以及查找和删除一个指定链结点的方法。还可以再想想其他有用的链表方法。例如，insertAfter()方法可以查找一个含有特定关键值的链结点，然后在它后面插入一个新的链结点。在本章最后讨论迭代器的时候，会看到这样一个方法。

双端链表

双端链表与传统的链表非常相似，但是它有一个新增的特性：即对最后一个链结点的引用，就像对第一个链结点的引用一样。图 5.9 显示了这么一个链表。

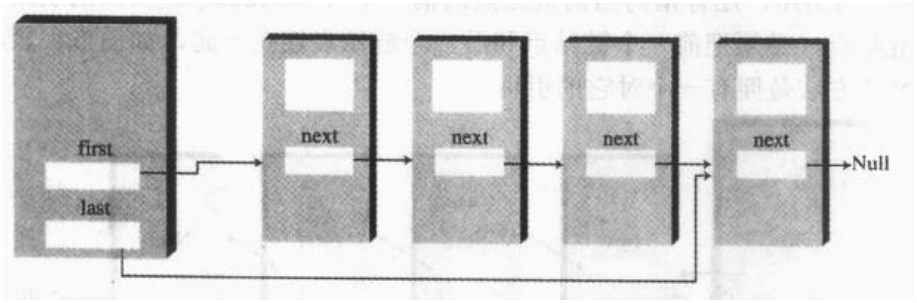


图 5.9 双端链表

对最后一个链结点的引用允许像在表头一样，在表尾直接插入一个链结点。当然，仍然可以在普通的单链表的表尾插入一个链结点，方法是遍历整个链表直到到达表尾，但是这种方法效率很低。

像访问表头一样访问表尾的特性，使双端链表更适合于一些普通链表不方便操作的场合，队列的实现就是这样一个情况；在下一节将看到这个机制如何运作。

清单 5.3 包含了 firstLastList.java 程序，这个程序示范了一个双端链表。（顺便提一句，不要把双端链表和双向链表搞混，双向链表在本章稍后会讨论。）

清单 5.3 firstLastList.java 程序

```
// firstLastList.java
// demonstrates list with first and last references
```

```
// to run this program: C>java FirstLastApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;                  // data item
   public Link next;                   // next link in list
// -------------------------------------------------------------
   public Link(long d)                 // constructor
      { dData = d; }
// -------------------------------------------------------------
   public void displayLink()           // display this link
      { System.out.print(dData + " "); }
// -------------------------------------------------------------
   }  // end class Link
////////////////////////////////////////////////////////////////
class FirstLastList
   {
   private Link first;                 // ref to first link
   private Link last;                  // ref to last link
// -------------------------------------------------------------
   public FirstLastList()              // constructor
      {
      first = null;                    // no links on list yet
      last = null;
      }
// -------------------------------------------------------------
   public boolean isEmpty()            // true if no links
      { return first==null; }
// -------------------------------------------------------------
   public void insertFirst(long dd)    // insert at front of list
      {
      Link newLink = new Link(dd);     // make new link

      if( isEmpty() )                  // if empty list,
         last = newLink;               // newLink <-- last
      newLink.next = first;            // newLink --> old first
      first = newLink;                 // first --> newLink
      }
// -------------------------------------------------------------
   public void insertLast(long dd)     // insert at end of list
      {
      Link newLink = new Link(dd);     // make new link
      if( isEmpty() )                  // if empty list,
         first = newLink;              // first --> newLink
      else
```

```
        last.next = newLink;        // old last --> newLink
      last = newLink;               // newLink <-- last
      }
// ---------------------------------------------------------
   public long deleteFirst()        // delete first link
      {                             // (assumes non-empty list)
      long temp = first.dData;
      if(first.next == null)        // if only one item
         last = null;               // null <-- last
      first = first.next;           // first --> old next
      return temp;
      }
// ---------------------------------------------------------
   public void displayList()
      {
      System.out.print("List (first-->last): ");
      Link current = first;         // start at beginning
      while(current != null)        // until end of list,
         {
         current.displayLink();     // print data
         current = current.next;    // move to next link
         }
      System.out.println("");
      }
// ---------------------------------------------------------
   } // end class FirstLastList
////////////////////////////////////////////////////////////////
class FirstLastApp
   {
   public static void main(String[] args)
      {                             // make a new list
      FirstLastList theList = new FirstLastList();

      theList.insertFirst(22);      // insert at front
      theList.insertFirst(44);
      theList.insertFirst(66);

      theList.insertLast(11);       // insert at rear
      theList.insertLast(33);
      theList.insertLast(55);

      theList.displayList();        // display the list

      theList.deleteFirst();        // delete first two items
      theList.deleteFirst();
```

```
      theList.displayList();          // display again
      } // end main()
   } // end class FirstLastApp
//////////////////////////////////////////////////////////////
```

为了简单起见，在这个程序中，把每个链结点中的数据字段个数从两个压缩到一个。这更容易显示链结点的内容。（记住，在一个正式的程序中，可能会有非常多的数据字段，或者对另外一个对象的引用，那个对象也包含很多数据字段。）

这个程序在表头和表尾各插入三个链点，显示插入后的链表。然后删除头两个链结点，再次显示。下面是输出：

```
List (first-->last): 66 44 22 11 33 55
List (first-->last): 22 11 33 55
```

注意在表头重复插入操作会颠倒链结点进入的顺序，而在表尾的重复插入则保持链结点进入的顺序。

双端链表类叫做 FirstLastList。正如前面讨论的，它有两个项，first 和 last，一个指向链表中的第一个链结点，另一个指向最后一个链结点。如果链表中只有一个链结点，first 和 last 就都指向它，如果没有链结点，两者都为 null 值。

这个类有一个新的方法 insertLast()，这个方法在表尾插入一个新的链结点。这个过程首先改变 last.next，使其指向新生成的链结点，然后改变 last，使其指向新的链结点，如图 5.10 所示。

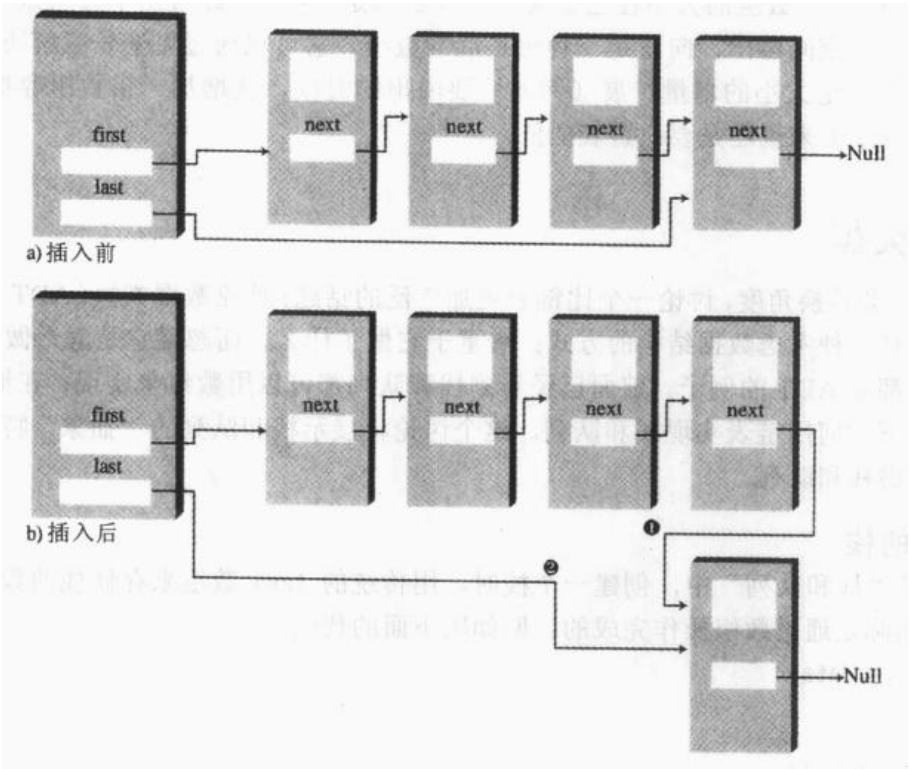


图 5.10 在表尾插入新链结点

插入和删除方法和普通链表的相应部分类似。然而，两个插入方法都要考虑一种特殊情况，即插入前链表是空的。如果 isEmpty()是真，那么 insertFirst()必须把 last 指向新的链结点，insertLast()也必须把 first 指向新的链结点。

如果用 insertFirst()方法实现在表头插入，first 就指向新的链结点，用 insertLast()方法实现在表尾插入，last 就指向新的链结点。如果链表只有一个链结点，那么从表头删除也是一种特殊情况：last 必须被赋值为 null 值。

不幸的是，用双端链表也不能有助于删除最后一个链结点，因为没有一个引用指向倒数第二个链结点。如果最后一个链结点被删除，倒数第二个链结点的 next 字段应该变成 null 值。为了方便的删除最后一个链结点，需要一个双向链表，马上将会讨论到它。（当然，也可以遍历整个链表找到最后一个链结点，但是那样做效率不是很高。）

链表的效率

在表头插入和删除速度很快。仅需要改变一两个引用值，所以花费 O(1)的时间。

平均起来，查找、删除和在指定链结点后面插入都需要搜索链表中的一半链结点。需要 O(N)次比较。在数组中执行这些操作也需要 O(N)次比较，但是链表仍然要快一些，因为当插入和删除链结点时，链表不需要移动任何东西。增加的效率是很显著的，特别是当复制时间远远大于比较时间的时候。

当然，链表比数组优越的另外一个重要方面是链表需要多少内存就可以用多少内存，并且可以扩展到所有可用内存。数组的大小在它创建的时候就固定了；所以经常由于数组太大导致效率低下，或者数组太小导致空间溢出。向量是一种可扩展的数组，它可以通过可变长度解决这个问题，但是它经常只允许以固定大小的增量扩展（例如快要溢出的时候，就增加一倍数组容量）。这个解决方案在内存使用效率上来说还是要比链表的低。

抽象数据类型

在本节，将会转换角度，讨论一个比链表更加广泛的话题：抽象数据类型（ADT）。什么是 ADT？简单说来，它是一种考虑数据结构的方式：着重于它做了什么，而忽略它是怎么做的。

栈和队列都是 ADT 的例子。前面已经看到栈和队列都可以用数组来实现。在继续 ADT 的讨论之前，先看一下如何用链表实现栈和队列。这个讨论将展示栈和队列的“抽象”特性：即如何脱离具体实现来考虑栈和队列。

用链表实现的栈

在第 4 章“栈和队列”中，创建一个栈时，用传统的 Java 数组来存储栈的数据。栈的 push()和 pop()操作实际是通过数组操作完成的。例如用下面的代码

```
arr[++top] = data;
```

和

```
data = arr[top--];
```

在数组中插入数据，或从数组中删除数据。

当然也可以用链表存储数据。在这种情况下，push()和 pop()操作通过类似于下面的操作来执行

```
theList.insertFirst(data)
```

和

```
data = theList.deleteFirst()
```

栈类的使用者调用 push()和 pop()方法来插入和删除栈中元素，他们不知道，也不需要知道栈是用链表还是用队列实现的。清单 5.4 显示了一个名为 LinkStack 的栈类是如何用 LinkList 类而不是用数组实现的。（纯的面向对象程序员可能认为 LinkStack 应该简单的称为 Stack，因为这个类的使用者不需要知道它是由链表实现的。）

清单 5.4 linkStack.java 程序

```
// linkStack.java
// demonstrates a stack implemented as a list
// to run this program: C>java LinkStackApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;              // data item
   public Link next;               // next link in list
// -------------------------------------------------------------
   public Link(long dd)            // constructor
      { dData = dd; }
// -------------------------------------------------------------
   public void displayLink()       // display ourself
      { System.out.print(dData + " "); }
   }  // end class Link
////////////////////////////////////////////////////////////////
class LinkList
   {
   private Link first;             // ref to first item on list
// -------------------------------------------------------------
   public LinkList()               // constructor
      { first = null; }            // no items on list yet
// -------------------------------------------------------------
   public boolean isEmpty()        // true if list is empty
      { return (first==null); }
// -------------------------------------------------------------
   public void insertFirst(long dd) // insert at start of list
      {                            // make new link
      Link newLink = new Link(dd);
      newLink.next = first;        // newLink --> old first
      first = newLink;             // first --> newLink
      }
```

```
// --------------------------------------------------------
   public long deleteFirst()      // delete first item
      {                           // (assumes list not empty)
      Link temp = first;          // save reference to link
      first = first.next;         // delete it: first-->old next
      return temp.dData;          // return deleted link
      }
// --------------------------------------------------------
   public void displayList()
      {
      Link current = first;       // start at beginning of list
      while(current != null)      // until end of list,
         {
         current.displayLink();   // print data
         current = current.next;  // move to next link
         }
      System.out.println("");
      }
// --------------------------------------------------------
   }  // end class LinkList
////////////////////////////////////////////////////////////////
class LinkStack
   {
   private LinkList theList;
//--------------------------------------------------------------
   public LinkStack()             // constructor
      {
      theList = new LinkList();
      }
//--------------------------------------------------------------
   public void push(long j)     // put item on top of stack
      {
      theList.insertFirst(j);
      }
//--------------------------------------------------------------
   public long pop()            // take item from top of stack
      {
      return theList.deleteFirst();
      }
//--------------------------------------------------------------
   public boolean isEmpty()       // true if stack is empty
      {
      return ( theList.isEmpty() );
      }
//--------------------------------------------------------------
```

```
   public void displayStack()
      {
      System.out.print("Stack (top-->bottom): ");
      theList.displayList();
      }
//--------------------------------------------------------------
   }  // end class LinkStack
////////////////////////////////////////////////////////////////
class LinkStackApp
   {
   public static void main(String[] args)
      {
      LinkStack theStack = new LinkStack(); // make stack
      theStack.push(20);                    // push items
      theStack.push(40);

      theStack.displayStack();              // display stack

      theStack.push(60);                    // push items
      theStack.push(80);

      theStack.displayStack();              // display stack

      theStack.pop();                       // pop items
      theStack.pop();

      theStack.displayStack();              // display stack
      }  // end main()
   }  // end class LinkStackApp
////////////////////////////////////////////////////////////////
```

main()方法创建了一个栈对象，压入了两个元素，显示这个栈，又压入两个元素，然后再次显示。最后，它弹出两个元素，第三次显示栈的内容。下面是输出：

```
Stack (top-->bottom): 40 20
Stack (top-->bottom): 80 60 40 20
Stack (top-->bottom): 40 20
```

注意整个程序的组织。LinkStackApp 类中的 main()方法只和 LinkStack 类有关。LinkStack 类只和 LinkList 类有关。main()方法和 LinkList 类是不进行通信的。

更特别的是，当 main()方法中的一语句调用 LinkStack 类中的 push()操作时，这个方法就调用 LinkList 类中的 insertFirst()方法插入数据。类似的，pop()方法调用 deleteFirst()删除一个数据项，displayStack()方法调用 displayList()方法显示栈。使用基于链表的 LinkStack 类还是使用基于数组的 stack 类来书写 main()方法中的代码，对于类的使用者来说是没有分别的。其中基于数组的 stack 类见第 4 章中的 stack.java 程序（清单 4.1）。

用链表实现的队列

下面是一个用链表实现 ADT 类似的例子。清单 5.5 显示了一个用双端链表实现的队列。

清单 5.5 linkQueue.java 程序

```
// linkQueue.java
// demonstrates queue implemented as double-ended list
// to run this program: C>java LinkQueueApp
////////////////////////////////////////////////////////////////
class Link
  {
  public long dData;                // data item
  public Link next;                 // next link in list
// -------------------------------------------------------------
  public Link(long d)               // constructor
    { dData = d; }
// -------------------------------------------------------------
  public void displayLink()         // display this link
    { System.out.print(dData + " "); }
// -------------------------------------------------------------
  } // end class Link
////////////////////////////////////////////////////////////////
class FirstLastList
  {
  private Link first;               // ref to first item
  private Link last;                // ref to last item
// -------------------------------------------------------------
  public FirstLastList()            // constructor
    {
    first = null;                   // no items on list yet
    last = null;
    }
// -------------------------------------------------------------
  public boolean isEmpty()          // true if no links
    { return first==null; }
// -------------------------------------------------------------
  public void insertLast(long dd) // insert at end of list
    {
    Link newLink = new Link(dd);    // make new link
    if( isEmpty() )                 // if empty list,
      first = newLink;              // first --> newLink
    else
      last.next = newLink;          // old last --> newLink
    last = newLink;                 // newLink <-- last
    }
// -------------------------------------------------------------
```

```
   public long deleteFirst()          // delete first link
      {                               // (assumes non-empty list)
      long temp = first.dData;
      if(first.next == null)          // if only one item
         last = null;                 // null <-- last
      first = first.next;             // first --> old next
      return temp;
      }
// ---------------------------------------------------------
   public void displayList()
      {
      Link current = first;           // start at beginning
      while(current != null)          // until end of list,
         {
         current.displayLink();       // print data
         current = current.next;      // move to next link
         }
      System.out.println("");
      }
// ---------------------------------------------------------
   } // end class FirstLastList
////////////////////////////////////////////////////////////////
class LinkQueue
   {
   private FirstLastList theList;
//---------------------------------------------------------
   public LinkQueue()                 // constructor
      { theList = new FirstLastList(); }  // make a 2-ended list
//---------------------------------------------------------
   public boolean isEmpty()           // true if queue is empty
      { return theList.isEmpty(); }
//---------------------------------------------------------
   public void insert(long j)         // insert, rear of queue
      { theList.insertLast(j); }
//---------------------------------------------------------
   public long remove()               // remove, front of queue
      {  return theList.deleteFirst();  }
//---------------------------------------------------------
   public void displayQueue()
      {
      System.out.print("Queue (front-->rear): ");
      theList.displayList();
      }
//---------------------------------------------------------
   } // end class LinkQueue
```

```
////////////////////////////////////////////////////////////////
class LinkQueueApp
  {
  public static void main(String[] args)
    {
    LinkQueue theQueue = new LinkQueue();
    theQueue.insert(20);                // insert items
    theQueue.insert(40);

    theQueue.displayQueue();            // display queue

    theQueue.insert(60);                // insert items
    theQueue.insert(80);

    theQueue.displayQueue();            // display queue

    theQueue.remove();                  // remove items
    theQueue.remove();

    theQueue.displayQueue();            // display queue
    }  // end main()
////////////////////////////////////////////////////////////////
```

这个程序创建了一个队列，插入两个元素，再插入两个元素，然后删掉两个元素；每一步做完后显示队列。下面是输出：

```
Queue (front-->rear): 20 40
Queue (front-->rear): 20 40 60 80
Queue (front-->rear): 60 80
```

这里的 insert()方法和 remove()方法是由 firstLastList 类中的 insertLast()方法和 deleteFirst()方法实现的。这里已经用链表替代数组实现了队列，而在第 4 章的 queue.java 程序（清单 4.4）中，是用数组来实现队列的。

linkStack.java 和 linkQueue.java 程序强调栈和队列是概念上的实体，独立于它们的具体实现。用数组或是用链表实现栈都是一样的。栈的重要性在于它的 push()操作和 pop()操作，以及如何使用它们；而不是实现这些操作的内在机制。

什么时候应该使用链表而不是队列来实现栈或队列呢？这一点要取决于是否能精确地预测栈或队列需要容纳的数据量。如果这点不甚清楚，链表就比数组表现出更好的适应性。两者都很快，所以速度可能不是考虑重点。

数据类型和抽象

“抽象数据类型”这个术语从何而来？首先看看“数据类型”这部分，再来考虑“抽象”。

数据类型

“数据类型”一词用在很多地方。它首先表示内置的类型，例如 int 型和 double 型。这可能是

听到这个词后首先想到的。

当谈论一个简单类型时，实际上涉及到两件事：拥有特定特征的数据项和在数据上允许的操作。例如，Java 中的 int 型变量是整数，取值范围在-2147483648 和+2147483647 之间，还能用各种操作符，+、-、*、/等等对其进行操作。数据类型允许的操作是它本身不可分离的部分，理解类型包括理解什么样的操作可以应用在该类型之上。

随着面向对象编程的出现，现在可以用类来创建自己的数据类型。这些数据类型中的一部分表示数值量，使用的方式和基本类型类似。例如，可以为时间定义一个类（拥有小时、分钟和秒等字段），为分数定义一个类（拥有分子和分母等字段），为超长数字定义一个类（字符串中的字符表示数字）。所有这些类都可以像 int 型和 double 型一样做加法和减法，只是在 Java 语言中必须使用方法而不是像‘+’或‘-’这样的操作符，这些方法使用类似于 add()和 sub()的函数标记符号。

“数据类型”这个短语好像很自然的适合那些数值量的类。然而，它也适用于没有这些量化属性的类。事实上，“任何”类都代表一个数据类型，从这个意义上来说，类是由数据（字段）和数据上允许的操作（方法）组成的。

更广泛一点，当一个数据存储结构（例如栈和队列）被表示为一个类时，它也成为了一个数据类型。栈和 int 类型在很多方面都不同，但它们都被定义为一组具有一定排列规律的数据和在此数据上的操作集合。

抽象

抽象这个词的意思是“不考虑细节的描述和实现。”抽象是事物的本质和重要特征。例如，总统办公室是一个抽象，它并不考虑哪一个人碰巧成为这个办公室的主人。总统办公室的权力和责任依旧，但是总统会随着任职和去职而来去。

因此，在面向对象编程中，一个抽象数据类型是一个类，且不考虑它的实现。它是对类中数据（域）的描述和能够在数据上执行的一系列操作（方法）以及如何使用这些操作的说明。每个方法如何执行任务的细节肯定不包括在内。作为类的用户，只会被告知可以调用哪些方法，如何调用它们，以及可望得到的结果，但是不包括内部如何运作的。

当“抽象数据类型”用于像栈和队列这样的数据结构时，它的意义被进一步扩展了。和其他类一样，它意味着数据和在数据上执行的操作，即使在这种情况下，如何存储数据的基本原则对于类用户来说也是不可见的。用户不仅不知道方法怎样运作，也不知道数据是如何存储的。

对于栈来说，用户只知道 push()方法和 pop()方法（也许还有一些其他方法）的存在和如何用它们工作。用户不需要（至少不经常需要）知道方法内部如何运作，或者数据是否存储在数组里、链表里或是例如树等其他数据结构中。

接口

ADT 有一个经常被叫做“接口”的规范。它是给类用户看的，通常是类的公有方法。在栈中，push()方法、pop()方法和其他类似的方法形成了接口。

ADT 列表

现在已经了解了什么是抽象数据类型，所以可以谈及另外一个术语：“列表”。列表（有时叫做线性表）是一组线性排列的数据项。也就是说，它们以一定的方式串接起来，像一根线上的珠子或一条街上的房子。列表支持一定的基本操作。可以插入某一项，删除某一项，还有经常从某个特定

位置读出一项（例如，读出第三项）。

不要把本章讲的链表和 ADT 表搞混。列表由它的接口定义：即用于和它交互的特定方法。这个接口可以用不同的结构实现，包括数组、链表。列表是这些数据结构的抽象。

作为设计工具的 ADT

ADT 的概念在软件设计过程中也是有用的。如果需要存储数据，那么就从考虑需要在数据上实现的操作开始。需要存取最后一个插入的数据项吗？还是第一个？是特定值的项？还是在特定位置的项？回答这些问题会引出 ADT 的定义。只有在完整定义了 ADT 后，才应该考虑细节问题，例如如何表示数据，如何编码使方法可以存取数据等等。

通过从 ADT 规范中剔除实现的细节，可以简化设计过程。在未来的某个时刻，易于修改实现。如果用户只接触 ADT 接口，应该可以在不“干扰”用户代码的情况下修改接口的实现。

当然，一旦设计好 ADT，必须仔细选择内部的数据结构，以使规定的操作的效率尽可能高。例如，如果需要随机存取元素 N，那么用链表表示就不够好，因为对链表来说，随机访问不是一个高效的操作。选择数组会得到较好的效果。

注意

记住，ADT 只是一个概念性的工具。数据存储结构不能截然地分成一些是 ADT，另一些用来实现 ADT。例如，链表不需要封装成一个列表接口才可用；它本身就可以作为一个 ADT，然而它也可以实现其他数据类型，例如队列。链表能够用数组来实现，一个数组类型的结构也可以用链表实现。在给定环境下，必须确定什么是 ADT，什么是更基本的结构。

有序链表

讨论至今，在链表中，还没有要求数据有序存储。然而对于某些应用来说，在链表中保持数据有序是有用的。具有这个特性的链表叫作“有序链表”。

在有序链表中，数据是按照关键值有序排列的。有序链表的删除常常是只限于删除在链表头部的最小（或者最大）链结点。不过，有时也用 find()方法和 delete()方法在整个链表中搜索某一特定点。

一般，在大多数需要使用有序数组的场合也可以使用有序链表。有序链表优于有序数组的地方是插入的速度（因为元素不需要移动），另外链表可以扩展到全部有效的使用内存，而数组只能局限于一个固定的大小中。但是，有序链表实现起来比有序数组更困难一些。

后面将看到一个有序链表的应用：为数据排序。有序链表也可以用于实现优先级队列，尽管堆是更常用的实现方法（参考第 12 章）。

在本章开头已经引入的 LinkList 专题 applet 不仅能演示普通链表，也能演示有序链表。现在观察有序链表如何运作：用 New 按钮创建一个有 20 个链结点的新链表，当提示出现时，点击“Sorted”选项。结果就出现一个数据有序排列的链表，如图 5.11 所示。

用 Ins 按钮插入一个新的数据项。键入一个数值，它的大小最好使新生成的链结点能落在链表中间的某个地方。算法遍历链表，寻找合适的插入位置的过程。找到正确位置后就插入新的链结点，如图 5.12 所示。

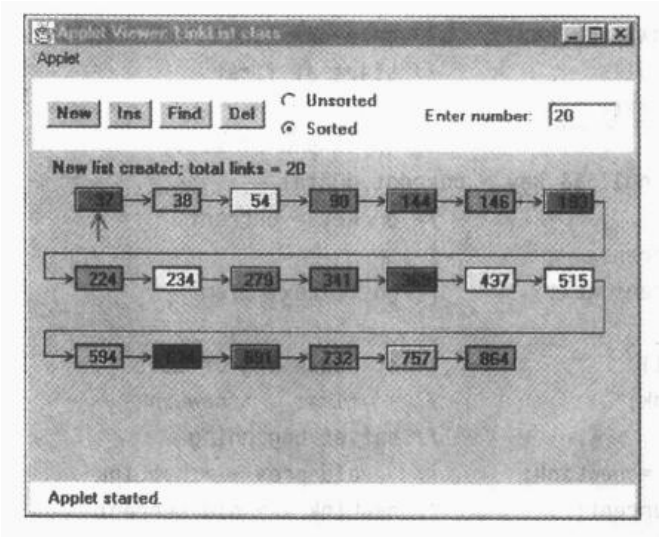


图 5.11 有序链表的 LinkList 专题 applet

再次点击 Ins 按钮，链表被重画以使它更美观。也可以用 Find 按钮找到特定链结点，或者用 Del 按钮删除特定链结点。

在有序链表中插入一个数据项的 Java 代码

为了在一个有序链表中插入数据项，算法必须首先搜索链表，直到找到合适的位置：它恰好在第一个比它大的数据项的前面，如图 5.12 所示。

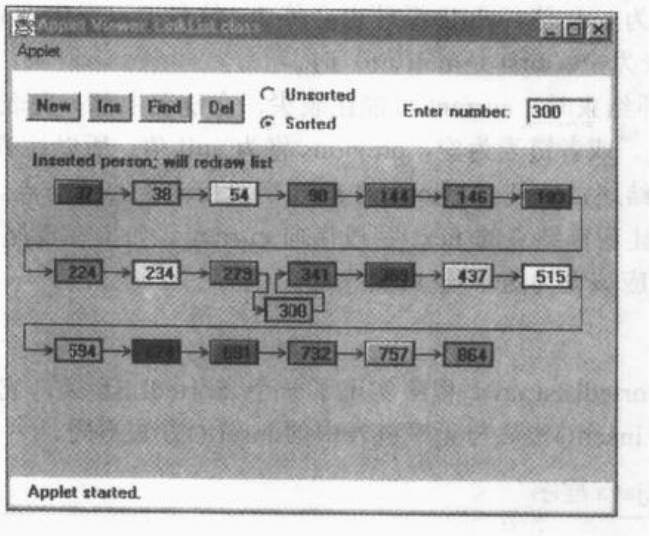


图 5.12 新插入的链结点

当算法找到了要插入的位置，用通常的方式插入数据项：把新链结点的 next 字段指向下一个链结点，然后把前一个链结点的 next 字段改为指向新的链结点。然而，需要考虑一些特殊情况：链结点有可能插在表头，或者插在表尾。看一下这段代码：

```
public void insert(long key) // insert in order
   {
```

```
Link newLink = new Link(key);     // make new link
Link previous = null;             // start at first
Link current = first;
                                  // until end of list,
while(current != null && key > current.dData)
   {                              // or key > current,
   previous = current;
   current = current.next;        // go to next item
   }
if(previous==null)                // at beginning of list
   first = newLink;               //    first --> newLink
else                              // not at beginning
   previous.next = newLink;       //    old prev --> newLink
newLink.next = current;           // newLink --> old current
} // end insert()
```

在链表上移动时，需要用一个 previous 引用，这样才能把前一个链结点的 next 字段指向新的链结点。创建新链结点后，把 current 变量设为 first，准备搜索正确的插入点。这时也把 previous 设为 null 值，这步操作很重要，因为后面要用这个 null 值判断是否仍在表头。

while 循环和以前用来搜索插入点的代码类似，但是有一个附加的条件。如果当前检查的链结点的关键值（current.dData）不再小于待插入的链结点的关键值（key），则循环结束；这是最常见的情况，即新关键值插在链表中部的某个地方。

然而，如果 current 为 null 值，while 循环也会停止。这种情况发生在表尾（最后一个元素的 next 域为 null 值），或者链表为空（first 是 null 值）时。

那么，当 while 循环结束时，current 可能在表头，在链表中部，在表尾，或者链表是空的。

如果 current 在表头，或者链表为空，previous 将为 null 值；所以让 first 指向新的链结点。否则 current 处在链表中部或结尾，就使 previous 的 next 字段指向新的链结点。

不论哪种情况，都让新链结点的 next 字段指向 current。如果在表尾，current 为 null 值，则新链结点的 next 字段也本应该设为这个值（null）。

sortedList.java 程序

清单 5.6 显示的 sortedLIst.java 程序实现了一个 SortedList 类，它拥有 insert()、remove()和 displayList()方法。只有 insert()方法与无序链表中的 insert()方法不同。

清单 5.6 sortedList.java 程序

```
// sortedList.java
// demonstrates sorted list
// to run this program: C>java SortedListApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;                 // data item
   public Link next;                  // next link in list
```

```
// ------------------------------------------------------
   public Link(long dd)                // constructor
      { dData = dd; }
// ------------------------------------------------------
   public void displayLink()           // display this link
      { System.out.print(dData + " "); }
   } // end class Link
////////////////////////////////////////////////////////////////
class SortedList
   {
   private Link first;                 // ref to first item on list
// ------------------------------------------------------
   public SortedList()                 // constructor
      { first = null; }
// ------------------------------------------------------
   public boolean isEmpty()            // true if no links
      { return (first==null); }
// ------------------------------------------------------
   public void insert(long key)        // insert, in order
      {
      Link newLink = new Link(key);    // make new link
      Link previous = null;            // start at first
      Link current = first;
                                       // until end of list,
      while(current != null && key > current.dData)
         {                             // or key > current,
         previous = current;
         current = current.next;       // go to next item
         }
      if(previous==null)               // at beginning of list
         first = newLink;              // first --> newLink
      else                             // not at beginning
         previous.next = newLink;      // old prev --> newLink
      newLink.next = current;          // newLink --> old current
      } // end insert()
// ------------------------------------------------------
   public Link remove()           // return & delete first link
      {                           // (assumes non-empty list)
      Link temp = first;               // save first
      first = first.next;              // delete first
      return temp;                     // return value
      }
// ------------------------------------------------------
   public void displayList()
      {
```

```
      System.out.print("List (first-->last): ");
      Link current = first;       // start at beginning of list
      while(current != null)      // until end of list,
         {
         current.displayLink();  // print data
         current = current.next; // move to next link
         }
      System.out.println("");
      }
   }  // end class SortedList
////////////////////////////////////////////////////////////////
class SortedListApp
   {
   public static void main(String[] args)
      {                           // create new list
      SortedList theSortedList = new SortedList();
      theSortedList.insert(20);    // insert 2 items
      theSortedList.insert(40);

      theSortedList.displayList(); // display list

      theSortedList.insert(10);    // insert 3 more items
      theSortedList.insert(30);
      theSortedList.insert(50);

      theSortedList.displayList(); // display list

      theSortedList.remove();      // remove an item

      theSortedList.displayList(); // display list
      }  // end main()
   }  // end class SortedListApp
////////////////////////////////////////////////////////////////
```

在 main()方法中，插入值为 20 和 40 的两个链结点。然后再插入三个链结点，分别是 10、30 和 50。这三个值分别插在表头、表中和表尾。这说明 insert()方法正确地处理了特殊情况。最后删除了一个链结点，表现出删除操作总是从表头进行。每一步变化后，都显示整个链表。下面是 sortedList.java 程序的输出：

```
List (first-->last): 20 40
List (first-->last): 10 20 30 40 50
List (first-->last): 20 30 40 50
```

有序链表的效率

在有序链表插入和删除某一项最多需要 O(N)次比较（平均 N/2），因为必须沿着链表上一步一

步走才能找到正确的位置。然而，可以在 O(1)的时间内找到或删除最小值，因为它总在表头。如果一个应用频繁地存取最小项，且不需要快速的插入，那么有序链表是一个有效的方案选择。例如，优先级队列可以用有序链表来实现。

表插入排序

有序链表可以用于一种高效的排序机制。假设有一个无序数组。如果从这个数组中取出数据，然后一个一个地插入有序链表，它们自动地按顺序排列。把它们从有序表中删除，重新放入数组，那么数组就会排好序了。

这种排序方式总体上比在数组中用常用的插入排序效率更高一些，这是因为这种方式进行的复制次数少一些，用数组的插入排序算法在第 3 章“简单排序”中有所描述。它仍然是一个时间级为 $O(N^2)$的过程，因为在有序链表中每插入一个新的链结点，平均要与一半已存在数据进行比较，如果插入 N 个新数据，就进行了 $N^2/4$ 次比较。每一个链结点只进行两次复制：一次从数组到链表，一次从链表到数组。在数组中进行插入排序需要 N^2 次移动，相比之下，2*N 次移动更好。

清单 5.7 显示了 listInsertionSort.java 程序，它创建一个 link 类型的无序数组，把它们插入到有序链表中（使用构造函数），然后再从链表中删除，放回原数组。

清单 5.7 listInsertionSort.java 程序

```
// listInsertionSort.java
// demonstrates sorted list used for sorting
// to run this program: C>java ListInsertionSortApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;                  // data item
   public Link next;                   // next link in list
// -------------------------------------------------------------
   public Link(long dd)                // constructor
      { dData = dd; }
// -------------------------------------------------------------
   } // end class Link
////////////////////////////////////////////////////////////////
class SortedList
   {
   private Link first;            // ref to first item on list
// -------------------------------------------------------------
   public SortedList()            // constructor (no args)
      { first = null; }                   // initialize list
// -------------------------------------------------------------
   public SortedList(Link[] linkArr)  // constructor (array
      {                               // as argument)
      first = null;                       // initialize list
      for(int j=0; j<linkArr.length; j++)  // copy array
         insert( linkArr[j] );            // to list
```

```
    }
// ------------------------------------------------------------
  public void insert(Link k)     // insert (in order)
    {
    Link previous = null;            // start at first
    Link current = first;
                                     // until end of list,
    while(current != null && k.dData > current.dData)
       {                             // or key > current,
       previous = current;
       current = current.next;       // go to next item
       }
    if(previous==null)               // at beginning of list
       first = k;                    // first --> k
    else                             // not at beginning
       previous.next = k;            // old prev --> k
    k.next = current;                // k --> old current
    } // end insert()
// ------------------------------------------------------------
  public Link remove()           // return & delete first link
    {                            // (assumes non-empty list)
    Link temp = first;               // save first
    first = first.next;              // delete first
    return temp;                     // return value
    }
// ------------------------------------------------------------
  } // end class SortedList
////////////////////////////////////////////////////////////////
class ListInsertionSortApp
  {
  public static void main(String[] args)
    {
    int size = 10;
                                 // create array of links
    Link[] linkArray = new Link[size];

    for(int j=0; j<size; j++)  // fill array with links
       {                             // random number
       int n = (int)(java.lang.Math.random()*99);
       Link newLink = new Link(n);  // make link
       linkArray[j] = newLink;      // put in array
       }
                                 // display array contents
    System.out.print("Unsorted array: ");
    for(int j=0; j<size; j++)
```

```
      System.out.print( linkArray[j].dData + " " );
    System.out.println("");
                              // create new list
                              // initialized with array
    SortedList theSortedList = new SortedList(linkArray);

    for(int j=0; j<size; j++)  // links from list to array
      linkArray[j] = theSortedList.remove();
                              // display array contents
    System.out.print("Sorted Array:   ");
    for(int j=0; j<size; j++)
      System.out.print(linkArray[j].dData + " ");
    System.out.println("");
    }  // end main()
  }  // end class ListInsertionSortApp
////////////////////////////////////////////////////////////////
```

这个程序显示了数组排序前后的内容。下面是这个例子的输出：

```
Unsorted array: 59 69 41 56 84 15 86 81 37 35
Sorted array:   15 35 37 41 56 59 69 81 84 86
```

每次的输出是不同的，因为原始数据是随机生成的。

SortedList 类的新构造函数把 Link 对象的数组作为参数读入，然后把整个数组内容插入到新创建的链表中。这样做以后，有助于简化客户（main()方法）的工作。

这个程序的 insert()方法也有一些变化。它现在接收 Link 对象作为参数，而不是 long 型对象。这样做的目的是为了能够在数组中存储 Link 对象，并能把它们直接插入到链表中。在 sortedList.java 程序中（清单 5.6），把 long 型的关键值作为参数，由 insert()方法创建每个 Link 对象，这样更方便。

和基于数组的插入排序相比，表插入排序有一个缺点，就是它要开辟差不多两倍的空间：数组和链表必须同时在内存中存在。但如果有现成的有序链表类可用，那么用表插入排序对不太大的数组排序是比较便利的。

双向链表

下面来讨论另一种链表的变型：双向链表（不要和双端链表产生混淆）。双向链表有什么优点呢？传统链表的一个潜在问题是沿链表的反向遍历是困难的。用这样一个语句

```
current=current.next
```

可以很方便地到达下一个链结点，然而没有对应的方法回到前一个链结点。根据应用的不同，这个限制可能会引起问题。

例如，假设一个文本编辑器用链表来存储文本。屏幕上的每一行文字作为一个 String 对象存储在链结点中。当编辑器用户向下移动光标时，程序移到下一个链结点操纵或显示新的一行。但是如果用户向上移动光标会发生什么呢？在普通的链表中，需要把 current 变量（或起同等作用的其他变量）调回到表头，再一步一步地向后步进到新的当前链结点，这样效率非常低。因此需要一个简单

的向回头方向走一步的操作。

双向链表提供了这个能力。即允许向前遍历，也允许向后遍历整个链表。其中秘密在于每个链结点有两个指向其他链结点的引用，而不是一个。第一个像普通链表一样指向下一个链结点。第二个指向前一个链结点。图 5.13 显示了这种链表。

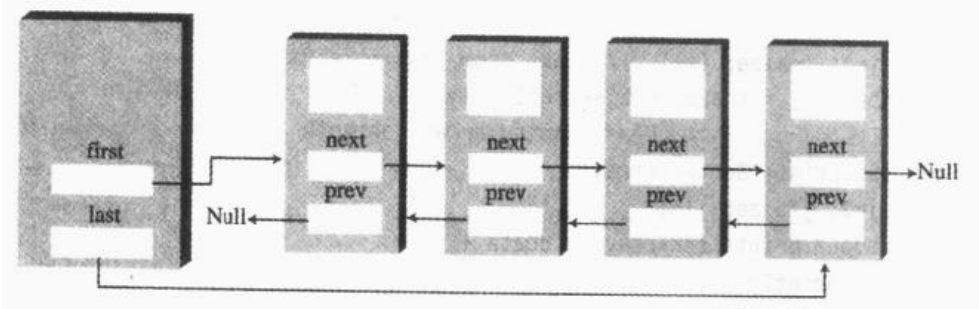


图 5.13 双向链表

在双向链表中，Link 类定义的开头是这样声明的：

```
class Link
   {
   public long dData;                  // data item
   public Link next;                   // next link in list
   public link previous;               // previous link in list
   ...
   }
```

双向链表的缺点是每次插入或删除一个链结点的时候，要处理四个链结点的引用，而不是两个：两个连接前一个链结点，两个连接后一个链结点。当然，由于多了两个引用，链结点的占用空间也变大了一点。

双向链表不必是双端链表（保持一个对链表最后一个元素的引用），但这种方式是有用的，所以在后面的例子中将包含双端的性质。

马上就会看到 doublyLinked.java 程序的全部代码清单，但首先还是来考察一下 doublyLinkedList 类的部分方法。

遍历

两个显示方法说明了双向链表的遍历。displayForward()方法和在普通链表中看到的 displayList()方法一样。displayBackward()方法与它们类似，但是从表尾开始，通过每个元素的 previous 域，一步一步向前到达表头。下面这段代码显示了这一过程。

```
Link current = last;              // start at end
while(current != null)            // until start of list,
   current = current.previous;    // move to previous link
```

顺便说一下，有些人带有这样的观点，因为你能够在双向链表中方便地向前或向后遍历，就没有一个明确的指向说明哪边是前，哪边是后，那么 previous 和 next 这些称谓就显得不太合适。如果读者愿意，可以用 left 和 right 这种方向中性的词来代替。

插入

在 DoublyLinkedList 类中已经包含了几个插入方法。insertFirst()方法在表头插入，insertLast()在表尾插入，insertAfter()在某一特定元素的后面插入。

除非这个链表是空的，否则 insertFirst()方法把原先 first 指向的链结点的 previous 字段指向新链结点，把新链结点的 next 字段指向前者。最后把 first 指向新链结点。这个过程如图 5.14 所示。

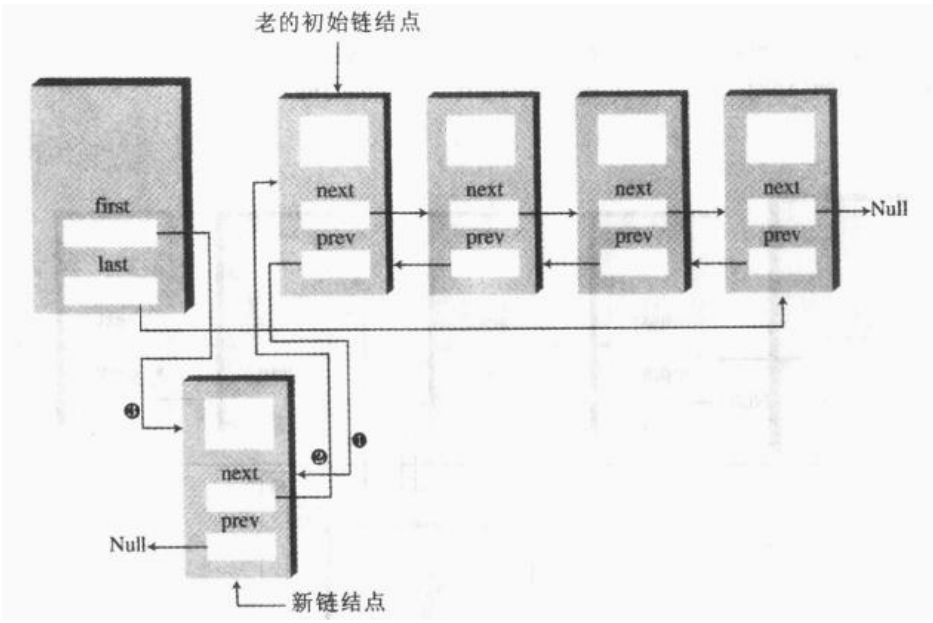


图 5.14 在表头插入

如果链表是空的，last 字段必须改变，而不是 first.previous 字段改变。下面是代码：

```
if( isEmpty() )                 // if empty list,
   last = newLink;              // newLink <-- last
else
   first.previous = newLink;    // newLink <-- old first
newLink.next = first;           // newLink --> old first
first = newLink;                // first --> newLink
```

insetLast()方法是一个同样的过程，只不过应用在表尾；它和 insertFirst()方法呈镜像。

insertAfter()方法在某个特定值的链结点后插入一个新的链结点。它有点复杂，因为需要建立四个连接。首先，必须要找到具有特定值的链结点。这个过程和 linkList2.java 程序（清单 5.2）中的 find()方法使用同样的方式。然后，假设插入点不在表尾，首先建立新链结点和下一个链结点之间的两个连接，接着是建立 current 所指链结点和新链结点之间的两个连接。图 5.15 显示了这个过程。

如果新链结点插在表尾，它的 next 字段必须设为 null 值，last 值必须指向新链结点。下面是处理链结点的 insertAfter()方法：

```
if(current==last)               // if last link,
   {
   newLink.next = null;         // newLink --> null
   last = newLink;              // newLink <-- last
```

```
    }
  else                          // not last link,
    {
    newLink.next = current.next; // newLink --> old next
                                 // newLink <-- old next
    current.next.previous = newLink;
    }
  newLink.previous = current;    // old current <-- newLink
  current.next = newLink;        // old current --> newLink
```

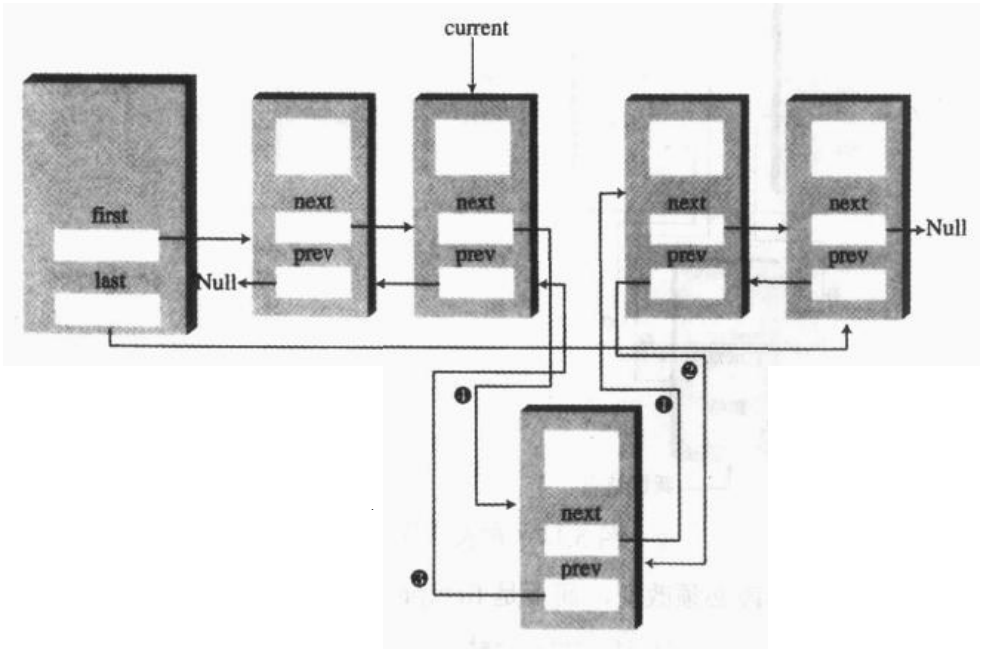


图 5.15 在任意的位置插入

有些读者可能还不太熟悉如何在一个表达式中使用两个句点的操作符。这是一个句点操作符的自然扩展。表达式

```
current.next.previous
```

的含义是 current 链结点的 next 字段引用的链结点的 previous 字段。

删除

这里有三个删除方法：deleteFirst()、deleteLast()和 deleteKey()。前两个相对容易。在 deleteKey()方法中，被删除的关键值链结点是 current 所指链结点。假设被删的链结点既不是第一个链结点，也不是最后一个，current.previous（被删链结点的前一个链结点）的 next 字段指向 current.next（被删链结点的后一个链结点），current.next 的 previous 字段指向 current.previous。这样就使 current 指向的链结点和链表断开了连接。图 5.16 显示了这个断开连接的过程，下面是执行这个过程的两行语句：

```
current.previous.next = current.next;
current.next.previous = current.previous;
```

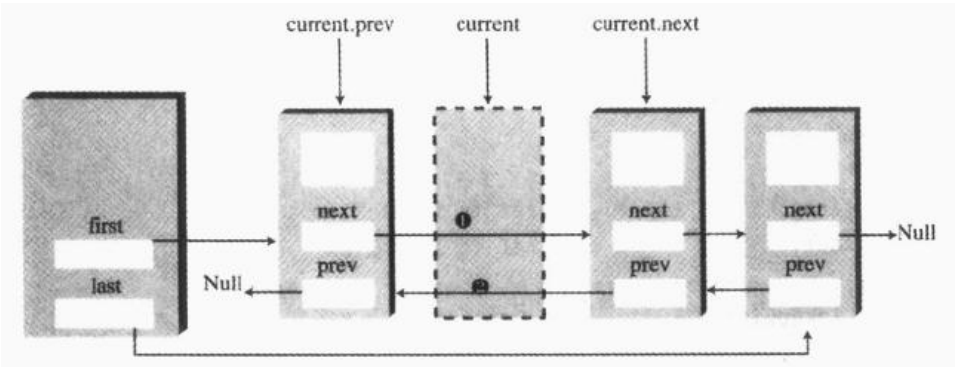


图 5.16 删除任意一个链结点

如果被删除的链结点是第一个或是最后一个，那么就产生了特殊情况，因为 first 或 last 必须指向前一个或后一个链结点。下面是 deleteKey()方法中处理链结点连接的代码：

```
if(current==first)                  // first item?
   first = current.next;            // first --> old next
else                                // not first
                                    // old previous --> old next
   current.previous.next = current.next;

if(current==last)                   // last item?
   last = current.previous;         // old previous <-- last
else                                // not last
                                    // old previous <-- old next
   current.next.previous = current.previous;
```

doublyLinked.java 程序

清单 5.8 显示了完整的 doublyLinked.java 程序，它包含了所有前面讨论的方法。

清单 5.8 doublyLinked.java 程序

```
// doublyLinked.java
// demonstrates doubly-linked list
// to run this program: C>java DoublyLinkedApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;                   // data item
   public Link next;                    // next link in list
   public Link previous;                // previous link in list
// -------------------------------------------------------------
   public Link(long d)                  // constructor
      { dData = d; }
// -------------------------------------------------------------
   public void displayLink()            // display this link
      { System.out.print(dData + " "); }
```

```
// --------------------------------------------------------
   } // end class Link
////////////////////////////////////////////////////////////////
class DoublyLinkedList
   {
   private Link first;              // ref to first item
   private Link last;               // ref to last item
// --------------------------------------------------------
   public DoublyLinkedList()        // constructor
      {
      first = null;                 // no items on list yet
      last = null;
      }
// --------------------------------------------------------
   public boolean isEmpty()         // true if no links
      { return first==null; }
// --------------------------------------------------------
   public void insertFirst(long dd) // insert at front of list
      {
      Link newLink = new Link(dd);  // make new link

      if( isEmpty() )               // if empty list,
         last = newLink;            // newLink <-- last
      else
         first.previous = newLink;  // newLink <-- old first
      newLink.next = first;         // newLink --> old first
      first = newLink;              // first --> newLink
      }
// --------------------------------------------------------
   public void insertLast(long dd)  // insert at end of list
      {
      Link newLink = new Link(dd);  // make new link
      if( isEmpty() )               // if empty list,
         first = newLink;           // first --> newLink
      else
         {
         last.next = newLink;       // old last --> newLink
         newLink.previous = last;   // old last <-- newLink
         }
      last = newLink;               // newLink <-- last
      }
// --------------------------------------------------------
   public Link deleteFirst()        // delete first link
      {                             // (assumes non-empty list)
      Link temp = first;
```

```
      if(first.next == null)         // if only one item
         last = null;                // null <-- last
      else
         first.next.previous = null; // null <-- old next
      first = first.next;            // first --> old next
      return temp;
      }
// ---------------------------------------------------------
   public Link deleteLast()          // delete last link
     {                               // (assumes non-empty list)
     Link temp = last;
     if(first.next == null)          // if only one item
        first = null;                // first --> null
     else
        last.previous.next = null;   // old previous --> null
     last = last.previous;           // old previous <-- last
     return temp;
     }
// ---------------------------------------------------------
                                    // insert dd just after key
   public boolean insertAfter(long key, long dd)
     {                              // (assumes non-empty list)
     Link current = first;          // start at beginning
     while(current.dData != key)    // until match is found,
        {
        current = current.next;     // move to next link
        if(current == null)
           return false;            // didn't find it
        }
     Link newLink = new Link(dd);   // make new link

     if(current==last)              // if last link,
        {
        newLink.next = null;        // newLink --> null
        last = newLink;             // newLink <-- last
        }
     else                           // not last link,
        {
        newLink.next = current.next; // newLink --> old next
                                     // newLink <-- old next
        current.next.previous = newLink;
        }
     newLink.previous = current;    // old current <-- newLink
     current.next = newLink;        // old current --> newLink
     return true;                   // found it, did insertion
```

```
    }
// ------------------------------------------------------------
  public Link deleteKey(long key)   // delete item w/ given key
    {                               // (assumes non-empty list)
    Link current = first;           // start at beginning
    while(current.dData != key)     // until match is found,
      {
      current = current.next;       // move to next link
      if(current == null)
        return null;                // didn't find it
      }
    if(current==first)              // found it; first item?
      first = current.next;         // first --> old next
    else                            // not first
                                    // old previous --> old next
      current.previous.next = current.next;

    if(current==last)               // last item?
      last = current.previous;      // old previous <-- last
    else                            // not last
                                    // old previous <-- old next
      current.next.previous = current.previous;
    return current;                 // return value
    }
// ------------------------------------------------------------
  public void displayForward()
    {
    System.out.print("List (first-->last): ");
    Link current = first;           // start at beginning
    while(current != null)          // until end of list,
      {
      current.displayLink();        // display data
      current = current.next;       // move to next link
      }
    System.out.println("");
    }
// ------------------------------------------------------------
  public void displayBackward()
    {
    System.out.print("List (last-->first): ");
    Link current = last;            // start at end
    while(current != null)          // until start of list,
      {
      current.displayLink();        // display data
      current = current.previous; // move to previous link
      }
```

```
      System.out.println("");
      }
// --------------------------------------------------------
   }  // end class DoublyLinkedList
////////////////////////////////////////////////////////////////
class DoublyLinkedApp
   {
   public static void main(String[] args)
      {                              // make a new list
      DoublyLinkedList theList = new DoublyLinkedList();

      theList.insertFirst(22);       // insert at front
      theList.insertFirst(44);
      theList.insertFirst(66);

      theList.insertLast(11);        // insert at rear
      theList.insertLast(33);
      theList.insertLast(55);

      theList.displayForward();      // display list forward
      theList.displayBackward();     // display list backward

      theList.deleteFirst();         // delete first item
      theList.deleteLast();          // delete last item
      theList.deleteKey(11);         // delete item with key 11

      theList.displayForward();      // display list forward

      theList.insertAfter(22, 77);   // insert 77 after 22
      theList.insertAfter(33, 88);   // insert 88 after 33

      theList.displayForward();      // display list forward
      }  // end main()
   }  // end class DoublyLinkedApp
////////////////////////////////////////////////////////////////
```

在 main()方法中，在表头和表尾分别插入一些链结点，正向或反向显示这些数据；然后删除第一个元素、最后一个元素和关键值为 11 的链结点，再次正向显示链表，接着用 insertAfter()方法插入两个新的链结点，又一次显示链表。下面是输出：

```
List (first-->last): 66 44 22 11 33 55
List (last-->first): 55 33 11 22 44 66
List (first-->last): 44 22 33
List (first-->last): 44 22 77 33 88
```

用删除方法和 insertAfter()方法时假设链表不空。所以在试图做插入和删除前，应该用 isEmpty()

方法来保证链表中是非空的。不过为了简便起见，在 main()方法中没有表达这一点。

基于双向链表的双端队列

前面的章节已经提及，双向链表可以用来作为双端队列的基础。在双端队列中，可以从任何一头插入和删除，双向链表提供了这个能力。

迭代器

已经看到链表的用户怎样使用 find()方法来查找一个含有给定值的链结点。这个方法从表头开始考察每个链结点，直到找到一个链结点的值和给定值匹配。其他的一些操作，例如删除指定链结点，在指定链结点的前面或后面插入新链结点，也含有链表上的搜索工作，以找到指定的链结点。然而，这些方法没有提供给用户任何遍历上的控制手段，以便找到指定链结点。

假定你要遍历一个链表，并在某些特定的链结点上执行一些操作。例如，用一个链表存储的职员表。你可能需要提高所有拿最低工资的员工的工资，而不影响那些已经高于最低工资的员工。或者假设一个订阅邮件用户的链表，你需要删除所有近六个月没有订阅任何邮件的用户。

在数组中，这种操作非常容易，因为可以用数组下标跟踪所在位置。可以在这个链结点上进行操作，然后通过下标指向下一项，看那一这项是否符合操作条件。然而在链表中，链结点没有固定的下标。怎样才能提供给链表用户类似于数组下标的东西呢？虽然可以反复使用 find()方法在链表中查找到合乎要求的链结点，但是为查找每个链结点这个方法需要进行很多次比较。如果能从链结点到链结点步进，检查每个链结点是否符合某个标准，若符合标准就执行适当的操作，这样效率会高得多。

放在链表内部吗？

作为类的用户，需要能存取指向任意链结点的引用。这样就可以考察和修改链结点。引用应该能递增，因此可以沿着整个链表遍历，依次查看每个链结点，而且可以访问这个引用所指向的链结点。

设想创建这么一个引用，把它安放在哪里？一个可能性是使用链表本身的一个字段，叫作 current 或是其他名字。可以用 current 存取一个链结点，然后使 current 递增，移动到下一个链结点。

这个方法存在一个问题是可能同时需要不止一个这种引用，就像经常要同时使用几个下标。多少个最适合？无法知道用户会需要几个。因此，很容易想到允许用户按使用的需求创建多个引用。为了在面向对象语言中使其成为可能，自然是考虑在一个类中嵌入每个引用。这个类不能和链表类相同，因为只有一个链表对象，所以把它做成另一个类。

迭代器类

迭代器类包含对数据结构中数据项的引用，并用来遍历这些结构的对象（有时，在某些 Java 类中，叫做“枚举器”）。下面是它们最初的定义：

```
class ListIterator()
   {
   private Link current;
   ...
   }
```

current 字段包含迭代器当前指向的链结点的一个引用。(这里使用的术语“指向”和 C++语言中的指针没有关系；正在使用的是它的“引用”含义。)

为了使用这样的迭代器，用户可以创建一个链表，然后创建一个和链表相关联的迭代器对象。实际上，当链表产生时，让它创建迭代器更容易一些，它能向迭代器传递必要的信息，例如它的 first 字段的一个引用。因此，在链表类中增加一个 getIterator()方法；这个方法返回给用户一个恰当的迭代器对象。下面是 main()方法中的部分代码，它显示了类的使用者如何调用迭代器：

```
public static void main(...)
   {
   LinkList theList = new LinkList();           // make list
   ListIterator iter1 = theList.getIterator();  // make iter
Link aLink = iter1.getCurrent();    // access link at iterator
iter1.nextLink();                   // move iter to next link
}
```

创建迭代器对象后，就可以通过它存取它指向的链结点，或者递增它以指向下一个链结点，正如后两行语句显示的那样。把迭代器对象叫作 iter1 是为了证明还可以以同样的方式声明更多的迭代器（iter2 等等）。

迭代器总是指向链表中的一些链结点。它同链表相关联，但并不等同于链表或是链结点。图 5.17 显示了指向链表中的链结点的两个迭代器。

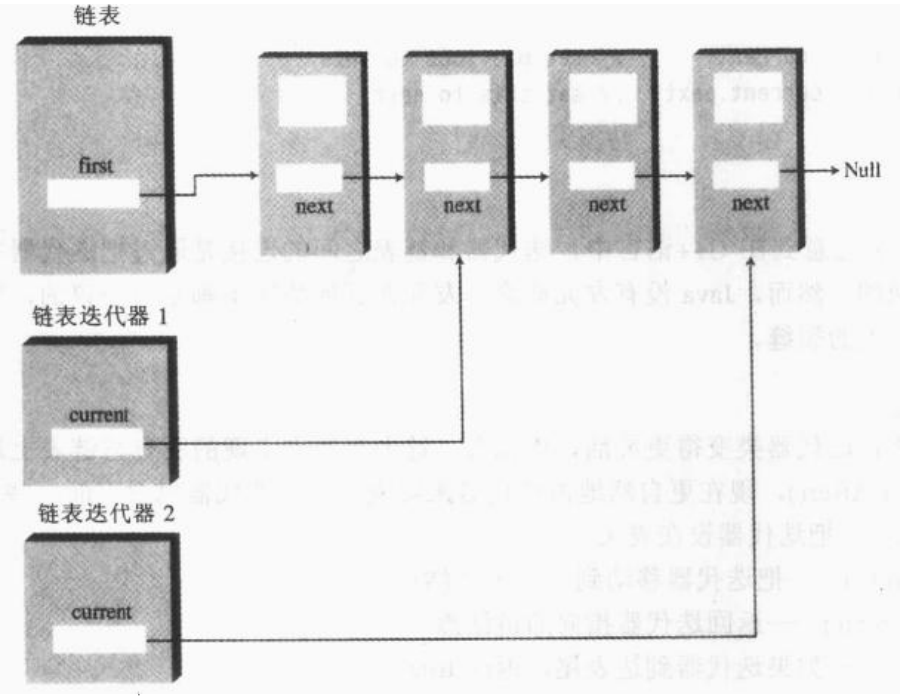


图 5.17 链表迭代器

迭代器的其他特性

前面已经看到在几个程序中都使用过 previous 字段，它使某些操作的执行变得简单，例如从任意位置删除一个链结点。这样的字段在迭代器中也是有用的。

而且，迭代器可能需要改变链表 first 字段的值——例如，如果在表头插入和删除一个数据项。如果迭代器是一个单独类的对象，它怎么能访问 first 这样的私有数据字段呢？一种解决办法是链表创建迭代器时，传递一个引用给迭代器。这个引用存储在迭代器的一个字段中。

链表必须提供公有方法，允许迭代器改变 first 的值。LinkList 类中的这些方法是 getFirst()和 setFirst()。（这种方式的缺点是这些方法允许任何人修改 first，这就引入了危险的因素。）

这里有一个修正的（尽管还不是完整的）迭代器类，这个类并入了一些其他的字段，以及 reset()方法和 nextLink()方法：

```
class ListIterator()
   {
   private Link current;       // reference to current link
   private Link previous;      // reference to previous link
   private LinkList ourList;   // reference to "parent" list

   public void reset()         // set to start of list
      {
      current = ourList.getFirst();  // current --> first
      previous = null;               // previous --> null
      }
   public void nextLink()      // go to next link
      {
      previous = current;      // set previous to this
      current = current.next;  // set this to next
      }
   ...
   }
```

C++程序员会注意到在 C++语言中，迭代器和链表之间的连接是通过把迭代器类设为链表类的“友元”来实现的。然而，Java 没有友元概念，友元在任何情况下都是有争议的，因为它如同是数据隐藏这个盔甲上的裂缝。

迭代器的方法

新加的方法让迭代器类变得更灵活，更强大。过去在类中实现的所有与链表上进行迭代有关的操作，例如 insertAfter()，现在更自然地由迭代器来实现。例如迭代器包含下面一些方法：

- reset()——把迭代器设在表头
- nextLink()——把迭代器移动到下一个链结点
- getCurrent()——返回迭代器指向的链结点
- atEnd()——如果迭代器到达表尾，返回 true
- insertAfter()——在迭代器后面插入一个新链结点
- insertBefore()——在迭代器前面插入一个新链结点
- deleteCurrent()——删除迭代器所指链结点

用户可以用 reset()方法和 nextLink()方法定位迭代器，用 atEnd()方法检查是否它在表尾，以及执行上述的其他操作。

哪些任务由迭代器完成，哪些任务由链表本身完成，这并不容易决定。insertBefore()方法最好在迭代器中工作，而 insertFirst()方法总是在表头插入新链结点，它可能更适合在链表类中实现。本例把 displayList()方法留在了链表类中，但是这个操作也可以通过调用迭代器中的 getCurrent()和 nextLink()方法来实现。

interIterator.java 程序

interIterator.java 程序包含一个交互界面，它允许使用者直接控制迭代器。启动程序后，可以通过键入正确的字母执行下面的动作：

- s（Show）——显示链表内容
- r（Reset）——把迭代器复位并设在表头
- n（Next）——到下一个链结点
- g（Get）——得到当前链结点的内容
- b（InsertBefore）——在当前链结点前插入新链结点
- a（InsertAfter）——在当前链结点后插入新链结点
- d（Delete）——删除当前链结点

清单 5.9 显示了完整的 interIterator.java 程序。

清单 5.9 interIterator.java 程序

```
// interIterator.java
// demonstrates iterators on a linked listListIterator
// to run this program: C>java InterIterApp
import java.io.*;                  // for I/O
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;              // data item
   public Link next;               // next link in list
// -------------------------------------------------------------
   public Link(long dd)            // constructor
      { dData = dd; }
// -------------------------------------------------------------
   public void displayLink()       // display ourself
      { System.out.print(dData + " "); }
   }  // end class Link
////////////////////////////////////////////////////////////////
class LinkList
   {
   private Link first;             // ref to first item on list

// -------------------------------------------------------------
   public LinkList()               // constructor
      { first = null; }            // no items on list yet
// -------------------------------------------------------------
```

```
   public Link getFirst()         // get value of first
      { return first; }
// -------------------------------------------------------------
   public void setFirst(Link f)   // set first to new link
      { first = f; }
// -------------------------------------------------------------
   public boolean isEmpty()       // true if list is empty
      { return first==null; }
// -------------------------------------------------------------
   public ListIterator getIterator()  // return iterator
      {
      return new ListIterator(this);  // initialized with
      }                               //    this list
// -------------------------------------------------------------
   public void displayList()
      {
      Link current = first;       // start at beginning of list
      while(current != null)      // until end of list,
         {
         current.displayLink();   // print data
         current = current.next;  // move to next link
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   } // end class LinkList
////////////////////////////////////////////////////////////////
class ListIterator
   {
   private Link current;          // current link
   private Link previous;         // previous link
   private LinkList ourList;      // our linked list
//-------------------------------------------------------------
   public ListIterator(LinkList list) // constructor
      {
      ourList = list;
      reset();
      }
//-------------------------------------------------------------
   public void reset()            // start at 'first'
      {
      current = ourList.getFirst();
      previous = null;
      }
//-------------------------------------------------------------
```

```
   public boolean atEnd()          // true if last link
      { return (current.next==null); }
//--------------------------------------------------------------
   public void nextLink()          // go to next link
      {
      previous = current;
      current = current.next;
      }
//--------------------------------------------------------------
   public Link getCurrent()        // get current link
      { return current; }
//--------------------------------------------------------------
   public void insertAfter(long dd)     // insert after
      {                                 // current link
      Link newLink = new Link(dd);

      if( ourList.isEmpty() )      // empty list
         {
         ourList.setFirst(newLink);
         current = newLink;
         }
      else                         // not empty
         {
         newLink.next = current.next;
         current.next = newLink;
         nextLink();               // point to new link
         }
      }
//--------------------------------------------------------------
   public void insertBefore(long dd)    // insert before
      {                                 // current link
      Link newLink = new Link(dd);

      if(previous == null)         // beginning of list
         {                         // (or empty list)
         newLink.next = ourList.getFirst();
         ourList.setFirst(newLink);
         reset();
         }
      else                         // not beginning
         {
         newLink.next = previous.next;
         previous.next = newLink;
         current = newLink;
         }
```

```
      }
//--------------------------------------------------------------
   public long deleteCurrent()     // delete item at current
      {
      long value = current.dData;
      if(previous == null)         // beginning of list
         {
         ourList.setFirst(current.next);
         reset();
         }
      else                         // not beginning
         {
         previous.next = current.next;
         if( atEnd() )
            reset();
         else
            current = current.next;
         }
      return value;
      }
//--------------------------------------------------------------
   }  // end class ListIterator
////////////////////////////////////////////////////////////////
class InterIterApp
   {
   public static void main(String[] args) throws IOException
      {
      LinkList theList = new LinkList();           // new list
      ListIterator iter1 = theList.getIterator();  // new iter
      long value;

      iter1.insertAfter(20);                       // insert items
      iter1.insertAfter(40);
      iter1.insertAfter(80);
      iter1.insertBefore(60);

      while(true)
         {
         System.out.print("Enter first letter of show, reset, ");
         System.out.print("next, get, before, after, delete: ");
         System.out.flush();
         int choice = getChar();          // get user's option
         switch(choice)
            {
            case 's':                     // show list
```

```
      if( !theList.isEmpty() )
         theList.displayList();
      else
         System.out.println("List is empty");
      break;
   case 'r':                    // reset (to first)
      iter1.reset();
      break;
   case 'n':                    // advance to next item
      if( !theList.isEmpty() && !iter1.atEnd() )
         iter1.nextLink();
      else
         System.out.println("Can't go to next link");
      break;
   case 'g':                    // get current item
      if( !theList.isEmpty() )
         {
         value = iter1.getCurrent().dData;
         System.out.println("Returned " + value);
         }
      else
         System.out.println("List is empty");
      break;
   case 'b':                    // insert before current
      System.out.print("Enter value to insert: ");
      System.out.flush();
      value = getInt();
      iter1.insertBefore(value);
      break;
   case 'a':                    // insert after current
      System.out.print("Enter value to insert: ");
      System.out.flush();
      value = getInt();
      iter1.insertAfter(value);
      break;
   case 'd':                    // delete current item
      if( !theList.isEmpty() )
         {
         value = iter1.deleteCurrent();
         System.out.println("Deleted " + value);
         }
      else
         System.out.println("Can't delete");
      break;
   default:
```

```
                System.out.println("Invalid entry");
            } // end switch
         } // end while
      } // end main()
//--------------------------------------------------------------
public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
//--------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//--------------------------------------------------------------
   } // end class InterIterApp
////////////////////////////////////////////////////////////////
```

main()方法使用迭代器和它的 insertAfter()方法插入四个数据项。然后等待使用者和它交互。在下面的示例操作中，使用者显示链表，迭代器复位指向链表开始部位，前进两个链结点，得到当前链结点的关键值（60），在它前面插入 100，在 100 后面插入 7，然后再次显示链表：

```
Enter first letter of
   show, reset, next, get, before, after, delete: s
20 40 60 80
Enter first letter of
   show, reset, next, get, before, after, delete: r
Enter first letter of
   show, reset, next, get, before, after, delete: n
Enter first letter of
   show, reset, next, get, before, after, delete: n
Enter first letter of
   show, reset, next, get, before, after, delete: g
Returned 60
Enter first letter of
```

```
   show, reset, next, get, before, after, delete: b
Enter value to insert: 100
Enter first letter of
   show, reset, next, get, before, after, delete: a
Enter value to insert: 7
Enter first letter of
   show, reset, next, get, before, after, delete: s
20 40 100 7 60 80
```

通过考察 interIterator.java 程序，可以弄明白迭代器如何沿着链表移动，以及它怎样在链表任意位置插入或删除链结点。

迭代器指向哪里？

迭代器类的一个设计问题是决定在不同的操作后，迭代器应该指向哪里。

当用 deleteCurrent()删除一项后，迭代器应该放在下一链结点，前一链结点，还是回到表头呢？把迭代器保持在被删除链结点的附近是方便的，因为类的用户将在那里执行其他的操作。然而，不能把它移动到前一个结果，因为无法把链表的 previous 字段置成前一项。（要完成这个任务，需要一个双向链表。）解决办法是把迭代器移动到被删除链结点的下一个链结点。如果恰巧删除链表的最后一个数据项，迭代器复位指向表头。

执行 insertBefore()方法和 insertAfter()方法后，让 current 指向新插入的链结点。

atEnd()方法

还有关于 atEnd()方法的另一个问题。有两种做法：当迭代器指向链表最后一个有效链结点时，它返回 ture，或者当迭代器指向最后一个链结点的“下一个”时（这时，它不是指向一个有效链结点），它返回 true。

用第一种方法，按这种循环条件，遍历链表显得很笨拙。因为需要在最后一个链结点上仍执行循环体内的操作，然而一旦检查出它是最后一个链结点，循环却中止了，没有来得及做操作。

在第二种方法中，一直不知是否到链表的结尾，一旦发现到了表尾，想对最后一个链结点做什么事都已经太晚了。（例如，找到最后一个链结点，并删除它。）这是因为当 atEnd()变成 true 时，迭代器将不再指向最后的链结点（或其他确切的有效链结点），迭代器在单链表中也不能“倒退”。

所以在这里还是采用第一种方法，用这种方法迭代器总是能指向一个有效链结点。然而，正如下面将要看到的那样，在写这种遍历链表的循环时要格外小心。

迭代的操作

已经看到，迭代器能遍历链表，在某些链结点上执行操作。下面的代码片段作用为显示链表的内容，它使用了迭代器，而不是链表的 displayList()方法：

```
iter1.reset();                          // start at first
long value = iter1.getCurrent().dData;  // display link
System.out.println(value + " ");
while( !iter1.atEnd() )                 // until end,
   {
   iter1.nextLink();                    // go to next link,
```

```
    long value = iter1.getCurrent().dData;  // display it
    System.out.println(value + " ");
    }
```

尽管这里没有写，但应该在调用 getCurrent()方法前用 isEmpty()方法检查以确定链表不空。

下面的代码显示了怎样删除所有关键值是 3 的倍数的数据项。这里只显示了修正的 main()方法；其他部分与 interIterator.java（清单 5.9）相同。

```
class InterIterApp
   {
   public static void main(String[] args) throws IOException
      {
      LinkList theList = new LinkList();           // new list
      ListIterator iter1 = theList.getIterator();  // new iter

      iter1.insertAfter(21);            // insert links
      iter1.insertAfter(40);
      iter1.insertAfter(30);
      iter1.insertAfter(7);
      iter1.insertAfter(45);

      theList.displayList();            // display list

      iter1.reset();                    // start at first link
      Link aLink = iter1.getCurrent();  // get it
      if(aLink.dData % 3 == 0)          // if divisible by 3,
         iter1.deleteCurrent();         // delete it
      while( !iter1.atEnd() )           // until end of list,
         {
      iter1.nextLink();                // go to next link

      aLink = iter1.getCurrent();      // get link
      if(aLink.dData % 3 == 0)         // if divisible by 3,
         iter1.deleteCurrent();        // delete it
      }
    theList.displayList();             // display list
    } // end main()
} // end class InterIterApp
```

进行如下操作：插入五个链结点，显示链表。然后迭代地遍历链表，删除那些能整除 3 的链结点，再次显示链表。下面是输出：

```
21 40 30 7 45
40 7
```

再次说明，尽管没有在这里写出，但是调用 deleteCurrent()方法前检查链表是否为空是非常重要的。

其他方法

可以为 ListIterator 类创建其他有用的方法。例如，find()方法可以返回包含特定值的数据项，就像以前在链表方法中看到的 find()方法那样。replace()方法能够把特定数据项替换成其他数据项。

因为这是单链表，可以只向前迭代地遍历链表。如果使用双向链表，可以双向走，需要有各种操作，例如在表尾删除，就像没有用迭代器一样。这个能力在某些应用中可能会是有益的。

小 结

- 链表包含一个 linkedList 对象和许多 Link 对象
- linkedList 对象包含一个引用，这个引用通常叫做 first，它指向链表的第一个链结点。
- 每个 Link 对象包含数据和一个引用，通常叫做 next，它指向链表的下一个链结点。
- next 字段为 null 值意味着链表的结尾。
- 在表头插入链结点需要把新链结点的 next 字段指向原来的第一个链结点，然后把 first 指向新链结点。
- 在表头删除链结点要把 first 指向 first.next。
- 为了遍历链表，从 first 开始，然后从一个链结点到下一个链结点，方法是用每个链结点的 next 字段找到下一个链结点。
- 通过遍历链表可以找到拥有特定值的链结点。一旦找到，可以显示、删除或用其他方式操纵该链结点。
- 新链结点可以插在某个特定值的链结点的前面或后面，首先要遍历找到这个链结点。
- 双端链表在链表中维护一个指向最后一个链结点的引用，它通常和 first 一样，叫做 last。
- 双端链表允许在表尾插入数据项。
- 抽象数据类型是一种数据存储类，不涉及它的实现。
- 栈和队列是 ADT。它们既可以用数组实现，又可以用链表实现。
- 有序链表中，链结点按照关键值升序（有时是降序）排列。
- 在有序链表中插入需要 O(N)的时间，因为必须找到正确的插入点。最小值链结点的删除需要 O(1)的时间。
- 双向链表中，每个链结点包含对前一个链结点的引用，同时有对后一个链结点的引用。
- 双向链表允许反向遍历，并可以从表尾删除。
- 迭代器是一个引用，它被封装在类对象中，这个引用指向相关联的链表中的链结点。
- 迭代器方法允许使用者沿链表移动迭代器，并访问当前指示的链结点。
- 能用迭代器遍历链表，在选定的链结点（或所有链结点）上执行某些操作。

问 题

下列问题作为读者的自测题。答案可见附录 C。

1. 下面哪种表达是不正确的？类对象的引用
   a. 能用来访问对象的公共方法。

b. 大小取决于它所指向的类。
c. 拥有这个类的数据类型。
d. 不含有这个类本身。

2. 在链表中访问链结点通常要_____链结点。
3. 当在链表中创建一个对链结点的引用时，它
a. 必须指向第一个链结点。
b. 必须指向 current 指向的链结点。
c. 必须指向 next 指向的链结点。
d. 可以指向任何链结点。
4. 在单链表的中部插入一个链结点要改变几个引用？
5. 在单链表的尾部插入一个链结点要改变几个引用？
6. 在 linkList.java 程序（清单 5.1）的 insertFirst()方法中，语句 newLink.next=first 的意思是
a. 下一个插入的新链结点将指向 first。
b. first 将指向新链结点。
c. 新链结点的 next 域将指向原来的第一个链结点。
d. nexLink.next 将指向链表的第一个链结点。
7. 假设 current 指向倒数第二个链结点，执行什么语句可以删除最后一个链结点？
8. 当链结点的所有引用都指向了其他链结点，那么这个链结点会怎么样？
9. 双端链表
a. 能从两端进行访问。
b. 是双向链表的另一个名字。
c. 有指针在链结点间前向或后向的移动。
d. 它的第一个链结点和它的最后一个链结点相连。
10. 当链表是_____时，插入和删除会出现特殊情况。
11. 假设拷贝比比较需要更长的时间，那么在链表中和在无序数组中删除一个特定值的数据项，哪个更快一些？
12. 要删除拥有最大值的数据项，需要遍历多少次单链表？
13. 本章讨论的各种链表中，哪个最适合用来实现队列？
14. 下面哪种表达是不正确的？迭代器在以下情况中有用：
a. 在链表中做插入排序。
b. 在表头插入新链结点。
c. 交换两个任意位置的链结点。
d. 删除所有包含特定值的链结点。
15. 你认为实现一个栈用哪种结构更好，单链表还是数组？

实 验

完成这些实验可以帮助深入理解本章的主题。不需要编程实现。

1. 用 LinkList 专题 applet 执行插入、查找和删除操作，在有序链表和无序链表上都尝试一下。对于这些操作，有序链表有什么优势吗？

2. 修改 linkList.java 程序（清单 5.1）的 main()方法，使它能够继续插入链结点，直到内存耗尽。每隔 1000 个数据项，让它显示一次插入的链结点数目。通过这种方法，你可以粗略知道你机器可以容纳多少个链结点。（当然，链结点数目的多少也依赖于内存中其他程序和许多其他因素。）如果它会使你所在的公共网络崩溃，就不要进行这个实验了。

编程作业

编程作业有助于巩固对本章内容的理解，并展示如何应用本章的概念。（在“简介”中提到过，资深教师可以从出版者的网站上得到编程作业的完整答案。）

5.1 实现一个基于有序链表的优先级队列。队列的删除操作应该删除具有最小关键字的链结点。

5.2 实现一个基于双向链表的双端队列。（参考前一章上机作业 4.2。）使用者应该能够执行双端队列的基本操作。

5.3 循环链表是一种链表，它的最后一个链结点指向第一个链结点。设计循环链表有许多方法。有时，用一个指向链表“开始”的指针。然而这样做使链表不像一个真正的环，而更像一个传统的链表，只不过这个链表的表头和表尾系在了一起。编写一个类代表循环单链表，它没有表头也没有表尾。访问这个链表的惟一方式是一个引用 current，它能指向任意一个链结点。这个引用在需要的时候可以沿链表移动。（参考上机作业 5.5，那种情况下用循环链表很合适。）你的链表应该能处理插入、查找和删除。可能会发现如果这些操作发生在 current 指向的下一个链结点时将会非常方便。（因为上一个链结点是单向连接的，不遍历整个循环链表，不可能得到它。）你也应该可以显示链表（尽管需要在循环链表的某处切断环，以把它们打印到屏幕上）。step()方法可以把 current 移动到下一个链结点，在这个程序中可能也会派上用场。

5.4 实现一个基于上题循环链表的栈类。这个练习并不是太难。（然而，实现一个队列有些难度，除非把循环链表做成双向的。）

5.5 Josephus 问题是古代一个著名的数学难题。围绕这个问题有很多故事。其中一个说 Josephus 是一群被罗马人抓获的犹太人中的一个，为了不被奴役，他们选择了自杀。他们排成一个圆圈，从某个人开始，沿着圆圈计数。每报第 n 个数的人就要离开圆圈去自杀。Josephus 不想死，所以他制定了规则，以使他成为最后一个离开圆圈的人。如果有（例如）20 个人的话，他就是从开头数第 7 个人，那么他让他们用什么数来进行报数呢？这个问题会越来越复杂，因为随着过程进行，圆圈在缩小。

使用 5.3 题的循环链表创建一个应用来模拟这个问题。输入是组成圆圈的人数，要报的数和第一个开始报的人的位置（通常是 1）。输出是被消去的人的列表。当一个人出了圆圈，再继续从他左边那个人开始计数（假设沿顺时针旋转）。这有一个例子。有 7 个人，从 1 到 7，从第一个人开始报数，报到 4 出圆圈，最后被消去的人的顺序是 4，1，6，5，7，3。最后剩下的人是 2。

5.6 下面尝试一些不同的事情：二维链表，通常叫做矩阵。这是二维数组的链表版本。它可以用于某些应用，例如电子表格程序。如果电子表格是基于数组的，那么在顶端附近插入一个新行时，

需要移动下面 N*M 个单元，这可能是一个非常缓慢的过程。如果电子表格用矩阵实现，只需要改变 N 个指针。

为了简单起见，假定用单链表方法（尽管双向链表方法可能对电子表格更为适合）。每个链结点（除了在第一行和左侧面的链结点）都由它上面和左面的链结点指着。就是说，可以从左上角的链结点开始移动，找到第三行第五列的链结点，只要沿着指针向下两行，向右四列即可。假定矩阵有固定的大小（例如 7×10）。应该可以在某个链结点处插入值，并显示矩阵的内容。

第 6 章

递　　归

本章重点

- 三角数字
- 阶乘
- 变位字
- 递归的二分查找
- 汉诺塔
- 归并排序
- 消除递归
- 一些有趣的递归应用

递归是一种方法（函数）调用自己的编程技术。这听起来似乎有点奇怪，或者甚至像是一个灾难性的错误。但是，递归在编程中却是最有趣，又有惊人高效的技术之一。就像拽着自己的鞋带拔高一样（你确实有鞋带，是吗？），在第一次遇到递归时，它似乎让人觉得难以置信。然而，递归不仅可以解决特定的问题，而且它也为解决很多问题提供了一个独特的概念上的框架。

在这一章中，将研究许多例子，以此来说明递归可以应用在多种情况下。首先将计算三角数字和阶乘，生成变位字，执行一个递归的二分查找，然后解决汉诺塔问题，最后还将研究一种被称为归并排序的排序技术。并且用专题 applet 来举例说明汉诺塔问题和归并排序问题。

同时还会讨论递归方法的优势和劣势，并且将说明一个递归的方法如何被转换成一个基于栈的非递归方法。

三角数字

据说毕达哥拉斯理论家，又称一群在毕达哥拉斯（以毕达哥拉斯理论闻名）领导下工作的古希腊的数学家，发现了在数字序列 1，3，6，10，15，21，...（省略号说明这个序列无限地继续下去）中有一种奇特的联系。你能知道这个序列的下一个数字是什么吗？

这个数列中的第 n 项是由第 n-1 项加 n 得到的。由此，第二项是由第一项（1）加上 2，得 3。第三项是由第二项（3）加上 3 得到 6，依此类推。

这个序列中的数字被称为三角数字，因为它们可以被形像化地表示成对象的一个三角形排列，如图 6.1 中小方块所示。

使用循环查找第 n 项

假设想要在这个数列中找到某任意项，第 n 项的值，比如说第 4 项（其值为 10）。你会如何计算它呢？看图 6.2，可以判定任何项的值都可以通过把所有竖直列上的小方块加起来而得到。

在第四项中，第一列有 4 个小方块，第二列有 3 个，依此类推。4+3+2+1 得到 10。

下面的 triangle()方法使用这种基于列的方法来找到一个三角数字。这种方法把所有的列都相加，从高度为 n 的列加到高度为 1 的列：

```
int triangle(int n)
   {
   int total = 0;
```

```
while(n > 0)           // until n is 1
   {
   total = total + n;  // add n (column height) to total
   --n;                // decrement column height
   }
return total;
}
```

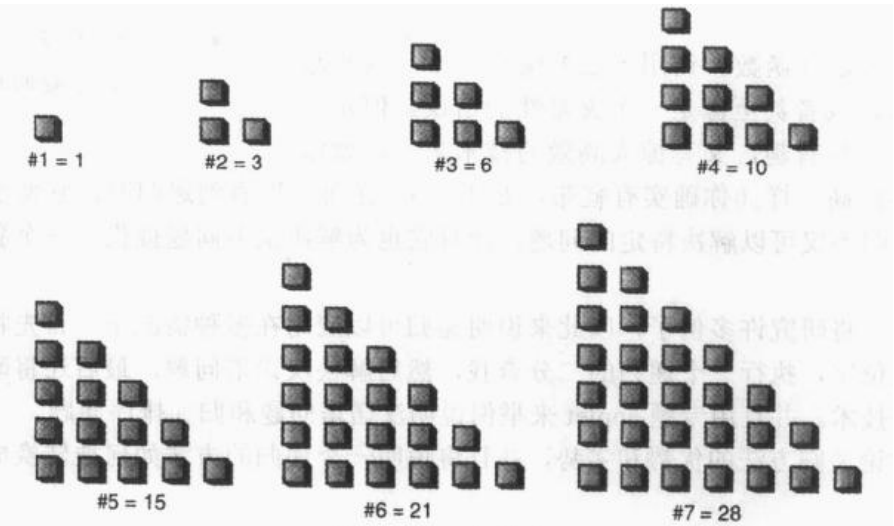


图 6.1 三角数字

这个方法循环了 n 次，total 值在第一次循环中加 n，第二次循环中加 n-1，如此循环一直到加 1，当 n 减小到 0 时退出循环。

使用递归查找第 n 项

循环的方法好像是非常易懂的，但是还可以通过另外一种方式来看这个问题。第 n 项的值可以被看成只是两个部分的和，而不是被看作整个序列的和。它们是：

1. 第一列（最高的一列），它的值为 n。
2. 所有剩余列的和。

如图 6.3 所示。

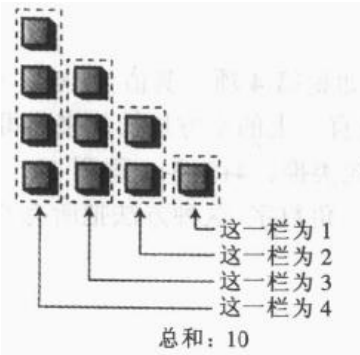


图 6.2 按列分割的三角数字

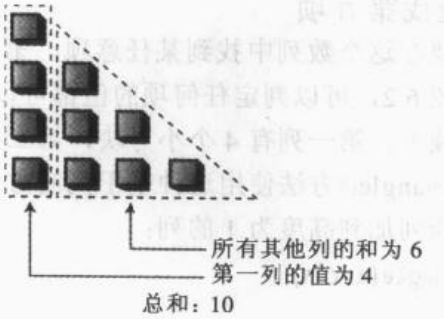


图 6.3 列加上三角形求三角数字

找到所有剩余列

如果知道一种可以找到所有剩余列的和的方法，就可以写一个 triangle()方法，这个方法返回第 n 个三角数字的值，如下：

```
int triangle(int n)
   {
   return( n + sumRemainingColumns(n) );  // (incomplete version)
   }
```

但是这时得到了什么？看起来写 sumRemainingColumns()方法就像在开始中写 triangle()方法一样困难。

但是，请注意，在图 6.3 中，为计算第 n 项的值时求所有剩余列的和与求第 n-1 项值时求所有列之和是一样的。这样，如果我们知道了一种可以求所有 n 列之和的方法，就可以用参数 n-1 来调用这个方法，以此来为第 n 项求其所有剩余列的和：

```
int triangle(int n)
   {
   return( n + sumAllColumns(n-1) );  // (incomplete version)
   }
```

但是仔细考虑这个问题时，就会发现 sumAllColumns() 方法所做的事情和 triangle()方法是完全一样的：把传递进来某个数字 n 作为参数，来计算所有 n 列的和。所以为什么不使用 triangle()方法本身来取代其他的方法呢？就像这样：

```
int triangle(int n)
   {
   return( n + triangle(n-1) );  // (incomplete version)
   }
```

可能会对一个方法能调用它自己觉得很怪异。怎么不这样试一下呢？其实方法调用就是把控制转移到这个方法的开始。这种控制的转移既可以发生在方法的外部，也可以发生在方法的内部。

把责任推给别人

所有的这些方法都可以看作是把责任推给别人。某人让我计算第九个三角数字。我知道这就是 9 加上第八个三角数字，所以我叫来 Harry 并让他来计算第八个三角数字。当我从 Harry 那里得到了返回的时候，我就可以用 9 去加上他告诉我的结果了，并且这就是我的答案了。

Harry 知道第八个三角数字是 8 加上第七个三角数字所得到的结果，所以他找到 Sally，并且让 Sally 去求第七个三角数字。这个过程持续不断地把问题从一个人这里传递到另一个人那里。

什么地方是这个传递的终结呢？在这个地方的人必须不再需要得到另外一个人的帮助就能够计算出结果。如果这种情况没有发生，那么就会有一个无限的一个人要求另外一个人的链——这是一种算法的庞氏骗局（Ponzi scheme），它将永远不会结束。在 triangle()方法中目前的情况，意味着调用自身的方法反复地无限执行，最终它将会使这个程序崩溃。

不推卸责任

为了防止无限重复调用自身的过程，在序列中第一个找到三角数字的人，也就是当 n 等于 1 时，他肯定知道结果是 1，没有更小的数字需要询问别人了，也没有什么数字可以去加到其他的数字上了，所以到此为止不再推卸责任。可以给 triangle()方法增加一个条件来表示：

```
int triangle(int n)
   {
   if(n==1)
      return 1;
   else
      return( n + triangle(n-1) );
   }
```

导致递归的方法返回而没有再一次进行递归调用，此时我们称为基值情况（base case）。每一个递归方法都有一个基值（终止）条件，以防止无限地递归下去，以及由此引发的程序崩溃。这一点是至关重要的。

triangle.java 程序

递归真的在运作吗？如果运行 triangle.java 程序，就会看到递归程序确实是这样。输入一个值 n 作为项数，那么程序将会显示对应的三角数字值。程序清单 6.1 显示了 triangle.java 程序。

清单 6.1 triangle.java 程序

```
// triangle.java
// evaluates triangular numbers
// to run this program: C>java TriangleApp
import java.io.*;                    // for I/O
////////////////////////////////////////////////////////////////
class TriangleApp
   {
   static int theNumber;

   public static void main(String[] args) throws IOException
      {
      System.out.print("Enter a number: ");
      theNumber = getInt();
      int theAnswer = triangle(theNumber);
      System.out.println("Triangle="+theAnswer);
      }  // end main()
//-------------------------------------------------------------
   public static int triangle(int n)
      {
      if(n==1)
         return 1;
      else
         return( n + triangle(n-1) );
      }
//-------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
```

```
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//-------------------------------------------------------------
   } // end class TriangleApp
////////////////////////////////////////////////////////////////
```

main()方法提示用户输入一个 n 值，调用 triangle()方法，并且显示返回的结果。triangle()方法反复地调用自身来做所有的工作。

这里是某个例子的输出结果：

```
Enter a number: 1000
Triangle = 500500
```

顺便提一下，如果你怀疑 triangle()的返回结果，可以通过下面的公式检查一下结果的正确性：

第 n 个三角数字 $= (n^2 + n)/2$

到底发生了什么?

下面来修改 triangle()方法，以清楚地了解到当递归执行的时候真正发生了什么。通过插入一些输出语句来跟踪观察参数和返回值：

```
public static int triangle(int n)
   {
   System.out.println("Entering: n=" + n);
   if(n==1)
      {
      System.out.println("Returning 1");
      return 1;
      }
   else
      {
      int temp = n + triangle(n-1);
      System.out.println("Returning " + temp);
      return temp;
      }
   }
```

这里用这个方法替代了早期的 triangle()方法，当用户的输入为 5 时，其输出结果如下：

```
Enter a number: 5

Entering: n=5
```

```
Entering: n=4
Entering: n=3
Entering: n=2
Entering: n=1
Returning 1
Returning 3
Returning 6
Returning 10
Returning 15

Triangle = 15
```

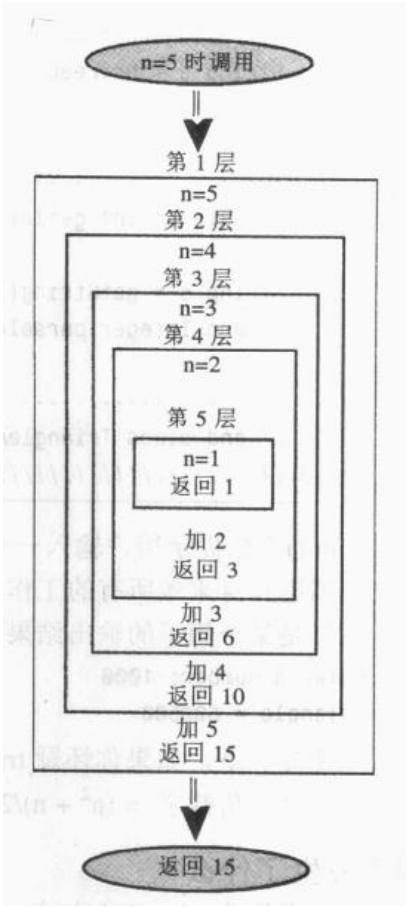


图 6.4 递归的 triangle()方法

triangle()方法调用自身时，它的参数从 5 开始，每次减 1。这个方法反复地进入自身，直到方法的参数减小到 1。于是方法返回。这会引发一系列的返回序列。这个方法像凤凰涅磐一样地返回，脱离被放弃的那一层。每当返回时，这个方法把调用它的参数 n 与其调用下一层方法的返回值相加。

返回值概括了三角数字序列，直到结果返回到 main()。图 6.4 表明了 triangle()方法的每一次调用是如何被想像成在上一层方法的内部的。

注意，在最内层返回 1 之前，实际上在同一时刻有 5 个不同的 triangle()方法实例存在。最外层传入的参数是 5；最内层传入的参数是 1。

递归方法的特征

尽管 triangle()方法很短，但是它拥有所有递归算法都具备的关键特征：

- 调用自身。
- 当它调用自身的时候，它这样做是为了解决更小的问题。
- 存在某个足够简单的问题的层次，在这一层算法不需要调用自己就可以直接解答，且返回结果。

在递归算法每次调用自身的过程中，参数变小（也许是被多个参数描述的范围变小），这反映了问题变小或变简单的事实。当参数或者范围达到一定的最小值时，将会触发一个条件，此时方法不需要调用自身而可以返回。

递归方法有效率吗？

调用一个方法会有一定的额外开销。控制必须从这个调用的位置转移到这个方法的开始处。除此之外，传给这个方法的参数以及这个方法返回的地址都要被压入到一个内部的栈里，为的是这个方法可以访问参数值和知道返回到哪里。

就 triangle()这个方法来讲，因为有上述开销而造成的结果，可能 while 循环方法执行的速度比递归的方法快。在此题中，递归的代价也许不算太大。但是如果由于递归方法的存在，造成了太大规模的方法调用的话，可能会考虑消除递归，在这一章的最后将会详谈一些这方面的问题。

另外一个低效性反映在系统内存空间存储所有的中间参数以及返回值，如果有大量的数据需要

存储，这就会引起栈溢出的问题。

人们常常采用递归，是因为它从概念上简化了问题，而不是因为它本质上更有效率。

数学归纳法

递归就是程序设计中的数学归纳法。数学归纳法是一种通过自身的语汇定义某事物自己的方法。（语汇也被用于描述证明原理的相关方法。）使用归纳法，可以用数学的方式定义三角数字：

tri(n) = 1　　if n = 1

tri(n) = n + tri(n-1)　　if n > 1

用自身来定义某事可能看起来是在转圈子，但是事实上它是完全正确的（假设有一个基值情况）。

阶　乘

阶乘在概念上和三角数字是类似的，只是用乘法取代了加法而已。得到第 n 个三角数字是通过 n 加上第 n-1 个三角数字的和，而 n 的阶乘则是通过 n 乘以 n-1 的阶乘来得到的。也就是说，第 5 个三角数字的值是 5+4+3+2+1，而 5 的阶乘是 5*4*3*2*1，结果等于 120。表 6.1 表示了前 10 个数的阶乘。

表 6.1　阶乘

数字	计算	阶乘
0	由定义	1
1	1 * 1	1
2	2 * 1	2
3	3 * 2	6
4	4 * 6	24
5	5 * 24	120
6	6 * 120	720
7	7 * 720	5040
8	8 * 5040	40320
9	9 * 40320	362880

0 的阶乘被定义为 1。就如你看到的一样，阶乘的数值增长非常快。

一个类似于 triangle()的递归方法可以用于计算阶乘。如下所示：

```
int factorial(int n)
   {
   if(n==0)
      return 1;
   else
      return (n * factorial(n-1) );
   }
```

factorial()和 triangle()方法只有两点不同。第一个是在表达式上 factorial()使用*来代替了+：

```
n * factorial(n-1)
```

第二点是，factorial()的基值条件是当 n 等于 0 时，而不是 n 等于 1 时。下面是这个方法应用于一个类似于 triangle.java 的程序时，某个例子的输出结果：

```
Enter a number: 6
Factorial =720
```

图 6.5 表明了当开始输入 n = 4 时 factorial()调用自己的递归层次是怎么样的。

计算阶乘是一个递归的经典例子，尽管阶乘不像三角数字那么直观。

其他很多数字学的问题都使用递归的类似方法解决，比如找两个数的最大公约数（用于分数化简），求一个数的乘方，等等。再说一次，尽管这些计算可以很好地说明递归，但是它们不太可能用于实际，因为基于循环的方法效率更高。

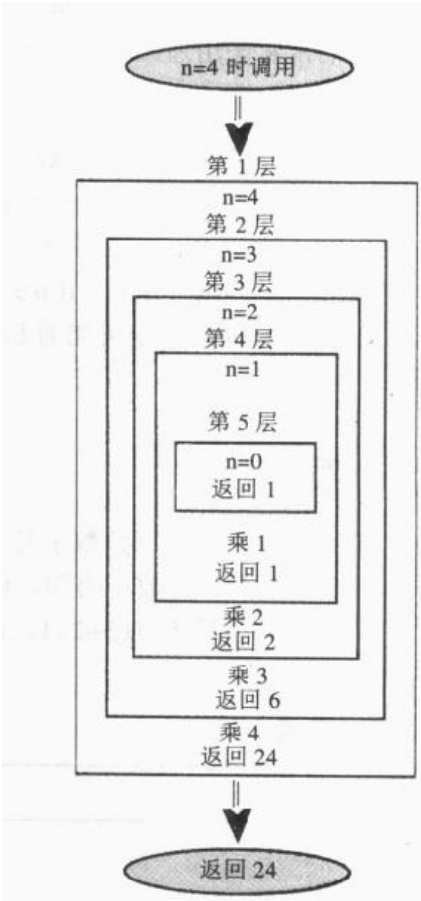


图 6.5 递归的 factorial()方法

变位字

这是递归应用的另外一种情况，在这种情况中递归提供了一种对问题的简捷解决方法。排列是指按照一定的顺序安排事物。假设想要列出一个指定单词的所有变位字，也就是列出该词的全排列（不管这些排列是否是真正的英语单词），它们都是由原来这个单词的字母组成。我们称这个工作是变位一个单词或称全排列一个单词。比如，全排列 cat，会产生

- cat
- cta
- atc
- act
- tca
- tac

试着自己全排列一些单词。会发现能写出的排列的数量是单词字母数的阶乘。对 3 个字母的单词有 6 种可能的词；4 个字母的有 24 个词；5 个字母的有 120 个；等等。（这里是假设所有的字母都是不同的；如果有同一字母出现多次，那么单词数就会少一些。）

你会怎么样写一个程序来全排列单词呢？这里有一个方法。假设这个词有 n 个字母。

1. 全排列最右边的 n-1 个字母。
2. 轮换所有 n 个字母。

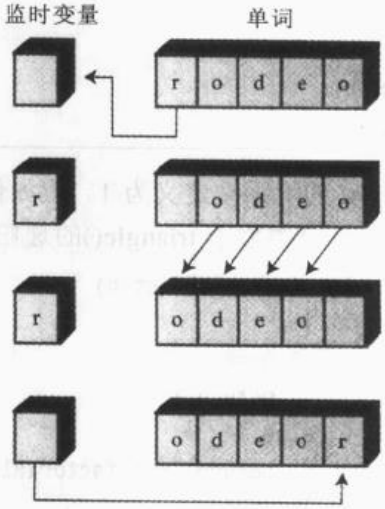


图 6.6 轮换一个单词

3. 重复以上步骤 n 次。

轮换这个词意味着所有的字母向左移一位，但最左边的字母例外，它“转换”至最右边字母的后边，如图 6.6 所示。

轮换单词 n 次，以给每一个字母一个排在单词开头的机会。当选定的字母占据第一个位置时，所有其他的字母被全排列（每一种可能的排列顺序）。对于 cat，只有 3 个字母，轮换剩下的两个字母只是简单地交换它们。结果如表 6.2 所示。

表 6.2 全排列单词 cat

单词	显示单词否?	首字母	剩余的字母	操作
cat	是	c	at	轮换 at
cta	是	c	ta	轮换 ta
cat	否	c	at	轮换 cat
atc	是	a	tc	轮换 tc
act	是	a	ct	轮换 ct
atc	否	a	tc	轮换 atc
tca	是	t	ca	轮换 ca
tac	是	t	ac	轮换 ac
tca	否	t	ca	轮换 tca
cat	否	c	at	完成

注意，在执行 3 个字母的轮换前其他两个字母必须轮换回开始的位置。这会导致出现像 cat，cta，cat 这样的序列。这个多余的单词将不显示。

如何来全排列最右边的 n-1 个字母？通过调用自己。递归的 doAnagram()方法把要被排列的单词的大小作为这个方法的惟一参数。这个单词被看成是这个完整的单词的最右边的 n 个字母。doAnagram()方法每调用自己一次，doAnagram()都会使这个词的字母比上一次少一个，如图 6.7 所示。

当被全排列的这个词的大小只剩一个字母的时候即出现基本情况终止条件发生。没有办法重新排列一个字母，所以方法立即返回。否则，方法把给定词的除了第一个字母之外的所有字母进行全排列，然后接着轮换整个单词。单词的大小为 n 时，这两个操作都执行 n 次。下面是递归的程序 doAnagram()：

```
public static void doAnagram(int newSize)
   {
   if(newSize == 1)                    // if too small,
      return;                          // go no further
   for(int j=0; j<newSize; j++)        // for each position,
      {
      doAnagram(newSize-1);            // anagram remaining
      if(newSize==2)                   // if innermost,
         displayWord();                // display it
      rotate(newSize);                 // rotate word
      }
   }
```

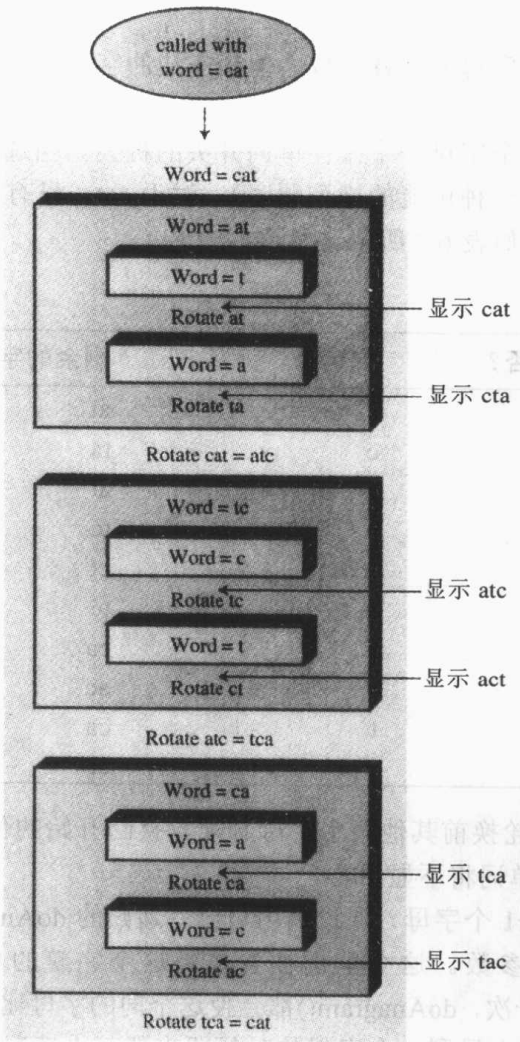


图 6.7 递归的 doAnagram()方法

doAnagram()每次调用自己的时候，词的大小都减少一个字母，并且开始的位置向右移动一位，如图 6.8 所示。

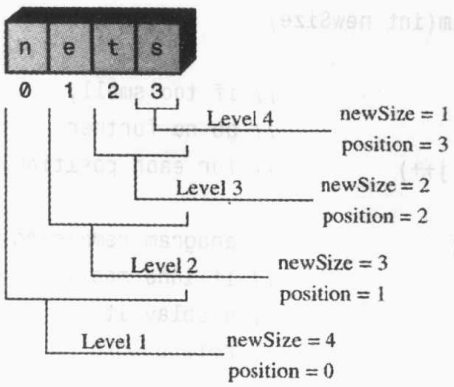


图 6.8 越来越小的词

程序清单 6.2 列出了完整的 anagram.java 程序。main()方法从用户那里得到一个单词，并把这个词插入到一个字符数组内以使它能够便于被处理，然后调用 doAnagram()。

清单 6.2 anagram.java 程序

```
// anagram.java
// creates anagrams
// to run this program: C>java AnagramApp
import java.io.*;
////////////////////////////////////////////////////////////////
class AnagramApp
   {
static int size;
static int count;
static char[] arrChar = new char[100];

public static void main(String[] args) throws IOException
   {
   System.out.print("Enter a word: ");    // get word
   String input = getString();
   size = input.length();                 // find its size
   count = 0;
   for(int j=0; j<size; j++)              // put it in array
      arrChar[j] = input.charAt(j);
   doAnagram(size);                       // anagram it
   }  // end main()
//-------------------------------------------------------------
public static void doAnagram(int newSize)
   {
   if(newSize == 1)                    // if too small,
      return;                          // go no further
   for(int j=0; j<newSize; j++)        // for each position,
      {
      doAnagram(newSize-1);            // anagram remaining
      if(newSize==2)                   // if innermost,
         displayWord();                // display it
      rotate(newSize);                 // rotate word
      }
   }
//-------------------------------------------------------------
// rotate left all chars from position to end
public static void rotate(int newSize)
   {
   int j;
   int position = size - newSize;
   char temp = arrChar[position];      // save first letter
```

```
    for(j=position+1; j<size; j++)       // shift others left
       arrChar[j-1] = arrChar[j];
    arrChar[j-1] = temp;                 // put first on right
    }
  //--------------------------------------------------------
  public static void displayWord()
    {
      if(count < 99)
         System.out.print(" ");
      if(count < 9)
         System.out.print(" ");
      System.out.print(++count + " ");
      for(int j=0; j<size; j++)
         System.out.print( arrChar[j] );
      System.out.print("   ");
      System.out.flush();
      if(count%6 == 0)
         System.out.println("");
      }
    //--------------------------------------------------------
    public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
    //--------------------------------------------------------
    } // end class AnagramApp
  ////////////////////////////////////////////////////////////////
```

rotate()方法就像前面描述的那样向左轮换一个字母。displayWord()方法显示整个词，并且，增加了一个计数以便能容易地看出显示了多少个单词。下面是程序的输入和输出结果：

```
Enter a word: cats
  1 cats     2 cast     3 ctsa     4 ctas     5 csat     6 csta
  7 atsc     8 atcs     9 asct    10 astc    11 acts    12 acst
 13 tsca    14 tsac    15 tcas    16 tcsa    17 tasc    18 tacs
 19 scat    20 scta    21 satc    22 sact    23 stca    24 stac
```

（scat是惟一的符合cats的排列吗？）你可以使用这个程序来全排列5个字母或者甚至是6个字母的单词。但是，由于6的阶乘等于720，全排列这么长的一个单词会产生出比想像中更多的单词。

递归的二分查找

还记得在第2章“数组”中讨论的二分查找吗？想要用最少的比较次数在一个有序的数组中找

到给定的一个数据项。方法是把数组从中间分成两半，然后看要查找的数据项在数组的哪一半，再次地折半，如此这样下去。下面是原来 find()方法的程序：

```
//--------------------------------------------------------
public int find(long searchKey)
   {
   int lowerBound = 0;
   int upperBound = nElems-1;
   int curIn;

   while(true)
      {
      curIn = (lowerBound + upperBound ) / 2;
      if(a[curIn]==searchKey)
         return curIn;              // found it
      else if(lowerBound > upperBound)
         return nElems;             // can't find it
      else                          // divide range
         {
         if(a[curIn] < searchKey)
            lowerBound = curIn + 1; // it's in upper half
         else
            upperBound = curIn - 1; // it's in lower half
         }  // end else divide range
      }  // end while
   }  // end find()
//--------------------------------------------------------
```

可能需要再读一遍第二章中对有序数组二分查找的部分，这一部分描述了这个方法是如何运作的。如果你想要看二分查找是如何操作的，也可以运行那一章中的 Ordered 专题 applet。

可以很容易地把这个基于循环的方法转换成递归的方法。在基于循环的方法中，改变 lowerBound 或者 upperBound 来确定新的查找范围，然后再循环一次。每经过一次循环，就把查找的范围（直接）折半。

递归取代循环

在递归的方法中，不用改变 lowerBound 或者 upperBound，而用 lowerBound 或者 upperBound 的新值作为参数反复调用 find()方法。循环没有了，它的位置被递归调用所取代。下面就是递归的程序：

```
private int recFind(long searchKey, int lowerBound,
                                    int upperBound)
   {
   int curIn;

   curIn = (lowerBound + upperBound ) / 2;
   if(a[curIn]==searchKey)
```

```
    return curIn;              // found it
  else if(lowerBound > upperBound)
    return nElems;             // can't find it
  else                         // divide range
    {
    if(a[curIn] < searchKey)   // it's in upper half
      return recFind(searchKey, curIn+1, upperBound);
    else                       // it's in lower half
      return recFind(searchKey, lowerBound, curIn-1);
    } // end else divide range
  } // end recFind()
```

用 main()代表的类用户，当它调用 find()方法时，可能不知道数组中有多少数据项，并且在任何情况下它都不必知道 upperBound 和 lowerBound 初始值是多少。因此，我们提供了一个中间公有方法。find()，main()用待查找关键字的值作为仅有的一个参数来调用这个 find()方法。find()方法为 lowerBound 和 upperBound（0 到 n-1 个数据项）提供了正确的初值，然后调用私有的递归方法 recFind()。find()方法如下：

```
public int find(long searchKey)
  {
  return recFind(searchKey, 0, nElems-1);
  }
```

清单 6.3 列出了完整的 binarySearch.java 程序的清单。

清单 6.3 binarySearch.java 程序

```
// binarySearch.java
// demonstrates recursive binary search
// to run this program: C>java BinarySearchApp
////////////////////////////////////////////////////////////////
class ordArray
  {
  private long[] a;                 // ref to array a
private int nElems;                 // number of data items
//--------------------------------------------------------
public ordArray(int max)            // constructor
  {
  a = new long[max];                // create array
  nElems = 0;
  }
//--------------------------------------------------------
public int size()
  { return nElems; }
//--------------------------------------------------------
public int find(long searchKey)
  {
```

```
   return recFind(searchKey, 0, nElems-1);
   }
//-------------------------------------------------------
private int recFind(long searchKey, int lowerBound,
                                    int upperBound)
   {
   int curIn;

   curIn = (lowerBound + upperBound ) / 2;
   if(a[curIn]==searchKey)
      return curIn;              // found it
   else if(lowerBound > upperBound)
      return nElems;             // can't find it
   else                          // divide range
      {
      if(a[curIn] < searchKey)   // it's in upper half
         return recFind(searchKey, curIn+1, upperBound);
      else                       // it's in lower half
         return recFind(searchKey, lowerBound, curIn-1);
      } // end else divide range
   } // end recFind()
//-------------------------------------------------------
public void insert(long value)    // put element into array
   {
   int j;
   for(j=0; j<nElems; j++)        // find where it goes
      if(a[j] > value)            // (linear search)
         break;
      for(int k=nElems; k>j; k--)    // move bigger ones up
         a[k] = a[k-1];
      a[j] = value;                  // insert it
      nElems++;                      // increment size
      } // end insert()
   //-------------------------------------------------------
   public void display()             // displays array contents
      {
      for(int j=0; j<nElems; j++)       // for each element,
         System.out.print(a[j] + " ");  // display it
      System.out.println("");
      }
   //-------------------------------------------------------
   } // end class ordArray
////////////////////////////////////////////////////////////////
class BinarySearchApp
   {
```

```
   public static void main(String[] args)
      {
      int maxSize = 100;            // array size
      ordArray arr;                 // reference to array
      arr = new ordArray(maxSize);  // create the array

      arr.insert(72);               // insert items
      arr.insert(90);
      arr.insert(45);
      arr.insert(126);
      arr.insert(54);
      arr.insert(99);
      arr.insert(144);
      arr.insert(27);
      arr.insert(135);
      arr.insert(81);
      arr.insert(18);
      arr.insert(108);
      arr.insert(9);
      arr.insert(117);
      arr.insert(63);
      arr.insert(36);
      arr.display();                // display array

      int searchKey = 27;           // search for item
      if( arr.find(searchKey) != arr.size() )
         System.out.println("Found " + searchKey);
      else
         System.out.println("Can't find " + searchKey);
      } // end main()
   } // end class BinarySearchApp
////////////////////////////////////////////////////////////////
```

在主程序 main()中，在数组中插入 16 个数据项。用 insert()方法插入使这些数据项排列有序；然后显示这些数据项。最后，使用 find()方法来找值为 27 的数据项。下面是例子的输出结果：

```
9 18 27 36 45 54 63 72 81 90 99 108 117 126 135 144
Found 27
```

在 binarySearch.java 程序中，数组里有 16 个数据项。图 6.9 显示了这个程序中 recfind()方法是如何反复地调用自己的，它每一次调用自己都比上一次的范围更小。当最内层的方法找到了指定的数据项，也就是值为 27 的数据项后，方法返回这个数据项的在数组中的下标，即 2（正如在有序数据的显示中看到的一样）。于是这个值依次从每一层 recFind()中返回；最后，find()返回值给类用户。

递归的二分查找和非递归的二分查找有同样的大 O 效率：O(logN)。递归的二分查找更为简洁一些，但是它的速度可能会慢一点。

分治算法

递归的二分查找法是分治算法的一个例子。把一个大问题分成两个相对来说更小的问题，并且分别解决每一个小问题。对每一个小问题的解决方法是一样的：把每个小问题分成两个更小的问题，并且解决它们。这个过程一直持续下去直到达到易于求解的基值情况，就不用再继续分了。

尽管正如在第 2 章中看到的二分查找算法一样，也可以使用非递归的算法，但是分治算法通常要回到递归。

分治算法常常是一个方法，在这个方法中含有两个对自身的递归调用，分别对应于问题的两个部分。在二分查找中，就有两个这样的调用，但是只有一个真的执行了。（调用哪一个取决于关键字的值。）在这一章中，后面将会遇到归并排序，它是真正执行了两个递归调用（对分成两半的数组分别进行排序）。

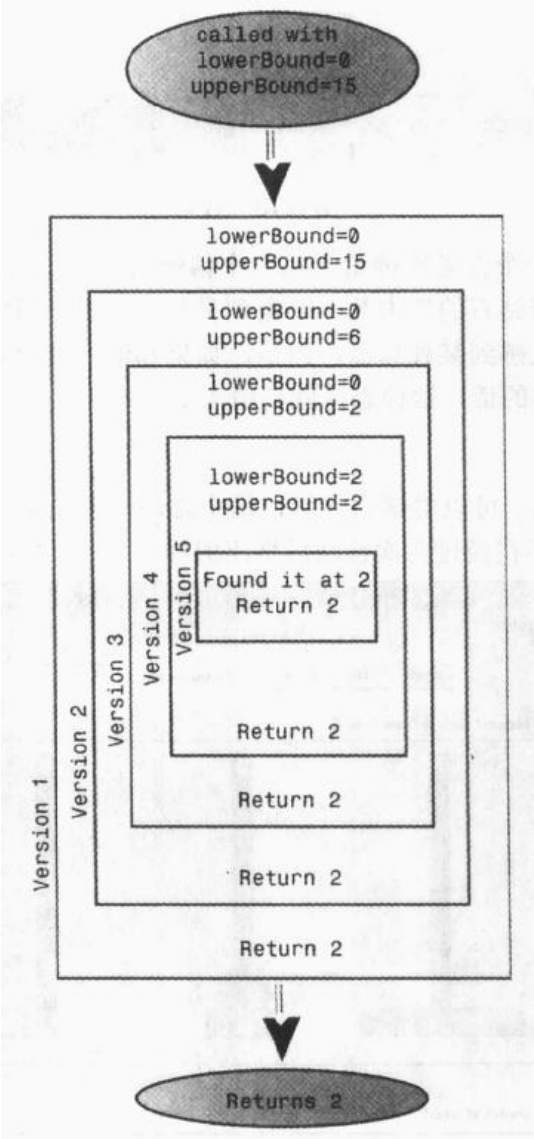


图 6.9 递归的 binarySearch()方法

汉诺（Hanoi）塔问题

汉诺塔问题是由很多放置在三个塔座上的盘子组成的一个古老的难题，如图 6.10 所示。

所有盘子的直径是不同的，并且盘子中央都有一个洞以使它们刚好可以放到塔座上。所有的盘子刚开始都放在塔座 A 上。这个难题的目标是将所有的盘子都从塔座 A 移动到塔座 C 上。每一次只可以移动一个盘子，并且任何一个盘子都不可以放在比自己小的盘子之上。

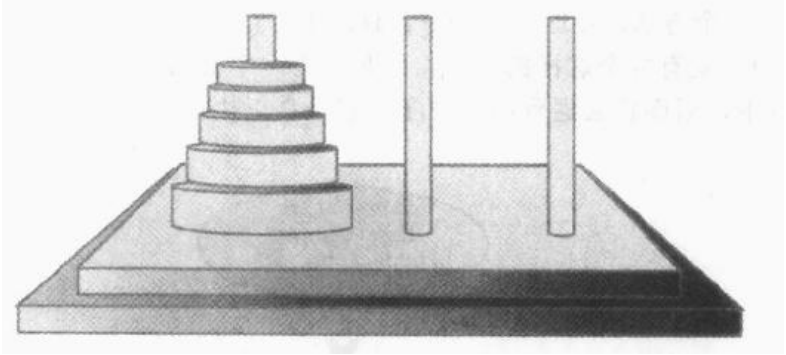


图 6.10 汉诺塔

在印度的某地流传着一个古老的神话，在一个偏僻的庙宇，僧侣们日夜不停的劳动，要把 64 个金制的盘子从三个镶嵌着钻石的塔中的一个搬到另外一个里。当他们完成这个任务的时候，世界就将灭亡了。这可能会使人感到某种恐慌，但是，如果知道在这个难题中即使只是搬比 64 少得多的盘子也要花费多么长时间的话，这种恐慌就会消失了。

Towers 专题 applet

启动 Tower 专题 applet。可以尝试自己解决这个难题，方法是用鼠标拖动最上面的盘子到另外一个塔座上。图 6.11 显示了在经过几次移动后塔座的样子。

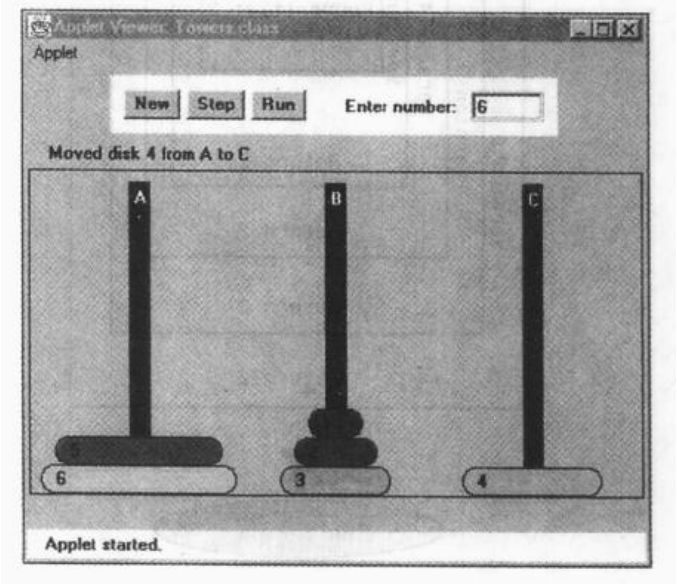


图 6.11 Towers 专题 applet

有三种使用 Towers 专题 applet 的方法：

- 可以手动解决这个难题，用鼠标把盘子从一个塔座拖动到另一个塔座。
- 可以重复点击 Step 按钮来观察这个算法是如何解决这个难题的。算法执行的每一步，都会显示信息，告知算法正在做什么。
- 可以点击 Run 按钮，并且在没有人干涉的情况下观察算法是如何解决这个难题的；盘子将在塔座之间来回快速移动。

重新运行这个 applet，键入想要搬的盘子数目，限于 1 到 10，然后点击 New 两次。（第一次点击 New 之后，要求验证确实想重新开始运行一遍程序。）指定数目的盘子排列在塔座 A 上。一旦用鼠标拖动一个盘子，就不能再使用 Step 按钮或者 Run 按钮了，否则必须用 New 重新开始。但是，当在步进（Step）或者运行（Run）的情况下可以切换成手动运行，而且在运行（Run）的状态下你也可以切换到步进（Step）状态，以及可以从步进的状态切换到运行状态。

试用少量的盘子来手动地解决这个难题，比如 3 个或者 4 个盘子。逐步增加盘子的数目。这个 applet 提供了一个直观学习的机会，以了解这个问题是怎样解决的。

移动的子树

在塔座 A 上盘子初始的树形（或金字塔形）排列称为一棵树。（这种树和本书其他地方提到的作为数据存储结构的树无关。）当用 applet 作实验时，可能会注意到生成盘子的较小的树形堆是问题解决过程中的一步。这些所含盘子数小于盘子总数的较小的树称为子树。举例来说，如果要移动 4 个盘子，会发现中间的一个状态是有 3 个盘子的子树在塔座 B 上，如图 6.12 所示。

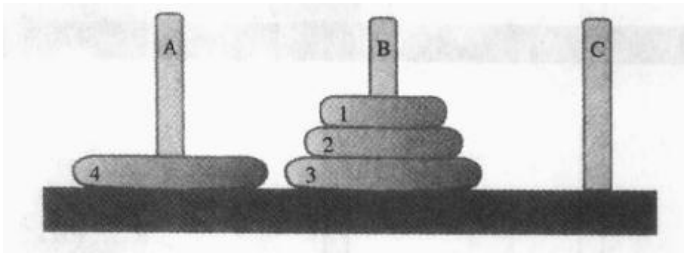


图 6.12 塔座 B 上的一颗子树

这些子树在这个难题的解决过程中会形成多次。子树形成多次是因为一棵子树的形式是把一个更大的盘子从一个塔座上转移到另一个塔座上的惟一方法：所有的小盘子都必须先放置在一个中介的塔座上，在这个中介的塔座上自然就会形成一棵子树。

当手动地解决这个难题的时候，有一个经验法则，可以提供帮助。如果试图要移动的子树含有奇数个盘子，开始时直接把最顶端的盘子移动到想要把这棵子树移动到的那个塔座上。如果试图要移动一棵含有偶数个盘子的子树，那么开始时要把最顶端的盘子移动到中介塔座上。

递归的算法

用子树的概念可以递归地表示出汉诺塔难题的解决办法。假设想要把所有的盘子从源塔座上（称为 S）移动到目标塔座上（称为 D）。有一个可以使用的中介塔座（称为 I）。假定在塔座 S 上有 n 个盘子。算法如下：

1. 从塔座 S 移动包含上面的 n-1 个盘子的子树到塔座 I 上。

2. 从塔座 S 移动剩余的盘子（最大的盘子）到塔座 D 上。

3. 从塔座 I 移动子树到塔座 D。

当开始的时候，源塔座是 A，中介塔座是 B，目标塔座是 C。图 6.13 显示了这种情况的三个步骤。

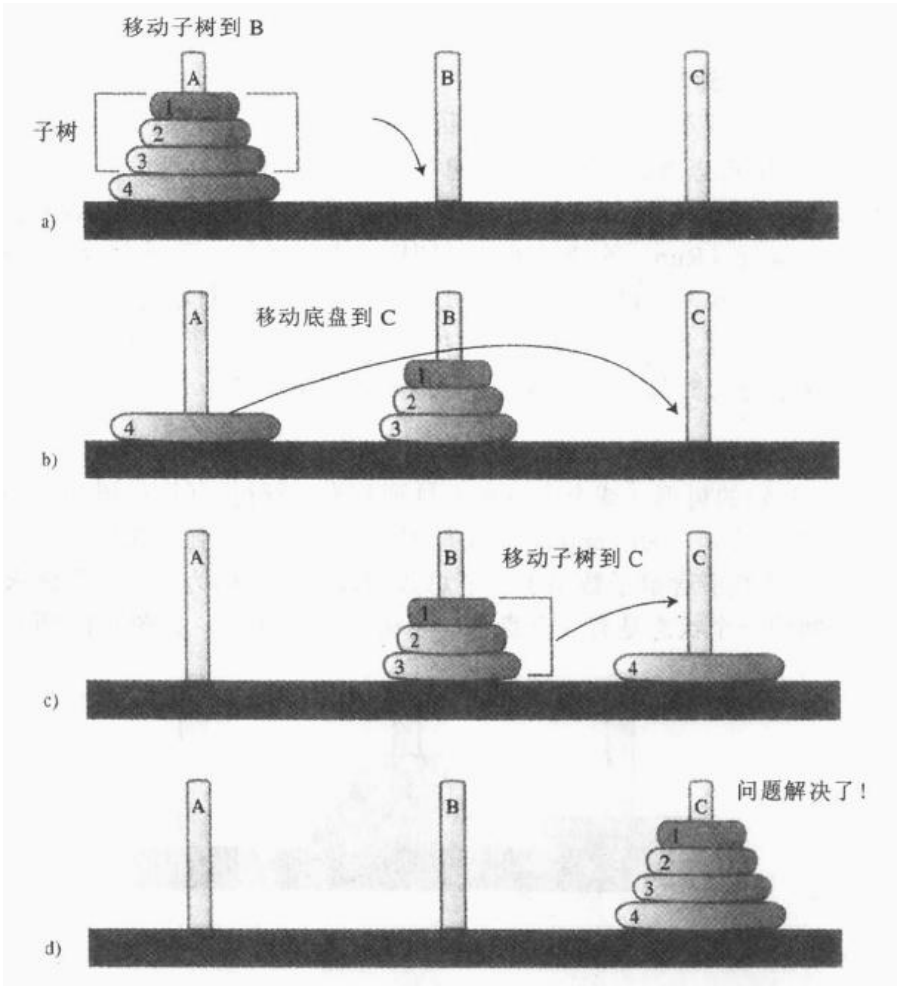


图 6.13 汉诺塔问题的递归解决办法

首先，包括盘子 1、2 和 3 的子树被移动到中介塔座 B 上。于是，最大的盘子 4，移动到塔座 C 上。然后子树从塔座 B 移动到塔座 C 上。

当然，这个方法没有解决如何把包括盘子 1、2 和 3 的子树移动到塔座 B 上的问题，因为不能一次性的移动一颗子树；每次只能移动一个盘子。移动三个盘子的子树不是那么容易的。但是，这比移动 4 个盘子要容易。

从塔座 A 上移动三个盘子到目标塔座 B 可以通过像移动四个盘子时一样的三个步骤来完成。那也就是说，从塔座 A 上移动包括最上面的两个盘子的子树到中介塔座 C 上；接着从 A 上移动盘子 3 到塔座 B 上。然后把子树从塔座 C 移回塔座 B。

如何把一棵有两个盘子的子树从塔座 A 上移动到塔座 C 上呢？从塔座 A 上移动只有一个盘子（盘子 1）的子树到塔座 B 上。这是基值条件：当移动一个盘子的时候，只要移动它就可以了；没

有其他的事情要做。然后从塔座 A 移动更大的盘子（盘子 2）到塔座 C，并且把这棵子树（盘子 1）重新放置在这个更大的盘子上。

towers.java 程序

towers.java 程序使用递归的办法解决了汉诺塔难题。这个程序通过显示来报告所发生的移动；这个递归算法比显示汉诺塔的代码要少得多。这个算法适合于人来读这个程序清单，然后实际执行这些移动。

这个程序的代码极为简单。main()方法调用了递归方法 doTowers()。然后 doTower()方法递归的调用自己，直到解决这个难题。在如程序清单 6.4 中显示的这个程序中，初始时只有三个盘子，但是可以用任何盘子数来重新编译这个程序。

清单 6.4 towers.java 程序

```
// towers.java
// solves the towers of Hanoi puzzle
// to run this program: C>java TowersApp
////////////////////////////////////////////////////////////////
class TowersApp
   {
   static int nDisks = 3;

   public static void main(String[] args)
      {
      doTowers(nDisks, 'A', 'B', 'C');
      }
   //-------------------------------------------------------------
   public static void doTowers(int topN,
                               char from, char inter, char to)
      {
      if(topN==1)
         System.out.println("Disk 1 from " + from + " to "+ to);
      else
         {
         doTowers(topN-1, from, to, inter);  // from-->inter

         System.out.println("Disk " + topN +
                            " from " + from + " to "+ to);
         doTowers(topN-1, inter, from, to);  // inter-->to
         }
      }
   //-------------------------------------------------------------
   } // end class TowersApp
////////////////////////////////////////////////////////////////
```

记住三个盘子从塔座 A 移动到塔座 C。下面是这个程序的输出结果：

```
Disk 1 from A to C
Disk 2 from A to B
Disk 1 from C to B
Disk 3 from A to C
Disk 1 from B to A
Disk 2 from B to C
Disk 1 from A to C
```

doTowers()的参数是要移动的盘子的数目，以及会使用到的源塔座（from）、中介塔座（inter）和目标塔座（to）。盘子数随着方法每调用一次就递减1。源塔座、中介塔座和目标塔座也就发生变化。

下面是有附加说明的输出结果，这些结果表明了何时进入doTowers()方法，何时从该方法返回，方法的参数，以及盘了是在基值情况下（子树只包含一个盘子）移动的还是在子树移走后，移动的是留在底部的大盘。

```
Enter (3 disks): s=A, i=B, d=C
   Enter (2 disks): s=A, i=C, d=B
      Enter (1 disk): s=A, i=B, d=C
         Base case: move disk 1 from A to C
      Return (1 disk)
      Move bottom disk 2 from A to B
      Enter (1 disk): s=C, i=A, d=B
         Base case: move disk 1 from C to B
      Return (1 disk)
   Return (2 disks)
   Move bottom disk 3 from A to C
   Enter (2 disks): s=B, i=A, d=C
      Enter (1 disk): s=B, i=C, d=A
         Base case: move disk 1 from B to A
      Return (1 disk)
      Move bottom disk 2 from B to C
      Enter (1 disk): s=A, i=B, d=C
         Base case: move disk 1 from A to C
      Return (1 disk)
   Return (2 disks)
Return (3 disks)
```

如果连同着 doTower()的源代码来一起研究这个输出结果，那么会很清楚确切地表示出这个方法是如何运作的。这是多么惊人呀，这么少量的代码就可以解决看起来如此复杂的问题。

归并排序

最后一个递归的例子是归并排序。归并排序比在第3章“简单排序”中看到的排序方法要有效得多，至少在速度上是这样的。冒泡排序、插入排序和选择排序要用 $O(N^2)$时间，而归并排序只要O(N*logN)。图2.9（第2章中）中的图表表明了归并排序要比简单排序快多少，如果N（被排序的

数据项的数目）是 10000，那么 N^2 就是 100000000，而 N*logN 只是 40000。如果为这么多数据项排序用归并排序的话需要 40 秒，那么用插入排序则会需要将近 28 个小时。

归并排序也相当容易实现。归并排序在概念上比将要在下一章中看到的快速排序和希尔排序都要容易理解。

归并排序的一个缺点是它需要在存储器中有另一个大小等于被排序的数据项数目的数组。如果初始数组几乎占满整个存储器，那么归并排序将不能工作。但是，如果有足够的空间，归并排序会是一个很好的选择。

归并两个有序数组

归并算法的中心是归并两个已经有序的数组。归并两个有序数组 A 和 B，就生成了第三个数组 C，数组 C 包含数组 A 和 B 的所有数据项，并且使它们有序的排列在数组 C 中。首先考察归并的过程；然后看它是如何在排序中使用的。

假设有两个有序数组，不要求有相同的大小。设数组 A 有 4 个数据项，数组 B 有 6 个数据项。它们要被归并到数组 C 中，开始时数组 C 有 10 个空的存储空间。图 6.14 显示了这些数组。

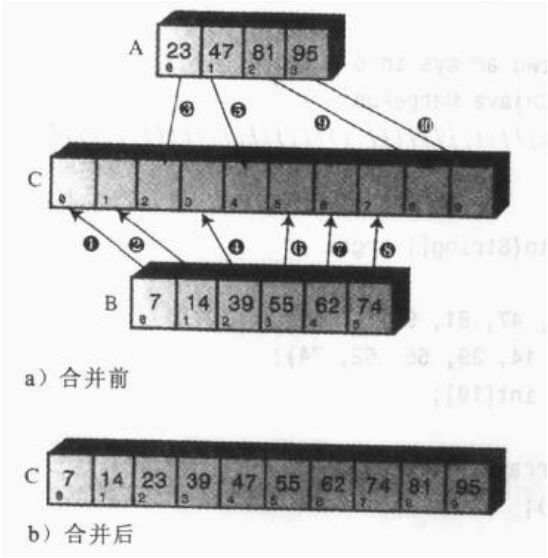


图 6.14 归并两个数组

在这个图中，带圈的数字显示了把数组 A 和 B 中的数据项转移到数组 C 中的顺序。表 6.3 显示了必要的比较，由此来决定要复制哪个数据项。表中的步骤对应于这个图中的步骤。在每一次比较之后，较小的数据项被复制到数组 A 中。

表 6.3 归并操作

步骤	比较（如果有的话）	复制
1	比较 23 和 7	复制 7 从 B 至 C
2	比较 23 和 14	复制 14 从 B 至 C
3	比较 23 和 39	复制 23 从 A 至 C
4	比较 39 和 47	复制 39 从 B 至 C

续表

步骤	比较（如果有的话）	复制
5	比较 55 和 47	复制 47 从 A 至 C
6	比较 55 和 81	复制 55 从 B 至 C
7	比较 62 和 81	复制 62 从 B 至 C
8	比较 74 和 81	复制 74 从 B 至 C
9		复制 81 从 A 至 C
10		复制 95 从 A 至 C

注意，由于数组 B 在第八步以后是空的，所以不需要再去比较了；只要把数组 A 中所有剩余的数据项复制到数组 C 即可。

清单 6.5 显示了一个如图 6.14 和表 6.3 所示的执行归并排序的 Java 程序。它不是递归的程序，只是理解归并排序的序曲。

清单 6.5 merge.java 程序

```
// merge.java
// demonstrates merging two arrays into a third
// to run this program: C>java MergeApp
////////////////////////////////////////////////////////////////
class MergeApp
   {
   public static void main(String[] args)
      {
      int[] arrayA = {23, 47, 81, 95};
      int[] arrayB = {7, 14, 39, 55, 62, 74};
      int[] arrayC = new int[10];

      merge(arrayA, 4, arrayB, 6, arrayC);
      display(arrayC, 10);
      } // end main()
   //-------------------------------------------------------------
                                    // merge A and B into C
     public static void merge( int[] arrayA, int sizeA,
                               int[] arrayB, int sizeB,
                               int[] arrayC )
      {
      int aDex=0, bDex=0, cDex=0;

      while(aDex < sizeA && bDex < sizeB)  // neither array empty
         if( arrayA[aDex] < arrayB[bDex] )
            arrayC[cDex++] = arrayA[aDex++];
         else
            arrayC[cDex++] = arrayB[bDex++];
```

```
    while(aDex < sizeA)                 // arrayB is empty,
       arrayC[cDex++] = arrayA[aDex++]; // but arrayA isn't

    while(bDex < sizeB)                 // arrayA is empty,
       arrayC[cDex++] = arrayB[bDex++]; // but arrayB isn't
    }  // end merge()
  //--------------------------------------------------------
                                    // display array
  public static void display(int[] theArray, int size)
    {
    for(int j=0; j<size; j++)
       System.out.print(theArray[j] + " ");
    System.out.println("");
    }
  //--------------------------------------------------------
  }  // end class MergeApp
////////////////////////////////////////////////////////////////
```

在 main()中创建数组 arrayA、arrayB 和数组 arrayC；然后调用 merge()方法把数组 A 和数组 B 归并到数组 C 中，并且显示数组 C 中的数据项。下面是输出结果：

```
7 14 23 39 47 55 62 74 81 95
```

merge()方法有三个 while 循环。第一个 while 循环是沿着数组 arrayA 和数组 arrayB 走，比较它们的数据项，并且复制它们中较小的数据项到数组 arrayC。

第二个 while 循环处理当数组 arrayB 的所有数据项都已经移出，而数组 arrayA 还有剩余数据项的情况。（这就是在本例中所发生的情况，数组 arrayA 中还有数据项 81 和 95。）这个循环把剩余的数据项直接从数组 arrayA 中复制到 arrayC 中。

第三个 while 循环处理相似的情况，即当数组 arrayA 所有的数据项都已经移出，而数组 arrayB 中还有剩余数据项的情况；那些剩余的数据项被复制到数组 arrayC 中。

通过归并进行排序

归并排序的思想是把一个数组分成两半，排序每一半，然后用 merge()方法把数组的两半归并成一个有序的数组。如何来为每一部分排序呢？这一章讲述的是递归，所以大概已经有答案了：把每一半都分成两个四分之一，对每个四分之一部分排序，然后把它们归并成一个有序的一半。

类似的，每一对八分之一部分归并成一个有序的四分之一部分，每一对十六分之一部分归并成一个有序的八分之一部分，依此类推。反复地分割数组，直到得到的子数组只含有一个数据项。这就是基值条件：设定只有一个数据项的数组是有序的。

前面已经看到递归方法在每次调用自身方法的时候通常某个参数的大小都会减小，并且方法每次返回时参数值又恢复到以前。在 mergeSort()方法中，每一次这个方法调用自身的时候排列都会被分成两部分，并且方法每一次返回时都会把两个较小的排列合并成一个更大的排列。

当 mergeSort()发现两个只有一个数据项的数组时，它就返回，把这两个数据项归并到一个有两个数据项的有序数组中。每个生成的一对两个数据项的数组又被合并成一个有 4 个数据项的有序数

组。这个过程一直持续下去，数组越来越大直到整个数组有序。当初始的数组大小是二的乘方的时候，是最容易看明白的，正如图 6.15 所示。

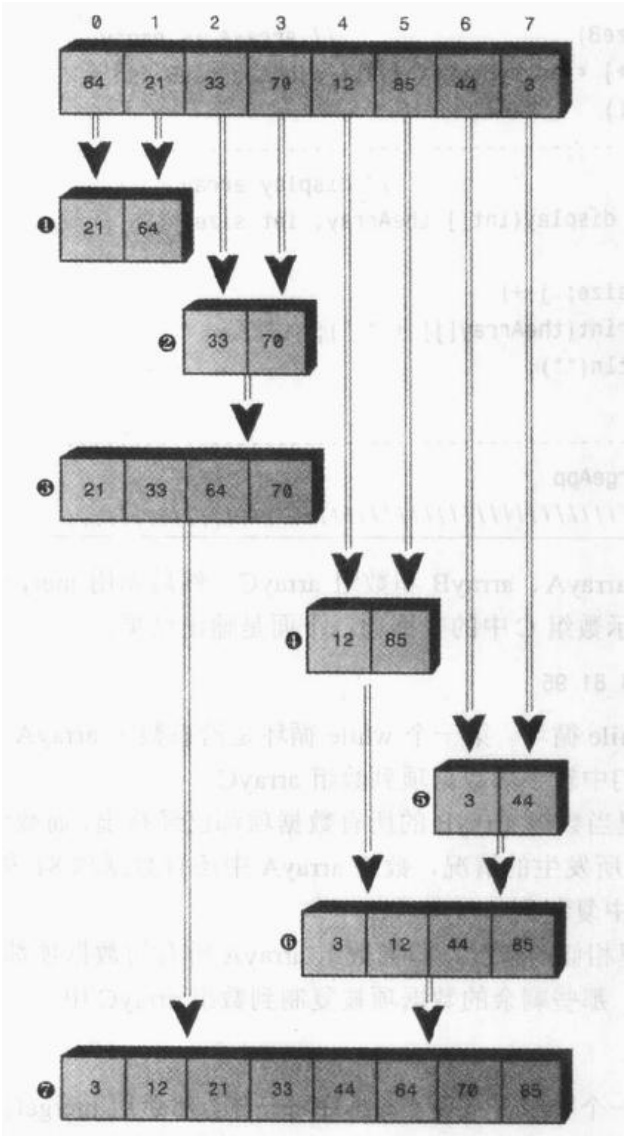


图 6.15 归并越来越大的数组

首先，在数组的下半区，位置 0-0 的数据项和位置 1-1 的数据项被归并到位置 0-1。当然，0-0 和 1-1 不是真正的排列；它们只是一个数据项，所以它们是基值条件。类似的，2-2 和 3-3 被归并到 2-3。于是排列 0-1 和排列 2-3 被归并到 0-3。

在数组的上半区，4-4 和 5-5 被归并到 4-5，6-6 和 7-7 被归并到 6-7，并且 4-5 和 6-7 被归并到 4-7。最后，前一部分 0-3 和后一部分 4-7 被归并到完全的数组 0-7 中，现在这个数组已经是有序的了。

当数组的大小不是 2 的乘方的时候，必须要归并不同大小的数组。例如，图 6.16 所显示的数组的大小是 12 的那种情况。这里是一个大小为 2 的数组要和一个大小为 1 的数组归并为一个大小为 3 的数组。

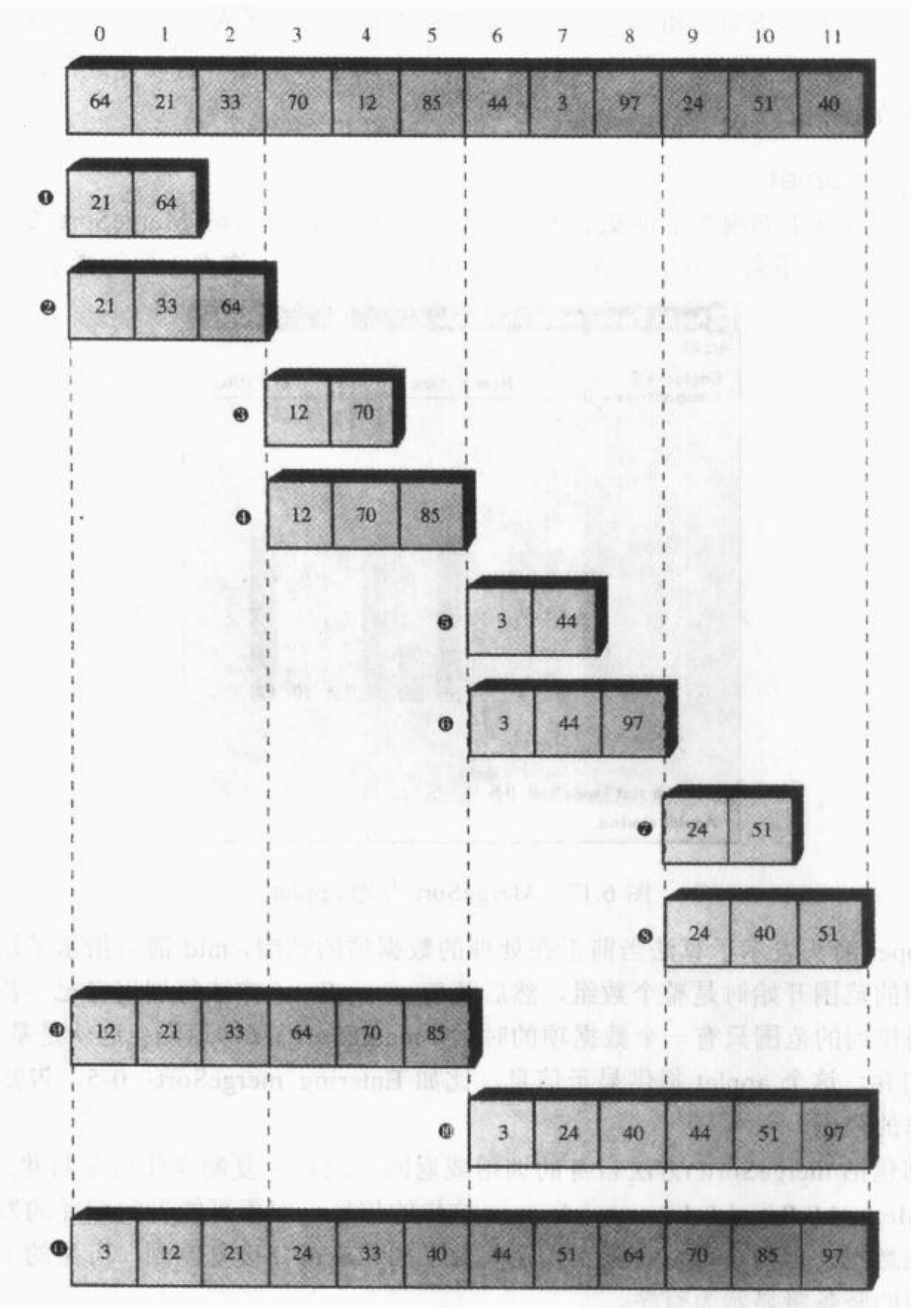


图 6.16 数组的大小不是 2 的乘方

首先，只有一个数据项的 0-0 和 1-1 被归并到有两个数据项的排列 0-1 上。然后排列 0-1 和只有一个数据项的 2-2 合并。这就生成了一个 3 个数据项的排列 0-2。它和三个数据项的排列 3-5 归并。这个过程一直持续到整个数组有序。

注意，在归并排序中，不能像在 merge.java 程序中演示的那样，把两个单独的数组归并为第三个数组，而要把一个数组的不同部分归并到这个数组中。

读者可能会问，所有的这些子数组都存放在存储器的什么地方。在这个算法中，创建了一个和

初始数组一样大小的工作空间数组。这些子数组存储在这个工作空间数组的这个部分中。这也就是说原始数组中的子数组被复制到工作空间数组的相应空间上。在每一次归并之后，工作数组的内容被复制回原来的数组中。

MergeSort 专题 applet

在亲眼见到眼前发生的现象后能更容易领悟这个排序过程。启动 MergeSort 专题 applet，反复点击 Step 按钮，归并排序会一步一步地执行。图 6.17 显示了前三次点击后的样子。

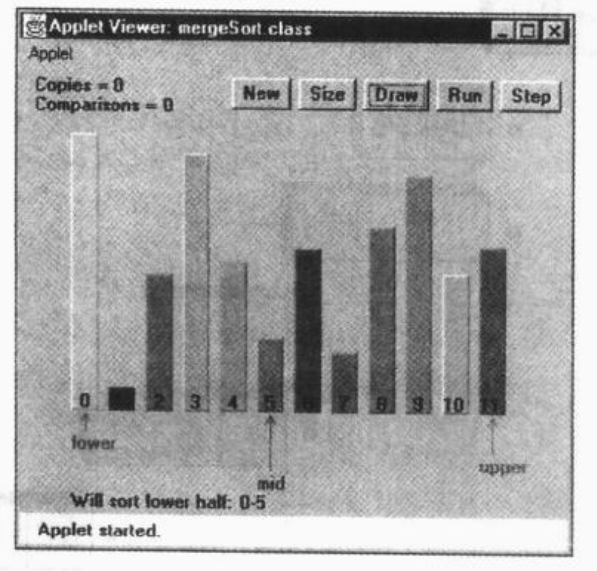


图 6.17 MergeSort 专题 applet

lower 和 upper 箭头表示了算法当前正在处理的数据项的范围，mid 箭头指示了这个范围的中间位置。这个排列的范围开始时是整个数组，然后随着 mergeSort()方法每调用自己一次，这个范围就被分成两半。当排列的范围只有一个数据项的时候，mergeSort()立即返回；这就是基值条件。否则，两个子数组被归并。这个 applet 提供显示信息，比如 Entering mergeSort: 0-5，告知 applet 正在做什么和正在操作的范围。

许多步骤都包括 mergeSort()方法自身的调用或返回。比较和复制操作只在归并的过程中执行，这时会看到像 Merged 0-0 and 1-1 into workspace 这样的信息。但不可能看到归并的发生，因为不显示这个工作空间数组。但是，可以看到工作空间数组的相应部分被复制回（可见的）原始数组的结果：指定范围内的竖条将显示为有序。

首先，前两个竖条被排序，接着是前三个竖条，然后是在位置 3-4 上的两个竖条，再下次是位置在 3-5 上的三个竖条，然后是在位置 0-5 上的 6 个竖条被排序，照此排列下去，如图 6.16 中显示的相应的序列。最后，所有的竖条都将有序。

可以通过点击 Run 按钮使这个算法连续运行。并且可以在任何时候点击 Step 来停止这个连续的过程，然后可以单步执行任意多次，如果再次点击 Run 可以继续连续运行。

正如在其他的排序专题 applet 中一样，点击 New 按钮来重新设置一组没有排序的竖条数组，而且竖条可以设置成任意顺序或者是逆序排列。Size 按钮可以选择竖条数为 12 或 100。

观察这个算法对 100 个逆序的竖条排序特别有指导意义。结果的模式清楚地显示了每段范围是

如何被分别排列为有序的，以及如何和它的另一半归并，它还显示了范围是如何变得越来越大的。

mergeSort.java 程序

马上就会看到完整的 mergeSort.java 程序。首先，把注意力集中到执行归并排序的方法。下面就是它的程序代码：

```
private void recMergeSort(long[] workSpace, int lowerBound,
                                            int upperBound)
   {
   if(lowerBound == upperBound)            // if range is 1,
      return;                              // no use sorting
   else
      {                                    // find midpoint
      int mid = (lowerBound+upperBound) / 2;
                                           // sort low half
      recMergeSort(workSpace, lowerBound, mid);
                                           // sort high half
      recMergeSort(workSpace, mid+1, upperBound);
                                           // merge them
      merge(workSpace, lowerBound, mid+1, upperBound);
      }  // end else
   }  // end recMergeSort
```

正如上面所看到的一样，除了基值条件之外，在这个方法中只有四条语句。一句是计算中间位置的，还有两个递归调用方法 recMergeSort()（每一个对应于数组的一半），最后一句是 merge()的调用，它来归并两个有序的部分。当这个范围只包含一个数组数据项（lowerBound == upperBound）的时候基值条件发生，并且立即返回。

在 mergeSort.java 程序中，mergeSort()方法是类用户可以实际上看到的一个方法。mergeSort()创建数组 workSpace[]，然后调用递归的程序 recMergeSort()来执行排序。workspace 数组的创建在 mergeSort()中实现，是因为在 recMergeSort()中创建数组会在每一次递归调用中都创建新数组，这是没有效率的。

在以前的 merge.java 程序（清单 6.5）中，merge()方法在三个独立的数组上操作：两个原数组以及一个目标数组。在 mergeSort.java 程序中的 merge()方法只在一个数组上进行操作：DArray 类的数据成员 theArray。merge()方法的参数是前半部分的子数组的开始位置，后半部分子数组的开始位置，以及后半部分子数组的上界。这个方法用这些信息来计算子数组的大小。

清单 6.6 显示了完整的 mergeSort.java 程序，这个程序使用了第 2 章中数组类的一个变型，在 DArray 类中增加了 mergeSort()方法和 recMergeSort()方法。main()方法创建了一个数组，插入 12 个数据项，显示数组，用 mergeSort()给数组数据项排序，并且会再次显示数组。

清单 6.6 mergeSort.java 程序

```
// mergeSort.java
// demonstrates recursive merge sort
// to run this program: C>java MergeSortApp
```

```
////////////////////////////////////////////////////////////////
class DArray
   {
   private long[] theArray;          // ref to array theArray
   private int nElems;               // number of data items
   //--------------------------------------------------------
   public DArray(int max)            // constructor
      {
      theArray = new long[max];      // create array
      nElems = 0;
      }
   //--------------------------------------------------------
   public void insert(long value)    // put element into array
      {
      theArray[nElems] = value;      // insert it
      nElems++;                      // increment size
      }
   //--------------------------------------------------------
   public void display()             // displays array contents
      {
      for(int j=0; j<nElems; j++)    // for each element,
         System.out.print(theArray[j] + " ");  // display it
      System.out.println("");
      }
   //--------------------------------------------------------
   public void mergeSort()           // called by main()
      {                              // provides workspace
      long[] workSpace = new long[nElems];
      recMergeSort(workSpace, 0, nElems-1);
      }
   //--------------------------------------------------------
   private void recMergeSort(long[] workSpace, int lowerBound,
                                               int upperBound)
      {
     if(lowerBound == upperBound)           // if range is 1,
        return;                             // no use sorting
     else
        {                                   // find midpoint
        int mid = (lowerBound+upperBound) / 2;
                                            // sort low half
        recMergeSort(workSpace, lowerBound, mid);
                                            // sort high half
        recMergeSort(workSpace, mid+1, upperBound);
                                            // merge them
        merge(workSpace, lowerBound, mid+1, upperBound);
```

```
      } // end else
    } // end recMergeSort()
  //--------------------------------------------------------
  private void merge(long[] workSpace, int lowPtr,
                     int highPtr, int upperBound)
    {
    int j = 0;                              // workspace index
    int lowerBound = lowPtr;
    int mid = highPtr-1;
    int n = upperBound-lowerBound+1;        // # of items

    while(lowPtr <= mid && highPtr <= upperBound)
      if( theArray[lowPtr] < theArray[highPtr] )
        workSpace[j++] = theArray[lowPtr++];
      else
        workSpace[j++] = theArray[highPtr++];

    while(lowPtr <= mid)
      workSpace[j++] = theArray[lowPtr++];

    while(highPtr <= upperBound)
      workSpace[j++] = theArray[highPtr++];

    for(j=0; j<n; j++)
      theArray[lowerBound+j] = workSpace[j];
    } // end merge()
  //--------------------------------------------------------
  } // end class DArray
////////////////////////////////////////////////////////////////
class MergeSortApp
  {
  public static void main(String[] args)
    {
    int maxSize = 100;              // array size
    DArray arr;                     // reference to array
    arr = new DArray(maxSize);      // create the array

    arr.insert(64);                 // insert items
    arr.insert(21);
    arr.insert(33);
    arr.insert(70);
    arr.insert(12);
    arr.insert(85);
    arr.insert(44);
    arr.insert(3);
```

```
      arr.insert(99);
      arr.insert(0);
      arr.insert(108);
      arr.insert(36);

      arr.display();                // display items

      arr.mergeSort();              // merge sort the array

      arr.display();                // display items again
      } // end main()
   } // end class MergeSortApp
////////////////////////////////////////////////////////////////
```

程序的输出结果简单地显示了没排序和已经排序的数组：

```
64 21 33 70 12 85 44 3 99 0 108 36
0 3 12 21 33 36 44 64 70 85 99 108
```

如果在 recMergeSort()方法中添加一些额外的语句，就可以在排序的过程中对程序正在做的事情生成运行时的注释。下面的输出结果表明了四个数据项的数组{64,21,33,70}是怎样排序的。（可以认为这就是图 6.15 中所示的下半部分数组一样。）

```
Entering 0-3
   Will sort low half of 0-3
   Entering 0-1
      Will sort low half of 0-1
         Entering 0-0
         Base-Case Return 0-0
      Will sort high half of 0-1
         Entering 1-1
         Base-Case Return 1-1
      Will merge halves into 0-1
   Return 0-1                       theArray=21 64 33 70
   Will sort high half of 0-3
   Entering 2-3
      Will sort low half of 2-3
         Entering 2-2
         Base-Case Return 2-2
      Will sort high half of 2-3
         Entering 3-3
         Base-Case Return 3-3
      Will merge halves into 2-3
   Return 2-3                       theArray=21 64 33 70
   Will merge halves into 0-3
Return 0-3                          theArray=21 33 64 70
```

如果它给四个数据项排序的话，这个结果和 MergeSort 专题 applet 输出的结果几乎是相同的。

研究这个输出结果，并且和 recMergeSort()的代码以及图 6.15 进行比较，就会看出这个排序过程的细节。

归并排序的效率

正如前面提到的那样，归并排序的运行时间是 O(N*logN)。如何知道这个时间的呢？首先看在这个算法执行的过程当中，如何计算一个数据项被复制的次数以及和其他数据项比较的次数。假设复制和比较是最费时的操作，递归的调用和返回不增加额外的开销。

复制的次数

考虑图 6.15。首行下面的每一个单元都代表了从数组复制到工作空间中的一个数据项。

把图 6.15 中所有的单元加在一起（7 个标数字的步骤）表明了需要有 24 次复制来给 8 个数据项排序。Log_2（8）等于 3，所以 8*log2(8)等于 24。这说明，对于有 8 个数据项的情况，复制的次数和 N*log2(N)成正比。

另一种用来看计算复制次数的方法是，排序 8 个数据项需要有 3 层，每一层包含 8 次复制。一层意味着所有元素都复制到相同大小的子数组中。在第一层中，有四个含有两个数据项的子数组；在第二层中，有两个含有四个数据项的子数组；在第三层中，有一个含有 8 个数据项的子数组。每一层含有 8 个数据项，所以也就是有 3*8 或者说有 24 次复制。

在图 6.15 中，只考虑图的一半，你可以看到对于一个有 4 个数据项的数组需要有 8 次复制（步骤 1、2 和 3），对于两个数据项来说需要 2 次复制。相似的计算可以得到对于更大的数组来说需要的复制次数。表 6.4 概略表明了这个信息。

表 6.4 当 N 是 2 的乘方时的操作的次数

N	log_2N	复制到工作区的次数	复制次数	最多比较次数
2	1	2	4	1(1)
4	2	8	16	5(4)
8	3	24	48	17(12)
16	4	64	128	49(32)
32	5	160	320	129(80)
64	6	384	768	321(192)
128	7	896	1792	769(448)

实际上，这些数据项不仅被复制到数组 workspace 中，而且也会被复制回原数组中。这会使复制的次数增加一倍，正如在总复制数一栏中显示的一样。表 6.4 的最后一栏表明了比较的次数，这一点马上会提到。

当 N 不是 2 的倍数的时候计算复制和比较的次数是比较困难的，但是这些次数会落在 2 的乘方之间。对于 12 个数据项，总共有 88 次复制，并且对于 100 个数据项，总共有 1344 次复制。

比较的次数

在归并排序算法中，比较的次数总是比复制的次数略微少一些的。那么少多少呢？假设数据项的个数是 2 的乘方，对于每一个独立的归并操作，比较的最大次数总是比正在被归并的数据项个数少一，并且比较的最少次数是正在被归并的数据项数目的一半。可以在图 6.18 中看到这为什么是正确的，这个图表明了试图归并各有四个数据项的两个数组时的两种可能性。

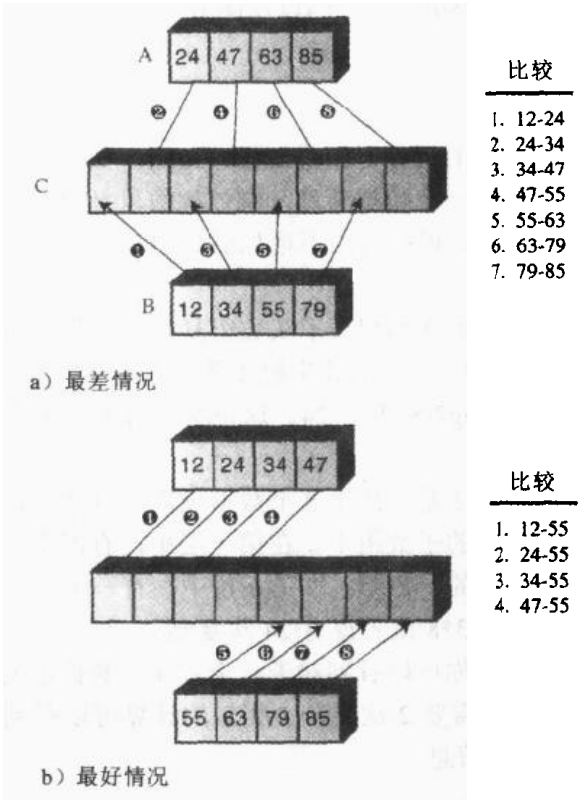


图 6.18 最大和最小比较次数

在第一种情况中，数据项大小交织排列，所以必须要进行七次比较来归并这些数据项。在第二种情况中，在一个数组中的所有数据项都比在另外一个数组中的数据项小，因此只需要有四次比较。

对每一次排序都要有多次的归并，所以必须把为每一次排序中的比较次数加起来。重新来看图6.15，可以看到为 8 个数据项进行排序，需要有七次归并操作。正在被归并的数据项个数以及相应的比较次数如表 6.5 所示。

表 6.5 包含 8 个数据项的比较次数

步骤	1	2	3	4	5	6	7	共计
被归并的项数(N)	2	2	4	2	2	4	8	24
最多比较(N-1)	1	1	3	1	1	3	7	17
最少比较(N/2)	1	1	2	1	1	1	4	12

对于每一次归并，最大的比较次数比数据项的个数少一。我们把所有归并的比较次数加在一起得到一个总数为 17。

最小的比较次数总是正在被归并的数据项个数的一半，若把所有归并中的比较次数加在一起得到 12。相似的算术运算得到如表 6.4 中“比较次数栏”的结果。排序一个指定数组的实际比较次数依赖于数据是如何排列的，但是这个数字一定会在最大比较次数和最小比较次数之间。

消除递归

有一些算法趋向于用递归的方法，另一些算法则不是。正如前面已经看到的那样，递归的triangle()和 factorial()方法可以用一个简单的循环来实现，那样效率更高。但是，各种分治算法，比如归并排序的递归函数，能工作得非常好。

一个算法作为一个递归的方法通常从概念上很容易理解，但是，在实际的运用中证明递归算法的效率不太高。在这种情况下，把递归的算法转换成非递归的算法是非常有用的。这种转换经常会用到栈。

递归和栈

递归和栈之间有一种紧密的联系。事实上，大部分的编译器都是使用栈来实现递归的。正如我们曾提到过的，当调用一个方法的时候，编译器会把这个方法的所有参数及其返回地址（这个方法返回时控制到达的地方）都压入栈中，然后把控制转移给这个方法。当这个方法返回的时候，这些值退栈。参数消失了，并且控制权重新回到返回地址处。

模拟一个递归方法

在这一节将举例说明任意一个递归的方法怎样转换成基于栈的方法。还记得这一章第一部分中所讲的递归的 triangle()方法吗？下面再次列出这个方法的代码：

```
int triangle(int n)
   {
   if(n==1)
      return 1;
   else
      return( n + triangle(n-1) );
   }
```

把这个算法分解成一个个单独的操作，使每一个操作对应于 switch 中的一条 case 语句。（可以在 C++和一些其他的语言中使用 goto 语句来执行相似的分解，但是 Java 语言不支持 goto 语句。）

switch 语句被封装在一个名为 step()的方法中。每一次调用 step()都会执行 switch 中的一个 case 语句。反复地调用 step()将最终执行程序中所有的代码。

刚才看到的 triangle()方法执行了两种操作。第一种是它执行必要的算术运算来计算三角数组。这包括检查 n 是否为 1，以及给上一次递归调用的结果加 n。但是，triangle()也执行了管理方法本身的一些必要操作，包括控制转移、参数的读取以及返回地址。这些操作不能通过阅读代码来看到；它们是嵌在所有方法内部的。下面粗略地说一下调用一个方法时所发生的事情：

- 当一个方法被调用时，它的参数以及返回地址被压入一个栈中。
- 这个方法可以通过获取栈顶元素的值来访问它的参数。
- 当这个方法要返回的时候，它查看栈以获得返回地址，然后这个地址以及方法的所有参数退栈，并且销毁。

stackTriangle.java 程序包含三个类：Params 类、StackX 类以及 StackTriangleApp 类。Params 类封装了返回地址和这个方法的参数，n；这个类的对象被压入到栈中。除了包含 Params 类的对象之外，StackX 和相关章中的这个类是相似的。StackTriangleApp 类包含四个方法：main()、rectriangle()、

step()和通常用于数字输入的 getInt()方法。

main()程序要求用户输入一个数字，调用 recTriangle()方法来计算对应于 n 的三角数字，并显示输出结果。

recTriangle()方法创建了一个 StackX 对象，并且初始化 codePart 为 1。然后它在反复调用 step()的地方设置一个 while 循环。直到这个方法运行到第六种情况，也就是这个方法的返回点，这时 step()返回 true，这个方法才退出循环。step()方法以一个很大的 switch 语句为基础，在这个 switch 语句中，每个 case 对应于原来 triangle()方法中的一部分代码。清单 6.7 显示了 stackTriangle.java 程序。

清单 6.7 stackTriangle.java 程序

```
// stackTriangle.java
// evaluates triangular numbers, stack replaces recursion
// to run this program: C>java StackTriangleApp
import java.io.*;                  // for I/O
////////////////////////////////////////////////////////////////
class Params     // parameters to save on stack
   {
   public int n;
   public int returnAddress;
   public Params(int nn, int ra)
      {
      n=nn;
      returnAddress=ra;
      }
   } // end class Params
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;          // size of StackX array
   private Params[] stackArray;
   private int top;              // top of stack
//--------------------------------------------------------------
   public StackX(int s)          // constructor
      {
      maxSize = s;               // set array size
      stackArray = new Params[maxSize];  // create array
      top = -1;                  // no items yet
      }
//--------------------------------------------------------------
   public void push(Params p)    // put item on top of stack
      {
      stackArray[++top] = p;     // increment top, insert item
      }
//--------------------------------------------------------------
   public Params pop()           // take item from top of stack
```

```
    {
    return stackArray[top--]; // access item, decrement top
    }
//--------------------------------------------------------------
  public Params peek()         // peek at top of stack
    {
    return stackArray[top];
    }
//--------------------------------------------------------------
  } // end class StackX
////////////////////////////////////////////////////////////////
class StackTriangleApp
  {
  static int theNumber;
  static int theAnswer;
 static StackX theStack;
 static int codePart;
 static Params theseParams;
//--------------------------------------------------------------
  public static void main(String[] args) throws IOException
    {
    System.out.print("Enter a number: ");
    theNumber = getInt();
    recTriangle();
    System.out.println("Triangle="+theAnswer);
    } // end main()
//--------------------------------------------------------------
  public static void recTriangle()
    {
    theStack = new StackX(10000);
    codePart = 1;
    while( step() == false)  // call step() until it's true
      ;                      // null statement
    }
//--------------------------------------------------------------
  public static boolean step()
    {
    switch(codePart)
      {
      case 1:                              // initial call
        theseParams = new Params(theNumber, 6);
        theStack.push(theseParams);
        codePart = 2;
        break;
      case 2:                              // method entry
```

```
            theseParams = theStack.peek();
            if(theseParams.n == 1)              // test
               {
               theAnswer = 1;
               codePart = 5;   // exit
               }
            else
               codePart = 3;   // recursive call
            break;
         case 3:                                // method call
            Params newParams = new Params(theseParams.n - 1, 4);
            theStack.push(newParams);
            codePart = 2;  // go enter method
            break;
         case 4:                                // calculation
            theseParams = theStack.peek();
            theAnswer = theAnswer + theseParams.n;
            codePart = 5;
            break;
         case 5:                                // method exit
            theseParams = theStack.peek();
            codePart = theseParams.returnAddress; // (4 or 6)
            theStack.pop();
            break;
         case 6:                                // return point
            return true;
         } // end switch
      return false;
      } // end triangle
//-------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//-------------------------------------------------------------
   } // end class StackTriangleApp
////////////////////////////////////////////////////////////////
```

这个程序就像在这一章开始部分所列的 triangle.java 程序（程序清单 6.1）所做的一样，计算三角数字。下面是某个例子的输出结果：

```
Enter a number: 100
Triangle=5050
```

图 6.19 显示了每一个 case 的代码部分是如何对应于算法的不同部分的。

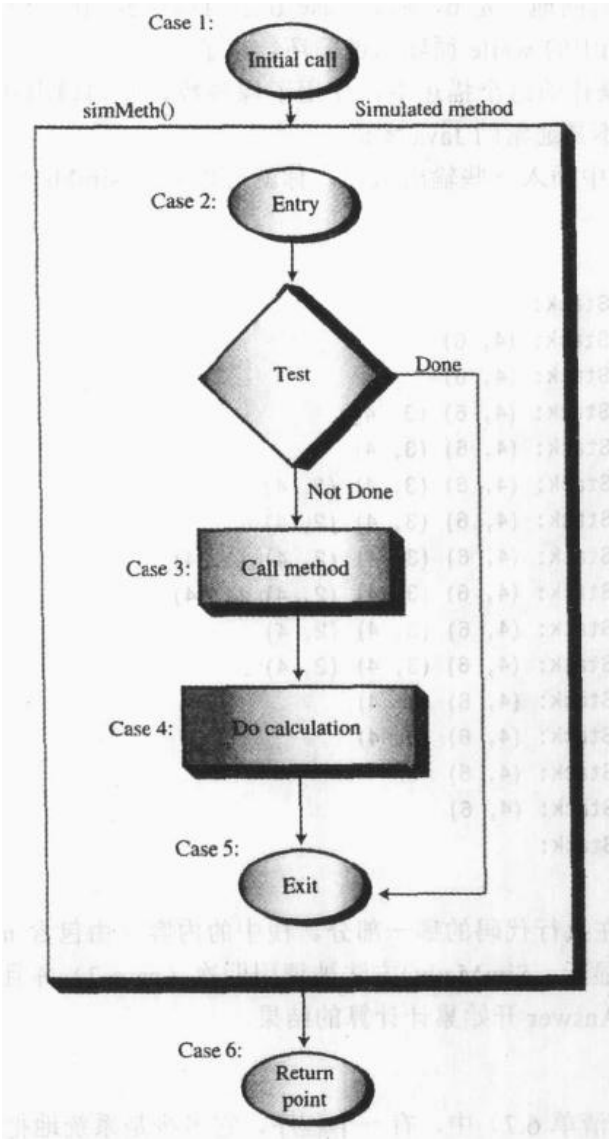


图 6.19 case 和 step()方法

这个程序模拟了一个方法，但是由于它不是一个真正的 Java 方法，所以它在这个程序清单中没有名字。这个模拟的方法称为 simMeth()。对 sinMeth()的初次调用（在 case 1 中）把用户输入的值以及返回值 6 压入栈中，并且移动到 simMeth()的入口处。

在 simMeth()方法的入口（case 2），simMeth()会检查它的参数是否为 1。它通过取得栈顶元素

的值来访问参数。如果参数是1，这就是基值条件，并且控制将转移到simMeth()的出口（case 5）。如果参数不是1，它将递归地调用自身（case 3）。这个递归的调用由参数n-1和返回地址4入栈，以及转移到方法的入口case 2来组成。

从递归调用返回的过程中，simMeth()对调用的返回值加上了它的参数n。最后，这个方法退出（case 5）。当simMeth()退出的时候，最后的Params对象退栈；这个信息不再需要了。

在初始调用中给定的返回地址是6，所以case 6是当程序返回时控制将要转到的地方。这段代码返回true使recTriangle()中的while循环知道循环结束了。

注意，在simMeth()操作的这个描述中，使用了像参数、递归调用和返回地址这样的术语来表示对这些特点的模拟，而不是通常的Java术语。

如果在每个case语句中插入一些输出语句，你就可以看到simMeth()正在做什么，可以有如下这样排列的输出结果：

```
Enter a number: 4
case 1. theAnswer=0  Stack:
case 2. theAnswer=0  Stack: (4, 6)
case 3. theAnswer=0  Stack: (4, 6)
case 2. theAnswer=0  Stack: (4, 6) (3, 4)
case 3. theAnswer=0  Stack: (4, 6) (3, 4)
case 2. theAnswer=0  Stack: (4, 6) (3, 4) (2, 4)
case 3. theAnswer=0  Stack: (4, 6) (3, 4) (2, 4)
case 2. theAnswer=0  Stack: (4, 6) (3, 4) (2, 4) (1, 4)
case 5. theAnswer=1  Stack: (4, 6) (3, 4) (2, 4) (1, 4)
case 4. theAnswer=1  Stack: (4, 6) (3, 4) (2, 4)
case 5. theAnswer=3  Stack: (4, 6) (3, 4) (2, 4)
case 4. theAnswer=3  Stack: (4, 6) (3, 4)
case 5. theAnswer=6  Stack: (4, 6) (3, 4)
case 4. theAnswer=6  Stack: (4, 6)
case 5. theAnswer=10 Stack: (4, 6)
case 6. theAnswer=10 Stack:
Triangle=10
```

case的数字表明了正在执行代码的哪一部分。栈中的内容（由包含n，以及后面的返回地址的Params对象所组成）也将显示。SimMeth()方法被调用四次（case 2）并且返回四次（case 5）。只有当它开始返回的时候，theAnswer开始累计计算的结果。

这证明了什么？

在stackTriangle.java（清单6.7）中，有一个程序，它多少是系统地把一个递归程序转换成了使用栈的程序。这表明对于任意一个递归程序都有可能做出这种转换，实际上这就是一个例证。

如果做一些额外的工作，可以系统地精炼这里给出的代码，并且还可以化简它，甚至可以完全消除switch语句，使代码更有效率。

然而在实践中，人们往往从一开始就重新思考基于栈的算法，而不是从递归的算法转化，这样作更为实用。清单6.8显示了用triangle()方法时发生的事情。

清单 6.8 stackTriangle2.java 程序

```
// stackTriangle2.java
// evaluates triangular numbers, stack replaces recursion
// to run this program: C>java StackTriangle2App
import java.io.*;                  // for I/O
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;        // size of stack array
   private int[] stackArray;
   private int top;            // top of stack
//--------------------------------------------------------------
   public StackX(int s)         // constructor
      {
      maxSize = s;
      stackArray = new int[maxSize];
      top = -1;
      }
//--------------------------------------------------------------
   public void push(int p)     // put item on top of stack
      { stackArray[++top] = p; }
//--------------------------------------------------------------
   public int pop()            // take item from top of stack
      { return stackArray[top--]; }
//--------------------------------------------------------------
   public int peek()           // peek at top of stack
      { return stackArray[top]; }
//--------------------------------------------------------------
   public boolean isEmpty()    // true if stack is empty
      { return (top == -1); }
//--------------------------------------------------------------
   }  // end class StackX
////////////////////////////////////////////////////////////////
class StackTriangle2App
   {
   static int theNumber;
   static int theAnswer;
   static StackX theStack;

   public static void main(String[] args) throws IOException
      {
      System.out.print("Enter a number: ");
      theNumber = getInt();
      stackTriangle();
      System.out.println("Triangle="+theAnswer);
```

```
    } // end main()
//--------------------------------------------------------------
  public static void stackTriangle()
    {
    theStack = new StackX(10000);      // make a stack

    theAnswer = 0;                     // initialize answer

    while(theNumber > 0)               // until n is 1,
      {
      theStack.push(theNumber);        // push value
      --theNumber;                     // decrement value
      }
    while( !theStack.isEmpty() )       // until stack empty,
      {
      int newN = theStack.pop();       // pop value,
      theAnswer += newN;               // add to answer
      }
    }
//--------------------------------------------------------------
  public static String getString() throws IOException
    {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String s = br.readLine();
    return s;
    }
//--------------------------------------------------------------
  public static int getInt() throws IOException
    {
    String s = getString();
    return Integer.parseInt(s);
    }
//--------------------------------------------------------------
  } // end class StackTriangle2App
```

在 stackTriangle()方法中有两个短的 while 循环取代了 stackTriangle.java 程序中的整个的 step()方法。当然，本程序中可以直接看到，能够完全消除栈，而使用一个简单的循环。但是，在更复杂的算法中必须要保留栈。

通常，需要根据试验来判断，在一个特定情况下，递归的方法，或者基于栈的方法，还是一个简单的循环方法，哪一个更有效率（或者更实用）。

一些有趣的递归应用

下面简要地看一些用到递归的其他情况。从这些例子的多样性中可以看到递归能在很多意想不

到的地方出现。我们将考察三个问题：求一个数的乘方，在背包中放入合适的物品，以及登山队选择成员组队的问题。这里只解释概念，而把实现留作练习。

求一个数的乘方

更复杂的便携(式)计算器能够求一个数的任意乘方。它们通常是一个有像 x^y 这样标记的按键，这个符号表示求 x 的 y 次方。如果计算器没有这个键，如何来做这个运算呢？可能会想到需要把 x 乘以 y 次。这也就是说，如果 x 为 2，并且 y 为 8(2^8)，你可以执行这样的算法运算：2*2*2*2*2*2*2*2。但是，对于很大的 y 值，这个方法就显得很冗长。有没有一种更快的方法？

一个解决的方法是重新组织这个问题，只要可能就拿 2 的方次相乘，而不是乘以 2。以 2^8 为例。最终，必须在这个乘法过程中含有八个 2。从 2*2 = 4 来开始。现在已经使用了两个 2，但是还有六个 2 没有用。然而，现在有一个可以起作用的新的数字：4。所以试一试 4*4=16。这使用了四个 2（因为每个 4 是两个 2 相乘的结果）。需要再去使用四个 2，但是现在有 16 可以起作用，并且 16*16 = 256 确实使用了八个 2（因为每个 16 有四个 2）。

所以只用三次乘法就可以求出 2^8 的结果，而没有使用七次乘法。也就是用时间 O(logN)代替了 O(N)。

可以把这个过程写成计算机可以执行的算法吗？这个方案以数学等式 $x^y = (x^2)^{y/2}$ 为基础。在我们所举的例子中，$2^8 = (2^2)^{8/2}$，或者 $2^8 = (2^2)^4$。这是正确的，因为有$(x^a)^b=x^{a*b}$的法则。

但是，现在假定计算机不能求一个数的乘方，所以不能处理$(2^2)^4$。我们来看一下是否能把它转换成一个只含有乘法的表达式。这个技巧是用新变量代替 2^2 作为开始值。

假定 $2^2 = a$。于是 2^8 等于$(2^2)^4$，也就是 a^4。但是，根据原始的等式，a^4 也可以写成是$(a^2)^2$，所以 $2^8 = (a^2)^2$。

再用一个新的变量来代替 a^2，假定 $a^2 = c$，那么$(c)^2$可以被写成是$(c^2)^1$，它也就是等于 2^8。

现在就有了一个用简单的乘法就可以解决的问题：c 乘以 c。

可以在递归方法中使用这个思想——它称为 power()方法——来计算乘方。它的参数是 x 和 y，并且这个方法返回 x^y。我们不需要再担心像 a 和 c 这样的变量，因为在方法每次调用自己的时候 x 和 y 都取得新的值。它的参数是 x*x 和 y/2。对于 x = 2 和 y = 8 的情况，参数的序列和返回值如下：

```
x=2, y=8
x=4, y=4
x=16, y=2
x=256, y=1
Returning 256, x=256, y=1
Returning 256, x=16, y=2
Returning 256, x=4, y=4
Returning 256, x=2, y=8
```

当 y 等于 1 时，就返回。答案是 256，在方法不断返回的过程中没有改变。

已经举了一个例子，在这个例子整个的除法运算序列中 y 都是一个偶数。情况并不会总是这样。那么如何修改算法以使它可以处理 y 是奇数的情况呢？在递归过程中使用整型除法，不用考虑 y 被 2 除后的余数。但是，在返回的过程中，只要 y 是一个奇数，就额外地乘以一个 x。下面是 3^{18} 的序列：

```
x=3, y=18
x=9, y=9
x=81, y=4
x=6561, y=2
x=43046721, y=1
Returning 43046721, x=43046721, y=1
Returning 43046721, x=6561, y=2
Returning 43046721, x=81, y=4
Returning 387420489, x=9, y=9  // y is odd; so multiply by x
Returning 387420489, x=3, y=18
```

背包问题

背包问题是计算机科学里的经典问题。在最简单的形式中，包括试图将不同重量的数据项放到背包中，以使背包最后达到指定的总重量。不需要把所有的选项都放入背包中。

举例来说，假设想要背包精确地承重 20 磅，并且有 5 个可以选择放入的数据项，它们的重量依次为 11 磅、8 磅、7 磅、6 磅和 5 磅。对于选择放入的数据项数量不大时，人类很善于通过观察就可以解决这个问题。于是大概可以计算出只有 8 磅、7 磅和 5 磅的数据项加在一起和为 20 磅。

如果想要计算机来解决这个问题，就需要给计算机更详细的指令。算法如下：

1. 如果在这个过程中的任何时刻，选择的数据项的总和符合目标重量，工作就完成了。

2. 从选择第一个数据项开始。剩余的数据项的加和必须符合背包的目标重量减去第一个数据项的重量；这是一个新的目标重量。

3. 逐个地试每种剩余数据项组合的可能性。但是，注意并不需要去试所有的组合，因为只要数据项的和大于目标重量的时候，就停止添加数据项。

4. 如果没有组合合适的话，放弃第一个数据项，并且从第二个数据项开始再重复一遍整个过程。

5. 继续从第三个数据项开始，如此下去直到你已经试过所有的组合，这时知道没有解决答案。

在刚刚描述的这个例子中，从 11 开始。现在想要剩余的数据项和为 9（20 减去 11）。对于 9，从很小的 8 开始。现在想要剩余的数据项和为 1（9 减去 8）。从 7 开始，但是它大于 1，于是尝试 6，然后试 5，它们都太大了。现在已经试过了所有的数据项，所以知道包含 8 的任何组合和都不可能为 9。接着尝试 7，于是现在开始找的目标为 2（9 减去 7）。用同样的方法继续，总结如下：

```
Items: 11, 8, 7, 6, 5
=======================================
11          // Target = 20, 11 is too small
11, 8       // Target = 9, 8 is too small
11, 8, 7    // Target = 1, 7 is too big
11, 8, 6    // Target = 1, 6 is too big
11, 8, 5    // Target = 1, 5 is too big. No more items
11, 7       // Target = 9, 7 is too small
11, 7, 6    // Target = 2, 6 is too big
11, 7, 5    // Target = 2, 5 is too big. No more items
11, 6       // Target = 9, 6 is too small
11, 6, 5    // Target = 3, 5 is too big. No more items
```

```
11, 5      // Target = 9, 5 is too small. No more items
8,         // Target = 20, 8 is too small
8, 7       // Target = 12, 7 is too small
8, 7, 6    // Target = 5, 6 is too big
8, 7, 5    // Target = 5, 5 is just right. Success!
```

现在可能已经认识到了，一个递归的程序能选择第一个数据项，并且，如果这个数据项比目标重量小，程序可以用一个新的目标值来调用自身以继续试探剩余数据项的总和。

组合：选择一支队

在数学中，组合是对事物的一种选择，而不考虑它们的顺序。例如，有 5 个登山队员，名字为 A、B、C、D 和 E。想要从这群登山队员中选择三个队员，组成一支登山队去攀登陡峭的冰雪覆盖的 Anaconda 峰。但是，由于担心登山队的成员如何相处，所以决定列出所有组队的可能性：也就是，挑出三个队员的所有可能组合。但是如果有一个计算机程序能打印出所有的组合就太好了。这样的程序能显示出 10 种可能的组合：

ABC, ABD, ABE, ACD, ACE, ADE, BCD, BCE, BDE, CDE

如何来写一个这样的程序呢？它生成了一个简洁的递归的解决办法。它包括把这些组合分成两部分：由 A 开始的组合和不是由 A 开始的组合。假设把从 5 个人中选出 3 个人的组合简写为(5,3)。规定 n 是这群人的大小，并且 k 是组队的大小。那么根据法则可以这么说：

(n, k) = (n-1, k-1) + (n-1, k)

对于从 5 个人中选择 3 个组队的例子，有

(5, 3) = (4, 2) + (4, 3)

现在已经把一个大问题转化成两个较小的问题了。从 4 个人的人群中做两次选择，而不是从一个 5 个人的人群中选择一次：首先，所有的选择是从 4 个人的人群中选择 2 个人，然后所有的选择是从一个 4 个人的人群中选择 3 个人。

从 4 个人的人群中选择 2 个人有 6 种选择方法。在(4, 2)项中——称为左项——表示有六种组合是

BC, BD, BE, CD, CE, DE

A 是一个隐在的成员，所以在这些组合前面要加上 A 组成三个人的队：

ABC, ABD, ABE, ACD, ACE, ADE

从 4 个人的人群中选择 3 个人有四种选择方法。(4, 3)项——右项——四种组合是

BCD, BCE, BDE, CDE

把从右项得到的 4 种组合和从左项得到的 6 种组合加在一起，就为(5, 3)得到了 10 种组合。

可以对每一个 4 个人的人群应用同样的分解过程。例如，(4, 2)等于(3, 1)加上(3, 2)。正如看到的那样，这是很自然的应用递归的地方。

可以把这个问题看成是一棵树，在第一行是（5, 3），在第二行是（4, 3）和（4,2），依此类推，这个树中的节点对应于递归方法的调用。图 6.20 显示了例子（5, 3）的样子。

基值条件是指没有意义的组合：某个数字是 0 以及队员数大于人群数的情况。组合（1, 1）是合法的，但是继续分解它就没有必要了。在这个图中，虚线表示了基值条件；这时需要返回而不是

继续分解。

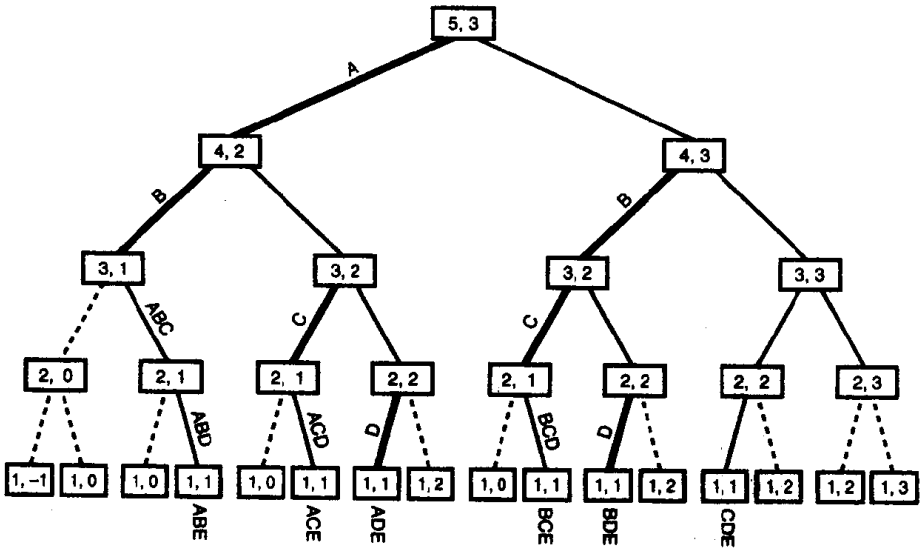


图 6.20 从 5 个人的人群中选择一支三个人的队伍

递归的深度和人群的人数是对应的：在首层的节点代表了成员 A，在下一层的两个节点代表了成员 B，依此类推。如果有 5 个成员，那么就会有 5 层。

当沿着树往下走时，需要记住访问过的节点序列。下面讲一下如何做到这些事：一旦调用了一次法则中的左项表达式，当离开该节点时，通过把它代表的字母加到序列中，由此来记录访问过的节点。这些左调用以及加到序列中的字母在图中用加粗的黑线显示。当返回的时候需要把序列带回。

为了记录所有的组合，可以随着遍历来显示它们。在作左调用时不显示任何东西。但是，当作右调用时，要检查这个序列；如果在一个合法的节点上，并且增加一个人就完成了组队，那么把这个节点加到序列上并且显示这次组建好的队。

小 结

- 一个递归的方法每次用不同的参数值反复调用自身。
- 某种参数值使递归的方法返回，而不再调用自身。这称为基值情况。
- 当递归方法返回时，递归过程通过逐渐完成各层方法实例的未执行部分，而从最内层返回到最外层的原始调用处。
- 三角数字是它本身以及所有比它小的数字的和。（在这一章中数字是指整数。）例如，4 的三角数字是 10，因为 4 + 3 + 2 +1 = 10。
- 一个数的阶乘是它自身和所有比它小的数的乘积。例如，4 的阶乘是 4*3*2*1 = 24。
- 三角数字和阶乘都可以通过递归的方法或者简单的循环方法来实现。
- 一个单词的全排列（它的 n 个字母的所有可能排列）可以通过反复地轮换它的字母以及全排列它最右边的 n-1 个字母来递归得到。
- 二分查找可以通过检查查找关键字在有序序列的哪一半，然后在这一半做相同的事情，这

些都可以用递归实现。

- 汉诺塔的问题包含三个塔座和任意数量的盘子。
- 汉诺塔难题可以递归来解决：把除了最低端盘子外的所有盘子形成的子树移动到一个中介塔座上，然后把最低端的盘子移到目标塔座上，最终把那个子树移动到目标塔座上。
- 归并两个有序数组意思是创建第三个数组，这个数组按顺序存储从这两个有序数组中取到的所有数据项。
- 在归并排序中，一个大数组的单个数据项的子数组归并为两个数据项的子数组，然后两个数据项的子数组归并为 4 个数据项的子数组，如此下去直到所有的数组数据项有序。
- 归并排序需要 O(N*logN)时间。
- 归并排序需要一个大小等于原来数组的工作空间。
- 对于三角数字、阶乘、单词字母全排列以及二分查找，它们的递归的方法只包含一次对自身的调用。（在二分查找的代码中显示有两次，但是在任何给定代码的运行中只有一次自身的调用执行了。）
- 对于汉诺塔和归并排序问题，它们的递归的方法包括两次递归调用。
- 任何可以用递归完成的操作都可以用一个栈来实现。
- 递归的方法可能效率低。如果是这样的话，有时可以用一个简单循环或者是一个基于栈的方法来替代它。

问 题

下列问题作为读者的自测题。答案可见附录 C。

1. 如果用户在 triangle.java 程序（清单 6.1）中输入 10，任何一次 triangle()方法“复制”的最大次数是多少（实际上只是它的参数的复制）？

2. 问题 1 中所提到的参数的复制是存储在什么地方的？

   a. triangle()方法中的一个变量

   b. TriangleApp 类的一个字段

   c. getString()方法中的一个变量

   d. 在栈中

3. 在问题 1 假定用户中输入 10。当 triangle()方法第一次返回值不是 1 时，n 的值是多少？

4. 假定还是问题 1 中的情况。当 triangle()方法将要返回到 main()方法时，n 的值是多少？

5. 判断题：在 triangle()方法中，返回值存储在栈中。

6. 在 anagram.java 程序（清单 6.2）中，在某个递归深度中 doAnagram()方法的参数是字符串“led”。当这个方法新调用一次自身方法时，这个新的方法的参数会是什么？

7. 现在已经知道递归可以取代循环，如在基于循环的 orderedArray.java 程序（清单 2.4）和递归的 binarySearch.java 程序（清单 6.3）中。如下描述中哪个是不正确的？

   a. 两个程序都反复地把排列分成两半。

   b. 如果没有找到关键字，循环的程序返回因为超出排列的界限，但是递归的程序可以执行，因为它达到了递归的底层。

c. 如果找到了关键字，循环的程序从整个方法中返回，而递归的程序只返回一层递归。

d. 在递归的程序中查找的范围必须用参数来指定，而在循序的程序中不需要。

8. 在 binarySearch.java 程序（清单 6.3）的 recFind()方法中，用什么取代了在非递归程序中的循环？

a. recFind()方法

b. recFind()方法的参数

c. recFind()方法的递归调用

d. main()调用 recFind()方法

9. binarySearch.java 程序是一个用_____方法解决问题的例子。

10. 当在 redFind()方法中反复地递归调用时什么值变得越来越小了？

11. 在 towers.java 程序（清单 6.4）中反复地递归调用时什么值变得越来越小了？

12. 在 towers.java 程序中的算法包括

a. 存储数据的“树”。

b. 偷偷的将小盘子放在大盘子下面。

c. 转变源塔座和目标塔座。

d. 移动一个小盘子，然后移动一堆大盘子。

13. 关于在 merge.java 程序（清单 6.5）中的 merge()方法哪个描述是不正确的？

a. 这个算法可以处理不同大小的数组。

b. 必须要查找目标数组来找到下一个数据项放在哪。

c. 没有递归。

d. 它不断的操作最小的数据项，而不考虑它在哪个数组中。

14. 归并排序的缺点是

a. 没有递归。

b. 它使用更多的存储空间。

c. 尽管比插入算法快，但是它比快速排序慢得多。

d. 实现起来很复杂。

15. 除了用循环外，_____常被用来取代递归。

实　验

完成这些实验可以帮助深入理解本章的主题。不需要编程实现。

1. 在 triangle.java 程序（清单 6.1）中，删除基值条件的代码（if(n ==1) return 1 和 else）。然后运行程序看看会发生什么？

2. 在手动操作的模式下使用 Towers 专题 applet 来解决有七个或者更多盘子的汉诺塔难题。

3. 重写 mergeSort.java 程序（清单 6.6）中的 main()部分，你可以给一个数组添加一个有成百上千个随机数。运行程序来给这些数排序，并且和第 3 章“简单排序”中的排序算法的速度进行比较。

编程作业

编程作业有助于巩固对本章内容的理解，并展示如何应用本章的概念。（在“简介”中提到过，资深教师可以从出版者的网站上得到编程作业的完整答案。）

6.1 假设买了一个价格便宜的掌上电脑，但是发现它内置的芯片不能做乘法，只可以做加法。摆脱这种窘境需要自己编写程序，写的是一个递归的方法 mult()，它的参数是 x 和 y 返回值是 x 乘 y。它可以通过 x 自身相加 y 次来执行 x 乘以 y 的乘法操作。写出这样一个程序后，用 main()调用它。当这个方法调用自身或者当它返回的时候，加法是否起作用了？

6.2 在第 8 章“二叉树”中，我们将看到二叉树，在它的每一个节点中确实有两个子树。如果使用字符在屏幕上画一棵二叉树，可以在顶层有一个节点，在下一层有两个，然后是 4、8、16 个，依此类推。下面是一棵最底层有 16 个字符的树的样子：

```
--------X--------
----X-------X---
--X---X---X---X-
-X-X-X-X-X-X-X-X
XXXXXXXXXXXXXXXX
```

（注意最下面的一行，应当向右移动半个字符宽，但是在字符模式中做不到。）可以使用递归的 makeBranches()方法来画这个树，这个方法的参数为 left 和 right，它们是水平范围的端点。当第一次进入这个程序的时候，left 等于 0，而且 right 是所有行中字符的数目（包括短线）减一。在这行范围的中间画一个 X。然后这个方法调用自己两次：一次为左一半的范围，一次为右一半的范围。当这个范围大小的时候返回。你可能想要把所有的短线和 X 都放到一个数组中，并且一次性显示数组，或许可以使用 display()方法。编写 main()调用 makeBranches()和 display()来画这棵树。允许 main()决定显示的每一行的宽度（32、64 或者其他的任何值）。确保存放显示字符的数组不会比所需要的空间大。行数（图中为五）和每一行的宽度有什么关系？

6.3 应用递归的算法来实现求一个数的乘方，如在这一章接近结尾处的“求一个数的乘方”部分所讲。写递归的 power()方法以及一个 main()来测试它。

6.4 写一个能解决背包问题的程序，任意给定背包的容量以及一系列物品的重量，设把这些重量值存在一个数组中。提示：递归方法 knapsack()的参数是目标重量和剩下物品开始位置的数组下标。

6.5 应用一个递归的方法来显示由一群人组队的所有可能方案（由 n 个每次挑 k 个）。编写递归的 showTeams()方法和一个 main()方法来提示用户输入人群的人数以及组队的人数，以此来作为 showTeams()的参数，然后显示所有的组合。

第7章 高级排序

本章重点
- 希尔排序
- 划分
- 快速排序
- 基数排序

在第 3 章中已经讨论过简单排序。那一章中讲述的排序算法——冒泡排序、选择排序和插入排序，都是一些容易实现的，但速度比较慢的算法。在第 6 章“递归”中讲述了归并排序。归并排序运行速度比简单排序要快，但是它需要的空间是原始数组空间的两倍；通常这是一个严重的缺点。

本章包含了两个高级的排序算法：希尔排序和快速排序。这两种排序算法都比简单排序算法快得多：希尔排序大约需要 $O(N*(\log N)^2)$时间，快速排序需要 $O(N*\log N)$时间。这两种排序算法都和归并排序不同，不需要大量的辅助存储空间。希尔排序几乎和归并排序一样容易实现，而快速排序是所有通用排序算法中最快的一种排序算法。本章的最后还简要地介绍了基数排序，它是一种不常用但很有趣的排序算法。

首先将研究希尔排序。然后是基于划分思想的快速排序，所以在讲解快速排序前，先用单独一节讨论一下划分思想。

希尔排序

希尔排序因计算机科学家 Donald L. Shell 而得名，他在 1959 年发现了希尔排序算法。希尔排序基于插入排序，但是增加了一个新的特性，大大地提高了插入排序的执行效率。

依靠这个特别的实现机制，希尔排序对于多达几千个数据项的，中等大小规模的数组排序表现良好。希尔排序不像快速排序和其他时间复杂度为 $O(N*\log N)$的排序算法那么快，因此对非常大的文件排序，它不是最优选择。但是，希尔排序比选择排序和插入排序这种时间复杂度为 $O(N^2)$的排序算法还是要快得多，并且它非常容易实现：希尔排序算法的代码既短又简单。

它在最坏情况下的执行效率和在平均情况下的执行效率相比没有差很多。（在本章的后面将谈到，除非采取了预防措施，否则快速排序在最坏情况下的执行效率会非常差。）一些专家（Sedgwick 参见附录 B，）提倡差不多任何排序工作在开始时都可以使用希尔排序算法，若在实际中证明它不够快，再改换成诸如快速排序这样更高级的排序算法。

插入排序：复制的次数太多

由于希尔排序是基于插入排序的，所以需要回顾一下第 3 章中的“插入排序”一节。回想一下在插入排序执行的一半的时候，标记符左边这部分数据项都是排过序的（这些数据项之间是有序的），而标记右边的数据项则没有排过序。这个算法取出标记符所指的数据项，把它存储在一个临时变量里。接着，从刚刚被移除的数据项的左边第一个单元开始，每次把有序的数据项向右移动一

个单元，直到存储在临时变量里的数据项能够有序回插。

下面是插入排序带来的问题。假设一个很小的数据项在很靠近右端的位置上，这里本来应该是值比较大的数据项所在的位置。把这个小数据项移动到在左边的正确位置上，所有的中间数据项（这个数据项原来所在的位置和它应该移动到的位置之间的数据项）都必须向右移动一位。这个步骤对每一个数据项都执行了将近 N 次的复制。虽不是所有数据项都必须移动 N 个位置，但是数据项平均移动了 N/2 个位置，这就执行了 N 次 N/2 个移位，总共是 $N^2/2$ 次复制。因此，插入排序的执行效率是 $O(N^2)$。

如果能以某种方式不必一个一个地移动所有中间的数据项，就能把较小的数据项移动到左边，那么这个算法的执行效率就会有很大的改进。

n-增量排序

希尔排序通过加入插入排序中元素之间的间隔，并在这些有间隔的元素中进行插入排序，从而使数据项能大跨度地移动。当这些数据项排过一趟序后，希尔排序算法减小数据项的间隔再进行排序，依此进行下去。进行这些排序时数据项之间的间隔被称为增量，并且习惯上用字母 h 来表示。图 7.1 显示了增量为 4 时对包含 10 个数据项的数组进行排序的第一个步骤的情况。在 0、4 和 8 号位置上的数据项已经有序了。

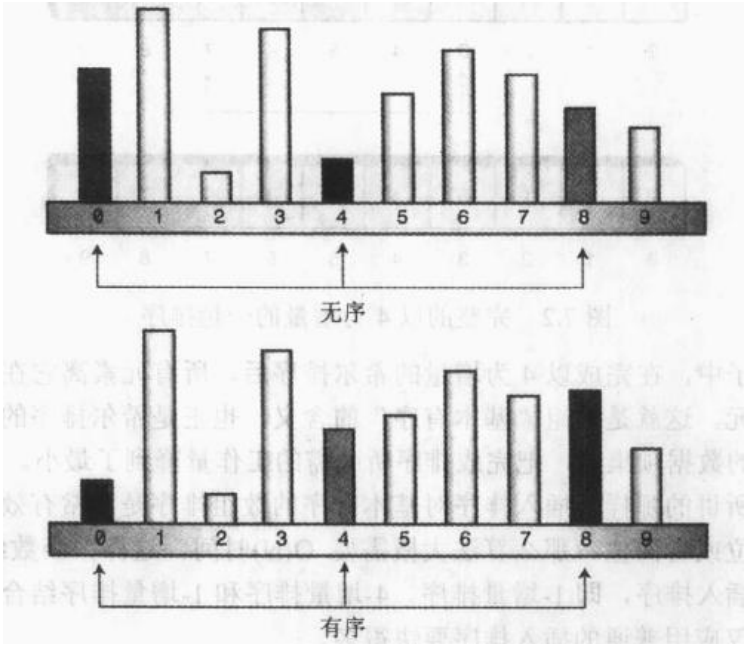


图 7.1　4-增量排序 0、4 和 8 号数据项

当对 0、4 和 8 号数据项完成排序之后，算法向右移一步，对 1、5 和 9 号数据项进行排序。这个排序过程持续进行，直到所有的数据项都已经完成了 4-增量排序，也就是说所有间隔为 4 的数据项之间都已经排列有序。这个过程如图 7.2 所示（使用更为简洁形象的图例表示）。

在完成以 4 为增量的希尔排序之后，数组可以看成是由 4 个子数组组成：（0，4，8），（1，5，9），（2，6）和（3，7），这四个子数组内分别是完全有序的。这些子数组相互交错着排列，然而彼此独立。

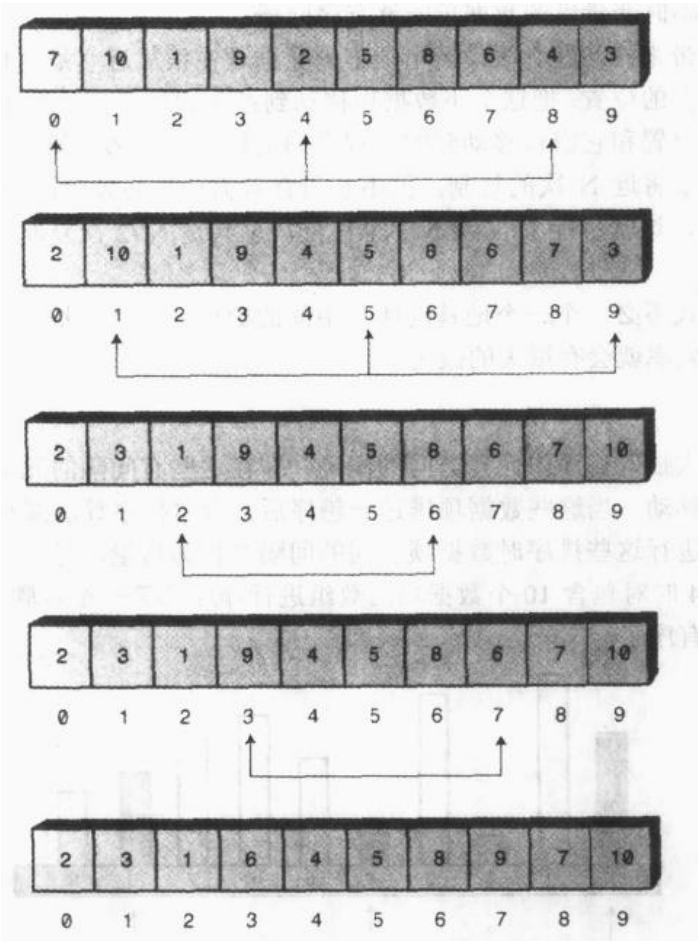


图 7.2 完整的以 4 为增量的一趟排序

注意，在这个例子中，在完成以 4 为增量的希尔排序后，所有元素离它在最终有序序列中的位置相差都不到两个单元。这就是数组“基本有序”的含义，也正是希尔排序的奥秘所在。通过创建这种交错的内部有序的数据项集合，把完成排序所必需的工作量降到了最小。

正如在第 3 章中所讲的那样，插入排序对基本有序的数组排序是非常有效的。如果插入排序只需要把数据项移动一位或者两位，那么算法大概需要 O(N)时间。这样，当数组完成 4-增量排序之后，可以进行普通的插入排序，即 1-增量排序。4-增量排序和 1-增量排序结合起来应用，比前面不执行 4-增量排序而仅仅应用普通的插入排序要快得多。

减小间隔

上面已经演示了以 4 为初始间隔对包含 10 个数据项的数组进行排序的情况。对于更大的数组，开始的间隔也应该更大。然后间隔不断减小，直到间隔变成 1。

举例来说，含有 1000 个数据项的数组可能先以 364 为增量，然后以 121 为增量，以 40 为增量，以 13 为增量，以 4 为增量，最后以 1 为增量进行希尔排序。用来形成间隔的数列（在本例中为 364，121，40，13，4，1）被称为间隔序列。这里所表示的间隔序列由 Knuth（见附录 B）提出，此序列是很常用的。数列以逆向的形式从 1 开始，通过递归表达式

h = 3*h + 1

来产生，初始值为 1。表 7.1 的前两栏显示了这个公式产生的序列。

表 7.1 Knuth 间隔序列

h	3*h+1	(h-1)/3
1	4	
4	13	1
13	40	4
40	121	13
121	364	40
364	1093	121
1093	3280	364

还有一些其他的方法也能产生间隔序列；后面会讲到这个问题。首先，来研究使用 Knuth 序列进行希尔排序的情况。

在排序算法中，首先在一个短小的循环中使用序列的生成公式来计算出最初的间隔。h 值最初被赋为 1，然后应用公式 h = 3*h + 1 生成序列 1，4，13，40，121，364，等等。当间隔大于数组大小的时候这个过程停止。对于一个含有 1000 个数据项的数组，序列的第七个数字，1093 就太大了。因此，使用序列的第六个数字作为最大的数字来开始这个排序过程，作 364-增量排序。然后，每完成一次排序例程的外部循环，用前面提供的此公式的倒推式来减小间隔：

h = (h-1)/3

它在表 7.1 的第三栏中显示。这个倒推的公式生成逆置的序列 364，121，40，13，4，1。从 364 开始，以每一个数字作为增量进行排序。当数组用 1-增量排序后，算法结束。

Shellsort 专题 applet

使用 Shellsort 专题 applet 来观察希尔排序算法是如何工作的。图 7.3 显示了 applet 对所有的竖条进行完 4-增量排序之后，开始 1-增量排序之前的情况。

单步执行这个算法，可以看到前面讨论时给出的解释做了稍微的简化。对于以 4-增量的序列实际上不是按（0，4，8），（1，5，9），（2，6），和（3，7）分组地进行，而是每一组三个数据项中的前两个数据项先排。首先排第一组的前两个数据项，然后排第二组的前两个数据项，照此下去。当所有组的前两个数据项都有序之后，算法返回，再对三个数据项的组进行排序。实际上的序列是（0，4），（1，5），（2，6），（3，7），（0，4，8），（1，5，9）。

首先对完整的子数组进行 4-增量排序似乎更为直观—（0，4），（0，4，8），（1，5），（1，5，9），（2，6），

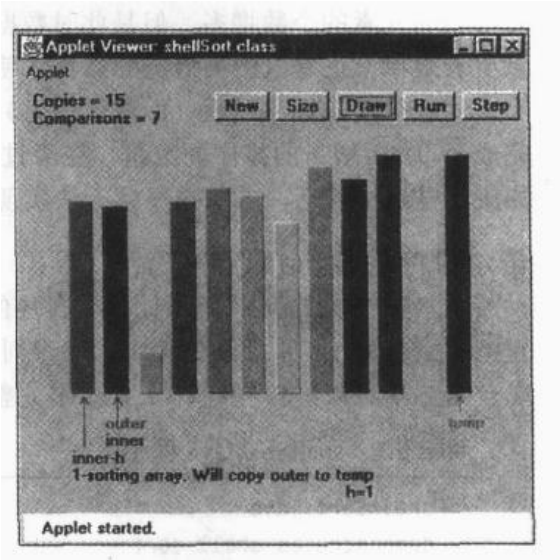


图 7.3 Shellsort 专题 applet

（3，7）—但是算法使用第一种方法处理数组下标更为有效。

希尔排序只对 10 个数据项进行排序效率不是很高，它所执行的交换和比较操作的次数和插入排序基本相当。但是，对 100 个竖条排序，希尔排序的改进就明显地表现出来了。

以 100 个逆序排列的竖条来运行这个专题 applet 是非常有意义的。（记住，正如在第 3 章中一样，第一次点击 New 按钮来创建任意排列的竖条，而第二次点击则创建了一个逆序排序的竖条序列。）图 7.4 显示了在第一趟排序之后竖条的情况，这时数组完成了以 40-增量的一趟排序。图 7.5 显示了下一趟排序后的情况，数组以 13-增量进行排序。h 每赋一个新的值，数组都更为有序。

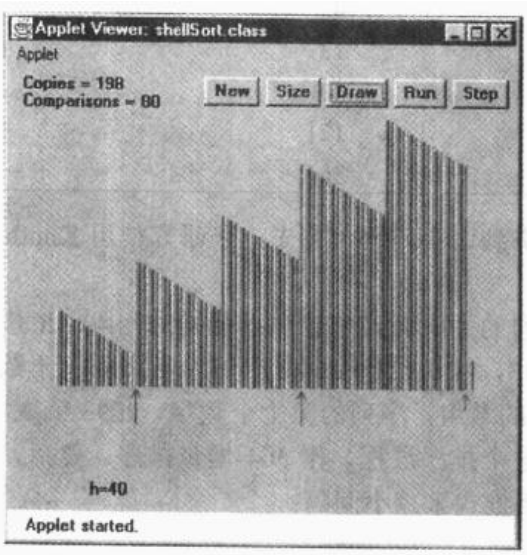


图 7.4 完成增量为 40 的排序之后

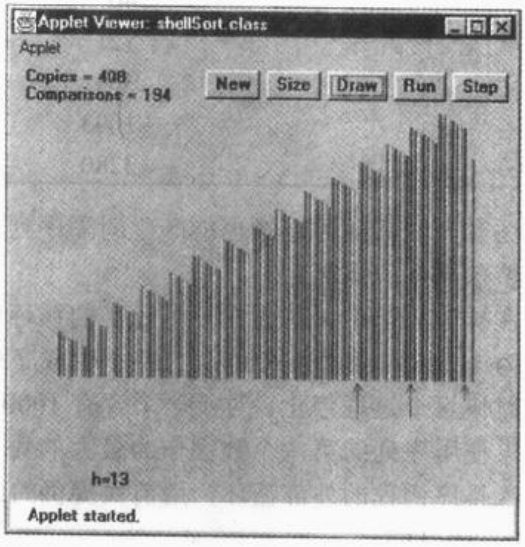


图 7.5 完成 13-增量排序之后

希尔排序比插入排序快很多，它是基于什么原因呢？当 h 值大的时候，数据项每一趟排序需要移动元素的个数很少，但数据项移动的距离很长。这是非常有效率的。当 h 减小时，每一趟排序需要移动的元素的个数增多，但是此时数据项已经接近于它们排序后最终的位置，这对于插入排序可以更有效率。正是这两种情况的结合才使希尔排序效率那么高。

注意后期的排序过程（h 取值小时）不撤销前期排序（h 取值大时）所做的工作。例如，已经完成了以 40-增量的排序的数组，在经过以 13-增量的排序后仍然保持了以 40-增量的排序的结果。如果不是这样的话，希尔排序就无法实现排序的目的。

希尔排序的 Java 代码

希尔排序的 Java 代码不比插入排序的代码复杂多少。只是在插入排序的开始部分，在合适的位置把 h 赋值为 1，并且添加生成间隔序列的公式。在 ArraySh 类中加入 shellSort()方法，其中数组类取自第 2 章“数组”。清单 7.1 显示了完整的 shellSort.java 程序。

清单 7.1 shellSort.java 程序

```
// shellSort.java
// demonstrates shell sort
// to run this program: C>java ShellSortApp
```

```
//--------------------------------------------------------------
class ArraySh
   {
   private long[] theArray;          // ref to array theArray
   private int nElems;               // number of data items
//--------------------------------------------------------------
    public ArraySh(int max)          // constructor
       {
       theArray = new long[max];     // create the array
       nElems = 0;                   // no items yet
       }
//--------------------------------------------------------------
    public void insert(long value)   // put element into array
       {
       theArray[nElems] = value;     // insert it
       nElems++;                     // increment size
       }
//--------------------------------------------------------------
    public void display()            // displays array contents
       {
       System.out.print("A=");
       for(int j=0; j<nElems; j++)   // for each element,
          System.out.print(theArray[j] + " ");  // display it
       System.out.println("");
       }
//--------------------------------------------------------------
    public void shellSort()
       {
       int inner, outer;
       long temp;

       int h = 1;                    // find initial value of h
       while(h <= nElems/3)
          h = h*3 + 1;               // (1, 4, 13, 40, 121, ...)

       while(h>0)                    // decreasing h, until h=1
          {
                                     // h-sort the file
          for(outer=h; outer<nElems; outer++)
             {
             temp = theArray[outer];
             inner = outer;
                                     // one subpass (eg 0, 4, 8)
             while(inner > h-1 && theArray[inner-h] >=  temp)
                {
```

```
            theArray[inner] = theArray[inner-h];
            inner -= h;
            }
         theArray[inner] = temp;
         } // end for
      h = (h-1) / 3;              // decrease h
      } // end while(h>0)
   } // end shellSort()
//--------------------------------------------------------------
} // end class ArraySh
////////////////////////////////////////////////////////////////
class ShellSortApp
   {
   public static void main(String[] args)
      {
      int maxSize = 10;            // array size
      ArraySh arr;
      arr = new ArraySh(maxSize);  // create the array

      for(int j=0; j<maxSize; j++) // fill array with
         {                         // random numbers
         long n = (int)(java.lang.Math.random()*99);
         arr.insert(n);
         }
      arr.display();               // display unsorted array
      arr.shellSort();             // shell sort the array
      arr.display();               // display sorted array
      } // end main()
   } // end class ShellSortApp
```

在 main()方法创建了一个 ArraySh 类型的对象，它的容量为 10，用随机数值给它们赋值，显示数组，对数组进行希尔排序，然后再次显示数组。下面是一个例子的输出结果：

```
A=20 89 6 42 55 59 41 69 75 66
A=6 20 41 42 55 59 66 69 75 89
```

可以使 maxSize 取更大的值，但是也不要太大。对 10000 个数据项需要将近 1 分钟的时间来完成排序。

尽管希尔排序的算法只需要几行代码来实现，但是跟踪这个算法也不是很简单的。要看这个算法操作的细节，用专题 applet 来执行对 10 个数据项的排序，同时将 applet 显示的信息与 shellSort() 方法的代码进行比较。

其他间隔序列

选择间隔序列可以称得上是一种魔法。至此书中只讨论了用公式 h = h*3+1 生成间隔序列，但是应用其他间隔序列也取得了不同程度的成功。只有一个绝对的条件，就是逐渐减小的间隔最后一

定要等于 1，因此最后一趟排序是一次普通的插入排序。

在希尔的原稿中，他建议初始的间距为 N/2，简单地把每一趟排序分成了两半。因此，对于 N = 100 的数组逐渐减小的间隔序列为 50，25，12，6，3，1。这个方法的好处是不需要在开始排序前为找到初始的间隔而计算序列；而只需要用 2 整除 N。但是，这被证明并不是最好的数列。尽管对于大多数的数据来说这个方法还是比插入排序效果好，但是这种方法有时会使运行时间降到 $O(N^2)$，这并不比插入排序的效率更高。

这个方法的一个变形是用 2.2 而非 2 来整除每一个间隔。对于 n = 100 的数组来说，会产生序列 45，20，9，4，1。这比用 2 整除显著改善了效果，因为这样避免了某些导致时间复杂度为 $O(N^2)$ 的最坏情况的发生。不论 N 为何值，都需要一些额外的代码来保证序列的最后取值为 1。这产生了和清单中所列的 Knuth 序列差不多的结果。

递减数列（来自 Flamig；参见附录 B）的另一个可能是

```
if(h < 5)
   h = 1;
else
   h = (5*h-1) / 11;
```

间隔序列中的数字互质通常被认为很重要；也就是说，除了 1 之外它们没有公约数。这个约束条件使每一趟排序更有可能保持前一趟排序已排好的效果。希尔最初以 N/2 为间隔的低效性就是归咎于它没有遵守这个准则。

或许还可以设计出像如上讲述的间隔序列一样好的（甚至更好的）间隔序列。但是不管这个间隔序列是什么，都应该能够快速地计算，而不会降低算法的执行速度。

希尔排序的效率

迄今为止，除了在一些特殊的情况下，还没有人能够从理论上分析希尔排序的效率。有各种各样基于试验的评估，估计它的时间级从 $O(N^{3/2})$到 $O(N^{7/6})$。

表 7.2 对比速度较慢的插入排序和速度较快的快速排序，列出了希尔排序一些估计的 O()值。它显示了对应于不同 N 值的理论时间。注意 $N^{x/y}$ 的意思是 N 的 x 方的 y 次方根。因此，如果 N 等于 100，$N^{3/2}$ 就是 100^3 的平方根，结果是 1000。另外，$(\log N)^2$ 意思是 N 对数的平方。它通常写作 $\log^2 N$，但这很容易和 $\log_2 N$（以 2 为底的 N 的对数）混淆。

表 7.2 希尔排序运行时间的估计

O()值	排序类型	10 项	100 项	1000 项	10000 项
N^2	插入等	100	10000	1000000	100000000
$N^{3/2}$	希尔排序	32	1000	32000	1000000
$N*(\log N)^2$	希尔排序	10	400	9000	160000
$N^{5/4}$	希尔排序	18	316	5600	100000
$N^{7/6}$	希尔排序	14	215	3200	46000
N*logN	快速排序等	10	200	3000	40000

对大多数数据来说，评估值越高，如 $N^{3/2}$，很可能就越真实。

划 分

划分是后面将要讨论的快速排序的根本机制，但是划分本身也是一个有用的操作，因此把它作为单独的一节在这里讲解。

划分数据就是把数据分为两组，使所有关键字大于特定值的数据项在一组，使所有关键字小于特定值的数据项在另一组。

很容易想像划分数据的情况。比如可以将职员记录分为两组：家住距离办公地点 15 英里以内的雇员和住在 15 英里以外的雇员。或者学校管理者想要把学生分成年级平均成绩高于 3.5 和低于 3.5 的两组，以此来判定哪些学生应该在系主任掌握的名单里。

Partition 专题 applet

Partition 专题 applet 演示了划分的过程。图 7.6 显示了划分前的 12 个竖条，图 7.7 显示了经过划分后的这 12 个竖条。

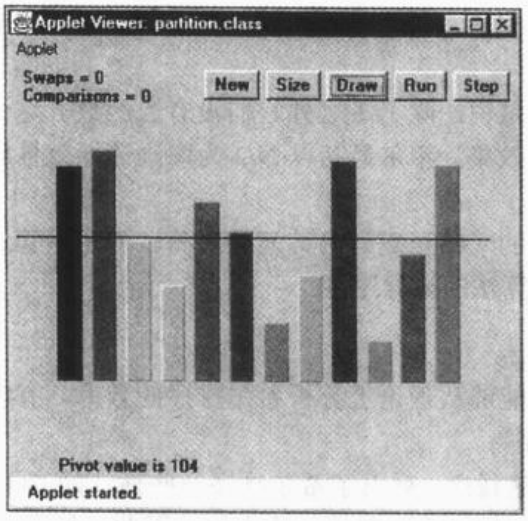


图 7.6 划分之前的 12 个竖条

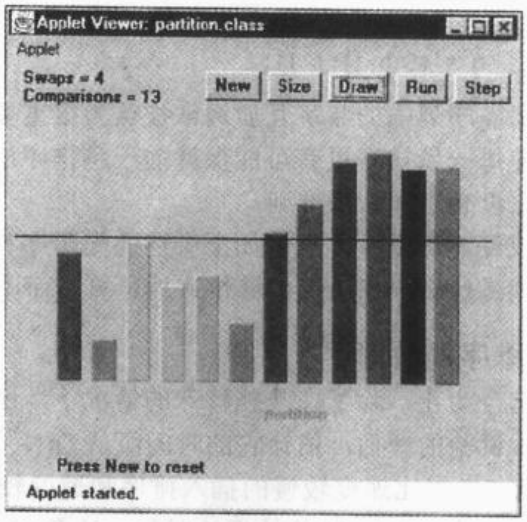


图 7.7 划分之后的 12 个竖条

水平线代表枢纽，这个值用来判定数据项应该在哪一组。关键字的值小于枢纽的数据项放在数组的左边部分，关键字的值大于（或者等于）枢纽的数据项则放在数组的右边部分。（在快速排序一节中，可以看到枢纽被定为一个具体数据项的关键字值，这个实际的数据项被称为枢纽或支点。对于这一节来说，它只是一个数字。）

标注为 partition 的箭头指示右侧（较大）子数组的最左边的数据项。这个值由划分方法来返回，因此其他的方法若需要知道在哪里进行划分，便可以使用这个值。

为了更生动地演示划分的过程，把 Partition 专题 applet 设置为 100 个竖条，然后点击 Run 按钮。leftScan 和 rightScan 指针朝着对方飞快地运动，移动的过程当中交换竖条。当 leftScan 和 rightScan 相遇时，本次划分结束。

根据划分的原因，可以选择任何值作为枢纽（例如选择年级平均成绩 3.5）。为了多样性，专题 applet 每点击一次 New 或者 Size 按钮，都会选择任意的数作为枢纽（以水平的黑线表示），但是这个值绝对不要和竖条的平均高度相差太远。

在完成划分之后，数据还不能称为有序；这只是把数据简单地分成了两组。但是，数据还是比没有划分之前要更接近有序了。在下一节中将会看到，要使它完全有序并不会很麻烦。

注意划分是不稳定的。这也就是说，每一组中的数据项并不是按照它原来的顺序排列的。事实上，划分往往会颠倒组中一些数据的顺序。

partition.java 程序

划分过程是如何执行的呢？下面来看一看例子的代码。清单 7.2 显示了 partition.java 程序，它包括划分数组的 partitionIt()方法。

清单 7.2　partition.java 程序

```
// partition.java
// demonstrates partitioning an array
// to run this program: C>java PartitionApp
////////////////////////////////////////////////////////////////
class ArrayPar
   {
   private long[] theArray;          // ref to array theArray
   private int nElems;               // number of data items
//--------------------------------------------------------------
   public ArrayPar(int max)          // constructor
      {
      theArray = new long[max];      // create the array
      nElems = 0;                    // no items yet
      }
//--------------------------------------------------------------
   public void insert(long value)    // put element into array
      {
      theArray[nElems] = value;      // insert it
      nElems++;                      // increment size
      }
//--------------------------------------------------------------
   public int size()                 // return number of items
      { return nElems; }
//--------------------------------------------------------------
   public void display()             // displays array contents
      {
      System.out.print("A=");
      for(int j=0; j<nElems; j++)    // for each element,
         System.out.print(theArray[j] + " ");  // display it
      System.out.println("");
      }
//--------------------------------------------------------------
    public int partitionIt(int left, int right, long pivot)
       {
```

```
      int leftPtr = left - 1;           // right of first elem
      int rightPtr = right + 1;         // left of pivot
      while(true)
         {
         while(leftPtr < right &&       // find bigger item
               theArray[++leftPtr] < pivot)
            ; // (nop)

         while(rightPtr > left &&       // find smaller item
               theArray[--rightPtr] > pivot)
            ; // (nop)
         if(leftPtr >= rightPtr)        // if pointers cross,
            break;                      //    partition done
         else                           // not crossed, so
            swap(leftPtr, rightPtr);    //    swap elements
         } // end while(true)
      return leftPtr;                   // return partition
      } // end partitionIt()
//--------------------------------------------------------------
   public void swap(int dex1, int dex2)  // swap two elements
      {
      long temp;
      temp = theArray[dex1];            // A into temp
      theArray[dex1] = theArray[dex2];  // B into A
      theArray[dex2] = temp;            // temp into B
      } // end swap()
//--------------------------------------------------------------
   } // end class ArrayPar
////////////////////////////////////////////////////////////////
class PartitionApp
   {
   public static void main(String[] args)
      {
      int maxSize = 16;             // array size
      ArrayPar arr;                 // reference to array
      arr = new ArrayPar(maxSize);  // create the array

      for(int j=0; j<maxSize; j++)  // fill array with
         {                          // random numbers
         long n = (int)(java.lang.Math.random()*199);
         arr.insert(n);
         }
      arr.display();                // display unsorted array

      long pivot = 99;              // pivot value
```

```
System.out.print("Pivot is " + pivot);
int size = arr.size();
                              // partition array
int partDex = arr.partitionIt(0, size-1, pivot);

System.out.println(", Partition is at index " + partDex);
arr.display();.               // display partitioned array
} // end main()
```

main()例程创建了一个可以容纳 16 个 long 类型数据项的 ArrayPar 对象。枢纽定为 99。例程在 ArrayPar 中插入 16 个任意值，显示它们，调用 partitionIt()方法对它们实行划分，然后再次显示它们。下面是某个例子的输出结果：

```
A=149 192 47 152 159 195 61 66 17 167 118 64 27 80 30 105
Pivot is 99, partition is at index 8
A=30 80 47 27 64 17 61 66 195 167 118 159 152 192 149 105
```

可以看出划分是成功的：前 8 个数字都比枢纽 99 小；后 8 个数字则都大于枢纽。

注意划分的过程不一定要像这个例子一样把数组划分成大小相同的两半；这取决于枢纽以及数据的关键字的值。有可能一组中的数据项个数多于另一组中的数据项个数。

划分算法

划分算法由两个指针开始工作，两个指针分别指向数组的两头。（这里使用“指针”这个词是指示数组数据项的，而不是 C++中所说的指针。）在左边的指针，leftPtr，向右移动，而在右边的指针，rightPtr，向左移动。注意在 partition.java 程序中的 leftPtr 和 rightPtr 分别对应于 Partition 专题 applet 中的 leftScan 和 rightScan。

实际上，leftPtr 初始化时是在第一个数据项的左边一位，rightPtr 是在最后一个数据项的右边一位，这是因为在它们工作之前，它们都要分别的加一和减一。

停止和交换

当 leftPtr 遇到比枢纽小的数据项时，它继续右移，因为这个数据项的位置已经处在数组的正确一边了。但是，当遇到比枢纽大的数据项时，它就停下来。类似的，当 rightPtr 遇到大于枢纽的数据项时，它继续左移，但是当发现比枢纽小的数据项时，它也停下来。两个内层的 while 循环，第一个应用于 leftPtr，第二个应用于 rightPtr，控制这个扫描过程。因为指针退出了 while 循环，所以它停止移动。下面是一段扫描不在适当位置上的数据项的简化代码：

```
while( theArray[++leftPtr] < pivot )   // find bigger item
   ; // (nop)
while( theArray[--rightPtr] > pivot )  // find smaller item
   ; // (nop)
swap(leftPtr, rightPtr);               // swap elements
```

第一个 while 循环在发现比枢纽大的数据项时退出；第二个循环在发现比枢纽小的数据项时退出。当这两个循环都退出之后，leftPtr 和 rightPtr 都指着在数组的错误一方位置上的数据项，所以交换这两个数据项。

交换之后，继续移动两个指针，当指向的数据项在数组的错误一方时，再次停止，然后交换数据项。所有的这些操作都包含在一个外部循环中，参见清单 7.2 中的方法 partitionIt()。当两个指针最终相遇的时候，划分过程结束，并且退出这个外层 while 循环。

当以 100 个竖条来运行 Partition 专题 applet 时，可以观察到指针的活动。这两个用蓝色箭头表示的指针开始时在数组的两头，然后相向移动，在移动的过程中停止并且交换数据项。在它们之间的竖条都是还没有进行划分的；那些它们已经走过的数据项则是划分完的。当两个指针相遇的时候，整个数组划分完毕。

处理异常数据

如果肯定数组最右端的数据项小于枢纽，并且数组最左端的数据项大于枢纽，那么前面看到的简化的 while 循环会很好地执行。遗憾的是，可能用算法来划分的那些数据没有排列得这么好。

例如，如果所有的数据都小于枢纽，leftPtr 变量将会遍历整个数组，徒劳地寻找大于枢纽的数据项，然后跑出数组的最右端，产生数组越界的异常。当所有的数据都大于枢纽的时候，类似的糟糕结果也会发生在 rightPtr 上。

为了避免这些问题，必须在 while 循环中增加数组边界的检测：在第一个循环中增加 leftPtr<right，第二个循环中增加 rightPtr>left。在清单 7.2 中可以看到这些检测的代码。

在快速排序一节，可以看到更为巧妙的枢纽选择过程，它可以去掉这种数组越界的检测。要使程序运行速度更快，在内部循环中削减代码总是一个好方法。

精巧的代码

这个 while 循环中的代码相当精巧。举例来说，想要从内部循环条件中除去加 1 操作符，并且用这个加 1 操作符代替空操作指令语句。（空操作指令指只包括一个分号的语句，它表示不做任何操作。）例如，可以把如下的代码：

```
while(leftPtr < right && theArray[++leftPtr] < pivot)
   ;  // (nop)
```

改为：

```
while(leftPtr < right && theArray[leftPtr] < pivot)
   ++leftPtr;
```

对于另一个内部 while 循环的改变是相似的。这些改变使指针的初始值分别设为 left 和 right 成为可能，这比设为 left-1 和 right+1 要更为清晰。

但是，这些改变导致只有在满足条件的情况下指针才会加 1。指针在任何情况下都必须移动，所以需要在外层的 while 循环中增加两个附加的语句强迫指针变化。空操作指令是最有效的解决办法。

相等的关键字

下面是要在 partitionIt()方法中做的另一个细微的改变。如果想要对所有与枢纽相等的数据项运行 partitionIt()方法，将发现每一次比较都会产生一次交换。交换关键字相等的数据项看起来是浪费时间的。while 循环中对枢纽和数组数据项进行比较的<和>操作符产生了这种额外的交换。然而，假设用<=和>=操作符代替<和>操作符，这确实防止了相等数据项的交换，但是这也使得在算法结束时 leftPtr 和 rightPtr 停在了数组的两端。正如将要在快速排序一节看到的一样，指针最后最好停在

数组的中间，停在数组的两端是非常糟糕的。因此，如果在快速排序中使用 partitionIt()方法，使用<和>操作符是正确的，即便这会引起一些不必要的交换。

划分算法的效率

划分算法的运行时间为 O(N)。在运行 Partition 专题 applet 时很容易明白为什么它的运行时间是 O(N)：两个指针开始时分别在数组的两端，然后以或大或小的恒定速度相向移动，停止移动并且在移动的过程中交换。当两个指针相遇时，划分完成。如果要划分两倍数目的数据项，指针以同样的速率移动，但是需要比较和交换两倍数目的数据项，因此这个过程耗时也是两倍。从而，运行时间和 N 成正比。

更为特别的，每一次划分都有 N+1 或者 N+2 次的比较。每个数据项都由这个或者那个指针指引参与比较，这产生了 N 次比较，但是在指针发现它们已经彼此“越过”之前，它们已经移动过头了，所以在划分完成之前多了一次或者两次额外的比较。比较的次数不取决于数据是如何排列的（除了在扫描结束时的一次或者两次额外比较的不确定性）。

但是，交换的次数确实是取决于数据是如何排列的。如果数据是逆序排列的，并且取的枢纽把数据项分为两半，那么每一对值都需要交换，也就是 N/2 次交换。（记住在 Partition 专题 applet 中枢纽是任意选择的，因此对于逆序排列的竖条的交换次数并不总是 N/2。）

对于任意的数据，在一次划分中交换的次数将小于 N/2，即使一半的竖条小于枢纽，一半的竖条大于枢纽。这是因为总会有一些竖条在正确的位置上（短的竖条在左面，长的竖条在右面）。如果枢纽比大多数的竖条都大（或者小），会有更少的交换次数，这是因为只有很少的大于（或者小于）枢纽的竖条需要交换。对于任意的数据，在平均情况下，大约执行数据项最大数目的一半的交换操作。

尽管交换的次数少于比较的次数，但它们都是和 N 成正比的。因此，划分过程运行 O(N)时间。运行专题 applet，可以看到对 12 个任意的竖条大约需要 3 次交换和 14 次比较，对于 100 个竖条需要有 25 次交换和 102 次的比较。

快速排序

毫无疑问，快速排序是最流行的排序算法，因为有充足的理由，在大多数情况下，快速排序都是最快的，执行时间为 O(N*logN) 级。（这只是对内部排序或者说随机存储器内的排序而言，对于在磁盘文件中的数据进行的排序，其他的排序算法可能更好。）快速排序是在 1962 年由 C.A.R. Hoare 发现的。

为了理解快速排序算法，对于前面一节所描述的划分算法应该非常熟悉。快速排序算法本质上通过把一个数组划分为两个子数组，然后递归地调用自身为每一个子数组进行快速排序来实现的。但是，对这个基本的设计还需要进行一些加工。算法还必须要选择枢纽以及对小的划分区域进行排序。看过这个主要算法的简化代码以后，还要对其进行求精。

在理解快速排序的道理之前想知道快速排序干的是什么，这一点有困难，所以在这里颠倒通常的讲解顺序，在给出 QuickSort1 专题 applet 之前，先列出快速排序的 Java 代码。

快速排序算法

基本的递归的快速排序算法的代码相当简单。下面就是一个实例：

```
public void recQuickSort(int left, int right)
   {
   if(right-left <= 0)          // if size is 1,
       return;                  //    it's already sorted
   else                         // size is 2 or larger
      {
                                          // partition range
      int partition = partitionIt(left, right);
      recQuickSort(left, partition-1);   // sort left side
      recQuickSort(partition+1, right);  // sort right side
      }
   }
```

正如大家看到的一样，有三个基本的步骤：

1. 把数组或者子数组划分成左边（较小的关键字）的一组和右边（较大的关键字）的一组。
2. 调用自身对左边的一组进行排序。
3. 再次调用自身对右边的一组进行排序。

经过一次划分之后，所有在左边子数组的数据项都小于在右边子数组的数据项。只要对左边子数组和右边子数组分别进行排序，整个数组就是有序的了。如何对子数组进行排序呢？通过递归的调用排序算法自身就可以。

传递给 recQuickSort()方法的参数决定了要排序数组（或者子数组）的左右两端的位置。这个方法首先检查数组是否只包含一个数据项。如果数组只包含一个数据项，那么就定义数组已经有序，方法立即返回。这是这个递归过程的基值（终止）条件。

如果数组包含两个或者更多的数据项，算法就调用在前面一节中讲过的 partitionIt()方法对这个数组进行划分。方法返回分割边界的下标数值（partition），它指向右边（较大的关键字）子数组最左端的数据项。划分标记给出两个子数组的分界。如图 7.8 所示。

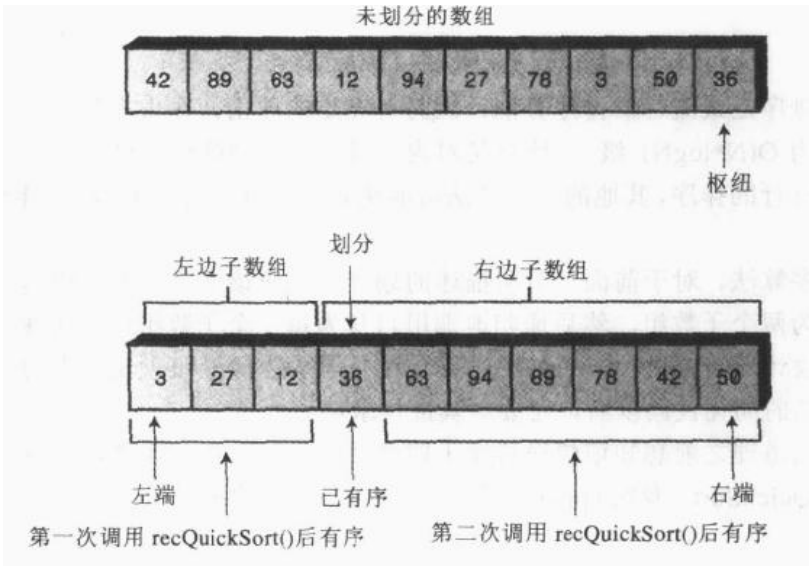


图 7.8 递归调用排序子数组

对数组进行划分之后，recQuickSort()递归地调用自身，数组左边的部分调用一次，对从 left 到 partition-1 位置上的数据项进行排序，数组右边的部分也调用一次，对从 partition+1 到 right 位置上的数据项进行排序。注意这两个递归调用都不包含数组下标为 paritition 的数据项。为什么不包含这个数据项呢？难道下标为 partition 的数据项不需要排序吗？这个原因在于枢纽的选择方法。

选择枢纽

partitionIt()方法应该使用什么样的枢纽呢？以下是一些相关的思想：

- 应该选择具体的一个数据项的关键字的值作为枢纽；称这个数据项为 pivot（枢纽）。
- 可以选择任意一个数据项作为枢纽。为了简便，我们假设总是选择待划分的子数组最右端的数据项作为枢纽。
- 划分完成之后，如果枢纽被插入到左右子数组之间的分界处，那么枢纽就落在排序之后的最终位置上了。

最后一条听起来似乎是不可能的，但请记住，正是因为使用枢纽的关键字的值来划分数组，所以划分之后的左边子数组包含的所有数据项都小于枢纽，右边子数组的所有数据项都大于枢纽。枢纽开始时在右端，但是如果以某种方式把它放在两个子数组之间，枢纽就会在正确的位置上了——也就是说，在它最终排序的位置上了。图 7.9 显示了用关键字为 36 的项作为枢纽的情况。

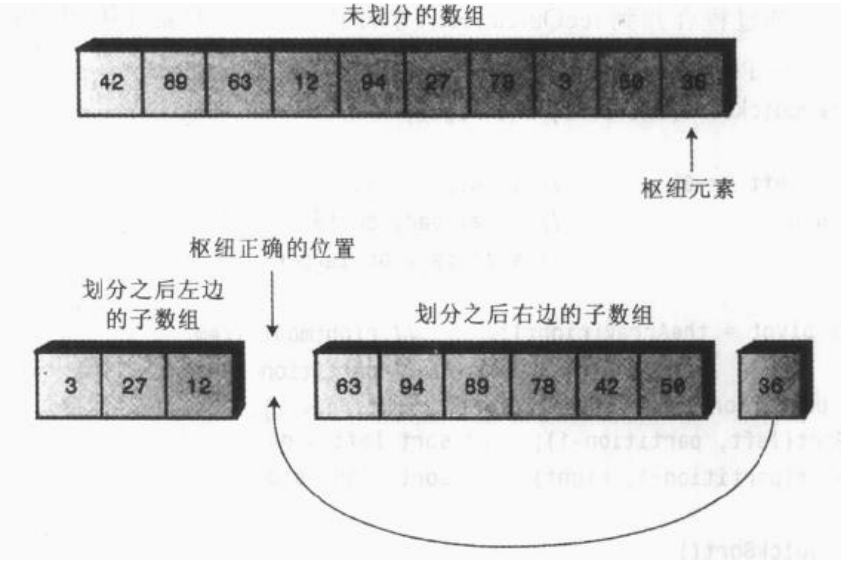


图 7.9 枢纽和子数组

因为不能真正像图中显示的那样把一个数组分开，所以这个图只是一个想像的情况。那么怎样才能把枢纽移动到它的正确位置上来呢？

可以把右边子数组的所有数据项都向右移动一位，以腾出枢纽的位置。但是，这样作既低效又不必要。记住尽管右边子数组的所有数据项都大于枢纽，但它们都还没有排序，所以它们可以在右边子数组内部移动，而没有任何影响。因此，为了简化把枢纽插入正确位置的操作，只要交换枢纽和右边子数组的最左端的数据项（目前是 63）就可以了。这个交换操作把枢纽放在了它正确的位置上，也就是左右子数组之间。63 跳到了最右端，但是由于它还保留在右边（较大的）的一组里，所以划分并没有被打乱。如图 7.10 所示。

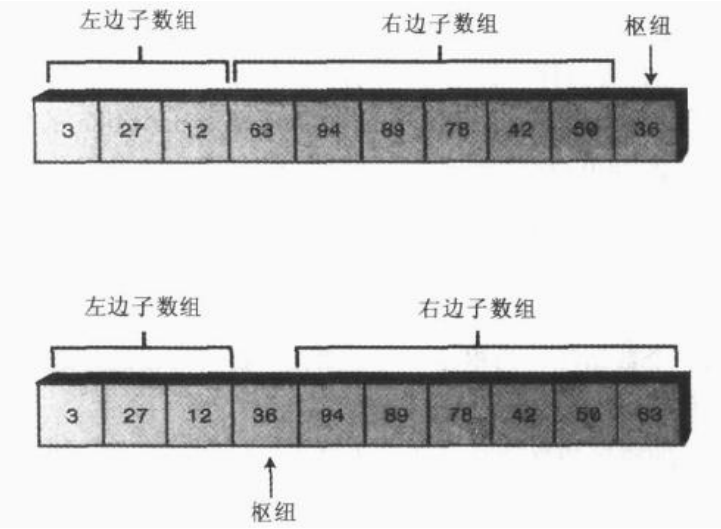


图 7.10 交换枢纽的位置

当枢纽被换到分界的位置时，它落在它最后应该在的位置上。以后所有的操作或者发生在左边或者发生在右边，枢纽本身不会再移动了。

为了把选择枢纽的过程合并到 recQuickSort()方法中，用一个明显的语句为枢纽赋值，并且把枢纽的值作为参数传递给 partitionIt()。代码如下：

```
public void recQuickSort(int left, int right)
      {
      if(right-left <= 0)          // if size <= 1,
          return;                  //    already sorted
      else                         // size is 2 or larger
         {
         long pivot = theArray[right];      // rightmost item
                                            // partition range
         int partition = partitionIt(left, right, pivot);
   recQuickSort(left, partition-1);   // sort left side
   recQuickSort(partition+1, right);  // sort right side
   }
} // end recQuickSort()
```

当使用选择数组最右端的数据项作为枢纽的方案时，需要修改 partitionIt()方法，从划分的过程中把最右端的数据项排除在外。毕竟，这个数据项在划分过程完成之后应该在什么位置已经很清楚：在两个子数组之间的划分点上。划分过程完成之后，还需要把枢纽从数组的最右端移动到划分点（partition）的位置上。清单 7.3 显示了综合了所有以上这些特性的 quickSort1.java 程序。

清单 7.3 quickSort1.java 程序

```
// quickSort1.java
// demonstrates simple version of quick sort
// to run this program: C>java QuickSort1App
////////////////////////////////////////////////////////////////
```

```
class ArrayIns
   {
   private long[] theArray;          // ref to array theArray
   private int nElems;               // number of data items
//--------------------------------------------------------------
   public ArrayIns(int max)          // constructor
      {
      theArray = new long[max];      // create the array
      nElems = 0;                    // no items yet
      }
//--------------------------------------------------------------
   public void insert(long value)    // put element into array
      {
      theArray[nElems] = value;      // insert it
      nElems++;                      // increment size
      }
//--------------------------------------------------------------
   public void display()             // displays array contents
      {
      System.out.print("A=");
      for(int j=0; j<nElems; j++)    // for each element,
         System.out.print(theArray[j] + " ");  // display it
      System.out.println("");
      }
//--------------------------------------------------------------
   public void quickSort()
      {
      recQuickSort(0, nElems-1);
      }
//--------------------------------------------------------------
   public void recQuickSort(int left, int right)
      {
      if(right-left <= 0)               // if size <= 1,
          return;                       //    already sorted
      else                              // size is 2 or larger
         {
         long pivot = theArray[right];      // rightmost item
                                            // partition range
         int partition = partitionIt(left, right, pivot);
         recQuickSort(left, partition-1);   // sort left side
         recQuickSort(partition+1, right);  // sort right side
         }
      }  // end recQuickSort()
//--------------------------------------------------------------
   public int partitionIt(int left, int right, long pivot)
```

```
        {
        int leftPtr = left-1;           // left    (after ++)
        int rightPtr = right;           // right-1 (after --)
        while(true)
           {                            // find bigger item
           while( theArray[++leftPtr] < pivot )
              ; // (nop)
                                        // find smaller item
           while(rightPtr > 0 && theArray[--rightPtr] > pivot)
              ; // (nop)

           if(leftPtr >= rightPtr)      // if pointers cross,
              break;                    //    partition done
           else                         // not crossed, so
              swap(leftPtr, rightPtr);  //    swap elements
           } // end while(true)
        swap(leftPtr, right);           // restore pivot
        return leftPtr;                 // return pivot location
        } // end partitionIt()
//--------------------------------------------------------------
   public void swap(int dex1, int dex2)  // swap two elements
      {
      long temp = theArray[dex1];        // A into temp
      theArray[dex1] = theArray[dex2];   // B into A
      theArray[dex2] = temp;             // temp into B
      } // end swap(
//--------------------------------------------------------------
   } // end class ArrayIns
////////////////////////////////////////////////////////////////
class QuickSort1App
   {
   public static void main(String[] args)
      {
      int maxSize = 16;             // array size
      ArrayIns arr;
      arr = new ArrayIns(maxSize);  // create array

      for(int j=0; j<maxSize; j++)  // fill array with
         {                          // random numbers
         long n = (int)(java.lang.Math.random()*99);
         arr.insert(n);
         }
      arr.display();                // display items
      arr.quickSort();              // quicksort them
      arr.display();                // display them again
```

```
    } // end main()
} // end class QuickSort1App
```

main()例程创建了一个 ArrayIns 类型的对象，在其中插入了 16 个任意的长整型（long）的数据项，显示数组，用 quickSort()方法对其进行排序，然后显示排序后的结果。下面是某个典型的输出结果：

```
A=69 0 70 6 38 38 24 56 44 26 73 77 30 45 97 65
A=0 6 24 26 30 38 38 44 45 56 65 69 70 73 77 97
```

在 partitionIt()方法中有趣的一点是，可以删除第一个内部 while 循环中对是否超越数组右端的检测。在之前的清单 7.2 中所列的 partition.java 程序中早期的 partitionIt()方法中的这个检测，即

```
leftPtr < right
```

在没有大于枢纽的数据项时，它防止了 leftPtr 右移越过数组的右端。为什么可以去除这个检测呢？这是因为选择最右端的数据项作为枢纽，所以 leftPtr 总会停在枢纽这个位置上。但是，第二个 while 循环中的 rightPtr 仍然需要这个检测。（后面会讲如何同样地消除这个检测。）

选择最右端的数据项作为枢纽不属于完全随意的选择，但是这样消除了不必要的检测，以此来加速代码的执行。选择某个其他位置上的数据项作为枢纽不能体现这个优势。

QuickSort1 专题 applet

现在，对快速排序算法的内容已经知道得足够多了，足可以理解 QuickSort1 专题 applet 中的微妙差别。

大图

为制作大图，先使用 Size 按钮为 applet 设置 100 个任意高度的竖条，然后点击 Run 按钮。如图 7.11 所示，显示了排序之后的状况。

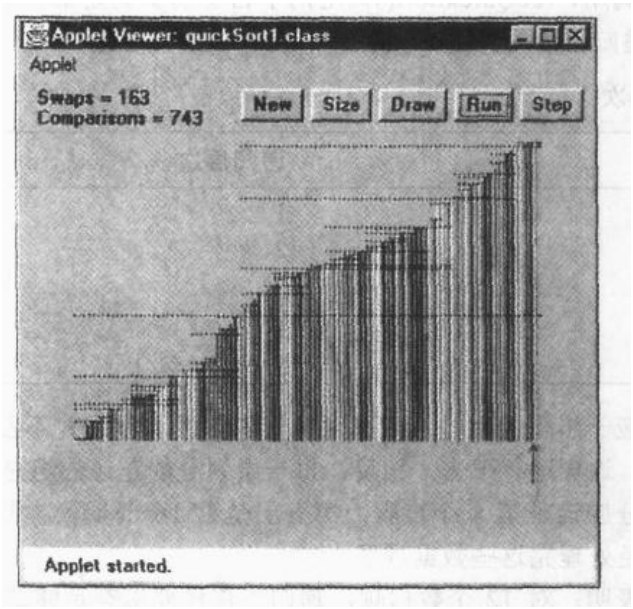


图 7.11 100 个竖条的 QuickSort1 专题 applet

观察算法如何把数组划分为两部分，然后把每一部分再划分为两部分，如此划分下去，创建越来越小的子数组。

当排序过程结束之后，每一条虚线都表示了一个已经排序好的子数组。虚线的水平长度范围内的竖条表示了一个子数组，并且虚线的垂直高度代表了枢纽（枢纽的高度）。所有这些显示的虚线的长度表明了算法为这个数组排序做了多少工作；后面会再次回到这个问题。

每一条虚线（除了最短的虚线之外）在其下都会有另外一条虚线（很可能被其他的更短的虚线所隔开），其上也会有一条虚线，把这两条线加到一起的长度和原来的虚线的长度（少一个竖条）一样。每个子数组都被分割成两个部分。

细节

为了更详细地了解快速排序的操作，在 QuickSort1 专题 applet 中设置 12 个竖条，并且单步执行排序的过程。这样可以看到枢纽是如何和在数组最右端的枢纽的高度保持一致的，以及算法如何划分数组，把枢纽交换到两个有序的组之间，对比枢纽低的组排序（使用很多递归的调用），然后再对比枢纽高的组排序。

图 7.12 显示了对 12 个竖条排序的所有的步骤。数组下面的水平大括弧中表示每一步正在被划分的子数组，圆圈中的数字表示执行这些划分的顺序。正在交换位置的枢纽用一个虚线箭头表示。枢纽的最后位置用虚线单元表示，以着重说明这个单元所含的是以后不会再改变的排好序的数据项。在一些单独的数据项下有水平大括弧（如步骤 5、6、7、11 和 12），这是调用 recQuickSort()的基值（终止）条件的情况，此时它们立刻返回。

有时，如在第 3 步和第 9 步中显示的那样，枢纽最后还在原来的位置上，即正在排序的数组的右侧。在这种情况下，只有一个子数组需要排序：枢纽左边的子数组。在枢纽的右边没有第二个子数组。

图 7.12 中的不同的步骤出现在不同的递归调用层次上，如表 7.3 所列。main()最初调用 recQuickSort()是第一层调用，recQuickSort()调用两个自身的实例是第二层递归，这两个实例又调用 4 个新的实例是第三层递归，依此类推。

表 7.3 图 7.12 的递归层次

步骤	递归层次
1	1
2,8	2
3,7,9,12	3
4,10	4
5,6,11	5

划分执行的顺序对应于步骤的序号，而不对应于递归的深度。并不都是首先处理第一层划分，然后再处理第二层划分，这样运行下去。相反，每一层划分的左边的组总是比右边的组先处理。

理论上，第四层划分应该是第 8 个步骤，第五层是第 16 个步骤，但是在这个小数组中，还没有执行这么多步骤就已经处理完这些数据项了。

表中所列的层次数表明：对 12 个数据项，递归工作栈要求有足够的空间来记录 5 个参数和返回地址集；每层递归对应于一个参数和返回地址集。正如将在后面看到的一样，它的大小略大于以

2 为底的数组数据项个数的对数：$\log_2 N$。递归工作栈的大小由特定的机器系统所决定。使用递归算法对规模非常大的数据项进行排序可能会引起栈溢出，导致存储错误。

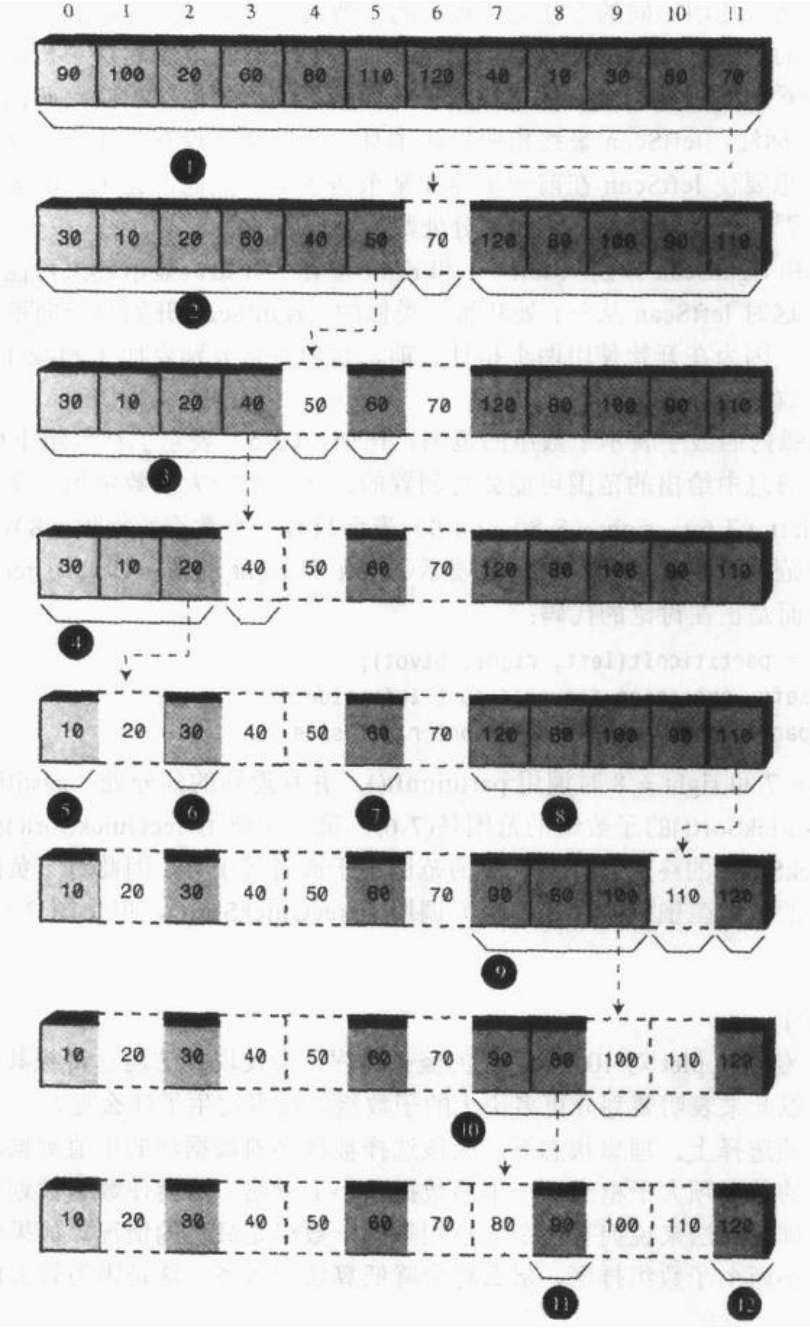


图 7.12 快速排序过程

注意的事情

下面是运行 QuickSort1 专题 applet 时需要注意的一些细节。

有人可能会认为像快速排序这样的高效的算法应该不能处理只有两三个数据项大小的子数组。

但是，这里讲的快速排序算法可以很好地对如此小的子数组进行排序；只是在 leftScan 和 rightScan 相遇之前，它们不会移动很大的距离。因此，对于很小的子数组不需要使用不同的排序方法。（不过，在后面可以看到，使用不同的方法处理很小的子数组可能会有很多好处。）

到每一趟扫描的最后，leftSan 变量最终总是指向划分处，也就是右边子数组的最左边一个数据项。然后正如已经看到的一样，交换枢纽和划分处的数据项，把枢纽放到正确的位置上。如图 7.12 步骤 3 和步骤 9 的标注，leftScan 最终指向枢纽本身，所以交换没有起作用。这似乎是一个多余的交换；有人可能会想要使 leftScan 在前一步停在某个竖条上。然而，让 leftScan 扫描全程一直到枢纽是非常重要的；否则，交换会使枢纽和划分处数据项排不成序。

注意 leftSacn 和 rightScan 分别从 left – 1 和 right 位置上开始。显示起来可能比较特殊，特别是当 left 等于 0 时；这时 leftScan 从－1 处开始。类似的，rightScan 开始时指向枢纽，而枢纽并不包括在划分过程当中。因为在开始使用两个指针之前，它们各自分别要加 1 和减 1，所以开始时它们都在要被划分的子数组之外。

applet 用圆括弧内的数字表示子数组的范围；比如，（2-5）表示了从数组下标 2 到数组下标 5 的子数组。在某些信息中给出的范围可能会是倒置的：从一个较大的数字到一个较小的数字，例如 Array partitioned; left（7-6），right（8-8）。（8-8）表示只有一个数组数据项（8），但是（7-6）表示什么意思呢？这个范围不是真正的；它只是表示了 left 和 right 的值，即调用 recQuickSort()方法时该方法的参数。下面是正在讨论的代码：

```
int partition = partitionIt(left, right, pivot);
recQuickSort(left, partition-1);   // sort left side
recQuickSort(partition+1, right);  // sort right side
```

例如，当 left = 7 和 right = 8 时调用 partitionIt()，并且返回的划分处（partition）恰好为 7，那么第一次调用 recQuickSort()的子数组的范围是(7-6)，第二次调用 recQuickSort()的范围是(8-8)。这是正常的。recQuickSort()的终止条件是数组的范围小于或者等于 1，因此对于负的数组范围，方法立即返回。尽管负的数组范围确实（简单地）调用了 recQuickSort()，但是图 7.12 中没有显示负的数组范围。

性能降到了 $O(n^2)$

用 QuickSort1 专题 applet 对 100 个逆序的竖条排序，会发现算法运行得极其缓慢，而且会产生更多的水平虚线，以此来表明要划分更多更大的子数组。这里发生了什么呢？

问题出在枢纽的选择上。理想状态下，应该选择被排序的数据项的中值数据项作为枢纽。也就是说，应该有一半的数据项大于枢纽，一半的数据项小于枢纽。这会使数组被划分成两个大小相等的子数组。对快速排序算法来说拥有两个大小相等的子数组是最优的情况。如果快速排序算法必须要对划分的一大一小两个子数组排序，那么将会降低算法的效率，这是因为较大的子数组必须要被划分更多次。

N 个数据项数组的最坏的划分情况是一个子数组只有一个数据项，另一个子数组含有 N-1 个数据项。（这种划分为 1 个数据项与 N-1 个数据项的情况在图 7.12 步骤 3 和步骤 9 中也可以看到。）如果在每一趟划分中都出现这种 1 个数据项和 N-1 个数据项的分割，那么每一个数据项都需要一次单独的划分步骤。在逆序排列的数据项中实际上发生的就是这种情况：所有的子数组中，枢纽都是

最小的数据项，因此每一次划分都产生一个有 N-1 个数据项的子数组以及另外一个只包含枢纽的子数组。

为了观察这种最坏情况的执行过程，以 12 个逆序排列的竖条为例运行 QuickSort1 专题 applet。注意与对 12 个任意排序的竖条排序相比它需要增加多少个步骤。在这种情况下，划分所带来的好处都没有了，算法的执行效率降低到 $O(N^2)$。

快速排序以 $O(N^2)$运行的时候，除了慢还有另外的一个潜在的问题。当划分的次数增加的时候，递归方法的调用次数也增加了。每一个方法调用都要增加所需递归工作栈的大小。如果调用次数太多，递归工作栈可能会发生溢出，从而使系统瘫痪。

总之：在 QuickSort1 applet 中，选择最右端的数据项作为枢纽。如果数据项真的是任意排列的，那么这个选择不会太坏，因为通常情况下枢纽不会太靠近数组的两端。但是，当数据是有序的或者是逆序时，从数组的一端或者另外一端选择数据项作为枢纽都不是好办法。能改进选择枢纽的方法吗？

“三数据项取中”划分

人们已经设计出了很多选择更好的枢纽的方法。方法应该简单但能避免出现选择最大或者最小数据项作为枢纽的情况。选择任意一个数据项作为枢纽的确是很简单的，但是正如我们已经看到的那样，这并不总是一个好的选择。可以检测所有的数据项，并且实际计算哪一个数据项是中值数据项。这应该是理想的枢纽选择，可是由于这个过程需要比排序本身更长的时间，因此它不可行。

折衷的方法是找到数组里第一个、最后一个以及中间位置数据项的居中数据项值，并且设此数据项为枢纽。选择第一个、最后一个以及中间位置数据项的中值被称为“三数据项取中”(median-of-three)方法，如图 7.13 所示。

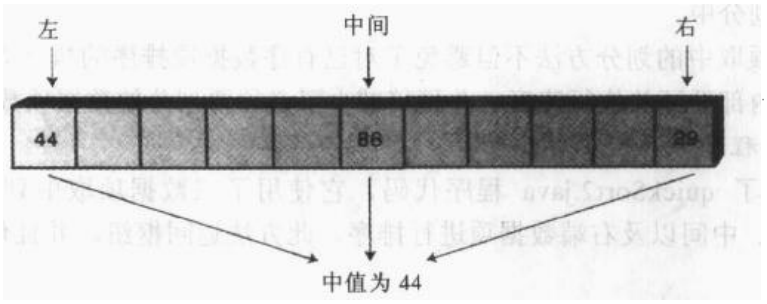


图 7.13 三数据项取中

查找三个数据项的中值数据项自然比查找所有数据项的中值数据项快很多，同时这也有效地避免了在数据已经有序或者逆序的情况下，选择最大的或者最小的数据项作为枢纽的机会。很可能存在一些很特殊的数据排列使得三数据项取中的方法很低效，但是通常情况下，对于选择枢纽它都是一个又快又有效的方法。

三数据项取中方法除了选择枢纽更为有效之外，还有一个额外的好处：可以在第二个内部 while 循环中取消 rightPtr>left 的测试，以略微提高算法的执行速度。这是怎样实现的呢？

因为在选择的过程中使用三数据项取中的方法不仅选择了枢纽，而且还对三个数据项进行了排序，所以可以消除这个测试。图 7.14 显示了这个操作过程。

当这三个数据项已经排好序，并且已经选择中值数据项作为枢纽后，这时就可以保证子数组最

左端的数据项小于（或者等于）枢纽，最右端的数据项大于（或者等于）枢纽。这就意味着即使取消 leftPtr>right 和 rightPtr<left 的检测，leftPtr 和 rightPtr 也不会分别越过数组的右端或者左端。（指针停止移动，考虑是否需要交换数据项，只要发现它和另外一个指针位置交叉，则划分过程结束。）left 和 right 的值充当了监视哨，以确保不会使 leftPtr 和 rightPtr 超出数组范围。

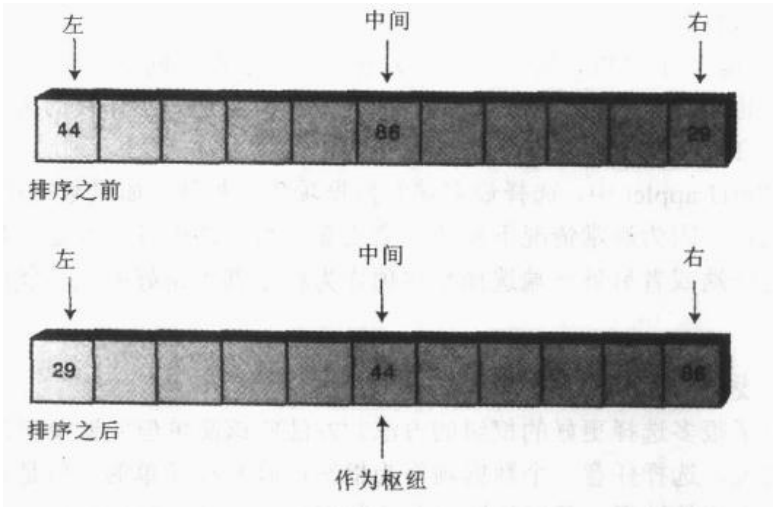


图 7.14 排序左端、中间以及右端的数据项

三数据项取中的另外一个小的好处是，对左端、中间以及右端的数据项排序之后，划分过程就不需要再考虑这三个数据项了。划分可以从 left+1 和 right−1 开始，因为 left 和 right 已经被有效地划分了。因为 left 在左边并且小于枢纽，所以它已经在正确的划分中了，right 在右边且大于枢纽，所以它也在正确的划分中。

这样，三数据项取中的划分方法不但避免了对已有序数据项排序的执行效率为 $O(N^2)$，而且它也提高了划分算法内部循环的执行速度，并稍稍减少了必须要划分的数据项数目。

quickSort2.java 程序

清单 7.4 显示了 quickSort2.java 程序代码，它使用了三数据项取中划分方法。独立的方法 medianOf3()对左端、中间以及右端数据项进行排序。此方法返回枢纽，并且传递给 partition()It 方法。

清单 7.4 quickSort2.java 程序

```
// quickSort2.java
// demonstrates quick sort with median-of-three partitioning
// to run this program: C>java QuickSort2App
////////////////////////////////////////////////////////////////
class ArrayIns
   {
   private long[] theArray;          // ref to array theArray
   private int nElems;               // number of data items
//--------------------------------------------------------------
   public ArrayIns(int max)          // constructor
```

```
      {
      theArray = new long[max];      // create the array
      nElems = 0;                    // no items yet
      }
//--------------------------------------------------------------
   public void insert(long value)    // put element into array
      {
      theArray[nElems] = value;      // insert it
      nElems++;                      // increment size
      }
//--------------------------------------------------------------
   public void display()             // displays array contents
      {
      System.out.print("A=");
      for(int j=0; j<nElems; j++)    // for each element,
         System.out.print(theArray[j] + " ");  // display it
      System.out.println("");
      }
//--------------------------------------------------------------
   public void quickSort()
      {
      recQuickSort(0, nElems-1);
      }
//--------------------------------------------------------------
   public void recQuickSort(int left, int right)
      {
      int size = right-left+1;
      if(size <= 3)                  // manual sort if small
         manualSort(left, right);
      else                           // quicksort if large
         {
         long median = medianOf3(left, right);
         int partition = partitionIt(left, right, median);
         recQuickSort(left, partition-1);
         recQuickSort(partition+1, right);
         }
      } // end recQuickSort()
//--------------------------------------------------------------
   public long medianOf3(int left, int right)
      {
      int center = (left+right)/2;
                                     // order left & center
      if( theArray[left] > theArray[center] )
         swap(left, center);
                                     // order left & right
```

```
      if( theArray[left] > theArray[right] )
         swap(left, right);
                                        // order center & right
      if( theArray[center] > theArray[right] )
         swap(center, right);

      swap(center, right-1);            // put pivot on right
      return theArray[right-1];         // return median value
      } // end median0f3()
//-------------------------------------------------------------
   public void swap(int dex1, int dex2)  // swap two elements
      {
      long temp = theArray[dex1];       // A into temp
      theArray[dex1] = theArray[dex2];  // B into A
      theArray[dex2] = temp;            // temp into B
      } // end swap(
//-------------------------------------------------------------
    public int partitionIt(int left, int right, long pivot)
       {
       int leftPtr = left;             // right of first elem
       int rightPtr = right - 1;       // left of pivot

       while(true)
          {
          while( theArray[++leftPtr] < pivot )  // find bigger
             ;                                  //    (nop)
          while( theArray[--rightPtr] > pivot ) // find smaller
              ;                                 //    (nop)
           if(leftPtr >= rightPtr)      // if pointers cross,
              break;                    //    partition done
           else                         // not crossed, so
              swap(leftPtr, rightPtr);  // swap elements
           } // end while(true)
        swap(leftPtr, right-1);         // restore pivot
        return leftPtr;                 // return pivot location
        } // end partitionIt()
//-------------------------------------------------------------
    public void manualSort(int left, int right)
       {
       int size = right-left+1;
       if(size <= 1)
          return;         // no sort necessary
       if(size == 2)
          {               // 2-sort left and right
          if( theArray[left] > theArray[right] )
```

```
            swap(left, right);
         return;
         }
      else                 // size is 3
         {                 // 3-sort left, center, & right
         if( theArray[left] > theArray[right-1] )
            swap(left, right-1);                // left, center
         if( theArray[left] > theArray[right] )
            swap(left, right);                  // left, right
         if( theArray[right-1] > theArray[right] )
            swap(right-1, right);               // center, right
         }
      }  // end manualSort()
//--------------------------------------------------------------
   }  // end class ArrayIns
////////////////////////////////////////////////////////////////
class QuickSort2App
   {
   public static void main(String[] args)
      {
      int maxSize = 16;             // array size
      ArrayIns arr;                 // reference to array
      arr = new ArrayIns(maxSize);  // create the array
      for(int j=0; j<maxSize; j++)  // fill array with
         {                          // random numbers
         long n = (int)(java.lang.Math.random()*99);
         arr.insert(n);
         }
      arr.display();                // display items
      arr.quickSort();              // quicksort them
      arr.display();                // display them again
      }  // end main()
   }  // end class QuickSort2App
```

程序使用了另一个新方法 manualSort()，对只有三个或者更少数据项的子数组进行排序。当子数组中只有一个数据项（或者更少）时方法立即返回，有两个数据项时，如果需要则交换这两个数据项，有三个数据项时对三个数据项进行排序。由于中值划分要求至少含有四个数据项，所以recQuickSort()例程不能应用于只有两个或者三个数据项的子数组。

main()例程以及 quickSort2.java 程序的输出结果与 quickSort1.java 类似。

QuickSort2 专题 applet

QuickSort2 专题 applet 显示了利用三数据项取中划分的快速排序算法。此 applet 和 QuickSort1 专题 applet 相似，但是它首先排序第一个数据项、中间数据项以及每个子数组的左边一个数据项，并且选择三个数据项的中值作为枢纽。至少，当数组的大小大于 3 时，applet 是这么做的。当子数

组只有两个或者三个数据项时，applet 只是“手工地”对其进行排序，而不使用划分或者递归调用。

注意此 applet 对 100 个逆序排列的竖条排序时，执行效率大大提高了。每个子数组不再被划分为 1 个数据项和 N-1 个数据项两组了，而是被划分成大致相等的两组。

QuickSort2 专题 applet 除了对有序数据项排序改进了效率之外，其他时候与 QuickSort1 执行效果相似。它对随机数据排序时并没有执行得更快，只是在对有序数据排序时它的优势才表现出来。

处理小划分

如果使用三数据项取中划分方法，则必须要遵循快速排序算法不能执行三个或者少于三个数据项的划分的规则。在这种情况下，数字 3 被称为切割点（cutoff）。在上面的例子中，是用一段代码手动地对两个或三个数据项的子数组来排序的。这是最好的方法吗？

对小划分使用插入排序

处理小划分的另一个选择是使用插入排序。当使用插入排序的时候，不用限制以 3 为切割点（cutoff）。可以把界限定为 10、20 或者其他任何数。试验不同切割点的值以找到最好的执行效率，这件事是很有意义的。Knuth（参见附录 B）推荐使用 9 作为切割点。但是，最好的选择值取决于计算机、操作系统、编译器（或者解释器）等。

quickSort3.java 程序如清单 7.5 所示，它使用插入排序来处理小于 10 个数据项的子数组。

清单 7.5 quickSort3.java 程序

```
// quickSort3.java
// demonstrates quick sort; uses insertion sort for cleanup
// to run this program: C>java QuickSort3App
////////////////////////////////////////////////////////////////
class ArrayIns
   {
   private long[] theArray;          // ref to array theArray
   private int nElems;               // number of data items
//--------------------------------------------------------------
   public ArrayIns(int max)          // constructor
      {
      theArray = new long[max];      // create the array
      nElems = 0;                    // no items yet
      }
//--------------------------------------------------------------
   public void insert(long value)    // put element into array
      {
      theArray[nElems] = value;      // insert it
      nElems++;                      // increment size
      }
//--------------------------------------------------------------
   public void display()             // displays array contents
      {
      System.out.print("A=");
      for(int j=0; j<nElems; j++)    // for each element,
```

```
        System.out.print(theArray[j] + " ");  // display it
      System.out.println("");
      }
//--------------------------------------------------------
   public void quickSort()
      {
      recQuickSort(0, nElems-1);
      // insertionSort(0, nElems-1); // the other option
      }
//--------------------------------------------------------
   public void recQuickSort(int left, int right)
      {
      int size = right-left+1;
      if(size < 10)                  // insertion sort if small
         insertionSort(left, right);
      else                           // quicksort if large
         {
         long median = medianOf3(left, right);
         int partition = partitionIt(left, right, median);
         recQuickSort(left, partition-1);
         recQuickSort(partition+1, right);
         }
      } // end recQuickSort()
//--------------------------------------------------------
   public long medianOf3(int left, int right)
      {
      int center = (left+right)/2;
                                     // order left & center
      if( theArray[left] > theArray[center] )
         swap(left, center);
                                     // order left & right
      if( theArray[left] > theArray[right] )
         swap(left, right);
                                     // order center & right
      if( theArray[center] > theArray[right] )
         swap(center, right);

      swap(center, right-1);         // put pivot on right
      return theArray[right-1];      // return median value
      } // end medianOf3()
//--------------------------------------------------------
   public void swap(int dex1, int dex2)  // swap two elements
      {
      long temp = theArray[dex1];        // A into temp
      theArray[dex1] = theArray[dex2];   // B into A
      theArray[dex2] = temp;             // temp into B
```

```
      } // end swap(
//--------------------------------------------------------------
   public int partitionIt(int left, int right, long pivot)
      {
      int leftPtr = left;             // right of first elem
      int rightPtr = right - 1;       // left of pivot
      while(true)
         {
         while( theArray[++leftPtr] < pivot )  // find bigger
            ;                                  // (nop)
         while( theArray[--rightPtr] > pivot ) // find smaller
            ;                                  // (nop)
         if(leftPtr >= rightPtr)       // if pointers cross,
            break;                     //    partition done
         else                          // not crossed, so
            swap(leftPtr, rightPtr);   // swap elements
         } // end while(true)
      swap(leftPtr, right-1);          // restore pivot
      return leftPtr;                  // return pivot location
      } // end partitionIt()
//--------------------------------------------------------------
                                       // insertion sort
   public void insertionSort(int left, int right)
      {
      int in, out;
                                       //  sorted on left of out
      for(out=left+1; out<=right; out++)
         {
         long temp = theArray[out];    // remove marked item
         in = out;                     // start shifts at out
                                       // until one is smaller,
         while(in>left && theArray[in-1] >= temp)
            {
            theArray[in] = theArray[in-1]; // shift item to right
            --in;                      // go left one position
            }
         theArray[in] = temp;          // insert marked item
         } // end for
      } // end insertionSort()
//--------------------------------------------------------------
   } // end class ArrayIns
////////////////////////////////////////////////////////////////
class QuickSort3App
   {
   public static void main(String[] args)
```

```
{
int maxSize = 16;            // array size
ArrayIns arr;                // reference to array
arr = new ArrayIns(maxSize); // create the array

for(int j=0; j<maxSize; j++) // fill array with
   {                         // random numbers
   long n = (int)(java.lang.Math.random()*99);
   arr.insert(n);
   }
arr.display();               // display items
arr.quickSort();             // quicksort them
arr.display();               // display them again
} // end main()
```

在这个特别的算法中，对小的子数组使用插入排序被证实为最快的一种方法，但是它不比在quickSort2.java 中对三个或者更少的数据项的子数组手动地排序快。比较和复制的次数在快速排序阶段都大大地减少了，但是在插入排序阶段却又上升了差不多相同的次数，因此并没有明显地节省时间。但是，要想把快速排序的性能发挥到极限，这个方法大概还是有用的。

快速排序之后的插入排序

另一个选择是对数组整个使用快速排序，不去考虑小于界限的划分的排序。在 quickSort()方法中的注释部分表明了这种做法。（如果使用这种做法，那么应该从 recQuickSort()中删除对方法 insertionSort()的调用。）当快速排序结束时，数组已经是基本有序的了。然后可以对整个数组应用插入排序。插入排序对基本有序的数组执行效率很高，而且很多专家都提倡使用这个方法，但是在我们的设置中，这个方法运行得很慢。插入排序显然更合适做很多小规模的排序，而不是一个大的排序。

消除递归

很多作者提倡的对快速排序算法的另一个可取的修改是消除递归。这包括重写算法以在栈中存储递推的子数组的上下界（left 和 right），以及使用循环代替递归来执行越来越小的子数组的划分。这么做的思想是通过取消方法调用来加速算法的执行。但是，这个思想源于早先的编译器以及计算机体系结构，对于每一次方法调用那种旧的系统都会导致大量的时间消耗。对于现在的系统来说，消除递归所带来的改进不是很明显，因为现在的系统可以更为有效地处理方法调用。

快速排序的效率

快速排序的时间复杂度为 O(N*logN)。正如第 6 章的归并排序中所讨论的那样，对于分治算法总的来说都是这样的，在分治算法中用递归的方法把一列数据项分为两组，然后调用自身来分别处理每一组数据项。这种情况下，算法实际上是以 2 为底的：运行时间和 $N* \log_2 N$ 成正比。

通过运行一次设置为 100 个随机排列的竖条的 quickSort 专题 applet，并且观察所显示的结果水平虚线，可以对快速排序 $N* \log_2 N$ 的运行时间的正确性有清楚的认识。

每一条虚线都表示了一个正在被划分的数组或者子数组：指针 leftScan 和 rightScan 相向移动，

比较每一对数据项并且在恰当的时候交换数据项。在“划分”一节中我们了解到一个单独划分的运行时间为 O(N)。这说明所有虚线的总长度和快速排序的运行时间成正比。但所有虚线总长是多少呢？用尺子在显示器上测量它们是很繁琐的，但是可以用其他的方法测量这些虚线。

总有一条虚线长度为整个图形的宽度，横跨了 N 个竖条。它是第一次划分的结果。还有两条虚线（在第一条产生的虚线上下各一条）的平均长度为 N/2 个竖条的宽度；它们的总长也是 N 个竖条的宽度。然后有 4 条平均长度为 N/4 个竖条宽度的虚线，其总长也为 N 个竖条的宽度，接着有 8 条虚线，16 条虚线，等等。图 7.15 显示了 1、2、4 和 8 条虚线时的情况。

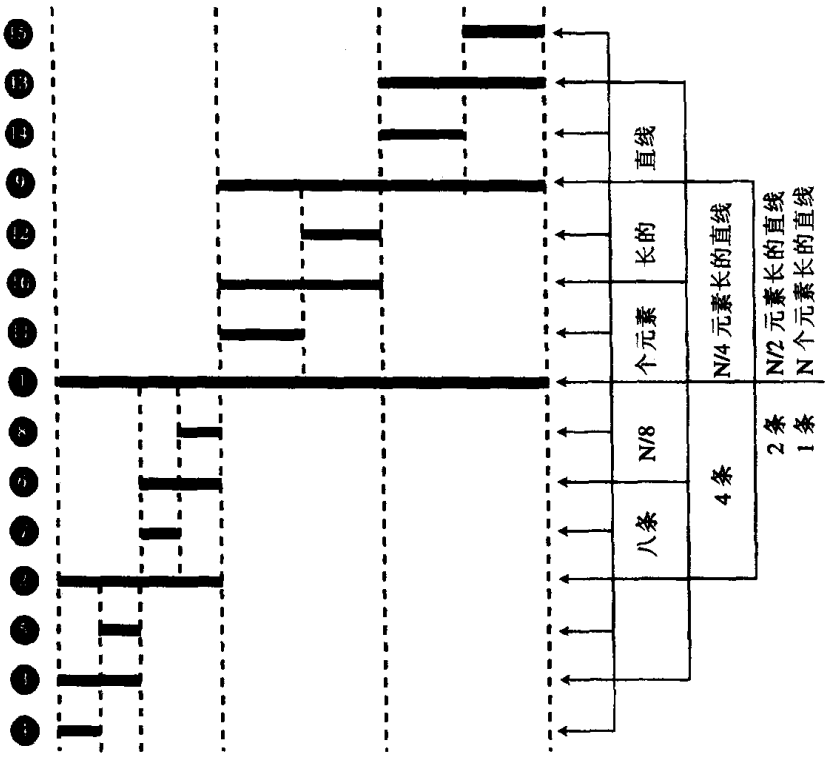


图 7.15 对应划分的直线

图中用水平的实线代表了快速排序 applet 中的水平虚线，类似“N/4 数据项长”的说明指的是平均长度，而不是直线的真实长度。左侧圆圈内的数字表示了生成直线的顺序。

每一组直线（例如，8 条 N/8 的直线）对应一层递归。第一次调用 recQuickSort()是第一层，产生第一条直线；第一次调用内部又两次调用 recQuickSort()，这是第二层调用，它产生了两条直线；依此类推。假设开始设置为 100 个竖条，结果如表 7.4 所示。

表 7.4 直线长度和递归

递归层次	图 7.15 中的步骤数	平均直线长度（元素数）	直线条数	总线（元素数）
1	1	100	1	100
2	2,9	50	2	100
3	3,6,10	25	4	100
	13			

续表

递归层次	图 7.15 中的步骤数	平均直线长度（元素数）	直线条数	总线（元素数）
4	4,5,7 8,11,12 14,15	12	8	96
5	未显示	6	16	96
6	未显示	3	32	96
7	未显示	1	64	64
				Total=652

划分过程在哪里结束呢？如果 100 不断地被 2 整除，算出划分的次数，得到数列 100，50，25，12，6，3，1，也就是大约有 7 层递归。在专题 applet 中可以看出这是正确的：在图中挑选一个竖条，计算所有向上或向下的虚线，平均大约为 7。（图 7.15 中，因为没有显示所有的递归，所以图中只有四条垂直虚线。）

表 7.4 显示总共有 652 个数据项。因为取整的误差，所以这只是一个近似值，但是它接近于 100 乘以以 2 为底的 100 的对数，后者为 6.65。因此，这个不正规的分析表明了快速排序 N* $\log_2 N$ 运行时间级的正确性。

更特殊的是，在划分一节中，我们发现应该有 N+2 次比较和少于 N/2 次交换。表 7.5 给出了对不同的 N 值，求 $\log_2 N$ 与比较和交换次数相乘的结果。

表 7.5 快速排序中的交换和比较

N	8	12	16	64	100	128
$\log_2 N$	3	3.59	4	6	6.65	7
$N*\log_2 N$	24	43	64	384	665	896
比较：$(N+2)*\log_2 N$	30	50	72	396	678	910
交换：fewer than $N/2*\log_2 N$	12	21	32	192	332	448

表 7.5 中用到的 $\log_2 N$ 只有在每一个子数组都划分为大小相等的两部分这种最好的情况下才是真正准确的。对于任意的数据，这个数字略大。不过，QuickSort1 和 QuickSort2 专题 applet 运行 12 个和 100 个竖条的运行结果和表中所列是很相近的，这可以通过运行 applet，并且观察交换和比较的信息来看到。

因为 QuickSort1 和 QuickSort2 有不同的切割点（cutoff），并且处理产生的小划分的方法不同，所以 QuickSort1 比 QuickSort2 执行的交换次数少，但比较次数多。表 7.5 中所列的交换次数是最大值（假设数据项是逆序排列的）。对任意排列的数据项，实际的交换次数是表中所列数字的二分之一或者三分之二。

基数排序

本章的最后简要地讨论一种完全不同的排序方法。前面的排序算法都是对简单数值型关键字的

处理，也就是通过比较关键字的值来实现排序的。基数排序则是把关键字拆分成数字位。并且按数字位的值对数据项进行排序。奇怪的是，实现基数排序不需要比较操作。

基数排序算法

这里讨论的基数排序都是普通的以 10 为基数的运算，因为这样更易于讲解。但是，以 2 为基数实现的基数排序也是非常高效的，这可以利用计算机高速的位运算。这里只考察基数排序，而不考察与基数排序相似但更复杂一些的基数交换排序。基数这个词的意思是一个数字系统的基。10 是十进制系统的基数，2 是二进制系统的基数。排序包括分别检测关键字的每一个数字，检测从个位（最低有效位）开始。

1. 根据数据项个位上的值，把所有的数据项分为 10 组。

2. 然后对这 10 组数据项重新排列：把所有关键字是以 0 结尾的数据项排在最前面，然后是关键字结尾是 1 的数据项，照此顺序直到以 9 结尾的数据项。这个步骤被称为第一趟子排序。

3. 在第二趟子排序中，再次把所有的数据项分为 10 组，但是这一次是根据数据项十位上的值来分组的。这次分组不能改变先前的排序顺序。也就是说，第二趟排序之后，从每一组数据项的内部来看，数据项的顺序保持不变；这趟子排序必须是稳定的。

4. 然后再把 10 组数据项重新合并，排在最前面的是十位上为 0 的数据项，然后是 10 位为 1 的数据项，如此排序直到十位上为 9 的数据项。

5. 对剩余位重复这个过程。如果某些数据项的位数少于其他数据项，那么认为它们的高位为 0。

下面是一个例子，有 7 个数据项，每个数据项都有三位。为了清晰起见显示第一位的 0。

```
421 240 035 532 305 430 124                 // unsorted array
(240 430) (421) (532) (124) (035 305)       // sorted on 1s digit
(305) (421 124) (430 532 035) (240)         // sorted on 10s digit
(035) (124) (240) (305) (421 430) (532)     // sorted on 100s digit
035 124 240 305 421 430 532                 // sorted array
```

圆括弧表示组。在同一组中对应位置上的值是相同的。为了证实这种方法是确实可行的，可以在纸上用几个数字自己动手做一做。

设计一个程序

实际应用中，原始数据很可能是来自于一个普通数组。那么 10 组数据项存放在哪里呢？可以利用另外一个数组或者利用一个含有 10 个数组的数组，但又出现了一个问题。个位，十位，百位……每一位数值的个数不可能完全相同，因此很难确定数组的大小。解决这个问题的一种方法是使用 10 个链表，而不用 10 个数组。链表可以根据需要扩展或收缩。这里将采用这个方法。

外层循环依次查看关键字的每一位。有两个内部循环：第一个循环从数组中取得数据项放到链表中；第二个循环则把数据项从链表复制回数组。这里必须使用一种恰当的链表。为了保证子排序的稳定，必须保证数据出链表的顺序与进链表的顺序相同。哪一种链表能够方便地做到这一点呢？具体的代码留作练习。

基数排序的效率

初看起来，基数排序的执行效率似乎好得让人无法相信。所有要做的只是把原始的数据项从数组复制到链表，然后再复制回去。如果有 10 个数据项，则有 20 次复制。对每一位重复一次这个过

程。假设对 5 位的数字排序，就需要 20*5，等于 100 次复制。如果有 100 个数据项，那么就有 200*5 等于 1000 次复制。复制的次数和数据项的个数成正比，即 O(N)，这是我们看到的效率最高的排序算法。

不幸的是，一般是这样：只要数据项越多，就需要越长的关键字。如果数据项增加 10 倍，那么关键字就需要再增加另一位。复制的次数是和数据项的个数与关键字的位数的乘积成正比。位数是关键字值的对数，因此在绝大多数的情况下，算法的执行效率倒退为 O(N*logN)，和快速排序算法相同。

尽管从数字中提取出每一位需要花费时间，但是没有比较。每两次复制需要一次位提取。然而，一台给定的计算机在二进制中的位提取操作快于比较操作。当然，与归并排序类似，基数排序所需要的存储空间是快速排序的二倍。

小 结

- 希尔排序将增量应用到插入排序，然后逐渐缩小增量。
- n-增量排序表示每隔 n 个元素进行排序。
- 被称为间隔序列或者间距序列的数列决定了希尔排序的排序间隔。
- 常用的间隔序列是由递归表达式 h=3*h+1 生成的，h 的初始值为 1。
- 一个容纳了 1000 个数据项的数组，对它进行希尔排序可以是间隔序列为 364，121，40，13，4，最后是 1 的增量排序。
- 希尔排序很难分析，但是它运行的时间复杂度大约为 $O(N*(\log N)^2)$。这比时间复杂度为 $O(N^2)$的排序算法要快，例如比插入排序快，但是比时间复杂度为 O(N*logN)的算法慢，例如比快速排序慢。
- 划分数组就是把数组分为两个子数组，在一组中所有的数据项关键字的值都小于指定的值，而在另一组中所有数据项关键字的值则大于或等于给定值。
- 枢纽是在划分的过程中确定数据项应该放在哪一组的值。小于枢纽的数据项都放在左边一组；而大于枢纽的数据项都放在右边一组。
- 在划分算法中，在各自的 while 循环中的两个数组下标的指针，分别从数组的两端开始，相向移动，查找需要交换的数据项。
- 当一个数组下标指针找到一个需要交换的数据项时，它的 while 循环中止。
- 当两个 while 循环都终止时，交换这两个数据项。
- 当两个 while 循环都终止时，并且两个子数组的下标指针相遇或者交错，则划分过程结束。
- 划分操作有线性的时间复杂度 O(N)，做 N+1 或 N+2 次的比较以及少于 N/2 次的交换。
- 划分算法的内部 while 循环需要额外的检测，以防止数组下标越界。
- 快速排序划分一个数组，然后递归调用自身，对划分得到的两个子数组进行快速排序。
- 只含有一个数据项的子数组定为已经有序，这一点可以作为快速排序算法的基值（终止）条件。
- 快速排序算法划分时的枢纽是一个特定数据项关键字的值，这个数据项称为 pivot（枢纽）。
- 在快速排序的简单版本中，总是由子数组的最右端的数据项作为枢纽。

- 划分的过程中枢纽总是放在被划分子数组的右界，它不包含在划分过程中。
- 划分之后枢纽也换位，被放在两个划分子数组之间。这就是枢纽的最终排序位置。
- 快速排序的简单版本，对已经有序（或者逆序）的数据项排序的执行效率只有 $O(N^2)$。
- 更高级的快速排序版本中，枢纽是子数组中第一个、最后一个以及中间一个数据项的中值。这称为“三数据项取中”（median-of-three）划分。
- 三数据项取中划分有效地解决了对已有序数据项排序时执行效率仅是 $O(N^2)$的问题。
- 在三数据项取中划分中，在对左端、中间以及右端的数据项取中值的同时对它们进行排序。
- 这个排序算法消除了划分算法内部 while 循环中对数据越界的检测。
- 快速排序算法的时间复杂度为 $O(N^* \log_2 N)$（除了用简单的快速排序版本对已有序数据项排序的情况）。
- 子数组小于一定的容量（切割界限，cutoff）时用另一种方法来排序，而不用快速排序。
- 通常用插入排序对小于切割界限的子数组排序。
- 在快速排序已经对大于切割界限的子数组排完序之后，插入排序也可用于整个的数组。
- 基数排序的时间复杂度和快速排序相同，只是它需要两倍的存储空间。

问 题

下列问题可以作为读者的自测题。答案见附录 C。

1. 希尔排序通过什么实现？
   a. 划分数组。
   b. 交换相邻的数据项。
   c. 处理一定间隔的数据项。
   d. 从普通的插入排序开始。
2. 对一个有 100 个数据项的数组，Knuth 算法开始的第一个间隔是_____。
3. 在把插入排序算法转换为希尔排序算法时，下列哪一步是不需要做的？
   a. 把 h 换为 1。
   b. 加入一个创建递减增量的算法。
   c. 在一个循环中包含普通的插入排序。
   d. 在内部循环中改变下标的方向。
4. 判断题：一个好的希尔排序的间隔序列是通过不断地平分数组得到的。
5. 填写大 O 值：希尔排序的执行速度大于_____，但是小于_____。
6. 划分是
   a. 把所有大于给定值的数据项都放到数组的一端。
   b. 平分一个数组。
   c. 对数组的一部分进行排序。
   d. 分别对每一半数组排序。
7. 在划分的时候，每个数组数据项和_____比较。

8. 在划分的时候，如果一个数据项和第 7 题的结果相等，那么
   a. 忽略该数据项。
   b. 是否忽略该数据项，由另一个数组数据项决定。
   c. 把该数据项放在枢纽的位置上。
   d. 交换。
9. 判断题：在快速排序中，枢纽可以是数组中任意的数据项。
10. 假设较大的关键字在右边，则划分是
   a. 左右子数组之间的数据项。
   b. 左右子数组之间的数据项的关键字的值。
   c. 右边子数组的左端数据项。
   d. 右边子数组的左端数据项关键字值。
11. 快速排序划分原始数组，然后_____。
12. 在简单版本的快速排序中，划分之后，枢纽可能
   a. 用于找数组的中值。
   b. 和右边子数组的一个数据项交换位置。
   c. 用作下一次划分的开始点。
   d. 被删除。
13. 三数据项取中划分是一种选择_____的方法。
14. 快速排序中，对于有 N 个数据项的数组，partitionIt()方法需要对每个数据项检查大约_____次。
15. 判断题：在划分区域的大小为 5 时停止划分，使用不同的排序方法完成排序，这样可以提高排序的速度。

实　验

做这些实验题可以帮助理解本章的主题。不需要编程实现。

1. 观察在 Partition 专题 applet 中运行 100 个逆序排列的竖条的情况。它的结果是基本有序的吗？

2. 修改 shellSort.java 程序（清单 7.1），使它在完成每一趟 n-增量排序后显示数组的完整内容。数组足够小，可以只显示在一行中。分析这些中间步骤，看看算法的执行是否和分析的一样。

3. 修改 shellSort.java 程序（清单 7.1）和 quickSort3.java 程序（清单 7.5），对大一些的数组排序，并且比较它们的速度。同时，比较它们和第 3 章中排序算法的速度。

编程作业

做上机作业编程有助于巩固理解本章内容，并演示本章的概念如何应用。（在“简介”中提到过，有资格证明的教师可以从出版者的网站上得到上机作业的完整答案。）

7.1　修改 partition.java 程序（清单 7.2），使 partitionIt()方法总是用具有最大的下标值的数组（最

右）数据项作为枢纽，而不是用任意一个数据项。（这和清单 7.3 的 quickSort1.java 程序相似。）确保程序对三个或少于三个数据项的数组也能执行。为了达到这个目的，需要增加一些额外的语句。

7.2 修改 quickSort2.java 程序（清单 7.4）以计算排序中复制和比较的次数，然后显示总数。这个程序应该和 QuickSort2 专题 applet 的执行性能相同，因此对逆序排列数据的复制和比较次数是一致的。（记住一次交换是三次复制。）

7.3 在第 3 章练习题 3.2 中，提示可以通过排序数据和挑选中间数据项来求一个数据集的中心值。读者可能会想，使用快速排序然后选择一个中间的数据项是找中心值最快的方法，但是还有一个更快的方法。利用划分算法求中心值，而不必对所有的数据项排序。

为了理解它是如何实现的，假设在划分数据时，偶然发现枢纽最后停在数组中间的位置上。工作就完成了！枢纽右边的所有数据项都大于（或等于）枢纽，而所有左边的数据项都小于（或等于）枢纽，所以如果枢纽停在数组正中间的位置上，那么它就是中心值。枢纽通常是不会停在数组中间位置上的，但是可以通过再划分包含了中间位置数据项的分组来找到它。

假设数组有 7 个数据项分布在数组的关键字从 0 到 6 的单元中。中间位置的数据项的下标为 3。如果划分这个数组，并且枢纽停在下标为 4 的位置上，那么需要对下标为 0 到 4 的数据项重新划分（这个划分包含了下标为 3 的数据项），而不包括下标为 5 到 6 的数据项。如果枢纽停在下标为 2 的位置，就需再划分下标从 2 到 4 的数据项，而不包含下标 0 到 1 的数据项。继续递归地划分对应的分区，总是检查枢纽是否在中间的位置。最后，枢纽会停在中间位置上的，程序也就结束。因为所需要的划分次数比快速排序少，所以这个算法更快。

扩展编程作业 7.1，来查找一个数组的中心值数据项。可以使用类似快速排序的递归调用，但是只用来划分子数组，而不是对整个数组排序。当找到中心值数据项时这个过程就停止，而不必等数组排序完成之后才停止。

7.4 选择的意思是从一个数组中找出第 k 大或者第 k 小的数据项。例如，要选择第 7 大的数据项。找中心值（如编程作业 7.2）是选择的一种特殊情况。可以使用同样的划分过程，但不是找中间位置的数据项，而是找一个指定下标的数据项。修改编程作业 7.2 中的程序，允许选择任意一个指定数据项。你的程序能处理多小的数组呢？

7.5 实现本章最后一节所讲的基数排序算法。它应该能处理不同数据量的数据项，以及关键字位数不同的数据。同时，也应该能够实现不同基数的排序（可以是 10 以外的其他数），但是，除非编写的程序能显示不同的基数值，否则会很难看清运行的情况。

第8章
二　叉　树

本章重点
- 为什么使用二叉树？
- 树的术语
- 类比
- 二叉搜索树如何工作？
- 查找节点
- 插入节点
- 遍历树
- 查找最大值和最小值
- 删除节点
- 二叉树的效率
- 用数组表示树
- 重复关键字
- 完整的 tree.java 程序
- 哈夫曼编码

在这一章，我们要从第 7 章“高级排序”的算法中跳出来，把注意力放到数据结构上去。二叉树是编写程序中使用的一种基本的数据存储结构。它的优势是前面介绍过的数据结构所没有的。在这一章中将学习树的用途、运行机制以及创建树的方法。

为什么使用二叉树？

为什么要用到树呢？因为它通常结合了另外两种数据结构的优点：一种是有序数组，另一种是链表。在树中查找数据项的速度和在有序数组中查找一样快，并且插入数据项和删除数据项的速度也和链表一样。下面，我们先来稍微思考一下这些话题，然后再深入地研究树的细节。

在有序数组中插入数据项太慢

假设数组中的所有数据项都有序的排列——这就是有序数组，和第 2 章“数组”中讲过的一样。本书已经讲过，用二分查找法可以在有序数组中快速地查找特定的值。它的过程是先查看数组的正中间的数据项，如果那个数据项值比要找的大，就缩小查找范围，在数组的后半段中找；如果小，就在前半段找。反复这个过程，查找数据所需的时间是 O(logN)。同时也可以迅速地遍历有序数组，按顺序访问每个数据项。

然而，想在有序数组中插入一个新数据项，就必须首先查找新数据项插入的位置，然后把所有比新数据项大的数据项向后移动一位，来给新数据项腾出空间。这样多次的移动很费时，平均来讲要移动数组中一半的数据项（N/2 次移动）。删除数据项也需要多次的移动，所以也很慢。

显而易见，如果要做很多的插入和删除操作，就不该选用有序数组。

在链表中查找太慢

另一方面，在第 5 章“链表”中，链表的插入和删除操作都很快。它们只需要改变一些引用的值就行了。这些操作的时间复杂度是 O(1)（是大 O 表示法中最小的时间复杂度）。

但是遗憾的是，在链表中查找数据项可不那么容易。查找必须从头开始，依次访问链表中的每一个数据项，直到找到该数据项为止。因此，平均需要访问 N/2 个数据项，把每个数据项的值和要找的数据项做比较。这个过程很慢，费时 O(N) (注意，对排序来说比较快的，对数据结构操作来说是比较慢的。)。

不难想到可以通过有序的链表来加快查找速度，链表中的数据项是有序的，但这样做是没有用的。即使是有序的链表还是必须从头开始依次访问数据项，因为链表中不能直接访问某个数据项，必须通过数据项间的链式引用才可以。（当然有序链表访问节点还是比无序链表快多了，但查找任意的数据项时它也无能为力了。）

用树解决问题

要是能有一种数据结构，既能像链表那样快速的插入和删除，又能像有序数组那样快速查找，那样就好了。树实现了这些特点，成为最有意思的数据结构之一。

树是什么？

本章中着重讲解的是一种特殊的树——二叉树，但在讲解二叉树之前，还是先从广义上讨论一下树。

树由边连接的节点而构成。图 8.1 就显示了一棵树。在图中（或者专题 applet 中）用圆代表节点，连接圆的直线代表边。

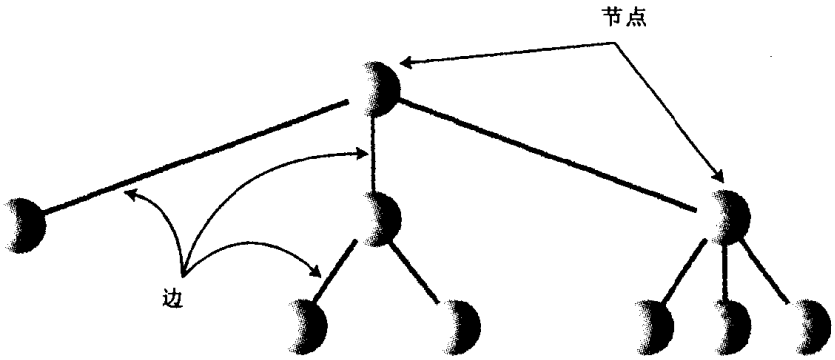


图 8.1 一棵普通的（非二叉）树

人们把树作为抽象的数学实体来广泛地研究，因此有大量的关于树的理论知识。其实树是范畴更广的图的特例，不过这里先不考虑图。在第 13 章“图”和第 14 章“带权图”将会讨论图。

计算机程序里，节点一般代表着那些实体，诸如人、汽车零件、预定的机票等等——这也就是其他常用的数据结构中存储的那些数据项。像 Java 这样的面向对象编程语言中，真实世界的实体常常用对象来表示。

节点间的直线（边）表示关联节点间的路径。一般，用边表示关联是方便的：因为如果节点间有边作为关联，对程序来说从一个节点到另一个节点是很容易的（也很快）。实际上，从一个节点到另一个节点的惟一方法是沿着一条顺着有边的道路前进。一般规定顺着边的一个方向走：即从根向下的方向。

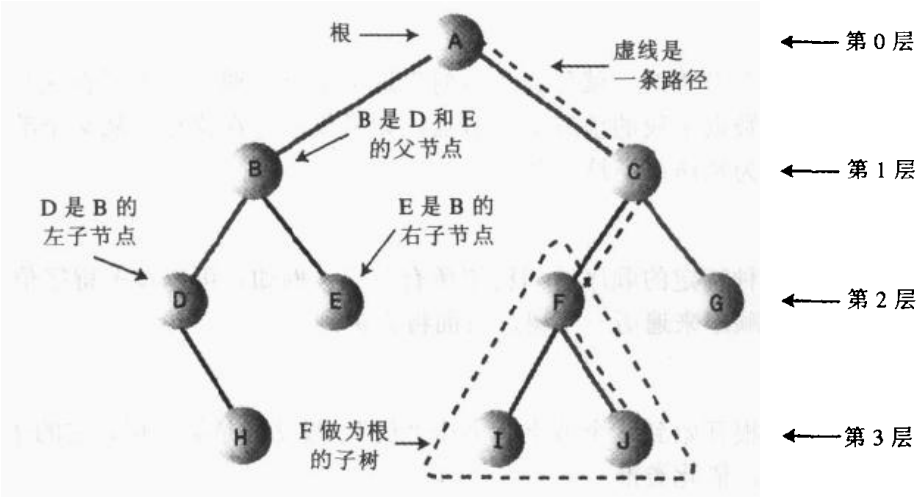
Java 语言编写的程序中常常用引用来表示边（或者 C/C++程序中可能会用指针）。

在树的顶层总是只有一个节点，它通过边连接到第二层的多个节点，然后第二层节点连向第三层更多的节点，依此类推。所以树的顶部小，底部大。这和真的树相比好像是颠倒过来了，但程序一般都从树的小的那端开始执行一种操作，而且（可以论证）从顶到底来思考问题更自然，就像人们阅读文字那样。

树有很多种。图 8.1 显示了每个节点有多于两个的子节点的树。（马上可以看到“子节点”是什么意思。）不过，这章里将讨论的是一种特殊的树——二叉树。二叉树的每个节点最多有两个子节点。更普通的树里，节点的子节点可以多于两个，这种树称为多路树。在第 10 章“2-3-4 树和外部存储”中可以看到多路树的例子。

树的术语

很多术语都可用来描述树的各个部分。为了让介绍的内容更容易理解，需要知道一些树的术语。幸运地是大部分术语都与真实世界的树相关，或者和家庭关系有关（如父节点和子节点），所以它们记忆起来一点也不难。图 8.2 展示了很多用于二叉树的一些树的术语。



H,E,I,J 和 G 是叶节点

图 8.2　树的术语

路径

设想一下顺着连接节点的边从一个节点走到另一个节点，所经过的节点的顺序排列就称为“路径”。

根

树顶端的节点称为“根”。一棵树只有一个根。如果要把一个节点和边的集合定义为树，那么从根到其他任何一个节点都必须有一条（而且只有一条）路径。图 8.3 展示的就不是树。可以看到它违反了上述这条规则。

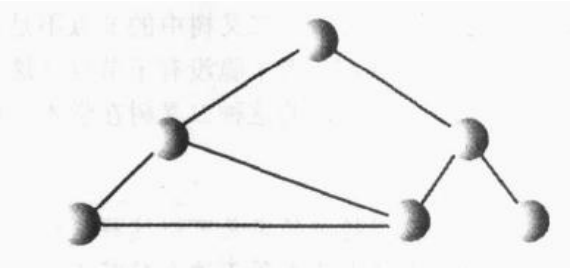


图 8.3　不是树

父节点

每个节点（除了根）都恰好有一条边向上连接到另一个节点，上面的这个节点就称为下面节点的“父节点”。

子节点

每个节点都可能有一条或多条边向下连接其他节点，下面的这些节点就称为它的“子节点”。

叶节点

没有子节点的节点称为“叶子节点”或简称“叶节点”。树中只有一个根，但是可以有很多叶节点。

子树

每个节点都可以作为“子树”的根，它和它所有的子节点，子节点的子节点等都含在子树中。就像家族中那样，一个节点的子树包含它所有的子孙。

访问

当程序控制流程到达某个节点时，就称为“访问”这个节点，通常是为了在这个节点处执行某种操作，例如查看节点某个数据字段的值或显示节点。如果仅仅是在路径上从某个节点到另一个节点时经过了一个节点，不认为是访问了这个节点。

遍历

遍历树意味着要遵循某种特定的顺序访问树中所有节点。例如，可以按关键字值的升序访问所有的节点。当然还有其他的顺序来遍历一棵树，后面将会讲到。

层

一个节点的层数是指从根开始到这个节点有多少“代”。假设根是第0层，它的子节点就是第1层，它的孙节点就是第2层，依此类推。

关键字

可以看到，对象中通常会有一个数据域被指定为关键字值。这个值常用于查询或其他的操作。在树的图形中，如果用圆表示保存数据项的节点，那么一般将这个数据项的关键字值显示在这个圆中。（后面会看到许多图，图中的节点有关键字。）

二叉树

如果树中每个节点最多只能有两个子节点，这样的树就称为“二叉树”。本章将会着重讲解二叉树，因为它是最简单的，也是最常用的。

二叉树每个节点的两个子节点称为“左子节点”和“右子节点”，分别对应于树图形中它们的位置，如图8.2所示。二叉树中的节点不是必须有两个子节点；它可以只有一个左子节点，或者只有一个右子节点，或者干脆没有子节点（这种情况下它就是叶节点）。

下面我们要学习的这种二叉树在学术上称为二叉搜索树。图8.4显示了一棵二叉搜索树。

注意

二叉搜索树特征的定义可以这样说：一个节点的左子节点的关键字值小于这个节点，右子节点的关键字值大于或等于这个父节点。

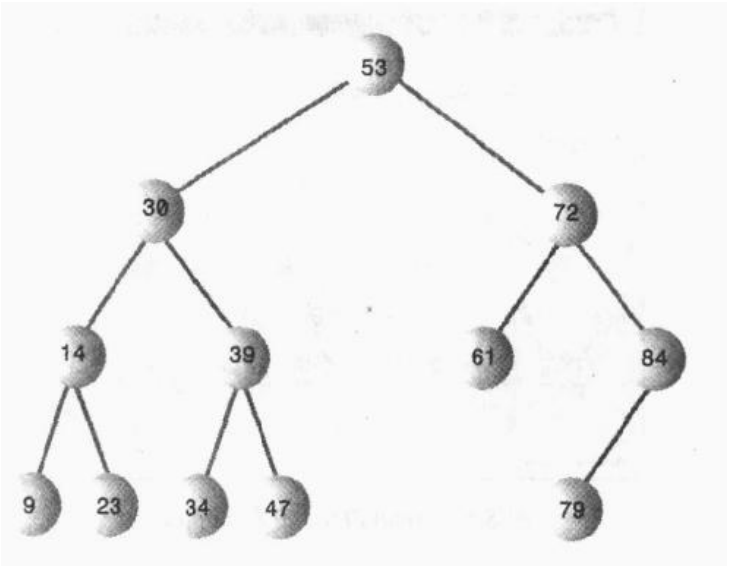


图 8.4 二叉搜索树

一个类比

计算机系统中，人们常遇到的树是分级文件结构。给定设备的根目录（在许多系统中，通过反斜线符号指定，如 C:\）是树的根。根目录下面的一层目录是根的子节点。子目录有许多层。文件代表叶节点；它们没有自己的子节点。

显然，分级文件结构不是二叉树，因为一个目录下可以有很多子节点。一个完整的路径名称，如 C:\SALES\EAST\NOVEMEBER\SMITH.DAT，对应着从根到 SMITH.DAT 叶的路径。应用于文件结构的术语，如根和路径等，是从树的理论中借来的。

分级文件结构和本章下面要讨论的树有明显的不同。文件结构中，子目录中不含有数据；它们只有其他子目录或文件的引用。只有文件中包含数据。而在树中，每个节点都包含数据（员工记录、汽车零件说明等等）。而且除了包含数据，除了叶子节点之外，每个节点还包含指向其他节点的引用。

二叉搜索树如何工作

下面来看看对于一个给定关键字的节点，在一棵普通的二叉树中怎么查找一个节点，怎么插入新节点，怎么遍历树，以及怎么删除节点。先来看看如何用 BinaryTree 专题 applet 来运行这些操作，之后再看对应的 Java 代码。

BinaryTree 专题 applet

启动 BinaryTree 专题 applet。屏幕显示的和图 8.5 差不多。不过，树在专题 applet 中是随机生成的，因此它不会和图中的树完全一样。

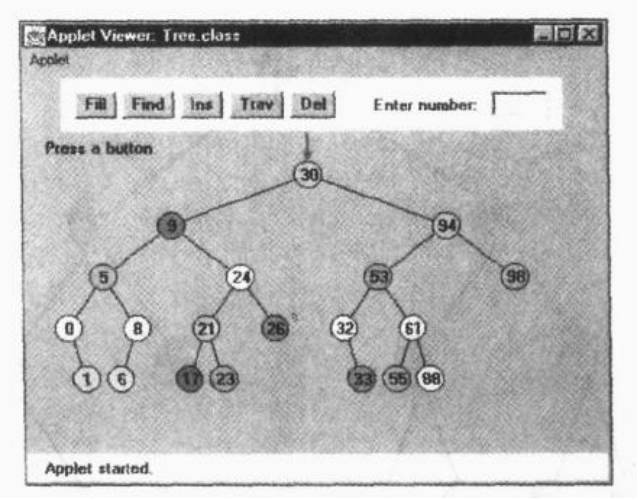


图 8.5 BinaryTree 专题 applet

应用专题 Applet

节点显示的关键字值是从 0 到 99 的数据。当然，在实际的树里可能会是更大的取值范围。例如，用员工的社会安全号码当作关键值时，它的取值范围的上界会到 999999999。

专题 applet 和实际的树之间的另一个区别是专题 applet 限制树的深度为 5；这就是说，从树根到树的底部最多有 5 层。这个限制保证了所有的节点都可以在屏幕上显示出来。而实际树的层数是无限的（直到溢出内存为止）。

应用专题 applet，可以随时创建一棵新树。点击 Fill 按钮创建新树。提示窗口会要求用户输入树的节点个数。个数的取值范围是 1 到 31，取值为 15 时会得到一棵很有代表性的树。输入个数之后，再点击 Fill 按钮几次来生成新树。试验创建有不同节点数的几棵树。

非平衡树

注意有些树是非平衡的；这就是说，它们大部分的节点在根的一边或者是另一边，如图 8.6 所示。个别的子树也可能是非平衡的。

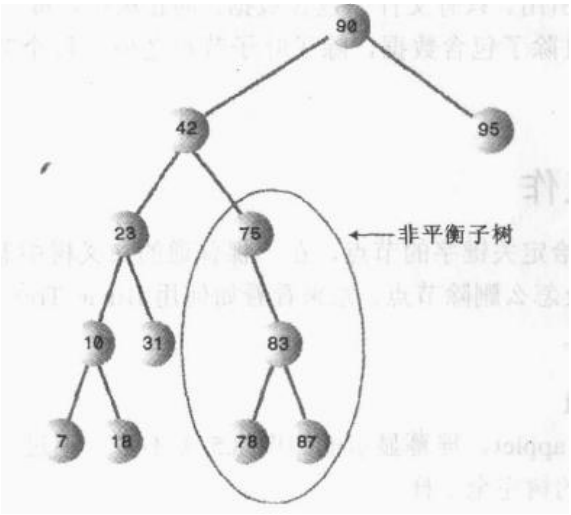


图 8.6 非平衡树（有一个非平衡子树）

树变得不平衡是由数据项插入的顺序造成的。如果关键字值是随机插入的，树会或多或少更平衡一点。但是，如果插入序列是升序（像 11，18，33，42，65，等等）或是降序，则所有的值都是右子节点（升序时）或者都是左子节点（降序时），这样树就会不平衡了。专题 apple 中关键字值是随机产生的，但无论如何还是会有些短的升序或降序数列出现，这就会使树的局部不平衡。在学习了如何用专题 applet 插入数据项之后，可以尝试着按这种有序序列插入数据项，看看会发生什么情况。

用 Fill 按钮建立树时，如果输入的节点数很大，有可能会得不到输入的那么多节点的树。这取决于树的不平衡的程度，所以有些分支可能不能容纳它的全部节点。这是因为专题 applet 中树的深度被限制为 5；实际的树中不会出现这样的问题。

如果树中关键字值的输入顺序是随机的，这样建立的较大的树，它的不平衡问题可能不会很严重，因为很长一列随机数值有序的机会是很小的。不过关键值也可以按一个严格的顺序输入；例如，负责数据录入的人员在输入数据之前，将一堆员工档案按员工号递增的顺序排列。如果这样做，那么树的效率就会严重退化。第 9 章中将会讨论不平衡的树以及该如何处理它们。

用 Java 代码表示树

下面来看看如何用 Java 语言实现二叉树。像其他数据结构一样，有很多方法可以在计算机内存中表示树。最常用的方法是把节点存在无关联的存储器中，而通过每个节点中指向自己子节点的引用来连接。

还可以在内存中用数组表示树，用存储在数组中相对的位置来表示节点在树中的位置。本章最后将讨论这种方法。实例中的 Java 代码采用的是用引用连接节点的方法。

注意：

下面分别讨论树的操作，会显示对应操作的代码片段。这些片段都是从本章最后的清单 8.1 中抽取出来的，清单 8.1 中是完整的程序。

Node 类

首先，需要有一个节点对象的类。这些对象包含数据，数据代表要存储的内容（例如，在员工数据库中的员工记录），而且还有指向节点的两个子节点的引用。下面就是这个类的描述语句：

```
class Node
   {
   int iData;                    // data used as key value
   double fData;                 // other data
   node leftChild;               // this node's left child
   node rightChild;              // this node's right child

   public void displayNode()
      {
      // (see Listing 8.1 for method body)
      }
   }
```

有些程序员也把节点的父节点的引用包括在 Node 类中。这样做会使一些操作简化，但使一些

别的操作复杂，所以这里不使用它。这个类中还有一个叫 displayNode()的方法，用它来显示节点数据，不过在这里没有写出它的代码。

还可以用其他方法来设计 Node 类，可以用引用，指向一个代表数据项的对象，而不是把数据项直接放到节点中：

```
class Node
   {
   person p1;                    // reference to person object
   node leftChild;               // this node's left child
   node rightChild;              // this node's right child
   }

class person
   {
   int iData;
   double fData;
   }
```

这种方法使节点类的概念更清楚，节点类和数据项包含的不是同样的东西，但这样做代码更复杂，所以本章还是沿用第一种方法。

Tree 类

还需要有一个表示树本身的类，由这个类实例化的对象含有所有的节点，这个类是 Tree 类。它只有一个数据字段：一个表示根的 Node 变量。它不需要包含其他节点的数据字段，因为其他节点都可以从根开始访问到。

Tree 类有很多方法。它们用来查询、插入和删除节点；进行各种不同的遍历；显示树。下面是这个类的骨架：

```
class Tree
   {
   private Node root;            // the only data field in Tree

   public void find(int key)
      {
      }
   public void insert(int id, double dd)
      {
      }
   public void delete(int id)
      {
      }
   // various other methods
   }  // end class Tree
```

TreeApp 类

最后，还需要有一个操作树的类。这里展示的是如何写一个有 main()方法的类来创建树，插入

节点和查询节点。这个类称为 TreeApp 类：

```
class TreeApp
   {
   public static void main(String[] args)
      {
      Tree theTree = new Tree;        // make a tree

      theTree.insert(50, 1.5);        // insert 3 nodes
      theTree.insert(25, 1.7);
      theTree.insert(75, 1.9);

      node found = theTree.find(25);  // find node with key 25
      if(found != null)
      System.out.println("Found the node with key 25");
   else
      System.out.println("Could not find node with key 25");
   }  // end main()
}  // end class TreeApp
```

提示：

在清单 8.1 中，main()还提供了简单的用户接口，用户可以用键盘控制来选择插入、查询、删除或是其他的操作。

下面逐个介绍树的操作：查找节点、插入节点、遍历树和删除节点。

查找节点

根据关键值查找节点是树的主要操作中最简单的，所以本章从查找开始学习。

回忆一下二叉搜索树中节点对应着含有信息的对象。它们可能是 person 对象，用员工号码作为关键字，也可能是用姓名、地址、电话号码、工资或其他字段。节点也可能代表汽车零件，用零件编号作为关键字，或者是手边的数量，价格等其他字段。不过，在专题 applet 中看到的每个节点只有两个特征：数字和颜色。节点在创建的时候得到这两个特征，并在生命周期内保持它们。

使用专题 applet 查找一个节点

专题 applet 里，选择一个节点，最好是选择靠近树底部的节点（尽可能的离根远一点）。这个节点里显示的数字就是它的关键字值。下面演示在专题 applet 中是如何查找给定关键字值的节点的。

为了讨论方便，假设要找的节点的关键字值是 57，像图 8.7 中的那样。当然，运行专题 applet 的时候，可能会得到不同的树，就需要选择其他不同关键字值的节点来查找。

点击 Find 按钮。提示框会请求输入要查找节点的值。输入 57（或选择的其他节点的数字）。再次点击 Find 按钮。

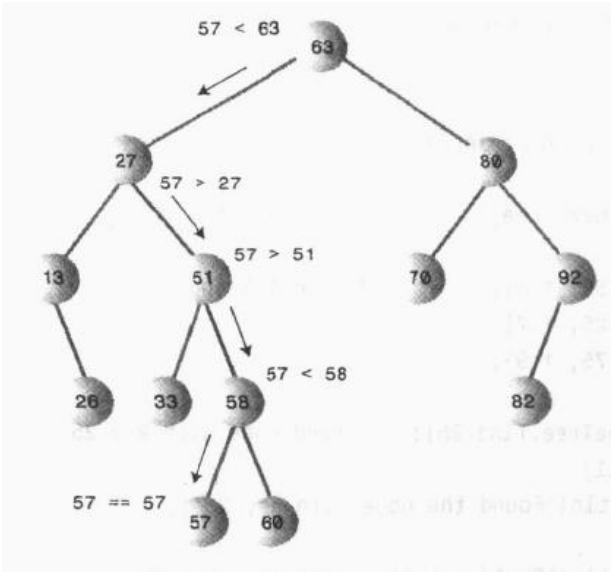


图 8.7 查找节点 57

在专题 applet 查找那个节点的时候，提示框会显示出“Going to left child”或者“Going to right child”，红色箭头就会向左或向右下移一层。

在图 8.7 中箭头从根开始。程序将关键字值 57 和根节点关键字值 63 相比较。57 小，因此程序知道要找的节点肯定在树的左部分中——可能是根的左子节点，或者是左子节点的子孙节点之一。根的左子节点的关键字值是 27，所以比较 27 和 57 后知道要找的节点在 27 的右子树中。箭头转向 51，这棵子树的根。这里，57 还是比 51 大，所以向右，到 58，然后向左，到 57。这次比较显示 57 和这个节点的关键字值相等，因此就找到了要找的节点。

专题 applet 在找到节点后不做任何操作，除了显示一条信息说明节点找到了。实际的程序一般会在找到节点后做一些操作，比如显示它的内容或改变它的一个字段等等。

查找节点的 Java 代码

下面是 find()例程的代码，它是 Tree 类中的一个方法：

```
public Node find(int key)       // find node with given key
   {                            // (assumes non-empty tree)
   Node current = root;                 // start at root
while(current.iData != key)             // while no match,
   {
   if(key < current.iData)              // go left?
      current = current.leftChild;
   else
      current = current.rightChild; // or go right?
   if(current == null)                  // if no child,
      return null;                      // didn't find it
   }
return current;                         // found it
}
```

这个过程用变量 current 来保存正在查看的节点。参数 key 是要找的值。查找从根开始。（它只能这样做，因为只有根节点可以直接访问。）因此，开始把 current 设为根。

之后，在 while 循环中，将要查找的值，key 与当前节点的 iData 字段的值（关键字字段）做比较。如果 key 小于这个数据域的值，current 设为此节点的左子节点。如果 key 大于（或等于）节点 iData 字段的值，current 设为此节点的右子节点。

找不到节点

如果 current 等于 null，在查找序列中找不到下一个子节点；到达序列的末端而没有找到要找的节点，表明了它不存在。返回 null 来指出这个情况。

找到节点

如果 while 循环的条件不满足，从循环的末端退出来，current 的 iData 字段和 key 相等：这就是说找到该节点了。返回这个节点，调用 find()方法的程序可以访问节点的任何字段。

树的效率

像上面看到的那样，查找节点的时间取决于这个节点所在的层数。在专题 applet 中，最多有 31 个节点，不超过 5 层——因此最多只需要 5 次比较，就可以找到任何节点。它的时间复杂度是 O(logN)，更精确地说是 $O(\log_2 N)$，以 2 为底的对数。本章最后将更深入的讨论这个问题。

插入一个节点

要插入节点，必须先找到插入的地方。这很像要找一个不存在的节点的过程，如前面说的找不到节点的那种情况。从根开始查找一个相应的节点，它将是新节点的父节点。当父节点找到了，新的节点就可以连接到它的左子节点或右子节点处，这取决于新节点的值是比父节点的值大还是小。

使用专题 applet 插入一个节点

用专题 applet 插入一个新节点，点击 Ins 按钮。提示框要求输入要插入节点的关键字值。假设要插入关键字值是 45 的节点，在文本框中输入这个数字。

程序插入节点的第一步是找到它应该插入的位置。图 8.8a 展示了这步是如何做的。

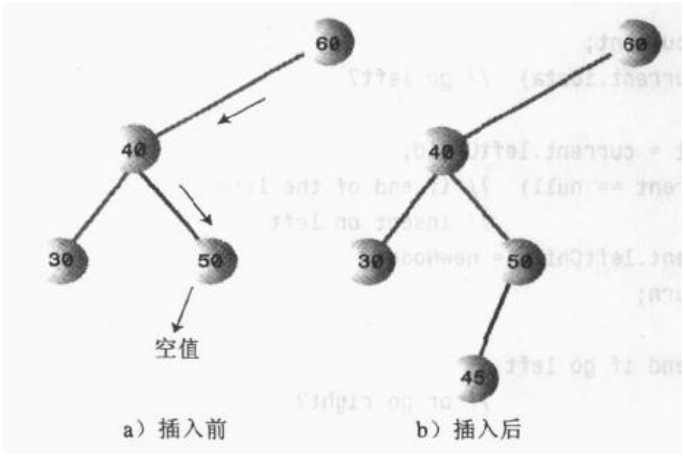


图 8.8 插入一个节点

45 比 60 小但比 40 大，所以转到节点 50。现在要向左，因为 45 比 50 小，但 50 没有左子节点；它的 leftChild 字段等于 null。找到 null 后，插入程序就找到新节点要接入的位置了。专题 applet 建立一个值为 45 的新节点（随机选择一种颜色）并把它连接到 50 的左子节点处，如图 8.8b 所示。

插入一个节点的 Java 代码

insert()方法从创建新节点开始，用提供的数据作为参数。

下一步，insert()必须先确定新节点插入的位置。这段代码与查找节点的代码大致相同，“查找一个节点的 Java 代码”这节中已经介绍过。区别是原来简单的查找节点时，遇到 null（不存在）值后，表明要找的节点不存在，于是就立即返回了。现在要插入一个节点，就在返回前插入（要是必要的话，要先创建）节点。

要找的值是从参数 id 传入的。while 循环用 true 作为它的条件，因为它不关心是否会遇到和 id 值一样的节点；它把那个有相同关键字值的节点当作关键字值比较大的节点处理。（本章后面将会讨论有重复节点的问题）。

插入节点的位置总会找到的（除非存储器溢出）；找到后，新节点被接到树上，while 循环从 return 处跳出。

下面是 insert()方法的代码：

```
public void insert(int id, double dd)
   {
   Node newNode = new Node();     // make new node
   newNode.iData = id;            // insert data
   newNode.dData = dd;
   if(root==null)                 // no node in root
      root = newNode;
   else                           // root occupied
      {
      Node current = root;        // start at root
      Node parent;
      while(true)                 // (exits internally)
         {
         parent = current;
         if(id < current.iData)   // go left?
            {
            current = current.leftChild;
            if(current == null)   // if end of the line,
               {                  // insert on left
               parent.leftChild = newNode;
               return;
               }
            } // end if go left
         else                     // or go right?
            {
             current = current.rightChild;
             if(current == null)  // if end of the line
```

```
                {                   // insert on right
                parent.rightChild = newNode;
                return;
                }
            }  // end else go right
        }  // end while
    }  // end else not root
}  // end insert()
// --------------------------------------------------------
```

这里用一个新的变量 parent（curent 的父节点），来存储遇到的最后一个不是 null 的节点（图 8.8 中的 50）。必须这样做，因为 current 在查找的过程中会变成 null，才能发现它查过的上一个节点没有一个对应的子节点。如果不存储 parent，就会失去插入新节点的位置。

为了插入新节点，把 parent（遇到的最后一个非 null 节点）中对应的子指针指向新节点。如果没找到 parent 的左子节点，就把新节点接到 parent 的左子节点处；没找到右子节点，就把新节点接到右子节点处。图 8.8 中，45 就接到 50 的左子节点处。

遍历树

遍历树的意思是根据一种特定顺序访问树的每一个节点。这个过程不如查找、插入和删除节点常用，其中一个原因是因为遍历的速度不是特别快。不过遍历树在某些情况下是有用的，而且在理论上很有意义。（遍历比删除简单，本章会尽量把删除操作放在后面讨论。）

有三种简单的方法遍历树。它们是：前序（preorder）、中序（inorder）、后序（postorder）。二叉搜索树最常用的遍历方法是中序遍历，所以先来看看中序遍历，再简要地学习其他那两种遍历方法。

中序遍历

中序遍历二叉搜索树会使所有的节点按关键字值升序被访问到。如果希望在二叉树中创建有序的数据序列，这是一种方法。

遍历树的最简单方法是用递归的方法（第 6 章中介绍过）。用递归的方法遍历整棵树要用一个节点作为参数。初始化时这个节点是根。这个方法只需要做三件事：

1. 调用自身来遍历节点的左子树。
2. 访问这个节点。
3. 调用自身来遍历节点的右子树。

记住访问一个节点意味着对这个节点做某种操作：显示节点，把节点写入文件，或其他别的操作。

遍历可以应用于任何二叉树，而不只是二叉搜索树。这个遍历的原理不关心节点的关键字值；它只是看这个节点是否有子节点。

遍历的 Java 代码

中序遍历实际的代码非常简单，在展示专题 applet 中树的遍历之前先来看看它的代码。inOrder()

例程中执行上面说的三个步骤。访问节点这个步骤把节点的内容显示出来。和其他递归程序一样，它必须有一个基值情况——在此情况让程序立即返回的条件，而不会再调用自身。在 inOrder()方法中，作为参数传入的节点等于 null 时，就是终止条件。下面就是 inOrder()方法的代码：

```
private void inOrder(node localRoot)
   {
   if(localRoot != null)
      {
      inOrder(localRoot.leftChild);

      System.out.print(localRoot.iData + " ");
      inOrder(localRoot.rightChild);
      }
   }
```

开始时用根作为参数调用这个方法：

```
inOrder(root);
```

之后，它就靠自己递归调用自己，直到所有节点都被访问过为止。

遍历一棵三节点树

下面来看一个简单的例子，以理解递归遍历例程的过程。假设遍历一棵只有三个节点的树：一个根节点（A），左子节点（B）和右子节点（C），如图 8.9 所示。

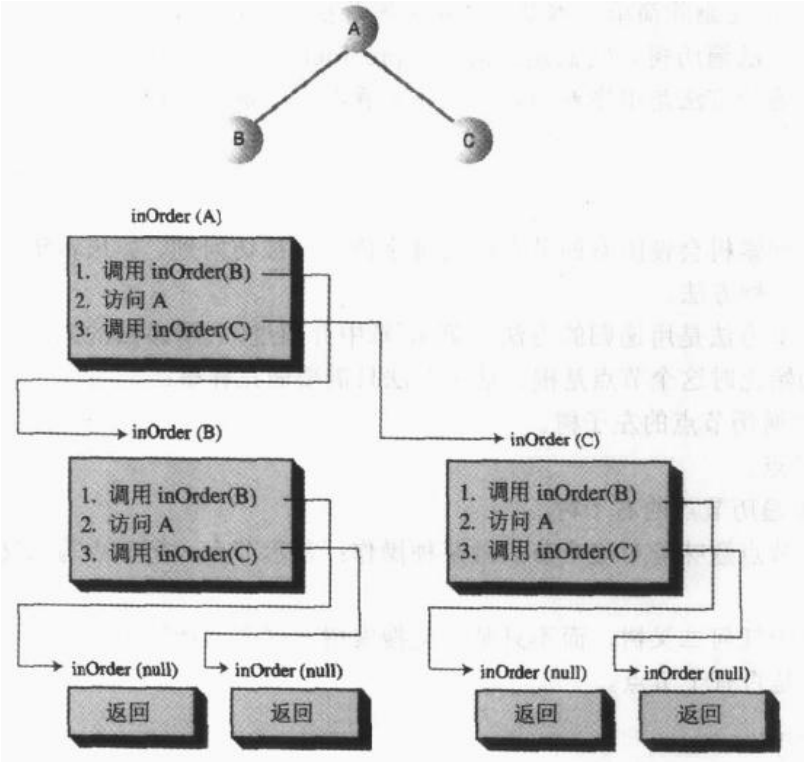


图 8.9　用 inOrder()方法遍历一棵三个节点树

首先，把根 A 作为参数来调用 inOrder()方法。这个调用的具体表示是 inOrder(A)。inOrder(A)首先用自己的左子节点 B 作为参数调用 inOrder()方法。这第二步调用用 inOrder(B)表示。

inOrder(B)同样先把自己的左子节点作为参数调用 inOrder()方法。但是，B 没有左子节点，所以参数就是 null。这个调用可以写为 inOrder(null)。现在，inOrder()方法存在着三个实例：inOrder(A)、inOrder(B)和 inOrder(null)。但是，inOrder(null)发现参数是 null 时会立即返回。（肯定都会出现这种情况。）

现在，inOrder(B)继续访问节点 B；假设这里就是把 B 节点显示出来。接着 inOrder(B)又调用 inOrder()，把它的右子节点作为参数。这次参数还是 null，所以第二次 inOrder(null)还是立即返回了。现在 inOrder(B)已经执行了步骤 1、步骤 2 和步骤 3，所以它可以返回了（并且不再存在了）。

然后回到 inOrder(A)，从遍历 A 的左子节点处返回。访问 A。然后再次调用 inOrder()，把 C 作为参数，创建 inOrder(C)。像 inOrder(B)一样，inOrder(C)没有子节点，所以步骤 1 什么也不做就返回了，步骤 2 访问 C，步骤 3 同样什么也不做就返回。现在，inOrder(C)返回到 inOrder(A)。

现在 inOrder(A)已经完成了，所以它返回，整个遍历就结束了。访问节点的顺序是：B，A，C；它们是被中序访问的。

可以用同样的方法来处理更复杂的树。每个节点处 inOrder()方法都调用自己，直到完成整棵树的遍历。

用专题 applet 遍历

为了看到专题 applet 中遍历是怎样的，需要反复地点击 Trav 按钮。（不需要输入任何数字。）

图 8.10 展示了专题 applet 中序遍历树时的过程。这比前面看到的三节点树略微复杂一点。红色箭头从根开始。表 8.1 列出了访问关键字值的顺序和显示的相应消息。关键字序列显示在专题 Applet 窗口的底部。

表 8.1 专题 applet 的遍历过程

步骤序号	红箭头所指节点	消息	访问过的节点序列
1	50(root)	Will check left child	
2	30	Will check left child	
3	20	Will check left child	
4	20	Will visit this node	
5	20	Will check right child	20
6	20	Will go to root of previous subtree	20
7	30	Will visit this node	20
8	30	Will check right child	20 30
9	40	Will check left child	20 30
10	40	Will visit this node	20 30
11	40	Will check right child	20 30 40
12	40	Will go to root of previous subtree	20 30 40
13	50	Will visit this node	20 30 40
14	50	Will check right child	20 30 40 50
15	60	Will check left child	20 30 40 50
16	60	Will visit this node	20 30 40 50
17	60	Will check right child	20 30 40 50 60
18	60	Will go to root of previous subtree	20 30 40 50 60
19	50	Done traversal	20 30 40 50 60

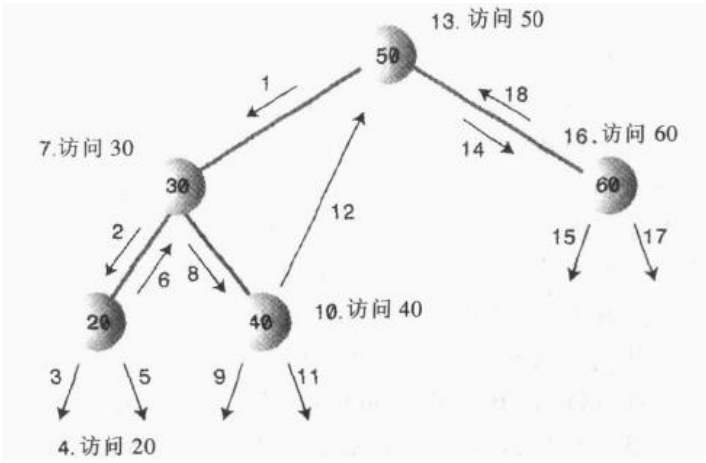


图 8.10 中序遍历树

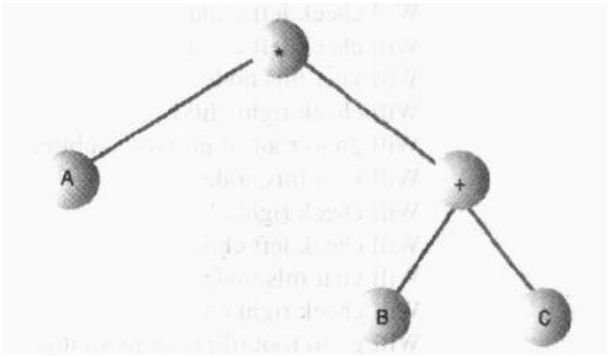
这个遍历的过程可能不那么明显的，但是对每个节点都是先遍历该节点的左子树，访问节点本身，再遍历节点的右子树。例如，对节点 30 就是这样，见表中第 2 步、第 7 步和第 8 步。

遍历算法不像看起来那么复杂。弄清它的最好方法就是用专题 applet 遍历多种不同的树，这样就会非常清楚遍历的过程了。

前序和后序遍历

除了中序之外，还有两种遍历方法；它们是前序和后序。要中序遍历一棵树的原因是很清楚的，但要通过前序或后序来遍历树的目的就不是那么清楚了。不过，如果要编写程序来解析或分析代数表达式，这两种遍历方法就很有用了。下面来看看为什么是这样的。

二叉树（不是二叉搜索树）可以用于表示包括二元运算符号+、-、/、*的算术表达式。根节点保存运算符号，其他节点或者存变量名（像 A、B 或 C），或者保存运算符号。每一棵子树都是一个合法的代数表达式。



中缀 : A*(B+C)
前缀 : *A+BC
后缀 : ABC+*

图 8.11 表示一个算术表达式的树

例如，图 8.11 中的二叉树就表示了一个算术表达式：

A*(B+C)

这样写的是中缀表达法；这种写法是算术中常用的。（要更多地了解中缀和后缀表达法，请参见第 4 章“栈和队列”中“解析算术表达式”一节。）中序遍历树得到正确的中序序列 A*B+C，不过需要自己加上小括号。

所有这些和前序和后序遍历有什么关系呢？还是来看看它们都包含了什么步骤吧。这两种遍历方法和中序遍历的三个步骤相同，只是步骤的顺序不同。下面是前序遍历 preOrder()的三个步骤：

1. 访问节点。
2. 调用自身遍历该节点的左子树。
3. 调用自身遍历该节点的右子树。

用前序遍历图 8.11 中的树会得到如下表达式：

*A+BC

这是前缀表达式。它的一个好处是不用加小括号；没有小括号表达式也不会混乱。从左边开始，每个操作符号作用于表达式中它后面的两个算式。对第一个操作符*来说，它作用于 A 和+BC。对第二个操作符+，它作用于 B 和 C，这是中缀表达式中最后的表达式 B+C。把它插到原先的前缀表达式，*A+BC（前序遍历）绘出的就是中序的 A*(B+C)。通过应用树的不同的遍历方法，可以把代数表达式的一种形式转换成另一种形式。

第三种遍历方法，后序遍历，把那三个步骤又换了顺序：

1. 调用自身遍历节点的左子树。
2. 调用自身遍历节点的右子树。
3. 访问该节点。

在图 8.11 中，用后序遍历方法遍历树将生成下面的表达式：

ABC+*

这是后缀表达式。它表示要把表达式中最后的操作符号*应用于第一和第二个算式。第一个是 A，第二个是 BC+。

BC+是要把表达式中最后的操作符号：+，应用于第一和第二个操作数。第一个是 B，第二个是 C，所以就得到中缀的(B+C)。把它插到原先的表达式 ABC+*（后缀表达式）中就得到了中缀表达式 A*(B+C)。

注意：

清单 8.1 中有前序遍历和后序遍历的代码，也有中序遍历的代码。

这里就不再详细讨论了，不过可以相当容易地根据输入的后缀表达式建立一棵像图 8.11 那样的树。方法类似于计算后缀表达式，如在 postfix.java 程序中看到的那样（第四章清单 4.8）。不过，这里不是在栈中存储操作数，而是存储整个子树。像 postfix.java 那样，顺序读取后缀字符串。这里是遇到操作数时的操作步骤：

1. 建立一棵树，它只有一个保存操作数的节点。
2. 这棵树入栈。

下面是遇到操作符时的操作步骤：

1. 两个操作数的树 B 和 C 出栈。
2. 建立一棵新的树 A，操作符是根。
3. 把 B 接到 A 的右子节点处。
4. 把 C 接到 A 的左子节点处。
5. 把得到的新树压入栈。

这样，在分解完后缀表达式之后，把栈中所剩的惟一的一个数据项出栈。奇妙的是，这个数据项是一棵完整的树，表示整个的算术表达式。可以通过前面说的方式遍历树，就从初始的后缀表达式得到前缀和中缀表达式（当然也可以得到后缀表达式）。这个问题的实现作为一个练习题。

查找最大值和最小值

顺便说一下，在二叉搜索树中得到最大值和最小值是轻而易举的事情。事实上，这个过程非常容易，因此没有包含在专题 applet 的功能中，清单 8.1 中也没有显示它的代码。不过，理解方法的原理还是有必要的。

要找最小值时，先走到根的左子节点处，然后接着走到那个子节点的左子节点，如此类推，直到找到一个没有左子节点的节点。这个节点就是最小值的节点，如图 8.12 所示。

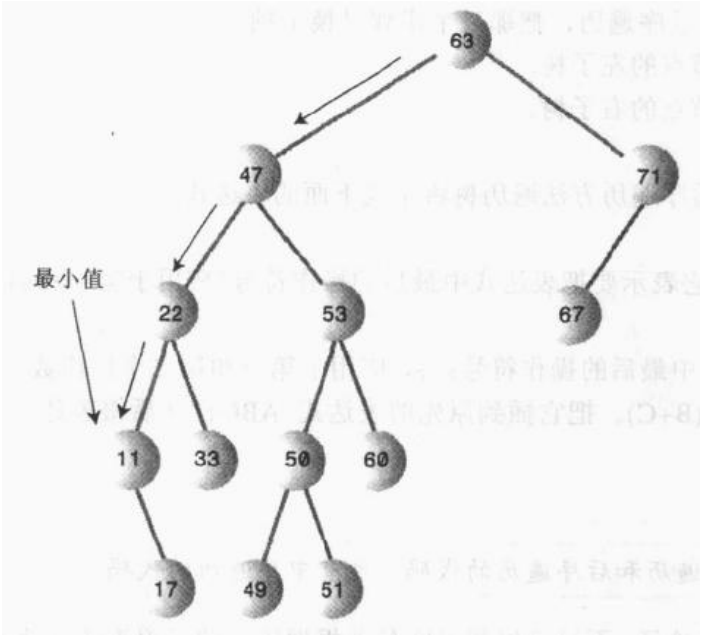


图 8.12 树中的最小值

下面是返回最小关键字值的节点的方法代码：

```
public Node minimum()      // returns node with minimum key value
   {
   Node current, last;
   current = root;                  // start at root
   while(current != null)           // until the bottom,
```

```
    {
    last = current;                 // remember node
    current = current.leftChild;    // go to left child
    }
 return last;
 }
```

在学习删除节点之前，需要了解如何找到最小值的节点。

按照相同的步骤来查找树中的最大值，不过要找到右子节点，一直向右找到没有右子节点的节点。这个节点就是最大值的节点。代码和上面的代码相同，除了把循环中的最后一句改成下面这句就可以了：

```
current = current.rightChild;  // go to right child
```

删除节点

删除节点是二叉搜索树常用的一般操作中最复杂的。但是，删除节点在很多树的应用中又非常重要，所以要详细研究并总结特点。

删除节点要从查找要删的节点开始入手，方法与前面介绍的 find()和 insert()相同。找到节点后，这个要删除的节点可能会有三种情况需要考虑：

1. 该节点是叶节点（没有子节点）。
2. 该节点有一个子节点。
3. 该节点有两个子节点。

下面将依次讲解这三种情况。第一种最简单；第二种也还是比较简单的；而第三种就相当复杂了。

情况 1：删除没有子节点的节点

要删除叶节点，只需要改变该节点的父节点的对应子字段的值，由指向该节点改为 null 就可以了。要删除的节点仍然存在，但它已经不是树的一部分了。如图 8.13 所示。

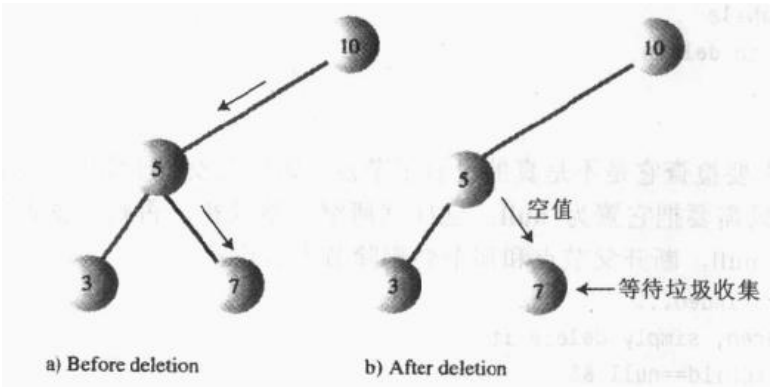


图 8.13 删除没有子节点的节点

因为 Java 语言有垃圾自动收集的机制，所以不需要非得把节点本身给删掉。一旦 Java 意识到程序不再与这个节点有关联，就会自动把它清理出存储器。（在 C 或 C++语言中需要运行 free()或

delete()方法来把节点从存储器中销毁。)

用专题 applet 删除没有子节点的节点

假设要删除图 8.13 中的节点 7。点击 Del 按钮，在提示框中输入 7。删除前要先找到这个节点。再次点击 Del，会从 10 找到 5 再找到 7。找到这个节点后，它就会被立刻删除。

删除没有子节点的节点的 Java 代码

delete()方法的第一部分和 find()和 insert()方法很像。它查找要删除的节点。和 insert()一样，需要保存要删节点的父节点，这样就可以修改它的子字段的值了。如果找到节点了，就从 while 循环中跳出，parent 保存要删除的节点。如果找不到要删除的节点，就从 delete()方法返回 false。

```
public boolean delete(int key) // delete node with given key
   {                           // (assumes non-empty list)
   Node current = root;
   Node parent = root;
   boolean isLeftChild = true;

   while(current.iData != key)        // search for node
      {
      parent = current;
      if(key < current.iData)         // go left?
         {
         isLeftChild = true;
         current = current.leftChild;
         }
      else                            // or go right?
         {
      isLeftChild = false;
      current = current.rightChild;
      }
   if(current == null)              // end of the line,
      return false;                 // didn't find it
   }  // end while
 // found node to delete
 // continues...
 }
```

找到节点后，先要检查它是不是真的没有子节点。如果它没有子节点，还需要检查它是不是根。如果它是根的话，只需要把它置为 null；这样就清空了整棵树。否则，就把父节点的 leftChild 或 rightChild 字段置为 null，断开父节点和那个要删除节点的连接。

```
// delete() continued...
// if no children, simply delete it
if(current.leftChild==null &&
                           current.rightChild==null)
   {
   if(current == root)              // if root,
      root = null;                  // tree is empty
```

```
      else if(isLeftChild)
         parent.leftChild = null;      // disconnect
      else                             // from parent
         parent.rightChild = null;
      }
   // continues...
```

情况 2：删除有一个子节点的节点

这第二种情况也不是很难。这个节点只有两个连接：连向父节点的和连向它惟一的子节点的。需要从这个序列中“剪断”这个节点，把它的子节点直接连到它的父节点上。这个过程要求改变父节点适当的引用（左子节点还是右子节点），指向要删除节点的子节点。如图 8.14 所示。

用专题 applet 删除有一个子节点的节点

假设要用专题 applet 处理图 8.14 的那棵树，要删除节点 71，此节点有左子节点，但没有右子节点。点击 Del 在提示框输入 71。持续点击 Del 直到箭头停在 71 上。节点 71 只有一个子节点，63。不用管 63 是否有自己的子节点；在这里 63 有一个子节点：67。

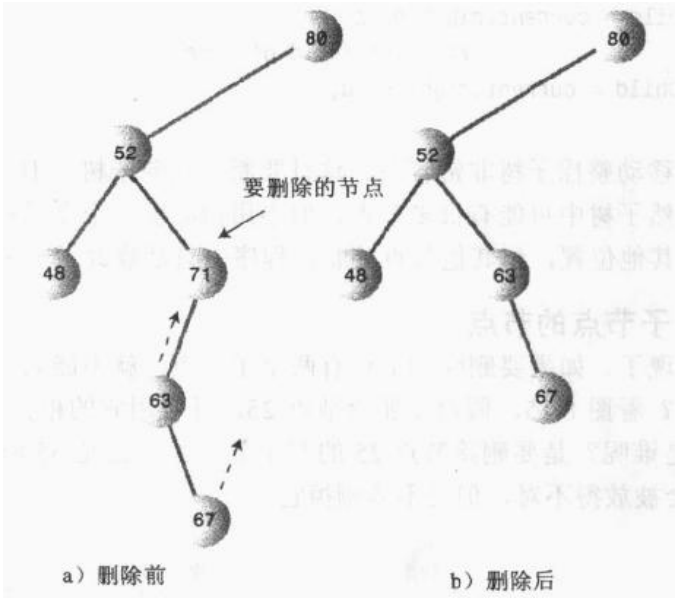


图 8.14　删除有一个子节点的节点

在点击 Del 一次，71 就被删掉了。它的位置被它的左子节点 63 取代了。实际上，63 作为根的整棵子树上移，作为 52 的新的右子节点插入。

用专题 applet 创建一棵有一个子节点的新的树，观察一下删除时的情况。查找一棵根为要删除节点的子节点的子树。无论这棵子树有多复杂，它只是上移并作为要删除节点的父节点的新的子节点。

删除有一个子节点的节点的 Java 代码

下面的代码用来处理有一个子节点的节点的情况。有四种不同的情况：要删除节点的子节点可能有左子节点或右子节点，并且每种情况中的要删除节点也可能是自己父节点的左子节点或右子节点。

这里还有一个特殊情况：被删除节点可能是根，它没有父节点，只是被合适的子树所代替。下

面是代码（从前面的删除没有子节点的节点的代码片段处继续）：

```
// delete() continued...
// if no right child, replace with left subtree
else if(current.rightChild==null)
   if(current == root)
      root = current.leftChild;
   else if(isLeftChild)           // left child of parent
      parent.leftChild = current.leftChild;
   else                           // right child of parent
      parent.rightChild = current.leftChild;

// if no left child, replace with right subtree
else if(current.leftChild==null)
   if(current == root)
      root = current.rightChild;
   else if(isLeftChild)           // left child of parent
      parent.leftChild = current.rightChild;
   else                           // right child of parent
      parent.rightChild = current.rightChild;
// continued...
```

注意应用引用使得移动整棵子树非常容易。这只要断开连向子树的旧的引用，建立新的引用连到别处去就可以了。虽然子树中可能有很多节点，但不用担心要一个个的移动它们。实际上，它们的移动只是概念上移到其他位置，和其他节点关联。程序中只是修改了子树的根的引用。

情况 3：删除有两个子节点的节点

下面有趣的情况出现了。如果要删除的节点有两个子节点，就不能只是用它的一个子节点代替它。为什么不能这样呢？看图 8.15，假设要删除节点 25，并且用它的根是 35 的右子树取代它。那么 35 的左子节点应该是谁呢？是要删除节点 25 的左子节点 15，还是 35 原来的左子节点 30？然而在这两种情况中 30 都会被放得不对，但又不能删掉它。

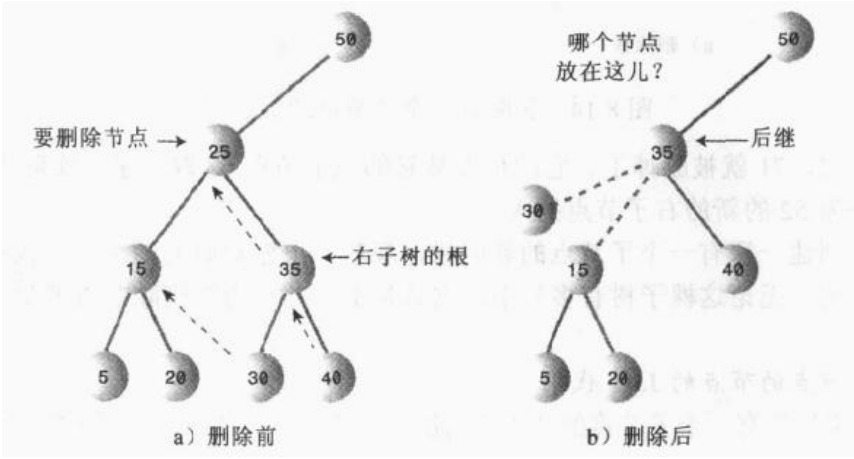


图 8.15 不能用子树替代

因此就需要用另一种方法。好的方面是此处有一个窍门。然而即使是有了窍门，还是有很多种情况要考虑。别忘了在二叉搜索树中的节点是按照升序的关键字值排列的。对每一个节点来说，比该节点的关键字值次高的节点是它的中序后继，可以简称为该节点的后继。在图 8.15a 中，节点 30 就是节点 25 的后继。

这就是窍门了：删除有两个子节点的节点，用它的中序后继来代替该节点。图 8.16 显示的就是要删除节点用它的后继代替它的情况。注意现在节点还是有序的。（这里还有更麻烦的情况是它的后继自己也有子节点，后面会讨论这种可能性。）

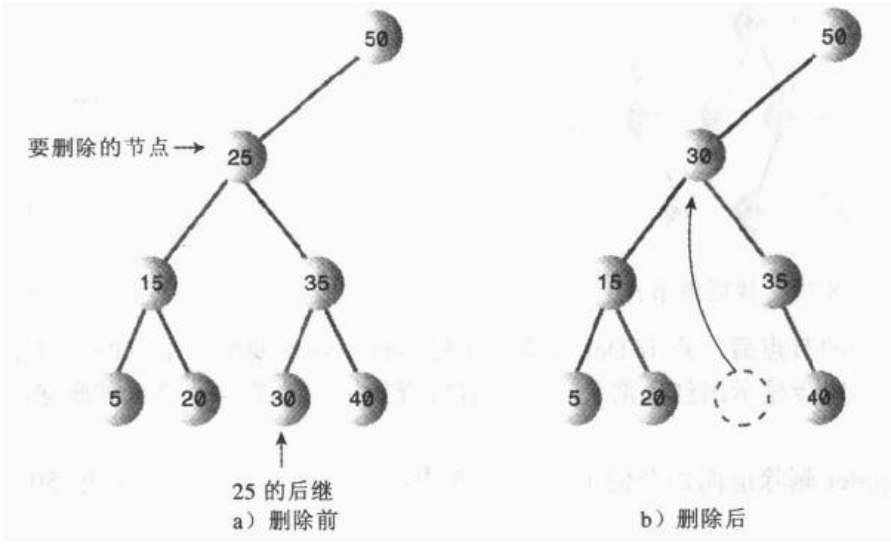


图 8.16　被后继替换的节点

找后继节点

怎么找节点的后继呢？对人来说，可以很快地找到节点的后继（不过也限于简单的树）。只要很快的扫一眼树，就可以找到比要删除节点大的下一个节点了。在图 8.16 中就可以很快看出 25 的后继是 30。没有其他的节点比 25 大，而比 30 小了。但是，计算机不能“扫一眼”就找到节点的后继；它需要有算法。下面就是算法：

首先，程序找到初始节点的右子节点，它的关键字值一定比初始节点大。然后转到初始节点的右子节点的左子节点那里（如果有的话），然后到这个左子节点的左子节点，以此类推，顺着左子节点的路径一直向下找。这个路径上的最后一个左子节点就是初始节点的后继，如图 8.17 所示。

为什么可以用这个算法呢？这里实际上是要找比初始节点关键值大的节点集合中最小的一个节点。当找到初始节点的右子节点时，这个以右子节点为根的子树的所有节点都比初始节点的关键字值大，因为这是二叉搜索树所定义的。现在要找到这棵子树中值最小的节点。本书已经讲过，找子树的最小值应该顺着所有左子节点的路径找下去。因此，这个算法可以找到比初始节点大的最小的节点；它就是要找的后继。

如果初始节点的右子节点没有左子节点，那么这个右子节点本身就是后继。如图 8.18 所示。

用专题 applet 删除有两个子节点的节点

用专题 applet 创建一棵树，选择一个有两个子节点的节点。现在在头脑中找出那个节点的后继，先到右子节点，再顺着这个右子节点的左子节点路径找（如果有的话）。需要确定后继没有自己的

子节点。如果有的话，情况就更复杂了，因为整棵子树都要旋转，而不是一个节点。

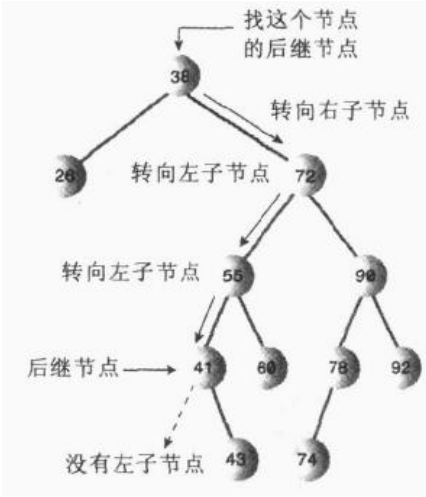


图 8.17 找后继节点

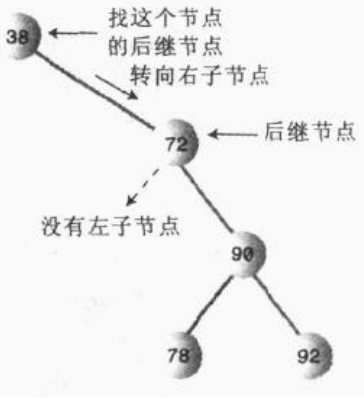


图 8.18 右子节点是后继节点

选定一个要删除的节点后，点击 Del 按钮。在提示框中输入要删除节点的关键字的值。输入之后，反复点击 Del 按钮会显示出红色箭头正在顺着树查找指定的节点。节点被删除时，该节点被它的后继所替代。

假设用专题 applet 删除前面那个例子里图 8.15 中的节点 30。红箭头会从根 50 到 25，之后 25 会被 30 代替。

找后继节点的 Java 代码

下面是 getSuccessor()方法的代码，它返回参数 delNode 的后继节点。（这个方法假设 delNode 有右子节点，因为已经判断过这个要删除的节点有两个子节点。）

```
// returns node with next-highest value after delNode
// goes to right child, then right child's left descendants

private node getSuccessor(node delNode)
   {
   Node successorParent = delNode;
Node successor = delNode;
Node current = delNode.rightChild;   // go to right child
while(current != null)               // until no more
   {                                 // left children,
   successorParent = successor;
   successor = current;
   current = current.leftChild;      // go to left child
   }
                                     // if successor not
if(successor != delNode.rightChild)  // right child,
   {                                 // make connections
   successorParent.leftChild = successor.rightChild;
```

```
        successor.rightChild = delNode.rightChild;
        }
    return successor;
    }
```

这个方法首先找到 delNode 的右子节点，然后，在 while 循环中，顺着这个右子节点所有左子节点的路径向下查找。当 while 循环中止时，successor 就存有 delNode 的后继。

找到后继后，还需要访问它的父节点，所以在 while 循环中还需要保留当前节点的父节点。

getSuccessor()方法除了找到后继，还有两个其他的功能。但是，为了理解它们，需要回顾并研究更全的代码。

正如看到的那样，后继节点可能与 current 有两种位置关系，current 就是要删除的节点。后继可能是 current 的右子节点，或者也可能是 current 右子节点的左子孙节点。下面来依次看看这两种情况。

后继节点是 delNode 的右子节点

如果后继是 current 的右子节点，情况就简单了一点，因为只需要把后继为根的子树移到删除的节点的位置。这个操作只需要两个步骤：

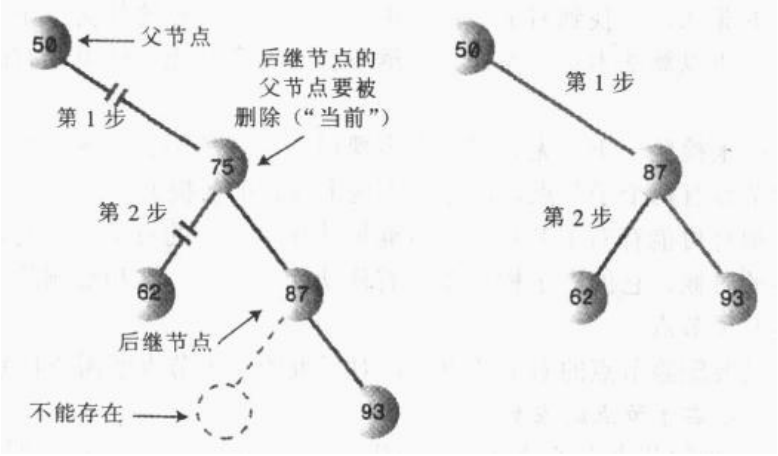
1. 把 current 从它父节点的 rightChild 字段删掉（当然也可能是 leftChild 字段），把这个字段指向后继。

2. 把 current 的左子节点移出来，把它插到后继的 leftChild 字段。

下面是执行这两个步骤的代码语句，是从 delete()中摘出来的：

```
1. parent.rightChild = successor;

2. successor.leftChild = current.leftChild;
```

图 8.19 总结了这种情况，显示了这两步的连接情况。



a）删除前　　b）删除后

图 8.19　后继节点是右子节点时的删除过程

下面的代码是前面代码的延续（接前面代码的 else-if 语句）：

```
// delete() continued
else  // two children, so replace with inorder successor
```

```
    {
    // get successor of node to delete (current)
    Node successor = getSuccessor(current);

    // connect parent of current to successor instead
    if(current == root)
       root = successor;
    else if(isLeftChild)
       parent.leftChild = successor;
    else
       parent.rightChild = successor;
    // connect successor to current's left child
    successor.leftChild = current.leftChild;
    }  // end else two children
// (successor cannot have a left child)
return true;
}  // end delete()
```

注意终于到 delete()方法的结尾一段了。下面回顾一下这两步的代码：

- 第一步：如果要删除的节点 current 是根，它没有父节点，所以就只需要把根置为后继。否则，要删除的节点或者是左子节点或者是右子节点了（图 8.19 中它是右子节点），因此需要把它父节点的对应的字段指向 successor。当 delete()方法返回，current 失去了作用范围后，就没有引用指向 current 保存的节点，它就会被 Java 的垃圾收集机制销毁。
- 第二步：把 successor 的左子节点指向的位置设为 current 的左子节点。

如果后继有子节点怎么办呢？首先，后继节点是肯定不会有左子节点的。无论后继是要删除节点的右子节点还是这个右子节点的左子节点之一，这条都能保证。为什么呢？

回忆一下找后继的算法，先找到右子节点，再找这个右子节点的任何左子节点。它会在没有左子节点的节点处停止，所以算法本身就保证了后继不会有左子节点。如果它有的话，则那个左子节点才应该是后继。

可以用专题 applet 来检验一下。无论创建多少棵树，永远都不会出现一个节点的后继有左子节点的情况（假设初始节点有两个子节点，这是产生这个问题的前提）。

另一方面，后继很有可能有右子节点。当后继是被删除节点的右子节点时，这种情况不会带来多大问题。移动后继的时候，它的右子树只要跟着移动就可以了。这和要删除节点的右子节点没有冲突，因为后继就是右子节点。

下一节中后继不是要删除节点的右子节点时，对后继的右子节点就需要特别地小心。

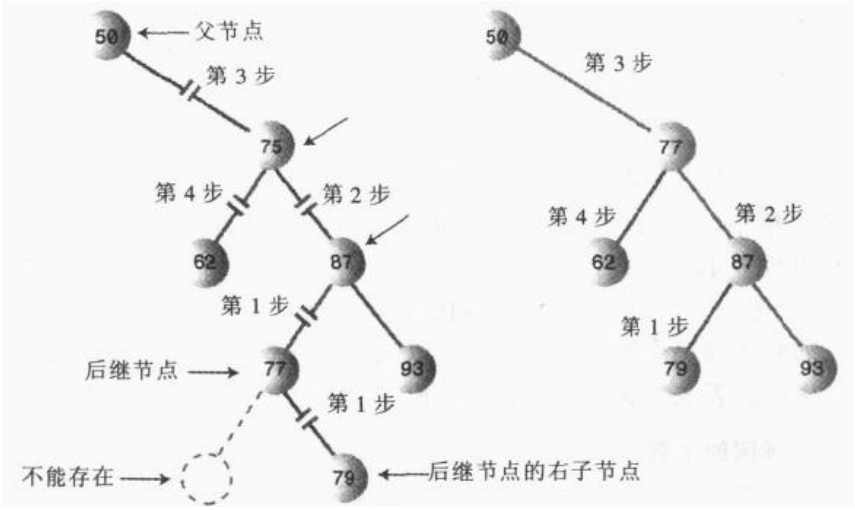
后继节点是 delNode 右子节点的左后代

如果 successor 是要删除节点右子节点的左后代，执行删除操作需要以下四个步骤：

1. 把后继父节点的 leftChild 字段置为 successor 的右子节点。
2. 把 successor 的 rightChild 字段置为要删除节点的右子节点。
3. 把 current 从它父节点的 rightChild 字段移除，把这个字段置为 successor。
4. 把 current 的左子节点从 current 移除，successor 的 leftChild 字段置为 current 的左子节点。

第 1 步和第 2 步由 getSuccessor()方法完成，第 3 步和第 4 步由 delete()方法完成。图 8.20 显示

这四步操作下节点间连接的变化情况。



a）删除前　　b）删除后

图 8.20　后继节点是左子节点时的删除过程

下面是这四步的代码：

```
1. successorParent.leftChild = successor.rightChild;
2. successor.rightChild = delNode.rightChild;
3. parent.rightChild = successor;
4. successor.leftChild = current.leftChild;
```

图 8.20 中标注的数字表示这四步改变连接的情况。第 1 步导致 successor 被它的右子树替代。第 2 步把要删除节点的右子节点挪到正确的位置（当后继是要删除节点的右子节点时这步就自动完成了）。第 1 步和第 2 步在前面讲的 getSuccessor()方法最后的 if 条件句里完成。下面列出这段代码：

```
                                    // if successor not
if(successor != delNode.rightChild) // right child,
   {                                // make connections
   successorParent.leftChild = successor.rightChild;
   successor.rightChild = delNode.rightChild;
   }
```

这两步在这个方法里比在 delete()中好，因为在 getSuccessor()中可以在树中向下寻找后继时顺便找到后继的父节点。

第 3 步和第 4 步已经讲过了；它们与后继是要删除节点的右子节点时的代码一样，就在 delete()中最后的 if 条件句中。

删除是必要的吗？

看到这里，就会发现删除是相当棘手的操作。实际上，因为它非常复杂，一些程序员都尝试着躲开它。他们在 node 类中加了一个 Boolean 的字段，名称如 isDeleted。要删除一个节点时，就把此节点的这个字段置为 true。其他操作，像 find()，在查找之前先判断这个节点是不是标志为已删除了。这样，删除的节点不会改变树的结构。当然，这样做存储中还保留着这种“已经删除”的节点。

这种方法或许有些逃避责任，但如果树中没有那么多删除操作时，这也不失为一个好方法。（例如，已经离职的员工的档案要永久保存在员工记录中。）

二叉树的效率

正如前面看到的一样，树的大部分操作都需要从上到下一层一层地查找某个节点。它这样做要花多少时间呢？一棵满树中，大约一半的节点在最底层。（更确切地说，如果是满树，最底层的节点个数比树的其他节点数多1。）因此，查找、插入或删除节点的操作大约有一半都需要找到最底层的节点。（另外还有四分之一节点的这些操作要到倒数第二层，依次类推。）

在查找过程中，需要访问每层的一个节点。所以只要知道有多少层就可以知道这些操作需要多长时间了。假设一棵满树，表8.2显示的就是容纳下给定数量节点所需要的层数。

表8.2 特定节点数量的满树的层数

节点数	层数
1	1
3	2
7	3
15	4
31	5
…	…
1023	10
…	…
32767	15
…	…
1048575	20
…	…
33554432	25
…	…
1073741824	30

这个情况和第2章讨论的有序数组很相似。在第2章中，二叉搜索比较的次数大致是数组中数据项个数的以2为底的对数。这里，设表中第一列节点个数为N，第二列层数为L，则N比2的L次方小1，即：

$N=2^L-1$

等式每边加1，得到：

$N+1=2^L$

这等于：

$L=\log_2(N+1)$。

因此，常见的树的操作时间复杂度大致是N以2为底的对数。在大O表示法中，表示为O(logN).

如果树不满，分析起来就很困难。不过，可以认为对给定层数的树，不满的树的平均查找时间

比满树要短，因为它在较低的层上完成查找的次数要比满树时少。

把树和前面讲过的那些数据结构做比较。在有 1000000 个数据项的无序数组或链表中，查找数据项平均会比较 500000 次，但在有 1000000 个节点的树中，只需要 20（或更少）次的比较。

有序数组可以很快的找到数据项，但插入数据项平均需要移动 500000 个数据项。在 1000000 个节点的树中插入数据项需要 20 次或更少的比较，再加上很短的时间来连接数据项。

同样，从有 1000000 个数据项的数组中删除一个数据项需要平均移动 500000 个数据项，而在 1000000 个节点的树中删除节点只需要 20 次或更少的比较来找到它，再加上（可能的话）一点比较的时间来找到它的后继，一点时间来断开这个节点的连接，以及连接它的后继。

因此，树对所有常用的数据存储操作都有很高的效率。

遍历不如其他操作快。但是，遍历在大型数据库中不是常用的操作。它更常用于程序中的辅助方法来解析算术或其他的表达式，而且表达式一般都不会很长。

用数组表示树

上面的实例代码基于这样的思想，树的边由每个节点的 leftChild 和 rightChild 字段中的引用表示。但是，还有一种完全不同的方法来表示树：用数组。

用数组的方法时，节点存在数组中，而不是由引用相连。节点在数组中的位置对应于它在树中的位置。下标为 0 的节点是根，下标为 1 的节点是根的左子节点，依次类推，按从左到右的顺序存储树的每一层。如图 8.21 所示。

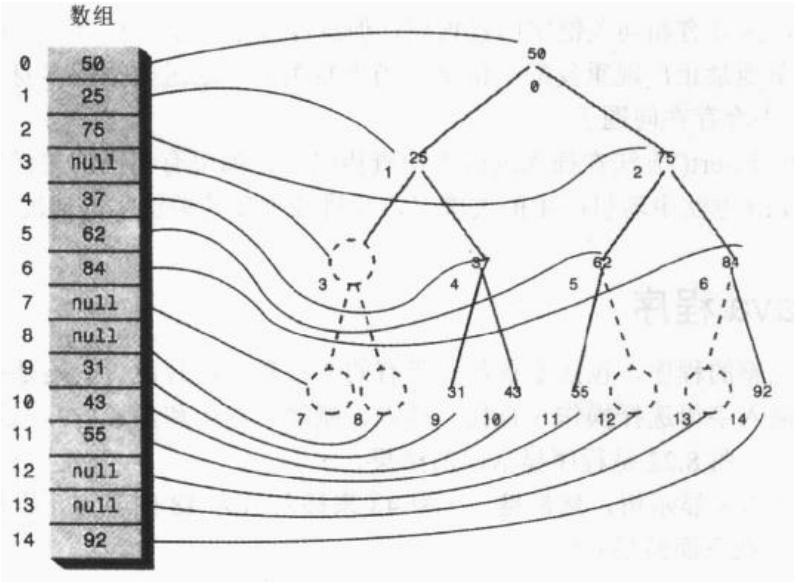


图 8.21 用数组表示的树

树中的每个位置，无论是否存在节点，都对应数组中的一个位置。把节点插入树的一个位置，意味着要在数组的相应位置插入一个数据项。树中没有节点的位置在数组中的对应位置用 0 或 null 来表示。

基于这种思想，找节点的子节点和父节点可以利用简单的算术计算它们在数组中的索引值。设

节点索引值为 index，则节点的左子节点是：

```
2*index + 1
```

它的右子节点是

```
2*index + 2
```

它的父节点是

```
(index-1) / 2
```

（"/" 符号表示整除运算）。可以对照图 8.21 来检验一下上面的算式。

大多数情况下用数组表示树不是很有效率。不满的节点和删除掉的节点在数组中留下了洞，浪费存储空间。更坏的是，删除节点时需要移动子树的话，子树中的每个节点都要移到数组中新的位置去，这在比较大的树中是很费时的。

不过，如果不允许删除操作，数组表示可能会很有用，特别是因为某种原因要动态地为每个节点分配空间非常耗时。数组表示法在其他一些特殊的情况下也很有用。例如，专题 applet 中的树，在内部就是由数组表示的，这样便于将数组中的节点画到屏幕上固定的位置上。

重复关键字

和其他数据结构一样，重复关键字问题是必须要提到的。insert()方法的代码中，以及在专题 applet 里，有重复关键字的节点都插到与它关键字相同的节点的右子节点处。

问题是 find()方法只能找到两个（或多个）相同关键字节点中的第一个。可以修改 find()方法来查找更多的数据项，区分有相同关键字的数据项，但这样做很（至少有点）消耗时间。

一种选择是简单地禁止出现重复的关键字。当重复关键字通过数据的本身被排除了（例如，员工代码的数字），就不会存在问题了。

否则，需要修改 insert()方法在插入过程中检查相同性，如果有相同的关键字时就放弃插入。

专题 applet 的 Fill 方法中随机产生的关键字，会排除重复的关键字的情况。

完整的 tree.java 程序

本节中要列出完整的程序，包括本章前面所有的方法和代码片段。它还是提供了简单的用户接口。允许用户通过输入字母选择操作（查找、插入、删除、遍历和显示树）。显示例程用字符输出，显示出一幅树的内容。图 8.22 是程序显示出的结果。

上图中，用户键入 s 显示树，然后键入 i 和 48 来插入值为 48 的节点，再次键入 s 显示新插入节点后的树。48 出现在下面的显示中。

可用的命令是字母 s、i、f、d 和 t，分别用于显示、插入、查找、删除和遍历。i、f 和 d 选项需要输入要操作节点的关键字值。t 选项要用户选择遍历的方式：1 是前序遍历，2 是中序遍历，3 是后序遍历。关键字值就按用户选择的遍历顺序显示出来。

显示方法可以把关键值按树形排列显示出来；但需要设想边的存在。两个短线符号（--）表示树中这个位置的节点不存在。程序初始化时创建一些节点，用户在没有做任何插入操作之前就可以看到它们。可以修改初始化的代码，从需要的任何节点开始，或没有任何节点（这是一种很好的状态）。

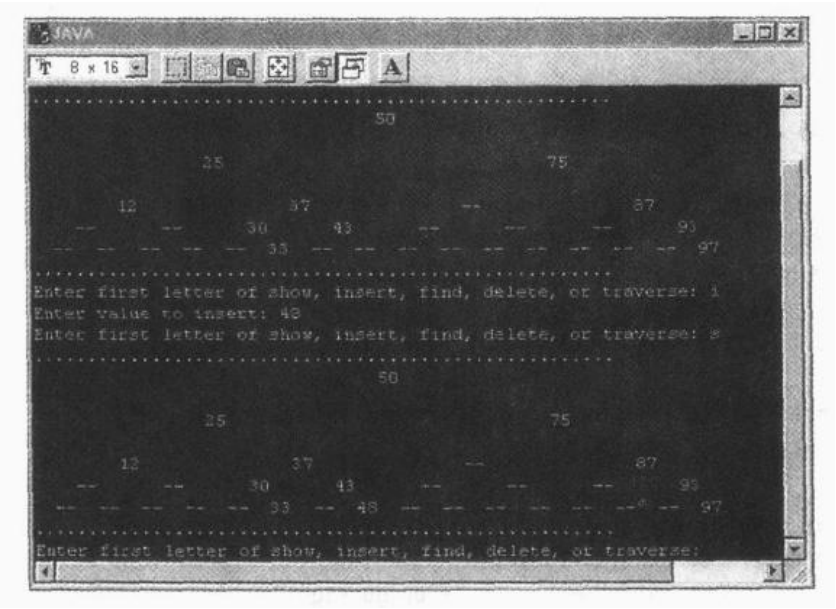


图 8.22 tree.java 程序的输出界面

像专题 applet 一样，可以实验清单 8.1 中的程序。不过不像专题 applet 那样能够显示出操作运行的步骤；程序中所有操作都是一下子运行完成的。

清单 8.1 tree.java 程序

```
// tree.java
// demonstrates binary tree
// to run this program: C>java TreeApp
import java.io.*;
import java.util.*;                 // for Stack class
////////////////////////////////////////////////////////////////
class Node
   {
   public int iData;                // data item (key)
   public double dData;             // data item
   public Node leftChild;           // this node's left child
   public Node rightChild;          // this node's right child

   public void displayNode()        // display ourself
      {
      System.out.print('{');
      System.out.print(iData);
      System.out.print(", ");
      System.out.print(dData);
      System.out.print("} ");
      }
   }  // end class Node
```

```
////////////////////////////////////////////////////////////////
class Tree
   {
   private Node root;             // first node of tree

// -------------------------------------------------------------
   public Tree()                  // constructor
      { root = null; }            // no nodes in tree yet
// -------------------------------------------------------------
   public Node find(int key)      // find node with given key
      {                           // (assumes non-empty tree)
      Node current = root;                // start at root
      while(current.iData != key)         // while no match,
         {
         if(key < current.iData)          // go left?
            current = current.leftChild;
         else                             // or go right?
            current = current.rightChild;
         if(current == null)              // if no child,
            return null;                  // didn't find it
         }
      return current;                     // found it
      } // end find()
// -------------------------------------------------------------
   public void insert(int id, double dd)
      {
      Node newNode = new Node();    // make new node
      newNode.iData = id;           // insert data
      newNode.dData = dd;
      if(root==null)                // no node in root
         root = newNode;
      else                          // root occupied
         {
         Node current = root;       // start at root
         Node parent;
         while(true)                // (exits internally)
            {
            parent = current;
            if(id < current.iData)  // go left?
               {
               current = current.leftChild;
               if(current == null)  // if end of the line,
                  {                 // insert on left
                  parent.leftChild = newNode;
                  return;
```

```
                }
             }  // end if go left
          else                    // or go right?
             {
             current = current.rightChild;
             if(current == null)  // if end of the line
                {                 // insert on right
                parent.rightChild = newNode;
                return;
                }
             }  // end else go right
          }  // end while
       }  // end else not root
    }  // end insert()
// ------------------------------------------------------------
   public boolean delete(int key) // delete node with given key
      {                           // (assumes non-empty list)
      Node current = root;
      Node parent = root;
      boolean isLeftChild = true;

      while(current.iData != key)        // search for node
         {
         parent = current;
         if(key < current.iData)         // go left?
            {
            isLeftChild = true;
            current = current.leftChild;
            }
         else                            // or go right?
            {
            isLeftChild = false;
            current = current.rightChild;
            }
         if(current == null)             // end of the line,
            return false;                // didn't find it
         }  // end while
      // found node to delete

      // if no children, simply delete it
      if(current.leftChild==null &&
                                 current.rightChild==null)
         {
         if(current == root)             // if root,
            root = null;                 // tree is empty
```

```
      else if(isLeftChild)
         parent.leftChild = null;      // disconnect
      else                             // from parent
         parent.rightChild = null;
      }

   // if no right child, replace with left subtree
   else if(current.rightChild==null)
      if(current == root)
         root = current.leftChild;
      else if(isLeftChild)
         parent.leftChild = current.leftChild;
      else
         parent.rightChild = current.leftChild;

   // if no left child, replace with right subtree
   else if(current.leftChild==null)
      if(current == root)
         root = current.rightChild;
      else if(isLeftChild)
         parent.leftChild = current.rightChild;
      else
         parent.rightChild = current.rightChild;

   else  // two children, so replace with inorder successor
      {
      // get successor of node to delete (current)
      Node successor = getSuccessor(current);

      // connect parent of current to successor instead
      if(current == root)
         root = successor;
      else if(isLeftChild)
         parent.leftChild = successor;
      else
         parent.rightChild = successor;

      // connect successor to current's left child
      successor.leftChild = current.leftChild;
      }  // end else two children
    // (successor cannot have a left child)
      return true;                                  // success
      }  // end delete()
// -------------------------------------------------------------
   // returns node with next-highest value after delNode
```

```
   // goes to right child, then right child's left descendents
   private Node getSuccessor(Node delNode)
      {
      Node successorParent = delNode;
      Node successor = delNode;
      Node current = delNode.rightChild;   // go to right child
      while(current != null)               // until no more
         {                                 // left children,
         successorParent = successor;
         successor = current;
         current = current.leftChild;      // go to left child
         }
                                           // if successor not
      if(successor != delNode.rightChild)  // right child,
         {                                 // make connections
         successorParent.leftChild = successor.rightChild;
         successor.rightChild = delNode.rightChild;
         }
      return successor;
      }
// -----------------------------------------------------------
   public void traverse(int traverseType)
      {
      switch(traverseType)
         {
         case 1: System.out.print("\nPreorder traversal: ");
                 preOrder(root);
                 break;
         case 2: System.out.print("\nInorder traversal:  ");
                 inOrder(root);
                 break;
         case 3: System.out.print("\nPostorder traversal: ");
                 postOrder(root);
                 break;
         }
      System.out.println();
      }
// -----------------------------------------------------------
   private void preOrder(Node localRoot)
      {
      if(localRoot != null)
         {
         System.out.print(localRoot.iData + " ");
         preOrder(localRoot.leftChild);
         preOrder(localRoot.rightChild);
```

```
        }
      }
// ------------------------------------------------------------
   private void inOrder(Node localRoot)
      {
      if(localRoot != null)
         {
         inOrder(localRoot.leftChild);
         System.out.print(localRoot.iData + " ");
         inOrder(localRoot.rightChild);
         }
      }
// ------------------------------------------------------------
   private void postOrder(Node localRoot)
      {
      if(localRoot != null)
         {
         postOrder(localRoot.leftChild);
         postOrder(localRoot.rightChild);
         System.out.print(localRoot.iData + " ");
         }
      }
// ------------------------------------------------------------
   public void displayTree()
      {
      Stack globalStack = new Stack();
      globalStack.push(root);
      int nBlanks = 32;
      boolean isRowEmpty = false;
      System.out.println(
      "......................................................");
      while(isRowEmpty==false)
         {
         Stack localStack = new Stack();
         isRowEmpty = true;

         for(int j=0; j<nBlanks; j++)
            System.out.print(' ');

         while(globalStack.isEmpty()==false)
            {
            Node temp = (Node)globalStack.pop();
            if(temp != null)
               {
               System.out.print(temp.iData);
```

```
            localStack.push(temp.leftChild);
            localStack.push(temp.rightChild);

            if(temp.leftChild != null ||
                              temp.rightChild != null)
               isRowEmpty = false;
            }
         else
            {
            System.out.print("--");
            localStack.push(null);
            localStack.push(null);
            }
         for(int j=0; j<nBlanks*2-2; j++)
            System.out.print(' ');
         } // end while globalStack not empty
      System.out.println();
      nBlanks /= 2;
      while(localStack.isEmpty()==false)
         globalStack.push( localStack.pop() );
      } // end while isRowEmpty is false
   System.out.println(
   "..................................................");
   } // end displayTree()
// --------------------------------------------------------
   } // end class Tree
////////////////////////////////////////////////////////////////
class TreeApp
   {
public static void main(String[] args) throws IOException
   {
   int value;
   Tree theTree = new Tree();

   theTree.insert(50, 1.5);
   theTree.insert(25, 1.2);
   theTree.insert(75, 1.7);
   theTree.insert(12, 1.5);
   theTree.insert(37, 1.2);
   theTree.insert(43, 1.7);
   theTree.insert(30, 1.5);
   theTree.insert(33, 1.2);
   theTree.insert(87, 1.7);
   theTree.insert(93, 1.5);
   theTree.insert(97, 1.5);
```

```
while(true)
   {
   System.out.print("Enter first letter of show, ");
   System.out.print("insert, find, delete, or traverse: ");
   int choice = getChar();
   switch(choice)
      {
      case 's':
         theTree.displayTree();
         break;
      case 'i':
         System.out.print("Enter value to insert: ");
         value = getInt();
         theTree.insert(value, value + 0.9);
         break;
      case 'f':
         System.out.print("Enter value to find: ");
         value = getInt();
         Node found = theTree.find(value);
         if(found != null)
            {
            System.out.print("Found: ");
            found.displayNode();
            System.out.print("\n");
              }
           else
              System.out.print("Could not find ");
              System.out.print(value + '\n');
           break;
        case 'd':
           System.out.print("Enter value to delete: ");
           value = getInt();
           boolean didDelete = theTree.delete(value);
           if(didDelete)
              System.out.print("Deleted " + value + '\n');
           else
              System.out.print("Could not delete ");
              System.out.print(value + '\n');
           break;
        case 't':
           System.out.print("Enter type 1, 2 or 3: ");
           value = getInt();
           theTree.traverse(value);
           break;
```

```
            default:
               System.out.print("Invalid entry\n");
            } // end switch
         } // end while
      } // end main()
// --------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
// --------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
//--------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
// --------------------------------------------------------
   } // end class TreeApp
////////////////////////////////////////////////////////////////
```

用组合键 Ctrl+G 退出程序；为了简单起见这个操作没有设单独的键来退出。

哈夫曼（Huffman）编码

二叉树并不全是搜索树。很多二叉树用于其他的情况。图 8.11 中的例子里二叉树表示了一个代数表达式。

本节讲解一个算法，它使用二叉树以令人惊讶的方式来压缩数据。1952 年 David Huffman 发现这种方法后，就称它为哈夫曼编码。数据压缩在很多领域中都是非常重要的。例如要通过互联网发送数据，尤其是通过拨号连接时，传送过程要花很长时间。这种方法的实现是很冗长的，这里就不给出完整的程序了。而把重点放在概念上，实现留作练习。

字符编码

计算机里每个字符在没有压缩的文本文件中由一个字节（如常见的 ASCII 码）或两个字节（如比较新的 Unicode，它可以在各种语言中通用）表示。这些方案中，每个字符需要相同的位数。表

8.3列出了在二进制中用ASCII码表示的一些字符。可以看到每个字符都用8个位表示。

表8.3 一些ASCII代码

字母	十进制	二进制
A	65	01000000
B	66	01000001
C	67	01000010
...	...	...
X	88	01011000
Y	89	01011001
Z	90	01011010

有很多压缩数据的方法。对文本来说，最常用的方法是减少表示最常用字符的位数量。如英语中，E是最常用的字母，所以用尽可能少的位为E编码是很合理的。反之，Z是很少用到的，所以用多些位表示也无所谓。

假设只用两位表示E，如01。那么就不能给字典中其他每个字母都用两位编码了，因为2位只有四种组合：00、01、10和11。可以用这四种组合表示四个最常用的字符吗？

很遗憾，不可以；必须小心的是，在编码序列中，如果用起始位组合相同的代码表示不同的字符，这样的情况是不能出现的。比如，如果E是01，而X是01011000，那么解码是就搞不清楚01011000起始的01是表示E还是表示X的开始部分了。这就产生了一个规则：每个代码都不能是其他代码的前缀。

还有一些时候有的信息里E并不是最常用的字符。比如，文本是Java源代码的时候，分号“;”可能都比E出现的次数更多。下面就来解决这个问题：为每种信息根据其特别的信息定制新的编码。假设要发送消息：SUSIE SAYS IT IS EASY。字母S出现得最多，其次是空格。用表格来列出每种字符出现的次数。这样的表叫频率表，如表8.4所示，。

表8.4 频率表

字符	出现次数
A	2
E	2
I	3
S	6
T	1
U	1
Y	2
空格符	4
换行符	1

编码时，出现次数最多的字符所占位数应该最少。表8.5显示了应如何为Susie消息中的字符编码。

表 8.5 哈夫曼编码

字符	编码
A	010
E	1111
I	110
S	10
T	0110
U	01111
Y	1110
空格符	00
换行符	01110

用 10 表示 S，用 00 表示空格后，就不能再用 01 和 11 了，因为它们是其他字符的前缀。那三位的组合怎么办呢？三位的组合有 8 种可能性：000、001、010、011、100、101、110 和 111。A 是 010，I 是 110。为什么不能用其他组合呢？这是因为已知不能用由 10 或 00 开始的组合；这就减少了四种选择。同时，011 用于 U 和换行符代码的开始，111 用于 E 和 Y 的开始。这样就只有两个三位代码留下来可以用了，它们就用来代表 A 和 I。同样可以理解为什么只有三个四位代码可用。

因此，整个消息编码后为：

```
10 01111 10 110 1111 00 10 010 1110 10 00 110 0110 00 110 10 00
➥ 1111 010 10 1110 01110
```

为了清楚，这里把这条消息的每个字符的代码分开显示。实际上所有位会连在一起；在二进制消息中没有空格字符，只有 0 和 1。

用哈夫曼树解码

稍后会看到如何建立哈夫曼编码。首先来看看简单一些的解码是怎么完成的。假设收到上节中的那条位字符串表示的消息。怎么才能把它转换回字符呢？可以使用一种叫做哈夫曼树的二叉树。如图 8.23 所示就是刚才讨论的代码的哈夫曼树。

消息中出现的字符在树中是叶节点。它们在消息中出现的频率越高，在树中的位置就越高。每个圆圈外面的数字就是频率。非叶节点外面的数字是它子节点的频率的和。稍后会看到这样做的重要性。

怎么用这棵树来解码呢？每个字符都从根开始。如果遇到 0，就向左走到下个节点，如果遇到 1，就向右。例如字符 A 的代码是 010。先向左，向右，再向左，说也奇怪，这样就找到了

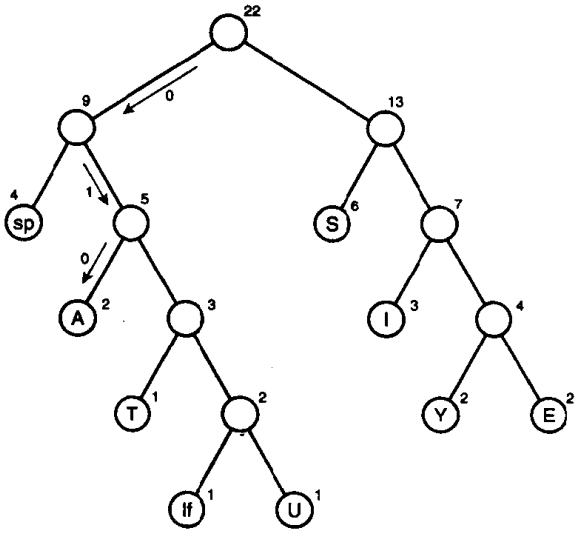


图 8.23 哈夫曼树

A 节点。这个过程如图 8.23 中的箭头所示。

解码其他字符也是一样的过程。有耐心的话，可以用这方法把整个比特串都解码。

创建哈夫曼树

上面已经学习了怎么用哈夫曼树解码，但怎么建立哈夫曼树呢？有很多方法来处理这个问题。本章的方法是基于清单 8.1 tree.java 程序中的 Node 类和 Tree 类（不过那些类中的应用于查找树的特殊方法，像 find()、insert()和 delete()都没有用了）。下面是建立哈夫曼树的算法：

1. 为消息中的每个字符创建一个 Node 对象（像 tree.java 中的那样）。在 Susie 示例中有九个节点。每个节点有两个数据项：字符和字符在消息中出现的频率。表 8.4 中列出了 Susie 消息中字符出现的频率。

2. 为这些节点创建 tree 对象。这些节点就是树的根。

3. 把这些树都插入到一个优先级队列中（第 4 章中讲过）。它们按频率排序，频率最小的节点有最高的优先级。因此，删除一棵树的时候，它就是队中最少用到的字符。

现在做下面的事情：

1. 从优先级队列中删掉两棵树，并把它们作为一个新节点的子节点。新节点的频率是子节点频率的和；它的字符字段可以是空的。

2. 把这个新的三节点树插回优先级队列里。

3. 反复重复第一步和第二步。树会越变越大，队列中的数据项会越来越少。当队中只有一棵树时，它就是所建的哈夫曼树了。

图 8.24 和 8.25 展示了如何对 Susie 的消息建立哈夫曼树。

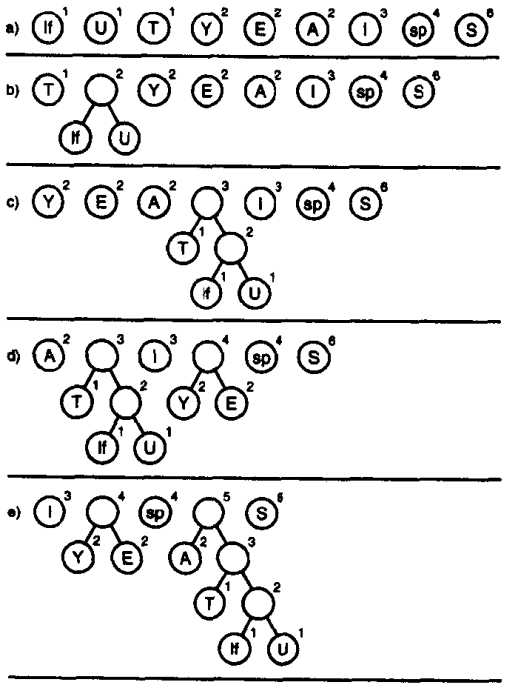


图 8.24 建立哈夫曼树，第一部分

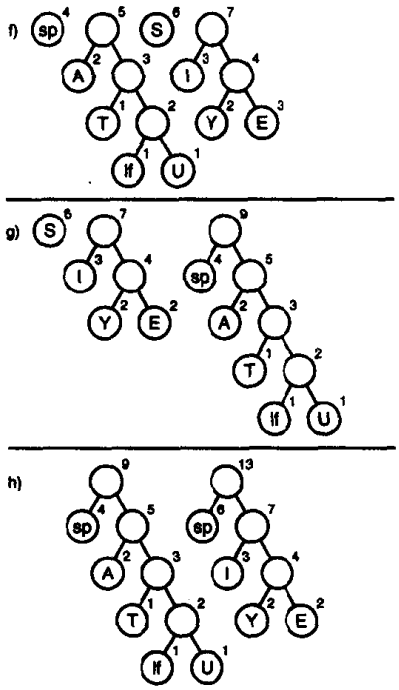


图 8.25 建立哈夫曼树，第二部分

信息编码

现在建好了哈夫曼树，怎么为一段信息编码呢？从建立代码表开始，这个表中列出了每个字符的哈夫曼代码。为了简化讨论，假设计算机中不是用 ASCII 代码，而是用简化的字母表，只有 28 个字符的大写字母。A 是 0，B 是 1，一直到 Z，是 25。空格是 26，换行符号是 27。为这些字符按数字顺序编码，它们的代码从 0 到 27。（这不是压缩编码，只是 ASCII 代码的简化，通常计算机中都是这么存储字符的。）

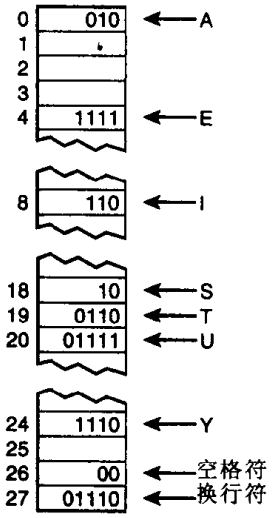


图 8.26 代码表

代码表就是一个有 28 个单元的数组。每个单元的下标就是每个字符的数字编码：0 是 A，1 是 B，依此类推。单元的内容将是对应字符的哈夫曼编码。不是每个单元都有哈夫曼编码的；只有出现在消息中的字符才会有。图 8.26 显示了对 Susie 消息的字符用这种格式保存的情况。

这样一个代码表就很方便编写信息的代码了：对每个原始消息中的每一个字符，都用代码表中对应的代码表示。然后再把对应的哈夫曼编码添加到信息的尾部，直到完成整个消息的代码转换为止。

创建哈夫曼编码

怎样建立哈夫曼编码并把它插入代码表中呢？这个过程类似于对一条信息解码的过程。从哈夫曼树的根开始并顺着哈夫曼树沿一条可能的路径到达叶节点。在顺着树往下走时，记录向左和向右的顺序，向左的边用 0 表示，向右的边就用 1 表示。到达某个叶节点后，0 和 1 的序列就是这个字符的哈夫曼编码。把这个代码插入代码表的正确位置就可以了。

这个过程的实现可以从根开始调用一个方法，然后对它的每个子节点递归地调用方法本身。最后，所有到叶节点的路径都被遍历了，代码表也完成了。

小 结

- 树由边（直线）连接的节点（圆）组成。
- 根是树中最顶端的节点；它没有父节点。
- 二叉树中，节点最多有两个子节点。
- 二叉搜索树中，所有 A 节点左边子孙节点的关键字值都比 A 小；所有右边子孙节点的关键字值都大于（或等于）A。
- 树执行查找、插入、删除的时间复杂度都是 O（logN）。
- 节点表示保存在树中的数据对象。
- 程序中通常用节点到子节点的引用来表示边（有时也用到节点的父节点的引用）。
- 遍历树是按某种顺序访问树中所有的节点。
- 最简单的遍历方法是前序、中序和后序。
- 非平衡树是指根左边的后代比右边多，或者相反，右边的后代比左边多。

- 查找节点需要比较要找的关键字值和节点的关键字值，如果要找节点关键值小就转向那个节点的左子节点，如果大就转向右子节点。
- 插入需要找到要插入新节点的位置并改变它父节点的子字段来指向它。
- 中序遍历按照关键值的升序访问节点。
- 前序和后序遍历对解析代数表达式是有用的。
- 如果一个节点没有子节点，删除它只要把它的父节点的子字段置为 null 即可。
- 如果一个节点有一个子节点，把它父节点的子字段置为它的子节点就可以删除它。
- 如果一个节点有两个子节点，删除它要用它的后继来代替它。
- A 节点的后继是以 A 的右子节点为根的子树中关键值最小的那个节点。
- 删除操作中，如果节点有两个子节点，会根据后继是被删节点的右子节点还是被删节点右子节点的左子孙节点出现两种不同的情况。
- 数组中重复关键字值的节点会产生一些问题，因为只有第一个能被查找到。
- 在计算机存储时可以用数组表示树，不过基于引用的方法更常用。
- 哈夫曼树是二叉树（但不是二叉搜索树），用于数据压缩算法，这种方法称为哈夫曼编码。
- 哈夫曼编码中，最经常出现的字符的编码位数最少，很少出现的字符编码位数要多一些。

问 题

下列问题作为读者的自测题。答案可见附录 C。

1. 树中插入和删除操作的时间复杂度是什么（用大 O 表示法表示）？
2. 下列哪种情况下二叉树是搜索树？

a. 每个非叶节点的子节点的关键字值都比其父节点的关键值小（或相等）。
b. 每个左子节点的关键字值比父节点小，每个右子节点的关键字值大于（或等于）父节点。
c. 从根到每个叶节点的路径上，每个节点的关键字值大于（或等于）它父节点的关键值。
d. 节点可能是两个子节点的最大值。

3. 判断题：不是所有树都是二叉树。
4. 有 20 个节点的完全二叉树中，根节点在第 0 层，第 4 层有多少个节点？
5. 二叉树的子树总是：

a. 有一个根，它为主树的根的子节点。
b. 有一个根，它与主树的根不相连。
c. 比主树节点少。
d. 有同样数目节点的兄弟节点。

6. 树的 Java 代码中，_____和_____一般分成两个类。
7. 在二叉搜索树中查找节点从一个节点走到另一个节点，需要知道：

a. 与要查找的节点相比节点关键字值是大还是小。
b. 节点关键字值与它的左子节点或右子节点比是大还是小。
c. 要找到的是哪个叶节点。

d. 现在在哪一层。

8. 不平衡树是下列哪一种树：

a. 大多数关键字值比平均值要大。

b. 行为不可预测。

c. 根或其他节点的左边节点比右边节点多，或者反之。

d. 形状像伞一样。

9. 插入节点操作的开始步骤和_____节点的开始操作的步骤一样。

10. 设节点 A 的后继节点是 S。那么 S 节点的关键字值一定比_____大，但小于或等于_____。

11. 二叉树用于表示数学表达式时，下列哪种说法是不正确的？

a. 操作符节点的两个子节点必须都是操作数。

b. 按照后序遍历，得到的算式不需要加小括号。

c. 按照中序遍历，得到的算式必须要加小括号。

d. 前序遍历时，在遍历子节点前先遍历根。

12. 用数组表示树时，节点下标值为 n，则它右子节点的下标是_____。

13. 判断题：从二叉搜索树中删除有一个子节点的节点需要找到这个节点的后继。

14. 哈夫曼树专用于_____文本。

15. 关于哈夫曼树，下列哪种说法是错误的？

a. 最常出现的字符总是在靠近树顶附近出现。

b. 通常，信息解码需要重复地顺着根到叶的路径走。

c. 为字符编码需要从叶开始再向上。

d. 哈夫曼树可以通过在优先级队列中的插入和移除操作来创建。

实　验

完成这些实验可以帮助深入理解本章的主题。不需要编程实现。

1. 用二叉树专题 applet 建立 20 棵树。出现严格的非平衡树的比例是多少？

2. 删除二叉搜索树中的节点时会有各种可能性，为此画出 UML 的活动图的情况（或比较过时的程序流程图）。应该详细地表示前面讲的三种情况。包括左右子节点不同以及根删除等特殊的情况。例如，第一种情况（左和右子节点）时有两种可能性。每个路径最后的方框中应表示出在此种情况下如何删除节点。

3. 在各种可能的情况下，用二叉树专题 applet 删除节点。

编程作业

编程作业有助于巩固对本章内容的理解，并展示如何应用本章的概念。（在“简介”中提到过，资深教师可以从出版者的网站上得到编程作业的完整答案。）

8.1　从 tree.java 程序（清单 8.1）出发，把它修改成用用户输入的字母的字符串建立二叉树（如

A、B 等等）。每个字母在各自的节点中显示。建立树，让每个包含字母的节点是叶节点。父节点可以有非字母标志如‘+’。保证每个父节点都恰好有两个子节点。不要担心树不平衡。注意这不是搜索树；没有快速的方法来查找节点。最后结果可以像下面所示的一样：

```
                  +
          +                   E
     +          D         -         -
  +    C    -    -    -    -    -    -
 A B  - -  - -  - -  - -  - -  - -  - -
```

一种方法是建立一个树的数组。（一组没有连接的树称为森林。）用用户输入的每个字母作为一个节点。把每个节点作为一棵树，节点就是根。现在把这些单节点的树放到数组中。下面开始建立一棵以‘+’为根的树，两个单节点树为它的子节点。依次把数组中的单节点树加到这棵大点的树中。不要担心它是不是非平衡树。实际可以用生成的中间树覆盖数组单元，该数组单元的内容已经加到树中。

find()、insert()和 delete()例程只用于搜索树，可以删掉了。保留 displayTree()方法和遍历方法，它们对所有二叉树都通用。

8.2 扩展编程作业 8.1 中的功能来创建平衡树。一种方法是保证叶节点尽可能出现在最底层。开始把每对单节点树建成三节点树，以‘+’为根。这样就有一个三节点树的森林了。合并每对三节点树变成七节点树的森林。随着每棵树节点数目的增加，树的数量减少，直到最后只有一棵树。

8.3 还是从 tree.java 程序开始，根据用户输入的字符创建树。这次，建立完全树——除了底层的最右边可能为空，其他层节点完全满的树。字母要从上到下以及从左到右有序，好像就是建立了一个字符的金字塔。（这种排列方法与本章前面讨论的三种遍历方法都不对应。）因此，字符串 ABCDEFGHIJ 会排列成下面的样子：

```
          A
     B         C
  D    E    F    G
 H I  J
```

建立这种树的一种方法是从上到下，而不是像编程作业中前两题那样从底向上。从建立一个作为最后树的根的节点开始。如果节点编号和字母排列顺序一样，1 是根，则编号为 n 的节点的左子节点的编号为 2*n，右子节点的编号为 2*n+1。可以用递归方法，创建两个子节点并在每个子节点中调用它自己。节点创建的顺序不需要和它们插入到树中的顺序相同。像编程作业中前面的题那样，可以删掉 Tree 类中搜索树的一些方法。

8.4 编写程序根据后缀表达式建立树，如本章图 8.11 中所示。需要修改 tree.java 程序（清单 8.1）中的 Tree 类，和第 4 章 postfix.java 程序（清单 4.8）中的 ParsePost 类。更多细节参考图 8.11 的讲解。

建好树之后，遍历树可以得到算术表达式相应的前缀、中缀和后缀表达式。中缀表达式需要小括号来避免产生模糊不清的表达式。InOrder()方法中，在第一次递归调用之前加入左括号，在第二次递归调用之后加入右括号。

8.5 编写程序实现哈夫曼编码和解码。需要做如下的工作：

Accept a text message, possibly of more than one line.

Create a Huffman tree for this message.

Create a code table.

Encode the message into binary.

Decode the message from binary back to text.

如果信息很短，程序可以显示建立好的哈夫曼树。编程作业 8.1、8.2 和 8.3 的方法会有所帮助的。可以用 String 变量把二进制数字存储为字符 1 和 0 的序列。除非确有必要，否则不需要做实际的位处理。

第 9 章

红-黑树

本章重点

- 本章讨论的方法
- 平衡树和非平衡树
- 用专题 applet 试验
- 旋转
- 插入一个新节点
- 删除
- 红-黑树的效率
- 红-黑树的实现
- 其他平衡树

在第 8 章“二叉树”中已经讲过，普通的二叉搜索树作为数据存储工具有重要的优势：可以快速地找到一个给定关键字的数据项，并且可以快速地插入和删除数据项。其他的数据存储结构，例如数组、有序数组以及链表，执行这些操作却很慢。因此，二叉搜索树似乎是理想的数据存储结构。

遗憾的是，二叉搜索树有一个很麻烦的问题。如果树中插入的是随机数据，则执行效果很好。但是，如果插入的是有序的数据（17，21，28，36，...）或者是逆序的数据（36，28，21，17，...），速度就变得特别慢。因为当插入的数值有序时，二叉树就是非平衡的了。而对于非平衡树，它的快速查找（插入，删除）指定数据项的能力就丧失了。

本章将讨论一种解决非平衡树问题的方法：红-黑树，它是增加了某些特点的二叉搜索树。

还有其他的一些方法来保证树是平衡的。在本章的最后将会提到一些，并将讨论几种第 10 章“2-3-4 树和外部存储”中讲的 2-3-4 树和 2-3 树。实际上，正如在那章中讲的那样，在 2-3-4 树上的操作和在红-黑树上的操作有惊人的对应之处。

本章讨论的方法

本章将要讨论的在红-黑树中的插入方法和在已经讲过的在其他数据结构中的方法有一点不同。理解红-黑树至关重要。因为这一点，也因为有大量的对称情况（如是左子节点或者是右子节点，内部或者外部的子孙节点，等等），实际的代码要比人们想像的长而且复杂。因此通过研究代码学习算法是十分困难的。所以，这一章中没有程序代码清单。可以利用第 10 章中 2-3-4 树的代码来创建相似的功能。不过，本章学习的概念将对理解 2-3-4 树有所帮助，况且它们本身也相当有趣。

概念

现在用 RBTree 专题 applet 来帮助理解红-黑树的概念。下面将讲解读者如何与 applet 共同操作，在树中插入一个新数据项。把人为的作用包括在插入例程中，肯定会使它的速度放慢，然而使读者更容易理解这个过程的执行情况。

红-黑树中的搜索方法和普通二叉树的一样。然而在另一方面，尽管插入和删除操作都基于普通树的算法，但有的改动非常大。因此，这一章的重点放在插入操作的处理上。

自顶向下插入

这里将讨论的插入算法称为自顶向下的插入，也就是说在搜索例程沿着树向下查找插入点，在

此进程中可能要对树的结构做一些改变。

另一种方法称为自底向上的插入。它需要找到插入新数据项的位置，然后沿着树向上改变树的结构。自底向上插入效率比较低，这是因为必须顺着树扫描两趟。

平衡树和非平衡树

在开始讲解红-黑树之前，先来回顾一下树是如何变得不平衡的。启动第 8 章中的 Binary Tree 专题 applet（不是这章中的 RBTree 专题 applet）。点击 Fill 按钮创建一棵只有一个节点的树，然后插入一系列关键字有序的数据项，其关键字是升序或者是降序的。所得到的结果类似图 9.1 所示。

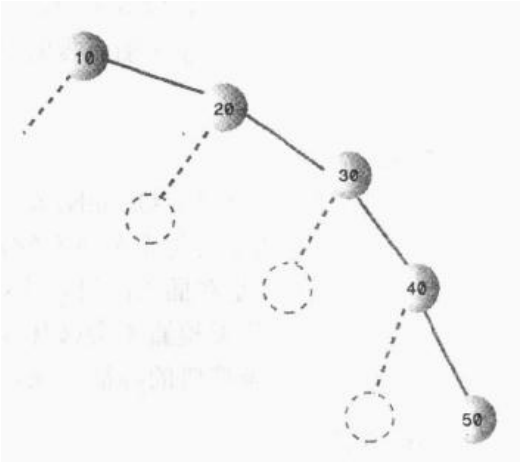


图 9.1 插入升序排序的数据项

这些节点排列在一条直线上，没有分支。若关键字为升序的，每一个节点都大于上一个插入的节点，每一个节点都是上一个节点的右子节点，所有的节点都在根的一侧。这棵树属于极端不平衡的情况。如果插入数据项的关键字是按降序排列的，则每一个节点都是它父节点的左子节点，节点都在另一侧，树也是极不平衡的。

时间复杂度降低到 O(N)

当树没有分支时，它其实就是一个链表。数据的排列是一维的，而不是二维的。不幸的是，和在链表中一样，现在就必须要（平均）查找一半的数据项来寻找要找的数据项。在这种情况下，查找的速度下降到 O(N)，而不是平衡树的 O(logN)。在一棵有 10000 个数据项的非平衡树中搜索数据项平均需要 5000 次比较，而在随机插入生成的平衡树中搜索数据项只需要 14 次比较。其实对于已经有序的数据首先使用链表也无妨。

由部分有序的数据所生成的树只是部分不平衡。如果用第 8 章中的 Binary Tree 专题 applet 生成 31 个节点的一棵树，可以看到这些节点中的一部分比另一部分不平衡，如图 9.2 所示。

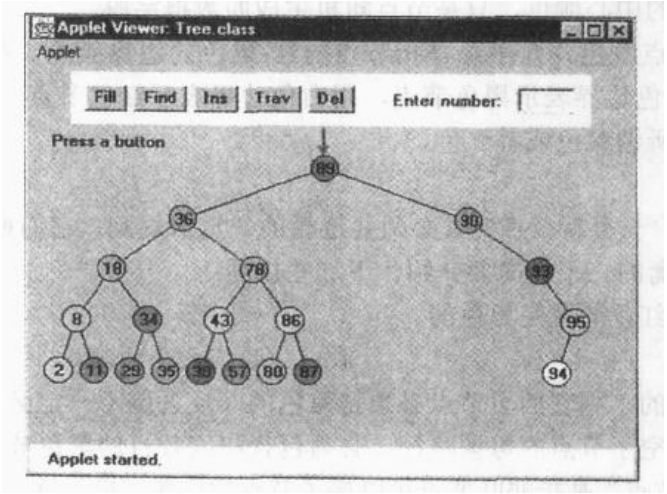


图 9.2 部分非平衡树

尽管它比最不平衡的树情况要好，但是在这种情况下的查找时间不是最佳的。

在 Binary Tree 专题 applet 中，甚至对随机生成的数据，树都可能是部分非平衡的，这是因为数据的数目太少，以至于很短的一段有序数据都会对树的平衡性造成很大的影响。同时，一个很小或很大的关键字值也会产生一个不平衡树，因为这将使很多节点不能插入到这个节点的一边或者另一边。例如，树的根是 3，在根的左边就只允许再插入两个数据项。

对于随机数据的实际数量来说，一棵树特别不平衡的情况是不大可能的。但是，可能会有一小部分有序数据使树部分非平衡。搜索部分非平衡树的时间介于 O(N)和 O(logN)之间，这取决于树的不平衡程度。

平衡的补救

为了能以较快的时间 O(logN)来搜索一棵树，需要保证树总是平衡的（或者至少大部分是平衡的）。这就是说对树中的每个节点在它左边的后代数目和在它右边的后代数目应该大致相等。

红-黑树的平衡是在插入的过程中（删除时也是，但暂时忽略这个问题）取得的。对一个要插入的数据项，插入例程要检查不会破坏树一定的特征。如果破坏了，程序就会进行纠正，根据需要更改树的结构。通过维持树的特征，保持了树的平衡。

红-黑树特征

这种树有什么神奇的特征呢？有两个特征，一个简单，另一个比较复杂：

- 节点都有颜色。
- 在插入和删除的过程中，要遵循保持这些颜色的不同排列的规则。

带颜色的节点

在红-黑树中，每一个节点或者是黑色的或者是红色的。也可以是任意的两种颜色；蓝色和黄色也是可以的。实际上，所说的节点有“颜色”是任意的比方。可以使用其他类似方法来表示：比如可以说每一个节点不是深色的就是是浅色的，或者说不是阴就是阳。不过，颜色便于标记。在节点类中增加一个数据字段，可以是 boolean 型的（例如，isRed），以此来表示颜色的信息。

在 RBTree 专题 applet 中，节点的红-黑特征通过节点边界的颜色来表示的。在前面章节的 Binary Tree 专题 applet 中节点的中心颜色，只是节点随机生成的数据字段。

在这一章中说到节点颜色，几乎总是指节点的红-黑色的边界颜色。在这一章的图中（除了图 9.3 之外），用加粗的黑色边界表示黑色节点，用白色边界表示红色节点。（注意有时显示的节点没有边界，表示了节点无所谓黑色或者红色。）

红-黑规则

当插入（或者删除）一个新节点时，必须要遵循的一定的规则，它们被称为红-黑规则。如果遵循这些规则，树就是平衡的。下面简要介绍一下这些规则：

1. 每一个节点不是红色的就是黑色的。
2. 根总是黑色的。
3. 如果节点是红色的，则它的子节点必须是黑色的（反之倒不一定必须为真）。
4. 从根到叶节点或空子节点的每条路径，必须包含相同数目的黑色节点。

规则 4 中的“空子节点”是指非叶节点可以接子节点的位置。换句话说，就是一个有右子节点的节点的可能接左子节点的位置，或者是有左子节点的节点的可能接右子节点位置。在下面的讲解中

会更表现出它的意义。

在从根到叶节点的路径上的黑色节点的数目称为黑色高度（black height）。规则 4 的另一种陈述方法是所有从根到叶节点路径上的黑色高度必须相同。

这些规则可能使人一头雾水。看不清楚这些规则怎么能使一棵树平衡，然而它们做到了；某些很聪明的人发明了它们。请把这些规则抄在一个便签上，贴在计算机旁边，因为在这一章的学习中需要经常查阅它们。

通过使用 RBTree 专题 applet 进行一些实验，以便了解上述这些规则是如何起作用的。但是在实验前，应当先知道，如果违犯了一条红-黑规则，应该采取什么措施来修正。

重复的关键字

如果有多于一个数据项的关键字值相同，将会出现什么情况呢？这给红-黑树提出了一个小问题。把有相同关键字的数据项分配到其他也有相同关键字数据项的两侧是很重要的。这也就是说，如果关键字的序列为 50，50，50，要把第二个 50 放到第一个 50 的右边，并且把第三个 50 放到第一个 50 的左边。否则，树将不平衡。

在插入算法中，用某种随机的过程处理相同关键字节点的分配问题。但是，如果要找到所有相同的关键字，插入过程就变得更复杂

因此不允许有关键字相同的数据项能使问题简单一些，目前的讨论假定不允许有重复关键字出现。

修正违规的情况

假设看到（或者 applet 提示）颜色的规则被违犯了。如何修正才能使树遵守上述规则呢？有两个，而且只有这两个可能的修正措施：

- 改变节点的颜色。
- 执行旋转操作。

在 applet 中，改变节点的颜色就是要改变节点的红-黑边界的颜色（而不是中心的颜色）。旋转是指重新排列节点，一次旋转使树更趋于平衡。

这些概念现在看起来可能很抽象，因此需要熟悉一下 RBTree 专题 applet，它有助于搞清楚这些概念。

使用 RBTree 专题 applet

图 9.3 显示了在 RBTree 专题 applet 中插入了一些节点后的情况。（在图中很难分区别出红色节点和黑色节点，但是在彩色显示器中它们的区别很明显。）

在 RBTree 专题 applet 中按钮相当多。这里将简要地介绍一下它们的功能，尽管目前有些说明可能有点难理解。

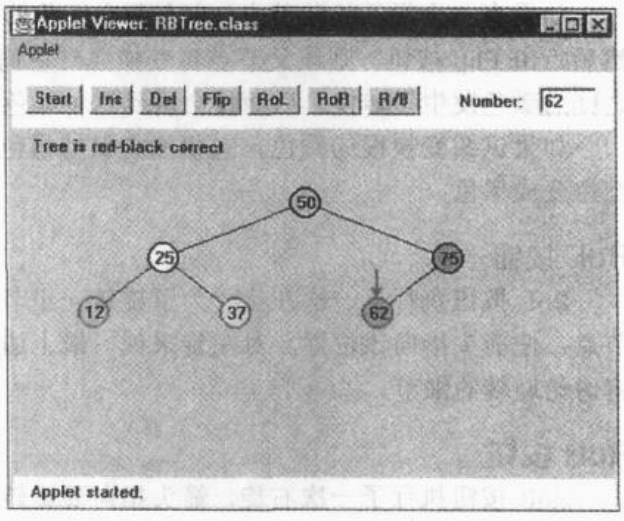


图 9.3 RBTree 专题 applet

点击一个节点

红色箭头指示当前选择的节点。正是这个节点要改变它的颜色或者是以它为轴进行旋转。用鼠标单击，选中一个节点，红色箭头便会指向这个节点。

Start 按钮

首次启动 RBTree 专题 applet 时，又同时点击 Start 按钮，就可以看到创建了一棵仅有一个节点的树。因为对红-黑树的理解都集中在插入的过程中使用红-黑规则，所以在开始时只有根。然后通过它插入其他节点建树。为了简化后面的操作，初始的根节点总是给定值 50。后面的插入就可以自己选择数字。

Ins 按钮

Ins 按钮可以创建一个新节点，在数字文本框中输入新节点的关键字值，然后把它插入到树中。（至少这是不需要颜色变换时的情况。颜色变换的更多信息参看 Flip 按钮部分。）

注意，只要点击一次 Ins 按钮就可以完成整个的插入操作，不需要多次点击这个按钮。这和在以前章节的 BinaryTree 专题 applet 中的要求不同。因此在点击按钮前键入关键字值是很重要的。在 RBTree 专题 applet 中关注的不是和在普通的二叉搜索树中类似的找节点插入位置的过程，而是保持树平衡的过程，所以 applet 不会显示插入过程的每个步骤。

Del 按钮

点击 Del 按钮将会删除在数组文本框中输入的关键字值的数据项。和 Ins 按钮一样，在第一次点击之后，删除立即执行；不需要多次点击。

Del 按钮和 Ins 按钮都使用基本的插入算法——和在 Tree 专题 applet 中的一样。这就在 applet 和用户之间进行了分工：applet 执行插入操作，但是用户对树做适当的改变，以确保遵守红-黑规则，并且由此保证平衡树是平衡的。

Flip 颜色变换按钮

如果有一个黑色的父节点，它有两个红色的子节点，用鼠标点击父节点，红色箭头便指向了它，然后点击 Flip 按钮，则该父节点将变换成红色而子节点们变换成黑色。这就是说，父节点和子节点之间的颜色发生了转换。后面将会讲到这种颜色交换的原因。

如果试图变换根的颜色，它仍然会保持黑色，否则将违反规则 2，但是它的子节点的颜色会从红色变成黑色。

RoL 按钮

RoL 按钮执行了一次左旋。为了旋转一组节点，首先用鼠标单击要旋转的一组节点的最上边的节点，使箭头指向该位置。对左旋来说，最上边的节点必须有一个右子节点。然后点击按钮。后面将讨论旋转的细节。

RoR 按钮

RoR 按钮执行了一次右旋。箭头指向要旋转的一组节点的最上边的节点，保证该节点有左子节点；然后点击按钮。

R/B 变色按钮

R/B 按钮把一个红色节点改变成黑色，或者把一个黑色节点改变成红色。单击鼠标使红色箭头指向该节点，然后再点击此按钮。（这个按钮只能改变一个节点的颜色；不要和 Flip 按钮混淆，Flip 按钮一次变换 3 个节点的颜色。）

文本信息

按钮下面的文本框中的信息表明树是否为正确的红黑树。如果树遵守了前面列出的规则 1 到规则 4，则树就是正确的红-黑树。如果它不是正确的，则可以在信息中看到它违反了哪一条规则。在某些情况下红色箭头会指向违反规则的位置。

Find 按钮在哪里?

在红-黑树中，查找例程和前面章节中讲过的二叉搜索树的完全相同。它也是从根开始，然后对每个遇到的节点（当前节点），通过比较当前节点的关键字和要查找的关键字来决定是转向左子节点还是转向右子节点。

在 RBTree 专题 applet 中不包含 Find 按钮，这是因为读者已经理解了查找的过程，并且本章的注意力放在操作树的红-黑变换方面。

用专题 applet 做试验

既然已经熟悉了 RBTree 专题 applet 的按钮，现在就来做一些简单的试验，亲身感受 applet 的功能。这里的目的是学会用 applet 的操作控制。后面将使用这些技巧来平衡一棵树。

试验 1：插入两个红色节点

点击 Start 清除多余的节点。于是只剩下根节点，它的值总是 50。

通过在数字文本框中输入数字并且点击 Ins 按钮，插入一个值比根小的新节点，比如 25。插入这个节点不会违背任何规则，因此信息继续显示为 Tree is red-black correct。

插入第二个节点，它的值大于根，比如 75。这棵树仍然是正确的红黑树。它也是平衡的；非叶子节点（根）右边的节点数目和在它左边的节点数目相等。结果如图 9.4 所示。

注意新插入的节点总是红色的（除了根节点）。这不是偶然的。而插入一个红色节点比插入黑色节点违背红-黑规则的可能性小。这是因为如果把新的红色节点连接到黑色节点上，不会违背规则。它不会造成有两个红色节点在一起的情况（规则 3），而且也不会改变任何路径上的黑色高度（规则 4）。当然，如果把新的红色节点连接到红色节点上，还是会违背规则 3。但是，不管怎么样只会有一半的机会发生这种情况。反之，如果加入一个新的黑色节点，总会改变这条路径上的黑色高度，从而违背了规则 4。

还有，违背规则 3（父和孩子都是红色的）比违背规则 4（黑色高度不同）更容易修正一些，后面将会讲到这一点。

试验 2：旋转

下面来做一些旋转的试验。如图 9.4 所示，开始时有三个节点。用鼠标点击根节点（50），使红色箭头指向它。这个节点将是旋转操作中的顶端节点。现在点击 RoR 按钮，执行一次右旋。所有的

节点都将变换到新位置，如图 9.5 所示。

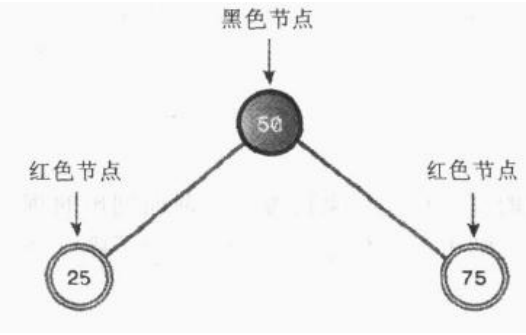


图 9.4 一棵平衡树

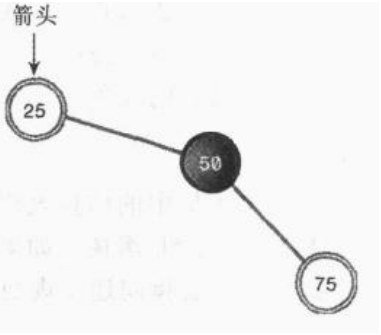


图 9.5 执行一次右旋之后

在这次右旋中，父节点或称顶端节点移到它的右子节点的位置，它的左子节点上移，并且取代了父节点的位置，右子节点下移成为新的顶端节点的子孙节点。

注意现在树是非平衡的；根右侧的节点比左侧的节点多。同时，消息也指出违背了红-黑规则，并指明是规则 2（根总是黑色的）。目前还不需要担心这个问题。

相反，向另一个方向做旋转。使红色箭头指向值为 25 的节点，它是当前的根节点（在上一次旋转之后箭头应该已经指向节点 25）。点击 RoL 按钮执行左旋。节点将回到如图 9.4 的位置上。

试验 3：颜色变换

开始时的位置如图 9.4 所示，除了节点 50 在根的位置上，还插入了节点 25 和节点 75。注意父节点（根）是黑色的，它的两个子节点是红色的。现在要插入另一个节点。不论节点值为多少，都会看到提示信息 Can't Insert: Needs color flip（不能插入，需要变换颜色）。

正如讲过的那样，只要在插入的过程中遇到一个有黑色节点，并且它还有两个红色的子节点，就需要进行颜色变换。

红色箭头应该已经指向了黑色父节点的位置（根节点），所以点击 Flip 按钮。根的两个子节点从红色变成黑色。通常，父节点会从黑色变成红色，但这里是一个特例，因为该节点是根：它仍然保持黑色以避免违背规则 2。现在所有的三个节点都是黑色的。树还是正确的红-黑树。

再点击 Ins 按钮来插入新的数据项。图 9.6 显示了插入一个关键字值为 12 的新节点的结果。

树还是正确的红-黑树。根是黑色的，没有出现父节点和子节点都是红色的情况，并且所有路径上都有相同数目的黑色节点（两个）。插入新的红色节点不会改变红-黑树的正确性。

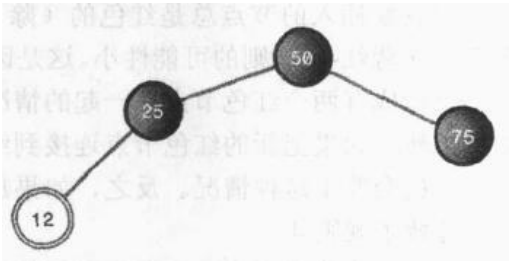


图 9.6 颜色变换，插入新节点

试验 4：非平衡树

现在来如果执行某些操作，导致了树的不平衡，这时，将会发生什么。在图 9.6 中一条路径比另一条路径多一个节点。这时是稍微有些不平衡，但是没有违反红-黑规则，所以人和红-黑算法都不必操心。但是，假设一条路径和另一条路径相差两层或者更多层（这里层就是路径上节点的数目），这种情况就总是违背红-黑规则的，需要重新平衡这

棵树。

在图 9.6 的树中插入一个节点 6。可以看到提示信息 Error: Parent and child are both red（错误：父和孩子都是红色的）。违反了规则 3，如图 9.7 所示。

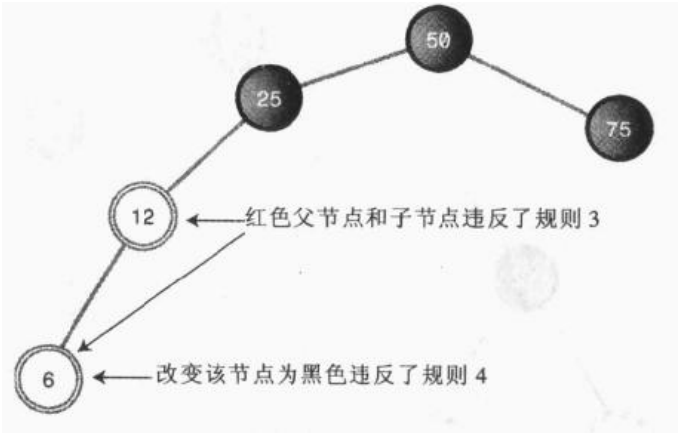


图 9.7 父节点和子节点都是红色的

应该怎样修改树就不违背规则 3 了呢？一个很明显的方法是把违反规则的一个节点变成黑色。下面试着改变子节点 6 的颜色。使红色箭头指向该节点，然后点击 R/B 按钮。该节点变为黑色。

好消息是解决了父节点和子节点都是红色的问题。坏消息是现在信息提示 Error: Black heights differ（错误：黑色高度不同）。根到节点 6 的路径上有三个黑色节点，而根到节点 75 的路径上却只有两个黑色节点。因此，违背了规则 4。这看起来修改没有成功。

这个问题可以通过一次旋转和几次颜色变换来解决。以后的章节将会讨论如何解决这个问题。

更多试验

用 RBTree 专题 applet 自己做更多的试验。插入更多的节点，观察会发生什么情况。看看是否能够通过旋转和变换颜色使树平衡。只要树不违背红-黑规则就能确保树是一棵平衡树吗？

试着插入升序排列的关键字序列（50，60，70，80，90），然后用 Start 按钮重启，再插入降序排列的关键字序列（50，40，30，20，10）。忽略提示信息；以后再讨论它们的意义。这种情况使普通的二叉搜索树陷入了困境。这里还能平衡这棵树吗？

红-黑规则和平衡树

试创建一棵树，它虽然已经超过两层不平衡了，但却要满足红-黑规则。事实证明，这是不可能的。这就是为什么红-黑规则能保证树是平衡的。如果一条路径上的节点数比另一条路径上的节点数多一个以上，那它要么有更多的黑色节点，违背了规则 4，要么有两个相邻接的红色节点，违背了规则 3。通过在 applet 上做试验来证实这一点。

空子节点

回忆一下规则 4，它明确指出从根到叶节点或空子节点的每条路径中，必须包含相同数目的黑色节点。空子节点是指非叶节点本可能有，但实际上没有的那个子节点。（即，如果节点只有右子节点，那么它的空缺的左子节点就是空子节点，反之亦然。）因此，在图 9.8 中从节点 50 到 25 再到 25 的右子节点（它是空子节点）的路径只有一个黑色节点，而从根到 6 以及根到 75 的路径上右两

个黑色节点。树的这种排列违背了规则 4，尽管两条到叶节点的路径上的黑色节点数相同。

“黑色高度”是指从根到指定节点路径上的黑色节点数目。图 9.8 中节点 50 的黑色高度为 1，节点 25 的高度也是 1，节点 12 的为 2，以此类推。

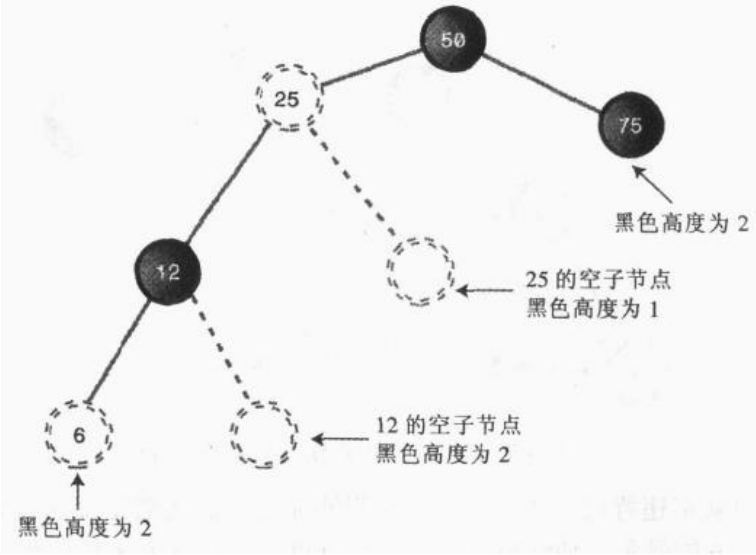


图 9.8 到空子节点的路径

旋 转

为了平衡一棵树，需要重新手动地排列节点。例如，如果所有的节点都在根的左侧，就需要把一些节点移到右侧。这使用旋转来实现。这一节将要学习什么是旋转以及如何执行旋转。旋转必须一次做两件事：

- 使一些节点上升，一些节点下降，帮助树平衡。
- 保证不破坏二叉搜索树的特征。

回想一下二叉搜索树的特征，即任何节点的左子节点及其子树的关键字值都小于该节点，而它的右子节点及其子树的关键字值都大于或等于它。如果旋转不能保证树是一棵合法的二叉搜索树，那它就没有什么用处，因为正如在前面章节已经讲过的，搜索算法的实现依赖于搜索树的这个排列特征。

注意利用颜色规则和节点颜色变换只是有助于决定何时执行旋转。光改变单个节点的颜色不能起到任何作用；旋转才是起关键作用的操作。颜色规则像盖房子时的拇指规则（例如“外面的门向里开”），而旋转则像实际盖房子时所需的锤子和锯。

简单旋转

在试验 2 中做过向左和向右旋转。这些选择都很容易看出来，因为它们只包含三个节点。下面来解释关于这个过程的一些问题。

什么是旋转？

“选择”这个术语可能会产生一点误导。节点自身是不会旋转的；旋转改变的只是节点之间的

关系。选择一个节点作为旋转的“顶端”(top)。如果做一次右旋，这个“顶端”节点将会向下和向右移动到它右子节点的位置。它的左子节点将会上移到它原来的位置上。

注意顶端节点不是旋转的“中心”。如果拿汽车轮胎作比较，顶端节点并不是指轮轴或者轮毂罩：它更像是轮胎胎面最上面的部分。

试验 2 中执行的旋转是以根作为顶端节点的，不过当然任何节点都可以作为旋转中的顶端节点，只要它有适当的子节点。

注意子节点

必须要确保，如果做右旋，顶端节点必须有一个左子节点。否则，将没有节点旋转到顶端节点原来所在的位置。类似的，如果做左旋，顶端节点必须有一个右子节点。

奇异的横向移动节点

旋转远比前面讨论过的三节点的例子要复杂得多。点击 Start，然后节点 50 就已经在根的位置上了，插入下列值的节点，顺序如下：25，75，12，37。

当要插入节点 12 时，可以看到 Can't insert: needs color flip（不能插入：需要颜色变换）的信息提示。点击 Flip 按钮。父节点和子节点的颜色发生改变。然后再次点击 Ins 按钮完成节点 12 的插入。最后，插入节点 37。插入的结果如图 9.9a 所示。

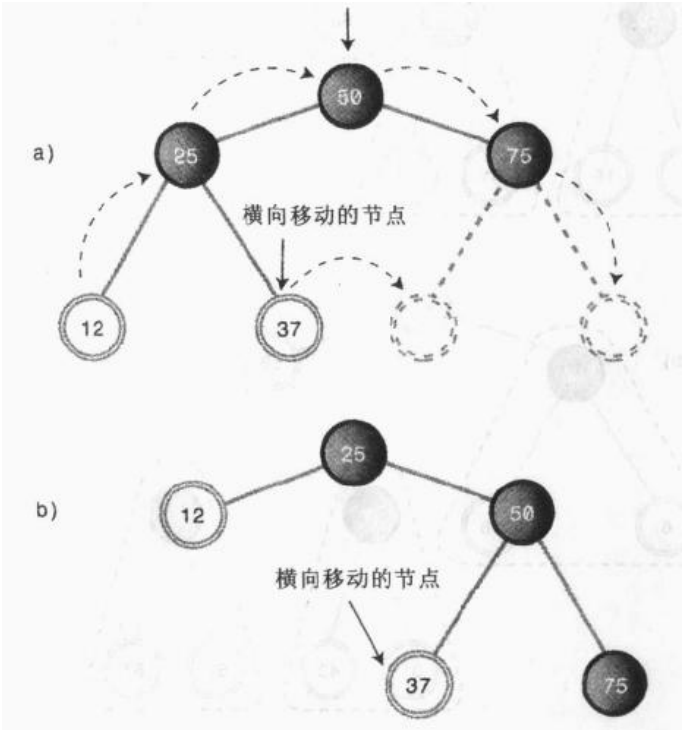


图 9.9 需要横向移动节点的旋转

现在试着做一次旋转。箭头指向根（不要忘了这一点！）然后点击 RoR 按钮。所有的节点都发生了移动。节点 12 跟着节点 25 上升，节点 50 跟着节点 75 下降。

但这是怎么回事？节点 37 虽然是 25 的右子节点，但它断开了和节点 25 的连接，并且取代 25 成为节点 50 的左子节点。一些节点上升，一些节点下降。但是节点 37 却横向移动了。结果如图 9.9b

所示。这次旋转破坏了规则4；后面将会看到如何解决这个问题。

在图9.9a显示的原来位置中，节点37称为顶端节点50的内侧子孙。（节点12是外侧子孙节点。）对内侧子孙节点而言，如果它是上移节点（在右旋中是顶端节点的左子节点）的子节点，它总是要断开和父节点的连接并且重新连接到它以前的祖父节点上。这就好像成为自己的叔叔一样。

移动子树

前面已经讲过了旋转过程中单个节点的位置改变，但是整棵子树也可以发生移动。为了解释这个问题，点击Start按钮，节点50为根，然后按下列顺序插入节点序列：25，75，12，37，62，87，6，18，31，43。只要因为提示信息Can't insert: needs color flip（不能插入，需要颜色变换），而不能完成插入操作时就点击Flip按钮。插入的结果如图9.10a所示。

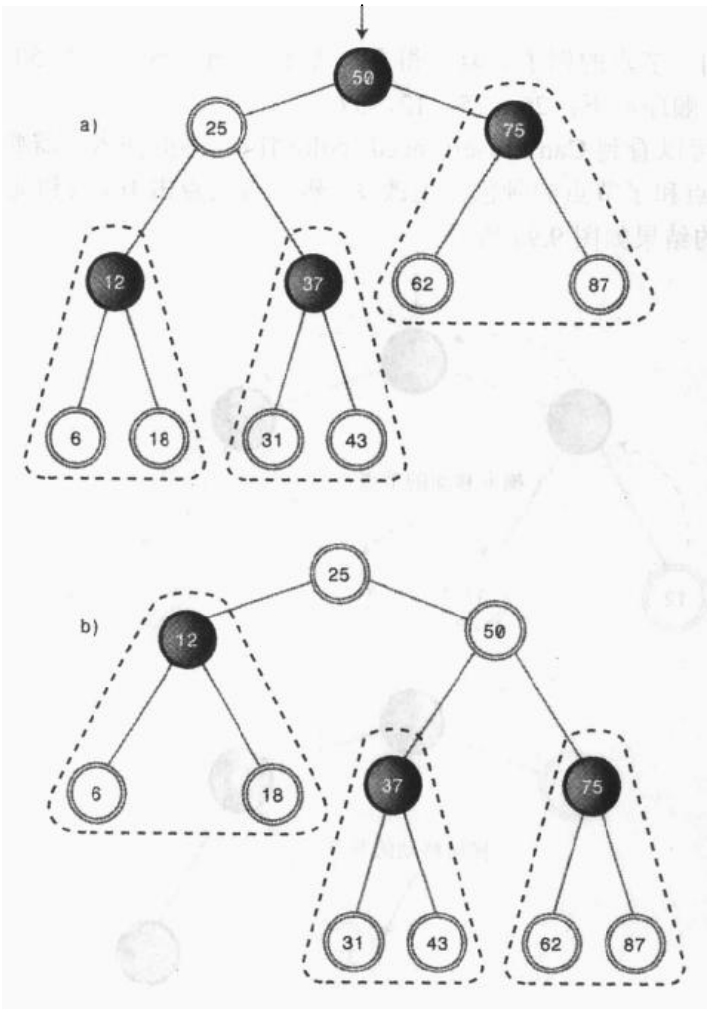


图9.10 旋转过程中的子树移动

箭头指向根节点50。现在点击RoR按钮。那么多节点的位置都发生了改变！结果如图9.10b所示。下面是发生的事情：

- 顶端节点（50）移动到它右子节点的位置。
- 顶端节点的左子节点（25）移动到顶端的位置上。

- 以节点 12 为根的整棵子树都向上移动。
- 以节点 37 为根的整棵子树横向移动，成为节点 50 的左子节点。
- 以节点 75 为根的整棵子树都向下移动。

将会出现提示信息 Error: root must be black（根必须是黑色的），但是这里先暂时地忽略它。可以用箭头指向顶端节点，然后交替地点击 RoR 和 Rol 按钮，这样可以来回地变换。照此操作，同时观察子树的情况，特别是节点 37 为根的子树的情况。

图 9.10 中，用虚线三角形标出了子树。注意旋转不会改变每棵子树中节点间的相互关系。整棵子树作为一个单元整体移动。子树可以比这个例子中所示的三个节点的子树大（有更多的子孙）。不论一棵子树中有多少个节点，在旋转的过程中它们都作为整体一起移动。

人类与计算机

到现在为止，已经讲解了关于旋转所有需要知道的知识。为了执行旋转，要把箭头指向顶端节点，然后点击 RoR 或者 RoL 按钮。当然，在真正的红-黑树插入算法中，旋转操作由程序控制，不需要人为地干预。

然而注意，作为人的一种能力，只需要通过观察树，然后对树作适当的旋转就可以平衡任何树。只要一个节点的左边有很多子孙节点而右边没有这么多节点，就可以向右旋转，反之亦然。

不幸的是，计算机并不善于“只靠观察”这种工作方式。如果计算机能遵守几个简单的规则，它们就会做得更好。这些规则就是在红-黑方法中提供的，表现形式为颜色码和四个红-黑规则。

插入一个新节点

现在已经有了足够的背景知识，可以学习红-黑树的插入算法如何利用旋转和颜色规则来保持树的平衡。

插入过程预览

下面将要简要地介绍本章描述插入过程的方法。这里如果不完全明白也不用担心；马上会更详细地讨论这个过程。

在下面的讨论中，使用 X、P 和 G 表示关联的节点。X 表示违反规则的节点。（有时 X 指一个新插入的节点，有时在父节点和子节点发生红-红颜色冲突时指子节点。）

- X 是一个特殊的节点。
- P 是 X 的父。
- G 是 X 的祖父节点（P 的父节点）。

在顺着树向下查找插入点时，只要发现一个黑色节点有两个红色的子节点（违反规则 2）就执行一次颜色变换。有时颜色变换会造成红-红颜色冲突（违反规则 3）。称红色子节点为 X，红色父节点为 P。只用一次或者两次旋转就可以解决这个冲突，旋转次数由 X 是 G 的外侧子孙节点还是内侧子孙节点来决定。执行了颜色变换和旋转之后，继续向下查找插入点，并且插入新的数据项。

在完成新节点 X 的插入之后，如果 P 是黑色的，只需要连接这个新的红色节点。如果 P 是红色的，则有两种可能性：X 是 G 的外侧子孙节点，或者 X 是 G 的内侧子孙节点。需要两次改变节点颜色（稍后将看到如何改变）。如果 X 是外侧节点，则执行一次旋转，而如果 X 是内侧节点，则执

行两次旋转。这能使树达到平衡状态。

现在要深入讨论一些细节问题。讨论分为三个部分，按复杂程度排列，分别是：

1. 在下行路途中的颜色变换。
2. 插入节点之后的旋转。
3. 在向下路途上的旋转。

如果要严格地按照时间的顺序来讨论这三部分，那应该先讨论第三部分再讨论第二部分。但是，在树底部的旋转比树中间的旋转容易，而且第一部分和第二部分中的操作比第三部分的操作更常用，所以这里先讨论第二部分，然后再讨论第三部分。

在下行路途中的颜色变换

红-黑树的插入例程的开始时所做的事和普通的二叉搜索树所做的基本上一样：沿着从根朝插入点位置走，在每一个节点处通过比较节点的关键字相对大小来决定向左走还是向右走。

但是，在红-黑树中，找到插入点更复杂，因为有颜色变换和旋转。在试验 3 中已经介绍了颜色变换；现在需要更详细地讨论一下。

假设插入例程顺着树向下执行，在每一个节点处向左走或者向右走，查找插入新节点的位置。为了不违反颜色规则，在必要的时候需要进行颜色变换。下面是规则：每当查找例程遇到一个有两个红色子节点的黑色节点时，它必须把子节点变为黑色，而把父节点变为红色（除非父节点为根节点，根总是黑色的）。

颜色变换对红-黑规则有什么影响呢？为了方便起见，称三角形顶端的节点为 P，该节点在颜色变换前是红色节点。这里称 P 的左子节点和右子节点分别为 X1 和 X2。图形如图 9.11a 所示。

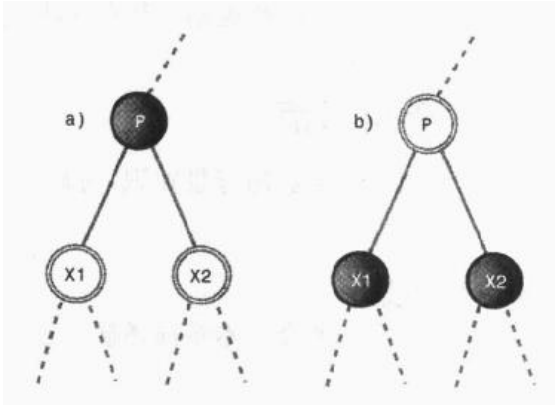


图 9.11 颜色变换

黑色高度不改变

图 9.11b 显示了颜色变换之后节点的情况。这次变换没有改变从根向下过 P 到叶节点或者空节点的路径上的黑色节点的数目。所有这样的路径都经过 P，然后或者经过 X1 或者经过 X2。在颜色变换前，只有 P 是黑色的，所以三角形（由 P、X1 和 X2 组成）在每条这样的路径上增加了一个黑色节点。

在颜色变换之后，P 不再是黑色的了，但是 X1 和 X2 都是黑色的，所以三角形仍然为通过它的每条路径贡献一个黑色节点。因此颜色变换不会造成违背规则 4。

颜色变换是有用的，因为这使红色的叶节点变为黑色的叶节点，因此在不违背规则 3 的情况下连接新的红色节点更容易。

违背规则 3

尽管颜色变换不会违背规则 4，但是可能会违背规则 3（一个节点和它的父节不能都是红色的）。如果 P 的父是黑色的，则 P 由黑色变为红色时不会有任何问题。但是，如果 P 的父节点是红色的，那么在 P 的颜色变化之后，就有两个红色节点相连接了。

这个问题需要在继续向下沿着路径插入新节点之前解决。可以通过旋转修正这个情况，马上就

可以看到。

根的情况

根怎么办呢？记住对根和它的两个子节点做颜色变换时，根和它的子节点一样都是黑色的。这样的颜色变换避免了违背规则 2。这会影响到其他的红-黑规则吗？显然，不会有红色节点连接红色节点的冲突，因为这使更多的节点变为黑色并且没有红色节点了。因此，不会违背规则 3。同时，因为根和它两个子节点中的一个子节点或另一个子节点在各自的路径上，每条路径上的黑色高度增加了相同的数量——都是 1。因此，也不会违背规则 4。

最后，插入节点

在顺着树的路径向下找到合适的位置之后，如果需要在下行路途中执行颜色变换（和旋转），然后就可以插入新的节点，过程和前面那章中描述的普通二叉搜索树一样。但是，这还不是这个问题的结尾。

插入节点之后的旋转

新数据项的插入可能会违背红-黑规则。因此，在插入之后，必须要检测是否违背规则，并采取相应的措施。

记住，正如前面讲过的一样，新插入的称为 X 的节点总是红色的。X 可能插入到相对于 P（X 的父节点）和 G（P 的父节点）不同的位置上，如图 9.12 所示。

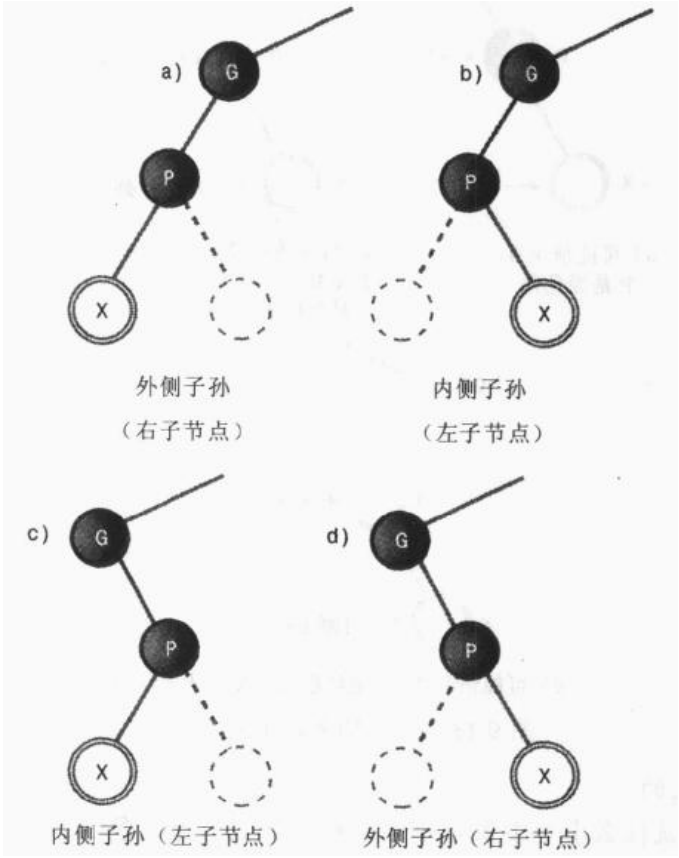


图 9.12　要插入节点的指向的变化

记住如果节点X在P的一侧与P在G的一侧相同，则X就是一个外侧子孙节点。这也就是说，如果节点X是P的左子节点并且P是G的左子节点（图9.12a），或者X是P的右子节点并且P是G的右子节点（图9.12d）时，X是一个外侧子孙节点。相反，如果节点X在P的一侧，而P在G的另一侧，则X就是一个内侧子孙节点（见图9.12b和图9.12c）。

图9.12显示了红黑树“指向”左或右变化的多样性，这是编写插入例程比较困难的原因之一。

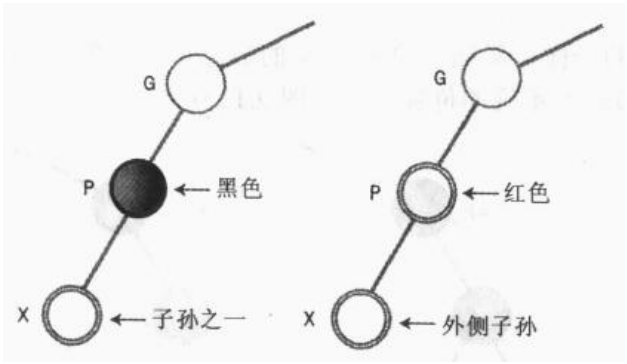
为恢复红-黑规则所采取的措施由颜色以及X和它亲属的布局所决定。可能很让人奇怪，节点只以三种方式排列（上面提到过的指向的变换不算在内）。每种可能性都必须由一种不同的方法处理，以保证红-黑树的正确性以及由此而生成的平衡树。下面将简要地列出这三种可能性，然后分别在各自的小节中详细讨论它们。图9.13显示了它们的情况。记住X总是红色的。

1. P是黑色的。

2. P是红色的，X是G的一个外侧子孙节点。

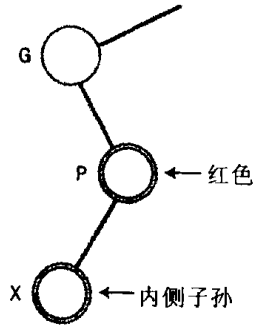
3. P是红色的，X是G的一个内侧子孙节点。

读者可能会认为这并没有包括所有的可能情况。先探讨一下这三种情况，然后再回到这个问题上来。



a）可能情况3：P是黑色的

b）可能情况2：P是红色的，X是外侧子孙



c）可能情况3：P是红色的，X是内侧子孙

图9.13 三种插入后的可能情况

可能性1：P是黑色的

如果P是黑色的，就什么事也不做。刚刚插入的节点总是红色的。如果它的父节点是黑色的，则没有红色节点连接红色节点的冲突（规则3）并且也不会增加黑色节点的数目（规则4）。因此，不会违背颜色规则。这时不需要做其他任何事情。插入完成了。

可能性 2：P 是红色的，X 是 G 的一个外侧子孙节点

如果 P 是红色的并且 X 是一个外侧子孙节点，则需要一次旋转和一些颜色的变化。在 RBTree 专题 applet 中设置这种情况，这样就可以看到正在讨论的情况。具体作法是初始只设有根节点 50，然后插入 25，75 和 12。在插入 12 之前需要做一次颜色变换。

现在插入 6，它是 X，为新节点。图 9.14a 显示了所得到的树。专题 applet 上的信息提示 Error: parent and child both red，所以需要做一些改动。

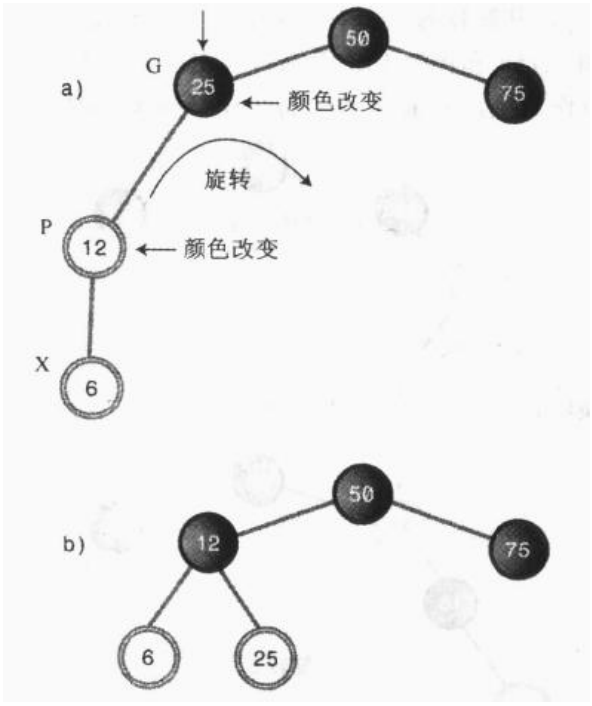


图 9.14　P 是红色的，X 是一个外侧子孙节点

在这种情况下，可以采取三个步骤使树重新符合红-黑规则，并由此使树平衡。下面就是这三个步骤：

1. 改变 X 的祖父节点 G（本例中是 25）的颜色。
2. 改变 X 的父节点 P（12）的颜色。
3. 以 X 的祖父节点 G（25）为顶旋转，向 X（6）上升的方向。在本例中是右旋。

已经讲过，要更改节点颜色，使箭头指向该节点，然后点击 R/B 按钮。要向右旋，使箭头指向顶端节点，然后点击 RoR 按钮。当执行完以上三个步骤之后，专题 applet 会提示 Tree is red/black correct（树是正确的红-黑树。）。树也比以前更趋于平衡了，如图 9.14b 所示。

在这个例子中，X 是一个外侧子孙节点而且是左子节点。X 是外侧子孙节点且为右子节点，是一种与此对称的情况。通过用 50，25，75，87，93 创建树（需要时变换节点颜色）来试验这种情况。通过改变节点 75 和 87 的颜色来修正树，然后以 75 为顶左旋树。树就再次平衡了。

可能性 3：P 是红色的，X 是 G 的一个内侧子孙节点

如果 P 是红色的，X 是内侧子孙节点，则需要两次旋转和一些颜色的改变。为了能实际地看到这种情况，使用 RBTree 专题 applet 创建节点为 50，25，75，12，18 的树。（同样，在插入节点 12

之前需要一次颜色变换。）结果如图 9.15a 所示。

注意节点 18 是一个内侧子孙节点。它和它的父节点都是红色的，所有又看到提示错误的信息 Error: parent and child both red（错误：子节点和父节点均为红色）。

修正这种情况稍微复杂一些。如果像在情况 2 中所做的那样，以祖父节点 G（25）为顶做右旋，内侧子孙节点 X（18）将会横向移动而不是向上移，所以树比以前更不平衡了。（试着这么做一下，然后以 12 为顶旋转它以恢复原来的树型。）因此需要一个不同的解决方法。

当 X 是内侧子孙节点时，其解决技巧是执行两次旋转而不是一次。第一次把 X 由内侧子孙节点变为外侧子孙节点，如图 9.15a 和图 9.15b 所示。现在的情况和可能情况 1 类似了，然后可以应用相同的旋转，用祖父节点作为顶，和前面所做的一样。结果如图 9.15c 所示。

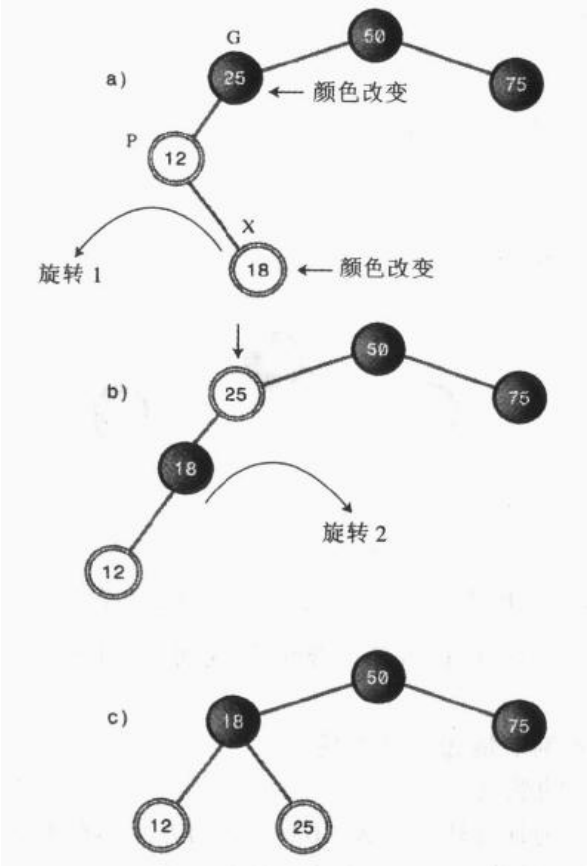


图 9.15 P 是红色的，X 是内侧子孙节点

同时还得重新为节点着色。在做任何旋转之前来做这件事情。（这个顺序其实没有什么关系，但是要是等到旋转完了之后再改变节点的颜色，就很难知道如何称呼它们。）步骤如下：

1. 改变 X 的祖父节点（本例中为 25）的颜色。
2. 改变 X（不是它的父节点；这里 X 是 18）的颜色。
3. 用 X 的父节点 P 作为顶（不是祖父节点；父节点是 12）旋转，向 X 上升的方向旋转（本例中是向左旋转）。
4. 再以 X 的祖父节点（25）为顶旋转，向 X 上升的方向旋转（本例为右旋）。

旋转和改变颜色使树恢复为正确的红-黑树，也（尽可能的）使树平衡。和可能性 2 一样，它也有一个类似的情况，即 P 是 G 的右子节点而不是左子节点。

其他可能情况怎样？

上面讨论的插入操作可能性真的包含了所有的情况吗？

例如，假设，X 有一个兄弟节点 S，它是 P 的另一个子节点。这种情景可能会使插入 X 所需的旋转更加复杂。但是如果 P 是黑色的，插入 X 没有任何问题（这是情况 1）。如果 P 是红色的，它的两个子节点必须都是黑色的（避免违背规则 3）。它不能有单独的一个黑色的子节点 S，因为这样 S 和空子节点的黑色高度会不同。但是，已知 X 是红色的，由此可以得出结论，X 不可能有一个兄弟节点，除非 P 是红色的。

另一种可能性是 P 的父节点 G 有一个子节点 U，U 是 P 的兄弟节点和 X 的叔节点。这种情景可能也会使任何需要的旋转复杂化。但是，如果 P 是黑色的，正如看到的一样，插入 X 时不需要旋转。所以假设 P 是红色的。那么 U 必须也是红色的；否则，从 G 到 P 的黑色高度就和从 G 到 U 的黑色高度不同了。但是有两个红色子节点的黑色父节点在沿着路径向下的时候颜色变换了，所以这种情况也是不存在的。

因此，上面讨论的三种可能性是全部可能存在的情况（除了这样的情况，即在可能性 2 和可能性 3 中，X 可能是右子节点或者左子节点，以及 G 可能是右子节点或者左子节点。）

颜色变换实现了什么

假设执行旋转和适当颜色改变造成了在树的上方出现其他红-黑规则违规情况。可以想像这就必须沿着树一直向上来做各种操作。执行旋转和颜色变化；直到消除所有违规情况。

幸运的是，这种情况不会发生。在下行的路途中使用颜色变换已经消除了旋转造成树的上方任何规则的违规情况。这保证了一次或两次旋转可以使整棵树再次成为正确的红-黑树。它的证明已经超出了本书的范围，但是这种证明是可行的。

在下行路途中的颜色变换使红-黑树的插入效率比其他平衡树，例如 AVL 树的插入效率更高。颜色变换保证了在下行的路途中仅在树上行走了一遍。

在下行路途中的旋转

现在来讨论插入一个节点时三种操作中的最后一种：在下行路途中查找插入点时所做的旋转。前面已经讲过，尽管最后讨论这个操作，但是它实际上发生在节点插入之前。直到现在才讨论它只是因为解释为刚刚插入的节点所作的旋转比解释树中间的节点的旋转容易。

在插入的过程中讨论颜色变换时，已经讲过颜色变换可能会造成对规则 3（父节点和子节点不能都是红色的）的违犯，还讲过旋转可以纠正这种违规的情况。

在向下的路径上有两种旋转的可能性，分别对应前面描述的在插入阶段的可能性 2 和可能性 3。违背规则的节点可能是一个外侧子孙节点，也可能是一个内侧子孙节点。（对应可能性 1 的情况，不需要做任何操作。）

外侧子孙节点

首先来看一个例子，例子中违背规则的节点是一个外侧子孙节点。“违背规则的节点”是指在造成红-红冲突的父-子节点对中的子节点。

从节点 50 开始建一棵新树，插入如下节点：25，75，12，37，6 和 18。在插入 12 和 6 时需要

做颜色变换。

现在要插入值为 3 的节点。注意，必须对节点 12 以及它的子节点 6 和 18 做颜色变换。点击 Flip 按钮。变换执行完毕，但是现在信息提示 Error: parent and child are both red（错误：父节点和子节点都是红色的），这是指 25 和它的子节点 12。此时生成的树如图 9.16a 所示。

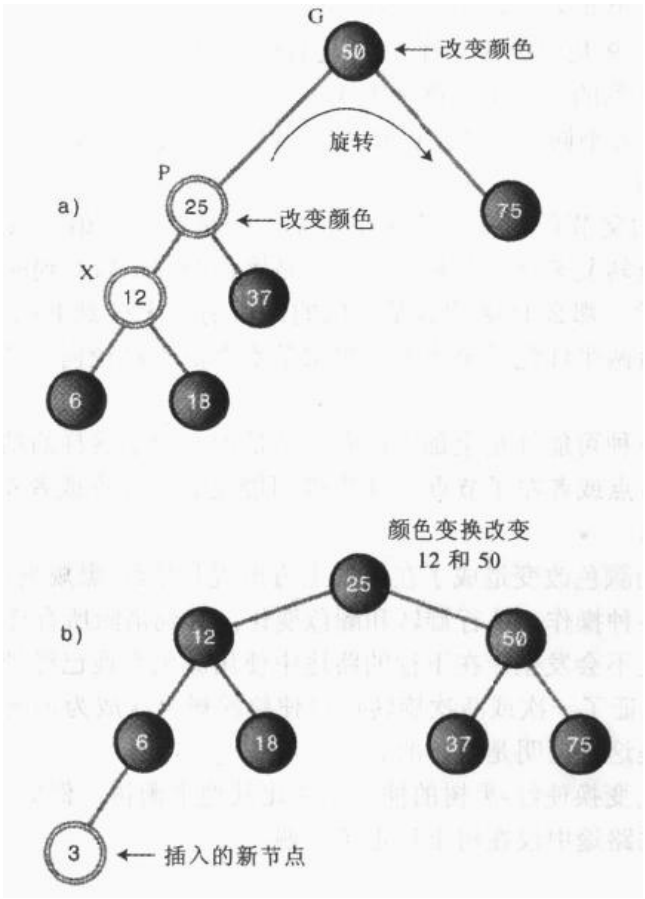


图 9.16 下行路途中的外侧子孙节点

纠正这种违规的过程和前面描述的插后操作中对外侧子孙节点的类似。必须要执行两次颜色改变和一次旋转。因此这里用前面讨论插入节点时的相同的术语，称刚作了颜色变换三角形的顶端节点（本例中是 12）为 X。这似乎有点古怪，因为以前是用 X 表示要插入的节点，但这里它甚至不是一个叶节点。然而，这些在下行路途中的旋转在树中的任何位置都可能发生。

X 的父节点是 P（此例中为 25），X 的祖父节点，即 P 的父节点，是 G（50）。遵守和前面讨论可能情况 2 时相同的那组规则：

1. 改变 X 的祖父节点 G（本例中是 50）的颜色。忽略根必须是黑色的提示信息。
2. 改变 X 的父节点 P（25）的颜色。
3. 以 X 的祖父节点 G（50）为顶旋转，向 X 上升的方向旋转（这里是右旋）。

突然之间，树就平衡了！同时树也令人满意地变成对称的了。这好象是发生了奇迹，但这只是遵循颜色规则的结果。

现在值为 3 的节点可以按通常的方法插入到树中。因为要连接到的节点（6）是黑色的，所以插入一点都不复杂。只需要一次颜色变换（在 50 处）。图 9.16b 显示了插入 3 之后的树。

内侧子孙节点

如果在下行路途中出现红-红冲突时，X 是内侧子孙节点，则需要两次旋转来改正它。这种情况和在前面讲过的插后操作可能性 3 即新插入节点 X 为内侧子孙节点的情况类似。在 RBTree 专题 applet 中点击 Start 创建一棵树，根为 50，插入 25，75，12，37，31 和 43。在插入 12 和 31 之前需要颜色变换。

现在试着插入一个新的节点，值为 28。提醒自己需要颜色变换（节点 37 处）。但当执行颜色变换时，37 和 25 都是红色的，出现提示信息 Error: parent and child are both red（错误：父节点和子节点都是红色的）。不要再点击 Ins 按钮。

此时 G 为 50，P 为 25，X 是 37，如图 9.17a 所示。

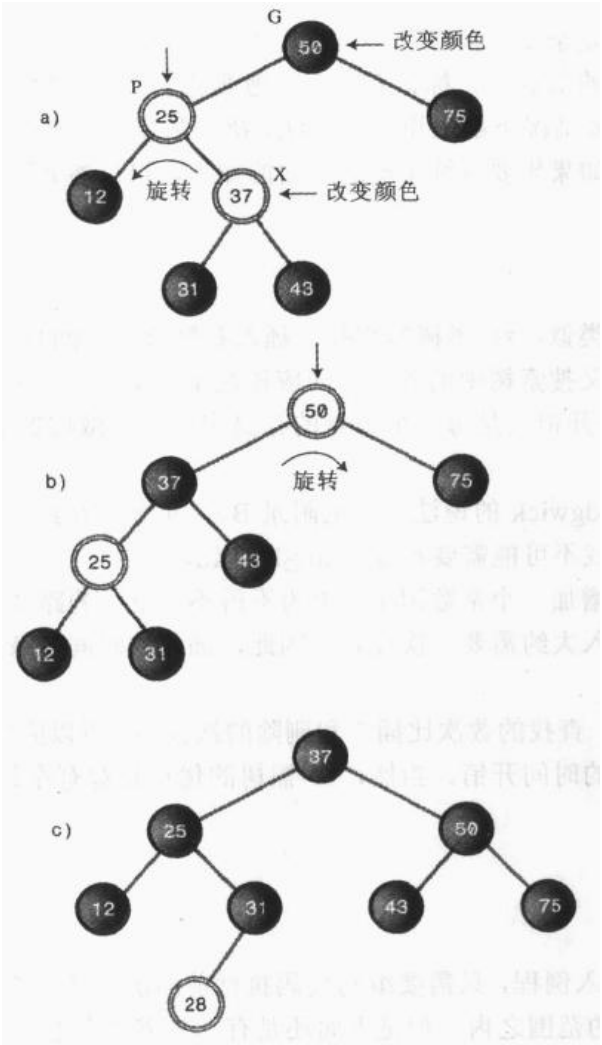


图 9.17　下行路途中的内侧子孙节点

为了解决红-红冲突，必须要做和可能性 3 相同的两次颜色改变和两次旋转：

1. 改变 G（50，忽略根必须是黑色的提示信息）的颜色。
2. 改变 X（37）的颜色。
3. 以 P（25）为顶旋转，向 X 上升的方向旋转（这里是左旋）。结果如图 9.17b 所示。
4. 以 G（50）为顶旋转，向 X 上升的方向旋转（这里是右旋）。

现在可以插入节点 28 了。当插入它颜色变换把 25 和 50 都变为黑色。结果如图 9.17c 所示。

由此可以得出结论，在插入的过程中，保持树的红-黑正确性，因此可以取得树的平衡。

删　除

可以回忆一下，在普通的二叉搜索树中编写删除操作的代码比编写插入操作的代码要难得多。在红-黑树中也是这样的，并且除此之外，可以想像得到删除过程由于需要在节点删除之后恢复树的红-黑正确性，变得更复杂。

实际上，删除过程太复杂了，很多程序员都用不同的方法来回避它。一种方法（和在普通的二叉树中一样）就是为删除的节点做个标记而不实际地删除它。任何找到该节点的查找例程都知道不用报告已找到该节点。很多情况下都应用这种方法，特别是在不经常执行删除操作时。不管怎么说，这里将不讨论删除过程。如果想要继续了解这方面的知识，可以参看附录 B“进一步阅读”。

红-黑树的效率

和一般的二叉搜索树类似，红-黑树的查找、插入和删除的时间复杂度为 $O(\log_2 N)$。在红-黑树中的查找时间和在普通二叉搜索树中的查找时间应该几乎完全一样，因为在查找的过程中并没有应用红-黑树的特征。额外的开销只是每一个节点的存储空间都稍微增加了一点，来存储红-黑的颜色（一个 boolean 变量）。

更为特别的，根据 Sedgwick 的说法（参见附录 B），实际上在红-黑树中的查找大约需要 $\log_2 N$ 次比较，并且也说明了查找不可能需要超过 $2*\log_2 N$ 次比较。

插入和删除的时间要增加一个常数因子，因为不得不在下行的路径上和插入点执行颜色变换和旋转。平均起来，一次插入大约需要一次旋转。因此，插入的时间复杂度还是 $O(\log_2 N)$，但是比在普通的二叉搜索树中要慢。

因为在大多数应用中，查找的数次比插入和删除的次数多，所以应用红-黑树取代普通的二叉搜索树总体上不会增加太多的时间开销。当然，红-黑树的优点是对有序数据的操作不会慢到 O(N)的时间复杂度。

红-黑树的实现

如果编写红-黑树的插入例程，只需要编写代码执行前面所描述的操作。前面已经讲过，显示和描述这样的代码不在本书的范围之内。但是下面还是有一些需要考虑的问题。

需要在 Node 类中增加一个标志红-黑的数据字段（可以是 boolean 类型）。

可以修改第 8 章中 tree.java 程序（清单 8.1）的插入例程。在向下到插入点的路径上，检查当

前节点是否为黑色，以及它的两个子节点是否都是红色的。如果是这样，改变这三个数据项的颜色（除非父节点是根，根总是保持黑色）。

在颜色变换之后，检查确实没有违背规则 3。如果有，执行适当的旋转：对外侧子孙节点旋转一次，对内侧子孙节点旋转两次。

当到达一个叶节点时，像在 tree.java 中一样插入新节点，保证这个节点是红色的。再次检查是否有红-红的冲突，然后执行需要的旋转操作。

读者也许会觉得比较奇怪，程序中不需要跟踪树不同路径的黑色高度（虽然在调试的过程中可能需要查看它）。只需要检查是否违背规则 3，即一个红色父节点有一个红色子节点，若有，当时就可以解决掉（检查黑色高度（规则 4）则不同，它需要更复杂的记录）。

如果执行前面描述的颜色变换、颜色改变以及旋转，节点的黑色高度应当能自动调整，并且树应该保持平衡。RBTree 专题 applet 提示黑色高度有错，只是因为没有强迫用户按正确的步骤完成插入算法。

其他平衡树

AVL 树是最早的一种平衡树。它以发明者的名字命名：Adelson-Velskii 和 Landis。在 AVL 树中每个节点存储一个额外的数据：它的左子树和右子树的高度差。这个差值不会大于 1。这也就是说，节点的左子树的高度和右子树的高度相差不会大于一层。

插入之后，检查新节点插入点所在的最低子树的根。如果它的子节点的高度相差大于 1，执行一次或者两次旋转使它们的高度相等。然后算法向上移动，检查上面的节点，必要时均衡高度。这个检测检查所有路径一直向上，直到根为止。

AVL 树查找的时间复杂度为 O(logN)，因为树一定是平衡的。但是，由于插入（或删除）一个节点时需要扫描两趟树，一次向下查找插入点，一次向上平衡树，AVL 树不如红-黑树效率高，也不如红-黑树常用。

另一个重要的平衡树是多叉树，每个节点可以有两个以上的子节点。本书将介绍多叉树的一种，在下一章中介绍 2-3-4 树。关于多路树的一个问题是每个节点都必须比二叉树的大，因为它需要保存它的每个子节点的引用。

小　结

- 保持二叉搜索树的平衡是非常重要的，这样可以使找到给定节点所必需的时间尽可能短。
- 插入有序的数据将创建最不平衡的树，它查找的时间复杂度为 O(N)。
- 在红-黑平衡的方法中，每个节点都有一个新的特征：它的颜色不是红的就是黑的。
- 一组称为红-黑规则的规则，详细说明了允许排列不同颜色节点的方法。
- 当插入（或删除）一个节点时应用这些规则。
- 一次颜色变换把一个黑色节点和它的两个红色子节点改成一个红色节点和两个黑色子节点。
- 在一次旋转中，指定一个节点为顶端节点。

- 右旋把顶端节点移到它的右子节点的位置，并把顶端节点的左子节点移到顶端节点的位置。
- 左旋把顶端节点移到它的左子节点的位置，并把顶端节点的右子节点移到顶端节点的位置。
- 当顺着树向下查找新节点的插入位置时，应用颜色变换，并且有时应用旋转。颜色变换通过简单的方法，就使树在插入后恢复成正确的红－黑树。
- 新节点插入之后，再次检查红-红冲突。如果发现有违背红-黑规则的现象，执行适当的旋转使树恢复红-黑正确性。
- 这些调整使树成为平衡的，或者至少大致平衡。
- 在二叉树中加入红-黑平衡对平均执行效率只有很小的负面影响，然而却避免了对有序的数据操作的最坏的性能。

问 题

下列问题作为读者的自测题。答案可见附录C。

1. 平衡的二叉搜索树是合乎需要的，因为它避免了在插入_____数据时的性能变慢。
2. 在一棵平衡树中，
   a. 在查找的过程中需要改变树的结构。
   b. 从根到叶子节点的所有路径大约长度都一样。
   c. 所有左子树的高度和所有右子树的高度都一样。
   d. 所有子树的高度都受严格控制。
3. 判断题：红-黑规则在树中重新排列节点使树平衡。
4. 一个空子节点是指
   a. 现在不存在但是下一步将要创建的子节点。
   b. 一个没有自己子节点的子节点。
   c. 一个叶节点的两个可能的子节点中的一个，将会在此插入新节点。
   d. 一个有右子节点的节点的不存在的左子节点（或者反之）。
5. 下列说法那个不是红-黑规则？
   a. 从根到每个叶节点或者到空子节点的每条路径，必须包含相同数目的黑色节点。
   b. 如果一个节点是黑色的，它的两个子节点必须是红色的。
   c. 根总是黑色的。
   d. 以上三条都是合法的规则。
6. 平衡一棵树的两种可能的操作是_____和_____。
7. 新插入的节点总是_____颜色的。
8. 以下说法哪个没有包含在旋转中？
   a. 重新排列节点以恢复二叉搜索树的特征
   b. 改变节点的颜色
   c. 保证遵守红-黑规则

d. 试图使树平衡

9. 一个“横行移动”的节点或者子树以_____开始，然后成为_____，或者反之。

10. 下列说法哪个是错误的？什么时候需要做旋转操作？

a. 插入一个节点之前

b. 插入一个节点之后

c. 查找插入点的过程中

d. 当查找一个给定值的节点时

11. 颜色变换包括改变_____和_____的颜色。

12. 外侧子孙节点是指

a. 它在父节点的一边，而它的父节点在它叔节点的另一边。

b. 它和它的父节点都在自己对应父节点的同一边。

c. 一个右子孙节点的左子孙节点（或者反之亦然）。

d. 它在父节点的一边，而它的兄弟节点在它们祖父节点的另一边。

13. 判断题：当一次旋转紧跟着另一次旋转时，它们旋转的方向相反。

14. 什么时候需要两次旋转？

a. 节点是内侧子孙节点并且父节点是红色的。

b. 节点是内侧子孙节点并且父节点是黑色的。

c. 节点是外侧子孙节点并且父节点是红色的。

d. 节点是外侧子孙节点并且父节点是黑色的。

15. 判断题：在红-黑树中删除时可能需要重新调整树的结构。

实　验

完成这些实验可以帮助深入理解本章的主题。不需要编程实现。

1. 对在红-黑树中插入一个新节点时，遇到的所有节点排列和颜色的情况，以及在每种情况下插入一个新节点应做的操作，画一个活动图或者流程图。

2. 使用 RBTree 专题 applet 建立实验 1 中的所有情况，并遵守插入操作的指令。

3. 做足够多的插入实验，以证明如果严格遵守了红-黑规则 1、2 和 3，那么将自动满足规则 4。

注意

由于这一章中没有列出代码，所以很难留任何的编程作业。如果想要给自己提出挑战，那么就实现红-黑树，可以从第 8 章的 tree.java 程序（清单 8.1）开始着手。

第 10 章

2-3-4 树和外部存储

本章重点

- 2-3-4 树的介绍
- 2-3-4 树的 Java 代码
- 2-3-4 树和红-黑树
- 2-3-4 树的效率
- 2-3 树
- 外部存储

在二叉树中，每个节点有一个数据项，最多有两个子节点。如果允许每个节点可以有更多的数据项和更多的子节点，就是多叉树（multiway tree）。本章第一部分将要介绍的 2-3-4 树，就是多叉树，它的每个节点最多有四个子节点和三个数据项。

因为几个原因，2-3-4 树非常有趣。首先，它像红-黑树一样是平衡树。它的效率比红-黑树稍差，但编程容易。其次，更重要的是，通过学习 2-3-4 树可以更容易地理解 B-树。

B-树是另一种多叉树，专门用在外部存储中来组织数据（外部的意思是主存储的外部；通常是指磁盘驱动器）。B-树中的节点可以有几十或几百个子节点。本章结尾部分会讨论外部存储和 B-树。

2-3-4 树的介绍

本节将介绍 2-3-4 树的特征。在这之后会看到专题 applet 怎样模拟 2-3-4 树以及如何用 Java 语言编写 2-3-4 树的程序。本章还会介绍 2-3-4 树和红-黑树的惊人的相似性。

图 10.1 展示了一棵小 2-3-4 树。每个菱形节点可以保存一个、两个或三个数据项。

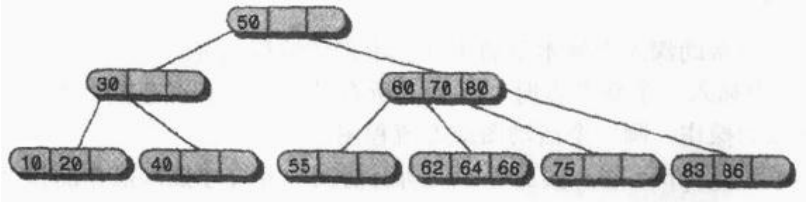


图 10.1　一棵 2-3-4 树

图中上面的三个节点有子节点，底层的六个节点都是叶节点，没有子节点。2-3-4 树中所有的叶节点总是在同一层上。

名字的含义？

2-3-4 树名字中的 2、3 和 4 的含义是指一个节点可能含有的子节点的个数。对非叶节点有三种可能的情况：

- 有一个数据项的节点总是有两个子节点。
- 有两个数据项的节点总是有三个子节点。
- 有三个数据项的节点总是有四个子节点。

简而言之，非叶节点的子节点数总是比它含有的数据项多 1。或者，用符号表示这个规则，设

子节点链接的个数是 L，数据项的个数是 D，那么

L=D+1

这个重要的关系决定了 2-3-4 树的结构。比较来说，叶节点没有子节点，然而它可能含有一个、两个或三个数据项。空节点是不会存在的。

因为 2-3-4 树最多可以有四个子节点的节点，也可以称它为 4 叉树。

为什么不称 2-3-4 树为 1-2-3-4 树呢？它的节点不能像二叉树中那样只有一个子节点吗？因为二叉树的每个节点最多有两个子节点，所以二叉树（第 8 章“二叉树”和第 9 章“红-黑树”讲过的）可以称为二叉的多叉树。但是，二叉树和 2-3-4 树有一点不同（除了节点的最大子节点个数之外）。二叉树中，节点最多有两个子节点的链接。它当然可以只有一个链接，指向它的左子节点或右子节点。它的另一个链接可以是 null 值。然而，在 2-3-4 树中，不允许只有一个链接。有一个数据项的节点必须总是保持有两个链接，除非它是叶节点，在那种情况下没有链接。

如图 10.2 所示。有两个链接的节点称为 2-节点，有三个链接的称为 3-节点，有四个链接的称为 4-节点，但没有称为 1-节点的节点。

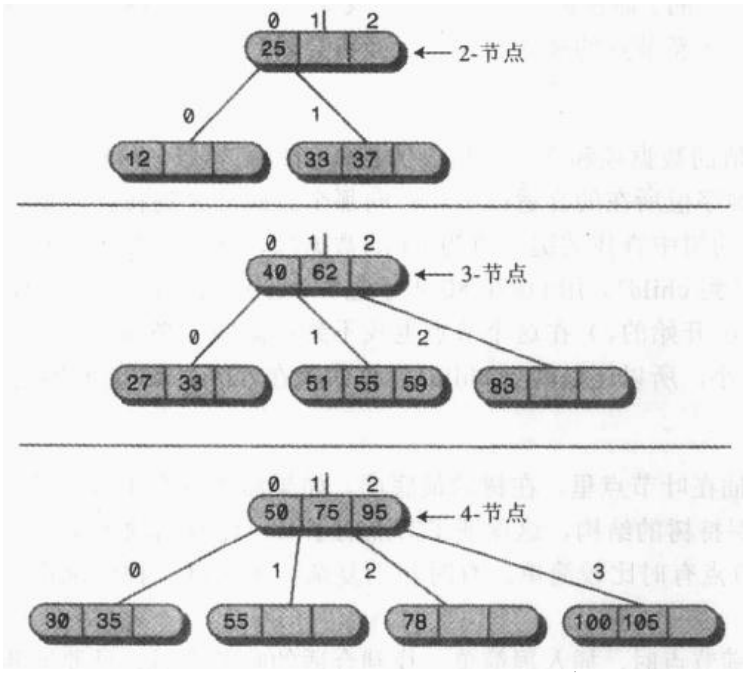


图 10.2 2-3-4 树中的节点

2-3-4 树的组织

为了方便起见，用从 0 到 2 的数字给数据项编号，用 0 到 3 给子节点链编号，如图 10.2 所示。节点中的数据项按关键字值升序排列，习惯上从左到右升序（小数到大数）。

树结构中很重要一点就是它的链与自己数据项的关键字值之间的关系。二叉树中，所有关键字值比某个节点值小的节点都在这个节点左子节点为根的子树上，所有关键字值比某个节点值大的节点都在这个节点右子节点为根的子树上。2-3-4 树中规则是一样的，还加上了以下几点：

- 根是 child0 的子树的所有子节点的关键字值小于 key0。
- 根是 child1 的子树的所有子节点的关键字值大于 key0 并且小于 key1。

- 根是 child2 的子树的所有子节点的关键字值大于 key1 并且小于 key2。
- 根是 child3 的子树所有子节点的关键字值大于 key2。

这种关系如图 10.3 所示。2-3-4 树中一般不允许出现重复关键字值，所以不用考虑比较相同的关键字值的情况。

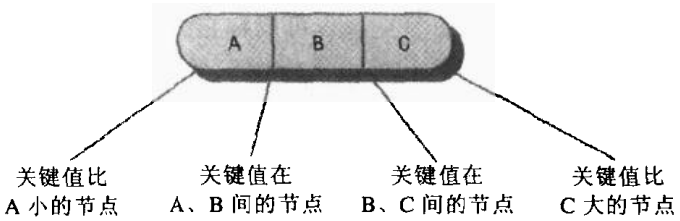


图 10.3 关键字和子节点

回到图 10.1 中的树。所有的 2-3-4 树中，叶节点都在同一层（最底层）。上面层的节点一般都不满；也就是说，它们可能只含有一个或两个数据项，而不是三个。

同样，注意树是平衡的。即使插入一列升序（或降序）排列的数据 2-3-4 树都能保持平衡。2-3-4 树的自我平衡能力取决于新节点的插入方式，后面将会看到。

搜索 2-3-4 树

查找特定关键字值的数据项和在二叉树中的搜索例程相类似。从根开始，除非查找的关键字值就是根，否则选择关键字值所在的合适范围，转向那个方向，直到找到为止。

例如，在图 10.1 的树中查找关键字值为 64 的数据项，从根开始。在根节点查找没有找到。因为 64 比 50 大，所以转到 child1，用 60/70/80 表示它。（记住 child1 是在右边，因为左边按数字标记子节点和链接时是从 0 开始的。）在这个节点也找不到数据项，必须转向下一个子节点。这里，因为 64 比 60 大且比 70 小，所以还是转到 child1。这次就在 62/64/66 节点中找到了 64。

插入

新的数据项总是插在叶节点里，在树的最底层。如果插入到有子节点的节点里，子节点的编号就要发生变化以此来保持树的结构，这保证了节点的子节点比数据项多 1。

2-3-4 树中插入节点有时比较简单，有时相当复杂。无论哪一种情况都是从查找适当的叶节点开始的。

查找时没有碰到满节点时，插入很简单。找到合适的叶节点后，只要把新数据项插入进去就可以了。图 10.4 显示了数据项 18 插入到 2-3-4 树中的情形。

插入可能会涉及到在一个节点中移动一个或者两个其他的数据项，这样在新数据项插入后关键值仍保持正确的顺序。在这个例子里 23 右移为 18 腾出位置。

节点分裂

如果往下寻找要插入位置的路途中，节点已经满了，插入就变得复杂了。发生这种情况时，节点必须分裂（split）。正是这种分裂过程保证了树的平衡。这里讨论的 2-3-4 树的是一种称为自顶向下的（top-down）2-3-4 树，因为是在向下找到插入点的路途中节点发生分裂。

把要分裂节点中的数据项设为 A、B 和 C。下面是分裂时的情况。（假设正在分裂的节点不是根；后面会讲解根的分裂方法。）

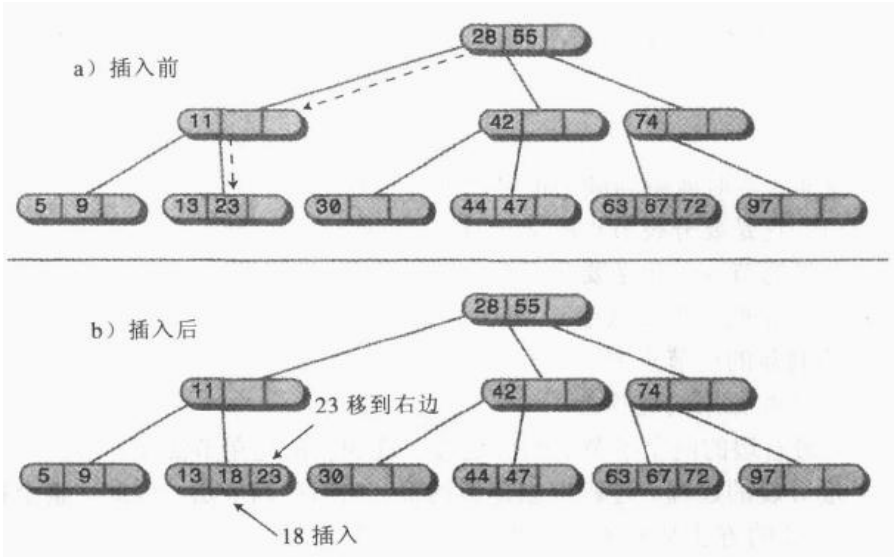


图 10.4　不分裂的插入情况

- 创建一个新的空节点。它是要分裂节点的兄弟，在要分裂节点的右边。
- 数据项 C 移到新节点中。
- 数据项 B 移到要分裂节点的父节点中。
- 数据项 A 保留在原来的位置上。
- 最右边的两个子节点从要分裂节点处断开，连到新节点上。

图 10.5 中显示的是一个节点分裂的例子。另一种描述节点分裂的方法是说 4-节点变成两个 2-节点。

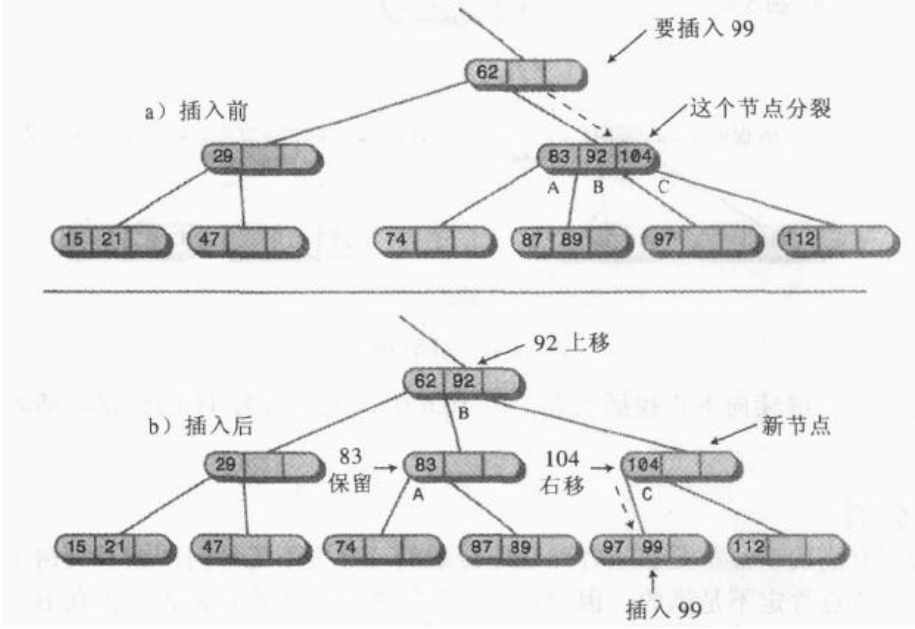


图 10.5　分裂节点

注意节点分裂是把数据向上和向右移动。正是这样的重新排列才可以保持树的平衡。

插入只需要分裂一个节点，除非插入路径上存在不止一个满的节点。这种情况就需要多重分裂。

根的分裂

如果一开始查找插入点时就碰到满的根时，插入过程更复杂一点：

- 创建新的根。它是要分裂节点的父节点。
- 创建第二个新的节点。它是要分裂节点的兄弟节点。
- 数据项 C 移到新的兄弟节点中。
- 数据项 B 移到新的根节点中。
- 数据项 A 保留在原来的位置上。
- 要分裂节点最右边的两个子节点断开连接，连到新的兄弟节点中。

图 10.6 显示了根分裂的过程。过程中创建新的根，比旧的高一层。因此，整个树的高度就增加了 1。另一种描述根分裂的方法是说 4-节点变成三个 2-节点。

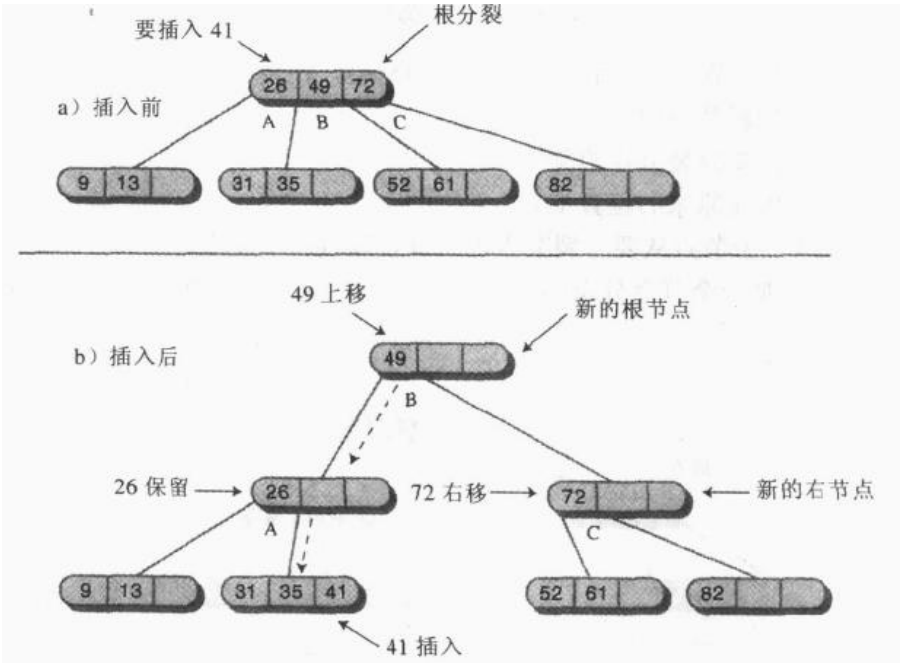


图 10.6 分裂根

顺着分裂的节点，继续向下查找插入点。图 10.6 中，关键值为 41 的数据项插入到合适的叶节点里。

在下行路途中分裂

注意，因为所有满的节点都是在下行路途中分裂的，分裂不可能向回波及到树上面的节点。任何要分裂节点的父节点肯定不是满的，因此该节点不需要分裂就可以插入数据项 B。当然，如果父节点的子节点分裂时它已经有两个子节点了，它就变满了。但是，这只是意味着下次查找碰到它时才需要分裂。

图 10.7 显示的是空树中的一系列插入过程。有四个节点分裂了，两个是根，两个是叶节点。

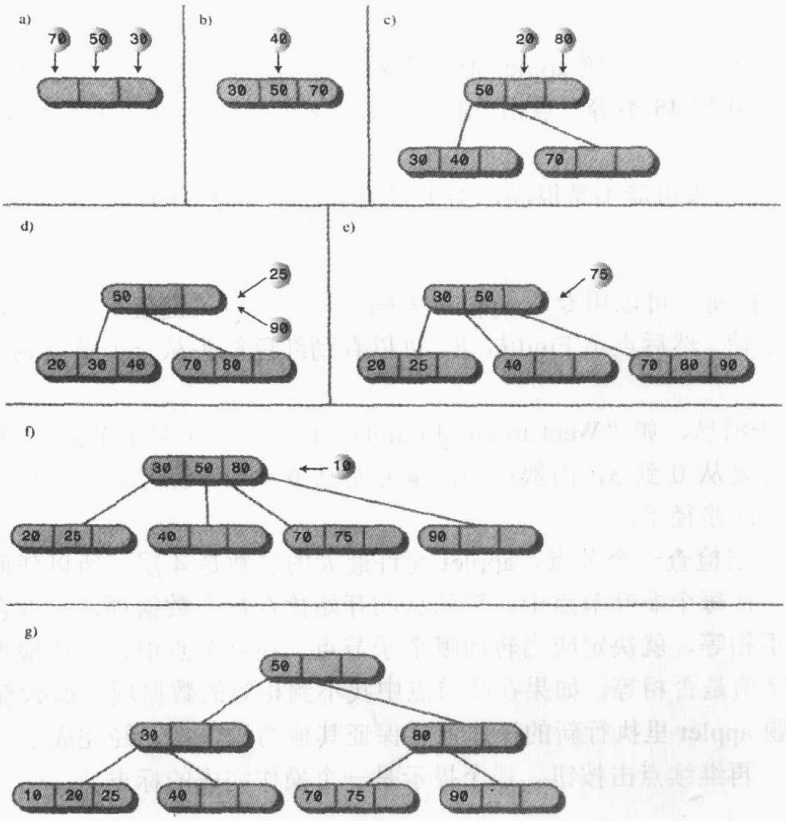


图 10.7 2-3-4 树中的插入

Tree234 专题 applet

操作 Tree234 专题 applet 可以很快地看到 2-3-4 树是怎么工作的。运行专题 applet，就可以看到与图 10.8 相似的屏幕。

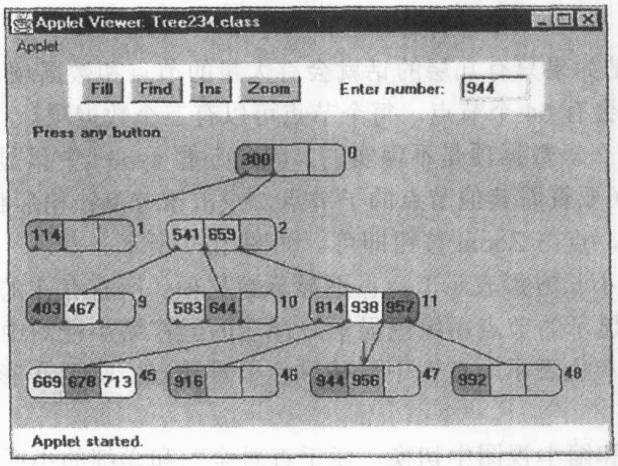


图 10.8 Tree234 专题 applet

Fill 按钮

第一次运行时，Tree234 专题 applet 把 7 个数据项插入到树中。可以用 Fill 按钮创建一棵新的树，数据项的个数从 0 到 45 不等。点击 Fill 按钮，并在提示框中输入一个数字。再点击 Fill，创建新的树。

45 个节点的树看上去可能不是很满，但更多的节点需要更多的层，可能显示不下。

Find 按钮

重复点击 Find 按钮，可以用专题 applet 来确定一个给定关键值字的数据项的位置。在提示框中输入要找的关键字值。然后点击 Find 按钮，可以看到红色箭头从一个节点到另一个节点查找数据项。

同时会显示一些消息，如“Went to child number 1”（转向 1 号子节点）。和前面说的一样，子节点从左到右的编号是从 0 到 3，而数据项的编号是从 0 到 2。练习运行几次 applet 之后，就能够预测出查找将要经历的路径了。

查找需要在每一层检查一个节点。applet 允许最大的层数是 4 层，所以任何数据项最多查找四个节点就可以找到。在每个非叶节点中，算法从左开始检查每个数据项，看看它们和要找的关键字值是否相等，要是不相等，就决定应当转向哪个子节点。在叶节点中，算法检查每个数据项，看看它们和要找的关键字值是否相等。如果在叶节点中找不到相等的数据项，那么查找失败。

在 Tree234 专题 applet 里执行新的操作前要保证其他的操作都已经完成了。等到出现消息提示“Press any button”，再继续点击按钮。那个提示是一个操作完成的标志。

Ins 按钮

Ins 按钮的功能是把文本框中的关键字值作为新的数据项插入到树中。算法首先查找适当的节点。如果遇到已经满的节点，就先分裂该节点，然后再继续下面的事。

实验一下这个插入过程。看看插入点路径中没有满的节点时插入的情况。这个过程很直接而且简单。然后再试试看插入路径末端是满节点的情况，可以发生在根，或者在叶节点，或在根与叶节点中间的某个地方。观察新的节点是如何形成的，被分裂节点中的值是如何分配的。

Zoom 按钮

2-3-4 树的一个问题是如果只有几层的话就会有大量的节点和数据项。Tree234 专题 applet 虽然只允许 4 层，但底层最多会有 64 个节点，每个节点可以有三个数据项。

在一行中一次显示这么多数据项是不现实的，因此专题 applet 中只显示了一部分数据项：只显示被选的节点的子节点。（要查看其他节点的子节点，点击该节点；稍后将讨论这个问题。）为了查看整个树的缩小过的视图，点击 Zoom 按钮即可。如图 10.9 所示。

在这样的视图下，用小长方形表示节点；不显示数据项。所有存在的且在放大视图中（可以再次点击 Zoom 按钮还原）显示的节点用绿色显示。存在的但在缩小视图中暂时不显示的节点用洋红色表示，不存在的节点用灰色表示。在图中这些颜色很难分辨；需要在彩色显示器中观看 applet，搞清显示的意思。

用 Zoom 按钮在放大和缩小视图中切换，便于查看全图和详细的小图，并且希望能在读者的脑子里把两幅图合二为一。

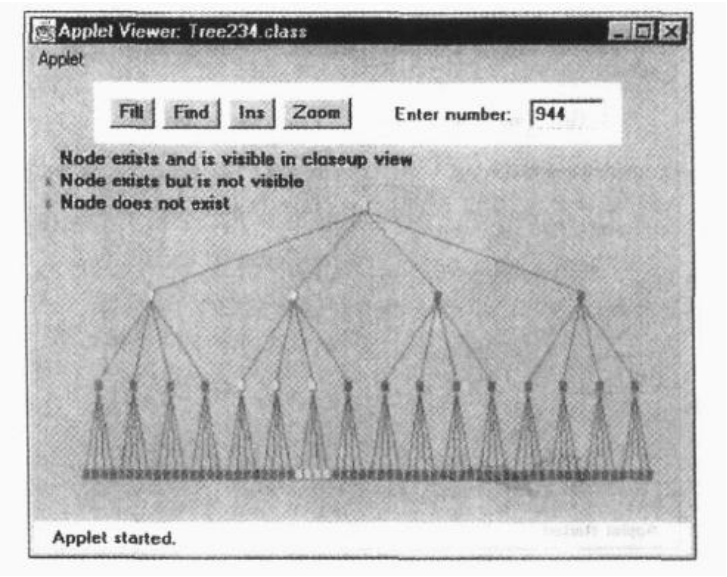


图 10.9 缩小过的视图

察看不同的节点

在放大视图中总是可以看到上面两层的所有节点：顶层只有一个节点，是根，第二层只有四个节点。第二层以下的情况就变得更复杂，因为节点太多不能都显示在屏幕中：第三层有 16 个节点，第四层有 64 个节点。但是，可以通过点击任何一个节点的父节点来查看那个节点，或者有时先点击它的祖先节点，然后再点击它的父节点。

节点底层的蓝色三角形显示的是子节点连接到节点的位置。如果节点的子节点当前可见，就可以看到从蓝色三角形连到子节点的直线。如果子节点当前不可见，就没有显示线，但蓝色三角形还在，表示节点是有子节点的。若点击父节点，它的子节点以及它到子节点的连线就都显示出来了。通过点击适当的节点，可以查看整个树。

为方便起见，把所有的节点编号，从根开始编号为 0，一直到底层最右端的节点编号为 85。号码显示在每个节点的右上角，如图 10.8 所示。无论节点存在与否节点都编了号，所以存在的节点的号码可能不是连续的。

图 10.10 显示的是第三层中有四个节点的一棵小树。用户可以点击节点 1，这样它的两个子节点，号码是 5 和 6 的节点就显示出来了。

如果用户点击节点 2，它的子节点 9 和节点 10 将出现，如图 10.11 所示。

这些图展示了如何通过点击第二层的节点来显示第三层的节点。要显示第四层节点，需要先点击第二层的祖先节点，然后再点击第三层的父节点。

用 Find 按钮和 Ins 按钮进行查找和插入时，视图会自动变化来显示当前红色箭头指向的节点。

实验

Tree234 专题 applet 提供了快捷的方法来学习 2-3-4 树。试向树中插入数据项，观察节点分裂。在将要发生分裂时暂停 applet，计算出分裂节点中的三个数据项的去向。之后再点击 Ins 按钮，检查计算结果是否正确。

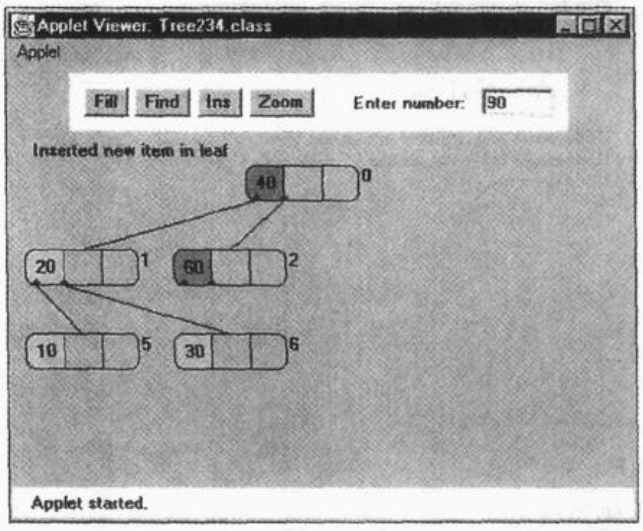


图 10.10 选择最左端的子节点

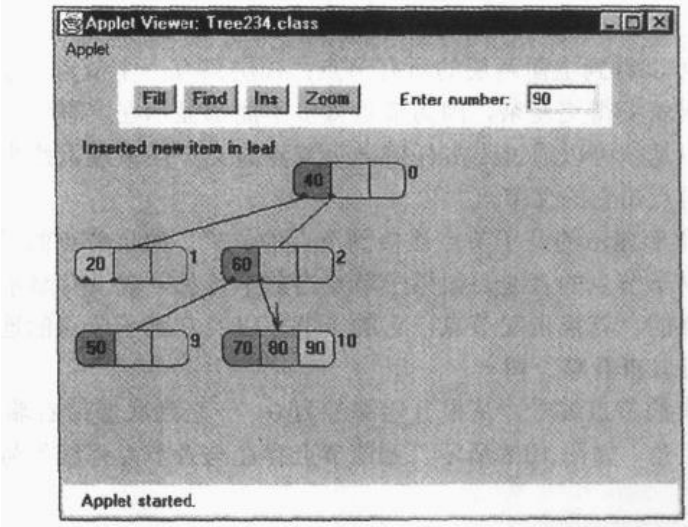


图 10.11 选择最右端的子节点

随着树的增大，需要来回地移动树来查看所有的节点。点击某个节点来查看它的子节点（以及子节点的子节点，等等）。如果找不到查看节点的路径了，可以用 Zoom 按钮来看大图。

可以向树中插入多少个数据项呢？因为树中只能有四层，所以会有限制。四层可以包括 1+4+16+64 个节点，总共有 85 个节点（都可以在缩小图中显示出来）。如果每个节点都是满的，有 3 个数据项，那就有 255 个数据项。但是，节点不能同时都是满的。在节点远没有满之前，另一个根就会分裂出来，数将变成 5 层了，但是这是不可能的，因为 applet 只允许有四层。

可以故意插入合适的数据项，使得路径上没有满的节点，以插入最多的数据项，这样节点就不需要分裂了。当然，对实际的数据来说这是不合理的过程。使用随机的数据可能连 50 个节点都插入不进 applet 去。Fill 按钮只允许 45 个数据项，以便使溢出的可能性最小化。

2-3-4 树的 Java 代码

本节讲述实现 2-3-4 树的 Java 程序。本节最后会显示完整的 tree234.java 程序。这个程序相当复杂，类之间也彼此关联，所以需要详细阅读整个代码清单来明白程序是如何运作的。

程序中有四个类：DataItem、Node、Tree234 和 Tree234App。下面依次来讨论它们。

DataItem 类

DataItem 类的对象表示存储在节点中的数据项。实际生活中每个对象可以表示一个完整的员工档案或目录的内容，但这里只有一条数据，long 类型，与每个 DataItem 对象关联。

这个类对象仅有的方法是初始化方法和显示方法。显示时在数据值前加上一个斜杠：/27。（Node 类中的显示例程会调用这个方法来显示节点中的所有数据项。）

Node 类

Node 类包括两个数组：childArray 和 itemArray。第一个数组有四个数据单元，保存节点可能会有的所有子节点的引用。第二个数组有三个数据单元，保存节点包含的 DataItem 类型的对象的引用。

注意 itemArray 中的数据项组成了有序数组。插入新数据项或删除原来的数据项时，它们还必须继续保持有序的状态（见第 2 章“数组”中的讨论）。数据项需要移位腾出空间来按序插入新数据项，或在删除某个数据项后前移以补上空着的数据单元。

这个类中还有表示节点中当前数据项个数的字段（numItems）和表示节点父节点的字段（parent）。不一定必须设有这些字段，可以删掉它们来缩小节点，不过，留着它们可以使程序更清楚，而且它们只是让节点增大了一点。

Node 类提供了各种通用的方法来管理子节点和父节点的链接，以及检查节点是否为满和是否为叶节点。其实主要的方法有 findItem()、insertItem()和 removeItem()，它们控制节点中的各个数据项。它们在节点中根据给定的关键字值查找数据项；在节点中插入新数据项，根据需要移动存在的数据项；以及删掉数据项，再根据需要移动存在的数据项。不要把这些方法和后面 Tree234 类中的 find()和 insert()方法混淆。

显示方法显示出了节点，用斜杠分割数据项，例如

/27/56/89/，/14/66/，或/45/。

不要忘记 Java 中创建对象时，引用被自动初始化为 null，数字初始化为 0，所以 Node 类不需要构造方法。

Tree234 类

Tree234 类的一个对象表示一棵完整的树。这个类只有一个字段：root，类型是 Node。所有操作都从根开始，因此树的所有需要记录的字段就是根。

查找

根据给定关键字值查找数据项由 find()方法执行。从根开始对每个节点调用节点的 findItem()方法来看数据项是否在那里。如果是的话，它返回数据项在节点的数据项数组中的索引值。

如果 find()方法在叶节点上并且没有找到数据项，查找就失败了，返回－1。如果在当前节点找

不到数据项，并且当前节点不是叶节点，find()调用 getNextChild()方法，找出下一步需要转向节点的那个子节点。

插入

除了找到满的节点就分裂节点之外，insert()方法开始的代码与 find()类似。同样地，它假设没有失败；不断查找，一层一层深入地找下去，直到找到一个叶节点。这时把新的数据项插入到叶节点中去。（叶节点中总会有空间；否则，叶节点就需要分裂。）

分裂

split()方法是这个程序中最复杂的方法。它会将要分裂的节点作为参数传入。首先，最右边的两个数据项从节点中删掉并保存起来。然后断开最右边两个子节点的连接；它们的引用也保存起来。

建立一个新节点，叫 newRight。它将置于被分裂节点的右边。如果要分裂的节点是根，还要再创建一个新节点：新的根节点。

下一步，把要分裂节点连到它父节点的合适位置上去。父节点可能是已经存在的，或如果根分裂时，父节点是新创建的根节点。设要分裂节点中的三个数据项是 A、B 和 C。数据项 B 插入到它的父节点中。如果有必要的话，父节点中已经存在的子节点先断开连接，再连接到右移一位的新的位置上，为新的数据项和新的连接腾出位置。newRight 节点连接到它的父节点上去。（参考图 10.5 和 10.6。）

现在关注 newRight 节点。数据项 C 插入到这个节点里，child2 和 child3——刚才从要分裂节点处断开的两个子节点，连接到这个节点上。分裂就完成了，split()例程返回。

Tree234App 类

在 Tree234App 类中，main()方法把一些数据项插入到树中去。它还提供了基于字符的用户接口，用户可以输入 s 来显示树，输入 i 来插入新的数据项，输入 f 来查找已经存在的数据项。下面是某个程序交互的例子：

```
Enter first letter of show, insert, or find: s
level=0 child=0 /50/
level=1 child=0 /30/40/
level=1 child=1 /60/70/

Enter first letter of show, insert, or find: f
Enter value to find: 40
Found 40

Enter first letter of show, insert, or find: i
Enter value to insert: 20
Enter first letter of show, insert, or find: s
level=0 child=0 /50/
level=1 child=0 /20/30/40/
level=1 child=1 /60/70/

Enter first letter of show, insert, or find: i
Enter value to insert: 10
```

```
Enter first letter of show, insert, or find: s
level=0 child=0 /30/50/
level=1 child=0 /10/20/
level=1 child=1 /40/
level=1 child=2 /60/70/
```

输出并不是很直观，但已经显示出了足够的信息可以画出树来。如上所示，首先是层数，根为0层，后面是子节点的编号。显示算法是按照深度优先的顺序，所以先显示出根，然后是根的第一个子节点，和以第一个子节点为根的子树，之后是第二个子节点和它的子树，依次类推。

上文输出的界面表示要插入两个数据项：20和10。第二个插入的数据项引起节点（根的child0）分裂。图10.12展示了插入完成后的树，是最后输入‘s’键的结果。

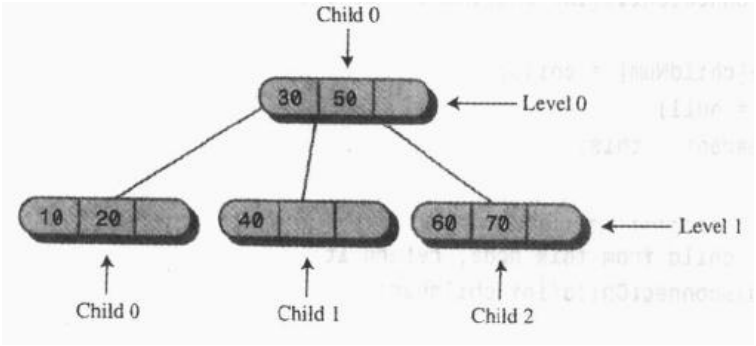


图 10.12　tree234.java 程序的输出例子

完整的 tree234.java 程序

清单10.1是完整的tree234.java程序，包括刚刚讨论过的所有的类。像大多数面向对象程序一样，最容易的方法是先查看上层主要的类，然后再进一步关注面向细节的类。程序中这样的顺序是Tree234App、Tree234、Node、DataItem。

清单 10.1　tree234.java 程序

```
// tree234.java
// demonstrates 234 tree
// to run this program: C>java Tree234App
import java.io.*;
////////////////////////////////////////////////////////////////
class DataItem
   {
   public long dData;          // one data item
//--------------------------------------------------------------
   public DataItem(long dd)    // constructor
      { dData = dd; }
//--------------------------------------------------------------
   public void displayItem()   // display item, format "/27"
      { System.out.print("/"+dData); }
//--------------------------------------------------------------
```

```
   } // end class DataItem
////////////////////////////////////////////////////////////////
class Node
   {
   private static final int ORDER = 4;
   private int numItems;
   private Node parent;
   private Node childArray[] = new Node[ORDER];
   private DataItem itemArray[] = new DataItem[ORDER-1];
// -------------------------------------------------------------
   // connect child to this node
   public void connectChild(int childNum, Node child)
      {
      childArray[childNum] = child;
      if(child != null)
         child.parent = this;
      }
// -------------------------------------------------------------
   // disconnect child from this node, return it
   public Node disconnectChild(int childNum)
      {
      Node tempNode = childArray[childNum];
      childArray[childNum] = null;
      return tempNode;
      }
// -------------------------------------------------------------
   public Node getChild(int childNum)
      { return childArray[childNum]; }
// -------------------------------------------------------------
   public Node getParent()
      { return parent; }
// -------------------------------------------------------------
   public boolean isLeaf()
     { return (childArray[0]==null) ? true : false; }
// -------------------------------------------------------------
   public int getNumItems()
     { return numItems; }
// -------------------------------------------------------------
   public DataItem getItem(int index)   // get DataItem at index
      { return itemArray[index]; }
// -------------------------------------------------------------
   public boolean isFull()
      { return (numItems==ORDER-1) ? true : false; }
// -------------------------------------------------------------
   public int findItem(long key)       // return index of
```

```
      {                                  // item (within node)
      for(int j=0; j<ORDER-1; j++)       // if found,
         {                               // otherwise,
         if(itemArray[j] == null)        // return -1
            break;
         else if(itemArray[j].dData == key)
            return j;
         }
      return -1;
      } // end findItem
// -------------------------------------------------------------
   public int insertItem(DataItem newItem)
      {
      // assumes node is not full
      numItems++;                        // will add new item
      long newKey = newItem.dData;       // key of new item

      for(int j=ORDER-2; j>=0; j--)      // start on right,
         {                               //    examine items
         if(itemArray[j] == null)        // if item null,
            continue;                    // go left one cell
         else                            // not null,
            {                            // get its key
            long itsKey = itemArray[j].dData;
            if(newKey < itsKey)          // if it's bigger
               itemArray[j+1] = itemArray[j]; // shift it right
            else
               {
               itemArray[j+1] = newItem; // insert new item
               return j+1;               // return index to
               }                         //    new item
            } // end else (not null)
         } // end for                    // shifted all items,
      itemArray[0] = newItem;            // insert new item
      return 0;
      } // end insertItem()
// -------------------------------------------------------------
   public DataItem removeItem()        // remove largest item
      {
      // assumes node not empty
      DataItem temp = itemArray[numItems-1];  // save item
      itemArray[numItems-1] = null;           // disconnect it
      numItems--;                             // one less item
      return temp;                            // return item
      }
```

```
// --------------------------------------------------------
   public void displayNode()           // format "/24/56/74/"
      {
      for(int j=0; j<numItems; j++)
         itemArray[j].displayItem();   // "/56"
      System.out.println("/");         // final "/"
      }
// --------------------------------------------------------
   } // end class Node
////////////////////////////////////////////////////////////////
class Tree234
   {
   private Node root = new Node();            // make root node
// --------------------------------------------------------
   public int find(long key)
      {
      Node curNode = root;
      int childNumber;
      while(true)
         {
         if(( childNumber=curNode.findItem(key) ) != -1)
            return childNumber;               // found it
         else if( curNode.isLeaf() )
            return -1;                        // can't find it
         else                                 // search deeper
           curNode = getNextChild(curNode, key);
         } // end while
      }
// --------------------------------------------------------
   // insert a DataItem
   public void insert(long dValue)
      {
      Node curNode = root;
      DataItem tempItem = new DataItem(dValue);

      while(true)
         {
         if( curNode.isFull() )               // if node full,
            {
            split(curNode);                   // split it
            curNode = curNode.getParent();    // back up
                                              // search once
            curNode = getNextChild(curNode, dValue);
            } // end if(node is full)
```

```
         else if( curNode.isLeaf() )          // if node is leaf,
            break;                            // go insert
         // node is not full, not a leaf; so go to lower level
         else
            curNode = getNextChild(curNode, dValue);
         } // end while

      curNode.insertItem(tempItem);       // insert new DataItem
      } // end insert()
// -------------------------------------------------------------
   public void split(Node thisNode)     // split the node
      {
      // assumes node is full
      DataItem itemB, itemC;
      Node parent, child2, child3;
      int itemIndex;

      itemC = thisNode.removeItem();    // remove items from
      itemB = thisNode.removeItem();    // this node
      child2 = thisNode.disconnectChild(2); // remove children
      child3 = thisNode.disconnectChild(3); // from this node
      Node newRight = new Node();       // make new node

      if(thisNode==root)                // if this is the root,
         {
         root = new Node();                // make new root
         parent = root;                    // root is our parent
         root.connectChild(0, thisNode);   // connect to parent
         }
      else                              // this node not the root
         parent = thisNode.getParent();    // get parent

      // deal with parent
      itemIndex = parent.insertItem(itemB); // item B to parent
      int n = parent.getNumItems();         // total items?

      for(int j=n-1; j>itemIndex; j--)          // move parent's
         {                                      // connections
         Node temp = parent.disconnectChild(j); // one child
         parent.connectChild(j+1, temp);        // to the right
         }
                                 // connect newRight to parent
      parent.connectChild(itemIndex+1, newRight);

      // deal with newRight
```

```
      newRight.insertItem(itemC);       // item C to newRight
      newRight.connectChild(0, child2); // connect to 0 and 1
      newRight.connectChild(1, child3); // on newRight
      } // end split()
// -------------------------------------------------------------
   // gets appropriate child of node during search for value
   public Node getNextChild(Node theNode, long theValue)
      {
      int j;
      // assumes node is not empty, not full, not a leaf
      int numItems = theNode.getNumItems();
      for(j=0; j<numItems; j++)          // for each item in node
         {                               // are we less?
         if( theValue < theNode.getItem(j).dData )
            return theNode.getChild(j);  // return left child
         } // end for                    // we're greater, so
      return theNode.getChild(j);        // return right child
      }
// -------------------------------------------------------------
   public void displayTree()
      {
      recDisplayTree(root, 0, 0);
      }
// -------------------------------------------------------------
   private void recDisplayTree(Node thisNode, int level,
                                          int childNumber)
      {
      System.out.print("level="+level+" child="+childNumber+" ");
      thisNode.displayNode();              // display this node

      // call ourselves for each child of this node
      int numItems = thisNode.getNumItems();
      for(int j=0; j<numItems+1; j++)
         {
         Node nextNode = thisNode.getChild(j);
         if(nextNode != null)
            recDisplayTree(nextNode, level+1, j);
         else
            return;
         }
      } // end recDisplayTree()
// -------------------------------------------------------------
   } // end class Tree234
////////////////////////////////////////////////////////////////
class Tree234App
```

```
{
public static void main(String[] args) throws IOException
   {
   long value;
   Tree234 theTree = new Tree234();

   theTree.insert(50);
   theTree.insert(40);
   theTree.insert(60);
   theTree.insert(30);
   theTree.insert(70);
    while(true)
       {
       System.out.print("Enter first letter of ");
       System.out.print("show, insert, or find: ");
       char choice = getChar();
       switch(choice)
          {
          case 's':
             theTree.displayTree();
             break;
          case 'i':
             System.out.print("Enter value to insert: ");
             value = getInt();
             theTree.insert(value);
             break;
          case 'f':
             System.out.print("Enter value to find: ");
             value = getInt();
             int found = theTree.find(value);
             if(found != -1)
                System.out.println("Found "+value);
             else
                System.out.println("Could not find "+value);
             break;
          default:
             System.out.print("Invalid entry\n");
          }  // end switch
       }  // end while
    }  // end main()
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
```

```
        String s = br.readLine();
        return s;
        }
//--------------------------------------------------------------
   public static char getChar() throws IOException
        {
        String s = getString();
        return s.charAt(0);
        }

//--------------------------------------------------------------
   public static int getInt() throws IOException
        {
        String s = getString();
        return Integer.parseInt(s);
        }
//--------------------------------------------------------------
   } // end class Tree234App
////////////////////////////////////////////////////////////////
```

2-3-4 树和红-黑树

2-3-4 树和红-黑树（第 9 章讲的）看上去可能完全不同。但是，在某种意义上它们又是完全相同的。一个可以通过应用一些简单的规则变成另一个，而且使它们保持平衡的操作也是一样的。数学上称它们为同构的。

可能在实际应用时不需要把 2-3-4 树转变成红-黑树，但这种结构的等价性可以更清楚地表明它们的操作，并可以用于分析它们的效率。

在历史上，2-3-4 树先被提出；后来红-黑树由它发展而来。

2-3-4 树转变为红-黑树

应用三条规则可以把 2-3-4 树转化成红-黑树，如图 10.13 所示：

- 把 2-3-4 树中的每个 2-节点转化成红-黑树的黑色节点，如图 10.13a 所示。
- 把每个 3-节点转化成一个子节点和一个父节点，如图 10.13b 所示。子节点有两个自己的子节点：W 和 X 或 X 和 Y。父节点有另一个子节点：Y 或 W。哪个节点变成子节点或父节点都无所谓。子节点涂成红色，父节点涂成黑色。
- 把每个 4-节点转化成一个父节点和两个子节点，如图 10.13c 所示。第一个子节点有它自己的子节点 W 和 X；第二个子节点拥有子节点 Y 和 Z。和前面一样，子节点涂成红色，父节点涂成黑色。

图 10.14 展示了一棵 2-3-4 树和应用这些规则转化后对应的红-黑树。虚线环绕的子树是由 3-节点和 4-节点变成的。转化后自动符合红黑树的规则。可以检验一下：两个红节点不会相连，每条从根到叶节点（或空子节点）的路径上黑节点的个数都是一样的。

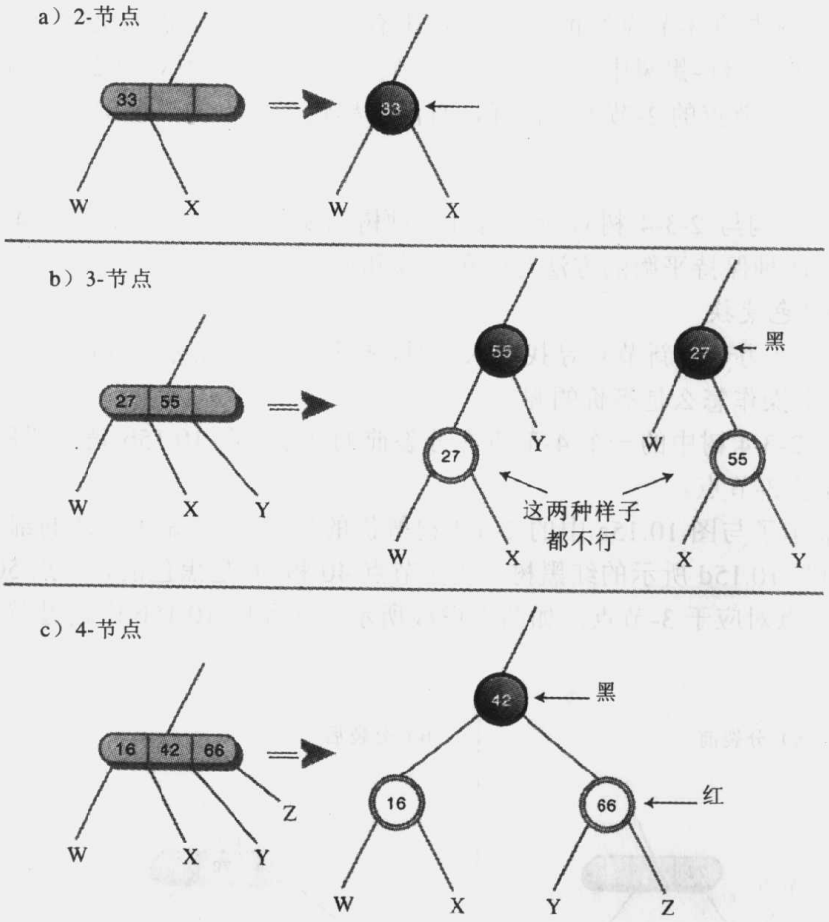


图 10.13　转换：2-3-4 树到红-黑树

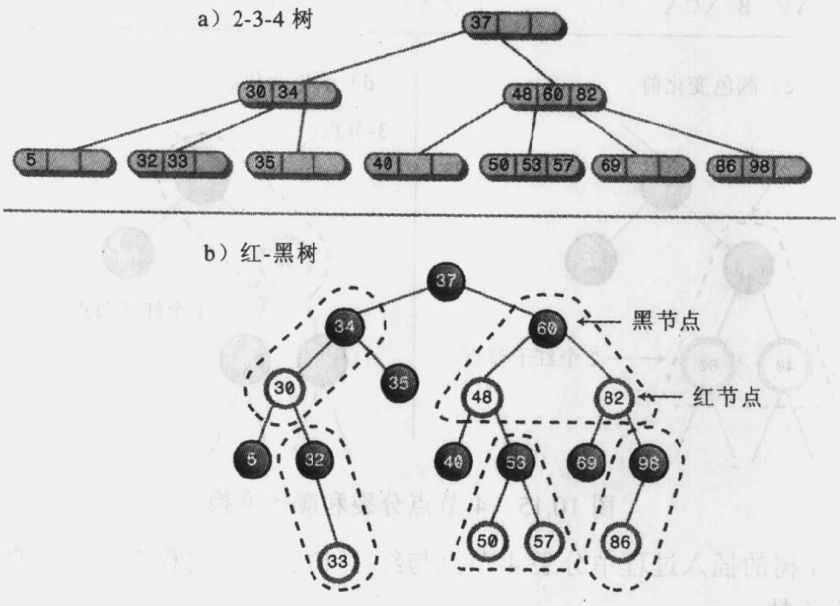


图 10.14　一棵 2-3-4 树和与它等价的红-黑树

可以说，2-3-4 树中的 3-节点等价于红-黑树中有一个红色子节点的父节点，4-节点等价于有两个红色子节点的父节点。红-黑树中有一个黑色子节点的黑色节点不对应 2-3-4 树中的 3-节点；它只表示有另一个 2-节点子节点的 2-节点。同样，有两个黑色子节点的黑色父节点也不代表 4-节点。

操作的等价性

不仅红-黑树的结构与 2-3-4 树对应，而且两种树的操作也是一样的。2-3-4 树中用节点分裂保持平衡。红-黑树中两种保持平衡的方法是颜色交换和旋转。

4-节点分裂和颜色变换

在 2-3-4 树中向下为一个新节点寻找插入点时，把每个 4-节点分裂成两个 2-节点。红-黑树中利用颜色变换。这两个操作怎么是等价的呢？

图 10.15a 中是 2-3-4 树中的一个 4-节点在分裂前的样子；图 10.15b 是分裂后的样子。4-节点的父节点 2-节点变成了 3-节点。

图 10.15c 中展示了与图 10.15a 中的 2-3-4 树等价的红-黑树。虚线环绕的部分等价于 4-节点。颜色变化后得到如图 10.15d 所示的红黑树。现在节点 40 和 60 是黑色的，节点 50 是红色的。因此，节点 50 和它的父节点对应于 3-节点，如图中虚线所示。这和图 10.15b 中通过节点分裂得到的 3-节点是一样的。

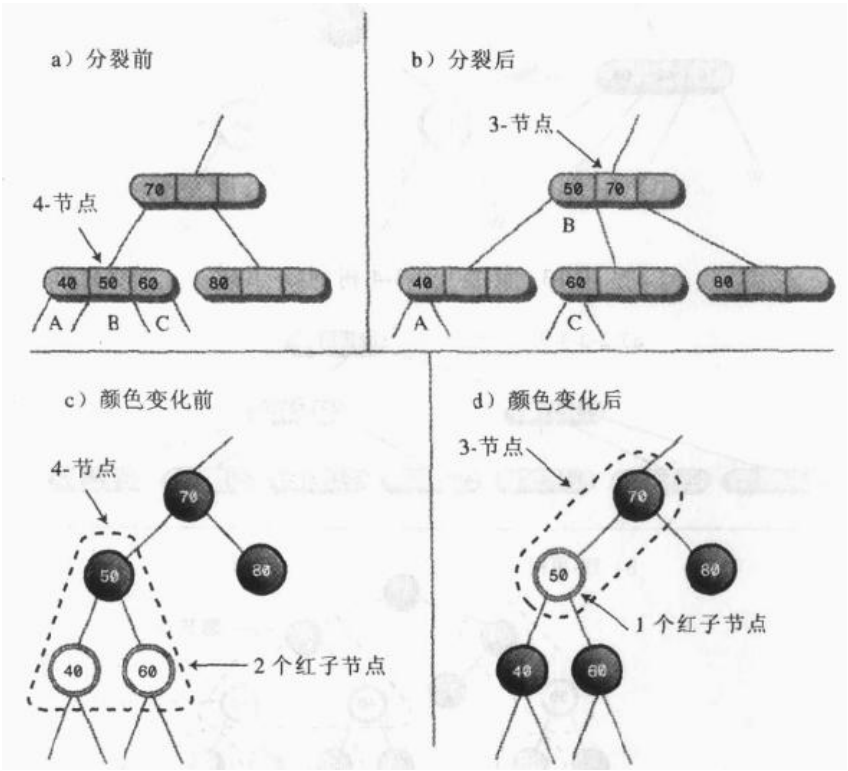


图 10.15　4-节点分裂和颜色变换

因此，在 2-3-4 树的插入过程中分裂 4-节点与红-黑树的插入过程中的颜色变换是等价的。

3-节点分裂和旋转

2-3-4 树中的 3-节点要转化成等价的红-黑树中的节点时，有两种可能的排列方式，如前面的图

10.13b 所示。两个数据项中的任何一个都可以成为父节点。选定一个做父节点后，子节点可以是左子节点或者是右子节点，就是说连接父节点和子节点的斜线可以向左斜或向右斜。

这两种排列都是允许的；但是它们对保持树的平衡来说作用可能不一样。在稍大的范围内来讨论这个问题。

图 10.16a 中是一棵 2-3-4 树，图 10.16b 和 10.16c 中是两棵应用转换规则从 2-3-4 树得到的等价红-黑树。它们之间的不同是选择 3-节点的两个数据项中的哪个来做父节点：在图 b 中是 80 做父节点；在图 c 中 70 做父节点。

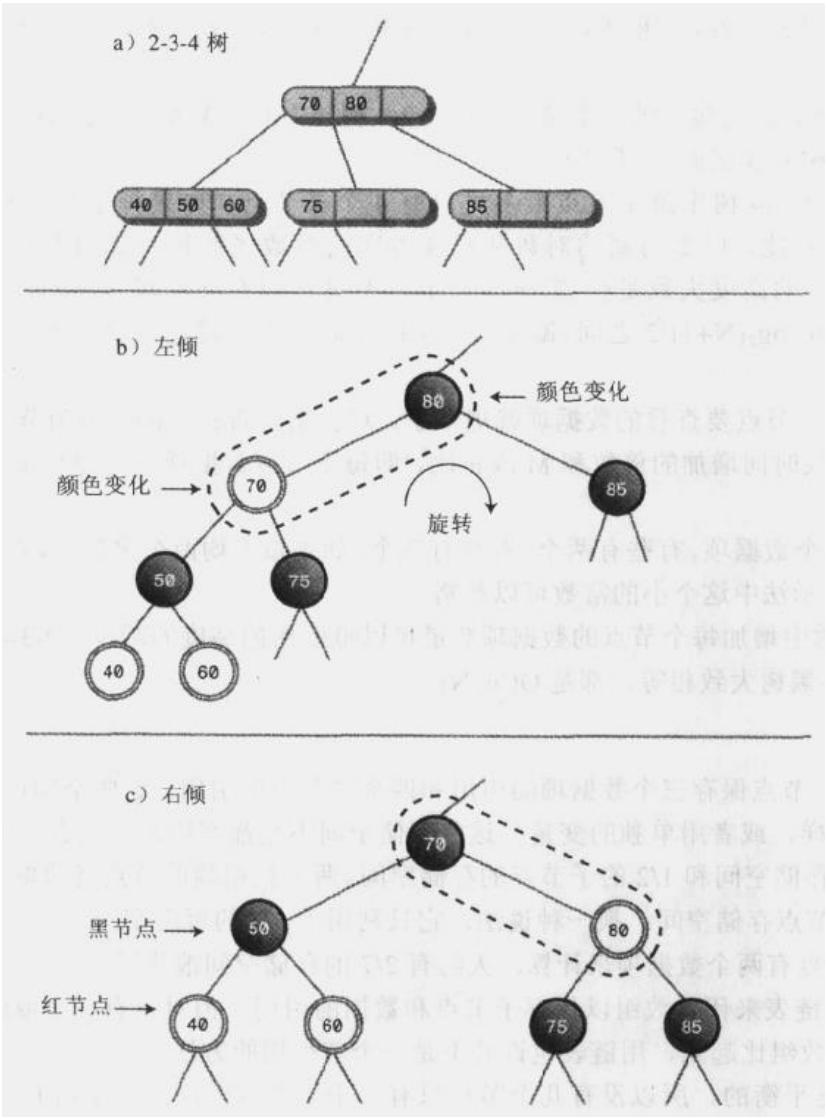


图 10.16　3-节点和旋转

虽然这些排列方法都是允许的，但可以看到 b）中的树是不平衡的，c）中则是平衡的。对 b）中的红-黑树，可以向右旋转（并进行颜色变幻）来保持平衡。令人惊异的是，旋转得到的树和 c）中的树是一样的。

因此，红-黑树中的旋转和把 2-3-4 树转换成红-黑树时选择哪个节点做父节点是等价的。虽然

没有提及，但对子孙节点中需要进行两次旋转时，也是等价的。

2-3-4 树的效率

分析 2-3-4 树的效率比红-黑树要困难，但还是可以从两者的等价性开始分析。

速度

如第 8 章所示，查找过程中红-黑树每层都要访问一个节点，可能是查找已经存在的节点，也可能是插入一个新节点。红-黑树的层数（平衡二叉树）大约是 $\log_2(N+1)$，所以搜索时间与层数成比例。

2-3-4 树中同样要访问每层的一个节点，但 2-3-4 树比有相同数据项的红-黑树短（层数少）。参考图 10.14，2-3-4 树有 3 层而红-黑树有 5 层。

更特别的是，2-3-4 树中每个节点最多可以有 4 个子节点。如果每个节点都是满的，树的高度应该和 $\log_4 N$ 成正比。以 2 为底的对数和以 4 为底的对数底数相差 2。因此，在所有节点都满的情况下，2-3-4 树的高度大致是红-黑树的一半。不过它们不可能都是满的，2-3-4 树的高度就大致在 $\log_2(N+1)$和 $\log_2(N+1)/2$ 之间。减少 2-3-4 树的高度可以使它的查找时间比红-黑树的短一些。

另一方面，每个节点要查看的数据项就更多了，这会增加查找时间。因为节点中用线性搜索来查看数据项，使查找时间增加的倍数和 M 成正比，即每个节点数据项的平均数量。总的查找时间和 $M*\log_4 N$ 成正比。

有些节点有一个数据项，有些有两个，有些有三个。如果按平均两个来计算，查找时间和 $2*\log_4 N$ 成正比。在大 O 表示法中这个小的常数可以忽略。

因此，2-3-4 树中增加每个节点的数据项数量可以抵偿树的高度的减少。2-3-4 树中的查找时间与平衡二叉树如红-黑树大致相等，都是 O(logN)。

存储需求

2-3-4 树中每个节点保存三个数据项的引用和四个子节点的引用。这些空间可以用数组的形式，像 tree234.java 那样，或者用单独的变量。这些存储空间不是都有用的。只有一个数据项的节点会浪费 2/3 的数据项存储空间和 1/2 的子节点的存储空间。两个数据项的节点就浪费了 1/3 的数据项存储空间和 1/4 的子节点存储空间；换一种说法，它只利用了 5/7 的可用空间。

平均按每个节点有两个数据项来计算，大约有 2/7 的存储空间浪费了。

可能会想到用链表来代替数组以保存子节点和数据的引用。但对只有三个或四个数据项来说，链表的额外开销和数组比起来，用链表也许并不是一个值得用的方法。

因为红-黑树是平衡的，所以没有几个节点只有一个子节点，因此几乎所有子节点引用的存储空间都利用了。同样，每个节点都保存了一个数据项的最大项数值。这使红-黑树在存储上比 2-3-4 树利用率更高。

Java 中，存储对象的引用而不是对象本身，在 2-3-4 树和红-黑树之间这个存储差异并不重要，而且对 2-3-4 树编程当然更容易些。但是，在不是用保存引用这种方法的编程语言中，2-3-4 树和红-黑树间存储的效率差异可能就很显著了。

2-3 树

下面将简单地讨论一下 2-3 树，因为在历史上它很重要，并且在很多应用程序中还在应用。同时，一些用于 2-3 树的技术会在 B-树中应用，下一节将要学习到 B-树。最后，因为仅仅修改了每个节点的子节点数目就会使树的算法有很大的变化，这个问题很有趣。

2-3 树和 2-3-4 树类似，除了从名字中就可以猜到的不同之处，2-3 树的节点比 2-3-4 树少存一个数据项和少一个子节点。2-3 树是第一种多叉树，由 J.E.Hopcroft 在 1970 年发明。B-树（2-3-4 树是它的特例）直到 1972 年才发明。

2-3 树的操作在很多方面都和 2-3-4 树类似。节点可以保存 1 个或 2 个数据项，可以有 0 个，1 个、2 个或 3 个子节点。其他的方面，父节点和子节点的关键字值的排列顺序是和 2-3-4 树一样的。向节点插入数据项简单一些，因为需要的比较和移动次数少了。和 2-3-4 树中一样，所有新数据项都插入到叶节点中去，而且所有的叶节点都在树的最底层。

节点分裂

查找一个存在的数据项的方法和 2-3-4 树一样，只是数据项和子节点的数量不一样。可能会认为插入过程也和 2-3-4 树类似，但分裂节点的过程有很大的不同。

下面来看为什么分裂过程那么不同。在这两种树中，无论哪一种树节点进行分裂都需要三个数据项：一个保留在要分裂的节点里，一个右移到新的节点里，还有一个要上移到父节点里。2-3-4 树中一个满的节点有三个数据项，它们都各自移到自己的位置中去。但是，在 2-3 树中满的节点只有两个数据项，到哪里去找第三个数据项呢？因此必须要用到新的数据项：即要插入到树中的数据项。

2-3-4 树中在所有分裂完成之后才插入新数据项。2-3 树中新的数据项必须参与分裂过程。它必须要插入到叶节点中去，这就不可能在下行路途中进行分裂。如果新数据项要插入的叶节点不满，新数据项可以立即插入进去，如果叶节点满了，该节点就得分裂。该节点的两个数据项和新数据项分在这三个节点里：已存在的节点、新节点和父节点。如果父节点不是满的，操作就完成了（在连接新节点之后）。这种情况如图 10.17 所示。

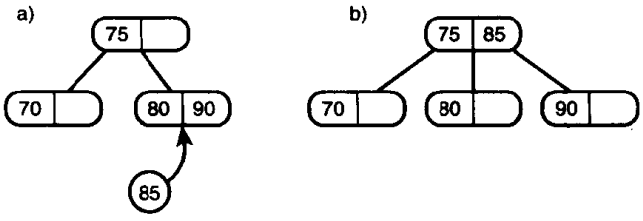


图 10.17 在非满的父节点插入的情况

但是，如果父节点是满的，父节点也必须要分裂。它的两个数据项和自己分裂的子节点传上来的数据项必须分配到父节点，父节点的新的兄弟节点，以及父节点的父节点中去。这种情况如图 10.18 所示。

如果父节点的父节点（叶节点的祖父节点）是满的，还是要分裂。分裂过程向上延续直到找到不满的父节点或者遇到根。如果根也是满的，就要创建一个新的根作为原来的根的父节点，如图 10.19

所示。

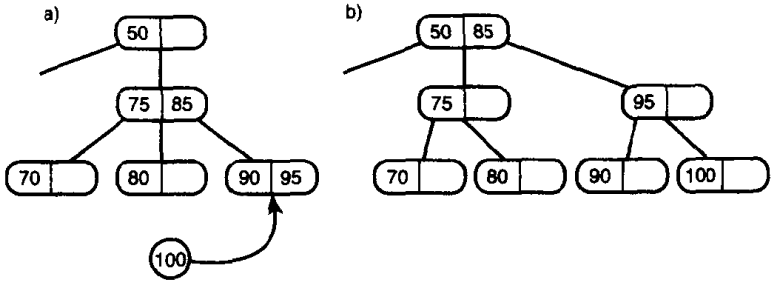


图 10.18 在满的父节点中插入的情况

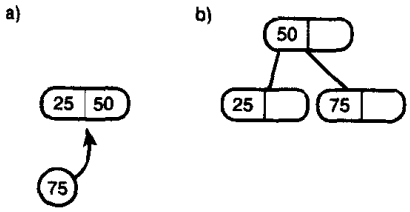


图 10.19 分裂根

图 10.20 展示的是顺着树向上延续分裂节点直到到达根为止的情况。

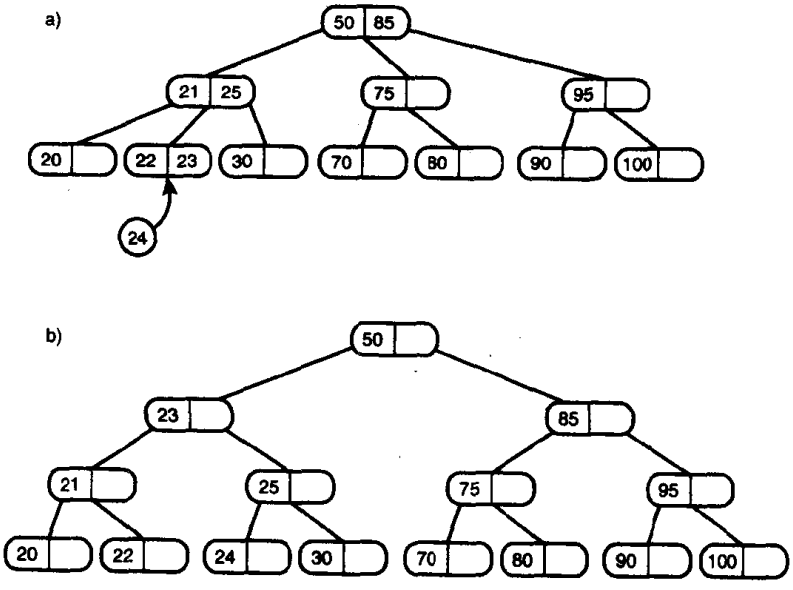


图 10.20 不断向上分裂的树

实现

本章把用 Java 完整地实现 2-3 树留作练习。但是，最后还是来提示一下如何处理分裂地过程。这只有一个方法（还有一种方法允许每个节点保存一个假的第四个子节点）。

向下插入过程的例程不理会遇到的节点是满还是不满。它顺着树向下查找直到找到要插入的叶节点。如果叶节点不满，就插入新的值。但是，如果叶节点满了，需要重新整理树来挪出位置。要做这件事，需要调用 split()方法。传入这个方法的参数是满了的叶节点和新的数据项。它会负责分裂这个节点并把新节点插入到新的叶节点里。

如果 split()方法发现叶节点的父节点也是满的，它递归调用自己来分裂父节点。split()方法一直调用自身，直到找到不满的叶节点或找到根为止。split()的返回值是新的右节点，它可以被前面调的 split()方法应用。

由于几个因素，编写分裂过程的代码是很复杂的。首先，2-3-4 树中三个要分开的数据项是已经有序的，但 2-3 树中新数据项的关键字值必须和其他两个叶节点中的数据项比较；然后根据比较结果分配这三个数据项。

另外，分裂父节点会创建第二个父节点，所以现在有一个左父节点（原来的）和一个新的右父节点。需要把连到一个父节点上的三个子节点的连接，变成两个父节点每个有两个子节点的状态。这里有三种情况，取决于哪个子节点（0，1 或 2）要分裂。如图 10.21 所示。

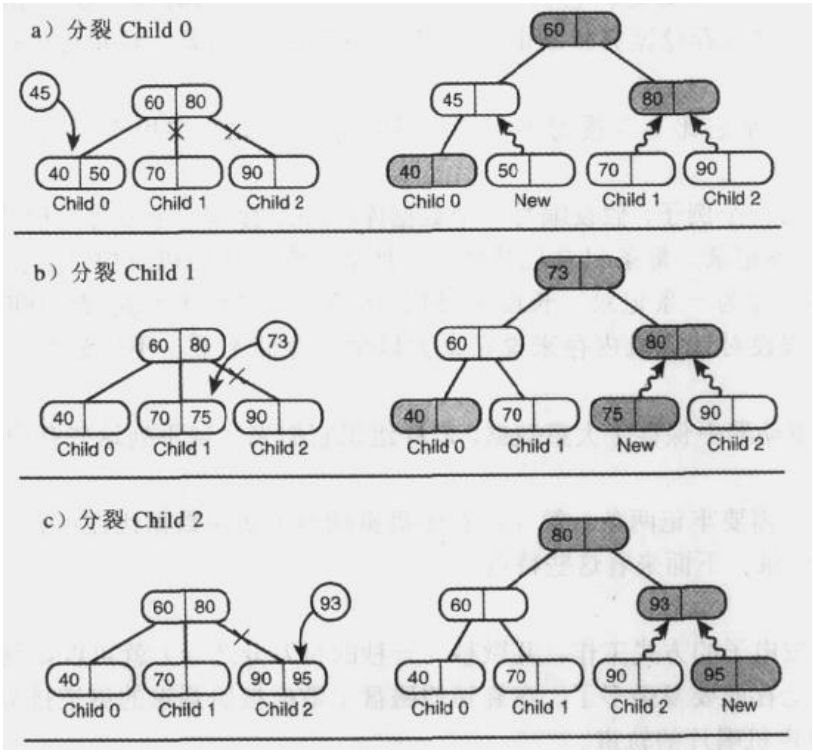


图 10.21 连接子节点

上图中分裂后创建的新节点用灰色显示，新的连接用曲线表示。

外部存储

2-3-4 树是多叉树的例子，多叉树是指节点的子节点多于两个并且数据项多于一个。另一种多叉树，B-树，尚在外部存储器上的数据时，它起很大作用。外部存储特指某类磁盘系统，例如在大

多数台式电脑或服务器中都有的硬盘。

本节中将从讨论外部文件管理的各种问题开始。用简单的方法来组织外部数据：即用顺序有序排列的方法。最后，学习 B-树，并解释为什么它和磁盘文件配合的那么好。本节以另一种外部存储的方法——“索引”来结束，这种方法可以单独使用也可以和 B-树一起用。

还要涉及外部存储的其他方面，例如检索技术。下一章会提到另一种外部存储的方法：哈希化。

外部存储技术实现的细节跟操作系统、语言，甚至和装置的硬件密切相关。因此，本节的讨论相比本书中的其他问题来说是相当概括的。

访问外部数据

到目前为止所讲过的数据结构都是假设数据存储在主存中的（通常称为 RAM，随机访问存储器）。但是，在很多情况下要处理的数据数量太大，不能都存储在主存中。这种情况下需要另一种存储方式。磁盘文件存储器一般比主存大得多；这使得存储每字节的花费比较低。

当然，磁盘还有另一个好处：它们的持久性。关掉计算机的时候（或者是断电了），在主存中的数据会丢失。磁盘文件存储器突然断电后还可以保存数据。但是，这里讨论的主要还是存储容量方面的差异。

外部存储的缺点是它比主存慢得多。由于速度不同因此需要用不同的技术来有效地管理它们。

作为外部存储的一个例子，假设编写一个数据库程序来管理一个中等规模城市的电话簿的数据——可能有 500000 条记录。每条记录包括姓名、地址、电话号码以及电话公司内部使用的各种其他数据。每条内容存储为一条记录，长度为 512 字节。则文件大小是 500000×512，256000000 字节，即 256M。假设对计算机内存来说它太大以至于存储不下，不过磁盘驱动器足够大可以存储它。

因此，在磁盘驱动器中保存了大量数据。怎样组织它们来实现下列这些有用的特点：快速查找、插入和删除呢？

为了找到答案，需要牢记两条。第一，在磁盘驱动器中访问数据比在主存中要慢得多。第二，一次需要访问很多记录。下面来看这些特点。

非常慢的存储

计算机的主存按电子的方式工作。几微秒（一秒的百万分之一）就可以访问一个字节。

在磁盘驱动器上存取要复杂多了。在旋转的磁盘上数据按照圆形的磁道排列，有点像压缩光盘（CD）或老式的留声机唱片的轨道。

要访问磁盘驱动器上的某段数据，读写头要先移到正确的磁道。通过步进的电动机或类似的设备完成；这样的机械运动需要好几毫秒（一秒的千分之几）。

一旦找到正确的磁道，读写头必须要等待数据旋转到正确的位置。平均来说，这一步要旋转半圈。即使磁盘每分钟转 10000 圈，那么在读到数据前还需要 3 毫秒。读写头就位后，就可以进行实际的读/写操作了；这可能还需要几毫秒。

因此，通常磁盘存取的时间大约是 10 毫秒。大约比访问主存慢了 10000 倍。

每年都会发展新技术来减少磁盘存取的时间，但主存访问的时间减少得更快，所以磁盘存取和

主存存取的时间差距将来会越来越大。

一次访问一个数据块

当读写头到达正确的位置后开始读（或写）过程，驱动器可以很快地把大量数据转移到主存中。因为这个原因，为了简化驱动器的控制装置，在磁盘上的数据按块存储，根据系统的不同称为块、页、分配单元，或其他的名字。这里称它们为块。

磁盘驱动器每次最少读或写一个数据块的数据。块的大小根据操作系统，磁盘驱动器的容量，以及其他因素而不同，但它总是 2 的倍数。在电话本的例子里，可以假设一个数据块的容量是 8192 字节（2^{13}）。因此，电话本数据库中需要 256000000 字节分为 8192 字节每块，一共是 31250 块。

在读/写操作时如果按块容量的倍数来操作是效率最高的。如果要读 100 字节，系统读取一块 8192 字节，只留下 100 字节，把其他的都扔了。或者如果要读 8200 字节，它会读两块，或者说 16284 字节，并把几乎一半的内容都扔了。通过组织软件使它每次操作一块数据，可以优化性能。

假设电话本记录的大小是 512 字节，在一块中存储 16 条记录（8192 除以 512），如图 10.22 所示。因此，效率最高的情况是一次读取 16 条记录（或这个数字的倍数）。

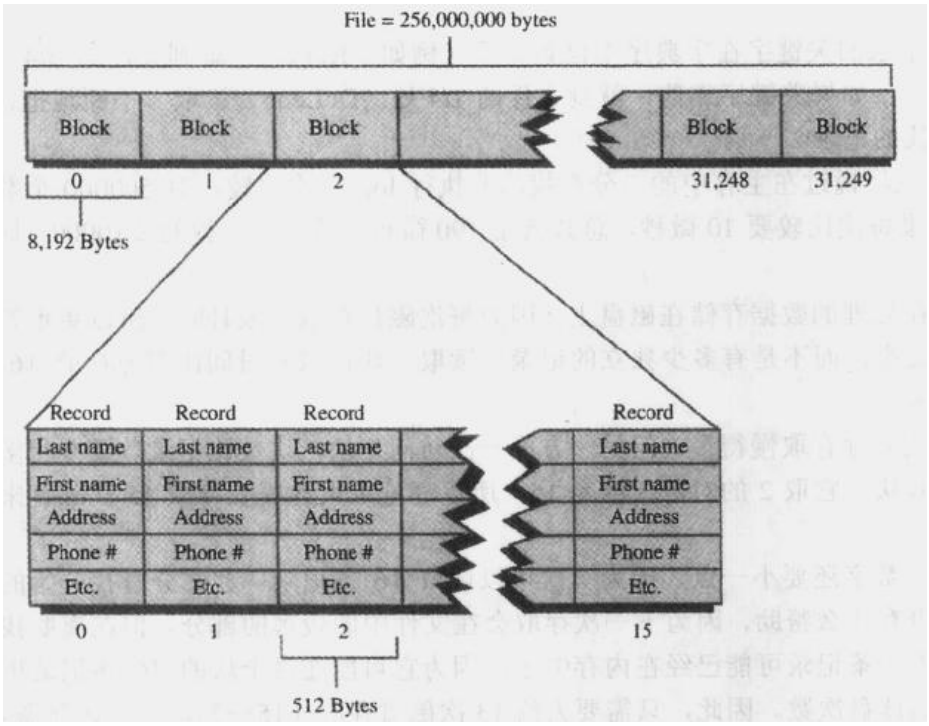


图 10.22 块和记录

注意让记录的大小是 2 的倍数也是很有用的。这样做，每块可以正好装下整数个记录。

当然，在这个电话本的实例中，记录，块和其他的数据的大小都只是示例性的；它们会根据记录的数目和大小，以及其他软件和硬件的限制等有很大的不同。通常一块中保存成百上千个记录，记录可能比 512 字节大也可能小。

一旦读写头已经就位，读取一块是相当快的，只需要几毫秒。因此，磁盘存取时读或写一

块的时间与块的容量关系不大。块越大，读或写一条记录的效率越高（假设用到的记录都在一块里）。

顺序有序排列

排列电话本数据的一种方法是在磁盘文件中按照某个关键字为所有的记录排序，如按照姓的字典序。Joseph Aardvark 记录可能会是第一个，依次类推。如图 10.23 所示。

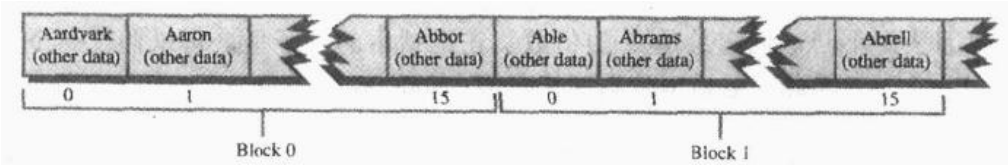


图 10.23 顺序有序排列

查找

在顺序有序排列的文件中查找某个姓的记录，如 Smith，可以用二分查找方法。需要从读取一块记录处于中间位置的记录开始。在主存中有 8192 字节的缓冲区，一次把一块中的 16 个记录都读到缓冲区里。

如果这些记录的关键字在字典序中位置靠后（例如，Keller），就到文件的 3/4 处（Prince）来读取那里的一块；如果关键字靠前，就到文件的 1/4 处（DeLeon）读取。不断地把范围缩小一半，最后会找到要找的记录。

第 2 章中已经说过在主存中的二分查找的要执行 $\log_2 N$ 次比较，对 500000 个数据项就是大约 19 次比较。如果每次比较要 10 微秒，总共就是 190 微秒，即大约一秒的 2/10000，比眨一下眼睛还要快。

但是，现在处理的数据存储在磁盘上。因为每次磁盘存取都很耗时，所以更重要的是要注意需要访问多少次磁盘，而不是有多少独立的记录。读取一块记录的时间比在内存的 16 个记录中查找的时间要长得多。

磁盘存取比内存存取慢得多，但另一方面一次访问一块，块数比记录数要少得多。在上面的例子中，有 31250 块。它取 2 的对数大约是 15，所以理论上大约需要存取 15 次磁盘来找到要找的记录。

实际上这个数字还要小一点，因为一次可以读取 16 条记录。在二分查找开始的阶段，内存中有多条记录并没有什么帮助，因为下一次存取会在文件中的较远的部分。但在离要找的记录很近的时候，要找的下一条记录可能已经在内存中了，因为它可能在这个块的 16 条记录里。这可能会减少了大约两次的读盘次数。因此，只需要大约 13 次磁盘存取（15－2），每次访问需要 10 毫秒，总共需要 130 毫秒，即 1/7 秒。这比内存访问要慢得多，但还不算太差。

插入

不幸的是，要在顺序有序排列的文件插入（或删除）一个数据项时情况要糟糕得多。因为数据是有序的，这两个操作平均都需要移动一半的记录，因此要移动大概一半的块。

移动每块都需要存取两次磁盘：一次读和一次写。找到插入点时，把包含插入点的数据块读入到存储缓冲区中。块中最后一条记录保存住，移动适当数目的记录为要插入的新记录腾地方。之后就把缓冲区的内容写回到磁盘中去。

下一步，第二块读到缓冲区中。保存它的最后一条记录，这块所有其他记录都移动，上一块的最后一条记录插入到缓冲区的开始处。之后缓冲区的内容再写回到磁盘中去。这个过程一直继续，直到所有在插入点后面的记录都重写过为止。

假设有 31250 块，需要读和写（平均）15625 块，每次读和写需要 10 毫秒的话总共要用 5 分钟来插入一条记录。如果要在电话本中插入上千条新名字，这显然太不理想了。

顺序有序排列的另一个问题是，如果它只有一个关键字，速度还比较快；比如这里的文件是按照姓排序的。但是假设需要查找某个电话号码，就不能用二分查找，因为数据是按姓排序的。这就得查找整个文件，用顺序访问的方法一块一块地找。这样查找需要读取平均一半的块，大约会需要 2.5 分钟，对一个简单的查找来说也是非常糟糕的。所以要寻找一种更有效的方法来保存磁盘中的数据。

B-树

怎样保存文件中的记录才能够快速地查找、插入和删除记录呢？前面已经讲过树是组织内存数据的一个好方法。那么树可以应用于文件吗？

当然可以，但对外部数据需要用和内存数据不一样的树。这种树是多叉树，有点像 2-3-4 树，但每个节点有更多的数据项；称它为 B-树。1972 年 R. Bayer 和 E. M. McCreight 首先提出用 B-树作为外部存储的数据结构。（严格地说，2-3 树和 2-3-4 树分别是 B-树的 3 节点和 4 节点的特例，但是 B-树通常更强调每个节点含有很多的子节点。）

每节点一块

为什么每个节点有那么多数据项呢？前面已经讲过一次读或写一块内的数据时效率最高。在树中，包含数据的实体是节点。把一整块数据作为树的节点是比较有意义的。这样做，读取一个节点可以在最短时间里访问最大数据量的数据。

一个节点中可以放多少数据呢？在电话本的例子里，每条数据记录 512 字节，可以把 16 条记录放在一个 8192 字节的块中。

但是在树中，还需要保存节点间的链接（就是链接到其他块去，因为一个节点对应一个块）。前面章节已经讨论过，在内存中的树里，这些链接是引用（或像 C++这样的语言里的指针），指向内存中其他部分的节点。在磁盘文件中存储的树，链接是文件中的块的编号（在电话本的例子中从 0 到 31249）。可以用一个 int 型的字段保存块号码，int 是 4 字节类型，可以保存 20 亿以上的块号码，基本上对大多数的文件都够用了。

现在不再把 16512 字节的记录挤到一块中去了，因为需要空间来保存指向子节点节点的引用。把块中记录的数量减少到 15 这样就有地方来保存链接了，不过更高效的方法是每个节点保存偶数个记录，这样（在恰当地分配管理后），把记录大小减少为 507 字节。将有 17 个子节点链接（比数据项个数多 1），这样链接需要 68 字节（17×4）。这样 16507 字节的记录填进块中后，还留下 12 字节的空间（507×16+68=8180）。树中一个节点对应一块，如图 10.24 所示。

在每个节点中数据是按关键字顺序有序排列的，像 2-3-4 树一样。实际上，B-树的结构很像 2-3-4 树，只是每个节点有更多的数据项和更多的指向子节点的链接。B-树的阶数由节点拥有最多的子节点数决定。在这个例子里是 17，所以这个树是 17 阶 B-树。

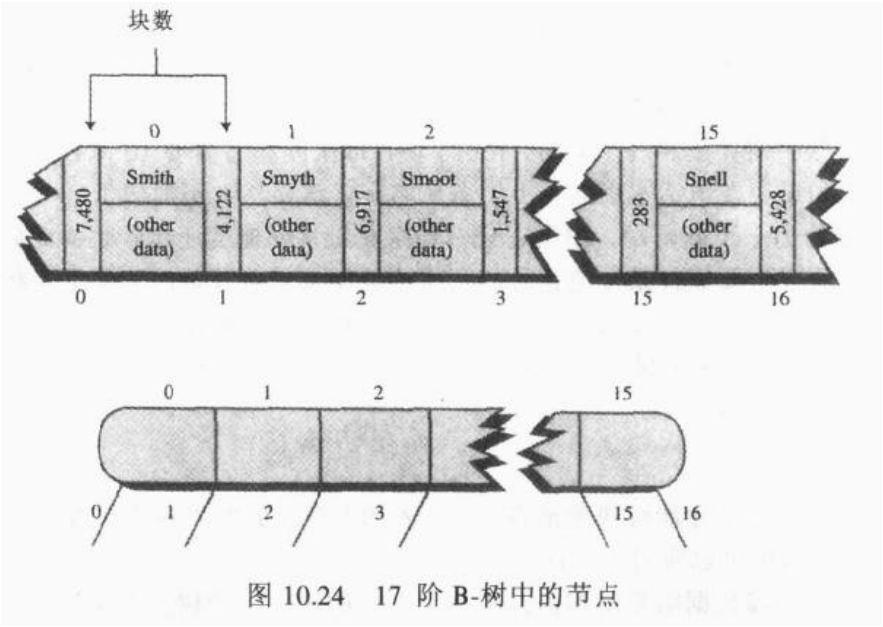


图 10.24 17 阶 B-树中的节点

查找

在记录中按关键字查找和在内存的 2-3-4 树中查找很类似。首先，含有根的块读入到内存中。然后搜索算法开始在这 15 个节点中查找（或者，如果块不满的话，有多少块就检查多少块），从 0 开始。当记录的关键字比较大时，需要找在这条记录和前一条记录之间的那个子节点。

持续这个过程直到找到正确的节点。如果到达叶节点还没有找到那条记录，则查找不成功。

插入

B-树中的插入过程更像 2-3 树，而不是 2-3-4 树。回忆一下 2-3-4 树有很多节点不满，实际上有的只有一个数据项。尤其是每次节点分裂时总会产生两个每个有一个数据项的节点。在 B-树中这不是最好的方法。

尽可能让 B-树节点满是非常重要的，这样每次存取磁盘时，读取整个节点，就可以获得最大数量的数据。为了达到这个目的，B-树的插入过程与 2-3-4 树的插入有下列三点不同：

- 节点分裂时数据项平分：一半到新创建的节点中去，一半保留在原来的节点中。
- 节点分裂像 2-3 树那样从底向上，而不是自顶向下。
- 同样，还是像 2-3 树一样，原节点内中间数据项不上移，而是加上数据项后所组成的节点数据项序列的中间数据项上移。

通过建立一棵小的 B-树来示范插入过程中的这些规则，如图 10.25 所示。因为没有那么大的地方来显示每个节点中的实际有的记录的数量，所以节点中只有四个记录；因此，这是一棵 5 阶 B-树。

图 10.25a 中显示的是一个已经满了的根节点；数据项的关键字是 20、40、60 和 80，它们已经插入到树中了。新的数据项 70 要插入，就导致节点分裂。下面是节点分裂的过程。因为是根要分裂，所以要创建了两个新节点（像 2-3-4 树一样）：一个新的根和一个作为要分裂节点的右兄弟的新节点。

为了决定新数据项要插在哪里，插入算法在内部缓冲区里把这 5 个的关键字排序。其中有 4 个

关键字在要分裂的节点中，第 5 个是要插入的新数据项。图 10.25 中，这 5 个数据项序列显示在树的旁边。第一步显示的序列是 20，40，60，70，80。

在第一步序列里的中间数据项是 60，它被提到新的根节点中。（在图中，一个箭头表示的中间数据项要上移。）所有中间数据项左边的数据项留在分裂节点中，所有右边的移到右边新节点中。结果如图 10.25b 所示。（在本章的电话本例子中，8 个数据项会移到每个子节点中，而不是图中所示的两个。）

图 10.25b 中再插入两个数据项，10 和 30。它们填满了左子节点，如图 10.25c 所示。下一个要插入数据项 15，则分裂这个左子节点，结果如图 10.25d 所示。数据项 20 被上移到根中。

下一步，三个数据项——75、85 和 90——要插入到树里。前两个填满第三个子节点，第三个数据项使它分裂，需要建立新节点并把中间数据项 80 提升到根中。结果如图 10.25e 所示。

又是三个数据项——25、35 和 50——要插入树里。前两个填满第二个子节点，第三个使它分裂，需要建立新节点并把中间数据项 35 提升到根中。结果如图 10.25f 所示。

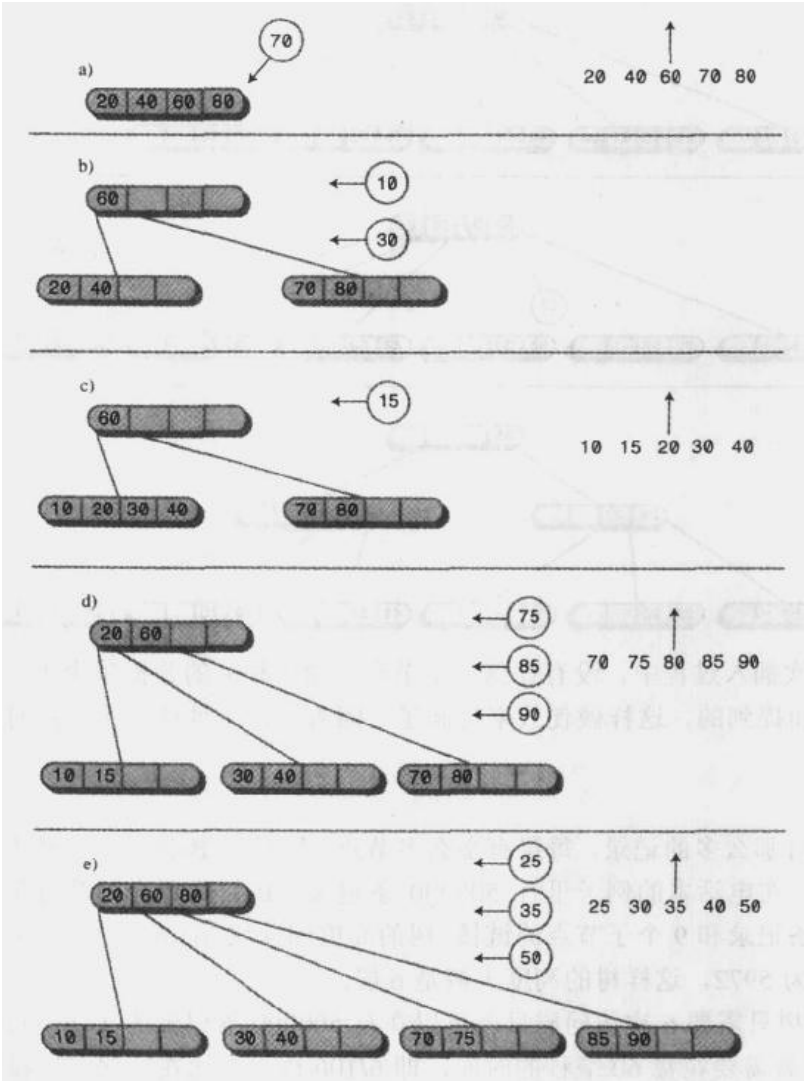
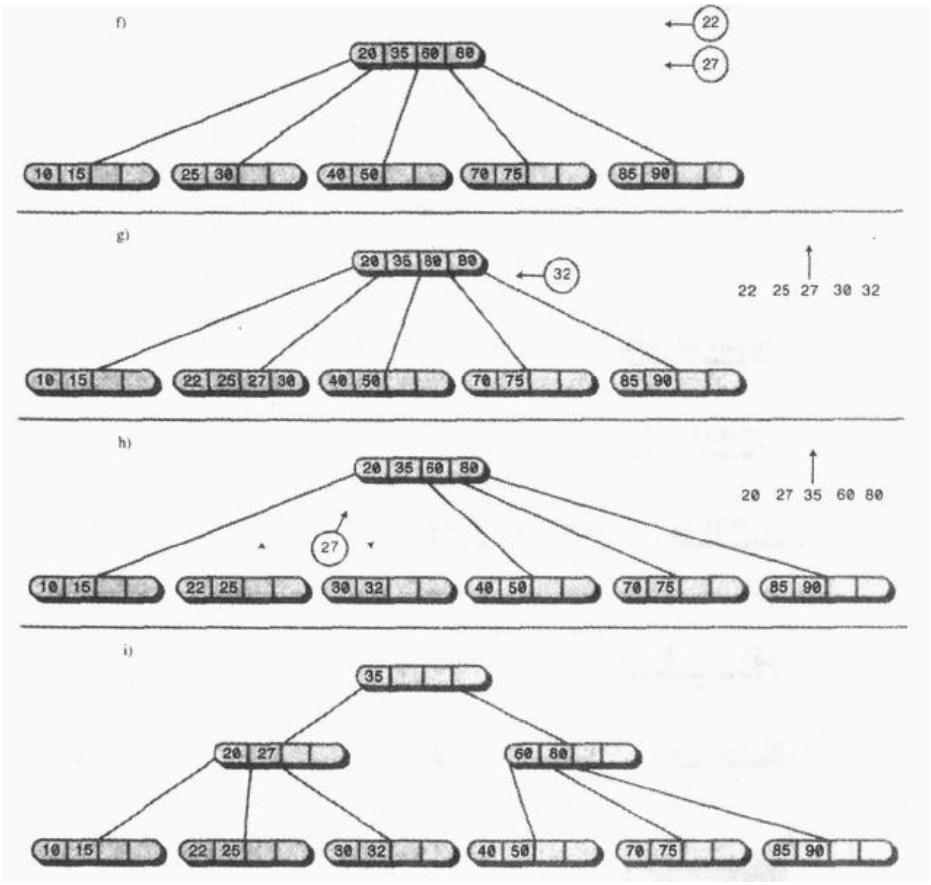


图 10.25　建立一棵 B 树

现在根已经满了。但是，随后的插入不需要分裂节点，因为只有新数据项要插入到一个满的节点时才需要分裂，而不是在树中向下查找时遇到满的节点就分裂。因此，22 和 27 插入到第二个子节点里，而不需要节点分裂，如图 10.25g 所示。

但是，下一个要插入的数据项 32 就导致了节点分裂；实际上是两个节点分裂。第二个子节点已经满了，所以它分裂，如图 10.25h 所示。但是这次分裂提出的数据项 27，没有地方去，因为根也是满的。因此，根必须也分裂，最后的分配结果如图 10.25i 所示。



注意在整个这次插入过程中，没有任何一个节点（除了根）的数据项少于一半，并且很多都比一半要满。正如上面提到的，这样做使效率提高了，因为访问文件读一个节点时总是能存取尽可能多的数据。

B-树的效率

因为每个节点有那么多的记录，每层有那么多节点，因此在 B-树上的操作非常快，这里假设数据都保存在磁盘上。在电话本的例子里有 500000 条记录。B-树中所有的节点至少是半满的，所以每个节点至少有 8 条记录和 9 个子节点的链接。树的高度因此比 $\log_9 N$ 小一点（N 以 9 为底的对数），N 是 500000。结果为 5972，这样树的高度大概是 6 层。

因此，使用 B-树只需要 6 次访问磁盘就可以在有 500000 条记录的文件中找到任何记录了。每次访问 10 毫秒，这就需要花费 60 毫秒的时间，即 6/100 秒。这比在顺序有序排列的文件中二分查找要快得多。

节点中记录越多，树的层数就越少。实例中的 B-树是 6 层，虽然节点只含有 16 条记录。相反，有 500000 个数据项的二叉树大约有 19 层，2-3-4 树会有 10 层。如果块中有成百条记录，可以减少树的层数并且大大提高访问时间。

虽然 B-树中的查找比在顺序有序排列的磁盘文件查找快，不过插入和删除操作才显示出 B-树最大的优越性。

先假设在 B-树中不需要节点分裂的插入情况。这种情况很少见，因为每个节点中会有大量的记录。在电话本实例中，已经看到只需要 6 次访问就可以找到插入点。之后还需要一次访问把保存了新插入记录的块写回到磁盘中去，一共是 7 次访问。

接着来看看节点需要分裂的情况。要读入分裂的节点，它的一半记录都要移动，并且要写回磁盘。新创建的节点要写入磁盘，必须要读取父节点，然后插入上移的记录，写回磁盘。这里就有 5 次访问，加上找到插入点需要的 6 次访问，一共是 12 次。相比在访问顺序文件中插入数据项所需要的 500000 次访问这是大大地改进了。

在其他的一些 B-树里，只有叶节点保存数据。非叶节点只保存关键字和块的号码。这使得操作更快，因为每一块可以保存更多块的号码。这样的高阶树层数更少，访问速度也提高了。但是，编程可能会复杂一些，因为这里有两种节点：叶节点和非叶节点。

索引

另一种加快文件访问速度的方法是用顺序有序排列存储记录但用文件索引连接数据。文件索引是由关键字-块对组成的列表，它按关键字排序。回忆一下电话本实例中有 500000 条 512 字节的记录，每块存 16 条记录，则有 31250 块。假设查找的关键字是姓，每条索引中有两个数据项：

- 关键字，比如 Jones。
- 块的号码，Jones 在文件中的定位。这个号码的范围从 0 到 31249。

用 28 个字节长度的字符串来保存关键字（足够保存最长的姓）以及 4 字节来保存块号码（Java 语言中的 int 类型）。索引中的每个记录需要 32 字节。这只是每条记录大小的 1/16。

索引中的记录根据姓按顺序有序排列。磁盘上原来那些记录可以按任何顺序有序排列。这就是说新记录可以简单地添加到文件末尾，这样记录按照插入时间排列。这种排列方法如图 10.26 所示。

内存中的文件索引

索引比文件中实际记录小得多。它甚至可以完全放在内存里。在电话本实例中，有 500000 条记录。每一条在索引中的记录是 32 字节，这样索引的大小是 32×500000 字节，即 1600000 字节那么长（1.6 兆）。把它放到现在的计算机的内存中不会有任何问题。索引可以保存在磁盘里，数据库程序启动后就读取到内存中来。这样，对索引的操作就可在内存中完成了。每天结束时（或更频繁也行），索引可以写回磁盘中永久保存。

查找

应用将索引放在内存中的方法，使得操作电话本的文件比直接在顺序有序排列记录的文件中执行操作更快。例如，二分查找需要 19 次索引访问。每次访问 20 微秒，这样只需要大约 4/10000 秒。然后在索引中找到实际的记录块的号码后，（不可避免地）要花时间从文件中访问它。不过，这一次访问磁盘的时间只需要 10 毫秒。

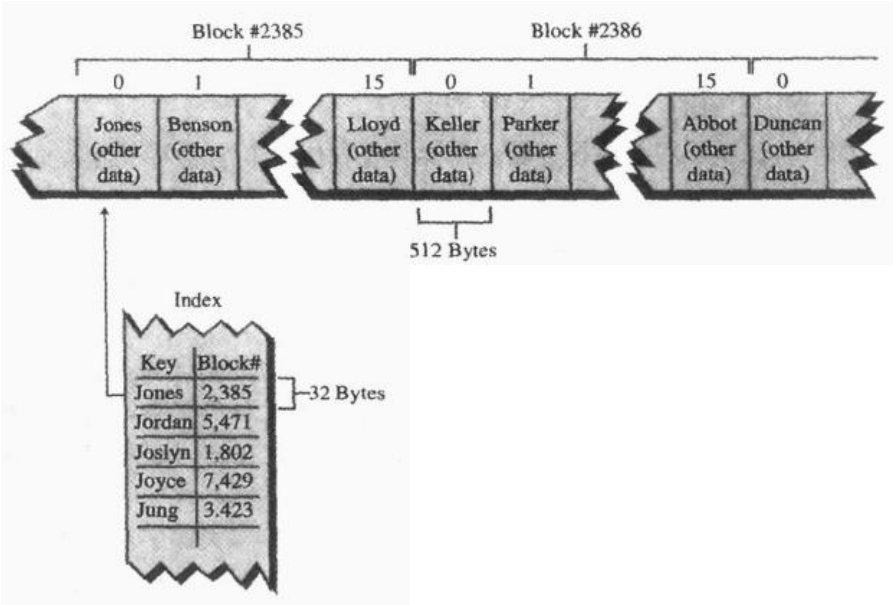


图 10.26 文件索引

插入

在索引文件中插入新数据项，要做两步。首先把这个数据项整个记录插入到主文件中去；然后把关键字和包括新数据项存储的块号码的记录插入到索引中。

因为索引是顺序有序排列的，要插入新数据项，平均需要移动一半的索引记录。设内存中 2 微秒移动一个字节，则需要 250000 乘以 32 再乘以 2，大约要 16 秒来插入一个新记录。这比没有索引，在顺序有序排列的文件插入一条新记录要 5 分钟还是快多了。（注意不需要移动主文件中的记录；只要在文件末尾处添加了一条新纪录即可。）

当然，可以用更复杂的方法在内存中保存索引。例如把它存为二叉树，2-3-4 树，或红-黑树。这些方法都大大减少了插入和删除的时间。每种情况下把索引存在内存中的方法都比文件顺序有序排列的方法快得多。有时比 B-树都快。

在索引文件的插入过程中真正的磁盘访问包括插入新纪录本身。通常，把文件的最后一块读入到内存中来，把新纪录添加在后面，然后把这块写回到磁盘上去。这个过程只需要两次文件访问。

多级索引

索引方法的一个优点是多级索引，同一个文件可以创建不同关键字的索引。在一个索引中关键字可以是姓；另一个索引中，是地址。索引和文件比起来很小，所以它并不会大量地增加数据存储量。当然，数据项从文件中删除的时候会麻烦一些，需要把所有索引中的那条索引记录删掉，但这里不讨论这个问题。

对内存来说索引太大

如果索引太大，不能放在内存中，它就需要按块分开存储在磁盘上。对大文件来说把索引保存成 B-树是很合适的。主文件中记录可以存成任何合适的顺序。

这种排列方法效率很高。把记录添加到主文件末尾很快，在索引中插入新记录的索引记录也很

快，因为索引按树形存储。对大文件来说这样做查找和插入操作都很快。

注意索引按 B-树存储时，每个节点保存 n 个子节点指针和 n-1 个数据项。子节点指针是索引中其他节点的块的号码。数据项保存关键字和指向主文件中一个块的指针。不要把这两种类型的块指针搞混。

组合搜索条件

组合搜索惟一可行的方法是顺序地读取文件中的每一块。假设电话本实例中要找一列记录，它们在电话本中名字都是 Frank，地址都是 Springfield，且电话号码中有三个 7。（这好像是在一宗谋杀中，受害者手中抓着的胡乱涂写的残片上所显示的线索。）

根据姓排列文件一点用也没有。甚至如果有索引文件，按名字和城市排序，也不能很方便地找到含有 Frank 和 Springfield 的记录。在这种情况下（在许多数据库中很常见），最快的方法可能就是顺序读取文件，一块块地检查每个记录看看它是否符合查找条件。

外部文件排序

归并排序是外部数据排序的首选方法。这是因为，这种方法比起其他大部分排序方法来说，磁盘访问更多的涉及临近的记录而不是文件中随机的部分。

回忆第 6 章“递归”中归并排序递归地调用它自身，来给越来越短的序列排序。一旦两个最小的序列（在内存执行的版本中是只有 1 字节）排好序，就把这两个合并成长一点的序列。越来越长的序列合并，直到最后整个文件有序。

外部存储的方法是相似的。不过最小的序列可能是从磁盘读取的一块中的一个记录。因此，需要两步过程。

第一步，读取一块，它的记录在内部排序，然后把排完序的块写回到磁盘中。下一块也同样排序并写回到磁盘中。直到所有的块内部都有序为止。

第二步，读取两个有序的块，合并成一个两块的有序的序列，再把它们写回到磁盘。下次，把每两块序列合成四块的序列。这个过程继续下去，直到所有成对的块都合并过了为止。每次，有序序列的长度增长一倍，直到整个文件有序。

图 10.27 中是对一个外部文件归并排序的过程。文件含有四块，每块有四条记录，一共有 16 条记录。每次只能把三块读到内存中。（当然，这些容量在实际情况下会大得多。）图 10.27 展示了排序前的文件；每条记录中的数字是它的关键字值。

块的内部排序

在第一步中文件中的所有块都在内部排序。把一块读入到内存中并且可以用任何内部排序的方法为它排序，如快速排序（或者是少量记录的希尔排序或插入排序）。块内部排序后的结果如图 10.27b 所示。

可以用第二个文件来保存有序的块，假设外部存储的容量不成问题。通常希望避免改变初始文件的内容。

归并

第二步，要合并有序的块。第一趟中把每两块合并成一个含有两块的序列。因此，两个块 2-9-11-14 和 4-12-13-16 要合并成 2-4-9-11-12-13-14-16。同样，3-5-10-15 和 1-6-7-8 合并成 1-3-5-6-7-8-10-15。结果如图 10.27c 所示。用第三个文件来保存这步合并的结果。

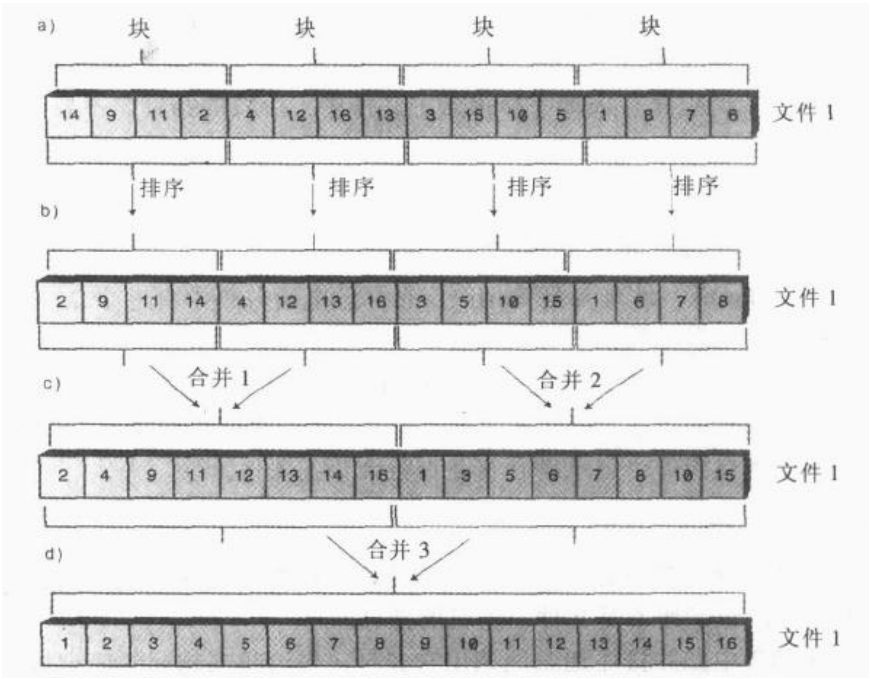


图 10.27 对外部文件进行归并排序

第二趟，两个 8 条记录的序列合并成 16 条记录的序列，可以把它写回到第二个文件里，如图 10.27d 所示。现在排序就完成了。当然，要排序更大的文件时需要合并更多的步数；步数和 $\log_2 N$ 成正比。归并的步骤可以在两个文件中交替存储（图 10.22 中的文件 2 和文件 3）。

内部数组

因为计算机内存中只能存放三个数据块，合并过程必须按一定的步骤来进行。设有三个数组，称为 arr1、arr2 和 arr3，每个可以存储一个数据块。

第一次归并时，块 2-9-11-14 读入到 arr1，4-12-13-16 读入到 arr2。这两个数组归并排序到 arr3。但是，因为 arr3 只能盛下一块，在排序完成前它就满了。当它满了以后，它的内容就写回到磁盘中去。然后排序就继续，再次填满 arr3。这就完成了排序，arr3 再次写回到磁盘中。下面的清单就是三次归并排序中每次的细节。

第一次归并：

1. 将 2-9-11-14 读入到 arr1 中。
2. 将 4-12-13-16 读入到 arr2 中。
3. 把归并结果 2，4，9，11 写入 arr3；写回磁盘。
4. 把归并结果 12，13，14，16 写入 arr3；写回磁盘。

第二次归并：

1. 将 3-5-10-15 读入到 arr1 中。
2. 将 1-6-7-8 读入到 arr2 中。
3. 把归并结果 1，3，5，6 写入 arr3；写回磁盘。
4. 把归并结果 7，8，10，15 写入 arr3；写回磁盘。

第三次归并：

1. 将 2-4-9-11 读入到 arr1 中。
2. 将 1-3-5-6 读入到 arr2 中。
3. 把归并结果 1，2，3，4 写入 arr3；写回磁盘。
4. 把 5，6 合并到 arr3（arr2 现在空了）。
5. 把 7-8-10-15 读入到 arr2 中。
6. 把 7,8 合并到 arr3 中，写回磁盘。
7. 把 9,10,11 合并到 arr3，（arr1 现在空了）。
8. 把 12-13-14-16 读取到 arr1 中。
9. 把 12 合并到 arr3 中；写回磁盘。
10. 把 13，14，15，16 合并到 arr3 中，写入磁盘。

这 10 步最后的序列相当长，因此在每步完成时检查数组中的内容是很有帮助的。图 10.28 展示了在三步合并的过程中这些数组的内容。

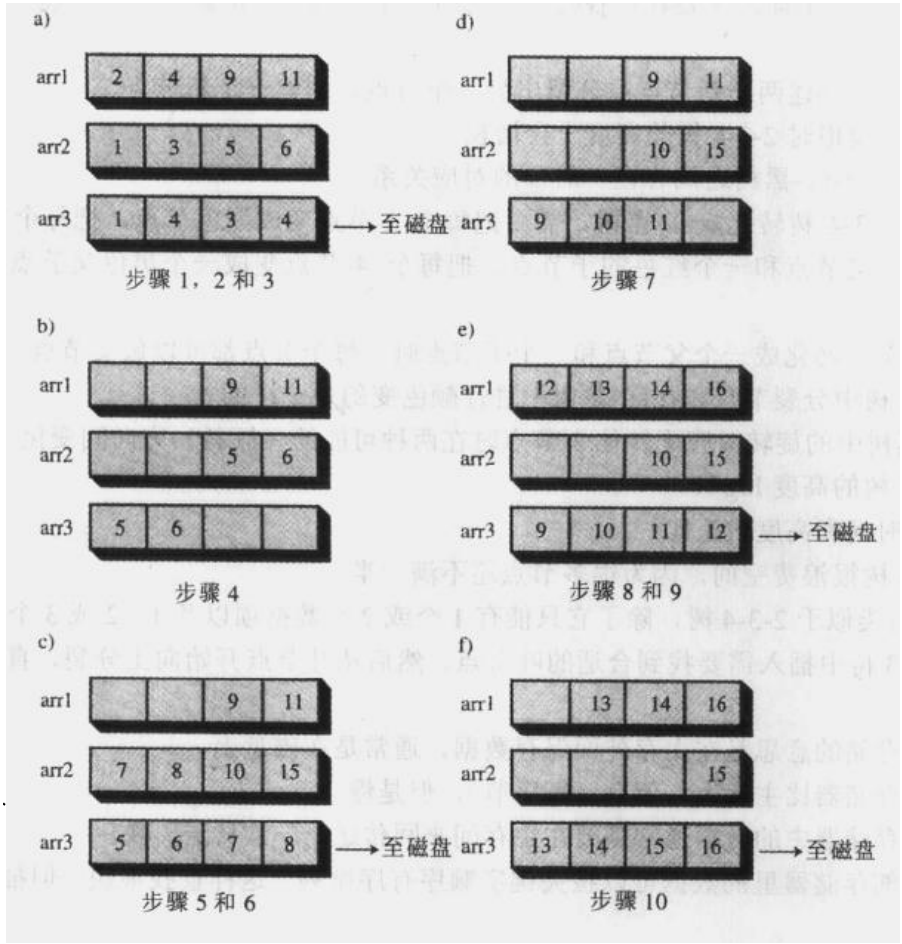


图 10.28　3 次归并中数组的内容

小 结

- 多叉树比二叉树有更多的关键字和子节点。
- 2-3-4 树是多叉树，每个节点最多有三个关键字和四个子节点。
- 多叉树中，节点中数据项按关键字升序排列。
- 2-3-4 树中，所有的插入都在叶节点上，所有的叶节点在同一层上。
- 在 2-3-4 树中有三种可能的节点：2-节点有 1 个关键字和 2 个子节点，3-节点有 2 个关键字和 3 个子节点，4-节点有 3 个关键字和 4 个子节点。
- 2-3-4 树中没有 1-节点。
- 在 2-3-4 树中查找时，检查每个节点的关键字。没有找到时，如果要查找节点的关键字比 key0 小，下一个节点就是 child0；如果要查找节点的关键字在 key0 和 key1 之间，下一个节点就是 child1；如果要查找节点的关键字在 key1 和 key2 之间，下一个节点就是 child2；如果要查找节点的关键字大于 key2，下一个节点就是 child3。
- 在 2-3-4 树中插入需要在查找插入点的过程中，顺着树的路径向下分裂路径上每个满的节点。
- 分裂根要创建两个新节点；分裂出另一个节点，创建一个新节点。
- 只有分裂根时 2-3-4 树的高度才会增长。
- 2-3-4 树和红-黑树之间存在一对一的对应关系。
- 要把 2-3-4 树转化成红-黑树，需要把每个 2-节点变成黑色节点，把每个 3-节点变成一个黑色的父节点和一个红色的子节点，把每个 4-节点变成一个黑色父节点和两个红色子节点。
- 当 3-节点转化成一个父节点和一个子节点时，每个节点都可以做父节点。
- 2-3-4 树中分裂节点和在红-黑树中进行颜色变幻是一样的。
- 红-黑树中的旋转对应于转化 3-节点时在两种可能的（倾斜）方向间变化。
- 2-3-4 树的高度 $\log_2 N$。
- 查找时间和高度成正比。
- 2-3-4 树很浪费空间，因为很多节点还不满一半。
- 2-3 树类似于 2-3-4 树，除了它只能有 1 个或 2 个数据项以及 1、2 或 3 个子节点。
- 在 2-3 树中插入需要找到合适的叶节点，然后从叶节点开始向上分裂，直到找到不满的节点。
- 外部存储的意思是在主存外面保存数据，通常是在磁盘上。
- 外部存储器比主存大，便宜（每字节），但是慢。
- 外部存储器中的数据通常需要在主存间来回传送，一次传送一块。
- 在外部存储器里的数据可以按关键字顺序有序排列。这样查找很快，但插入（或删除）很慢。
- B-树是多叉树，每个节点可以有几十或上百个关键字和子节点。
- B-树中子节点的个数总是比关键字的个数多 1。
- 为达到最好的性能，B-树通常在一个节点中保存一块的数据。

- 如果查找条件涉及多个关键字，在文件所有的记录中顺序查找可能是最实用的方法。

问 题

下列问题作为读者的自测题。答案可见附录 C。

1. 2-3-4 树这样命名是因为它的一个节点可以有：
   a. 3 个子节点和 4 个数据项。
   b. 2 个、3 个或 4 个子节点。
   c. 2 个父节点、3 个子节点和 4 个数据项。
   d. 2 个父节点、3 个数据项和 4 个子节点。
2. 2-3-4 树比二叉搜索树高级，因为它_____。
3. 假设父节点节点的数据项是 25、50 和 75。如果它的一个子节点节点的数据项是 60 和 70，这个子节点的编号是_____。
4. 判断题：数据项只能在叶节点里。
5. 每次节点分裂时下列说法那一项是不正确的？
   a. 恰好创建一个新节点。
   b. 把新的数据项正确的插入到树中。
   c. 把一个数据项从分裂节点移到它的父节点里。
   d. 把一个数据项从分裂节点移到它的新兄弟节点里。
6. 2-3-4 树在_____时高度增加。
7. 在 2-3-4 树中查找包括：
   a. 如果需要的话，分裂查找路径上的节点。
   b. 根据节点中的数据项，选择转向适当的子节点。
   c. 如果没有找到要找的关键字，就停在叶节点处。
   d. 在每个访问的节点里至少查看一个数据项。
8. 在 2-3-4 树中非根节点分裂后，它的新右子节点含有它以前的哪个数据项，0，1 还是 2？
9. 2-3-4 树中一个 4-节点分裂相当于在红-黑树中_____。
10. 下列关于 2-3 树（不是 2-3-4 树）的节点分裂操作哪个描述是不对的？
    a. 分裂节点的父节点如果是满的也必须分裂。
    b. 要分裂节点中最小的数据项总是留在原来的节点中。
    c. 当父节点分裂后，child2 总是断开它旧的父节点的连接，连到新的父节点上去。
    d. 分裂过程从叶节点开始并向上执行。
11. 用大 O 表示法表示 2-3 树的时间复杂度是多少？
12. 访问磁盘驱动器的数据时：
    a. 插入数据很慢，但找写数据的位置却很快。
    b. 移动数据来给更多的数据腾地是很快的，因为一次可以访问很多数据项。
    c. 删除数据超常的快。
    d. 找到要写数据的位置相当慢，但大量的数据可以很快地写入。

13. B-树中每个节点含有_____个数据项。

14. 判断题：B-树中节点分裂和 2-3 树中节点分裂相似。

15. 外部存储中，索引的含义是保存一个文件的：

a. 关键字和它们对应的块。

b. 记录和它们对应的块。

c. 关键字和它们对应的记录。

d. 姓和它们对应的关键字。

实　验

完成这些实验可以帮助深入理解本章的主题。不需要编程实现。

1. 用笔画出下面每一次插入完成后 2-3-4 树的样子：10，20，30，40，50，60，70，80 和 90。不要用 Tree234 专题 applet。

2. 用笔画出下面每一次插入完成后 2-3 树的样子，插入序列同实验 1。

3. 思考怎么从 2-3-4 树中删除一个节点。

编程作业

编程作业有助于巩固对本章内容的理解，并展示如何应用本章的概念。（在“简介”中提到过，资深教师可以从出版者的网站上得到编程作业的完整答案。）

10.1　这个作业很容易。编写一个方法能返回 2-3-4 树中的最小值。

10.2　编写方法中序遍历 2-3-4 树。它能有序地显示所有的数据项。

10.3　2-3-4 树可以用于排序。编写 sort()方法，从 main()方法中传入关键字的数组并在排序后把它们写回数组中去。

10.4　修改 tree234.java 程序（清单 10.1），使它可以创建并操作 2-3 树。要求显示树并可以查找。还要能够插入数据项，但只限在叶节点（要分裂的）的父节点不需要再分裂的情况。这就是说 split()方法不需要递归。编写 insert()方法，记得在找到要插入的合适的叶节点之前不需要节点分裂。找到后，如果叶节点满了就要分裂。也需要能够分裂根，但只限它是叶节点的情况下。根据这个限制，只能插入少于 9 个的数据项，否则程序就会起冲突了。

10.5　扩展编程作业 10.4 的程序，使 split()方法是递归的并可以处理一个满的子节点的满父节点的情况。这就允许不限个数地插入节点。注意在递归的 split()例程中，在决定数据项要转向何处和子节点要连接到哪里之前要先分裂父节点。

第11章 哈希表

本章重点

- 哈希化简介
- 开放地址法
- 链地址法
- 哈希函数
- 哈希化的效率
- 哈希化和外部存储

哈希表是一种数据结构，它可以提供快速的插入操作和查找操作。第一次接触哈希表时，它的优点多得让人难以置信。不论哈希表中有多少数据，插入和删除（有时包括删除）只需要接近常量的时间：即O(1)的时间级。实际上，这只需要几条机器指令。

对哈希表的使用者——人来说，这是一瞬间的事。哈希表运算得非常快，在计算机程序中，如果需要在一秒种内查找上千条记录，通常使用哈希表（例如拼写检查器）。哈希表的速度明显比树快，正如前面几章看到的，树的操作通常需要O(N)的时间级。哈希表不仅速度快，编程实现也相对容易。

哈希表也有一些缺点：它是基于数组的，数组创建后难于扩展。某些哈希表被基本填满时，性能下降得非常严重，所以程序员必须要清楚表中将要存储多少数据（或者准备好定期地把数据转移到更大的哈希表中，这是个费时的过程）。

而且，也没有一种简便的方法可以以任何一种顺序（例如从小到大）遍历表中数据项。如果需要这种能力，就只能选择其他数据结构。

然而，如果不需要有序遍历数据，并且可以提前预测数据量的大小，那么哈希表在速度和易用性方面是无与伦比的。

哈希化简介

本节将要介绍哈希表和哈希化。其中一个重要的概念是如何把关键字转换成数组下标。在哈希表中，这个转换通过哈希函数来完成。然而，对于特定的关键字，并不需要哈希函数；关键字的值可以直接用于数组下标。先来看一个简单的例子；然后再逐步展示，若关键字值不能恰好用于数组下标时，如何使用哈希函数进行转换。

雇员号码作为关键字

假设现在要写一个程序，存取一个公司的雇员记录，这个小公司大约有1000个员工。每个雇员记录需要1000个字节的存储空间。因此，整个数据库的大小约1MB，一般的计算机内存都可以满足。

计算机部门的领导已经强调必须要尽可能快地存取每个雇员记录。而且，每个雇员有一个特定的号码，从1（公司创立者）到1000（最近雇佣的工人）。这个雇员号码作为存取记录的关键字；事实上，用其他关键字进行存取完全没有必要。雇员很少被解雇，但是即使当他们不在公司了，他们的记录也要保存在数据库中以供参考（涉及到退休金等等问题）。在这种情况下需要使用什么数

据结构呢？

关键字作为索引

一种可能是用数组。每个雇员号码占数组的一个单元。单元的数组下标是当前记录的雇员号码。数组的类型如图 11.1 所示。

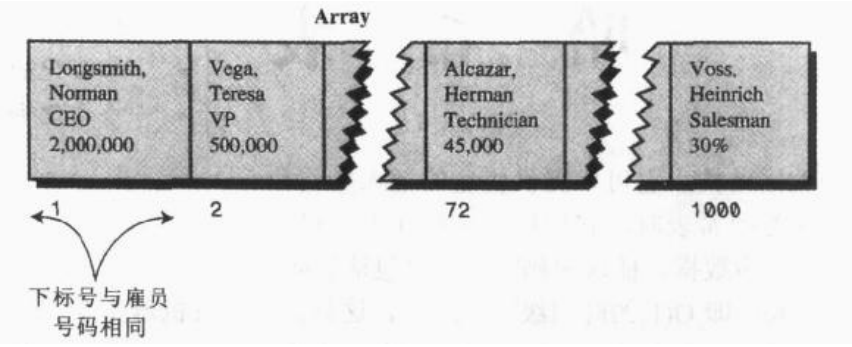


图 11.1 雇员号码作为数组下标

众所周知，如果知道数组下标，要访问特定的数组数据项非常方便。操作员要查找 Herman Alcazar，他知道 Alcazar 的雇员号码是 72，所以他键入这个号码，程序直接检索数组下标为 72 的单元。总共只需要一条语句：

```
empRecord rec = databaseArray[72];
```

增加一个新项也非常快。只需要把它插到最后一个数据项的后面。下一个记录（Jim Chan，是最新雇佣的员工，雇员号码是 1001）将被放在第 1001 个单元。插入新记录也只需要一条语句：

```
databaseArray[totalEmployees++] = newRecord;
```

数组大概需要比当前的雇员数量多少大一些，为扩展留出空间，但不能指望可以大量扩展数组容量。

不总是如此有序

使用基于数组的数据库，使得存储数据速度快且非常简单，这很吸引人。然而这个例子之所以能运行，仅仅是因为关键字组织得异乎寻常的好。关键字从 1 到一个已知的最大值，这个最大值对数组来说是合理的。没有删除，所以序列中不存在浪费内存的断裂带。新数据可以添加在数组的尾端，并且数组容量不需要比当前数据项的数量大很多。

字典

前面描述的雇员信息数据库的关键字“表现”得很好，而在许多应用中，情况并非如此。经典的例子是字典。如果想要把一本英文字典的每个单词，从 a 到 zyzzyva（这当然是一个单词），都写入计算机内存，以便快速读写，那么哈希表是一个不错的选择。

哈希表还在另一个类似的领域得到广泛应用，这就是高级计算机语言的编译器，它们通常用哈希表保留符号表。符号表记录了程序员声明的所有变量和函数名，以及它们在内存中的地址。程序需要快速地访问这些名字，所以哈希表是理想的数据结构。

例如，想在内存中存储 50000 个英文单词。起初可能考虑每个单词占据一个数组单元，那么数组的大小是 50000，同时可以使用数组下标存取单词。这样，存取确实很快。但是数组下标和单词

有什么关系呢？例如给出一个单词 morphosis，怎么能找到它的数组下标呢？

把单词转化成数组下标

现在需要的是把单词转化成适当下标的系统。现在已经知道，计算机应用不同的编码方案，用数字代表单个的字符。其中一种是 ASCII 编码，其中 a 是 97，b 是 98，依此类推，直到 122 代表 z。

然而，ASCII 码从 0 到 255，可以容纳字母、标点等字符。英文单词中只有 26 个字母，所以可以设计出一种自己的编码方案，它可以潜在地存储内存空间。其中 a 是 1，b 是 2，c 是 3，依此类推，直到 26 代表 z。还要把空格用 0 代表，所以有 27 个字符。（这个字典中不使用大写字母。）

如何把代表单个字母的数字组合成代表整个单词的数字呢？这有许多方法。下面看两种有代表性的方法，以及它们的优点和缺点。

把数字相加

一个转换单词的简单方法是把单词每个字符的代码求和。例如把单词 cats 转换成数字。首先，用前面创造的编码方案转换单词：

c = 3

a = 1

t = 20

s = 19

然后把它们相加：

3 + 1 + 20 + 19 = 43

那么，在字典中，单词 cats 存储在数组下标为 43 的单元中。所有的英文单词都可以用这个办法转换成数组下标。

这个方法工作得好吗？假设约定单词有十个字母。那么（记住空位是 0）字典的第一个单词 a 的编码是

0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 1 = 1

字典最后一个可能的单词是 zzzzzzzzzz（10 个 z）。所有的字符编码的和是

26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 = 260

因此，单词编码的范围是从 1 到 260。不幸的是，词典中有 50000 个单词，所以没有足够的数组下标数来索引那么多的单词。每个数组数据项大概要存储 192 个单词（50000 除以 260）。

很显然，用一个单词占用一个数组单元的方案会发生问题。也许可以考虑每个数组数据项包含一个子数组或链表。不幸的是，这个办法严重降低了存取速度。存取数据项确实很快，但是要在 192 个单词中找到其中一个，速度就很慢。

所以，第一次把单词转化成数字的尝试还留下一些问题要解决，比如有太多的单词需要用同一个数组下标。（例如，was、tin、give、tend、moan、tick、bails、dredge 和其他一百多个单词用编码方案求得的和都是 43，和 cats 一样。）那么就得到这样的结论，这个方法没有把单词分得足够开，所以结果数组能表达的元素太少。需要扩展数组表达下标的空间。

幂的连乘

我们来尝试一下用另外一种方法把单词映射成数字。如果数组容量太小，就确保它有足够的空

间。如果创建一个数组，使得每个单词，实际上是每个潜在的单词，从 a 到 zzzzzzzzzz，都可以占据数组中一个单元的话，那么将会发生什么？

如果要这么做，首先需要确定单词中的每个字符可以通过一种独一无二的方法得到最终的数字。

下面考虑一种用数字替换单词的类似方法。在一个大于 10 的数字中，每一数位代表用数字乘以从当前位到个位的位数那么多个 10。那么 7564 的意思是

7*1000 + 5*100 + 4*10 + 6*1

或者写成 10 的幂的连乘：

$7*10^3 + 5*10^2 + 4*10^1 + 6*10^0$

（在计算机程序中，输入例程应用一个类似的连乘和加法，把从键盘输入的数列转换成内存中的数。）

在这个系统中，把数字拆成数列，用适当的 10 的幂乘以这些数位（因为有 10 个可能的数位），然后把乘积相加。

类似的方法，可以把单词分解成字母组合，把字母转化成它们的数字代码，乘以适当的 27 的幂（因为有 27 个可能的字符，包括空格），然后将结果相加。这就给出了每个单词对应的独一无二的数字。

例如要把单词 cats 转换成数字。首先像前面一样把数转化成数列，每个数字乘以相应的 27 的幂，然后将结果相加：

$3*27^3 + 1*27^2 + 20*27^1 + 19*27^0$

计算幂的值，得到

3*19683 + 1*729 + 20*27 + 19*1

字母代码与幂的值相乘，得到

59049 + 729 + 540 + 19

和为 60337。

这个过程确实可以为每个可能的单词创建一个独一无二的整数。刚刚只是计算了一个 4 个字母的单词。较长的单词会发生什么？不幸的是，整数的范围会变得非常大。最长的 10 个字母的单词，zzzzzzzzzz，将转化成

$26*27^9 + 26*27^8 + 26*27^7 + 26*27^6 + 26*27^5 + 26*27^4 + 26*27^3 + 26*27^2 + 26*27^1 + 26*27^0$

仅 27^9 就超过 7000000000000，所以可以看到结果非常巨大。在内存中的数组根本不可能有这么多的单元。

这个问题的出现是因为这个方案为每个可能的单词分配了一个数组单元，不管这个单词是不是真正的英语单词。因此数组单元就从 aaaaaaaaaa，aaaaaaaaab，aaaaaaaaac，一直到 zzzzzzzzzz。这些单元中只有一小部分存放了存在的英语单词，这是必需的，而大多数单元都是空的。如图 11.2 显示的情况。

第一种方案（把数字相加求和）产生的数组下标太少。第二种方案（与 27 的幂相乘并求和）产生的数组下标又太多。

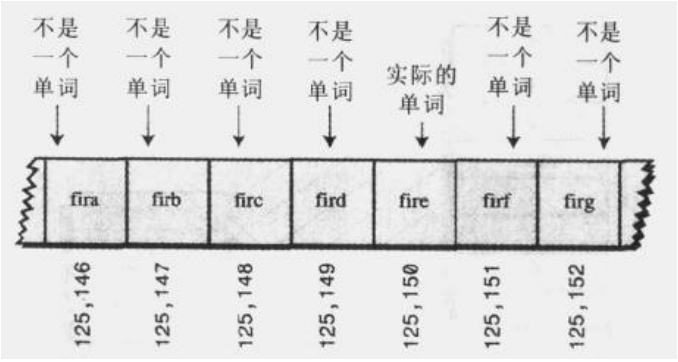


图 11.2 每个可能单词的索引

哈希化

现在需要一种压缩方法，把数位幂的连乘系统中得到的巨大的整数范围压缩到可接受的数组范围中。

对于英语词典，多大的数组才合适？如果只有 50000 个单词，可能会假设这个数组大概就有这么多空间。但实际上，需要多一倍的空间容纳这些单词。（后面就会知道为什么这么说。）所以，最终需要容量为 100000 的数组。

现在，就找一种方法，把 0 到超过 7000000000000 的范围，压缩为从 0 到 100000。有一种简单的方法是使用取余操作符，它的作用是得到一个数被另外一个数整除后的余数。

为了看到这个方法如何工作，首先来看在一个较小的，较易理解的数组范围如何运作。假设把从 0 到 199 的数字（用变量 largeNumber 代表），压缩为从 0 到 9 的数字（用变量 smallNumber 代表）。后者有 10 个数，所以说变量 smallRange 值为 10。而变量 largeNumber 的值是多少并不重要（除非它超过编程语言规定的变量大小）。这个转换的 Java 表达式为

```
smallNumber = largeNumber % smallRange;
```

当一个数被 10 整除时，余数一定在 0 到 9 之间；例如，13%10 为 3，157%10 为 7。如图 11.3 所示。这样，就把 0～199 的范围压缩到 0～9 的范围，压缩率为 20：1。

也可以用类似的方法把表示单词的惟一的数字压缩成数组的下标：

```
arrayIndex = hugeNumber % arraySize;
```

这就是一种哈希函数。它把一个大范围的数字哈希（转化）成一个小范围的数字。这个小的范围对应着数组的下标。使用哈希函数向数组插入数据后，这个数组就称为哈希表。（本章稍后部分将会详细介绍哈希函数的设计。）

回忆一下：通过把单词每个字母乘以 27 的适当次幂，使单词成为了一个巨大的数字。

hugeNumber = ch0*27^9 + ch1*27^8 + ch2*27^7 + ch3*27^6 + ch4*27^5 + ch5*27^4 + ch6*27^3 + ch7*27^2 + ch8*27^1 + ch9*27^0

然后，使用取余操作符（%），把得到的巨大整数范围转换成两倍于要存储内容的数组下标范围。下面是哈希函数的例子：

```
arraySize = numberWords * 2;
arrayIndex = hugeNumber % arraySize;
```

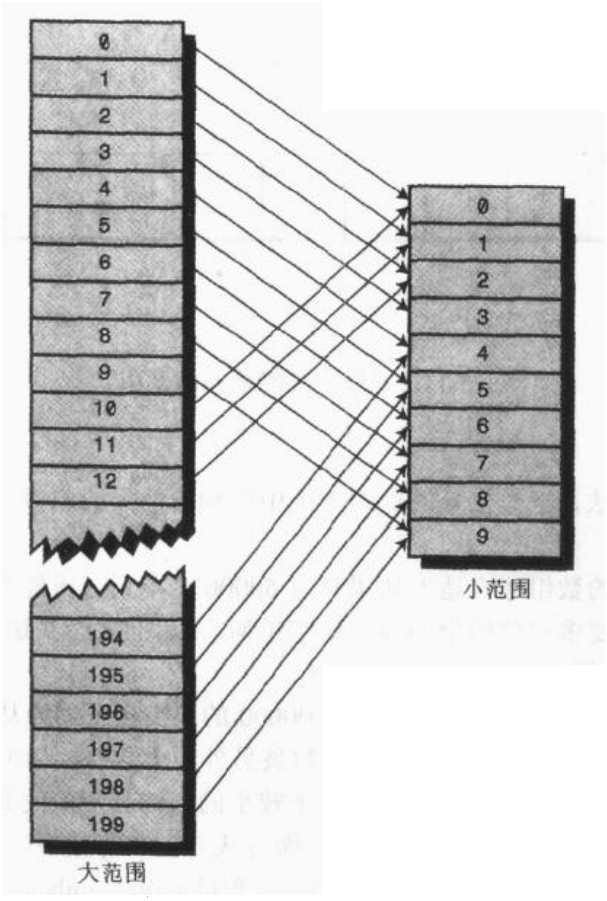


图 11.3 范围转换

在原始的范围中，每个数字代表一个潜在的数据项（一种字母组合），但是它们中间只有很少一部分代表真实数据（英语单词）。哈希函数把这个巨大的整数范围转换成小的多的数组的下标范围。期望的数组应该有这样的特点，平均起来，每两个数组单元，就有一个单词。有些单元没有单词；而有些有多个单词。

这个方案的实际实现会遇到问题，因为 hugeNumber 可能会超出变量范围，即使变量类型为 long。后面会看到如何处理这个问题。

冲突

把巨大的数字空间压缩成较小的数字空间，必然要付出代价，即不能保证，每个单词都映射到数组的空白单元。

这个情况和把字符代码相加时的情况类似，但是现在更麻烦。当把字符代码相加时，只有约 260 种可能（对于最多只有 10 个字符的单词）。现在，把这个可能性扩展到了 50000 种。

虽然如此，不可能避免把几个不同的单词哈希化到同一个数组单元，至少偶尔会这样。当然希望每个数组下标对应一个数据项，但是通常这不可能。只能寄希望于没有太多的单词有同样的数组下标。

假设要在数组中插入单词 melioration。通过哈希函数得到了它的数组下标后，发现那个单元已

经有一个单词（demystify）了，因为这个单词哈希化后得到的数组下标与 melioration 相同（对一个特定大小的数组）。如图 11.4 所示，这种情况称为冲突。

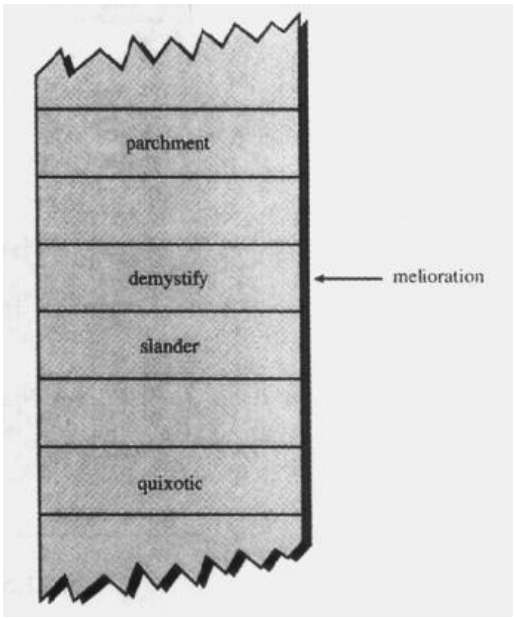


图 11.4 冲突

冲突的可能性会导致哈希化方案无法实施，实际上，可以通过另外的方式解决这个问题。记住，前面已经说过，指定的数组大小两倍于需要存储的数据量。因此，可能一半的单元是空的。当冲突发生时，一个方法是通过系统的方法找到数组的一个空位，并把这个单词填入，而不再用哈希函数得到的数组下标。这个方法叫做开放地址法。例如，如果 cats 哈希化的结果是 5421，但它的位置已经被 parsnip 占用，那么可能会考虑把 cats 放在 5422 的位置上。

第二种方法（前面提到过）是创建一个存放单词链表的数组，数组内不直接存储单词。这样，当发生冲突时，新的数据项直接接到这个数组下标所指的链表中。这种方法叫做链地址法。

本章的剩余部分将讨论开放地址法和链地址法，最后回到哈希函数的问题。

迄今为止，一直都在关注如何哈希化字符串。这是很现实的，因为许多哈希表用来存储字符串。然而，也有不少哈希表是存储数字的，例如前面提到的雇员号码的例子。在下面的讨论，以及专题 applet 中，都使用数字（而不是字符串）作为关键字。这样事情容易理解，编程也会变得容易。然而，应该记住在许多情况中，这些数字是从字符串中得到的。

开放地址法

在开放地址法中，若数据不能直接放在由哈希函数计算出来的数组下标所指的单元时，就要寻找数组的其他位置。下面要探索开放地址法的三种方法，它们在找下一个空白单元时使用的方法不同。这三种方法分别是线性探测、二次探测和再哈希法。

线性探测

在线性探测中，线性地查找空白单元。如果 5421 是要插入数据的位置，它已经被占用了，那么就使用 5422，然后是 5423，依此类推，数组下标一直递增，直到找到空位。这就叫做线性探测，因为它沿着数组的下标一步一步顺序地查找空白单元。

Hash 专题 applet

Hash 专题 applet 演示了线性探测。打开这个 applet，看到类似图 11.5 的界面。

在这个 applet 中，关键字的范围从 0 到 999。数组的初始大小为 60。哈希函数必须把关键字的范围压缩到数组的范围。用前面用过的取余操作符（%）来完成：

```
arrayIndex = key % arraySize;
```

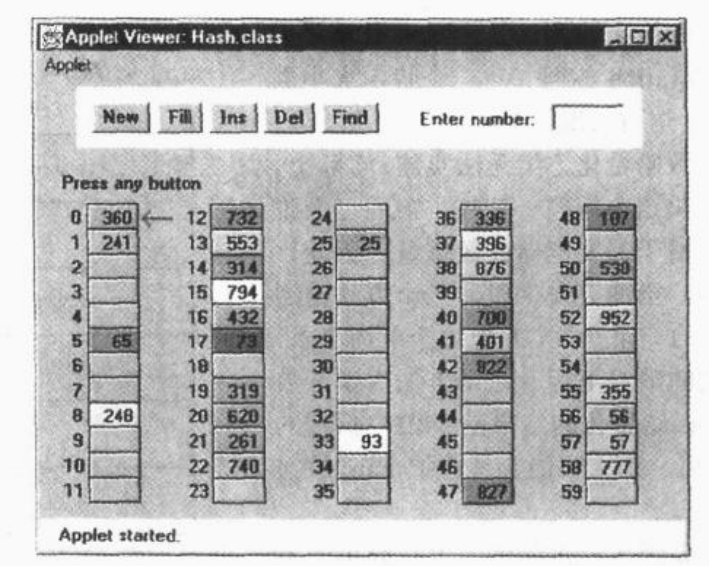


图 11.5 开始时的 Hash 专题 applet

针对数组的大小为 60，即

```
arrayIndex = key % 60;
```

这个哈希函数足够简单，可以手算出来。对给定的关键字，每次都减掉 60，直到结果小于 60。例如，哈希化 143，减 60 是 83，再减 60 是 23。这就是算法决定的 143 应该放的位置。因此，可以容易地检查算法是否把关键词映射到正确的地址。（大小为 10 的数组更容易计算，因为关键字的最后一位就是哈希化的结果。）

像其他的 applet 一样，这个 applet 也要反复按同一个按钮来执行操作。例如，为了找到特定的关键字，反复点击 Find 按钮。记住，使用其他按钮前，先完成某个按钮的该执行的操作。例如，直到 Press any key 的消息显示后，再从 Fill 按钮换到其他按钮。

所有的操作需要在序列的开始，键入一个数值。例如，Find 按钮要求键入一个数值，而 New 按钮要求键入新哈希表的大小。

New 按钮

用 New 按钮可以创建指定大小的哈希表。最大值为 60；由于这个 applet 窗口显示的单元数有限，所以数组的大小有限制。原始大小是 60，使用这个数是因为用它能很方便地检测哈希值是否合适，但正如后面看到的，在通用的哈希表中，数组的大小应该是素数，所以，59 是更好的选择。

Fill 按钮

最开始，哈希表包含 30 项，所以，有一半空间是满的。然而，也可以用 Fill 按钮向哈希表填充指定数目的数据。点击 Fill，当出现提示时，点击要填入的数据项数目。当数据项数目占哈希表长的一半，或最多到三分之二（60 个单元的表容纳 40 个数据项）时，哈希表的性能最好。

可以看出，已填充的单元的分布并不均匀。有时有一串空白单元，有时又有一串已填充单元。

哈希表中，一串连续的已填充单元叫做填充序列。增加越来越多的数据项时，填充序列变的越

来越长。这叫做聚集，如图 11.6 所示。

使用 applet 时，注意，如果把哈希表填得太满（例如，在 60 个单元的表中试图容纳 59 个数据项），那么在表中每填入一个数据项都要花费很长时间。可能会认为程序已经停止。但是请忍耐，哈希表在几乎被填满的数组中添加数据项，效率的确非常低。

而且，注意如果哈希表被完全填满，算法就会停止工作；在 applet 中假定，哈希表至少有一个空白单元。

Find 按钮

Find 按钮对键入的关键字值应用哈希函数。结果是一个数组下标。这个下标所指单元可能是要寻找的关键字；这是最好的情况，并且立即报告查找成功。

然而，也可能这个单元被其他关键字占据。这就是冲突；会看到红色箭头指向被占用的单元。根据冲突的位置，查找算法依次查找下一个单元。查找合适单元的过程叫做探测。

根据冲突的位置，查找算法只是沿着数组一个一个地察看每个单元。如果在找到要寻找的关键字前遇到一个空位，说明查找失败。不需要再做查找，因为插入算法本应该把这个数据项插在那个空位上（如果不是前面那样的话）。图 11.7 显示了成功和不成功的线性探测。

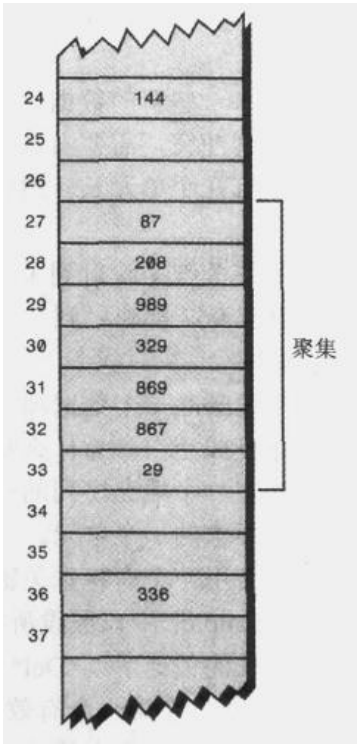
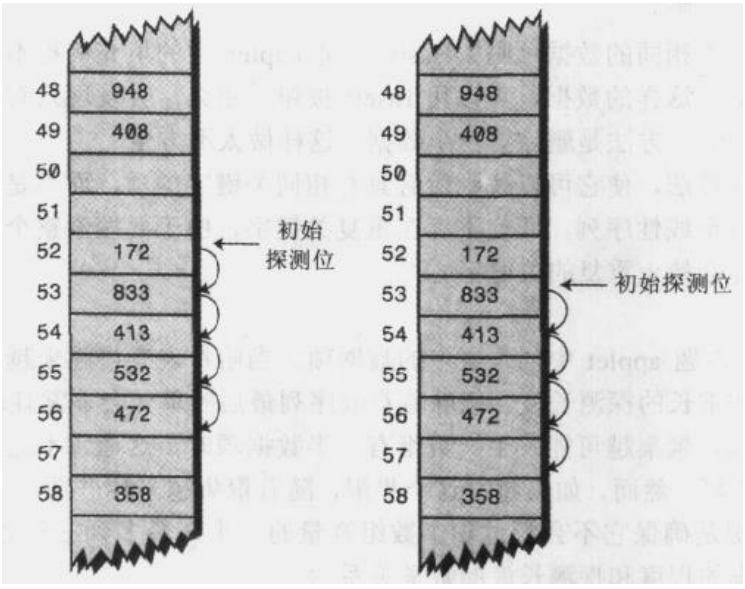


图 11.6 聚集的例子



a）对 472 的成功查找　　b）对 893 的不成功查找

图 11.7 线性探测

Ins 按钮

Ins 按钮向哈希表中插入一个数据项，它的关键字值由用户键入。插入算法使用和查找算法同

样的算法找到合适的单元。如果最初的单元被占用，它会用线性探索，寻找空白单元。当找到后，就插入数据。

试着插入一些新的数据项。键入一个三位数，看发生了什么。大多数数据项会一次插入成功，但有些会遇到冲突，需要沿数组步进，查找新的空白单元。它们走过的步数成为探索长度。大多数探索长度只有几个单元长。然而，有时会达到四个或五个单元长，当数组过分满的时候，探索长度可能更长。

注意哪些关键字映射到了相同的下标。如果数组大小是 60，那么关键字 7，67，127，187，247 等等，一直到 967 都映射到 7。试着插入这个序列或类似的序列。这样的序列证明了线性探测方法。

Del 按钮

Del 按钮删除一个数据项，它的关键字同样由用户输入。删除不是简单地把某个单元的数据项删除，把它变成空白。为什么呢？记住，在插入操作中，探测过程走过一系列单元，查找一个空白单元。如果在一个填充序列的中间有一个空位，查找算法就会在看到它时中途放弃查找，即使最终本可以到达要求的那个单元。

因此，要用一个有特殊关键字值的数据项替代要被删除的数据项，以此标识此处的数据已不存在。在这个 applet 中，假设所有有效的关键字值都是正数，所以被删除的数据变成－1。被删除的数据项用特殊的关键字（*Del*）标记。

Insert 按钮将在第一个有效空位或值为*Del*的位置插入一个新数据项。Find 按钮会把值为*Del*的数据项作为一个已存在的项，为的是可以跨过这项，查找更多的数据项。

如果做了很多次删除操作；哈希表就会充满值为*Del*的数据项，这使得哈希表效率下降。因此许多哈希表不允许删除操作。如果实现了删除，应该尽量有节制地使用删除。

允许有重复的值吗？

在哈希表中允许相同的数据项吗？Hash 专题 applet 中的填充例程不允许有相同关键字的数据项，但是如果要插入这样的数据，可以用 Insert 按钮。那么，会发现只有第一个数据可以被存取。存取第二个数据的惟一方法是删除第一个数据。这样做太不方便。

可以重写查找算法，使它可以找到所有具有相同关键字的项，而不是只找第一个。然而，这需要搜索它遇到的每个线性序列。即使不存在重复关键字，由于要搜索整个表，所以非常耗时。在大多数情况下，可能会禁止重复的关键字。

聚集

试着在 Hash 专题 applet 中插入更多的数据项。当哈希表变得越来越满时，聚集变得越来越严重。这导致产生非常长的探测长度。意味着存取序列最后的单元会非常耗时。

数组填得越满，聚集越可能发生。数组有一半数据项时，这通常不是问题，当三分之二满的时候，情况也不会太坏。然而，如果超过这个界限，随着聚集越来越严重，性能下降也很严重。因此，设计哈希表的关键是确保它不会超过整个数组容量的一半，最多到三分之二。（在本章最后将讨论哈希表的装填数据的程度和探测长度的数学关系。）

线性探测哈希表的 Java 代码

在线性探测哈希表中，编写方法实现查找、插入和删除都不困难。下面展示这些方法的 Java 代码，然后是完整的 hash.java 程序，它包含了所有这些方法。

find()方法

find()方法首先调用 hashFunc()方法把待查找关键字转换成数组下标 hashVal。hashFunc()方法把%操作符应用于查找关键字和数组容量，这在前面已经看到。

接着，在 while 循环中，find()方法检查这个下标所指单元是否为空（null）。如果不为空，它检查这个数据项是否包含待检查关键字。如果包含，find()方法返回这个数据项。如果不包含，find()递增 hashVal，并且回到 while 循环的开始，检查下一个单元是否被占用。下面是 find()方法的代码：

```
public DataItem find(int key)     // find item with key
// (assumes table not full)
   {
   int hashVal = hashFunc(key);  // hash the key

   while(hashArray[hashVal] != null)  // until empty cell,
      {                               // found the key?
      if(hashArray[hashVal].getKey() == key)
         return hashArray[hashVal];   // yes, return item
      ++hashVal;                  // go to next cell
      hashVal %= arraySize;       // wrap around if necessary
      }
   return null;                   // can't find item
   }
```

随着 hashVal 递增，它最终到达数组的尾端。这种情况发生时，我们希望它能再次回到数组的起始端。用一个 if 语句来检查这种情况，只要 hashVal 的值等于数组大小，就把它的值置为 0。然而，也可以对 hashVal 的值进行取余操作，效果是一样的。

谨慎的程序员可能想到要判断哈希表是否满，代码中没有进行判断。哈希表不应该允许被填满，如果它被填满，这个方法会永远循环下去。简单起见，我们并没有检查这个情况。

insert()方法

下面显示的 insert()方法使用和 find()方法类似的算法定位数据项。然而，它要寻找一个空位，或者一个被删除的数据项（关键字为-1），而不是特定项。当找到一个空位后，insert()方法插入新的数据项。

```
public void insert(DataItem item) // insert a DataItem
// (assumes table not full)
   {
   int key = item.getKey();      // extract key
   int hashVal = hashFunc(key);  // hash the key
                                 // until empty cell or -1,
   while(hashArray[hashVal] != null &&
                            hashArray[hashVal].iData != -1)
      {
      ++hashVal;                 // go to next cell
      hashVal %= arraySize;      // wrap around if necessary
      }
```

```
    hashArray[hashVal] = item;     // insert item
    } // end insert()
```

delete()方法

下面的 delete()方法用 find()方法的算法找到一个已存在的项。当找到这个数据项后，delete()方法用特殊的数据项 nonItem 覆盖原来的数据，这个变量事先定义为-1。

```
public DataItem delete(int key)  // delete a DataItem
   {
   int hashVal = hashFunc(key);  // hash the key

   while(hashArray[hashVal] != null)  // until empty cell,
      {                               // found the key?
      if(hashArray[hashVal].getKey() == key)
      {
      DataItem temp = hashArray[hashVal]; // save item
      hashArray[hashVal] = nonItem;       // delete item
      return temp;                        // return item
      }
   ++hashVal;                 // go to next cell
   hashVal %= arraySize;      // wrap around if necessary
   }
return null;                  // can't find item
} // end delete()
```

hash.java 程序

清单 11.1 显示了完整的 hash.java 程序。在这个程序中，DataItem 对象只包含一个域，即作为关键字的整数。跟前面讨论的数据结构一样，这个对象可以包含其他数据或对其他类对象（例如 employee 或 partNumber）的引用。

HashTable 类的主要字段是名为 hashArray 的数组。其他的字段有数组的大小，和用于删除的 nonItem 对象。

清单 11.1 hash.java 程序

```
// hash.java
// demonstrates hash table with linear probing
// to run this program: C:>java HashTableApp
import java.io.*;
////////////////////////////////////////////////////////////////
class DataItem
   {                              // (could have more data)
   private int iData;             // data item (key)
//-------------------------------------------------------------
   public DataItem(int ii)        // constructor
      { iData = ii; }
//-------------------------------------------------------------
   public int getKey()
```

```
    { return iData; }
//--------------------------------------------------------------
  } // end class DataItem
////////////////////////////////////////////////////////////////
class HashTable
  {
  private DataItem[] hashArray;    // array holds hash table
  private int arraySize;
  private DataItem nonItem;        // for deleted items
// -------------------------------------------------------------
  public HashTable(int size)       // constructor
    {
    arraySize = size;
    hashArray = new DataItem[arraySize];
    nonItem = new DataItem(-1);    // deleted item key is -1
    }
// -------------------------------------------------------------
  public void displayTable()
    {
    System.out.print("Table: ");
    for(int j=0; j<arraySize; j++)
      {
      if(hashArray[j] != null)
        System.out.print(hashArray[j].getKey() + " ");
      else
        System.out.print("** ");
      }
    System.out.println("");
    }
// -------------------------------------------------------------
  public int hashFunc(int key)
    {
    return key % arraySize;        // hash function
    }
// -------------------------------------------------------------
  public void insert(DataItem item) // insert a DataItem
  // (assumes table not full)
    {
    int key = item.getKey();       // extract key
    int hashVal = hashFunc(key);   // hash the key
                                   // until empty cell or -1,
    while(hashArray[hashVal] != null &&
                  hashArray[hashVal].getKey() != -1)
      {
      ++hashVal;                   // go to next cell
```

```
        hashVal %= arraySize;       // wraparound if necessary
        }
      hashArray[hashVal] = item;    // insert item
      } // end insert()
// --------------------------------------------------------
   public DataItem delete(int key)  // delete a DataItem
      {
      int hashVal = hashFunc(key);  // hash the key

      while(hashArray[hashVal] != null)  // until empty cell,
         {                               // found the key?
         if(hashArray[hashVal].getKey() == key)
            {
            DataItem temp = hashArray[hashVal]; // save item
            hashArray[hashVal] = nonItem;       // delete item
            return temp;                        // return item
            }
         ++hashVal;                 // go to next cell
         hashVal %= arraySize;      // wraparound if necessary
         }
      return null;                  // can't find item
      } // end delete()
// --------------------------------------------------------
   public DataItem find(int key)    // find item with key
      {
      int hashVal = hashFunc(key);  // hash the key

      while(hashArray[hashVal] != null)  // until empty cell,
         {                               // found the key?
         if(hashArray[hashVal].getKey() == key)
            return hashArray[hashVal];   // yes, return item
         ++hashVal;                 // go to next cell
         hashVal %= arraySize;      // wraparound if necessary
         }
      return null;                  // can't find item
      }
// --------------------------------------------------------
   } // end class HashTable
////////////////////////////////////////////////////////////////
class HashTableApp
   {
   public static void main(String[] args) throws IOException
      {
      DataItem aDataItem;
      int aKey, size, n, keysPerCell;
```

```
                              // get sizes
System.out.print("Enter size of hash table: ");
size = getInt();
System.out.print("Enter initial number of items: ");
n = getInt();
keysPerCell = 10;
                              // make table
HashTable theHashTable = new HashTable(size);

for(int j=0; j<n; j++)        // insert data
   {
   aKey = (int)(java.lang.Math.random() *
                                    keysPerCell * size);
   aDataItem = new DataItem(aKey);
   theHashTable.insert(aDataItem);
   }

while(true)                   // interact with user
   {
   System.out.print('Enter first letter of ");
   System.out.print('show, insert, delete, or find: ");
   char choice = getChar();
   switch(choice)
      {
      case 's':
         theHashTable.displayTable();
         break;
      case 'i':
      System.out.print("Enter key value to insert: ");
         aKey = getInt();
         aDataItem = new DataItem(aKey);
         theHashTable.insert(aDataItem);
         break;
      case 'd':
         System.out.print("Enter key value to delete: ");
         aKey = getInt();
         theHashTable.delete(aKey);
         break;
      case 'f':
         System.out.print("Enter key value to find: ");
         aKey = getInt();
               aDataItem = theHashTable.find(aKey);
               if(aDataItem != null)
                  {
                  System.out.println("Found " + aKey);
```

```
                }
             else
                System.out.println("Could not find " + aKey);
             break;
          default:
             System.out.print("Invalid entry\n");
          } // end switch
       } // end while
    } // end main()
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
//--------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//--------------------------------------------------------------
   } // end class HashTableApp
////////////////////////////////////////////////////////////////
```

HashTableApp 类的 main()方法包含一个用户界面，允许用户显示哈希表的内容（键入 s），插入数据项（键入 i），删除数据项（键入 d），或者寻找数据项（键入 f）。

最开始，程序要求用户输入哈希表的大小和要存储的数据项个数。能够输入任意大小的数，从几项到 10000 项。（构建比这更大的哈希表需要费些时间。）如果哈希表有几百项，就不要使用 s 选项（速度太慢）；显示哈希表会翻好几屏，而且花费很多时间。

main()方法中的变量 keysPerCell 指定了关键字范围的比率，它是以数组大小为基础的。在程序清单中，它的值为 10。意思是如果指定表长为 20，那么关键字就从 0 到 200。

如果想看哈希表的执行过程，最好创建一个少于 20 项的哈希表，为的是所有数据项都可以显示在一行中。下面是 hash.java 的一个应用实例：

```
Enter size of hash table: 12
Enter initial number of items: 8
```

```
Enter first letter of show, insert, delete, or find: s
Table: 108 13 0 ** ** 113 5 66 ** 117 ** 47

Enter first letter of show, insert, delete, or find: f
Enter key value to find: 66
Found 66

Enter first letter of show, insert, delete, or find: i
Enter key value to insert: 100
Enter first letter of show, insert, delete, or find: s
Table: 108 13 0 ** 100 113 5 66 ** 117 ** 47

Enter first letter of show, insert, delete, or find: d
Enter key value to delete: 100
Enter first letter of show, insert, delete, or find: s
Table: 108 13 0 ** -1 113 5 66 ** 117 ** 47
```

关键字从 0 到 119（12 乘以 10，减 1）。**符号标志某个单元是空的。关键字为 100 的项插在位置 4（从 0 开始计数），因为 100%12 为 4。注意，删除该项时，100 是如何变成－1 的。

扩展数组

当哈希表变得太满时，一个选择是扩展数组。在 Java 中，数组有固定的大小，而且不能扩展。编程时只能另外创建一个新的更大的数组，然后把旧数组的所有内容插入到新的数组中。

记住，哈希函数根据数组大小计算给定数据项的位置，所以，这些数据项不能再放在新数组中和老数组相同的位置上。因此，不能简单地从一个数组向另一个数组拷贝数据。需要按顺序遍历老数组，用 insert()方法向新数组中插入每个数据项。这叫作重新哈希化。这是一个耗时的过程，但如果数组要进行扩展，这个过程就是必要的。

扩展后的数组容量通常是原来的两倍。实际上，因为数组容量应该是一个质数，所以新数组要比两倍的容量多一点。计算新数组的容量是重新哈希化的一部分。

下面是帮助找到新数组容量（或原始的数组容量，因为可能会怀疑用户没有选择一个质数，这种情况经常发生）的几个例程。

```
private int getPrime(int min)   // returns 1st prime > min
      {
      for(int j = min+1; true; j++)    // for all j > min
         if( isPrime(j) )              // is j prime?
            return j;                  // yes, return it
      }
// ------------------------------------------------------------
   private boolean isPrime(int n)      // is n prime?
      {
      for(int j=2; (j*j <= n); j++)    // for all j
         if( n % j == 0)               // divides evenly by j?
            return false;              // yes, so not prime
```

```
return true;                    // no, so prime
}
```

例程最终不会混合。例如，在 getPrime()方法中，可以检查 2，然后只检查奇数，而不是每个数字。然而，这种优化没有多少用，因为只需检查几个数字，就会找到一个质数。

Java 提供了 Vector 类，这个类似数组的数据结构的类可以扩展。然而，它没有太大帮助，因为数组容量改变后需要重新哈希化所有数据项。

二次探测

前面已经看到，在开放地址法的线性探测中会发生聚集。一旦聚集形成，它会变得越来越大。那些哈希化后的落在聚集范围内的数据项，都要一步一步移动，并且插在聚集的最后，因此使聚集变得更大。聚集越大，它增长得也越快。

这就像人群，当某个人在商场晕倒，人群就慢慢聚集。最初的人聚过来是因为看到了那个倒下的人；后面的人聚过来因为他们想知道每个人都在看什么。人群聚得越大，吸引的人就会越多。

已填入哈希表的数据项和表长的比率叫做装填因子。有 10000 个单元的哈希表填入 6667 个数据后，它的装填因子是 2/3。

```
loadFactor = nItems / arraySize;
```

当装填因子不太大时，聚集分布得比较连贯。 哈希表的某个部分可能包含大量的聚集，而另一个部分还很稀疏。聚集降低了哈希表的性能。

二次探测是防止聚集产生的一种尝试。思想是探测相隔较远的单元，而不是和原始位置相邻的单元。

步骤是步数的平方

在线性探测中，如果哈希函数计算的原始下标是 x，线性探测就是 x+1, x+2, x+3，依此类推。而在二次探测中，探测的过程是 x+1, x+4, x+9, x+16, x+25，依此类推。到原始位置的距离是步数的平方：$x+1^2$, $x+2^2$, $x+3^2$, $x+4^2$, $x+5^2$，等等。

图 11.8 显示了一些线性探测的情况。

当二次探测的搜索变长时，好像它变得越来越绝望。第一次，它查找相邻的单元。如果这个单元被占用，它认为这里可能有一个小的聚集，所以，它尝试距离为 4 的单元。如果这里也被占用，它变得有些焦虑，认为这里有一个大的聚集，然后尝试距离为 9 的单元。如果这里还被占用，它感到一丝恐慌，跳到距离为 16 的单元。很快，它会歇斯底里地飞跃整个数组空间。当哈希表几乎填满时，在 HashDouble 专题 applet 中执行查找操作，就会出现这种情况。

包含二次查找的 HashDouble Applet

HashDouble 专题 applet 允许使用两种处理冲突的方法：二次探测和再哈希法。（下节将讲解再哈希法。）这个 applet 生成的界面与 Hash 专题 applet 的基本相同，除了它多了一个单选按钮，用以选择二次探测或再哈希法。

为了看到二次探测如何工作，打开这个 applet，用 New 按钮创建一个 59 项的哈希表。当要求选择二次探测还是再哈希法时，点击 Quad 按钮。新表创建后，用 Fill 按钮填数据项，直到哈希表达到 4/5 满（有 47 个数据项）。这时表太满，但是会产生更长的探测，因此可以研究探测算法。

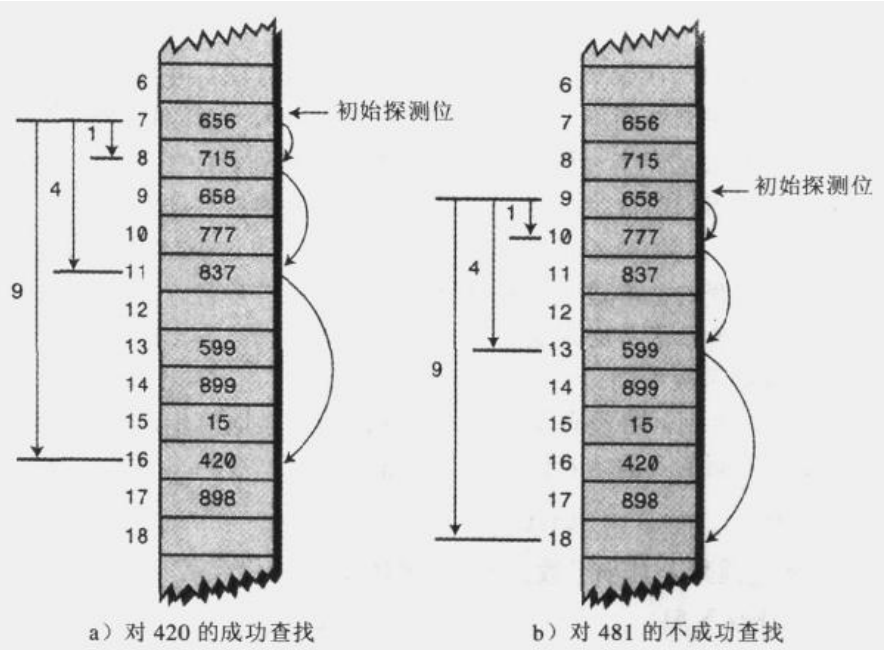


图 11.8 二次探测

顺便提一下，如果试图把哈希表填得太满，就会看到这样的信息：Can't complete fill （不能全部填满）。当探测序列太长时，就会出现这个提示。探测序列每增加一步，跨过的单元就越多。如果序列太长，步长最终会超过整型变量的范围，从而导致这个 applet 停止填表的过程。

填好表后，选择一个已存在的关键字值，用 Find 按钮看算法是否可以找到它。通常，可以在初始的单元找到该关键字，或是在相邻的单元。然而，如果有耐心可以找一个需要探测三、四步的关键字值，这样可以看到每一步的步长。也可以找一个不存在的值，这样的查找会一直持续，直到遇到一个空位。

提示

重要：数组容量总是选一个质数。例如用 59 代替 60。（小于 60 的质数还有 53，47，43，41，37，31，29，23，19，17，13，11，7，5，3 和 2。）如果数组容量不是质数，在探测过程中，步长序列就会变得无限长。如果在填表操作中发生这种情况，这个 applet 将会崩溃。

二次探测的问题

二次探测消除了在线性探测中产生的聚集问题，这种聚集问题叫做原始聚集。然而，二次探测产生了另外一种，更细的聚集问题。之所以会发生，是因为所有映射到同一个位置的关键字在寻找空位时，探测的单元都是一样的。

比如将 184，302，420 和 544 依次插入到表中，它们都映射到 7。那么 302 需要以一为步长的探测，420 需要以四为步长的探测，544 需要以九为步长的探测。只要有一项，其关键字映射到 7，就需要更长步长的探测。这个现象叫做二次聚集。

二次聚集不是一个严重的问题，但是，二次探测不会经常使用，因为还有稍微好些的解决方案。

再哈希法

为了消除原始聚集和二次聚集，可以使用另外的一个方法：再哈希法。二次聚集产生的原因是，二次探测的算法产生的探测序列步长总是固定的：1，4，9，16，依此类推。

现在需要的一种方法是产生一种依赖关键字的探测序列，而不是每个关键字都一样。那么，不同的关键字即使映射到相同的数组下标，也可以使用不同的探测序列。

方法是把关键字用不同的哈希函数再做一遍哈希化，用这个结果作为步长。对指定的关键字，步长在整个探测中是不变的，不过不同的关键字使用不同的步长。

经验说明，第二个哈希函数必须具备如下特点：

- 和第一个哈希函数不同。
- 不能输出 0（否则，将没有步长；每次探测都是原地踏步，算法将陷入死循环）。

专家们已经发现下面形式的哈希函数工作得非常好：

```
stepSize = constant - (key % constant);
```

其中，constant 是质数，且小于数组容量。例如，

```
stepSize = 5 - (key % 5);
```

这是用在 HashDouble 专题 applet 中的第二个哈希函数。不同的关键字可能映射到相同的数组下标，但是，它们会（很有可能）产生不同的步长。用这个函数，步长的范围是从 1 到 5。如图 11.9 所示。

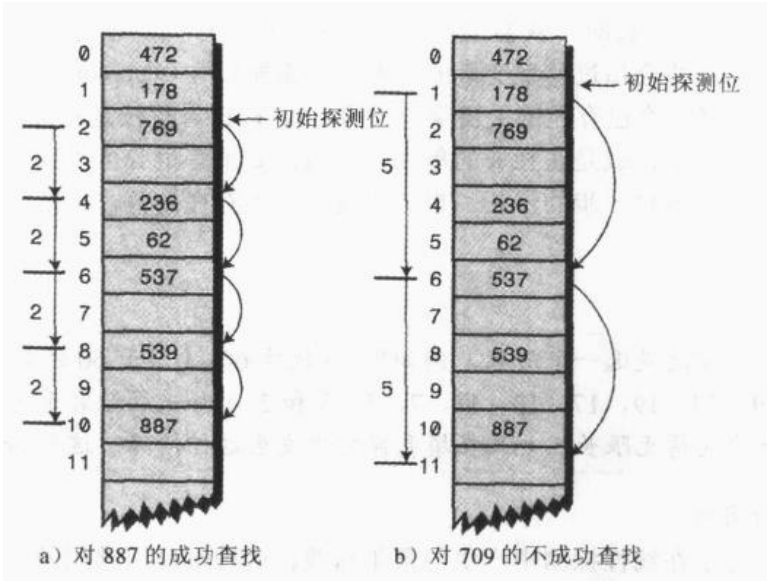


图 11.9 再哈希法

包含再哈希法的 HashDouble applet

可以使用 HashDouble 专题 applet 观察再哈希法如何工作。这个 applet 自动以再哈希法模式启动，但是，如果它是二次探测模式，需要用 New 按钮创建一个新数组，当出现提示时，点击 Double 按钮，以此切换至再哈希法模式。为了得到最好的效果，需要把表填得相对满一些，大约 9/10 或更大的比例。尽管有这么大的装填因子，大多数数据项还是能用一次哈希函数找到：只有一小部分需要使用扩展的探测序列。

试着找一个已存在的关键字，当有一个需要探测序列时，会看到，对给定的关键字每步走相同的步长，而不同的关键字有不同的（从 1 到 5）步长。

再哈希法的 Java 代码

清单 11.2 显示了 hashDouble.java 的完整清单，这个程序使用了再哈希法。它和 hash.java 程序（清单 11.1）类似，但是使用了两个哈希函数：一个用于找到原始位置，另一个生成步长。在这之前，用户可以显示表的内容，插入数据项，删除数据项，以及查找数据项。

清单 11.2 hashDouble.java 程序

```
// hashDouble.java
// demonstrates hash table with double hashing
// to run this program: C:>java HashDoubleApp
import java.io.*;
////////////////////////////////////////////////////////////////
class DataItem
   {                                // (could have more items)
   private int iData;               // data item (key)
//-----------------------------------------------------------
   public DataItem(int ii)          // constructor
      { iData = ii; }
//-----------------------------------------------------------
   public int getKey()
      { return iData; }
//-----------------------------------------------------------
   } // end class DataItem
////////////////////////////////////////////////////////////////
class HashTable
   {
   private DataItem[] hashArray;    // array is the hash table
   private int arraySize;
   private DataItem nonItem;        // for deleted items
// -----------------------------------------------------------
   HashTable(int size)              // constructor
      {
      arraySize = size;
      hashArray = new DataItem[arraySize];
      nonItem = new DataItem(-1);
      }
// -----------------------------------------------------------
   public void displayTable()
      {
      System.out.print("Table: ");
      for(int j=0; j<arraySize; j++)
         {
         if(hashArray[j] != null)
```

```
            System.out.print(hashArray[j].getKey()+ " ");
         else
            System.out.print("** ");
         }
      System.out.println("");
      }
// --------------------------------------------------------
   public int hashFunc1(int key)
      {
      return key % arraySize;
      }
// --------------------------------------------------------
   public int hashFunc2(int key)
      {
      // non-zero, less than array size, different from hF1
      // array size must be relatively prime to 5, 4, 3, and 2
      return 5 - key % 5;
      }
// --------------------------------------------------------
                                    // insert a DataItem
   public void insert(int key, DataItem item)
   // (assumes table not full)
      {
      int hashVal = hashFunc1(key);  // hash the key
      int stepSize = hashFunc2(key); // get step size
                                    // until empty cell or -1
      while(hashArray[hashVal] != null &&
                     hashArray[hashVal].getKey() != -1)
         {
         hashVal += stepSize;       // add the step
         hashVal %= arraySize;      // for wraparound
         }
      hashArray[hashVal] = item;    // insert item
      }  // end insert()
// --------------------------------------------------------
   public DataItem delete(int key)   // delete a DataItem
      {
      int hashVal = hashFunc1(key);      // hash the key
      int stepSize = hashFunc2(key);     // get step size
      while(hashArray[hashVal] != null)  // until empty cell,
         {                               // is correct hashVal?
         if(hashArray[hashVal].getKey() == key)
            {
            DataItem temp = hashArray[hashVal]; // save item
            hashArray[hashVal] = nonItem;       // delete item
```

```
            return temp;                        // return item
            }
         hashVal += stepSize;                   // add the step
         hashVal %= arraySize;                  // for wraparound
         }
      return null;                              // can't find item
      }  // end delete()
// ---------------------------------------------------------
   public DataItem find(int key)     // find item with key
   // (assumes table not full)
      {
      int hashVal = hashFunc1(key);      // hash the key
      int stepSize = hashFunc2(key);     // get step size

      while(hashArray[hashVal] != null)  // until empty cell,
         {                               // is correct hashVal?
         if(hashArray[hashVal].getKey() == key)
            return hashArray[hashVal];   // yes, return item
         hashVal += stepSize;            // add the step
         hashVal %= arraySize;           // for wraparound
         }
      return null;                       // can't find item
      }
// ---------------------------------------------------------
   }  // end class HashTable
////////////////////////////////////////////////////////////////
class HashDoubleApp
   {
   public static void main(String[] args) throws IOException
      {
      int aKey;
      DataItem aDataItem;
      int size, n;
                                  // get sizes
      System.out.print("Enter size of hash table: ");
 size = getInt();
 System.out.print("Enter initial number of items: ");
 n = getInt();
                            // make table
 HashTable theHashTable = new HashTable(size);

 for(int j=0; j<n; j++)       // insert data
    {
    aKey = (int)(java.lang.Math.random() * 2 * size);
    aDataItem = new DataItem(aKey);
```

```
      theHashTable.insert(aKey, aDataItem);
      }

   while(true)                  // interact with user
      {
      System.out.print("Enter first letter of ");
      System.out.print("show, insert, delete, or find: ");
      char choice = getChar();
      switch(choice)
         {
         case 's':
            theHashTable.displayTable();
            break;
         case 'i':
            System.out.print("Enter key value to insert: ");
            aKey = getInt();
            aDataItem = new DataItem(aKey);
            theHashTable.insert(aKey, aDataItem);
            break;
         case 'd':
            System.out.print("Enter key value to delete: ");
            aKey = getInt();
            theHashTable.delete(aKey);
            break;
         case 'f':
            System.out.print("Enter key value to find: ");
            aKey = getInt();
            aDataItem = theHashTable.find(aKey);
            if(aDataItem != null)
               System.out.println("Found " + aKey);
            else
                     System.out.println("Could not find " + aKey);
                  break;
               default:
                  System.out.print("Invalid entry\n");
               } // end switch
            } // end while
         } // end main()
   //--------------------------------------------------------------
      public static String getString() throws IOException
         {
         InputStreamReader isr = new InputStreamReader(System.in);
         BufferedReader br = new BufferedReader(isr);
         String s = br.readLine();
         return s;
```

```
    }
//-------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//-------------------------------------------------------------
   } // end class HashDoubleApp
////////////////////////////////////////////////////////////////
```

输出和这个程序的操作与 hash.java 类似。表 11.1 显示了当 21 个数据项插入到容量为 23 的哈希表的情况，处理冲突的方法使用再哈希法。步长从 1 到 5。

表 11.1 用再哈希法填充容量为 23 的表

项数	关键字	哈希值	步长大小	探查序列中的单元
1	1	1	4	
2	38	15	2	
3	37	14	3	
4	16	16	4	
5	20	20	5	
6	3	3	2	
7	11	11	4	
8	24	1	1	2
9	5	5	5	
10	16	16	4	20 1 5 9
11	10	10	5	
12	31	8	4	
13	18	18	2	
14	12	12	3	
15	30	7	5	
16	1	1	4	5 9 1 3
17	19	19	1	
18	36	13	4	17
19	41	18	4	22
20	15	15	5	20 2 7 12 17 22 4
21	25	2	5	7 12 17 22 4 9 14 19 16

前面 15 个关键字大多数映射到空白单元（第 10 个是不规则的）。这以后，随着数组越来越满，探测序列也越来越长。下面是数组的内容：

** 1 24 3 15 5 25 30 31 16 10 11 12 1 37 38 16 36 18 19 20 ** 41

表的容量是一个质数

再哈希法要求哈希表的容量是一个质数。为了考察为什么会有这个限制，假设表的容量不是质数。例如，假设表长是 15（下标从 0 到 14），有一个特定关键字映射到 0，步长为 5。探测序列是 0，5，10，0，5，10，依此类推，一直循环下去。算法只尝试这三个单元，所以不可能找到某些空白单元，例如位置 1，2，3 或其他位置。算法最终会导致崩溃。

如果数组容量是 13，即一个质数，探测序列最终会访问所有单元。即 0，5，10，2，7，12，4，9，1，6，11，3，一直下去。只要表中有一个空位，就可以探测到它。用质数作为数组容量使得任何数想整除它都是不可能的，因此探测序列最终会检查所有单元。

类似的影响在二次探测中也存在。然而，由于每步的步长都在变化，且最终会超出变量的范围，所以避免了无限的循环。

使用开放地址策略时，探测序列通常用再哈希法生成。

链地址法

开放地址法中，通过在哈希表中再寻找一个空位解决冲突问题。另一个方法是在哈希表每个单元中设置链表。某个数据项的关键字值还是像通常一样映射到哈希表的单元，而数据项本身插入到这个单元的链表中。其他同样映射到这个位置的数据项只需要加到链表中；不需要在原始的数组中寻找空位。图 11.10 显示了链地址法如何做的。

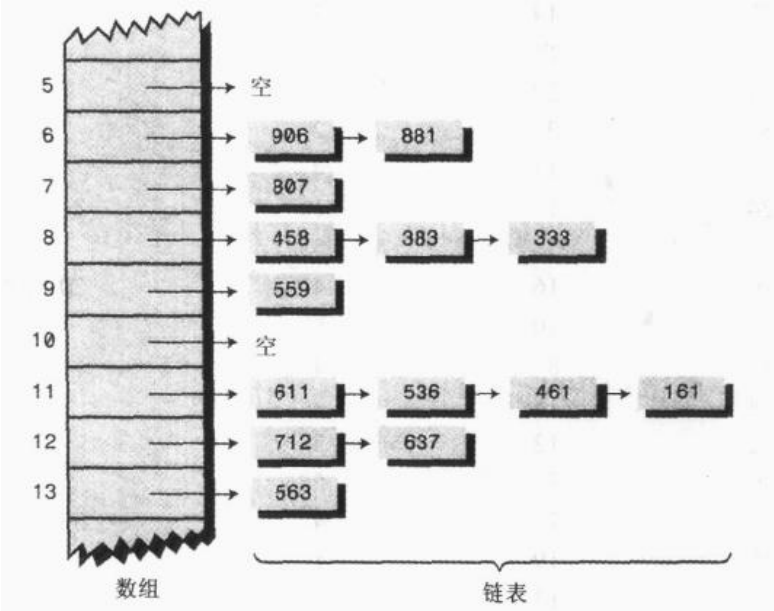


图 11.10 链地址法的例子

链地址法在概念上比开放地址法中的几种探测策略要简单。然而，代码会比其他的长，因为必

须要包含链表机制，这就要在程序中增加一个类。

HashChain 专题 Applet

为了考察链地址法如何工作，启动 HashChain 专题 applet。它显示一个带链表的数组，如图 11.11 所示。

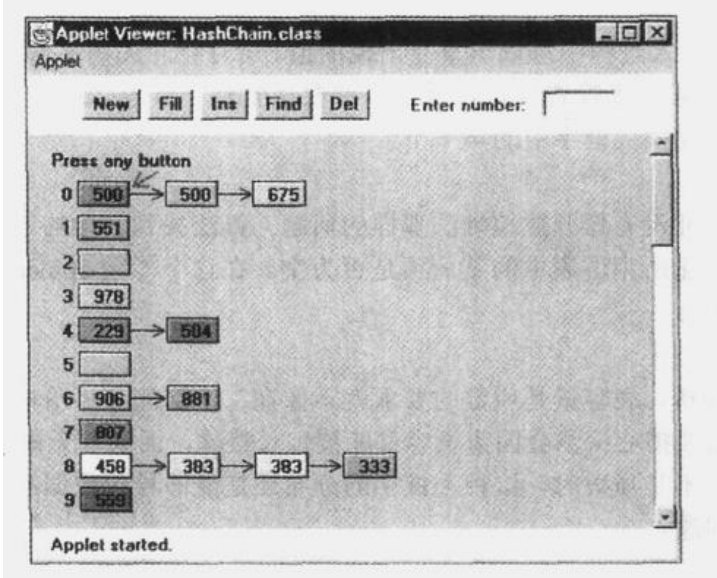


图 11.11 包含链地址法的 HashChain 专题 applet

数组的每个数据项占用显示区的一行，链表从左向右扩展。开始，数组中有 25 个单元（25 个链表）。一屏已经显示不下；可以拖动滚动条，以看到整个数组。每个链表只能显示前六项。可以创建一个带 100 个链表的哈希表，装填因子最大到 2.0。太高的装填因子可能导致链表超过六个数据项，那么多余的就会挤到屏幕外，不可能看到全部的数据项。（即使在装填因子为 2.0 的时候，这种情况也不常发生。）

用这个 applet 做个实验。用 Ins 按钮插入一些新的数据项。会看到红色箭头迅速移动到正确的链表中，把数据项插入到链表的表头。HashChain applet 中的链表并不排序，所以插入不需要搜索整个链表。（例程将示范有序链表。）

试用 Find 按钮找到指定的数据项。在查找操作中，如果链表中有好几项，红色箭头就逐项的查找正确的关键字。对于成功的查找，需要检查链表中平均一半的数据项，正如第五章中讨论的那样。对于不成功的查找，会检查所有的数据项。

装填因子

链地址法中的装填因子（数据项数和哈希表容量的比值）与开放地址法的不同。在链地址法中，需要在有 N 个单元的数组中装入 N 个或更多的数据项；因此，装填因子一般为 1，或比 1 大。这没有问题；因为，某些位置包含的链表中包含两个或两个以上的数据项。

当然，如果链表中有许多项，存取时间就会变长，因为存取特定数据项平均需要搜索链表的一半的数据项。找到初始的单元需要 O(1)的时间级，而搜索链表的时间与 M 成正比，M 为链表包含的平均项数。即 O(M)的时间级。因此，并不希望链表太满。

正如在刚才的专题 applet 中看到的，装填因子通常为 1。在这种情况下，大约三分之一的单元

是空白单元，三分之一的单元有一个数据项，三分之一有两个或更多的数据项。

在开放地址法中，当装填因子超过二分之一或三分之二后，性能下降得很快。在链地址法中，装填因子可以达到1以上，且对性能影响不大。因此，链地址法是更健壮的机制，特别是当事先难以确定哈希表要存储多少数据时更是如此。

重复值

这里允许重复，在填入过程中可以填重复出现的值。所有相同关键字值的项都放在同一链表中。所以如果需要找到所有项，不管查找是否成功，都要搜索整个链表。这会使性能略微下降。applet中的查找操作只能找到相同关键字中的第一个。

删除

链地址法中，删除并没有像开放地址法那样的问题。算法找到正确的链表，从链表中删除数据项。因为不需要探测，无所谓链表中的某一项是否为空。在这个专题applet中包含了Del按钮，可以看到删除操作如何进行。

表的容量

如果用链地址法的话，表容量是质数的要求就不像在二次探测和再哈希法中显得那么重要。这里没有探测，所以不需要担心探测会因为表容量能被步长整除，而陷入无限的循环中。

但是，当数组容量不是质数时，这种关键字的分布还是能够导致数据的聚集。在讨论哈希函数的时候还会谈到这个问题。

桶

另一种方法类似于链地址法，它在哈希表的每个单元中使用数组，而不是链表。这样的数组有时称为桶。然而，这个方法不如链表有效，因为桶容量不好选择。如果容量太小，可能会溢出。如果太大，又浪费空间。链表是动态分配的，所以没有这个问题。

链地址法的Java代码

hashChain.java程序包含了一个SortedList类和一个相关的Link类。有序链表不能加快成功的搜索，但可以减少不成功搜索中一半的时间。（只要有一项比要查找的值大，平均起来是在一半的位置，查找就宣告失败。）

删除的时间级也减少一半；然而，插入的时间延长了，因为新数据项不能只插在表头；插入前，必须找到有序表中的正确位置。如果链表很短，插入操作额外增加的时间不是很重要。

hashChain.java程序（如清单11.3所示）从构造一个哈希表开始，表长和项目数由用户决定。然后，用户能够插入，查找和删除数据项，并显示链表。为了整个哈希表可以显示在屏幕上，表长不要超过16。

清单11.3 hashChain.java程序

```
// hashChain.java
// demonstrates hash table with separate chaining
// to run this program: C:>java HashChainApp
import java.io.*;
////////////////////////////////////////////////////////////////
class Link
  {                                   // (could be other items)
```

```
   private int iData;                  // data item
   public Link next;                   // next link in list
// -------------------------------------------------------------
   public Link(int it)                 // constructor
      { iData= it; }
// -------------------------------------------------------------
   public int getKey()
      { return iData; }
// -------------------------------------------------------------
   public void displayLink()           // display this link
      { System.out.print(iData + " "); }
   }  // end class Link
////////////////////////////////////////////////////////////////
class SortedList
   {
   private Link first;                 // ref to first list item
// -------------------------------------------------------------
   public void SortedList()            // constructor
      { first = null; }
// -------------------------------------------------------------
   public void insert(Link theLink)    // insert link, in order
      {
      int key = theLink.getKey();
      Link previous = null;            // start at first
      Link current = first;
                                       // until end of list,
      while( current != null && key > current.getKey() )
         {                             // or current > key,
         previous = current;
         current = current.next;       // go to next item
         }
      if(previous==null)               // if beginning of list,
         first = theLink;              //    first --> new link
      else                             // not at beginning,
         previous.next = theLink;      //    prev --> new link
      theLink.next = current;          // new link --> current
      }  // end insert()
// -------------------------------------------------------------
   public void delete(int key)         // delete link
      {                                // (assumes non-empty list)
      Link previous = null;            // start at first
      Link current = first;
                                       // until end of list,
      while( current != null && key != current.getKey() )
         {                             // or key == current,
```

```
         previous = current;
         current = current.next;      // go to next link
         }
                                      // disconnect link
      if(previous==null)              //   if beginning of list
         first = first.next;          //      delete first link
      else                            //   not at beginning
         previous.next = current.next; //    delete current link
      } // end delete()
// ---------------------------------------------------------
   public Link find(int key)          // find link
      {
      Link current = first;           // start at first
                                      // until end of list,
      while(current != null &&  current.getKey() <= key)
         {                            // or key too small,
         if(current.getKey() == key)    // is this the link?
            return current;           // found it, return link
         current = current.next;      // go to next item
         }
      return null;                    // didn't find it
      } // end find()
// ---------------------------------------------------------
   public void displayList()
      {
      System.out.print("List (first-->last): ");
      Link current = first;       // start at beginning of list
      while(current != null)      // until end of list,
         {
         current.displayLink();   // print data
         current = current.next;  // move to next link
         }
      System.out.println("");
      }
   } // end class SortedList
////////////////////////////////////////////////////////////////
class HashTable
   {
   private SortedList[] hashArray;   // array of lists
   private int arraySize;
// ---------------------------------------------------------
   public HashTable(int size)        // constructor
      {
      arraySize = size;
      hashArray = new SortedList[arraySize];  // create array
```

```
      for(int j=0; j<arraySize; j++)          // fill array
         hashArray[j] = new SortedList();     // with lists
      }
// -------------------------------------------------------------
   public void displayTable()
      {
      for(int j=0; j<arraySize; j++) // for each cell,
         {
         System.out.print(j + ". "); // display cell number
         hashArray[j].displayList(); // display list
         }
      }
// -------------------------------------------------------------
   public int hashFunc(int key)       // hash function
      {
      return key % arraySize;
      }
// -------------------------------------------------------------
   public void insert(Link theLink)  // insert a link
      {
      int key = theLink.getKey();
      int hashVal = hashFunc(key);   // hash the key
      hashArray[hashVal].insert(theLink); // insert at hashVal
      }  // end insert()
// -------------------------------------------------------------
   public void delete(int key)       // delete a link
      {
      int hashVal = hashFunc(key);   // hash the key
      hashArray[hashVal].delete(key); // delete link
      }  // end delete()
// -------------------------------------------------------------
   public Link find(int key)         // find link
      {
      int hashVal = hashFunc(key);   // hash the key
      Link theLink = hashArray[hashVal].find(key);  // get link
      return theLink;                // return link
      }
// -------------------------------------------------------------
   }  // end class HashTable
////////////////////////////////////////////////////////////////
class HashChainApp
   {
   public static void main(String[] args) throws IOException
      {
int aKey;
Link aDataItem;
```

```
int size, n, keysPerCell = 100;
                                // get sizes
System.out.print("Enter size of hash table: ");
size = getInt();
System.out.print("Enter initial number of items: ");
n = getInt();
                                // make table
HashTable theHashTable = new HashTable(size);

for(int j=0; j<n; j++)          // insert data
   {
   aKey = (int)(java.lang.Math.random() *
                                      keysPerCell * size);
   aDataItem = new Link(aKey);
   theHashTable.insert(aDataItem);
   }
while(true)                     // interact with user
   {
   System.out.print("Enter first letter of ");
   System.out.print("show, insert, delete, or find: ");
   char choice = getChar();
   switch(choice)
      {
      case 's':
         theHashTable.displayTable();
         break;
      case 'i':
         System.out.print("Enter key value to insert: ");
         aKey = getInt();
         aDataItem = new Link(aKey);
         theHashTable.insert(aDataItem);
         break;
      case 'd':
         System.out.print("Enter key value to delete: ");
         aKey = getInt();
         theHashTable.delete(aKey);
         break;
      case 'f':
         System.out.print("Enter key value to find: ");
               aKey = getInt();
               aDataItem = theHashTable.find(aKey);
               if(aDataItem != null)
                  System.out.println("Found " + aKey);
               else
                  System.out.println("Could not find " + aKey);
```

```
                break;
             default:
                System.out.print("Invalid entry\n");
             } // end switch
          } // end while
       } // end main()
    //--------------------------------------------------------------
       public static String getString() throws IOException
          {
          InputStreamReader isr = new InputStreamReader(System.in);
          BufferedReader br = new BufferedReader(isr);
          String s = br.readLine();
          return s;
          }
    //--------------------------------------------------------------
       public static char getChar() throws IOException
          {
          String s = getString();
          return s.charAt(0);
          }
    //--------------------------------------------------------------
       public static int getInt() throws IOException
          {
          String s = getString();
          return Integer.parseInt(s);
          }
    //--------------------------------------------------------------
       } // end class HashChainApp
    ////////////////////////////////////////////////////////////////
```

下面是一段输出。用户创建了一个 20 个链表的哈希表，插入 20 个数据项，并且用 s 选项显示它：

```
Enter size of hash table: 20
Enter initial number of items: 20
Enter first letter of show, insert, delete, or find: s
0. List (first-->last): 240 1160
1. List (first-->last):
2. List (first-->last):
3. List (first-->last): 143
4. List (first-->last): 1004
5. List (first-->last): 1485 1585
6. List (first-->last):
7. List (first-->last): 87 1407
8. List (first-->last):
9. List (first-->last): 309
```

```
10. List (first-->last): 490
11. List (first-->last):
12. List (first-->last): 872
13. List (first-->last): 1073
14. List (first-->last): 594 954
15. List (first-->last): 335
16. List (first-->last): 1216
17. List (first-->last): 1057 1357
18. List (first-->last): 938 1818
19. List (first-->last):
```

如果插入更多的项，会看到表变得越来越长，但依然有序。当然也可以删除它们。

后面讨论哈希表效率时，会再回到何时应用链地址法的问题上来。

哈希函数

本节中将会讨论一下设计好的哈希函数所需的因素，并看看是否可以改进本章前面提出的哈希化字符串的方法。

快速的计算

好的哈希函数很简单，所以它能快速计算。哈希表的主要优点是它的速度。如果哈希函数运行缓慢，速度就会降低。哈希函数中有许多乘法和除法是不可取的。(Java 或 C++语言中有位操作，例如，除以 2 的倍数，使得每位都向右移动，这种操作有时很有用。)

哈希函数的目的是得到关键字值的范围，把它用一种方式转化成数组的下标值，这种方法应该使关键字值随机地分布在整个哈希表中。关键字可能完全随机，但也有可能不那么完全随机。

随机关键字

所谓完美的哈希函数把每个关键字都映射到表中不同的位置。只有在关键字组织得异乎寻常的好，且它的范围足够小，可以直接用于数组下标（例如本章开头提到的雇员号码的例子）的时候，这种情况才可能出现。

大多数情况下，这种情况不会发生，哈希函数需要把较大的关键字值范围压缩成较小的数组下标的范围。

在特定的数据库中，关键字值的分布决定哈希函数需要做什么。在本章，假设要数据随机地分布在整个表中。这时，哈希函数

```
index = key % arraySize;
```

是令人满意的。它只包含一个数学操作，如果，关键字真是随机的，得到的下标也是随机的，就会有良好的分布情况。

非随机关键字

然而，数据通常不是随机分布的。假设数据库使用汽车零件号码作为关键字。也许这些号码是这样的形式

033-400-03-94-05-0-535

下面是各个数字的意义：

- 0-2 位：厂商号码（1 到 999，目前最大到 70）
- 3-5 位：分类代号（100，150，200，250，一直到 850）
- 6-7 位：引进的月份（1 到 12）
- 8-9 位：引进的年份（00 到 99）
- 10-11 位：序列号（1 到 99，不可能超过 100）
- 12 位：是否有毒标志（0 或 1）
- 13-15 位：校验和（其他字段的和，对 100 取模）

用作零件号的关键字可能是 0334000394050535。然而，这样的关键字不是随机分布的。从 0 到 9999999999999999 中的大部分数实际不可能出现（例如，大于 70 的供应商号码，不是 50 倍数的分类代号，以及从 13 到 99 的月份）。而且，校验和并不独立于其他数字。对这些零件号码还需要做一些工作，以确保它们形成一个更靠近随机数分布的范围。

不要使用无用数据

压缩关键字字段时，要把每个位都计算在内。例如，分类代码应压缩到 0 到 15 之间。而且，校验和应该舍弃，因为它没有提供任何有用的信息；在压缩中是多余的。各种调整位的技术都可以用来压缩关键字中的不同字段。

使用所有的数据

关键字的每个部分（除了无用数据，如前所述）都应该在哈希函数中有所反映。不要只使用头四位数字，或类似的删除其他位的情况。关键字中提供的数据越多，哈希化后，越可能覆盖整个下标范围。

有时，关键字的范围太大，超出了 int 和 long 类型的规定。马上会讲到哈希化字符串，将可以看到如何控制溢出。

总结：关于哈希函数的窍门是找到既简单又快的哈希函数，而且要去掉关键字中的无用数据，并尽量使用所有的数据。

使用质数作为取模的基数

通常，哈希函数包含对数组容量的取模操作。前面已经看到，使用二次探测和再哈希法时，要求数组容量为质数是多么的重要。然而，如果关键字本身不是随机分布的，不论使用什么哈希化系统，都应该要求数组容量是质数。

这个论述是正确的，因为如果许多关键字共享一个数组容量作为除数，它们会趋向于映射到相同的位置，这会导致聚集。使用质数，可以消除这种可能性。例如，如果在这个汽车零件系统的例子中，表容量是 50 的倍数，那么分类代码全都映射到是 50 倍数的数组下标。然而，如果使用质数，例如 53，可以保证关键字会较平均地映射到数组中。

所以，应该仔细检查关键字，并修正哈希算法，删除任何关键字分布不规则的地方。

哈希化字符串

在本章的开头，已经看到了如何把短小的字符串转换成数字，方法是每个数位乘以对应的一个常数的幂。特别的，看到一个例子，即把四个字母的单词 cats 用下面的公式转化成一个数字

$$key = 3*27^3 + 1*27^2 + 20*27^1 + 19*27^0$$

这个方法有一个令人满意的特性，它包含了字符串的所有可能。计算的值用通常的方法哈希化成数组下标：

```
index = (key) % arraySize;
```

下面是哈希化字符串的 Java 代码：

```
public static int hashFunc1(String key)
   {
   int hashVal = 0;
   int pow27 = 1;                        // 1, 27, 27*27, etc

   for(int j=key.length()-1; j>=0; j--) // right to left
      {
      int letter = key.charAt(j) - 96;  // get char code
      hashVal += pow27 * letter;        // times power of 27
      pow27 *= 27;                      // next power of 27
      }
   return hashVal % arraySize;
   }  // end hashFunc1()
```

循环从单词最右边的字母开始。如果有 N 个字母，就是从第 N-1 个字母单元开始。字母的数字代码放在变量 letter 中；这个代码依据本章开头得到的编码系统（a=1，依此类推）。然后乘以 27 的若干次幂，对于第 N-1 个字母，就是一次幂，对于第 N-2 个字母就是二次幂，依此类推。

hashFunc1()方法不如想像的那么有效。除了字符转换外，在循环中有两次相乘和一次相加。还可以用一种叫做 Horner 方法（Horner 是英国数学家，1773-1827）的数学恒等式取代乘法。这个方法的表述是，如下的表达式

$var4*n^4 + var3*n^3 + var2*n^2 + var1*n^1 + var0*n^0$

可以写成下面的形式

$(((var4*n + var3)*n + var2)*n + var1)*n + var0$

为了对这个等式求值，从最内侧的括号开始，逐渐向外运算。如果把它转化成 Java 代码，就得到下面的方法：

```
public static int hashFunc2(String key)
      {
      int hashVal = key.charAt(0) - 96;
 for(int j=1; j<key.length(); j++)    // left to right
    {
    int letter = key.charAt(j) - 96;  // get char code
    hashVal = hashVal * 27 + letter;  // multiply and add
    }
 return hashVal % arraySize;          // mod
 }  // end hashFunc2()
```

下面从单词最左边的单词开始（有时比从右边开始更自然），这样每次在循环中只有一次乘法

和一次加法（除了从字符串中提取出字符外）。

不幸的是，hashFunc2()方法不能处理大于 7 个字符的字符串。更长的字符串会导致 hashVal 的值超出 int 类型的范围。（如果使用 long 类型，在字符串太长的时候也会出现同样的问题。）

可以修正这种基本的方法避免变量溢出吗？注意，最后得到的关键字总比数组容量小，因为使用了取模操作。那么，这个超大的整数不是最终的数组下标；它只是中间结果。

由此可以得出结论，在 Horner 公式的每一步中，都可以应用取模操作符（%）。这和最后只应用一次取模操作符的结果是一样，但是避免了溢出。（循环中增加一个操作。）下面是修正过的 hashFunc3()方法：

```
public static int hashFunc3(String key)
   {
   int hashVal = 0;
   for(int j=0; j<key.length(); j++)    // left to right
      {
      int letter = key.charAt(j) - 96;  // get char code
      hashVal = (hashVal * 27 + letter) % arraySize; // mod
      }
   return hashVal;                      // no mod
   }  // end hashFunc3()
```

这种方法或类似的方法通常用来哈希化字符串。也可以应用不同的位操作技巧，例如使用 32（或者更大的 2 的幂）作为模基取代 27，这样，乘法可以结合右移操作符（>>）增加效率，右移操作比取模更快。

可以使用这个方法把任意长度的字符串转换成适当的数字。字符串可以是单词、名字或其他字符的组合。

折叠

另一个哈希函数是把关键字的数字分成几组，然后几个组相加。这样做确保了所有数字对哈希值都有贡献。一个组拥有几个数字，是和数组容量相对应的。即，对于有 1000 项的数组，每组包含 3 个数字。

例如，假设要哈希化一个 9 位数的社会安全号码，使用线性探测。如果数组容量是 1000 项，就把 9 位数分成三组。如果某个号码是 123-45-6789，然后按公式 123+456+789=1368，计算关键字。可以使用取模操作符截短这个和，使得最大数组下标是 999。在这里，1368%1000=368。如果数组容量是 100，就需要把 9 位数分成 4 个两位数和 1 个一位数的组：12+34+56+78+9=189，然后 189%100=89。

当数组容量是 10 的倍数时，容易考虑这个过程如何进行。然而，要得到最好的结果，它应该是质数，正如其他的哈希函数一样。这个方案的实现留作习题。

哈希化的效率

已经注意到在哈希表中执行插入和搜索操作可以达到 O(1)的时间级。如果没有发生冲突，只需要使用一次哈希函数和数组的引用，就可以插入一个新数据项或找到一个已存在的数据项。这是最

小的存取时间级。

如果发生冲突，存取时间就依赖后来的探测长度。在一次探测中，每个单元的存取时间要加上寻找一个空白单元（插入时）或一个已存在单元的时间。在一次存取中，要检查这个单元是不是空的，以及（查找或删除时）这个单元是不是包含要寻找的数据项。

因此，一次单独的查找或插入时间与探测的长度成正比。这里还要加上哈希函数的常量时间。

平均探测长度（以及平均存取时间）取决于装填因子（表中项数和表长的比率）。随着装填因子变大，探测长度也越来越长。

下面会看到在不同的哈希表中，探测长度和装填因子之间的关系。

开放地址法

随着装填因子变大，效率下降的情况，在不同开放地址法方案中比链地址法中更严重。

开放地址法中，不成功查找比成功的查找花费更多的时间。在探测序列中，只要找到要查找的数据项，算法就能停止，平均起来，这会发生在探测序列的一半。另一方面，要确定算法不能找到这样的项，就必须走过整个探测序列。

线性探测

下面的等式显示了线性探测时，探测序列（P）和装填因子（L）的关系，对成功的查找来说

$$P = (1 + 1/(1 - L)^2) / 2$$

而对于不成功的查找来说

$$P = (1 + 1/(1 - L)) / 2$$

这些公式来自 Knuth（参考附录 B，“进一步阅读”），把它们推导出来相当复杂。图 11.12 显示了用坐标表示这个等式的结果。

当装填因子为 1/2 时，成功的搜索需要 1.5 次比较，不成功的搜索需要 2.5 次。当装填因子为 2/3 时，分别需要 2.0 次和 5.0 次比较。如果装填因子更大，比较次数会变得非常大。

正如我们看到的那样，应该使装填因子保持在 2/3 以下，最好在 1/2 以下。另一方面，装填因子越低，对于给定数量的数据项，就需要越多的空间。实际情况中，最好的装填因子取决于存储效率和速度之间的平衡。随着装填因子变小，存储效率下降，而速度上升。

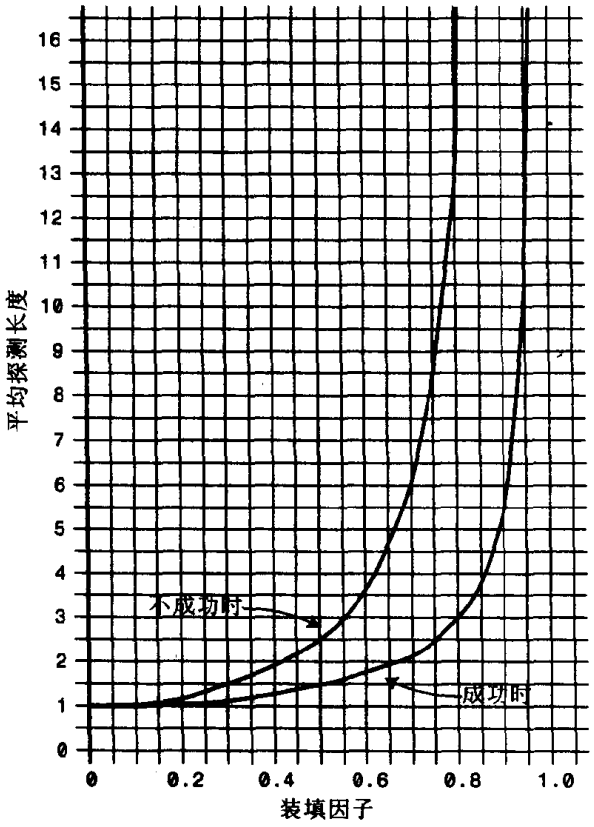


图 11.12 线性探索的性能

二次探测和再哈希法

二次探测和再哈希法的性能相当。它们的性能比线性探测略好。对成功的搜索，公式（仍然来自 Knuth）是

$$-\log_2(1-\text{loadFactor}) / \text{loadFactor}$$

对于不成功的查找，公式是

```
1 / (1-loadFactor)
```

图 11.13 显示了这两个公式的图。当装填因子为 0.5 时，成功和不成功的查找都平均需要两次比较。当装填因子为 2/3 时，分别需要 2.37 次和 3.0 次比较。当装填因子为 0.8 时，分别需要 2.9 和 5.0 次。因此，对于较高的装填因子，对比线性探测，二次探测和再哈希法还是可以忍受的。

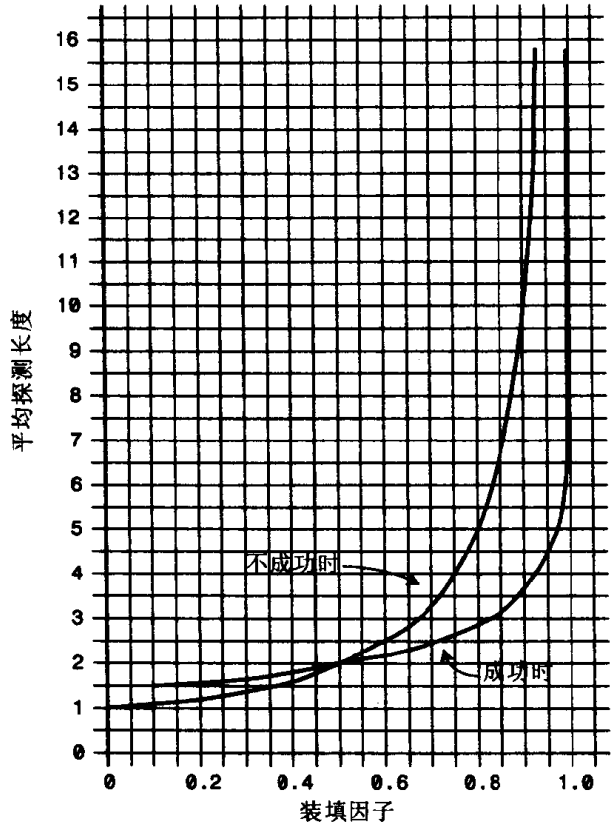


图 11.13 二次探测和再哈希法的性能

链地址法

链地址法的效率分析有些不同，一般来说比开放地址法简单。

我们想知道要查找或插入一个新项需要多少时间。假设这个操作的最耗费时间的部分是比较待查找项和表中其他项的关键字。同时假设，哈希函数执行的时间和判断是否到达表尾的时间与一次关键字比较的时间相等。因此，所有的操作需要 1+nComps 的时间，其中，nComps 是关键字比较次数。

假设哈希表包含 arraySize 个数据项，每个数据项有一个链表，在表中有 N 个数据项。那么，平均起来每个链表有 N 除 arraySize 个数据项：

```
AverageListLength = N / arraySize
```

这和装填因子的定义是相同的：

```
loadFactor = N / arraySize
```

所以平均表长等于装填因子。

查找

在成功的查找中，算法映射到正确的链表，然后在链表中查找数据项。平均起来，找到正确项之前，要检查一半的数据项。因此，查找时间是

```
1 + loadFactor / 2
```

不管链表是否有序，都遵循这个公式。在不成功的查找中，如果链表不是有序的，要检查所有的数据项，所以时间是

```
1 + loadFactor
```

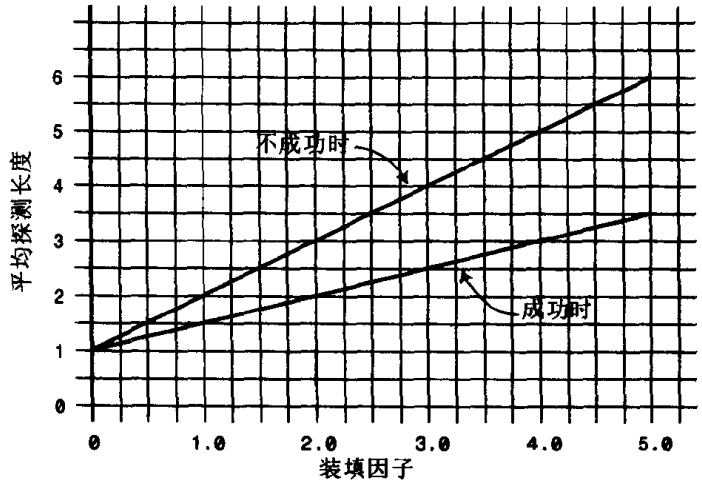


图 11.14 链地址法的性能

对于有序表，在不成功查找中只需要检查一半的数据项，所以时间与成功查找的时间相同。

在链地址法中，通常装填因子为 1（数据项的个数和数组容量相同）。较小的装填因子不能显著地提升性能，但是，所有操作的时间都会随着装填因子的变大而增长，所以，不宜把装填因子提升到 2。

插入

如果链表是无序的，插入操作是立即完成的，这里不需要比较。哈希函数必须计算，所以插入的时间是 1。

如果链表有序，那么，由于存在不成功的查找，平均要检查一半的数据项，所以插入的时间是

```
1 + loadFactor / 2
```

开放地址法和链地址法的比较

如果使用开放地址法，对于小型的哈希表，再哈希法似乎比二次探测的效果好。有一个情况例外，就是内存充足，并且哈希表一经创建，就不再改变其容量；在这种情况下，线性探测相对容易实现，并且，如果装填因子低于 0.5，几乎没有什么性能下降。

如果在哈希表创建时，要填入的项数未知，链地址法要好过开放地址法。如果用开放地址法，随着装填因子变大，性能会下降很快，但是用链地址法，性能只能线性地下降。

当两者都可选时，使用链地址法。它需要使用链表类，但回报是增加比预期更多的数据时，不会导致性能的快速下降。

哈希化和外部存储

在第 10 章的最后，讨论了使用 B-树作为外部（基于磁盘）存储的数据结构。下面简要看一下哈希表用作外部存储的情况。

回忆一下第 10 章，磁盘由许多包含文件的“块”组成，存取一个块的时间比任何在内存中存取数据的时间都要长得多。因此，在设计外部存储策略时，不考虑最小化存取块的数量。

另一方面，外部存储器单位容量相当便宜。所以，可以使用更大数量的存储器，超过要存储的项数，如果这样的话，会加快存取时间。这就有可能使用哈希表。

文件指针表

外部哈希化的关键部分是一个哈希表，它包含块成员，指向外部存储器中的块。哈希表有时叫做索引（就像书的索引）。它可以存储在内存中，如果它太大，也可以存储在磁盘上，而把它的一部分放在内存中。即使把哈希表都放在内存中，也要在磁盘中保存一个备份，文件打开时，把它读入内存。

未填满块

这里还要使用一下第 10 章的例子。在这个例子中，块大小为 8192 字节，一个记录为 512 字节。因此，一个块可容纳 16 个记录。哈希表的每个条目指向某个块。假设一个特殊的文件有 100 个块。

内存中的索引（哈希表）记录了文件块的指针，第一个块为 0，一直到 99。

在外部哈希化中，重要的是块不要填满。因此，每个块平均存储 8 个记录。有的块多些，有的少些。在文件中大概有 800 个记录。排列如图 11.15 所示。

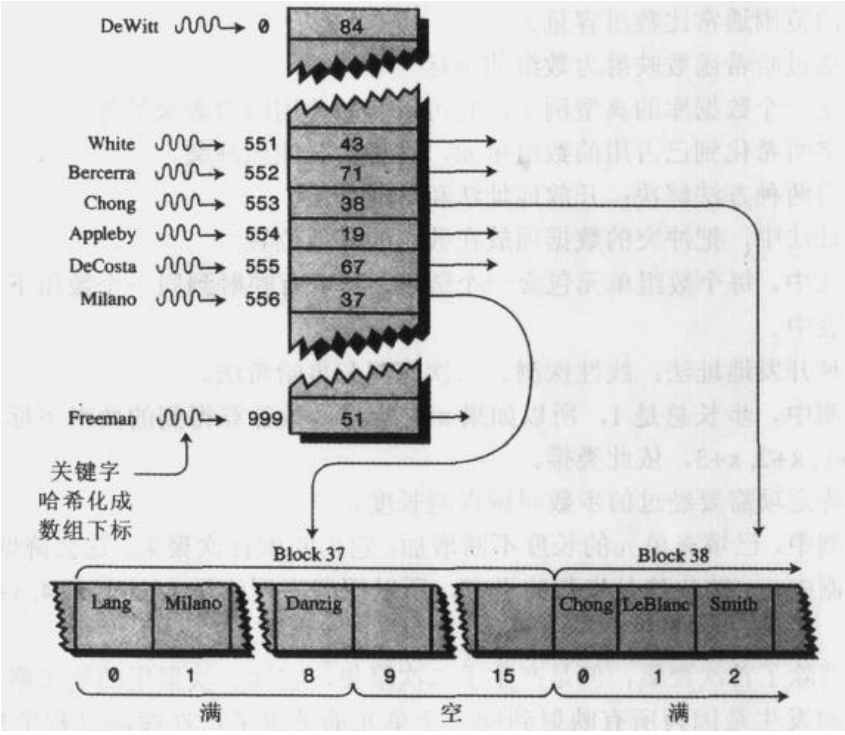


图 11.15 外部哈希化

所有关键字映射为同一个值的记录都定位到相同块。为找到特定关键字的一个记录，搜索算法哈希化关键字，用哈希值作为哈希表的下标，得到某个下标中的块号，然后读取这个块。

这个过程是有效的，因为定位一个特定的数据项，只需要访问一次块。缺点是相当多的磁盘空间被浪费了，因为设计时规定，块是不允许填满的。

为了实现这个方案，必须仔细选择哈希函数和哈希表的大小，为的是限制映射到相同的值关键字的数量。在本例中，平均每个值只对应 8 个记录。

填满的块

即使用一个好的哈希函数，块偶尔也会填满。这时，可以使用在内部哈希表中讨论的处理冲突的不同方法：开放地址法和链地址法。

在开放地址法中，插入时，如果发现一个块是满的，算法在相邻的块插入新记录。在线性探测中，这是下一个块，但也可以用二次探测或再哈希法选择。在链地址法中，有一个溢出块；当发现块已满时，新记录插在溢出块中。

填满的块是不合需要的，因为有了它就需要额外的磁盘访问，这就需要两倍的访问时间。然而，如果这种情况不经常发生，也可以接受。

这里只讨论了用于外部存储的最简单的哈希表实现。还有很多更复杂的方法，已经超出了本书的范围。

小 结

- 哈希表基于数组。
- 关键字值的范围通常比数组容量大。
- 关键字值通过哈希函数映射为数组的下标。
- 英文字典是一个数据库的典型例子，它可以有效的用哈希表来处理。
- 一个关键字哈希化到已占用的数组单元，这种情况叫做冲突。
- 冲突可以用两种方法解决：开放地址法和链地址法。
- 在开放地址法中，把冲突的数据项放在数组的其他位置。
- 在链地址法中，每个数组单元包含一个链表。把所有映射到同一个数组下标的数据项都插在这个链表中。
- 讨论了三种开发地址法：线性探测、二次探测和再哈希法。
- 在线性探测中，步长总是 1，所以如果 x 是哈希函数计算得到的数组下标，那么探测序列就是 x, x+1, x+2, x+3，依此类推。
- 找到一个特定项需要经过的步数叫做探测长度。
- 在线性探测中，已填充单元的长度不断增加。它们叫做首次聚集，这会降低哈希表的性能。
- 在二次探测中，x 的位移是步数的平方，所以探测序列就是 x, x+1, x+4, x+9，x+16，依此类推。
- 二次探测消除了首次聚集，但是产生了二次聚集，它比首次聚集的危害略小。
- 二次聚集的发生是因为所有映射到同一个单元的关键字，在探测过程中执行了相同的序

列。

- 发生上述情况是因为步长只依赖于哈希值，与关键字无关。
- 在再哈希法中，步长依赖于关键字，且从第二个哈希函数中得到。
- 在再哈希法中，如果第二个哈希函数返回一个值 s，那么探测序列就是 x, x+s, x+2s, x+3s, x+4s，依此类推，这里 s 由关键字得到，但探测过程中保持常量。
- 装填因子是表中数据项数和数组容量的比值。
- 开放地址法中的最大装填因子应该在 0.5 附近，若具有相同的装填因子，对于再哈希法来说，查找的平均探测长度是 2。
- 在开放地址法中，当装填因子接近 1 时，查找时间趋于无限。
- 在开放地址法中，关键是哈希表不能填得太满。
- 对于链地址法，装填因子为 1 比较合适。
- 这时，成功的探测长度平均是 1.5，不成功的是 2.0。
- 链地址法的探测长度随着装填因子变大而线性增长。
- 字符串可以这样哈希化，每个字符乘以常数的不同次幂，求和，然后用取模操作符（%）缩减结果，以适应哈希表的容量。
- 如果在 Horner 方法中用多项式表达哈希化，每一步中都应用取模操作符，以免发生溢出。
- 哈希表的容量通常是一个质数。这在二次探测和再哈希法中非常重要。
- 哈希表可用于外部存储，一种做法是用哈希表的单元存储磁盘文件的块号码。

问　题

下列问题作为读者的自测题。答案可见附录 C。

1. 使用大 O 表示法，说明在哈希表中查找一个数据项需要多少时间（理想状况）。
2. _____把关键字值的范围转换成数组下标的范围。
3. 开发地址法
   a. 保证数组中有大量空白单元。
   b. 可以随意使用那些地址。
   c. 沿着单元 x+1, x+2 一直往下探测，直到找到一个空位。
   d. 当找到的单元被占用时，在数组中寻找另一个单元。
4. 如果探测不成功，就使用下一个可用单元，这种探测方法叫做_____。
5. 二次探测中，前五步的步长是什么？
6. 二次聚集会发生，是因为
   a. 许多关键字映射到相同的位置。
   b. 探测序列总是相同的。
   c. 插入了太多有相同关键字的项。
   d. 哈希函数不完美。
7. 链地址法在每个单元中使用了_____。
8. 链地址法中，合理的装填因子是_____。

9. 判断题：字符串的一个可能的哈希函数是每个字符乘以递增的幂。
10. 当填入的数据项数未知时，最好的方法是
    a. 线性探测。
    b. 二次探测。
    c. 再哈希法。
    d. 链地址法。
11. 如果哈希函数用数字折叠的方法，每组包含几个数字应该根据_____。
12. 判断题：线性探测中，不成功查找比成功的查找更耗时。
13. 在链地址法中，插入新数据项的时间
    a. 随装填因子线性增长。
    b. 和表中项数成正比。
    c. 和链表数目成正比。
    d. 和数组已占用单元的百分比成正比。
14. 判断题：在外部哈希化中，重要的是块中不能填满记录。
15. 在外部哈希化中，所有关键字映射到相同的值的记录被放在_____。

实　验

完成这些实验可以帮助深入理解本章的主题。不需要编程实现。

1. 在线性探测中，不成功查找的时间和聚集的规模有关。使用 Hash 专题 applet，创建一个有 60 个单元的哈希表，并填入 30 个数据项，即装填因子为 0.5，找出它的平均聚集规模。认为一个孤立的单元（即，左右都是空白单元）聚集规模为 1。为了找到这个平均值，应该计算每个聚集包含的项数，然后除以聚集的个数，但是有一个更简单的方法。是什么呢？重复 6 次这个实验，每次填入 30 个数据项，并求平均聚集规模。令装填因子分别为 0.6，0.7，0.8 和 0.9，重复整个过程，可以得到如图 11.12 所示的结果吗？

2. 用 HashDouble 专题 applet，创建一个小的二次探测哈希表，它的容量不是一个质数，比如 24。把它填得非常满，比如 16 项。现在查找一个不存在的关键字值。实验不同的值，直到发现有一个可以造成二次探测陷入无限循环的值。这种情况之所以会发生，是因为探测步长（对不是质数的数组容量取模）形成了不断重复的序列。所以要记住，数组容量一定要是质数。

3. 用 HashChain applet，创建一个有 25 个单元的数组，然后，填入 50 个数据项，这时装填因子为 2.0。检查显示的链表。把所有链表长度相加，然后除以链表个数，就得到平均链表长度。在不成功查找中，平均起来，就需要搜索这个长度。（实际上，有更快的方法找到这个长度，是什么？）

编程作业

编程作业有助于巩固对本章内容的理解，并展示如何应用本章的概念。（在“简介”中提到过，资深教师可以从出版者的网站上得到编程作业的完整答案。）

11.1　修改 hash.java 程序（清单 11.1），改用二次探测。

11.2 实现用一个线性探测哈希表存储字符串。需要一个把字符串转换成数组下标的哈希函数：参考本章“哈希化字符串”一节。假设字符串全是小写字母，所以 26 个字符就足够了。

11.3 写一个哈希函数，实现数字折叠的方法（正如本章“哈希函数”一节描述的）。程序应该可以适应任何数组容量和任何关键字长度。使用线性探测。存取某个数中的一组数字比想像的要简单。如果数组容量不是 10 的倍数，有影响吗？

11.4 为 hash.java 程序写一个 rehash()方法。只要装填因子大于 0.5，insert()方法会调用它，把整个哈希表复制到比它大两倍的数组中，新的数组容量应该是一个质数。参考本章“扩展数组”一节。不要忘记，必须处理已经“删除”的数据项，那里的值为-1。

11.5 使用二叉搜索树，而不使用链地址法中的链表解决冲突。即创建一个树的数组作为哈希表。可以使用 hahsChain.java 程序（清单 11.3）和第 8 章 tree.java 程序（清单 8.1）中的 Tree 类作为基础。为了显示这个基于树的哈希表，每棵树使用中序遍历。

树比链表好的地方是搜索只需要 O(logN)的数量级，而不是 O(N)。如果装填因子很大，省下的这部分时间是很可观的。在链表中，检查 15 个数据项最少用 5 次比较，而在树中只用 4 次。

重复项在树和哈希表中都会出现问题，所以要增加一些代码，防止在哈希表中插入重复项。（小心：Tree 类中的 find()方法假设是一棵非空树。）为了缩短这个程序的清单，不用考虑删除，对树来说，删除需要大量的代码。

第 12 章 堆

本章重点

- 堆的介绍
- 堆的 Java 代码
- 基于树的堆
- 堆排序

第 4 章中介绍了优先级队列，它是一种对最小（或最大）关键字的数据项提供便利访问的数据结构。

优先级队列可以用于计算机中的任务调度，在计算机中某些程序和活动需要比其他的程序和活动先执行，因此要给它们分配更高的优先级。

另一个例子是在武器系统中，比如在一个海军巡洋舰系统中。会探测到许多对巡洋舰的威胁，如来自飞机、导弹、潜水艇等的攻击，需要为这些威胁分优先级。例如，离巡洋舰距离近的导弹就比距离远的飞机的优先级更高，这样对应的防范系统（例如地对空导弹）会首先对付它。

优先级队列也可以用在其他计算机算法的内部。在第 14 章中，可以看到图算法中应用了优先级队列，例如在 Dijkstra 算法中。

优先级队列是一个抽象数据类型（ADT），它提供了删除最大（或最小）关键字值的数据项的方法，插入数据项的方法，有时还有一些其他操作的方法。配合不同的 ADT，优先级队列可以用不同的内部结构来实现。第 4 章中可以看到，优先级队列是用有序数组来实现的。这种作法的问题是，尽管删除最大数据项的时间复杂度为 O(1)，但是插入还是需要较长的 O(N)时间。这是因为必须移动数组中平均一半的数据项以插入新数据项，并在完成插入后，数组依然有序。

本章将介绍实现优先级队列的另一种结构：堆。堆是一种树，由它实现的优先级队列的插入和删除的时间复杂度都是 O(logN)。尽管这样删除的时间变慢了一些，但是插入的时间快得多了。当速度非常重要，且有很多插入操作时，可以选择堆来实现优先级队列。

注意

这里的“堆”是指一种特殊的二叉树，不要和 Java 和 C++等编程语言里的“堆”混淆，后者指的是程序员用 new 能得到的计算机内存的可用部分。

堆的介绍

堆是有如下特点的二叉树：

- 它是完全二叉树。这也就是说，除了树的最后一层节点不需要是满的，其他的每一层从左到右都完全是满的。图 12.1 显示了完全二叉树和非完全二叉树。
- 它常常用一个数组实现。第 8 章中介绍了如何用数组而不是由引用连接起来的各个节点来存储二叉树。
- 堆中的每一个节点都满足堆的条件，也就是说每一个节点的关键字都大于（或等于）这个

节点的子节点的关键字。

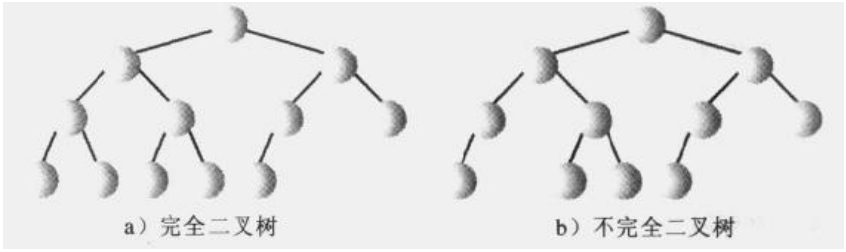


图 12.1　完全二叉树和非完全二叉树

图 12.2 显示了堆与实现它的数组之间的关系。堆在存储器中的表示是数组；堆只是一个概念上的表示。注意树是完全的二叉树，并且所有的节点都满足堆的条件。

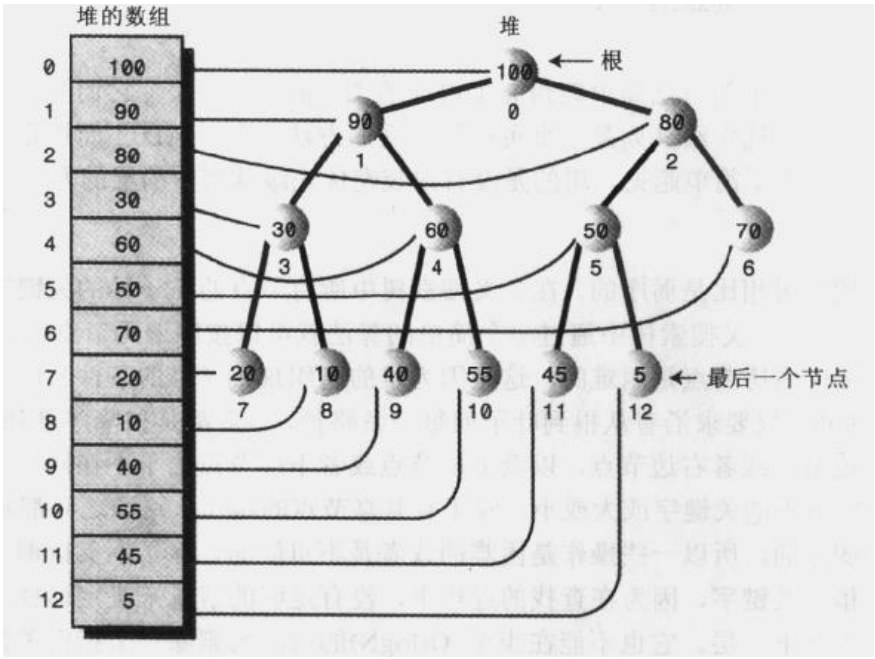


图 12.2　堆和它的实现数组

堆是完全二叉树的事实说明了表示堆的数组中没有“洞”。从下标 0 到 N-1，每一个数据单元都有数据项。（图 12.2 中 N 等于 13。）

本章中假设最大的（而不是最小的）关键字在根节点上。基于这种堆的优先级队列是降序的优先级队列。（在第 4 章中讨论了升序优先级队列。）

优先级队列、堆和抽象数据类型（ADT）

本章讨论堆，堆主要用于实现优先级队列，优先级队列和实现它的堆之间有非常紧密的关系。下面简化的代码表明了这种关系：

```
class Heap
   {
   private Node heapArray[];
```

```
    public void insert(Node nd)
       {  }
    public Node remove()
       {  }
    }
class priorityQueue
    {
    private Heap theHeap;

    public void insert(Node nd)
       { theHeap.insert(nd); }
    public Node remove()
       ( return theHeap.remove() }
    }
```

priorityQueue 类中的方法简单地封装了内部的 Heap 类的方法，它们的功能相同。这个例子在概念上清楚地表明了优先级队列是一个可以用不同的方法实现的 ADT，而堆是一种更为基础的数据结构。在本章中，为了简单起见，用的是没有封装在优先级队列中的堆的方法。

弱序

堆和二叉搜索树相比是弱序的，在二叉搜索树中所有节点的左子孙的关键字都小于右子孙的关键字。这说明在一个二叉搜索树中通过一个简单的算法就可以按序遍历节点，正如以前看到的那样。

在堆中，按序遍历节点是困难的，这是因为堆的组织规则（堆的条件）比二叉搜索树的组织规则弱。对于堆来说，只要求沿着从根到叶子的每一条路径，节点都是按降序排列的。如图 12.2 所示，指定节点的左边节点或者右边节点，以及上层节点或者下层节点由于不在同一条路径上，它们的关键字可能比指定节点的关键字或大或小。除了有共享节点的路径，路径之间都是相互独立的。

由于堆是弱序的，所以一些操作是困难的或者是不可能的。除了不支持遍历以外，也不能在堆上便利地查找指定关键字，因为在查找的过程中，没有足够的信息来决定选择通过节点的两个子节点中的哪一个走向下一层。它也不能在少至 O(logN)的时间内删除一个指定关键字的节点，因为没有办法能找到这个节点。（这些操作可以通过按顺序查找数组的每一个单元来执行，但是只能以较慢的 O(N)时间执行。）

因此，堆的这种组织似乎非常接近无序。不过，对于快速移除最大节点的操作以及快速插入新节点的操作，这种顺序已经足够了。这些操作是使用堆作为优先级队列时所需的全部操作。后面将主要讨论这些操作是如何完成的，然后再在专题 applet 中实际观察这些操作。

移除

移除是指删除关键字最大的节点。这个节点总是根节点，所以移除是很容易的。根在堆数组中的索引总是 0：

```
maxNode = heapArray[0];
```

问题是一旦移除了根节点，树就不再是完全的了；数组里就有了一个空的数据单元。这个“洞”必须要填上，可以把数组中所有数据项都向前移动一个单元，但是还有快得多的方法。下面就是移除最大节点的步骤：

1. 移走根。

2. 把最后一个节点移动到根的位置。

3. 一直向下筛选这个节点，直到它在一个大于它的节点之下，小于它的节点之上为止。

最后一个节点（last）是树最低层的最右端的数据项，它对应于数组中最后一个填入的数据单元。（在图 12.2 中，可以看到最后一个节点的下标为 12，值为 5。）把这个节点直接复制到根节点处：

```
heapArray[0] = heapArray[N-1];
N--;
```

移除了根，使数组容量的大小减一。

向上或向下“筛选”（也可以用“冒泡”或者“渗滤”）一个节点是指沿着一条路径一步一步地移动此节点，和它前面的节点交换位置，每一步都检查它是否处在了合适的位置。在步骤 3 中，在根上的节点对根来说太小了，所以要把它筛选到堆的合适的位置。后面将看到它的代码。

步骤 2 恢复了堆的完全性的特征（没有洞），而步骤 3 恢复了堆的条件（每个节点都大于它的子节点）。移除过程如图 12.3 所示。

图中 a）部分把最后一个节点（30）复制到根，根节点已经移除了。b）、c）和 d）中这个节点被向下筛选到合适的位置，它的合适的位置在最底层。（并不总是这种情况；筛选的过程也可能在中间某层停止。）图中 e）显示的就是节点在正确位置时的情景。

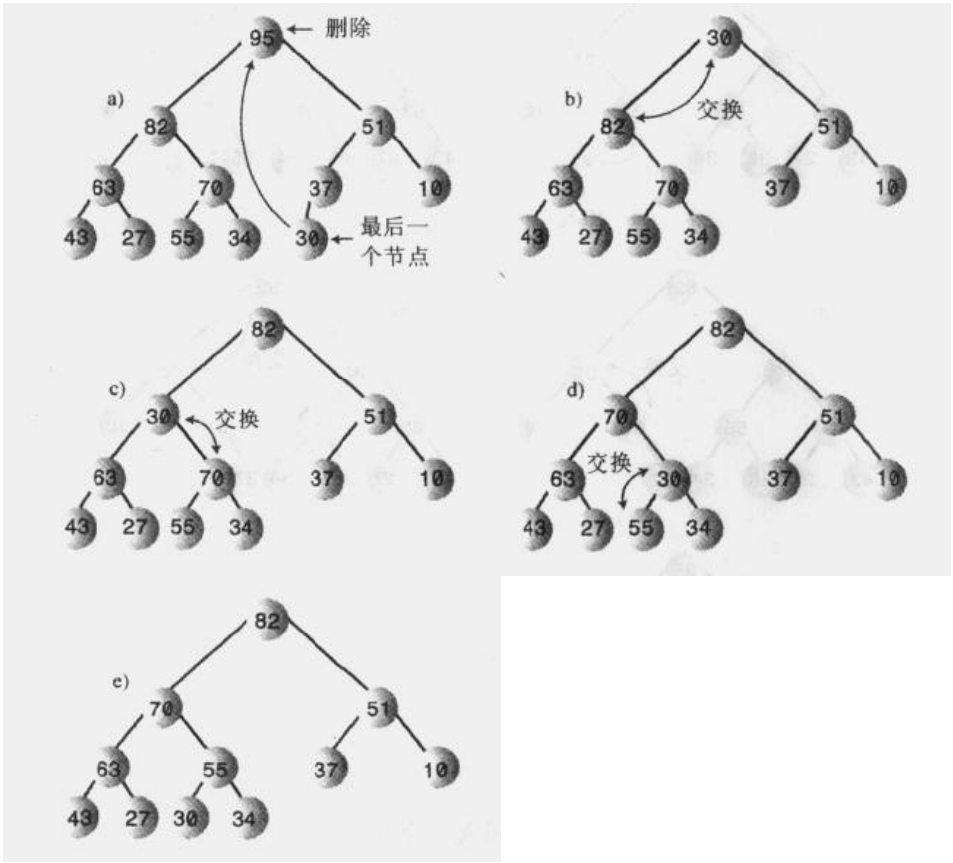


图 12.3 移除最大的节点

在被筛目标节点的每个暂时停留的位置上，向下筛选的算法都要检查哪一个子节点更大。然后目标节点和较大的子节点交换位置。如果要把目标节点和较小的子节点换位，那么这个子节点就会变成大子节点的父节点，这就违背了堆的条件。图 12.4 显示了正确的和不正确的交换情况。

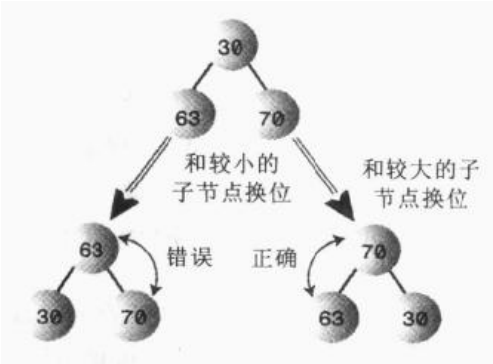


图 12.4 哪个子节点该交换？

插入

插入节点也是很容易的。插入使用向上筛选，而不是向下筛选。节点初始时插入到数组最后第一个空着的单元中，数组容量大小增一。

```
heapArray[N] = newNode;
N++;
```

问题是这样可能会破坏堆的条件。如果新插入的节点大于它新得到的父节点时，就会发生这种情况。因为父节点在堆的底端，它可能很小，所以新节点就显得比较大。因此，需要向上筛选新节点，直到它在一个大于它的节点之下，在一个小于它的节点之上。插入过程如图 12.5 所示。

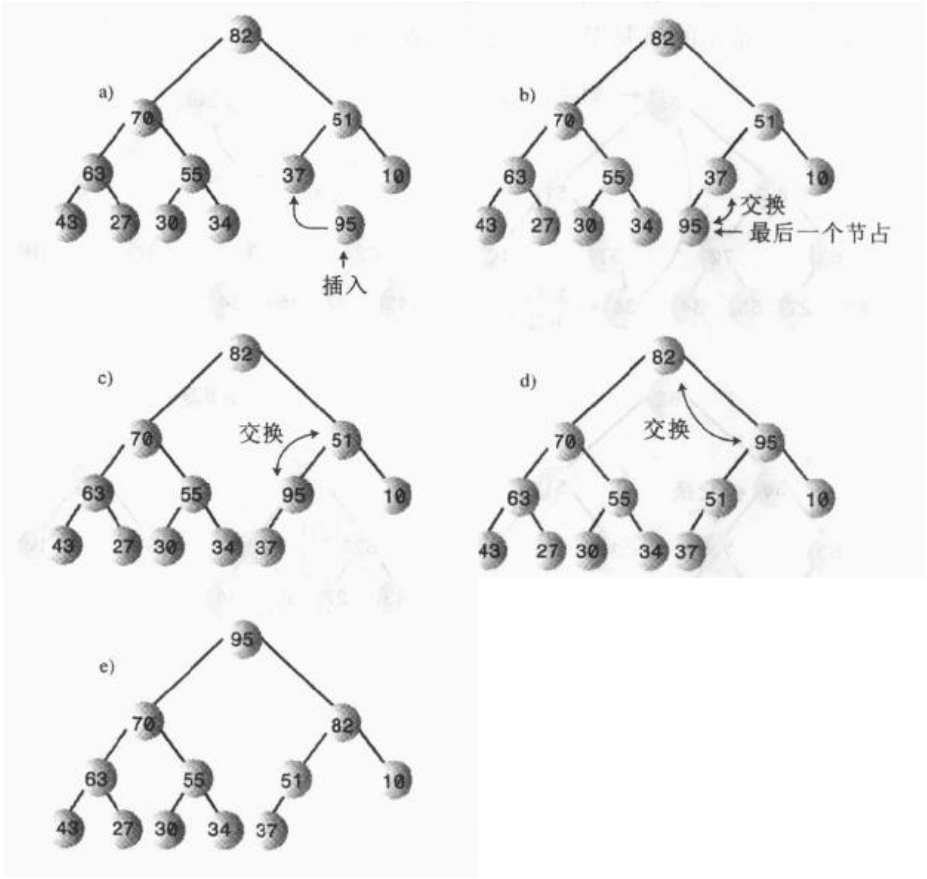


图 12.5 插入一个节点

向上筛选的算法比向下筛选的算法相对简单，因为它不用比较两个子节点的关键字大小。节点只有一个父节点，目标节点只要和它的父节点换位即可。在图中，新插入节点的最后位置恰好在根

上．但是新节点最后也可能落在中间层上。

比较图 12.4 和图 12.5，发现如果先移除一个节点再插入相同的一个节点，结果并不一定是恢复为原来的堆。一组给定的节点可以组成很多合法的堆，这取决于节点插入的顺序。

不是真的交换

图 12.4 和 12.5 显示了向下筛选和向上筛选的过程中节点交换位置的情况。换位是在概念上理解插入和删除最简单的方法，并且实际上某些堆的实现也确实使用了换位。图 12.6a 显示了在向下筛选的过程中使用换位的简化版本。在三次换位之后，节点 A 将停在 D 的位置上，并且节点 B、C 和 D 都会向上移动一层。

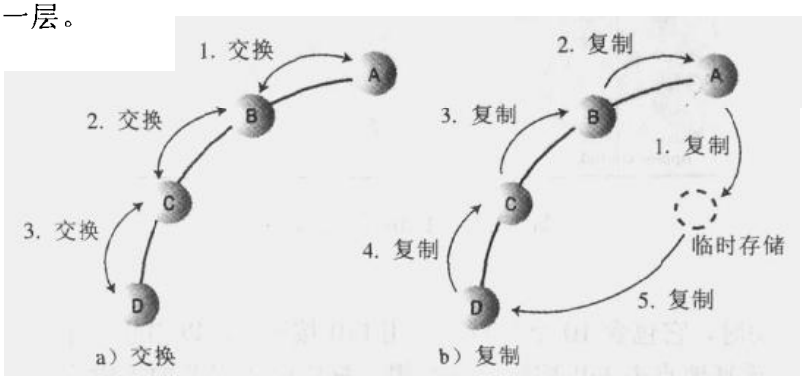


图 12.6 筛选过程中的交换和复制

但是，一次交换需要三次复制，因此在如图 12.6a 所示的三次交换中就需要九次复制。通过在筛选的算法中使用复制取代交换，可以减少所需的复制总数。

图 12.6b 显示了如何用五次复制做三次换位。首先，暂时保存节点 A；然后 B 覆盖 A，C 覆盖 B，D 覆盖 C；最后，再从临时存储中取出 A 复制到以前 D 的位置上。这样就把复制的次数从九次减少到了五次。

在图中节点 A 移动了三层。当层数增加时，复制节省的时间更多，因为从临时存储区复制以及复制到临时存储区的两次复制在复制的总数中所占的比例更少了。对于很多层的堆，节省的复制次数接近三的倍数。

另一种观察用复制执行向上筛选和向下筛选过程的方法是用“洞”的思想（空节点的数组元素单元），在向上筛选时洞向下移动，向下筛选时洞向上移动。例如，图 12.6b 中把 A 复制到 Temp 中，在 A 点创建一个“洞”。“洞”实际上保留着被移走节点原来的值，它还在那，但这没关系。把 B 复制到 A，“洞”由 A 移动到 B，和节点位置的变化方向相反。“洞”一步一步地向下移动。

Heap 专题 applet

Heap 专题 applet 演示了上面所讨论的操作：它允许在堆中插入新节点以及移除最大的节点。除此之外，还可以改变指定数据项的优先级。

当启动 Heap 专题 applet 时，可以看到如图 12.7 显示的界面。

这里有四个按钮：Fill、Chng、Rem 和 Ins，分别对应添加、改变、移除和插入操作。下面就来看看它们是怎么工作的。

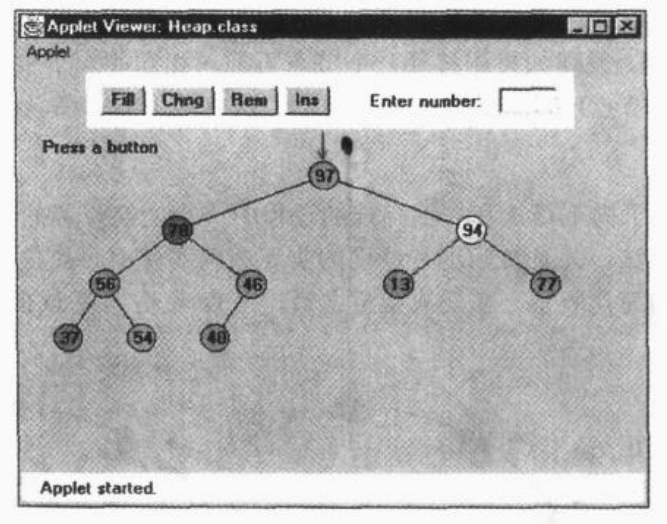


图 12.7 Heap 专题 applet

Fill 按钮

当 applet 开始启动时，它包含 10 个节点。使用 Fill 按钮，可以创建一个新的堆，节点值可以是 1 到 31 中的任意值。重复地点击 Fill 按钮，并在提示框中输入想要的关键字值。

Change 按钮

可以改变一个已存在节点的优先级。这个过程在很多情况下都很有用。例如，在前面的巡洋舰的例子中，一个正在接近的飞机可能会改变航向飞离航空母舰；它的优先级在这种新的情况下应该变低，尽管飞机还会保留在优先级队列中，直到它出了雷达感应的范围。

为了改变一个节点优先级，重复点击 Chng 按钮。当出现提示时，用鼠标点击该节点。红色箭头将指向它。然后，在提示框中键入节点的新的优先级。

如果节点的优先级上升了，节点会向上移动到新位置上。如果优先级降低，节点会向下筛选。

Remove 按钮

重复点击 Rem 按钮会移除关键字最大的节点，它在根的位置上。可以看到该节点消失了，然后被最底层的最后（最右端）的一个节点取代了。这个节点一直向下筛选直到它到达能恢复堆顺序的位置为止。

Insert 按钮

节点初始插入时总是插入到第一个可用的数组单元中，也就是堆的最底层的最后一个节点的右侧。节点从这个位置开始向上筛选直到合适的位置。重复点击 Ins 按钮执行这个操作。

堆的 Java 代码

这一节的后面将显示完整的 heap.java 代码。在列出程序清单之前，先来分别介绍一下插入、移除和改变优先级操作的代码。

下面是第 8 章中关于用数组表示一棵树的一些要点。若数组中节点的索引为 x，则

- 它的父节点的下标为(x-1)/2。
- 它的左子节点的下标为 2*x + 1。
- 它的右子节点的下标为 2*x + 2。

图 12.2 显示了这种关系。

注意

记住“/”这个符号，应用于整数的算式时，它执行整除，且得到的结果是下取整的值。

插入

向上筛选的算法放在它自己的方法中。insert()方法直接调用这个 trickleUp()方法：

```
public boolean insert(int key)
   {
   if(currentSize==maxSize)            // if array is full,
      return false;                    //    failure
   Node newNode = new Node(key);       // make a new node
   heapArray[currentSize] = newNode;   // put it at the end
   trickleUp(currentSize++);           // trickle it up
   return true;                        // success
   }  // end insert()
```

这里需要先检查以确定数组不满，然后用参数传递的关键字值创建一个新的节点。把这个节点插入到数组的末端。最后，调用 trickleUp()例程把这个节点向上移动到适当的位置。

trickleUp()方法（将在后面介绍）的参数是新插入的数据项的下标。找到这个位置的父节点，然后把这个节点保存在一个名为 bottom 的变量里。在 while 循环内部，变量 index 沿着路径朝根的方向上行，依次指向每一个节点。只要没有到达根（index>0），且 index 的父节点的关键字（iData）小于这个新节点，while 循环就一直执行。

while 循环体执行向上筛选过程中的一个步骤。首先把父节点复制到 index 所指的单元中，即向下移动了父节点。（这就使“洞”上移了。）然后通过把父节点的下标传给 index，使得 index 上移了，并且还把父节点的下标给予它的父节点。

```
public void trickleUp(int index)
   {
   int parent = (index-1) / 2;
   Node bottom = heapArray[index];

   while( index > 0 &&
         heapArray[parent].getKey() < bottom.getKey() )
      {
      heapArray[index] = heapArray[parent]; // move node down
      index = parent;                       // move index up
      parent = (parent-1) / 2;        // parent <- its parent
      }  // end while
   heapArray[index] = bottom;
   }  // end trickleUp()
```

最后，当退出循环时，临时存储在 bottom 里的要插入的新节点，插入到 index 所指示的单元中。这就是第一个它不大于它的父节点的位置，因此在这里插入它才满足堆的条件。

移除

如果在移除的例程中包含了向下筛选的算法，移除算法也不复杂。保存根节点，把最后一个节点（下标为 currentSize-1）放到根的位置上，然后调用 trickleDown()，把这个节点放到适当的位置。

```
public Node remove()           // delete item with max key
   {                           // (assumes non-empty list)
   Node root = heapArray[0];   // save the root
   heapArray[0] = heapArray[--currentSize];  // root <- last
   trickleDown(0);             // trickle down the root
   return root;                // return removed node
   }  // end remove()
```

这个方法返回被移除的节点；堆的用户常常用其他方法处理这个节点。

trickleDown()例程比 trickleUp()复杂，因为在 trickleDown()中必须判断哪个子节点更大。首先，把下标为 index 的节点保存到一个变量 top 中。如果是 romove()调用 trickleDown()，index 就是指根，不过，正如将要看到的一样，其他例程也可以调用它。

只要 index 没有到最底层——这就是说，只要它还有至少一个子节点，while 循环就一直运行。在循环内部，检查是否有右子节点（可能只有一个左子节点），如果有，比较两个子节点的关键字，给 largerChild 设置合适的值。

然后检查原来节点（现在在根位置上的节点）的关键字是否大于 largerChild；如果是，向下筛选的过程就完成了，并且退出循环。

```
public void trickleDown(int index)
   {
   int largerChild;
   Node top = heapArray[index];       // save root
   while(index < currentSize/2)       // while node has at
      {                               //    least one child,
      int leftChild = 2*index+1;
      int rightChild = leftChild+1;
                                      // find larger child
   if( rightChild < currentSize &&  // (rightChild exists?)
                     heapArray[leftChild].getKey() <
                     heapArray[rightChild].getKey() )
      largerChild = rightChild;
   else
      largerChild = leftChild;
                                   // top >= largerChild?
   if(top.getKey() >= heapArray[largerChild].getKey())
      break;
                                   // shift child up
   heapArray[index] = heapArray[largerChild];
```

```
    index = largerChild;            // go down
    } // end while
heapArray[index] = top;             // index <- root
} // end trickleDown()
```

在退出循环时，只需要把存储在 top 中的节点恢复到适当的位置上，即 index 所指的位置。

关键字更改

在有了 trickleDown()和 trickleUp()方法之后，很容易实现改变节点的优先级（关键字）算法，先更改节点关键字的值，然后再把节点向上或者向下移动到适当的位置。change()方法实现了上述操作：

```
public boolean change(int index, int newValue)
      {
      if(index<0 || index>=currentSize)
         return false;
      int oldValue = heapArray[index].getKey(); // remember old
      heapArray[index].setKey(newValue);  // change to new

      if(oldValue < newValue)             // if raised,
         trickleUp(index);                // trickle it up
      else                                // if lowered,
         trickleDown(index);              // trickle it down
      return true;
      } // end change()
```

例程首先检查给定的第一个参数 index 是否合法，如果它是合法的，改变这个下标所指节点的 iData 字段，赋值为第二个参数的值。

然后，如果节点优先级提高了，就向上筛选节点；如果优先级下降了，就向下筛选节点。

实际上，改变节点的优先级的困难部分没有在这个例程中显示出来，困难是要查找更改优先级的节点。在刚刚显示的 change()方法中，提供下标作为一个参数，而且在 Heap 专题 applet 中，用户只要通过点击节点作选择即可。在实际的应用中，则需要有查找适当节点的机制；正如前面已经讲过的，在堆中通常易于访问的节点只有关键字最大的那个节点。

通过顺序地扫描数组可以在线性的时间 O(N)内解决查找指定节点的这个问题。或者，当节点在优先级队列中移动的时候，用新的下标值去更新另外一个数据结构（可能是哈希表），这样可以快速地访问任何节点。当然，维护另一个数据结构更新的本身也是耗时的。

数组的大小

应该注意，数组的大小，也就是记录堆中节点的数目，是 Heap 类中关于堆状态的一个很重要的信息，也是 Heap 类中关键的字段。从最后一个位置复制的节点并不清除，所以对于算法来说，判断最后一个堆占用单元位置的惟一方法是根据数组当前的大小求得。

heap.java 程序

heap.java 程序，如清单 12.1 所示，使用了只有一个 iData 字段的 Node 类，用 iData 变量保存节

点的关键字。通常，在实际应用的程序中，这个类可以有许多其他的字段。Heap 类中包括前面所讨论的方法，并增加了 isEmpty()和 displayHeap()方法，displayHeap()用字符形式显示一个堆，尽管很粗略，但是易于理解。

清单 12.1 heap.java 程序

```
// heap.java
// demonstrates heaps
// to run this program: C>java HeapApp
import java.io.*;
////////////////////////////////////////////////////////////////
class Node
   {
   private int iData;              // data item (key)
// -------------------------------------------------------------
   public Node(int key)            // constructor
      { iData = key; }
// -------------------------------------------------------------
   public int getKey()
      { return iData; }
// -------------------------------------------------------------
   public void setKey(int id)
      { iData = id; }
// -------------------------------------------------------------
   } // end class Node
////////////////////////////////////////////////////////////////
class Heap
   {
   private Node[] heapArray;
   private int maxSize;            // size of array
   private int currentSize;        // number of nodes in array
// -------------------------------------------------------------
   public Heap(int mx)             // constructor
      {
      maxSize = mx;
      currentSize = 0;
      heapArray = new Node[maxSize];  // create array
      }
// -------------------------------------------------------------
   public boolean isEmpty()
      { return currentSize==0; }
// -------------------------------------------------------------
   public boolean insert(int key)
      {
      if(currentSize==maxSize)
```

```
        return false;
      Node newNode = new Node(key);
      heapArray[currentSize] = newNode;
      trickleUp(currentSize++);
      return true;
      }  // end insert()
// -------------------------------------------------------------
   public void trickleUp(int index)
      {
      int parent = (index-1) / 2;
      Node bottom = heapArray[index];
      while( index > 0 &&
             heapArray[parent].getKey() < bottom.getKey() )
         {
         heapArray[index] = heapArray[parent];  // move it down
         index = parent;
         parent = (parent-1) / 2;
         }  // end while
      heapArray[index] = bottom;
      }  // end trickleUp()
// -------------------------------------------------------------
   public Node remove()           // delete item with max key
      {                           // (assumes non-empty list)
      Node root = heapArray[0];
      heapArray[0] = heapArray[--currentSize];
      trickleDown(0);
      return root;
      }  // end remove()
// -------------------------------------------------------------
   public void trickleDown(int index)
      {
      int largerChild;
      Node top = heapArray[index];       // save root
      while(index < currentSize/2)       // while node has at
         {                               //    least one child,
         int leftChild = 2*index+1;
         int rightChild = leftChild+1;
                                         // find larger child
         if(rightChild < currentSize &&  // (rightChild exists?)
                         heapArray[leftChild].getKey() <
                         heapArray[rightChild].getKey())
            largerChild = rightChild;
         else
            largerChild = leftChild;
                                         // top >= largerChild?
```

```
        if( top.getKey() >= heapArray[largerChild].getKey() )
           break;
                                    // shift child up
        heapArray[index] = heapArray[largerChild];
        index = largerChild;        // go down
        }  // end while
      heapArray[index] = top;       // root to index
      }  // end trickleDown()
// -------------------------------------------------------------
   public boolean change(int index, int newValue)
      {
      if(index<0 || index>=currentSize)
         return false;
      int oldValue = heapArray[index].getKey(); // remember old
      heapArray[index].setKey(newValue);  // change to new

      if(oldValue < newValue)             // if raised,
         trickleUp(index);                // trickle it up
      else                                // if lowered,
         trickleDown(index);              // trickle it down
      return true;
      }  // end change()
// -------------------------------------------------------------
   public void displayHeap()
      {
      System.out.print("heapArray: ");    // array format
      for(int m=0; m<currentSize; m++)
         if(heapArray[m] != null)
            System.out.print( heapArray[m].getKey() + " ");
         else
            System.out.print( "-- ");
      System.out.println();
                                    // heap format
      int nBlanks = 32;
      int itemsPerRow = 1;
      int column = 0;
      int j = 0;                    // current item
      String dots = "...............................";
      System.out.println(dots+dots);      // dotted top line

      while(currentSize > 0)              // for each heap item
         {
         if(column == 0)                  // first item in row?
            for(int k=0; k<nBlanks; k++)  // preceding blanks
               System.out.print(' ');
```

```
                                    // display item
        System.out.print(heapArray[j].getKey());
        if(++j == currentSize)          // done?
           break;

        if(++column==itemsPerRow)       // end of row?
           {
           nBlanks /= 2;                // half the blanks
           itemsPerRow *= 2;            // twice the items
           column = 0;                  // start over on
           System.out.println();        //    new row
           }
        else                            // next item on row
           for(int k=0; k<nBlanks*2-2; k++)
              System.out.print(' ');    // interim blanks
        } // end for
     System.out.println("\n"+dots+dots); // dotted bottom line
     } // end displayHeap()
// --------------------------------------------------------------
   } // end class Heap
////////////////////////////////////////////////////////////////
class HeapApp
   {
   public static void main(String[] args) throws IOException
      {
      int value, value2;
      Heap theHeap = new Heap(31);  // make a Heap; max size 31
      boolean success;

      theHeap.insert(70);           // insert 10 items
      theHeap.insert(40);
      theHeap.insert(50);
      theHeap.insert(20);
      theHeap.insert(60);
      theHeap.insert(100);
      theHeap.insert(80);
      theHeap.insert(30);
      theHeap.insert(10);
      theHeap.insert(90);

      while(true)                   // until [Ctrl]-[C]
         {
         System.out.print("Enter first letter of ");
         System.out.print("show, insert, remove, change: ");
         int choice = getChar();
```

```
            switch(choice)
               {
               case 's':                        // show
                  theHeap.displayHeap();
                  break;
               case 'i':                        // insert
                  System.out.print("Enter value to insert: ");
                  value = getInt();
                  success = theHeap.insert(value);
                  if( !success )
                     System.out.println("Can't insert; heap full");
                  break;
               case 'r':                        // remove
                  if( !theHeap.isEmpty() )
                     theHeap.remove();
                  else
                     System.out.println("Can't remove; heap empty");
                  break;
               case 'c':                        // change
                  System.out.print("Enter current index of item: ");
                  value = getInt();
                  System.out.print("Enter new key: ");
                  value2 = getInt();
                  success = theHeap.change(value, value2);
                  if( !success )
                     System.out.println("Invalid index");
                  break;
               default:
                  System.out.println("Invalid entry\n");
               } // end switch
            } // end while
         } // end main()
      //-------------------------------------------------------------
         public static String getString() throws IOException
            {
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            String s = br.readLine();
            return s;
            }
      //-------------------------------------------------------------
         public static char getChar() throws IOException
            {
            String s = getString();
            return s.charAt(0);
```

```
    }
//--------------------------------------------------------------
  public static int getInt() throws IOException
    {
    String s = getString();
    return Integer.parseInt(s);
    }
//--------------------------------------------------------------
 } // end class HeapApp
////////////////////////////////////////////////////////////////
```

堆的根在数组中的下标是 0。某些堆的实现，根在数组中的下标为 1，用下标为 0 的位置作为一个最大可能关键字的标记值。它省去了在某些算法中的一条指令，但却使概念变得复杂了。

HeapApp 中的 main()例程创建了一个堆，最大节点数为 31（在显示例程中规定了这个限制），然后插入 10 个任意关键字的节点。然后程序进入循环，用户可以输入 s、i、r 或者 c，分别对于操作显示、插入、移除或者更改关键字。

下面是程序输入输出的某个例子：

```
Enter first letter of show, insert, remove, change: s
heapArray: 100 90 80 30 60 50 70 20 10 40
.....................................................
                          100
            90                          80
      30          60          50          70
   20     10     40
.....................................................
Enter first letter of show, insert, remove, change: i
Enter value to insert: 53
Enter first letter of show, insert, remove, change: s
heapArray: 100 90 80 30 60 50 70 20 10 40 53
.....................................................
                          100
            90                          80
      30          60          50          70
   20     10     40     53
.....................................................
Enter first letter of show, insert, remove, change: r
Enter first letter of show, insert, remove, change: s
heapArray: 90 60 80 30 53 50 70 20 10 40
.....................................................
                          90
            60                          80
      30          53          50          70
   20     10     40
```

```
.....................................................
Enter first letter of show, insert, remove, change:
```

用户选择显示堆，然后增加一个关键字为 53 的数据项，再次显示堆，移除最大关键字的数据项，第三次显示堆。show()例程既显示数组也显示堆的树型表示。节点之间的连线需要用户自己想像。

扩展堆数组

在程序运行的过程中，如果插入太多的数据项，超出了堆数组的容量会发生什么情况呢？可以创建一个新的数组，把数据从旧数组中复制到新的数组中。（和哈希表中的情况不同，改变堆的大小不需要重新排列数据。）执行复制操作的时间是线性的，但是增大数组容量的操作并不会经常发生，特别是当每次扩展数组容量时，数组的容量都充分地增大了（例如，增大一倍）。

提示

在 Java 中，用 Vector 类对象取代数组对象；Vector 类对象可以动态地扩展。

堆操作的效率

对有足够多数据项的堆来说，向上筛选和向下筛选算法是本章已讲过的所有操作中最费时的部分。这两个算法的时间都花费在一个循环中，沿着一条路径重复地向上或者向下移动节点。所需的复制次数和堆的高度有关；如果树的高度为五层，需要执行四次复制把“洞”从顶层移动到底层。（这里忽略将最后一个节点复制到临时存储区以及又复制回来的两次移动；它们都是必须做的，因此也会消耗一定的时间。）

tickleUp()方法在它的循环里只有一个主要的操作：比较新插入节点的关键字和当前位置节点的关键字。tickleDown()方法需要两次比较：一次找到最大的子节点，一次比较这个最大的子节点和“最后”的节点。它们必须都要从顶层到底层或者从底层到顶层复制节点来完成操作。

堆是一种特殊的二叉树，正如第 8 章中讲过的一样，二叉树的层数 L 等于 $\log_2(N+1)$，其中 N 为节点数。tickleUp()和 tickleDown()例程中的循环执行了 L-1 次，所以 tickleUp()执行的时间和 $\log_2 N$ 成正比，tickleDown()执行时间略长一点，因为它需要执行额外的比较。总之，这里讨论的堆操作的时间复杂度都是 O(logN)。

基于树的堆

在本章所显示的图中，堆显示的样子和树差不多，因为通过树的方式更容易让读者清楚堆，但是它的实现是基于数组的。不过，也可以基于真正的树来实现堆。这棵树可以是二叉树，但不会是二叉搜索树，因为前面已经讲过，它的有序规则不是那么强。它也是一棵完全树，没有空缺的节点，称这样的树为树堆（tree heap）。

关于树堆的一个问题是找到最后一个节点。移除最大数据项的时候需要找到这个节点，因为这个节点将要插入到已移除的根的位置（然后再向下筛选）。同时，也需要找到第一个空节点，因为它是插入新节点位置（然后再向上筛选）。由于不知道它们的值，况且它不是一棵搜索树，不能直接查找这两个节点。但是，在完全树中它们的位置不是很难找的，只要根据树中的节点数目就可以找到。

正如在第 8 章讨论哈夫曼树时看到的那样，可以用二进制码表示从根到叶子的路径，用二进制数字指示从每个父节点到它子节点的路径：0 表示左子节点，1 表示右子节点。

事实证明，树中的节点数目和到最后一个节点路径的二进制编号存在简单的关系。假设根的编号为 1；下一层节点的编号为 2 和 3；第三层节点为 4、5、6 和 7；以此类推。以要查找的节点的编号开始。它就是最后一个节点或者是第一个空节点。把这个节点的编号转变为二进制数。例如，假设树中有 29 个节点，现在想要查找最后一个节点。十进制 29 转化为二进制是 11101。移除开始的 1，保留 1101。这就是从根到编号为 29 的节点的路径：向右，向右，向左，向右。第一个空的节点编号为 30，它的路径也就是二进制序列 1110（移除第一个 1 之后）：向右，向右，向右，向左。

为了执行这个运算，可以重复使用 % 操作符求出节点数目 n 被 2 整除时得出的余数（0 或者 1），并再用 / 操作符执行真正的整除 n/2。当 n 小于 1 时，操作完成。所得的余数序列，可以保存在一个数组或者字符串中，就是二进制码字（最小有效位对应路径的最底端。）下面就是执行运算的代码：

```
while(n >= 1)
   {
   path[j++] = n % 2;
   n = n / 2;
   }
```

也可以使用递归的方法来实现，每次由方法调用自身来求出余数并且每次返回时决定适当的方向。

找到适当的节点（或者空子节点）以后，堆操作就非常简单了。当向上或者向下筛选时，树的节构不改变，所以不需要移动实际的节点。只需要把一个节点复制另一个节点的位置上。用这种方法，不必为了一次移动连接或者断开所有父节点和子节点之间的关系。Node 类中需要有一个数据域保存父节点的信息，因为当向上筛选时需要访问父节点。本章将把堆树的实现留作编程作业。

树堆操作的时间复杂度为 O(logN)。因为基于数组的堆操作的大部分时间都消耗在向上筛选和向下筛选的操作上了，它操作的时间和树的高度成正比。

堆排序

堆数据结构的效率使它引出一种出奇简单，但却很有效率的排序算法，称为堆排序。

堆排序的基本思想是使用普通的 insert()例程在堆中插入全部无序的数据项，然后重复用 remove()例程，就可以按序移除所有数据项。下面就是操作的情况：

```
for(j=0; j<size; j++)
   theHeap.insert( anArray[j] );    // from unsorted array
for(j=0; j<size; j++)
   anArray[j] = theHeap.remove();   // to sorted array
```

因为 insert()和 remove()方法操作的时间复杂度都是 O(logN)，并且每个方法必须都要执行 N 次，所以整个排序操作需要 O(N*logN)时间，这和快速排序一样。但是，它不如快速排序快，部分原因是 trickleDown()里 while 循环中的操作比快速排序里循环中的操作要多。

但是，几个技巧可以使堆排序更有效。其一是节省时间，其二是节省内存。

向下筛选到适当的位置上

如果在堆中插入 N 个新数据项，则需要应用 trickleUp()方法 N 次。但是，可能所有的数据项在数组中都是任意排列的，然后再重新排列成堆，这样就只应用了 N/2 次的 trickleDown()方法。这可以使速度稍稍提升。

由两个正确的子堆形成一个正确的堆

这个方法是这样工作的：如果有一个不遵守堆有序条件的项占据了根的位置，而它的两个子堆却都是正确的堆，用 trickleDown()方法也能够创建一个正确的堆。（根本身可以是子堆的根，也可以是整个堆的根。）如图 12.8 所示。

这就提出了一个把无序的数组变成堆的方法。对（潜在的）堆底层的节点应用 trickleDown()方法——也就是说，从数组末端的节点开始，然后上行直到（下标为 0 的）根的各个节点都应用此方法。在每一步应用方法时，该节点下面的子堆都是正确的堆，因为已经对它们应用过 trickleDown()方法了。在对根应用了 trickleDown()之后，无序的数组将转化成堆。

不过，注意在底层的终端节点，就是那些没有子节点的节点，都已经是正确的堆了，因为它们是单节点的树，没有违背堆的条件。因此，不用对这些节点应用 trickleDown()。可以从节点 N/2-1，即最右一个有子节点的节点开始，而不是从节点 N-1，最后一个节点开始应用 trickleDown()。这样，筛选操作只需要执行 insert()方法 N 次的一半次数就够了。图 12.9 显示了使用向下筛选算法的次序；堆中一共有 15 个节点，从节点 6 开始筛选。

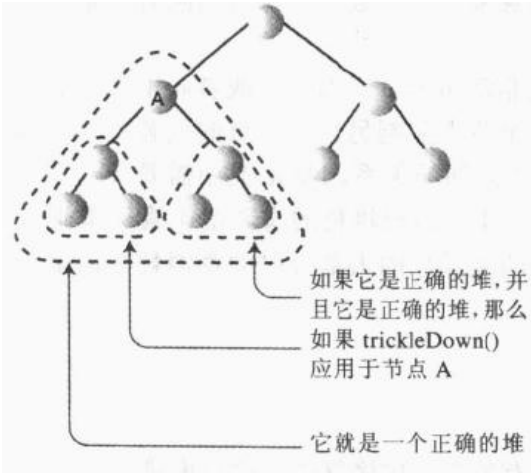


图 12.8 两个子堆必须都是正确的

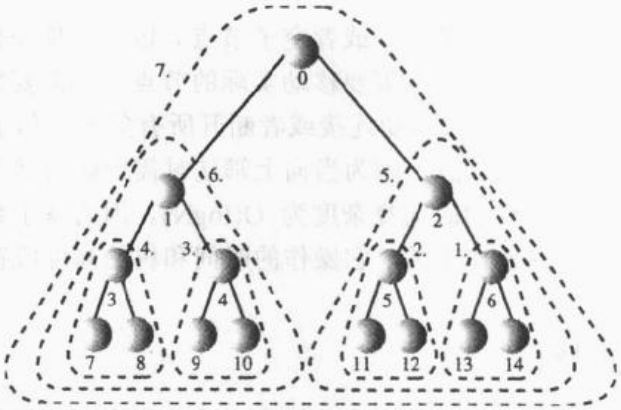


图 12.9 应用 trickleDown() 的次序

下面的代码片段将 trickleDown()应用于全部的节点，除了在底层的节点之外，从 N/2－1 处的节点开始一直到根节点为止：

```
for(j=size/2-1; j >=0; j--)
   theHeap.trickleDown(j);
```

递归的方法

也可以用递归的方法把数组变成一个堆。heapify()方法应用于根。它对根的两个子节点又调用自身，然后对这两个子节点的两个子节点分别调用自身，以此类推。最后，它执行到最底层，当它遇到没有子节点的节点时立即返回。

当它对两个子堆调用自身之后，heapify()对子树的根应用 trickleDown()。这保证了子树是正确的堆。然后 heapify()返回，对上一层子树运作。

```
heapify(int index)      // transform array into heap
   {
   if(index > N/2-1)    // if node has no children,
      return;           //    return
   heapify(index*2+2);  // turn right subtree into heap
   heapify(index*2+1);  // turn left subtree into heap
   trickleDown(index);  // apply trickle-down to this node
   }
```

递归方法可能不如简单的循环方法效率高。

使用同一个数组

原始代码片段显示了数组中的无序数据。然后把数据插入到堆中，最后从堆中移除它并把它有序地写回到数组中。这个过程需要两个大小为 N 的数组：初始数组和用于堆的数组。

事实上，堆和初始数组可以使用同一个数组。这样堆排序所需的存储空间减小了一半；除了初始数组之外不需要额外的存储空间。

本章已经讲过可以通过对数组中的一半元素进行 trickleDown()操作，而把整个数组转变成堆。只需要一个数组就能把无序的数组数据项转移到堆的适当位置上。因此，堆排序的第一步只需要一个数组。

但是，堆重复应用 remove()时的情况就复杂了。把移除的数据项放到哪里去呢？

每从堆顶移除一个数据项，堆数组的末端单元就变为空的；堆减小一个节点。可以把最近一次移除的节点放到这个新空出的单元中。移除的节点越来越多，堆数组就越来越小，但是有序数据的数组却越来越大。因此，只要稍微计划一下，有序数组和堆数组就可以共同使用一块存储空间。如果 12.10 所示。

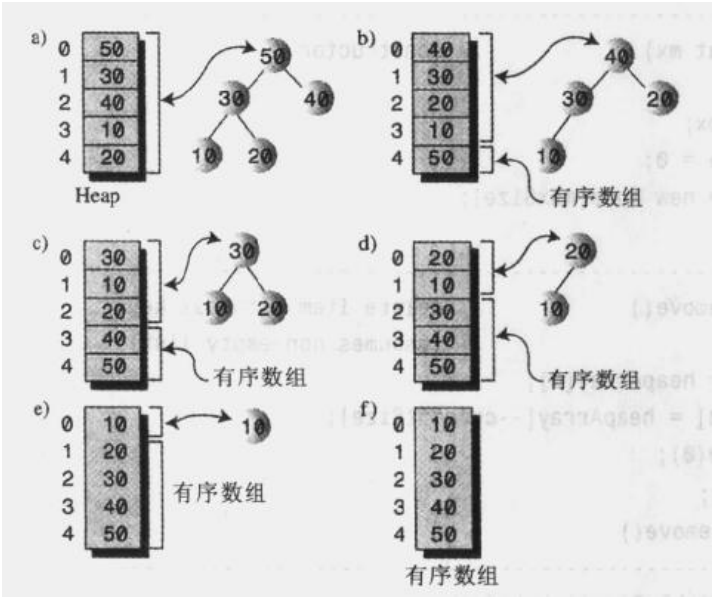


图 12.10　双用数组

heapSort.java 程序

执行堆排序的程序中可以一起使用这两个技巧——采用没有 insert()的 trickleDown()方法，以及对初始数据和堆使用同一个数组。清单 12.2 显示了完整的 heapSort.java 程序。

清单 12.2 heap.java 程序

```
// heapSort.java
// demonstrates heap sort
// to run this program: C>java HeapSortApp
import java.io.*;
////////////////////////////////////////////////////////////////
class Node
   {
   private int iData;              // data item (key)
// -------------------------------------------------------------
   public Node(int key)            // constructor
      { iData = key; }
// -------------------------------------------------------------
   public int getKey()
      { return iData; }
// -------------------------------------------------------------
   }  // end class Node
////////////////////////////////////////////////////////////////
class Heap
   {
   private Node[] heapArray;
   private int maxSize;            // size of array
   private int currentSize;        // number of items in array
// -------------------------------------------------------------
   public Heap(int mx)             // constructor
      {
      maxSize = mx;
      currentSize = 0;
      heapArray = new Node[maxSize];
      }
// -------------------------------------------------------------
   public Node remove()            // delete item with max key
      {                            // (assumes non-empty list)
      Node root = heapArray[0];
      heapArray[0] = heapArray[--currentSize];
      trickleDown(0);
      return root;
      }  // end remove()
// -------------------------------------------------------------
   public void trickleDown(int index)
```

```
    {
    int largerChild;
    Node top = heapArray[index];          // save root
    while(index < currentSize/2)          // not on bottom row
       {
       int leftChild = 2*index+1;
       int rightChild = leftChild+1;
                                          // find larger child
       if(rightChild < currentSize &&     // right ch exists?
                         heapArray[leftChild].getKey() <
                         heapArray[rightChild].getKey())
          largerChild = rightChild;
       else
          largerChild = leftChild;
                                          // top >= largerChild?
       if(top.getKey() >= heapArray[largerChild].getKey())
          break;
                                          // shift child up
       heapArray[index] = heapArray[largerChild];
       index = largerChild;               // go down
       }  // end while
    heapArray[index] = top;               // root to index
    }  // end trickleDown()
// -------------------------------------------------------------
   public void displayHeap()
      {
      int nBlanks = 32;
      int itemsPerRow = 1;
      int column = 0;
      int j = 0;                          // current item
      String dots = "................................";
      System.out.println(dots+dots);      // dotted top line

      while(currentSize > 0)              // for each heap item
         {
         if(column == 0)                  // first item in row?
            for(int k=0; k<nBlanks; k++)  // preceding blanks
               System.out.print(' ');
                                          // display item
         System.out.print(heapArray[j].getKey());

         if(++j == currentSize)           // done?
            break;

         if(++column==itemsPerRow)        // end of row?
```

```
            {
            nBlanks /= 2;                 // half the blanks
            itemsPerRow *= 2;             // twice the items
            column = 0;                   // start over on
            System.out.println();         //    new row
            }
         else                             // next item on row
            for(int k=0; k<nBlanks*2-2; k++)
               System.out.print(' ');     // interim blanks
         } // end for
      System.out.println("\n"+dots+dots); // dotted bottom line
      } // end displayHeap()
// -------------------------------------------------------------
   public void displayArray()
      {
      for(int j=0; j<maxSize; j++)
         System.out.print(heapArray[j].getKey() + " ");
      System.out.println("");
      }
// -------------------------------------------------------------
   public void insertAt(int index, Node newNode)
      { heapArray[index] = newNode; }
// -------------------------------------------------------------
   public void incrementSize()
      { currentSize++; }
// -------------------------------------------------------------
   } // end class Heap
////////////////////////////////////////////////////////////////
class HeapSortApp
   {
   public static void main(String[] args) throws IOException
      {
      int size, j;
      System.out.print("Enter number of items: ");
      size = getInt();
      Heap theHeap = new Heap(size);

      for(j=0; j<size; j++)       // fill array with
         {                        //    random nodes
         int random = (int)(java.lang.Math.random()*100);
         Node newNode = new Node(random);
         theHeap.insertAt(j, newNode);
         theHeap.incrementSize();
         }
```

```
      System.out.print("Random: ");
         theHeap.displayArray();  // display random array

      for(j=size/2-1; j>=0; j--)  // make random array into heap
         theHeap.trickleDown(j);

      System.out.print("Heap:   ");
      theHeap.displayArray();     // display heap array
      theHeap.displayHeap();      // display heap

      for(j=size-1; j>=0; j--)    // remove from heap and
         {                        //    store at array end
         Node biggestNode = theHeap.remove();
         theHeap.insertAt(j, biggestNode);
         }
      System.out.print("Sorted: ");
      theHeap.displayArray();     // display sorted array
      } // end main()
// --------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//---------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
// --------------------------------------------------------
  } // end class HeapSortApp
```

Heap 类和 heap.java 程序（清单 12.1）中的完全一样，只是为了节省空间去掉了 trickleUp()和 insert()方法，堆排序不需要这两种方法。同时还增加了 insertAt()方法，它可以把元素直接插入到堆数组中。

注意这次增加的方法没有依照面向对象编程的思想。Heap 类的接口应该对类的用户屏蔽掉堆内部的实现。内部的数组应该是不可见的，但是 insertAt()可以直接访问内部数组。这里允许违背 OOP 的原则是因为数组和堆结构的联系太紧密了。

堆类中增加的另一个方法是 incrementSize()。看起来它似乎可以和 insertAt()方法合并，但当插入时是把数组用作存储有序数据的数组的，并不希望增大堆的大小，所以仍然把这个两个方法分开。

HeapSortApp 类中的 main()例程执行如下操作：

1. 从用户那里得到数组的大小。
2. 用随机的数据填满数组。
3. 执行 N/2 次 trickleDown()方法，把数组转变为堆。
4. 从堆中移除数据项，并把它们写回数组的末端。

每执行一步都显示数组的内容。用户可以选择数组的大小。下面是 heapSort.java 某个输出的例子：

```
Enter number of items: 10
Random: 81 6 23 38 95 71 72 39 34 53
Heap:   95 81 72 39 53 71 23 38 34 6
..............................................................
                              95
              81                              72
       39             53              71              23
   38      34      6
..............................................................
Sorted: 6 23 34 38 39 53 71 72 81 95
```

堆排序的效率

前面已经讲过，堆排序运行的时间复杂度为 O(N*logN)。尽管它比快速排序略慢，但它比快速排序优越的一点是它对初始数据的分布不敏感。在关键字值按某种排列顺序的情况下，快速排序运行的时间复杂度可以降到 $O(N^2)$ 级，然而堆排序对任意排列的数据，其排序的时间复杂度都是 O(N*logN)。

小 结

- 在一个升序优先级队列中，最大关键字的数据项被称为有最高的优先级。（在降序优先级队列中优先级最高的是最小的数据项。）
- 优先级队列是提供了数据插入和移除最大（或者最小）数据项方法的抽象数据类型（ADT）。
- 堆是优先级队列 ADT 的有效的实现形式。
- 堆提供移除最大数据项和插入的方法，时间复杂度都为 O(logN)。
- 最大数据项总是在根的位置上。
- 堆不能有序地遍历所有的数据，不能找到特定关键字数据项的位置，也不能移除特定关键字的数据项。
- 堆通常用数组来实现，表现为一棵完全二叉树。根节点的下标为 0，最后一个节点的下标为 N-1。
- 每个节点的关键字都小于它的父节点，大于它的子节点。
- 要插入的数据项总是先被存放到数组第一个空的单元中，然后再向上筛选它至适当的位置。
- 当从根移除一个数据项时，用数组中最后一个数据项取代它的位置，然后再向下筛选这个节点至适当的位置。

- 向上筛选和向下筛选算法可以被看作是一系列的交换，但更有效的作法是进行一系列复制。
- 可以更改任一个数据项的优先级。首先，更改它的关键字。如果关键字增加了，数据项就向上筛选；而如果关键字减小了，数据项就向下筛选。
- 堆的实现可以基于二叉树（不是搜索树），它映射堆的结构；称为树堆。
- 存在在树堆中查找最后一个节点或者第一个空的单元的算法。
- 堆排序是一种高效的排序过程，它的时间复杂度为 O(N*logN)。
- 在概念上堆排序的过程包括先在堆中插入 N 次，然后再作 N 次移除。
- 通过对无序数组中的 N/2 个数据项施用向下筛选算法，而不作 N 次插入，可以使堆排序的运行速度更快。
- 可以使用同一个数组来存放初始无序的数据、堆以及最后有序的数据。因此，堆排序不需要额外的存储空间。

问 题

下列问题作为读者的自测题。答案见附录 C。

1. 二叉树中“完全”的意思是什么？
   a. 已经插入了所有需要插入的数据。
   b. 除了最底层可能不满之外，所有层的节点都是满的。
   c. 所有存在的节点都包含数据。
   d. 节点排列满足堆的条件。
2. 堆中的“弱序”的含义是什么？
3. 被移除的节点总是在_____。
4. 在降序堆中“向上筛选”一个节点是指
   a. 重复地交换节点和它的父节点的位置，直到它大于它的父节点。
   b. 重复地交换节点和它的子节点的位置，直到它大于它的子节点。
   c. 重复地交换节点和它的子节点的位置，直到它小于它的子节点。
   d. 重复地交换节点和它的父节点的位置，直到它小于它的父节点。
5. 堆可以用数组表示，因为堆
   a. 是完全的。
   b. 是弱序的。
   c. 是二叉树。
   d. 满足堆的条件。
6. 堆中的最后一个节点
   a. 总是左子节点。
   b. 总是右子节点。
   c. 总是在最底层。
   d. 不会小于它的兄弟。

7. 用堆可以实现优先级队列，就像用_____可以实现栈。
8. 在降序排列的堆中插入数据项需要作_____筛选。
9. 堆排序包括
   a. 从堆中移除数据项然后再把它插入。
   b. 在堆中插入数据项然后再移除它。
   c. 从一个堆中复制数据到另一个堆中。
   d. 从表示堆的数组中复制数据到堆中。
10. 执行一次堆排序需要多少个可以容纳全部数据的数组？

实　验

通过做这些实验题，可以帮助理解本章的主题。不需要编程实现。

1. 数据插入堆中的顺序会影响堆的排列吗？使用 Heap 专题 applet 试验找出答案。

2. 使用专题 applet 的 Ins 按钮在空的堆中按升序插入 10 个数据项。如果用 Rem 按钮移除这些数据项，它们是按逆序移除的吗？

3. 插入一些关键字相等的数据项。然后移除它们。你能根据它们的执行结果判断出堆排序是否稳定吗？节点的颜色是辅助的数据项。

编程作业

编程作业有助于巩固理解本章内容，并演示本章的概念如何应用。（在“简介”中提到过，资深教师可以从出版者的网站上得到编程作业的完整答案。）

12.1　修改 heap.java 程序（清单 12.1），使堆变为升序的，而不是降序的堆。（这就是说，根节点上的数据项是最小的，而不是最大的。）确保所有的操作都能正确地执行。

12.2　用 heap.java 程序中的 insert()方法在堆中插入一个新节点，确保不违背堆的条件。另编写 toss()方法，把新节点放到堆数组中，不需要保持堆的条件。（可能每个新节点只是简单地放在数组的末端。）再编写一个 restoreHeap()方法，它能在全堆内恢复堆的条件。重复应用 toss()，一次插入大量的数据项，然后再用一次 restoreHeap()，比反复应用 insert()的效率要高。查看堆排序的描述找到暗示。用 insert()插入几个数据，再用 toss()方法放置几个数据，然后再用 restoreHeap()恢复堆，以此测试编写的程序。

12.3　应用堆取代数组来实现 priorityQ.java 程序（清单 4.6）中 PriorityQ 类。可以不用修改 heap.java 程序（清单 12.1）中的 Heap 类，直接应用它。把它作成一个降序队列（移除最大数据项）。

12.4　用基于数组的堆实现的优先级队列存在一个问题，就是数组的大小为固定的。如果数据量超过了数组容量，就需要扩展数组，就像在第 11 章的编程作业 11.4 中对哈希表所做的修改一样。用普通的二叉搜索树而不是堆来实现优先级队列就可以避免这个问题。一棵树可以要多大就有多大（只要不超过系统内存的容量就行）。

从 tree.java 程序（清单 8.1）中的 Tree 类出发，修改这个类，使它支持优先级队列，增加移除

最大数据项的方法 removeMax()。这一点在堆中是很容易作到的，但是在树中稍微有点困难。如何在树中找到最大的数据项呢？当移除一个节点时需要考虑它的子节点吗？编写 change()方法是一种选择。用它在二叉搜索树中移除老数据项和插入另一个不同关键字的新数据项是很容易的。

应用代码应该只与 PriorityQ 类有关，因此在 main()中 Tree 类是不可见的（也许除了当调试时要显示树之外）。插入操作和 removeMax()的时间复杂度都是 O(logN)。

12.5 编写程序实现本章中所讨论过的树堆（基于树而实现的堆），确保能够进行移除最大的数据项，插入数据项，以及修改数据项关键字的操作。

第 13 章

图

本章重点

- 图简介
- 搜索
- 最小生成树
- 有向图的拓扑排序
- 有向图的连通性

在计算机程序设计中，图是最常用的结构之一。一般来说，用图来帮助解决的问题类型与本书中已经讨论过的问题类型有很大差别。如果处理一般的数据存储问题，可能用不到图，但对某些问题（经常是一些有趣的问题），图是必不可少的。

对图的讨论分成两章。本章讨论不带权的图，展示这些图的算法，并且演示两个模拟不带权图的专题 applet。下一章讨论关于带权图的更复杂的算法。

图简介

图是一种与树有些相像的数据结构。实际上，从数学意义上说，树是图的一种。然而，在计算机程序设计中，图的应用方式与树不同。

本书前面讨论的数据结构都有一个框架，这个框架都是由相应的算法规定的。例如，二叉树是那样一个形状，就是因为那样的形状使它容易搜索数据和插入新数据。树的边表示了从一个节点到另一个节点的快捷方式。

另一方面，图通常有一个固定的形状，这是由物理或抽象的问题所决定的。例如，图中节点表示城市，而边可能表示城市间的班机航线。另一个更抽象的例子是一个代表了几个单独任务的图，这些任务是完成一个项目所必需的。在图中，节点可能代表任务，有向边（在一端带一个箭头）指示某个任务必须在另一个任务前完成。在这种情形下，图的形状取决于真实世界的具体情况。

在继续讨论下去之前，必须说明，当讨论图时，节点通常叫做顶点。可能是因为图的命名在几个世纪前的数学领域就有了，所以它比树的命名要早。树和计算机科学联系得更紧密一些。但是，这些术语或多或少会交替使用。

定义

图 13.1a 显示了美国加利福尼亚 San Jose 周边地区的简化的高速公路网。图 13.1b 是模拟这些高速公路的图。

在图中，圆圈代表高速公路的交汇点，连接圆圈的直线代表高速公路段。圆圈是顶点，线是边。顶点通常用一些方法来标识——正如图中显示的那样，用字母表中的字母来表示。每条边由两个顶点作为两端。

图并不是要试图反映地图上的地理位置；它只是显示了顶点和边的关系——即，哪些边连接着哪些顶点。它本身不涉及物理的远近和方向。而且一条边可能代表几条不同的公路，例如从 I 到 H 的情形中，就包含了 101 号公路、84 号公路和 280 号公路。两个交叉点间的连通性（或不连通性）

是重要的，而实际的路线并不重要。

邻接

如果两个顶点被同一条边连接，就称这两个顶点是邻接的。图 13.1 中，顶点 I 和 G 是邻接的，但 I 和 F 就不是。和某个指定顶点邻接的顶点有时叫做它的邻居。例如 G 的邻居是 I、H 和 F。

路径

路径是边的序列。图 13.1 显示了一条从顶点 B 到顶点 J 的路径，这条路径通过了顶点 A 和顶点 E。这条路径称作 BAEJ。这两个顶点之间还有其他路径：从 B 到 J 的另一条路径是 BCDJ。

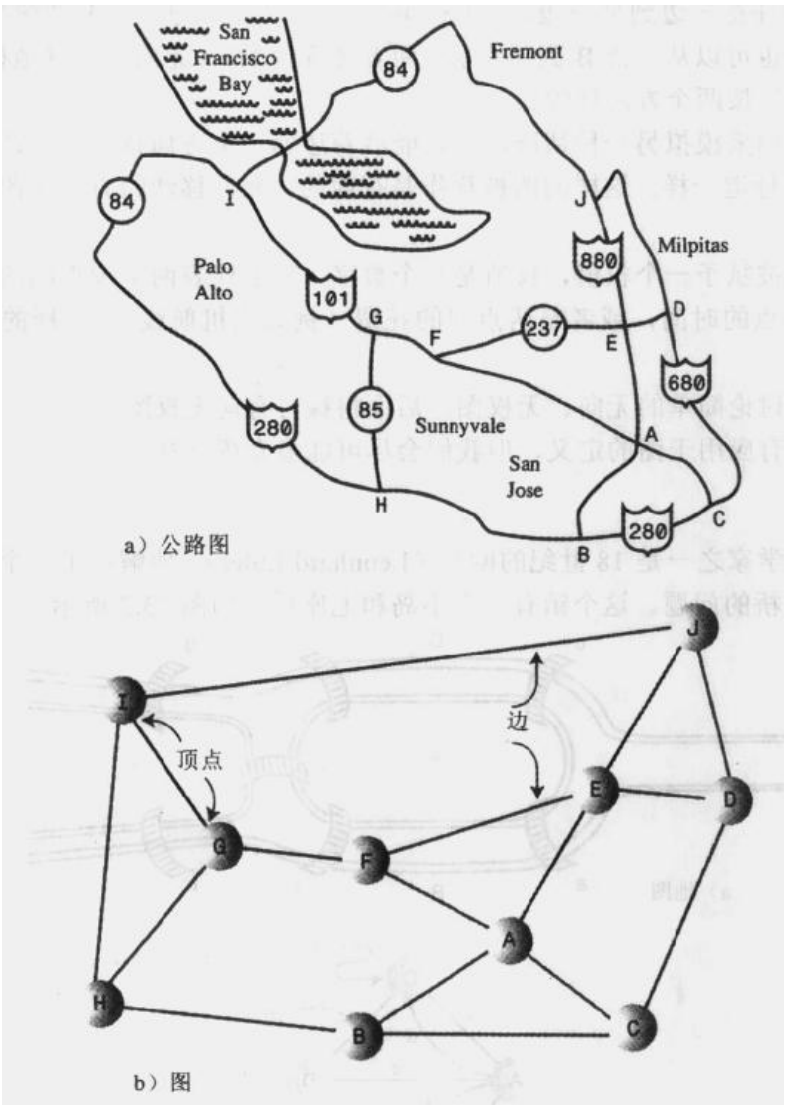


图 13.1　公路图和图

连通图

如果至少有一条路径可以连接起所有的顶点，那么这个图被称作连通的，例如图 13.2a。然而，如果“不能从这里到那里”（就像 Vermont 的农民通常向停下来问路的城里人说的那样），那么这个图就是非连通的，例如图 13.2b。

非连通图包含几个连通子图。在图 13.2b 中，A 和 B 是一个连通子图，C 和 D 也是一个连通子图。

简便起见，本章讨论的算法都是应用在连通图或非连通图的连通子图中。如果需要，进行一些小的修正就可以使它们应用在非连通图中。

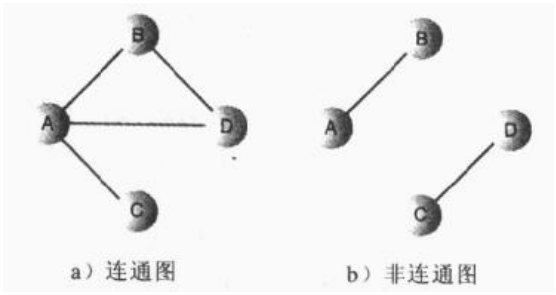


图 13.2 连通图和非连通图

有向图和带权图

图 13.1 和图 13.2 中的图是无向图。这说明图中的边没有方向；可以从任意一边到另一边。所以，可以从顶点 A 到顶点 B，也可以从顶点 B 到顶点 A，两者是等价的。（无向图很好地模拟了高速公路网，因为在一条公路上可以按两个方向行驶。）

然而，图还经常用来模拟另一种情况，即只能沿着边朝一个方向移动——只能从 A 到 B，而不能从 B 到 A，就像单行道一样。这样的图被称作是有向的。允许移动的方向在图中通常用边一端的箭头表示。

在某些图中，边被赋予一个权值，权值是一个数字，它能代表两个顶点间的物理距离，或者从一个顶点到另一个顶点的时间，或者是两点间的花费（例如飞机航线）。这样的图叫做带权图。下一章将讨论它们。

本章开始我们将讨论简单的无向、无权图，后面将探讨有向无权图。

我们无法覆盖所有应用于图的定义，但我们会尽可能多介绍一些。

历史的笔记

研究图最早的数学家之一是 18 世纪的欧拉（Leonhard Euler）。他解决了一个著名的难题，关于波兰 Königsberg 镇的桥的问题。这个镇有一个小岛和七座桥，如图 13.3 所示。

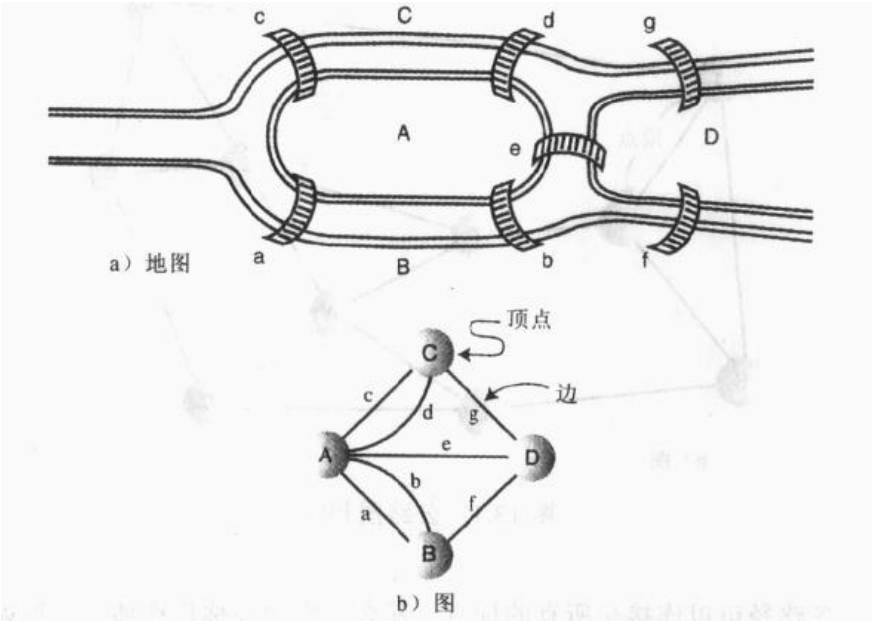


图 13.3 Königsberg 的桥

这个问题已经被市民讨论很多次了，它要寻找到一条走遍所有桥的路线，但不允许包含任何交

义路线。在这里不打算叙述 Euler 的解决方案，它的结果是没有这样一条通路。然而，解决问题的关键所在是把这个问题用图表示，其中把岛的区域作为顶点，桥作为边，如图 13.3b 所示。这可能是把图用于表示真实世界的一个具体问题的第一个例子。

在程序中表示图

计算机发明以前，以抽象的方式考虑图是非常恰当的，就像 Euler 和其他数学家做的那样，但是现在需要用计算机来描述图。什么样的软件结构适合于模拟图？首先来看顶点的情况，然后是边的情况。

顶点

在非常抽象的图的问题中，只是简单地把顶点编号，从 0 到 N-1（这里 N 是顶点数）。不需要任何变量类型存储顶点，因为它们的用处来自于它们之间的相互关系。

然而在大多数情况下，顶点表示某个真实世界的对象，这个对象必须用数据项来描述。例如，如果在一个飞机航线模拟程序中，顶点代表城市，那么它需要存储城市名字、海拔高度、地理位置和其他相关信息。因此，通常用一个顶点类的对象来表示一个顶点。示例程序仅存储了一个字母（例如 A），用来标识顶点，同时还有一个标志位，后面将会看到它在搜索算法中的作用。Vertex 类如下所示：

```
class Vertex
   {
   public char label;        // label (e.g. 'A')
   public boolean wasVisited;

   public Vertex(char lab)   // constructor
      {
      label = lab;
      wasVisited = false;
      }
   } // end class Vertex
```

顶点对象能放在数组中，然后用下标指示。在本例中，用数组 vertexList 存储顶点对象。顶点也可以放在链表或其他数据结构中。不论使用什么结构，存储只为了使用方便。这与边如何连接点没有关系。要达到这个目的，还需要其他机制。

边

在第 9 章“红-黑树”中，可以看到计算机程序能用几种方式表示树。大多数情况下讨论的树都是每个节点包含它的子节点的引用，但也提到过用数组表示树，数组中的节点位置决定了它和其他节点的关系。在第 12 章“堆”中，就用数组表示了一种树的类型，叫做“堆”。

然而，图并不像树，拥有几种固定的结构。二叉树中，每个节点最多有两个子节点，但图的每个顶点可以与任意多个顶点连接。例如，图 13.2a 中，顶点 A 与三个顶点连接，而 C 只与一个顶点连接。

为了模拟这种自由形式的组织结构，需要用一种不同的方法表示边，比树的表示方法更合适些。一般用两个方法表示图：即邻接矩阵和邻接表。（记住，如果一条边连接两个顶点，这两个顶点就是邻接的。）

邻接矩阵

邻接矩阵是一个二维数组，数据项表示两点间是否存在边。如果图有 N 个顶点，邻接矩阵就是 N*N 的数组。表 13.1 显示了图 13.2a 中图的邻接矩阵。

表 13.1 邻接矩阵

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

顶点被用作行和列的标题。两个顶点间有边则标识为 1，没有边则标识为 0（也可以使用布尔变量的 true/false 值来标识）。如表所示，顶点 A 和另外三个顶点邻接，B 和 A、D 邻接，C 只与 A 邻接，而 D 与 A 和 B 邻接。在本例中，顶点与其自身的“连接”设为 0，从 A-A 到 D-D 称为主（同一）对角线，所以这条从左上角到右下角的对角线全部是 0。主对角线上的实体不代表任何真实世界的信息，所以为了方便，也可以把主对角线的值设为 1。

注意，这个矩阵的上三角是下三角的镜像；两个三角包含了同样的信息。这个冗余信息看似低效，但在大多数计算机语言中，创造一个三角形数组比较困难，所以只好求其次接受这个冗余。这也要求当增加一条边时，必须更新邻接矩阵的两部分，而不是一部分。

邻接表

表示边的另一种方法是邻接表。邻接表中的表指的是第 5 章“链表”中讨论的那种链表。实际上，邻接表是一个链表数组（或者是链表的链表）。每个单独的链表表示了有哪些顶点与当前顶点邻接。表 13.2 显示了图 13.2a 中图的邻接表。

表 13.2 邻接表

顶点	包含邻接顶点的链表
A	B—>C—>D
B	A—>D
C	A
D	A—>B

在这个表中，符号—>表示链表中的一个节点。链表中每个节点都是一个顶点。在这里的每个链表中，顶点按字母顺序排列，不过这并不是必需的。不要把邻接表的内容与路径混淆。邻接表表示了当前顶点与哪些顶点连接——即两个顶点间存在边，而不是表示顶点间的路径。

后面会看到什么时候使用邻接矩阵，而什么时候使用邻接表。本章的所有专题 applet 都使用邻接矩阵，但有时链表的方法可能效率更高。

在图中添加顶点和边

为了向图中添加顶点，必须用 new 保留字生成一个新的顶点对象，然后插入到顶点数组 vertexList 中。在模拟真实世界的程序中，顶点可能包含许多数据项，但是为了简便起见，这里假定

顶点只包含单一的字符。因此顶点的创建用下面的代码：

```
vertexList[nVerts++] = new Vertex('F');
```

这就插入了顶点 F，nVerts 变量是图中当前顶点数。

怎样添加边取决于用邻接矩阵还是用邻接表表示图。假定使用邻接矩阵并考虑在顶点 1 和顶点 3 之间加一条边。这些数字对应 vertexList 数组的下标，顶点存储在数组的对应位置。首次创建邻接矩阵 adjMat 时，初值为 0。添加边的代码如下：

```
adjMat[1][3] = 1;
adjMat[3][1] = 1;
```

如果使用邻接表，就把 1 加到 3 的链表中，然后把 3 加到 1 的链表中。

Graph 类

下面看一下 Graph 类，它包含创建邻接表和邻接矩阵的方法，以向 Graph 对象插入顶点和边的方法：

```
class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // array of vertices
   private int adjMat[][];      // adjacency matrix
   private int nVerts;          // current number of vertices
// ----------------------------------------------------------
   public Graph()               // constructor
      {
      vertexList = new Vertex[MAX_VERTS];
                                        // adjacency matrix
      adjMat = new int[MAX_VERTS][MAX_VERTS];
       nVerts = 0;
       for(int j=0; j<MAX_VERTS; j++)      // set adjacency
          for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
             adjMat[j][k] = 0;
       } // end constructor
 // ----------------------------------------------------------
    public void addVertex(char lab)    // argument is label
       {
       vertexList[nVerts++] = new Vertex(lab);
       }
 // ----------------------------------------------------------
    public void addEdge(int start, int end)
       {
       adjMat[start][end] = 1;
       adjMat[end][start] = 1;
       }
 // ----------------------------------------------------------
    public void displayVertex(int v)
```

```
        {
        System.out.print(vertexList[v].label);
        }
    // ---------------------------------------------------------
    }  // end class Graph
```

在 Graph 类中，vertexList 数组的下标惟一地表示一个顶点。

上面已经讨论了这里的大多数方法。为了显示矩阵，只需要打印顶点的字符。

邻接矩阵（或者邻接表）提供了关于当前顶点的位置信息。特别是，当前顶点通过边与哪些顶点相连。为了回答关于顶点序列的更一般问题，就必须求助于其他的算法。下面从搜索开始。

搜　索

在图中实现的最基本的操作之一就是搜索从一个指定顶点可以到达哪些顶点。例如，可以想像要找出美国有多少个城市可以从 Kansas 城乘坐旅行列车到达（假定中途不换车）。一些城市可以直达，而有些城市因为没有旅行列车服务而不能直达。有些地方即使有列车服务也不能到达，因为他们的铁轨系统（例如 Hayfork-Hicksville RR 用窄铁轨标准）不能和出发地或者沿途的标准铁轨系统相连。

还有另外一种情形可能需要找到所有当前顶点可到达的顶点。设想需要设计一个印刷电路板，就像在计算机中使用的这一个。（打开机箱，看看它！）不同的部分[大多数是集成电路块（IC）]放置在电路板上，集成电路的针脚插在电路板的预留孔中。IC 芯片焊接在恰当位置，它们的针脚通过连线（trace）与其他针脚保持电路连接。连线是在电路板表面的细小金属线，如图 13.4 所示。（并不需要了解这幅图的细节。）

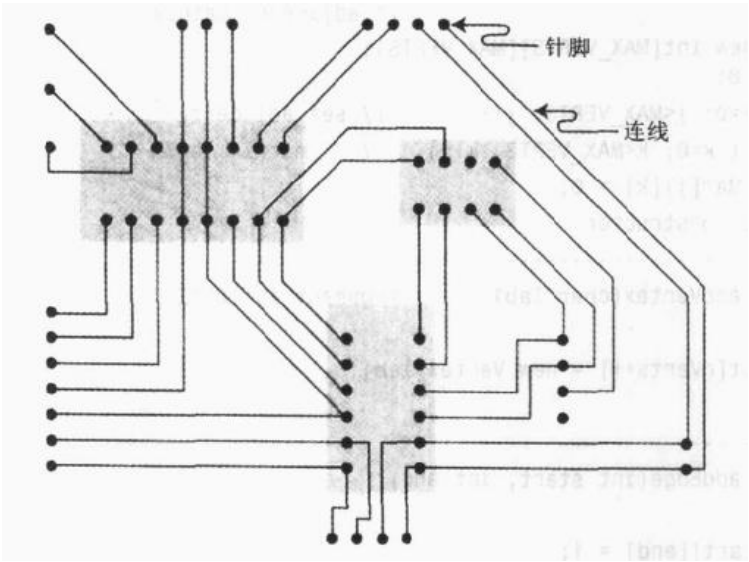


图 13.4　电路板的针脚和连线

在图中，顶点代表每个针脚，边代表每条连线。在电路板上，有许多线路是互不相连的，所以图绝不是连通图。因此，设计过程中，创建一个图，利用它找到哪些针脚连接到同一电路中，真是

太有用了。

假设已经创建了这么一个图。现在需要一种算法来提供系统的方法，从某个特定顶点开始，沿着边移动到其他顶点。移动完毕后，要保证访问了和起始点相连的每一个顶点。正如在第 8 章“二叉树”中讨论的二叉树一样，访问意味着在顶点上的某种操作，例如显示操作。

有两种常用的方法可用来搜索图：即深度优先搜索（DFS）和广度优先搜索（BFS）。它们最终都会到达所有连通的顶点。深度优先搜索通过栈来实现，而广度优先搜索通过队列实现。正如我们即将看到的那样，不同的机制导致了搜索的不同方式。

深度优先搜索

在搜索到尽头的时候，深度优先搜索用栈记住下一步的走向。这里展示了一个例子，最好实验一下 GraphN 专题 applet，最后考察一下执行搜索的具体代码。

一个示例

下面讨论图 13.5 所示的深度优先搜索思想。图中的数字显示了顶点被访问的顺序。

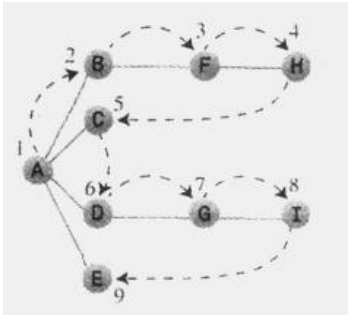


图 13.5 深度优先搜索

为了实现深度优先搜索，找一个起始点——本例为顶点 A。需要做三件事：首先访问该顶点，然后把该点放入栈中，以便记住它，最后标记该点，这样就不会再访问它。

下面可以访问任何与顶点 A 相连的顶点，只要还没有访问过它。假设顶点按字母顺序访问，所以下面访问顶点 B。然后标记它，并放入栈中。

现在已经访问了 B，做相同的事情：找下一个未访问的顶点，也就是 F。这个过程称作规则 1。

规则 1

如果可能，访问一个邻接的未访问顶点，标记它，并把它放入栈中。

再次应用规则 1，这次访问顶点 H。然而，这里还需要做一些事情，因为没有和 H 邻接的未访问顶点。下面是规则 2。

规则 2

当不能执行规则 1 时，如果栈不空，就从栈中弹出一个顶点。

根据这条规则，从栈中弹出 H，这样就又回到了顶点 F。F 也没有与之邻接且未访问的顶点了。那么再弹出 F，这回到顶点 B。这时只有顶点 A 在栈中。

然而 A 还有未访问的邻接点，所以访问下一个顶点 C。但是 C 又是这一条路线的终点，所以从栈中弹出它，再次回到 A 点。接着访问 D、G 和 I。当到达 I 时，把它们都弹出栈。现在回到 A，然后访问 E，最后再次回到 A。

然而这次 A 也没有未访问的邻接点，所以把它也弹出栈。现在栈中已无顶点。下面是规则 3。

规则 3

如果不能执行规则 1 和规则 2，就完成了整个搜索过程。

表 13.3 表示了对图 13.5 进行深度优先搜索过程中栈的内容。

表 13.3 深度优先搜索中的栈内容

事件	栈
VisitA	A
VisitB	AB
VistF	ABF
VisitH	ABFH
PopH	ABF
PopF	AB
PopB	A
VisitC	AC
PopC	A
VisitD	AD
VisitG	ADG
VisitI	ADGI
PopI	ADG
PopC	AD
PopG	A
VisitE	AE
PopE	A
PopA	
Done	

栈的内容就是刚才从起始顶点到各个顶点访问的整个过程。从起始顶点出发访问下一个顶点时，就把这个顶点入栈。回到起始顶点时，出栈。访问顶点的顺序是 ABFHCDGIE。

深度优先搜索算法要得到距离起始点最远的顶点，然后在不能继续前进的时候返回。使用深度这个术语表示与起始点的距离，便可以理解“深度优先搜索”的意义。

模拟问题

深度优先搜索与迷宫问题类似。迷宫在英国很流行，可以由一方给另一方设置障碍，由另一方想办法通过。迷宫由狭窄的过道（认为是边）和过道的交汇点（顶点）组成。

假设有个人在迷宫中迷路。她知道有一个出口，并且计划系统地搜索迷宫找到出口。幸运的是，她有一团线和一支笔。她从某个交汇点开始，任意选择一个通路，从线团上退下一些线。在下一个交汇点，她继续随机选择一条通路，再退下一些线，直到最后她到达死胡同。

到达死胡同时，她按原路返回，把线再绕上，直到到达前一个交汇点。她标记了以前走过的路径，所以不会重复走那些通路，而选择未走过的通路。当她标记了这个交汇点的所有通路，就会再回到上一个交汇点，并且重复这个过程。

线代表栈：它“记住”了走向某个特定点的路径。

GraphN 专题 Applet 和 DFS

通过点击 GraphN 专题 applet 中的 DFS 按钮可以执行深度优先搜索。（N 表示无方向，无权值。）

运行 applet。开始时并没有顶点或边，只是一个空的长方形。在希望创建顶点的区域双击来建立顶点。第一个顶点自动标记为 A，第二个点为 B，依此类推。每个点的颜色是随机的。

要增加一条边，从一个顶点起托拽鼠标到另一个顶点。图 13.6 显示了用这个 applet 创建的图 13.5 中的图。

由于无法删除单个顶点或边，所以一旦出错，就需要点击 New 按钮重新开始，这时会擦除所有存在的顶点和边。（执行删除前会有提示。）点击 View 按钮，切换成图的邻接矩阵形式，如图 13.7 所示。再次点击 View 按钮切回图形模式。

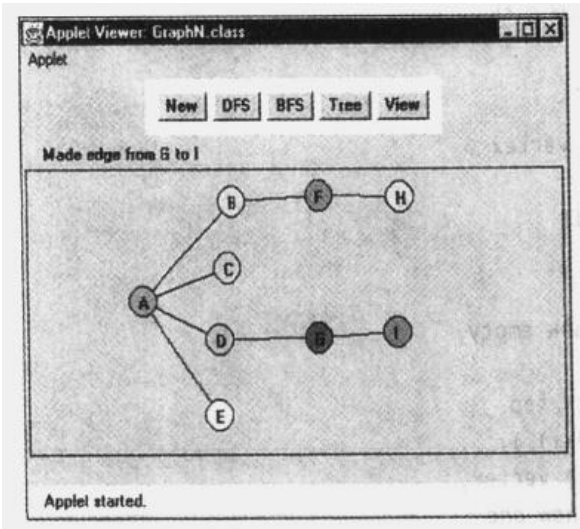


图 13.6 GraphN 专题 applet

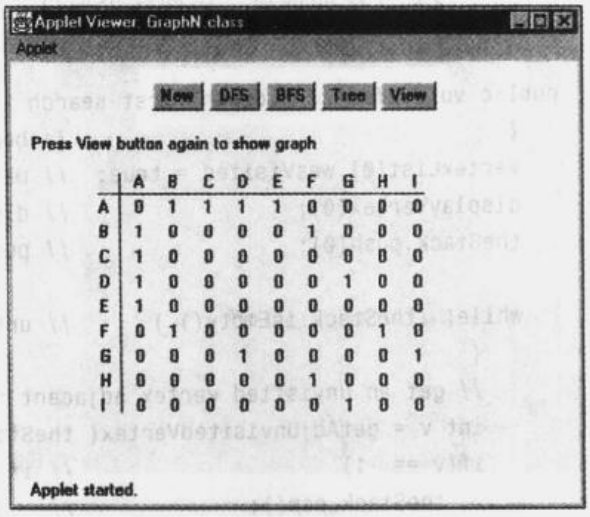


图 13.7 GraphN 中的邻接矩阵

要运行深度优先搜索算法，再次点击 DFS 按钮。这个过程开始后，系统提示单击（不是双击）作为出发点的顶点。

可以重新创建图 13.6 的图，或者创建复杂度不同的各种图。只要使用一会儿，就可以知道下一步算法会如何进行（除非图太复杂）。

如果在非连通图中应用这个算法，它就只能找到那些与起始点连通的顶点。

Java 代码

深度优先搜索的关键在于能够找到与某一顶点邻接且没有访问过的顶点。如何做呢？邻接矩阵是关键。找到指定顶点所在的行，从第一列开始向后寻找值为 1 的列；列号是邻接顶点的号码。检查这个顶点是否未访问过，如果是这样，那么这就是要访问的下一个顶点。如果该行没有顶点既等于 1（邻接）且又是未访问的，那么与指定点相邻接的顶点就全部访问过了。这个过程在 getAdjUnvisitedVertex()方法中实现：

```
// returns an unvisited vertex adjacent to v
public int getAdjUnvisitedVertex(int v)
   {
   for(int j=0; j<nVerts; j++)
      if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
         return j;                  // return first such vertex
```

```
    return -1;                      // no such vertices
    } // end getAdjUnvisitedVertex()
```

现在开始考察 Graph 类中的 dfs()方法，这个方法实际执行了深度优先搜索。下面会看到这段代码如何包含了前面提出的三条规则。它循环执行，直到栈为空。每次循环中，它做四件事：

1. 用 peek()方法检查栈顶的顶点。
2. 试图找到这个顶点还未访问的邻接点。
3. 如果没有找到，出栈。
4. 如果找到这样的顶点，访问这个顶点，并把它放入栈。
5. 下面是 dfs()方法的代码：

```
public void dfs()  // depth-first search
   {                                // begin at vertex 0
   vertexList[0].wasVisited = true; // mark it
   displayVertex(0);                // display it
   theStack.push(0);                // push it

   while( !theStack.isEmpty() )     // until stack empty,
      {
      // get an unvisited vertex adjacent to stack top
      int v = getAdjUnvisitedVertex( theStack.peek() );
      if(v == -1)                   // if no such vertex,
         theStack.pop();            //    pop a new one
   else                            // if it exists,
      {
      vertexList[v].wasVisited = true;  // mark it
      displayVertex(v);                 // display it
      theStack.push(v);                 // push it
      }
   } // end while

   // stack is empty, so we're done
   for(int j=0; j<nVerts; j++)    // reset flags
      vertexList[j].wasVisited = false;
} // end dfs
```

在 dfs()方法的最后，重置了所有 wasVisited 标记位，这样就可以在稍后继续使用 dfs()方法。栈此时已为空，所以不需要重置。

现在 Graph 类已经有了需要的所有片段。下面是创建图对象的代码，并在图中加入一些顶点和边，然后执行深度优先搜索：

```
Graph theGraph = new Graph();
theGraph.addVertex('A');    // 0  (start for dfs)
theGraph.addVertex('B');    // 1
theGraph.addVertex('C');    // 2
theGraph.addVertex('D');    // 3
```

```
theGraph.addVertex('E');    // 4

theGraph.addEdge(0, 1);     // AB
theGraph.addEdge(1, 2);     // BC
theGraph.addEdge(0, 3);     // AD
theGraph.addEdge(3, 4);     // DE

System.out.print("Visits: ");
theGraph.dfs();             // depth-first search
System.out.println();
```

图 13.8 显示了这段代码创建的图，下面是输出：

```
Visits: ABCDE
```

可以改变上述代码创建其他图，运行它，并观察如何执行深度优先搜索。

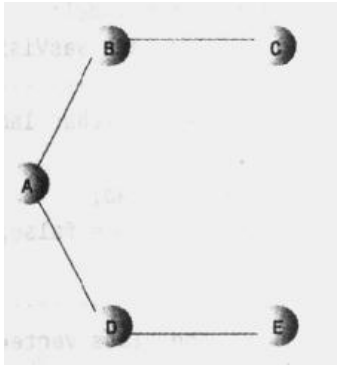


图 13.8 dfs.java 和 bfs.java 使用的图

dfs.java 程序

清单 13.1 显示了 dfs.java 程序，它包含 dfs()方法，同时含有 StackX 类，这个类来自第 4 章“栈和队列”。

清单 13.1 dfs.java 程序

```
// dfs.java
// demonstrates depth-first search
// to run this program: C>java DFSApp
////////////////////////////////////////////////////////////////
class StackX
   {
   private final int SIZE = 20;
   private int[] st;
   private int top;
// ----------------------------------------------------------
   public StackX()           // constructor
      {
      st = new int[SIZE];    // make array
      top = -1;
      }
// ----------------------------------------------------------
   public void push(int j)   // put item on stack
      { st[++top] = j; }
// ----------------------------------------------------------
   public int pop()          // take item off stack
      { return st[top--]; }
// ----------------------------------------------------------
   public int peek()         // peek at top of stack
      { return st[top]; }
// ----------------------------------------------------------
```

```
   public boolean isEmpty()  // true if nothing on stack-
      { return (top == -1); }
// ------------------------------------------------------------
   }  // end class StackX
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // label (e.g. 'A')
   public boolean wasVisited;
// ------------------------------------------------------------
   public Vertex(char lab)   // constructor
      {
      label = lab;
      wasVisited = false;
      }
// ------------------------------------------------------------
   }  // end class Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // list of vertices
   private int adjMat[][];      // adjacency matrix
   private int nVerts;          // current number of vertices
   private StackX theStack;
// ------------------------------------------------------------
   public Graph()               // constructor
      {
      vertexList = new Vertex[MAX_VERTS];
                                          // adjacency matrix
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // set adjacency
         for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
            adjMat[j][k] = 0;
      theStack = new StackX();
      }  // end constructor
// ------------------------------------------------------------
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// ------------------------------------------------------------
   public void addEdge(int start, int end)
      {
```

```
      adjMat[start][end] = 1;
      adjMat[end][start] = 1;
      }
// ---------------------------------------------------------
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }
// ---------------------------------------------------------
   public void dfs()  // depth-first search
      {                                 // begin at vertex 0
      vertexList[0].wasVisited = true;  // mark it
      displayVertex(0);                 // display it
      theStack.push(0);                 // push it

      while( !theStack.isEmpty() )      // until stack empty,
         {
         // get an unvisited vertex adjacent to stack top
         int v = getAdjUnvisitedVertex( theStack.peek() );
         if(v == -1)                    // if no such vertex,
            theStack.pop();
         else                           // if it exists,
            {
            vertexList[v].wasVisited = true;  // mark it
            displayVertex(v);                 // display it
            theStack.push(v);                 // push it
            }
         }  // end while

      // stack is empty, so we're done
      for(int j=0; j<nVerts; j++)          // reset flags
         vertexList[j].wasVisited = false;
      }  // end dfs
// ---------------------------------------------------------
   // returns an unvisited vertex adj to v
   public int getAdjUnvisitedVertex(int v)
      {
      for(int j=0; j<nVerts; j++)
         if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
            return j;
      return -1;
      }  // end getAdjUnvisitedVertex()
// ---------------------------------------------------------
   }  // end class Graph
////////////////////////////////////////////////////////////////
```

```
class DFSApp
  {
  public static void main(String[] args)
     {
     Graph theGraph = new Graph();
     theGraph.addVertex('A');    // 0  (start for dfs)
     theGraph.addVertex('B');    // 1
     theGraph.addVertex('C');    // 2
     theGraph.addVertex('D');    // 3
     theGraph.addVertex('E');    // 4

     theGraph.addEdge(0, 1);     // AB
     theGraph.addEdge(1, 2);     // BC
     theGraph.addEdge(0, 3);     // AD
     theGraph.addEdge(3, 4);     // DE

     System.out.print("Visits: ");
     theGraph.dfs();             // depth-first search
     System.out.println();
     }  // end main()
  }  // end class DFSApp
//////////////////////////////////////////////////////////////
```

深度优先搜索和游戏仿真

深度优先搜索通常用在游戏仿真中（以及真实世界中与游戏相似的情况）。在一般的游戏中，可以在几个可能的动作中选择一个。每个选择导致更进一步的选择，这些选择又产生了更多的选择，这样就形成了一个代表可能性的不断伸展的树形图。一个选择点代表一个顶点，采取的特定选择代表边，由它可以到达下一个选择顶点。

想像一个名为 tic-tac-toe 的游戏。如果你先走，可以在九种可能的移动中选择一种。而对手只能在八种可能的移动中选择一种，依此类推。每次移动都导致了对手可以对当时的情况做出选择，而她的选择又影响到你下一步的走向，直到最后一个方块被填充。

当考虑如何走时，一种方法是在心中默想一步移动，然后对手可能的对策，然后是自己的对策，一直进行下去。可以通过考察哪一步移动能产生最好的结果来选择下一步怎么走。在 tic-tac-toe 这种简单游戏中，可能的移动选择很有限，所以可以很轻松地沿每种路径考虑到游戏的最后一步。当分析了全部的路径后，就可以知道哪条路径最佳。这可以通过图来表示。图中的一个顶点代表首次移动，它连接着八个顶点，代表对手的八种移动可能，它们每一个又连接着七个顶点，代表相应的移动可能性，依此类推。从起始点到终点的所有路径都包含九个顶点。要做一个完整的分析，就需要画九张图，每张图代表不同的起始点。

即使在这种简单的游戏中，搜索路径的数量也是出奇的大。如果可根据忽略所作的简化，九张图就会有 9*8*7*6*5*4*3*2*1 条路径。这是 9 的阶乘（9!），或 362880。在国际象棋这种游戏中，可能的移动数量还要大得多，因此，即使最强大的计算机（例如 IBM 的“深蓝”）也不能“看到”比赛的结果。它们只能沿某条路径到达一定的深度，然后估算棋盘上的形式，来判断是否这个选择

比其他的选择要好。

用计算机考察这种问题，通常的方法是使用深度优先搜索。在每个顶点决定下一步的移动，就像 dfs.java 程序（清单 13.1）中 getAdjUnvisitedVertex()方法执行的那样。如果还有未访问的顶点（选择点），就把当前顶点入栈，然后继续选择。如果发现在某一点已经不能再作出选择（getAdjUnvisitedVertex()返回-1），就从栈中弹出一个顶点，并回到这个顶点处，然后看这里是否还有未试过的选择。

可以把游戏中移动的序列考虑成一棵树，节点代表移动。首次移动是根节点。在 tic-tac-toe 游戏中，首次移动后只有八种可能的移动，那么有八个第二层节点与根节点连接。接下来只有七种可能的移动，那么每个第二层节点就有七个第三层节点与之相连。从根节点到叶节点，为构造这棵树需要有 9！条路径。这叫作决策树。

实际上，决策树中的每一支的数量可以缩减，因为在所有方格填满前，游戏可能已经结束了。然而，tic-tac-toe 游戏的决策树还是非常庞大和复杂，不过和国际象棋等游戏比起来，它还算是比较简单的游戏。

决策树中只有几条路径有成功解。例如，有些会导致对手获胜。当到达叶节点时，必须回复到，或者说回溯到上一个节点，尝试另一条路径。用这样的方法探索整棵树，直到找到获得成功解的路径。那么就可以根据这条路径作出第一步选择。

广度优先搜索

正如深度优先搜索中看到的，算法表现得好像要尽快地远离起始点似的。相反，在广度优先搜索中，算法好像要尽可能地靠近起始点。它首先访问起始顶点的所有邻接点，然后再访问较远的区域。这种搜索不能用栈，而要用队列来实现。

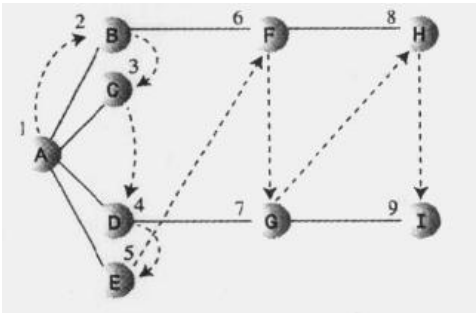


图 13.9 广度优先搜索

一个例子

图 13.9 中的图与图 13.5 中的相同，但是这里应用广度优先搜索策略。号码依旧表明访问顶点的顺序。

A 是起始点，所以访问它，并标记为当前顶点。然后应用下面几条规则：

规则 1

访问下一个未来访问的邻接点（如果存在），这个顶点必须是当前顶点的邻接点，标记它，并把它插入到队列中。

规则 2

如果因为已经没有未访问顶点而不能执行规则 1，那么从队列头取一个顶点（如果存在），并使其成为当前顶点。

规则 3

如果因为队列为空而不能执行规则 2，则搜索结束。

因此，需要首先访问所有与 A 邻接的顶点，并在访问的同时把它们插入到队列中。现在已经访问了 A、B、C、D 和 E。这时队列（从头到尾）包含 BCDE。

已经没有未访问的且与顶点 A 邻接的顶点了，所以从队列中取出 B，寻找那些和 B 邻接的顶点。这时找到 F，所以把 F 插入到队列中。已经没有未访问的且与顶点 B 邻接的顶点了，所以从队列中取出 C，它没有未访问的邻接点，因此取出 D 并访问 G。D 没有未访问的邻接点，所以取出 E。现在队列中有 FG。再取出 F，访问 H。然后取出 G，访问 I。

现在队列中有 HI，但当取出它们时，发现没有其他的未访问顶点，这时队列为空，所以从过程中退出。表 13.4 显示了这个过程。

表 13.4 广度优先搜索过程中的队列内容

事件	队列（队头到队尾）
VisitA	
VisitB	B
VisitC	BC
VisitD	BCD
VisitE	BCDE
RemoveB	CDE
VisitF	CDEF
RemoveC	DEF
RemoveD	EF
VisitG	EFG
RemoveE	FG
RemoveF	G
VisitH	GH
RemoveG	H
VisitI	HI
RemoveH	I
RemoveI	
Done	

在每一时刻，队列所包含的顶点是那些本身已经被访问，而它的邻居还有未被访问的顶点。（对比深度优先搜索，栈的内容是起始点到当前顶点经过的所有顶点。）顶点访问的顺序为 ABCDEFGHI。

GraphN 专题 Applet 和 BFS

在 GraphN 专题 Applet 中，点击 BFS 按钮就开始执行广度优先搜索。可以用图 13.9 所示的图来实验，也可以换用别的图。

注意广度优先搜索和深度优先搜索的异同点。

可以认为广度优先搜索就像往水中投入石块时，水波纹扩展的过程——对于喜爱流行病学的人来说，就好比流感细菌通过航空旅客从一个城市传播到另一个城市。首先是相距起始点只有一条边（飞机航线）的所有顶点被访问，然后是相距两条边的所有顶点被访问，依此类推。

Java 代码

Graph 类的 bfs()方法和 dfs()方法类似，只是用队列代替了栈，嵌套的循环代替了单层循环。外层循环等待队列为空，而内层循环依次寻找当前顶点的未访问邻接点。下面是代码：

```
public void bfs()                  // breadth-first search
   {                               // begin at vertex 0
   vertexList[0].wasVisited = true; // mark it
   displayVertex(0);               // display it
   theQueue.insert(0);             // insert at tail
   int v2;

   while( !theQueue.isEmpty() )    // until queue empty,
      {
      int v1 = theQueue.remove();  // remove vertex at head
      // until it has no unvisited neighbors
      while( (v2=getAdjUnvisitedVertex(v1)) != -1 )
         {                                 // get one,
         vertexList[v2].wasVisited = true; // mark it
         displayVertex(v2);                // display it
         theQueue.insert(v2);              // insert it
         }  // end while(unvisited neighbors)
      }  // end while(queue not empty)

   // queue is empty, so we're done
   for(int j=0; j<nVerts; j++)            // reset flags
      vertexList[j].wasVisited = false;
   }  // end bfs()
```

假定使用 dfs.java（图 13.8 所示）中的图，则 bfs.java 的输出结果为

```
Visits: ABDCE
```

bfs.java 程序

清单 13.2 所示的 bfs.java 程序与 dfs.java 类似。只是用 Queue 类（根据第 4 章的版本修改）代替了 StackX 类，并且用 bfs()方法替代了 dfs()方法。

清单 13.2 bfs.java 程序

```
// bfs.java
// demonstrates breadth-first search
// to run this program: C>java BFSApp
////////////////////////////////////////////////////////////////
class Queue
   {
   private final int SIZE = 20;
   private int[] queArray;
   private int front;
   private int rear;
```

```
// --------------------------------------------------------
   public Queue()            // constructor
      {
      queArray = new int[SIZE];
      front = 0;
      rear = -1;
      }
// --------------------------------------------------------
   public void insert(int j) // put item at rear of queue
      {
      if(rear == SIZE-1)
         rear = -1;
      queArray[++rear] = j;
      }
// --------------------------------------------------------
   public int remove()       // take item from front of queue
      {
      int temp = queArray[front++];
      if(front == SIZE)
         front = 0;
      return temp;
      }
// --------------------------------------------------------
   public boolean isEmpty()  // true if queue is empty
      {
      return ( rear+1==front || (front+SIZE-1==rear) );
      }
// --------------------------------------------------------
   } // end class Queue
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // label (e.g. 'A')
   public boolean wasVisited;
// --------------------------------------------------------
   public Vertex(char lab)   // constructor
      {
      label = lab;
      wasVisited = false;
      }
// --------------------------------------------------------
   } // end class Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
```

```
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // list of vertices
   private int adjMat[][];      // adjacency matrix
   private int nVerts;          // current number of vertices
   private Queue theQueue;
// ----------------
   public Graph()               // constructor
      {
      vertexList = new Vertex[MAX_VERTS];
                                          // adjacency matrix
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // set adjacency
         for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
            adjMat[j][k] = 0;
      theQueue = new Queue();
      }  // end constructor
// ------------------------------------------------------------
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// ------------------------------------------------------------
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      adjMat[end][start] = 1;
      }
// ------------------------------------------------------------
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }
// ------------------------------------------------------------
   public void bfs()                  // breadth-first search
      {                               // begin at vertex 0
      vertexList[0].wasVisited = true; // mark it
      displayVertex(0);               // display it
      theQueue.insert(0);             // insert at tail
      int v2;

      while( !theQueue.isEmpty() )    // until queue empty,
         {
         int v1 = theQueue.remove();  // remove vertex at head
         // until it has no unvisited neighbors
```

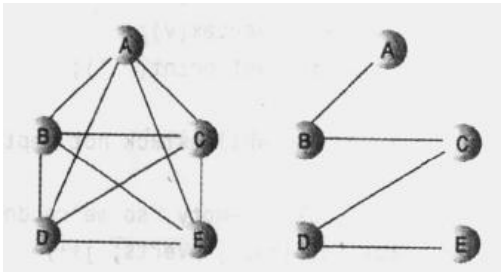
```
        while( (v2=getAdjUnvisitedVertex(v1)) != -1 )
           {                                  // get one,
           vertexList[v2].wasVisited = true;  // mark it
           displayVertex(v2);                 // display it
           theQueue.insert(v2);               // insert it
           }  // end while
        }  // end while(queue not empty)

     // queue is empty, so we're done
     for(int j=0; j<nVerts; j++)             // reset flags
        vertexList[j].wasVisited = false;
     }  // end bfs()
// -------------------------------------------------------------
   // returns an unvisited vertex adj to v
   public int getAdjUnvisitedVertex(int v)
     {
     for(int j=0; j<nVerts; j++)
        if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
           return j;
     return -1;
     }  // end getAdjUnvisitedVertex()
// -------------------------------------------------------------
   }  // end class Graph
////////////////////////////////////////////////////////////////
class BFSApp
   {
   public static void main(String[] args)
     {
     Graph theGraph = new Graph();
     theGraph.addVertex('A');    // 0  (start for dfs)
     theGraph.addVertex('B');    // 1
     theGraph.addVertex('C');    // 2
     theGraph.addVertex('D');    // 3
     theGraph.addVertex('E');    // 4

     theGraph.addEdge(0, 1);     // AB
     theGraph.addEdge(1, 2);     // BC
     theGraph.addEdge(0, 3);     // AD
     theGraph.addEdge(3, 4);     // DE

     System.out.print("Visits: ");
     theGraph.bfs();             // breadth-first search
     System.out.println();
     }  // end main()
   }  // end class BFSApp
////////////////////////////////////////////////////////////////
```

广度优先搜索有一个有趣的属性：它首先找到与起始点相距一条边的所有顶点，然后是与起始点相距两条边的顶点，依此类推。如果要寻找起始顶点到指定顶点的最短距离，那么这个属性非常有用。首先执行 BFS，当找到指定顶点时，就可以说这条路径是到这个顶点的最短路径。如果有更短的路径，BFS 算法就应该已经找到过它了。

最小生成树

假设我们设计了一个如图 13.4 所示的印刷电路板，可能需要确定是否使用了最少的连线。也就是说，针脚之间不要有多余的连接；多余的连接必定占用多余的空间，使布线变得更加困难。

对于任何针脚和连线的连通部分（用图的术语说就是顶点和边），如果有一种算法可以去掉多余的连线，那真是太好了。执行这种算法的结果就是产生了一种图，它用最少的边连接了所有的顶点。例如，图 13.10a 显示了五个点，它们之间有许多多余的边，而图 13.10b 也显示了五个点，它们之间只有最少数量的边保证它们彼此连通。这组成了最小生成树（MST）。



a）多余边　　b）最少数量的边

图 13.10　最小生成树

对给定的一组顶点，可能有很多种最小生成树。图 13.10b 显示了边 AB、BC、CD 和 DE 组成的最小生成树，而边 AC、CE、ED 和 DB 同样组成最小生成树。注意，最小生成树边 E 的数量总比顶点 V 的数量小 1：

E = V – 1

记住，不必关心边的长度。并不需要找到一条最短路径，而是要找最少数量的边。（在下一章讨论带权图时，这点就会发生改变。）

创建最小生成树的算法与搜索的算法几乎是相同的。它同样可以基于广度优先搜索或深度优先搜索。本例中使用深度优先搜索。

在执行深度优先搜索过程中，记录走过的边，就可以创建一棵最小生成树，这可能会使人感到有些奇怪。下面要看到的最小生成树算法 mst()和前面看到的深度优先搜索算法 dfs()之间的惟一区别就是 mst()方法必须记录走过的边。

GraphN 专题 applet

重复点击 GraphN 专题程序中的 Tree 按钮，可以创建任意一个图的最小生成树。用不同的图来生成各自的最小生成树。你会看到算法与使用 DFS 按钮执行搜索时的步骤完全一样。然而，当使用 Tree 按钮时，如果算法把某条边算作最小生成树的一部分，那么它就变黑。算法结束时，applet 去掉所有未变黑的线，只留下最小生成树。最后如果希望下次继续使用原图，点击按钮保存它。

最小生成树的 Java 代码

下面是 mst()方法的代码：

```
while( !theStack.isEmpty() )          // until stack empty
   {                                  // get stack top
```

```
    int currentVertex = theStack.peek();
    // get next unvisited neighbor
    int v = getAdjUnvisitedVertex(currentVertex);
    if(v == -1)                     // if no more neighbors
       theStack.pop();              //    pop it away
    else                            // got a neighbor
       {
       vertexList[v].wasVisited = true;  // mark it
       theStack.push(v);                 // push it
                                    // display edge
       displayVertex(currentVertex);     // from currentV
       displayVertex(v);                 // to v
       System.out.print(" ");
       }
    } // end while(stack not empty)

    // stack is empty, so we're done
    for(int j=0; j<nVerts; j++)          // reset flags
       vertexList[j].wasVisited = false;
} // end mst()
```

正如刚才看到的，这段代码与dfs()方法非常类似。然而，在else语句中，当前顶点与它下一个未访问邻接点被显示。这两个顶点决定了一条边，沿着这条边，算法将要访问下一个新的顶点。所有这样的边就组成了最小生成树。

在mst.java程序的main()方法中，用下面的语句创建一个图：

```
Graph theGraph = new Graph();
theGraph.addVertex('A');    // 0  (start for mst)
theGraph.addVertex('B');    // 1
theGraph.addVertex('C');    // 2
theGraph.addVertex('D');    // 3
theGraph.addVertex('E');    // 4

theGraph.addEdge(0, 1);     // AB
theGraph.addEdge(0, 2);     // AC
theGraph.addEdge(0, 3);     // AD
theGraph.addEdge(0, 4);     // AE
theGraph.addEdge(1, 2);     // BC
theGraph.addEdge(1, 3);     // BD
theGraph.addEdge(1, 4);     // BE
theGraph.addEdge(2, 3);     // CD
theGraph.addEdge(2, 4);     // CE
theGraph.addEdge(3, 4);     // DE
```

这个图的结果如图13.10a所示。当mst()方法运行完毕，只有4条边被保留，如图13.10b所示。下面是mst.java程序的输出：

```
Minimum spanning tree: AB BC CD DE
```

注意，这里仅是能够创建的众多最小生成树中的一种可能。例如，用不同的起始点，可以产生不同的树。所以可以在代码中作些小的调整，例如在 getAdjUnvisitedVertex()方法中从数组 vertexList[]的最大下标开始，而不是从 0 下标开始。

最小生成树比较容易从深度优先搜索得到，这是因为 DFS 访问所有顶点，但只访问一次。它绝不会两次访问同一个顶点。当它看到某条边将到达一个已访问的顶点，它就不会走这条边。它从来不遍历那些不可能的边。因此，DFS 算法走过整个图的路径必定是最小生成树。

mst.java 程序

清单 13.3 显示了 mst.java 程序。它与 dfs.java 程序类似，只是 main()中创建的图和 mst()方法不同。

清单 13.3　mst.java 程序

```
// mst.java
// demonstrates minimum spanning tree
// to run this program: C>java MSTApp
////////////////////////////////////////////////////////////////
class StackX
   {
   private final int SIZE = 20;
   private int[] st;
   private int top;
// ----------------------------------------------------------
   public StackX()            // constructor
      {
      st = new int[SIZE];     // make array
      top = -1;
      }
// ----------------------------------------------------------
   public void push(int j)    // put item on stack
      { st[++top] = j; }
// ----------------------------------------------------------
   public int pop()           // take item off stack
      { return st[top--]; }
// ----------------------------------------------------------
   public int peek()          // peek at top of stack
      { return st[top]; }
// ----------------------------------------------------------
   public boolean isEmpty()   // true if nothing on stack
      { return (top == -1); }
// ----------------------------------------------------------
   }  // end class StackX
////////////////////////////////////////////////////////////////
class Vertex
```

```
   {
   public char label;        // label (e.g. 'A')
   public boolean wasVisited;
// ------------------------------------------------------------
   public Vertex(char lab)   // constructor
      {
      label = lab;
      wasVisited = false;
      }
// ------------------------------------------------------------
   } // end class Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // list of vertices
   private int adjMat[][];      // adjacency matrix
   private int nVerts;          // current number of vertices
   private StackX theStack;
// ------------------------------------------------------------
   public Graph()               // constructor
      {
      vertexList = new Vertex[MAX_VERTS];
                                          // adjacency matrix
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // set adjacency
         for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
            adjMat[j][k] = 0;
      theStack = new StackX();
      } // end constructor
// ------------------------------------------------------------
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// ------------------------------------------------------------
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      adjMat[end][start] = 1;
      }
// ------------------------------------------------------------
   public void displayVertex(int v)
      {
```

```
      System.out.print(vertexList[v].label);
      }
// ------------------------------------------------------------
   public void mst()  // minimum spanning tree (depth first)
      {                                 // start at 0
      vertexList[0].wasVisited = true;  // mark it
      theStack.push(0);                 // push it

      while( !theStack.isEmpty() )      // until stack empty
         {                              // get stack top
         int currentVertex = theStack.peek();
         // get next unvisited neighbor
         int v = getAdjUnvisitedVertex(currentVertex);
         if(v == -1)                    // if no more neighbors
            theStack.pop();             //    pop it away
         else                           // got a neighbor
            {
            vertexList[v].wasVisited = true;  // mark it
            theStack.push(v);                 // push it
                                           // display edge
            displayVertex(currentVertex);     // from currentV
            displayVertex(v);                 // to v
            System.out.print(" ");
            }
         }  // end while(stack not empty)

         // stack is empty, so we're done
         for(int j=0; j<nVerts; j++)          // reset flags
            vertexList[j].wasVisited = false;
      }  // end tree
// ------------------------------------------------------------
   // returns an unvisited vertex adj to v
   public int getAdjUnvisitedVertex(int v)
      {
      for(int j=0; j<nVerts; j++)
         if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
            return j;
      return -1;
      }  // end getAdjUnvisitedVertex()
// ------------------------------------------------------------
   }  // end class Graph
////////////////////////////////////////////////////////////////
class MSTApp
   {
   public static void main(String[] args)
```

```
    {
    Graph theGraph = new Graph();
    theGraph.addVertex('A');    // 0  (start for mst)
    theGraph.addVertex('B');    // 1
    theGraph.addVertex('C');    // 2
    theGraph.addVertex('D');    // 3
    theGraph.addVertex('E');    // 4

    theGraph.addEdge(0, 1);     // AB
    theGraph.addEdge(0, 2);     // AC
    theGraph.addEdge(0, 3);     // AD
    theGraph.addEdge(0, 4);     // AE
    theGraph.addEdge(1, 2);     // BC
    theGraph.addEdge(1, 3);     // BD
    theGraph.addEdge(1, 4);     // BE
    theGraph.addEdge(2, 3);     // CD
    theGraph.addEdge(2, 4);     // CE
    theGraph.addEdge(3, 4);     // DE

    System.out.print("Minimum spanning tree: ");
    theGraph.mst();             // minimum spanning tree
    System.out.println();
    } // end main()
  } // end class MSTApp
////////////////////////////////////////////////////////////////
```

main()中的语句创建了一个类似五角星的图，它的每个顶点都有边与其他顶点相连接。下面是输出：

```
Minimum spanning tree: AB BC CD DE
```

有向图的拓扑排序

拓扑排序是可以用图模拟的另一种操作。它可用于表示一种情况，即某些项目或事件必须按特定的顺序排列或发生。下面看一个例子。

实例：课程的优先关系

在高中或大学中，学生们发现（他们会很沮丧）不能随心所欲地选择课程。一些课程有先修课程——这些课程必须先学。确实，修够必要的课程也是得到相关专业学位的“先决条件”。图 13.11 显示了要得到数学学位所必要的课程，这里它们的排列关系多少有些随意的成分。

为了得到学位，必须完成高级研讨会课程和（由于来自英语系的压力）比较文学课程。但是如果不选高等代数和解析几何，就不能修高级研讨会课程，同样不修英文写作也不能选比较文学课程。同时，要修几何，以便学习解析几何，以及修代数，以便学习高等代数和解析几何。

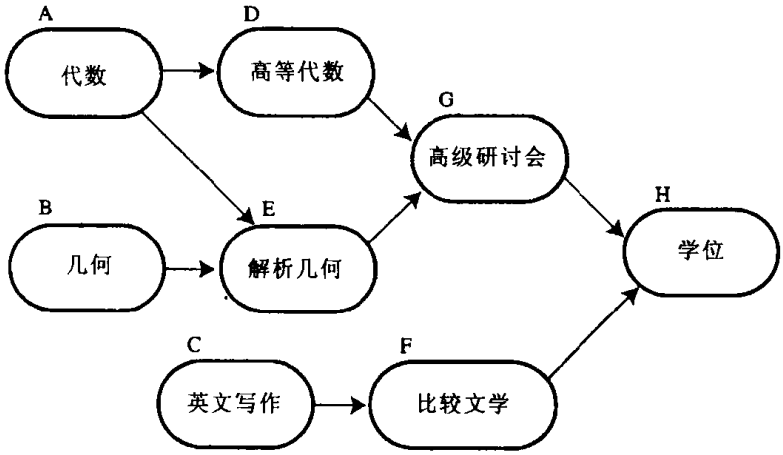


图 13.11 课程的优先关系

有向图

如图 13.11 所示，图可以表示这类关系。然而，图需要有一种前面没有涉及过的特性：边有方向。这时，图叫做有向图。在有向图中，只能沿着边指定的方向移动。图中的箭头表示了边的方向。

在程序中，有向图和无向图的区别是有向图的边在邻接矩阵中只有一项。图 13.12 显示了一个小的有向图；表 13.5 是它的邻接矩阵。

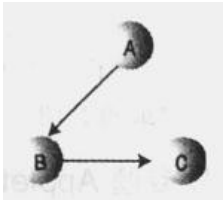


图 13.12 一个小的有向图

表 13.5 有向图的邻接矩阵

	A	B	C
A	0	1	0
B	0	0	1
C	0	0	0

1 代表一条边。行标表示边从哪里开始，列标表示边到哪里结束。因此，行 A 列 B 的值为 1 表示从 A 到 B 的边。如果边的方向是相反的，即从 B 到 A，那么行 B 列 A 的值就是 1。

对于前面讨论的无向图，邻接矩阵的上下三角是对称的，所以一半的信息是冗余的。而有向图的邻接矩阵中所有行列值都包含必要的信息。它的上下三角不是对称的。

对于有向图，增加边的方法只需要一条语句，

```
public void addEdge(int start, int end)  // directed graph
   {
   adjMat[start][end] = 1;
   }
```

而在无向图中要两条语句。

如果用邻接表表示图，那么 A 在它的链表中有 B，但不同于无向图的是，B 的链表中不包含 A。

拓扑排序

假设存在一个课程列表，包含了要得到学位必修的所有课程，就像图 13.11 中显示的课程一样。下面按课程的先后关系排列它们。得到学位是列表的最后一项，这就得到下面的序列：

```
BAEDGCFH
```

这种方式的排列，叫做为图进行拓扑排序。那些在某些课程前面学习的课程在列表中相应地排在前面。

实际上，许多可能的排序都符合课程的优先关系。例如可以先修 C 和 F：

```
CFBAEDGH
```

这也满足优先关系。而且还有许多其他的排序关系也符合。当用算法生成一个拓扑序列时，使用的方法和代码的细节决定了产生哪种拓扑序列。

除了课程优先关系以外，拓扑排序还可以对其他情况进行建模。工作进度是一个重要的例子。如果正在制造小汽车，就需要对工序进行排列，使得刹车片在安装车轮前安装，以及在上底盘前组装好发动机。汽车制造商用图来模拟制造过程中的上千个操作。这样可以保证每件事按恰当的顺序进行。

用图对工作进度建模叫做关键路径分析。尽管这里没有提到带权图，但是会用到它（下一章讨论）。带权图允许图带有时间的信息，这个时间是项目中完成不同任务所需的时间。图会指出完成整个工程的最短时间。

GraphD 专题 Applet

GraphD 专题 applet 模拟了有向图。这个 applet 的操作与 GraphN 基本相同，但是在每条边的尾端加了一个点，以表示边的方向。请注意：用拖拽鼠标的方式生成边时，拖拽的方向就是边的方向。图 13.13 显示了用 GraphD 专题 applet 模拟的课程优先关系图。

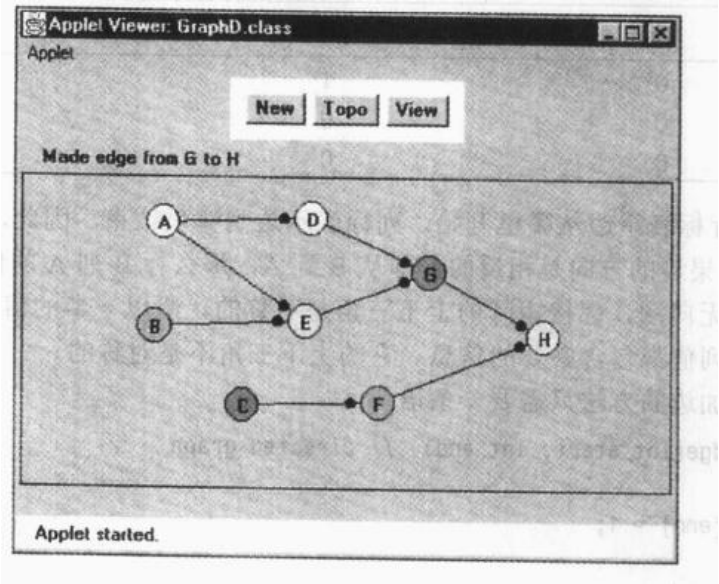


图 13.13 GraphD 专题 applet

拓扑排序算法的思想虽然不寻常但是很简单。有两个步骤是必需的：

步骤 1

找到一个没有后继的顶点。

顶点的后继也是一些顶点，它们是该节点的直接“下游”——即，该节点与它们由一条边相连，并且边的方向指向它们。如果有一条边从 A 指向 B，那么 B 是 A 的后继。图 13.11 中，惟一没有后继的顶点是 H。

步骤 2

从图中删除这个顶点，在列表的前面插入顶点的标记。

重复步骤 1 和步骤 2，直到所有顶点都从图中删除。这时，列表显示的顶点顺序就是拓扑排序的结果。

使用 GraphD applet 可以观看这个过程。构造一个图 13.11 中显示的图（或者其他形状的图），方法是双击产生顶点，拖拽产生边。然后反复点击 Topo 按钮。当有一个顶点被删除时，其标记就被放在图下面的列表开头。

删除顶点似乎是一个极端步骤，但是它是算法的核心。如果第一个顶点不处理，算法就不能计算出要处理的第二个顶点。如果需要，也可以在其他地方存储图的数据（顶点列表或邻接矩阵），然后在排序完成后恢复它们，GraphD applet 正是这么做的。

算法能够执行是因为，如果一个顶点没有后继，那么它肯定是拓扑序列中的最后一个。一旦删除它，剩下的顶点中必然有一个没有后继，所以它成为下一个拓扑序列中的最后一个，依此类推。

拓扑排序算法既可以用于连通图，也可以用于非连通图。这可以模拟另外一种有两个互不相关的目标的情况，例如同时得到数学学位和急救资格的证书。

环和树

有一种图是拓扑排序算法不能处理的，那就是有环图。什么是环？它是一条路径，路径的起点和终点都是同一个顶点。图 13.14 的路径 B-C-D-B 就形成一个环。（注意 A-B-C-A 不是环，因为不能从 C 到 A。）

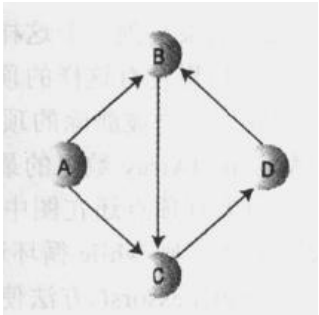


图 13.14 有环图

环模拟了令人左右为难的情况（这个情况是某些学生提出在现行制度中有矛盾的地方），这种情况下，课程 B 是课程 C 的先修课程，C 是 D 的先修课程，D 是 B 的先修课程。

不包含环的图叫做树。本书前面讨论的二叉树和多叉树就是这个意义上的树。然而在图中提出的树比二叉树和多叉树更具有一般意义，因为二叉树和多叉树定死了子节点的最大个数。在图中，树的顶点可以连接任意数量的顶点，只要不存在环即可。

要计算出无向图是否存在环也很简单。如果有 N 个顶点的图有超过 N-1 条边，那么它必定存在环。可以尝试着画一个没有环而有 N 个顶点，N 条边的图，这样就可以理解这个问题了。

拓扑排序必须在无环的有向图中进行。这样的图叫做有向无环图，缩写为 DAG。

Java 代码

下面是 topo()方法的代码，这个方法执行了拓扑排序：

```
public void topo()              // topological sort
   {
   int orig_nVerts = nVerts;  // remember how many verts

   while(nVerts > 0)          // while vertices remain,
      {
      // get a vertex with no successors, or -1
      int currentVertex = noSuccessors();
      if(currentVertex == -1)       // must be a cycle
         {
         System.out.println("ERROR: Graph has cycles");
         return;
         }
    // insert vertex label in sorted array (start at end)
    sortedArray[nVerts-1] = vertexList[currentVertex].label;

    deleteVertex(currentVertex);  // delete vertex
    }  // end while

 // vertices all gone; display sortedArray
 System.out.print("Topologically sorted order: ");
 for(int j=0; j<orig_nVerts; j++)
    System.out.print( sortedArray[j] );
 System.out.println("");
 }  // end topo
```

主要工作在 while 循环中进行。这个循环直到顶点数目为 0 时才退出。下面是步骤：

1. 调用 noSuccessors()找到任意一个没有后继的顶点。
2. 如果找到一个这样的顶点，把顶点放入数组 sortedArray[]，并且从图中删除顶点。
3. 如果没有这样的顶点，则图必然存在环。

最后一个被删除的顶点出现在列表的开头，所以，随着 nVerts（图中顶点个数）逐渐变小，顶点从 sortedArray 数组的最后开始，依次向前排列。

如果有顶点还在图中，但它们都有后继，那么图必然存在环，算法会显示一条信息并退出。如果没有环，则 while 循环退出，显示 sortedArray 数组中的数据，这时顶点是按拓扑有序排列。

noSuccessors()方法使用邻接矩阵找到没有后继的顶点。在外层 for 循环中，沿着每一行考察每个顶点。在每行中，用内层 for 循环扫描列，查找值为 1 的顶点。如果找到一个，就说明顶点有后继，因为从这个顶点到其他点有边存在。当找到一个 1 时，跳出内层循环，考察下一个顶点。

只有整个一行都没有 1 存在，才说明有一个顶点没有后继；这时，就返回它的行号。如果没有这样的顶点，就返回-1。下面是 noSuccessors()方法：

```
public int noSuccessors()  // returns vert with no successors
   {                       // (or -1 if no such verts)
 boolean isEdge;  // edge from row to column in adjMat
```

```
for(int row=0; row<nVerts; row++)  // for each vertex,
   {
   isEdge = false;                 // check edges
   for(int col=0; col<nVerts; col++)
      {
      if( adjMat[row][col] > 0 )   // if edge to
         {                         // another,
         isEdge = true;
         break;                    // this vertex
         }                         //    has a successor
      }                            //    try another
   if( !isEdge )                   // if no edges,
      return row;                  //    has no successors
   }
return -1;                         // no such vertex
} // end noSuccessors()
```

除了一些细节外，删除一个顶点很简单。顶点从 vertexList[]数组删除，后面的顶点向前移动填补空位。同样的，顶点的行列从邻接矩阵中删除，下面的行和右面的列移动来填补空位。这些任务由 deleteVertex()、moveRowUp()和 moveColLeft()方法来完成。这些方法将在 topo.java 程序的完整代码（清单 13.4）中看到。对于这个算法，用邻接表表示图效率更高，但要使用更多空间。

main()例程调用一些方法创建如图 13.10 所示的图，这些方法与前面看到的类似。应该注意到，addEdge()方法只在邻接矩阵中插入一个数，因为这是有向图。下面是 main()方法的代码：

```
public static void main(String[] args)
   {
   Graph theGraph = new Graph();
   theGraph.addVertex('A');    // 0
   theGraph.addVertex('B');    // 1
   theGraph.addVertex('C');    // 2
   theGraph.addVertex('D');    // 3
   theGraph.addVertex('E');    // 4
   theGraph.addVertex('F');    // 5
   theGraph.addVertex('G');    // 6
   theGraph.addVertex('H');    // 7

   theGraph.addEdge(0, 3);     // AD
   theGraph.addEdge(0, 4);     // AE
   theGraph.addEdge(1, 4);     // BE
   theGraph.addEdge(2, 5);     // CF
   theGraph.addEdge(3, 6);     // DG
   theGraph.addEdge(4, 6);     // EG
   theGraph.addEdge(5, 7);     // FH
   theGraph.addEdge(6, 7);     // GH
```

```
theGraph.topo();            // do the sort
} // end main()
```

创建图后，main()方法调用 topo()方法对图排序，并显示结果。下面是输出：

```
Topologically sorted order: BAEDGCFH
```

当然，可以重写 main()方法生成其他图。

完整的 topo.java 程序

现在已经看到了 topo.java 程序的大多数例程。清单 13.4 显示了完整的程序。

清单 13.4 topo.java 程序

```
// topo.java
// demonstrates topological sorting
// to run this program: C>java TopoApp
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // label (e.g. 'A')
// ----------------------------------------------------------
   public Vertex(char lab)   // constructor
      { label = lab; }
   } // end class Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // list of vertices
   private int adjMat[][];      // adjacency matrix
   private int nVerts;          // current number of vertices
   private char sortedArray[];
// ----------------------------------------------------------
   public Graph()               // constructor
      {
      vertexList = new Vertex[MAX_VERTS];
                                          // adjacency matrix
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // set adjacency
         for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
            adjMat[j][k] = 0;
      sortedArray = new char[MAX_VERTS];  // sorted vert labels
      } // end constructor
// ----------------------------------------------------------
   public void addVertex(char lab)
      {
```

```
      vertexList[nVerts++] = new Vertex(lab);
      }
// ------------------------------------------------------------
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      }
// ------------------------------------------------------------
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }
// ------------------------------------------------------------
   public void topo()  // topological sort
      {
      int orig_nVerts = nVerts;  // remember how many verts

      while(nVerts > 0)  // while vertices remain,
         {
         // get a vertex with no successors, or -1
         int currentVertex = noSuccessors();
         if(currentVertex == -1)       // must be a cycle
            {
            System.out.println("ERROR: Graph has cycles");
            return;
            }
         // insert vertex label in sorted array (start at end)
         sortedArray[nVerts-1] = vertexList[currentVertex].label;

         deleteVertex(currentVertex);  // delete vertex
         }  // end while

      // vertices all gone; display sortedArray
      System.out.print("Topologically sorted order: ");
      for(int j=0; j<orig_nVerts; j++)
         System.out.print( sortedArray[j] );
      System.out.println("");
      }  // end topo
// ------------------------------------------------------------
   public int noSuccessors()  // returns vert with no successors
      {                       // (or -1 if no such verts)
      boolean isEdge;  // edge from row to column in adjMat

      for(int row=0; row<nVerts; row++)  // for each vertex,
         {
```

```
        isEdge = false;                 // check edges
        for(int col=0; col<nVerts; col++)
           {
           if( adjMat[row][col] > 0 )   // if edge to
              {                         // another,
              isEdge = true;
              break;                    // this vertex
              }                         //    has a successor
           }                            //    try another
        if( !isEdge )                   // if no edges,
           return row;                  //    has no successors
        }
     return -1;                         // no such vertex
     } // end noSuccessors()
// ------------------------------------------------------------
   public void deleteVertex(int delVert)
      {
      if(delVert != nVerts-1)       // if not last vertex,
        {                           // delete from vertexList
        for(int j=delVert; j<nVerts-1; j++)
           vertexList[j] = vertexList[j+1];
                                    // delete row from adjMat
        for(int row=delVert; row<nVerts-1; row++)
           moveRowUp(row, nVerts);
                                    // delete col from adjMat
        for(int col=delVert; col<nVerts-1; col++)
           moveColLeft(col, nVerts-1);
        }
     nVerts--;                      // one less vertex
     } // end deleteVertex
// ------------------------------------------------------------
   private void moveRowUp(int row, int length)
      {
      for(int col=0; col<length; col++)
         adjMat[row][col] = adjMat[row+1][col];
      }
// ------------------------------------------------------------
   private void moveColLeft(int col, int length)
      {
      for(int row=0; row<length; row++)
         adjMat[row][col] = adjMat[row][col+1];
      }
// ------------------------------------------------------------
   } // end class Graph
////////////////////////////////////////////////////////////////
```

```
class TopoApp
   {
   public static void main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A');    // 0
      theGraph.addVertex('B');    // 1
      theGraph.addVertex('C');    // 2
      theGraph.addVertex('D');    // 3
      theGraph.addVertex('E');    // 4
      theGraph.addVertex('F');    // 5
      theGraph.addVertex('G');    // 6
      theGraph.addVertex('H');    // 7
      theGraph.addEdge(0, 3);     // AD
      theGraph.addEdge(0, 4);     // AE
      theGraph.addEdge(1, 4);     // BE
      theGraph.addEdge(2, 5);     // CF
      theGraph.addEdge(3, 6);     // DG
      theGraph.addEdge(4, 6);     // EG
      theGraph.addEdge(5, 7);     // FH
      theGraph.addEdge(6, 7);     // GH

      theGraph.topo();            // do the sort
      }  // end main()
   }  // end class TopoApp
//////////////////////////////////////////////////////////////////
```

在下一章，我们将会看到当边既有方向又有权值时会发生什么（情况）。

有向图的连通性

前面已经看到在无向图中，如何利用深度优先搜索或广度优先搜索找到所有相互连通的点。当试图找到有向图中所有的连通点时，事情变得更加复杂了。因为不能从任意一个顶点开始，并期望找到所有其他的连通的顶点。

考虑图 13.15 中的图。如果从 A 开始，可以到达 C，但是不能到达其他任何顶点。如果从 B 开始，就不能到达 D，如果从 C 开始，更不能到达其他顶点。关于连通性的问题是：如果从一个指定顶点出发，能够到达哪些顶点？

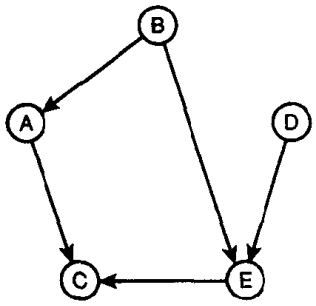


图 13.15 有向图

连通性表

我们可以很容易地改变 dfs.java 程序（清单 13.1），依次从每个顶点开始进行搜索。对于图 13.15 所示的图，输出如下所示：

```
AC
BACE
C
DEC
EC
```

这是有向图的连通性表，第一个字母是起始点，接下来的字母显示能够到达的顶点（直接或者通过其他顶点）。

Warshall 算法

在有些应用中，需要快速地找出是否一个顶点可以从其他顶点到达。例如想沿 Hubris 航线从 Athens 到 Murmansk，如果不考虑中间要转多少次站，那么可能到达吗？

可以检索连通性表，但是那要查看指定行的所有表项，需要 O(N)的时间（这里 N 是指定顶点可到达的顶点数目平均值）。如果时间紧迫，可否有更快的方式？

可以构造一个表，这个表将立即（即，O(1)的复杂度）告知一个顶点对另一个点是否是可达的。这样的表可以通过系统地修改图的邻接矩阵得到。由这种修正过的邻接矩阵表示的图，叫做原图的传递闭包。

记住，在原来的邻接矩阵中，行号代表边从哪里开始，列号代表边到哪里结束。（在连通表中排列是类似的。）行 C 列 D 的交叉点为 1，表示从顶点 C 到顶点 D 有一条边。可以一步从一个顶点到另一个顶点。（当然，在有向图中，从 D 到 C 不能通过这条从 C 到 D 的边。）表 13.6 显示了图 13.15 中所示的图的邻接矩阵。

表 13.6 邻接矩阵

	A	B	C	D	E
A	0	0	1	0	0
B	1	0	0	0	1
C	0	0	0	0	0
D	0	0	0	0	1
E	0	0	1	0	0

可以使用 Warshall 算法把邻接矩阵变成图的传递闭包。算法用有限的几行做了很多工作。它基于一个简单的思想：

如果能从顶点 L 到 M，并且能从顶点 M 到 N，那么可以从 L 到 N。

这里已经从两个一步路径得到一个两步路径。邻接矩阵显示了所有的一步路径，所以它可以很好地应用这个规则。

你可能想知道这个算法是否可以找到比两步还长的路径。毕竟规则只讨论了从两个一步路径合并成一个两步路径。在它执行时，算法建立在用前面发现的多步路径来创建任意长度的路径的基础上。将要描述的实现保证了这个结果，但是它的证明超出了本书的范围。

下面是它的实现过程。这里使用表 13.6 作为例子，要检查邻接矩阵的每个单元，一次一行。

行 A

从行 A 开始。列 A 和列 B 为 0，但列 C 为 1,所以在这里停下来。

现在，1 说明从 A 到 C 有一条路径。如果这时知道从某个顶点 X 到 A 有一条路径，那么就知道有条路径从顶点 X 到 C。以 A 为终点的边（如果存在）在哪里？它只可能在列 A 中出现：结果发现行 B 是 1。这说明从 B 到 A 有一条路径。所以就知道有一条边从 B 到 A，另一条（前面开始的那条）从 A 到 C。那么这就暗示可以通过两步从 B 到 C。通过图 13.15 可以检验这个过程的正确性。

为了记录这个结果，把行 B 和列 C 的交叉点置为 1。结果如图 13.16a 所示。

行 A 余下的单元为空。

行 B、C 和 D

下面来看行 B。在列 A，也就是第一个单元，值就是 1，表示有一条边从 B 到 A。有没有边是以 B 为结尾的？那么看列 B，发现它是空的，所以就知道，列 B 没有 1 存在，这样就没有发现更长的路径，因为没有以 B 为终点的边。

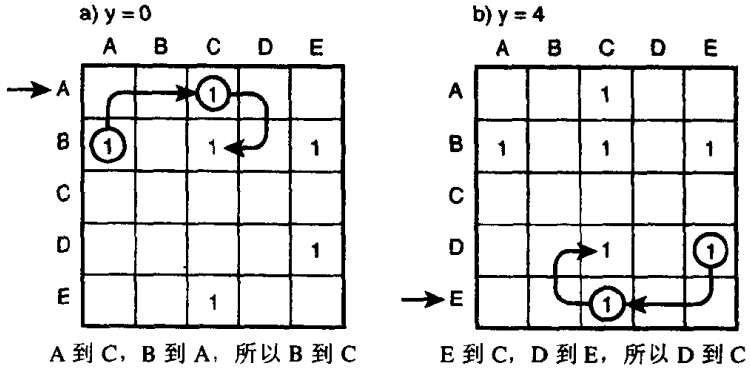


图 13.16 Warshall 算法步骤

行 C 没有 1 存在，所以直接到行 D。这时发现有一条边从 D 到 E。然而，列 D 是空的，所有没有其他边以 D 结尾。

行 E

在行 E，有一条边从 E 到 C。考察列 E，E 的第一个条目是从 B 到 E 的边，所以根据 B 到 E 和 E 到 C，可以知道有一条路径从 B 到 C。然而，它已经被发现了，所以相应的位置已经被置为 1。

列 E 还有一个 1，在行 D。这条从 D 到 E 的边加上从 E 到 C 的边说明有一条路径从 D 到 C，所以把相应单元置 1。结果如图 13.16b 所示。

Warshall 算法现在结束。由于向邻接矩阵增加了两个 1，现在邻接矩阵可以显示某个顶点通过任意步可以从另一个顶点到达。如果基于新矩阵画一个图，它就是图 13.15 所示的图的传递闭包。

Warshall 算法的实现

实现 Warshall 算法的一个方法是用三层嵌套的循环（就像 Sedgewick 建议的那样；参考附录 B “更进一步的阅读”）。外层循环考察每一行；称它为变量 y。它里面的一层循环考察行中的每个单元，它使用变量 x。如果在单元(x,y)发现 1，那么表明有一条边从 y 到 x，这时执行最里层循环，它使用变量 z。

第三个循环检查列 y 的每个单元，看是否有边以 y 为终点。（注意 y 在第一个循环中作为行，在这个循环中作为列。）如果行 z 列 y 值为 1，说明有一条边从 z 到 y。一条边从 z 到 y，另一条从 y 到 x，就可以说有一条路径从 z 到 x，所以把(z,x)置为 1。至于算法的细节将留作练习。

小 结

- 图包含由边连接的顶点。
- 图可以表示许多真实世界的情况，包括飞机航线、电子线路和工作调度。
- 搜索算法以一种系统的方式访问图中的每个顶点。搜索是其他行为的基础。
- 两个主要的搜索算法是深度优先搜索（DFS）和广度优先搜索（BFS）。
- 深度优先搜索通过栈实现；广度优先搜索通过队列实现。
- 最小生成树（MST）包含连接图中所有顶点所需要的最少数量的边。
- 在不带权的图中，简单修改深度优先搜索算法就可以生成它的最小生成树。
- 在有向图中，边有方向（通常用箭头表示）。
- 拓扑排序算法创建这样一个顶点的排列列表：如果从 A 到 B 有一条边，那么在列表中顶点 A 在顶点 B 的前面。
- 拓扑排序只能在 DAG 中执行，DAG 表示有向无环（没有环存在）图。
- 拓扑排序的典型应用是复杂项目的调度，它包含了任务和任务之间的前后关系。
- Warshall 算法能找到任意顶点和另外顶点之间是否有连接，不管是通过一步还是任意步到达。

问 题

下列问题作为读者的自测题。答案可见附录 C。

1. 在图中，一条_____连接两个_____。
2. 在无向图中，单凭邻接矩阵，如何知道图有多少条边？
3. 在游戏仿真中，什么图对应为了要移动而作出的选择？
4. 有向图的特点是
   a. 只能沿着最小生成树访问。
   b. 只能从顶点 A 到顶点 B 再到顶点 C，依此类推。
   c. 从一个顶点到另一个只能沿着一个方向。
   d. 对于给定的路径只能沿着一个方向行进。
5. 如果邻接矩阵为{0,1,0,0}，{1,0,1,1}，{0,1,0,0}，{0,1,0,0}，那么对应的邻接表是什么？
6. 最小生成树是这样的图
   a. 连接所有顶点的边的数量尽可能少。
   b. 边的数量和顶点的数量相同。
   c. 所有不必要的顶点都被删除。
   d. 用最少的边连接每两个顶点。
7. 在三个顶点和三条边的无向图中，有多少种最小生成树？
8. 下列哪一种情况下，无向图必定有环？
   a. 任何顶点都可以从其他顶点到达。
   b. 路径的数量大于顶点的数量。

c. 边的数量等于顶点的数量。

d. 路径的数量小于边的数量。

9. _____是无环图。

10. 无向图的最小生成树有环吗？

11. 判断题：某个给定的图可能有多种正确的拓扑排序序列。

12. 拓扑排序导致

a. 顶点排序后，边按同一个方向排列。

b. 顶点按边的递增编号排列。

c. 顶点按字母顺序排列。

d. 排在后面的顶点是前面顶点的下游。

13. 什么是 DAG？

14. 树有环吗？

15. 在 topo.java 程序（清单 13.4）中，用什么证据判断图有环？

实　验

完成这些实验可以帮助深入理解本章的主题。不需要编程实现。

1. 用 GraphN 专题 applet 画五个顶点、七条边的图。然后不用 View 按钮，记下图的邻接矩阵。做完后，按 View 按钮，查看结果是否一致。

2. tic-tac-toe 游戏一般在 3*3 的棋盘上进行，但是为了简便，考虑一个 2*2 的 tic-tac-toe 游戏，这时游戏者只需要 2 个 X 和 2 个 O 就可以胜出。用 GraphN applet 创建一个对应这个游戏的图。确实需要 4 层的图吗？

3. 创建一个有五个顶点的邻接矩阵，随机插入 0 和 1。不必考虑对称。现在，不使用 View 按钮，用 GraphD 专题 applet 创建对应的有向图。完成后，点击 View 按钮看图与邻接矩阵是否一致。

4. 在 GraphD 专题 applet 中，看是否能够创建一个带环的图，这样的图拓扑排序不能处理。

编程作业

编程作业有助于巩固对本章内容的理解，并展示如何应用本章的概念。（在“简介”中提到过，资深教师可以从出版者的网站上得到编程作业的完整答案。）

13.1　修改 bfs.java 程序（清单 13.2），通过广度优先搜索找到最小生成树，在 mst.java 程序（清单 13.3）中，这个工作是由深度优先搜索完成的。在 main()中，创建带有 9 个顶点和 12 条边的图，然后找到最小生成树。

13.2　修改 dfs.java 程序（清单 13.1），用邻接表而不是邻接矩阵执行深度优先搜索。可以通过修改第 5 章 linkList2.java 程序(清单 5.2)的 Link 类和 LinkList 类得到链表。修改 LinkList 中的 find()方法，搜索一个未访问的顶点，而不是一个关键字值。

13.3　修改 dfs.java 程序（清单 13.1）显示有向图的连通性表，就像“有向图的连通性”那节描述的一样。

13.4 实现 Warshall 算法来找到一个图的传递闭包。可以从编程作业 13.3 题开始。它有助于显示在算法的不同阶段显示邻接矩阵。

13.5 骑士旅行是一个古老而著名的象棋谜题。题目是在一个空的棋盘上移动一个骑士，从一个方块到另一个，直到踏遍了棋盘的所有的方块。写一个程序，用深度优先搜索解决这个问题。最好使棋盘的大小可变，这样可以在较小的棋盘上解决这个问题。8*8 的棋盘中，用个人电脑大概需要几年的时间解决这个问题。5*5 的棋盘只需要几分钟而已。

参考本章“深度优先搜索和游戏仿真”一节。容易想到当发生移动时，创建一个新的骑士，并放在新的方块中。如果这样，骑士对应一个顶点，而骑士的序列被放到栈中。当棋盘被骑士完全填满（栈满），任务就完成了。这个问题中，棋盘一般按顺序编号，左上角为 1，右下角为 64（或者在 5*5 的棋盘上为 1 到 25）。当查找下一步移动时，骑士不仅要按规则规定的方式走，还不能踏出棋盘或已经被占用（已经访问）的方块。如果写程序绘制棋盘并在每步移动后等待键盘输入，就能看到算法的过程，这时越来越多的骑士放在了棋盘上，当骑士无路可走时，删除一些骑士，尝试一些其他的移动。在下一章将讨论这个问题的复杂性。

第 14 章
带 权 图

本章重点

- 带权图的最小生成树
- 最短路径问题
- 每一对顶点之间的最短路径问题
- 效率
- 难题

在前一章已经看到了图的边可以有方向。本章中，将要探讨边的另一个特性：权值。例如，如果带权图的顶点代表城市，边的权可能代表城市间的距离，或者城市间的飞行费用，或者两城市间每年的汽车流量（高速公路建设者往往对这个感兴趣）。

当把权值当作边的特性时，一些有趣和复杂的问题也就出现了。什么是带权图的最小生成树？什么是两个顶点间的最短（或造价最低）距离？这些问题在现实世界中有重要的意义。

首先来看有权的无向图以及它的最小生成树。在本章后半部分，将讨论带有方向和权值的图，同时探讨著名的 Dijkstra 算法，并利用它寻找两点间的最短路径。

带权图的最小生成树

要引入带权图，首先回到最小生成树的问题。在有向图中创建这样一棵树比在无向图中要复杂一点。当所有的边拥有相同的权值，问题变得简单了（正如第 13 章中看到的），算法可以任意选择一条边加入最小生成树。但是当边具有不同的权值时，需要用一些算法策略来选择正确的边。

一个实例：丛林中的有线电视

假设要在一个虚拟的国家 Magnaguena 的 6 个城市之间架有线电视网，把它们都连接起来。五条边可以连接六个点，但是应该是哪五条边呢？连接每两个城市的造价不同，所以必须仔细选择线路使得总体造价最低。

图 14.1 显示了一个有 6 个顶点的带权图，代表 6 个城市，Ajo、Bordo、Colina、Danza、Erizo 和 Erizo。每条边有一个权值，标记在边的旁边。假设这些数字代表在两城市间架设电缆的造价，1 代表一百万 Magnaguenian 元。（注意，由于地形原因会导致距离太远，所以一些边是不必要的；例如，假设从 Ajo 到 Colina 或从 Danza 到 Flor 距离太远，所以这些边不予考虑，也没有出现在图中。）

选择哪些边架设电缆，能使得安装有线电视系统的造价最低呢？答案是利用最小生成树。它将有五条边（比城市的数量少 1），连接六个城市，并具有建立连接所需的最小代价。通过看图 14.1 中的图，能否直接找出这些边呢？如果不能，就需要用 GraphW 专题 applet 来解决问题。

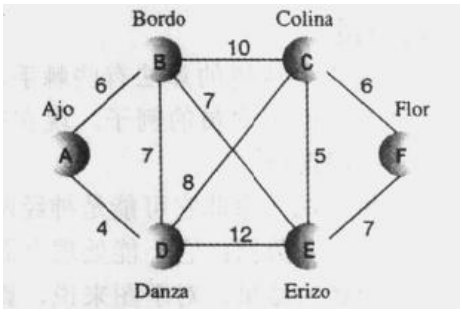


图 14.1　带权图

GraphW 专题 Applet

GraphW 专题 applet 类似于 GraphN 和 GraphD，但是它创建带权无向图。在用鼠标拖拽创建边之前，必须在右上角的输入框中输入边的权值。

这个 applet 只执行一个算法：反复点击 Tree 按钮，它就可以找到任意一个图的最小生成树。New 和 View 按钮像前面的 applet 一样，擦除掉旧图，显示邻接矩阵。

用这个 applet 创建一个小图并找到它的最小生成树。（由于设置的限制需要小心安置顶点位置，为的是权值不会彼此覆盖。）

执行算法时，会看到当顶点和边加入最小生成树时，顶点变成红色的圆圈，边也变粗了。在图的下面，左边显示了加入到树中的顶点。右边显示了优先级队列的内容。优先级队列中的每一项都是边。例如，AB6 表示从 A 到 B 的边，且权值为 6。展示算法后，我们会解释优先级队列的作用。

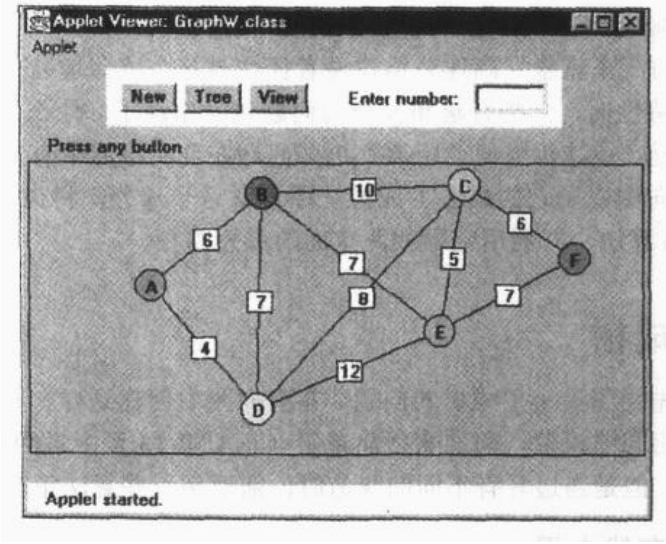


图 14.2 GraphW 专题 applet

现在，用 Tree 按钮一步一步执行算法，找到图的最小生成树。结果应该如图 14.3 所示。

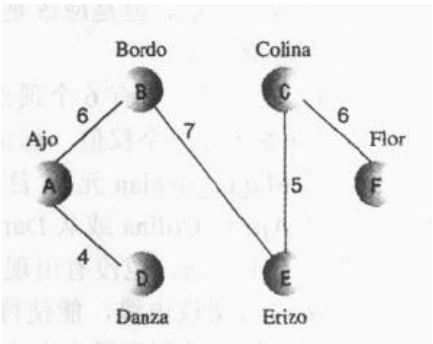


图 14.3 最小生成树

applet 应该发现最小生成树包含边 AD、AB、BE、EC 和 CF，总权值为 28。这些边的顺序是不重要的。如果从另一个顶点开始，还是会创建这样一棵树，但顺序就会不一样了。

派遣调查员

构造最小生成树的算法有些棘手，为了介绍这个算法先引入一个有线电视雇员的例子。现在有一个雇员（当然是经理）和一些调查员。

计算机算法（除非它可能是神经网络）不能一次“知道”给定问题的所有数据；它不能处理全面的大图，只能一点一点地得到数据，随着处理过程的深入，不断修改问题的结果。对于图来说，算法从某个顶点开始工作，首先得到这个点附近的数据，然后找到更远顶点的数据。在前面一章，深度优先搜索和广度优先搜索就是同样的例子。

同样，假设开始不知道在 Magnaguena 任意两个城市安装电缆需要多少钱。得到这些数据需要时间。这时就需要调查员。

从 Ajo 开始

现在从在 Ajo 建立办事处开始（可以在任意一个城市开始，但是 Ajo 有最好的餐馆）。从 Ajo 只能到两个城市：Bordo 和 Danza（参考图 14.1）。雇佣两个强壮且熟悉丛林环境的调查员，把他们派到危险的荒野中去，一个前往 Bordo，一个前往 Danza。这个工作是为了确定在这两条路线上安装有线电视的造价。

第一个调查员到达了 Bordo，完成了他的调查，他通知总部说在 Ajo 和 Bordo 之间安装电缆需要六百万。第二个调查员遇到了鳄鱼，晚些时候从 Danza 报告，Ajo 到 Danza 的电缆穿过更多的村庄，却只需要四百万。现在得到一个表单：

- Ajo-Danza, $4,000,000
- Ajo-Bordo, $6,000,000

然后按价值大小排序；很快将会看到为什么这是一个好方法。

建立 Ajo-Danza 的连接

这时，可以正式派出建筑队安装从 Ajo 到 Danza 的电缆。如何确定 Ajo-Danza 一线最终是造价最低方案（最小生成树）的一部分？因为迄今为止，只知道整个系统的两条连线。难道不需要其他的信息了吗？

为了对这个情况有一个直观感受，试想有其他的连接比当前从 Ajo 到 Danza 的连接更省钱。如果它不是直接连接 Danza，那么它必定是绕过 Bordo，然后迂回到 Danza，可能会穿过一个或更多的城市。但是现在已经知道到 Bordo 的连接需要六百万，比到 Danza 的连接（4 百万）更加昂贵。所以假设有其余可想像的路线更廉价，但是因为穿过了 Bordo 而导致那些路线的造价比与 Danza 直接连接还要高，并且经过 Bordo 到达环线上的其他城市比从 Danza 经过要昂贵。

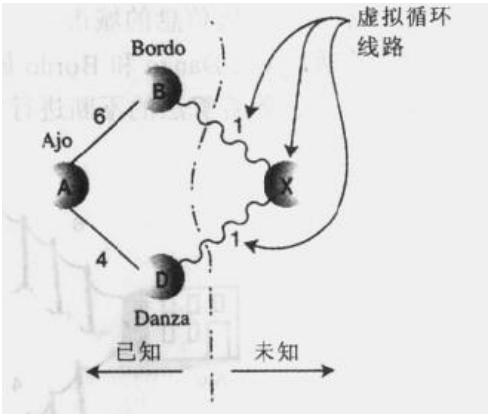


图 14.4 想像的环线

所以结论就是 Ajo-Danza 是最小生成树的一部分。这里虽没有进行形式化的证明（证明超出本书范围），但是它确实表明选择目前造价最低的路线是最好的作法。所以应该安装从 Ajo 到 Danza 的电缆，并在 Danza 建办事处。

为什么需要办事处？这是 Magnaguena 政府的要求，在从一个城市派出调查员到邻近的城市前，必须在这个城市建办事处。用图的术语说，在得到某个顶点与之相连的边的权值前，必须把顶点加入到树中。有办事处的城市都用电缆彼此连接，没有办事处的城市都还没有连接。

建立 Ajo-Bordo 的连接

现在已经完成了 Ajo-Danza 的连接，并且在 Danza 修建了办事处，可以从 Danza 派出调查员去能够到达的城市，分别是 Bordo、Colina 和 Erizo。调查员到达各自的终点，并报告三条连接的造价分别为 7 百万、8 百万和 1 千 2 百万。（当然，不需要派调查员去 Ajo，因为已经调查了它们之间的路线，并安装了电缆。）

现在已经知道了从有办事处的城市到没有办事处的城市的 4 条路线，分别是：

- Ajo-Bordo, $6,000,000
- Danza-Bordo, $7,000,000
- Danza-Colina, $8,000,000
- Danza-Erizo, $12,000,000

为什么 Ajo-Danza 这条线路不在表单中？因为那里已经安装了电缆；没有必要对它作进一步的考虑，所以已经安装的路线一律从表单中删除。

现在，接着做什么不是很明显。有许多潜在的路线可供选择。哪一条是最佳选择呢？下面是规则：

规则

在表单中，总是选择造价最低的边。

实际上，刚才在选择从 Ajo 出发的线路时，已经使用了这个规则；Ajo-Danza 是造价最低的。这时，造价最低的边是 Ajo-Bordo，所以花费 6 百万安装从 Ajo 到 Bordo 的电缆，并在 Bordo 建办事处。

在这里暂停一下，做个小结。在某个特定时间，电缆系统构造过程中，有三类城市：

1. 已经有办事处，并且用电缆连接的城市。（在图中，它们是最小生成树的顶点。）

2. 还没有连接，也没有办事处的城市，但是已知把它们连接到至少一个已有办事处的城市电缆的安装造价。这些城市叫做“边缘”城市。

3. 还不知道任何信息的城市。

这样的话，Ajo, Danza 和 Bordo 属于第一类，Colina 和 Erizo 属于第二类，而 Flor 属于第三类，如图 14.5 所示。随着算法的不断进行，城市从第三类变成第二类，再变成第一类。

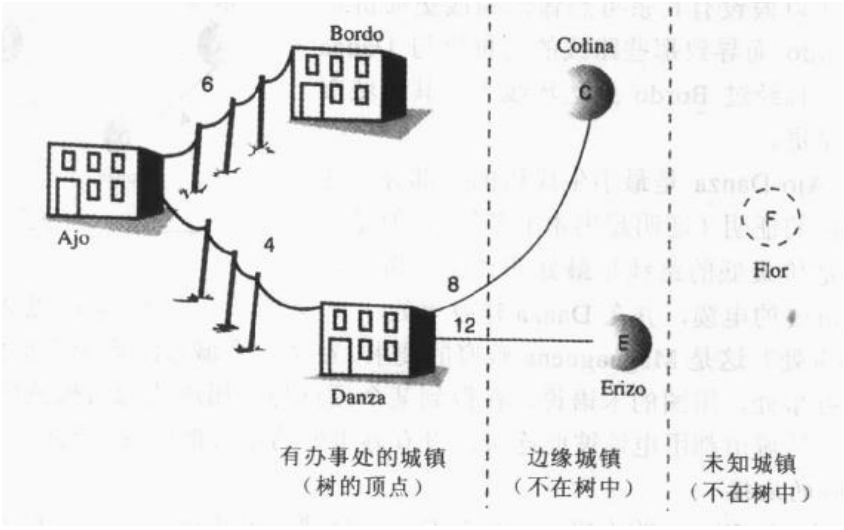


图 14.5 最小生成树算法的中间过程

建立 Bordo-Erizo 的连接

目前，Ajo、Danza 和 Bordo 已接入电缆系统，也有了办事处。现在已经知道从 Ajo 和 Danza 到第二类城市的造价，而不知道从 Bordo 到第二类城市的造价。所以需要从 Bordo 派遣调查员到

Colina 和 Erizo。他们报告说连接 Colina 需要 1 千万，连接 Erizo 需要 7 百万。下面是新的表单：

- Bordo-Erizo, $7,000,000
- Danza-Colina, $8,000,000
- Bordo-Colina, $10,000,000
- Danza-Erizo, $12,000,000

注意，Danza-Bordo 的连接原来在表单中，现在却没有了，因为没有必要考虑已连接的城市之间的连接问题，即使它们不是直接相连的。

从这个表单中可以看到，造价最低的路线是 Bordo-Erizo，需要 7 百万。所以修建了这段电缆，并在 Erizo 建立办事处（参考图 14.3）。

建立 Erizo-Colina 的连接

从 Erizo，调查员报告连接 Colina 需要 5 百万，连接 Flor 需要 7 百万。Danza-Erizo 的连接必须从表单中删除，因为 Erizo 现在是一个已连接的城市。新的表单是

- Erizo-Colina, $5,000,000
- Erizo-Flor, $7,000,000
- Danza-Colina, $8,000,000
- Bordo-Colina, 10,000,000

造价最低的连接是 Erizo-Colina，所以修建这条连接，并在 Colina 建立办事处。

最后建立 Colina-Flor 的连接

选择的范围现在变窄了，除去所有已经连接的城市，表单上只有两项：

- Colina-Flor, $6,000,000
- Erizo-Flor, $7,000,000

安装从 Colina 到 Flor 的电缆，在 Flor 建立办事处后，所有工作就完成了。由于每个城市都有了办事处，所以工作肯定完成了。建立的电缆路线是 Ajo-Danza、Ajo-Bordo、Bordo-Erizo、Erizo-Colina 和 Colina-Flor，如图 14.3 所示。这是连接 Magnaguena 六个城市造价最低的方案。

设计算法

这个想像中安装有线电视系统的例子，已经阐释了寻找带权图最小生成树算法的主要思想。现在看一下如何根据这个过程设计算法。

优先级队列

正如前面例子描述的那样，执行算法的关键行为是保存两城市间建立连接的造价表。通过选择造价最低的项，就可以决定下一条连接建在何处。

建议用优先级队列来实现这个用于反复选择最小造价值的表，而不用链表或数组。这是解决最小生成树的有效方式。在正式的程序中，优先级队列可能基于堆来实现，正如第 12 章“堆”所描述的。这会加快在较大的优先级队列中的操作。然而，在实例程序中，将用数组实现优先级队列。

算法要点

下面用图的术语（相对于有线电视的术语）重申一下算法：

从一个顶点开始，把它放入树的集合中。然后重复做下面的事情：

1. 找到从最新的顶点到其他顶点的所有边，这些顶点不能在树的集合中。把这些边放入优先级

队列。

2. 找出权值最小的边，把它和它所到达的顶点放入树的集合中。

重复这些步骤，直到所有顶点都在树的集合中。这时，工作完成。

在步骤1，“最新的”意味着最近放入树中的。此步骤的边可以在邻接矩阵中找到。步骤1完成后，表中包含了所有的边，这些边都是从树中顶点到它们的不在树中的邻接点（边缘点）的连接。

无用边

在表剩余条目中，想要把某些连接删除比较困难，它们都是从当前城市到已经拥有办事处的城市的连接。但是如果不做这个工作，就可能会导致安装不必要的有线电缆。

在程序的算法中，也要确保优先级队列中不能有连接已在树中的顶点的边。每次向树中增加顶点后，都要遍历优先级队列查找并删除这样的边。这样做了以后，要使优先级队列中在任意时刻只保持一条从树中顶点到某边缘点的边就变得容易了。也就是说，优先级队列中应该只包含一条到达某个第二类顶点的边。

下面将看到在GraphW专题applet中发生的事情。在优先级队列中的边比期望的要少——每个第二类顶点只有一个条目。一步一步地生成如图14.1所示的最小生成树，验证其中发生的事情。表14.1显示了优先级队列中，到目标地点重复的边是如何被删除的。

表14.1 边的剪除

步骤号码列表	没剪除的边列表	剪除的边列表（在优先级列列中）	优先级队列中因重复删掉的边
1	AB6,AD4	AB6,AD4	
2	DE12,DC8,DB7,AB6	DE12,DE8,AB6	DB7(AB6)
3	DE12,BC10,DC8,BE7	DC8,BE7	DE12(BE7),BC10(DC8)
4	BC10,DC8,EF7,EC5	EF7,EC5	BC10(EC5),DC8(EC5)
5	EF7,CF6	CF6	EF7

记住，一条边包含源（起始）点的字母，汇（结束）点的字母和权值。表中的第二列对应构造有线电视系统时的表。它显示了从第一类顶点（已在树中的顶点）到第二类顶点（至少有一条边与第一类顶点相连）的边。

第三列是运行GraphW applet时，优先级队列的变化情况。任何与其他边有相同终点的边，如果权值较大，就从队列中删除。

第四列显示了已经被删除的边，括号中是权值较小的边，它取代了权值较大的边，保留在队列中。记住，在步进执行的过程中，表中最后一项肯定要被删除，因为它已经加到树中。

在优先级队列中寻找目的地重复的边

如何保证以每个第二类顶点为目的地的边只有一条？每次向树中增加边的时候，一定要确保没有其他边也到达同样的顶点。如果有，只保留最小权值的边。

这使得在优先级队列中逐项查找成为必要的一步操作。查找的目的是看是否有这样的重复边。优先级队列设计初衷本不为随机访问的，所以这不是一个有效的方法。然而，在这种情况下，必须违反一下优先级队列的设计思想。

Java 代码

根据前面的算法要点，编制有向图最小生成树的方法 mstw()。正如其他图的程序一样，假定在 vertexList[]数组中有一个顶点列表，并且从下标为 0 的顶点开始。currentVert 代表最近加到树中的顶点。下面是 mstw()方法的代码：

```
public void mstw()            // minimum spanning tree
   {
   currentVert = 0;           // start at 0

   while(nTree < nVerts-1)    // while not all verts in tree
      {                       // put currentVert in tree
      vertexList[currentVert].isInTree = true;
      nTree++;

      // insert edges adjacent to currentVert into PQ
      for(int j=0; j<nVerts; j++)   // for each vertex,
         {
         if(j==currentVert)         // skip if it's us
            continue;
         if(vertexList[j].isInTree) // skip if in the tree
            continue;
         int distance = adjMat[currentVert][j];
         if( distance == INFINITY)  // skip if no edge
            continue;
         putInPQ(j, distance);      // put it in PQ (maybe)
         }
      if(thePQ.size()==0)           // no vertices in PQ?
            {
            System.out.println(" GRAPH NOT CONNECTED");
            return;
            }
      // remove edge with minimum distance, from PQ
      Edge theEdge = thePQ.removeMin();
      int sourceVert = theEdge.srcVert;
      currentVert = theEdge.destVert;

      // display edge from source to current
      System.out.print( vertexList[sourceVert].label );
      System.out.print( vertexList[currentVert].label );
      System.out.print(" ");
      }  // end while(not all verts in tree)

   // mst is complete
   for(int j=0; j<nVerts; j++)     // unmark vertices
      vertexList[j].isInTree = false;
   }  // end mstw()
```

算法在 while 循环中执行，循环结束条件是所有顶点都已在树中。循环中完成下面的操作：

1. 当前顶点放在树中。
2. 连接这个顶点的边放到优先级队列中（如果合适）。
3. 从优先级队列中删除权值最小的边。这条边的目的顶点变成当前顶点。

再看看这些步骤的细节。在步骤 1 中，通过标记 currentVert 所指顶点的 isInTree 字段来表示该顶点放入树中。

在步骤 2 中，连接这个顶点的边插入优先级队列。通过在邻接矩阵中扫描行号是 currentVert 的行寻找需要的边。只要下面任意一个条件为真，这条边就不能放入队列中：

- 源点和终点相同。
- 终点在树中。
- 源点和终点之间没有边（邻接矩阵中对应的值等于无限大）。

如果没有一个条件为真，调用 putInPQ()方法把这条边放入队列中。实际上，正如下面要看到的，这个例程并不总把边放到队列中。

在步骤 3 中，将最小权值的边从优先级队列中删除。把这条边和该边的终点加入树，并显示源点（currentVert）和终点。

在 mstw()方法最后，所有顶点的 isInTree 变量被重置，即从树中删除。在该程序中这样做，是因为根据这些数据只能创建一棵树。然而，在完成一项工作后，最好把数据恢复到原始的形态。

正如前面所强调的，在优先级队列中应该只含有一条到达某个特定目标顶点的边。putInPQ()方法保证了这一点。它调用 PriorityQ 类的 find()方法，这个方法经过修正，可以寻找到达指定点的边。如果不存在到达指定点的边，find()方法返回－1，这时 putInPQ()方法只要把新边插入优先级队列中即可。然而，如果有到达指定点的老边存在，putInPQ()方法就要检查老边是否比新边有更小的权值。如果老边的权值小，就不需要作什么变化。如果新边有更小的权值，就要把老边从队列中删除，把新边放进去。下面是 putInPQ()方法的代码：

```
public void putInPQ(int newVert, int newDist)
   {
   // is there another edge with the same destination vertex?
   int queueIndex = thePQ.find(newVert);  // got edge's index
   if(queueIndex != -1)                   // if there is one,
      {                                   // get edge
      Edge tempEdge = thePQ.peekN(queueIndex);
      int oldDist = tempEdge.distance;
      if(oldDist > newDist)               // if new edge shorter,
         {
         thePQ.removeN(queueIndex);  // remove old edge
         Edge theEdge = new Edge(currentVert, newVert, newDist);
         thePQ.insert(theEdge);      // insert new edge
         }
      // else no action; just leave the old vertex there
      }  // end if
   else  // no edge with same destination vertex
```

```
        {                               // so insert new one
        Edge theEdge = new Edge(currentVert, newVert, newDist);
        thePQ.insert(theEdge);
        }
    }  // end putInPQ()
```

mstw.java 程序

PriorityQ 类用数组存储对象。正如前面提到的，在处理大量数据的程序中，用堆存储对象比数组更合适。PriorityQ 类已经增加了不同的方法，如前所示，它用 find()方法找到到达指定顶点的边；用 peekN()方法得到任意一个数据项；用 removeN()方法删除任意一个数据项。程序的其余部分前面已经看到。清单 14.1 显示了完整的 mstw.java 程序。

清单 14.1　mstw.java 程序

```
// mstw.java
// demonstrates minimum spanning tree with weighted graphs
// to run this program: C>java MSTWApp
////////////////////////////////////////////////////////////////
class Edge
   {
   public int srcVert;   // index of a vertex starting edge
   public int destVert;  // index of a vertex ending edge
   public int distance;  // distance from src to dest
// ------------------------------------------------------------
   public Edge(int sv, int dv, int d)  // constructor
      {
      srcVert = sv;
      destVert = dv;
      distance = d;
      }
// ------------------------------------------------------------
   }  // end class Edge
////////////////////////////////////////////////////////////////
class PriorityQ
   {
   // array in sorted order, from max at 0 to min at size-1
   private final int SIZE = 20;
   private Edge[] queArray;
   private int size;
// ------------------------------------------------------------
   public PriorityQ()            // constructor
      {
      queArray = new Edge[SIZE];
      size = 0;
      }
```

```
// ------------------------------------------------------------
   public void insert(Edge item)  // insert item in sorted order
      {
      int j;

      for(j=0; j<size; j++)          // find place to insert
         if( item.distance >= queArray[j].distance )
            break;

      for(int k=size-1; k>=j; k--)    // move items up
         queArray[k+1] = queArray[k];

      queArray[j] = item;             // insert item
      size++;
      }
// ------------------------------------------------------------
   public Edge removeMin()            // remove minimum item
      { return queArray[--size]; }
// ------------------------------------------------------------
   public void removeN(int n)         // remove item at n
      {
      for(int j=n; j<size-1; j++)     // move items down
         queArray[j] = queArray[j+1];
      size--;
      }
// ------------------------------------------------------------
   public Edge peekMin()          // peek at minimum item
      { return queArray[size-1]; }
// ------------------------------------------------------------
   public int size()              // return number of items
      { return size; }
// ------------------------------------------------------------
   public boolean isEmpty()       // true if queue is empty
      { return (size==0); }
// ------------------------------------------------------------
   public Edge peekN(int n)       // peek at item n
      { return queArray[n]; }
// ------------------------------------------------------------
   public int find(int findDex)  // find item with specified
      {                          // destVert value
      for(int j=0; j<size; j++)
         if(queArray[j].destVert == findDex)
            return j;
      return -1;
      }
```

```
// --------------------------------------------------------
   } // end class PriorityQ
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // label (e.g. 'A')
   public boolean isInTree;
// --------------------------------------------------------
   public Vertex(char lab)   // constructor
      {
      label = lab;
      isInTree = false;
      }
// --------------------------------------------------------
   } // end class Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
   private final int INFINITY = 1000000;
   private Vertex vertexList[]; // list of vertices
   private int adjMat[][];      // adjacency matrix
   private int nVerts;          // current number of vertices
   private int currentVert;
   private PriorityQ thePQ;
   private int nTree;           // number of verts in tree
// --------------------------------------------------------
   public Graph()               // constructor
      {
      vertexList = new Vertex[MAX_VERTS];
                                          // adjacency matrix
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // set adjacency
         for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
            adjMat[j][k] = INFINITY;
      thePQ = new PriorityQ();
      } // end constructor
// --------------------------------------------------------
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// --------------------------------------------------------
   public void addEdge(int start, int end, int weight)
```

```
        {
        adjMat[start][end] = weight;
        adjMat[end][start] = weight;
        }
    // ---------------------------------------------------------
       public void displayVertex(int v)
        {
        System.out.print(vertexList[v].label);
        }
    // ---------------------------------------------------------
     public void mstw()              // minimum spanning tree
       {
       currentVert = 0;              // start at 0

       while(nTree < nVerts-1)       // while not all verts in tree
          {                          // put currentVert in tree
          vertexList[currentVert].isInTree = true;
          nTree++;

          // insert edges adjacent to currentVert into PQ
          for(int j=0; j<nVerts; j++)   // for each vertex,
             {
             if(j==currentVert)         // skip if it's us
                continue;
             if(vertexList[j].isInTree) // skip if in the tree
                continue;
             int distance = adjMat[currentVert][j];
             if( distance == INFINITY)  // skip if no edge
                continue;
             putInPQ(j, distance);      // put it in PQ (maybe)
             }
          if(thePQ.size()==0)           // no vertices in PQ?
                {
                System.out.println(" GRAPH NOT CONNECTED");
                return;
                }
          // remove edge with minimum distance, from PQ
          Edge theEdge = thePQ.removeMin();
          int sourceVert = theEdge.srcVert;
          currentVert = theEdge.destVert;

          // display edge from source to current
          System.out.print( vertexList[sourceVert].label );
          System.out.print( vertexList[currentVert].label );
          System.out.print(" ");
```

```
      } // end while(not all verts in tree)

   // mst is complete
   for(int j=0; j<nVerts; j++)     // unmark vertices
      vertexList[j].isInTree = false;
   } // end mstw
// -------------------------------------------------------------
   public void putInPQ(int newVert, int newDist)
      {
      // is there another edge with the same destination vertex?
      int queueIndex = thePQ.find(newVert);
      if(queueIndex != -1)              // got edge's index
         {
         Edge tempEdge = thePQ.peekN(queueIndex);  // get edge
         int oldDist = tempEdge.distance;
         if(oldDist > newDist)          // if new edge shorter,
            {
            thePQ.removeN(queueIndex);  // remove old edge
            Edge theEdge =
                        new Edge(currentVert, newVert, newDist);
            thePQ.insert(theEdge);      // insert new edge
            }
         // else no action; just leave the old vertex there
         } // end if
      else  // no edge with same destination vertex
         {                              // so insert new one
         Edge theEdge = new Edge(currentVert, newVert, newDist);
         thePQ.insert(theEdge);
         }
      } // end putInPQ()
// -------------------------------------------------------------
   } // end class Graph
////////////////////////////////////////////////////////////////
class MSTWApp
   {
   public static void main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A');    // 0  (start for mst)
      theGraph.addVertex('B');    // 1
      theGraph.addVertex('C');    // 2
      theGraph.addVertex('D');    // 3
      theGraph.addVertex('E');    // 4
      theGraph.addVertex('F');    // 5
```

```
        theGraph.addEdge(0, 1, 6);  // AB  6
        theGraph.addEdge(0, 3, 4);  // AD  4
        theGraph.addEdge(1, 2, 10); // BC 10
        theGraph.addEdge(1, 3, 7);  // BD  7
        theGraph.addEdge(1, 4, 7);  // BE  7
        theGraph.addEdge(2, 3, 8);  // CD  8
        theGraph.addEdge(2, 4, 5);  // CE  5
        theGraph.addEdge(2, 5, 6);  // CF  6
        theGraph.addEdge(3, 4, 12); // DE 12
        theGraph.addEdge(4, 5, 7);  // EF  7

        System.out.print("Minimum spanning tree: ");
        theGraph.mstw();            // minimum spanning tree
        System.out.println();
        } // end main()
    } // end class MSTWApp
////////////////////////////////////////////////////////////////
```

MSTWApp 类的 main()方法创建了一棵图 14.1 所示的树。下面是输出：

```
Minimum spanning tree: AD AB BE EC CF
```

最短路径问题

可能在带权图中最常遇到的问题就是，寻找两点间的最短路径问题。这个问题的解决方法可应用于现实生活中的很多情况，从印刷电路板的布局到项目调度都适用。但是它比前面见到的问题更复杂一些，所以首先还是来看一个真实世界的场景，它还发生在前面一节引入的虚拟的 Magnaguena 国。

铁路线

这次需要考虑铁路线而不是有线电视系统。然而，这个项目不像上一个那么浩大。这次并不是要修建铁路；铁路已经存在了。这次只是想找到从一个城市到另一个城市的费用最低的路线。

旅客在两个城市间搭乘火车需要固定的费用。图 14.6 显示了这些费用。也就是说，从 Ajo 到 Bordo 需要 50 元，从 Bordo 到 Danza 需要 90 元，等等。费用与两城市间的铁路长度无关（这不同于现在飞机费用的情况）。

图 14.6 的边是有向的。它们代表铁路是单向的，（为了安全）火车只允许朝一个方向开。例如，可以坐火车从 Ajo 到 Bordo，但是反过来不行。

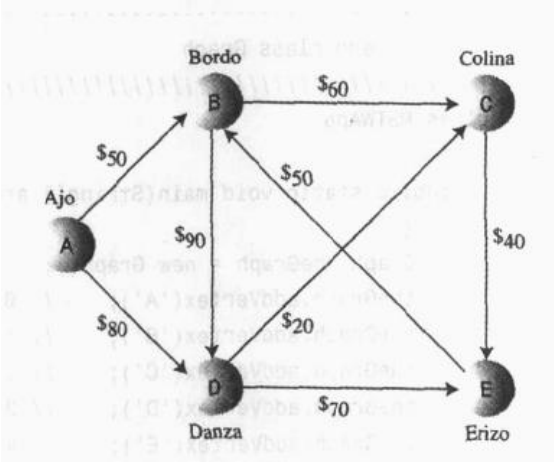


图 14.6 Magnaguena 的铁路费用

不管怎样，最关心的还是路费便宜。在图论中，这类问题叫做最短路径问题（SPP）。这里所说的最短并不总是指距离上的最短；它也可以代表最便宜，最快或最好的路径，或其他衡量标准。

最便宜的费用

任何两个城市间都有几条可能的路线。例如，为了从 Ajo 到达 Erizo，可以通过 Danza，或通过 Bordo 和 Colina，也可以通过 Danza 和 Colina，还可以选择其他的路线。（现在不可能通过火车到达 Flor，因为要经过崎岖的 Sierra Descaro 山脉，所以在图中没有显示。这是比较幸运的，因为缩小了要解决的问题的规模。）

最短路径问题是这样的：对一个给定的源点和终止点，走哪条路线费用最低？在图 14.6 中，能够看出（需要费些精力）从 Ajo 到 Erizo 费用最低的路线是通过 Danza 和 Colina；它只需要花费 140 元。

有向带权图

正如前面提到的，铁路线只有一个方向，所以火车在任何两个城市间只能朝一个方向行进。这相当于一个有向图。本来应该描绘一个更接近现实的情况，即乘客可以花同样的钱在两个城市间往返。那就相当一个无向图。然而最短路径问题在这两种情况下是类似的。所以为了体现多样性，我们来看这个问题在有向图中如何解决。

Dijkstra 算法

为解决最短路径问题而提出的方法叫做 Djikstra 算法，Edsger Dijkstra 在 1959 年首次解决了这个问题。这个算法的实现基于图的邻接矩阵表示法。让人感到有些惊奇的是它不仅能够找到任意两点间的最短路径，还可以找到某个指定点到其他所有顶点的最短路径。

代理人和铁路路线

为了了解 Dijkstra 算法如何工作，假设需要找到从 Ajo 到 Magnaguena 所有其他城市的费用最低的路线。这里要有扮演计算机程序的角色来执行 Dijkstra 算法（雇佣的代理人也是如此）。当然，现实中可以得到一本关于所有铁路费用的说明。然而算法在任意时刻只能看到一条信息，所以（如前面章节所示）假定不能直接看到整个图的样子。

在每个城市，站长会告知从该站到可以直达的其他城市的费用（即单段铁路，不包括穿过其他城市的铁路）。而且，他不能告知从该站跨过一个以上的城市到达另一个城市的费用。这里需要用到一个记事本，本子上为每个城市留一列的位置，并且希望每列的最后显示从源点到其他城市的最低费用路线。

第一个代理人：在 Ajo

最终，需要在每个城市放一个代理人；这个代理人的工作是保持到其他城市费用的信息。你自己是 Ajo 的代理人。

Ajo 的站长所能告诉你的是到 Bordo 需要 50 元，到 Danza 需要 80 元，把这些信息记在本子上。如表 14.2 所示。

表 14.2 第一步：Ajo 的代理人

从 Ajo 至→	Brodo	Colina	Danza	Erizo
Step1	50(via Ajo)	inf	80(via Ajo)	inf

表中“inf”是“无穷大”的缩写，意味着不能从 Ajo 到达表中每一列的列头所示的城市，或者至少目前为止不知道如何到达那里。（在算法中，用一个非常大的数代表无穷大。将会看到，这有助于进行计算。）圆括号中的内容显示了在到达不同的目的地之前，最后访问的城市。待会将看到为什么要有这个信息。现在将要做什么？下面是必须遵循的规则：

规则

总是把代理人派到下一个城市，从源点（Ajo）到这个城市的费用最小。

不必考虑那些已经有代理人的城市。注意，这个规则和最小生成树中问题（安装有线电视系统）的规则不同。在那个问题中，需要找到从已连接城市到未连接城市的造价最低连接（边）。在这个问题中，需要找到从 Ajo 到没有代理人的城市的最低费用路线。在这个研究项目的特殊情况下，两者做同样的事情，因为所有从 Ajo 出发的路线只包含一条边，但是，当把代理人派到更多城市时，从 Ajo 出发的路线会变成几条有向边的和。

第二个代理人：在 Bordo

从 Ajo 出发的费用最低路线是到 Bordo 的路线，为 50 元。所以雇佣一个路人前往 Bordo，在那里他成为代理人。当他到达那里时，他通过电话告知，Bordo 的站长告诉他从 Bordo 到 Colina 的费用是 60 元，到 Danza 的费用是 90 元。

经过简单计算即知，从 Ajo 经过 Bordo 到 Colina 的费用是 50 元加 60 元，即 110 元。所以修改笔记本上 Colina 的条目。同时可以看到，从 Ajo 经过 Bordo 到 Danza 必须花 50 元加 90 元，即 140 元。然而（这是关键部分）刚才已经知道从 Ajo 直接到 Danza 只需要 80 元。由于只关心从 Ajo 出发的最低费用路线，所以忽略这条较贵的路线，笔记本上的相应条目也不作修改。修改后的结果见表 14.3 的最后一行。图 14.7 显示了地理上的情况。

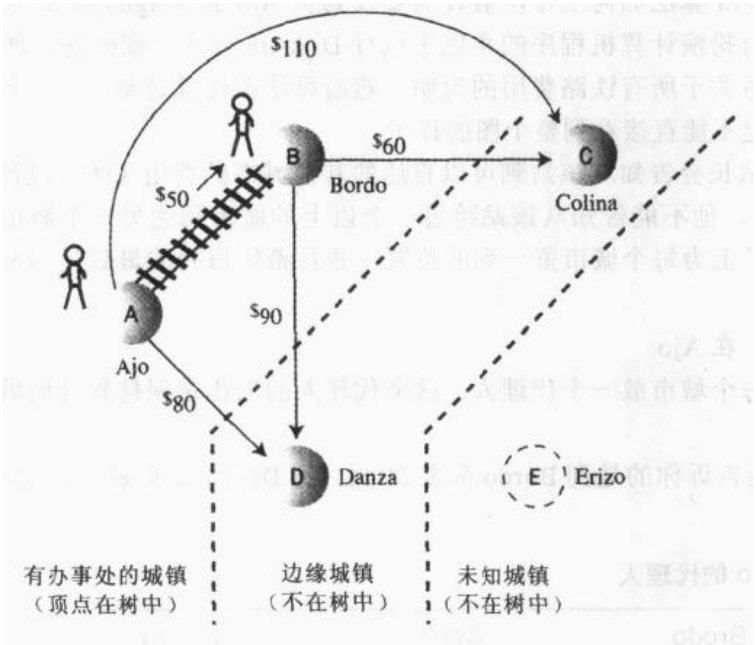


图 14.7 最短路径算法的第二步

表 14.3 第二步：在 Ajo 和 Bordo 的代理人

从 Ajo 至→	Bordo	Colina	Danza	Erizo
Step1	50(via Ajo)	inf	80(via Ajo)	inf
Step2	50(via Ajo)*	110(via Bordo)	80(via Ajo)	inf

在某个城市有代理人后，可以确定这个代理人走过的路线是费用最低的路线。为什么呢？考虑现在的情形。如果有另外一条线路比从 Ajo 到 Bordo 的直接连接更便宜，它需要通过其他的城市。但是从 Ajo 出发的另一条路线是到达 Danza，它已经比到 Bordo 的直接连接更贵了。加上从 Danza 到 Bordo 的费用，使得这条路线更加昂贵。

因此可以确定，从现在开始，不需要改动从 Ajo 到 Bordo 的最低费用。不管找到什么城市，这个费用都不会改变。在它旁边标注一个*，表示在这个城市有一个代理人，并且到它的最低费用是固定的。

三种城市

和最小生成树算法一样，需要把城市分成三种类型：

1. 已经有代理人的城市；它们已在树中。
2. 从有代理人的城市出发已知到达所需费用的城市；它们是边缘城市。
3. 未知城市。

现在，Ajo 和 Bordo 是第一类的城市，因为它们那里已经有代理人。第一类城市形成了一棵树，它包含了所有从源点出发，到达各个目的顶点的路径。（显然，这和最小生成树不同。）

还有一些城市没有代理人，但是已经知道了到它们那里的费用，因为在邻接的第一类城市中有代理人。比如现在知道从 Ajo 到 Danza 的费用是 80 元，从 Bordo 到 Colina 的费用是 60 元。因为这些费用是已知的，Danza 和 Colina 是第二类（边缘）城市。

现在还不知道关于 Erizo 的情况，它是“未知”顶点。图 14.7 显示了当前在算法中的分类情况。

和最小生成树算法一样，随着算法的执行，城市从未知类型，变到边缘类型，最后进入树中。

第三个代理人：在 Danza

现在，已经知道从 Ajo 到没有代理人的任意城市的最低费用是 80 元，它是从 Ajo 到 Danza 的直接线路。Ajo-Bordo-Colina 一线需要 110 元，Ajo-Bordo-Danza 一线需要 140 元，都比 Ajo-Danza 贵。

雇佣另外一个路人，把他派到 Danza。他报告从 Danza 花费 20 元可到 Colina，70 元可到 Erizo。现在可以修改笔记本上 Colina 的条目。之前，从 Ajo 转道到 Colina 需要 110 元。而如果转道从 Danza 到 Colina 只需要 100 元。而且可以知道从 Ajo，通过 Danza 到以前未知的城市 Erizo 的费用是 150 元。注意这些变化，如图 14.8 和表 14.4 所示。

表 14.4 第三步：在 Ajo、Bordo 和 Danza 的代理人

从 Ajo 至→	Bordo	Colina	Danza	Erizo
Step1	50(via Ajo)	inf	80(via Ajo)	inf
Step2	50(via Ajo)*	110(via Bordo)	80(via Ajo)	inf
Step3	50(via Ajo)*	100(via Danza)	80(via Ajo)*	150(via Danza)

第四个代理人：在 Colina

现在到达没有代理人的任意城市的最低费用是 100 元，即从 Ajo 穿过 Danza 到达 Colina。因此，派一个代理人沿这条路径到达 Colina。他报告从那到 Erizo 需要 40 元。现在能够计算出，既然 Ajo 到 Colina（通过 Danza）是 100 元，从 Colina 到 Erizo 是 40 元，那么从 Ajo 到 Erizo 的最低费用可以从 150 元（Ajo-Danza-Erizo 一线）降低到 140 元（Ajo-Danza-Colina-Erizo 一线）。由此更新笔记本的内容，如图 14.9 和表 14.5 所示。

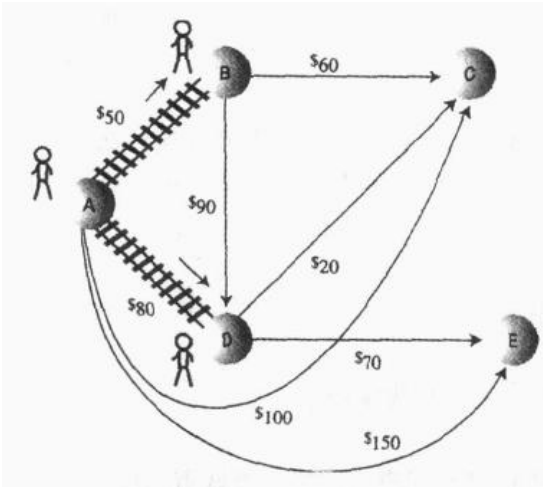


图 14.8 最短路径算法的第三步

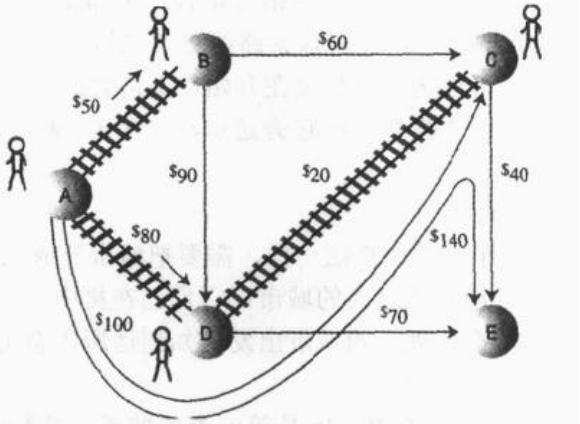


图 14.9 最短路径算法的第四步

表 14.5 第四步：在 Ajo、Bordo、Danza 和 Colina 的代理人

从 Ajo 至→	Bordo	Colina	Danza	Erizo
Step1	50(via Ajo)	inf	80(via Ajo)	inf
Step2	50(via Ajo)*	110(via Bordo)	80(via Ajo)	inf
Step3	50(via Ajo)*	100(via Danza)	80(via Ajo)*	150(via Danza)
Step4	50(via Ajo)*	100(via Danza)*	80(via Ajo)*	140(via Colina)

最后一个代理人：在 Erizo

现在到达没有代理人的任意城市的最低费用是 140 元，即从 Ajo 依次穿过 Danza 和 Colina 到达 Erizo。派一个代理人到 Erizo，但是，他报告说那里没有到达其他没有代理人的城市的线路。（有一条到 Bordo 的线路，但 Bordo 已经有代理人了。）表 14.6 显示了笔记本的最后一行。所做的全部工作就是在 Erizo 条目旁加星号，表示那里已经有代理人了。

表 14.6 第五步：在 Ajo、Bordo、Danza、Colina 和 Erizo 的代理人

从 Ajo 至→	Bordo	Colina	Danza	Erizo
Step1	50(via Ajo)	inf	80(via Ajo)	inf
Step2	50(via Ajo)*	110(via Bordo)	80(via Ajo)	inf
Step3	50(via Ajo)*	100(via Danza)	80(via Ajo)*	150(via Danza)
Step4	50(via Ajo)*	100(via Danza)*	80(via Ajo)*	140(via Colina)
Step5	50(via Ajo)*	100(via Danza)*	80(via Ajo)*	140(via Colina)*

当每个城市都有代理人后，就知道了 Ajo 到每个城市的费用。所以要求的工作就完成了。不再需要额外的计算，笔记本的最后一行显示了从 Ajo 到其他所有城市的费用最低路线。

这段描述说明了 Dijkstra 算法的本质。关键是

- 每次派一个代理人到新的城市，用这个代理人提供的新信息修正费用表单，在表中只保留从源点到某个城市现在已知的最低费用。
- 不断向某个新城市派代理人，条件是从源点到那个城市的路线费用最低（并不是从某个有代理人的城市出发的费用最低的路线，那是在最小生成树中用到的。）

使用 GraphDW 专题 Applet

下面看一下在 GraphDW（有向和带权的）专题 applet 中如何执行 Dijkstra 算法。用 applet 创建一个如图 14.6 所示的图。结果在图 14.10 中显示。（后面会看到如何得到出现在图下面的表。）这是一个有向带权图，所以要加一条边，必须在拖拽前键入一个数，然后从起点到终点，沿着正确的方向拖拽。

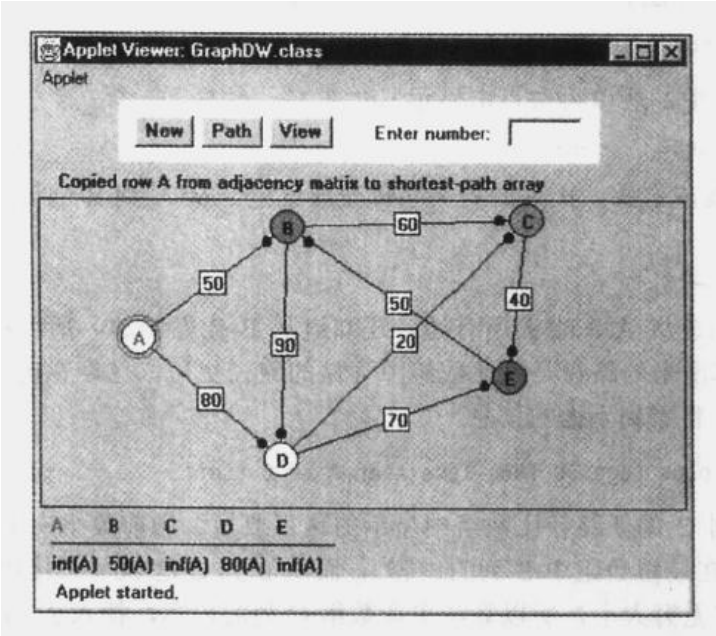


图 14.10 GraphDW 中铁路场景

当图完成后，点击 Path 按钮，提示出现后，点击顶点 A。再点击 Path 按钮，顶点 A 变成红色的空心圆圈，表示把 A 放入树中。

最短路径数组

再次点击 Path 按钮，图下面会出现一个表，如图 14.10 所示。对应的信息是“Copied row A from adjacency matrix to shortest-path array”。Dijkstra 算法首先把邻接矩阵中适当的行（即源点的行）复制到数组中。记住，在任何时候都可以点击 View 按钮查看邻接矩阵。）

这个数组叫做“最短路径”数组。在决定 Magnaguena 的费用最低火车线路中，笔记本上记载的最新一行内容对应着这个数组。这个数组将存储到其他顶点最短路径的最新情况，其他顶点都是终点。表 14.7 中每列的列头表示这些终点。

表 14.7 第一步：最短路径数组

A	B	C	D	E
inf(A)	50(A)	inf(A)	80(A)	inf(A)

在 applet 中，数组中最短路径值后面跟着一个括起来的“父顶点”。这个父节点是到达目的顶点之前到达的顶点。现在，父节点全是 A，因为只从 A 移动了一步。

如果费用未知（或无意义，例如从 A 到 A），它显示为无穷大，用“inf”代表，就像在记事本上写的一样。注意，如果顶点已经在树中，那么该列的开头字母为红色。这列的内容也不会再有变化。

最短距离

最初，算法知道从 A 点到那些直接连接的顶点的距离。只有 B 和 D 与 A 邻接，所以表中只显示了到 B 和 D 的距离。算法找到最小距离。再次点击 Path，显示如下信息

```
Minimum distance from A is 50, to vertex B
```

算法把顶点加入树中，所以再次点击显示

```
Added vertex B to tree
```

现在，图中的 B 变成红圈，B 列的列头字母变成红色。从 A 到 B 的边变得更黑，表明它是树的一部分。

最短路径数组的每一列

现在，算法不仅知道从 A 出发的所有边，还知道从 B 出发的边。所以在最短路径数组中按列查找，看用新信息计算出来的路径是否比数组中存储的路径更短。已经在树中的顶点（A 和 B）被跳过。首先检查 C 列。将看到下面的信息

```
To C: A to B (50) plus edge BC (60) less than A to C (inf)
```

算法已经发现，到 C 的新路径比数组中显示的路径要短。当前数组中 C 列显示的是无穷大。但是从 A 到 B 是 50（在最短数组中 B 列的内容），而从 B 到 C 是 60（邻接矩阵中行 B 列 C 的值为 60）。和为 110。110 比无穷大小，所以算法更新数组 C 列的内容，插入了 110。括号中的父顶点变成 B，因为这是到达 C 之前的最后一个顶点；B 是 C 的父节点。

接着，检查 D 列。将看到下面的信息

```
To D: A to B (50) plus edge BD (90) greater than or equal to A to D (80)
```

算法比较了原来显示的从 A 到 D 的距离（长度为 80，直接连接），和一条穿过 B 的可能路径（即 A-B-D）的距离。但是路径 A-B 是 50，路径 B-D 是 90，和为 140。这比 80 大，所以保持 80 不变。

对 E 列，显示的信息是

```
To E: A to B (50) plus edge BE (inf) greater than or equal to A to E (inf)
```

新计算的从 A 通过 B 到 E 的路径（50 加无穷大）仍然大于或等于数组中的当前值（无穷大），所以 E 列也没有改变。表 14.8 显示了新的最短路径数组内容。

表 14.8 第二步：最短路径数组

A	B	C	D	E
inf(A)	50(A)	110(B)	80(A)	inf(A)

现在可以更清楚地了解到显示在每个距离后面的父顶点的作用。每列显示从 A 到相应顶点的距离。父节点是终点沿此路径上的直接前趋。在 C 列，父顶点是 B，意味着从 A 到 C 的最短路径在到达 C 之前穿过 B。算法用这条信息选择合适的边放入树中。（当距离是无穷大时，父顶点是无意义的，如 A 列所示。）

新的最短距离

现在，最短路径数组已经更新了，算法找到数组中最短距离，正如点击 Path 按钮后看到的。显示的消息是

```
Minimum distance from A is 80, to vertex D
```

然后，消息

```
Added vertex D to tree
```

出现，新的顶点和边 AD 加入树中。

重复这个工作

现在，算法继续扫描最短路径数组，检查和更新从 A 到没有在树中的终点的距离；只有 C 和 E 仍然属于这一类。C 列和 E 列被更新。结果如表 14.9 所示。

表 14.9 第三步：最短路径数组

A	B	C	D	E
inf(A)	50(A)	100(D)	80(A)	150(D)

从 A 到一个非树中顶点的最短路径是 100，到顶点 C，所以 C 加入树中。

下次扫描最短路径数组，只需要考虑到 E 的距离。通过 C 的路径可以缩短这个距离，所以修改 E 列，如表 14.10 所示。

表 14.10 第四步：最短路径数组

A	B	C	D	E
inf(A)	50(A)	100(D)	80(A)	140(C)

现在，最后的顶点 E，也加入到树中，任务完成。最短路径数组显示了从 A 到所有其他顶点的最短距离。树包含了所有顶点和边 AB、AD、DC 和 CE，这些边用粗线表示。

可以根据到达任何顶点的最短路径，回溯地重构顶点序列。例如对于到 E 的最短路径，E 的父节点显示在数组 E 列的括号中，为 C。C 的前驱，重新扫描数组会发现是 D，然后 D 的前驱是 A。所以从 A 到 E 的最短路径是这样一条路径：A-D-C-E。

用 GraphDW 构建其他图来做实验，从小一点的图开始。经过实验一阵子，你就可以预测算法

下一步将如何走，这时就说明你已经逐步理解了 Dijkstra 算法。

Java 代码

最短路径算法的代码是本书中最复杂的代码之一，虽然如此，它并没有超出一般人的理解能力。首先看一个辅助类，然后是执行算法的主要方法，path()，最后是由 path()方法调用的两个方法，它们执行了特定的任务。

sPath 数组和 DistPar 类

正如前面看到的，最短路径算法的关键数据结构是一个数组，它保持了从源点到其他顶点（终点）的最短路径。在算法的执行过程中这个距离是变化的，直到最后，它存储了从源点开始的真正最短距离。在例子代码中，这个数组叫做 sPath[]（最短路径的缩写）。

正如前面看到的，不仅应该记录从源点到每个终点的最短路径，还应该记录走过的路径。幸运的是，不必要明确记录整个路径。只需要记录终点的父节点。父节点就是到达终点前到达的顶点。在专题 applet 中已经看到这一点，如果 100(D)出现在 C 列中，它意味着从 A 到 C 的费用最低路径是 100，且 D 是这条路径上到达 C 之前的最后一个顶点。

有几种方法保留父节点顶点的信息，但是，本例选用了把父节点和距离放在一起的方法，并把结果的对象放到 sPath[]数组中。这个对象类叫做 DistPar（距离－父节点）。

```
class DistPar             // distance and parent
   {                      // items stored in sPath array
   public int distance;   // distance from start to this vertex
   public int parentVert; // current parent of this vertex

   public DistPar(int pv, int d)  // constructor
      {
      distance = d;
      parentVert = pv;
      }
   }
```

path()方法

path()方法执行真正的最短路径算法。它使用 DistPar 类和 Vertex 类，这个类在 mstw.java 程序（清单 14.1）中见到过。path()方法是 Graph 类的一个成员，也会在 mstw.java 程序中见到，这个程序版本和以前的有所不同。

```
public void path()               // find all shortest paths
   {
   int startTree = 0;            // start at vertex 0
   vertexList[startTree].isInTree = true;
   nTree = 1;                    // put it in tree

   // transfer row of distances from adjMat to sPath
   for(int j=0; j<nVerts; j++)
      {
      int tempDist = adjMat[startTree][j];
```

```
      sPath[j] = new DistPar(startTree, tempDist);
      }

   // until all vertices are in the tree
   while(nTree < nVerts)
      {
      int indexMin = getMin();    // get minimum from sPath
      int minDist = sPath[indexMin].distance;

      if(minDist == INFINITY)     // if all infinite
         {                        // or in tree,
         System.out.println("There are unreachable vertices");
         break;                   // sPath is complete
         }
      else
         {                        // reset currentVert
         currentVert = indexMin;  // to closest vert
         startToCurrent = sPath[indexMin].distance;
         // minimum distance from startTree is
         // to currentVert, and is startToCurrent
         }
      // put current vertex in tree
      vertexList[currentVert].isInTree = true;
      nTree++;
      adjust_sPath();             // update sPath[] array
      }  // end while(nTree<nVerts)

   displayPaths();                // display sPath[] contents

   nTree = 0;                     // clear tree
   for(int j=0; j<nVerts; j++)
      vertexList[j].isInTree = false;
   }  // end path()
```

源点总在 vertexList[]数组下标为 0 的位置。path()方法的第一个任务是把这个顶点放入树中。算法执行过程中，将会把其他顶点也逐一放入树中。把顶点放入树中的操作是设置标志位，并把 nTree 变量增 1，这个变量记录了树中有多少个顶点。

第二，path()方法把邻接矩阵的对应行表达的距离的值复制到 sPath[]中。实际总是先从第 0 行复制，为了简单，假定源点的下标总为 0。最开始，所有 sPath[]数组中的父节点字段为 A，即源点。

现在进入算法的主 while 循环。等到所有的顶点都放入树中，这个循环就结束。在这个循环中有三个基本动作：

1. 选择 sPath[]数组中的最小距离。
2. 把对应的顶点（这个最小距离所在列的题头）放入树中，这个点变成“当前顶点”，currentVert。
3. 根据 currentVert 的变化，更新所有的 sPath[]数组内容。

如果 path()方法发现最小距离是无穷大，它就知道有些顶点从源点不能到达。为什么？因为不是所有的顶点都在树中（while 循环没有结束），尚无方法可以到达那些树外的顶点；如果有，就不会是无穷大的距离。

返回前，paht()方法调用 displayPaths()方法，显示 sPaht[]数组的最后内容。这是程序的惟一输出。而且 path()方法把 nTree 归 0，把所有顶点的 isInTree 标志位复位，这样就可以在其他算法中再次使用（尽管它们不在这个程序中）。

用 getMin()找到最小距离

为了找到 sPath[]数组中的最小距离，paht()方法调用 getMin()方法。整个例程比较简单。它扫描整个 sPath[]数组，返回最小距离项的列号（下标）。

```
public int getMin()                 // get entry from sPath
   {                                //    with minimum distance
   int minDist = INFINITY;          // assume large minimum
   int indexMin = 0;
   for(int j=1; j<nVerts; j++)      // for each vertex,
      {                             // if it's in tree and
      if( !vertexList[j].isInTree &&  // smaller than old one
                          sPath[j].distance < minDist )
        {
        minDist = sPath[j].distance;
        indexMin = j;               // update minimum
        }
     }  // end for
  return indexMin;                  // return index of minimum
  }  // end getMin()
```

正如在前面寻找最小生成树所做的那样，本应该用优先级队列作为最短路径算法的基础。但如果是那样的话，getMin()方法就不是必须的了；最小权值的边会自动出现在队头。然而，用数组的方法可以更清楚地显示实际进行的过程。

用 adjust_sPath()方法更新 sPath[]数组

从刚加入到树中的顶点获得新的信息，由 adjust_sPath()方法来更新 sPath[]数组。当例程被调用时，currentVert 刚被放入树中，startToCurrent 是 sPath[]数组中的当前项。adjust_sPath()方法现在检查 sPath[]数组中的每一项，它使用循环计数器 column 依次指向每个顶点。对于 sPath[]数组的每一项，如果对应顶点不在树中，它就做三件事：

1. 把到当前顶点的距离（已经计算过，并在 startToCurrent 变量中）加到从 currentVert 顶点到 column 顶点的距离上。结果叫做 startToFringe。

2. 把 startToFringe 与 sPath[]数组的当前项进行比较。

3. 如果 startToFringe 更小，就替换当前项。

这是 Dijkstra 算法的核心。它让 sPath[]数组总是存储从源到所有已知顶点的当前最短路径。下面是 adjust_sPath()方法的代码：

```
public void adjust_sPath()
   {
```

```
    // adjust values in shortest-path array sPath
    int column = 1;                 // skip starting vertex
    while(column < nVerts)          // go across columns
       {
       // if this column's vertex already in tree, skip it
       if( vertexList[column].isInTree )
          {
          column++;
          continue;
          }
       // calculate distance for one sPath entry
                    // get edge from currentVert to column
       int currentToFringe = adjMat[currentVert][column];
                    // add distance from start
       int startToFringe = startToCurrent + currentToFringe;
                    // get distance of current sPath entry
       int sPathDist = sPath[column].distance;

       // compare distance from start with sPath entry
       if(startToFringe < sPathDist)  // if shorter,
          {                           // update sPath
          sPath[column].parentVert = currentVert;
          sPath[column].distance = startToFringe;
          }
       column++;
       }  // end while(column < nVerts)
    }  // end adjust_sPath()
```

path.java 程序的 main()例程创建了一个如图 14.6 所示的图，并显示了它的最短路径数组。下面是代码：

```
    public static void main(String[] args)
       {
       Graph theGraph = new Graph();
       theGraph.addVertex('A');     // 0  (start)
       theGraph.addVertex('B');     // 1
       theGraph.addVertex('C');     // 2
       theGraph.addVertex('D');     // 3
       theGraph.addVertex('E');     // 4

       theGraph.addEdge(0, 1, 50);  // AB 50
       theGraph.addEdge(0, 3, 80);  // AD 80
       theGraph.addEdge(1, 2, 60);  // BC 60
       theGraph.addEdge(1, 3, 90);  // BD 90
       theGraph.addEdge(2, 4, 40);  // CE 40
       theGraph.addEdge(3, 2, 20);  // DC 20
```

```
   theGraph.addEdge(3, 4, 70);  // DE 70
   theGraph.addEdge(4, 1, 50);  // EB 50

   System.out.println("Shortest paths");
   theGraph.path();             // shortest paths
   System.out.println();
   } // end main()
```

程序的输出结果是

```
A=inf(A) B=50(A) C=100(D) D=80(A) E=140(C)
```

path.java 程序

清单 14.2 显示了 path.java 程序的完整代码。它的各个部分在前面都讨论过了。

清单 14.2 path.java 程序

```
// path.java
// demonstrates shortest path with weighted, directed graphs
// to run this program: C>java PathApp
////////////////////////////////////////////////////////////////
class DistPar             // distance and parent
   {                      // items stored in sPath array
   public int distance;   // distance from start to this vertex
   public int parentVert; // current parent of this vertex
// --------------------------------------------------------------
   public DistPar(int pv, int d)  // constructor
      {
      distance = d;
      parentVert = pv;
      }
   } // end class DistPar
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // label (e.g. 'A')
   public boolean isInTree;
// --------------------------------------------------------------
   public Vertex(char lab)   // constructor
      {
      label = lab;
      isInTree = false;
      }
// --------------------------------------------------------------
   } // end class Vertex
////////////////////////////////////////////////////////////////
class Graph
```

```
   {
   private final int MAX_VERTS = 20;
   private final int INFINITY = 1000000;
   private Vertex vertexList[]; // list of vertices
   private int adjMat[][];      // adjacency matrix
   private int nVerts;          // current number of vertices
   private int nTree;           // number of verts in tree
   private DistPar sPath[];     // array for shortest-path data
   private int currentVert;     // current vertex
   private int startToCurrent;  // distance to currentVert
// -------------------------------------------------------------
   public Graph()               // constructor
      {
      vertexList = new Vertex[MAX_VERTS];
                                        // adjacency matrix
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      nTree = 0;
      for(int j=0; j<MAX_VERTS; j++)     // set adjacency
         for(int k=0; k<MAX_VERTS; k++)  //     matrix
            adjMat[j][k] = INFINITY;     //     to infinity
      sPath = new DistPar[MAX_VERTS];    // shortest paths
      }  // end constructor
// -------------------------------------------------------------
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// -------------------------------------------------------------
   public void addEdge(int start, int end, int weight)
      {
      adjMat[start][end] = weight;  // (directed)
      }
// -------------------------------------------------------------
   public void path()               // find all shortest paths
      {
      int startTree = 0;            // start at vertex 0
      vertexList[startTree].isInTree = true;
      nTree = 1;                    // put it in tree

      // transfer row of distances from adjMat to sPath
      for(int j=0; j<nVerts; j++)
         {
         int tempDist = adjMat[startTree][j];
         sPath[j] = new DistPar(startTree, tempDist);
```

```
        }

      // until all vertices are in the tree
      while(nTree < nVerts)
         {
         int indexMin = getMin();    // get minimum from sPath
         int minDist = sPath[indexMin].distance;

         if(minDist == INFINITY)     // if all infinite
            {                        // or in tree,
            System.out.println("There are unreachable vertices");
            break;                   // sPath is complete
            }
         else
            {                        // reset currentVert
            currentVert = indexMin;  // to closest vert
            startToCurrent = sPath[indexMin].distance;
            // minimum distance from startTree is
            // to currentVert, and is startToCurrent
            }
         // put current vertex in tree
         vertexList[currentVert].isInTree = true;
         nTree++;
         adjust_sPath();             // update sPath[] array
         }  // end while(nTree<nVerts)

      displayPaths();                // display sPath[] contents

      nTree = 0;                     // clear tree
      for(int j=0; j<nVerts; j++)
         vertexList[j].isInTree = false;
      }  // end path()
// ------------------------------------------------------------
   public int getMin()               // get entry from sPath
      {                              //    with minimum distance
      int minDist = INFINITY;        // assume minimum
      int indexMin = 0;
      for(int j=1; j<nVerts; j++)    // for each vertex,
         {                           // if it's in tree and
         if( !vertexList[j].isInTree &&  // smaller than old one
                              sPath[j].distance < minDist )
            {
            minDist = sPath[j].distance;
            indexMin = j;            // update minimum
            }
```

```
        } // end for
      return indexMin;                  // return index of minimum
      } // end getMin()
// ------------------------------------------------------------
   public void adjust_sPath()
      {
      // adjust values in shortest-path array sPath
      int column = 1;                   // skip starting vertex
      while(column < nVerts)            // go across columns
         {
         // if this column's vertex already in tree, skip it
         if( vertexList[column].isInTree )
            {
            column++;
            continue;
            }
         // calculate distance for one sPath entry
                     // get edge from currentVert to column
         int currentToFringe = adjMat[currentVert][column];
                     // add distance from start
         int startToFringe = startToCurrent + currentToFringe;
                     // get distance of current sPath entry
         int sPathDist = sPath[column].distance;

         // compare distance from start with sPath entry
         if(startToFringe < sPathDist)  // if shorter,
            {                           // update sPath
            sPath[column].parentVert = currentVert;
            sPath[column].distance = startToFringe;
            }
         column++;
         } // end while(column < nVerts)
   } // end adjust_sPath()
// ------------------------------------------------------------
   public void displayPaths()
      {
      for(int j=0; j<nVerts; j++) // display contents of sPath[]
         {
         System.out.print(vertexList[j].label + "="); // B=
         if(sPath[j].distance == INFINITY)
            System.out.print("inf");                  // inf
         else
            System.out.print(sPath[j].distance);      // 50
         char parent = vertexList[ sPath[j].parentVert ].label;
         System.out.print("(" + parent + ") ");       // (A)
```

```
        }
      System.out.println("");
      }
// --------------------------------------------------------
   }  // end class Graph
////////////////////////////////////////////////////////////////
class PathApp
   {
   public static void main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A');     // 0  (start)
      theGraph.addVertex('C');     // 2
      theGraph.addVertex('B');     // 1
      theGraph.addVertex('D');     // 3
      theGraph.addVertex('E');     // 4

      theGraph.addEdge(0, 1, 50);  // AB 50
      theGraph.addEdge(0, 3, 80);  // AD 80
      theGraph.addEdge(1, 2, 60);  // BC 60
      theGraph.addEdge(1, 3, 90);  // BD 90
      theGraph.addEdge(2, 4, 40);  // CE 40
      theGraph.addEdge(3, 2, 20);  // DC 20
      theGraph.addEdge(3, 4, 70);  // DE 70
      theGraph.addEdge(4, 1, 50);  // EB 50

      System.out.println("Shortest paths");
      theGraph.path();             // shortest paths
      System.out.println();
      }  // end main()
   }  // end class PathApp
////////////////////////////////////////////////////////////////
```

每一对顶点之间的最短路径问题

在第 13 章讨论连通性的时候，曾经想知道如果不介意中途的换乘，是否可以从 Athens 飞到 Murmansk。在带权图中，可以回答第二个问题，这个问题在 Hubris 航线柜台前等待时可能会产生：这趟旅行的花费是多少？

为了看出某条路线是否可能，需要创建一个连通表。在带权图中，用一张表给出任意两个顶点间的最低耗费，这对顶点可能通过几条边相连。这种问题叫做每一对顶点之间的最短路径问题。

依次用每个顶点做起始点，执行 path.java 程序，就可以得到这样一张表。如表 14.11 所示。

表 14.11 每一对顶点之间的最短路径表

	A	B	C	D	E
A	—	50	100	80	140
B	—	—	60	90	100
C	—	90	—	180	40
D	—	110	20	—	60
E	—	50	110	140	—

在前面的章节中会发现，Warshall 算法可以很快的创建这样的表，来显示从某个顶点通过一步或若干步到达其他顶点。带权图中类似的方法叫做 Floyd 算法，它是由 Robert Floyd 在 1962 年发明的。这是创建如表 14.11 所示的表格的另外一种方法。

下面用一个简单的图来讨论 Floyd 算法。图 14.11 显示了一个有向带权图和它的邻接矩阵。

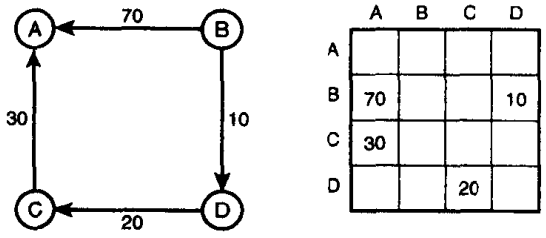


图 14.11 带权图和它的邻接矩阵

邻接矩阵显示了所有单段路径的耗费。现在想扩展这个矩阵，显示所有路径的耗费，但不考虑它的长度。例如，从图 14.11 可以看出，从 B 到 C 的耗费是 30（从 B 到 D 是 10，从 D 到 C 是 20，和为 30）。

正如在 Warshall 算法中那样，可以系统的修改邻接矩阵。检查每行的每个单元。如果有非 0 的权值（例如 C 行 A 列的 30），那么就看列 C（因为 C 是 30 所在的行）。如果在 C 列找到一个非 0 权值（例如，D 行的 20），那么就可以知道从 C 到 A 有一条权值为 30 的路径，从 D 到 C 有一条权值为 20 的路径。由此可以断定从 D 到 A 有一条两条边组成的路径，权值为 50。图 14.12 显示了在如图 14.11 所示图中应用 Floyd 算法的步骤。

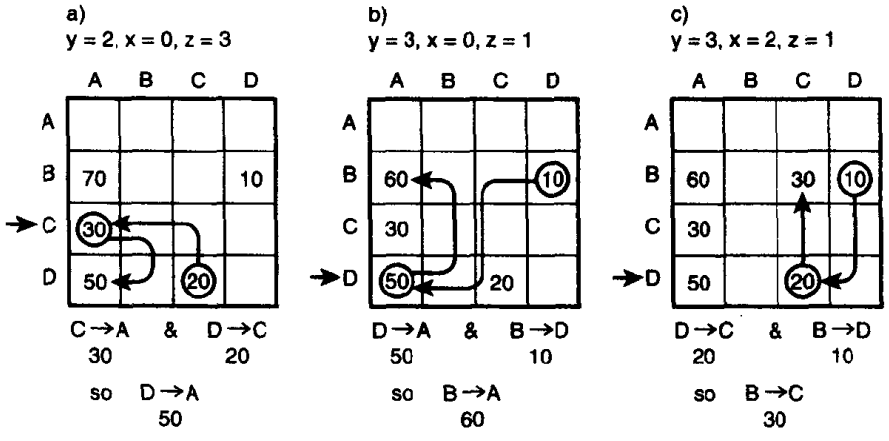


图 14.12 Floyd 算法

行 A 是空的，所以不需要做什么。在 B 行，A 列处有 70，D 列处有 10，但是 B 列是空的，所以 B 列的这些项不可能和以 B 为终点的边结合。

然而，在 C 行，发现在 A 列有 30。下面看 C 列，发现在 D 行有 20。现在有从 C 到 A 的路径

权值为30，从D到C的路径权值为20，所以，得到从D到A的路径权值为50。

D行出现了一个有趣的情况：可以得到一个比已存在的权值更低的权值。在列A为50，在B行D列为10，所以知道从B到A的路径权值为60。然而，在这个单元已经有一个数字－70。那么该怎么办？因为60小于70，所以用60取代70。在这种两点间多路径的情况下，要求这个表要保持有最小耗费的路径。

Floyd算法的实现与Warshall算法类似。然而，在Warshall算法中当找到一个两段路径时，只是简单的在表中插入1，在Floyd算法中，需要把两端的权值相加，并插入它们的和。算法的细节在练习中给出。

效　率

迄今为止，还没有讨论各种图的算法的效率。由于图的表示法有两种，邻接矩阵和邻接表，使得这个问题变得相对复杂。

如果使用邻接矩阵，前面讨论的算法大多需要$O(V^2)$的时间级，这里V是顶点的数量。为什么？如果分析这些算法，就发现它们几乎都检查了一遍所有顶点，具体方法是在邻接矩阵中扫描每一行，依次查看每一条边。换句话讲，邻接矩阵的每个单元，一共有V^2个单元，都被扫描过。

对于规模大的矩阵，$O(V^2)$的时间级不是非常好的性能。如果图是密集的，那就没有什么提高性能的余地。（正如前面看到的，密集意味着图有很多边，而它的邻接矩阵的许多或大部分单元被占。）

然而，许多图是稀疏的，与稠密相反。其实并没有一个确定数量的定义说明多少条边的图是稠密的或是稀疏的，但是如果在一个大图中每个顶点只有很少几条边相连，那么这个图通常被认为是稀疏的。

在稀疏图中，使用邻接表的表示方法代替邻接矩阵，可以改善运行时间。这很容易理解：不必浪费时间来检索邻接矩阵中没有边的单元。

对于无权图，邻接表的深度优先搜索需要$O(V+E)$的时间级，这里V是顶点数量，E是边的数量。对于带权图，最小生成树算法和最短路径算法都需要$O((E+V)\log V)$的时间级，在大型的稀疏图中，与邻接矩阵方法的时间级相比，这样的时间级可使性能大幅提升。然而，算法多少会更加复杂一些，这就是为什么在本章中仍然使用邻接矩阵方法的原因。可以参考Sedgewick（附录B，“进一步阅读”）和其他作者用邻接表方法实现图的算法的例子。

Warshall算法和Floyd算法比到目前为止本书讨论过的所有算法花费的时间都要长。它们的执行需要$O(V^3)$的时间级。这是因为在它们的实现中使用了三层嵌套的循环。

难　题

本书中，已经看到了大O的值，从O(1)，通过O(N)，$O(N*\log N)$，$O(N^2)$，一直到（Warshall算法和Floyd算法）$O(N^3)$。甚至$O(N^3)$时间级的问题当N为几千时，也能在可接受的时间内完成。这些大O值的算法可以用来解决绝大多数实际问题。

然而，有些算法的大O值非常大，以至于只有在N值非常小的时候才有解。许多真实世界需

要用这种算法来解决的难题，简直不能在合理的时间内完成。这样的问题称作“难题”。（这样的问题有另外一个术语叫做 NP 完全性，NP 的意思是“不确定多项式”。对这个定义的解释超出了本书的范围。）

骑士周游

骑士周游（第 13 章的编程作业 13.5）是一个难题的例子，因为可能的移动步数非常大。可能的移动序列的总数虽然很难计算，但是可以估计它。每次移动最多有八个选择。如果考虑移动后骑士出了棋盘或目标位置已经访问过，那么这个移动数字会缩减。在周游初期，接近八种移动方向，但是随着棋盘被填满，这个数字逐渐下降。假设（保守地）从每个位置只有两步可能的移动。除了初始化位置，骑士要访问 63 个方块。因此，一共有 2^{63} 种可能的移动。这大约是 10^{19}。如果一台计算机一秒钟可以计算一百万次移动（10^6）。一年大约是 10^7 秒，那么这台计算机一年可以计算 10^{13} 次移动。如此解决这个问题就需要 10^6，也就是一百万年的时间。

如果使用决策树剪枝的策略，这个特殊问题可以转化为可解的。其中有一种是 Warnsdorff 的启发式策略（H.C. Von Warnsdorff，1823），他指出每次要移动到出路可能性最小的方块上去。

旅行商问题

下面是另一个著名的难题：假定有一个商人，需要驾车到所有客户所在的城市，但是他希望走过的路线最短。商人知道从每个城市到另外的城市的距离。商人要从自己的城市出发，在每个客户城市访问一次且仅一次，然后回到自己的城市。以什么顺序访问这些城市可以使走过的路程最短？在图论中，这是旅行商问题，简称 TSP。

图 14.13 显示了一组城市和它们之间距离的图。从 A 出发，访问所有城市再回到 A，哪条路线是最短的？注意，每对城市不必都是直接相连。例如，由于地形的原因，他必须经过 Philadelphia，就能从 Washington, D.C.到达 New York。

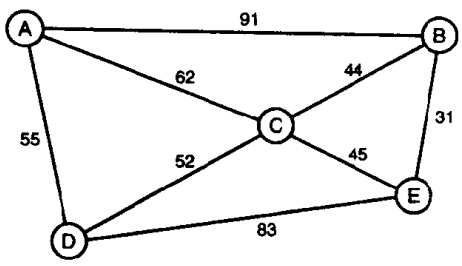


图 14.13 城市和距离

为了找到最短距离，他列出所有可能的城市组合（Boston-Seattle-Miami, Boston-Miami-Seattle, Miami-Boston-Seattle 等等），然后计算每种组合的总里程数。路线 ABCEDA 总长为 318。路线 ABCDEA 不存在，因为没有从 E 直接到 A 的路线。

不幸的是，组合的数量非常巨大：它是城市数量的阶乘（不计算出发城市）。如果有六个城市要访问，那么在出发城市就有六个城市可以选择，第二个有 5 个，第三个有 4 个，依次类推；一共有 6*5*4*3*2*1，即 720 种可能的路线。这个问题的规模如果是 50 个城市，就是不切实际的。另外，有一些策略可以去掉那些不可能的序列，但是这样做帮助不大。用带权图来解决这个问题，权值代表英里数，顶点代表城市。如果像一般的理解那样，从 A 到 B 与从 B 到 A 的距离相同，那么这个图是无向图。如果权值代表的是飞机票价，那么不同方向可能就不同，就应该使用有向图来表示。

汉密尔顿回路

这个问题与 TSP 相似，但是更加抽象，它要找到图的汉密尔顿回路。正如前面说到的，回路是一条源点和终点都是同一个顶点的路径。汉密尔顿回路中访问一次且只访问一次其他顶点。与 TSP

的区别是，这里不关心距离；惟一关心的是是否有这样的回路存在。在图 14.13 中，ABCEDA 是一个汉密尔顿回路，而 ABCDEA 不是。骑士周游问题是汉密尔顿回路问题的一个实例（假设骑士每次回到他出发的方块）。

找汉密尔顿回路与 TSP 一样，都需要 O(N!)的时间级。通常用术语“指数级时间”表示大 O 值为 2^N 或 N!（增长的速度比 2^N 还要快）的时间级。

小 结

- 带权图中，边带有一个数字，叫做权，它可能代表距离、耗费、时间或其他意义。
- 带权图的最小生成树中有所有的顶点和连接它们的必要的边，且这些边的权值最小。
- 优先级队列的算法可用于寻找带权图的最小生成树。
- 带权图的最小生成树模拟了真实世界的许多情况，例如在两城市间铺设线缆。
- 无权图的最短路径问题要找到两点间的路径，且路径长度最短。
- 带权图的最短路径问题产生的路径总权值和最小。
- 带权图的最短路径问题可以用 Dijkstra 算法解决。
- 对于大型的稀疏图，用邻接表表示，相对于用邻接矩阵表示，可以提高算法的运行效率。
- 每一对顶点间的最短路径问题是要找到图中每对顶点间的权值和。Floyd 算法用来解决这个问题。
- 有些图的算法需要指数级时间，因此在顶点多的图中应用不太实际。

问 题

下列问题作为读者的自测题。答案可见附录 C。

1. 带权图的权值是图的_____的属性。
2. 在带权图中，最小生成树（MST）试图最小化
   a. 从源点到指定顶点的边的数量。
   b. 连接所有顶点的边的数量。
   c. 从源点到指定顶点的边的总权值。
   d. 连接所有顶点的边的总权值。
3. 判断题：MST 的权值取决于源点的选择。
4. 在 MST 算法中，从优先级队列中删除什么？
5. 在有线电视的例子中，加入到 MST 中的每条边连接
   a. 源点和它的一个邻接点。
   b. 已连接的城市和未连接的城市。
   c. 当前顶点和它的邻接点。
   d. 两个有办公室的城市。
6. 当边连接一个_____的顶点，MST 算法从表单中“剪除”这条边。
7. 判断题：最短路径问题（SPP）必须在有向图中执行。

8. Dijkstra 算法找到_____的最短路径。
   a. 从一个指定顶点到所有其他的顶点。
   b. 从一个指定顶点到另一个指定顶点。
   c. 从所有顶点到相应的邻接点
   d. 从所有顶点到相应的非邻接点（路径包含一条以上的边）。
9. 判断题：Dijkstra 算法的规则是总是把最接近源点的顶点放入树中。
10. 在铁路费用的例子中，边缘城市是
   a. 到这点的距离已知，但是从这点出发的距离未知。
   b. 在树中。
   c. 到这点的距离已知，且在树中。
   d. 完全未知。
11. 每一对顶点间的最短路径问题需要找到_____的最短路径。
   a. 从源点到其他的每个顶点。
   b. 从每个顶点到其他的每个顶点。
   c. 从源点到每个邻接点。
   d. 从源点到每个非邻接点（路径包含一条以上的边）。
12. Floyd 算法用于带权图，_____用于不带权的图。
13. Floyd 算法使用_____表示图。
14. 要试图解决骑士周游问题，需要的大 O 时间级为多少？
15. 在图 14.13 中，路线 ABCEDA 是旅行商问题的一个解决方案吗？

实　验

完成这些实验可以帮助深入理解本章的主题。不需要编程实现。

1. 使用 GraphW 专题 applet 找到图 14.6 所示的图的最小生成树。考虑图是无向的，即忽略箭头。

2. 使用 GraphDW 专题 applet 解决图 14.6 所示的图的最短路径问题，但是所有边要用新的权值，方法是从 100 中减去原来的权值。

3. 画一个五个顶点，五条边的图。然后在纸上实现这个图的 Djikstra 算法。显示每一步的树型和最短路径数组。

编程作业

编程作业有助于巩固对本章内容的理解，并展示如何应用本章的概念。（在“简介”中提到过，资深教师可以从出版者的网站上得到编程作业的完整答案。）

14.1　修改 path.java 程序（清单 14.2），打印一张任意两点间最小耗费的表。这个练习需要修改原来假设源点总是 A 的例程。

14.2　迄今为止已经用邻接矩阵和邻接表实现了图。另一个方法是用 Java 的引用代表边，这样

Vertex 对象就包含一个对其邻接点的引用列表。在有向图中，这种方式的引用尤其直观。因为它“指明”从一个顶点到另一个顶点。写程序实现这个表示方法。main()方法应该与 path.java 程序（清单 14.2）的 main()方法类似，为的是它可以调用相同的 addVertex()方法和 addEdge()方法创建如图 14.6 的图。然后，它应该显示图的连通表，以证明图被正确的构建。还要在某处存储每条边的权值。一个方法是使用 Edge 类，类中存储权值和边的终点。每个顶点拥有一个 Edge 对象的列表——即，从这个顶点开始的边。

14.3 实现 Floyd 算法。可以从 path.java 程序（清单 14.2）开始，适当地修改它。例如，可以删除所有的最短路径代码。保留对不可到达顶点为无穷大的表示法。这样作，当比较新得到的耗费和已存在的耗费时，就不再需要检查 0。所有可能到达的路线的耗费都比无穷大小。可以在 main()方法中构建任意复杂的图。

14.4 实现本章“难题”中描述的旅行商问题。不考虑它的难度，当 N 较小时，比如 10 或更小，这个问题很好解决。在无向图上尝试一下。使用穷举法测试每种可能的城市序列。序列改变的方式，参考第 6 章“递归”中的 anagram.java 程序（清单 6.2）。用无穷大表示不存在的边。这样做，当从一个城市到下一个城市的边不存在的情况发生时，不需要中止对序列的计算；任何无穷大的权值总和都是不可能的路线。而且，不必考虑消除对称路线的情况。例如 ABCDEA 和 AEDCBA 都要显示。

14.5 编写程序找到并显示带权无向图的所有汉密尔顿回路。

第 15 章

应 用 场 合

本章重点

- 通用数据结构
- 专用数据结构
- 排序
- 图
- 外部存储

本章简要地总结了到目前为止所学的知识，并着眼于讨论在不同的情况如何决定使用哪种数据结构和算法。

本章还有必要提醒读者，每一个真实世界的问题都有各自的特性，因此我们在这里所说的并不一定就是你所遇问题的正确答案，这种情况是经常出现的。下面把本章的内容大致分一下组：

- 通用数据结构：数组，链表，树，哈希表
- 专用数据结构：栈，队列，优先级队列
- 排序：插入排序，希尔排序，快速排序，归并排序，堆排序
- 图：邻接矩阵，邻接表
- 外部存储：顺序存储，索引文件，B-树，哈希方法

注释

想了解这些主题的细节信息，请参阅本书前面的不同章节。

通用数据结构

若想存储真实世界中的类似人事记录、存货目录、合同表或销售业绩等数据，则只需要一般用途的数据结构。在本书中属于这种类型的结构有数组、链表、树和哈希表。它们被称之为通用的数据结构是因为它们通过关键字的值来存储并查找数据，这一点在通用数据库程序中常见到(栈等特殊结构正好相反，它们只允许存取一定的数据项)。

对于一个给定问题，这些通用数据结构中哪一种是合适的呢？图 15.1 对这个问题给出了一个大致的解法。但是除去图中表示出来的因素之外，还有许多条件可以影响最终结果。为了更加详细地讨论这个问题，我们会先探讨一些通用的因素，然后再集中精力于个别的结构。

速度与算法

通用数据结构可以完全按照速度的快慢来分类：

数组和链表是最慢的，树相对较快，哈希表是最快的。

但是请不要从图 15.1 中得到这样的结论：使用最快的结构永远是最好的方案。这些最快的结构也有缺陷。首先，它们的程序在不同程度上比数组和链表的复杂；其次，哈希表要求预先知道要存储多少数据，数据对存储空间的利用率也不是非常高。普通的二叉树对顺序的数据来说，会变成缓慢的 O(N)级操作；而平衡树虽然避免了上述的问题，但是它的程序编制起来却比较困难。

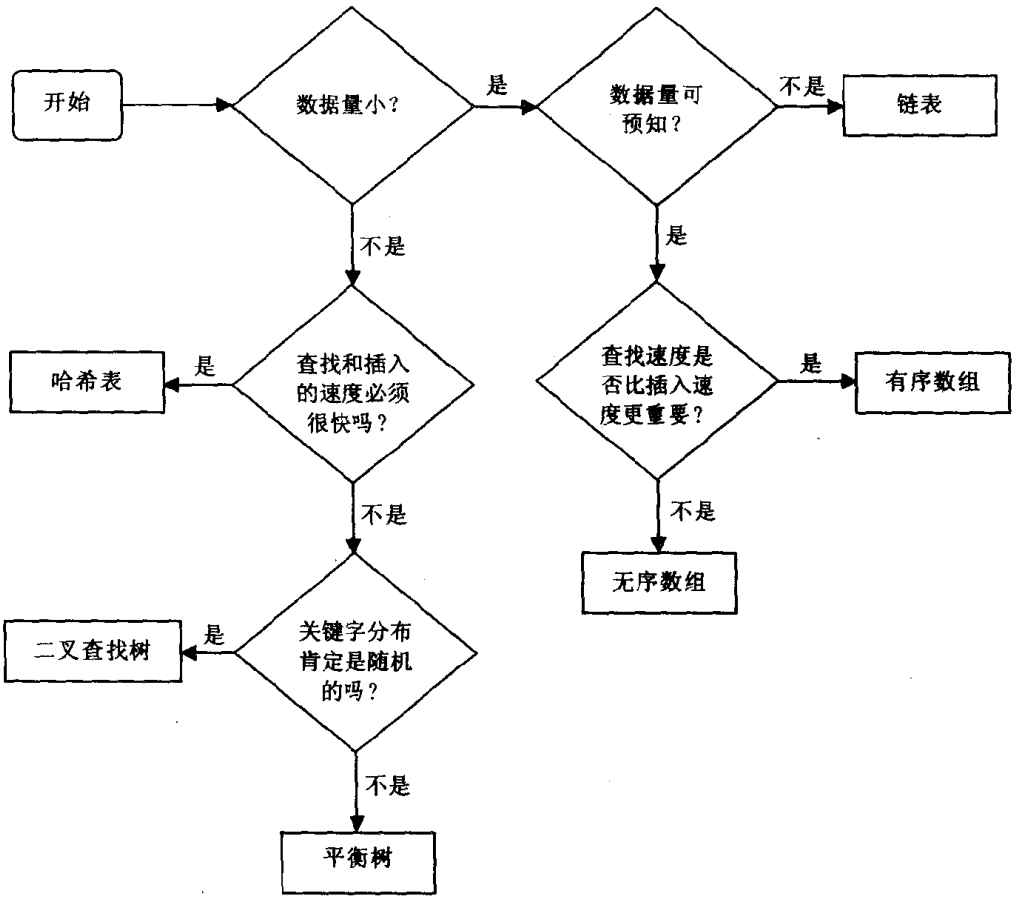


图 15.1 通用数据结构的关系

处理器速度：一个变动着的目标

快速的结构都有缺陷，而计算机的另一个发展因素却能使低速的结构更加具有吸引力。新计算机的 CPU 和存取速度每一年都有提升。Moore 定律（Gordon Moore 在 1965 年提出）声明了 CPU 的速度每 18 个月翻一倍。这造成了早期计算机和现今应用的计算机在性能方面的惊人差异，而且目前没有任何理由能认为这个增长速度会减慢。

假设几年前的一台电脑可以在一个可接受的时间内处理含有 100 个对象的数组，而现在的计算机则快多了，因此可以在同样的时间里处理含有 10000 个对象的数组。许多作者都对某种数据结构的最大可应用范围给出了定量的估计，超出了这个范围会变得很慢。请不要相信这些估计（包括本书在内）。今天的估计不能应用于明天。

请从简单数据结构入手考虑：除非它们明显是太慢了，否则就用数组或链表编写程序，看看结果究竟怎样。如果能在一个可接受的时间内运行完毕，那么就采用它，不必再找别的了。没有人会留意用的是数组或别的什么结构，为什么一定要拼命地写出一个平衡树的算法？甚至必须面对成千上千万、百万的数据项进行操作时，不妨先看一看数组或链表处理表现的情况，这也还是值得的。只有在实验中发现这些简单结构的性能太慢时，才回过头来采用那些更加复杂的数据结构。

Java 引用的优点

在操作对象的速度上，Java 与其他语言相比有极大的优势，那是由于对于大多数数据结构来说，Java 只存储引用（reference）而不是实际的对象，因此相对于那些在数据结构中实际为对象开辟了空间的语言来说，大多数 Java 算法的执行速度更快。在分析算法时，不是从对象的真实存储空间出发，因而“移动”对象的速度也不依赖于对象的大小，而只是考虑对象引用的移动，因此对象本身的大小就不重要了。

当然在类似 C++等其他语言中，也可以存储对象的指针而不是存储对象本身，这与使用引用的效果是一样的，但是它们的语法更加复杂。

类库

在所有主要的编程语言中，都具备商用数据结构库。在这些语言中有一些内置的结构。比如在 Java 中，有向量、栈和哈希表类；在 C++中有标准模板库，其中含有许多数据结构和算法。

使用一个商用类库可能会去掉或至少削减创建本书描述的数据结构的必要编程。当处于这种情况时，更加吸引人的是使用一个如平衡树这样的复杂结构或象快排这样的精巧算法。然而，采用时必须确保这个类真正能够适应你的特定的情况。

数组

当存储和操作数据时，在大多数情况下数组是首先应该考虑的结构。数组在下列情况下很有用：

- 数据量较小。
- 数据量的大小事先可预测。

如果存储空间足够大的话，可以放松第二条，创建一个足够大的数组来应付所有可以预见的数据输入。

如果插入速度很重要的话，使用无序数组。如果查找速度很重要的话，使用有序数组，并用二分查找。数组元素的删除总是很慢，这是由于为了填充空出来的单元，平均半数以上的数组元素要被移动。在有序数组中的遍历（指按关键字的有序遍历——译者注）是很快的，而在无序的数组不支持这种功能。

向量（如 Java 中的向量类）是一种当数据太满时可以自己扩充空间的数组。向量可以应用于数据量不可预知的情况下。然而，在向量扩充时，要将旧的数据拷入一个新的空间中，这一过程会造成程序明显的周期性暂停。

链表

如果需要存储的数据量不能预知或者需要频繁地插入删除数据元素时，考虑使用链表。当有新的元素加入时，链表就开辟新的所需要的空间，所以它甚至可以占满全部可用内存；在删除过程中没有必要像数组那样添补“空洞”。

在一个无序的链表中，插入是相当快的。查找和删除却很慢(尽管比数组的删除快一些)，因此，与数组一样，链表最好也应用于数据量相对较小的情况。

对于编程而言，链表比数组复杂，但它比树或哈希表简单。

二叉搜索树

当确认数组和链表过慢时，二叉树是最先应该考虑的结构。树可以提供快速的 O(logN)级的插

入、查找和删除。遍历的时间复杂度是O(N)级的，这是任何数据结构遍历的最大值（根据定义，必须访问所有的数据）。对于遍历一定范围内的数据可以很快得出访问数据的最大值和最小值。

对于程序来说，不平衡的二叉树要比平衡二叉树简单得多，但不幸的是，有序数据能将它的性能降至O(N)级，不比一个链表好多少。然而如果可以保证数据是随机进入的，就不需要用平衡二叉树。

平衡树

在众多平衡树中，我们讨论了红-黑树和2-3-4树，它们都是平衡树，并且无论输入数据是否有序，它们都能保证性能为O(logN)。然而对于编程来说，这些平衡树都是很有挑战性的，其中最难的是红-黑树。它们也因用了附加存储而产生额外耗费，这对系统或多或少有些影响。

如果利用树的商用类可以降低编程的复杂性。有些情况下，选择哈希表比平衡树更好。即便当数据有序时，哈希表的性能也不降低。

平衡树有许多种，其中包括AVL树、splay树、2-3树等，但它们不如红-黑树使用得广泛。

哈希表

哈希表在数据存储结构中速度最快。这使它成为计算机而不是人与数据交互时的必需。哈希表通常用于拼写检查器和作为计算机语言编译器中的符号表，在这些应用中，程序必须在几分之一秒的时间里检查上千的词或符号。

当人而不是计算机初始化操作数据的存储时，哈希表会很有用。正如前文所提到的，哈希表对数据插入的顺序并不敏感，因此可以取代平衡树。但是哈希表的编程却比平衡树简单多了。

哈希表需要有额外的存储空间，尤其是对于开放定址法。因为哈希表用数组作为基本结构，所以，必须预先精确地知道待存储的数据量。

用链地址法处理冲突的哈希表是最健壮的实现办法。若能预先精确地知道数据量，在这种情况下用开放定址法编程最简单，因为不需要用到链表类。

哈希表并不能提供任何形式的有序遍历，或对最大最小值元素进行存取。如果这些功能重要的话，使用二叉搜索树更好一些。

通用数据存储结构的比较

表15.1用大O表示法总结了不同的通用数据存储结构的速度。

表15.1 通用数据存储结构

数据结构	查找	插入	删除	遍历
数组	O(N)	O(1)	O(N)	—
有序数组	O(logN)	O(N)	O(N)	O(N)
链表	O(N)	O(1)	O(N)	—
有序链表	O(N)	O(N)	O(N)	O(N)
二叉树（一般情况）	O(logN)	O(logN)	O(logN)	O(N)
二叉树（最坏情况）	O(N)	O(N)	O(N)	O(N)
平衡树（一般情况和最坏情况）	O(logN)	O(logN)	O(logN)	O(N)
哈希表	O(1)	O(1)	O(1)	—

假设未经排序的数组的插入是在数组的后端进行的。有序数组在进行二分查找时很快，但是插入和删除却很慢，因为这些操作平均要移动一半的元素。遍历意味着通过升序或降序的关键字来对数据进行访问，— 符号表示不支持这个操作。

专用数据结构

本书中讨论的专用数据结构有栈、队和优先级队列。这些结构不是为了用户可访问的数据库而建立的，通常用它们在程序中辅助实现一些算法。我们在本书中见到了许多这样的例子，譬如在第13章“图”和第14章“带权图”的图的算法中都用到栈、队和优先级队列。

栈、队和优先级队列是抽象数据类型（ADT），它们又由一些更加基础的结构如数组、链表或堆（如在优先级队列中）组成。这些 ADT 只提供给用户简单的接口，一般仅允许插入和访问或者删除一个数据项。

这些数据项是：

对于栈：最后被插入的数据项

对于队：最先被插入的数据项

对于优先级队列：具有最高优先级的数据项

这些 ADT 可以被当作帮助理解的概念。它们的功能可以通过直接使用基础结构(如数组)来实现，但是它们提供的精简的接口可以简化许多问题。

栈

栈用在只对最后被插入数据项访问的时候，它是一个后进先出（LIFO）的结构。

栈往往通过数组或链表实现，通过数组实现很有效率，因为最后被插入的数据总是在数组的最后，这个位置的数据很容易被删除。栈的溢出有可能出现，但当数组的大小被合理的规划后，溢出并不常见，因为栈很少会拥有大量的数据。

如果栈拥有许多数据，并且数量不可精确预测（当用栈实现递归时）时，用链表比数组更好一些，这是由于从表头的位置删除或插入一个元素很方便。除非整个内存满了，栈的溢出不可能出现。链表比数组稍慢一些，因为对于插入一个新链接必须分配内存，从表中某链接点上删除元素后回收分配内存亦是必需的。

队

队用在只对最先被插入数据项访问的时候，它是一个先进先出（FIFO）的结构。

同栈相比，队同样可以通过数组和链表实现。这两种方法都很有效率。数组需要附加的程序来处理队在数组尾部回绕的情况。链表必须是双端的，这样才能从一端插入从另一端删除。

用数组还是链表来实现队的选择是通过数据量是否可以被很好地预测来决定的。如果知道会有多少数据量的话，就使用数组；否则的话就用链表。

优先级队列

优先级队列用在只对访问最高优先级数据项访问的时候，最高优先级数据项就是含有最大(有时最小)的关键字的项。

优先级队列可以用有序数组或堆来实现。向有序数组中插入是很慢的，但是删除很快。使用堆来实现优先级队列，插入和删除的时间复杂度都是 O(logN)级。

当插入速度不重要时，可以使用数组或双端链表。当数据量可以被预测时，使用数组；当数据量未知时，使用链表。如果速度很重要的话，选择堆更好一些。

专用结构的比较

表 15.2 用大 O 表示法比较了栈、队和优先级队列。这些结构都不支持查找或遍历。

表 15.2 专用数据存储结构

数据结构	插入	删除	注释
栈（数组或链表）	O(1)	O(1)	删除最先插入的
队（数组或链表）	O(1)	O(1)	删除最后插入的
优先级队列（有序数组）	O(N)	O(1)	删除优先级最高的
优先级队列（堆）	O(logN)	O(logN)	

排 序

当选择数据结构时，可以先尝试一种较慢但简单的排序，例如插入排序。如果采用了这些方法，现代计算机的快速处理速度也有可能在恰当的时间内将较大的数据量排序。（比较粗略的估计是，较慢的排序对少于 1000 的数据量为宜。）

插入排序对几乎已排好的文件也很有效，如果没有太多的元素处于乱序的位置上，操作的时间复杂度大约在 O(N)级。这通常发生在往一个已排好序的文件中插入一些新的数据元素的情况。

如果插入排序显得太慢，下一步可以尝试希尔排序。它很容易实现，并且使用起来不会因为条件不同而性能变化差距太大；Sedgewick 估计它在数据量为 5000 以下时很有用。

只有当希尔排序变得很慢时，你才应该使用那些更复杂但更快速的排序方法：归并排序、堆排序或快速排序。归并排序需要辅助存储空间，堆排序需要有一个堆的数据结构，前两者都比快速排序在某些程度上慢，所以当需要最短的排序时间时经常选择快速排序。

然而，快速排序在处理非随机性数据时性能不大可靠，因为那时它的速度有可能蜕化至 $O(N^2)$ 级。

对于那些有可能是非随机性的数据来说，堆排更加可靠。当快速排序没有被正确地实现时，它会产生微小偏差。在代码中细小的错误会使它对按某些顺序排列的数据无能为力，而诊断到这种情况却又相当难。

表 15.3 总结了不同排序算法的运行时间。在比较那一列中尝试对具有相同的平均时间复杂度的排序算法间的细微速度差别进行比较。（希尔排序没有比较，因为没有其他的算法与之有相同的大 O 性能级别。）

表 15.3 排序算法的比较

排序	平均情况	最坏情况	比较	附加存储
冒泡排序	$O(N^2)$	$O(N^2)$	及格	不需要
选择排序	$O(N^2)$	$O(N^2)$	良好	不需要
插入排序	$O(N^2)$	$O(N^2)$	优良	不需要
希尔排序	$O(N^{3/2})$	$O(N^{3/2})$	—	不需要
快速排序	O(N*logN)	$O(N^2)$	优良	不需要
归并排序	O(N*logN)	O(N*logN)	良好	需要
堆排	O(N*logN)	O(N*logN)	良好	不需要

图

图在数据存储结构的神殿（pantheon）中与众不同。它们并不存储通用数据，也并不会在其他算法中成为程序员的工具，正相反，它们直接模拟现实世界的情况。图的结构直接反映了问题的结构。

当需要用图时，没有其他的数据结构可以取代之，所以对于何时选择图并没有太多可说的。主要的选择还是如何表示图：使用邻接矩阵或邻接表。这个选择依赖于图的疏密程度，稠密的图用邻接矩阵，稀疏的图用邻接表。

邻接矩阵表示的图的深度优先搜索和广度优先搜索的时间复杂度为 $O(V^2)$级，V 是顶点的个数。邻接表表示的图的这两种操作的时间复杂度为 O(V+E)级，E 是边的条数。最小生成树和最短路径在使用邻接矩阵表示时为 $O(V^2)$级，邻接表为 O((E+V)logV)级。请先估计图中的 V 和 E，并通过计算来判断哪种表示方法更加合适。

外部存储

在先前的讨论中，我们假设了数据被存放在内存中。然而如果数据量大到内存容不下时，只能被存到外部存储空间，它们经常被称为磁盘文件。在第 10 章的第二部分“2-3-4 树和外部存储”和第 11 章“哈希表”中讨论了外部存储。

存在磁盘文件中具有固定大小单元的数据被称为块（block），每一个块都存储一定数量的记录。（磁盘文件中的记录拥有与主存中对象相同的数据类型。）与对象一样，每条记录都有一个关键字值，通过它可以访问到这条记录。

同样，我们假设了读写操作总是在一个单一的块中进行，这些读写操作比对主存中的数据进行任何操作都要耗时得多。因此，为了提高操作速度必须将磁盘的存取次数减到最小。

顺序存储

通过特定的关键字进行搜索的最简单的方法是随机存储记录然后顺序读取。新的记录可以被简单地插入在文件的最后。已删除的记录可以标记为已删除，或将记录顺次移动(如同数组中)来填补空缺。

就平均而言，查找和删除会涉及读取半数的块，所以顺序存储并不很快，时间复杂度为 O(N)。但是对于小量数据来说它仍然是令人满意的。

索引文件

当使用索引文件时，速度会明显的提高。在这种方法中关键字的索引和相应块的号数被存放在内存中。当通过一个特殊的关键字访问一条记录时，程序会先向索引询问。索引提供这个关键字的块号数，然后只需要读取这一个块，仅耗费了 O(1)级的时间。

可以使用不同种类的关键字来做多种索引（名字做一个，社会安全号码做一个，等等）。只要索引数量能在内存的存储的范围之内，这种方法表现得很好。

通常，索引文件存储在磁盘上，只有在需要时才复制进内存中。

索引文件的缺点是必须先创建索引，这有可能对磁盘上的文件进行顺序读取，所以创建索引是很慢的。同样，当记录被加入到文件中时，索引还需要更新。

B-树

B-树是多叉树，通常用于外部存储，树中的节点对应于磁盘中的块。同其他树一样，通过算法来遍历树，在每一层上读取一个块。B-树可以在 O(logN)级的时间内进行查找、插入和删除。这是相当快的，并且它对很大的文件也很有效。但是，它的编程很繁琐。

哈希方法

如果可以占用一个文件通常大小两倍以上的外部存储空间的话，外部哈希会是一个很好的选择。它同索引文件一样有着相同的存取时间，O(1)，但它可以对更大的文件进行操作。

图 15.2 形象地表示了如何选择外部存储结构。

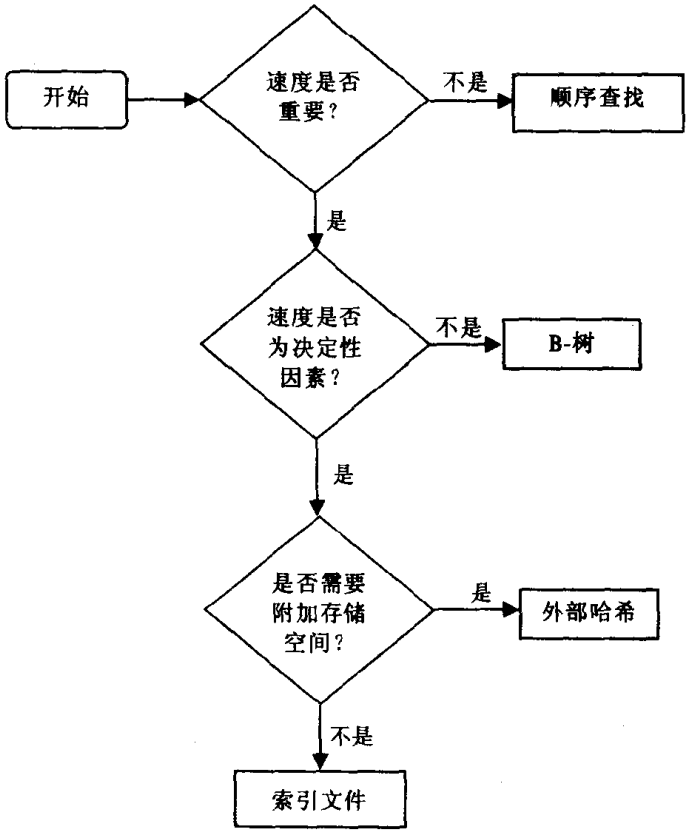


图 15.2 外部存储选择的关系

虚拟内存

有时可以通过操作系统的虚拟内存的能力(如果它有的话)来解决磁盘存取问题，而不需通过编程。

如果读取一个大小超过主存的文件，虚拟内存系统会读取合适主存大小的部分并将其他存储在磁盘上。当访问文件的不同部分时，它们会自动从磁盘读入并放置于内存中。

可以对整个文件使用内部存储的算法，使它们好像同时都在内存中一样；如果文件的那个部分不在内存中，也让操作系统去读取它们。

当然，这样的操作比整个文件在内存中的速度要慢得多，但是通过外部存储算法一块一块地处理文件的话，速度也是一样的慢。不要在乎文件的大小适合放在内存中，在虚拟内存的帮助下验证算法工作得好坏是有益的，尤其是对那些比可用的内存大不了多少的文件来说，这更是一个简单的解决方案。

前　进

至此，我们到达了对数据结构和算法探索旅程的终点。这个题目很大而且很复杂，没有哪一本书可以让你成为专家，但我们希望本书可以使你学习基础知识更加容易一些。附录 B“进一步学习”中将向你提供更进一步学习的建议。

附录 A
运行专题 applet 和示例程序

本附录重点

- 专题 applet
- 示例程序
- Sun Microsystem 软件开发工具集
- 重名的类文件
- 其他开发系统

在这个附录中讨论运行专题 applet 和示例程序的细节。

- 专题 applet 是图形化的演示程序，它们显示了树和其他数据结构是什么样子的。
- 示例程序的代码写在书中，它们是可执行的 Java 代码。

我们同样讨论了 Sun Microsystem Java 第二版（J2SE）软件开发工具集（SDK），通过它不仅可以运行本书中的 applet 和示例程序，并且可以修改程序代码或编写自己的程序。

可以在 Sams 网站：www.samspublishing.com中下载本书的不同版本的 applet 和示例程序。登录后使用本书的国际标准书目号（ISBN）访问该书的网页，在那里可以找到下载的连接。

专题 applet

一个 applet 是一个特殊的 Java 程序，它可以轻松地发布到国际互联网上。因为 Java applet 是为互联网设计的，它们可以在有合适网页浏览器或 applet 察看器的任何计算机平台上运行。

本书中，专题 applet 为文章中讨论的概念提供了活跃的，互动图形化演示。例如，第 8 章“二叉树”包括了一个专题 applet 在一个 applet 窗口中显示了一棵树。点击 applet 的按钮会演示向树中插入一个新节点的每一步，删除一个存在的节点，遍历树等等。其他章节中都包括了相应的专题 applet。

在下载了专题 applet 之后，可以使用流行的网页浏览器立即运行它们。这些浏览器包括最近版本的 Microsoft Internet Explorer 和 Netscape Communicator。商用 Java 开发产品也包含有 applet 察看器通用程序能够运行 applet，还可以使用包括在 SDK 中的 appletviewer 工具来运行专题 applet。

下面是如何用一个典型的网页浏览器运行 applet 脱机工作，从文件菜单下选择打开，并选择到合适的目录。每一个专题 applet 都有一个子目录，子目录下有若干带有.class 后缀的文件和一个带有.html 后缀的文件。打开.html 文件后，applet 会出现在屏幕上。

示例程序

示例程序用来尽可能简单地显示书中讨论的数据结构和算法是如何用 Java 实现的。这些 Java 程序包含有 Java 应用（与 applet 相对）。Java 应用不是用来发布在网上的，它是用来当作一个普通的程序在特定的计算机上运行的。

Java 应用可以在控制台（console）模式或图形模式下运行。为了更加简单，本书中的示例程序运行在控制台模式下，这种模式的输出是文本格式，输入需要用户从键盘输入。在 Windows 环境下，控制台模式运行于一个 MS-DOS 窗口中。在这种模式里不能显示图形。

书中有示例程序的源代码，与书中源程序相同的文件可以在 Sams 网站中下载。

Sun Microsystem 软件开发工具集

示例程序和专题 applet 都可以使用 Sun 的 SDK 的程序所提供的工具来运行。SDK 可以从 Sun 的网站中下载: www.sun.com。请在其中找到 Java 2 Standard Edition(J2SE) Software Development Kit。这个软件包很大，它不仅可以提供运行书中提供的程序和 applet 的环境，而且可以用它开发自己的 Java applet 和应用。

命令行程序

SDK 运行在文本模式下，使用命令行来启动各种程序。在 Windows 中，可以通过开启 MS-DOS 窗口来进行命令行操作。点击 Start（开始）按钮，找到名为 MS-DOS Prompt 的程序，它可能在附件文件夹中，并有可能叫做其他名称，例如叫 Command Prompt。

然后在 MS-DOS 下使用 cd(转换目录)命令来移动至硬盘上的合适目录下，这个目录下应存有专题 applet 或示例程序。按下文所介绍的详细方法使用适当的 SDK 工具运行这些程序。

设置路径

在 Windows 中，SDK 工具程序的位置必须在 autoexec.bat 文件中声明，使这些程序可以从任何子目录下方便地访问。当运行 SDK 安装程序时，PATH 语句可能会自动放在 autoexec.bat 文件中。否则的话请使用记事本程序在 autoexec.bat 文件中插入

```
SET PATH=C:\JDK1.4.0\BIN;
```

这句话可以放在任何 SET PATH 命令行之后。autoexec.bat 文件位于根目录下。关闭 MS-DOS 窗口，打开一个新的窗口以激活这个新路径。（在必要时修改版本号和目录名称。）

观看专题 applet

若通过 SDK 运行专题 applet，首先在 MS-DOS 中使用 cd 命令移动至所需的子目录下。例如，运行第 2 章“数组”中的数组专题 applet，移动至它所在的目录下：

```
C:\>cd javaapps
C:\javaapps>cd chap02
C:\javaapps\chap02>cd Array
```

然后使用 SDK 中的 appletviewer 工具运行 applet 的.html 的文件：

```
C:\javaapps\chap02\Array>appletviewer Array.html
```

applet 现在应该开始运行。（有时 applet 需要用一段时间来装载，所以请耐心等待。）applet 的外观与书中所示的屏幕截图应较接近，但不会完全一样，因为每一个 applet 观察程序和浏览器解释 HTML 和 Java 格式都有所不同。

正如我们所提到的，你可以使用大多数网页浏览器执行 applet。

专题 applet 的操作

每章中都对操作该章的专题 applet 进行了用法说明。一般来说，请记住在大多数情况下只需重复按一个按钮来执行一个操作。例如，在数组专题 applet 中，每按一下 Ins 按钮就会完成插入过程的一步。通常，每一步都会有一个信息出现在屏幕上，告知现在的状态。

请在结束某一特定操作后再点击其他的操作按钮。例如，点击查找按钮直至找到含有特定关键字的元素，随后可以看到 Press any button（按任意键）的信息，只有此时才可以转换至另一个操作，例如通过 Ins 键插入一个新的元素。

第 3 章“简单排序”和第 7 章“高级排序”的排序 applet 有一个 Step 按钮，使用它可以观察排序过程中的每一步。它们还有一个 Run 键，通过它可以快速观察排序过程而不需要按钮。点击 Run 键一次，就可以观察这些直条自动排序。如果要暂停的话，可以在任何时候点击 Step 按钮。当点击 Run 键后排序过程再次继续运行。

本书不倾向于研究专题 applet 的代码，因为其中大部分代码都是关于图形表示的，所以，本书也未提供源代码清单。

运行示例程序

每一个示例程序都有一个子目录，其中包含有一个.java 文件和一些.class 文件。.java 文件是源程序，它同时出现在书中，运行前必须先编译。.class 文件已经被编译好了，如果有 Java 解释器的话，可以直接运行。

可以使用 Sun 的 SDK 的 Java 解释器直接运行.class 文件的示例程序。对于每一个应用程序来说，.class 文件名称以 App 结尾。这种文件必须用 java 调用。

在 MS-DOS 提示符下，转到合适的子目录(使用 cd 命令)并且找到当前目录下的 App 文件。例如第 3 章中的 insertSort 程序，转到 InsertSort 子目录下。（不要将 applet 的目录和示例程序的目录搞混。）在这个目录下可以找到一个.java 文件和一些.class 文件。其中有一个是 insertSortApp.class。执行该文件，请输入

```
C:\chap03\InsertSort>java insertSortApp
```

不要在文件名后键入文件后缀。insertSort 程序应该开始运行，同时屏幕上由文本显示出未经排序的和已排序的数据。在某些示例程序中会有提示符提醒输入数据，则可以通过键盘输入。

编译示例程序

可以尝试修改示例程序，编译并运行修改过的版本。还可以试着编写自己的应用，编译并运行它。要编译一个 Java 应用，请使用 javac 程序来调用示例中的.java 文件。例如编译 insertSort 程序，应先转至 insertSort 目录下并输入

```
C:\chap03\insertSort>javac insertSort.java
```

请注意输入中加上了.java 后缀。这条命令会将.java 文件编译成与程序中类的数目一样多的.class 文件。如果源程序有错误，这些错会显示在屏幕上。

编辑源代码

许多文本编辑器都适用于修改或创建.java 的源程序。例如，可以从命令行调用一个名为 edit

的 MS-DOS 编辑器，或是 Windows 自带的记事本。许多商用的文本编辑器也可以用。

请不要使用那些太高档的文字处理软件来编辑源程序，例如 Microsoft Word 文字处理软件，它通常会在输出文件中加入一些古怪的字符和格式信息，而这些是 Java 解释器所不能理解的。

终止示例程序

通过按 Control-C 组合键（同时按下 Control 键和 C 键），可以终止任何运行在控制台模式下的程序，包括所有的示例程序。课文中谈到的一些示例程序有特殊的结束操作，例如在一行的开始按 Enter，但其他的必须按 Control-C 才能结束。

重名的类文件

若干专题 applet 或示例程序经常会使用相同名字的.class 文件，然而要注意这些文件不见得一定是相同的。applet 或示例程序若使用了错误的类文件则不能正常运行，即使这个文件的名字是正确的。

调用错误文件的问题并不常见，因为一个程序的所有文件都被放置在同一子目录下。然而如果手动移动文件，请注意不要漫不经心地将文件复制到错误的文件夹中。这种错误会引出许多问题，并且很难找出原因。

其他开发系统

除了 Sun 的 SDK 之外还有许多 Java 开发系统。例如 Symantec、Microsoft、Borland 等公司的软件都很好。Sun 自己有一个 Java 开发系统，称作 Sun ONE Studio 4（曾被称为 Forte）。这些软件通常比 SDK 更快更方便。它们基本上将所有的功能：编辑、编译和执行整合到一个窗口中。

为了执行本书的示例程序，这些开发系统应能处理 Java 1.4.0 或更近一些的版本。许多示例程序（尤其是包含有用户输入的那些）不能被 Java 的早期版本所编译。（然而经过第 1 章中提到过的微小修改，旧版本的系统就可以编译.java 文件了。）

附录 B
进一步学习

本附录重点

- 数据结构和算法
- 对象程序语言
- 面向对象设计(OOD)和软件工程

本附录将提到一些涉及软件开发不同方面的书籍，其中也包括数据结构和算法。下面是我个人提出的列表，其中列出了与本书讨论过主题有关的许多优秀书籍。

数据结构和算法

对于任何数据结构和算法的研究来说，最权威的参考书是：Standford 大学的 Donald E.Knuth 编写的《The Art of Computer Programming》（Addison Wesley,1998）。这套本源性的巨著在 20 世纪 70 年代就已出版，现在已经出到了第三版。它包括三卷（最近又有精装版出现）：《Volume 1:Fundamental Algorithms，Volume 2:Seminumerical Algorithms 和 Volume 3:Sorting and Searching》。在这三卷中，第三卷与本书中的内容最相关。这套专著中的数学很深，阅读起来并不容易，但它是所有沉湎于这个领域进行认真研究的人的圣经。

另一本读起来稍微容易一些的书是 Robert Sedgewick 的《Algorithms in C++》（Addison Wesley, 1998）。此书改编自早期用 Pascal 写的 Algorithms(Addison Wesley,1988)。它是一本综合且权威的专著。其中的文字和代码很简洁，需要读者认真阅读。Robert Sedgewick 和 Michael Schidlowsky 编写的《Algorithms in Java》（Addison Wesley,2002）用 Java 讨论了数据结构和算法。

另一本为本科生的数据结构和算法课程而写的好书是：《Data Abstraction and Problem Solving with C++:Walls and Mirrors》，它是由 Janet J. Prichard 和 Frank M. Carrano (Benjamin Cummings,2001) 编写的。该书中有许多插图，章末都有练习和编程作业（project）。该书的 Java 版本是：《Data Abstraction and Problem Solving with Java:Walls and Mirrors》，同样由 Frank M.Carrano 和 Janet J. Prichard 编写。

Bryan Flamig 编写的《Practical Algorithms in C++》（John Wiley and Sons,1995）除了涵盖通常的专题外，还包括了许多其他书所不常提到的问题，例如算法生成器（Algorithm generator）和字符串查找。

Jon Louis Bentley 编写的《Programming Pearls，Second Edition》[1]（Addison Wesley,1999）实际上最初在 1986 年就写成了，但其中包括了许多对于程序员来说至关重要的建议。书中的大部分都与数据结构和算法有关。

其他一些有关数据结构和算法的值得一读的著作有 Timothy A.Budd 编写的《Classic Data Structures in C++》（Addison Wesley,2000），Mark Allen Weiss 编写的《Data Structures and Problem

1 该书中文版《编程珠玑》（第二版）已由中国电力出版社出版。

Solving Using C++》（Addison Wesley,1999）和 Yedidyah Langsam 等编写的《Data Structures Using C and C++》（Prentice Hall,1996）。

面向对象程序语言

若想对 Java 语言和面向对象程序设计有一个全面且深入的了解，请参考 Stephen Gilbert 和 Bill McCarty 编写的《Object-Oriented Programming in Java》（Waite group Press,1997）一书。

如果对 C++有兴趣的话，可以参考 Robert Lafore 编写的《Object-Oriented Programming in C++,Fourth Edition》[2]（Sams Publishing,2001）。

Ken Arnold,James Gosling 和 David Holmes 编写的《The Java Programming Language,Third Edition》[3]（Addison Wesley,2000）讲解了 Java 的语法，它是一本绝对权威的书（尽管比许多其他的书简短），这是由于工作在 Sun 公司的 Gosling 是 Java 的创作者。

Cay S. Horstmann 和 Gary Cornell 编写的《Core Java 2,Fifth Edition》（Prentice Hall,2000）有许多卷，书中的理论知识较深，但它会使读者很容易就理解 Java 编程中的各个方面。

面向对象设计（OOD）和软件工程

若要对软件工程有一个轻松且非纯学院派的了解，请尝试阅读 Scott W. Ambler 编写的《The Object Primer: The Application Developer's Guide to Object-Orientation, Second Edition》（Cambridge University Press,2001）一书。这本书利用平常的语言解释了如何设计一个大型的应用程序。本书的书名有些用词不当，它超出了纯 OO 的概念。

Stephen Gilbert 和 Bill McCarty 编写的《Object-Oriented Design in Java》（Waite Group Press, 1998）是一本极易理解的书。

在 OOD 领域中的经典之作是 Grady Booch 编写的《Object-Oriented Analysis and Design with Applications》一书。该书作者是本领域的倡导者之一，还是用来描述类关系的 Booch 标记法（Booch notation）的发明人。该书对初学者来说不太容易，但对高级读者来说却是不可或缺的。

有关 OOD 的一本较早的书是 Frederick P. Brooks,Jr.编写的《The Mythical Man-Month》（Addison Wesley,1975,1995 再版），该书清晰且文学化地解释了优秀的软件设计的必要。这本书据说是世界上卖得最多的计算机类书籍。

其他关于 OOD 的好课本有 Timothy Budd 编写的《An Introduction to Object-Oriented Programming,Third Edition》（Addison Wesley,2002），Arthur J.Riel 编写的《Object-Oriented Design Heuristics》（Addison Wesley,1996）和 Erich Gamma 等编写的《Design Patterns:Elements of Reusable Object-Oriented Software》（Addison Wesley,1995）。

2 该书中文版《C++面向对象程序设计》（第四版）已由中国电力出版社出版。
3 该书中文版《Java 编程语言》（第三版）已由中国电力出版社出版。

附录C
问题答案

本附录重点

- 第1章 综述
- 第2章 数组
- 第3章 简单排序
- 第4章 栈和队列
- 第5章 链表
- 第6章 递归
- 第7章 高级排序
- 第8章 二叉树
- 第9章 红-黑树
- 第10章 2-3-4树和外部存储
- 第11章 哈希表
- 第12章 堆
- 第13章 图
- 第14章 带权图

第1章，综述

问题答案

1. 插入，查找，删除
2. 排序
3. c
4. 搜索关键字
5. b
6. a
7. d
8. 方法
9. 点
10. 数据类型

第2章，数组

问题答案

1. d
2. 真
3. b
4. 假
5. new
6. d
7. 接口
8. d
9 指数
10. 3
11. 8
12. 6

13. 假
14. a
15. 常量
16. 对象

第 3 章，简单排序

问题答案

1. d
2. 比较并交换（或复制）
3. 假
4. a
5. 假
6. b
7. 假
8. 三
9. 下标小于等于 outer 的项进行排序。
10. c
11. d
12. 复制（copies）
13. b
14. 下标小于 outer 的项部分有序。
15. b

第 4 章，栈与队列

问题答案

1. 10
2. b
3. 先进后出和先进先出
4. 假。正好相反。
5. b
6. 根本没有移动。
7. 45
8. 假。它们用了 O(1)时间。
9. c
10. O(N)
11. c

12. 真
13. b
14. 是，需要一个查找最小值的方法。
15. a

第 5 章，链表

问题答案

1. b
2. 第一个
3. d
4. 2
5. 1
6. c
7. current.net = null;
8. Java 的垃圾回收处理销毁了它。
9. a
10. 空
11. 一个链表。
12. 一次，如果链表中包含有一个 previous 引用
13. 双端链表
14. b
15. 经常是链表。它们的 push()和 pop()都是 O(1)时间级，但是链表使用内存空间更有效率。

第 6 章，递归

问题答案

1. 10
2. d
3. 2
4. 10
5. 假
6. “ed”
7. b
8. c
9. 分治
10. 用来搜索的单元范围
11. 用来传递的磁盘个数

12. c
13. b
14. b
15. 栈

第 7 章，高级排序

问题答案

1. c
2. 40
3. d
4. 假
5. O(N*logN)，$O(N^2)$
6. a
7. 枢轴
8. d
9. 真
10. c
11. 划分结果中的子数组
12. b
13. 枢轴
14. $\log_2 N$
15. 真

第 8 章，二叉树

问题答案

1. O(logN)
2. b
3. 真
4. 5
5. c
6. 节点，树
7. a
8. c
9. 查找
10. A，A 的左子孙
11. d

12. 2*n+1
13. 假
14. 压缩
15. c

第9章，红-黑树

问题答案

1. 顺序（或逆序）
2. b
3. 假
4. d
5. b
6. 旋转，改变节点颜色
7. 红
8. a
9. 左子节点，右子节点
10. d
11. 一个节点，它的两个子节点
12. b
13. 真
14. a
15. 真

第10章，2-3-4树和外部存储

问题答案

1. b
2. 平衡的
3. 2
4. 假
5. b
6. 根分裂
7. a
8. 2
9. 颜色变换
10. b
11. O(logN)

12. d
13. 许多
14. 真
15. a

第 11 章，哈希表

问题答案

1. O(1)
2. 哈希函数
3. d
4. 线性探测
5. 1，4，9，16，25
6. b
7. 链表
8. 1.0
9. 真
10. d
11. 数组大小
12. 假
13. a
14. 假
15. 同一块

第 12 章，堆

问题答案

1. b
2. 左右子节点都不大于父节点
3. 根
4. a
5. a
6. c
7. 数组（或链表）
8. 上
9. b
10. 一个

第13章，图

问题答案

1. 边，节点（或顶点）
2. 数出矩阵中1的个数并除以2即为所求。（假设主对角线上都为0。）
3. 节点
4. d
5. A:B,B:A->C->D,C:B,D:B
6. a
7. 3
8. c
9. 树
10. 不是
11. 真
12. d
13. 有向无环图
14. 不。由定义。
15. 有些顶点保留，但没有一个有后继节点。

第14章，带权图

问题答案

1. 边
2. d
3. 假
4. 权值最低（最便宜）的边
5. b
6. 已经是一个权值更低的边的终点。
7. 假
8. a
9. 真
10. a
11. b
12. 沃赛尔（Warshall）算法
13. 邻接矩阵
14. 2^N，N是板上的正方形的个数减一
15. 不是