

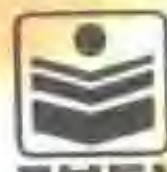
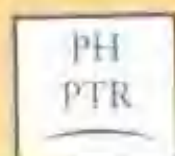
# 数据结构与算法分析

## (Java 版)

A Practical Introduction to Data Structures  
and Algorithm Analysis Java Edition

[美] Clifford A. Shaffer 著

张 铭 刘晓丹 译



电子工业出版社

Publishing House of Electronics Industry  
URL: <http://www.phei.com.cn>

国外计算机科学教材系列

# 数据结构与算法分析 (Java 版)

A Practical Introduction to Data Structures and Algorithm Analysis  
Java Edition

[美] Clifford A. Shaffer 著

张 铭 刘晓丹 译

電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING

## 内 容 提 要

作为《数据结构与算法分析(C++版)》的姊妹篇,本书采用了当前十分流行且适合于 Internet 环境的面向对象程序设计语言 Java 作为算法描述语言。本书利用 Java 的接口(Interface)来定义抽象数据类型,这比使用 C++ 的类更自然。本书把数据结构原理和算法分析技术有机地结合在一起,系统地介绍了各种类型的数据结构和排序、检索的各种方法。作者非常注意对每一种数据结构的不同存储方法及有关算法进行分析比较。本书还引入了一些比较高级的数据结构与先进的算法分析技术,并介绍了可计算性理论的一般知识。

本书概念清楚,逻辑性强,内容新颖,可作为大专院校计算机软件专业与计算机应用专业学生的教材和参考书,也可供计算机工程技术人员参考。

Authorized translation from the English language edition published by Prentice-Hall, Inc.

本书中文简体专有翻译出版权由美国 Prentice-Hall, Inc. 授予电子工业出版社。其原文版及中文翻译出版权受法律保护。未经许可,不得以任何形式或手段复制或抄袭本书内容。

Copyright (C) 1998 Prentice-Hall, Inc. All rights Reserved. No Part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Prentice-Hall, Inc.

### 图书在版编目(CIP)数据

数据结构与算法分析(Java 版)/(美)沙佛(Shaffer, C. A.)著;张铭译.—北京:电子工业出版社,2001.1

国外计算机科学教材系列

ISBN 7-5053-6497-9

I. 数... II. ①沙...②张... III. ①数据结构-教材②算法分析-教材 IV. TP311.12

中国版本图书馆 CIP 数据核字(2001)第 03483 号

丛 书 名: 国外计算机科学教材系列

书 名: 数据结构与算法分析(Java 版)

原 书 名: A Practical Introduction to Data Structures and Algorithm Analysis Java Edition

著 者: [美] Clifford A. Shaffer

译 者: 张 铭 刘晓丹

责任编辑: 吴 源

排版制作: 电子工业出版社计算机排版室监制

印 刷 者: 北京东光印刷厂

出版发行: 电子工业出版社 URL: <http://www.phei.com.cn>

北京市海淀区万寿路 173 信箱 邮编: 100036

经 销: 各地新华书店

开 本: 787×1092 1/16 印张: 21 字数: 537 千字

版 次: 2001 年 2 月第 1 版 2001 年 2 月第 1 次印刷

书 号: ISBN 7-5053-6497-9  
TP·3566

印 数: 10100 册 定价: 35.00 元

版权贸易合同登记号 图字: 01-2000-3484

凡购买电子工业出版社的图书,如有缺页、倒页、脱页、所附磁盘或光盘有问题者,请向购买书店调换。若书店售缺,请与本社发行部联系调换 电话 68279077

## 译者序

数据结构以及算法分析是计算机专业十分重要的基础课,计算机科学各领域及各种应用软件都要使用相关的数据结构和算法。

当面临一个新的设计问题时,设计者需要选择适当的数据结构并设计出满足一定时间和空间限制的有效算法。本书作者把数据结构和算法分析有机地结合在一本教材中,有助于读者根据问题的性质选择合理的数据结构,并对时间空间复杂性进行必要的控制。

本书采用当前十分流行的 Java 作为算法描述语言,Java 的接口(Interface)可以使抽象数据类型(ADT)的概念得到更自然的体现。而且随着网络技术的迅速发展,Java 这种平台无关且天生适合于因特网的面向对象程序设计语言得到了非常广泛的应用。广大 Java 程序员十分需要直接用 Java 语言描述经典数据结构和算法的技术书籍。

本书包括四大部分内容,第一部分是准备工作,介绍了一些基本概念和术语以及基本的数学知识。

第二部分介绍了最基本的数据结构,依次为线性表(包括栈和队列)、二叉树、树和图。每种数据结构都从其数学特性入手,先介绍抽象数据类型,然后再讨论不同的存储方法,并且研究不同存储方法的可能算法。

作为最常用的算法,排序和检索历来是数据结构讨论的重点问题,这在第三部分作了详尽的讨论。排序算法最能体现算法分析的魅力,它的算法速度要求非常高。第 8 章证明了所有基于比较的排序算法的时间代价是  $\Theta(n \log n)$ ,这也是排序问题的时间代价。检索则考虑怎样提高检索速度,这往往是与存储方法有关的。书中介绍了几种高效的数据结构,例如自组织线性表、散列表、B 树和 B<sup>+</sup> 树等,都具有极好的检索性能。

第四部分介绍了数据结构的应用与一些高级主题。例如跳跃表、广义表和稀疏矩阵等更复杂的线性表结构,Trie 结构、伸展树等复杂树结构,k-d 树、PR 四分树等空间数据结构。另外还简单介绍了求和、递归关系分析和均摊分析等高级算法分析技术。这些技术对于提高程序员的算法分析能力具有重要的作用。

附录部分介绍了对于理解本书例子必要的一些 Java,从而使得不懂 Java 语法的 C/C++ 和 Pascal 程序员能够更好地理解本书。作者提供的参考书目也颇有价值。

本书的前言及第 1 章至第 8 章由张铭翻译,第 9 章至第 15 章由刘晓丹翻译,肖毅、柴栗、肖之屏、刘莹等人参与了本书的核对工作,译者在此表示感谢。由于水平有限,难免有不妥之处,欢迎批评指正。



## 前 言

我们研究数据结构的目的是为了学会编写效率更高的程序。现在的计算机速度一年比一年快,为什么还需要高效率的程序?这是由于人类解决问题的雄心与能力是同步增长的。现代计算技术在计算能力和存储容量上的革命,仅仅提供了计算更复杂问题的有效工具,而程序的高效性要求永远也不会过时。

程序高效性的要求不会,也不应该与合理的设计和简明清晰的编码相矛盾。高效程序的设计基于良好的信息组织和优秀的算法,而不是基于“编程小技巧”。程序员如果没有掌握设计简明清晰程序的基本原理,就不可能编写有效的程序。反过来说,简洁的程序需要合理的数据组织和清晰的算法。大多数计算机科学系的课程设置都意识到要培养良好的程序设计技能,首先应该强调基本的软件工程原理。因此,一旦程序员学会了设计和实现简明清晰程序的原理,下一步就应该学习有效的数据组织和算法,以提高程序的效率。

**途径:**本书描述了许多表示数据的技术。这些技术包括以下原则:

1. 每一种数据结构和每一个算法都有其时间、空间的开销和效率。当面临一个新的设计问题时,设计者要透彻掌握怎样权衡时间、空间开销和算法有效性的方法,以适应问题的需要。这就需要懂得算法分析的原理,而且还需要了解所使用的物理介质的特性(例如,数据存储在磁盘上与存储在主存上,就有不同的考虑)。

2. 与开销和效率有关的是时空权衡。例如,人们通常增加空间开销来减少运行时间,或者相反。程序员所面对的时空权衡问题普遍存在于软件设计和实现的各个阶段,因此这个概念必须牢记在心。

3. 程序员应该充分了解一些现成的方法,以免作不必要的重复开发工作。因此,学生需要了解经常使用的数据结构和相关算法。

4. 数据结构服从于应用需求。学生必须把分析应用需求放在第一位,然后再寻找一个与实际应用相匹配的数据结构。要做到这一点,需要应用上述三条原则。

**组织:**数据结构和算法设计的书籍往往囿于下面这两种情形之一:一种是教材,一种是百科全书。有的书籍试图融合这两种编排,但通常是二者都没有组织好。本书是作为教材来编写的。我相信了解选择或设计解决问题的高效数据结构的基本原理是十分重要的,这比死记硬背书本内容重要得多。因此,我在本书中涵盖了大多数但不是所有的标准数据结构。为了阐述一些重要原理,也包括了某些并非广泛使用的数据结构。另外,还介绍了一些相对较新但即将得到广泛应用的数据结构。

本书可以作为本科生一个学期的教学内容,也可以作为专业技术人员的自学教材。读者应该具有编程经验,最好学过相当于两个学期的结构化程序设计语言 Pascal 或 C 语言。在第 2 章中,复习了一些数学预备知识,早已熟悉数学归纳法和递归的读者会容易掌握一些。

尽管本书应该一个学期完成,但书中超过了一个学期的内容,这可以为教师提供一些选择的余地。二年级学生的基本数据结构和算法分析背景不太多,可以对他们详细讲解第 1 章~第 12 章的内容,再从第 13 章中选择一些专题来讲解,我就是这样来给二年级学生讲课的。背

景知识更丰富的学生,可以先读第 1 章,跳过第 2 章中除参考书目之外的内容,简要地浏览第 3 章和第 4 章(请着重阅读 4.1.3 小节),然后详细阅读其余章节。另外,教师可以根据程序设计实习的需要,选择第 13 章以后的某些专题内容。

第 13 章是针对做较大的程序设计练习而编写的。我建议所有选修数据结构的学生,都应该做一些高级树结构或其他较复杂的动态数据结构的上机实习,比如第 12 章中的跳跃表(Skip List)或稀疏矩阵。所有这些数据结构都不比二叉检索树更难,而且学完第 5 章的学生都有能力来实现它们。

我尽量合理地安排内容顺序。教师可以根据需要自由地重新组织内容。读者掌握了第 1 章至第 6 章后,以下的内容就相对独立了。显然,外排序依赖于内排序和磁盘文件系统。Kruskal 最小支撑树算法使用了 6.2 节关于 UNION/FIND 的算法。10.2 节的自组织线性表提到了 9.3 节讨论的缓冲区置换技术。第 14 章的讨论基于本书的例题。15.3 节依赖于图论知识。一般情况下,大多数主题都只依赖于同一章中讨论过的内容。

**关于 Java:**本书的示例程序是用 Java 来写的。像其他程序设计语言一样,Java 有利有弊。Java 是一种小型语言,往往只有一种方法来解决某个问题。程序员只要正确使用 Java,就能得到清晰的程序结构。从这个角度来看,它比 C 或 C++ 优越。Java 可以很好地定义和使用大多数传统的数据结构,如线性表和树。另一方面,Java 的文件处理能力很弱,笨拙而低效,而且它的内存控制能力也很弱。例如,12.4 节讨论的存储管理,就很难用 Java 来编写。由于我希望全书能够使用同一种语言,所以像其他程序员一样,我只能接受 Java 的这些弱点,瑕不掩瑜嘛。最重要的是表达算法的思想,并不需要考虑这些思想是否适合某种具体的语言。大多数程序员将使用多种程序设计语言,本书所描述的概念将在不同的环境中被证明是有用的。

我并不想难倒那些对 Java 不熟悉的读者。在保持 Java 优点的同时,我尽量使示例程序简明、清晰。Java 在本书中只作为阐释数据结构的工具。值得庆幸的是,对于 C 或 Pascal 程序员而言,Java 是一种很容易掌握的语言,只需要学习很少的与面向对象程序设计有关的语法即可。特别是我用到了 Java 隐蔽实现细节的特性,例如类(class)、私有成员(private class member)和接口(Interface)。这些特性支持了一个关键的概念:体现于抽象数据类型(abstract data type)中的逻辑设计与体现于数据结构中的物理实现的分离。

我没有在本书中讲授 Java 语言的意图,只是提供了一个附录来解释一些基本的 Java 语义和概念,这对读懂示例程序是必要的。另外还提供了通过匿名 FTP 得到本书中 Java 源代码的途径。

本书很少使用继承(Inheritance)这一面向对象程序设计的关键特征。类的继承是避免重复编码和降低程序错误率的重要工具;但是从教育学的标准观点来看,类的继承在若干类中分散了数据元素的描述,从而使得程序更难理解。因此,我对于一些对象的类定义,比如线性表和树结点的定义,就没有充分继承前面示例编码中的类定义。这并不意味着程序员也应该这样做。避免代码重复和减少错误是很重要的目标,请不要把本书中的示例程序直接拷贝到自己的程序中,而只是把它们看作是对数据结构原理的阐释。

本书中的示例程序提供了有关数据结构原理的具体描述,而不是一系列具有商业质量的类实现。这些例子中所作的参数检查,比起从事商业软件设计的程序员在编程中所作的要少得多。某些参数检查是以调用类 Assert 中函数的形式包含进来的,这些函数模仿了 C 的标准库函数 assert。方法 Assert.notFalse 的输入是一个布尔表达式,一旦这个表达式的值为假

(false), 程序就立即终止。方法 `Assert.notNull` 的输入是类 `Object` 的一个引用(reference), 如果这个引用的值为空(null), 程序就会终止。更准确地说, 这些函数产生一种 `IllegalArgumentException` (非法参数异常)。除非程序员处理这个异常(exception), 否则将导致程序终止。当一个函数接收到非法参数时终止程序, 这种做法在实际程序中是不合适的, 但是对于理解一个数据结构怎样运作是十分有用的。在实际程序应用中, 应该使用 Java 的异常处理功能来处理输入数据错误。

在示例程序中, 我严格区分了“Java 实现”和“伪码”(pseudocode)。一个标明“Java 实现”的示例程序至少在一个编译器中被真正编译过。伪码的示例通常具有与 Java 接近的语法, 但是一般包含一行以上的更高级的描述。当我发现简单的、尽管并不十分精确的描述具有更好的教学效果时, 就使用伪码。

几乎每一章都是以“深入学习导读”一节来结束的。它并不是那一章的综合参考索引, 而是为了通过这些导读书籍或文章提供给读者更广泛的信息和乐趣。有些情况下我也提供了某个知名计算机科学家的重要背景文章。

**习题和项目设计:** 只靠读书是不能学会灵活使用数据结构的, 一定要通过编制实际的程序, 比较不同的数据结构技术来观察在一种给定的条件下哪一种结构更有效。另外, 学生也需要通过编程序来提高他们的算法分析与设计能力。这里提供了 300 多个习题和项目设计的程序实习题, 希望读者能够很好地利用它们。

**与作者联系的方法以及相关资料的获取:** 本书难免有一些错误, 有些方面还有待进一步研究。作者非常欢迎读者指正, 并提出建设性意见。作者在因特网(Internet)上的电子邮件(E-mail)地址是 `shaffer@cs.vt.edu`。也可以写信给以下地址:

Cliff Shaffer  
Department of Computer Science  
Virginia Tech  
Blacksburg, VA 24061  
USA

与本书有关的一套基于 LATEX 系统的幻灯片材料可以通过 FTP 的匿名帐号(anonymous), 从 `ftp.prenhall.com` 的目录中获得。该目录为:

`pub/esm/computer_science.s-041/shaffer/ds/supplements/transparencies`

例题的 Java 代码可以从相同 FTP 站点下面的目录获得:

`pub/esm/computer_science.s-041/shaffer/ds/code`

弗吉尼亚技术学院二年级数据结构课程的 WWW 主页(Home Page)之 URL 为:

`http://ci.cs.vt.edu/~cs2604`

该网址上同时可以看到一个针对数据结构的可视化和图形化调试工具, 称为 SWAN 系统。

本书是用 LATEX 的写作系统编排的, LATEX 是 TEX 系统的一个软件包。参考书目(bibliography)是用 BIBTEX 编排的, 词汇表用 `makeindex` 编写。图形主要是用 Xfig 来制作, 但图 3.1 和图 10.5 实验性地使用了 Mathematica。

**致谢:**本书得到了许多友人的帮助。我想特别感谢其中的几位,他们对本书的出版贡献最大。对于没有被提及的朋友,在此表示歉意。系主任 Jack Carroll 对本书给予了重要的精神上的帮助。弗吉尼亚技术学院在 1994 年秋季的学术休假中使得整个出书的事情成为可能,我是从那时开始着手准备的。Mike Keenan, Lenny Heath 和 Jeff Shaffer 对本书最初版本的内容提供了有价值的意见。尤其是 Lenny Heath 多年来一直与我深入讨论算法设计和分析的有关问题以及怎样把二者讲授给学生的方法。Layne Watson 提供了有关 Mathematica 的帮助, Bo Begole, Philip Isenhour 和 Craig Struble 提供了一些技术上的帮助。Steve Edwards, Mark Abrams 和 Dennis Kafura 回答了一些有关 C++ 和 Java 的问题。

对于许多评阅了本书初稿的朋友,本人欠情甚深。这些评阅者是:J. David Bezek (University of Evansville)、Douglas Campbell (Brigham Young University)、Karen Davis (University of Cincinnati)、Vijay Kumar Garg (University of Texas-Austin)、Jim Miller (University of Kansas)、Bruce Maxim (University of Michigan-Dearborn)、Jeff Parker (Agile Networks/Harvard)、Dana Richards (George Mason University)、Jack Tan (University of Houston) 和 Lixin Tao (Concordia University)。要不是他们的热心帮助,本书会出现更多技术上的错误,内容也将更肤浅。

没有 Prentice Hall 公司众多朋友的帮助,不可能有本书的出版,因为作者不可能自己印出书来。因此,我要感谢 Laura Steele 和 Alan Apt 这两位编辑。感谢本书 C++ 版的责任编辑 Kathleen Caren 和 Java 版的责任编辑 Ed DeFelipis,他们在本书接近出版的最紧张的日子里,保持各个方面运作良好。感谢 Bill Zobrist 和 Bruce Gregory 使我着手此事。感谢 Prentice Hall 的 Truly Donovan、Linda Behrens 和 Phyllis Bregman 在本书出版过程中提供的帮助。可能还有许多没有被提及的 Prentice Hall 公司的朋友,默默地提供了帮助。

我十分感激 Hanan Samet 传授给我有关数据结构的知识。我从他那里学到了许多原理,当然本书中可能的错误并不是他的责任。感谢我的妻子 Terry 对我的爱和支持。最后,也是最重要的是,要感谢这些年来选修数据结构的学生,是他们使我知道了在数据结构课程中什么是重要的而什么应该忽略,许多深入的问题也是他们提供的。这本书献给他们。

克利福德·沙佛 (Clifford A. Shaffer)

福吉尼亚州,布莱克斯堡 (Blacksburg, Virginia)



# 目 录

## 第一部分 预备知识

<b>第 1 章 数据结构和算法</b>	( 2 )
1.1 数据结构的原则	( 2 )
1.1.1 学习数据结构的必要性	( 2 )
1.1.2 代价与效益	( 3 )
1.1.3 本书的目的	( 4 )
1.2 抽象数据类型和数据结构	( 4 )
1.3 问题、算法和程序	( 6 )
1.4 算法的效率	( 8 )
1.5 深入学习导读	( 8 )
1.6 习题	( 9 )
<b>第 2 章 数学预备知识</b>	(11)
2.1 集合	(11)
2.2 常用数学术语	(12)
2.3 对数	(13)
2.4 递归	(14)
2.5 级数求和与递归	(16)
2.6 数学证明方法	(17)
2.6.1 反证法	(18)
2.6.2 数学归纳法	(18)
2.7 评估	(20)
2.8 深入学习导读	(21)
2.9 习题	(22)
<b>第 3 章 算法分析</b>	(25)
3.1 概述	(25)
3.2 最佳、最差和平均情况	(28)
3.3 换一台更快的计算机,还是换一种更快的算法	(29)
3.4 渐进分析	(31)
3.4.1 上限	(31)
3.4.2 下限	(32)
3.4.3 $\Theta$ 表示法	(33)
3.4.4 化简法则	(33)
3.5 程序运行时间的计算	(34)
3.6 问题的分析	(37)

3.7 多参数问题	(38)
3.8 空间代价	(39)
3.9 实际操作中的一些因素	(41)
3.10 深入学习导读	(42)
3.11 习题	(43)
3.12 项目设计	(45)

## 第二部分 基本数据结构

<b>第4章 线性表、栈和队列</b>	(48)
4.1 线性表	(48)
4.1.1 顺序表的表示法	(50)
4.1.2 链表	(53)
4.1.3 线性表实现方法的比较	(62)
4.1.4 元素的表示	(63)
4.1.5 双链表	(63)
4.1.6 循环链表	(66)
4.2 栈	(67)
4.2.1 顺序栈	(67)
4.2.2 链式栈	(69)
4.2.3 顺序栈与链式栈的比较	(70)
4.2.4 递归的实现	(70)
4.3 队列	(73)
4.3.1 顺序队列	(73)
4.3.2 链式队列	(76)
4.3.3 顺序队列与链式队列的比较	(77)
4.4 习题	(77)
4.5 项目设计	(79)
<b>第5章 二叉树</b>	(80)
5.1 定义及主要特性	(80)
5.1.1 满二叉树定理	(82)
5.1.2 二叉树的抽象数据类型	(83)
5.2 周游二叉树	(84)
5.3 二叉树的实现	(84)
5.3.1 使用指针实现二叉树	(84)
5.3.2 空间开销	(88)
5.3.3 使用数组实现完全二叉树	(89)
5.4 Huffman 编码树	(90)
5.4.1 建立 Huffman 编码树	(91)
5.4.2 Huffman 编码及其用法	(94)

5.5	二叉检索树	(96)
5.6	堆与优先队列	(102)
5.7	深入学习导读	(107)
5.8	习题	(108)
5.9	项目设计	(109)
<b>第 6 章</b>	<b>树</b>	<b>(111)</b>
6.1	树的定义与术语	(111)
6.1.1	树结点的 ADT(抽象数据类型)	(112)
6.1.2	树的周游	(112)
6.2	父指针表示法	(113)
6.3	树的实现	(118)
6.3.1	子结点表表示法	(118)
6.3.2	左子结点/右兄弟结点表示法	(119)
6.3.3	动态结点表示法	(120)
6.3.4	动态“左子结点/右兄弟结点”表示法	(121)
6.4	K 叉树	(121)
6.5	树的顺序表示法	(122)
6.6	深入学习导读	(124)
6.7	习题	(124)
6.8	项目设计	(125)
<b>第 7 章</b>	<b>图</b>	<b>(127)</b>
7.1	术语和表示法	(127)
7.2	图的实现	(129)
7.3	图的周游	(136)
7.3.1	深度优先搜索	(137)
7.3.2	广度优先搜索	(139)
7.3.3	拓扑排序	(139)
7.4	最短路径问题	(142)
7.4.1	单源最短路径	(142)
7.4.2	每对顶点间的最短路径	(145)
7.5	最小支撑树	(147)
7.5.1	Prim 算法	(147)
7.5.2	Kruskal 算法	(149)
7.6	深入学习导读	(151)
7.7	习题	(152)
7.8	项目设计	(153)

### 第三部分 排序和检索

<b>第 8 章</b>	<b>内排序</b>	<b>(156)</b>
--------------	------------	--------------

8.1	排序的术语及记号 .....	(156)
8.2	三种代价为 $\Theta(n^3)$ 的排序方法 .....	(157)
8.2.1	插入排序 .....	(157)
8.2.2	起泡排序 .....	(158)
8.2.3	选择排序 .....	(159)
8.2.4	交换排序算法的时间代价 .....	(160)
8.3	Shell 排序 .....	(161)
8.4	快速排序 .....	(163)
8.5	归并排序 .....	(168)
8.6	堆排序 .....	(171)
8.7	分配排序和基数排序 .....	(172)
8.8	对各种排序算法的实验比较 .....	(177)
8.9	排序问题的下限 .....	(178)
8.10	深入学习导读 .....	(181)
8.11	习题 .....	(181)
8.12	项目设计 .....	(184)
<b>第 9 章</b>	<b>文件管理和外排序</b> .....	(185)
9.1	主存储器和辅助存储器 .....	(185)
9.2	磁盘和磁带驱动器 .....	(187)
9.2.1	磁盘访问的代价 .....	(190)
9.2.2	磁带 .....	(192)
9.3	缓冲区和缓冲池 .....	(192)
9.4	程序员的文件视图 .....	(194)
9.5	外部排序 .....	(195)
9.6	外部排序的简单方法 .....	(197)
9.7	置换选择排序 .....	(199)
9.8	多路归并 .....	(201)
9.9	深入学习导读 .....	(203)
9.10	习题 .....	(204)
9.11	项目设计 .....	(205)
<b>第 10 章</b>	<b>检索</b> .....	(207)
10.1	检索已排序的数组 .....	(207)
10.2	自组织线性表 .....	(208)
10.3	集合的检索 .....	(211)
10.4	散列方法 .....	(212)
10.4.1	散列函数 .....	(213)
10.4.2	开散列方法 .....	(215)
10.4.3	闭散列方法 .....	(216)
10.5	深入学习导读 .....	(224)

10.6	习题 .....	(224)
10.7	项目设计 .....	(226)
<b>第 11 章</b>	<b>索引技术 .....</b>	<b>(227)</b>
11.1	线性索引 .....	(228)
11.2	ISAM .....	(230)
11.3	树形索引 .....	(231)
11.4	2-3 树 .....	(232)
11.5	B 树 .....	(238)
11.5.1	B <sup>+</sup> 树 .....	(239)
11.5.2	B 树分析 .....	(244)
11.6	深入学习导读 .....	(245)
11.7	习题 .....	(245)
11.8	项目设计 .....	(246)
 <b>第四部分 应用与高级话题</b> 		
<b>第 12 章</b>	<b>线性表和数组高级技术 .....</b>	<b>(250)</b>
12.1	跳跃表 .....	(250)
12.2	广义表 .....	(254)
12.3	矩阵的表示方法 .....	(256)
12.4	存储管理 .....	(259)
12.4.1	动态存储分配 .....	(259)
12.4.2	失败处理策略和无用单元收集 .....	(266)
12.5	深入学习导读 .....	(269)
12.6	习题 .....	(269)
12.7	项目设计 .....	(271)
<b>第 13 章</b>	<b>高级树形结构 .....</b>	<b>(272)</b>
13.1	Trie 结构 .....	(272)
13.2	伸展树 .....	(275)
13.3	空间数据结构 .....	(278)
13.3.1	k-d 树 .....	(279)
13.3.2	PR 四分树 .....	(282)
13.3.3	其他空间数据结构 .....	(284)
13.4	深入学习导读 .....	(285)
13.5	习题 .....	(286)
13.6	项目设计 .....	(286)
<b>第 14 章</b>	<b>分析技术 .....</b>	<b>(288)</b>
14.1	求和技术 .....	(288)
14.2	递归关系 .....	(290)
14.2.1	估计上下限 .....	(290)



14.2.2	扩展递归 .....	(291)
14.2.3	分治法递归 .....	(292)
14.2.4	快速排序平均情况分析 .....	(293)
14.3	均摊分析 .....	(294)
14.4	深入学习导读 .....	(296)
14.5	习题 .....	(296)
14.6	项目设计 .....	(298)
<b>第 15 章</b>	<b>计算的限制 .....</b>	<b>(299)</b>
15.1	简介 .....	(299)
15.2	归约 .....	(299)
15.3	难解问题 .....	(302)
15.3.1	NP 完全性 .....	(303)
15.3.2	绕过 NP 完全性问题 .....	(306)
15.4	不可解问题 .....	(306)
15.4.1	不可数性 .....	(307)
15.4.2	停机问题的不可解性 .....	(309)
15.4.3	确定程序行为是不可解的 .....	(310)
15.5	深入学习导读 .....	(311)
15.6	习题 .....	(311)
15.7	项目设计 .....	(312)
<b>附录 A</b>	<b>C 和 Pascal 程序员的 Java 导引 .....</b>	<b>(313)</b>
A.1	例 1:线性表的接口 .....	(313)
A.2	例 2:基于数组的线性表实现 .....	(314)
A.3	例 3:链表的实现 .....	(316)
<b>参考文献</b>	<b>.....</b>	<b>(319)</b>

# 第一部分 预备知识

第 1 章 数据结构和算法

第 2 章 数学预备知识

第 3 章 算法分析

# 第 1 章 数据结构和算法

信息的表示是计算机科学的基础。大多数计算程序的主要目标与其说是完成运算,倒不如说是存储和检索信息。从存储空间和运行时间的现实角度来看,这些程序必须组织信息,以支持高效的信息处理过程。因此,研究数据结构和算法以有效地支持程序的实现就成了计算机科学的核心问题

## 1.1 数据结构的原理

### 1.1.1 学习数据结构的必要性

有人可能认为,随着计算机功能的日益强大,程序的运行效率变得越来越不重要了。然而,计算机功能越强大,人们就越想去尝试更复杂的问题。而更复杂的问题需要更大的计算量,这就使得对高效程序的需求更加明显,工作愈复杂就愈偏离人们的日常经验。当今的计算机科学家必须训练出彻底理解隐藏在高效程序设计后面的一般原理的能力,因为他们的日常生活经验并不具备这些能力。

一般说来,一种数据结构就是一类普通数据的表示及其相关操作。从数据表示的观点来看,即使是存储在计算机中的一个整数或者一个浮点数也是一个简单的数据结构。更典型地,一个数据结构被认为是一组数据项的组织或者结构。存储在数组中的一个有序整数表就是这种结构的一个例子。

如果有足够的空间来存储一组数据项,总会有可能在这个数据项集合中查找出指定的数据项、打印数据项或将这些数据项处理成任何期望得到的顺序,或者更改任何特定数据项的值。因此,就有可能对任何数据结构施加所有必要的运算。然而,选择不同的数据结构可能会产生很大的差异:同样一个程序,可能在几秒钟内运行完毕,也可能需要几天时间才能完成运行。

毋庸置疑,人们编写程序是为了解决问题,这个道理本来不用作者再次提出。然而,在选择数据结构解决特定问题时,头脑中有这个不言而喻的道理是具有重要意义的。只有通过预先分析问题来确定必须达到的性能目标,才有希望挑选出正确的数据结构。有相当多的程序设计人员却忽视了这一分析过程,而直接选用某一个他们习惯使用的、但是与问题不相称的数据结构,结果设计出一个低效率的程序。相反,当使用简单的设计能够达到性能目标时,选用复杂的数据表示来改进这个程序也是没有道理的。

**定义 1.1** 一个算法如果能在所要求的资源限制(resource constraint)内将问题解决好,则称这个算法是有效率的(efficient)。例如,一个资源限制是:可用来存储数据的全部空间——可以分为内存空间限制和磁盘(外存)空间限制——和允许执行每一个子任务所需要的时间。一个算法如果比其他已知解所需要的资源都少,这个算法也可以称为是有效率的。一个算法的代价(cost)是指这个算法消耗的资源量。一般说来,代价是由一个关键资源例如时间来评估的,这意味着这个算

法满足其他资源限制

当为解决某一问题而选择数据结构时,应该完成以下几步:

1. 分析问题以确定任何算法均会遇到的资源限制。
2. 确定必须支持的基本运算,并度量每种运算所受的资源限制。基本运算的实例包括向数据结构中插入一个数据项、从数据结构中删除一个数据项和查找指定的数据项。
3. 选择最接近这些开销的数据结构。

根据这三个步骤来选择数据结构,实际上贯彻了一种以数据为中心的设计观点。先定义数据和对数据的操作,然后确定数据的表示方法,最后是数据表示的实现。

某些重要的操作,例如查找、插入和删除数据记录的资源限制通常决定了数据结构的选择过程。对于这些操作相对重要性的争论焦点集中在以下三个问题中,无论什么时候,只要选择数据结构,就应该仔细考虑这三个问题:

- 开始时将所有数据项都插入数据结构,还是与其他操作混合在一起插入?
- 数据项可以删除吗?
- 所有数据项是安排在某一个已经定义好的序列中,还是允许随机进入?

显然,非集中插入、允许删除数据项和支持数据项的随机进入都需要更复杂的表示方法。

### 1.1.2 代价与效益

每一个数据结构都将代价与效益联系在一起。如果有人说某个算法在所有情况下都比其他算法好,这通常是不正确的。因为本书提到的几乎每一个数据结构和算法,我们都会发现一些例子说明它在什么地方才是最好的选择,其中有些例子是出人意料的。

一个数据结构需要一定的空间来存储它的每一个数据项,一定的时间来执行单个基本操作,一定的程序设计工作。每一个问题都有可利用的空间和时间的限制。问题的每一个解决方案都利用了一定比例的相关基本操作,数据结构的选择过程必须考虑到这一点。只有对问题的特性仔细分析之后,才能得到执行这项任务的最好的数据结构。

**例 1.1** 设计一个支持银行顾客账户记录的数据库系统。这个数据库必须可以添加或者删除顾客的账户,并且可以让顾客查询账户信息以及存款取款。顾客愿意在开户或者销户时等上几分钟,却不愿意为个别账目业务(如存钱、取钱)等候几秒钟。

银行提供一种自动取款机(ATM)让顾客使用,以查询余额、存款或者取款。比较特殊的是,这些 ATM 业务并不太更改数据库(为简单起见,只是假设钱增加了或减少了,这项业务仅仅改变账户记录里存储的值)。向数据库中添加一个新账户需要花费几分钟的时间(一般的顾客并不经常开户,而且开户时顾客在银行经理办公室等候)。销户没有时间限制,因为从顾客的角度来说,重要的只是所有的钱被取回(跟取款过程相同);从银行的角度来说,账户记录可以在营业时间之后从数据库中删除。

一个删除效率很低、查找效率极高,并且具有适度插入效率的数据结构,应该符合上述问题所要求的资源限制。通过账号很容易取得账户记录(有时称之为“精确匹配”检索方法)。符合这些要求的数据结构就是在 10.4 节中描述的散列表。散列表具有非常快的精确匹配检索。当修改操作不影响记录长度时,记录可以很快被修改。散列表也支持新记录的高效率插入。散列表还能够支持高效率删除,但是删除得太多会导致其他操作性能的降低。然而,散列表可以定期重组,以将系统还原到最高效率状态。这种重组应该脱机进行,以免影响 ATM 业务。

### 1.1.3 本书的目的

本书有三个主要目的。第一个目的是讲授常用的数据结构,这些数据结构形成了一个程序员基本数据结构工具箱(toolkit)。对于许多问题,工具箱里的数据结构是理想的选择。

第二个目的是讲授并加强“权衡”(tradeoff)的概念,每一个数据结构都有相关的代价和效率的权衡。本书通过列举出不同的数据结构,并指出它们应用于实际问题时的代价和效率来讨论“权衡”的概念。

第三个目的是讲授如何评估一个数据结构或算法的有效性。只有通过这样的分析,才能够确定对于一个新问题其最合适的数据结构是工具箱中的哪一个。这种技术也使得能够判断自己或别人发明的新数据结构的价值。

## 1.2 抽象数据类型和数据结构

前面提到了术语“数据项”和“数据结构”,却没有给出它们的确切定义。这一部分介绍术语,有助于体现上述选择数据结构三个步骤的设计过程。

**定义 1.2** 类型(type)是一组值的集合。例如:布尔类型由 true 和 false 这两个值组成;整数也构成一个类型。数据类型(data type)是指一个类型以及定义在这个类型上的一组操作。例如,一个整数变量是整数数据类型的一个成员。数据项(data item)是一条信息或者其值属于某个类型的一条记录。数据项是数据类型的成员(member)。整数是简单数据项(simple data item),因为它不包含子结构。而银行的账户记录包含许多项信息,例如:姓名、地址、账号、余额。这样一条记录是复杂数据项(aggregate data item)的一个例子。

数据类型的描述与它在计算机程序中的实现有很重要的区别。例如,实现线性表数据类型有两种传统的数据结构:链表(linked list)和顺序表(array-based list,基于数组的线性表)。因此,可以在链表或者数组之间选择一种来实现线性表数据类型。但是“数组”(array)是一个模棱两可的概念,因为它既可以指一种数据类型,又可以指一种实现方式。“数组”在计算机程序设计中常用来指一块连续的内存空间,每一个内存空间存储一个固定长度的数据项。从这个意义上讲,数组是一种特殊的数据结构。然而,数组也能够表示一个由一组结构相同的数据项组成的数据类型,每一个数据项由一个特定的索引号(即数组中的下标)来标识。这样看来,数组可以采用多种不同的方法来实现。例如:12.3 节中用来实现稀疏矩阵(只有极少非零元素的大型二维矩阵)的数据结构就和传统的占用连续内存空间的数组大不相同。

**定义 1.3** 抽象数据类型(abstract data type,简称 ADT)是指基于一个逻辑类型的类型的数据类型以及这个类型上的一组操作。每一个操作由它的输入和输出定义。一个 ADT 的定义并不涉及它的实现细节,这些实现细节对于 ADT 的用户是隐藏的。隐藏实现细节的过程称为封装(encapsulation)。数据结构(data structure)是 ADT 的物理实现。ADT 的每一个操作均由一个或者多个子程序来实现。术语“数据结构”常指存储在计算机内存中的数据。与其相关的术语文件结构(file structure)常指外存储器(如磁盘驱动器、磁带)中数据的组织。



**例 1.2** 整数的数学概念和施加到整数的运算构成一个 ADT。Java 的变量类型 `int` 就是对这个抽象整型的一种物理实现。遗憾的是,由于 `int` 型变量有一定的取值范围,所以它对这个抽象整型的实现并不完全正确。如果无法接受这些限制,就必须引进一些其他的实现方法来实现这个抽象整型。

**例 1.3** 一个整数线性表的 ADT 应包含下列操作:

- 把一个新整数插入到线性表的结尾。
- 按线性表元素的顺序依次打印整数。
- 如果线性表中存在一个特殊的整数,则返回 `true`, 否则返回 `false`。
- 删除线性表中特定位置上的整数。

通过上述描述,每个操作的输入输出都清晰可见,但是线性表的实现还未详细说明。

**例 1.4** 驾驶汽车的主要操作有控制方向盘、加速和刹车。几乎所有的汽车都通过转动方向盘控制方向,踩油门加速,踩车闸刹车。汽车的这种设计以及“控制方向盘”、“加速”、“刹车”的操作可以看作是一个 ADT。两辆汽车可以用截然不同的方式实现这些操作,但是大多数司机都能够驾驶许多不同的汽车,因为 ADT 提供了一致的操作方法。

任何成功的计算机科学家都懂得一个重要原理:将复杂的问题抽象化。ADT 的概念就是这样的一个例子。计算机科学的主题是问题的复杂性和处理它的技术。为了解决复杂性,人们首先给一个物体或概念集合赋予一个称号(label),然后用这个称号代替其实体来执行有关的操作。我认识的一位心理学家称这样的一个称号为隐喻(metaphor)。一个特定的称号可能与其他信息或者其他称号有关,这个集合被依次分配给一个称号,从而形成概念和称号的层次。称号的这种层次结构能够使我们重视主要问题而忽略不必要的细节。

**例 1.5** 称号“硬盘驱动器”指在某种类型的存储设备上处理数据的硬件集合。称号“CPU”指控制计算机指令执行的硬件。这两个称号以及其他一些称号合起来从属于称号“计算机”。由于即使小型家用电脑也要由数百万个部件组成,所以在弄清楚计算机是怎样进行工作之前,一些抽象的形式是十分必要的。

想像一个处理 ADT 的复杂计算机程序。ADT 通过一种特定的数据结构在程序的某个部分得以实现,而在设计使用 ADT 的那部分程序时,我们只关心这个数据类型上的操作,而不关心数据结构的实现。因此,在思考一个复杂的程序时,如果不能将它简化,就没有希望理解或者实现它。

**定义 1.4** 数据项有逻辑形式(logical form)和物理形式(physical form)两个方面。用 ADT 给出的数据项的定义是它的逻辑形式,数据结构中对数据项的实现是它的物理形式。

**例 1.6** 某个 Java 编译器可能会提供管理整数线性表的库函数。线性表的逻辑形式由函数集和它们的输入、输出来定义(即一个 ADT)。从本质上讲,可以用各种物理形式来实现整数线性表。4.1 节描述了线性表的几种物理实现。

图 1.1 说明了数据项的逻辑形式和物理形式之间的关系。

本书的一些章节侧重于对给定数据结构的一种或各种物理实现,而有些章节则用逻辑 ADT 来实现高层任务中的数据类型。为了区分数据类型的逻辑 ADT 和作为数据结构的物理实现,本书给出的程序实例用 Java 语言编写。作为一种面向对象的程序设计语言,Java 提供

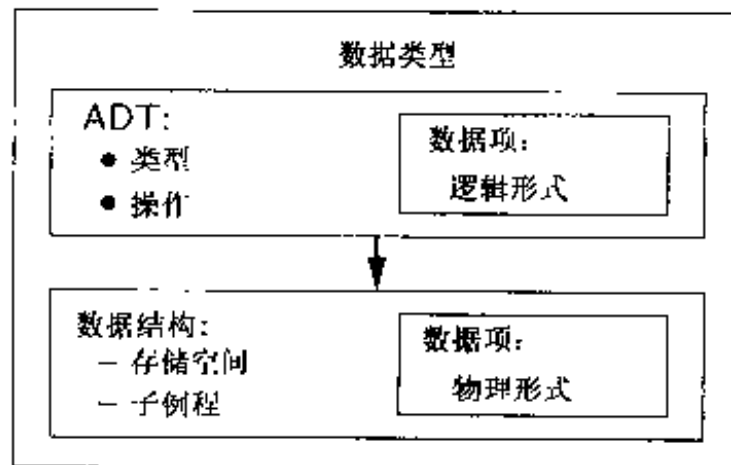


图 1.1 数据项、抽象数据类型和数据结构的关系。ADT 定义了数据类型的逻辑形式,数据结构是实现数据类型的物理形式

了许多特性来支持封装。只要读者具备结构化程序设计语言(如 Pascal 或 C)的编程经验,就会很容易理解本书中的程序实例。如果由于对 Java 语言不熟悉而在阅读程序时遇到困难,可以仔细阅读附录。附录介绍了本书中所用到的 Java 语法和概念。它虽然不能教会读者如何使用 Java 语言编写程序,但是对于理解本书中的 Java 程序实例是足够的。早已熟悉 Java 语言的读者可以略过附录。

### 1.3 问题、算法和程序

本书是关于数据结构和相关算法的,而程序设计人员总是需要与问题、算法和计算机程序打交道。这是三个不同的概念。

**问题(Problem):**从直觉上讲,问题无非是一个需要完成的任务,即对应一组输入就有一组相应的输出。问题的定义不能包含有关怎样解决问题的限制。只有在问题被准确定义并完全理解后才能研究问题的解决方法。然而,问题的定义应该包含对任何可行方案所需资源的限制。对于计算机要解决的任何一个问题,总有一些直接的或者间接的资源限制。例如,任何计算机程序只能使用可用的主存储器和磁盘空间,而且必须在合理的时间内完成运行。

从数学角度讲,可以把问题看作函数。

**定义 1.5** 函数(function)是输入(即定义域, domain)和输出(即值域, range)之间的一种映射关系。函数的输入可以是一个值或者一些信息,这些值组成的输入称为函数的参数(parameter)。不同的输入可以产生不同的输出,但是对于给定的输入,每次计算函数得到的输出必须相同。

这种将问题看作函数的概念可能不符合我们对计算机程序行为的直觉。我们知道,在不同的两种情况下,给程序输入同样的值可能得到两个不同的输出结果。例如,在许多计算机上键入命令“date”可以得到当前的日期。当然,即使给出的是同样的命令,不同的日子所得到的日期也不同。然而,日期程序的输入显然比运行这个程序所键入的命令要多,日期程序执行的是一个函数。在任意一个确定的日子里,运行程序 date 所得到的结果是惟一的。也就是说,对于一个完全确定的输入,正确运行日期程序只能得到惟一的结果。对于所有的计算机程序,输出完全由程序的整个输入决定,即使是“随机数生成器”,也完全由它的输入决定(尽管一些随机函数生成系统从表面上看是一个不受用户控制的物理过程,是从一个物理随机数发生器

接受随机输入的)。程序和函数的关系将在 15.4 节进一步研究。

**算法(algorithm):** 算法是指解决问题的一种方法或者一个过程。如果将问题看作函数,那么算法就是把输入转化为输出。一个问题可以用多种算法来解决,一个给定的算法解决一个特定的问题(例如,计算一个特殊函数)。本书涉及了许多问题,对于其中几个问题给出了不止一种算法: 对于重要的排序问题,差不多给出了 12 种算法。

知道一个问题多种解法的好处在于换一种解法可能对问题的几个特定变量更有效,或者对同一个问题的不同输入更有效。例如,有的排序算法适合于数目较少的序列,有的算法适合于数目较多的序列,而有的算法则适合于可变长度的字符串。

一个算法应该包含以下几条性质:

1. 正确性(correct)。也就是说,它必须完成所期望的功能,把每一次输入转化为正确的输出。
2. 具体步骤(concrete steps)。一个算法应该由一系列具体步骤组成。“具体”意味着每一步所描述的行为对于必须完成算法的人或机器是可读的、可执行的。每一步必须在有限的时间内执行完毕。因此,算法好像给我们一个通过一系列步骤解决问题的“工序”,其中的每一步都是我们力所能及的,是否能够完成每一步依赖于谁或者什么来执行这个工序。例如,烹饪书中关于小甜饼的制作方法对于指导一位厨师是足够具体的,但是对于一个自动小甜饼加工工厂是不够的。
3. 确定性(no ambiguity)。下一步(通常是指算法描述中的下一步)应执行的步骤必须明确。选择语句(例如 Java 中的 if 和 switch 语句)是任何算法描述语言的组成部分,它允许对下一步执行的语句进行选择,但是选择过程必须是确定的。
4. 有限性(finite)。一个算法必须由有限步骤组成。如果一个算法的描述是由无限步组成的,我们就不可能将它写出来,也不可能将它作为计算机程序来实现。大多数算法描述语言(包括英语或“伪码”)均提供一些实现重复行为的方法,如循环。例如 Java 中的 while 和 for 循环结构。循环结构具有简短的描述,但是实际执行的次数由输入来决定。
5. 可终止性(terminable)。算法必须可以终止,即不能进入死循环。

**程序(program):** 一个计算机程序被认为是使用某种程序设计语言对一个算法的具体实现。本书中几乎所有算法都给出了全部程序或者部分程序。当然,由于使用任何一种现代计算机程序设计语言都可以实现任何一个算法,所以可能有许多程序都是同一个算法的实现(尽管一些程序设计语言可能使得编程人员的工作容易一些)。本书在下面的内容中为了简化表达,常常混用“算法”和“程序”,尽管它们实际上是互相独立的两个概念。定义一个算法时,必须提供足够多的细节,以便必要时转化为程序。

算法必须可终止意味着不是所有的计算机程序都是算法。操作系统是一个程序,而不是一个算法。然而,我们可以把操作系统的各种任务看成是一些单独的问题,每一个问题由一部分操作系统程序通过某个算法来实现,得到输出结果后便终止。

**定义 1.6** 问题(problem)是一个函数,或者是从输入到输出的一种映射。算法(algorithm)是一个能够解决问题的、有具体步骤的方法。算法步骤必须无二义性,算法必须正确,长度有限,必须对所有输入都能终止。程序(program)是算法在计算机程序设计语言中的实现。

## 1.4 算法的效率

一个问题有多种解法,选择哪一种呢? 计算机程序设计的核心有两个目标(有时它们互相冲突):

1. 设计一个容易理解、编码和调试的算法。
2. 设计一个能有效利用计算机资源的算法。

在理想情况下,达到这两个目标的最终程序都是正确的,有时也说是“完美的”。本书给出的算法和程序实例在这种意义上讲是趋于完美的。与目标 1 有关的问题不是本书的目的,它们主要涉及到的是软件工程原理。而本书主要讲的是与目标 2 有关的问题。

怎样度量效率呢? 第 3 章将给出估算一个算法或者一个计算机程序效率的方法,称为算法分析(algorithm analysis)。它还可以度量一个问题的内在复杂程度。以后的章节中只要介绍算法就会用到算法分析方法。用这种方法可以清楚地看到在解决同一个问题时不同算法在效率上的差异。

## 1.5 深入学习导读

数据结构和算法方面最早的权威性著作是 Donald E. Knuth 所著的系列丛书《The Art of Computer Programming》(《计算机程序设计技巧》),其中第 1 卷和第 3 卷是有关数据结构的研究[Knu73, Knu81]。Robert Sedgewick 著的《Algorithms》(《算法》)[Sed 88]采用现代百科全书的方式组织,如果已经掌握了数据结构和算法的基本原理,这本书比较容易理解。Udi Manber 所著的《Introduction to Algorithms: A Creative Approach》(《算法介绍:一种创造性的方法》)[Man 89],是一部优秀的、可读性较强的高级著作,它介绍了算法、算法设计和算法分析。Cormen、Leiserson 和 Rivest 合著的《Introduction to Algorithms》(《算法概论》)[CLR 90],采用了百科全书式的组织方法,内容比较先进。

《Abstract Data Types: Their Specification, Representation, and Use》(《抽象数据类型的定义、表示和使用》)[TRE 88],由 Thomas、Robinson 和 Emms 著,介绍了 ADT 和程序规范说明。

所有现代程序设计语言可以实现任何算法(准确地讲应该是任何一个对于某种程序设计语言是可计算的函数,对于其他任何具备标准功能的程序设计语言也是可计算的),这是可计算性理论的一条重要结论。Lewis 和 Papadimitriou 合著的《Elements of the Theory of Computation》(《计算理论的要素》)[LP 81],在这方面作了介绍。

许多人致力于计算机科学中的问题求解,实际上这正是它吸引许多人进入这个领域的地方。George Pólya 著的《How to Solve it》(《如何解决问题》)[Pól 57]被认为是提高问题求解能力的经典著作。另一本值得推荐的书是 James L. Adams 著的《Conceptual Blockbusting》[Ada 79]。

Julian Jaynes 著的《The Origin of Consciousness in the Breakdown of the Bicameral Mind》[Jay 90],仔细讨论了如何利用隐喻解决复杂性问题(complexity)。

数据结构可以满足高层次程序设计的需要,许多人学习数据结构是为了编写出更好的程序。要想使程序正确、有效地运行,首先,这个程序对于自己和合作者应该是可理解的。

Kernighan 和 Plauger 著的《The Elements of Programming Style》(《程序设计风格的要素》)[KP 78]介绍了怎样养成良好的程序设计风格。Frederick P. Brooks 著的《The Mythical Man-Month: Essays on Software Engineering》[Bro 75]介绍了编写大程序的困难,该书十分出色,而且富有娱乐性。

最后,要成为一个成功的 Java 程序设计人员,手边应该有几本好的参考书。Winston 和 Narasimhan 合著的《On to Java》(《关于 Java》)[WN96]是一本介绍 Java 基本内容的好书。David Flanagan 的《Java in a Nutshell》[Fla96]给那些有一定 Java 基础的读者提供了很好的参考

## 1.6 习题

本章与本书其他章节的习题不同之处在于本章大部分习题在后面的章节中得到了解答。但是请你不要到后面的章节中找答案,这些习题的目的就是要让你思考一些后面将要讨论的问题,并用当前你所知道的知识尽可能地回答。

- 1.1 从你以前用过的程序中找出一个慢得无法接受的程序,找出使程序速度慢的个别操作和使程序执行得足够快的其他基本操作。
- 1.2 大多数程序设计语言有内建的整数数据类型。在正常情况下这种表示方法有固定的长度(表示一个整数所用的位数),因此限制了整型变量值的大小。请给出一种无长度限制(除计算机可用内存的限制之外)的整数表示方法,这样存储的整型变量大小就不受限制了。请用这种表示方法简要地说明如何实现加、乘、指数操作。
- 1.3 为字符串定义一个 ADT,要求包含字符串的典型操作,每一个操作定义为一个函数,每一个函数由它的输入、输出来定义。
- 1.4 为一个整数线性表定义一个 ADT。它包含线性表常用的操作(一个函数对应一个操作),每一个函数由它的输入、输出定义。
- 1.5 为一个整数集合定义一个 ADT(注意集合中无重复元素)。它包含整数集合的常见运算,每一个运算对应一个函数,每一个函数由它的输入、输出定义。
- 1.6 简要概括整型变量在计算机上是如何表示的(如果对这些不熟悉,可以在计算机科学入门教材或者《数字逻辑》中查阅“反码”和“补码”)。为什么整数的这种表示方法符合定义 1.3 中所定义的数据结构?
- 1.7 为二维整数数组定义一个 ADT,详细而准确地说明可以在此数组上完成的操作。然后,试着将它用于一个 1 000 行、1 000 列的数组,其中非零元素远少于 10 000 个。为这个数组设计两种不同的实现方法,使它们比使用标准的二维数组所需要的 100 万( $1\,000 \times 1\,000$ )个位置的实现方法节省空间。
- 1.8 每一个问题都有一个算法吗?
- 1.9 设计一个在家用电脑上运行的拼写检查程序,它能够快速处理少于 20 页的文献。假设这个程序有一个大约 20 000 字的以 ASCII 码形式存储的字典。这个字典必须实现的原语操作是什么?每一个操作的合理时间限制是多少?
- 1.10 设想你被雇用去设计一个包含美国城市和乡村信息的数据库服务系统,有成千个城市和乡村。此数据库程序应当允许用户按照名字检索某一个地方的信息,用户根据



位置或者人口等属性的某一个特定值或某一范围内的值来查找满足条件的所有地方。按照用户应该看到的操作,描述系统的基本功能,并列出每个操作的时间限制。

- 1.11 假设有一组记录,按照每个记录都包含的一些关键码字段排序。给出两种不同的方法检索有特定关键码值的记录,你认为哪种更好,为什么?
- 1.12 怎样比较两种对一组整数进行排序的算法? 特别地,
  - (a) 作为比较两种排序算法的基础,用什么代价度量比较合适?
  - (b) 在这些代价度量方式下,用什么测试方法来判断这两种算法的性能?
- 1.13 编译器和文本编辑器的一个普遍问题是判断一个字符串中的圆括号(或其他括号)是否平衡并恰好匹配。例如,字符串“((( )))”中的圆括号平衡且恰好匹配,但是字符串“())(”中的圆括号不平衡,字符串“( )”中的圆括号不匹配。
  - (a) 给出一个算法,当字符串中的圆括号恰好平衡且匹配时返回 true,否则返回 false。提示:从左到右扫描一个合法的字符串,保证任何时候所遇到的右圆括号不比左圆括号多。
  - (b) 给出一个算法,如果字符串中圆括号不平衡或者不匹配,则返回字符串中第一个非法圆括号的位置。也就是说,如果发现一个多余的右圆括号,则返回它的位置;如果有多个左圆括号,则返回第一个多余的左圆括号的位置;如果字符串平衡且恰好匹配,则返回 -1。
- 1.14 一个图由顶点集和边集组成,每条边连接两个顶点,任意一对顶点间只能连一条边。至少给出两种不同的方法,在计算机中表示由图的顶点和边的定义所确定的连接关系,你的表示方法要能用来确定给定一对顶点间是否有边相连。
- 1.15 尽可能地写出你能想到的对 1000 个数进行排序的方法。其中哪一种(些)最好?

## 第2章 数学预备知识

本章介绍了本书所使用的数学符号、背景知识和方法。它主要用于复习和参考。在后面的章节中遇到不熟悉的符号或数学方法时,可以学习本章的相关部分。

2.7节介绍的评估对于许多读者来说可能比较陌生。评估不是一种数学方法,而是一种以前可能没有遇到过的工程上的通用技巧。评估对于计算机科学家的设计工作有很大用途,因为对于任何可能方案,如果它的估计资源需求超出了问题的资源限制,就可以立即放弃。

### 2.1 集合

数学意义上的集合概念在计算机科学上有广泛的用途。

**定义 2.1** 集合(set)是由互不相同的成员(member)或者元素(element)构成的一个整体。成员取自一个更大的范围,称为基类型(base type)。集合的每个成员或者是基类型的一个基本元素(Primitive element),或者它本身也是一个集合。集合中没有重复的概念,取自基类型的每个值要么在集合内要么不在集合内。

例如,集合  $R$  由整数 3、4、5 组成,此时  $R$  的成员是 3、4 和 5,  $R$  的基类型是整型。依赖于集合的基类型,它的成员经常有一个线性顺序。

**定义 2.2** 线性顺序(linear order)有以下性质:

1. 对于集合  $S$  中的任意两个元素  $a$  和  $b$ ,三个表达式  $a < b$ 、 $a = b$ 、 $b < a$  中有且只有一个为真。
2. 对于集合  $S$  中的任意三个元素  $a$ 、 $b$ 、 $c$ ,如果  $a < b$ 、 $b < c$ ,那么  $a < c$ ,这称为传递性(transitivity)。

有线性顺序的基类型实例包括整型、字符型和实型。但是,如果一个集合是由水果组成的,那么在苹果和桔子之间并不存在一个被普遍接受的相对顺序的标准。如果需要人为地给它们规定一个顺序,可以使用某些程序设计语言中支持的枚举类型。

图 2.1 给出了表示集合和它们之间关系的常用符号。下面给出使用这种表示方法的实例。首先定义两个集合  $R$  和  $S$ :

$$R = \{2, 3, 5\}, \quad S = \{5, 10\}$$

则有  $|R| = 3$  (因为  $R$  有 3 个成员),  $|S| = 2$  (因为  $S$  有 2 个成员)。  $R$  和  $S$  的并(union),记为  $R \cup S$ ,是由  $R$  或  $S$  中的元素组成的集合,为  $\{2, 3, 5, 10\}$ 。  $R$  和  $S$  的交(intersection),记为  $R \cap S$ ,是由既在  $R$  中又在  $S$  中的元素组成的集合,为  $\{5\}$ 。  $R$  和  $S$  的差(difference),记为  $R - S$ ,是由在  $R$  中但不在  $S$  中的元素组成的集合,为  $\{2, 3\}$ 。注意:总有  $R \cup S = S \cup R$ ,  $R \cap S = S \cap R$ ,但通常  $R - S \neq S - R$ ,本例中  $S - R = \{10\}$ 。

一个与集合有关的概念是序列。

$\{1, 4\}$	由数字1和4组成的集合
$\{x x \text{ 是正整数}\}$	使用规范形式定义的集合 例: 所有正整数集合
$x \in A$	$x$ 是集合 $A$ 的一个元素
$x \notin A$	$x$ 不是集合 $A$ 中的元素
$\emptyset$	空集
$ A $	基数: 集合 $A$ 的大小或者元素数目
$A \subseteq B, B \supseteq A$	集合 $A$ 包含在集中 $B$ 中 集合 $A$ 是集合 $B$ 的一个子集 集合 $B$ 是集合 $A$ 的一个超集
$A \cup B$	集合的并: 所有属于集合 $A$ 或者集合 $B$ 的元素
$A \cap B$	集合的交: 所有属于集合 $A$ 而且属于集合 $B$ 的元素
$A - B$	集合的差: 所有属于集合 $A$ 而不属于集合 $B$ 的元素

图 2.1 集合的表示方法

**定义 2.3** 长度为  $n$  的一个有限序列 (finite sequence) 是定义域为集合  $\{0, 1, \dots, n-1\}$  的一个函数  $f$ 。这个定义的含义与集合的不同之处为:

- (i) 序列的元素有一个顺序 (第 0 个成员, 第 1 个成员, 等等)。
- (ii) 成员可以重复, 但是它们是序列的不同成员 (因为  $f(i)$  可能与  $f(j)$  相等)。

## 2.2 常用数学术语

**计量单位:**按照 IEEE 规定的表示法标准, 字节缩写为“B”, 位缩写为“b”, 兆字节 ( $2^{20}$  字节) 缩写为“MB”, 千字节 ( $2^{10} = 1024$  字节) 缩写为“KB”, 毫秒 ( $1/1000$  秒称为 1 毫秒) 缩写为“ms”。数字与以 2 为底数的缩写单位之间不应该有空格。因此, 硬盘的大小为 540 兆字节 ( $1 \text{ 兆字节} = 2^{20} \text{ 字节}$ ) 就记为“540MB”。如果单位是以 10 为底数缩写的, 那么它与数字之间应该有空格。因此, 2000 位记为“2 Kb”, 而“2Kb”代表 2048 比特位。2000 毫秒记为“2000 ms”。本书中度量大量存储空间一般都使用 2 作为底数, 很少使用 10 作为底数。

**阶乘函数 (factorial function):**阶乘函数  $n!$  是指从 1 到  $n$  之间所有整数的连乘, 其中  $n$  为大于 0 的整数。因此,  $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$ 。特别地,  $0! = 1$ 。阶乘函数随着  $n$  的增大迅速增长。由于直接计算阶乘函数非常耗时, 所以有时使用一个公式来做近似计算是非常有用的。Stirling 近似公式  $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$ , 其中  $e \approx 2.71828$  ( $e$  是自然对数的底数)<sup>①</sup>。

**排列 (permutation):**一个序列的排列就是把这个序列的成员按照一定的顺序组织起来。

<sup>①</sup> 符号“ $\approx$ ”表示“约等于”。

例如,数字 1 到  $n$  的排列可以把这些数字按照任意顺序放置。如果一个序列有  $n$  个不同的成员,那么这个序列有  $n!$  种不同的排列。因为排列中的第一个成员有  $n$  种选择方法;对于每个选定的第一个成员,第二个成员有  $n-1$  种选择方法;依此类推。有时候需要得到某个序列的一个随机排列(random permutation),也就是说, $n!$  种可能排列中的任何一种被选中的概率都相同。下面是一个产生随机排列的简单 Java 函数。序列的  $n$  个值存储在数组  $A$  的元素  $A[0]$  到  $A[n-1]$  中,函数  $\text{swap}(A, i, j)$  在数组  $A$  中交换元素  $i$  和元素  $j$  的值。函数  $\text{Dsu-til.random}(n)$  返回一个 0 到  $n-1$  之间的整数,程序如下:

```
static void permute(Object[] A) {
    for (int i = A.length; i > 0; i--) // for each i
        swap(A, i, Dsu-til.random(i)); // swap A[i] with
    } // a random element
```

**布尔变量(Boolean Variable):**布尔变量是一个只能取值为 true 或 false 的变量(Java 中的 boolean 型变量)。这两个值常常分别与值 1 和 0 对应,这样规定并没有任何特殊的理由。依靠 0 与 false 的对应关系进行编程练习是非常愚蠢的(事实上,Java 不允许这样做)。

**取下整和取上整(floor and ceiling):**实数  $x$  的取下整函数(floor)(记为  $\lfloor x \rfloor$ ) 返回不超过  $x$  的最大整数。例如,  $\lfloor 3.4 \rfloor = 3$ , 与  $\lfloor 3.0 \rfloor$  的结果相同。实数  $x$  的取上整函数(ceiling)(记为  $\lceil x \rceil$ ), 返回不小于  $x$  的最小整数。例如,  $\lceil 3.4 \rceil = 4$ , 与  $\lceil 4.0 \rceil$  的结果相同。Java 中等价的库函数为  $\text{Math.floor}$  和  $\text{Math.ceil}$ 。

**取模操作符(modulus, 或者 mod):**取模函数返回整除后的余数。有时在数学表达式中用  $n \bmod m$  表示,在 Java 中取模操作符的表示为  $n \% m$ 。从余数的定义可知,  $n \bmod m$  得到一个整数  $r$ , 满足  $n = qm + r$ , 其中  $q$  为一个整数, 且  $0 \leq r < m$ 。也可以将取模结果表示为  $n - m \lfloor n/m \rfloor$ 。  $n \bmod m$  的结果一定在 0 到  $m-1$  之间。例如,  $5 \bmod 3 = 2$ ,  $25 \bmod 3 = 1$ ,  $5 \bmod 7 = 5$ ,  $5 \bmod 5 = 0$ ,  $-3 \bmod 5 = 2$ 。最后一个实例所给出的结果是以数学上取模的定义为基础得到的。在某些程序设计语言中,取模操作符的任何一个操作数为负,运算结果将随着编译器的不同而不同。本书中取模操作的所有操作数均为正。

## 2.3 对数

以  $b$  为底  $y$  的对数(logarithm)定义为使得  $b$  的某次幂等于  $y$  的那个指数,记为  $\log_b y = x$ 。从而可得:

$$\log_b y = x \Leftrightarrow b^x = y \Leftrightarrow b^{\log_b y} = y$$

其中符号“ $\Leftrightarrow$ ”表示“等价于”。

编程人员经常使用对数,它有两个典型的用途。第一,许多程序需要对一些对象进行编码,那么表示  $n$  个编码至少需要多少位呢? 答案为  $\lceil \log_2 n \rceil$  位。例如,如果要存储 1000 个不同的编码,至少需要  $\lceil \log_2 1000 \rceil = 10$  位(10 位可以产生 1024 个不同的可用编码)。第二,对数普遍用于分析把问题分解为更小子问题的算法。在一个线性表中查找指定值所使用的二分法检索就是这样一种算法。二分法检索首先与中间元素进行比较,以确定下一步是在上半部分进行查找还是在下半部分进行查找,然后继续将适当的子表分半,直到找到指定的值(二分法

检索在 3.5 节详细描述)。一个长度为  $n$  的线性表被逐次分半,直到最后的子表中只有一个元素,一共需要分多少次呢? 答案是  $\log_2 n$  次。

本书中用到的对数几乎都以 2 为底,这是因为数据结构和算法总是把事情一分为二,或者用二进制位来存储编码。本书中用  $\log n$  表示  $\log_2 n$ ,任何不以 2 为底的对数都会把底数清楚地写出来

对于任意正数  $m, n, r$ , 任意正整数  $a$  和  $b$ , 对数有下列性质:

1.  $\log nm = \log n + \log m$
2.  $\log n/m = \log n - \log m$
3.  $\log n^r = r \log n$
4.  $\log_a n = \log_b n / \log_b a$

前两个性质表明两个数相乘(或相除)的对数,等于两个数分别取对数再相加(或相减)<sup>①</sup>。性质 3 是对性质 1 的推广。性质 4 表明对于变量  $n$  和任意两个整数变量  $a$  和  $b$ ,  $\log_a n$  与  $\log_b n$  只相差常数因子  $\log_b a$ ,而与  $n$  的值无关。本书中的大多数代价分析都忽略常数因子。性质 4 表明这种分析与对数的底数无关,因为它们对整体开销只是改变了一个常数因子。

在讨论对数时,容易与指数混淆。性质 3 告诉我们  $\log n^2 = 2 \log n$ 。那么对数的平方应该如何表示呢? 应该记为  $(\log n)^2$ ,也习惯记为  $\log^2 n$ 。同样, $n$  的对数的对数应该记为  $\log \log n$ 。

## 2.4 递归

如果一个算法调用自己来完成它的部分工作,就称这个算法是递归的(recursive)。这种方法要想取得成功,必须在比原始问题小的问题上调用自己。总而言之,一个递归算法必须有两个部分:初始情况(base case)和递归部分。初始情况只处理可以直接解决而不需要再次递归调用的简单输入。递归部分包含对算法的一次或者多次递归调用,每一次的调用参数都在某种程度上比原始调用参数更接近初始情况。下面给出一个计算  $n!$  的 Java 递归函数。对一个较小的  $n$ ,4.2.4 小节给出了这个函数的执行过程。

```
static long fact(int n) : // Must have n < 21 to fit in long
    Assert.notFalse((n >= 0) && (n < 21), "Input out of range");
    if (n <= 1) return 1; // Base case: return base solution
    return n * fact(n-1); // Recursive call for n > 1
}
```

函数的前两行构成初始情况。如果  $n \leq 1$ ,那么初始情况计算出问题的一个解<sup>②</sup>,否则, fact 调用一个知道如何得到  $(n-1)!$  的函数。当然,函数 fact 本身就能做这件事! 递归算法

---

① 这些性质是形成计算尺的基础。两个数相加可以看作把两个长度连接起来,测量它们的总长,而相乘却不是那么容易做的。但是,如果先把这些数转化为它们的对数,然后相加,最后对得到的结果取对数的逆,就可以得出相乘的结果(这正是对数的性质 1)。计算尺计算出来的是数的对数长度,你只是滑动木条把这些长度累加起来,最后对累加的结果取对数的逆,就可得到正确结果。

② 函数 Assert.notFalse 仿照标准 C++ 库函数 assert, 它的输入参数是一个布尔表达式。如果这个表达式的值为 false, Assert.notFalse 会使程序终止;如果表达式的值为 true,那么 Assert.notFalse 什么也不做。函数 Assert.notNull 对一个 Object 引用进行类似的操作。



的设计总能够使用下面的方法实现。首先写出初始情况,然后考虑通过组合一个或多个较小但是类似子问题的结果来解决问题。如果你编写的算法是正确的,那么你当然可以依靠它来递归地解决更小的子问题。还有什么方法更简单呢?

递归方法在日常生活的问题求解中没有类似的概念。由于它需要使用一种新的思维方式来考虑问题,所以这个概念很难掌握。为了有效使用递归,必须强制自己尽量不用非递归方法来思考问题。只要在不超出递归调用的范围内分析递归过程,子问题就会迎刃而解。

阶乘函数的递归实现看上去并不需要那么复杂,因为用一个 while 循环就可以达到同样的效果。下面给出的另一个实例基于著名的“汉诺塔”(Tower of Hanoi)问题。它的 Java 实现有多个递归调用,它不是那么容易就能够使用 while 循环改写的。

汉诺塔问题首先给出 3 根柱子和  $n$  个圆盘,所有圆盘均在最左边的柱子上(记为柱 1)。各个圆盘之间大小不同,按照大的圆盘在下面的顺序依次往上堆放,如图 2.2(a)所示。问题是要通过一系列步骤把这些圆盘从最左边的柱子上移到最右边的柱子上(记为柱 3)。每一步只能把某个柱子最上面的一个圆盘移到另一个柱子的上面,圆盘移到哪根柱子上不受限制,但是任何一个圆盘都不能放到比它小的圆盘的上面。

怎样解决这个问题呢?如果不费劲地去考虑细节,这个问题是非常容易的。只要考虑所有的圆盘都必须从柱 1 移到柱 3 上,因此必须首先把最下面(最大)的圆盘移到柱 3 上。要达到这个目的,柱 3 必须是空的,而且柱 1 上只能有最下面的一个圆盘,因此其余的  $n-1$  个圆盘只能在柱 2 上,如图 2.2(b)所示。

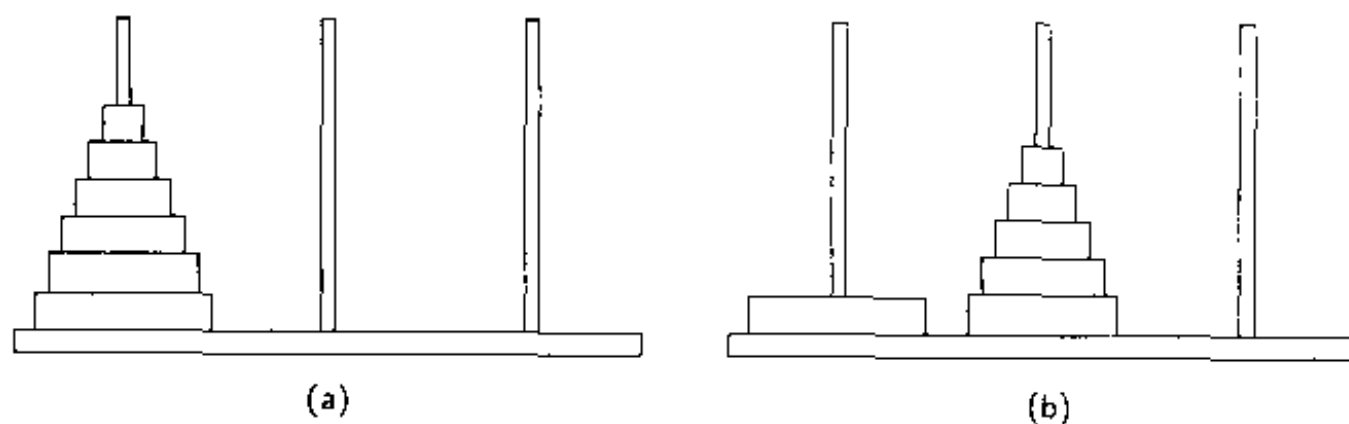


图 2.2 汉诺塔示例

(a)这个问题有 6 个圆盘时的初始状态。(b)得到解的过程中必须经过的一个中间步骤

这该如何实现呢?假设  $X$  是一个函数,可以把柱 1 上面的  $n-1$  个圆盘移动到柱 2 的上面,然后把柱 1 最下面的一个圆盘移动到柱 3 上;最后,再用函数  $X$  把其余的  $n-1$  个圆盘从柱 2 移动到柱 3 上即可。在这两种情况中,“函数  $X$ ”只不过是一个调用更小问题的汉诺塔函数而已。

成功的秘密在于汉诺塔算法为我们做了这些工作。我们不必关心汉诺塔的子问题如何解决这些细节,只要做好两件事,问题就迎刃而解了。第一,必须有一个初始情况(如果只有一个圆盘怎么做),以便递归过程不会永远进行下去。第二,对汉诺塔问题的递归调用只能用来解决更小的问题,而且只有一种正确的形式(一种满足汉诺塔问题初始定义的形式,假定对柱子适当地重命名)。

下面给出汉诺塔递归算法的一种 Java 实现,函数 `move(start, goal)` 把柱 `start` 最上面的圆盘移动到柱 `goal` 上。如果函数 `move` 的功能是打印它的参数值,那么递归调用 TOH 的结果将

给出解决此问题的圆盘移动序列。

```
static void TCH(int n, Pole start, Pole goal, Pole temp) {
    if (n == 0) return;          // Base case
    TCH(n-1, start, temp, goal); // Recursive call : n-1 rings
    move(start, goal);           // Move bottom disk to goal
    TCH(n-1, temp, goal, start); // Recursive call : n-1 rings
}
```

把递归作为一种主要用于设计和描述简单算法的工具,对于不熟悉它的编程人员是很难接受的。递归算法通常不是解决问题最有效的计算机程序,因为递归包含函数调用,比其他替代选择诸如 while 循环等,所花费的代价更大。但是,递归通常提供了一种能合理有效地解决第3章中所讨论问题的算法(但不总是,请参看习题2.3)。需要的时候,可以对递归方法进行修改,以便更快地实现算法,这部分内容在4.2.4小节将作进一步讨论。

## 2.5 级数求和与递归

大多数程序都具有循环结构,当分析带有循环程序的运行时间开销时,我们需要把每次循环执行的时间累加起来,这就是一个级数求和(summation)的例子。级数求和,简单地讲,就是把函数在一定范围内取的值加起来,它一般采用下面的“ $\sum$ ”表示法:

$$\sum_{i=1}^n f(i)$$

这个记号表示对作用于某个(整数)范围内的值  $i$  的函数  $f(i)$  之值求和,表达式的参数和它的初值写在  $\sum$  符号的下面。这里,记号  $i=1$  表明参数是  $i$ ,它的初值是1。 $\sum$  符号的上面是表达式  $n$ ,表示参数  $i$  的最大值。因此,这种表示法表示当  $i$  从1变到  $n$  时对  $f(i)$  的值求和,也可以把它写成:

$$f(1) + f(2) + \cdots + f(n-1) + f(n)$$

有时  $\sum$  表示法的排版形式是写在同一行的,如  $\sum_{i=1}^n f(i)$ 。

给出一个级数求和,我们希望用一个能直接计算级数求和的等式来代替它,这样的等式称为“闭合形式解”(closed form solution)。例如,级数求和  $\sum_{i=1}^n 1$  就是把数值“1”累加  $n$  次(注意  $i$  从1变到  $n$ )。由于  $n$  个1的和为  $n$ ,所以这个级数求和的闭合形式解为  $n$ 。下面给出本书中出现的所有级数求和以及它们的闭合形式解。

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \quad (2.1)$$

$$\sum_{i=1}^n i^2 = \frac{2n^3 + 3n^2 + n}{6} \quad (2.2)$$

$$\sum_{i=1}^{\log n} n = n \log n \quad (2.3)$$

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \quad (0 < a < 1) \quad (2.4)$$

作为公式(2.4)的特例,有:

$$\sum_{i=1}^n \frac{1}{2^i} = 1 - \frac{1}{2^n} \quad (2.5)$$

和

$$\sum_{i=0}^n a^i = \frac{a^{n+1} - 1}{a - 1} \quad (a > 1) \quad (2.6)$$

另一种形式为:

$$\sum_{i=1}^n \frac{i}{2^i} = 2 - \frac{n+2}{2^n} \quad (2.7)$$

作为公式 2.6 的特例,有:

$$\sum_{i=0}^n 2^i = 2^{n+1} - 1 \quad (2.8)$$

作为公式 2.8 的推论,有:

$$\sum_{i=0}^{\log n} 2^i = 2^{\log n + 1} - 1 = 2n - 1 \quad (2.9)$$

最后,从 1 到  $n$  的倒数之和称为调和级数(Harmonic Series),记为  $H_n$ ,它的近似闭合形式解如下:

$$H_n = \sum_{i=1}^n \frac{1}{i}; \quad \log_e n < H_n < 1 + \log_e n \quad (2.10)$$

这些等式中的大多数容易由数学归纳法证明(见 2.6.2 小节),但是数学归纳法不能帮助我们推出闭合形式解。推出闭合形式解的方法将在 14.1 节进行讨论,那些方法的原理与数据结构和算法分析无关。

递归算法的运行时间最易于用递归表达式来表示,因为它包括了运行递归调用的时间。递归关系(recurrence relation)用一个表达式定义了一个函数,这个表达式包括它本身的一个或者多个(更小的)实例。一些经典的实例包括阶乘函数的递归定义:

$$n! = (n-1)! \cdot n \quad 1! = 0! = 1$$

和 Fibonacci(斐波那契)序列:

$$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2); \quad \text{Fib}(1) = \text{Fib}(2) = 1$$

从这个定义可以得到 Fibonacci 序列的前 7 个数是:

$$1, 1, 2, 3, 5, 8 \text{ 和 } 13$$

这个定义包含两部分: $\text{Fib}(n)$ 的一般定义和初始情况  $\text{Fib}(1)$ 与  $\text{Fib}(2)$ 。同样,阶乘函数的定义也包括递归部分和初始情况。

递归关系经常用来计算递归函数的开销。例如, $n$ 个圆盘的汉诺塔算法需要移动圆盘的次数为  $T(n) = 2T(n-1) + 1$ 。因为要解决这个问题,必须先解决  $n-1$ 个圆盘的问题,做一次移动,然后再解决另外一次  $n-1$ 个圆盘的问题。

就级数求和而言,我们希望用它的闭合形式解来代替递归关系。本书中使用了少量递归关系,而且使用递归关系时给出了相应的闭合形式解。寻找递归关系的闭合形式解在 14.2 节进行讨论,这些方法与数据结构和算法分析无关。

## 2.6 数学证明方法

这部分简要介绍本书中最常用的两种证明方法:反证法和数学归纳法。

### 2.6.1 反证法

推翻一个定理或者命题的最简单方法就是找一个反例。然而,支持一个定理的实例再多也不足以证明这个定理的正确性。反证法(Proof by contradiction)是一种类似于使用反例进行反证的方法。要使用反证法证明一个定理,我们首先假设这个定理是错误的,然后找出由这个假设导致的逻辑上的矛盾。如果寻找矛盾的逻辑是正确的,那么惟一解决矛盾的方法就是纠正我们所做的定理错误的假设,即定理是正确的。

**例 2.1** 下面是使用反证法的简单证明。

**定理 2.1** 没有最大的整数。

**证明:**反证法。

**第 1 步,反面假设:**假设存在一个最大整数,记为  $B$ 。

**第 2 步,由该假设导出矛盾:**考虑  $C = B + 1$ ,因为  $C$  是两个整数的和,所以  $C$  也是整数,而且  $C > B$ ,因此导出矛盾。我们推理过程中的惟一漏洞就是开始时假设定理是错误的,从而得出结论:定理是正确的。

### 2.6.2 数学归纳法

数学归纳法(Proof by Mathematical Induction)很像递归,它对许多定理都适用。归纳法还提供了--种考虑算法设计的有用方法,因为它促使人们从简单的子问题入手考虑问题的求解。

设  $T$  是一个要证明的定理,用一个正整数参数  $n$  来表达  $T$ 。数学归纳法表明,如果下面两个条件为真,那么对于参数  $n$  的任何值,  $T$  都是正确的(对于  $n \geq c$ ,  $c$  是一个较小的常量):

1. **初始情况(Base Case):**  $n = c$  时  $T$  成立。

2. **归纳步骤(Induction Step):** 如果  $n - 1$  时  $T$  成立,则  $T$  对于  $n$  也成立。

证明初始情况通常很容易,只需用一些较小的值(如 1)代替定理中的  $n$ ,然后应用一些简单的代数或简单的逻辑来证明定理即可。证明归纳步骤有时容易有时难。强归纳法(strong induction)的归纳步骤有所变化,为:

2a. **归纳步骤:** 如果对于所有满足条件  $c \leq k < n$  的  $k$ ,  $T$  都成立,则  $T$  对于  $n$  也成立。

这种强归纳法,保证归纳过程的每一步都是正确的(即初始情况对于不同的值都成立),然后得出一个完满的证明。

把构成归纳法的两个条件合起来表明  $n = 2$  时  $T$  成立是对事实  $n = 1$  时  $T$  成立的扩展,再把这个事实和条件 2 或 2a 结合起来就得出  $n = 3$  时  $T$  也成立,依此类推。因此,只要证明了这两个条件,就有  $T$  对所有的  $n$  值都成立。

数学归纳法如此强大(而且许多初学者都觉得神秘)的原因是:可以充分利用  $T$  对所有小于  $n$  的值都成立的假设来帮助我们证明  $T$  对  $n$  也成立。这个假设称为归纳假设(induction hypothesis)。有了这个假设,归纳步骤的证明要比直接处理定理容易一些。

递归和归纳法有相似之处,它们都是由一个或者多个初始情况来终止的。递归函数能够调用它自己得到的此问题更小实例的解。同样,归纳法证明依靠归纳假设的事实来证明定理。

**例 2.2** 下面是一个用数学归纳法证明的例子。

**定理 2.2** 前  $n$  个自然数的和为  $n(n+1)/2$ 。

**证明:**数学归纳法。

1. 检查初始情况。 $n=1$ 时,和显然为1,公式的值为 $n=1, 1(1+1)/2=1$ 。因此,对于初始情况公式成立。

2. 提出归纳假设。归纳假设为:

$$\sum_{i=1}^{n-1} i = \frac{(n-1)((n-1)+1)}{2} = \frac{(n-1)(n)}{2}$$

3. 利用 $n-1$ 的归纳假设,说明结果对于 $n$ 也是正确的。归纳假设说明由于 $\sum_{i=1}^{n-1} i = (n-1)(n)/2$ ,且由于 $\sum_{i=1}^n i = \sum_{i=1}^{n-1} i + n$ ,因此可以推导出:

$$\sum_{i=1}^n i = \sum_{i=1}^{n-1} i + n = \frac{(n-1)(n)}{2} + n = \frac{n^2 - n + 2n}{2} = \frac{n(n+1)}{2}$$

所以,由数学归纳法得出 $\sum_{i=1}^n i = n(n+1)/2$ 。

**例 2.3** 下面给出另一个使用归纳法简单证明的例子,它说明了要为归纳法选择一个合适的变量。我们想要证明前 $n$ 个正奇数的和为 $n^2$ 。首先要有一种描述第 $n$ 个奇数的方法,即 $2n-1$ 。这样我们就可以得出一个级数求和的定理。

**定理 2.3**  $\sum_{i=1}^n (2i-1) = n^2$

**证明:**由 $n=1$ 的初始情况得 $1=1^2$ ,结论正确。归纳假设为:

$$\sum_{i=1}^{n-1} (2i-1) = (n-1)^2$$

现在用归纳假设来说明定理对于 $n$ 成立。前 $n$ 个奇数的和等于前 $n-1$ 个奇数的和加上第 $n$ 个奇数。即:

$$\begin{aligned} \sum_{i=1}^n (2i-1) &= \left( \sum_{i=1}^{n-1} (2i-1) \right) + 2n-1 \\ &= (n-1)^2 + 2n-1 \\ &= n^2 - 2n + 1 + 2n - 1 \\ &= n^2 \end{aligned}$$

因此由数学归纳法得出 $\sum_{i=1}^n (2i-1) = n^2$

**例 2.4** 下面的例子使用了归纳法,但不涉及级数求和,它说明了初始情况的一种更灵活的定义。

**定理 2.4** 用2分和5分的邮票可以构成任何票面金额的邮资(对于金额 $\geq 4$ )

**证明:**首先注意定理所定义的问题是对于金额 $\geq 4$ 的情况成立,而对于1和3都不成立。所以用4作为初始情况,4分的金额能由两张2分邮票组成。归纳假设金额为 $n-1$ 时可以由2分和5分邮票组合而成。现在用归纳假设说明如何推出金额为 $n$ 的组合。金额为 $n-1$ 的组合中或者包含5分邮票,或者不包含。如果包含5分邮票,则用3个2分邮票来代替。如果不包含,那么它的组合中至少包含两张2分的邮票(因为金额至少是4,而且只含有2分邮票),这种情况下用一张5分邮票代替两张2分邮票。无论哪种情况,我们都可以得到金额为 $n$ 的邮资可以由2分和5分邮票组成。因此,由数学归纳法推导出定理正确。

**例 2.5** 下面是一个使用强归纳法的实例。

**定理 2.5** 对于所有大于1的整数 $n$ , $n$ 能被某个素数整除。

**证明:**  $n = 2$  为初始情况, 2 可以被素数 2 整除。归纳假设对于所有的值  $a, 2 \leq a < n, a$  能被某个素数整除。为了证明定理对于  $n$  成立, 下面要考虑两种情况。如果  $n$  是一个素数, 那么  $n$  可以被它自己整除。如果  $n$  不是素数, 则  $n = a \times b$ , 其中  $a, b$  均小于  $n$  且大于 1。由归纳假设  $a$  可以被某个素数整除, 从而  $n$  也可以被这个素数整除。因此, 由数学归纳法可知, 定理成立。

**例 2.6** 数学归纳法的最后一个例子证明一个几何定理。它也说明了一种归纳法证明的技术, 即原本有  $n$  个物体, 为了使用归纳假设, 要任意去掉一个物体。

“着双色”(two-coloring)是指已知一个区域集以及两种颜色, 要为每一个区域分配一种颜色, 使得有共享边的区域不同色。例如, 国际象棋棋盘就是一个“着双色”的图。图 2.3 显示了有三条线的平面的“着双色”, 假设两种颜色为黑和白。

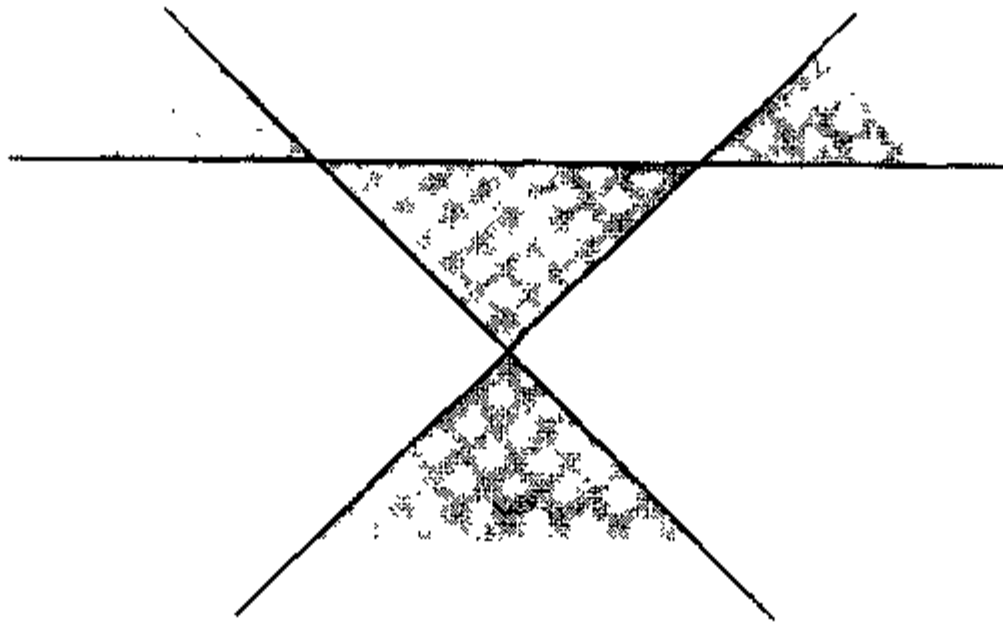


图 2.3 平面上由三条线形成的区域的“着双色”

**定理 2.6** 由平面上的  $n$  条直线形成的区域集可以实现“着双色”。

**证明:** 初始情况为平面上有一条直线。它将平面分为两个区域, 一个区域着黑色, 另一个区域着白色就得到一个合法的“着双色”。归纳假设是  $n - 1$  条直线形成的区域集可以被“着双色”。为了证明定理对  $n$  也成立, 我们先考虑删去  $n$  条直线中的任意一条, 由剩下的  $n - 1$  条直线形成的区域集。由归纳假设, 这个区域集可以被“着双色”。现在把第  $n$  条线放回来, 它把平面分为两个半平面, 每一个(互相独立)都实现了合法的“着双色”。但是, 被第  $n$  条直线分割的区域却违反了“着双色”的规则。因此, 我们把在第  $n$  条直线一侧的所有区域的颜色都取反, 现在被第  $n$  条直线分割的区域也符合“着双色”的规则了, 因为这些区域在第  $n$  条直线一侧的部分为黑色, 另一侧的部分为白色。因此, 由数学归纳法可得整个平面实现了“着双色”。

## 2.7 评估

你可以从计算机科学训练中获得的最有用的生活技能之一, 就是怎样对一个问题的求解进行快速评估(estimate)。有时也称作“餐巾纸背面计算”(back of the napkin calculation)或者“信封背面计算”(back of the envelope calculation), 这两个别名都表明只需要进行粗略的评估。



评估技术是工程学课程的基本内容之一,但是在计算机科学中却常常被忽视。它不能代替对一个问题的严格细节分析,但是当严格的分析被保证时,可以用它来指示:如果评估表明一个方法不可行,那么进一步的分析可能就没有必要了。

评估可以被形式化为以下三步:

1. 确定影响问题的主要参数。
2. 推导出一个与问题的参数有关的公式。
3. 选择参数值,由该公式得出一个评估解。

为了确保自己的评估是合理的,最好使用两种不同的方法进行评估。总的来说,如果想知道一个系统到底怎样,可以直接对它进行评估,也可以估计系统的输入参数是什么(假设输入系统的参数,一定有相应的结果)。如果两种(相互独立的)方法得出同样的结果,那么你对于自己的评估能力一定会信心倍增。

评估时一定要保证计量单位的统一,例如,不要把英尺和英镑相加。一定要验证结果单位的正确性。时刻记住一次计算的结果只与本次的输入参数有关。第3步输入的参数值越不确定,输出的值也就越不确定。然而,“信封背面”的计算经常只意味着得到一个大致正确或者50%正确的结果即可。因此在评估之前,应该规定一个误差允许的范围,如10%以内、50%以内等等。一旦评估值落在误差允许范围内就不用再管它了!如果没有必要,就不用再费劲去得到一个更精确的值。

**例 2.7** 下面给出一个“信封背面”计算的实例。要存放共有100万页的书籍需要多少个图书馆书架?我估计,一本500页的书需要在图书馆书架上占一英寸,因此100万页的书需要占200英尺的书架空间。如果书架有4英尺宽,则需要50层。如果一个书架可以放5层书,则需要图书馆的10个大书架。为了得出这个结论,我需要估计每英寸的页数、图书馆书架的宽度和每个图书馆书架的层数。我的估计可能没有一个是准确的,但是我相信我的结论有50%正确的可能(写完此例后,我去图书馆看了一些实际的书籍和书架。书架只有2英尺宽,但是一个书架有7层共21英尺的可存书宽度。因此在书架的容量方面,我只有10%的误差,基本上是正确的,这远比我所期望或需要的更高)。

**例 2.8** 另一个日常生活中的评估实例。买一辆每加仑汽油可以行驶20英里的汽车是否比买一辆每加仑行驶30英里但贵1000美元的汽车更合算呢?普通汽车每年大约行驶12000英里。如果油价是每加仑1美元,则低效率的汽车每年买汽油需要花600美元,而节油的汽车需要花400美元。如果忽略诸如把1000美元存入银行得到利息等问题,需要5年来弥补价格上的差异。此时,买主就必须决定是否价钱是惟一的标准,是否5年的弥补时间可以接受。当然,车行驶的距离越长弥补差异就越快,而且汽油价格的变化也会大大影响结果。

## 2.8 深入学习导读

本章涉及的大多数问题都是离散数学的问题。有关这个领域的介绍请参见 Susanna S. Epp 著的《Discrete Mathematics with Applications, 2nd Edition》[Epp95]。Graham、Knuth 和 Patashnik 合著的《Concrete Mathematics: A Foundation for Computer Science》[GKP89],是一本关于在计算机科学中有一些数学问题的高级处理方法的书籍。

《IEEE Spectrum》1995年2月份期刊上的文章“Technically Speaking”[Sel95]讨论了本书

中用到的计算机存储单元的标准。

Udi Manker 著的《Introduction to Algorithms》[Mar 89]扩展了数学归纳法,从而使之成为一种设计算法的技术。

阅读 Eric S. Roberts 著的《Thinking Recursively》[Rob86]可以得到有关递归的进一步知识。为了正确掌握递归,应该了解一下 LISP 程序设计语言,尽管没有必要去编写一个 LISP 程序。特别是 Frideman 和 Felleisen 著的《The Little LISP》[FF89]能教你如何考虑递归,也教你 LISP 语言。这是一本有趣的书。

Daniel Solow 著的《How to Read and Do Proofs》[Sol90]是一本有关书写数学证明的好书。Leslie Lamport 著的《How to Write a Proof》[Lam93]也是如此。

John Louis Bentley 撰写的题为《The Back of the Envelope》和《The Envelope is Back》[Ben84, Ben86a, Ben86b, Ben88]的有关程序设计的姐妹篇著作可以使我们获得有关“信封背面”计算的进一步知识。James Gleick 所著的《Genius: The Life and Science of Richard Feynman》[Gle92],深刻地指出了“信封背面”计算对于计算机科学的重要性,它一点儿也不亚于原子弹的开发者对现代理论物理学的贡献。

## 2.9 习题

- 2.1 不使用递归,改写 2.4 节的阶乘函数。
- 2.2 把 2.2 节中用 for 循环编写的产生随机排列的函数改写为一个递归函数。
- 2.3 下面是一个计算 Fibonacci 序列的简单递归函数。

```
static long fibr(int n) : // Recursive Fibonacci generator
{
    Assert.notFalse((n > 0) && (n < 92), "Parameter out of range");
    if ((n == 1) || (n == 2)) return 1;    // Base case
    return fibr(n-1) + fibr(n-2);        // Recursive call
}
```

这个算法非常慢,调用 fibr 的总次数多于 Fib( $n$ )次。将它与下面的迭代算法进行比较:

```
static long fibi(int n) : // Iterative fibonacci generator
{
    Assert.notFalse((n > 0) && (n < 92), "Parameter out of range");
    long curr, prev;
    if ((n == 1) || (n == 2)) return 1;
    curr = prev - 1;    // curr holds current Fib value
    for (int i = 3; i <= n; i++) { // Compute next value
        curr = prev + curr;
        prev = curr - prev;    // prev holds previous Fib value
    }
    return curr;
}
```

函数 fibi 执行  $n - 2$  次 for 循环。请解释为什么 fibr 比 fibi 慢得多。

- 2.4 将汉诺塔问题推广,初始状态每个圆盘可能在任何一个柱子上,只要没有大圆盘放在小圆盘的上面即可。编写一个递归函数解决这个问题。
- 2.5 用反证法证明素数的个数是无限的。
- 2.6 证明 $\sqrt{2}$ 是无理数。
- 2.7 解释为什么下式成立:

$$\sum_{i=1}^n i = \sum_{i=1}^n (n - i + 1) = \sum_{i=0}^{n-1} (n - i)$$

- 2.8 使用数学归纳法证明公式(2.2)。
- 2.9 使用数学归纳法证明公式(2.5)。
- 2.10 使用数学归纳法证明公式(2.8)。
- 2.11 证明前  $n$  个偶数的和为  $n^2 + n$ 。
- (a)用例 2.3 对奇数的证明间接地进行证明。
- (b)用数学归纳法直接进行证明。
- 2.12 证明  $\text{Fib}(n) < \left(\frac{5}{3}\right)^n$ 。
- 2.13 证明:当  $n \geq 1$  时:

$$\sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4}$$

- 2.14 下面的定理称为鸽笼原理(Pigeonhole Principle)。
- 定理 2.7**  $n+1$  只鸽子要在  $n$  个鸽笼中栖息,那么至少有一个鸽笼中有 2 只鸽子。
- (a)用反证法证明鸽笼原理。
- (b)用数学归纳法证明鸽笼原理。
- 2.15 假设有一个  $n$  位整数(以标准二进制表示)按照相同的概率在 0 到  $2^n - 1$  之间取值。
- (a)对于每一个比特位,它取 1 的概率是多少? 取 0 的概率是多少?
- (b)对于一个  $n$  位的随机整数,值为“1”的位平均有多少个?
- (c)最左边一个为“1”的位所在位置的期望值是多少? 也就是说,从最左边一位开始向右移动直到遇到第一个“1”时平均检查了多少位?
- 2.16 用公升度量,你的总体积有多大(也可以用加仑作为单位)?
- 2.17 一位艺术史学家有一个包含 20 000 幅全屏彩色图像的数据库。
- (a)大约需要多少存储空间? 存储这个数据库需要多少张 CD-ROM(一张 CD-ROM 的容量为 600MB)? 请说出为推出结论你所做的所有假设。
- (b)现在假设你已经掌握了一种图像压缩技术,此时存储一幅图像只需要不压缩时所需存储空间的  $1/10$ 。如果压缩图像,整个数据库能放在一张 CD-ROM 上吗?
- 2.18 密西西比河每天的水流量大约是多少立方英里? 请给出为了推出该结论你所做的所有假设,不要查找答案或任何辅助事实。
- 2.19 分期付款购买房产时,你可以选择先付一些钱(称为“折扣点”),以获得一个较好的借款利率。假设你可以在两种 15 年期的抵押贷款中进行选择,一种为 8% 的利

率,另一种为  $7\frac{3}{4}\%$  的利率,但是要多预付房款总金额的 1%。如果选择低利率的抵押贷款,那么总房款 1% 的费用要多长时间才能平衡? 另外,请更精确地估计一下,如果选择高利率的抵押,而且把等价的总房款 1% 的费用存入利率为 5% 的银行,那么考虑付款额和利息,需要多长时间才能够得到回报? 不要使用纸或计算器来演算。

- 2.20 下面的问题用来测试有关计算机操作速度的知识。访问磁盘驱动器的时间通常是用毫秒(千分之一秒)或微秒(百万分之一秒)度量的吗? RAM 访问一个字的时间是多于 1 微秒还是少于 1 微秒? 如果计算机不停地运行着,那么 CPU 一年中能执行多少条指令? 不用纸或计算器,推出你的结论。
- 2.21 你家里的所有书加起来有 100 万页吗? 你所在学校图书馆的藏书总共有多少页?
- 2.22 本书中共有多少单词?
- 2.23 100 万秒是多少小时? 多少天? 用心算回答这些问题。
- 2.24 美国有多少城市和乡村?
- 2.25 一位男士开车去拜访他的亲戚。整个距离为 60 英里,他出发时的速度为 60 英里/小时。恰好行驶 1 英里后,他的旅行兴趣有所减少,所以立即减速到 59 英里/小时。再行驶一英里后速度降为 58 英里/小时。这样继续下去,每行驶 1 英里减速 1 英里/小时,直到走完全程。
- (a)他到达亲戚家所需的时间是多少?
- (b)如果速度随着距离均匀地连续减慢,行驶 1 英里正好总共减速了 1 英里/小时,那么他的旅行驾驶时间是多少?

## 第3章 算法分析

本章将介绍算法分析的一些基础知识,算法分析是评估算法所消耗资源的方法。我们可以据此对解决同一问题的两种或两种以上算法的代价加以比较,算法设计者也可以据此判断一种算法在实现时是否会遇到资源限制的问题。学习完本章之后,读者应该掌握以下知识点:

- 增长率(growth rate)的概念,即当问题的规模增大时,算法代价增长的速度。
- 增长率的上限和下限的概念,即怎么对简单程序、算法或问题的上下限作出估算。
- 能够区分一种算法(或程序)的代价和一个问题的代价。

本章最后还讨论了通过实验方式测算程序时间代价时可能遇到的一些实际问题,并介绍了通过代码调整法改善程序效率的一些原则。

### 3.1 概述

如何比较两种算法解决问题的效率呢?一种办法就是用源程序分别实现这两种算法,然后输入适当的数据运行,测算两个程序各自的开销。但是这种方法并不尽如人意。第一,编写两个程序来测算两种算法将花费较多的时间和精力,而我们至多只需要保留其中之一。第二,仅凭实验来比较两种算法,很有可能因为一个程序比另一个“写得好”,而使得算法的真正质量没有得到很好的体现。第三,测试数据的选择可能对其中的一个算法有利。第四,你可能会发现即使是较好的那种算法也超出了预算开销,这意味着你不得不再重复一遍这样的过程——寻找一种新的算法,再编写一个程序实现它。

有一种办法能够解决所有这些问题,我们称之为渐进算法分析(asymptotic algorithm analysis),简称算法分析(algorithm analysis)。它可以估算出当问题规模变大时,一种算法及实现它的程序的效率和开销。这种方法实际上是一种估算方法,如果两个程序中一个总是比另一个“稍快一点”,它并不能判断那个“稍快一点”的程序的相对优越性。但是在实际应用中,它被证明是很有效的,尤其是当科学家们确定某种算法是否值得实现的时候。

运行时间通常是算法代价的一个关键方面,但是也不能片面地注重运行速度,而应该同时考虑其他因素,如运行该程序所需要的空间代价(包括内存和磁盘空间)。通常我们需要分析一种算法(或者是实现该算法的一个程序实例)所花费的时间,以及一种数据结构所占用的空间。

许多因素都会影响程序的运行时间。有些因素与程序的编译和运行环境有关,如计算机主频、总线 and 外部设备等。如果与其他用户共享计算机资源,有时会使程序慢得像蜗牛爬行。程序设计使用的语言和编译系统生成的机器代码的质量会产生很大的影响,编程人员用程序实现算法的效率也会在很大程度上影响运行的速度。如果你要在一台指定的机器上,在给定的时间和空间限制下运行一个程序,以上这些因素都会对结果产生影响。但是,这些因素与两种算法或数据结构的差异无关。为了公平起见,同一个问题的两种算法所对应的两个程序,应该在同样的条件下用同一个编译器编译,在同一台计算机上运行。并且,两次编程所花费的精

力也应该尽可能地相等,以使得算法的实现“等效”。做到以上几点,上面提到的那些因素就不会对结果产生影响,因为它们对每一个算法都是公平的

如果你真的想知道一种算法的运行时间,只考虑主频、编程语言、编译器之类的因素是不够的。从理论上说,要在标准的环境下测算一种算法的时间代价,然而事实上,我们只是在某台计算机上运行算法的载体。惟一的选择是选用另一种尺度来代替运行时间。

判断算法性能的一个基本考虑是处理一定“规模”(size)的输入时该算法所需要执行的“基本操作”(basic operation)数。“基本操作”和“规模”这两个名词的含义都是模糊不清的,而且要视具体算法而定。“规模”一般是指输入量的数目。比如,在排序问题中,问题的规模就可以很典型地用被排序的元素个数来衡量。一个“基本操作”必须具有这样的性质:完成该操作所需时间与操作数的具体取值无关。在大多数高级语言中,两个整数相加以及比较两个整数的大小都是基本操作,而  $n$  个整数累加就不是基本操作,因为其代价(cost)依赖于  $n$  的值(即大小)。

**例 3.1** 下面是查找一维  $n$  元整数数组中最大元素的算法。该算法依次遍历数组中的元素,并保存当前的最大元素,称为“最大元素顺序检索”。下面就是使用 Java 语言编写的程序:

```
static int largest(int[] array) ,           // Find largest value
    int currLargest = 0;                    // Store largest value
    for (int i = 0; i < array.length; i++) // For each array element
        if (array[i] > currLargest)         // If this is largest
            currLargest = array[i];         // remember it
    return currLargest;                     // Return largest value
;
```

其中,问题的规模为  $n$ ,这些整数存放在数组 `array` 中,基本操作是“检查”一个整数,即把一个存放现有最大整数的变量与它作比较。我们可以认为,这样检查数组中的某个整数所需要的时间就是—定的,与该整数的大小或其在数组中的位置无关。

因为影响时间代价的最主要因素一般来说是输入的规模,我们经常把执行算法所需要的时间  $T$  写成输入规模  $n$  的函数,记作  $T(n)$ 。注意:我们总是假设  $T(n)$  为非负值。

我们把 `largest` 函数中检查一个元素所需要的时间记为  $c$ 。 $c$  中包括变量  $i$  增值的时间(这是处理数组中每一个元素都要做的工作),还有当找到一个新的最大元素时的实际赋值操作。现在不考虑  $c$  的实际值,也不考虑函数初始化时所需要的一小部分额外时间——我们只想得到执行该算法的一个合理的近似时间。因此,运行 `largest` 函数的总时间可近似地认为是  $cn$  (共需要  $n$  步检查工作,每一步需要时间  $c$ )。我们说 `largest` 函数(或者说最大元素顺序检索法)的时间代价可以用下面的等式来表示:

$$T(n) = cn$$

这个等式表明了最大元素顺序检索法时间代价的增长率。

**例 3.2** 把一个整数数组的第一个元素值赋给另一个变量,只要复制这个元素的值就可以了。我们认为完成这一功能所需要的时间是固定的,与这个元素的具体取值无关。在一台特定的计算机中,无论这个数组有多大(只要内存和定义的数组大小允许),复制该数组第一个元素值的时间总是确定的,记作  $c_1$ 。因此该算法时间代价的等式就是:

$$T(n) = c_1$$



输入规模  $n$  对运行时间不产生影响。这称为常数运行时间(constant running time)。

**例 3.3** 让我们来看一看下面的 Java 程序段:

```
sum = 0;
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        sum++;
```

如何计算这个程序段的运行时间呢? 显然,随着  $n$  的增大,其运行时间也会增大。本例的基本操作是变量  $sum$  的累加,我们可以认为该操作所需要的时间是一定的,记为  $c_2$  (我们在此可以忽略初始化  $sum$  和循环变量  $i$  与  $j$  累加的时间。事实上,这些时间开销都可以计入  $c_2$ 。本章后面的内容还将对此作进一步的解释)。要执行的基本操作总数为  $n^2$ , 因此,运行时间函数为:

$$T(n) = c_2 n^2$$

增长率的概念是非常重要的。它可以帮助我们比较两个算法的时间代价,而不用真的编写出两个程序,并在同一台计算机上运行。

图 3.1 给出了五个运行时间函数的曲线,每个多项式反映一个程序或者一种算法的时间代价,图中显示了不同算法的增长率。标记为  $10n$  和  $20n$  的两个函数图像为直线,表达式为  $cn$  ( $c$  为任意正常数)的增长率称为线性增长率(linear growth rate)或者线性时间代价(linear time cost)。这说明,当  $n$  增大时,算法的运行时间以相同的比例增加。 $n$  增大一倍,运行时间也增加一倍。如果算法的运行时间函数中含有形如  $n^2$  的高次项,则称为二次增长率(quadratic growth rate)。在图 3.1 中,标有  $2n^2$  的那条曲线就代表二次增长率。标有  $2^n$  的曲线属于指数增长率(exponential growth rate),这是因为  $n$  出现在指数位置而得名的。

从图中可以看到,运行时间函数式分别为  $T(n) = 10n$  和  $T(n) = 2n^2$  的两种算法有着天壤之别。当  $n > 5$  时,相应的  $T(n) = 2n^2$  的算法要慢得多,尽管  $10n$  的系数比  $2n^2$  的系数要大。比较标有  $20n$  和  $2n^2$  的两条曲线,我们还会发现,改变一个函数的常数系数只改变两个曲线的交点,当  $n > 10$  时,对应于  $T(n) = 2n^2$  的算法比对应于  $T(n) = 20n$  的算法慢。该图还表明,运行时间函数为  $T(n) = 5n \log n$  的曲线增长速度比  $T(n) = 10n$  和  $T(n) = 20n$  都稍快,但是又比  $T(n) = 2n^2$  慢。当  $a, b$  为大于 1 的任意正常数时,  $n^a$  的增长速度比  $\log^b n$  和  $\log n^b$  快。最后还应该指出,即使  $n$  值很小,运行时间函数为  $T(n) = 2^n$  的算法时间开销也很大。注意:当  $a, b \geq 1$  时,  $a^n$  增长速度比  $n^b$  快。

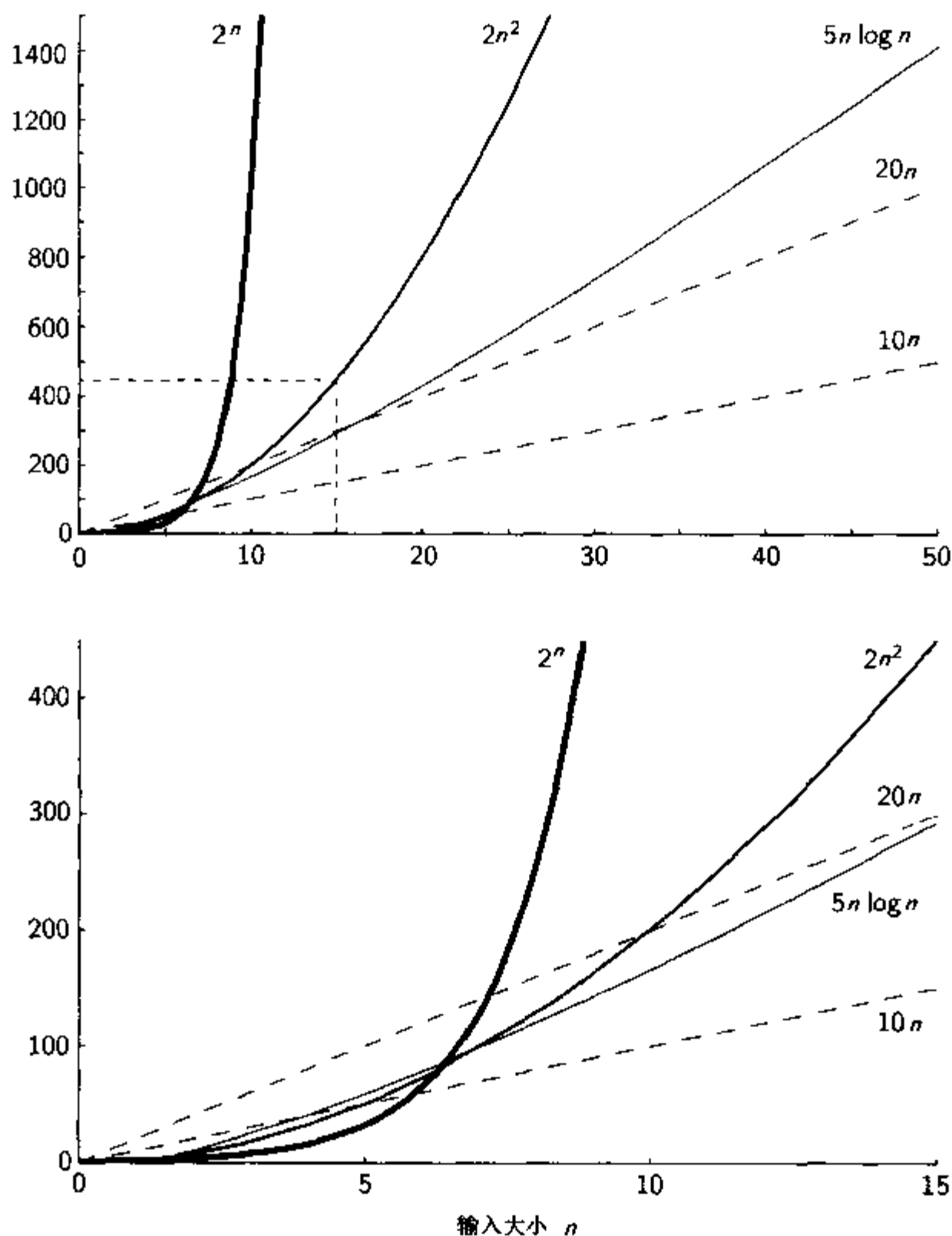


图 3.1 这是一幅图的两个视图,分别对应 5 个函数的增长率。下图是上图左下角的放大。水平轴代表输入规模,垂直轴表示时间、空间或其他开销

### 3.2 最佳、最差和平均情况

对于某些算法,即使问题规模相同,如果输入数据不同,其时间开销也不同。例如,现在要 从一个  $n$  元一维数组中找出一个给定的  $K$  (假设该数组中有且仅有一个元素值为  $K$ )。顺序 检索法将从第一个元素开始,依次检索每一个元素,直到找到  $K$  为止。一旦找到了  $K$ ,算法也 就完成了。这与例 3.1 的最大元素顺序检索不同,后者必须检查每一个元素的值。

这样,顺序检索法的时间开销可能在很大的一个范围内浮动。数组中的第一个元素可能 恰恰就是  $K$ ,于是只要检索一个元素就行了。在这种情况下,运行时间很短。这叫作算法的 最佳情况(best case)——顺序检索算法不可能执行比检索一个元素更少的操作了。另一种情 况,如果数组的最后一个元素是  $K$ ,运行时间就会相当长,因为这个算法要检查所有的  $n$  个元

素。这是算法的最差情况(worst case)——该算法不可能检索  $n$  个以上的元素。如果用一个程序来实现顺序检索法并用该程序对许多不同的  $n$  元数组检索,或者在同一个数组中检索不同的  $K$  值,就会发现,平均检索到整个数组的一半就能找到  $K$ 。也就是说,这种算法平均要检索  $n/2$  个元素,我们称之为算法的平均情况(average case)时间代价。

分析一种算法时,应该研究最佳、最差还是平均情况呢?一般来说,我们对最佳情况没有多大兴趣,因为它发生的概率太小,而且对于条件的考虑太乐观了。换言之,最佳情况不能作为算法性能的代表。不过,也有一小部分情况,最佳情况分析是有用的——尤其是当最佳情况出现概率较大的时候。在第8章,我们会看到一个排序算法的例子,可以用最佳情况运行时间来分析该算法,非常迅速而有效。

那么最差情况呢?分析最差情况有一个好处:它能让你知道算法至少能做得多快。这一点在实时系统中尤其重要,例如空运处理系统。在这个系统中,一个“绝大部分”情况下能管理  $n$  架飞机的算法,如果它不能在规定的时间内管理来自于同一方向的  $n$  架飞机,那么它是不能被接受的。

在另外一些情况下——特别是程序要对许多不同的输入运行多次时——最差情况分析就不适合于用来衡量一种算法的性能了。通常我们会更希望知道平均情况的时间代价,也就是说,当输入规模为  $n$  时算法的“典型”表现。可惜,平均情况分析并不总是可行的。首先,它要求我们清楚数据是如何分布的。例如,上面提到过顺序检索法平均情况下要检查数组中一半的元素。但是这是基于  $K$  在数组中每个位置出现概率相等的假设之上的。如果这个假设不成立,那么算法的平均情况就不一定是检查一半的元素了。我们将在10.2节就这个问题作进一步的讨论。

数据分布的特点对于很多检索算法都会有很大影响,例如10.4节提到的散列检索和5.5节的检索树。不正确的假设有时会给程序的时间或空间性能带来灾难性的影响。另一方面,一些特殊的数据分布也会带来好处,我们会在10.2节的例子中很好地看到这一点。

总之,在实时系统中,我们比较关注最差情况算法分析。在其他情况下,通常考虑平均情况,只要我们知道计算平均情况所需要的输入数据的分布即可。否则,就只能求助于最差情况分析了。

### 3.3 换一台更快的计算机,还是换一种更快的算法

假设你要解决一个问题,而且已经知道一种算法,其时间代价为  $cn^2$  ( $c$  为常数)。但是,程序运行时间比规定的时间慢了10倍。如果现在换一台运行速度比原有计算机快10倍的计算机,这种算法是否就可行了呢?如果问题规模不变,也许这台新计算机的确可以按时完成任务,虽然你的算法增长率很高。但是,对于大多数拥有更先进计算机的人来说,事情并不那么简单——他们并不想做相同的问题,而是要解决更大规模的问题!假定你希望原来的计算机对10 000个数进行排序,因为这是一个午餐休息时间所能完成的任务,那么现在对于这台新的计算机,你可能会要求它在相同的时间内对100 000个数进行排序。午餐时间不会缩短,因此你希望解决一个更大规模的问题。既然新机器快了10倍,你很自然希望它解决问题的规模也增大10倍。

如果你的算法增长率是线性的(即运行时间函数  $T(n) = cn$ ,  $c$  为常数),那么新计算机处

理100 000个数据的时间与原来的计算机处理10 000个数据的时间相同。如果算法的增长率高于 $cn$ (如 $c_1n^2$ ),在相同的时间里就不能在一台速度提高10倍的计算机上完成一个规模扩大10倍的问题

那么,在给定的时间内,速度变快的机器能够处理问题的规模扩大了几倍呢?假设新的计算机运行速度是原来的10倍,原有机器1小时能完成的问题规模为 $n$ ,那么新机器1小时最多可以解决问题的规模有多大?图3.2给出了图3.1中列出的5个运行时间函数可解决问题的规模

$f(n)$	$n$	$n'$	变化	$n'/n$
$10n$	1 000	10 000	$n' = 10n$	10
$20n$	500	5 000	$n' = 10n$	10
$5n \log n$	250	1 842	$\sqrt{10}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
$2^n$	13	16	$n' = n + 3$	--

图 3.2 规定时间内,速度是原来10倍的计算机所能处理问题规模的增长情况。第一栏列出了图3.1中给出的5个运行时间函数。不妨假设原来的计算机能在1小时内完成10 000个基本操作。第二栏显示了10 000个基本操作所能完成的 $n$ 的最大值。第三栏是 $n'$ 的值,即新机器解决该问题的最大值。第四栏给出了 $n$ 与 $n'$ 之间的函数关系。第五栏给出了 $n'$ 与 $n$ 的比例

从表中可以看出许多重要内容。前两个函数是线性的,只是常数系数不同。新机器处理二者时,问题规模的增长都是10倍。也就是说,系数大小虽然影响一定时间内能够解决问题的规模,却不能影响机器速度加快时问题规模的增长。无论算法增长率是多少,下面这个结论总是成立的:常数系数不改变机器速度加快时问题规模的增长倍数。

运行时间 $T(n) = 2n^2$ 的算法问题规模增长的倍数要比线性增长的算法小。后者增长了10倍,前者只是它的平方根: $\sqrt{10} \approx 3.16$ 。可见,增长率高的算法从机器升级中得益较少,能解决的问题规模也较小。

$T(n) = 5n \log n$ 的算法比二次增长率的算法提高较多,但是也不如线性增长。注意指数增长率的算法有一点儿特别。图3.1中显示,增长率为 $2n$ 的算法对应的曲线增长得很快。图3.2中,新的计算机可以处理的问题的规模约为 $n + 3$ (确切地说, $n + \log_2 10$ )。其问题规模的增长只是加上了一个常数,而不是乘上一个因数。原来 $n$ 值为13,新的问题规模就是16。假如你明年再换一台计算机,比现在还要快10倍,它也不过只能处理规模为19的问题。假如你还有另一个指数增长率的算法,某台计算机能在1小时内处理规模为1000的该问题,那么,换一台速度加快10倍的机器,1小时也只能解决规模是1003的问题!可见,指数增长率的算法与图3.2中其他算法有根本的不同。关于这种差异的意义,将在第15章中作进一步的探讨。

看来,你不应该急着买一台新机器,而应该考虑使用另一种运行时间为 $n \log n$ 的算法来代替现有的运行时间为 $n^2$ 的算法。在图3.1中,水平轴线代表时间。如果在规定时间内,两条对应不同增长率的曲线已经相交,那么增长速度较慢的那种算法就会较快。当问题规模 $n = 1024$ 时,运行时间 $T(n) = n^2$ 的算法需要 $1024 \times 1024 = 1\,048\,576$ 个单位时间, $T(n) = n \log n$ 的算法需要 $1024 \times 10 = 10\,240$ 个单位时间,两者之比远远大于10倍。因为 $n > 58$ 时,有 $n^2 > 10n \log n$ ,所以如果问题规模大于58,你最好换一种算法,而不是换一台计算机。

而且,即使你买了一台速度更快的计算机,就同一时间所能解决问题规模的增长情况而言,增长率较小的算法才能从中获益较多

### 3.4 渐进分析

虽然图 3.1 中标有  $10n$  的曲线系数较大, $2n^2$  曲线还是在  $n=5$  时就超过它了。如果把线性方程的系数再增加一倍,结果将如何呢? 如图所示,当  $n=10$  时, $20n$  对应的曲线也被  $2n^2$  超过。这两个线性增长率的系数并没有产生很大影响,只不过改变了交点的横坐标值。推而广之,改变其中一个函数的常系数,只改变两个曲线“在何处”相交,而不能改变它们“是否”相交。

当你拥有一台更快的计算机或者一个更快的编译系统时,一定时间内可以完成问题的规模增长倍数是一个定值,与运行时间函数中的系数无关。类似地,两个增长率不同的算法对应的时间曲线总是会相交的,与运行时间函数的系数无关。因此,当我们估算一种算法的时间或者其他开销时,经常忽略其系数。这样能够简化算法分析,并且使得我们的注意力集中在最重要的一点上:增长率。这称为渐进算法分析。准确地说,渐进分析是指当输入规模很大,或者说达到极限(微积分意义上)时,对一种算法的研究。实践证明忽略这些系数很有用,因此渐进分析也被广泛应用于算法比较。

并不是任何时候都能够忽略常数。当算法要解决的问题规模  $n$  很小时,系数就会起到举足轻重的作用。例如,现在要对 5 个数排序,那么用来给成千上万个数排序的算法可能就并不很适合,尽管它的渐进分析表明该算法性能良好。也有少量的例子,两种被比较的算法,其中具有较小增长率的算法,由于其较大的系数而对大多数的情况不适宜。渐进分析是对算法资源开销的一种不精确的估算,是一种“信封背面”估算。它提供了对算法资源开销进行评估的简单化的模型。但是千万不要忘记渐进分析的局限性,尤其是在那些系数也起到重要作用的少数情况下。

#### 3.4.1 上限

有几个术语是专门用于描述算法时间函数的。这些术语及其符号能够准确反映函数某一方面的特性。算法运行时间的上限(upper bound)就是其中之一,用来表示该算法可能有的最高增长率。

上限与输入规模  $n$  一定时的最差情况代价不同。准确地说,它是一个函数所能表示的增长率的上限。因此,如果要作一个关于算法上限的论断,它应与输入规模  $n$  有关。我们几乎总是考虑最佳、最差、平均情况下的上限,因此不能说“这种算法增长率的上限为  $n^2$ ”,而应该说“这种算法平均情况下增长率的上限为  $n^2$ ”。

“增长率的上限为  $f(n)$ ”这句话太长了,但是它在分析算法时又极其常用,于是我们采用一种特殊的表示法,称为大 O 表示法(big-Oh notation),读作“大欧”表示法。如果某个算法的增长率上限(最差情况下)是  $f(n)$ ,那么就说这种算法“在集合  $O(f(n))$  中”,或者说“在  $O(f(n))$  中”。例如,如果在最差情况下  $T(n)$  增长速度与  $n^2$  相同,则称算法最差情况代价在  $O(n^2)$  中。

下面给出了上限的一个精确定义。其中  $T(n)$  表示算法的实际运行时间, $f(n)$  则是上限

的一个函数表达式。

**定义 3.1** 对非负函数  $T(n)$ , 若存在两个正常数  $c$  和  $n_0$ , 对任意  $n > n_0$ , 有  $T(n) \leq cf(n)$ , 则称  $T(n)$  在集合  $O(f(n))$  中。

常数  $n_0$  是使上限成立的  $n$  的最小值。一般情况下  $n_0$  都很小, 如取 1, 但是并不一定要如此。必须能够找出这样的一个常数  $c$ , 而  $c$  确切是多少却无关紧要。换言之, 定义指出对于问题的所有(比如最差情况)输入, 只要输入规模足够大(即  $n > n_0$ ), 该算法总是能在  $cf(n)$  步以内完成,  $c$  是某个确定的常数。

**例 3.4** 考虑找出数组中某个元素的顺序检索法。如果访问并检查数组中的一个元素需要时间  $c_s$  ( $c_s$  为正数), 那么在平均情况下  $T(n) = c_s n/2$ 。对于  $n > 1$ ,  $c_s n/2 \leq c_s n$ 。所以根据定义,  $T(n)$  在  $O(n)$  中,  $n_0 = 1, c = c_s$ 。

**例 3.5** 某一算法平均情况下  $T(n) = c_1 n^2 + c_2 n$ ,  $c_1, c_2$  为正数。若  $n > 1$ ,  $c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2 \leq (c_1 + c_2) n^2$ 。因此取  $c = c_1 + c_2$ ,  $n_0 = 1$ , 有  $T(n) \leq cn^2$ 。根据定义,  $T(n)$  在  $O(n^2)$  中。

**例 3.6** 把数组中的第一个元素值赋给一个变量, 这个算法的运行时间是一定的, 与数组大小无关。因此, 在最佳、最差和平均情况下恒有  $T(n) = c$ 。我们可以认为在这种情况下  $T(n)$  在  $O(c)$  中。不过, 按照传统的说法, 运行时间上限为常数的算法在  $O(1)$  中。

要知道, 某个算法在  $O(f(n))$  中只是说事情顶多能坏到某种地步。事实上也许并不那么糟。如果我们知道顺序检索法在  $O(n)$  中, 那么也可以说它在  $O(n^2)$  中。但是顺序检索法对于很大的  $n$  也是可行的, 其他在  $O(n^2)$  中的算法就不一定如此了。我们总是试图给算法的时间代价找到一个最“紧”(即最小)的上限, 因此, 一般说顺序检索法在  $O(n)$  中。这也说明了为什么我们用“在  $O(f(n))$  中”或用记号“ $\in O(f(n))$ ”的表示方法, 而不用“是  $O(f(n))$ ”或“ $= O(f(n))$ ”。使用大  $O$  表示法没有严格的等号。 $O(n)$  在  $O(n^2)$  中, 但  $O(n^2)$  不一定在  $O(n)$  中。

### 3.4.2 下限

大  $O$  表示法描述上限, 也就是说, 当某一类数据的输入规模为  $n$  时(通常为最差情况, 所有可能输入情况的平均, 或最佳情况输入), 一种算法消耗某种资源(通常是时间)的最大值。

相似的表示方法可以用来描述算法在某类数据输入时所需要的最少资源。与大  $O$  表示法类似, 它也是算法增长率的一个衡量尺度。它同样可以表示任何资源, 但是一般衡量最小时间代价。还有一点类似之处, 对于输入规模  $n$ , 我们针对一些特殊的输入来估计资源开销: 最差、最佳和平均情况下的资源开销。

算法(或以后将讲到的问题)的下限用符号  $\Omega$  来表示, 读作“大欧米伽(Omega)”或“欧米伽”。下面给出  $\Omega$  的定义, 它与大  $O$  表示法的定义极其相似。

**定义 3.2** 若存在两个正常数  $c$  和  $n_0$ , 对于  $n > n_0$ , 有  $T(n) \geq cg(n)$ , 则称  $T(n)$  在集合  $\Omega(g(n))$  中。

**例 3.7** 假定  $T(n) = c_1 n^2 + c_2 n$  ( $c_1, c_2 > 0$ ), 则有:

$$c_1 n^2 + c_2 n \geq c_1 n^2 \quad (n > 1)$$



因此,取  $c = c_1, n_0 = 1$ , 有  $T(n) \geq cn^2$ , 根据定义,  $T(n)$  在  $\Omega(n^2)$  中。

也可以说例 3.7 中  $T(n)$  也在  $\Omega(n)$  中。但是,正如大  $O$  表示法一样,我们同样希望找到一个最“紧”的可能限制(对于  $\Omega$  表示法<sup>①</sup>,是最大的)。因此,一般说这个  $T(n)$  在  $\Omega(n^2)$  中。

回忆一下在数组中寻找值为  $K$  的元素的顺序检索法。在平均情况和最差情况下,这个算法在  $\Omega(n)$  中,因为在这两种情况下,都至少要检索  $cn$  个元素(在平均情况下  $c = 1/2$ , 在最差情况下  $c = 1$ )。

### 3.4.3 $\Theta$ 表示法

大  $O$  表示法和  $\Omega$  表示法使我们能够描述某一算法的上限(如果能找到某一类输入下开销最大的函数)和下限(如果能找到某一类输入下开销最小的函数)。当上、下限相等时,我们可能用  $\Theta$  表示法,读作“西塔(Theta)”(或“大西塔”)。如果一种算法既在  $O(h(n))$  中,又在  $\Omega(h(n))$  中,则称其为  $\Theta(h(n))$ 。注意在  $\Theta$  表示法中,我们不再说“在……中”,因为两个  $\Theta$  相同的函数有交换性,也就是说,若  $f(n)$  是  $\Theta(g(n))$ , 则  $g(n)$  是  $\Theta(f(n))$ 。

因为在平均情况下,顺序检索法既在  $O(n)$  中,又在  $\Omega(n)$  中,所以说平均情况下它是  $\Theta(n)$ 。

给出一个反映某算法时间代价的算术表达式,其上、下限通常都是相等的。这是因为从某种意义上说,我们已经对该算法有了一个精确的分析,用运行时间函数表示出来。对于很多算法(或者其实现形式,如程序),我们能够很容易地写出反映它们时间代价的函数。本书所列举的绝大部分算法都很浅显易懂,并且可以作出  $\Theta$  分析。但是,在第 15 章,我们会看到整整一类算法无法用  $\Theta$  表示法分析,有的甚至不能用大  $O$  和  $\Omega$  表示法分析。习题 3.11 给出了一个简短的程序,但是目前还没有人能够求出它的确切上下限。因此,当我们对算法了解较深时,一般采用  $\Theta$  表示法,但是限于分析能力,在有的算法中还会用到  $O$  或者  $\Omega$  表示法。

### 3.4.4 化简法则

一旦知道了算法的运行时间函数,从中推导出大  $O$ 、 $\Omega$  和  $\Theta$  表达式并不是一件很困难的

① 也可以用另一种方法定义  $\Omega$ :

**定义 3.3** 若存在正常数  $c$ , 有无穷多个  $n$  使  $T(n) \geq cg(n)$  成立, 则称  $T(n)$  在集合  $\Omega(n)$  中。

简而言之,这个定义要求在无限多的情况下,该算法的时间代价超过  $cg(n)$ 。注意该定义与大  $O$  表示法的定义不太相似。当  $g(n)$  是下限时,定义并不要求对所有大于某一常数的  $n$ , 都有  $T(n) \geq cg(n)$  成立。它只要求这种状况发生得足够频繁,特别是有无穷多个  $n$  满足这个条件。这种定义的优点可以在下面这个例子中发现。

**例 3.8** 假定某个算法表达式如下:

$$T(n) = \begin{cases} n & n \text{ 为奇数且 } n \geq 1 \\ n^2/100 & n \text{ 为偶数且 } n \geq 0 \end{cases}$$

$n \geq 0$  且  $n$  为偶数时,  $n^2/100 \geq (1/100)n^2$ 。取  $c = 1/100$ , 则有无数多个  $n$  (偶数的  $n$ ) 使  $T(n) \geq cn^2$  成立。根据定义,  $T(n)$  在  $\Omega(n^2)$  中。

对例 3.8 的运行时间函数而言,输入规模为  $n$  时时间代价至少为  $cn$ 。但是,有无穷多个  $n$ , 当输入规模为  $n$  时,运行时间为  $cn^2$ , 因此我们更倾向于说该算法在  $\Omega(n^2)$  中。但是,定义 3.2 会得到下限为  $\Omega(n)$ , 因为不能找到哪个常数  $c$  和  $n_0$ , 使任何  $n > n_0$  都有  $T(n) \geq cn^2$ 。而根据定义 3.3, 的确能够得到该算法的下限为  $\Omega(n^2)$ , 这更符合我们通常的标准。幸而在现实生活中,很少有程序或算法会有如此奇怪的特性,定义 3.2 一般还是能得到正确结果的。

从以上讨论中我们可以看到,渐进分析的下限表示法不是一种自然存在的规律,而只不过是数学家们发明的一种有用的表示工具,被电脑科学家改进后用以描述算法性能罢了。

事。我们并不需要严格遵循定义来推导,可以用下面的法则来求得其最简形式:

**定义 3.4** 渐进分析化简四法则:

1. 若  $f(n)$  在  $O(g(n))$  中,且  $g(n)$  在  $O(h(n))$  中,则  $f(n)$  在  $O(h(n))$  中。
2. 若  $f(n)$  在  $O(kg(n))$  中,对于任意常数  $k > 0$  成立,则  $f(n)$  在  $O(g(n))$  中。
3. 若  $f_1(n)$  在  $O(g_1(n))$  中,且  $f_2(n)$  在  $O(g_2(n))$  中,则  $f_1(n) + f_2(n)$  在  $O(\max(g_1(n), g_2(n)))$  中。
4. 若  $f_1(n)$  在  $O(g_1(n))$  中,且  $f_2(n)$  在  $O(g_2(n))$  中,则  $f_1(n)f_2(n)$  在  $O(g_1(n)g_2(n))$  中。

第一条法则说,如果  $g(n)$  是算法代价函数的一个上限,则  $g(n)$  的任意上限也是该算法代价的上限。对  $\Omega$  表示法有类似的性质:若  $g(n)$  是算法代价函数的一个下限,  $g(n)$  的任意下限也是该算法代价的下限,  $\Theta$  表示法同理。

法则 2 的意义在于使我们能够忽略大  $O$  表示法中的常数因子。对于  $\Omega$  和  $\Theta$  表示法同样有这样的性质。

法则 3 说明,顺序给出一个程序的两个部分(两组语句或两段代码),我们只需要考虑其中开销较大的部分。类似地,在  $\Omega$  与  $\Theta$  表示法中,也只看开销大的部分就可以了。

法则 4 用于分析程序中的简单循环。如果要有限次地重复某种操作,且每次重复的开销相等,则总开销为每次的开销与重复次数之积。对于  $\Omega$  和  $\Theta$  表示法结论也成立。

综合考虑前三条性质,我们可以在计算任何算法开销的渐进增长率时,忽略所有的常数和低次项。忽略常数的优点和不足在本节前面的内容中已有论述。进行算法分析时忽略低次项也合乎情理。因为当  $n$  增大时,相对高次项来说,低次项在总开销中所占比例微乎其微。因此,如果  $T(n) = 3n^4 + 2n^2$ ,可以说  $T(n)$  在  $O(n^4)$  中,因为  $n^2$  项对于总体开销来说无足轻重。

我们在本书的其他内容中讨论算法或程序的开销时,会不断使用这些化简法则。

### 3.5 程序运行时间的计算

这一节给出了几个简单程序段的分析。

**例 3.9** 首先来看一个给整型变量赋值的简单语句:

```
a = b;
```

该语句执行时间为一个常量,为  $\Theta(1)$ 。

**例 3.10** 再来看一个简单的 for 循环:

```
sum = 0;
for (i = 1; i <= n; i++)
    sum += n;
```

第一个语句的时间代价为  $\Theta(1)$ 。for 循环重复了  $n$  次,第三个语句时间代价为一个常量,根据 3.4.4 小节中的化简法则 4,后两行的 for 循环总时间代价为  $\Theta(n)$ 。根据法则 3,整个程序段的代价也是  $\Theta(n)$ 。

**例 3.11** 下面是一个含有多个 for 循环的程序段,其中有些是嵌套的。

```
sum = 0;
for (j = 1; j <= n; j++)           // First for loop
    for (i = 1; i <= j; i++)         // is a double loop
        sum++;
for (k = 0; k < n; k++)             // Second for loop
    A[k] = k;
```

该程序段有三个相对独立的片断:一个赋值语句和两个 for 循环结构。同样地,赋值语句时间代价为常量,记作  $c_1$ ;第二个 for 循环与例 3.10 相似,时间代价为  $c_2 n = \Theta(n)$ 。

第一个 for 循环是一个双重循环,需要一点特殊的技巧。我们从内层循环入手:运行  $\text{sum}++$  需要时间为一常量,记作  $c_3$ ,内层循环执行  $j$  次,根据法则 4,时间开销为  $c_3 j$ 。外层循环共执行  $n$  次,但是每一次内层循环的时间开销都因  $j$  的变化而不同。可以看到,第一次执行外层循环时  $j = 1$ ,第二次执行时  $j = 2$ 。每执行一次外层循环, $j$  就以 1 的步长递增,直至最后一次  $j = n$ 。因此,总的时间开销是从 1 累加到  $n$  再乘以  $c_3$ ,根据公式 2.1,可以得出:

$$\sum_{j=1}^n j = \frac{n(n+1)}{2}$$

即  $\Theta(n^2)$ ,根据法则 3,总运行时间为  $\Theta(c_1 + c_2 n + c_3 n^2)$ ,可简化为  $\Theta(n^2)$ 。

**例 3.12** 比较下面两段程序的算法分析:

```
sum1 = 0;
for (i = 1; i <= n; i++)           // First double loop
    for (j = 1; j <= n; j++)         // do n times
        sum1++;

sum2 = 0;
for (i = 1; i <= n; i++)           // Second double loop
    for (j = 1; j <= i; j++)         // do i times
        sum2++;
```

在第一个双重循环中,内层 for 循环总是执行  $n$  次。因为外层循环执行  $n$  次,所以  $\text{sum}++$  语句显然恰好执行  $n^2$  次。而第二个循环与上题的例子相仿,时间代价为  $\sum_{j=1}^n j$ ,近似于  $\frac{1}{2} n^2$ 。因此,两个二重循环的时间代价都为  $\Theta(n^2)$ ,不过第二个程序段的运行时间约为第一个的一半。

**例 3.13** 并非所有的嵌套 for 循环的时间代价都是  $\Theta(n^2)$ 。

以下面的两个嵌套循环为例:

```
sum1 = 0;
for (k = 1; k <= n; k *= 2)
    for (j = 1; j <= n; j++)
        sum1++;
```

```

sum2 = 0;
for (k = 1; k <= n; k *= 2)
    for (j = 1; j <= k; j++)
        sum2++;

```

第一段程序的外层 for 循环执行  $\log n$  次, 因为每循环一次  $k$  就乘以 2, 直至  $k > n$ 。由于内层循环执行次数恒为  $n$ , 所以第一段程序的总时间代价可以表示为  $\sum_{i=1}^{\log n} n$ 。这里有一个变量替换:  $k = 2^i$ 。根据公式(2.3), 其和为  $\Theta(n \log n)$ 。在第二段程序中, 外层循环也执行  $\log n$  次, 而内层循环重复  $k$  次, 随着外层循环的增长而成倍递增。其总和可以表示为  $\sum_{i=0}^{\log n} 2^i$ , 其中  $n$  假定恰为 2 的幂, 且  $k = 2^i$ 。因为外层循环执行  $\log n$  次, 而内层循环执行  $i$  次,  $i$  每次都成倍递增, 由公式(2.9)知, 其和为  $\Theta(n)$ 。

那么其他控制语句又如何呢? while 循环的分析方法与 for 循环类似。if 语句的最差情况时间代价是 then 和 else 语句中时间代价较大的那一个。对于平均情况代价也是如此, 假如  $n$  的取值与执行哪一条指令的概率无关(通常情况下都是如此, 但不一定如此)。switch 语句的最差情况代价是所有分支中开销最大的那一个。子程序调用时, 只要加上执行子程序的时间即可。

但是, 计算一个递归子程序的执行时间是一件令人头疼的事。递归过程的运行时间一般都能通过一个递归关系式得到很好的体现。例如, 2.4 节提到的递归函数 fact 每递归调用自身一次, 问题规模就减少 1。调用返回的结果与输入参数相乘, 这个操作的运行时间是一个常量。因此, 函数的时间代价就等于该常数加上执行递归调用的时间, 可以表示成:

$$T(n) = T(n-1) + c$$

可求得结果为  $\Theta(n)$ 。

有少数情况, 执行 if 或 switch 语句中某一个分支的概率是输入规模的函数。例如, 输入规模为  $n$  时, if 语句中的 then 语句执行的概率为  $1/n$ 。举个简单的例子, 某个 if 语句规定, 当对  $n$  个数中最小的那一个进行操作时, 才执行 then 语句。对这类问题进行分析时, 不能简单地将其处理成开销较大分支的时间代价。这时, 均摊分析方法(amortized analysis, 见 14.3 节)的技巧就很有用了。

本节的最后一个实例是比较两个完成检索功能的算法。在以前的学习中, 我们已经知道当被检索元素  $k$  在数组中任意一个位置出现的概率相等时, 顺序检索法的平均和最差情况代价都是  $\Theta(n)$ 。下面将其与二分法检索(binary search)的运行开销作比较, 假设数组元素按照从小到大的顺序存储。

二分法检索从检查数组中间位置的元素开始; 把这个位置记为 mid, 相应的元素值记为  $k_{\text{mid}}$ 。如果  $k_{\text{mid}} = K$ , 那么检索工作就完成了。当然, 这是不太可能的事情。不过, 这个中间元素值还是能够给我们一些有用的信息, 帮助我们继续进行检索。当  $k_{\text{mid}} > K$  时, 我们知道  $K$  不可能在 mid 后面的位置上出现, 因此, 在以后的检索中不必再考虑后半部分的元素。相反, 如果  $k_{\text{mid}} < K$ , 就可以忽略 mid 前面的部分。无论哪种情况, 都可以缩小一半范围。二分法的下一步工作是检查  $K$  可能存在的那部分元素的中间位置。这个位置上的元素值使我们又能缩小一半检索范围。重复这个过程, 就能够找到指定的元素(或者确定它不在数组中)。这个过程至多需要重复  $\log n$  次。图 3.3 给出了二分法检索的图示。下面给出它的 Java 程序:

位置	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
关键码	11	13	21	26	29	36	40	41	45	51	54	56	65	72	77	83

图 3.3 16 个元素的顺序数组二分法检索示意图。考察  $K = 45$  的检索。首先检查 7 号位置的元素。因为  $41 < K$ ,  $K$  不可能存放在小于 7 的位置。第二步,二分法检查 11 号位置的元素。 $56 > K$ ,被检索元素(若存在)必须在位置 7~11 之间。9 号元素是下一个要检索的元素,还是太大了。最后一个要检查的是 8 号位置的元素,恰好是要检索的元素。这样, `binary` 函数的最终返回值为 8。如果  $K = 44$ ,检索过程几乎完全相同,不过检查完 8 号元素之后, `binary` 将返回“UNSUCCESSFUL”(检索失败)

```
static int binary(int K, int[] array, int left, int right) {
    // Return position of the element in array (if any) with value K
    int l = left - 1;
    int r = right + 1;           // l and r are beyond the bounds of array
    while (l + 1 != r) {         // Stop when l and r meet
        int i = (l + r) / 2;     // Look at middle of remaining subarray
        if (K < array[i]) r = i; // In left half
        if (K == array[i]) return i; // Found it
        if (K > array[i]) l = i; // In right half
    }
    return UNSUCCESSFUL; // Search value not in array
}
```

函数 `binary` 的功能是查找  $K$  的(惟一)位置,并返回该位置。若  $K$  不在数组中,则返回一个特定信息。还可以对此算法稍加改动,使之能够返回  $K$  在数组中第一次出现的位置(若数组元素允许有重复),或者当  $K$  不在数组中时返回小于  $K$  的最大元素的位置。

对比顺序检索和二分法检索,可以看到顺序检索法的平均和最差情况代价  $\Theta(n)$  将远远大于二分法的代价  $\Theta(\log n)$ 。孤立地看,二分法比顺序检索法效率高得多。但是,注意顺序检索法的时间代价几乎总是相差不多的,无论数组中的元素是否按照顺序保存。相反,二分法检索要求元素必须按从低到高的顺序保存。根据二分法使用的环境,这个排序的要求可能会对时间代价产生损害,因为要保持数组的顺序性,在插入新元素时会增加时间代价。这里有一个权衡的问题:使用二分法检索比较容易,但是维持一个有序的数组比较费时,怎样权衡其利弊呢?只有在解决具体问题时才能知道是否利大于弊。

### 3.6 问题的分析

通常,我们用“算法”分析的技巧来分析算法或者其对应的程序。其实,还可以用这种技巧来分析问题的开销。应该指出,一个问题开销的上限不应该超过我们已经知道的最优解法的开销上限。但是一个问题开销的下限是什么意思呢?说一个问题在  $\Omega(f(n))$  中比说一个算法(或程序)在  $\Omega(f(n))$  中困难得多。因为一个问题在  $\Omega(f(n))$  中意味着所有可能的解法都

在  $\Omega(f(n))$  中,即使是我们尚未考虑到的解法。

迄今为止,我们给出的所有算法分析的例子都有明确的结果,而且大  $O$  与  $\Omega$  可以统一。为了更好地理解大  $O$ 、 $\Omega$  和  $\Theta$  三种表示法是如何描述一个问题或算法的特性的,最好来看一个你可能所知不详的问题。

现在让我们一起来分析一个排序的问题。所有排序算法的最差情况代价中最小可能值是多少?显然,任何算法必须先读入待排序的  $n$  个结点,排序完成以后应该将结果显示出来。于是,仅仅为了做输入和输出工作就至少要花  $cn$  时间。在很多类似的问题中,根据观察可以发现  $n$  个数需要输入,于是很容易得到  $\Omega(n)$  为一个下限。

在以前的学习中,你可能看到过最差情况代价在  $O(n^2)$  中的排序算法。在程序设计入门课程中通常作为典型范例的和直接选择排序的时间代价就在  $O(n^2)$  中。因此,  $O(n^2)$  是排序问题的一个上限。但是,  $\Omega(n)$  与  $O(n^2)$  之间的情况如何呢?是否存在更好的排序方法:如果不能找到最差情况代价小于  $O(n^2)$  的算法,也不能找到一种分析方法说明排序问题最差情况代价的下限大于  $\Omega(n)$ ,那就不能确定是否存在一个更好的算法。

第 8 章给出了一种排序方法,时间代价在  $O(n \log n)$  中。这使得上下限之间的差距大大缩小了。现在我们知道了一个下限  $\Omega(n)$  和一个上限  $O(n \log n)$ 。是否还能找到更快的排序算法呢?人们为此付出了很大努力,但是徒劳无功。幸好(抑或不幸?)第 8 章中还给出了一种证明:任何排序算法的最差情况代价都在  $\Omega(n \log n)$  中<sup>①</sup>。这个证明是算法分析领域的一个重要结果,它说明对于规模为  $n$  的排序问题,没有一种算法的运行时间比  $n \log n$  短。因此,我们可以得出结论:排序问题的最差情况代价为  $\Theta(n \log n)$ ,因为上限和下限重合了。

### 3.7 多参数问题

在有些情况下,要准确分析一个算法并给出其时间代价函数需要多个参数。为了阐释这一概念,我们来看看根据一幅图中各像素值出现的频率对这些值进行排序的一个算法。图通常用一个二维数组来表示,每个像素都是图的一个单元。像素的对应值是图中这个点的颜色或密度。假设每个像素的取值都可以是 0 到  $C-1$  的任意整数。现在问题是如何计算出每种取值所对应的像素个数,并根据每种取值在图中出现的次数对其进行排序。假定该图是一个包含  $P$  个像素的矩形。下面是对应的算法:

```
for (i = 0; i < C; i++)          // Initialize count
    count[i] = 0;
for (i = 0; i < P; i++)          // Look at all of the pixels
    count[value(i)]++;           // Increment proper pixel value count
sort(count);                     // Sort pixel value counts
```

在本例中, `count` 是一个一维  $C$  元数组,用来存储每种颜色值的像素个数。函数 `value(i)` 返回像素  $i$  的颜色值。

第一个 `for` 循环(初始化 `count`)的运行时间是由颜色的种类  $C$  决定的。第二个循环(计算每种颜色的像素个数)的运行时间是  $\Theta(P)$ 。最后一个语句(调用 `sort` 函数)的时间代价依赖

<sup>①</sup> 幸运的是我们可以证明这一结论,不幸的是排序问题是  $\Theta(n \log n)$ ,而非  $\Theta(n)$ 。



于这个排序算法的运行时间。根据上一节的讨论,我们可以认为排序算法对  $P$  个元素进行排序的时间代价为  $\Theta(P \log P)$ 。因此可以得出整个算法的代价为  $\Theta(P \log P)$ 。

但是这个表达式是否合理呢?真正被排序的对象是什么?不是像素,而是颜色的取值。那么如果  $C$  远远小于  $P$  时情况如何呢?我们看到,  $\Theta(P \log P)$  这个估计太悲观了。因为事实上被排序的对象远远少于  $P$ 。应该用  $P$  作分析变量来考察检索像素的那几步,而用  $C$  作分析变量处理颜色的那几步。于是,可以得出初始化循环时间开销为  $\Theta(C)$ ,像素计数循环开销为  $\Theta(P)$ ,而排序操作的开销为  $\Theta(C \log C)$ 。这样,总的时间代价为  $\Theta(P + C \log C)$ 。

为什么不能简单地用  $C$  衡量输入规模,认为算法的开销为  $\Theta(C \log C)$  呢?因为  $C$  总是比  $P$  小得多。例如,一幅图可能含有  $1000 \times 1000$  个像素,但是只有 256 种可能的颜色值。于是,  $P$  等于 100 万,  $C \log C$  与之相比简直是九牛一毛。但是,如果  $P$  小一些,而  $C$  大一些(即使它可能仍小于  $P$ ),那么  $C \log C$  就不能等闲视之了。由此可见,两个变量都有其存在的意义,缺一不可。

### 3.8 空间代价

除了时间以外,空间代价也是程序员要经常考虑的计算机资源。近年来,计算机运行速度提高的同时,其存储能力也大大增强了。虽然如此,可利用的磁盘或内存空间大小仍是对算法设计者的重要限制。

用来分析空间代价的技巧与分析时间代价的技巧类似。不同的是,时间代价是相对于处理某个数据结构的算法而言的,而空间代价是相对于这个数据结构本身而言的。渐进分析中增长率的概念对于空间代价同样适用。

**例 3.14** 一个包含  $n$  个整数的一维数组空间代价为多少?如果每个整数占用  $c$  字节,则整个数组需要  $cn$  字节的空間,即  $\Theta(n)$ 。

**例 3.15** 假设我们要记录  $n$  个人互相之间的朋友关系,就可以用一个  $n \times n$  数组来实现。数组的每一行表示某人的所有朋友,每一列则显示谁是朋友。例如,某位  $i$  认识某位  $j$ ,就可以在第  $i$  行第  $j$  列上作一个记号。同样,可以在第  $j$  行第  $i$  列上作个记号,因为这种关系是相互的。对于  $n$  个人来说,数组总规模为  $\Theta(n^2)$ 。

一个数据结构的主要目的是用恰当的方法存储数据,使我们能够简单而有效地对其进行访问。为此,我们必须在这个数据结构中加上一些附加信息,指明数据存放在何处。例如,链表中的每个元素都带有一个指针,指向表中的下一个元素。所有这类并非真正数据的附加信息称为结构性开销(overhead)。理论上,这种结构性开销应该尽量小,而访问路径又应该尽可能多且有效。这种互相矛盾的目标之间的权衡正是研究数据结构的乐趣和魅力所在。

**定义 3.5** 算法设计有一个重要原则:空间/时间权衡原则(space/time trade-off)。牺牲空间或者其他替代资源,通常都可以减少时间代价。

许多程序经过对信息的压缩或者加密后,都可以节省存储空间。然而,解压缩和解密的过程又需要额外的时间。这样一来,得到的程序空间代价小了,但是时间代价大了。相反,许多程序也可以预先存放部分结果或者对信息进行重组,以提高运行速度,但代价是占用了较多的存储空间。通常情况下,这些时间空间上的变化都是通过常数因子来改变的。

下面存储一个包含 32 个布尔值的集合的例子可以说明时间/空间权衡原则。一种方法是

存储 32 个整型值,每个整数代表一个布尔值。Java 定义一个 int(整型)为 4 个字节长,因此 32 个布尔值占用的空间总量为 128 个字节。也可以利用 Java 处理单字节 byte 类型变量的功能,用 32 个字节存储 32 个布尔值。因为每个布尔变量只能在 true 或 false 两种情况中选择一个,最有效的方法是用 32 个位变量存储这 32 个布尔变量,并将其压缩成一个 4 字节的 int 型变量<sup>①</sup>。不过,在大多数机器上,对位域变量进行赋值或读取的操作比对字符型或整型变量作同样操作慢得多。在我的计算机上,最快的实现方式是使用 32 个 byte 型变量来存储这 32 个布尔值。采用 int 或 boolean 型变量时,赋值操作所用时间几乎一样。若对一个 int 型变量进行位域操作,赋值或读取时间将是这个时间的两倍。在你的机器上,这些具体数据可能会有所不同,但是程序的相对快慢是差不多的。因此,实现时你可以选择对空间进行优化还是对时间进行优化。对任何一种实现方式而言, $n$  个布尔值都需要  $\Theta(n)$  的空间代价,所不同的只是其常数因子而已。

空间/时间权衡原则的另一个例子是查找表(lookup table)。查找表中预先存放了一些函数值,从而不必每次调用时都重新计算。例如阶乘函数,如果使用 32 位的 int 型变量来存储,12! 是允许范围内的最大函数值。如果一个程序中需要多次重复计算阶乘,那么把这 12 个函数值预先存放在一个查找表中将会大大减少运行时间。程序需要  $n!$  ( $n < 12$ ) 的值时,只需要简单地查一下查找表就行了(如果  $n > 12$ ,值就太大了,造成存储的不便)。与每次计算阶乘的时间代价相比,牺牲一点点空间来存放查找表实在太合算了。

查找表的另一个用途是存放某些具有较大开销的函数的近似值,如 sin、cos 等。如果只想在一定的精度要求下计算这类函数,或者允许结果在一定误差范围内,那么就可以使用查找表来存放一定精度的计算结果,而不必重复运行该函数。注意初始化查找表也需要一定的时间,因此程序中使用查找表的次数应该足够多,以保证初始化过程的开销是物有所值的。

下面一个空间/时间权衡的典型例子是程序员试图减少空间代价时经常会遇到的情形。这里有一个对一组记录排序的简单程序段,每个记录有一个关键码字段作为排序码。函数  $x.key()$  返回记录  $x$  的关键码值。共有  $n$  个记录,其关键码值是从 0 到  $n-1$  的一个排列(即其中无重复值出现)。这里用到了分配排序(Binsort),具体情况将在 8.7 节介绍。分配排序根据每个记录的关键码值在数组中分配给它相应的位置。

```
for (i = 0; i < n; i++)
    B[A[i].key()] = A[i];
```

这种做法很简单,也很迅速,时间代价为  $\Theta(n)$ 。但是,它需要两个长度为  $n$  的数组。另一种做法是按照排列的顺序逐一定位,不过可以在一个数组内完成(因此这是一个“原地”排序):

```
for (i = 0; i < n; i++)
    while (A[i].key() != i) // Swap element A[i] with A[A[i].key()]
        Dsutil.swap(A, i, A[i].key());
```

函数  $swap(A, i, j)$  用来交换变量  $i, j$  的值。第二段程序对记录进行排序的过程可能不太明显。为了弄清楚其工作方式,请注意:每做一次 for 循环,至少关键码值为  $i$  的记录已被放

---

<sup>①</sup> 理想情况下,这 32 个标志变量可以存储在一个有 32 个布尔变量的数组中。但是,大多数 Java 虚拟机用一个字节的物理表示来实现一个位的布尔类型。

到正确的位置,而且 $A[i].key()$ 一定大于等于 $i$ 。程序中至多会发生 $n$ 次`swap`操作,因为一旦一个记录已被放到正确的位置,就不会再移出去了,而且每执行一次`swap`操作至少能够把一个记录放到正确的位置。因此,这个程序段的时间代价也是 $\Theta(n)$ 。但是事实上,它的运行时间比第一段程序慢。在我的计算机上,第二种方法的时间代价是第一种方法的两到三倍,不过第二种方法节省了一半的空间。

程序空间、时间代价相互关系的另一个原则是对存储在磁盘上的信息(外存文件信息)进行处理的程序而言的,我们将在第9章以后进一步讨论这类程序。有意思的是,基于磁盘的空间/时间权衡原则与上面所说的基于内存的空间/时间权衡原则几乎是完全相反的。

**定义 3.6** 基于磁盘的空间/时间权衡原则(disk-based space/time tradcoff)为:在磁盘上的存储开销越小,程序运行得越快。这是因为从磁盘上读取数据的时间开销远远大于用于计算的时间开销,于是几乎所有用于对数据进行解压缩的额外操作的时间开销都小于减少存储开销后节约下来的读盘时间。

当然,这一原则也并不是对任何情况都成立的,不过在设计处理磁盘上数据的程序时,最好能知道这一原则。

#### 一个真实的故事

几年前,我的一个学生接到了一个课题。他的论文课题中涉及到基于一个庞大数据库的好几个复杂操作,而他已经进入了最后阶段。“Shaffer 博士,”他对我说,“我正在运行这个程序,它的运行时间看来太长了。”我们检查了算法之后,发现它的时间代价是 $\Theta(n^2)$ ,可能需要一两周的时间才能运行结束。即使计算机真的能不受干扰地运行那么久,他的论文和毕业日期也等不了那么久。值得庆幸的是,我们发现有一种相当简单的方法可以改变算法,而使其时间代价缩短为 $\Theta(n \log n)$ 。第二天他改写了这个程序,几个小时之后,运行结果就出来了,他的毕业论文也按时完成了。

### 3.9 实际操作中的一些因素

在实际工作中,时间代价分别为 $\Theta(n)$ 和 $\Theta(n \log n)$ 的两种算法之间的差别可能并不很大。但是,时间代价为 $\Theta(n \log n)$ 和 $\Theta(n^2)$ 的两种算法的差别却是巨大的。在学习普通数据结构和算法的课程中,你应该已经接触过不少这样的问题了,对于这些问题,某些简明解法的时间代价为 $\Theta(n^2)$ ,但是还有其他解法的时间代价为 $\Theta(n \log n)$ 。排序和检索——计算机的两大重要问题——都属于此类。

还有一种方法,虽然可能不如改变算法而降低增长率那么重要,但是也可以使程序的运行时间得到大幅度的改进,这就是所谓的“代码调整法”(code tuning)。这是人工方式优化程序的一种艺术,从而使程序运行得更快,或者需要更小的空间。对于很多程序,代码调整法可以把运行时间缩短到原来的十分之一,或者把存储开销降至原来的二分之一,甚至更少。我曾经对一个程序的关键函数进行了这样的代码调整,并没有改变其基本算法,却使其运行速度整整提高了200倍。当然,为了达到这个效果,我在信息的存储方式上作了很大改动,将符号编码结构转化成了数字编码结构,从而能够直接对信息进行运算等处理。

以下是一些如何使用代码调整法来提高程序运行速度的方法。

首先应该明白程序中的大多数语句对于程序的运行时间并没有太大的影响。通常只是几个关键的子程序,甚至是这些关键子程序中的几行关键命令,占用了绝大部分的运行时间。对于那种只占总时间 1% 的子程序进行调整,即使能使其运行时间减少一半,也是没有很大意义的,我们应该把精力集中在那些关键部分上。

调整代码时,注意收集好时间统计数据,许多编译器和操作系统中都带有剖视工具(profiler)和其他特殊的工具,可以帮助你得到有关时间、空间开销情况的数据。当你想提高程序效率时,这些都是很有用处的。

不要玩弄小技巧,使得程序的可读性降低。大多数代码调整都是使写得较粗糙的程序变得简明清晰,而不是在原来就很清晰的程序上加上一些小把戏。特别是应该更好地理解 and 利用先进编译器的能力来优化表达式。这里,“优化表达式”是指重新安排一些算术或逻辑表达式,使之运行得更加有效。当你对表达式进行手工优化时,不要破坏了编译器自身作同类优化的能力。一定要在一套可作为参照的输入数据下比较调整前后程序的运行时间,以检查你的“优化”是否真正使程序得到了改进。有很多次,我以为对程序进行了积极的代码调整,事实却证明适得其反,尤其是在优化表达式的时候。要比编译器做得更好似乎实在不是件容易的事。

有时会有这样的情形发生:你要对两个不同的程序加以比较,以确定哪个更适于完成某项任务。比较两个程序的时间代价是件难事,因为结果经常受制于某些不可控制的因素(如系统负载、所使用的语言或编译器,等等)所引起的实验误差。最重要的一点是你不能对其中一个程序有所偏爱,否则必然会影响其运行时间的比较结果。看一看处于竞争中的软件或硬件厂商各自的广告,你就会明白这一点。这里最容易出现的陷阱就是在编写程序时,你会下意识地对其中之一的代码调整下较大的功夫。如上所述,代码调整常常会使时间开销缩小到十分之一。如果原先两个程序的运行时间只是相差一个常数因子(即它们的增长率相同),那么代码调整产生的差异就会很容易影响你的比较结果。

但是,最巨大的时间和空间改进还是源于更好的数据结构或算法。本章的最后忠告是:

先调整算法,后调整代码!

### 3.10 深入学习导读

算法分析领域的探索性著作包括 Donald E. Knuth 的《The Art of Computer Programming》[Knu73, Knu81]; Aho、Hopcroft、Ullman 合著的《The Design and Analysis of Computer Algorithms》[AHU83]。使用“ $T(n)$  在  $O(f(n))$  中”这一表示法代替传统的“ $T(n) = O(f(n))$ ”,是我参考了 Brassard 和 Bratley [BB88] 后引进的,不过在他们之前显然已经出现了这种表示法。对于想进一步了解算法分析技巧的读者, Gregory J. E. Rawlins 的《Compare to What?》[Raw92] 是一本值得一读的好书。

Beatley [Ben88] 描述了数字分析领域的一个问题,自从 1945 年以来,该问题最著名的算法已经从  $O(n^7)$  降到了  $O(n^3)$ 。对于  $n = 64$  的输入规模来说,这一改进引起的程序运行速度的提高就几乎相当于同时期内计算机硬件方面取得的所有进展所提高速度的总和。

算法是决定程序效率最重要的方面,但优化程序代码也是改进程序性能不容忽视的手段。正如 Frederick P. Brooks 著的《The Mythical Man-Month》[Bro75] 一书中所说的,一位高效的

程序员所设计的程序常常比低效程序员的程序快 5 倍左右,即使两个人都不在优化代码方面下什么特别的功夫。如果你想看看关于如何提高代码效率的优秀文章,或者想学习优化代码的方法,不妨看一下 Jon Bentley 的[Ben82, Ben86a, Ben88]。

算法分析中还有一件趣事,写一个正确的二分法检索算法居然也不是件简单的事。Knuth[Kn81]中提到,第一个二分法检索算法早在 1946 年就出现了,但是第一个无错误、无故障的二分法检索算法却直到 1962 年才出现!有兴趣的读者可以看看[Kn81]中 6.2.1 小节中的一段论述。Bentley 在[Ben86a]的《Writing Correct Programs》中写道,90%的计算机专家不能在两小时内写出完全正确的二分法检索算法。

### 3.11 习题

- 3.1 对于图 3.1 中的 5 个表达式,说出每个表达式当  $n$  取什么值时效率最高。
- 3.2 画出下列表达式的函数图,并说出当  $n$  取什么值时每个表达式的效率最高。
- $$4n^2 \quad \log_3 n \quad 3^n \quad 20n \quad 2 \quad \log_2 n \quad n^{2/3}$$
- 3.3 按照增长率从低到高的顺序排列以下表达式:
- $$4n^2 \quad \log_3 n \quad 3^n \quad 20n \quad 2 \quad \log_2 n \quad n^{2/3}$$
- 又: $n!$  应该排在第几位?
- 3.4 (a)假设某一算法的时间开销为  $T(n) = 3 \times 2^n$ ,对于输入规模  $n$ ,在某台计算机上实现并完成该算法的时间为  $T$  秒。现在另有一台计算机,运行速度为第一台的 64 倍,那么  $T$  秒内新机器上能完成输入规模为多大的问题?
- (b)假设又有一种算法,  $T(n) = n^2$ ,其余条件不变,那么新机器  $T$  秒内能完成的输入规模是多少?
- (c)若算法  $T(n) = 8n$ ,其余条件不变,那么在新机器上  $T$  秒内能够处理多少输入数据?
- 3.5 硬件厂商 XYZ 公司宣称他们最新研制的微处理器运行速度为其竞争对手 Prunes 公司同类产品的 100 倍。若 Prunes 公司的计算机能在 1 小时内完成输入规模为  $n$  的某程序,对算法增长率分别为  $n, n^2, n^3, 2^n$  的程序,分别计算 XYZ 公司的计算机 1 小时内能完成的输入规模为多少。
- 3.6 根据大  $O$  和  $\Omega$  的定义,写出下列表达式的上限和下限。请注意确定适当的  $c$  和  $n_0$ :
- (a)  $c_1 n$
- (b)  $c_2 n^3 + c_3$
- (c)  $c_4 n \log n + c_5 n$
- (d)  $c_6 2^n + c_7 n^6$
- 3.7 是否每种算法都有  $\Theta$  等式? 也就是说,是否任何算法运行时间的上下限都相等?
- 3.8 对于下列各组函数式,  $f(n)$  与  $g(n)$  的关系为,或者  $f(n)$  在  $O(g(n))$  中,或者  $f(n)$  在  $\Omega(g(n))$  中,或者  $f(n) = \Theta(g(n))$ 。对于每一组函数,确定两个函数究竟是哪种关系,并简述理由。
- (a)  $f(n) = \log n^2$ ;  $g(n) = \log n + 5$
- (b)  $f(n) = \sqrt{n}$ ;  $g(n) = \log n^2$

- (c)  $f(n) = \log^2 n$ ;  $g(n) = \log n$   
 (d)  $f(n) = n$ ;  $g(n) = \log^2 n$   
 (e)  $f(n) = n \log n + n$ ;  $g(n) = \log n$   
 (f)  $f(n) = 10$ ;  $g(n) = \log 10$   
 (g)  $f(n) = 2^n$ ;  $g(n) = 10n^2$   
 (h)  $f(n) = 2^n$ ;  $g(n) = 3^n$

3.9 写出下列程序段平均情况下时间代价的  $\Theta$  表示式。假设所有变量类型都为 int:

(a)  $a = b + c$ ;

$d = a + e$ ;

(b)  $\text{sum} = 0$ ;

for ( $i = 0$ ;  $i < 3$ ;  $i++$ )

for ( $j = 0$ ;  $j < n$ ;  $j++$ )

$\text{sum}++$ ;

(c)  $\text{sum} = 0$ ;

for ( $i = 0$ ;  $i < n * n$ ;  $i++$ )

$\text{sum}++$ ;

(d) 假设数组  $A$  中含有  $n$  个元素。

for ( $i = 0$ ;  $i < n$ ;  $i++$ ) {

for ( $j = 0$ ;  $j < n$ ;  $j++$ )

$A[i] = \text{Dsutil.random}(n)$ ; // random takes constant time

sort( $A, n$ ); // sort takes  $n \log n$  time

}

(e) 假设数组  $A$  元素为从 0 到  $n - 1$  的任意一个排列。

$\text{sum3} = 0$ ;

for ( $i = 0$ ;  $i < n$ ;  $i++$ )

for ( $j = 0$ ;  $A[j] \neq i$ ;  $j++$ )

$\text{sum3}++$ ;

(f)  $\text{sum} = 0$ ;

if (EVEN( $n$ ))

for ( $i = 0$ ;  $i < n$ ;  $i++$ )

$\text{sum}++$ ;

else

$\text{sum} = \text{sum} + n$ ;

3.10 定理 3.1  $\log n! = \Theta(n \log n)$

证明该定理最简单的方法是独立地证明  $n!$  的上下限。

(a) 证明  $\log n!$  在  $\Omega(n \log n)$  中。

(b) 证明  $\log n!$  在  $O(n \log n)$  中。

3.11 下面是一个确定  $n$  的初始值的函数。尽你所能, 给出该段程序的下限:

while ( $n > 1$ )

```

if (ODD(n))
    n = 3 * n + 1;
else
    n = n / 2;

```

你认为其上限是否与你给出的下限相等?

- 3.12 是否对每个问题都可以找到这样的算法:它的运行时间能用  $\Theta$  表示法表示,即其上下限相等?
- 3.13 现在有一个已排序的数组,请改写二分法检索子程序,使之当  $K$  在数组中重复出现时能返回第一个  $K$  出现的位置。注意保证算法代价为  $\Theta(\log n)$ ,即当找到某个元素值为  $K$  后,不要用顺序法继续查找。
- 3.14 现在有一个已排序的数组,请改写二分法检索子程序,使之当  $K$  本身在数组中不出现时,能返回数组中小于  $K$  的最大元素的位置。如果数组中的所有元素都大于  $K$ ,则返回 ERROR。
- 3.15 设计一个“拼图游戏”算法。假定每一块拼板都有四个边,每个拼板最终的方向已知,为 top、bottom 等。假设已有一个函数

```
bool compare(Piece a, Piece b, Side ad, side bd)
```

该函数能够在常数时间内判断拼板  $a$  的  $ad$  面与拼板  $b$  的  $bd$  面是否耦合(可以拼接在一起)。算法的输入是一个随机生成的  $n \times m$  二维数组,每个数组元素为一个拼板;算法把这些拼板放置在数组中正确的位置上。从渐近分析的角度说,你的算法应该尽可能地有效。写出对  $n$  块拼板进行拼图时该算法运行时间的总和,并由此得出相应的闭合形式解。

- 3.16 证明:如果一个算法平均情况代价为  $\Theta(f(n))$ ,则其最差情况代价在  $\Omega(f(n))$  中。

## 3.12 项目设计

- 3.1 在你的计算机上实现 3.8 节说明的空间/时间权衡原则的布尔值一例。分别用 byte、boolean、int 或压缩存储在 int 变量中的位域变量来存储这些布尔值,比较读取这些值的时间。编写程序时请注意两个问题。首先,程序读取变量的次数应该足够多,以保证比较有意义——只进行一次操作的时间开销太小了,不能进行有效的比较。其次,应该保证程序的运行时间尽可能多地用于读取变量,而不是调用计时程序或者循环变量增值之类的操作。
- 3.2 在计算机上实现顺序检索法和二分法检索法。对于元素个数  $n = 10^i$  的数组( $i$  从 1 到你的计算机内存和编译器允许的最大值),比较两种算法的时间代价。数组元素从 0 至  $n-1$  顺序存储;对于每个  $n$ ,从 0 到  $n-1$  中随机选取一些元素作为要检索的元素进行检索。将结果用函数曲线图进行表示。



# 第二部分 基本数据结构

第 4 章 线性表、栈和队列

第 5 章 二叉树

第 6 章 树

第 7 章 图

## 第4章 线性表、栈和队列

本章描述了线性表的表示方法以及两种重要的、操作受限的线性表——栈(stack)和队列(queue)。4.1节从定义线性表的一种抽象数据类型(ADT)出发,详细介绍了线性表ADT的两种实现方式——顺序表(数组)和使用指针的动态链表,并且对它们的相对优点进行了讨论。4.2节和4.3节分别介绍了栈和队列。本章的每个数据结构都给出了Java语言的实现。

### 4.1 线性表

线性表(list)是由叫作元素(element)的数据项组成的一种有限并且有序的序列。这个定义中的有序是指线性表中的每一个元素都有自己的位置(position)。换句话说,就是线性表中有第一个元素,第二个元素等等。每一个元素也都有一种数据类型。虽然概念上并不反对线性表具有不同数据类型元素(参见12.2节),但是在这一章讨论的简单线性表实现中,表中的所有元素都具有相同的数据类型。操作被定义为线性表ADT的一部分,它并不依赖于元素的数据类型。例如,线性表ADT可以用来实现整数线性表、字符线性表等等。

线性表中不包含任何元素时,我们称之为空表(empty list)。当前存储的元素数目叫作线性表的长度(length)。线性表的开始结点叫作表头(head),结尾结点叫作表尾(tail)。表中元素的值与它的位置之间可以有联系,也可以没有联系。例如,有序线性表(sorted list)的元素按照值的递增顺序排列,而无序线性表(unsorted list)在元素的值与位置之间就没有特殊的联系。本章只考虑无序线性表。如何创建并有效地查找有序线性表的问题将在第8章和第10章进行论述。

当讲到线性表的内容时,我们的表示法是把元素写在括号内,并用逗号隔开。例如,线性表( $a_0, a_1, \dots, a_{n-1}$ )包含 $n$ 个元素。下标表示元素在线性表中的位置<sup>①</sup>。对于所有的 $i > 0$ ,下标为 $i$ 的元素紧跟着下标为 $i-1$ 的元素。在这种表示法中,空表记作 $()$ 。

在选择线性表的表示方法之前,程序设计人员首先应该考虑到这种表示法要支持的基本操作。人们关于线性表的一般直觉告诉我们:一个线性表在长度上应该能够增长和缩短。应该能够在线性表的任何地方插入或者移动元素。应该有办法获得元素的值、读出这个值或者改变这个值。必须能够生成和清除(或重新初始化)线性表。由当前结点找到它的前驱和后继的操作也应该是方便的。

下一步是依据一个线性表的一组操作来定义该对象的抽象数据类型(ADT)。在Java中,对象使用类(class)来表示。本节描述了每个线性表操作的目的,但是没有说明这些操作是如何实现的。后面有两个完整的实现,它们用相同的线性表ADT来定义自己的运算操作,但是

---

<sup>①</sup> 为了与Java数组的表示法一致,表中第一个位置使用0而不是使用1来表示。因此,如果表中有 $n$ 个元素,它们的位置就从0到 $n-1$ 给出。这对熟悉诸如Pascal语言的程序员来说似乎有些不习惯,但是比起在某一些程序例子中再换另一种编号方式来说,始终如一地采用这种表示法可以减少很多易混淆之处。

操作的实现方式则有显著差别。图 4.1 以 Java 接口 (Interface) 的形式表示了线性表 ADT 的公有方法。这是一个 Java 程序员要用来实现一种特殊线性表的全部知识。图 4.1 假定线性表元素是对 Object 类型的引用 (reference)。实际创建线性表时, 元素的实际类型可以使用应用程序中任何方便的数据类型来代替, 比如用简单的整型 Integer 或者用户自定义的更复杂的结构来代替。

```
interface List {
    // List ADT
    public void clear();           // Remove all Objects from list
    public void insert(Object item); // Insert Object at curr position
    public void append(Object item); // Insert Object at tail of list
    public Object remove();        // Remove/return current Object
    public void setFirst();        // Set current to first position
    public void next();            // Move current to next position
    public void prev();           // Move current to prev position
    public int length();          // Return current length of list
    public void setPos(int pos);   // Set current to specified pos
    public void setValue(Object val); // Set current Object's value
    public Object currValue();     // Return value of current Object
    public boolean isEmpty();      // Return true if list is empty
    public boolean isInList();     // True if current is within list
    public void print();          // Print all of list's elements
} // interface List
```

图 4.1 线性表类的接口, 它给出了一些成员函数

图 4.1 中几个成员函数的描述中涉及到了“当前”位置。例如: 成员函数 setFirst 把当前位置设置为表中第一个元素, 而成员函数 next 和 prev 则分别把当前位置移到下一个或者前一个元素。其含义是指任何线性表的实现都支持当前位置这个概念。

这个 List 接口中的第一个成员函数是 clear 函数。这个函数清除表中所有元素, 返回线性表的初始化(空)状态。成员函数 insert 在当前位置插入一个值。这个函数允许在表中任意位置插入新的元素。例如: 如果表为 (12, 32, 15), 而且当前位置指向元素 32, 那么以值 99 来调用 insert 就会把这个线性表转化成 (12, 99, 32, 15)。

成员函数 append 在表的最后添加一个元素。函数 remove 从表中删除当前元素。成员函数 setFirst 把当前位置的指示符设置为表中第一个元素。成员函数 prev 和 next 分别把当前位置的指示符设置为上一个或者下一个元素。函数 length 返回表中当前元素的数目。成员函数 setPos(i) 把当前位置设置为表中第 i 个元素 (线性表的开始元素位置为 0)。成员函数 setValue 替换当前元素的值。函数 currValue 返回当前元素的值。注意到初始化了的线性表和清除后的线性表都没有任何元素, 因此它们可以没有当前元素。对空表调用 currValue 或 setValue 是不正确的。任何一种线性表的实现方式都应该能够检测出这种线性表对象的误用。

函数 isEmpty 在表为空时返回 true, 否则返回 false。如果当前位置确实指向表中的一个位置 (而不是脱离表的末端或与空表相关), 则函数 isInList 返回 true。当按照位置顺序处理表中的每一个元素时, 这是有用的, 就像下面的源程序片断所展示的那样。

```
for(MyList.first(); MyList.isInList(); MyList.next())
    DoSomething(MyList.currValue());
```

最后,成员 `print` 打印出线性表的内容。

这里所给出的线性表类说明仅仅是众多可行的线性表实现中的一种。图 4.1 给出了大部分操作,人们很希望这些操作能在线性表中完成,并且能说明与线性表数据结构的实现相关的问题。作为使用线性表 ADT 的一个例子,我们可以创建一个函数,找到表中下一个(从当前位置开始)出现某个特殊关键码值的地方。函数 `find` 不需要知道线性表的具体实现。但是 `find` 需要知道线性表中元素的类型。像本书许多 ADT 一样,我们假设线性表中存储的数据类型支持接口 `Elem`,定义如下:

```
interface Elem {                // Interface for generic element type
    public abstract int key(); // Key used for search and ordering
} // interface Elem
```

接口 `Elem` 简单地保证了元素类型支持成员函数 `key`,该函数返回一个整型值作为查找关键码(search key)。在此限制之内,线性表元素可以是任何类型的。

下面是 `find` 函数,它找到线性表中第一个出现值  $K$  的元素,并且把线性表的当前位置设置为那个元素。如果值  $K$  找到了,函数 `find` 返回 `true`,否则返回 `false`。如果表中没出现  $K$  值,就重新设置线性表的位置,以便使得 `isInList` 函数返回 `false`。请注意线性表是从当前位置开始查找的。

```
// Starting at current position.
// find Elem with the next occurrence of key value K in List L
public static Elem find(List L, int K) {
    while (L.isInList())
        if (((Elem)L.currValue()).key() == K)
            return (Elem)L.currValue();
        else L.next();
    return null;
}
```

### 4.1.1 顺序表的表示法

线性表的实现有两种标准方法——顺序表(array-based list 或 sequential list)和链表(linked list)。本小节讨论的是顺序表的实现方法。4.1.2 节将讨论链表,4.1.3 节将讨论、比较这两种实现方式的时间、空间效率。

顺序表的实现是用数组来存储表中的元素,这就意味着将要分配固定长度的数组。因此当线性表生成时数组的长度必须是已知的。既然每个线性表可以有一个不等长的数组,那么这个长度就必须由线性表对象来记录。在任何给定的时刻,线性表事实上都有一定数目的元素,这个数目应该小于数组允许的最大值。线性表中当前的实际元素数目也必须在线性表中记录。

顺序表把表中的元素存储在数组中相邻的位置上。数组的位置与元素的位置相对应。换句话说,就是表中第  $i$  个元素存储在数组的第  $i$  个单元中。表头总是在第 0 个位置,这就使得

对表中任意一个元素的随机访问相当容易。给出表中的某一个位置,这个位置对应元素的值就可以直接获取。因此,用 `setPos` 函数访问任意元素只需要花费  $\Theta(1)$  时间。

既然顺序表把表中的元素定义为存储在数组的相邻单元中, `insert`、`append` 和 `remove` 函数就必须能够支持这一定义。在表尾插入和删除元素是很容易的。添加操作 `append` 要花费  $\Theta(1)$  时间。但是,如果我们想按图 4.2 所示的那样要在表头插入一个元素,那么当前表中所有元素都必须向表尾移动一个位置以腾出空间。如果表中已经有了  $n$  个元素,那么这个过程就要花费  $\Theta(n)$  时间。如果想要在有  $n$  个元素的表中的第  $i$  个位置插入,那么  $n - i$  个元素都必须向表尾移动。从表头删除一个元素也是如此,数组中所有元素都要向前移动一个位置以填满空间。要删除第  $i$  个元素,  $n - i - 1$  个元素都要向前移动。平均说来,插入和删除要移动一半元素,也即需要  $\Theta(n)$  时间。

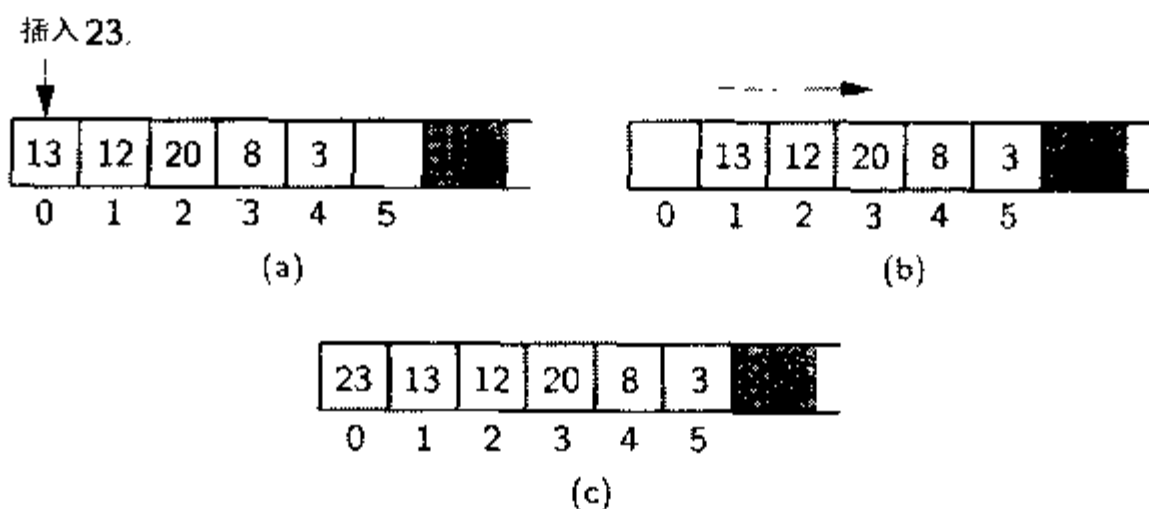


图 4.2 在顺序表的表头插入一个元素,需要表中所有元素向表尾移动一个位置

(a)在插入值为 23 的元素之前,表中包含 5 个元素

(b)所有元素向右移动了一个位置之后的表。

(c)在表中的第 0 个位置插入了 23。阴影部分表示数组中未用的空间

图 4.3 是顺序表实现的类说明,称为 `AList`。请注意这个类实现了 `List` 接口,这意味着它实现了 `List` 定义的每一个成员函数。类 `AList` 有 4 个数据成员,其中包括存储线性表元素的数组 `listArray`, 表的最大长度 `msize`, 当前实际长度 `numInList` 和当前元素在数组中的位置 `curr`。既然这些变量被说明为私有的(`private`),这就意味着它们不能被类的成员以外的其他函数调用。

```
class AList implements List { // Array-based list implementation

    private static final int defaultSize = 10; // Default array size

    private int msize; // Maximum size of list
    private int numInList; // Actual number of Objects in list
    private int curr; // Position of current Object
    private Object[] listArray; // Array holding list Objects

    AList() { setup(defaultSize); } // Constructor: use default size

    AList(int sz) { setup(sz); } // Constructor: user-specified size
```

---

```

private void setup(int sz) { // Do actual initialization work
    msize = sz;
    numInList = curr = 0;
    listArray = new Object[sz]; // Create listArray
}

public void clear() // Remove all Objects from list
{ numInList = curr = 0; } // Simply reinitialize values

public void insert(Object it) { // Insert Object at current position
    Assert.notFalse(numInList < msize, "List is full");
    Assert.notFalse((curr >= 0) && (curr <= numInList),
        "Bad value for curr");
    for (int i = numInList; i > curr; i--) // Shift Objects to make room
        listArray[i] = listArray[i-1];
    listArray[curr] = it;
    numInList++; // Increment list size
}

public void append(Object it) { // Insert Object at tail of list
    Assert.notFalse(numInList < msize, "List is full");
    listArray[numInList++] = it; // Increment list size
}

public Object remove() { // Remove and return current Object
    Assert.notFalse(! isEmpty(), "Can't delete from empty list");
    Assert.notFalse(isInList(), "No current element");
    Object it = listArray[curr]; // Hold removed Object
    for (int i = curr; i < numInList - 1; i++) // Shift elements down
        listArray[i] = listArray[i+1];
    numInList--; // Decrement list size
    return it;
}

public void setFirst() { curr = 0; } // Set curr to first position
public void prev() { curr--; } // Move curr to previous position
public void next() { curr++; } // Move curr to next position

public int length() // Return length of list
{ return numInList; }

public void setPos(int pos) // Set curr to specified position
{ curr = pos; }

```

```

public void setValue(Object it) { // Set current Object's value
    Assert.notNull(isInList(), "No current element");
    listArray[curr] = it;
}

public Object currValue() { // Return current Object's value
    Assert.notNull(isInList(), "No current element");
    return listArray[curr];
}

public boolean isEmpty() // Return true if list is empty
{ return numInList == 0; }

public boolean isInList() // True if curr is within list
{ return (curr >= 0) && (curr < numInList); }

public void print() { // Print all list's Objects
    if (isEmpty()) System.out.println("");
    else {
        System.out.print("(");
        for (setFirst(); isInList(); next())
            System.out.print(currValue() + " ");
        System.out.println(")");
    }
}

// class AList

```

图 4.3 顺序表类的实现

AList 类中头两个成员函数是它的构造函数 (constructor), 当生成一个线性表时, 构造函数控制初始化。第一个构造函数没有参数, 而第二个构造函数的参数为一个整数, 该参数表示线性表的最大长度。这个长度将成为 listArray 的长度。如果创建线性表时没有说明最大长度, defaultSize 值将用来设置 listArray 的长度。这两个构造函数都调用一个私有成员函数 setup 来完成实际的初始化工作, 包括给 listArray 分配空间和初始化其他数据成员。

其他成员函数实现了 List 接口中说明的其他函数。

#### 4.1.2 链表

第二种传统的实现线性表的方法是利用指针, 这种表示法通常称为链表 (Linked List)。链表是动态的 (dynamic), 也就是说, 它能够按照需要为表中新的元素分配存储空间。

链表是由一系列叫作表的结点 (node) 的对象组成的。因为结点是一个独立的对象 (这一点与数组的一个元素相反), 所以它能够很好地实现独立的结点类。建立结点类还有一个好处就是它能够被栈和队列的链接实现方式重用 (reuse), 栈和队列数据结构的实现将在本章后面介绍。图 4.4 表示了结点的完整定义, 叫作 Link 类。Link 类中的对象包含一个存储元素值



的 element 域和一个存储表中下一个结点指针的 next 域。因为在由这种结点建立的链表中, 每个结点只有一个指向表中下一个结点的指针, 所以叫作单链表(singly linked list)。

```
class Link { // A singly linked list node
    private Object element; // Object for this node
    private Link next; // Pointer to next node in list
    Link(Object it, Link nextval) // Constructor 1
    { element = it; next = nextval; } // Given Object
    Link(Link nextval) { next = nextval; } // Constructor 2
    Link next() { return next; }
    Link setNext(Link nextval) { return next = nextval; }
    Object element() { return element; }
    Object setElement(Object it) { return element = it; }
} // class Link
```

图 4.4 单链表结点类的定义

Link 类非常简单。它的构造函数有两种形式, 一个函数有初始化元素的值, 而另一个没有。其他函数帮助用户访问两个(私有)数据成员。使用这个类的用户可以取得并设置 next 和 element 域的值, 这样使得 Link 类的使用者可以方便地设置这些域值。原则上这种方法比让数据成员公有化要好。这是由于这些设置(set)函数可以用来控制值的改变, 只接受合理的赋值。使用线性表实现的用户不必知道 Link 类的细节。Link 类实际上属于链表实现, 而不是线性表类公共接口。必要时, 可以使用 Java 的包(package)来禁止链表用户使用 Link 对象。

图 4.5 是单链表类的定义, 称为 LList。它也实现了 List 接口。

```
class LList implements List { // Linked list class
    private Link head; // Pointer to list header
    private Link tail; // Pointer to last Object in list
    protected Link curr; // Pointer to current Object

    LList(int sz) { setup(); } // Constructor - - Ignore sz
    LList() { setup(); } // Constructor

    private void setup() // Do initialization
    { tail = head = curr = new Link(null); } // Create header node

    public void clear() { // Remove all Objects from list
        head.setNext(null); // Drop access to rest of links
        curr = tail = head; // Reinitialize
    }

    // Insert Object at current position
    public void insert(Object it) {
        Assert.notNull(curr, "No current element");
```

```

curr.setNext(new Link(it, curr.next()));
if (tail == curr) // Appended new Object
    tail = curr.next();
}

// Insert Object at end of list
public void append(Object it) {
    tail.setNext(new Link(it, null));
    tail = tail.next();
}

public Object remove() { // Remove and return current Object
    if (! isInList()) return null;
    Object it = curr.next().element(); // Remember value
    if (tail == curr.next()) tail = curr; // Removed last; set tail
    curr.setNext(curr.next().next()); // Remove from list
    return it; // Return value removed
}

public void setFirst() // Set curr to first position
{ curr = head; }

public void next() // Move curr to next position
{ if (curr != null) curr = curr.next(); }

public void prev() { // Move curr to previous position
    if ((curr == null) || (curr == head)) // No previous Object
        { curr = null; return; } // so just return
    Link temp = head; // Start at front of list
    while ((temp != null) && (temp.next() != curr))
        temp = temp.next();
    curr = temp; // Found previous link
}

public int length() { // Return current length of list
    int cnt = 0;
    for (Link temp = head.next(); temp != null; temp = temp.next())
        cnt + +; // Count the number of Objects
    return cnt;
}

public void setPos(int pos) { // Set curr to specified position
    curr = head;
    for (int i = 0; (curr != null) && (i < pos); i + +)

```

```

        curr = curr.next();
    }

    public void setValue(Object it) // Set current Object's value
    { Assert.notFalse(isInList()); curr.next().setElement(it); }

    public Object currValue() , // Return value of current Object
    { if (! isInList()) return null;
      return curr.next().element();
    }

    public boolean isEmpty() // Return true if list is empty
    { return head.next() == null; }

    public boolean isInList() // True if curr is within list
    { return (curr != null) && (curr.next() != null); }

    public void print() { // Print out the list's elements
        if (isEmpty()) System.out.println("()");
        else {
            System.out.print("( ");
            for (setFirst(); isInList(); next())
                System.out.print(currValue() + " ");
            System.out.println(")");
        }
    }
} // class LList

```

图 4.5 链表类的实现

一个线性表对象有一个指向第一个结点的头指针(head)和一个指向最后一个结点的尾指针(tail)。当前位置是由一个指向当前结点的指针表示的,这与数组中使用下标表示的方法不同。

图 4.6(a)是存储有 4 个整数的链表示例。指针变量中存储的值使用指向某个地方的箭头来表示。Java 的实现中使用一个特殊的记号 null 来表示不指向任何地方的指针值,例如最后一个元素的 next 域就是这样。null 指针在图解中表示为穿过指针变量方框的对角线。

图 4.6(a)中链表的 curr 指针直接指向当前结点。起初,这似乎是很明显的实现方式。但是如果我们想要在这个位置向表中插入一个新的结点,想想看会怎样。要在图 4.6(a)中链表的当前位置插入 10 这个值,并且要变成图 4.6(b)所示的链表。调用我们定义的 insert,新元素就插入到了表中的当前位置。这种定义似乎是合理的,并且在以数组为基础的实现中没有出现问题。但是,就图 4.6(a)所示的链表实现方式而言,就会出现问题。为了把包含新元素的结点“镶嵌”到链表中,存放 23 的结点就必须使它的后继指针(next)指向新的结点。但是,这样就没有一种方便的方法指向 curr 所指结点的前驱结点了。

一种代价较大的方法是从表头开始做,直到找到当前结点的前驱结点。另外一种更可行的

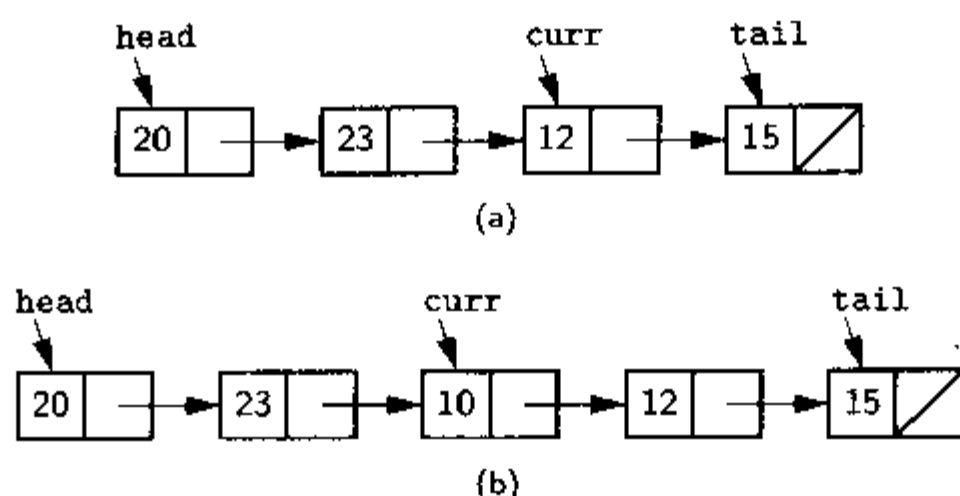


图 4.6 链表的实现示例,其中指针 curr 直接指向当前结点  
(a)插入数值 10 之前的链表。(b)插入数值 10 之后的预期结果

办法是把新元素的值复制到当前结点中,在其后插入一个新的结点,并把原来的当前结点值赋给这个新结点。这样,只花费一个不必要的拷贝代价就完成了插入操作。可惜的是,如果我们要删除表中的最后一个元素,这种 curr 指针表示法就行不通了。例如,假设当前 curr 指针指向图 4.6 (a)中值为 15 的结点,我们想要删除这个结点就不可避免地要改变值为 12 的结点。

另一种可选的方案是让 curr 指针指向当前元素的前驱结点。换句话说,如果图 4.6(a)中标号为 12 的结点是当前结点,那么 curr 指针事实上应该指向标着值 23 的结点。这就使得插入操作相当容易。

这种新的当前结点定义方式带来了新的问题。如果表中只有一个元素,那么 curr 指针所要指向的前驱元素也就不存在了。因此,表中没有结点或只有一个结点的情况就比较特殊。当我们把第一个元素作为当前位置时这就是一个特例。所有这些特例需要在链表实现时具有附加的源代码,而且这也增加了源代码的复杂性和程序出现故障的机会。实际上,如果定义 curr 指针直接指向当前结点,也会出现一些特殊情况。

这些特例可以通过增加特殊的表头结点(header node)来解决。这个表头结点是表中的第一个结点,它与表中其他元素一样,只是它的值被忽略,不被看作表中的实际元素。因为我们不再需要考虑空链表、表中只有一个结点或当前元素为表中第一个元素这些特殊情况,所以表头结点节省了源代码。这种源代码简化的代价是增加了表头结点空间的开销,但是源程序的复杂性降低了。既然处理特殊情况的语句被节省了,源程序的规模就变得更小了。实际上,这种方法显著地节省了空间,远远多于表头结点所需要的那一点儿冗余,节省空间的大小依赖于新建的表数目。图 4.7 表示的是使用表头结点初始化一个空链表的情形。图 4.8 是使用了表头结点对图 4.6 所示情况进行插入操作的示例,并且规定 curr 指针指向当前结点的前驱结点。

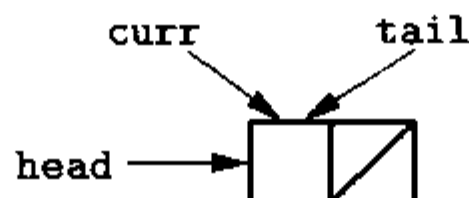


图 4.7 带有表头结点的单链表的初始状态

定义当前元素为 curr 指针所指结点的后继结点还会带来一个好处,我们可以在表中任意

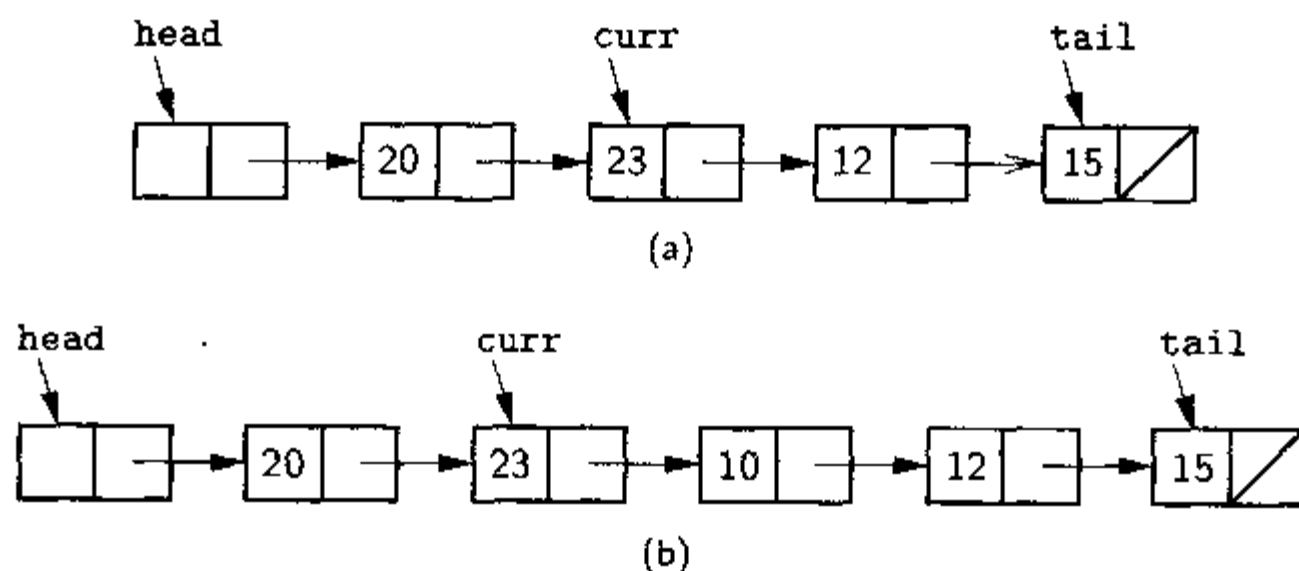


图 4.8 使用带有表头结点和转义 curr 指针的插入图示

(a)插入前的链表。当前结点是含有值 12 的结点。

(b)插入含有值 10 的结点之后的链表

一个位置插入新元素。再次考虑一下 curr 指针直接指向当前元素的那种朴素的实现方式。既然我们定义 insert 是在当前位置之前插入一个新元素,那么在这种定义的情况下就无法使用 insert 在表尾插入新元素了。如果 curr 指针指向最后一个元素,那么插入就会把新元素放在最后一个元素的后继位置。但是,我们对 curr 指针的定义作了修改之后就允许 insert 函数在 curr 指针指向表中最后一个存在的结点时,能够在表尾插入一个新结点。如果 curr 指针指向表头结点,那么插入就会把新元素放在链表开始的位置。

链表类大多数成员函数的函数实现都是很直截了当的。但是 insert 和 remove 就值得仔细研究了。

在链表中当前位置插入一个新元素包括三个步骤。首先,要创建一个新的结点,并且赋给它一个新的值。其次,新结点的 next 域要赋值给当前结点的元素。第三,当前结点元素前驱的 next 域要指向新插入的结点。既然 curr 指针已经指向了当前元素的前驱结点,这项工作就很容易完成了。下面是图 4.5 的 insert 函数中的一行语句,实际上它完成了所有这三个步骤:

```
curr.setNext( new Link( it, curr.next() ) );
```

运算符 new 创建了一个新的链表结点。函数 curr.setNext() 使当前结点的前驱结点的 next 域指向新插入的结点。运算符 new 调用了 Link 类有两个参数的那个构造函数。第一个参数是元素的值,叫 it。第二个参数是要放在 Link 结点 next 域的值,在此是由函数调用 curr.next() 所返回的值。图 4.9 就显示了这三个步骤。插入需要花费  $\Theta(1)$  时间。

从链表中删除一个结点只需要能够在被删除的结点放一个恰当的指针。下面图 4.5 中 remove 函数的一行语句就是这样做的。

```
curr.setNext( curr.next().next() );    // Remove from list
```

但是,我们必须谨慎,不能“丢失”被删除结点的内存。此内存(如图 4.10 中的 it 指针所指结点)应该返回给存储器。因此,把要删除的指针赋值给临时指针 it。图 4.10 显示了 remove 过程。删除一个元素需要  $\Theta(1)$  时间。

成员函数 next 只是将 curr 指针向表尾移动一个位置,这需要  $\Theta(1)$  时间。成员函数 prev 把 curr 指针向表头移动一个位置,但是它的实现方式更困难一些。在单链表中,没有指向前

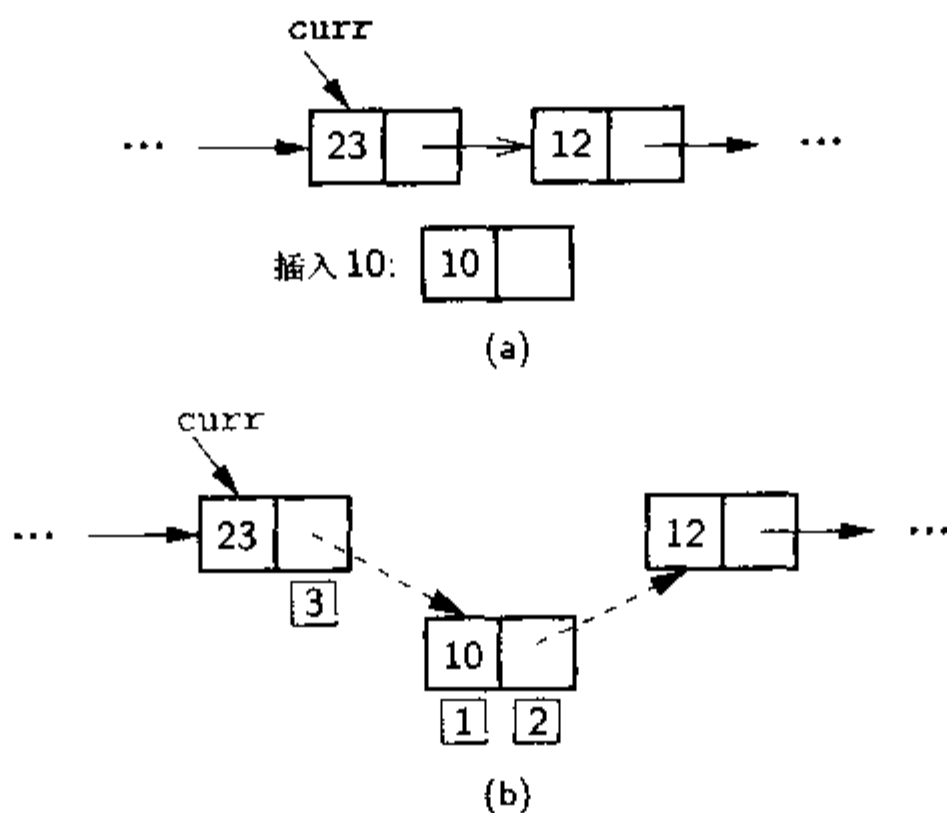


图 4.9 链表结点的插入过程

(a)插入前的链表。(b)插入后的链表。①表示新链表结点的元素域。②表示新结点的 next 域,它指向插入前原来的值为 12 的当前结点。③表示链表中当前结点的前驱结点的 next 域。插入前它指向含有值 12 的结点,插入后它指向含有值 10 的新插入结点

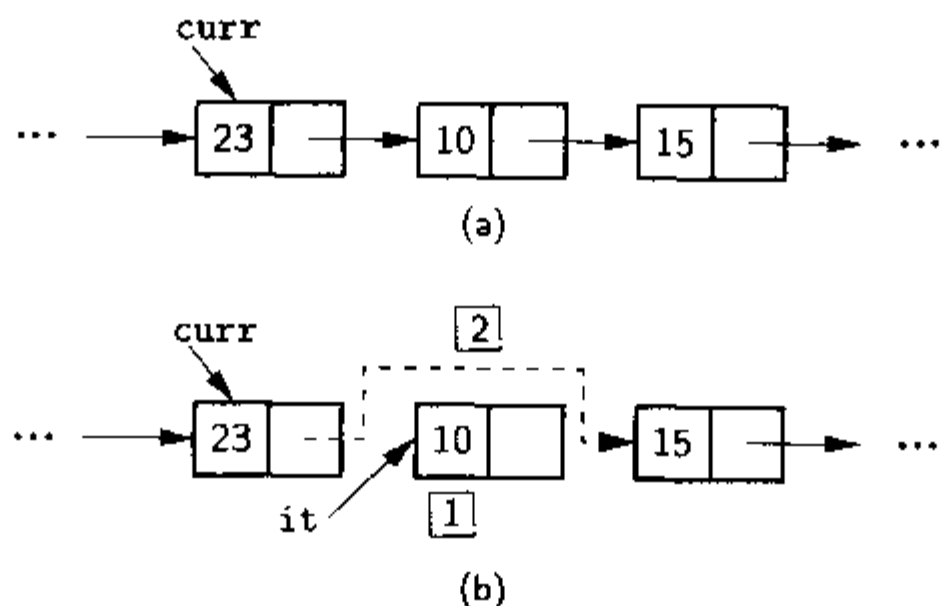


图 4.10 链表结点的删除过程

(a)删除值为 10 的结点前的链前。(b)删除结点后的链表。①表示被删除的结点——指针 it 指向这个结点。②表示被删除结点的前驱结点的 next 域,它将指向被删除结点的后继结点

驱结点的结点。因此,惟一的选择就是从头开始沿链表移动,直到我们找到了当前结点(既然这个结点是我们想要的,就一定要记住它前面的结点)。平均说来,要花费  $\Theta(n)$  时间。成员函数 `setPos` 的实现中找第  $i$  个位置需要从表头向后移动  $i$  个位置,花费  $\Theta(i)$  时间。其余操作的实现是很简单的。

### 可利用空间表

Java 的空闲存储空间的分配操作 `new` 相对来说较难使用。C、C++、Pascal 和许多其他编程语言中等价的函数也是如此。究其原因,存储的分配和回收必须尽量处理非固定模式的

分配和回收需求,这些需求所要求的内存大小非常悬殊。Java的“无用存储空间回收器”(garbage collector)负责发现程序不再使用的存储空间并回收到空闲空间表中去,这是额外的开销。直到最近,许多C和Pascal编译器还不能正确完成它们的空闲存储空间回收功能。如果按照某种模式使用,有的会失败,有的会毫无理由地变慢。在极端情况下,负责返回内存给空闲空间表的函数根本就没有实现!编译器只是简单丢弃所有返回的存储空间。

现在,大多数编译器都能实现合理的存储分配和回收,但是存储分配和回收管理的一些固有特性却使它们的效率不太高。在链表的实现中,线性表结点的建立和删除使得编写Link类的程序员能够提供简单而有效的内存管理例程,以代替系统级的存储分配和回收操作符。Link类能管理它自己的可利用空间表(freelist),来取代反复调用的new。还可以不依赖“无用存储单元回收器”而收集被删除结点的空间。可利用空间表存放当前不用的那些线性表结点,从一个链表上删除的结点就被放到可利用空间表的首端。当需要把一个新元素增加到链表上时,先检查可利用空间表,看一看是否有可用的线性表结点。如果有空结点,则从可利用空间表中取走一个结点。只有当可利用空间表为空时,才调用标准操作符new。

当链表周期性地增长然后缩短时,可利用空间表是非常有用的。可利用空间表的大小绝对不会超过链表的长度。(线性表缩短之后)需要增加新结点时可以通过可利用空间表来处理。

可利用空间表本身被一个称为freelist的静态(static)数据成员访问。如果一个数据成员被声明为静态的,该类中所有对象共享同一个数据成员。这样,所有Link对象都可以引用同一个freelist变量。

我们实现可利用空间表的方法是在Link类实现中增加两个新函数get和release。图4.11是Link类的freelist版。需要修改图4.5的LList实现来调用新函数。图4.12是修改后的insert、append和remove函数。注意函数get被声明为static,这是因为不应该对特殊Link对象调用get。例如,修改后的insert引用get的方式为Link.get()。

```
class Link { // Singly linked list node with freelist
    private Object element; // Object for this Link
    private Link next;      // Pointer to next Link in list
    Link(Object it, Link nextval) // Constructor 1
    { element = it; next = nextval; } // Given Object
    Link(Link nextval) { next = nextval; } // Constructor 2
    Link next() { return next; }
    Link setNext(Link nextval) { return next = nextval; }
    Object element() { return element; }
    Object setElement(Object it) { return element = it; }

    // Extensions to support freelists
    static Link freelist = null; // Freelist for the class

    static Link get(Object it, Link nextval) { // Get new link
        if (freelist == null)
            return new Link(it, nextval); // Get from free store
```



```

    Link temp = freelist;           // Get from freelist
    freelist = freelist.next();
    temp.setElement(it);
    temp.setNext(nextval);
    return temp;
}

void release() {                   // Return Link to freelist
    element = null;                // Drop reference to the element
    next = freelist;
    freelist = this;
}
} // class Link

```

图 4.11 使用可利用空间表的 Link 类实现。增加了 get 和 release 函数,以及静态的 freelist 变量

可利用空间表的 get 和 release 函数的运行时间都为  $\Theta(1)$ ,除非可利用空间表用完了而需要调用系统级的标准 new 操作。

```

// Insert Object at current position
public void insert(Object it) {
    Assert.notNull(curr, "No current element");
    curr.setNext(Link.get(it, curr.next())); // Get Link
    if (tail == curr)                        // Appended new Object
        tail = curr.next();
}

public void append(Object it) { // Insert Object at tail of list
    tail.setNext(Link.get(it, null));        // Get Link
    tail = tail.next();
}

public Object remove() { // Remove and return current Element
    Assert.notNull(isInList(), "No current element");
    Object it = curr.next().element();       // Remember value
    if (tail == curr.next()) tail = curr;    // Removed last: set tail
    Link tempPtr = curr.next();
    curr.setNext(curr.next().next());        // Remove from list
    tempPtr.release();                       // Release Link
    return it;                              // Return value removed
}

```

图 4.12 使用可利用空间表的 LList 类实现中的 insert、append 和 remove 函数。请注意 Link.get()和 Link.release()的调用形式

### 4.1.3 线性表实现方法的比较

前面已经给出线性表的两种截然不同的实现方法,人们自然要问,哪一种更好呢?如果有特殊任务必须使用一个线性表来完成,那么应该选择哪一种实现方法呢?

顺序表的缺点是它的大小事先固定。虽然便于给数组分配空间,但它不仅不能超过预定的长度,而且当线性表中只有几个元素时,浪费了相当多的空间。链表的优点是只有实际在链表中的对象需要空间,只要有可用的内存空间分配,链表中元素的个数就没有限制。链表的空间需求为  $\Theta(n)$ ;而顺序表的空间需求是  $\Omega(n)$ ,而且还可能更多。

顺序表的优点是对于表中的每一个元素没有浪费空间,而链表需要在每个结点上附加一个指针。如果 element 域占据的空间较小,则链表的结构性开销就占去了整个存储空间的大部分。当顺序表被填满时,存储上没有结构性开销。在这种情况下,顺序表有更高的空间效率。

有一个简单的公式可用来确定在特殊情况下,顺序表和链表的实现哪一种空间效率更高。设  $n$  表示线性表中当前元素的数目, $P$  表示指针的存储单元大小(通常为 4 个字节), $E$  表示数据元素的存储单元大小(可以任意取值,最小为一位的布尔变量), $D$  表示可以在数组中存储的线性表元素的最大数目,则顺序表的空间需求为  $DE$ 。如果不考虑任何给定时刻链表中实际存储元素的数目,则链表的空间需求为  $n(P+E)$ 。对于给定的  $n$  值,这两个表达式中值较小的那一个对于  $n$  个元素的实现有更高的空间效率。在一般情况下,当线性表中的元素相对较少时,链表的实现比顺序表的实现更节省空间。反之,当数组几乎被填满时,顺序实现方法空间效率更高。为此,我们可以求出  $n$  的临界值,即  $n > DE/(P+E)$ ,超过这个值时,顺序表的空间效率就更高。如果  $P = E$ (例如:4 个字节的 long 类型元素数据和 4 个字节的指针),则临界值为  $n = D/2$ 。

作为一般规律,当线性表元素数目变化较大或者未知时,最好使用链表实现。而如果用户事先知道线性表的大致长度,使用顺序表的空间效率会更高。

像取出线性表中第  $i$  个元素这样的按位置的随机访问,使用顺序表更快一些;通过 next 和 prev 可以很容易调整当前位置向前或者向后,这两种操作需要的时间为  $\Theta(1)$ 。相比之下,单链表不能直接访问前面的元素,按位置访问只能从表头开始,直到找到那个特定的位置。这两种操作需要的平均时间和最差时间为  $\Theta(n)$ 。

给出指向链表中合适位置的指针后,insert 和 remove 函数所需要的时间仅为  $\Theta(1)$ 。而顺序表必须在数组内将其余的元素向前或者向后移动。这种方法所需要的平均时间和最差时间均为  $\Theta(n)$ 。对于许多应用,插入和删除是最主要的操作,因此它们的时间效率是举足轻重的。仅就这个原因而言,链表往往比顺序表更好。

实现顺序表时,程序员可以用 Java 的 Vector 对象来代替简单的数组。不同之处在于 Java 的 Vector 对象用动态数组(dynamic array)来实现数据结构。动态数组根据实际存储的元素数目增长或缩小。采用这种办法,程序员可以避免标准数组一旦创建就不能更改数组大小的限制。这也意味着使用前不用给 Vector 对象分配空间。这种方法的缺点是需要时间来调整 Vector 对象的存储空间。Vector 对象元素个数增长时,它的元素内容要被拷贝下来。一个实现得较好的 Vector 类将如此增长和缩小数组,使得一系列的插入/删除操作代价相对较低。我们需要用到 14.3 节的均摊分析(amortized analysis)技术来分析使用 Vector 类的时间代价。

#### 4.1.4 元素的表示

前面所介绍的顺序表和链表都存储了一个 Object 引用。对于大一些的元素,这是必须的。因为这样可以避免把元素拷贝到表中的开销,如果需要还可以允许多个表元素指向同一个对象。这样做不仅节省空间,而且还意味着修改一个元素的值会自动影响到所有与它相关的位置。

既然线性表元素是指向 Object 的引用,我们就把线性表元素存为指针。当然,每个指针需要它自己的空间。如果元素都不重复,这自然会增加不必要的结构性开销。对于 int 变量这样的小对象来说,总的结构性开销是可观的。

使用指针或者元素本身哪一个更好依赖于实际应用。大体上讲,元素越大而且重复越多,使用指向元素的指针就会更好。

实现线性表类(或存储一组用户定义的数据元素的任何数据结构)面临的第二个问题是:是否要求这些元素类型相同。这在数据结构中称为同构性(homogeneity)。在某些应用中,用户可能把线性表元素定义为一个类,而不允许不同类的对象存储在这个线性表中。有很多技术可以保证给定线性表的元素类型是固定的,而同时又允许不同线性表存储不同的元素类型。一种方法是把具有给定类型的对象存储在线性表的头结点中(例如可以把该对象作为线性表构造函数的一个参数),然后检查该线性表的所有插入操作,保证插入了适当的元素类型。另一种方法是对每个 insert 操作都检查被插入的元素与线性表的第一个元素(不是头结点)类型是否相同。

#### 4.1.5 双链表

4.1.2 小节介绍的单链表只允许从一个表结点直接访问它的后继结点,而双链表(double linked list)可以从一个表结点出发,方便地在线性表中访问它的前驱结点和后继结点。双链表存储了两个指针,使得这些成为可能:一个指向它的后继结点(与单链表相同),另一个指向它的前驱结点。图 4.13 说明双链表的结构。

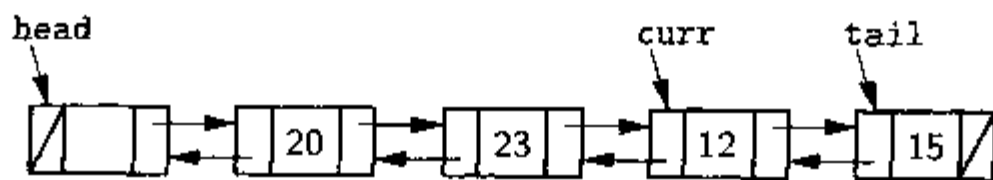


图 4.13 一个双链表

双链表非常有用,主要是因为它比单链表更容易实现。因为指针 prev 可以访问前一个结点,所以可以让指针 curr 直接指向双链表中的当前结点。但是习惯上让指针 curr 指向包含当前元素的结点的前驱结点,这样做有两个原因:(1)避免在空表中插入元素的特殊情况;(2)可以把结点插入到表中的任意位置。如果 curr 直接指向当前结点,insert 就不能把一个结点插入到表尾。

线性表的实现采用单链还是双链对 List 类的用户应该是透明的。图 4.14 给出使用双链表的一个 Link 类的完整实现。图 4.16 给出了 insert、append、remove 和 prev 四个双链表成员函数的实现。其他双链表类的成员函数与图 4.5 完全相同。双链表实现是 List 接口的又一次实现。

```

class DLink {
    // A doubly - linked list node
    private Object element; // Object for this node
    private DLink next;    // Pointer to next node in list
    private DLink prev;    // Pointer to previous node in list
    DLink(Object it, DLink n, DLink p) // Constructor 1
    { element = it; next = n; prev = p; } // Given Object
    DLink(DLink n, DLink p) { next = n; prev = p; } // Constructor 2
    DLink next() { return next; }
    DLink setNext(DLink nextval) { return next = nextval; }
    DLink prev() { return prev; }
    DLink setPrev(DLink prevval) { return prev = prevval; }
    Object element() { return element; }
    Object setElement(Object it) { return element = it; }
} // class DLink

```

图 4.14 双链表结点的实现

双链表的 insert 函数非常简单。图 4.15 给出了链表在插入一个值为 10 的结点之前和之后的情形。图 4.16 中的三行代码完成了这项工作。

```

curr.setNext(new Link(it, curr.next(), curr));
if (curr.next().next() != null)
    curr.next().next().setPrev(curr.next());

```

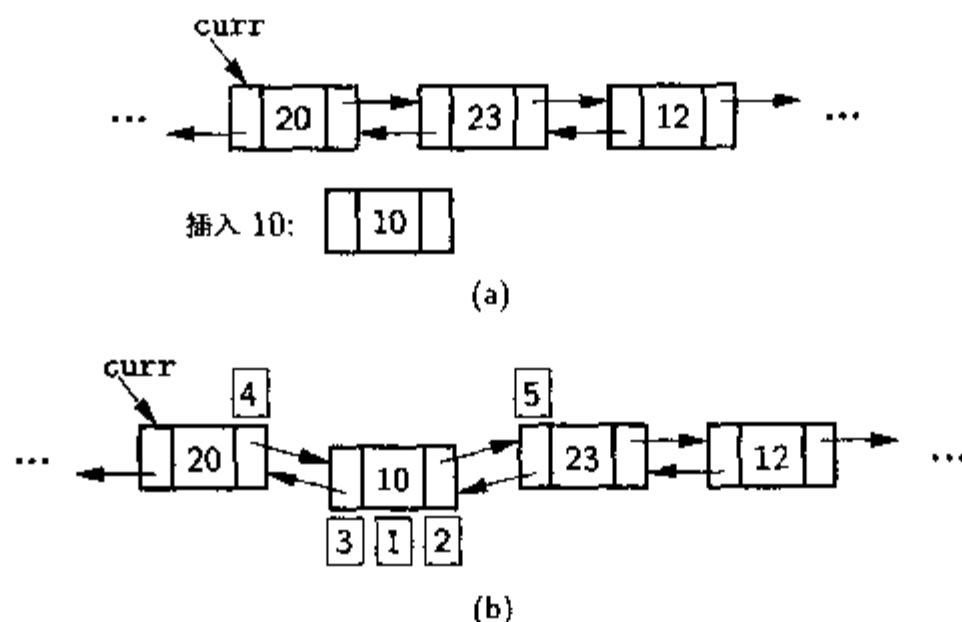


图 4.15 双链表的插入。方框标号 1、2 和 3 对应于链表结点构造函数所做的赋值操作。方框标记 4 对 curr→next 的赋值。方框标记 5 对新插入结点的后继结点的 prev 指针赋值

```

// Insert Object at current position
public void insert(Object it) {
    Assert.notNull(curr, "No current element");
    curr.setNext(new DLink(it, curr.next(), curr));
    if (curr.next().next() != null)
        curr.next().next().setPrev(curr.next());
    if (tail == curr) // Appended new Object
        tail = tail.next();
}

```

```

;

public void append(Object it) { // Insert Object at tail of list
    tail.setNext(new DLink(it, null, tail));
    tail = tail.next();
}

public Object remove() { // Remove and return current Object
    Assert.notFalse(isInList(), "No current element");
    Object it = curr.next().element(); // Remember Object
    if (curr.next().next() != null)
        curr.next().next().setPrev(curr);
    else tail = curr; // Removed last Object: set tail
    curr.setNext(curr.next().next()); // Remove from list
    return it; // Return value removed
}

public void prev() // Move curr to previous position
{ if (curr != null) curr = curr.prev(); }

```

图 4.16 双链表类 insert、append、remove 和 prev 四个成员函数的实现

new 操作符的 3 个参数允许表结点类的构造函数分别对新结点的 element、next 和 prev 这三个域赋值,并且返回指向新建立结点的指针。new 操作返回被创建的新指针,然后变量 curr 的 next 域被设置为指向新结点,最后一步修改新插入结点的下一个结点的 prev 域。if 语句用来保证这样的当前结点 curr 真的存在(而不是被插入到表尾)。

图 4.16 中 List 类的 append 函数因为不关心下一个结点而更简单。下面给出关键的代码:

```

tail.setNext(new DLink(it, null, tail));
tail = tail.next();

```

这里,当执行 new 操作时,Link 类的构造函数设置新结点的 element、next 和 prev 这三个域。由于新结点要被放到表尾,所以它的 next 域初始化为空。表中最后一个结点(由 tail 指向)的 next 域设置为指向新结点。最后,设置 tail 指向新添加的结点。

函数 remove(参见图 4.17)的程序代码尽管有些长,但是容易理解。首先,变量 it 指向要被删除的元素,下面四行代码进行指针的调整:

```

if (curr.next().next() != null)
    curr.next().next().setPrev(curr);
else tail = curr; // Removed last Object: set tail
curr.setNext(curr.next().next()); // Remove from list

```

头两行在被删除结点存在后继结点时设置后继结点的 prev 域。如果不存在后继,则进行第三行的调整使得 tail 必须指向被删除结点的前驱结点。最后,修改被删除结点的前驱结点的 next 域。函数 remove 的最后一步返回被删除元素的值。

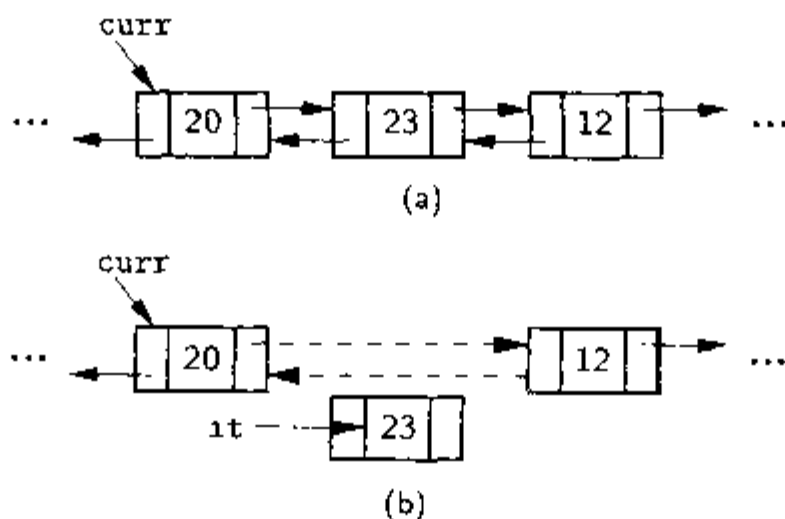


图 4.17 双链表的删除。指针 *it* 指向当前结点,然后调整被删除结点两边结点的指针

双链表与单链表相比唯一的缺点就是使用的空间更多。双链表的每一个结点需要两个指针。在上面介绍的实现方法中,它需要的结构性开销是单链表的两倍。在某些语言<sup>①</sup>中,有一种节省空间的方法可以用来消除额外的空间需求,尽管它会使实现变得复杂而且速度稍微减慢。这种方法是一个空间/时间权衡使用的例子。它基于异或函数(XOR)的以下性质(下面用“^”代表异或操作):

$$(L \oplus R) \oplus R = L$$

$$(L \oplus R) \oplus L = R$$

也就是说,给出两个值,并将它们异或,则其中任何一个值都可以由另外一个值与它们异或后的结果再异或而得到。因此,双链表可以通过在一个指针域中存储两个指针值的异或结果来实现。当然,要恢复其中一个值必须知道另外一个值。只要给出指向表中某一个结点的指针以及它的两个链接域中的任何一个值,就可以按照顺序访问到表中所有其他结点,这是因为指向这个结点的指针与此结点的后继结点的 *prev* 域的指针是相同的。因此,只要分解开链接域就能够顺着链表走下去,像打开拉链一样。

这种方法的原理值得我们注意,因为它有许多应用。在计算机图形学中就广泛利用了这一原理。例如,把计算机屏幕上的一个区域与一个矩形图像异或,可以使屏幕上该区域的图像成为高亮的,再次与这个矩形图像异或就恢复成屏幕的原来内容。另一个例子用下面的代码来说明,不用临时变量交换两个变量的内容(以三个 XOR 操作为代价):

```
a = a ^ b;
b = a ^ b;    // Now b contains the original value of a
a = a ^ b;    // Now a contains the original value of b
```

#### 4.1.6 循环链表

有些应用不需要线性表中有明显的头尾元素。这种情况下,可能需从最后一个元素方便地访问到第一个元素。一般的链表最后一个元素的 *next* 域存储的值为 *null*,如果把它存储为指向线性表中第一个元素的指针,就建立了一个循环链表(*circular list*)。在这种实现方法中就不需要 *tail* 指针了。图 4.18 是一个循环单链表的示例。图 4.19 是一个循环双链表的示

<sup>①</sup> Java 不支持这种技术,因为它不支持异或操作,也不允许针对指针的算术操作。

例。这种循环链表实现方法的危险之处在于由于表中没有明显的尾端,可能使得链表的处理操作进入死循环。然而,可以使用 head 指针来标记表的处理操作是否周游了整个表。

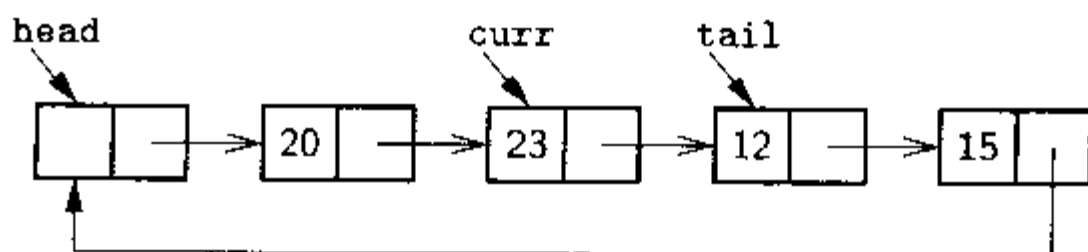


图 4.18 一个循环单链表

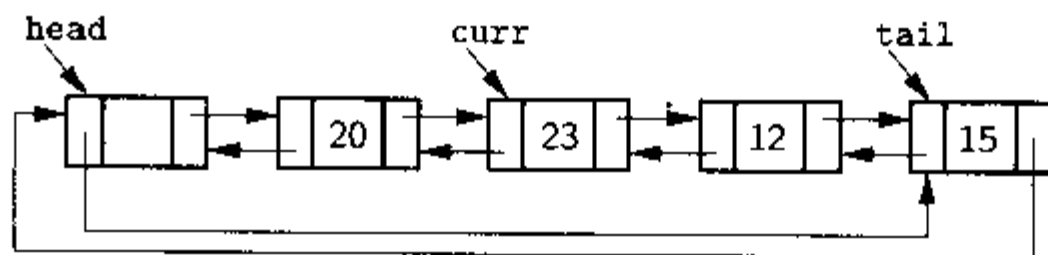


图 4.19 一个循环双链表

## 4.2 栈

栈(stack)是限定仅在一端进行插入或删除的线性表。虽然这个限制减小了栈的灵活性,但是它也使得栈更有效且更容易实现。许多应用都只需要提供受限制的插入和删除操作形式,在这种情况下使用较简单的栈这种数据结构比使用一般的线性表更有效。例如,4.1.2 小节的可利用空间表实际上就是一个栈。

尽管栈受到限制,它还是有广泛的应用,因此形成了栈的一个特殊术语集。早在计算机发明之前,会计就使用过栈的记号,他们称栈为“LIFO”线性表,意思是“后进先出”(Last In First Out)。“LIFO”的原则隐含着栈存储和删除元素的顺序与元素到达的顺序相反。

习惯上称栈的可访问元素为栈顶(top)元素,元素插入栈称作压栈(push),删除元素时称作出栈(pop)。

就线性表而言,实现栈的方法多种多样。这里介绍两种方法:顺序栈(array-based stack)和链式栈(linked stack),它们分别类似于顺序表和链表。

### 4.2.1 顺序栈

图 4.20 给出了顺序栈类的整个实现。其中, listarray 在建立栈时必须说明一个固定的长度。在栈的构造函数中, sz 用来表示栈的大小。成员 top 表示当前所在的位置值,亦指当前栈中元素的数目。由于当前位置总是在栈顶,所以不需要另外一个成员来保留栈的当前位置。

```
class AStack {                                // Array based stack class

    private static final int defaultSize = 10;
```



```

private int size;           // Maximum size of stack
private int top;           // Index for top Object
private Object[] listarray; // Array holding stack Objects

AStack() { setup(defaultSize); }
AStack(int sz) { setup(sz); }

public void setup(int sz)
{ size = sz; top = 0; listarray = new Object[sz]; }

public void clear()           // Remove all Objects from stack
{ top = 0; }

public void push(Object it) { // Push Object onto stack
    Assert.notFalse(top < size, "Stack overflow");
    listarray[top++] = it;
}

public Object pop() { // Pop Object from top of stack
    Assert.notFalse(! isEmpty(), "Empty stack");
    return listarray[--top];
}

public Object topValue() { // Return value of top Object
    Assert.notFalse(! isEmpty(), "Empty stack");
    return listarray[top - 1];
}

public boolean isEmpty() // Return true if stack is empty
{ return top == 0; }
} // class AStack

```

图 4.20 顺序栈类的实现

顺序栈的实现,本质上是顺序表实现的简化。惟一重要的是确定应该用数组的哪一端表示栈顶。一种选择是把数组的第 0 个位置作为栈顶。根据线性表的函数,所有的插入(insert)和删除(remove)操作都在第 0 个位置的元素上进行。由于这时每次 push(insert)或者 pop(remove)操作都需要把当前栈中的所有元素在数组中移动一个位置,因此效率不高。如果栈中有  $n$  个元素,则时间代价为  $\Theta(n)$ 。另一种选择是当栈中有  $n$  个元素时把位置  $n - 1$  作为栈顶。也就是说,当向栈中压入元素时,把它们添加到线性表的表尾,成员函数 pop 也是删除表尾元素。在这种情况下,每次 push 或者 pop 操作的时间代价仅为  $\Theta(1)$ 。

对于图 4.20 的实现方法, `top` 被定义为表示栈中的第一个空闲位置<sup>①</sup>。成员函数 `push` 和 `pop` 只是从 `top` 指示的数组中的位置插入或者删除一个元素。因为 `top` 表示第一个空闲位置, 所以 `push` 首先把一个值插入到栈顶位置, 然后把 `top` 加 1。同样, `pop` 首先把 `top` 减 1, 然后删除栈顶元素。

对顺序栈的实现方法稍作修改就可以支持变长元素, 如字符串。假定把栈数组分为等长的存储单元, 例如一个字符或者整数长度, 每个栈元素可能存放在若干个基本存储单元中。如果能把栈元素所占用存储单元的数目存放在一个存储单元内, 那么每个栈元素所占用的存储单元的数目, 就可以存放在为这个栈元素分配的若干个存储单元中最上面的那个存储单元内。 `push` 操作将把一个需要  $i$  个存储单元的元素存储到从 `top` 的当前值开始的  $i$  个位置中, 然后在 `top + i` 的位置存放数值  $i$ , 把 `top` 的值重置为 `top + i + 1`。 `pop` 操作需要先查看存储在 `top - 1` 位置的长度值, 然后弹出适当数目的存储单元。图 4.21 是一个存储变长字符串的顺序栈的示例。

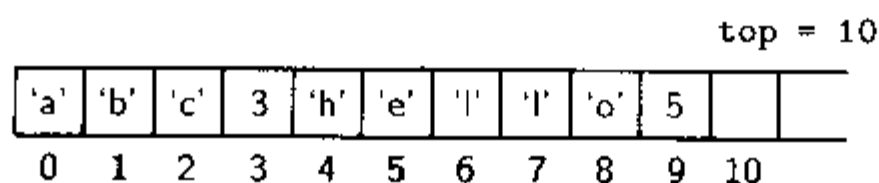


图 4.21 一个存储变长字符串的顺序栈。每个位置存储一个字符或者一个整数, 这个整数说明栈中位于其左边的字符串长度

### 4.2.2 链式栈

链式栈的实现是对链表实现的简化。4.1.2 小节的可利用空间表就是一个链式栈的实例。它的元素只能在表头进行插入和删除。由于空栈或者只有一个元素的栈都不需要特殊情形的结点, 所以它不需要表头结点。图 4.22 给出了链式栈的全部类实现。其中惟一的一个数据成员是 `top`, 它是一个指向链式栈第一个结点(栈顶)的指针。

```
class LStack {                                     // Linked stack class
private Link top;                                  // Pointer to list header

public LStack() { setup(); }                       // Constructor
public LStack(int sz) { setup(); }                 // Constructor: ignore sz

private void setup() { top = null; }               // Initialize stack

public void clear() { top = null; }                // Remove Objects from stack

public void push(Object it)                       // Push Object onto stack
{ top = new Link(it, top); }

public Object pop() {                             // Pop Object at top of stack
```

<sup>①</sup> `top` 也可以定义为表示栈中最上面那个元素的位置, 而不表示第一个空闲位置。如果这样的话, 则空线性表应该把 `top` 初始化为 -1。

```

    Assert.assertFalse(! isEmpty(), "Empty stack");
    Object it = top.element();
    top = top.next();
    return it;
}

public Object topValue()           // Get value of top Object
{ Assert.assertFalse(! isEmpty(), "No top value");
  return top.element(); }

public boolean isEmpty()           // Return true if empty stack
{ return top == null; }
} // class LStack

```

图 4.22 链栈类的实现

成员 `push` 首先修改新产生的链表结点的 `next` 域并指向栈顶, 然后设置 `top` 指向新的链表结点。成员 `pop` 也十分简单: 变量 `it` 用来存储栈顶结点的值, 把 `top` 指向当前栈顶链接到的下一个结点, 原来栈顶的值 `it` 作为 `pop` 函数的返回值。

### 4.2.3 顺序栈与链式栈的比较

实现顺序栈和链式栈的所有操作都只需要常数时间, 因此惟一可以比较的是所需要的时间。下面的分析与对线性表的实现所做的分析类似。初始时顺序栈必须说明一个固定的长度, 当栈不够满时, 一些空间被浪费掉了。链式栈的长度可变, 但是对于每个元素都需要一个链接域, 从而产生了结构性开销。

当需要实现多个栈时, 可以充分利用顺序栈单向延伸的特性。因此, 可以使用一个数组来存储两个栈, 每个栈从各自的端点向中间延伸, 如图 4.23, 这样浪费的空间就会减少。但是, 只有当两个栈的空间需求有相反的关系时这种方法才奏效。也就是说, 最好一个栈增长时另一个栈缩短。当需要从一个栈中取出元素放入另一个栈时, 这种方法非常有效。反之, 如果两个栈同时增长, 则数组中间的可用空间很快就会用尽。

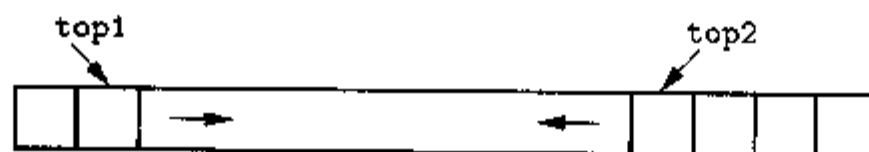


图 4.23 存储在同--一个数组中, 两个彼此迎面增长的栈

### 4.2.4 递归的实现

栈的最广泛用途用户也许是看不到的, 这就是大多数程序设计语言都有的子程序调用。子程序调用通过把有关子程序的必要信息(包括返回地址、参数、局部变量)存储到一个栈中来实现, 这块信息称为活动记录(activation record)。紧接着的子程序调用都把活动记录压入栈中。每次从子程序中返回时, 就从栈中弹出一个活动记录。图 4.24 从编译器的角度说明了

## 2.4 节递归阶乘函数的实现。

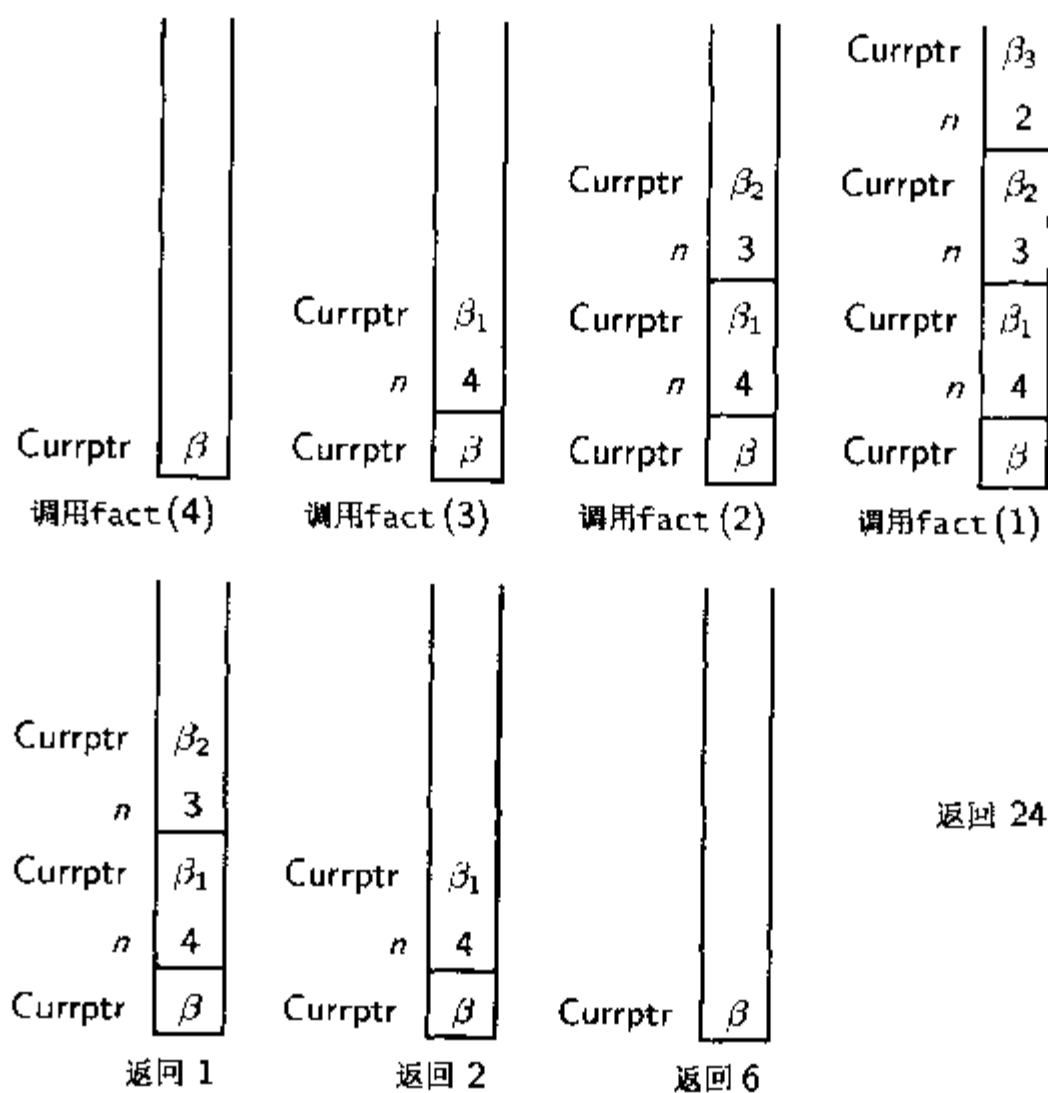


图 4.24 用栈实现递归。 $\beta$  值表示完成当前函数调用后要返回到的程序指令地址。每一次递归调用函数  $\text{fact}$  (参见 2.4 节) 都需要保存返回地址和当前的  $n$  值。每次从  $\text{fact}$  返回都要把最上面的活动记录弹出栈

由于需要生成一个活动记录并把它压入栈中,所以每一次子程序调用都是一个相对来说耗时较多的操作。使用递归方法实现比较容易而且清晰,但是有时我们希望避免由递归函数调用而产生的庞大的时间和空间开销。可能的情况下,最有效的方法是用迭代来代替递归,例如阶乘函数就很容易用迭代来实现。

不幸的是,并不总能用迭代来代替递归。当实现有多个分支的算法时,很难用迭代来代替递归,所以必须使用递归或与递归等价的算法。例如,汉诺塔算法、周游二叉树算法以及第 8 章的归并排序和快速排序算法都必须使用递归。值得庆幸的是,可以使用栈来模拟递归。

下例用栈代替递归实现了阶乘函数的非递归版本。

```
static long fact(int n) { // Must have n < 21 to fit in long
    Assert.IsFalse((n >= 0) && (n < 21), "Input out of range");
    Stack S = new AStack(n - 1); // Make stack just big enough
    while (n > 1) S.push(new Integer(n--));
    long result = 1;
    while (!S.isEmpty())
        result = result * ((Integer)S.pop()).longValue();
    return result;
}
```

在此,我们简单地把  $n-1$  压入栈顶,同时把  $n$  减少 1。依此类推,不断地操作,直到遇到基本条件为止。然后连续地弹出栈顶元素,并把该值与结果 result 相乘<sup>①</sup>。

实际上阶乘函数的迭代实现比这个用栈实现的版本简单快捷得多。但是,让我们来看汉诺塔函数的非递归实现,它是无法用迭代来完成的。问题是图 2.2 的 TOH 函数有两个递归调用,第一个把  $n-1$  个圆盘与最底层的圆盘分离,另一个把这  $n-1$  个圆盘放回到目标柱。为了消除递归,我们使用栈来存储代表 TOH 必须执行的三个操作:两个递归调用和一个移动操作。为此,我们要用一个类来表示这三个不同的操作,该类的对象就是存储在栈中的元素。下面是类说明。

```
class TOHobj {      // Stack element for Tower of Hanoi
    public int op;    // Operation (imitate recursion or move)
    public int num;   // Size of pile to move (recursion level)
    public Pole start, goal, temp;

    TOHobj(int o, int n, Pole s, Pole g, Pole t) // TOH
    { op = o; num = n; start = s; goal = g; temp = t; }

    TOHobj(int o, Pole s, Pole g)                // MOVE
    { op = o; start = s; goal = g; }
}
```

类 TOHobj 存储了五个域:一个操作域(说明是一个移动还是一个新的 TOH 操作)、圆盘的数目和三个柱子。注意移动操作实际上只需要存储两个柱子的信息。有两个构造函数:一个存储模拟递归调用时的状态,另一个存储移动操作的状态。TOH 的非递归版本如下:

```
static final int MOVE = 1;      // Move operation indicator
static final int TOH = 2;       // TOH operation indicator
static void TOH(int n, Pole start, Pole goal, Pole temp) {
    Stack S = new AStack(2 * n + 1); // Make stack just big enough
    S.push(new TOHobj(TOH, n, start, goal, temp)); // Initial form
    while (! S.isEmpty()) {
        TOHobj it = (TOHobj)S.pop(); // Grab next task
        if (it.op == MOVE)           // Do a move
            move(it.start, it.goal);
        else if (it.num > 0) {        // Imitate three statements in TOH
            // recursive solution (in reverse)
            S.push(new TOHobj(TOH, it.num-1, it.temp, it.goal, it.start));
            S.push(new TOHobj(MOVE, it.start, it.goal)); // A move to do
            S.push(new TOHobj(TOH, it.num-1, it.start, it.temp, it.goal));
        }
    }
}
```

<sup>①</sup> 注意 Stack 类存储的元素类型为 Object,也就是说元素一定要是真正的类类型(class type),因此简单的(基本类型) int 变量不能在此使用。可以用 Integer 类型来代替,从 int 到 long 类型都需要显式地转换。

首先定义两个常量 TOH 和 MOVE 来指示要执行的操作。注意我们使用了顺序栈,因为我们知道栈中恰好要存放  $2n + 1$  个元素。一开始,TOH 的新版本就把  $n$  个圆盘的初始状态存入栈中。函数剩余部分只是一个简单的 while 循环:压栈,执行相应的操作。对于 TOH 的一个操作( $n > 0$  时),我们存储递归版本三个操作的栈状态。当然,这些操作必须在栈中逆序存放,这样才能正确地弹出来。

一些“自然递归”的应用实例都已经用栈有效地实现了,这是由于需要用来描述它们的子问题的信息不多。例如,第 8.4 节的图 8.9 描述了快速排序的一种基于栈的实现方法。

### 4.3 队列

同栈一样,队列(queue)也是一种受限的线性表。队列元素只能从队尾插入(称为入队操作,enqueue),从队首删除(称为出队操作,dequeue)。队列操作像在电影院前排队买票一样<sup>①</sup>。如果没有人为的破坏,那么新来者应该站到队列的后端,在队列最前面的人是下一个要被服务的对象。注意,队列是按照到达的顺序存储的。早在计算机出现之前,会计就使用过队列,他们称队列为“FIFO”线性表,意思是“先进先出”(First In First Out)。这一部分介绍队列的两种实现方法:顺序队列和链式队列。

#### 4.3.1 顺序队列

有效地实现顺序队列(array-based queue)有些棘手,因为如果只是对顺序表的实现进行简单转化,效率不会很高。

假设队列中有  $n$  个元素,实现顺序表需要把所有元素都存储在数组的前  $n$  个位置上。如果我们选择把队列的尾部元素放在位置 0,则 dequeue 操作的时间开销仅为  $\Theta(1)$ ,因为队列最前面的一个元素(要被删除的元素)是数组最后面的一个元素(处于位置  $n - 1$ )。但是 enqueue 操作的时间开销为  $\Theta(n)$ ,因为必须把队列中当前  $n$  个元素的每一个都在数组中移动一个位置。如果反过来,把队列的尾部元素放在位置  $n - 1$ ,则 enqueue 操作相当于线性表的 append 操作,时间开销仅为  $\Theta(1)$ ,但是此时 dequeue 操作的时间开销就为  $\Theta(n)$  了。因为为了保持剩下的  $n - 1$  个队列元素仍然在数组的前  $n - 1$  个位置,所有的元素都必须移动一位。

如果放宽队列的所有元素必须处在数组的前  $n$  个位置这一条件,就可以得到一种更有效的实现方法。仍然保证队列的元素存储在连续的数组位置中,但是队列的内容允许在数组中移动,如图 4.25 所示。此时 enqueue 操作和 dequeue 操作的时间开销均为  $\Theta(1)$ ,因为队列中没有任何需要移动的元素。

但是这种实现方法有一个新的问题,如下面的情况所示。假设初始时队列的头在位置 0,新元素连续插入到数组中编号较高的位置上。当从队列中删除元素时,front 的值增加。随着时间的推移,整个队列向数组中编号较高的位置移过去。尽管此时数组的低端还可能有空闲的位置,即先前从队列中删除的元素所占用的位置,一旦一个元素被插入到数组中编号最高的

<sup>①</sup> 在英国,一队人称为一个“queue”(队列),进入队列等候服务称为“queuing up”(排队)。

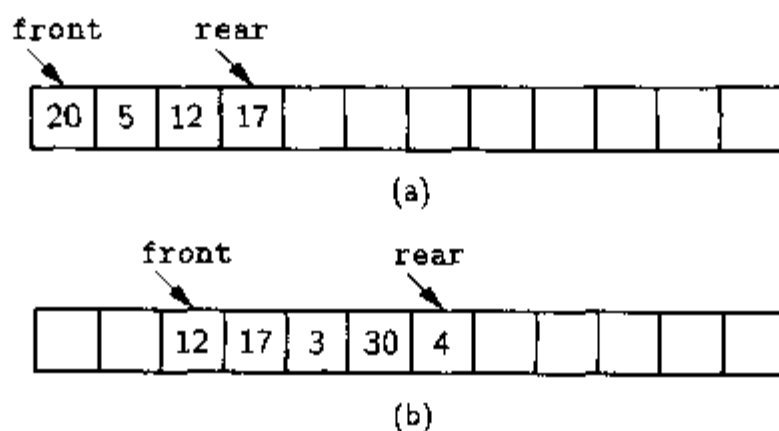


图 4.25 经过多次使用后,顺序队列的元素将移动到数组的后面

(a)插入初始四个数 20、5、12 和 17 后的队列。

(b)先删除 20 和 5,然后插入 3、30 和 4 之后的队列

位置上之后,队列的空间就用尽了。

“移动队列”问题可以通过假定数组是循环的来解决,即允许队列直接从数组中编号最高的位置延续到编号最低的位置。使用取模操作可以很容易地实现它。在这种方法中,数组中位置的编号为 0 到  $size - 1$ ,  $size - 1$  被定义为位置 0 (等价于位置  $size \% size$ ) 的前趋。图 4.26 说明了这种方法。

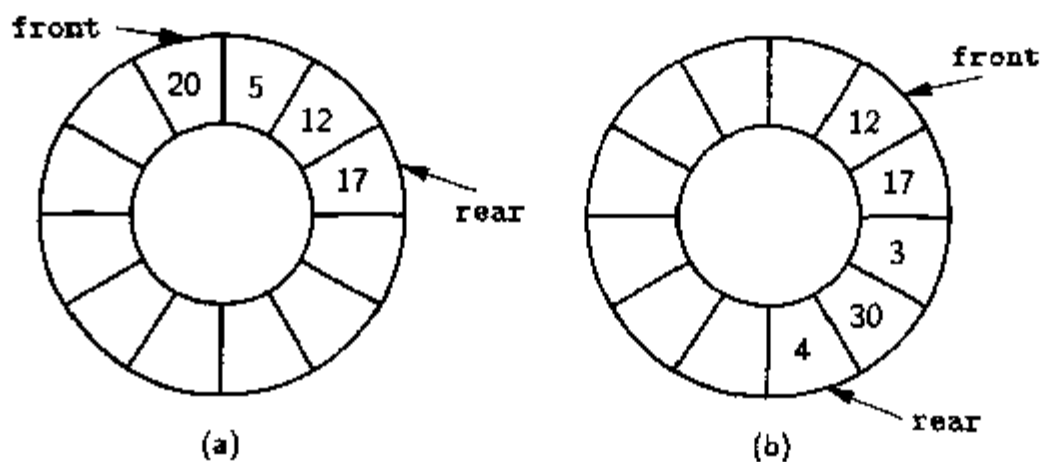


图 4.26 数组位置顺时针增长的循环队列

(a)插入初始四个数 20、5、12 和 17 后的队列。

(b)先删除 20 和 5,然后插入 3、30 和 4 后的队列

对于顺序队列的实现还有一个虽然小却十分重要的问题,就是如何判断队列是空还是满? 假设  $front$  存储队列中队首元素的位置号,  $rear$  存储队尾元素的位置号。如果  $front$  和  $rear$  的位置相同,那么在这种策略下队列中一定只有一个元素。因此,如果  $rear$  比  $front$  小 1 就表示是空队列(考虑循环队列,位置  $size - 1$  被认为比位置 0 小 1)。但是如果队列满会怎样呢? 也就是说当一个有  $n$  个可用数组位置的队列含有  $n$  个元素时情形会怎样呢? 在这种情况下,如果队列头的元素在位置 0,则队列尾的元素一定在位置  $size - 1$ 。这就意味着如果考虑循环队列,则  $rear$  的值比  $front$  的值小 1,即满队列和空队列无法区分!

你可能会认为问题出在假设中把  $front$  和  $rear$  分别定义为记录队列头和队列尾元素的位置号上,只要在定义中稍作修改就会得出结果。但是,只对  $front$  和  $rear$  的定义作简单的修改并不能补救这个问题,因为队列可能处于许多条件或状态之中。忽略队首元素的实际位置,忽略队列中存储的元素的实际值,有多少种不同的状态呢? 队列中可能没有元素、有 1 个元素、有 2 个元素,等等。如果数组有  $n$  个位置,则队列中最多有  $n$  个元素。也就是说,队列有  $n +$

1 种不同的状态(可能有 0 到  $n$  个元素)。

如果固定 front 的值,则 rear 应该有  $n + 1$  种不同的取值来区分  $n + 1$  种状态。但是实际上 rear 只有  $n$  种可能的取值,除非我们为队空发明一种特殊情形。这是习题 2.14 中定义的鸽笼原理的一个实例。鸽笼原理的意思是给定  $n$  个鸽笼和  $n + 1$  只鸽子,当所有的鸽子都进入笼中时,我们可以确信至少有一个鸽笼中的鸽子多于 1 只。同理,用 front 和 rear 的相对值, $n + 1$  种状态中必定有两种不能区分。因此我们必须寻求其他途径来区分满队列和空队列。

一种显然的方法是显式存储队列是否为空,用一个布尔变量来实现。这样就只剩下  $n$  个位置和  $n$  种状态了。另一种方法是设置数组的大小为  $n + 1$ ,但是只能存储  $n$  个元素。采用哪一种方法完全取决于你的兴趣,我的选择是使用大小为  $n + 1$  的数组。

图 4.27 介绍了一种顺序队列类的实现方法。通常, listArray 是一个指向存放队列元素数组的指针,队列的构造函数提供可选参数来设置队列最大长度。为了区分空队列和满队列,数组的大小实际上要比队列允许的最大长度大 1。成员 size 用来控制队列的循环(它是取模操作符的基数)。成员 rear 表示队尾元素的位置。为了简化实现,成员 front 表示队首元素的前趋位置。因此, front 和 rear 相等的队列为空队列。front 采用这种定义的优点在于使出队操作先给 front 加 1,再返回元素的值。

```
class AQueue { // Array-based queue class
    private static final int defaultSize = 10;
    private int size; // Maximum size of queue
    private int front; // Index prior to front item
    private int rear; // Index of rear item
    private Object[] listArray; // Array holding Objects

    AQueue() { setup(defaultSize); } // Constructor: default size
    AQueue(int sz) { setup(sz); } // Constructor: set size

    void setup(int sz) // Initialize queue
    { size = sz + 1; front = rear = 0; listArray = new Object[sz + 1]; }

    public void clear() // Remove all Objects from queue
    { front = rear = 0; }

    public void enqueue(Object it) { // Enqueue Object at rear
        Assert.notFalse(((rear + 1) % size) != front, "Queue is full");
        rear = (rear + 1) % size; // Increment rear (in circle)
        listArray[rear] = it;
    }

    public Object dequeue() { // Dequeue Object from front
        Assert.notFalse(! isEmpty(), "Queue is empty");
        front = (front + 1) % size; // Increment front
```



```

        return listArray[front];          // Return value
    }

    public Object firstValue() .          // Return value of front Object
    {
        Assert.notFalse(! isEmpty(), "Queue is empty");
        return listArray[(front + 1) % size];
    }

    public boolean isEmpty()              // Return true if queue is empty
    {
        return front == rear;
    }
} // class AQueue

```

图 4.27 顺序队列类的实现

在这种实现方法中,队首存放在数组中编号较低的位置(图 4.26 中沿顺时针方向),队尾存放在数组中编号较高的位置。因此,enqueue 增加 rear 指针的值(对 size 取模),dequeue 增加 front 指针的值(对 size 取模)。所有成员函数的实现都简单易懂。

### 4.3.2 链式队列

链式队列(linked queue)的实现对链表的实现做了简单的修改。图 4.28 给出链式队列类的说明。成员 front 和 rear 分别是指向队首和队尾元素的指针。链式队列的实现也不需要表头结点。链式队列的成员函数只需要对链表的相应部分稍作改动。因为这种实现方法没有表头结点,所以必须检查由 dequeue 函数导致的空队列的特殊情形。从本质上讲,enqueue 只是简单地把新元素放到链表尾部(rear 指向的结点),然后修改 rear 指针指向新的链表结点。dequeue 只是简单地去掉表中的最前面一个结点,并修改 front 指针。

```

class LQueue {                          // Linked queue class
    private Link front;                  // Pointer to front node
    private Link rear;                   // Pointer to rear node

    public LQueue() { setup(); }          // Constructor
    public LQueue(int sz) { setup(); }    // Constructor: Ignore sz

    private void setup()                 // Initialize queue
    {
        front = rear = null;
    }

    // Remove all Objects from queue
    public void clear() { front = rear = null; }

    // Enqueue Object at rear of queue
    public void enqueue(Object it) {
        if (rear != null) {              // Queue not empty: add to end
            rear.setNext(new Link(it, null));
            rear = rear.next();
        }
    }
}

```

```

|
| else front = rear = new Link(it, null); // Empty queue
|
|

public Object dequeue() :           // Dequeue Object from front
    Assert.notNull(! isEmpty());    // Must be something to dequeue
    Object it = front.element();     // Store dequeued Object
    front = front.next();            // Advance front
    if (front == null) rear = null;  // Dequeued last Object
    return it;                       // Return Object

public Object firstValue()           // Return value of top Object
{ Assert.notNull(! isEmpty()); return front.element(); }

public boolean isEmpty()             // Return true if queue is empty
{ return front == null; }
| // class LQueue

```

图 4.28 链式队列类的实现

### 4.3.3 顺序队列与链式队列的比较

实现顺序队列和链式队列的所有成员函数都需要常数时间。空间比较问题与栈实现类似。只是顺序队列不像顺序栈那样,它不能在一个数组中存储两个队列,除非总是有数据项从一个队列转入另一个队列。在顺序队列中存储变长记录的方式与顺序栈的方式相同。

## 4.4 习题

- 4.1 假设一个线性表包含下列元素:(2,23,15,5,9),使用线性表 List 接口编写一些 Java 语句删除值为 15 的元素。
- 4.2 4.1.3 小节中指出“实现顺序表的空间需求为  $\Omega(n)$  或者更多”,解释其原因。
- 4.3 使用图 4.1 的 ADT 写出线性表在经过每一步操作后的形式,假定线性表 L1 和 L2 在每一组操作时,都初始化为空,说出线性表中当前元素的位置。
  - (a) L1.append(10);  
     L1.append(20);  
     L1.append(15);
  - (b) L2.append(10);  
     L2.append(20);  
     L2.append(15);  
     L2.first();  
     L2.insert(39);  
     L2.next();

`T2.insert(12);`

- 4.4 编写一些 Java 语句,用线性表的 ADT 建立一个能存放 20 个元素而实际只存储了序列(2,23,15,5,9)的线性表。
- 4.5 4.1.3 小节介绍了一个公式,用来确定线性表的两种实现方法的空间需求临界值。其变量为  $D$ 、 $E$ 、 $P$  和  $n$ ,每个变量的单位是什么? 方程两边按单位平衡吗?
- 4.6 已知元素的大小为 8 个字节,一个指针的大小为 4 个字节,数组的大小为 20 个元素,用 4.1.3 小节的空间方程求顺序表和链表实现的临界值。
- 4.7 修改图 4.5 的代码,实现循环单链表。
- 4.8 修改图 4.16 的代码,实现循环双链表。
- 4.9 使用图 4.1 的线性表 ADT 编写一个函数,交换当前元素和它在表中的后继元素。
- 4.10 在链表类的实现中增加一个成员函数,实现对表中元素置逆的操作(设原链表为  $a_1, a_2, \dots, a_{n-1}, a_n$ ; 则置逆后的序列为  $a_n, a_{n-1}, \dots, a_2, a_1$ )。对于具有  $n$  个元素的线性表,你的算法的运行时间应为  $\Theta(n)$ 。
- 4.11 修改图 4.20 的代码,实现存储于一个数组的两个栈。
- 4.12 4.3.1 小节中把成员 `front` 定义为表示队首元素的前趋位置,是为了使出队操作先给 `front` 加 1,再返回元素的值。请解释为什么这样做比把 `front` 定义为表示队首元素要好。
- 4.13 确定你的计算机上 Java 的 `long` 类型变量和指针的大小,然后按 4.1.3 小节的方法计算临界值,超过这个临界值时,对元素类型为 `long` 的线性表,使用顺序表实现比使用链表实现的空间效率更高。
- 4.14 修改图 4.27 顺序队列的定义,使用一个独立的布尔成员记录队列是否为空,而不用在数组中留一个空位置。
- 4.15 回文(`palindrome`)是指一个字符串从前面读和从后面读都一样。仅使用若干栈和队列、栈和队列的 ADT 函数以及若干个 `int` 类型和 `char` 类型的变量,编写一个算法来判断一个字符串是否为回文。假设字符串从标准输入设备一次一个字符地读入,算法的输出结果应为 `true` 或 `false`。
- 4.16 已知  $Q$  是一个非空队列,  $S$  是一个空栈。仅用栈和队列的 ADT 函数和一个成员变量  $X$  编写一个算法,使得  $Q$  中的元素倒置。
- 4.17 一个未排序的整数顺序表允许在常数时间内简单地把一个新整数插入表尾。但是在有  $n$  个整数的线性表中查找一个关键码值为  $X$  的整数需要的平均时间为  $\Theta(n)$ 。对比之下,对一个已排序的包含  $n$  个整数的顺序表使用二分法检索需要的时间是  $\Theta(\log n)$ 。然而,由于数组中的许多整数可能被移动,所以插入一个新整数的时间为  $\Theta(n)$ 。如果想使得插入和检索均在  $\Theta(\log n)$  时间完成,数据应该如何组织?
- 4.18 编译器和文本编辑器的一个普遍问题是判断一个字符串中的圆括号(或者其他括号)是否平衡且匹配。例如,字符串“((( )))”中的圆括号恰好平衡且匹配,但是字符串“())( )”中的圆括不平衡,字符串“())”中的圆括号不匹配。  
(a) 给出一个算法,当字符串中的圆括号恰好平衡且匹配时返回 `true`, 否则返回 `false`。用一个栈来记录当前扫描到的未匹配的左圆括号。提示:从左到右扫描

一个合法的字符串,保证任何时候所遇到的右圆括号不比左圆括号多。

- (b)给出一个算法,如果字符串不平衡或者不匹配则返回字符串中第一个非法圆括号的位置。也就是说,如果发现一个多余的右圆括号,则返回它的位置;如果有多个左圆括号,则返回第一个多余的左圆括号的位置;如果字符串平衡且恰好匹配,则返回-1。使用一个栈记录当前扫描到的左圆括号的数目和位置。

## 4.5 项目设计

- 4.1 修改链表的实现,以支持有序线性表。也就是说,线性表的元素值按照递增的顺序从低到高存储。首先要修改 List 的接口,因为某些成员函数不再有效。例如,不可能把一个元素插入到有序线性表的任意位置。
- 4.2 4.1 节介绍的线性表的 Java 实现不支持元素类型不同的线性表。修改 List 类的 ADT,以及顺序表和链表的实现,允许用户创建 List 对象时指明元素所属的类。List 类的成员函数应该保证只有属于这种类型的元素才可以插入到线性表中。
- 4.3 使用单链表实现不限大小的整数,应实现加、减、乘和指数操作,指数运算限制为正整数,则每次操作的渐进运行时间是多少?用每个函数的两个操作数的位数(数字的个数)来表示。
- 4.4 用无序线性表实现一个城市数据库。每条数据库记录包括城市名(任意长的字符串)和城市的坐标(用整数  $x$  和  $y$  表示)。你的数据库应该允许插入记录、按照名字或者坐标删除或检索记录,还应该支持打印在指定点给定距离内的所有记录。先使用顺序表实现,然后用链表实现。记录用这两种方法实现的每次操作的运行时间。这两种实现方法的相对优、缺点是什么?如果按照城市名的字母顺序来存储记录,使得线性表成为有序的,这样能加速一些操作吗?这种按照城市名排序的线性表会减慢一些操作吗?

## 第5章 二叉树

第4章讲述的链表的实现都有一个基本限制:要么检索速度快,要么易于插入新结点,但是不能二者兼备。对于已有的数据,树型结构能够进行高效的创建与更新。尤其是二叉树,它被广泛地应用,并且相对容易实现。本章首先介绍二叉树的定义与主要特性。5.2节讨论如何使数据有序地存放到树中。5.3节介绍实现二叉树的各种方法。5.4节至5.6节介绍二叉树的特殊应用:用于文件压缩的 Huffman 编码树,利用二叉检索树进行搜索,以及用堆实现优先队列(priority queue)。

### 5.1 定义及主要特性

**定义 5.1** 二叉树(binary tree)由结点(node)的有限集合组成,这个集合或者为空(empty),或者由一个根结点(root)以及两棵不相交的二叉树组成,这两棵二叉树分别称作这个根的左子树(left subtree)和右子树(right subtree)。这两棵子树的根称为此二叉树根结点的子结点(children)。从一个结点到它的两个子结点都有边(edge)相连,这个结点称为它的子结点的父结点(parent)。

如果一棵树的一串结点  $n_1, n_2, \dots, n_k$  有如下关系:结点  $n_i$  是  $n_{i+1}$  的父结点 ( $1 \leq i < k$ ),就把  $n_1, n_2, \dots, n_k$  称为一条由  $n_1$  到  $n_k$  的路径(path)。这条路径的长度(length)是  $k-1$ 。如果有一条路径从结点 R 至结点 M,那么 R 就称为 M 的祖先(ancestor),而 M 则称为 R 的子孙(descendant)。因此,所有的结点都是根结点的子孙,而根结点是它们的祖先。

**定义 5.2** 结点 M 的深度(depth)就是从根结点到 M 的路径的长度。树的高度(height)等于最深的结点的深度+1。任何深度为  $d$  的结点的层数(level)都为  $d$ 。根结点的层数为 0,深度也为 0。没有非空子树的结点称为叶结点(leaf)。至少有一个非空子树的结点称为分支结点或称为内部结点(internal node)。

图 5.1 解释了定义 5.1 与定义 5.2。图 5.2 说明了二叉树结构的几个要点。由于必须区分二叉树结点的左右子结点,所以图 5.2(a)与图 5.2(b)表示的是不同的二叉树。

请注意下面定义的特殊二叉树的名称。

**定义 5.3** 满二叉树(full binary tree)的每一个结点或者是一个分支结点,并恰有两个非空子结点;或者是叶结点。完全二叉树(complete binary tree)有严格的形状要求:从根结点起每一层从左到右填充。一棵高度为  $d$  的完全二叉除了  $d-1$  层以外,每一层都是满的。底层叶结点集中在左边的若干个位置上。

图 5.3 说明了满二叉树与完全二叉树的区别。二者之间并没有任何特别的关系。即:二

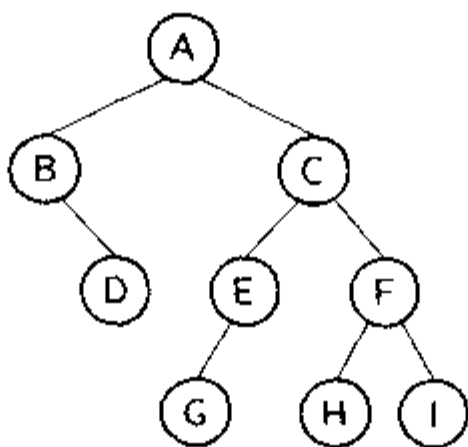


图 5.1 二叉树举例：结点 A 是根结点，B、C 是 A 的子结点。结点 B 与 D 一起是一棵子树。B 的两个子结点：左子结点是空树，右子结点是 D。结点 A、C 和 E 是 G 的祖先，结点 D、E 和 F 的层数为 2，结点 A 的层数为 0。从 A 到 C 到 E 到 G 的边形成了一条长度为 3 的路径。结点 D、G、H 和 I 是叶结点，A、B、C、E 和 F 是内部结点。结点 I 的深度为 3。这棵树的高度是 4。

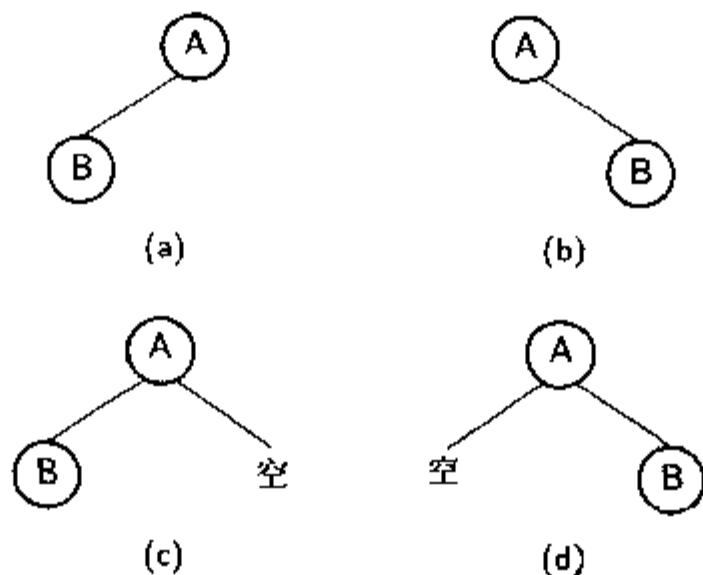


图 5.2 两棵不同的二叉树

(a) 根结点有非空左子结点。(b) 根结点有非空右子结点。(c) 显式地标明了二叉树(a)的右子结点为空。(d) 显式地标明了二叉树(b)的左子结点为空。

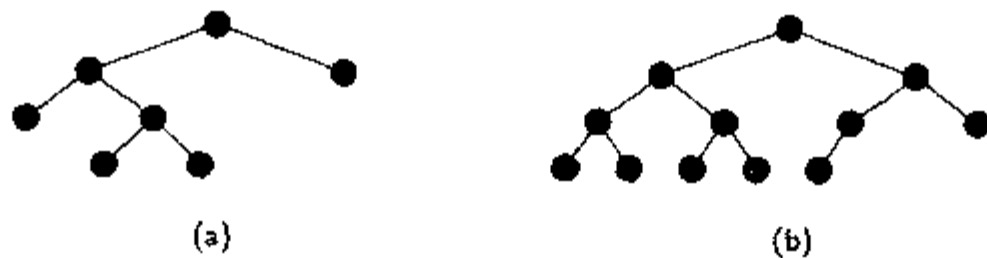


图 5.3 满二叉树与完全二叉树

(a) 满二叉树(非完全二叉树)。(b) 完全二叉树(非满二叉树)

叉树(a)是满二叉树，但不是完全二叉树。(b)是完全二叉树，但不是满二叉树<sup>①</sup>。Huffman 编码树(5.4 节)是满二叉树。堆数据结构(5.6 节)是一种完全二叉树。

<sup>①</sup> 虽然本书关于满二叉树、完全二叉树的定义是最广泛使用的一种，但是它们不是普遍被接受的，有些书甚至可能把两个定义倒过来。由于满与完全这两个词的意义十分接近，所以没有比记住定义的方法能更好区别它们了。这里介绍一种帮助记忆的方法：“完全”的词义比“满”宽一些，而完全二叉树一般比满二叉树要宽，因为完全二叉树的每一层都尽可能地宽。

### 5.1.1 满二叉树定理

某些二叉树的实现只用叶结点存储数据,而用分支结点来存储结构信息。更一般地说,二叉树的实现需要用一定空间来存储分支结点,这个空间的大小可能与存储叶结点的空间不同。因此,为了分析这种实现方式的空间开销,知道一棵具有  $n$  个分支结点的二叉树的叶结点在全部结点中可能的最小与最大比例是十分有用的。

但是,一棵有  $n$  个分支结点的二叉树可能只包含一个叶结点。例如,这些分支结点构成一个链,而以一个叶结点结束(如图 5.4)。在这种情况下,叶结点很少,因为每个分支结点都仅有一个非空子结点。为了使叶结点尽量多而分支结点仍为  $n$ ,首先想到的是使每一个分支结点都有两个非空子结点,即满二叉树。但是,这并没有说明什么形状的二叉树的叶结点所占比率会达到最高百分比。不过没有关系,因为任何有  $n$  个分支结点的满二叉树都有相同的叶结点数目。这个事实使我们能够计算出叶结点与分支结点占有不同空间的满二叉树的空间开销。下面的定理说明了任何有  $n$  个分支结点的满二叉树的叶结点数目。



图 5.4 包含  $n$  个分支结点与一个惟一叶结点的二叉树

**定理 5.1 满二叉树定理:**非空满二叉树的叶结点数等于其分支结点数加 1。

**证明:**对  $n$ (分支结点数)作数学归纳法。这是一个使用归纳法证明的例子:我们把一种任意参数为  $n$  的情况归约到一种参数为  $n-1$  的特例,而参数为  $n-1$  的情况符合归纳假设,因此定理得证。

- 初始情况:没有分支结点的非空二叉树有一个叶结点。有一个分支结点的满二叉树有两个叶结点,即当  $n=0$  及  $n=1$  时定理成立。

- 归纳假设:设任意一棵有  $n-1$  个分支结点的满二叉树有  $n$  个叶结点。

- 归纳步骤:假设树  $T$  有  $n$  个分支结点,取一个左右子结点均为叶结点的分支结点  $I$ 。去掉  $I$  的两个子结点,则  $I$  成为叶结点,把新树记为  $T'$ ,  $T'$  有  $n-1$  个分支结点,根据归纳假设,  $T'$  有  $n$  个叶结点,现在把两个叶结点归还给  $I$ 。我们又得到树  $T$  有  $n$  个分支结点。但是有几个叶结点呢? 既然  $T'$  有  $n$  个,再加上两个就有  $n+2$  个,但是在  $T'$  中结点  $I$  被计算为叶结点,而现在则是分支结点,于是,树  $T$  有  $n+1$  个叶结点和  $n$  个分支结点。

因此,根据归纳原理,定理对任意  $n \geq 0$  成立。

当度量二叉树的空间开销时,知道有多少个空子树是很有用的。简单的满二叉树定理的推论告诉我们任意一棵二叉树中空子树的数目,无论它是否为满二叉树。有两种方法可以证明,这两种方法对我们理解二叉树是有帮助的。

**定理 5.2 一棵非空二叉树空子树的数目等于其结点数加 1。**

**证明 1:**设二叉树  $T$ ,将其所有空子树换成叶结点,把新的二叉树记为  $T'$ 。所有原来树  $T$  的结点现在是树  $T'$  的分支结点。由于树  $T$  中的所有分支结点都有两个子结点,并且树  $T$  中的

每个叶结点在树  $T'$  中都有两个叶结点,所以树  $T'$  是满二叉树。根据满二叉树定理,新添加的叶结点数目等于树  $T$  的结点数目加 1,而每个新添加的叶结点对应树  $T$  的一棵空子树,因此树  $T$  中空子树的数目等于树  $T$  中结点数目加 1。

**证明 2:** 根据定义,树  $T$  中每个结点都有两个子结点,因此一棵(实际上)有  $n$  个结点的二叉树有  $2n$  个子结点。除了根结点以外,每个结点都有一个父结点,于是共有  $n-1$  个父结点,换句话说,就有  $n-1$  个非空子结点。既然子结点数目为  $2n$ ,则其中有  $n+1$  个为空。

### 5.1.2 二叉树的抽象数据类型

在讨论二叉树的应用之前,我们首先应该考虑实现二叉树时,怎样才能适合于二叉树的各种应用。例如,必须能够初始化二叉树,或者能够说明它为空。有些操作是二叉树应用所独有的。例如,我们希望使用把两个二叉树的根结点作为一个新根结点的两个子结点的方法来合并两棵二叉树。另外有些应用是围绕结点进行的,例如我们会需要访问某个结点的左子结点、右子结点、父结点,或者访问结点存储的数据。

显然,对于二叉树的操作,有些是关于结点的(如指向父结点或子结点),而另一些则是关于整棵树的(如初始化)。这说明在 Java 中必须分别实现二叉树与结点的类。这里提供了一种二叉树结点类的定义,这个类适合于本章下面将要出现的各种不同的二叉树类型。二叉树类的定义例子也将在讨论二叉树的应用时给出。

图 5.5 给出了一个二叉树结点的 Java 接口,称为 `BinNode`。与第 4 章中 `List` 类的实现一样,`BinNode` 类中存储了指向 `Object` 类的引用。创建二叉树时,可以根据应用需要而采用实际的数据类型。成员函数包括返回元素的值,返回左、右结点指针,设置元素的值,以及标志该结点是否为叶结点。

```
interface BinNode { // ADT for binary tree nodes
    // Return and set the element value
    public Object element();
    public Object setElement(Object v);

    // Return and set the left child
    public BinNode left();
    public BinNode setLeft(BinNode p);

    // Return and set the right child
    public BinNode right();
    public BinNode setRight(BinNode p);

    // Return true if this is a leaf node
    public boolean isLeaf();
} // interface BinNode
```

图 5.5 二叉树结点的 ADT



## 5.2 周游二叉树

我们经常通过访问每个结点来访问整个二叉树,每次完成一项工作,如打印出每个结点的内容。按照一定顺序访问二叉树的结点,称为一次周游或遍历(traversal)。对每个结点都进行一次访问并列出来,称为二叉树结点的枚举(enumeration)。有些应用并不要求按照某种顺序访问每个结点,尽管每个结点都只访问一次。而另一些应用则必须按照一定的顺序访问。例如,可能要求先访问结点,后访问其子结点,这称为前序周游(preorder traversal)。把图 5.1 的二叉树按照前序周游枚举出来的结果为:

A B D C E G F H I

打印的第一个结点是根结点,接下来打印所有左子树的结点,最后将右子树的结点打印出来。

类似地,我们也可以先访问结点的子结点(包括它们的子树),然后再访问该结点,这称为后序周游(postorder traversal)。把图 5.1 的二叉树按照后序周游枚举出来的结果就是:

D B C E H I F C A

中序周游(inorder traversal)则先访问左子结点(包括整个子树),然后是该结点,最后访问右子结点(包括整个子树)。对图 5.1 二叉树中序周游枚举出来的结果就是:

B D A G E C H F I

周游路线可以很容易地使用递归函数来表达。函数的输入项是指向结点 R 的指针。初始调用时传入根结点指针,然后按照既定的顺序周游 R 及其子结点(如果存在)。例如前序周游要求先访问 R,再访问它的子结点。这可以很容易地用 Java 语言来实现:

```
void preorder(BinNode rt) // rt is the root of the subtree
{
    if (rt == null) return; // Empty subtree
    visit(rt);
    preorder(rt.left());
    preorder(rt.right());
}
```

preorder 函数首先检查树是否为空(如果为空,则周游完成,并且函数直接返回),否则对根结点调用 visit 函数(例如打印结点的值,或者按照需要完成某些计算)。接着对左子树递归调用本函数,访问子树中的全部结点。最后,对右子树进行同样的操作。后序与中序周游的函数是类似的,只需要适当改变对结点与子结点访问的顺序即可。

## 5.3 二叉树的实现

这一节首先介绍利用指针来实现二叉树。接着从技术上讨论二叉树实现的空间开销。本节还包括使用数组实现完全二叉树。

### 5.3.1 使用指针实现二叉树

根据定义,每一个结点都有两个子结点,不论二者有一个为空还是都为空。通常二叉树的

结点都包含一个数据区,数据区所需空间大小根据需要而定。最常见的结点实现方法包含一个数据区和两个指向子结点的指针。图 5.6 给出了一个二叉树结点类 BinNodePtr 的说明。

```
// Binary tree node with pointers to children
class BinNodePtr implements BinNode {
    private Object element; // Object for this node
    private BinNode left;   // Pointer to left child
    private BinNode right;  // Pointer to right child

    public BinNodePtr() { left = right = null; } // Constructor 1
    public BinNodePtr(Object val) {                // Constructor 2
        left = right = null;
        element = val;
    }

    public BinNodePtr(Object val, BinNode l, BinNode r) // Construct 3
    { left = l; right = r; element = val; }

    // Return and set the element value
    public Object element() { return element; }
    public Object setElement(Object v) { return element = v; }

    // Return and set the left child
    public BinNode left() { return left; }
    public BinNode setLeft(BinNode p) { return left = p; }

    // Return and set the right child
    public BinNode right() { return right; }
    public BinNode setRight(BinNode p) { return right = p; }

    public boolean isLeaf() // Return true if this is a leaf node
    { return (left == null) && (right == null); }
} // class BinNodePtr
```

图 5.6 二叉树结点类的声明

BinNode 类包括一个 Object 类型的数据成员,称为 element。每个 BinNode 都有两个指针,一个指向左子结点,另一个指向右子结点。除了构造函数以外,还包括以下成员:返回和设置元素值、结点的左右子结点,以及判断该结点是否为叶结点。图 5.7 给出了用指针实现的二叉树的结构图。

有一些特殊的应用会增加一个指向父结点的指针,以便于向上搜索。增加一个父指针有点像在双链表中增加的指向前一结点的指针 prev。实际上,父指针通常是不必要的,并且增加了许多结构性开销。

在利用指针实现的二叉树中,叶结点与分支结点是否使用相同的类定义十分重要。有一

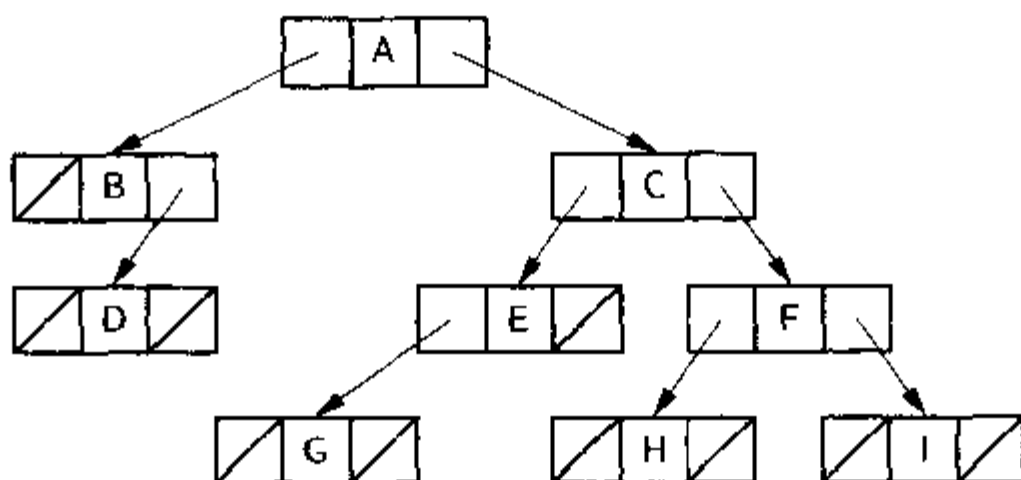


图 5.7 典型的使用指针实现的二叉树结构,其中每个结点存储两个子结点的指针和一个值。一些应用只需要用叶结点存储数据。还有一些应用要求分支结点与叶结点存储不同类型的数据。例如 13.1 节的二叉树,13.3 节的 PR 四分树,以及本节后面介绍的表达式树。根据定义,只有分支结点有非空子结点。因此,分别定义分支结点与叶结点将节省存储空间。

叶结点与分支结点分别存储不同类型数据的例子可以参考图 5.8 的表达式树。这个表达式树表示了一个代数式,其中包括双目运算符,如加、减、乘和除。分支结点存储运算符,而叶结点存储操作数。图 5.8 的表达式树表示表达式  $4x(2x + a) - c$ 。叶结点所需要的存储空间与分支结点是不同的,分支结点存储元素数目很少的操作符集合中的一个操作符,因此分支结点可以存储这个操作符的代码或者用一个字节存储其图形符号。叶结点则存储不同的变量名或数值,所以叶结点必须有足够大的数据区来存储各种可能的值。同时,叶结点不必存储子结点的指针。所以分别定义叶结点与分支结点能够节省空间。

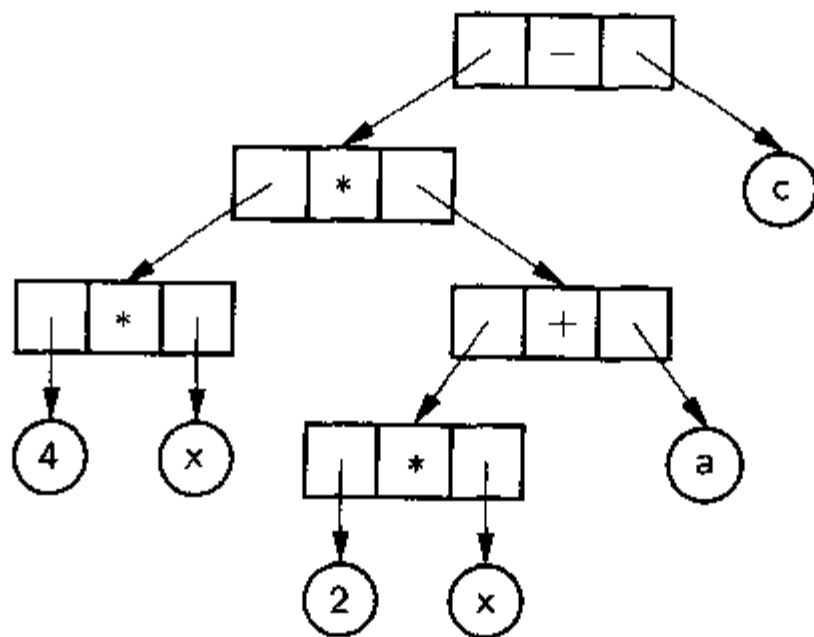


图 5.8 表达式  $4x(2x + a) - c$  的表达式树

许多程序设计语言支持分别定义叶结点与分支结点,例如 Pascal 和 C++ 分别采用变体记录 and 联合结构来实现,然而并不是所有语言都支持联合。另外,如果叶结点的子类型与分支结点的子类型的长度差别太大,就会使联合结构变得低效了。这是由于联合结构要求每个结点都足够大,以便存储最大的子类型。Java 的类继承可以提供更好的解决方案。在示例中,我们把 BinNode 接口作为基类,派生出两个独立的子类来分别实现叶结点和分支结点。分支结点(由 IntlNode 实现)存储操作符和指向类型 BinNode 的指针,这些指针并不知道子结点的真实类型。叶结点(由 LeafNode 实现)只存储叶结点值。检查一个结点时, isLeaf 函数返回该结点的真实子类型。图 5.9 给出了示例程序。

```

class LeafNode implements BinNode { // Leaf node subclass
    private String var;                // Operand value

    public LeafNode(String val) { var = val; } // Constructor
    public Object element() { return var; }
    public Object setElement(Object v) { return var = (String)v; }
    public BinNode left() { return null; }
    public BinNode setLeft(BinNode p) { return null; }
    public BinNode right() { return null; }
    public BinNode setRight(BinNode p) { return null; }
    public boolean isLeaf() { return true; }
} // class LeafNode

class IntlNode implements BinNode { // Internal node subclass
    private BinNode left;              // Left child
    private BinNode right;             // Right child
    private Character opx;              // Operator value

    public IntlNode(Character op, BinNode l, BinNode r)
    { opx = op; left = l; right = r; } // Constructor
    public Object element() { return opx; }
    public Object setElement(Object v) { return opx = (Character)v; }
    public BinNode left() { return left; }
    public BinNode setLeft(BinNode p) { return left = p; }
    public BinNode right() { return right; }
    public BinNode setRight(BinNode p) { return right = p; }
    public boolean isLeaf() { return false; }
} // class IntlNode

// Traverse a binary tree implemented with variable
// BinNode implementation
static void traverse(BinNode rt) { // Preorder traversal
    if (rt == null) return;        // Nothing to visit
    if (rt.isLeaf())                // Process leaf node
        VisitLeafNode(rt.element());
    else {                          // Process internal node
        VisitInternalNode(rt.element());
        traverse(rt.left());
        traverse(rt.right());
    }
}

```

图 5.9 使用 Java 的类继承,对 BinNode 接口实现不同的分支结点与叶结点

两个派生类 `LeafNode` 和 `IntlNode` 各自包含了对 `isLeaf` 的实现。`IntlNode` 类通过基类指针(即 `BinNode` 的指针类型)指向其子结点。函数 `traverse` 说明了这些定义的用法。当 `traverse` 调用“`rt.isLeaf()`”时,Java 运行环境判断当前结点 `rt` 所属的子类,并调用这个子类的 `isLeaf` 函数。而后,`isLeaf` 函数给出结点的真正结点类型。可以类似地调用这两个派生子类的其他成员函数,因为运行环境知道一个给定对象的具体类型是什么。

### 5.3.2 空间开销

本小节介绍二叉树实现的结构性开销的计算。回顾一下,结构性开销是指为了实现数据结构所花费的空间——换句话说,即那些不用来存储数据的空间。结构性开销的大小取决于许多因素,包括哪些结点存储数据(全部结点或只是叶结点),是否有父指针,以及是否为满二叉树等。

在简单的基于指针的二叉树中(如图 5.6 所示),每个结点都有两个指针指向其子结点(即使子结点为空)。定理 5.2 说明大约一半指针被浪费在存储 `null` 值上,目的仅仅是为了描述树的结构。

如果只是叶结点存储数据,那么结构性开销在全部开销中所占的比例就取决于二叉树是否“满”。如果二叉树不满,有可能在一串分支结点之后仅有一个叶结点。在非满二叉树中,结构性开销将占有很大比例。二叉树越接近满的程度,结构性开销所占的比例就越低,而满二叉树则达到最低。在这种情况下,大约一半结点是分支结点,都属于结构性开销。

如果简单地把每个结点实现为存储两个子结点指针和一个数据,那么有  $n$  个结点的二叉树一共需要空间  $n(2p + d)$ ,这里  $p$  代表一个指针所需要的空间, $d$  代表数据所需要的空间。对于整个二叉树来说,结构性开销为  $2pn$ 。这样,结构性开销所占比例就为  $2p/(2p + d)$ 。表达式的值取决于指针与数据区的相对大小。如果简单地假设  $p = d$ ,那么满二叉树的空间将有  $2/3$  被结构性开销占据。在满二叉树中,去掉叶结点中的指针会省下大量空间。因为分支结点与树叶大约各占一半,而现在只有分支结点包含结构性开销,这样,结构性开销所占比例将接近于:

$$\frac{\frac{n}{2}(2p)}{\frac{n}{2}(2p) + dn} = \frac{p}{p + d}$$

如果  $p = d$ ,那就意味着结构性开销将达到全部空间的大约一半。但是,如果只有叶结点存储有用的信息,结构性开销所占的比例大约为四分之三。因为有一半的“数据”区没有使用。

对于只在叶结点中存储数据的满二叉树来说,较好的实现方法是分支结点只存储两个指针,没有数据区,而叶结点则只包含一个数据区。这种方法需要  $2pn + d(n + 1)$  个空间单元。如果  $p = d$ ,结构性开销约为  $2p/(2p + d) = 2/3$ 。

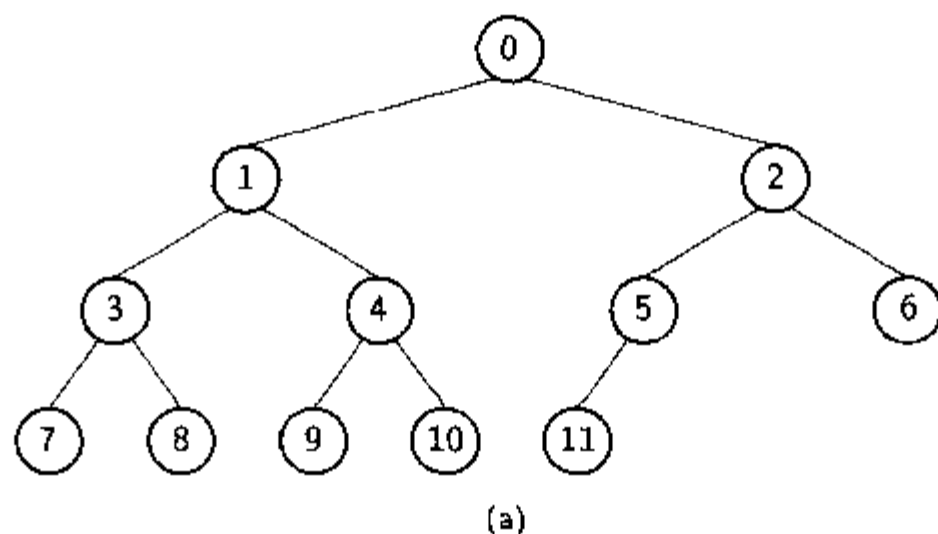
这种分析有一个严重的缺陷。如果分别实现分支结点与叶结点,那么就需要有一种方法来区别两种结点类型。假如使用 Java 的子类来分别实现两种结点类型,运行环境会为每个对象存储信息,以便调用函数 `isLeaf` 时能够判断其所属的子类。这就意味着每个结点都增加了额外的空间开销。而实际上只需要一位(的空间)就可以区别两种情形。有些方法在结点值区域中找到一位,用于表示所属的结点类型。还有一种方法则使用结点指针的一个空闲位来存储结点所属的类型。通常这是能够办到的,前提是程序语言允许对指针变量进行算术运算,而且编译器要求结构与类必须以一个字(word)的边界开始,使得指针值的最后一位总是为零。

于是,这一位就可以用来存储结点类型的标志。并且当指针被重新赋值时,这一位重新设置为零。另一种方法是:如果叶数据区比一个指针小,就将指向叶结点的指针换成该叶结点的值。如果空间非常有限,类似这样的技术往往是能否成功的关键。在其他情况下,应该避免使用这种“缩位”(bit packing)技巧,因为“可读性第一,效率其次”才是我们所追求的。

### 5.3.3 使用数组实现完全二叉树

上一小节指出二叉树的实现,有很大比例的空间被结构性开销所占用,而不是用于存储有用的数据。本小节介绍一种简单、紧凑的实现方法。回顾一下,完全二叉树的每一层(除了最底层)都是满的,并且最底层的结点从左到右填充。因此, $n$ 个结点的二叉树只可能有一种形状。你也许会认为完全二叉树极少出现,因而没有必要专门设计一种方法来实现它。事实上,完全二叉树有实际的用途,最重要的就是5.6节讨论的堆数据结构。堆经常被用来实现优先队列(5.6节)和外排序算法(9.7节)。

假设在完全二叉树中,如图5.10(a)所示,逐层而下、从左到右,结点的位置由其序号完全确定。数组可以有效地存储二叉树的数据,把每一个数据存放在其结点对应序号的位置上。图5.10(b)的图表列出了每个结点的子结点、父结点及兄弟结点。从图5.10(b)中可以看出每个结点的亲属在数组中的位置关系,可以推导出计算每个结点亲属下标的简单公式。不需要任何指向左右子结点的指针。这意味着如果对有 $n$ 个结点的二叉树使用大小为 $n$ 的数组来实现,就不存在结构性开销。



结点/索引	0	1	2	3	4	5	6	7	8	9	10	11
父结点	-	0	0	1	1	2	2	3	3	4	4	5
左子结点	1	3	5	7	9	11	-	-	-	-	-	-
右子结点	2	4	6	8	10	-	-	-	-	-	-	-
左兄弟结点	-	-	1	-	3	-	5	-	7	-	9	-
右兄弟结点	-	2	-	4	-	6	-	8	-	10	-	-

(b)

图 5.10 完全二叉树及其数组实现

(a)有 12 个结点的完全二叉树。(b)数组标明了每个结点之间的相互关系,符号“-”表示不存在相互关系

下面就是计算各亲属结点下标的公式。公式中 $r$ 表示结点的下标, $n$ 表示二叉树结点的总数。

- $\text{Parent}(r) = (r - 1) / 2$ , 当  $0 < r < n$  时

- $\text{Leftchild}(r) = 2r + 1$ , 当  $2r + 1 < n$  时
- $\text{Rightchild}(r) = 2r + 2$ , 当  $2r + 2 < n$  时
- $\text{Leftsibling}(r) = r - 1$ , 当  $r$  为偶数并且  $0 < r < n$  时
- $\text{Rightsibling}(r) = r + 1$ , 当  $r$  为奇数并且  $r + 1 < n$  时

## 5.4 Huffman 编码树

3.8 节的时间/空间权衡原则指出,通常可以通过牺牲运行时间代价来换取空间开销的降低。有许多应用技巧都是很好的权衡方法。一个典型的例子就是把文件存储到磁盘中。如果这个文件不是经常用到,使用者就可以压缩它以节省空间,以后使用时再解压缩,这会花费一些时间,但是只需要做一次。

进行程序设计时,我们经常通过给每一个元素标记一个单独的代码来表示某一组元素,例如标准的 ASCII 码把每个字符分别用一个 8 位的二进制数来表示。这种方法使用最少的位表示了所有的字符。它用  $\log 128$  即 7 位提供了 128 个不同的代码来表示 ASCII 码表中的 128 种字符<sup>①</sup>。

假设所有代码都等长,则表示  $n$  个不同的代码需要  $\log n$  位,称为固定长度编码方法(a fixed-length coding scheme)。ASCII 码就是一种固定长度编码。如果每个字符的使用频率都相等,固定长度编码是空间效率最高的方法。但是,你可能注意到,并非每个字符的使用频率都是一样的。

图 5.11 给出了在典型的英语文献中字母表中各个字母出现的相对频率。通过这个表,可以看出字母“E”的出现频率为“Z”的 60 倍。在 ASCII 码中,单词“DEED”和“FUZZ”需要相同的空间(4 个字节)。经常出现的单词例如“DEED”似乎应该比“FUZZ”这类相对较少出现的单词使用更少的空间存储以使得总的存储空间变小。

字母	频率	字母	频率
A	77	N	67
B	17	O	67
C	32	P	20
D	42	Q	5
E	120	R	59
F	24	S	67
G	17	T	85
H	50	U	37
I	76	V	12
J	4	W	22
K	7	X	4
L	42	Y	22
M	24	Z	2

图 5.11 26 个字母在英语文献中出现的相对频率。摘自[Wel88],见“参考文献”。

“频率(Frequency)”表示每 1000 个字母中字母出现的次数,不区分大小写

<sup>①</sup> ASCII 标准是 8 位,而不是 7 位,尽管它只表示了 128 个字符。第 8 位用以校验传输中的错误,或支持扩展 ASCII 码,表示另外的 128 个字符。

如果某些字符比其他字符更常用,是否有可能利用这一点来缩短代码呢?代价也许是其他字符需要使用更长的代码来表示。但是如果这些字符很少出现,这是值得的。这个概念是今天广泛使用的文件压缩技术的核心。下一小节介绍一种不等长编码(variable-length code),称作 Huffman 编码。虽然 Huffman 编码的最简形式在文件压缩技术中并不常用(有更好的方法),但是它给出了这种编码方法的思想。

#### 5.4.1 建立 Huffman 编码树

Huffman 编码将为字母分配代码。代码长度取决于对应字母的相对使用频率或者“权”,因此它是一种不等长编码。如果预计的字母出现频率与实际资料显示的情况相符,那么所得到的代码长度将明显小于使用固定长度编码所获得的代码。每个字母的 Huffman 编码是从被称为 Huffman 编码树(Huffman coding tree)或者 Huffman 树(Huffman tree)的满二叉树中得到的。Huffman 树的每个叶结点对应于一个字母。其目的在于按照最小外部路径权重(minimum external path weight)建立一棵树。一个叶结点的加权路径长度(weighted path length)定义为权乘以深度。这样建立的二叉树使给定叶结点的带权路径长度达到最小。“权”大的叶结点深度小,因而它相对总路径长度的花费最小。因此,其他的叶结点如果“权”小,就被推到树的较深处。

建立 Huffman 树的过程很简单。首先,按照“权”(例如频率)的大小顺序将字母排为一列。接着,拿走前两个字母(“权”最小的两个字母),再把它们标记为 Huffman 树的叶结点,把这两个叶结点标记为一个分支结点的两个子结点,而这个结点的权即为两个叶结点的权之和。把所得的“权”放回序列中适当的位置,使得“权”的顺序保持为升序。重复上述步骤,直至序列中只剩下一个元素,则 Huffman 树建立完毕。

图 5.13 说明了使用图 5.12 的 8 个字母建立部分 Huffman 树的过程。其中字母 D 与 L 权相同,可以任意排列。8 个字母根据出现频率排列如下:

Z	K	F	C	U	D	L	E
2	7	24	32	37	42	42	120

字母	C	D	E	F	K	L	U	Z
频率	32	42	120	24	7	42	37	2

图 5.12 8 个字母的相对使用频率

这些各自分离的树最终将组成一棵 Huffman 树。头两个字母是 Z 和 K,它们便首先被挑选出来组成树。二者成为一个权为 9 的内部结点的子结点。因而,一个根结点的权为 9 的树被放回序列中,占据第一位。接着把 9 与 24 取出(对应上一步建立的包含两个叶结点的子树,和表示存储字母 F 的子树)并将它们合并。所得到的根结点的权为 33,因此这个树被放在权为 32(表示字母 C)和权为 37(表示字母 U)的树之间。这个过程一直持续到建立一个根结点权为 306 的树为止。

图 5.14 给出了两个类的 Java 说明。第一个类 LettFreq 存储了字母/频率对。第二个类是 Huffman 树本身。Huffman 树的每一个结点存储一个指向 LettFreq 对象的指针。图 5.15



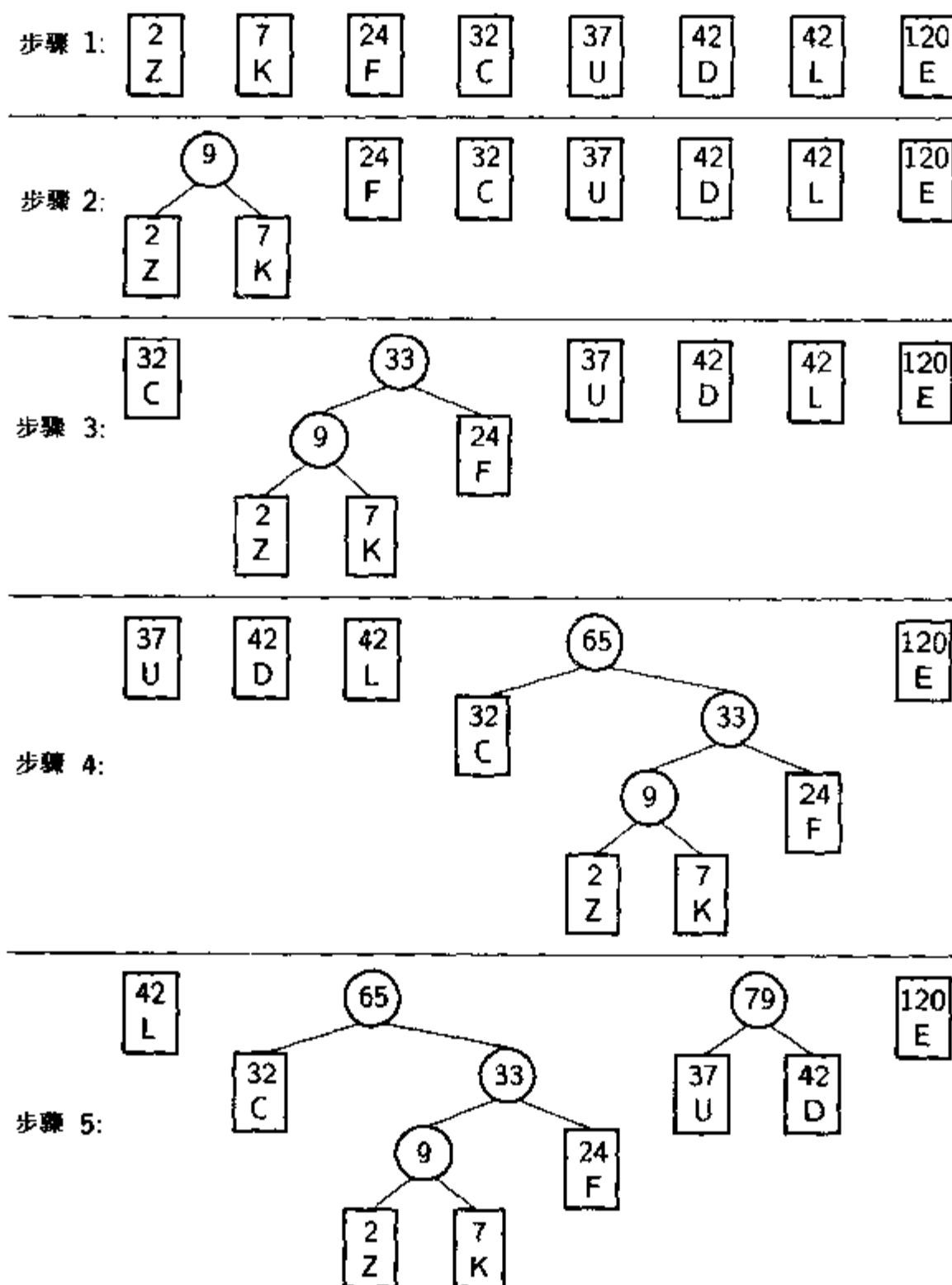


图 5.13 建立 Huffman 树样例的前 5 个步骤

给出了建立 Huffman 树的 Java 代码。

```

class LettFreq { // A letter/frequency pair
    private char lett; // The letter
    private int freq; // Frequency for the letter

    public LettFreq(int f, char l) { freq = f; lett = l; }
    public LettFreq(int f) { freq = f; }
    public int weight() { return freq; } // Return the weight
    public char letter() { return lett; } // Return the letter
} // class LettFreq

class HuffTree { // A Huffman coding tree
    private BinNode root; // Root of the Huffman coding tree

```

```

public HuffTree(LettFreq val)
{ root = new BinNodePtr(val); }
public HuffTree(LettFreq val, HuffTree l, HuffTree r)
{ root = new BinNodePtr(val, l.root(), r.root()); }

public BinNode root() { return root; }
public int weight()    // Weight of tree is weight of root node
{ return ((LettFreq)root.element()).weight(); }
} // class HuffTree

```

图 5.14 Huffman 树类的说明

```

// Build a Huffman tree from list hufflist
static HuffTree buildTree(List hufflist) {
    HuffTree temp1, temp2, temp3;
    LettFreq tempnode;

    for(hufflist.setPos(1); hufflist.isInList(); hufflist.setPos(1)) {
        // While at least two items left
        hufflist.setFirst();
        temp1 = (HuffTree)hufflist.remove();
        temp2 = (HuffTree)hufflist.remove();
        tempnode = new LettFreq(temp1.weight() + temp2.weight());
        temp3 = new HuffTree(tempnode, temp1, temp2);

        // return to the list in sorted order
        for (hufflist.setFirst(); hufflist.isInList(); hufflist.next())
            if (temp3.weight() <=
                ((HuffTree)(hufflist.currValue())).weight())
                { hufflist.insert(temp3); break; } // Put in list
        if (! hufflist.isInList())                // It is heaviest value
            hufflist.append(temp3);
    }
    hufflist.setFirst();    // Tree now only element on list
    return (HuffTree)hufflist.remove(); // Return the tree
}

```

图 5.15 建构 Huffman 树的实现。函数 build-tree 传入的参数 hufflist 是 Huffman 树部分树的表,初始时如图 5.13 中步骤 1 显示的那样,表元素都只是叶结点。函数 build-tree 的主体主要是一个 for 循环语句。每循环一次,最前面的两个部分树就被取出存放到变量 temp1 和 temp2 中。产生一个新的根结点(temp3),并将这两个部分树作为其子结点。最后,temp3 被放回到 hufflist 中

Huffman 树的建立方法是贪心算法(greedy algorithm)的一个例子。每一步,“权”最小的两棵子树被结合为一棵新的子树。这使得算法简单,但是是否能够得到所要的结果呢?本小节证明 Huffman 树确实给出了给定字母的最佳排列。

证明需要用到下面的引理。

**引理 5.1** 一棵至少包含两个结点的 Huffman 树,会把字母使用频率最小的两个字母作为兄弟结点存储,其深度不比树中其他任何叶结点小。

**证明:**记使用频率最低的两个字母为  $l_1$  和  $l_2$ 。由于 buildTree 在构造过程的第一步就选择了它们,所以它们一定是兄弟。假设  $l_1$  和  $l_2$  并不是二叉树中最深的结点。这样,Huffman 树或者像图 5.16 的样子,或者与之相似。当这种情况发生时, $l_1$  和  $l_2$  的父结点  $V$  一定会有比结点  $X$  更大的“权”。否则,函数 buildTree 会选择结点  $V$  而不是  $X$  作为结点  $U$  的子结点。然而,由于  $l_1$  和  $l_2$  是使用频率最小的字母,这种情况不可能发生。

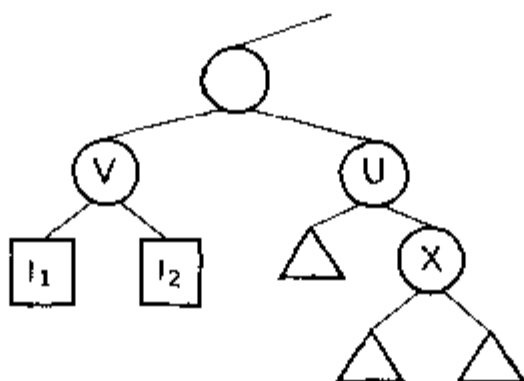


图 5.16 一棵不可能实现的 Huffman 树,权最小的两个结点  $l_1$  和  $l_2$  不是最深的结点。三角形表示子树

**定理 5.3** 对于给定的一组字母,Huffman 树实现了“最小外部路径权重”(the minimum external path weight)。

**证明:**对字母个数  $n$  作归纳进行证明。

- **初始情况:**令  $n = 2$ , Huffman 树一定有最小外部路径权重,因为只可能有两种树,并且两个叶结点的加权路径长度相等。

- **归纳假设:**假设有  $n - 1$  个叶结点的 Huffman 树有最小外部路径权重。

- **归纳步骤:**设一棵 Huffman 树  $T$  有  $n$  个叶结点,  $n \leq 2$ ,并假设  $w_1 \leq w_2 \leq \dots \leq w_n$ ,这里  $w_1$  到  $w_n$  代表字母的“权”。记  $V$  是频率为  $w_1$  和  $w_2$  的两个字母的父结点。根据引理,由于它们已经是树  $T$  中最深的结点,不能用深度更深而“权”也较之更大的结点替换它们以减小外部路径长度。记 Huffman 树  $T'$  与  $T$  完全相同,除了把结点  $V$  换为一个叶结点  $V'$ ,其权等于  $w_1 + w_2$ 。根据归纳假设, $T'$  具有最小的外部路径长度。把两个子结点  $w_1$  和  $w_2$  归还给  $V'$ ,还原为  $T$ ,则  $T$  也应该有最小的外部路径长度。

因此,根据归纳原理,定理成立。

#### 5.4.2 Huffman 编码及其用法

一旦 Huffman 树构造完成,很容易就能把各个字母用代码标记上。从根结点开始,分别把“0”或“1”标于树的每条边上。“0”对应于连接左子结点的那条边,“1”则对应于连接右子结点的边。图 5.17 表示了这个过程。字母的 Huffman 编码就是从根结点到对应于该字母叶结点路径的二进制代码。因此,由于从根结点到对应于 E 的叶结点的路径只是左边的一个分支,字母 E 对应代码“0”。由于到对应 K 的结点的路径为 4 条右分支,接着一个左分支,最后

又是一个右分支,所以 K 的代码为“111101”。图 5.18 列出了 8 个字母的对应代码。

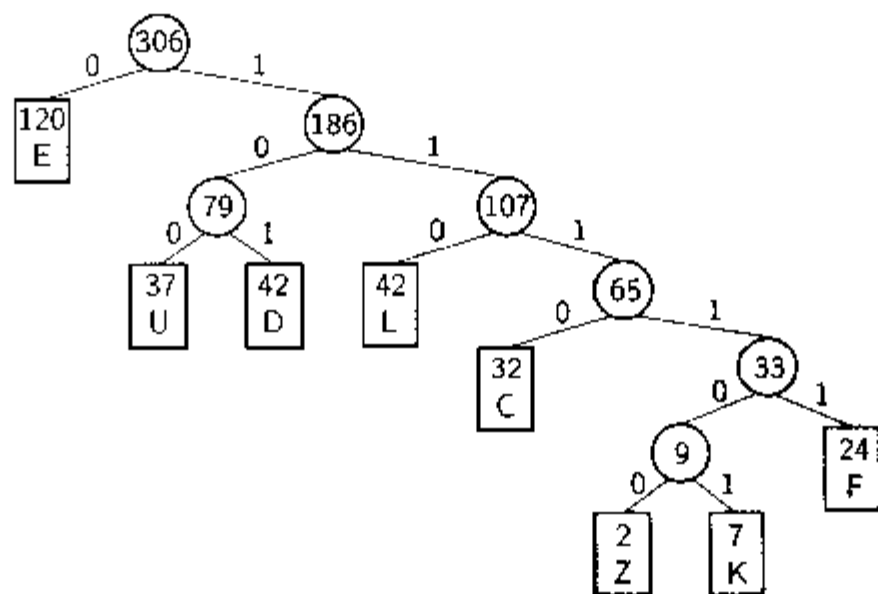


图 5.17 根据图 5.12 的字母建立的 Huffman 树

字母	频率	代码	位数
C	32	1110	4
D	42	101	3
E	120	0	1
F	24	11111	5
K	7	111101	6
L	42	110	3
U	37	100	3
Z	2	111100	6

图 5.18 对应图 5.12 字母的 Huffman 码

给出了字母各自的代码,将文本信息用这些代码来表示就变得很容易了。只需要简单地把字母用对应的二进制代码替换就行了。这可以通过查表来完成。用我们举例的 Huffman 树生成的代码,单词“DEED”可以用数字串“10100101”表示,而单词“FUZZ”可用数字串“11111100111100111100”来表示。

对信息代码反编码的过程为:从左到右逐位判别代码串,直至确定一个字母。这可以对 Huffman 树用其生成代码过程的逆过程来实现。从树的根结点开始对数字串进行反编码。根据每一位的值是“0”或者“1”确定选择左分支还是右分支——直至到达一个叶结点。这个叶结点包含的字母就是文本信息的第一个字母。然后从下一位代码开始,自根结点出发,开始下一个字母的翻译。

对数字串“1011001110111101”反编码,从根结点开始,由于第一位是“1”,所以选择右分支。下一位是“0”,所以选左分支。接着选择另一个右分支(因为第 3 位是“1”),到达叶结点对应的字母 D。这样,被编码单词的第一个字母就是 D。接着从根结点出发,从第 4 位开始,它是“1”,选择右分支,接着是两个左分支(因为接下来的两位是“0”),就到达了对应字母 U 的叶结点。于是第二个字母为 U。类似地,完成全部反编码可以发现最后两个字母是 C 和 K,它们组成单词“DUCK”。

如果一组代码中的任何一个代码都不是另一个代码的前缀,称这组代码符合前缀特性(prefix property)。这种前缀特性保证了代码串被反编码时不会有多种可能。换句话说,在反编码的过程中,一旦到达某个代码的最后一位,我们就能够判断出它所代表的字母。由于任何一个代码的前缀对应一个分支结点,而每个代码都对应一个字母, Huffman 代码当然符合前缀特性。例如

F 的代码为“11111”。在图 5.17 的 Huffman 树中选择 5 个右分支,就得到包含 F 的叶结点。我们可以肯定没有一个字母代码为“111”,因为它对应树的一个分支结点,而在建立树的过程中,只把字母存放在叶结点中。

Huffman 编码的效率如何呢?理论上,一旦知道实际的频率,它是一种优化编码方法。但是在实际应用中,字母的使用频率取决于具体文本信息的情况。例如,尽管在字母表的各个字母中 E 是英语文献里最常见的,但是作为单词的第一个字母,字母 T 更常见。这就是为什么商业压缩工具都不采用 Huffman 编码作为其基本编码方式的原因。

另一个影响 Huffman 编码效率的因素是字母的相对使用频率。与固定长度编码相比,有些频率模式并不节省空间;有些则能够产生极大的压缩比。一般情况下,如果字母频率的变化范围很大,则 Huffman 编码是很有效的。如图 5.18 显示的特殊情况,如果所编码文本的实际字母频率与预期的相符,我们就能确定 Huffman 编码所节省的空间。

由于图 5.18 中频率的总和为 306,并且 E 的频率为 120,预计在一段包含 306 个字母的文章中 E 出现 120 次。实际的文章不一定符合这个假设。字母 D、L 和 U 的代码长度为 3,在 306 个字母中预计可能出现 121 次。字母 C 的代码长度为 4,预计在 306 个字母中出现 32 次。字母 F 的代码长度为 5,预计在 306 个字母中出现 24 次。最后,字母 K 与 Z 的代码长度均为 6,预计只出现 9 次。预计平均每个字母的代码长度等于每个代码的长度( $c_i$ )乘以其出现的概率( $p_i$ ),即:

$$c_1 p_1 + c_2 p_2 + \cdots + c_n p_n$$

也可以记为:

$$\frac{c_1 f_1 + c_2 f_2 + \cdots + c_n f_n}{f_T}$$

这里  $f_i$  为第  $i$  个字母的相对频率,而  $f_T$  为所有字母的总频率。在这组频率下,每个字母的期望代码长度为:

$$[(1 \times 120) + (3 \times 121) + (4 \times 32) + (5 \times 24) + (6 \times 9)] / 306 = 785 / 306 \approx 2.57$$

对于这 8 个字母使用固定长度编码,每个字母需要  $\log 8 = 3$  位,而 Huffman 编码只需要 2.57 位。因此对于这组字母,Huffman 编码预计可以节省大约 12% 的空间。

对所有 ASCII 字符的 Huffman 编码效果要比这个好得多。图 5.18 的字母并不典型,因为相对于较少使用的字母来说,普通字母太多了。26 个字母的 Huffman 编码导致每个字母的期望代码长度为 4.29 位,而等价的固定长度编码将需要 5 位。这样对于固定长度编码来说多少有些不公平,因为实际上 5 位可以提供 32 个代码,但是字母只有 26 个。更进一步说,如果我们设 ASCII 编码为每个字符 8 位,Huffman 编码对于典型的文本文件将比 ASCII 编码节省大约 40% 的空间。对于二进制代码文件(例如可执行文件),Huffman 编码会有极为不同的频率分布组合,因而也会有不同的压缩比率。大多数的商业压缩程序都是采用 2 到 3 种编码方式,以应付各种类型的文件。

在前面的例子中,“DEED”编码为 8 位,比固定长度编码所需要的 12 位节省 33%。当然,“FUZZ”需要 20 位,比固定长度编码使用的空间多,问题在于“FUZZ”由那些预计不常使用的字母组成。如果文本的字母频率与预计不符,当然编码的长度也不会是预计的那样。

## 5.5 二叉检索树

4.5 节项目设计的 4.6 题要求使用线性表实现一个简单的城市数据库。数据库中的每一

条记录包含城市的名字以及它的  $xy$  坐标。如果记录的存储没有任何特别的顺序,那么插入一条新记录将会很快,只需要将其放在末端。但是,在一个没有顺序的链表中根据城市的名称查找一条特殊记录的平均检索时间为  $\Theta(n)$ 。对于一个大型数据库,这很可能太慢了。可供选择的方法之一就是要把记录按照城市名字的字典顺序来存储。如果线性表使用链表来实现,顺序存储记录并不会提高检索速度。如果线性表使用数组实现,那么使用第 3.5 节的二分法检索只需要时间  $\Theta(\log n)$ 。但是插入则需要时间  $\Theta(n)$ ,因为当在已排序的表中找到新记录恰当的位置时,需要移动许多记录以便为新的记录腾出地方。有没有哪一种组织记录的方法使得记录的插入与检索都能很快地完成呢?

本节介绍能很好地解决这个问题的二叉检索树(Binary Search Tree,即 BST)。

在讨论 BST 之前,我们考虑一下要查找的到底是什么。上例中,一条城市记录至少包含 3 个信息: $x$  坐标、 $y$  坐标和城市名。因此城市名成为检索关键码(search key)。在以后的篇幅中,我们将讨论用于检索特殊记录的数据结构,以及用于组织一组记录的不同排序算法。假定我们将讨论的数据记录是从一个称为 Elem 的接口派生的。Elem 接口定义如下:

```
interface Elem {                               // Interface for generic element type
    public abstract int key();                 // Key used for search and ordering
} // interface Elem
```

接口 Elem 简单地保证了记录有一个返回 int 类型值的 key 函数,它可以被排序或检索算法用来比较元素值。当然这是一种过于简单的描述,因为有些情况下关键码不会只是一个整数(例如用城市名来组织城市记录)。但是,这种简化并不影响我们讨论检索和排序的原理,实际应用中可以修改具体的实现而满足复杂关键码的情形。

BST 是满足下面所给条件的二叉树。

**定义 5.4** 二叉检索树的任何一个结点,设其值为  $K$ ,则该结点左子树中任意一个结点的值都小于  $K$ 。该结点右子树中任意一个结点的值都大于或等于  $K$ 。

图 5.19 给出了对应一组数值的两棵二叉检索树(BST)。二叉检索树的特点就是,如果按照中序周游将各个结点打印出来,就会得到由小到大的排列。

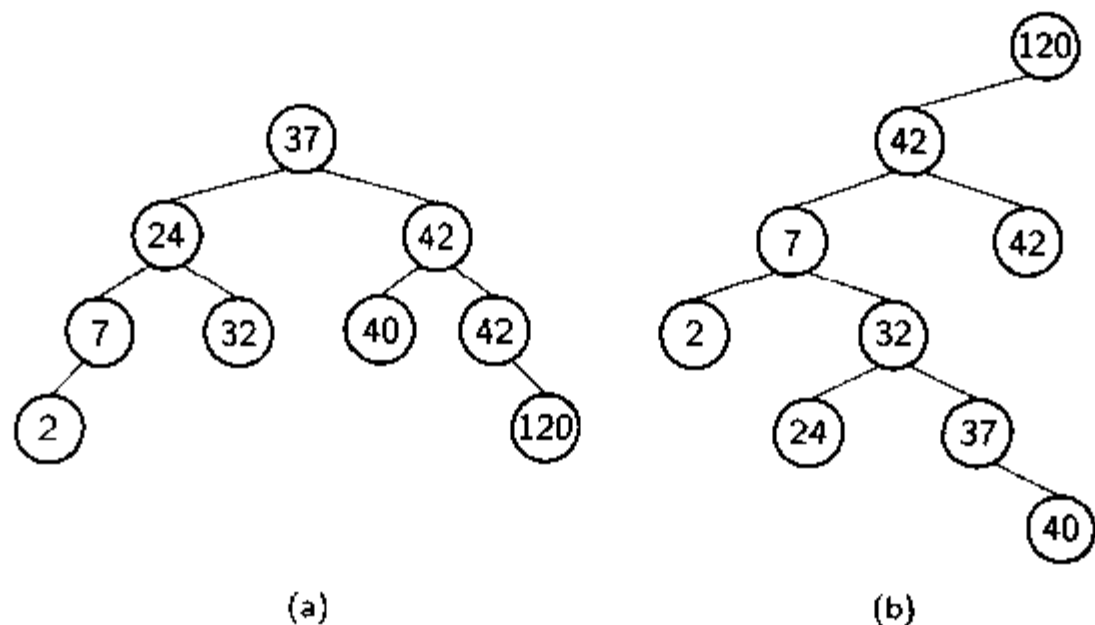


图 5.19 一组给定数值的两棵二叉检索树。按照(37,24,42,7,2,40,42,32,120)的顺序将各个结点插入,得到二叉树(a)。按照(120,42,42,7,2,32,37,24,40)的顺序将各个结点插入,则得到二叉树(b)

图 5.20 给出了二叉检索树的部分描述。包括所有公共函数,如 BST 的构造函数、重新初始化 BST 的函数 clear、插入和删除记录的函数、查找给定关键码值的函数以及按升序打印关键码值的函数。

```
class BST { // Binary Search Tree implementation
    private BinNode root; // The root of the tree

    public BST() { root = null; } // Initialize root to null
    public void clear() { root = null; } // Throw the nodes away
    public void insert(Elem val) { root = inserthelp(root, val); }
    public void remove(int key) { root = removehelp(root, key); }
    public Elem find(int key) { return findhelp(root, key); }
    public boolean isEmpty() { return root == null; }

    public void print() { // Print out the BST
        if (root == null)
            System.out.println("The BST is empty.");
        else {
            printhelp(root, 0);
            System.out.println();
        }
    }
} // class BST
```

图 5.20 二叉检索树的类说明

从根结点开始,在 BST 中检索值  $K$ 。如果根结点存储的值为  $K$ ,则检索结束。如果不是,则必须检索树的更深层。BST 的效率就在于只需要检索两个子树之一。如果  $K$  小于根结点的值,则只需检索左子树,如果  $K$  大于根结点的值,就只检索右子树。这个过程一直持续到  $K$  被找到或者遇到了一个叶结点为止。如果遇到叶结点仍没有发现  $K$ ,那么  $K$  就不在这个 BST 中。

例如在图 5.19(a)的二叉树中检索值 32。由于 32 小于根结点的值 37,检索过程进入左子树。由于 32 比 24 大,我们检索 24 的右子树。此时找到了包含值 32 的结点。如果检索值为 35,检索的路径是相同的,直到找到包含 32 的结点。由于这个结点没有子结点,我们可以判断出 35 不在这个 BST 中。

注意图 5.20 中,find 只是简单地调用私有成员函数 findhelp。函数 find 的显式参数是检索的值,其实它的隐含参数是该值所在的 BST。检索时将子树的根结点及检索的值作为参数,使用递归函数可以很容易实现。成员 findhelp 正是满足这种要求的递归子程序,其实现如下。

```
private Elem findhelp(BinNode rt, int key) {
    if (rt == null) return null;
    Elem it = (Elem)rt.element();
    if (it.key() > key) return findhelp(rt.left(), key);
    else if (it.key() == key) return it;
    else return findhelp(rt.right(), key);
}
```

```

}

```

一旦找到所检索的记录,它将沿着递归调用链将值传回给 findhelp。

要插入一个值  $K$ ,首先必须找出它应该放在树的什么地方。这就把我们带到一个叶结点或者一个分支结点,它在待插入的方向上没有子结点<sup>①</sup>。记这个结点为  $R'$ 。接着,把一个包含  $K$  的结点作为  $R'$  的子结点加上去。图 5.21 分析了这个操作。值 35 作为包含值 32 结点的右子结点被加上去。下面是 inserthelp 的实现。

```

private BinNode inserthelp(BinNode rt, Elem val) {
    if (rt == null) return new BinNodePtr(val);
    Elem it = (Elem)rt.element();
    if (it.key() > val.key())
        rt.setLeft(inserthelp(rt.left(), val));
    else
        rt.setRight(inserthelp(rt.right(), val));
    return rt;
}

```

图 5.21 BST 中插入值为 35 的结点。值为 32 的结点是包含值为 35 的新结点的父结点

要注意 inserthelp 实现中的一个重要细节。inserthelp 返回一个 BinNode 引用。从逻辑的角度来说,返回的是与原先的子树一样的子树,只是新子树中包含新插入的结点。从根结点到被插入结点的父结点的路径上的各个结点都被赋予了相应的子结点值。其实除该路径的最后一个结点外,其他结点都不会改变子结点指针值。从这一点来说,许多赋值都是不必要的。但是由于这种方法的简捷性,这些额外的赋值是值得的。

BST 的形状取决于各个元素被插入二叉树的先后顺序。一个新的元素作为一个新的叶结点被添加到二叉树中,有可能增加树的深度。图 5.19 表示对于一组给定元素的两棵不同的 BST。一棵包含  $n$  个结点的 BST 有可能是一条  $n$  个结点的链,而树的高度为  $n$ 。例如,所有元素按照已排列好的顺序插入时,这种情况就会发生。通常情况下 BST 的高度越小越好。

从 BST 中删除一个结点需要一些技巧,但是如果分别考虑各种可能情况就不算太难。在着手解决挪走一般结点的过程之前,我们首先讨论如何挪走一个子树中值最小的结点。这个方法以后将被删除一般结点的函数所采用。为了挪走子树中值最小的结点,首先应该找到这个结点。这只需要沿着左边的链不断向下移,直到已经没有左边的链可以继续下移,这样就找到了值最小的结点,记为  $S$ 。挪走  $S$  只需要简单地把  $S$  的父结点中原来指向  $S$  的指针改变为指向  $S$  的右子结点。我们肯定  $S$  没有左子结点(因为如果  $S$  有左子结点,它就不是值最小的结点)。这样改变指针,删除  $S$  结点,二叉检索树的性质保持不变。完成上述查找最小值和删除最小值功能的函数分别为 getmin 和 deletemin,其代码如下:

```

private Elem getmin(BinNode rt) {
    if (rt.left() == null)

```

<sup>①</sup> 这里假设二叉树中没有一个结点的值与被插入结点的值相同。假如我们确实发现某个结点的值与被插入结点的值相同,有两种选择:如果该应用不允许结点有相同的值,就必须把这个插入作为错误处理;如果允许有相同的值,我们的习惯是将其插在右子树中,如成员 inserthelp 所作的那样。



```

        return (Elem)rt.element();
    else return getmin(rt.left());
}

private BinNode deletemin(BinNode rt) {
    if (rt.left() == null)
        return rt.right();
    else {
        rt.setLeft(deletemin(rt.left()));
        return rt;
    }
}

```

图 5.22 表示了 `deletemin` 的过程。从值为 10 的根结点开始, `deletemin` 沿着左边的链下移, 直到再没有左边的链, 此时到达值为 5 的结点。因为 `rt` 是一个引用方式的指针, 此时, 它是值为 10 的结点的成员 `left` 的一个别名。接着变量 `rt` 被改变为指向值为 5 的结点的右子结点, 当然也改变了值为 10 的结点的成员 `left` 的值。在图 5.22 中用一条带箭头的线来表示。

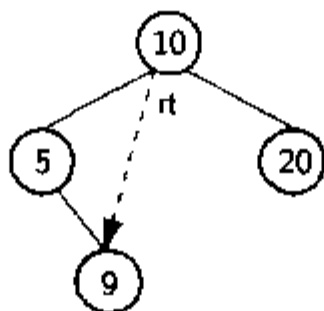


图 5.22 删除最小值示例。这棵二叉树中, 包含最小值 5 的结点是根结点的左子结点。所以, 根结点的左指针改为指向 5 的右子结点。变量 `rt` 是根结点左指针的别名

从 BST 中删除一个任意的结点, 首先必须找到 `R`, 接着将它从二叉树中删掉。因此, 删除操作的第一步就是检索 `R` 的存在。如果找到 `R`, 有多种可能。如果 `R` 没有子结点, 那么就将 `R` 的父结点指向它的指针改为 `null`。如果 `R` 有一个子结点, 那就将 `R` 的父结点指向它的指针改为指向 `R` 的子结点(与 `deletemin` 相似)。如果 `R` 有两个子结点, 问题就来了。一种较简单但代价高昂的解决办法就是让 `R` 的父结点指向 `R` 的一棵子树, 然后将剩下的另一棵子树的结点一个一个地重新插入。较好的解决办法是从某个子树中找出一个能代替 `R` 的值。这样就出现了一个问题: 什么值能够替代那个被删除的值呢?

由于必须在保证树的结构不发生巨大变化的同时又保持二叉检索树的性质, 因而并不是一个任意的值都可以用来替换。哪一个值最像被替换的值呢? 答案是那些大于(或等于)被替换值中的最小者, 或者那些小于被替换值中的最大者。如果用这两者之一去替换, 就能够保持二叉检索树的性质。举个例子, 假设我们希望从图 5.19(a) 的二叉检索树中删除值 37。我们删除右子树中值最小的结点(用 `getmin` 和 `deletemin` 操作), 而不删除根结点。这个值就可以用来代替根结点的值。在这个例子中, 由于 40 是右子树中的最小值, 我们先删除包含值 40 的结点, 接着用 40 代替 37 作为根结点的新值。图 5.23 表示了这个过程。

当二叉树中没有重复值出现时, 用左子树中的最大值还是右子树中的最小值来代替并没

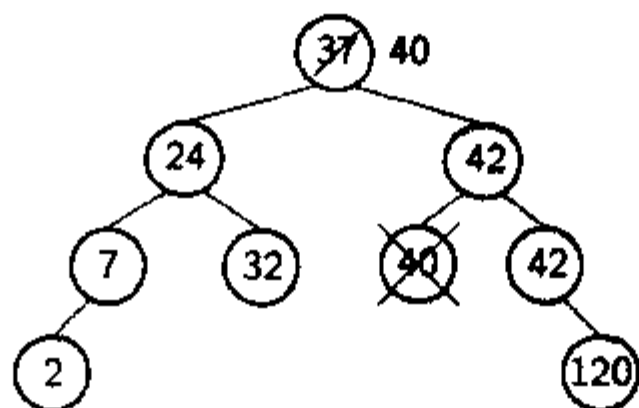


图 5.23 从二叉检索树中删除值 37 示例。包含这个值的结点有两个子结点,用结点右子树中的最小值 40 来代替值 37

有什么区别。如果有重复的值,那么就应该从右子树中选择替代者。究其原因,假设左子树中的最大值为  $G$ ,如果左子树的其他结点也有值  $G$ ,那么选择  $G$  作为根结点就会导致一棵二叉树的左子树中具有与子树根结点值  $G$  相同的结点。作为特例,假如图 5.19(b)中我们使用左子树中的最大值来替换 120,这种错误就出现了。从右子树中选择最小值就不会有类似的问题,因为具有相同值的结点只出现在右子树中,这样操作不会破坏二叉检索树的性质。

综上所述,如果想把一个有两个子结点的结点的值删去,只需要对其右子树调用函数 `getmin` 和 `deletemin`,并将 `getmin` 的返回值代替被删除的值。下面是 `removehelp` 的源代码。

```
private BinNode removehelp(BinNode rt, int key) {
    if (rt == null) return null;
    Elem it = (Elem)rt.element();
    if (key < it.key())
        rt.setLeft(removehelp(rt.left(), key));
    else if (key > it.key())
        rt.setRight(removehelp(rt.right(), key));
    else { // Found it
        if (rt.left() == null)
            rt = rt.right();
        else if (rt.right() == null)
            rt = rt.left();
        else { // Two children
            Elem temp = getmin(rt.right());
            rt.setElement(temp);
            rt.setRight(deletemin(rt.right()));
        }
    }
    return rt;
}
```

`findhelp` 和 `inserthelp` 的时间代价取决于结点被找到/插入的深度。`removehelp` 的时间代价取决于被删除结点的深度。如果结点有两个子结点,则取决于其右子树中包含最小值的结点的深度。这样,这几个操作中任意一个的最差情况都等于该树的深度。这就是为什么要尽量保持二叉检索树的平衡,也就是使其高度尽可能小。如果二叉树是平衡的,则有  $n$  个结点的二叉树的高度大约为  $\log n$ 。但是如果二叉树完全不平衡,例如成一个链表的形状,则其高

度可以达到  $n$ 。因而,平衡二叉树每次操作的平均时间代价为  $\Theta(\log n)$ ,而严重不平衡的 BST 在最差情况下平均每次操作时间代价为  $\Theta(n)$ 。假设我们按照一个一个地插入的方式创建一棵有  $n$  个结点的 BST,并且很幸运地遇到了每个结点的插入都使树保持平衡的情况(“随机”的顺序很可能较好地实现了这种目的),那么每次插入的平均时间代价为  $\Theta(\log n)$ ,共为  $\Theta(n \cdot \log n)$ 。但是,如果结点按照递增的顺序插入,就会得到一条高度为  $n$  的链,插入的时间代价为  $\sum_{i=1}^n i$ ,也即  $\Theta(n^2)$ 。

不论该树的形状如何周游二叉树的时间代价为  $\Theta(n)$ ,每个结点恰好被访问一次,每个指针也恰好被引用一次。下例函数 `printhelp` 完成二叉检索树的中序周游,把结点的值按照从小到大的升序打印出来。

```
private void printhelp(BinNode rt, int level) {
    if (rt == null) return;
    printhelp(rt.left(), level + 1);
    for (int i = 0; i < level; i++) // Indent based on level
        System.out.print(" ");
    System.out.println((Elem)rt.element()); // Print node value
    printhelp(rt.right(), level + 1);
}
```

虽然当二叉树平衡的时候,其实现很简单并且效率很高,但是它成为非平衡状态的可能性却很大。有许多组织二叉树使其形态良好的技巧。例如 13.2 节的伸展树(Splay Tree)。有一些检索树肯定是平衡的,例如 11.4 节的 2-3 树。

## 5.6 堆与优先队列

在现实生活和计算机应用中,存在许多需要从一群人、一系列任务或一些对象中找出“下一个最重要”目标的情况。例如,医院急诊室大夫常常是选择“下一个最重要”的病人,而不是选择那个最先来到的病人。在多任务操作系统的作业调度中,任何时刻都可能有多程序(通常称为“作业”,即 job)等待运行。操作系统往往选择具有最高优先级(priority)的那个作业。优先级是与作业有关的一个特殊值。操作系统选择作业运行时,应该从等待队列中选择具有最高优先级的那个作业。

一些按照重要性或优先级来组织的对象称为优先队列(priority queue)。在普通队列数据结构中查找具有最高优先级的元素的时间代价为  $\Theta(n)$ ,因而普通队列不能有效地实现优先队列。在有序或无序线性表中插入和删除的时间代价都是  $\Theta(n)$ 。可以考虑使用把记录按优先级组织的 BST(二叉检索树),平均情况下操作的总时间代价为  $\Theta(n \log n)$ 。但是 BST 可能会变得不平衡,这将导致 BST 性能变得很差。对于这种特殊的应用,我们希望发现一种新的数据结构以保证较高的操作效率。

本节介绍堆数据结构。堆由两条性质来定义。首先,它是一棵完全二叉树,所以往往如 5.3.3 小节中用数组表示完全二叉树那样用数组来实现。其次,堆中存储的数据是局部有序的。也就是说,结点存储的值与其子结点存储的值之间存在某种联系。有两种不同的堆,取决于有关联系的定义。

最大值堆(max-heap)的性质是任意一个结点的值都大于或者等于其任意一个子结点存储的值。由于根结点包含大于或等于其子结点的值,而其子结点又依次大于或等于各自子结点的值,所以根结点存储着该树的所有结点中的最大值。

另一种堆叫最小值堆(min-heap)。最小值堆的每一个结点存储的值都小于或等于其子结点存储的值。由于根结点包含小于或等于其子结点的值,而其子结点又依次小于或等于各自子结点的值,所以根结点存储了该树的所有结点的最小值。

无论最小值堆还是最大值堆,任何一个结点与其兄弟之间都没有必然的联系。例如,有可能根结点左子树的所有结点的值都比右子树中任意一个结点的值大。

两种堆都有其用处。例如,8.6节中堆排序使用了最大值堆,而9.7节的置换-选择排序算法中使用了最小值堆。本节的以下部分都使用最大值堆。

学生们经常把堆的逻辑表示与利用基于数组的完全二叉树的物理实现混淆。二者并非同义的,虽然堆的典型实现方法是使用数组,但是从逻辑的角度看,堆实际上是一种树形结构。

图5.24给出了最大值堆的实现。与BST一样,堆的元素类型为Elem,其函数key返回用于排序的关键码值。

```
public class MaxHeap {                // Max-heap implementation
private Elem[] Heap;                // Pointer to the heap array
private int size;                    // Maximum size of the heap
private int n;                      // Number of elements now in heap

public MaxHeap(Elem[] h, int num, int max) // Constructor
{ Heap = h; n = num; size = max; buildheap(); }

public int heapsize() // Return current size of the heap
{ return n; }

public boolean isLeaf(int pos) // True if pos is a leaf position
{ return (pos >= n/2) && (pos < n); }

// Return position for left child of pos
public int leftchild(int pos) {
    Assert.notFalse(pos < n/2, "Position has no left child");
    return 2 * pos + 1;
}

// Return position for right child of pos
public int rightchild(int pos) {
    Assert.notFalse(pos < (n-1)/2, "Position has no right child");
    return 2 * pos + 2;
}

public int parent(int pos) // Return position for parent
```

```

    Assert.notFalse(pos > 0, "Position has no parent");
    return (pos - 1) / 2;
}

public void buildheap()           // Heapify contents of Heap
{ for (int i = n / 2 - 1; i >= 0; i--) siftDown(i); }

private void siftDown(int pos) { // Put element in its correct place
    Assert.notFalse((pos >= 0) && (pos < n), "Illegal heap position");
    while (!isLeaf(pos)) {
        int j = leftChild(pos);
        if ((j < (n - 1)) && (Heap[j].key() < Heap[j + 1].key()))
            j++; // j is now index of child with greater value
        if (Heap[pos].key() >= Heap[j].key()) return; // Done
        DSUtil.swap(Heap, pos, j);
        pos = j; // Move down
    }
}

public void insert(Elem val) { // Insert value into heap
    Assert.notFalse(n < size, "Heap is full");
    int curr = n++;
    Heap[curr] = val;           // Start at end of heap
    // Now sift up until curr's parent's key > curr's key
    while ((curr != 0) && (Heap[curr].key() > Heap[parent(curr)].key())) {
        DSUtil.swap(Heap, curr, parent(curr));
        curr = parent(curr);
    }
}

public Elem removeMax() { // Remove maximum value
    Assert.notFalse(n > 0, "Removing from empty heap");
    DSUtil.swap(Heap, 0, --n); // Swap maximum with last value
    if (n != 0) // Not on last element
        siftDown(0); // Put new heap root val in correct place
    return Heap[n];
}

// Remove element at specified position
public Elem remove(int pos) {
    Assert.notFalse((pos > 0) && (pos < n), "Illegal heap position");
    DSUtil.swap(Heap, pos, --n); // Swap with last value
    if (n != 0) // Not on last element
        siftDown(pos); // Put new heap root val in correct place
}

```

```

    return Heap[n];
}
// class MaxHeap

```

图 5.24 最大值堆类的 Java 实现

类 `MaxHeap` 对基于数组的实现作了两项调整。首先,堆的结点根据其在堆中的逻辑位置指示,而不是使用指针指向。实际上,堆中的逻辑位置在数值上对应于其在数组中的物理位置。其次,指向所使用数组的指针作为构造函数的参数。这种方法为使用堆提供了最大的灵活性,因为所有的数据都能够直接装入数组中。这种方法的优点下面将作叙述。构造函数使用一个整型参数表示堆的初始大小(取决于初始时装入数组元素的数目),另一个整型参数表示堆允许的最大结点数目。成员函数 `heapsize` 返回堆当时的大小。如果所处的位置 `pos` 在堆 `H` 中是叶结点,则调用 `H.isLeaf(pos)` 将返回 `true`。成员 `leftchild`、`rightchild` 和 `parent` 分别返回传入参数的左子结点、右子结点和父结点的位置(实际为数组的下标)。

一种建立堆的方法就是把元素一个接一个地插入堆中。成员函数 `insert` 将新元素 `V` 插入堆中,你可能会认为堆的插入过程与 `BST` 的插入过程相似,从根开始往下。但是,这种方法不大可能有效,因为堆必须保持完全二叉树的形状。换句话说,如果调用 `insert` 之前堆占用数组的前  $n$  个位置,则调用之后占用前  $n+1$  个位置。为此, `insert` 首先将 `V` 置于堆的末尾位置  $n$ 。当然, `V` 此时很可能不在正确的位置上,将其与父结点相比较,以便它移到正确的位置。如果它小于或等于其父结点的值,则它已经处于正确的位置, `insert` 过程完成。如果 `V` 的值大于其父结点,则两个元素交换位置。 `V` 与其父结点的比较一直持续到 `V` 到达其正确位置为止。

插入的值可能要从树的底部移动到顶端,这是最长的距离。因此每次调用 `insert` 最差情况时间代价为  $\Theta(\log n)$ 。因而插入  $n$  个值的时间代价为  $\Theta(n \log n)$ 。

如果建立堆时全部  $n$  个值都已知,则可以更高效地建立堆。我们可以利用它们在一起这个特点来加快建立过程,而不必将值一个一个地插入堆中。图 5.25(a)给出了通过交换值建立堆的方法。记住,该图给出了输入数据作为完全二叉树的逻辑形式,但是应该清楚,实际上这些值是物理地存储于数组中的。所有的交换都在结点与其某一个子结点之间进行。通过交换过程形成堆。图 5.25(a)中右边树的数组如下:

7	4	6	1	2	3	5
---	---	---	---	---	---	---

图 5.25(b)给出了另一种交换方式,同样建立堆,但效率更高。从这个例子可以很明显地看出,对于给定的一组数,堆并不是惟一的,并且某些建堆方法比其他方法重新排列这些数据时所花费的交换次数相对少一些。那么如何选择最佳的重排列方法呢?有一种源于归纳法的较好算法。假设根的左、右子树都已经是堆,并且根的元素名为 `R`。图 5.26 表示了这种情况。在这种情况下,有两种可能:(1) `R` 的值大于或等于其两个子结点,此时堆结构已经完成;(2) `R` 的值小于某一个或全部两个子结点的值,此时 `R` 应与两个子结点中值较大的一个交换,结果得到一个堆,除非 `R` 仍然小于其新子结点的一个或全部两个。在这种情况下,我们只需要简单地继续这种将 `R`“拉下来”的过程,直至到达某一层使它大于它的子结点,或者它成了叶结点。这个过程用堆类的私有成员函数 `siftdown` 实现。图 5.27 演示了 `siftdown` 的操作。

这种方法假设子树已经是堆,说明完整的算法可以用按照某种顺序访问结点来实现,例如

在访问结点本身之前先访问其子结点。

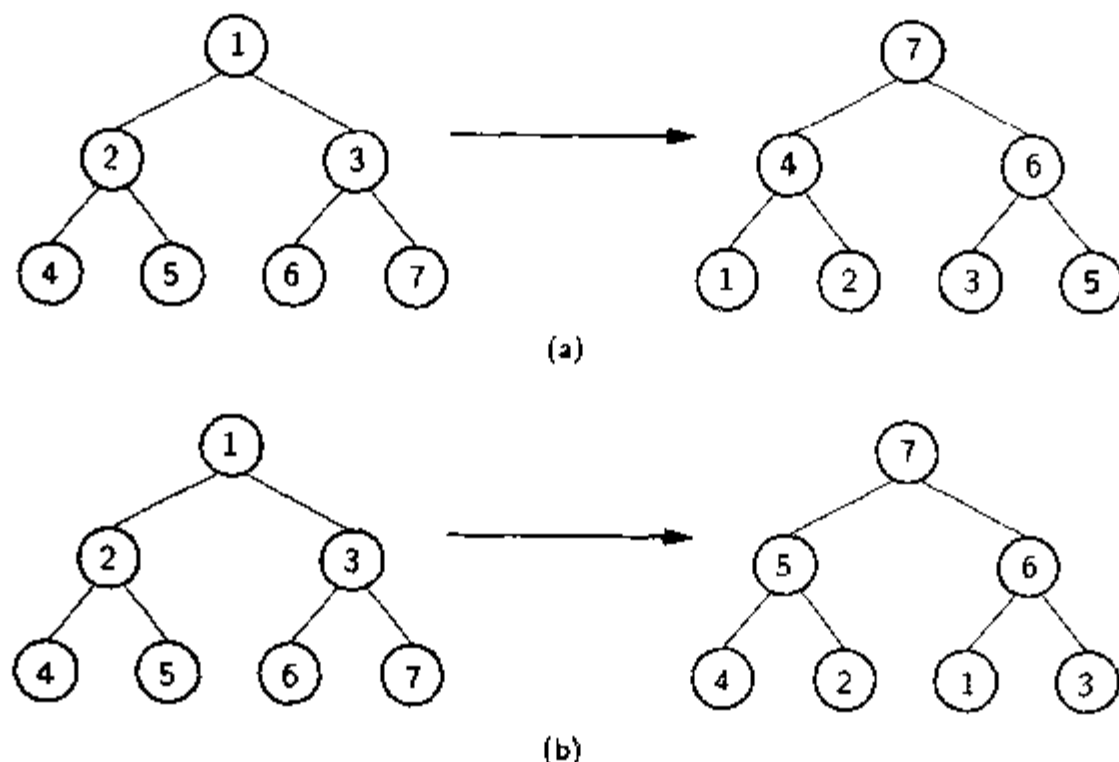


图 5.25 两种交换方法建堆

(a)这个堆通过 9 次交换建成,交换依次为(4-2),(4-1),(2-1),(5-2),(5-4),(6-3),(7-5),(7-6),(7-1)。(b)这个堆通过 4 次交换建成,依次为(5-2),(7-3),(7-1),(6-1)

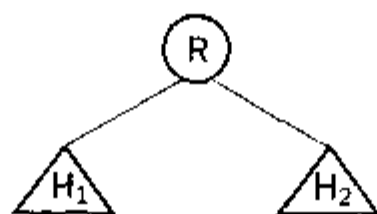


图 5.26 建堆算法的最后一个步骤。 $R$  的两棵子树都已经是堆  
只需要将  $R$  向下推,直至到达它在堆中的适当层

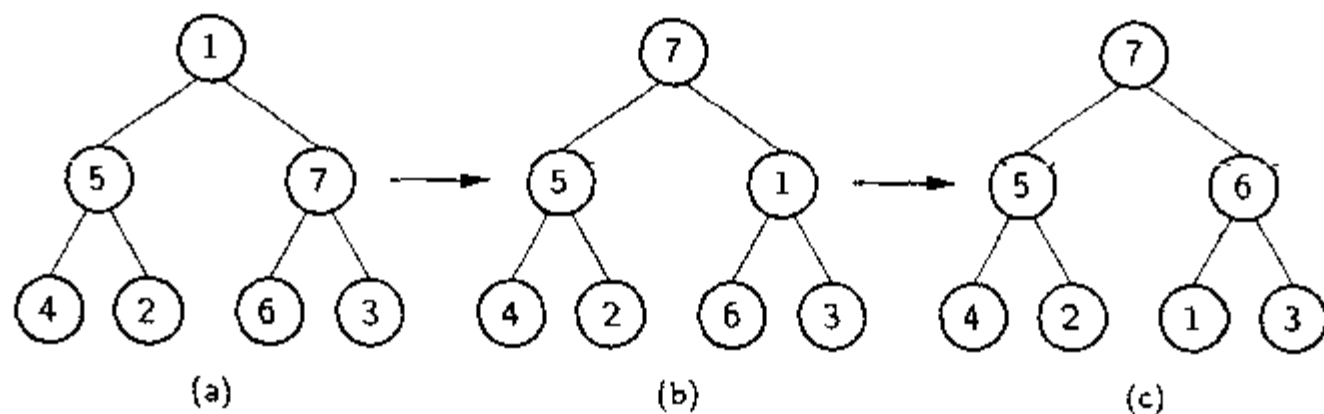


图 5.27 sift-down 操作。假设根的子树都已经是堆

(a)部分完成的堆。(b)值 1 与 7 交换。(c)值 1 与 6 交换,最后形成堆

一种简单的方法是从数组的较大序号结点向较小序号结点顺序访问。实际上,建堆过程不必访问叶结点(由于它们已经在最底层,不能再往下移了),所以建堆的算法从数组中部的第一个分支结点开始。图 5.25(b)表示的就是这个过程的结果。成员函数 `buildheap` 实现建堆的算法。

`buildheap` 的开销如何? 一种计算方法是算出每个元素到达其最终层所需移动的距离。

我们将只考虑一个元素在堆中向下移动的距离。堆中的某些元素是向上移动的,但是每向上一步都对应另外一个元素向下一步。由于没有哪个元素会先向上移动然后又被移下来,我们可以忽略那些向上的移动。这是因为一旦某个结点被处理过了,则其下面的每一个结点的值都比它小,因而它不可能又被向下拉回去。

按照这种方法,我们知道一个高度为  $d$  的堆中大约一半的结点深度为  $d-1$ ,而它们根本不能再向下移动。四分之一的结点深度为  $d-2$ ,而它们至多能向下移动一层。树中每向上一层,结点的数目就为前一层的一半,而子树的高度加 1。因而元素移动的最大距离总数为:

$$\sum_{i=1}^{\log n} (i-1) \frac{n}{2^i} = \Theta(n)$$

所以,这种算法的时间代价为  $\Theta(n)$ 。

如何在保持完全二叉树的形状的前提下,将最大值(根结点)从一个包含  $n$  个元素的堆中移走,并且剩下的  $n-1$  个结点值仍然符合堆的性质呢? 我们可以通过把堆中最后一个位置上的元素(数组中实际的最后一个元素)移到根位置上的方法来保持恰当的形状。我们现在考虑的是少了一个元素的堆。但是,新根结点的值很可能不是新堆中的最大值。这个问题利用 `siftdown` 对堆重新排序可以很容易解决。由于堆有  $\log n$  层深,删除最大元素的平均时间代价与最差时间代价为  $\Theta(\log n)$ 。

堆是前面讨论的优先队列的一种自然实现方法。需要时,作业可以添加到堆中(以其优先权值作为排序码)。操作系统选择新作业运行时,可以调用函数 `removemax`。

有些优先队列的应用要求能够改变已存储于队列中的对象的优先权。这可能要改变对象在堆中的存储位置。但是,最大值堆在查找一个任意值时效率不高:它只适合于查找最大值。不管怎样,如果我们知道对象在堆中的下标位置,那么更改其优先权(包括改变其位置以保持堆的性质)或者将其删除都是很简单的。`remove` 成员函数传入的参数是要删除的结点在堆中的位置。可以改变优先权的优先队列的典型实现方法需要一个辅助数据结构,以便高效地检索对象(例如使用二叉检索树 BST)。辅助数据结构的记录存储对象在堆中的下标,以便从堆中把对象删除,并根据其新优先权重新插入(参看项目设计 5.5)。7.5.1 小节给出了可以更改优先权的优先队列的应用。

## 5.7 深入学习导读

有关二叉树实现分支结点指针直接存储叶结点值的例子,请参考 shaffer and Braon [SB93]。

5.4.1 小节中 Huffman 编码树有最小带权外部路径长度的证明选自 Knuth [Knu73]。关于数据压缩技术的详细内容,参考 James A. Store 所著的《Data Compression: Methods and Theory》[Sto88]和 Dominic Welsh 所著的《Codes and Cryptography》[Wel88]。表 5.11 和 5.12 摘自 [Wel88]。

关于堆数据结构的讨论及其用法,请参考 Bentley 所著《Programming Pearl “Thanks, Heaps”》[Ben85, Ben88]。

现在已经有许多技术可以处理不友好的插入与删除操作,以保持合理的平衡二叉树结构。例如由 Adelson-Velskii 和 Landis 发明的,Knuth [Knu81]中讨论的 AVL 树。AVL 树实际上是



一个 BST,在其插入与删除过程中重新组织树的结构,以保证任何结点左右子树的高度至多相差 1。另一个例子是 13.2 节讨论的伸展树[ST85]。

## 5.8 习题

- 5.1 5.1.1 小节中称满二叉树非空叶结点的比率是最高的,证明这个事实。
- 5.2 定义一个结点的度数(degree)为其非空子结点的数目。作为满二叉树定理的推论,用归纳法证明任何二叉树中度数为 2 结点的数目为叶结点数目减 1。
- 5.3 (a)修改 5.2 节的前序周游算法为二叉树的中序周游。  
(b)修改 5.2 节的前序周游算法为二叉树的后序周游。
- 5.4 计算下列满二叉树实现的结构性开销占总开销的比率。各空间要求如下:  
(a)所有的结点都存储数据、两个子结点的指针以及一个父指针。数据域占 4 个字节,每个指针占 4 个字节。  
(b)叶结点与分支结点都存储数据和两个子结点的指针。数据域占 16 个字节,每个指针占 4 个字节。  
(c)叶结点与分支结点都存储数据,所有结点存储一个父指针,分支结点存储两个子结点指针。数据域占 8 个字节,每个指针占 4 个字节。  
(d)只有叶结点存储数据;分支结点存储两个子结点的指针。数据域占 4 个字节,每个指针占 2 个字节。
- 5.5 编写一个递归函数 `search`,传入参数为一棵二叉树(不是二叉检索树!)和一个值  $K$ ,如果值  $K$  出现在树中则返回 `true`,否则返回 `false`。函数原型如下:

```
boolean search(BinNode rt, int K)
```

- 5.6 编写一个算法,传入参数为二叉树根结点的指针,并按照层次顺序将结点的值打印出来。层次顺序首先打印根结点,接着是第 1 层的所有结点,再接着是第 2 层的所有结点,依此类推。  
提示:前序周游利用栈递归调用。考虑使用其他数据结构实现层次顺序周游。
- 5.7 定义二叉树的内部路径长度为所有分支结点深度的总和,外部路径长度则是所有叶结点深度的总和。用归纳法证明包含  $n$  个分支结点的满二叉树  $T$ ,如果内部路径长度为  $I$ ,外部路径长度为  $E$ ,则  $E = I + 2n$ ,对  $n \geq 0$  成立。
- 5.8 根据下面给定的字母和权建立 Huffman 编码树,并给出各字母的代码。

A	B	C	D	E	F	G	H	I	J	K	L
2	3	5	7	11	13	17	19	23	31	37	41

一段根据这样的分布频率包含  $n$  个字母的信息,其预期存储长度为多少位?

- 5.9 如果给定一组 16 个字母的权都相等,Huffman 编码树将会是怎样的形状? 在这种情况下,每个字母的平均代码长度是多少? 它与对 16 个字母的固定长度编码的最小可能长度的差别如何?
- 5.10 一组包含不同权的字母已经对应好 Huffman 编码,如果某一个字母对应编码 001,

则:

- (a)其他什么代码不可能对应字母?
- (b)其他什么代码肯定对应某个字母?

5.11 假设某个字母表各个字母的权如下:

Q	Z	F	M	T	S	O	E
2	3	10	10	10	15	20	30

- (a)按照这个字母表,一个包含  $n$  个字母的字符串采用 Huffman 编码在最差情况下需要多少位? 什么样的串会出现最差情况?
  - (b)按照这个字母表,包含  $n$  个字母的字符串采用 Huffman 编码在最佳情况下需要多少位? 什么样的串会出现最佳情况?
  - (c)按照这个字母表,一个字母平均需要多少位?
- 5.12 一个高度为  $h$  的堆最大和最小元素数目各为多少?
- 5.13 在最小值堆中,最大的元素的位置可能在哪儿?
- 5.14 本书给出了一个基于对已排序线性表进行操作创建 Huffman 编码树的算法。这需要  $\Theta(n^2)$  的时间代价,因为往线性表中插入一个中间结果的 Huffman 树开销大约为  $\Theta(n)$ 。请使用基于最小值堆而非线性表的优先队列改写这个算法。
- 5.15 为什么二叉检索树性质规定与根结点值相同的结点只能出现在根结点的右子树中,而不是在左右子树中都可出现?
- 5.16 画出图 5.19(a)中的 BST 加上值 5 以后的形状。
- 5.17 画出图 5.19(b)中的 BST 删除值 7 以后的形状。
- 5.18 编写一个递归函数 `count`,传入一棵二叉检索树和值  $K$ ,返回值小于或等于  $K$  的结点数目。函数 `count` 应尽可能少地访问 BST 的结点,函数原型如下:
- ```
int count(BinNode root, int K)
```
- 5.19 编写一个递归函数 `printRange`,传入一个 BST,一个较小的值和一个较大的值,按照顺序打印出介于两个值之间的所有结点。函数 `printRange` 应尽可能少地访问 BST 的结点。函数原型如下:
- ```
void printRange(BinNode root, int low, int high)
```
- 5.20 画出对下列存储于数组中的值执行 `buildheap` 后得到的最大值堆:
- 10   5   12   3   2   1   8   7   9   4
- 5.21 (a)画出从图 5.25(b)的最大值堆中删除最大元素后得到的堆。  
(b)画出从图 5.25(b)的最大值堆中删除元素 5 后得到的堆。
- 5.22 修改图 5.24 的堆定义,实现最小值堆。成员函数 `removemax` 应该使用新的函数 `removemin` 来代替。

## 5.9 项目设计

- 5.1 完成根据 5.4 节所给代码建立 Huffman 编码树的源代码。包括计算各个字母对应

代码的函数,以及对信息进行编码与解码的函数。这个对象可以进一步扩展以支持对文件的压缩。为此必须增加两个步骤:(1)扫描整个文件以生成文件中各个字母的实际使用频率;(2)在编码文件的开头存储 Huffman 树,以便解码函数使用。如果设计这种 Huffman 树存储表示有困难,请参考 6.5 节。

- 5.2 完成 5.4 节关于建立 Huffman 编码树的代码实现。修改图 5.15 中的函数 `buildTree`,用一个最小值堆替换已排序的线性表,以存储部分 Huffman 编码树。
- 5.3 在二叉树中处理 null 指针问题的一种方法是利用其空间做其他事情。例如,穿线二叉树(threaded binary tree)。穿线树的每个结点存储两个附加位,以指示成员 `left` 和 `right` 是正常指向子结点还是作为线索。如果 `left` 不是一个指向子结点的指针(例如,在普通的二叉树中它将为 null),则它线索化地存储了其结点在中序下前驱(in-order predecessor)的指针。中序下的前驱是指中序周游下,将当前结点之前打印出来的结点。如果 `right` 不是指向子结点的指针,则它线索化地存储了该结点在中序下后继(inorder successor)的指针。中序下的后继是指中序周游中,将当前结点之后打印出来的结点。穿线树的主要优点是一些操作,如中序周游,可以不用递归或栈而实现。

请把 BST 实现为穿线树,其中包括一个非递归的前序周游函数。

- 5.4 利用 BST 存储数据库记录实现一个城市数据库。每个数据库结点包含城市名称(一个任意长度的串)和以整数  $x$  与  $y$  表示的城市坐标。根据城市的名称组织该 BST。这个数据库应该能够允许记录的插入,根据名称或坐标删除,以及根据名称或坐标进行检索。另一个应该实现的操作是打印出与指定点的距离在给定值之内的所有城市记录。统计各个操作的运行时间。

利用 BST 时,哪些操作的实现理所当然地效率较高(例如,平均情况下时间代价为  $\Theta(\log n)$ )? 如果增加一个或多个根据坐标组织的 BST,能使数据库系统变得更高效吗?

- 5.5 利用图 5.24 的最大值堆实现一个优先队列。对于队列的操作应该支持下列几种指令:

```
void enqueue(int ObjectID, int Priority);
int dequeue();
void changeweight(int ObjectID, int newPriority);
```

函数 `enqueue` 向优先队列中插入一个 ID 号为 `ObjectID`、优先级为 `priority` 的新对象。函数 `dequeue` 从优先队列中删除优先级最高的对象,并返回该对象的 ID 号。函数 `changeweight` 将 ID 号为 `ObjectID` 的对象的优先级改为 `newPriority`。类型 `Elem` 应该为一个存储对象 ID 及其优先级的类或结构。你需要一种机制,以便获取所需对象在堆中的位置。利用一个数组,将 `ObjectID` 值为  $i$  的对象存放在数组位置  $i$  处(记住测试时应该保证 `ObjectID` 的数值在数组的边界限定之内)。你还需要对堆的实现进行修改,以存储对象在数组中的位置,以便堆中对象的修改可以在辅助数组结构中记录下来。

## 第 6 章 树

本章扩展了第 5 章对树的讨论,树结点可以有任意数目的子结点。这种属性使得树在实现上比二叉树要困难得多。

6.1 节给出了有关树的术语。6.2 节给出一个简单的表示方法,解决了对等价类问题的处理。6.3 节包括几个基于指针的实现方法。6.4 节推广了二叉树的性质,定义了每个结点可以有  $K$  个子结点的  $K$  叉树。6.5 节给出了利用线性结构实现树结构的方法。

### 6.1 树的定义与术语

一棵树(tree)  $T$  是由一个或一个以上结点组成的有限集,其中有一个特定的结点  $R$  称为  $T$  的根结点。集合  $(T - \{R\})$  中的其余结点可被划分为  $n \geq 0$  个不相交的子集  $T_1, T_2, \dots, T_n$ , 其中每个子集都是树,并且其相应的根结点  $R_1, R_2, \dots, R_n$  是  $R$  的子结点。子集  $T_i (1 \leq i \leq n)$  称为树  $T$  的子树(subtree)。子树可以如下排序:  $T_i$  排在  $T_j$  之前,当且仅当  $i < j$ 。为方便起见,子树从左到右排列,其中  $T_1$  被称为  $R$  的最左子结点。结点的出度(out degree)被定义为该结点的子结点数目。森林(forest)定义为--棵或更多棵树的集合。

图 6.1 中树的表示法是从第 5 章二叉树的表示方法推广而来的。一些新增加的术语是树所特有的。

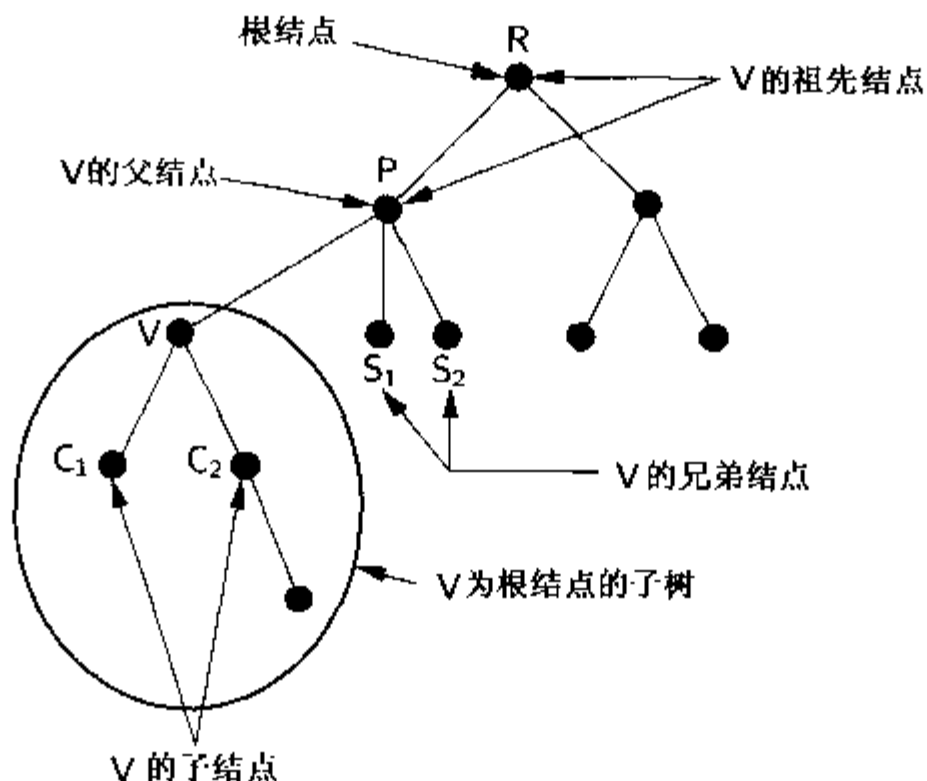


图 6.1 树的表示法。P 是 V 的父结点,也是  $S_1$  和  $S_2$  的父结点。因此,  $V$ 、 $S_1$  和  $S_2$  是 P 的子结点。R 和 P 是 V 的祖先结点。V、 $S_1$  与  $S_2$  互称兄弟结点(sibling)。椭圆形圈住的部分是以 V 为根结点的子树

除了根没有父结点外,树中每个结点都正好只有一个父结点。根据这一观察,立刻就能

得出结点为  $n$  的树必然有  $n - 1$  条边, 因为除了根以外的每个结点都有一条边连接它的父结点。

### 6.1.1 树结点的 ADT(抽象数据类型)

在讨论树的实现之前, 应该先明确这种实现所必须支持的操作。任何实现都必须可以初始化一棵树。给出一棵树, 我们会想到直接访问根结点。典型的情况下, 我们希望可以按照某种顺序处理一个结点的子结点。这需要结点的数据域值、结点的最左子结点, 结点的下一个(右侧的)兄弟结点或结点的父结点, 同样需要从树中删除结点的方法。

在上述操作中, 有些是针对结点的(到达父结点、兄弟结点或子结点), 也有一些是针对整棵树的(初始化、返回根结点)。于是, 类似于二叉树, 树的实现包括树类与结点类。图 6.2 给出了这两个类的 Java 接口。

```
interface GNode {                                     // General tree node ADT
    public Object value();                             // Return the value
    public boolean isLeaf();                           // TRUE if this node is a leaf
    public GNode parent();                             // Return the parent
    public GNode leftmost_child();                     // Return the leftmost child
    public GNode right_sibling();                       // Return the right sibling
    public void setValue(Object value);                 // Set the value
    public void setParent(GNode par);                   // Set the parent
    public void insert_first(GNode n);                  // Add a new leftmost child
    public void insert_next(GNode n);                   // Insert a new right sibling
    public void remove_first();                         // Remove the leftmost child
    public void remove_next();                          // Remove the right sibling
} // interface GNode

interface GenTree {                                   // General tree ADT
    public void clear();                               // Clear the tree
    public GNode root();                               // Return the root
    // Make the tree have a new root with children first and sib
    public void newroot(Object value, GNode first, GNode sib);
} // interface GenTree
```

图 6.2 树结点接口和树接口

### 6.1.2 树的周游

二叉树有三种周游方法: 前序周游、后序周游、中序周游。对树而言, 前序及后序周游的定义与二叉树很相似。树的先根周游(前序)先访问根结点, 再依次由左至右先根周游每棵子树。树的后根周游(后序)先由左至右依次后根周游每棵子树, 再访问根结点。中根周游对树不具有自然的定义。因为树内部的结点不具有特定数目的子结点。可以给出一些很随便的定义, 例如: 先中根周游最左子树, 然后访问根结点, 再依次中根周游其余子树。一般不使用树的中

根周游。

对图 6.3 中树的先根周游按照顺序 RACDEBF 访问树的结点。而后根周游将按照顺序 CDEAFBR 访问树的结点。

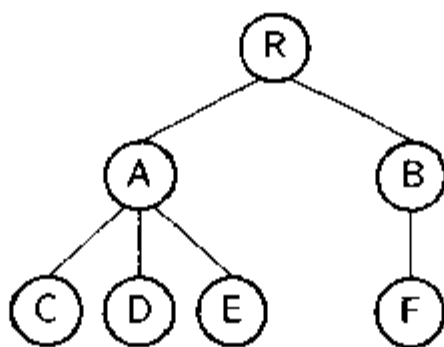


图 6.3 树的一个示例

要先根周游一棵树,必须对每一个结点(譬如说结点为 R)都从左到右周游其子结点。可以从 R 的最左子结点(假定为 T)开始,从 T 结点转到 T 的右兄弟结点,再转到 T 的右兄弟的右兄弟,依此类推。

下面给出了一个利用图 6.2 中的 GNode 接口,按照先根顺序打印树中结点的 Java 函数。

```
static void preorder(GNode rt) { // Preorder traversal
    if (rt.isLeaf()) System.out.print( "Leaf: ");
    else System.out.print( "Internal: ");
    System.out.println( rt.value() );      // Print or take other action
    GNode temp = rt.lefmost_child();
    while (temp != null) {
        preorder(temp);
        temp = temp.right_sibling();
    }
}
```

## 6.2 父指针表示法

或许实现树的最简单方法就是对每个结点只保存一个指针域指向其父结点,这种实现被称为父指针(parent pointer)表示法。很明显,这种实现并非出于一般性的目的,因为它对诸如找到一个结点的最左子结点或右侧兄弟结点这样的重要操作是不够的。那么用这种方法来实现树看来是没有价值的。然而,父指针表示法精确地保存了用于解答下面有用问题所需的信息:“给出两个结点,它们是否在同一棵树中?”为了解答这个问题,仅仅需要跟着由每个结点到它相应根结点的祖先指针序列即可。如果两个结点到达同一根结点,它们一定在同一棵树中。查找一个给定结点的根结点的过程称为 FIND。如果找到的根结点是不同的,那么两个结点就不在同一棵树中。

一个典型的应用就是判断两个结点是否在同一个集合中,如果不是,则将两个集合归并为一个。由于集合被合并,这个过程称为 UNION,并且整个操作以“UNION/FIND 算法”命名。

“UNION/FIND”算法用一棵树代表一个集合。如果两个结点在同一棵树中,则认为它们在同一集合中。树中的每个结点(除根结点以外)有且只有一个父结点。这样,每个结点可

用同样大小的存储空间来实现。结点中仅需保存结点的父指针信息,树本身可以存储为一个以其结点为元素的数组。根结点指针保存 null 指针值。图 6.4 中给出了树结点和树的父指针实现。这些类已经比图 6.2 大大简化了,因为我们仅仅需要一般树操作集中有限的子集。类 GenTree 给出了两个新的函数 differ 和 UNION。函数 differ 检查两个结点是否在不同的集合中,函数 UNION 归并两个集合。私有函数 FIND 用来找到目标结点的根结点。

```

class GNode { // General tree node for UNION/FIND
    private GNode par; // Parent pointer
    public GNode() { par = null; } // Constructor
    public GNode parent() { return par; } // Return node's parent
    public GNode setParent(GNode newpar) // Set the parent pointer
    { return par = newpar; }
} // class GNode

class GenTree { // General Tree class for UNION/FIND
    private GNode[] array; // Node array

    public GenTree(int size) { // Constructor
        array = new GNode[size]; // Create node array
        for (int i = 0; i < size; i++)
            array[i] = new GNode();
    }

    public boolean differ(int a, int b) { // Nodes in different trees?
        GNode root1 = FIND(array[a]); // Find root of node a
        GNode root2 = FIND(array[b]); // Find root of node b
        return root1 != root2; // Compare roots
    }

    public void UNION(int a, int b) { // Merge two subtrees
        GNode root1 = FIND(array[a]); // Find root of node a
        GNode root2 = FIND(array[b]); // Find root of node b
        if (root1 != root2) root2.setParent(root1); // Merge subtrees
    }

    private GNode FIND(GNode curr) { // Find root
        while (curr.parent() != null) curr = curr.parent();
        return curr; // At root
    }
} //Class GenTree

```

图 6.4 树的 UNION/FIND 算法的实现

使用 UNION/FIND 算法的程序中应该有一个  $n$  个结点的集合,其中每个结点被赋给 0

到  $n-1$  范围内惟一的下标值。类 GenTree 产生并初始化 UNION/FIND 数组,而函数 UNION 与 differ 只需把结点下标值作为输入。

图 6.5 阐释了父指针表示法。注意在数组内结点可能以任意顺序出现,并且数组可以存储任意多棵无关的树。例如图 6.5 在同一个数组中存储了两棵树。这样,一个数组就能存储这样的一组数据项,这些数据项分属于数目随意变化的不同集合。

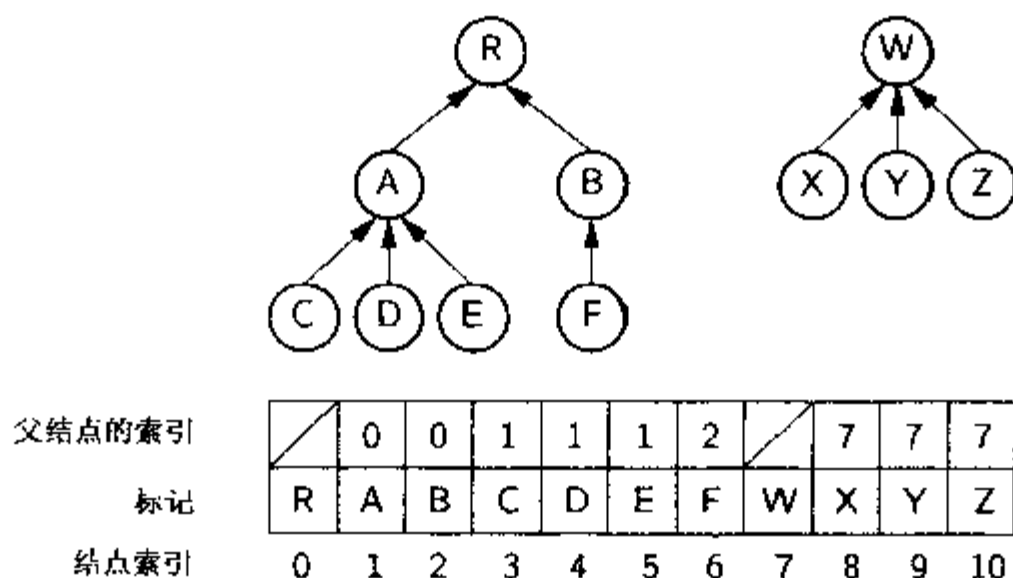


图 6.5 父指针数组表示法。每个结点存储其值以及一个指向父结点的指针。为了简明起见,父指针表示为父结点在数组中位置的下标值。树的根结点存储 null 值,在图中表示为斜线。本图表示了存储在同一个数组中的两棵树,一棵以 R 为根结点,一棵以 W 为根结点。

考虑把一个集合中的元素分配到称为等价类 (equivalence class) 的不相交子集中的问题。输入为一系列等价对,一个等价对可能会是 A 等价于 C (亦即 A 与 C 在同一子集中)。如果又有一个等价对为 A 与 B,那么由传递性可知 C 与 B 等价。这样,一个等价对归并了两个子集,其中每个子集各包含了几个元素。

UNION/FIND 算法可以很容易地解决等价类问题。开始时,每个元素都在独立的只包含一个结点的树中,而它自己就是根结点。通过使用函数 differ 可以检查一个等价对中的两个元素是否在同一棵树中。如果是,由于它们已经在同一个等价类中,不需要作变动。否则两个等价类可以用 UNION 函数归并。

作为等价类问题的一个例子,考虑字符集 (A, B, C, D, E, F, G, H, I, J)。首先,我们假定每个字符均在相异的等价类中。通过把每个字符存储为其所属树的根结点可以代表这种情况。图 6.6(a) 所示就是父指针表示法数组的初始状态。

现在考虑处理等价关系 (A, B) 时的情况。包含 A 的树的根结点为 A,包含 B 的树的根结点为 B。这样,两个结点中的一个就被设置为指向另一个。在这种情况下把哪一个指向另一个是没有影响的,因此可以随意选择在字母表中排在前面的作为根结点。在父指针数组中,只须把 B 的父指针设置为指向 A 即可。可以同样处理等价对 (C, H), (G, F), (D, E)。类似地处理等价对 (I, F)。I 与 F 均为其所属树的根结点,所以 I 被设置为指向 F。请注意这也将使 G 与 I 等价。图 6.6(b) 即为对这 5 个等价对的处理结果。

父指针表示法并不限制共享同一父结点的结点数目。为了使等价对的处理尽可能高效,每个结点与其相应的根结点的距离应尽可能小。这样,当我们把两个等价类归并到一起时,可以保持树的高度较小。在理想情况下,每棵树的结点均应该直接指向根结点。为了达到此目的,可能需要过多的附加处理以致于得不偿失。



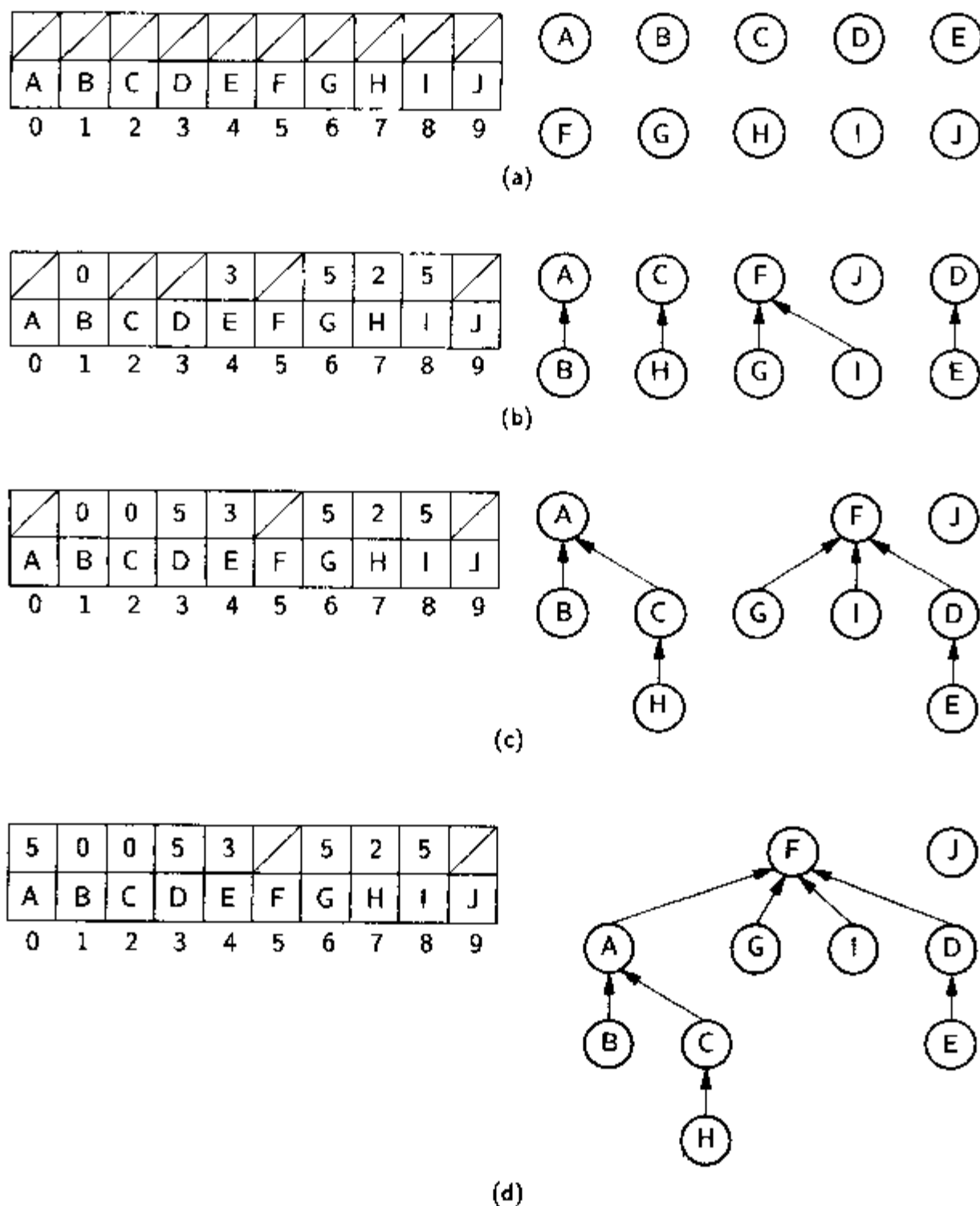


图 6.6 等价类处理一例

(a) 10 个数据项分别在 10 个等价类中的初始结构。(b) 对 5 个等价对 (A,B), (C,H), (G,F), (D,E), (I,F) 的处理。(c) 对另外两个等价对 (H,A) 与 (E,G) 的处理。(d) 对最后一个等价对 (H,E) 的处理

降低树高度比较省力的途径应该是使两棵树合并的方式灵活一些。一个简单的技术,称作“重量权衡合并规则”(weighted union rule),在把结点较少的一棵树与结点较多的一棵树归并时,将结点较少树的根结点指向结点较多树的根结点。因为在较小树中所有结点的深度均增加 1,而合并后树的结点数至少是较小树的两倍,这可以把树的整体深度限制在  $O(\log n)$ 。因此,任何结点的深度最多只会增加  $\log n$  次。

处理图 6.6(b) 中等价对 (I,F) 时,以 F 为根结点的树有两个结点,而以 I 为根结点的树只有一个结点。因此把 I 指向 F,而不是相反。图 6.6(c) 是对另外两个等价对 (H,A) 和 (E,G) 的处理结果。第一个等价对中 H 所属树的根结点是 C,而 A 所属树的根结点是它本身。每棵树均包含两个结点,因此,哪一个根结点作为归并后的根结点是没有关系的。现在看等价对 (E,G) 的情形。E 的根结点是 D,G 的根结点为 F。而 F 为较大树的根结点,所以 D 被设置为指向 F。

不是处理等价对时都会归并两棵树。例如,在图 6.6(c)状态下处理等价对(F,G)时,因为 F 已经是 G 的根结点,所以不会有什么变动。

重量权衡合并规则有助于减小树的深度,但是我们还有更好的办法。路径压缩(path compression)是一种可以产生极浅的树的方法。当查找一个结点 X 的根结点时可以采用路径压缩。设根结点为 R,则路径压缩把由 X 到 R 的路径上每个结点的父指针均设置为直接指向 R。首先要查找 R,然后顺着由 X 到 R 的路径把每个结点的父指针域均设置为指向 R。图 6.7 提供了一种可供选择的递归算法。函数 FIND 不仅返回当前结点的根结点,而且把当前结点所有祖先结点的父指针都指向根结点。

```

GNode FIND(GNode curr) {
    if (curr.parent() == null) return curr; // At root
    return curr.setParent(FIND(curr.parent()));
}

```

图 6.7 带路径压缩的 FIND 函数的 Java 代码

路径压缩使 FIND 的开销接近于常数。更精确地讲,是“非常接近于常数”。对  $n$  个结点进行  $n$  次 FIND 操作的路径压缩开销(用重量权衡合并规则来归并集合)为  $\Theta(n \log^* n)$ 。 $\log^* n$  是在  $n \leq 1$  之前要进行的对  $n$  取对数操作的次数。例如,  $\log 65536 = 16$ ,  $\log 16 = 4$ ,  $\log 4 = 2$ ,  $\log 2 = 1$ , 因此  $\log^* 65536 = 4$ (4 次  $\log$  操作)。由于  $\log^* n$  增长非常缓慢,因此,一系列  $n$  个 FIND 操作的开销非常接近  $\Theta(n)$ 。

注意到上面的讨论并不意味着处理  $n$  对等价对后得到的树的深度必然为  $\Theta(\log^* n)$ 。可以构造出一系列等价对操作,使产生的树深度为  $\Theta(\log n)$ 。但是,这个序列中许多等价对操作会只依赖于被归并的树的根结点,而只需很少的处理时间。所需要的总的处理时间为  $\Theta(\log^* n)$ ,对于每个等价对的操作时间接近于常数。这是均摊分析方法的一个例子,将在 14.3 节讨论。

图 6.6(d)是用没有路径压缩的标准重量权衡合并规则处理图 6.6(c)中等价对(H,E)的结果。图 6.8 所示是用路径压缩处理同一等价对的结果。在找到 H 结点的根结点后,可以用路径压缩使 H 直接指向根结点 A。类似地,E 也被设置为直接指向根结点 F。最后,A 被设置为指向根结点 F。

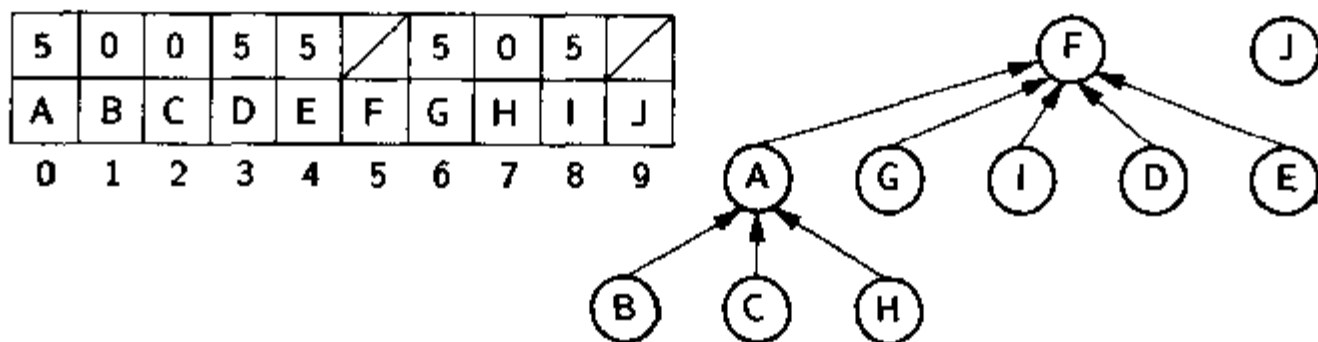


图 6.8 路径压缩一例。所示为对图 6.6(c)中等价对(H,E)的处理结果

## 6.3 树的实现

我们现在着手给出树的表示法并要求图 6.2 的 ADT 中所有成员函数均可以有效使用。本节给出几种表示树的方法。每种表示法在存储每个结点所需要的空间以及关键操作的简明性方面各有优劣。

树的表示法不应限制每个结点的子结点数目。在某些应用中,结点一旦生成,其子结点的数目便不再改变。在这种情况下,当一个结点产生时,可以基于其子结点的数目为它分配一块固定的存储空间。如果子结点可以插入或删除,情况就会变得更加复杂,这要求该结点的存储空间也要进行相应调整。

### 6.3.1 子结点表表示法

首先介绍一种“子结点表表示法”(list of children)。这是一种很简单的表示方法,其中每个分支结点均存储其子结点按照一定顺序形成的一个链表。如图 6.9 所示。

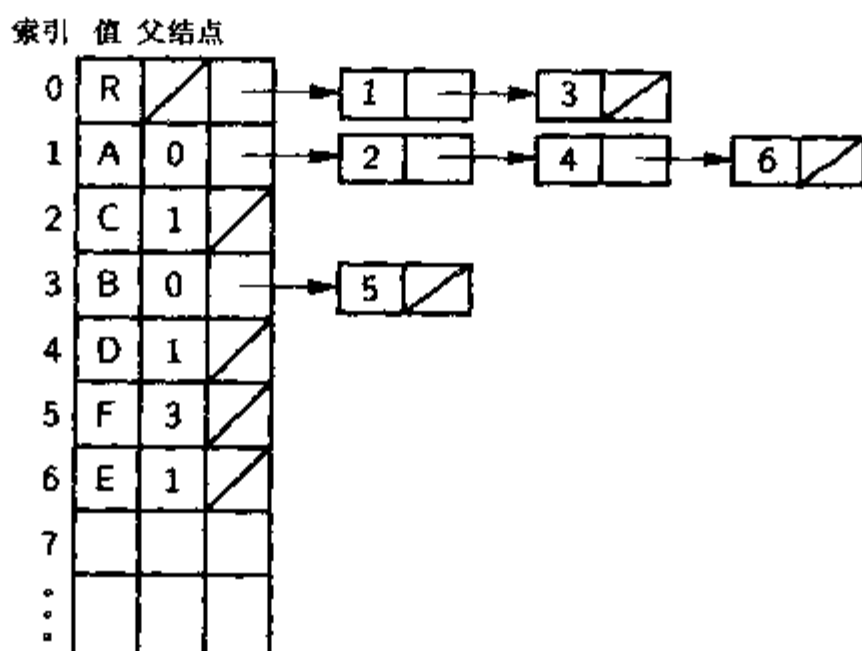


图 6.9 以“子结点表”表示法实现图 6.3 中的树。最左侧的一列数标明数组的下标。值 (“Val”)列存储结点值。父结点 (“Par”)列存储父指针。为了简明起见,这些指针实际上是数组的下标值。对于分支结点,最后一列值存储指向子结点链表的指针。链表上的每个表项均存储着指向结点的某个子结点的指针(如图所示目标结点的下标)

“子结点表”表示法在数组中存储树的结点。每个结点包括结点值、一个父指针以及一个指向子结点链表的指针,链表中子结点的顺序由左至右。每个链表表项均包含指向一个子结点的指针。这样,结点的最左子结点可由链表的第一个表项直接找到。但是,找到结点的右侧兄弟结点要困难一些。考虑结点 M 与其父结点 P,为了找到 M 的右侧兄弟结点,必须沿着 P 的子结点表移动,直至找到一个表项,其中存储着指向 M 的指针。下一个表项就存储着指向 M 的右侧兄弟结点的指针。因此在最差情况下,必须查找 M 的父结点的全部子结点。

如果两棵树分别存储在不同的数组中,使用这种表示方法会给归并这两棵树带来困难。如果两棵树存储在同一个结点数组中,则要添加树 T 成为结点 R 的子树,只需要将 T 的根结点添加到 R 的子结点表中即可。

### 6.3.2 左子结点/右兄弟结点表示法

“子结点表”表示法使得存取一个结点的右侧兄弟结点较为困难。图 6.10 中给出了一个改进的实现。每个结点都存储结点的值,以及指向父结点、最左子结点和右侧兄弟结点的指针。这样,抽象数据类型(ADT)的基本操作可通过读取结点中的一个值来实现。如果两棵树存储在同一个数组中,那么把其中一个添加为另一棵树的子树只需简单设置三个指针值即可。图 6.11 就是使用这种方法归并树的图示。这种表示法比子结点表表示法空间效率更高,而且结点数组中的每个结点仅需要固定大小的存储空间。

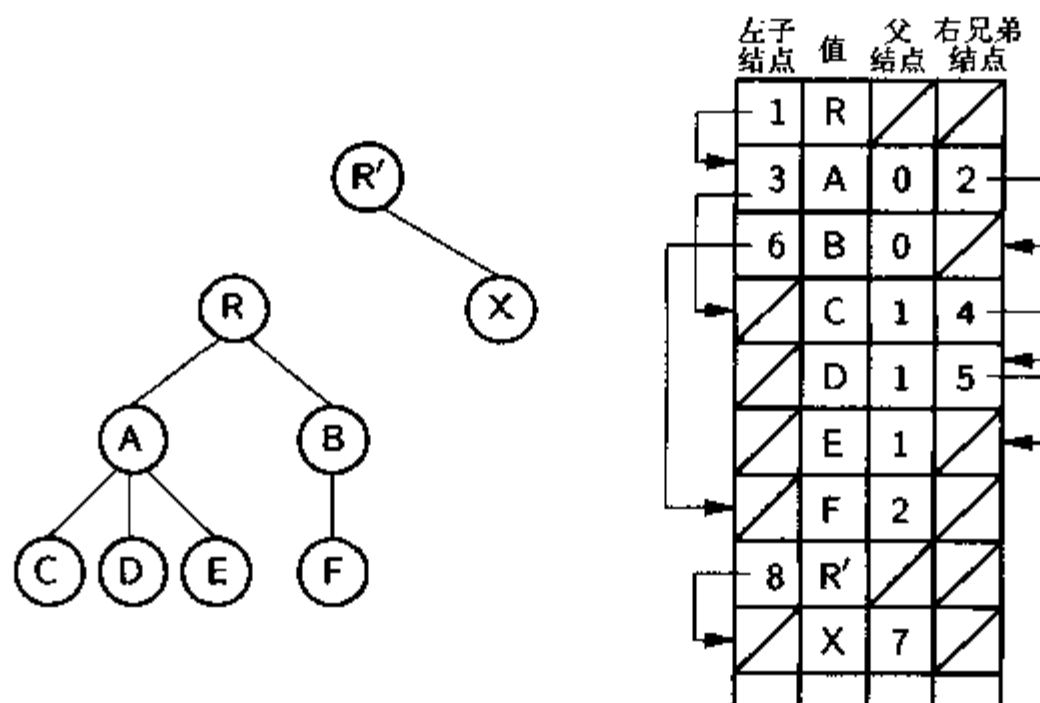


图 6.10 “左子结点/右兄弟结点”(left child/right sibling)表示法

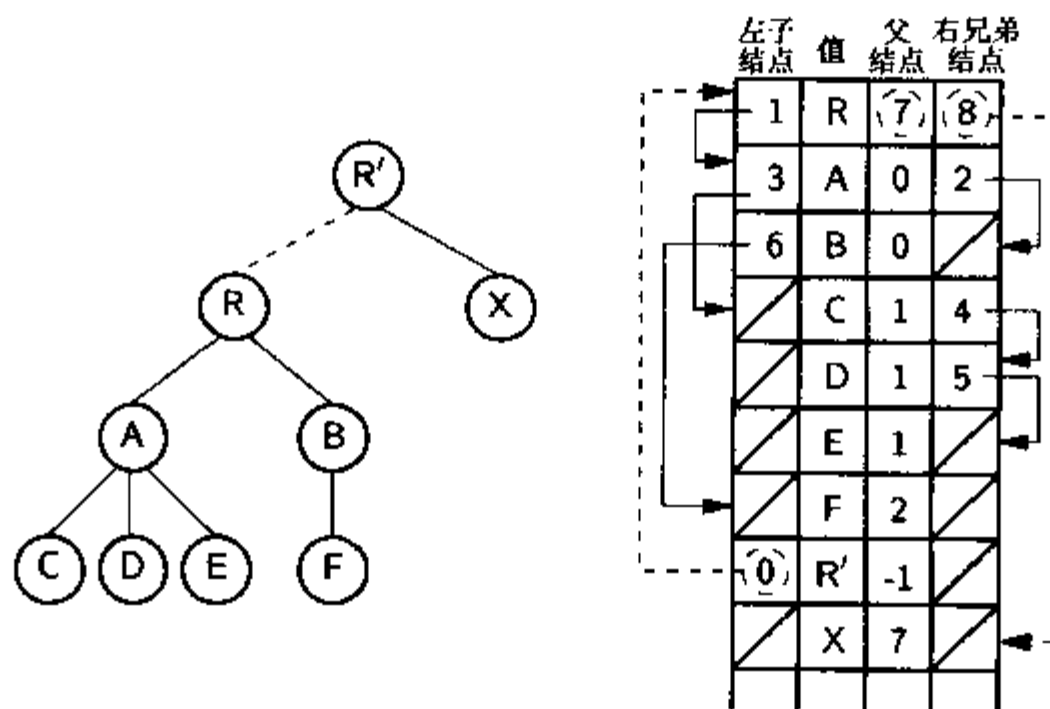


图 6.11 使用“左子结点/右兄弟结点”实现对树的归并。图 6.10 中以 R 为根结点的树成为 R' 的最左子树。结点数组中有三个指针被调整: R' 的左子结点指针指向 R, R 的右侧兄弟结点指针指向 X。R 的父指针指向 R'

### 6.3.3 动态结点表示法

上述两种实现树的方法使用数组存储结点。我们接着尝试将链接表示法扩展到树。在二叉树的链接表示法中,每个结点存储独立的动态对象,包含其值以及指向两个子结点的指针。但是,树的结点可以有任意数目的子结点,并且这个数目在结点的生存周期中还可能发生变化。树结点的表示法必须支持这个属性。一种实现方法是对任意一个结点可能有的子结点数目加以限制,并且对每个结点分配确定数目的指针域。这种方法有两个主要缺点。首先,它对子结点的数目强加了不必要的限制,使得某些树用这种方法无法表示。其次,这可能会造成空间的极度浪费,因为大多数结点不会有那么多的子结点,因此有些指针的位置是空闲的。

另外一种实现方法是为每个结点分配可变的存储空间。这有两种基本途径。其一是将一个指向子结点的指针数组作为结点的一部分分配给结点。实质上,每个结点存储一个基于数组的子结点指针表。图 6.12 体现了这种想法。这种方法要求在一个结点生成时就知道它的子结点数目,这对某些应用可能适合,而对其他应用就可能不适合。在子结点的数目不变时,它工作得最好。如果子结点的数目发生变动(特别是增加),就必须想出一些特殊办法来进行校正,以使子结点指针数组的规模可变。一种可能的方法是从可利用空间中以恰当的大小分配一个新结点,然后把原来的结点返还给可利用空间,以便重新利用。这种方法在像 Java 这样收集无用存储单元的语言环境下工作得相当好。例如,一个结点 M 起初有两个子结点,当 M 生成时,两个子结点指针的空间被分配给 M。如果第三个子结点添加到 M,则应分配一个有三个子结点指针的新结点, M 的内容复制到新的结点空间中, M 被返还给可利用空间。这种方法要求对可变长存储单元进行存储管理,这将在 12.4 节予以讨论。在图 6.12 中应注意到当前子结点数目存储在子结点数(size)域中。子结点指针存储在一个有 size 个元素的数组中。

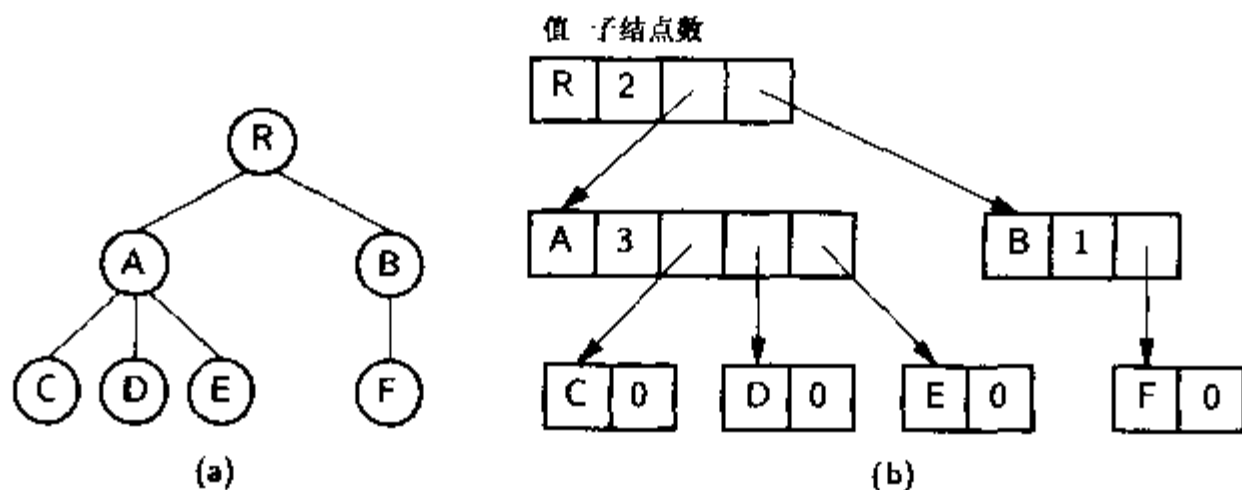


图 6.12 树的动态表示法(dynamic node implementation),子结点指针存储在规模固定的数组中

(a)树。(b)树的实现。对每个结点在第一个域中存储其值,而第二个域中存储子结点指针数组的规模

另一种途径更具有灵活性,但是需要更多的空间,如图 6.13 所示,每个结点存储一条子结点指针的链表。这种表示法本质上与 6.3.1 小节中“子结点表”表示法相同,但是它动态地分配结点空间,而不是把结点分配在数组中。

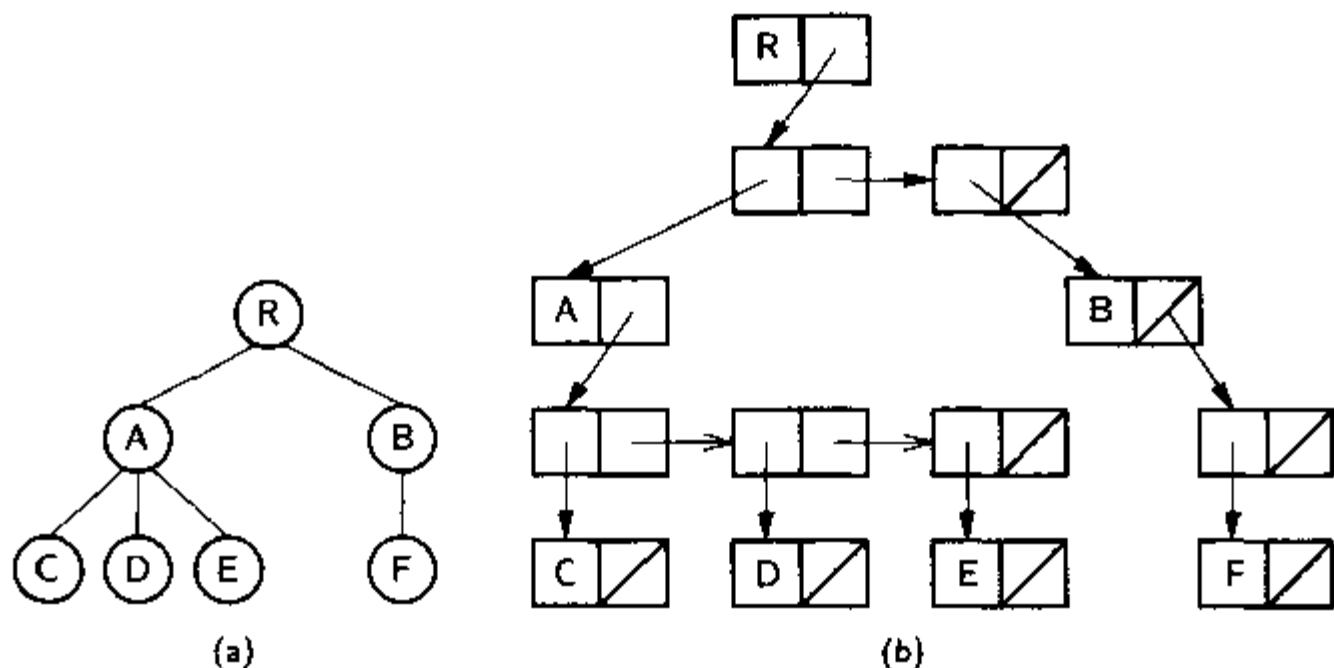


图 6.13 带有子结点指针链表的动态树表示法  
(a)树, (b)树的实现

#### 6.3.4 动态“左子结点/右兄弟结点”表示法

6.3.2 小节的“左子结点/右兄弟结点”表示法对每个结点存储固定数目的指针。毫无疑问,它适合于动态实现。本质上,我们使用二叉树来替换树(或森林)。“左子结点/右兄弟结点”表示法中的每个结点在一棵新的二叉树结构中指向两个“子结点”。这个新的结构中左子结点在树中是结点的最左子结点。右子结点是结点原来的右侧兄弟结点。图 6.14 把森林转化为一棵二叉树。我们简单地添加了指向右侧兄弟结点的指针,删除了除指向最左子结点以外其他子结点的指针。

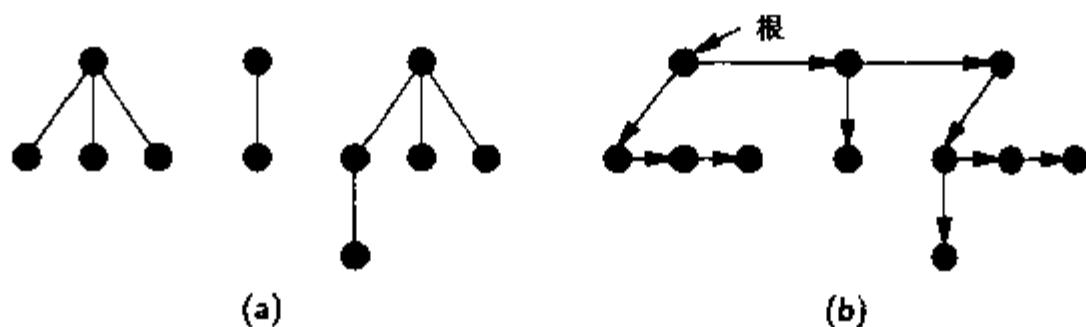


图 6.14 将森林转化为单一的二叉树。每个结点存储指向它的左子结点和右侧兄弟结点的指针。为了进行转化,假定树的根结点为兄弟结点

由于树的每个结点均包含固定数目的指针,而且树的 ADT 的每个函数均能有效实现,因此动态“左子结点/右兄弟结点”表示法(dynamic “Left Child/Right Sibling”)比 6.3.1 小节到 6.3.3 小节中的其他方法更为常用。

### 6.4 K 叉树

K 叉树(K-ary Tree)的结点有  $K$  个子结点。这样,二叉树就是 2 叉树。在 13.3 节中将讨论的 PR 四分树就是 4 叉树的特例。与树相区别,K 叉树的结点有  $K$  个子结点,子结点数目是固定的,因此相对来说容易实现些。通常 K 叉树与二叉树有许多相似之处,并且对 K 叉树

结点可以使用与二叉树类似的实现。注意到当  $K$  变大时,空指针的潜在数目会增加,并且叶结点与分支结点在所需空间大小上的差异也会更显著。这样,当  $K$  增加时,对叶结点与分支结点采用不同实现的优点便会更加明显。

相应的,满(full) $K$ 叉树和完全(complete) $K$ 叉树与满二叉树和完全二叉树是类似的。图 6.15 中对  $K=3$  给出了完全  $K$  叉树和满  $K$  叉树。

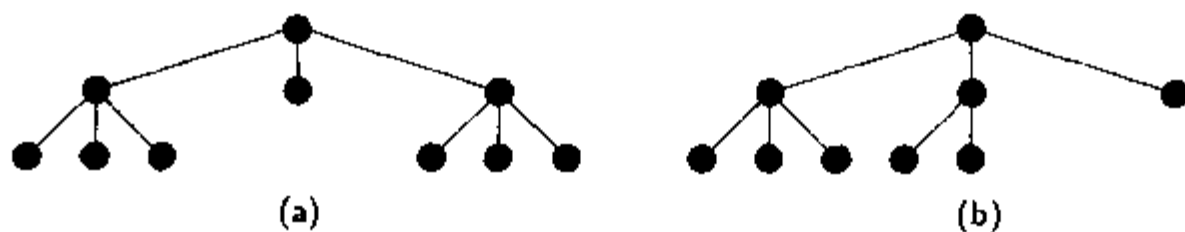


图 6.15 完全 3 叉树与满 3 叉树

(a)这棵树是满的(并不完全)。(b)这棵树是完全的(并不满)

二叉树的许多性质可以推广到  $K$  叉树。在考虑  $K$  叉树中空指针的数目、 $K$  叉树叶结点数目与分支结点数目之间的关系时,同样可推出类似于 5.1.1 小节的定理。我们也可以像 5.3.3 小节中一样把完全  $K$  叉树存储在一个数组中。

## 6.5 树的顺序表示法

现在我们来考虑一种与前面所述有本质区别的实现树的途径。其目的在于存储一系列结点的值,其中具有尽可能少但是对于重建树结构必不可少的信息。这种方法称为顺序树表示法(sequential tree implementation),由于没有存储指针固具有节省空间的优点。其缺点是对任何结点的存取,都必须顺序查找所有在结点表中排在它前面的结点。换句话说,对结点的存取必须从结点表的头部开始,按照存储顺序依次查找,直到找到要处理的结点。这样,就失去了这部分讨论的其他实现方法的一个主要优点:对树中任意结点的高效率存取(典型时间代价为  $\Theta(\log n)$ )。由于顺序表示法节省空间,因此它是把树压缩在磁盘上以备以后使用的理想方法,而且树结构在日后处理需要时可以很容易重建。

树的顺序表示法把结点值按照它们在先根周游中出现的顺序存储起来,有关树的形状的充足信息也同时被存储起来。如果树具有受限的形状,例如一棵满二叉树,那么需要存储的有关结构信息就可以少一些。由于普通的树具有比较灵活的结构,因此需要更多的附加信息来描述形状。树的顺序表示法有很多可选方案。我们将从适合于二叉树的方法开始,然后将其推广为与一般树结构相适应的表示法。

由于二叉树的每个结点或者是叶结点,或者有两棵子树(可能为空),我们可以利用这一点来隐式地实现树结构。最直接的顺序表示法是把树的结点值按照先根周游的顺序依次列出。但是其中并没有足够的信息来确定叶结点。可以把所有的非空结点均视为有两棵空子树的分支结点。只有空的指针值 `null` 时才被当作叶结点。这样的结点表提供足够的信息来记录树结构。对于图 6.16 中的二叉树,结点表如下(假定“/”代表空指针):

$$AB/D//CEG///FH//I// \quad (6.1)$$

为了恢复原来的树结构,我们从设置  $A$  为根结点开始。 $A$  的左子结点为  $B$ ,  $B$  的左子树为空,因此  $D$  一定是  $B$  的右子结点。 $D$  有两个空子结点,因此  $C$  一定是  $A$  的右子结点。

为了说明使用树的顺序表示法进行处理所遇到的困难,可以考虑查找根结点右子结点的问题。我们首先必须顺序地沿着结点表查找完整的左子树,只有这样,才能找到根结点的右子结点的值。很显然,顺序表示法在空间上是高效率的,但在时间上效率并不高。

假定每个结点的值所需要的存储空间均为常量(例如,如果结点的值为正整数,而空指针用零值表示)。由 5.1.1 小节的满二叉树定理可以知道结点表的大小是结点数目的两倍。额外的空间被空指针占用了。我们应该能够更紧凑地存储结点表,并且任何顺序表示法都要求在到达叶结点时能够识别出来(叶结点意味着子树的结束)。实现这个目标的途径之一就是显式地在每个结点后面标识出它是叶结点还是分支结点。如果  $X$  是分支结点,我们就知道在结点表紧随其后的是  $X$  的两个子结点(有可能为子树)。如果  $X$  为叶结点,则在结点表中紧随其后的是  $X$  的某个祖先的右子结点,而不是  $X$  的右子结点。具体地说,下一个结点是  $X$  的祖先结点中,尚未出现右子结点的距  $X$  最近的那个祖先结点的右子结点。然而,这仍需假定每个分支结点确实有两个子结点。换句话说,树是满的。空子结点必须在子结点表中显式地表示出来。假定分支结点加标记(')而叶结点不加任何标记;分支结点的空子结点以"/"表示,而叶结点的空子结点不加表示。我们可以把图 6.16 中的树表示如下:

$$A'B'/DC'E'G/F'HI \quad (6.2)$$

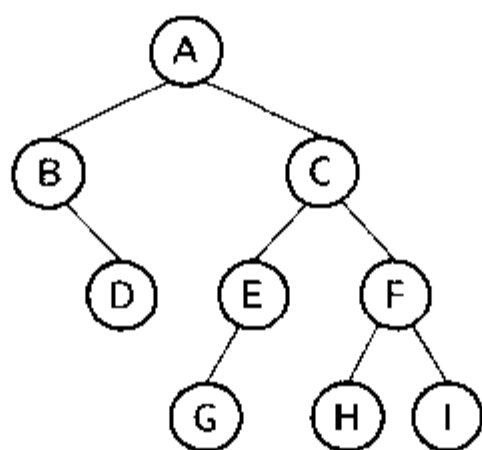


图 6.16 树的顺序表示法的二叉树示例

采用这种方法,每个结点都需要增加一位的记录域进行存储。存储  $n$  位相对于存储  $n$  个空指针来说节省了相当多的空间。注意到在这种实现中满二叉树不需要存储空指针,所以需要的开销更少。

用顺序表示法存储一般的树需要在结点表中包含更多的显式结构信息。不仅要给出一个结点是叶结点还是分支结点,还必须给出有多少个子结点的信息。作为一种替代方法,也可以给出一个结点的子结点表结束的位置。下面的一个例子中对分支结点与叶结点没有设置标志。相反,它使用特殊标记“)”来标明子结点表的结束。所有叶结点后面都跟着一个“)”,因为它们没有子结点。如果一个叶结点是其父结点的最后一个子结点,则其后将有两个连续的“)”。对于图 6.3 中的树,结点表为:

$$RAC)D)E))BF))) \quad (6.3)$$

请注意  $F$  后面跟着连续三个“)”标记,这是因为它既是叶结点,也是  $B$  的最右子树的最后结点,同时又是  $R$  的最右子树的最后结点。



## 6.6 深入学习导读

6.2 节中的表达式  $\log^* n$  与阿克曼(Ackeman)函数的逆函数密切相关。若要进一步了解有关阿克曼函数以及路径压缩开销,请参看 Robert E. Tarjan 的论文“On the efficiency of a good but not linear set merging algorithm”[Tar75]。Galil 与 Italiano 所写的文章“Data Structures and Algorithms for Disjoint Set Union Problems”[GI91]中涵盖了等价类问题的许多方面。

Hanan Samet 所著的《Application of Spatial Data Structures》[Sam90a]在 K 叉树的背景下将各种树的实现方法进行了详细论述。Samet 也像本章及第 5 章一样将顺序表示法、数组表示法与链接表示法均收在书中。虽然这些书与空间数据结构相关,但是其中涉及的许多概念对于任何需要实现树结构的应用都是适合的。

## 6.7 习题

- 6.1 编写出一个算法来判断两棵树是否相同。分析算法的运行时间代价。
- 6.2 编写出一个算法来判断两棵树在不考虑子树顺序的前提下是否相等。
- 6.3 编写出树的后根周游函数,它与 6.1.2 小节的先根周游函数 preorder 是相似的。
- 6.4 编写出一个函数,以一棵树为输入,返回树的结点数目。要求使用 6.2 节的 GTNode 接口。
- 6.5 描述如何有效实现重量权衡合并规则。具体描述出每个结点必须存储哪些信息,在两棵树合并时信息如何更新。
- 6.6 使用重量权衡合并规则与路径压缩,对下列从 0 到 15 之间的数的等价对进行归并,并给出所得树的父指针表示法的数组表示。在初始情况下,集合中的每个元素分别在独立的等价类中。当两棵树规模同样大时,使结点值较大的根结点作为值较小的根结点的子结点。  
(0,2) (1,2) (3,4) (3,1) (3,5) (9,11) (12,14) (3,9)  
(4,14) (6,7) (8,10) (8,7) (7,0) (10,15) (10,13)
- 6.7 给出一系列等价关系以使得一个有 16 个数据项的集合在同时使用重量权衡合并规则与路径压缩时得到高度为 5 的树。进行这一系列操作需要变动的父指针总数是多少?
- 6.8 分析“子结点表”表示法,“左子结点/右兄弟结点”表示法以及 6.3.3 小节中的两种链接实现的结构性开销比率。相比之下它们各自的空间效率如何?
- 6.9 使用 6.2 节的树 ADT,编写出一个函数,它以一棵树的根为输入,返回一棵用图 6.14 所示的转化过程得到的二叉树。
- 6.10 利用 5.3.3 小节的完全树顺序表示法存储 K 叉树,请导出非空完全 K 叉树结点之间关系的公式。
- 6.11 对于具有下面的空间要求的满 K 叉树实现方法,计算其结构性空间开销比率。  
(a)所有的结点均存储数据、K 个子结点指针和一个父指针。数据域需要 4 个字节,每个指针需要 4 个字节。

- (b)叶结点与分支结点均存储数据及  $K$  个子结点指针。数据域需要 16 个字节,每个指针需要 4 个字节。
- (c)叶结点与分支结点均存储数据。所有的结点都存储父指针。并且分支结点存储  $K$  个子结点指针。数据域需要 8 个字节,每个指针需要 4 个字节。
- (d)只有叶结点存储数据,只有分支结点存储  $K$  个子结点指针。数据域需要 4 个字节,每个指针需要 2 个字节。
- 6.12 使用数学归纳法证明非空满  $K$  叉树的叶结点数目为  $(K-1)n+1$ , 其中  $n$  为分支结点数目。
- 6.13 (a)使用(6.1)式的编码方法写出对图 6.17 中树的线性实现。  
(b)使用(6.2)式的编码方法写出对图 6.17 中树的线性实现。

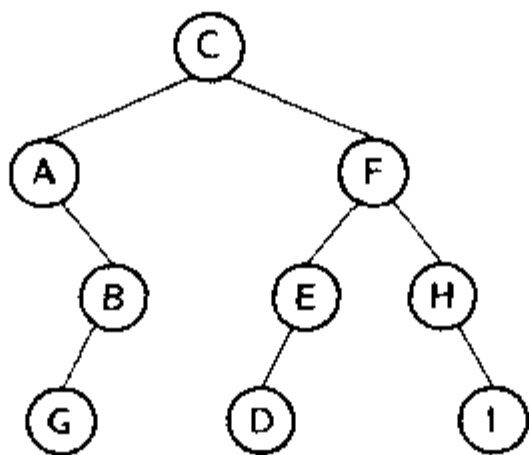


图 6.17 习题 6.13 所用的例子

- 6.14 对下列用(6.3)式编码方法写出的树的顺序表示,画出树的形状。  
XPC)Q)RV)M)))))
- 6.15 (a)编写一个函数,对公式(6.1)的顺序表示方法所表示的二叉树进行解码。输入为结点表,输出为指向所生成二叉树根结点的指针。  
(b)编写一个函数,对公式(6.2)的顺序表示方法所表示的满二叉树进行解码。输入为结点表,输出为指向二叉树的根结点的指针。  
(c)编写一个函数,对公式(6.3)的顺序表示方法所表示的树进行解码。输入为结点表,输出为指向生成树的根结点的指针。
- 6.16 给出 Huffman 树的一个顺序表示,以适用于作为文件压缩实用程序的一部分而使用。(见项目设计 5.1)

## 6.8 项目设计

- 6.1 使用 6.3.4 小节中的动态“左子结点/右兄弟结点”表示法,实现图 6.2 中的树接口。
- 6.2 使用图 6.12 带子结点指针数组的树的链接表示法,实现图 6.2 中的树接口。此实现应该只支持固定大小的结点:它们一旦生成,子结点的数目便不会再发生变化。然后,用如图 6.13 的子结点指针链接表示法重新实现这些类。比较而言,这两种实现在时间、空间效率以及程序编写容易程度方面各如何?
- 6.3 使用图 6.12 带子结点指针数组的树的链接表示法,实现图 6.2 中树的接口。此实

现必须能支持结点的子结点数目的变化。在生成时,结点应该只被分配足够的存储其子结点集的空间。当一个新的子结点添加到结点上以致于数组溢出时,则从可利用空间中分配一个为原来数组两倍大的数组。

- 6.4 实现一个 BST(二叉检索树)的文件档案库。你的程序应该在主存中产生一棵 BST,它使用的是图 5.21 中的实现方法,然后将它用 6.5 节中某一种顺序表示法写入磁盘。它也应该能够读出使用你的顺序表示法存储的磁盘文件,并在主存中产生相应的等价 BST。
- 6.5 使用 UNION/FIND 算法解决下面的问题:用点的  $(x, y)$  坐标给出一个点集,把点分配到集群(cluster)中去。两个距离不超过  $d$  的点定义为在一个集群中。集群是等价关系,此为这个问题的目的(练习等价类的归并)。换句话说,点 A、B 和 C 定义为在一个集群中,如果 A 与 B 距离小于  $d$ ,且 A 与 C 距离小于  $d$ ,即使 B 与 C 距离大于  $d$  也是成立的。为了解决这个问题,计算每一对点之间的距离,当出现两个点在特定距离内时,使用等价处理算法来归并集群。这个算法的渐近复杂度如何?处理的瓶颈在何处?

## 第7章 图

本章介绍图这一数据结构。一个图由两部分组成,一部分是结点,在图的术语中也称为顶点(vertex);另一部分是顶点的偶对,称为边(edge)。通常图的任意一对顶点间都允许有一条边。树和链表也可以看作受限图,因此,从某种意义上说,图是最基本的数据结构。

图被广泛应用于模拟真实事件或抽象问题中,并被在成百上千个应用问题中经常运用。下面列出图常用来解决的几类主要问题:

1. 模拟计算机与通信网络的联接。
2. 表示一张地图的一组坐标以及坐标之间的距离,以求得最短路径。
3. 模拟交通网络的流量。
4. 寻找从开始状态到目标状态的路径,如人工智能问题求解。
5. 模拟计算机算法,显示程序状态的变化。
6. 为复杂活动各子任务的完成寻找较优顺序,如大型建筑的建造。
7. 模拟家族、商业活动或军事组织和自然科学中动植物分类中的各种关系。

本章将给出一些解决上述问题的图算法的例子。各节内容安排如下:7.1节首先介绍一些图的基本术语,然后定义图的两种基本表示方法:相邻矩阵和邻接表;7.2节介绍一个基于相邻矩阵和邻接表的图的抽象数据类型(ADT)及其简单实现;7.3节介绍两种最常用的图的周游算法:深度优先搜索和广度优先搜索,以及它们在拓扑排序中的应用;7.4节介绍解决寻找图的最短路径问题的算法;最后,7.5节介绍寻找最小支撑树的算法,这对于确定网络联接的最小代价很有价值。除了本身的实用性和趣味性以外,上述算法使用了前几章介绍过的大部分数据结构。

### 7.1 术语和表示法

图可用  $G = (V, E)$  来表示,每个图都包括一个顶点集合(记为  $V$ )和一个边集合(记为  $E$ ),其中  $E$  中每条边都是  $V$  中某一对顶点的连接。顶点总数记为  $|V|$ ,边的总数记为  $|E|$ , $|E|$  的取值范围是 0 到  $\Theta(|V|^2)$ 。边数较少的图称为稀疏图(sparse graph),边数较多的图称为密集图(dense graph),包括所有可能边的图称为完全图(complete graph)。

如果图的边限定为从一个顶点指向另一个顶点,则称此图为有向图(directed graph 或 digraph)。如果图中的边没有方向性,则称为无向图(undirected graph)。如果图中各顶点均带有标号,则称为标号图(labeled graph)。一条边所连接的两个顶点是相邻的(adjacent),称为邻接点(neighbors)。连接一对邻接点  $u$ 、 $v$  的边称为与顶点  $u$ 、 $v$  相关联(incident)的边,记作  $(u, v)$ 。每条边都可能附有其值或权(weight)。边上标有权的图称为带权图(weighted graph)。

如果从  $v_i$  到  $v_{i+1}$  ( $1 \leq i \leq n$ ) 的边均存在,则称顶点序列  $v_1, v_2, \dots, v_n$  构成一条长度为  $n-1$  的路径(path)。如果路径上各顶点均不同,则称此路径为简单路径(simple path)。路径长

度(length)是指路径包含的边数。如果一条路径将某个顶点(如  $v_i$ ) 连接到它本身,且其长度大于等于 3,则称此路径为回路(cycle)。如果构成回路的路径是简单路径,特别当首尾两顶点不相同时,称此回路为简单回路(simple cycle)。图 7.1 给出了上面定义的图的术语图解。

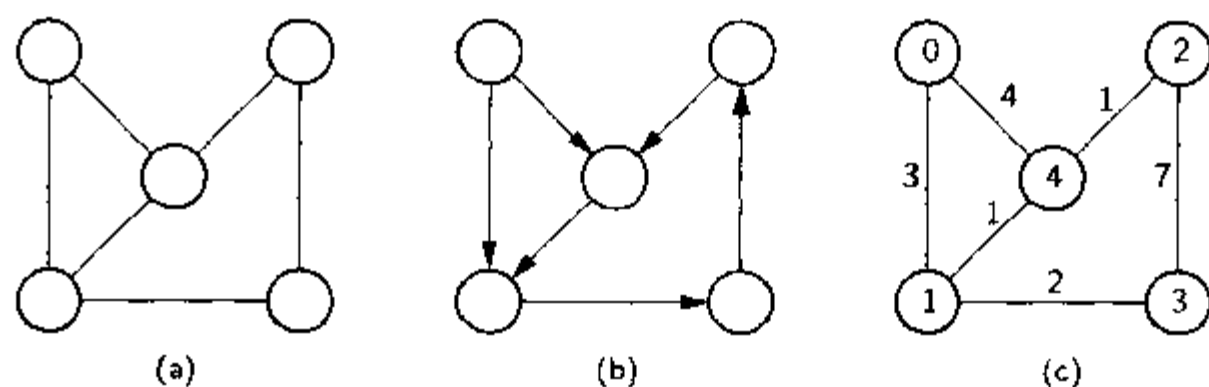


图 7.1 图及其术语示例

(a) 一个图。(b) 一个有向图(digraph)。(c) 一个有向标号图,且边上标有权。这个例子中从顶点 0 到顶点 3 存在一条包括顶点 0、顶点 1 和顶点 3 的简单路径。顶点 0, 1, 3, 2, 4 再到顶点 1 也构成一条路径,但不是简单路径。顶点 1, 3, 2, 4 再到顶点 1 构成一条简单回路

子图(subgraph)  $S$  是指由图  $G$  中选出其顶点集的一个子集  $V_S$  以及与  $V_S$  中顶点相关联的一些边所构成的图。

如果一个无向图中任意一个顶点到其他任意顶点都至少存在一条路径,则称此无向图为连通的(connected)。无向图的最大连通子图称为连通分量(connected component)。图 7.2 给出了一个有三个连通分量的无向图示例。

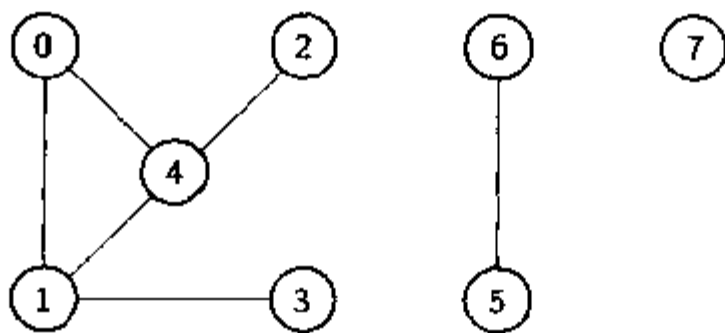


图 7.2 有三个连通分量的无向图。顶点 0, 1, 2, 3 和顶点 4 构成一个连通分量。顶点 5 和 6 构成第二个连通分量。顶点 7 单独构成第三个连通分量

不带回路的图称为无环图(acyclic graph)。不带回路的有向图则称为有向无环图(directed acyclic graph 或 DAG)。

一棵自由树(free tree)就是一个不带简单回路的无向图,它是连通的,且有  $|V| - 1$  条边。

图有两种常用的表示方法。图 7.3(b) 是相邻矩阵(adjacency matrix)表示法的图示。图的相邻矩阵是一个  $|V| \times |V|$  矩阵。假设  $|V| = n$ , 各顶点依次记为  $v_0, v_1, \dots, v_{n-1}$ , 则相邻矩阵的第  $i$  行包括所有以  $v_i$  为起点的边。如果从  $v_i$  到  $v_j$  存在一条边,则第  $i$  行的第  $j$  个元素标记为 1, 否则不作标记。因此相邻矩阵的每个元素需要占用一个字节。但是如果我们希望用数字来标记每条边,例如构成边的两个顶点间的距离,则矩阵的每个元素必须占足够大的空间来存储。不管哪种情况,相邻矩阵的空间代价为  $\Theta(|V|^2)$ 。

图的第二种常见表示法为邻接表(adjacency list)。如图 7.3(c) 所示,邻接表是一个以链表为元素的数组。这个数组包含  $|V|$  个元素,其中第  $i$  个元素存储的是指向顶点  $v_i$  的边表的

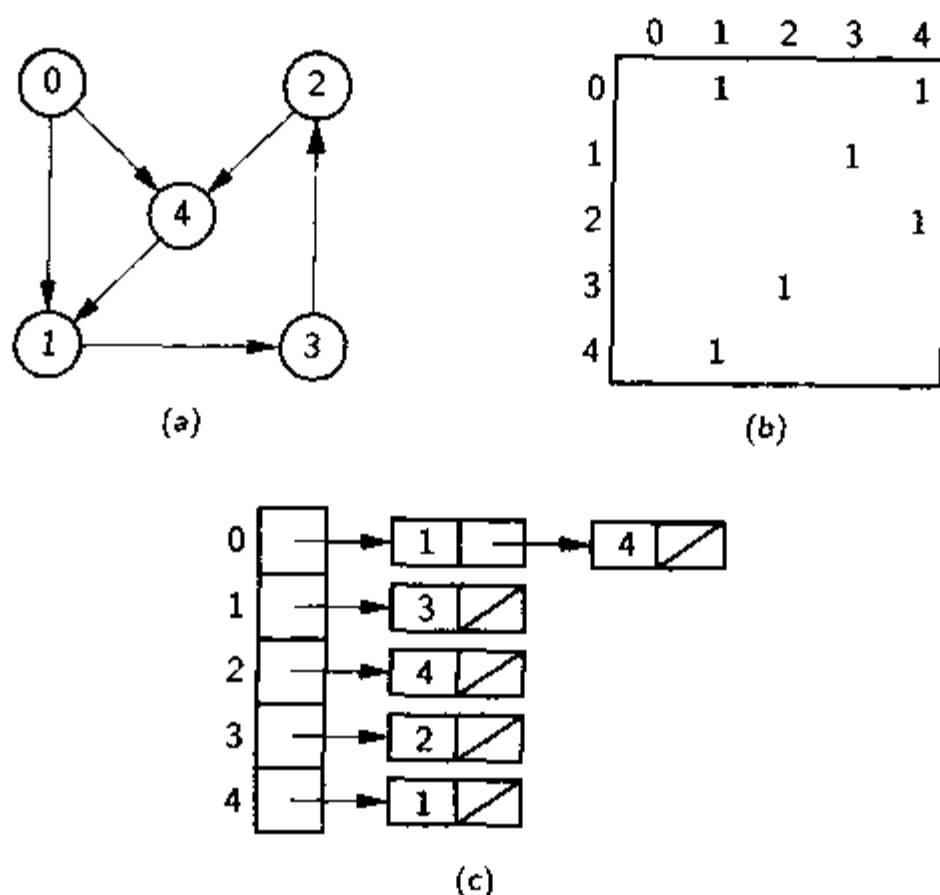


图 7.3 两种图的表示方法

(a)一个有向图。(b)为(a)图的相邻矩阵。(c)为(a)图的邻接表

指针。前述边表是由顶点  $v_i$  的邻接点构成的链表。因此,邻接表可说是 6.3.1 小节介绍的树的“子结点表”表示法的推广。

邻接表的空间代价与图的边及顶点数目均有关。每个顶点都要占据一个数组元素的位置(即使该顶点没有邻接点,此时该顶点的边表中没有元素),且每条边必须出现在其中某个顶点的边表中。所以,邻接表的空间代价为  $\Theta(|V| + |E|)$ 。

相邻矩阵和邻接表均可用于存储有向图或无向图。无向图中连接某两个顶点  $u$  和  $v$  的边可用两条有向边来代替:一条从  $u$  到  $v$ ,一条从  $v$  到  $u$ 。图 7.4 描述了无向图相邻矩阵和邻接表的用法。

哪种表示法的存储效率更高取决于图中边的数目。邻接表仅存储实际出现在图中的边信息,而相邻矩阵则需要存储所有可能的边。但是相邻矩阵不需要指针的结构性开销,而这可能是一个巨大的开销,特别是在仅需要一个比特位来存储边信息以表明边的存在时更是这样。图越密集,相邻矩阵的空间效率相应就越高。对稀疏图使用邻接表表示法,也可获得较高的空间效率。

与邻接表相比,相邻矩阵在图的算法中常常导致相对较高的时间代价。其原因是:访问某个顶点所有邻接点的图算法相当普遍,如果使用邻接表,则只需检查连接此顶点与它相邻顶点的实际存在的边;而使用相邻矩阵,则必须查看它的所有  $|V|$  条可能的边。例如,对一个稀疏图,如果算法使用邻接表,则仅需查找  $\Theta(|V| + |E|)$  次;而使用相邻矩阵却需要查找  $\Theta(|V|^2)$  次。

## 7.2 图的实现

接着我们来讨论怎样实现一个 Graph 类的问题。顶点和边可以用不同的类来实现,但是

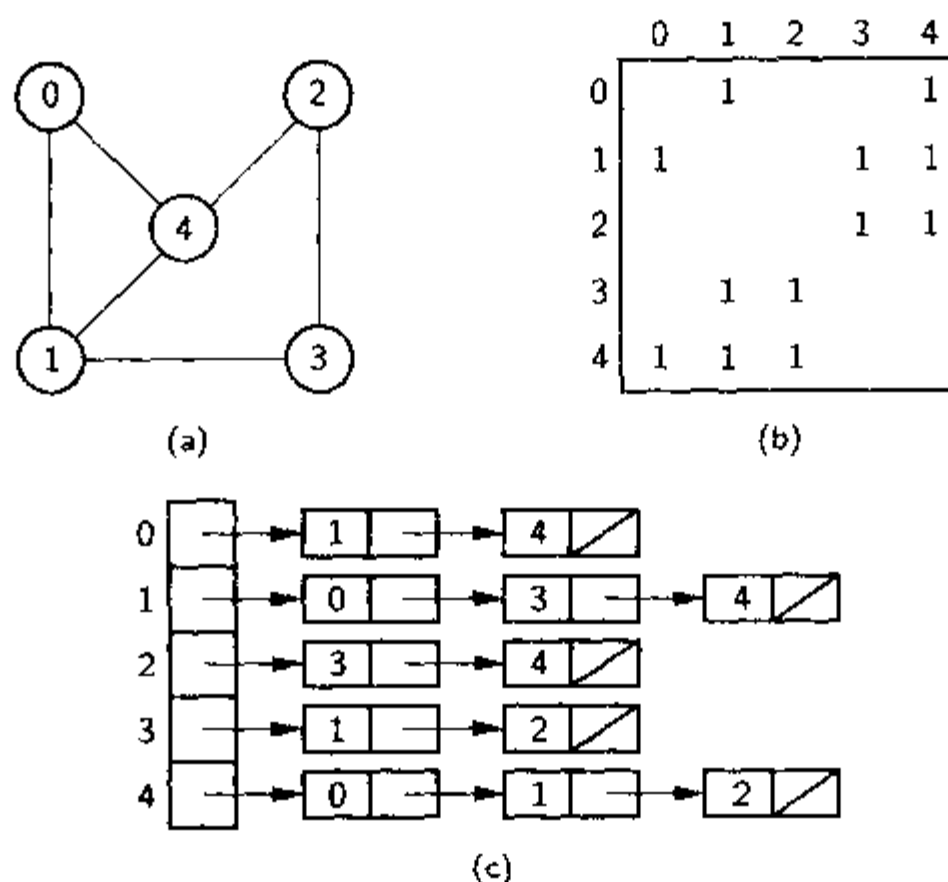


图 7.4 无向图的表示

(a)一个有向图。(b)为(a)图的相邻矩阵。(c)为(a)图的邻接表

为简单有效起见,下面将不使用这种实现方法。图 7.5 给出了一个图的接口,并列出了其主要函数。不妨设一个顶点数为  $n$  的图,其图中顶点标号值为从 0 到  $n-1$ 。

```

interface Graph {
    public int n();           // Graph class ADT
    public int e();           // Number of vertices
    public Edge first(int v); // Number of edges
    public Edge next(Edge w); // Get first edge for vertex
    public boolean isEdge(Edge w); // Get next edge for a vertex
    public boolean isEdge(int i, int j); // True if this is an edge
    public int v1(Edge w); // True if this is an edge
    public int v2(Edge w); // Where edge came from
    public void setEdge(int i, int j, int weight); // Where edge goes to
    public void setEdge(Edge w, int weight); // Set edge weight
    public void delEdge(Edge w); // Delete edge w
    public void delEdge(int i, int j); // Delete edge (i, j)
    public int weight(int i, int j); // Return weight of edge
    public int weight(Edge w); // Return weight of edge
    public void setMark(int v, int val); // Set Mark for v
    public int getMark(int v); // Get Mark for v
} // interface Graph

```

图 7.5 一个图的 ADT(抽象数据类型)

Graph 类具有返回顶点数和边数的函数(分别为函数  $n$  和  $e$ )。因为图的算法中一个基本的操作就是从--一个给定的顶点出发,顺序访问--一系列相连的顶点,所以给出三个与访问线性表

的函数工作方式类似的函数。其中 `first` 函数返回与给定顶点关联的第一条边。`next` 函数返回此顶点的下一条与之关联的边。`isEdge` 函数用来判断给定边在图中是否存在。这可以用来判断边表是否已经处理完毕,因为 `next` 函数在到达最后一条边时将返回 `null` 值。任意给定一条边,函数 `v1` 和 `v2` 分别返回它的起点和终点。需要注意的是,这里给出了两个 `weight` 函数,第一个返回给定边的权,第二个返回给定起点和终点的边的权。如果这样的边不存在,权就定义为 `INFINITY`(无穷大)。函数 `setEdge` 把一条边添加到图中,函数 `delEdge` 从图中删除一条边。这两个函数分别有两种版本,一种用 `Edge` 边对象作为参数,另一种用起点和终点作为参数。假定边的权不为零。函数 `getMark` 得到 `Mark` 数组(下面将会介绍 `Mark` 数组)的值,而函数 `setMark` 给 `Mark` 数组赋值。

上述图的 ADT 函数中的参数用索引值来表示顶点。换句话说,顶点分别为顶点 0、顶点 1,依此类推。毫无疑问,实际应用中可以用适当的信息来描述顶点,例如名字或与应用相关的值。

图的 ADT 把边看作特殊的对象。边可以有其特殊信息,图的实现可以利用这些信息。图的应用尤其需要得到边所涉及到的两个顶点的信息。下面是边的接口。

```
interface Edge {
    // Interface for graph edges
    public int v1(); // Return the vertex it comes from
    public int v2(); // Return the vertex it goes to
} // interface Edge
```

给出了上述图和边的 ADT 定义,可以很容易地用相邻矩阵或邻接表来实现。这些实现并没有考虑图究竟如何生成的问题。为此,在具体应用中,图程序员必须给出一些附加函数,比如从某个文件中读出图的描述。可以利用函数 `setEdge` 来创建图。

图 7.6 是相邻矩阵表示法的图实现。其中 `Mark` 数组在本节将要介绍的图算法中用来判断一个给定顶点是否已被访问过。函数 `getMark` 和 `setMark` 访问和改变 `Mark` 数组的值。对  $n$  个顶点的图,用一个  $n \times n$  的整型数组 `matrix` 来实现表示它的边矩阵。矩阵位置  $(i, j)$  的元素存储实际存在的对应边的权。如果  $(i, j)$  没有对应的边,则该位置的矩阵元素值为 0。类 `Edgem` 实现了 `Edge` 接口,它存储了边所连接的两个顶点。

```
// Edge class for Adjacency Matrix graph representation
class Edgem implements Edge {
    private int vert1, vert2; // The vertex indices

    public Edgem(int vt1, int vt2) { vert1 = vt1; vert2 = vt2; }
    public int v1() { return vert1; }
    public int v2() { return vert2; }
} // class Edgem

class Graphm implements Graph { // Graph: Adjacency matrix
    private int[][] matrix; // The edge matrix
    private int numEdge; // Number of edges
    public int[] Mark; // The mark array
```



---

```

public Graphm(int n) { // Constructor
    Mark = new int[n];
    matrix = new int[n][n];
    numEdge = 0;
}

public int n() { return Mark.length; } // Number of vertices

public int e() { return numEdge; } // Number of edges

public Edge first(int v) { // Get the first edge for a vertex
    for (int i = 0; i < Mark.length; i++)
        if (matrix[v][i] != 0)
            return new Edgem(v, i);
    return null; // No edge for this vertex
}

public Edge next(Edge w) { // Get next edge for a vertex
    if (w == null) return null;
    for (int i = w.v2() + 1; i < Mark.length; i++)
        if (matrix[w.v1()][i] != 0)
            return new Edgem(w.v1(), i);
    return null; // No next edge;
}

public boolean isEdge(Edge w) { // True if this is an edge
    if (w == null) return false;
    else return matrix[w.v1()][w.v2()] != 0;
}

public boolean isEdge(int i, int j) // True if this is an edge
    { return matrix[i][j] != 0; }

public int v1(Edge w) { return w.v1(); } // Where edge comes from

public int v2(Edge w) { return w.v2(); } // Where edge goes to

public void setEdge(int i, int j, int wt) { // Set edge weight
    Assert.notFalse(wt != 0, "Cannot set weight to 0");
    matrix[i][j] = wt;
    numEdge++;
}

public void setEdge(Edge w, int weight) // Set edge weight

```

```

    } if (w != null) setEdge(w.v1(), w.v2(), weight); }

public void delEdge(Edge w) {          // Delete edge w
    if (w != null)
        if (matrix[w.v1()][w.v2()] != 0) {
            matrix[w.v1()][w.v2()] = 0;
            numEdge--;
        }
}

public void delEdge(int i, int j) { // Delete edge (i, j)
    if (matrix[i][j] != 0) {
        matrix[i][j] = 0;
        numEdge--;
    }
}

public int weight(int i, int j) { // Return weight of edge
    if (matrix[i][j] == 0) return Integer.MAX_VALUE;
    else return matrix[i][j];
}

public int weight(Edge w) {          // Return weight of edge
    Assert.notNull(w, "Can't take weight of null edge");
    if (matrix[w.v1()][w.v2()] == 0) return Integer.MAX_VALUE;
    else return matrix[w.v1()][w.v2()];
}

public void setMark(int v, int val) { Mark[v] = val; } // Set Mark

public int getMark(int v) { return Mark[v]; }          // Get Mark
} // class Graphm

```

图 7.6 边和图的相邻矩阵实现

给定顶点  $v$ , `first` 函数返回与它相关联的第一条边(如果存在)在 `matrix` 中对应元素的地址。其方法是从  $(v, 0)$  开始顺序扫描第  $v$  行,直到找到一条边。如果没有边与  $v$  相关联, `first` 函数就返回 `null` 值。`next` 函数从顶点  $i$  所在的那一行的第  $j + 1$  列开始,不断地向右扫描第  $i$  行,直至寻找到边  $(i, j)$  的下一条边(如果存在)。`isEdge` 函数用来判断是否已到达边表的末尾。函数 `setEdge` 把一条边添加到图中,函数 `delEdge` 从图中删除一条边。`weight` 函数返回给定边的权值。

图 7.7 是邻接表表示法的边类 `Edge1` 和图类 `Graph1` 的实现。`Graph1` 类利用了第 4 章标准链表类 `LList` 的扩展定义 `GraphList`。扩展功能可以访问链表对象内当前指针 `curr`,这有利于

Edge 对象在邻接表中直接访问相关边。否则,通过 Edge 对象查找给定边的相应邻接顶点,需要在顶点邻接表中从表头开始逐个查找。下面是扩展图 4.5 链表类 LList 定义的 GraphList 实现。

```
class GraphList extends LList {
    public Link currLink() { return curr; }
    public void setCurr(Link who) { curr = who; }
} // class GraphList
```

Edge 类存储它关联到的两个顶点和它在邻接表中的相应边。实现了 isEdge 后,Graph 的其他成员函数就相对容易实现了。isEdge 必须判断给定的一条边是否存在于图中。边可以由 Edge 对象或两个顶点来表示。请注意下面的副作用:由于 isEdge 调用了函数 setCurr,该边所属顶点邻接表中当前指针 curr 会有所改变。因此,像 setEdge 这样的函数先要验证边是否存在。若存在,则在相应的顶点邻接表中修改当前边的值,否则在相应位置插入该边。其他函数实现方法类似。有两个顶点 v1 和 v2 作为输入参数的 isEdge 相当直观,沿着 v1 的邻接表一直找到 v2 为止。以 Edge 对象为输入参数的 isEdge 要验证存储在 Edge 对象内的邻接表中边结点是否与该边的两个顶点相匹配。

```
// Edge class for Adjacency List graph representation
class Edgel implements Edge {
    private int vert1, vert2; // Indices of v1, v2
    private Link itself; // Pointer to node in the adjacency list

    public Edgel(int vt1, int vt2, Link it) // Constructor
    { vert1 = vt1; vert2 = vt2; itself = it; }

    public int v1() { return vert1; }
    public int v2() { return vert2; }
    Link theLink() { return itself; } // Access into adjacency list
} // class Edgel
```

```
class Graph1 implements Graph { // Graph: Adjacency list
    private GraphList[] vertex; // The vertex list
    private int numEdge; // Number of edges
    public int[] Mark; // The mark array

    public Graph1(int n) // Constructor
    {
        Mark = new int[n];
        vertex = new GraphList[n];
        for (int i = 0; i < n; i++)
            vertex[i] = new GraphList();
        numEdge = 0;
    }
}
```

```

public int n() { return Mark.length; } // Number of vertices

public int e() { return numEdge; } // Number of edges

public Edge first(int v) { // Get the first edge for a vertex
    vertex[v].setFirst();
    if (vertex[v].currValue() == null) return null;
    return new Edge(v, ((int[])vertex[v].currValue())[0],
                    vertex[v].currLink());
}

public boolean isEdge(Edge e) { // True if this is an edge
    if (e == null) return false;
    vertex[e.v1()].setCurr(((Edge)e).theLink());
    if (! vertex[e.v1()].isInList()) return false;
    return (((int[])vertex[e.v1()].currValue())[0] == e.v2());
}

public int v1(Edge e) { return e.v1(); } // Where edge comes from

public int v2(Edge e) { return e.v2(); } // Where edge goes to
public boolean isEdge(int i, int j) { // True if this is an edge
    GraphList temp = vertex[i];
    for (temp.setFirst(); ((temp.currValue() != null) &&
        (((int[])temp.currValue())[0] < j)); temp.next());
    return (temp.currValue() != null) &&
        (((int[])temp.currValue())[0] == j);
}

public Edge next(Edge e) { // Get next edge for a vertex
    vertex[e.v1()].setCurr(((Edge)e).theLink());
    vertex[e.v1()].next();
    if (vertex[e.v1()].currValue() == null) return null;
    return new Edge(e.v1(), ((int[])vertex[e.v1()].currValue())[0],
                    vertex[e.v1()].currLink());
}

public void setEdge(int i, int j, int weight) { // Set edge weight
    Assert.notFalse(weight != 0, "Cannot set weight to 0");
    int[] currEdge = { j, weight };
    if (isEdge(i, j)) // Edge already exists in graph
        vertex[i].setValue(currEdge);
    else // Add new edge to graph

```

```

        vertex[i].insert(currEdge);
        numEdge++;
    }
}

public void setEdge(Edge w, int weight) // Set edge weight
{ if (w != null) setEdge(w.v1(), w.v2(), weight); }

public void delEdge(int i, int j) // Delete edge
{ if (isEdge(i, j)) { vertex[i].remove(); numEdge--; } }

public void delEdge(Edge w) // Delete edge
{ if (w != null) delEdge(w.v1(), w.v2()); }

public int weight(int i, int j) { // Return weight of edge
    if (isEdge(i, j)) return ((int[])vertex[i].currValue())[1];
    else return Integer.MAX_VALUE;
}

public int weight(Edge e) { // Return weight of edge
    if (isEdge(e)) return ((int[])vertex[e.v1()].currValue())[1];
    else return Integer.MAX_VALUE;
}

public void setMark(int v, int val) { Mark[v] = val; } // Set Mark

public int getMark(int v) { return Mark[v]; } // Get Mark
} // class Graph1

```

图 7.7 边和图的邻接表实现

### 7.3 图的周游

一般说来,基于图的拓扑结构,以特定的顺序依次访问图中各顶点是很有用的,这被称为图的周游(graph traversal),从概念上讲与树的周游类似。让我们回顾一下,树的周游是指以某种特定的顺序,如先根周游或后根周游,对每个结点恰好访问一次。标准的图的周游也存在类似的顺序,而且每种顺序适用于解决某些特定的问题。特别是,人工智能程序设计中的许多问题就使用图来建模。例如领土问题:假设有许多国家,有些国家两两相互连接,要求从某个指定的起始国出发,通过国家之间的连接,从一国到另一国,最后到达另一个指定的终点国。通常情况下,起始国和终点国之间并不直接相连。

图的周游算法典型地是从一个起点出发,试探性地访问其余顶点。它还必须处理若干棘

手的问题。首先,从起点出发可能到达不了所有其他顶点,非连通图就可能发生这种情况。其次,有些图存在回路,我们必须确定算法不会因回路而陷入死循环。

为了避免发生上述两种情况,图的周游算法通常为图的每个顶点保留一个标志位(mark bit)。算法开始时,所有顶点的标志位清零。周游过程中,当某个顶点被访问时,其标志位就被标记。如果在周游中遇到被标记过的顶点,则不再访问它。这样就可以避免程序遇到回路时陷入无限循环。

周游算法一结束,就可以通过检查标志位数组来查看是否已处理了所有顶点。如果还有顶点未被标记,可以从某个未被标记的顶点开始继续周游。注意:这个处理过程与图是有向的还是无向的没有关系。为了保证访问到所有的顶点,图 G 的周游函数 graphTraverse 可以这样实现:

```
void graphTraverse(Graph G) {
    for (v = 0; v < G.n(); v++)
        G.setMark(v, UNVISITED); // Initialize mark bits
    for (v = 0; v < G.n(); v++)
        If (G.getMark(v) == UNVISITED)
            doTraverse(G, v)
}
```

其中“doTraverse”函数可用下几小节将介绍的任何一种图的周游方式代替。

### 7.3.1 深度优先搜索

第一种系统的图周游方式称为深度优先搜索(depth-first search 或 DFS)。在周游过程中,某个顶点 v 被访问后,递归地访问它的所有尚未被访问的相邻顶点。另一种方案是,先访问 v,把所有与 v 相关联的边存入栈;弹出栈顶元素,栈顶元素代表的边所关联到的另一个顶点就是要访问的下一个元素,对该元素重复对 v 的操作;依此类推直至栈中所有元素都被处理完毕。其结果是沿着图的某一分支搜索直到它的末端,然后回溯,沿着另一分支搜索,依此类推。深度优先搜索将产生一棵深度优先搜索树(depth-first search tree)。这棵树由周游过程中所有连接某一新的(未被访问的)顶点的边组成,而不包括那些连接已访问顶点的边。DFS 适用于有向图和无向图。下面给出 DFS 算法的一种实现:

```
static void DFS(Graph G, int v) { // Depth first search
    PreVisit(G, v); // Take appropriate action
    G.setMark(v, VISITED);
    for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
        if (G.getMark(G.v2(w)) == UNVISITED)
            DFS(G, G.v2(w));
    PostVisit(G, v); // Take appropriate action
}
```

这个实现调用了 PreVisit 和 PostVisit 函数。它们的功能是在搜索过程中指定程序该干什么。类似于树的先根周游在子树被访问前需要对根结点进行处理,一些图的周游也要求在进入 DFS 的下一层递归前对顶点进行处理。而另一些应用问题则要在处理完剩余顶点后再处理当前顶点,这时就需要调用 PostVisit 函数。



DFS 算法对有向图的每条边都恰处理好一次。在无向图中,DFS 对每条边分别沿两个方向进行处理,且每个顶点必须被访问,所以总的时间代价为  $\Theta(|V| + |E|)$ 。

### 7.3.2 广度优先搜索

我们要介绍的第二种图周游算法是广度优先搜索(breadth-first search 或 BFS)。BFS 在更进一步访问其他顶点前,检查起点的所有邻接点。除了用队列代替了递归栈外,BFS 的实现与 DFS 相似。注意:当图是一棵树且起点为树的根结点时,BFS 将由顶至底逐层对各个结点进行访问(树的层次周游)。下面给出 BFS 算法的一种实现:

```
static void BFS(Graph G, int start) { // Breadth first search
    Queue Q = new AQueue(G.n()); // Use a Queue
    Q.enqueue(new Integer(start));
    G.setMark(start, VISITED);
    while (! Q.isEmpty()) { // Process each vertex on Q
        int v = ((Integer)Q.dequeue()).intValue();
        PreVisit(G, v); // Take appropriate action
        for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
            if (G.getMark(G.v2(w)) == UNVISITED) { // Put neighbors on Q
                G.setMark(G.v2(w), VISITED);
                Q.enqueue(new Integer(G.v2(w)));
            }
        PostVisit(G, v); // Take appropriate action
    }
}
```

图 7.10 给出一个图及其广度优先搜索树。图 7.11 给出图 7.10(a)的 BFS 算法处理过程的图示。

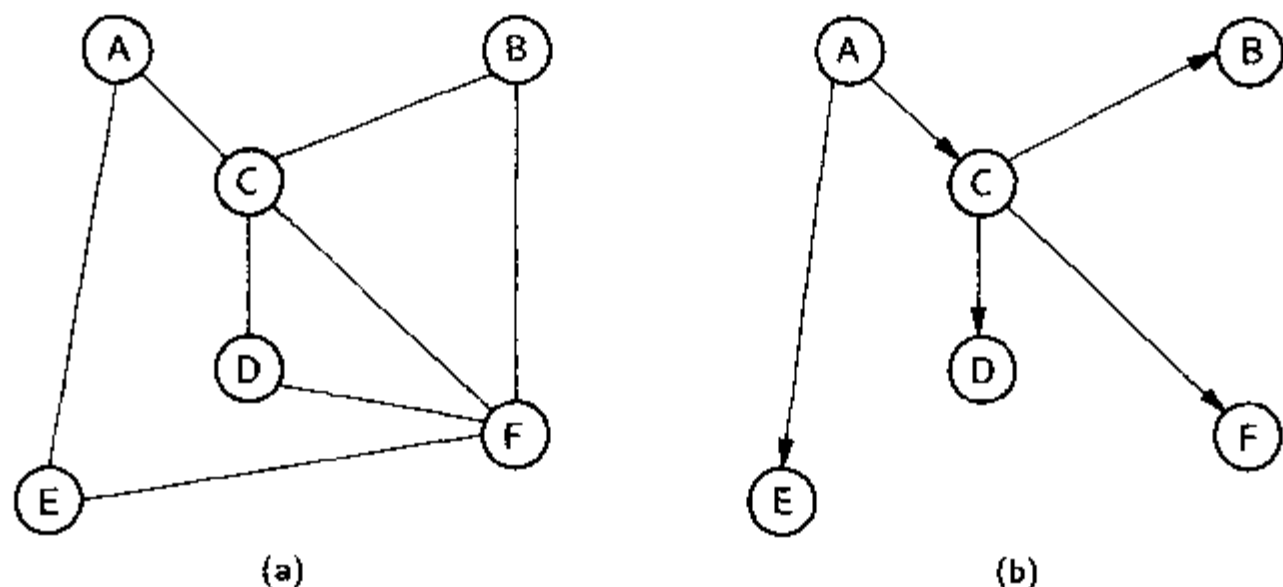


图 7.10 (a)一个图。(b)由顶点 A 开始的广度优先搜索树

### 7.3.3 拓扑排序

图的周游可用于解决先决条件问题。假设我们要安排一系列任务,如轮班或分工,而只有



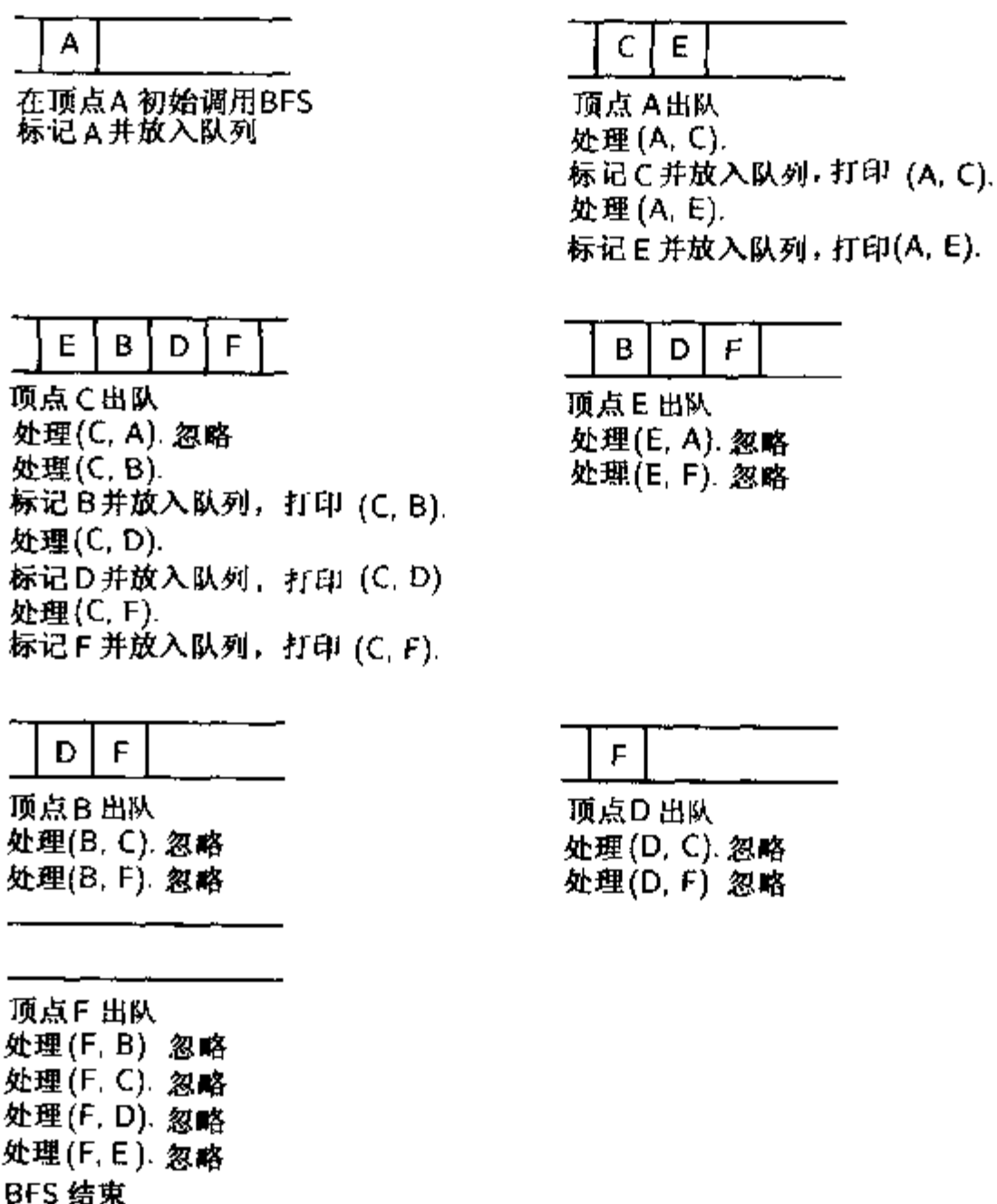


图 7.11 图 7.10(a)从顶点 A 开始的广度优先搜索过程的详细图示。描绘了引起队列变化的每个步骤

在某个任务的先决条件具备时才能着手执行这个任务。我们希望以某种线性顺序组织这些任务,以便在能够满足先决条件的情况下逐个完成各项任务。可以使用一个有向无环图(DAG)来模拟这个问题。因为任务之间存在先决条件,即顶点之间有方向性,因此图是有向的。图又需要是无回路的,因为回路中隐含了相互冲突的条件,从而使某些条件不可能在不违反任何先决条件的情况下得到实现。将一个 DAG 中所有顶点在不违反先决条件规定的基础上排成线性序列的过程称为拓扑排序(topological sort)。图 7.12 是前述问题的图示。本例的一个可行拓扑序列为 J1, J2, J3, J4, J5, J6, J7。

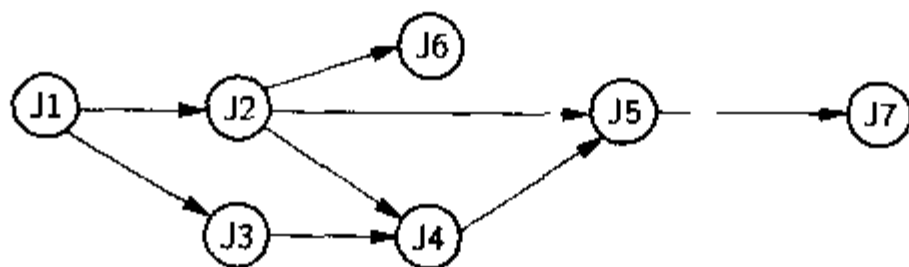


图 7.12 拓扑排序的例图。7 个任务之间的依赖关系如图所示

拓扑序列可以通过对图进行深度优先搜索来寻找:某个顶点被访问时,不进行任何处理(即 PreVisit 函数什么也不做);当递归返回到此顶点时,PostVisit 函数打印这个顶点。这就产生一个逆序的拓扑序列。序列从哪个顶点开始并不重要,因为所有顶点最终都被访问到。下面给出上述算法的实现:

```
static void topsort(Graph G) { // Topological sort: recursive
    for (int i = 0; i < G.n(); i++) // Initialize Mark array
        G.setMark(i, UNVISITED);
    for (int i = 0; i < G.n(); i++) // Process all vertices
        if (G.getMark(i) == UNVISITED)
            tophelp(G, i); // Call recursive helper function
}

static void tophelp(Graph G, int v) { // Topsort helper function
    G.setMark(v, VISITED);
    for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
        if (G.getMark(G.v2(w)) == UNVISITED)
            tophelp(G, G.v2(w));
    printout(v); // PostVisit for Vertex v
}
```

使用此算法从 J1 开始,依次访问相邻顶点,图 7.12 所示图的顶点将以 J7,J5,J4,J6,J2,J3,J1 的顺序打印出来。将其顺序颠倒之后,就得到一个合乎要求的拓扑序列 J1,J3,J2,J6,J4,J5,J7。

我们也可以使用队列代替递归来实现拓扑排序。做法如下:首先访问所有的边,计算指向每个顶点的边数(即计算每个顶点的先决条件个数),将所有无先决条件的顶点放入队列,然后开始处理队列。从队列中删除一个顶点并打印,同时将其所有相邻顶点的先决条件计数减 1。当某个相邻顶点的计数为 0 时,就将其插入队列。如果在所有顶点都被打印前队列已为空,则图中必包含回路(即不可能不违反任何先决条件为这些任务安排一个合理顺序)。使用队列对图 7.12 所示图进行拓扑排序,其顶点的打印顺序将是 J1,J2,J3,J6,J4,J5,J7。图 7.13 给出基于队列的拓扑排序算法的实现:

```
static void topsort(Graph G) { // Topological sort: Queue
    Queue Q = new AQueue(G.n());
    int[] Count = new int[G.n()];
    int v;
    for (v = 0; v < G.n(); v++) Count[v] = 0; // Initialize
    for (v = 0; v < G.n(); v++) // Process every edge
        for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
            Count[G.v2(w)]++; // Add to v2's prereq count
    for (v = 0; v < G.n(); v++) // Initialize Queue
        if (Count[v] == 0) // Vertex has no prerequisites
            Q.enqueue(new Integer(v));
}
```

```

while (! Q.isEmpty()) {          // Process the vertices
    v = ((Integer)Q.dequeue()).intValue();
    printout(v);                 // PreVisit for Vertex V
    for (Edge w = G.first(v); G.isEdge(w); w = G.next(w)) {
        Count[G.v2(w)] --;      // One less prerequisite
        if (Count[G.v2(w)] == 0) // This vertex is now free
            Q.enqueue(new Integer(G.v2(w)));
    }
}

```

图 7.13 基于队列的拓扑排序算法

## 7.4 最短路径问题

在交通地图上,城市之间的公路通常标有长度。我们可以使用边上标记数值的有向图来模拟公路网。这些数值代表公路的距离或其他类型的开销,例如旅游所用的时间;它们被称为权(weight)、开销(cost)或长度(distance),视具体应用问题而定。对于一个带权图,典型的问题就是寻找两个指定顶点之间最短路径的总长。这并不是一个简单的问题,因为最短路径不一定恰好就是连接这两个顶点的边(如果这样的边存在),而可能是一条包括一个或多个中间顶点的路径。例如,图 7.14 中,从 A 经 B 到 D 的路径长度为 15,直接从 A 到 D 的边长度为 20,从 A 到 C 再经 B 到 D 的路径长度为 10。因此,从 A 到 D 的最短路径长度为 10(但它并不是从 A 直接到 D 的边)。我们用记号  $d(A, D) = 10$  表示从 A 到 D 的最短路径长度为 10。在图 7.14 中,从 E 到 B 不存在任何路径,则令  $d(E, B) = \infty$ 。注意,  $w(D, A) = \infty$ ,因为图 7.14 所示图是有向的。我们假设所有权都是正值。

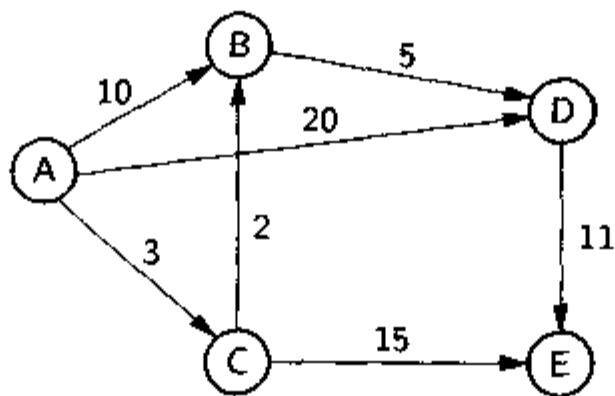


图 7.14 最短路径定义的例图

### 7.4.1 单源最短路径

我们要研究的第一个最短路径问题称为单源最短路径(single-source shortest path)问题。已知图  $G = (V, E)$ , 找出从某个给定源顶点  $s \in V$  到  $V$  中每个顶点的最短路径。也许我们只需要顶点  $s$  与  $t$  之间的最短路径,但是很可惜,同寻找到其他所有顶点的最短路径算法相比,并没有适合仅寻找到某一顶点最短路径的更好的算法(在最差情况下)。

这里所介绍的算法将仅计算从给定顶点到其他所有顶点的最短路径长度,而不记录实际路径。记录这些路径需要对算法做一些改动,在后面的习题中将让读者自己解决这个问题。

单源最短路径的一个实例是关于计算机网络的问题:怎样找到一种最廉价的方式从一台计算机向网上所有其他计算机发送一条信息。问题中的网络可以使用一个带权图来模拟,图中的权代表向一个网络邻居发送一条信息所需要的时间或其他开销。

对于无权图(或图中所有边等权),单源最短路径可以通过简单的广度优先搜索找到。但是加了权以后,特别是各边的权不全相等时,这个方法就不行了。

解决这个问题的一个方法是,当图中各边的权不相等时,用固定的顺序对顶点进行处理。将顶点依次记为  $v_0$  到  $v_{n-1}$ ,并使  $s = v_0$ 。处理  $v_1$  时,取边  $(v_0, v_1)$  边。处理  $v_2$  时,将  $v_0$  到  $v_1$  再到  $v_2$  两条边的长度之和与  $v_0$  到  $v_2$  的边的长度比较,取较小者为  $v_0$  到  $v_2$  的最短路径长度。处理  $v_i$  时,利用已经处理过的从  $v_0$  到  $v_{i-1}$  的最短路径长度。但是这样可能会产生下述问题:也许从  $v_0$  到  $v_i$  的真正最短路径要经过  $v_j$ , 而  $j > i$ 。使用这个算法将会遗漏这种路径。不过,如果我们以从  $s$  出发所到达点的距离(递增)为顺序对各个顶点进行处理就能避免上述问题了。假设处理了从  $s$  出发所到达的距离最小的前  $i-1$  个顶点,称这些顶点的集合为  $S$ 。现在准备依此顺序处理第  $i$  个顶点,称之为  $x$ 。则从  $s$  到  $x$  的最短路径的倒数第二个顶点一定在集合  $S$  中。因此我们有

$$d(s, x) = \min_{u \in S} (d(s, u) + w(u, x)).$$

换句话说,从  $s$  到  $x$  的最短路径长度为:从集合  $S$  中任取顶点  $u$ , 计算从  $s$  到集合  $S$  中任意顶点  $u$  的长度,加上  $u$  到  $x$  的边的长度之和,取这些和中的最小值。

这个方法通常被称为 Dijkstra 算法。它的技巧在于为  $V$  中所有顶点保留了一个最短路径长度的估计值  $D(x)$ , 并且按照从  $s$  出发所到达点的距离(递增)为顺序处理各个顶点。处理某一个顶点  $v$  时,它的任意一个相邻顶点  $x$  的  $D(x)$  值都可能随之改变。图 7.15 是 Dijkstra 算法的实现。算法结束时,数组  $D$  将包括所有的最短路径长度。

```
// Compute shortest path distances from s, store them in D
static void Dijkstra(Graph G, int s, int[] D) {
    for (int i = 0; i < G.n(); i++) // Initialize
        D[i] = Integer.MAX_VALUE;
    D[s] = 0;
    for (int i = 0; i < G.n(); i++) { // Process the vertices
        int v = minVertex(G, D); // Find the next - closest vertex
        G.setMark(v, VISITED);
        if (D[v] == Integer.MAX_VALUE) return; // Unreachable vertices
        for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
            if (D[G.v2(w)] > (D[v] + G.weight(w)))
                D[G.v2(w)] = D[v] + G.weight(w);
    }
}
```

图 7.15 Dijkstra 算法的实现

有两种理论上可行的方法可以解决在每次主 for 循环中寻找未访问顶点最小  $D$  值的问

题。第一种方法比较简单,通过扫描整个包含 $|V|$ 个元素的表来搜索最小值。算法中使用函数 `minVertex` 实现了这种方法。

```
static int minVertex(Graph G, int[] D) {
    int v = 0; // Initialize v to any unvisited vertex;
    for (int i = 0; i < G.n(); i++)
        if (G.getMark(i) == UNVISITED) { v = i; break; }
    for (int i = 0; i < G.n(); i++) // Now find smallest value
        if ((G.getMark(i) == UNVISITED) && (D[i] < D[v]))
            v = i;
    return v;
}
```

因为扫描需要进行 $|V|$ 次,而且每条边需要相同的次数来更新  $D$  值,所以本方法的总时间代价为  $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$ , 因为 $|E|$ 在  $O(|V|^2)$  中。

第二种方法是将未被处理的顶点以  $D$  值大小为顺序保存在一个最小堆中,可以使用  $\Theta(\log |V|)$  次搜索找出下一个最近顶点。每次改变  $D(x)$  值时,都可以通过先删除再重新插入的方法改变顶点  $x$  在堆中的位置。这是一个 5.6 节中介绍过的在一个优先队列中进行优先更新的例子。为了实现优先更新,我们需要将每个顶点连同它的数组下标存储在堆中。一个更简单的办法是仅为某个顶点添加一个新的(更小的)最短路径的长度作为堆中的新元素(而不作删除旧值的操作)。当前在堆中的某个指定顶点最短路径估计长度的最小值将首先被找到,而其后找到的较大值将被忽略,因为此时顶点已被标记为 `VISITED`。重复插入最短路径长度的惟一缺点是:在最差情况下,它将使堆中元素数目由  $\Theta(|V|)$  增加到  $\Theta(|E|)$ 。总的开销为  $\Theta((|V| + |E|) \cdot \log |E|)$ , 因为处理每条边时,都必须对堆进行一次重排。

我们需要在堆中存储实现 `Elem` 接口的对象(例如有函数 `key` 的对象)。本例中,对象存储的是一个顶点以及它与给定开始顶点之间的(当前最短)距离。`Key` 函数返回该顶点与开始顶点之间的当前最短距离。可以用下面的 `DijkElem` 类来达到这个目的。

```
class DijkElem implements Elem {
    private int vertex;
    private int distance;

    public DijkElem(int v, int d) { vertex = v; distance = d; }
    public DijkElem() { vertex = 0; distance = 0; }

    public int key() { return distance; }
    public int vertex() { return vertex; }
} // class DijkElem
```

图 7.16 给出了使用优先队列实现的 Dijkstra 算法。

```
// Dijkstra's shortest - paths algorithm; priority queue version
static void Dijkstra(Graph G, int s, int[] D) {
    int v; // The current vertex
    DijkElem[] E = new DijkElem[G.e()]; // Heap with lots of space
```

```

E[0] = new DijkElem(s, 0);           // Initialize heap array
MinHeap H = new MinHeap(E, 1, G.e()); // Create the heap
for (int i = 0; i < G.n(); i++)       // Initialize distance array
    D[i] = Integer.MAX_VALUE;
D[s] = 0;
for (int i = 0; i < G.n(); i++) {     // For each vertex
    do { v = ((DijkElem)H.removeMin()).vertex(); } // Get position
    while (G.getMark(v) == VISITED);
    G.setMark(v, VISITED);
    if (D[v] == Integer.MAX_VALUE) return; // Unreachable vertices
    for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
        if (D[G.v2(w)] > (D[v] + G.weight(w))) { // Update D
            D[G.v2(w)] = D[v] + G.weight(w);
            H.insert(new DijkElem(G.v2(w), D[G.v2(w)]));
        }
    }
}

```

图 7.16 利用优先队列实现的 Dijkstra 算法

在密集图中,即 $|E|$ 接近 $|V|^2$ 时,通过扫描顶点表来寻找最小值效率更高。而稀疏图使用优先队列更有效,因为它的时间代价是 $\Theta((|V| + |E|) \cdot \log |E|)$ 。但对密集图而言,这个代价将大到 $\Theta(|V|^2 \log |E|)$ 。

图 7.17 是 Dijkstra 算法的图示。源顶点为 A。除了 A 以外,其余各个顶点的最短路径长度估计值均赋初值为 $\infty$ 。处理完顶点 A 后,它的相邻顶点最短路径长度估计值被更新为到 A 的直接距离。处理完 C(离 A 最近的顶点)后,顶点 B、E 的估计值被更新为对应的经过 C 的最短路径长度。余下顶点按 B、D、E 的顺序被处理。

	A	B	C	D	E
初始状态	0	$\infty$	$\infty$	$\infty$	$\infty$
处理 A	0	10	3	20	$\infty$
处理 C	0	5	3	20	18
处理 B	0	5	3	10	18
处理 D	0	5	3	10	18
处理 E	0	5	3	10	18

图 7.17 Dijkstra 算法作用于图 7.14 时各处理步骤,源顶点为 A

### 7.4.2 每对顶点间的最短路径

接下来我们研究寻找图中每一对顶点间最短路径长度的问题,这也被称为每对顶点间的最短路径(all-pairs shortest-paths)问题。更具体地说,就是对任意的 $u, v \in V$ ,计算 $d(u, v)$ 的值。

解决这个问题的一种方法是使用 $|V|$ 次 Dijkstra 算法,每次从不同的顶点出发计算最短路径。如图 G 是稀疏图( $|E| = \Theta(|V|)$ ),这不失为一种好方法,因为这种 Dijkstra 算法是基

于优先队列的,所以总的时间代价为  $\Theta(|V|^2 + |V||E|\log|V|) = \Theta(|V|^2\log|V|)$ 。对于密集图来说,基于优先队列的 Dijkstra 算法时间代价为  $\Theta(|V|^3\log|V|)$ ,而前面介绍的使用 MinVertex 的 Dijkstra 算法在这里的时间代价将为  $\Theta(|V|^3)$ 。

另一种称为 Floyd 算法的方法,不论有多少条边均能把处理时间限制在  $\Theta(|V|^3)$ 。定义  $k$ -path 为任意一条从顶点  $v$  到  $u$  的、中间顶点(除  $v$  和  $u$  外)序号小于  $k$  的路径。0-path 即为直接从  $v$  到  $u$  的边。图 7.18 是  $k$ -path 概念的图示。

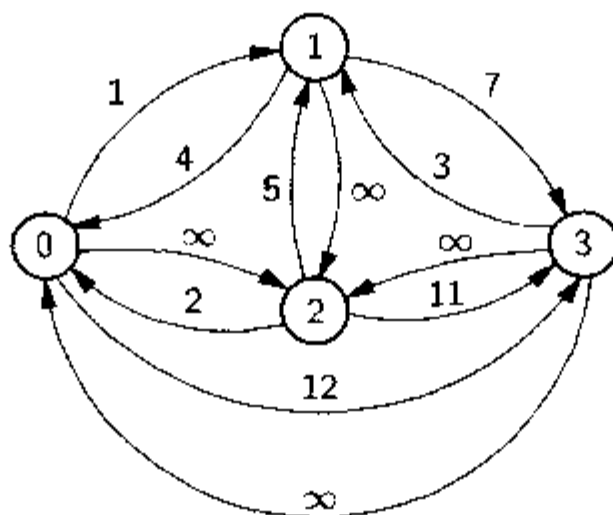


图 7.18 Floyd 算法中  $k$ -path 的一个实例。根据定义,路径 1,3 是 0-path。路径 2, 0, 1 不是 0-path, 而是 1-path(也可说是 2-path 或 3-path 等),因为序号最小的中间顶点为 0 顶点。路径 0,3, 1 是 4-path 而不是 3-path,因为中间顶点是顶点 3。图中最终所有路径均为 4-path(扩展了所有 4 个顶点后的结果)

定义  $D_k(v, u)$  为从  $v$  到  $u$  长度最小的  $k$ -path。假设我们已知从  $v$  到  $u$  的最短  $k$ -path,则最短的  $(k+1)$ -path 要么经过,要么不经过顶点  $k$ 。如果它经过顶点  $k$ ,则最短  $(k+1)$ -path 的一部分是从  $v$  到  $k$  的最短  $k$ -path,另一部分是从  $k$  到  $u$  的最短  $k$ -path。否则,我们保持其值为最短  $k$ -path 不变。Floyd 算法仅通过一个三重循环检查了所有的可能性。下面给出 Floyd 算法的实现。在算法结束时,  $D$  数组存储了所有成对顶点间的最短路径长度。

```
// Compute all - pairs shortest paths
static void Floyd(Graph G, int[][] D) {
    for (int i = 0; i < G.n(); i++) // Initialize D with initial weights
        for (int j = 0; j < G.n(); j++)
            D[i][j] = G.weight(i, j);
    for (int k = 0; k < G.n(); k++) // Compute all k paths
        for (int i = 0; i < G.n(); i++)
            for (int j = 0; j < G.n(); j++)
                if ((D[i][k] != Integer.MAX_VALUE) &&
                    (D[k][j] != Integer.MAX_VALUE) &&
                    (D[i][j] > (D[i][k] + D[k][j]))))
                    D[i][j] = D[i][k] + D[k][j];
}
```

很明显,这个算法需要  $\Theta(|V|^3)$  运行时间,但是它是密集图的最好选择,因为它相对来说较快,而且易于实现。

## 7.5 最小支撑树

本节介绍求图的最小支撑树(minimum-cost spanning tree 或 MST)的两个算法。给定一个连通无向图  $G$ , 且它的每条边均有相应的长度或权值, 则 MST 是一个包括  $G$  的所有顶点及其边子集的图, 边的子集满足下列条件:

- (1) 这个子集中所有边的权之和为所有子集中最小的。
- (2) 子集中的边能够保证图是连通的

解决 MST 问题的方法可用于以下应用问题: 怎样使连接电路板上一系列接头所需焊接的线路最短, 或者怎样使得在几个城市之间建立电话网所需的线路最短。

MST 中没有回路, 因为如果 MST 的边集中有回路, 显然可以通过去掉回路中某条边而得到开销更小的 MST。因此, MST 是一棵有  $|V| - 1$  条边的自由树。之所以称之为最小支撑树, 是因为一方面满足 MST 要求的边集所构成的树支撑起了所有的顶点(即把它们联接起来了), 另一方面此边集的代价最小。图 7.19 给出了一个图及其 MST 的图示。

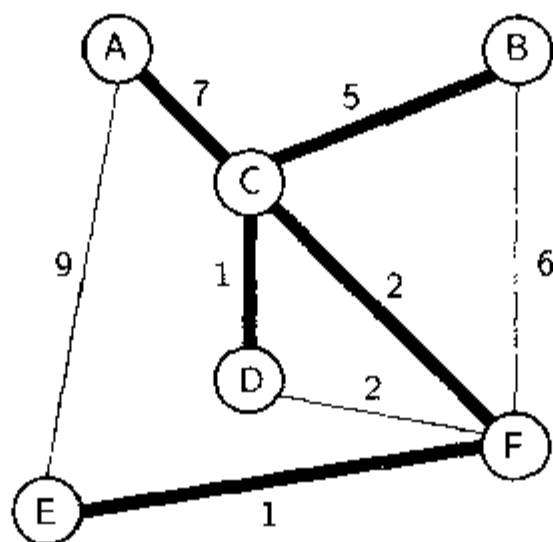


图 7.19 一个图及其 MST。所有的线都是原图的边, MST 中边的子集用粗线表示。注意, 如果使用边  $(C, F)$  代替边  $(D, F)$ , 可得到另一个 MST。

### 7.5.1 Prim 算法

我们要介绍的求最小支撑树的两种算法中, 第一种通常被称为 Prim 算法。它很简单, 由图中任意一个顶点  $N$  开始, 初始化 MST 为  $N$ 。选出与  $N$  相关联的边中权最小的一条, 设其连接  $N$  与另一顶点  $M$ 。把顶点  $M$  和边  $(N, M)$  加入 MST 中。然后, 选出与  $N$  或  $M$  相关联的边中权最小的一条, 设其连接另一个新顶点, 将此边和新顶点添加到 MST 中。反复进行这样的处理, 每一步都通过选出连接当前已在 MST 中的某个顶点以及另一个不在 MST 中顶点的代价最小的边而扩展 MST。

Prim 算法与寻找一个顶点到其他顶点最短路径的 Dijkstra 算法十分相似, 主要区别在于我们不是寻找下一个离起点最近的顶点, 而是下一个与 MST 中某个顶点相距最近的顶点。图 7.20 给出了 Prim 算法的实现, 它通过搜索距离矩阵(即相邻矩阵)来找下一个最近顶点。数组  $V[i]$  存储刚被访问的离顶点  $i$  最近的顶点, 这样就可以知道处理到顶点  $i$  时应加入 MST 的是哪条边。



```

// Compute a minimal - cost spanning tree
static void Prim(Graph G, int s, int[] D) {
    int[] V = new int[G.n()]; // V[i] stores closest vertex to i
    for (int i = 0; i < G.n(); i++) // Initialize
        D[i] = Integer.MAX_VALUE;
    D[s] = 0;
    for (int i = 0; i < G.n(); i++) { // Process the vertices
        int v = minVertex(G, D);
        G.setMark(v, VISITED);
        if (v != s) AddEdgeToMST(V[v], v);
        if (D[v] == Integer.MAX_VALUE) return; // Unreachable vertices
        for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
            if (D[G.v2(w)] > G.weight(w)) {
                D[G.v2(w)] = G.weight(w);
                V[G.v2(w)] = v;
            }
    }
}

```

图 7.20 Prim 算法的实现

我们也可以像图 7.21 那样用一个优先队列来寻找下一个最近顶点,从而实现 Prim 算法的另外一种方案。与用优先队列实现的 Dijkstra 算法类似,堆的 Elem 类型存储 DijElem 对象, key 函数返回顶点的  $D$  值,  $D$  值是该顶点与已标记顶点相距最近的距离。

```

// Prim's MST algorithm: priority queue version
static void Prim(Graph G, int s, int[] D) {
    int v; // The current vertex
    int[] V = new int[G.n()]; // V[i] stores closest vertex to i
    DijElem[] E = new DijElem[G.e()]; // Heap with lots of space
    E[0] = new DijElem(s, 0); // Initialize heap array
    MinHeap H = new MinHeap(E, 1, G.e()); // Create the heap
    for (int i = 0; i < G.n(); i++) // Initialize distance array
        D[i] = Integer.MAX_VALUE;
    D[s] = 0;
    for (int i = 0; i < G.n(); i++) { // Now, get distances
        do { v = ((DijElem)H.removeMin()).vertex(); } // Get position
        while (G.getMark(v) == VISITED);
        G.setMark(v, VISITED);
        if (v != s) AddEdgeToMST(V[v], v); // Add this edge to MST
        if (D[v] == Integer.MAX_VALUE) return; // Unreachable vertices
        for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
            if (D[G.v2(w)] > G.weight(w)) { // Update D
                D[G.v2(w)] = G.weight(w);
            }
    }
}

```

```

V[G.v2(w)] = v;           // Update who it came from
H.insert(new DijkElem(G.v2(w), D[G.v2(w)]));

```

图 7.21 使用优先队列实现 Prim 算法

Prim 算法是贪心算法的一个实例。每执行一次 for 循环,都选出一条连接一个已标记顶点和另一个未标记顶点的具有最小权的边。但是 MST 真正应该包含所谓的下一条权最小的边吗?这就导致了一个重要的问题:Prim 算法正确吗?显然,它会生成一棵支撑树(因为每次 for 循环都将一个未标记顶点加入此支撑树中,直到所有顶点都被添加进去为止),但是这棵树的总开销是最小的吗?

**定理 7.1** Prim 算法产生的是最小支撑树。

**证明:**本定理可以使用反证法来证明。设图  $G = (V, E)$  不能通过 Prim 算法生成最小支撑树。根据 Prim 算法中各顶点加入 MST 的顺序依次定义图  $G$  中各顶点为:  $v_0, v_1, \dots, v_{n-1}$ 。令  $e_i$  代表边  $(v_x, v_i)$ , 其中  $x < i$  且  $i \geq 1$ , 令  $e_j$  为 Prim 算法添加的序号最小的那条(第一条)出现以下情况的边:加入  $e_j$  后的边集不能被扩展而构成图  $G$  的一个 MST。换句话说,  $e_j$  是 Prim 算法发生错误的第一条边。设  $T$  为“真正的”MST。令  $v_p$  为被边  $e_j$  所关联的点, 即  $e_j = (v_p, v_j)$ 。

因为  $T$  是一棵树, 所以  $T$  中将存在一条连接  $v_p$  和  $v_j$  的路径, 且此路径中一定会存在某条边  $e'$  连接  $v_u$  和  $v_w$ , 其中  $u < j$ ,  $w \geq j$  (即顶点  $v_u$  处于图 7.22 中左边已标记点的集合中, 而  $v_w$  处于图 7.22 右边未标记点的集合中, 才有可能成为联系图中各顶点的支撑树  $T$ )。因为把  $e_j$  加入到  $T$  会构成一个回路, 所以  $e_j$  不属于  $T$ 。又因为 Prim 算法不能生成一个 MST, 所以  $e'$  比  $e_j$  的权更小(否则就去掉边  $e'$ , 换成边  $e_j$  所形成的树的总权开销更小)。这种情形在图 7.22 中给出了图示。但是, Prim 算法应该是选择可能的最小权边, 它一定会选择  $e'$ , 而不是  $e_j$ 。这与 Prim 算法选错了边  $e_j$  的假设相矛盾。因此, Prim 算法一定是正确的。

下面给出一个说明 Prim 算法如何工作的例子。对于图 7.19 所示的图来说, 设从标记顶点  $A$  开始。从  $A$  出发权最小的边连接顶点  $C$ 。把  $C$  和边  $(A, C)$  加入 MST 中。从 MST 中现有顶点出发的权最小的边是  $(C, D)$ 。接着被标记的是顶点  $F$ 。因为边  $(C, F)$  与  $(D, F)$  相等, 选择哪条边都可以。再下一步标记顶点  $E$  并把边  $(F, E)$  加入 MST。按此方法继续, 顶点  $B$  (通过边  $(C, B)$ ) 被标记, 此时算法结束。

### 7.5.2 Kruskal 算法

下一个要介绍的 MST 算法通常称为 Kruskal 算法。它也是一个简单的贪心算法。首先, 我们将顶点集分为  $|V|$  个等价类, 每个等价类包括一个顶点。然后以权的大小为顺序处理各条边。如果某条边连接两个不同等价类的顶点, 就把这条边添加到 MST, 并把两个等价类合并为一个。反复执行此过程直至只剩下一个等价类。

以权为顺序处理各条边可以通过使用 min-heap(最小值堆)来实现。这通常比首先对边进行排序更快, 因为实际上在完成 MST 前仅需访问一小部分边。这是一个在表中查找少数最小元素的例子, 我们将在 8.6 节进一步讨论。

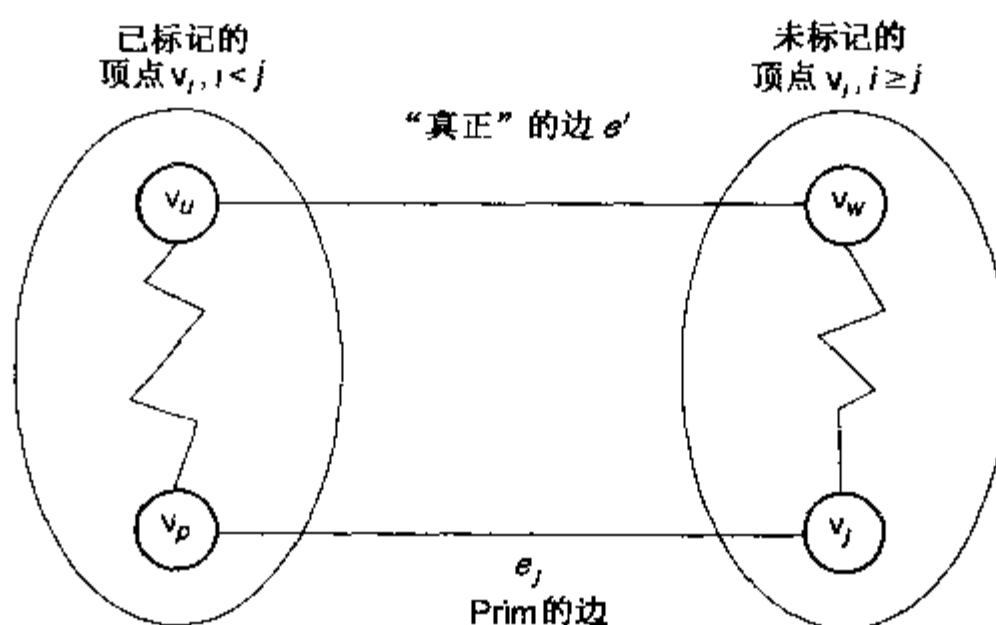


图 7.22 Prim 算法证明过程图示。左边的椭圆包括 Prim 算法产生的 MST 和“真正的”MST 相吻合的那部分图。右边的椭圆包括图剩余未作标记处理的部分。图的两部分(至少)由边  $e_j$  (根据 Prim 算法的选择应属于 MST 的边)和  $e'$  (“真正”应属于 MST 的边)所连接。注意,从  $v_w$  到  $v_{j+1}$  的路径中不能包括任何已标记的顶点  $v_i$ , 其中  $i \leq j$ , 因为这样将构成一个回路

算法中惟一需要技巧的部分是怎样确定两个顶点是否属于同一等价类。幸运的是,我们可以使用理想的数据结构来实现这一点。可以简单地使用 6.2 节所介绍的树的基于父指针表示法的 UNION/FIND 算法。图 7.23 是 Kruskal 算法求 MST 前三步的图示。图 7.24 是这个算法的实现,其中 KruskalElem 类用于在最小值堆中存储边。

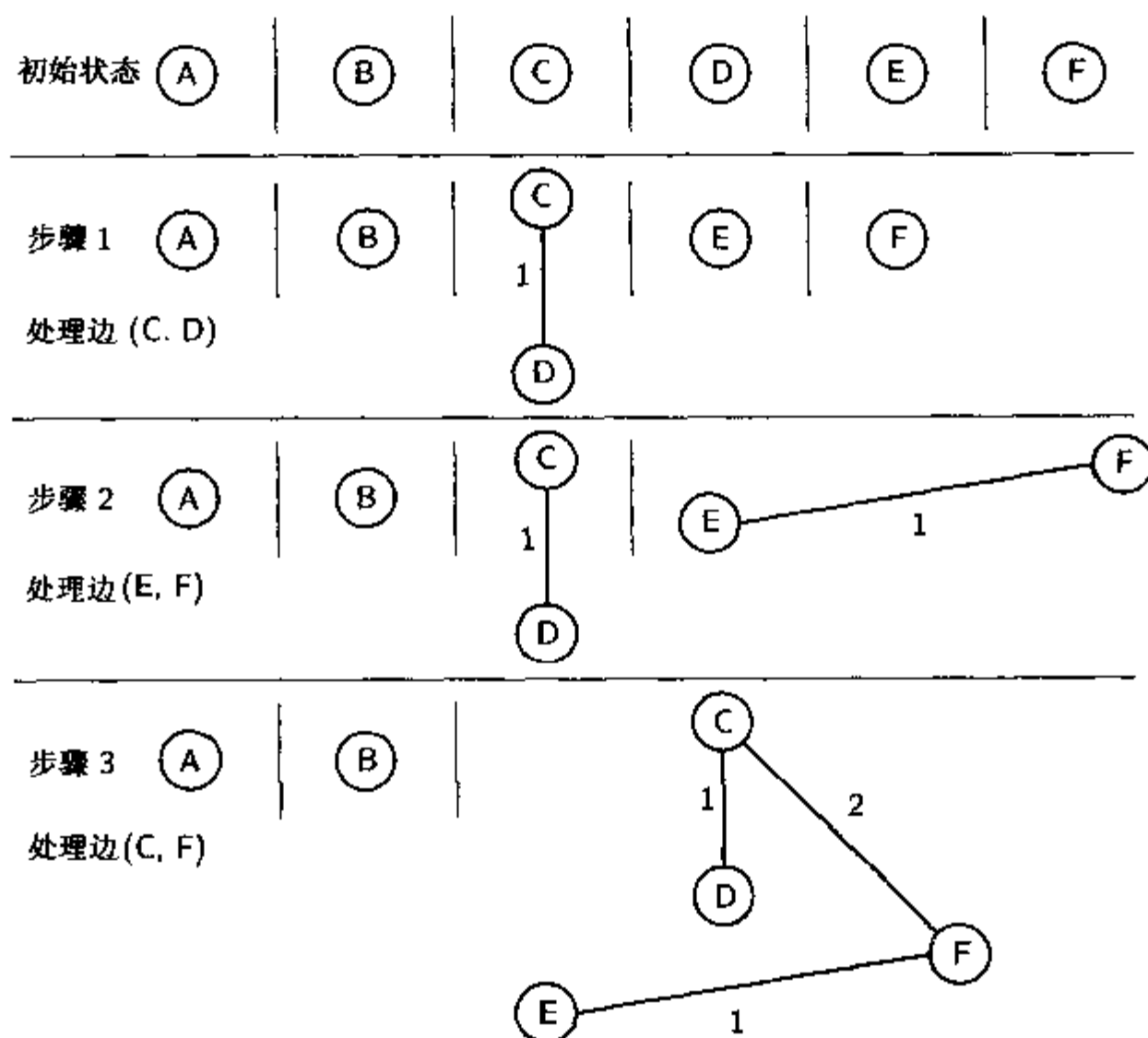


图 7.23 应用于图 7.19 所示图的 Kruskal 算法前三步图示

Kruskal 算法的代价由按权处理各条边所需要的时间确定。如果使用了路径压缩,则 `differ` 和 `UNION` 函数几乎接近常数。因此,最差情况下此算法的总时间代价为  $\Theta(|E|\log|E|)$ , 而一般情况下的代价则接近  $\Theta(|V|\log|E|)$ 。

```
class KruskalElem implements Elem {
    private Edge edge;
    private Graph G;

    public KruskalElem(Graph inG, Edge w) { G = inG; edge = w; }
    public int key() { return G.weight(edge); }
    public Edge edge() { return edge; }
} // class KruskalElem

static void Kruskal(Graph G) {          // Kruskal's MST algorithm
    GerfTree A = new GerfTree(G.n()); // Equivalence class array
    KruskalElem[] E = new KruskalElem[G.e()]; // Minheap array
    int edgecnt = 0; // Count of edges

    for (int i = 0; i < G.n(); i++)      // Put the edges on the array
        for (Edge w = G.first(i); G.isEdge(w); w = G.next(w))
            E[edgecnt++] = new KruskalElem(G, w);
    MinHeap H = new MinHeap(E, edgecnt, edgecnt); // Heapify the edges
    int numMST = G.n();                    // Initially n equiv classes
    for (int i = 0; numMST > 1; i++) {      // Combine equiv classes
        KruskalElem temp = (KruskalElem)H.removeMin(); // Next cheapest
        Edge w = temp.edge();
        int v = G.v1(w); int u = G.v2(w);
        if (A.differ(v, u)) {              // If in different equiv classes
            A.UNION(v, u);                  // Combine equiv classes
            AddEdgeToMST(G.v1(w), G.v2(w)); // Add this edge to MST
            numMST--;                       // One less MST
        }
    }
}
```

图 7.24 Kruskal 算法的实现

## 7.6 深入学习导读

可以利用《Stanford Graphbase》中的程序来研究图的许多有趣性质。此书是有关标准数据库和图处理程序的选集。请参见文献[Knu94]。

## 7.7 习题

- 7.1 用归纳法证明  $n$  个顶点的图最多只能有  $n(n-1)/2$  条边。
- 7.2 证明下述关于自由树的描述:
- (a) 一个没有简单回路的连通无向图有  $|V| - 1$  条边。
  - (b) 一个有  $|V| - 1$  条边的无环图一定是连通的。
- 7.3 (a) 画出图 7.25 所示图的相邻矩阵表示。  
 (b) 画出此图的邻接表表示。  
 (c) 如果每个指针需要 4 个字节, 每个顶点的标号占 2 个字节, 每条边的权占 2 个字节, 此图采用哪种表示法所需要的空间较多?

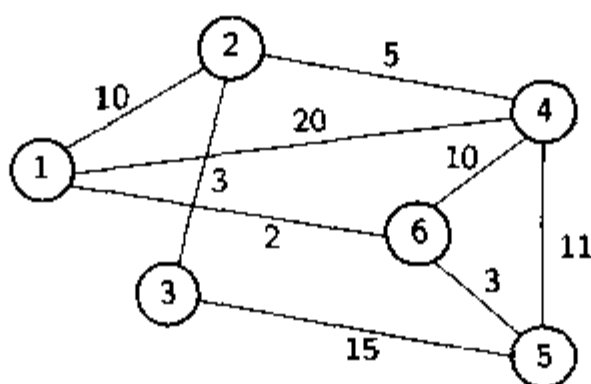


图 7.25 第 7 章习题例图

- 7.4 给出图 7.25 所示图从顶点 1 开始的 DFS 树。
- 7.5 给出此图从顶点 1 开始的 BFS 树。
- 7.6 BFS 拓扑排序算法在遇到回路时会返回报告“存在回路”的信息。修改此算法使之能打印所遇到的第一个回路(假如存在)。
- 7.7 写出对图 7.25, 从顶点 4 出发, 使用 Dijkstra 最短路径算法产生的单源最短路径表。请像图 7.17 所示那样, 画出距离数组  $D$  中的数值变化过程。
- 7.8 修改单源最短路径算法, 使它确实能存储并返回最短路径而不仅仅计算其长度。
- 7.9 一个 DAG 的根结点是指某个结点  $R$ , 该 DAG 的任意一个结点都可以从  $R$  出发通过有向路径到达。写出一个算法, 使之以一个有向图作为输入, 如果有一个根, 请确定此图的根结点。
- 7.10 写一个算法查找一个 DAG 中的最长路径, 这里路径长度由该路径所包含边的数目确定。你的算法的渐近复杂度是多少?
- 7.11 写一个算法以确定有  $n$  个顶点的有向图是否包含回路。此算法的时间代价应该是  $\Theta(n)$ 。
- 7.12 写一个算法以确定有  $n$  个顶点的无向图是否包含回路。此算法的时间代价应该是  $\Theta(n)$ 。
- 7.13 有向图的单目标最短路径(single-destination shortest path)问题是找出从各顶点到某一指定顶点的最短路径。写一个算法解决此问题。
- 7.14 给出对图 7.25 所示图使用 Floyd 每对顶点间最短路径算法时的结果。
- 7.15 7.4.2 小节给出的 Floyd 算法对邻接表来说效率并不高, 这是因为初始化  $D$  数组

时,边的访问顺序不佳。那么,对邻接表来说,初始化的时间代价是多少?怎样改进初始化这一步使之在最差情况下时间代价为  $\Theta(|V|^2)$ ?

- 7.16 说明你所认为的每对顶点间最短路径问题的最大可能下限,并验证你的答案。
- 7.17 列出对图 7.25 从顶点 3 开始使用 Prim 的 MST 算法时各边被访问的顺序,并给出最终的 MST。
- 7.19 列出对图 7.25 运行 Kruskal 最小支撑树算法时,逐步访问各边的顺序。每加入一条边到 MST 时,等价类数组的结果是什么(如图 6.6 那样显示该数组内容)?
- 7.19 写出求最大支撑树的算法,即此支撑树有最大可能开销。
- 7.20 在什么情况下 Prim 算法与 Kruskal 算法生成不同的 MST?
- 7.21 证明如果图  $G$  所有边的代价均不相等时,它只存在一棵 MST。
- 7.22 如果图中有一部分边的权为负值,那么 Prim 算法或 Kruskal 算法是否可行?
- 7.23 Dijkstra 最短路径算法是否给出一棵支撑树(不论开销是否最小)?它是否生成一棵 MST 呢?请说明理由。
- 7.24 写一个算法来标记一个无向图的连通分量。换句话说,第一个连通分量的所有顶点给以第一分量的标记,第二个连通分量的所有顶点给以第二分量的标记,依次类推。

## 7.8 项目设计

- 7.1 设计一个格式以使用文件方式存放图,再实现一个函数以便从某个文件中将图读入主存。实现另一个函数将一个图写入文件。这样检验你的函数:实现一个完整的 MST 程序,从一个文件中读入一个无向图,构造 MST,再将表示此 MST 的有向图写入另一个文件。

原书空白

# 第三部分 排序和检索

第 8 章 内排序

第 9 章 文件管理和外排序

第 10 章 检索

第 11 章 索引技术



## 第8章 内 排 序

排序是数据处理中经常使用的一种很重要的运算,因此人们已经对它进行了深入细致的研究,并且已经设计出了一些巧妙的算法。但是,仍然有一些与排序相关的问题尚未解决,适应各种不同要求的新算法也不断被开发出来并得到了改进。本章在对排序问题进行讲解的同时,也涉及到了许多重要算法的分析。排序算法涉及到广泛的算法分析技术。排序问题的研究也促进了文件处理技术的发展,第9章将介绍文件处理技术。许多程序要对不能在内存中直接处理的大数据集进行排序,而基于磁盘的排序需要特殊的技术。第9章在介绍了基于磁盘的处理原则之后,将介绍外存排序技术。

这一章介绍几种实用的内排序(internal sorting)方法。首先对三个简单但是相对较慢的算法进行分析,它们在平均和最差情况下的时间代价是  $\Theta(n^2)$ 。有一些较好算法的时间代价是  $\Theta(n \log n)$ 。最后介绍一种在最佳情况下只用  $\Theta(n)$  的算法。本章还将证明排序算法一般来说在最差情况下的时间代价为  $\Omega(n \log n)$ 。

### 8.1 排序的术语及记号

如果没有特别说明,本章中的数据类型都是存储在数组中的一组记录。数组中的每一个记录内都有一个域称为排序关键码(sort key),或者简称为关键码(key)。这个关键码可以是任何一种可比的有序数据类型,关键码的类型可能是:字符、字符串、整数、实数或者其他更复杂的类型,前提是可以找到一种能够比较关键码之间顺序的函数。我们认为对于任何一种记录都可以找到一个取得它的关键码的函数,记为函数 `key`,而且假设比较关键码的运算符已经有了定义。因此,本章的排序中比较记录 `R` 和 `S` 就有以下形式:

```
if (R.key() <= S.key()) ...
```

我们还假设对每种记录类型已定义了用于交换两个记录的函数 `Dstul.swap`。该函数以数组和两个被交换记录的下标为变量,在数组中交换这两个记录的位置。

**定义 8.1** 排序问题:给定一组记录  $r_1, r_2, \dots, r_n$ , 其关键码分别为  $k_1, k_2, \dots, k_n$ , 将这些记录排成顺序为  $r_{s_1}, r_{s_2}, \dots, r_{s_n}$  的一个序列  $S$ , 满足条件  $k_{s_1} \leq k_{s_2} \leq \dots \leq k_{s_n}$ 。换句话说,排序就是要重排一组记录,使其关键码域的值具有不减的顺序。

根据定义,排序问题中的记录可以具有相同的关键码。有些应用中要求输入没有重复关键码的一组记录。除非特别说明,本章与第9章中所有排序算法都适用于处理具有重复关键码的问题。

当允许关键码重复时,也许具有相同关键码的记录之间本身就有某种内在的顺序,一般为它们的输入顺序。有的应用可能要求不改变具有相同关键码的记录的原始输入顺序。如果一种排序算法不改变这种相对顺序则称之为稳定的(stable)。本章中的大多数(但不是全部)排序算法都是稳定的。

当比较两个排序算法时,最直接了当的方法是对它们进行编程,然后比较它们的运行时间。这种时间比较的例子请参见图 8.14 和图 8.15。但是,有些算法的运行时间依赖于原始输入记录的情况,因此这种比较方法也就失去了意义。特别是记录的数量、记录的大小、关键码的可操作区域以及输入记录的原始有序程度,这些都会大大影响排序算法的相对运行时间。

分析排序算法时,传统方法是衡量关键码之间进行比较的次数。这种方法通常与算法消耗的时间有关,而与机器和数据类型无关。但是在一些情况下,记录也许很大,以致于它们的移动是影响程序整个运行时间的重要因素。在这种情况下,应该统计算法中所使用的交换次数。在大多数情况下,我们可以假设所有记录及关键码都具有固定长度,因此做一次比较或者交换所用的时间也是固定的,不用考虑所比较的是哪一个关键码。一些特定的应用可以采取较灵活的比较方法。例如,一个应用中不同的记录或关键码的长度差别很大(例如对一个长度不同的字符串序列进行排序),可以采用一些特殊的算法。一些实例中只对少量记录进行排序,但是排序操作的频率很高,例如仅对 5 个记录反复排序。在这种情况下,在渐进分析的运行时间公式中常常被忽略的常量系数和常数项就变得十分重要了。另外,还有一些实例要求占用的内存尽量少。

## 8.2 三种代价为 $\Theta(n^2)$ 的排序方法

本节介绍了三种简单的排序算法。尽管这些算法简单易懂且易于实现,但是很快你就会发现对于待排序的记录数较多的排序算法来说,它们的速度令人无法忍受。可是,在某些情况下这些最简单的算法可能是最好的算法。

### 8.2.1 插入排序

这里将要介绍的第一种排序方法称为插入排序法(Insert Sort)。插入排序逐个处理待排序的记录,每个新记录与前面已排序的子序列进行比较,将它插入到子序列中正确的位置。下面是使用 Java 编写的函数,其输入是一个记录数组,数组中存放着  $n$  个记录。

```
static void insort(Elem[] array) { // Insertion Sort
    for (int i = 1; i < array.length; i++) // Insert i'th record
        for (int j = i; (j > 0) && (array[j].key() < array[j-1].key()); j--)
            DSUtil.swap(array, j, j-1);
}
```

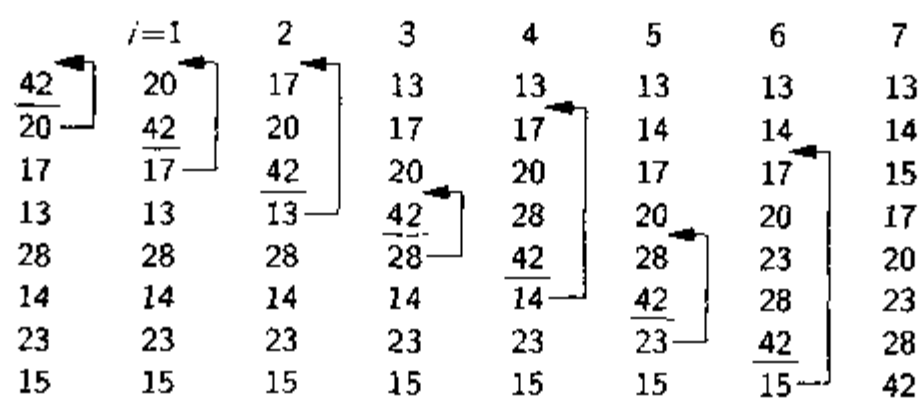


图 8.1 插入排序说明。图中的每一列数都表示以该列顶部的  $i$  值进行一次 for 循环后数组中的内容。每列中横线以上的记录是已排序的。每个箭头都指示了该元素应该插入的位置

考虑一下 `inssort` 处理第  $i$  个记录的情况,记录的值设为  $X$ 。当记录  $X$  比它上面的那个值小时,就向上移动它。直到遇到一个比它小或者与它相等的值,本次插入才完成,因为再往前就一定都比它小了。

插入排序的程序体是由嵌套的两个 `for` 循环组成的。外层插入循环要做  $n-1$  次。里面的插入循环次数分析起来要更困难一些,因为该循环次数依赖于在第  $i$  个记录前的  $i-1$  个记录中有多少个关键码值小于第  $i$  个记录的关键码值。最差的情况是:每个记录都必须移动到数组的顶端。如果原来数组中的原始数据是逆序的,这种情况将会发生。这时内部循环总共要做  $i-1$  次比较。因此,比较次数最多为:

$$\sum_{i=2}^n i = \Theta(n^2)$$

相反,考虑最佳情况,此时数组中的关键码就是从小到大按照正序排列的。在这种情况下,每个结点刚进入 `for` 循环就退出,没有记录需要移动。总的比较次数为  $n-1$  次,即外层 `for` 循环的执行次数。因此最佳情况下插入排序的时间代价为  $\Theta(n)$ 。

虽然最佳情况要比最差情况快得多,但是往往最差情况的性能是“典型”运行时间性能的可信指标。尽管如此,在有些情况下,待排序数据可能是已经有序或者基本有序。例如,在一个已排序序列上略作修改,如果改动不太大,最好用插入排序把它恢复成有序的。8.3 节的 Shell 排序及 8.4 节的快速排序中,将给出利用插入排序最佳情况的例子。

那么,插入排序的平均执行时间到底是多少呢?当处理到第  $i$  个记录时,内层 `for` 循环的执行次数依赖于该记录离最终位置的距离。也就是说,第  $i$  个记录前面的 0 到  $i-1$  个记录中,比第  $i$  个记录大的那些记录都会引起 `for` 循环执行一步。例如,在图 8.1 最左边的值 15 前面有 5 个比它大的数。每一个这样的数称为 1 个逆置(*inversion*)。逆置的数目(例如,数组中位于一个给定值之前,并比它大的值的数目)将决定比较及交换的次数。我们需要判断对于第  $i$  个记录来说,其平均逆置有多少。平均情况下,在数组的前  $i-1$  个记录中有一半值比第  $i$  个记录的值大。因此,平均的时间开销就是最差情况的一半,仍然为  $\Theta(n^2)$ 。因此,在渐近复杂性的意义上,平均情况也并不比最差的情况好多少。

计算比较及交换的次数所得出的结果是相近的,因为要比较一次就要交换一次(除了每一轮插入时的最后一次比较找到了应该插入的位置,此时没有发生交换)。因此,总交换次数是总比较次数减去  $n-1$ ,它在最佳情况下为 0,在最差及平均情况下为  $\Theta(n^2)$ 。

### 8.2.2 起泡排序

下面要介绍的是一种称为起泡排序(*Bubble Sort*)的算法。起泡排序常常在计算机科学的一些入门课程中作为例题给初学程序设计者讲述。这其实并不合适,因为起泡排序并没有那么浅显。它是一种较慢的排序,而且不同于插入排序的是它没有较好的最佳情况执行时间。尽管如此,起泡排序给下面将要讨论的一种更好的排序提供了基础。

起泡排序包括一个简单的双重 `for` 循环。第一次的内循环从数组的底部比较到顶部,比较相邻的关键码。如果下面的关键码比其上邻居的关键码小,则将二者交换顺序。如此反复地做下去。一旦遇到一个最小关键码,这个过程将使它像个“气泡”似地被推到数组的顶部(就像水底的气泡冒到水面上一样)。第二次再重复调用上面的过程。但是,既然我们知道最小元素第一次就被排到了数组的最上面,因此就没有必要再比较最上面的两个元素了。同样,每一

轮循环都是比较相邻的关键码,但是都将要比上一轮循环少比较一个关键码。图 8.2 阐明了起泡排序的工作过程。一个 Java 函数如下:

	i=0	1	2	3	4	5	6
42	13	13	13	13	13	13	13
20	42	14	14	14	14	14	14
17	20	42	20	15	15	15	15
13	17	20	42	20	17	17	17
28	14	17	15	42	20	20	20
14	28	15	17	17	42	23	23
23	15	28	23	23	23	42	28
15	23	23	28	28	28	28	42

图 8.2 起泡排序说明。图中每一列数都表示以该列顶部的  $i$  值进行一次 for 循环后数组中的内容。箭头指示一个循环中每次交换的具体位置

```
static void buksort(Elem[] array) {    // Bubble Sort
    for (int i = 0; i < array.length - 1; i++)    // Bubble up i'th record
        for (int j = array.length - 1; j > i; j--)
            if (array[j].key() < array[j - 1].key())
                DSUtil.swap(array, j, j - 1);
}
```

分析起泡排序的比较次数十分简单。不去考虑数组中结点的组合情况,比较的次数总是  $i$ , 因此时间代价为:

$$\sum_{i=1}^n i = \Theta(n^2)$$

起泡排序的最佳、平均、最差情况的运行时间几乎是相同的。

一个结点比它前一个结点的关键码值小的概率有多大就决定了交换的次数。我们可以假定这个概率为平均情况下比较次数的一半,因此代价为  $\Theta(n^2)$ 。事实上起泡排序的交换次数与插入排序的交换次数相同。

### 8.2.3 选择排序

最后介绍的一种  $\Theta(n^2)$  级排序是选择排序(Selection Sort)。选择排序第  $i$  次是选择数组中第  $i$  小的记录。并将这个记录放到数组的第  $i$  个位置。换句话说,选择排序首先从未排序的序列中找到最小关键码,接着是次小的,如此反复找出较小的关键码,直到完全排序。它比较独特的地方是很少交换。在寻找下一个较小的数时,需要检索整个未排序的序列,但是只用一次交换即可将待排序的记录放到正确位置。这样最多需要交换  $n - 1$  次。

图 8.3 解释了如何进行选择排序。下面是用 Java 编写的函数:

```
static void selSort(Elem[] array) {    // Selection Sort
    for (int i = 0; i < array.length - 1; i++) {    // Select i'th record
        int lowindex = i;    // Remember its index
        for (int j = array.length - 1; j > i; j--)    // Find the least value
            if (array[j].key() < array[lowindex].key())
                lowindex = j;    // Put it in place
    }
}
```

```
DSUtil.swap(array, i, lowindex);
```

	i=0	1	2	3	4	5	6
42	13	13	13	13	13	13	13
20	20	14	14	14	14	14	14
17	17	17	15	15	15	15	15
13	42	42	42	17	17	17	17
28	28	28	28	28	20	20	20
14	14	20	20	20	28	23	23
23	23	23	23	23	23	28	28
15	15	15	17	42	42	42	42

图 8.3 选择排序的一个示例。每列代表以  $i$  值为循环值的外层 for 循环执行后数组中的记录情况。在每列中划横线以上的元素都已经是有序的了,而且都在它们的最终位置上

选择排序实质上就是起泡排序,我们记住选择的最小元素的位置并最后用一次交换使它到位,而不是不断地交换相邻记录以使下一个最小记录到位。因此,比较的次数仍为  $\Theta(n^2)$ ,但是交换的次数要比起泡排序少得多。对于处理那些作一次交换花费时间较多的问题,选择排序是很有效的。例如当元素是较长的字符串或者是其他大型记录时,其他情况下也要比起泡排序有效。

还有一种方法可以降低各种排序算法用于交换记录所用的时间,尤其是当记录很大的时候。这就是使数组中的每一个元素存储指向该元素记录的指针。在这种实现中,交换只需要对指针所指的关键码进行操作。图 8.4 是该技术的一个示例。虽然需要一些空间来存放指针,但换来了更高的效率。注意本章排序算法的实现都采用了基类型为 Elem 的数组作为输入参数。由于 Java 数组存储指向对象的引用,而不存储对象本身,所以 Java 自然鼓励交换指针的实现方式。

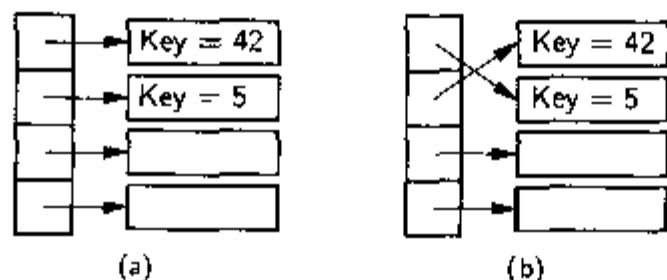


图 8.4 交换指向记录的指针的示例

(a) 4 个记录的序列, 关键码值为 42 的记录排在关键码值为 5 的记录前面。(b) 上面两个指针交换后的 4 个记录, 现在关键码值为 5 的记录排在关键码值为 42 的记录前面

#### 8.2.4 交换排序算法的时间代价

图 8.5 列出了插入排序、起泡排序和选择排序<sup>①</sup> 分别在最佳、平均和最差情况下的比较次数与交换次数。三种方法在平均及最差情况下的时间代价皆为  $\Theta(n^2)$ 。

在一般情况下,本章下面要介绍的排序算法比上述三种快得多。但是在继续介绍其他算法之前,有必要讨论一下这三种排序算法如此之慢的原因。关键的瓶颈是只比较相邻的元素。

<sup>①</sup> 选择排序不同于插入排序和起泡排序之处在于选择排序在最佳情况下的交换次数大于后两者。

	插入	起泡	选择
比较次数:			
最佳情况	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
平均情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
最差情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
移动次数:			
最佳情况	0	0	$\Theta(n)$
平均情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
最差情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

图 8.5 三种简单排序算法的渐近时间复杂性

因此,比较和移动只能一步一步地进行(除了选择排序外)。交换相邻记录叫作一次交换(exchange)。因此,有时这些排序被称为交换排序(exchange sort)。

任何一种交换排序的时间代价是数组中所有记录移到“正确”位置所要求的总步数。为了确定平均情况下交换排序的最佳时间代价,我们需要计算每一个记录的当前位置与其最终在排好序的数组中位置的差别,然后把这些差别累计起来。

那么交换排序的最小时间代价(平均来说)到底是多少呢?考虑一个有  $n$  个元素的序列  $L$ 。 $L$  中有  $n(n-1)/2$  对不同的元素,每一对都可能是一个逆置,每一种这样的对一定在  $L$  中或在  $L_R$  ( $L$  的逆置序列)中。因此, $L$  及  $L_R$  最多可有  $n(n-1)/2$  对逆置,而平均起来它们每个序列只有  $n(n-1)/4$  对逆置。因此,我们可以说任何一种将比较限制在相邻两个元素之间进行的交换算法的平均时间代价都是  $\Theta(n^2)$ 。

### 8.3 Shell 排序

我们将要分析的下一算法通常称为 Shell 排序(Shell Sort),这是以它的发明者 D.L.Shell 的名字来命名的。也有人称之为缩小增量排序法(diminishing increment sort)。与交换排序不同,Shell 排序在不相邻的记录之间进行比较与交换。Shell 排序利用了插入排序的最佳时间代价特性。Shell 排序试图将待排序序列变成“近似排序”状态(我们称之为“基本有序”,almost sorted),然后再用插入排序来完成最后的排序工作。如果实现得正确,在最差情况下 Shell 排序肯定具有比  $\Theta(n^2)$  好得多的性能。

Shell 排序是这样来分组并排序的:将序列分成子序列,然后分别对子序列进行排序,最后将子序列组合起来。Shell 排序将数组元素分成“虚拟”子序列,每个子序列用插入排序方法进行排序。另一组子序列也是如此选取,然后排序……依此类推。

在执行每一次循环时,Shell 排序把序列分为互不相连的子序列,并且使各个子序列中的元素在整个数组中的间距相同。例如,我们设数组中元素的个数为偶数  $n$ ,是 2 的整数次幂。Shell 排序首先将它分成  $n/2$  个长度为 2 的子序列。如果数组的下标为 0~15 的 16 个记录,那么首先是将它分成 8 个各有两个记录子序列,第一个序列元素的下标是 0 和 8,第二个下标是 1 和 9……依此类推。

第二轮 Shell 排序将处理数量少一些的子序列,但是每个子序列都更长了。对于我们的例子来说会有  $n/4$  个长度为 4 的子序列。因此,第二次分割的第一个子序列中有位于 0,4,8,12 的 4 个元素,第二个子序列的元素位于 1,5,9,13……依此类推。每一个子序列仍然用插入

排序法进行排序。

第三轮将对两个子序列进行排序,其中一个包含原数组中的奇数位上的元素,另一个包含偶数位上的元素。

最后一轮将是一次完全的插入排序。图 8.6 解释了一个具有 16 个元素的数组的 Shell 排序过程,其中元素间距的增量分别为 8, 4, 2 和 1。下面是用 Java 编写的 Shell 排序函数:

```
static void shellsort(Elem[] array) { // Shellsort
    for (int i = array.length/2; i > 2; i /= 2) // For each increment
        for (int j = 0; j < i; j++) // Sort each sublist
            inssort2(array, j, i);
    inssort2(array, 0, 1); // Could call regular inssort here
}

// Modified version of Insertion Sort for varying increments
static void inssort2(Elem[] A, int start, int incr) {
    for (int i = start + incr; i < A.length; i += incr)
        for (int j = i; (j >= incr) && (A[j].key() < A[j - incr].key()); j -= incr)
            DSUtil.swap(A, j, j - incr);
}
```

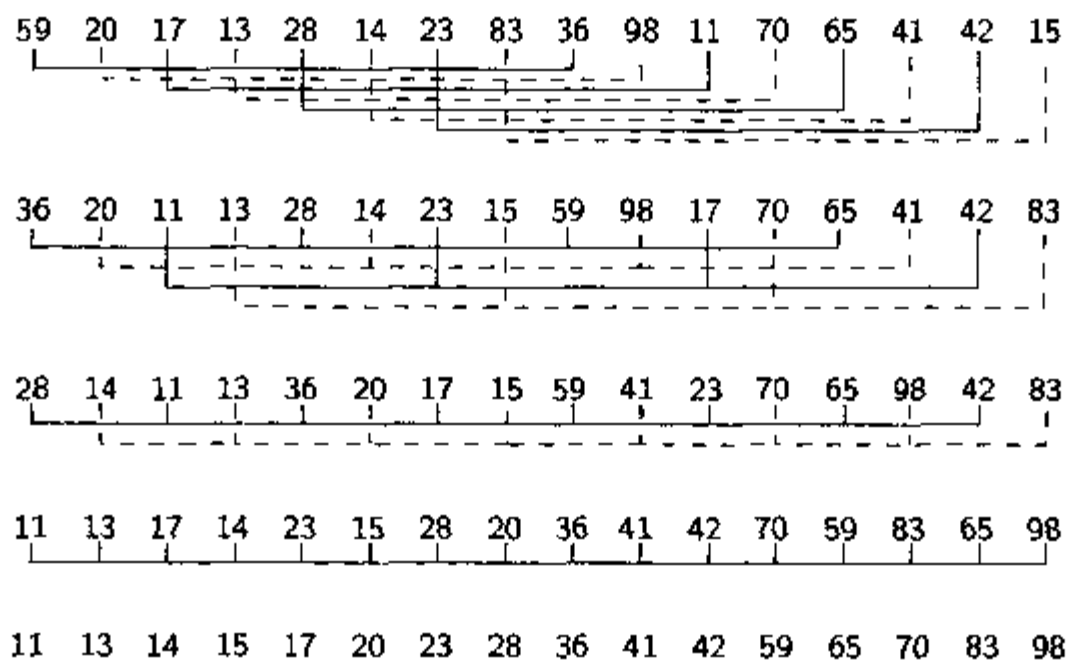


图 8.6 Shell 排序的一个实例。用 4 个回合对 16 个元素进行。第一轮处理 8 个长度为 2 的子序列,第二轮处理 4 个长度为 4 的子序列,第三轮处理 2 个长度各为 8 的子序列。第四轮处理长度为 16 的整个数组

Shell 排序并不关心分割的子序列中元素的间隔(尽管最后的间隔为 1,是一个常规的插入排序)。其目的是经过每次对子序列的处理可以使待排序的数组更加有序。当最后一轮调用插入排序时,数组已经是基本有序的了。

选择适当的增量序列可以使 Shell 排序比其他排序法更有效。一般来说,增量序列为( $2^k$ ,  $2^{k-1}$ , ..., 2, 1)时并没有多大效果,而“增量每次除以 3”所选择的序列如(..., 121, 40, 13, 4, 1)时效果很好。

分析 Shell 排序是很困难的,因此我们必须不加证明地承认 Shell 排序的平均运行时间是

$\Theta(n^{1.5})$ (对于选择“增量每次除以3”递减)。选取其他增量序列可以减少这个上界。因此, Shell 排序确实比插入排序或在 8.2 节中讲到的任何一种运行时间为  $\Theta(n^2)$  的排序算法要快。Shell 排序说明了有时我们可以利用一个算法的特殊性能(如本例中的插入排序), 尽管一般情况下该算法可能会慢得令人难以忍受。

## 8.4 快速排序

快速排序(quick sort)是个恰当的命名, 因为当用得恰到好处时, 它是迄今为止所有内排序算法中最快的。快速排序应用广泛, 典型的应用是 UNIX 系统调用库函数例程中的 qsort 函数。有趣的是, 快速排序往往由于最差时间代价的性能而在某些应用中无法采用。

在我们讲解快速排序之前, 先来考虑一个用二叉树进行排序的实例。你可以将所有结点放到一个 BST(二叉检索树)中, 然后再按照中序方法周游。结果得到一个有序数组。但是这种方法有许多弊端。首先, 为了存储一棵二叉树要占用大量结点空间。其次, 把结点插入 BST 中需要花很多时间。但是这种方法给了我们一些有用的启示: BST 的根结点将树分为两部分, 所有比它小的记录结点都在左子树, 所有比它大的记录结点都位于其右子树。这样 BST 隐含地实现了“分治法”(divide and conquer), 对其左、右子树分别进行处理。快速排序以一种更有效的方式来实现了“分治法”的思想。

快速排序首先选择一个轴值(pivot), 假设输入的数组中有  $k$  个小于轴值的结点, 于是这些结点被放在数组最左边的  $k$  个位置上, 而大于轴值的结点被放在数组最右边的  $n - k$  个位置上。这称为数组的一个分割(partition)。在给定的分割中的点不必被排序, 只要求所有结点都放到了正确的分组位置中。而轴值的位置就是下标  $k$ 。快速排序再对轴值的左右子数组分别进行类似的操作, 其中一个子数组有  $k$  个元素, 而另一个有  $n - k + 1$  个元素。那么这些值是如何进行排序的呢? 由于快速排序是一个好算法, 可以对这两个子数组继续使用快速排序法。下面是一个用 Java 编写的快速排序算法。参数  $i$  与  $j$  分别是待排序子序列左、右两端的下标。第一次调用时的形式是 qsort(array, 0,  $n - 1$ )。

```
static void qsort(Elem[] array, int i, int j) { // Quicksort
    int pivotindex = findpivot(array, i, j); // Pick a pivot
    DSUtil.swap(array, pivotindex, j); // Stick pivot at end
    // k will be the first position in the right subarray
    int k = partition(array, i - 1, j, array[j].key());
    DSUtil.swap(array, k, j); // Put pivot in place
    if ((k - i) > 1) qsort(array, i, k - 1); // Sort left partition
    if ((j - k) > 1) qsort(array, k + 1, j); // Sort right partition
}
```

请注意在调用 partition 之前, 轴值已被放在数组的最后一个位置上了。函数 partition 将返回值  $k$ , 这是分割后的右半部分的起始位置。函数分割一定不能影响到数组中  $j$  所指的记录。然后轴值被放到下标为  $k$  的位置上, 这就是它在最终排好序的数组中的位置。要做到这一点, 我们必须保证在递归调用 qsort 的过程中轴值不再移动。即使是在最差情况下选择了一个不好的轴值, 导致分割出了一个空子数组, 而另一个子数组中起码有  $n - 1$  个记录, 也不能



移动。

选择轴值有多种方法。最简单的方法是使用第一个记录的关键码。但是,如果输入的数组是正序的或者是逆序的,就会将所有结点分到轴值的一边。较好的方法是随机选取轴值。这样可以减少由于原始输入对排序造成的影响。可惜,随机选取轴值的开销较大,我们可以用选取数组中间点的方法代替。下面是一个简单的 findpivot 函数:

```
static int findpivot(Elem[] array, int i, int j)
    { return (i + j)/2; }
```

我们现在来看函数 partition。如果我们事先知道有多少个结点比轴值小,partition 只需将关键码比轴值小的  $k$  个结点放到数组的前  $k$  个位置上,关键码比轴值大的元素放到最后。由于事先并不知道有多少关键码比中心点(轴值)小,我们可以用一种较为巧妙的方法来解决:从数组的两端移动下标,必要时交换记录,直到数组两端的下标相遇为止。下面是用 Java 编写的算法:

```
static int partition(Elem[] array, int l, int r, int pivot) {
    do {
        // Move the bounds inward until they meet
        while (array[ ++l ].key() < pivot); // Move left bound right
        while ((r != 0) && (array[ --r ].key() > pivot)); // Move right bound
        DSUtil.swap(array, l, r); // Swap out-of-place values
    } while (l < r); // Stop when they cross
    DSUtil.swap(array, l, r); // Reverse last, wasted swap
    return l; // Return first position in right partition
}
```

图 8.7 阐明了函数的执行过程。开始时参数  $l$  和  $r$  紧挨着数组的实际边界。每一轮执行外层 do 循环时,都将它们向数组的中间移动,直到它们相遇为止。请注意每次内层的 while 循环时,边界下标都先移动,然后再与轴值进行比较。这是为了保证每个 while 循环都有所进展,即使当最后一次 do 循环中两个被交换的值都等于轴值时也同样处理<sup>①</sup>。另外请注意在第二个 while 循环中  $r$  保持正值。这就保证了当轴值所分割出来的左半部分的长度为 0 时, $r$  不至于会超出数组的下界(下溢出)。函数 partition 返回右半部分的第一个下标的值,因此我们可以确定递归调用 qsort 的子数组的边界。图 8.8 演示了快速排序算法的全部执行过程。

为了分析快速排序函数的执行过程,我们首先分析一下对长度为  $k$  的子数组进行 findpivot 和 partition 操作的例子。显然 findpivot 使用的是常数时间。partition 函数包含 1 个 do 循环和 2 个 while 循环。分割操作的总时间消耗取决于  $l$  和  $r$  这两个下标要向中间移动多久才能相遇。一般说来,如果子数组的长度为  $s$ ,那么两个下标变量一共要走  $s$  步。但是,这并没有直接告诉我们 while 循环要消耗多少时间。do 循环每执行一次指针都向前移动至少一步。每一个 while 循环至少移动一次相应的下标(除非  $r$  下标遇到了数组的左边界,但是这种情况至多发生一次)。因此,do 循环至多执行  $s$  次,总的移动下标的次数最多是  $s$  次,并且每一个 while 循环最多执行  $s$  次。于是最大的时间消耗为  $\Theta(s)$ 。

<sup>①</sup> 这将使得最后一次 do 循环停留在  $l = r + 1$  的条件下,而且发生了多余的交换,必须进行一个处理 DSUtil.swap(array, l, r)来纠正。

初始状态	72	6	57	88	85	42	83	73	48	60	
										r	
第1趟	72	6	57	88	85	42	83	73	48	60	
										r	
第1次交换	48	6	57	88	85	42	83	73	72	60	
										r	
第2趟	48	6	57	88	85	42	83	73	72	60	
						r					
第2次交换	48	6	57	42	85	88	83	73	72	60	
						r					
第3趟	48	6	57	42	85	88	83	73	72	60	
				r							
第3次交换	48	6	57	85	42	88	83	73	72	60	
				r							
反转交换	48	6	57	42		85	88	83	73	72	60

图 8.7 快速排序的分割步骤。第一行给出了数组的初始情况。轴值是被交换到数组最后位置上的 60。do 循环做了三次,每一次将两个下标变量向数组中央移动一格,直到第三个循环结束时相遇。最终一次交换是对最后一趟循环中不必要交换的纠正。于是,左边子数组有 4 个结点,而右边子数组有 6 个。轴值最后置于数组的第 4 个位置上。

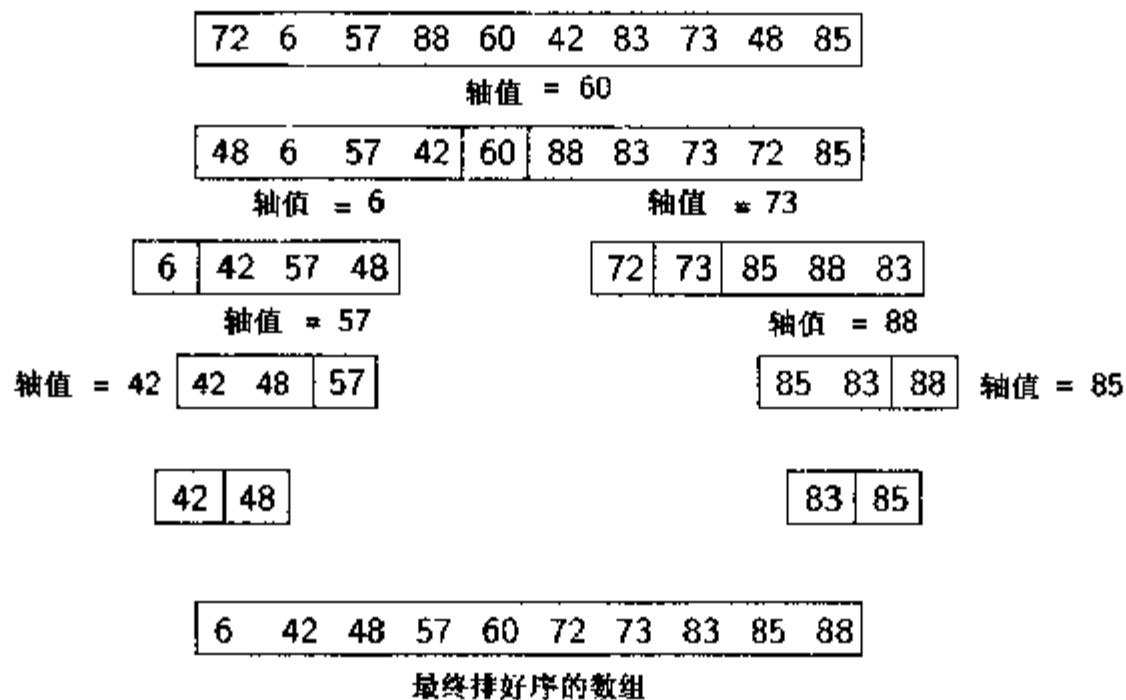


图 8.8 快速排序图示

知道了 findpivot 和 partition 的时间,就可以分析快速排序的时间代价。我们首先分析一下最差情况。最差情况出现在轴值未能很好分割数组时,即一个子数组中没有结点,而另一个数组中有  $n-1$  个结点。在这种情况下,分割过程未能很好地完成任务。因此,下一次处理的子数组只比原数组小 1。如果这种情况发生在每一次分割过程中,那么算法的总时间代价为:

$$T(n) = cn + \frac{1}{n} \sum_{k=0}^{n-1} [T(k) + T(n-1-k)], \quad T(0) = T(1) = c$$

在最差情况下,快速排序的时间代价为  $\Theta(n^2)$ 。这是很糟糕的,并不比起泡排序<sup>①</sup> 更好。那么什么时候会出现这种最差情况呢? 仅仅在每个轴值都未能将数组分割好时会出现。如果数组的记录是随机选取的,那么这种情况并不太可能发生。即使选取中间点作为轴值也不太可能出现。因此,这种最差的情况并不影响快速排序的工作。

当每个轴值都将数组分成相等的两部分时,出现了快速排序的最佳情况。快速排序一直将数组分割下去,直到最后,如图 8.8 所示。在最佳情况下,要分割  $\log n$  次。最上层原始待排序数组有  $n$  个记录,第二层分割的数组是 2 个长度各为  $n/2$  的子数组,第 3 层分割的子数组是 4 个长度为  $n/4$  的子数组……依此类推。因此,在每一层中,所有分割的步骤之和是  $n$ ,于是整个算法的时间代价为  $\Theta(n \log n)$ 。

快速排序的平均情况介于最佳与最差两种情况之间。平均情况应考虑到所有可能的输入情况,对各种情况下所耗费的时间求和,然后除以总的情况数。我们做一个合理的简化设想,在每一次分割时,轴值处于最终排好序的数组中位置的概率是一样的。换句话说,轴值将数组分成长度为 0 和  $n-1$ , 1 和  $n-2$ ……依此类推。这些分组的概率是相等的。

在这种假设下,平均时间代价可以推算为:

这是一个递归公式。 $T(k)$ 是指处理长度为  $k$  的数组时快速排序算法所花费的时间。递归关系将在第 14 章进行讨论,在 14.2.4 小节将解决这个问题。这个等式说明了在长度为  $n$  的数组中,数组被分割为 0 和  $n-1$ , 1 和  $n-2$ ……的概率都是  $1/n$ 。表达式“ $T(k) + T(n-1-k)$ ”是分别递归处理长度  $k$  和  $n-1-k$  的子数组时所用的时间。最前面的项  $cn$  是 `findpivot` 和 `partition` 函数所用的时间。根据公式所推算出来的时间为  $\Theta(n \log n)$ 。因此,快速排序平均要使用  $\Theta(n \log n)$  时间。

快速排序算法是可以改进的(改变常数因子)。最明显的可改进之处与函数 `findpivot` 有关。快速排序的最差情况发生在轴值将数组分成一个空的和一个长度为  $n-1$  的数组时。如果我们再多花些精力寻找一个更好的轴值,这个坏轴值的影响会减少甚至消失。一种较好的方法是“三者取中法”,即取三个随机值的中间一个。用一个随机函数耗时较多,因此较普遍的方法是看当前子数组中第一个、中间一个及最后一个位置的数值。

事实上,当  $n$  很小时,快速排序是很慢的,于是我们还可以从这个方面做一些改进。如果每一次都处理大数组可能不需要管它,也不用管快速排序偶尔对小数组排序时所耗费的时间多长,因为它仍能较快完成。但我们应该注意到快速排序本身就在不断地对小的数组进行排序。这是分治法自然带来的副产品。

一个简单的改进是用能较快处理较小数组的方法来替换快速排序。如插入排序及选择排序。但是,有一种更有效也很简单的优化方法。当快速排序的子数组小于某个长度时,什么也不要做。那些子数组中的数值是无序的,但是,我们知道左边数组的关键码都要小于右边数组的关键码。因此,虽然快速排序只是大致将排序码移到了接近正确的位置,不过数组已是基本有序的了。这样的待排序数组正适合使用插入排序。最后一步仅仅是调用插入排序从而将整个数组排序。经验表明,最好的组合方式是当  $n$ (子数组的长度)减小至 9 或更小值时就选择使用插入排序。

最后想到的缩短运行时间的方法是与递归调用有关的。快速排序本质上是递归的,由于

<sup>①</sup> 我能想到的具有最糟糕结果的排序算法。

每个快速排序操作都要对两个子序列排序,因此无法使用一种简单的方法来转换成等价的循环算法。但是,当需要存储的信息不是很多时,可以使用栈来模拟递归调用,以实现快速排序。事实上,我们没有必要存储子数组的拷贝,只需将子数组的边界存起来。进一步观察,如果注意调整快速排序的递归调用顺序,堆栈的深度可以保持较小。我们也可以将函数 `findpivot` 及 `partition` 的代码变为直接的编码形式嵌入算法中,因为函数调用比直接编码耗时更多。如果按照我们以上建议的那样,不处理长度为 9(或更短)的子数组,不管这个长度规定是多少,已经减少了 80% 到 90% 的函数调用。因此,消除其余的函数调用只能提高有限的速度。图 8.9 是使用栈代替递归进行快速排序,并且把函数 `findpivot` 和 `partition` 改写为直接代码的算法。

```
// Non-recursive Quicksort
static void qsort(Elem[] array, int oi, int oj) {
    int[] Stack = new int[MAXSTACKSIZE]; // Stack for array bounds
    int listsize = oj - oi + 1;
    int top = -1;
    int pivot;
    int pivotindex, l, r;

    Stack[++top] = oi; // Initialize stack
    Stack[++top] = oj;

    while (top > 0) { // While there are unprocessed subarrays
        // Pop Stack
        int j = Stack[top--];
        int i = Stack[top--];

        // Findpivot
        pivotindex = (i + j) / 2;
        pivot = array[pivotindex].key();
        DSUtil.swap(array, pivotindex, j); // Stick pivot at end

        // Partition
        l = i - 1;
        r = j;
        do {
            while (array[++l].key() < pivot);
            while ((r != 0) && (array[--r].key() > pivot));
            DSUtil.swap(array, l, r);
        } while (l < r);
        DSUtil.swap(array, l, r); // Undo final swap
        DSUtil.swap(array, l, j); // Put pivot value in place

        // Put new subarrays onto Stack if they are small
        if ((l - i) > THRESHOLD) { // Left partition
```

```

        Stack[ ++top ] = i;
        Stack[ ++top ] = l - 1;
    }
    if ((j - 1) > THRESHOLD) { // Right partition
        Stack[ ++top ] = l + 1;
        Stack[ ++top ] = j;
    }
}
inssort(array); // Final Insertion Sort
}

```

图 8.9 快速排序的优化。递归调用已被一个栈代替,函数 `findpivot` 和 `partition` 被改写成直接的代码,数组的长度小于给定的阈值 `THRESHOLD` 时将不处理。最后,调用一次插入排序来对整个数组排序,以完成那些短子序列的最后排序

## 8.5 归并排序

归并是一种想法很简单的排序算法,而且不论在理论上还是在实践上,都证明该算法的速度是很快。可惜,它在实际应用中仍有一些问题。像快速排序一样,归并排序(Mergesort)也是基于分治法的。归并排序将一个数组分成两个长度相等的子数组,为每一个子数组排序,然后再将它们合并成一个数组。将两个有序数组合并成一个有序数组称为归并(Merging),归并排序的运行时间并不依赖于输入数组中元素的组合方式,这样,它就避免了快速排序中的最差情况。可是,在某些特殊数组中,归并排序并不一定比快速排序更快。图 8.10 阐释了归并排序的实现过程。下面是一个实现归并排序的伪码框架。

```

List mergesort(List inlist) {
    if (length(inlist) <= 1) return inlist;
    List l1 = half of the items from inlist;
    List l2 = other half of the items from inlist;
    return merge(mergesort(l1), mergesort(l2));
}

```

36	20	17	13	28	14	23	15
20	36	13	17	14	28	15	23
13	17	20	36	14	15	23	28
13	14	15	17	20	23	28	36

图 8.10 归并排序步骤说明。第一行给出的是 8 个待排序的数。归并排序首先将线性表分成 8 个只有一个元素的子线性表,然后对子表进行重组。第二行是进行第一轮归并后四个长度为 2 的数组。第三行是对第二行的子线性表进行第二轮归并后,形成的两个长度为 4 的子线性表。第四行是对第三行的两个子线性表进行归并后,最后形成的一个完成排序的线性表

在讨论如何进行归并排序之前,我们首先考虑一下归并函数 `merge`。将两个已排序的数组合并成为一个有序数组是很简单的。函数 `merge` 首先对两个数组的第一个记录进行比较,并将较小的作为合并数组中的最小值。把这个最小值从它所在的数组中取出并放到输出数组的第一个位置。继续用这种方法,不断比较两个数组中未被处理序列的最前端的元素,并将结果中较小的依次放入输出数组中。

要实现归并排序仍然有一些困难。第一个困难是怎样才能重组线性表。由于归并排序并不需要随机选取中心点,所以可以很好地处理一个单链表。因此,当输入的待排序数据存储在链表中时,归并排序是一个很好的选择。将两个链表实行归并,非常直接了当,因为我们只需把待归并的链表中的第一个结点取出来,然后链接到结果链表的尾部。将输入链表分成两个等长的子链表看来有些困难,但是我们只需将数组分成前后两个部分。然而,尽管事先知道链表的长度,也要周游半个链表以得到后半部分的开始结点。一个不需要知道链表长度的简单方法是交替地将链表的元素分配给两个子链表。链表的第一个结点分配给第一个子链表,第二个结点分配给第二个子链表,第三个结点分配给第一个子链表,第四个结点分配给第二个子链表……依此类推。这需要完全周游输入链表来建立子链表。

当使用归并排序算法为一个数组排序时,如果事先知道数组的边界,确定两个子数组是比较容易的。如果将归并结果放到另一个数组中,那么归并的过程也是很简单的。但是,这将使得归并排序的空间开销是我们上面所讨论的排序方法的两倍,这是归并排序的严重缺陷。不用额外的数组是可以的,但是这样做起来极其困难,因此并不可取。将两个数组归并后放到另一个数组中又要消耗空间。用一个辅助数组来实现归并排序,虽然简单,但是也有其麻烦之处。考虑一下归并排序是如何将待排序的数组分成子数组的,如图 8.10 所示,归并排序一直调用分割的过程,直到子数组的长度为 1。共需要  $\log n$  次递归。这些子数组再归并成为长度为 2 的子数组,然后是长度为 4……依此类推。我们需要避免每一次归并都使用一个新的数组。尽管有一点困难,仍然可以设计一个算法,只使用两个数组轮换进行排序。一个较好的方法是将排好序的子数组首先复制到辅助数组中,然后再将它们归并回原数组。下面是一个用这种方法编写的完整的归并程序。

```
static void mergesort(Elem[] array, Elem[] temp, int l, int r) {
    int mid = (l+r)/2;           // Select midpoint
    if (l == r) return;          // List has one element
    mergesort(array, temp, l, mid); // Mergesort first half
    mergesort(array, temp, mid+1, r); // Mergesort second half
    for (int i = l; i <= r; i++) // Copy subarray to temp
        temp[i] = array[i];
    // Do the merge operation back to array
    int i1 = l; int i2 = mid + 1;
    for (int curr = l; curr <= r; curr++) {
        if (i1 == mid+1)          // Left sublist exhausted
            array[curr] = temp[i2++];
        else if (i2 > r)           // Right sublist exhausted
            array[curr] = temp[i1++];
        else if (temp[i1].key() < temp[i2].key()) // Get smaller value
```

```

        array[curr] = temp[i1++];
    else array[curr] = temp[i2++];
    }
}

```

下面是 R. Sedgewick 发明的一个优化归并排序方法。这个算法十分巧妙,因为它在开始复制时将第二个子数组中元素的顺序颠倒了一下。现在两个子数组从两端开始运行,向中间推进,这就使得这两个子数组的两端互相成为另一个数组的“监视哨”,从而不用像上面的算法那样需要检查子序列被处理完的情况。这个版本也用到了插入排序来处理较短的子数组。

```

static void mergesort(Elem[] array, Elem[] temp, int l, int r) {
    int i, j, k, mid = (l+r)/2; // Select the midpoint
    if (l == r) return; // List has one element
    if ((mid-l) >= THRESHOLD) mergesort(array, temp, l, mid);
    else inssort(array, l, mid-1+1);
    if ((r-mid) > THRESHOLD) mergesort(array, temp, mid+1, r);
    else inssort(array, mid+1, r-mid);
    // Do the merge operation. First, copy 2 halves to temp.
    for (i=l; i<=mid; i++) temp[i] = array[i];
    for (j=1; j<=r-mid; j++) temp[r-j+1] = array[j+mid];
    // Merge sublists back to array
    int a = temp[i].key(); int b = temp[r].key();
    for (i=l, j=r, k=1; k<=r; k++)
        if (a < b) { array[k] = temp[i++]; a = temp[i].key(); }
        else { array[k] = temp[j--]; b = temp[j].key(); }
}

// Insertion Sort for subarrays: sort len elements from index start
static void inssort(Elem[] array, int start, int len) {
    for (int i = start+1; i < start+len; i++) // Insert i'th record
        for (int j = i; (j > start) &&
            (array[j].key() < array[j-1].key())); j--)
            DSUtil.swap(array, j, j-1);
}

```

对归并算法的分析非常直观,尽管它是一个递归程序。设  $i$  为两个要排序子数组的总长度,归并过程要花费  $\Theta(i)$  时间。需要排序的数组一直不断被分成两半,直到子数组长度为 1;然后开始将它们归并为长度为 2 的子数组,然后是长度为 4……依此类推。就像图 8.10 中所示的那样。因此,当被排序元素的数目为  $n$  时,递归的深度为  $\log n$ 。第一层递归可以认为是对一个长度为  $n$  的数组排序,下一层是对两个长度为  $n/2$  的子数组排序,再下一层是对 4 个长度为  $n/4$  的子数组排序,最后一层对  $n$  个长度为 1 的子数组排序。显然,对  $n$  个长度为 1 的子数组归并,需要  $n$  步;对  $n/2$  个长度为 2 的数组归并,需要  $n$  步……依此类推。在所有  $\log n$  层递归中,每一层都需要  $\Theta(n)$  的时间开销,因此总的时间代价为  $\Theta(n \log n)$ 。这个时间

代价并不依赖于待排序数组中数值的相对顺序。因此,这也就是归并排序的最佳、平均、最差的运行时间。

## 8.6 堆排序

我们关于快速排序的讨论,是从利用二叉检索树(BST)进行排序开始的。BST 占用的空间太大,而且由于插入结点需要花费一些时间,它比快速排序和归并排序都慢。如果 BST 不平衡,可能导致最差运行时间为  $\Theta(n^2)$ 。二叉树中子树的平衡与快速排序中对数组的分割很相似。快速排序的轴值与 BST 根结点的值起着相同的作用,左半部分(左子树)的值都小于轴值(根结点值),右半部分(右子树)的值都大于等于轴值(根结点值)。

人们设计了一个基于树结构的更好的排序算法。特别地,我们希望二叉树是平衡的,消耗的存储空间小,并且运算速度快。注意排序这种特殊的应用,一开始就全部给出了应该存储而用于排序的所有值,这就意味着没有必要将结点一次一个地插入二叉树中。

堆排序(HeapSort)基于 5.6 节的堆数据结构。堆排序具有许多优点。整个树是平衡的,而且它的数组实现方式对空间的利用效率也很高,我们可以利用有效的建堆函数 `buildheap` 一次性地把所有值装入数组中。堆排序的最佳、平均、最差执行时间均为  $\Theta(n \log n)$ 。平均情况下它比快速排序要慢常数因子,但是堆排序更适于外排序,处理那些数据集太大而不适于在内存中排序的情况。详见第 9 章。

一个基于“最大值堆”(max-heap)排序算法的思想是很直接的。首先我们用 5.6 节介绍的那种方法将数组转化为一个满足堆定义的序列。然后将堆顶的最大元素取出,再对剩下的数排成堆并取出堆顶数值……如此下去,直到堆为空。应该注意的是每次应将堆顶的最大元素取出放到数组的最后。假设  $n$  个元素存在数组中的 0 到  $n-1$  个位置上。把堆顶元素取出时,应该将它置于数组的第  $n-1$  个位置,这时堆中元素的数目为  $n-1$  个。再按照堆的定义重新排列堆,取出最大值并放入数组的第  $n-2$  个位置。到最后结束时,就排出了一个由小到大排列的数组。这就是为什么要用最大值堆而不用最小值堆。图 8.11 介绍子堆排序的过程。下面是完整的 Java 实现。

```
static void heapsort(Elem[] array) { // Heapsort
    MaxHeap H = new MaxHeap(array, array.length, array.length);
    for (int i = 0; i < array.length; i++) // Now sort
        H.removeMax(); // RemoveMax places max value at end of heap
}
```

因为建堆要用  $\Theta(n)$  时间,并且  $n$  次取堆的最大元素要用  $\Theta(\log n)$  时间,因此整个时间消耗为  $\Theta(n \log n)$ ,这是堆排序的最佳、平均、最差时间代价。尽管有时要比快速排序慢,但是堆排序有其独特的优点。建堆是很快,只要用  $\Theta(n)$  时间,将堆顶元素移走要用  $\Theta(\log n)$  时间。因此,如果希望找到数组中第  $k$  大的元素,可以用  $\Theta(n + k \log n)$  时间。如果  $k$  很小,它的速度要比前面所讲述的方法快得多。7.5.2 小节的 Kruskal 最小支撑树(MST)算法就利用了这个特点,该算法要求按照递增顺序访问带权边,因此应该用最小值堆(min-heap)。而且最小支撑树一旦形成,算法就立即结束,因而只需要对为数很少的边排序。



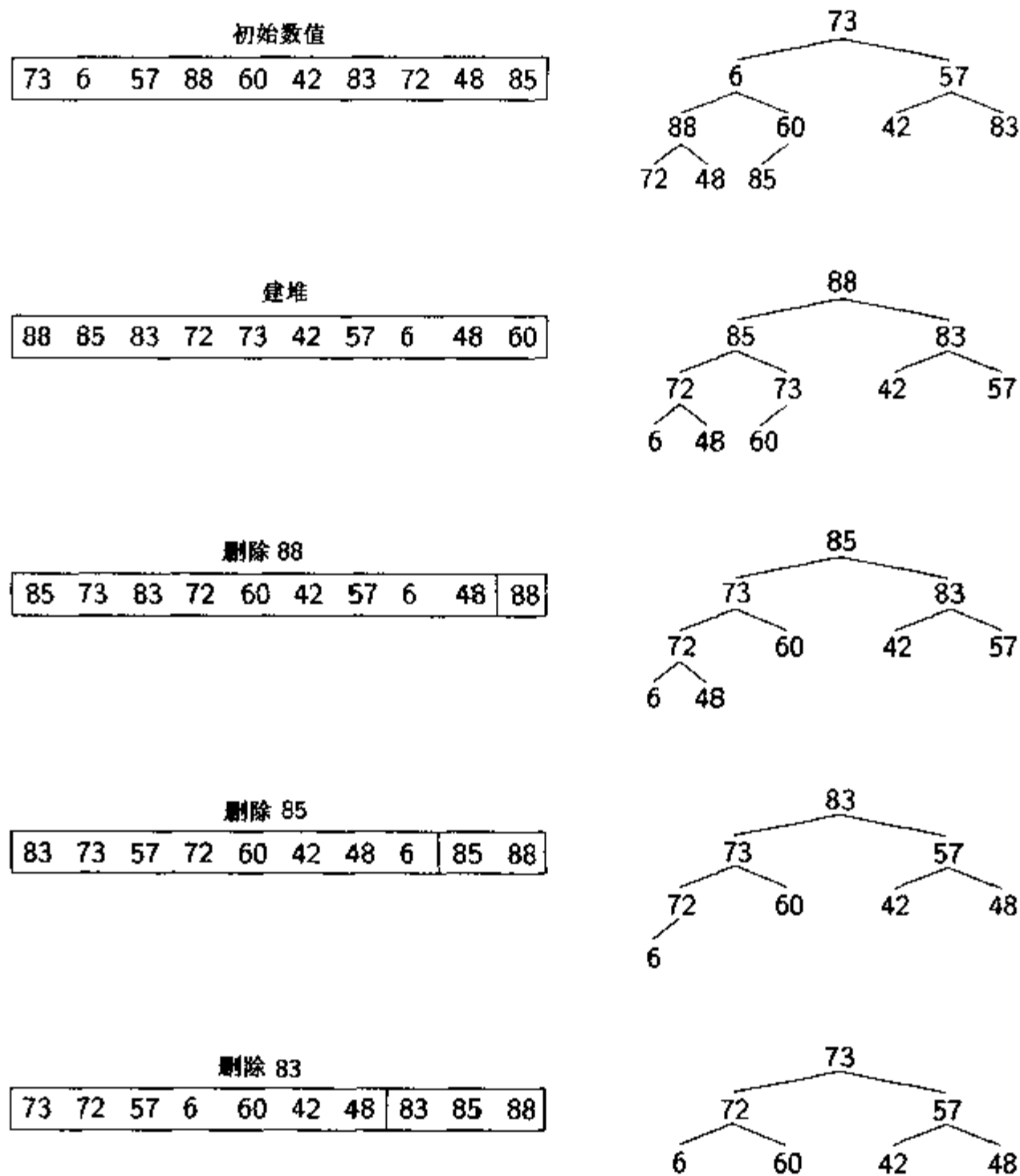


图 8.11 堆排序的步骤。第一行是初始数组。第二行是建立堆后的情形。第三行给出了执行操作 `deletemax` 取出最大值 88 后的情形。第四行显示了第二次执行 `deletemax` 取出关键码 85 后的情形。第五行是第三次 `deletemax` 取出关键码 83 后的状态,此时数组中最后三个位置按增序存放着数组中最大的三个值。堆排序照此反复进行下去,直到整个数组有序

## 8.7 分配排序和基数排序

下面是 3.8 节中一段从 0 到  $n-1$  的循环。

```
for (i = 0; i < n; i++)
    B[A[i].key()] = A[i];
```

这里的关键码用来确定一个记录在排序中的最终位置。关键码用于将记录放入“盒子”里,是分配排序(Binsort)的一种最基本的例子。这种算法很棒,无论初始数组的关键码顺序如何,都只用  $\Theta(n)$  时间代价。这比我们目前所讨论过的所有算法都要快许多。但惟一的问题是它只能对 0 到  $n-1$  的一个排列进行排序。

有许多方法可以对这种简单分配排序算法进行扩展,使它更有实用价值。最简单的方法

是允许关键码重复。使数组元素成为可变长的盒子,也就是使数组  $B$  中的每个元素成为一个链表的头结点,于是所有具有关键码  $i$  的记录就被放于  $B[i]$  的盒子里。另一个扩展是允许关键码大于  $n$ 。例如,一个具有  $n$  个记录的数组可以有  $1 \sim 2n$  范围内的关键码,惟一的要求是每个记录在数组  $B$  中都有一个相应的盒子。扩展的分配排序如下:

```
void binsort(ELEM *A, int n) {
    list B[MaxKeyValue];
    for (i = 0; i < n; i++) B[A[i].key()].append(A[i]);
    for (i = 0; i < MaxKeyValue; i++)
        for (B[i].first(); B[i].isInList(); B[i].next())
            output(B[i].currValue());
}
```

这个分配排序算法可以对关键码处于 0 到  $\text{MaxKeyValue} - 1$  之间的序列进行排序。所需工作量只是要花时间将各个记录放入盒子中,然后再从盒子中收集所有记录。因此,需要处理每个记录两次,每次  $\Theta(n)$ 。

可惜,有一个虽小但是很关键的问题忘记分析了。在分配排序中,必须检查每个盒子来确认里面是否有记录。不管实际上哪个盒子中有记录,算法必须处理  $\text{MaxKeyValue}$  (最大关键码)个盒子。如果  $\text{MaxKeyValue}$  比  $n$  小,那么这并不成问题。如果  $\text{MaxKeyValue}$  为  $n^2$ ,时间代价就是  $\Theta(n + n^2) = \Theta(n^2)$ 。这就导致了一种很差的排序算法。如果  $n$  与  $\text{MaxKeyValue}$  相差更加悬殊,算法还会更差。另外,大的关键码范围需要使用较大的数组  $B$  来存储,因此即使是扩充的分配排序也只适用于有限的关键码范围。

分配排序的一个简单扩展是桶式排序(bucket sort)。每一个盒子并非仅与一个关键码相联系,而是与一组关键码有关。桶式排序将记录放入“桶”中,然后借助于其他排序技术对每个“桶”中的记录排序。其目的是用开销相对较小的分“桶”处理技术,只分配较少的记录给每个“桶”,然后用比较快的“收尾排序”(cleanup sort)来对每一桶中的记录进行排序。

有一种方法可以尽量减少盒子的数目并缩短排序时间。考虑关键码范围为  $0 \sim 99$  的一个序列。假如有 10 个盒子,首先我们可以取其关键码模 10 并赋给盒子中心记录,这样每个关键码都以它的个位为标准放到 10 个不同的盒子中。然后,我们按照顺序从盒子中收集这些记录,并且按照最高位对它们进行排序。换句话说,将数组中记录  $i$  按照  $\text{key}(A[i])/10$  的值再放入到盒子里。图 8.12 展示了这种算法。

在这个例子中,盒子数  $r = 10$ ,待排序的关键码数  $n = 12$ ,关键码值介于  $0 \sim r^2 - 1$  之间。由于我们对每个记录和每个盒子的处理时间是常数,因此总的计算时间为  $O(n)$ 。这是对简单分配排序的一个很大的改进,因为简单分配排序需要长度为关键码值范围的数组。注意到例子中用  $r = 10$  来使分配排序算法易于观察,各个元素以个位的大小顺序放入盒子中。然后再按照较左边一位的大小重排……依此类推,直至最左边的一位处理完毕。其实用多少个盒子都可以。这是一个基数排序(Radix Sort)的例子,之所以这样说,是因为它基于对关键码的某个位来分配盒子。这个排序算法可以扩展到对任意位长的关键码进行比较,当然关键码的数目可以有任意多个。我们从最右边的位(个位)开始,到最左边的位(最高位)为止,每次按照关键码某位的数字把它分配到盒子中。如果关键码有  $k$  位数,那么就需要  $k$  次对盒子分配关键码元素。

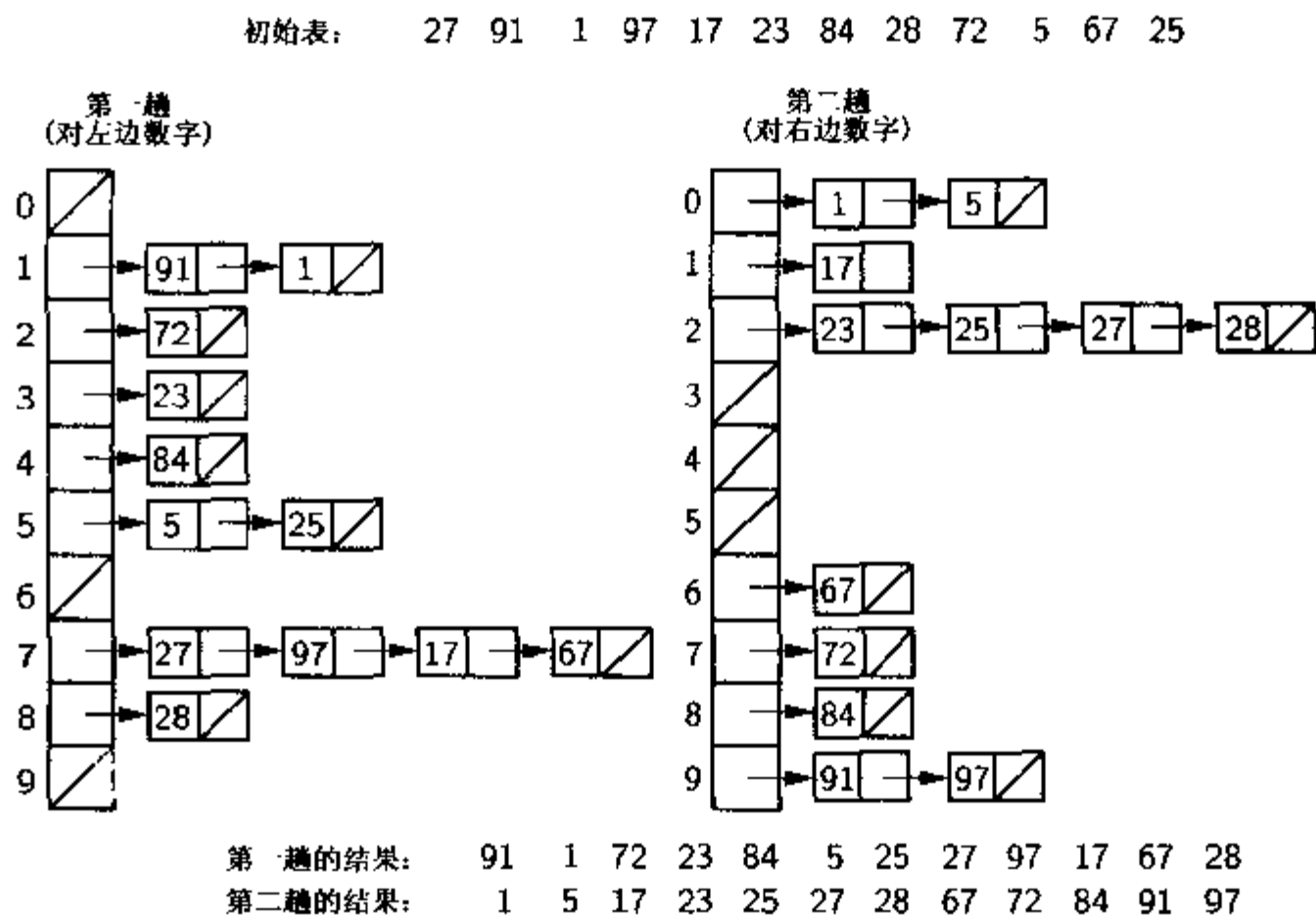


图 8.12 一个基数排序(Radix Sort)的例子。此例中的数字为两位数,因此要用两次分配过程来完成

正如归并排序一样,基数排序也有一个棘手的问题。人们愿意对数组排序,以避免链表的处理。如果事先知道每个盒子里有多少个元素,那么我们可以使用一个长度为  $n$  的辅助数组。例如,如果在第一轮中第 0 个盒子接收 3 个记录,第 1 个盒子接收 5 个记录,那么我们可以简单地把前 3 个位置空出来留给第 0 个盒子使用,把接着的 5 个位置空出来留给第 1 个盒子使用。下面的 Java 算法实现了这个思想。在每一轮分配结束时,记录都被复制回原数组。

```
static void radix(Elem[] A, Elem[] B, int k, int r, int[] count) {
    // Count[i] stores number of records in bin[i]
    int i, j, rtok;

    for (i = 0, rtok = 1; i < k; i++, rtok *= r) { // For k digits
        for (j = 0; j < r; j++) count[j] = 0; // Initialize count

        // Count the number of records for each bin on this pass
        for (j = 0; j < A.length; j++) count[(A[j].key()/rtok) % r]++;

        // Index B: count[j] will be index for last slot of bin j.
        for (j = 1; j < r; j++) count[j] = count[j-1] + count[j];

        // Put records into bins working from bottom of each bin.
        // Since bins fill from bottom, j counts downwards
        for (j = A.length-1; j >= 0; j--)
            B[--count[(A[j].key()/rtok) % r]] = A[j];
    }
}
```

```

for (j = 0; j < A.length; j++) A[j] = B[j]; // Copy B back to A
}
}

```

第一个内部 for 循环是初始化数组 count。第二个内部 for 循环计算要放入每个盒子的记录数目。第三个内部 for 循环将 count 数组中的值设置为该盒子在数组 B 中的下标位置。请注意 count[j] 中的下标值是第 j 个盒子的最后一个下标, 是从下到上往盒子中放置关键码的。第四个内部 for 循环把记录分配到数组 B 的盒子中。最后一个内部 for 循环简单地把记录复制到 A, 以便下一轮分配。变量 rtok 存储数值  $r^k$ , 用于截取某位数值的计算。图 8.13 展示了该算法处理图 8.12 中输入数据的过程。



图 8.13 对于图 8.12 中的输入, 函数 radix 的运行示例。第一行显示输入数组中的初始数据。第二行是数组 count 计算每个盒子中记录数之后的值。第三行是存于数组 count 中的下标值。例如, count[0] 为 0, 表示第 0 个盒子中没有记录; count[1] 为 2, 表示数组 B 的第 0 个和第 1 个位置将存放第 1 个盒子中的记录; count[2] 为 3, 表示数组 B 的第 2 个位置将存放第 2 个盒子中的记录(仅有一个); count[7] 为 11, 表示数组 B 的第 7 个到第 10 个位置将存放第 7 个盒子的 4 个记录。第四行显示第一趟基数排序后的结果。第五至第七行展示了第二趟基数排序的类似步骤

对于  $n$  个数据的序列, 假设基数为  $r$ , 这个算法需要  $k$  趟分配工作。每趟分配的时间为  $\Theta(n + r)$ , 因此总时间开销为  $\Theta(nk + rk)$ 。这个算法与  $n$  的关系如何呢? 因为  $r$  是基数, 它是比较小的。可以用 2 或者 10 作为基数, 对于字符串的排序, 采用 26 作为基数比较好(因为有 26 个

英文字母)。为了考查算法的渐近复杂性,可以把  $r$  看成一个常数值,而忽视它的影响。变量  $k$  与关键码长度有关,它是以  $r$  为基数时关键码可能具有的最大位数。在一些应用中我们可以认为  $k$  是有限的,因此也可以把它看成是常数。在这种假设下,基数排序的最佳、平均、最差时间代价都是  $\Theta(n)$ ,这使得基数排序成为我们所讨论过的具有最好渐近复杂性的排序算法。

把  $k$  看成常数合理吗?  $k$  与  $n$  之间有关系吗? 如果关键码的长度是有限的,而且关键码允许重复,那么在  $k$  与  $n$  之间也许没有任何关系。但是,考虑没有重复关键码的情况,如果有  $n$  个互不相同的关键码,那么将需要  $n$  个不同的编码来表示它们。为了表示  $n$  个不同的编码值,最少需要  $\log_2 n$  个二进制位(binary bit)。当基数为  $r$  个时,我们需要  $\log_r n$  位。总的来说,至少需要  $\Omega(\log n)$  位(在常数因子范围内)来存储这  $n$  个不同的值。换句话说,  $k$  在  $\Omega(\log n)$  中。因此,基数排序要耗用  $\Omega(n \log n)$  的时间代价。

关键码可能长得多,  $\log_r n$  仅仅是对于  $n$  个不同排序码的最佳情况。因此,用  $\log_r n$  来估算  $k$  就未免有些太乐观了。但是这种分析的合理之处在于,对于具有  $n$  个不同关键码的一般情况,基数排序的最佳时间代价为  $\Omega(n \log n)$ 。

基数排序还有待改进,可以使基数  $r$  尽量大些。考虑整型关键码的情况,令  $r = 2^i$ ,  $i$  为某个整数。换句话说,  $r$  的大小与每一趟分配时可以处理的位数(bits)有关。如果  $r$  增加一倍,则分配的趟数可以减少一半。当处理整型关键码时,令  $r = 256 = 2^8$ ,即一趟分配可以处理 8 个二进制位。那么处理一个 32 位的关键码只需要 4 趟分配。对于大多数计算机来说可以采用  $r = 2^{16} = 64K$ ,只需两趟分配。当然,这需要一个长度为 64K 的 count 数组。只有当待排序的记录接近或超过 64K 以上时,算法的性能才是较好的。换句话说,要使基数排序的效率更高,一定要仔细研究记录数目和关键码长度。在许多基数排序的应用中,都可以调整  $r$  的值而获得较好的性能。

对于某一位数值,基数排序要把它确定地分配到若干个盒子中的一个。基数排序依赖于这种分配能力,也依赖于随机访问盒子的能力。因此,基数排序对一些数据类型来说是较难实现的。例如,如果关键码的数据类型为实型或者不等长的字符串,就会需要一些特殊处理。特别是基数排序中在确定实数的“最后一个数字”或者变长字符串的“最后一个字符”时,需要做一些处理。对于这些情形,Trie 数据结构更合适些,这将在 13.1 节详细论述。

讨论到现在,一些敏锐的读者可能会对产生怀疑,前面所假设的关键码比较为常数时间是正确的吗? 如果关键码是存于数组中的普通整数,比如说是一个长整数变量,那么它的大小与  $n$  相比会怎样呢? 在实际的排序应用中,32 (Java 中 int 类型所占的二进制位数)几乎都比  $\log n$  大。就这一点来说,比较两个较长整数的时间代价为  $\Omega(\log n)$ 。

计算机都是在特定长度的存储单元中进行算术运算的,比如采用一个 32 位的字长来运算。不管变量有多大,关键码的比较都采用本机字长来进行,因此比较运算花费常数时间。事实上 32 往往比  $\log n$  大很多。有些比较时间的差异是由于观察者的观看差异引起的。对计算机体系结构的门电路级来说,是比较不同的位。因此,事实上对于多数计算机来说,整数的比较花费常数时间的假设是正确的。相反,基数排序必须对关键码值进行几次算术运算,计算的次数与关键码长度相关。因此,处理  $n$  个不同关键码时,基数排序实际的时间代价为  $\Theta(n \log n)$ 。

## 8.8 对各种排序算法的实验比较

哪一种算法是最快的呢? 渐近复杂性分析方法可以让我们区分  $\Theta(n^2)$  算法和  $\Theta(n \log n)$  算法, 但是它不能区分具有相同渐近复杂性的算法。渐近分析也没有指出对于比较小的序列排序, 哪一种算法是最好的。为了回答这些问题, 我们必须根据测试结果进行经验分析。

图 8.14 及 8.15 给出了本章所述各种算法的时间开销。其中包括: 插入排序、起泡排序、选择排序、Shell 排序、快速排序、归并排序、堆排序和基数排序。快速排序给出了两种算法的比较: 8.4 节的基本方法, 以及非递归且不再分割长度小于 10 的子序列的方法。归并排序展示了 8.5 节中的基本方法, 以及改进了的优化算法, 优化算法的另外一个改进是对于长度小于 10 的子序列调用选择排序进行处理。

算法	10	100	1,000	10,000	正序	逆序
插入排序	0.10	9.5	957.9	98 086	22	192 656
起泡排序	0.13	1463	1470.3	157 230	101 926	193 337
选择排序	0.11	9.9	1018.9	104.897	102 398	102 711
Shell 排序	0.09	25	45.6	829	247	484
快速排序	0.15	1.8	23.6	291	176	193
快速排序/O	0.10	1.6	20.9	274	127	170
归并排序	0.12	2.4	36.8	505	395	390
归并排序/O	0.08	1.8	28.0	390	302	450
堆排序	-	50.0	60.0	880	980	940
基数排序/1	0.87	8.6	89.5	939	923	923
基数排序/2	0.44	4.3	44.5	478	461	462
基数排序/4	0.23	2.3	2265	236	231	236
基数排序/8	0.19	1.2	11.5	115	120	121

图 8.14 在 IBM 兼容 PC 机上, 运行 Windows 95 操作系统和 Visual J++ 编译器时, 各排序算法的实验比较数据。时间单位为毫秒。对于每种排序算法, 所给时间是分别对 10、100、1 000、10 000 和 30 000 个记录排序的运行时间。还给出了对升序(“正序”所表明的那列)的 10 000 个记录和降序(“逆序”所表明的那列)的 10 000 个记录分别排序的运行时间。快速排序和归并排序都给出了普通及优化算法。基数排序给出了每趟处理 1、4 和 8 位的不同算法

每个算法的输入数据都是随机的整数数组。这对于某些排序算法有所影响。例如, 当一个记录的空间较小时, 选择排序显示不出它的优势。基数排序方法自然利用了这种短关键码的好处, 并且没有完成必要的对更长关键码的考察。事实上基数排序假设关键码值小于  $2^{16}$ 。如果要支持 32 位整数, 那么基数排序所花的时间将加倍。图 8.14 中给出在 IBM 兼容 PC 机上, 当软件环境为 Windows 95 操作系统和 Visual J++ 编译器时, 应用各种排序算法分别对长度为 10、100、1 000、10 000 和 30 000 的序列排序所花费运行时间。图 8.15 中给出了相应序列在 Sun SPARC 工作站的 UNIX 操作系统环境下的运行时间。在每个图中的最后两列是长度为 10 000 的序列分别为正序(已排序)及逆序(逆排序)时的情形。这两列显示出对某些算法的最佳情形, 以及对某些算法的最差情形。从这两列也可以看出, 输入的顺序对某些算法几乎没有影响。

算法	10	100	1 000	10 000	正序	逆序
插入排序	0.66	65.9	6423	661 711	129	1 176 829
起泡排序	0.90	85.5	8447	1 068 268	673 057	1 411 337
选择排序	0.73	67.4	6678	668 056	686 090	682 790
Shell 排序	0.62	18.5	321	5 593	1 717	3 259
快速排序	0.92	12.7	169	1 836	1 187	1 384
快速排序/O	0.65	10.7	141	1 781	907	1 165
归并排序	0.76	16.8	234	3 231	2 493	2 446
归并排序/O	0.53	11.8	189	2 649	1 955	2 985
堆排序	-	41.0	565	7 973	9 151	8 013
基数排序/1	7.40	67.6	794	6 895	6 850	6 898
基数排序/2	3.90	33.6	331	3 628	3 340	3 266
基数排序/4	2.10	18.7	160	1 678	1 693	1 648
基数排序/8	4.10	11.5	97	808	815	805

图 8.15 在 Sun SPARC 工作站上运行 UNIX 系统时,图 8.14 中所示  
各算法所用的时间。时间单位为毫秒

这些数据给出了一些有趣的结果。正如我们想像的那样,快速排序具有一致的好性能。出乎意料的是,快速排序的优化版只比原始版稍好。这种优化与语言相关,等价的 C/C++ 优化版节省将近四分之一的的时间。优化的归并排序显然比非优化的好。堆排序在两种机器中都很慢,但是在 UNIX 上要更慢些。对于短序列,插入排序与快速排序和归并排序一样好。

对于长序列,每趟处理 8 位的基数排序是最快的。请注意这里关键码值小于 16 位的二进制数值,如果用 32 位的关键码,那么基数排序所花的时间将加倍,而其他排序方法所用的时间并无显著改变。

## 8.9 排序问题的下限

本书有许多算法的分析,这些分析基本上明确了算法在最差和平均情况下的上限及下限。对于迄今为止所讲到的算法,算法分析都比较简单。本节有一个更为艰巨的分析,分析一个问题的时间代价(cost of a problem),而不是某个算法的时间代价。一个问题的上限可以定义为已知算法中速度最快的渐近时间代价;其下限解决这个问题所有算法的最佳可能效率,包括那些尚未设计出来的算法。一旦问题的上限与下限相同,我们就知道,从渐近分析的意义上说,不可能有更有有效的算法了。

一种估计问题下限的简单方法是计算必须读入的输入长度及必须写出的输出长度。任何算法的时间代价当然都不可能小于它的 I/O 时间,于是我们就知道没有任何排序算法能够将时间下限降到  $\Omega(n)$  以下,因为算法至少要花  $n$  步来读入  $n$  个待排序的数据,输出排序后的  $n$  个结果。就我们实际所知,我们可以说排序的时间在  $\Omega(n)$  到  $O(n \log n)$  之间。

计算机专家们花了许多时间和精力,希望能够找到一种更快的算法,但是没有有一个算法能

够在平均及最差情况下比  $O(n \log n)$  快。那么我们有必要继续找更快的算法吗? 或者是找到一个严格的下限, 证明不可能有更快的排序算法?

本节介绍了计算机科学中最重要并且也是最有用的论证之一: 没有任何一种基于关键码比较的排序算法可以将最差执行时间降低到  $\Omega(n \log n)$  以下。主要有以下三个原因。首先, 广泛应用的优化排序算法已令人满意, 这就是说没有必要死死地追寻  $O(n)$  那样快的算法(起码没有人去寻求基于关键码比较的更好算法)。其次, 这个论证也是证明问题下限的重要方法之一。也就是说, 这个证明提供了一个实例, 说明了下限要比仅仅估计  $1/O$  时间的  $\Omega(n)$  严格得多。因此, 它可以作为证明其他问题下限时时的一个参考模式。最后, 知道了排序问题的下限, 也就知道了可以用来解决排序问题的其他问题的下限。从一个问题的渐近上限、下限推导出其他问题的上限、下限, 这种方法叫做归约, 这个概念将在第 15 章介绍。

除了基数排序及分配排序以外, 本章所有的排序算法都决定于两个关键码的直接比较。例如, 插入排序不断地比较待插入关键码与数组中元素的大小关系, 直到找到正确位置。相反, 基数排序并没有直接比较关键码值, 而是取决于关键码值中各位数字的值。因此有可能使排序并不只是基于关键码的比较。当然, 基数排序最后并不比基于比较的排序算法更有效。因此, 实验数据表明, 基于比较的排序算法是较好的<sup>①</sup>。

下面将证明: 在最差情况下, 任何一种基于比较的算法都需要  $\Omega(n \log n)$  时间代价。首先, 可以发现, 所有的比较判断都可以用一棵二叉树来模拟。也就是说, 所有基于比较的算法都可以归结为一棵二叉树, 树的一个内部结点对应于一个比较。其次, 二叉树树叶数目最小值是  $n$  的阶乘。最后, 具有  $n!$  个树叶的二叉树的最小深度是  $\Omega(n \log n)$ 。

在证明  $\Omega(n \log n)$  是下限之前, 我们首先应该讲一下判定树(decision tree)。一棵判定树是可以模拟任何判定算法处理过程的一颗二叉树。每个判断都是二叉树的一个分支。为了建立排序算法的判定树, 我们把所有对关键码值的比较看作是一个判断。当比较两个关键码时, 如果第一个小于第二个, 那么第一个关键码就作为判定树的左分支; 否则如果第一个大于第二个, 那么第一个关键码就作为判定树的右分支。

图 8.16 给出了插入排序的一棵判定树。第一个输入的是  $X$ , 第二个是  $Y$ , 第三个是  $Z$ 。初始时它们依次存放于输入数组  $A$  的 0, 1, 2 位置。让我们考虑一下可能的输出。刚开始时我们并不知道这 3 个值在排序后的输出数组中将处于什么位置。正确的输出结果可能是这 3 个值全排列中的任何一个。对于这 3 个值, 有  $n! = 6$  个排列。因此在根结点中存有 6 个排列, 它们都可能是算法的最后输出结果。

当  $n = 3$  时, 第一次比较在数组中的第二个元素( $Y$ )与数组中的第一个元素( $X$ )之间进行。有两种可能: 要么  $Y$  比  $X$  小, 要么  $Y$  不小于  $X$ 。这种判定在二叉树第一层分支分开。如果  $Y$  小于  $X$ , 那么取左分支, 并且在最终输出中  $Y$  应该在  $X$  之前。只有三个排列满足  $Y$  出现于  $X$  之前的条件, 于是左子树的根结点中列出了  $YXZ, YZX, ZYX$ 。同理, 如果  $Y$  不小于  $X$ , 那么要取右子树, 只有三个排列满足  $Y$  出现于  $X$  之后的条件, 分别为  $XYZ, XZY, ZXY$ 。这些在右子树的根结点中列出。

让我们假设  $Y$  小于  $X$ , 因此取左子树。在这种情况下, 插入排序交换这两个值, 数组中存

<sup>①</sup> 实际结果比这句话的论断还强些。实际上, 基数排序也基于比较, 因而也可以用本节介绍的基数来建立模型。结论是甚至于像基数排序这样的算法在平均情况下的下限也是  $\Omega(n \log n)$ 。



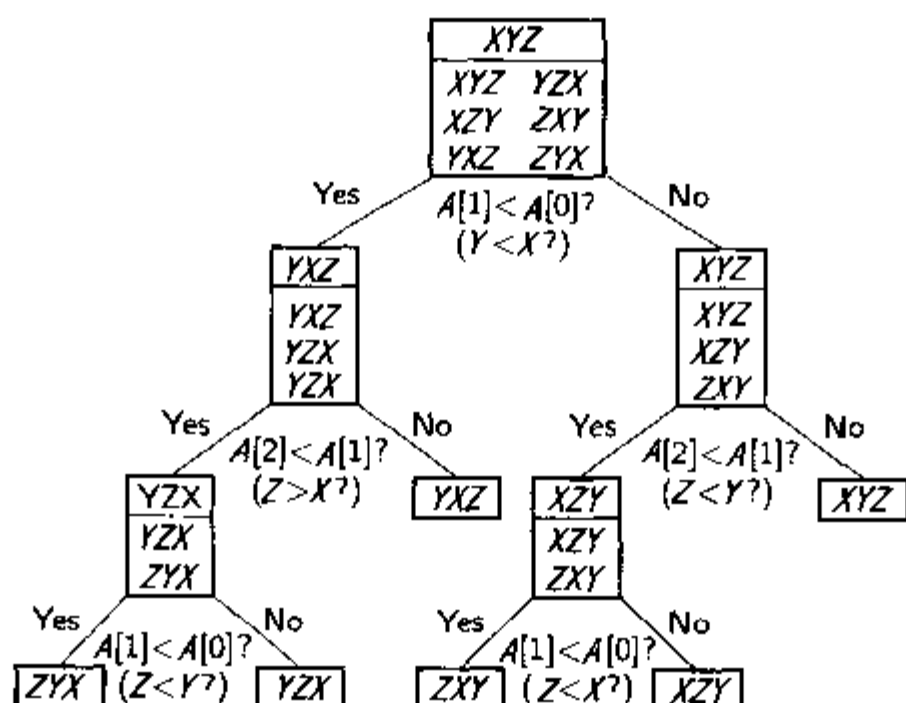


图 8.16 插入排序的判定树,对三个值  $X$ 、 $Y$ 、 $Z$  进行排序,初始时它们分别存于输入数组  $A$  的 0、1 和 2 三个位置中

储着  $YXZ$ 。此时图 8.16 根结点左右方框中的横线上标明为  $YXZ$ 。下一步是比较数组中第三个与第二个的关系(此时是  $Z$  与  $X$  相比)。又有两种可能:如果  $Z$  小于  $X$ ,那么应该交换这两个值,取左分支;否则  $Z$  不小于  $X$ ,那么插入排序完成了,取右分支。注意到右分支只有一个叶结点,而叶结点中只有一个排列顺序  $YZX$ 。这说明如果取这条路径,只能达到这个结点,也只能得到一个结果。换句话说,插入排序已经找到了原始输入的惟一排列顺序,从而导出了排序结果。同样,如果第二步取的是左分支,那么第三次比较无论结果如何,都只有一个排列顺序。于是,插入排序也找到了排序结果。

无论输入数据有多少,所有基于比较的排序都可以用一个判定树来模拟。于是,算法可以看成是寻找正确的顺序。我们可以这样来看待每个基于比较的排序:基于对关键码比较的结果,再取树的分支,直至到达一个具有惟一排列顺序的结点为止。

那么判定树是如何表现最差排序时间的呢?判定树显示了对于一个给定的输入大小,算法对所有可能输入所作出的判断。任何一个从树根到叶的路径都表示算法的一个可能的判断过程。最深结点的高度,表示算法得到一个结果所需要的最长判断路径。

有许多基于比较的排序算法,每个算法都有其特定的判定树。有些判定树较为平衡,而有些可能是不平衡的。有些树比别的树结点多(那些多余的结点可能是作了无用的比较)。事实上,一个较差的排序算法可能会有一个结点很多的判定树,树也很高。没有方法知道最差的排序算法有多慢。我们感兴趣的只是最好的排序算法在最差情况下的最小时间代价。换句话说,我们想知道对于所有算法来说,树中最深结点的可能最小深度是多少。

最深结点的最小深度依赖于树中结点的数目。我们当然希望将树中的结点“向上提”,但是这些结点之上的树空间是有限的。高度为 1 的树只能放 1 个结点(根结点),高度为 2 的只能放 3 个结点,高度为 3 的只能放 7 个,等等。

**需记住的事实:**一棵高为  $n$  的树最多可以放  $2^n - 1$  个结点。等价地,  $n$  个结点的树至少要  $\log(n + 1)$  层。

那么对于某种基于比较的排序算法来说,处理  $n$  个结点,其判定树中有多少个结点呢?

由于排序算法需要决定输入的排列中哪一个将对应于排好序的序列,于是所有的排序算法对于每一个排列都要有一个树叶结点。 $n$ 个数有 $n!$ 个排列(参见2.2节)。

既然至少要有 $n!$ 个结点,我们知道树至少有 $\Omega(n \log n)$ 层。根据2.2节的分析,知道 $\log(n!)$ 应为 $\Omega(n \log n)$ 。对于任何一种基于比较的排序算法来说,判定树有 $\Omega(n \log n)$ 层结点。于是,在最差情况下,这类问题的任何算法都需要 $\Omega(n \log n)$ 次比较。

在最差情况下任何排序算法需要 $\Omega(n \log n)$ 次比较,因此最差情况时间代价就是 $\Omega(n \log n)$ 。既然所有排序算法所需要的运行时间是 $\Omega(n \log n)$ ,那么排序问题需要的运行时间就是 $\Omega(n \log n)$ 。我们已经知道所有排序算法都需要 $O(n \log n)$ 的运行时间,因此可以推导出排序问题需要 $\Theta(n \log n)$ 的运行时间。从理论上说,我们知道不基于比较的排序算法可以对现有的 $\Theta(n \log n)$ 排序算法进行改进,这种改进有可能不只是常数因子。

## 8.10 深入学习导读

Donald E. Knuth 的《Sorting and Searching》[Knu81]有关于排序的详尽讨论。该书对许多细节进行了探讨,包括对 $n$ 很小的情况,以及特殊用途的排序网络。这本书是对排序问题的全面(尽管有些过时)探讨。更新的关于快速排序及其全部优化的评估,请参阅 Robert Sedgewick 的《Quicksort》[Sed80]。Sedgewick 的《Algorithms》[Sed88]讨论了这里所讨论的所有排序算法,并且注重于其有效实现。

尽管 $\Omega(n \log n)$ 是排序的理论下限,但是在很多情况下输入的数据往往都是接近排序的,因此利用这个特点也有可能加速排序过程。一个简单的例子是插入排序的最佳执行时间。运行时间受输入数据有序程度影响较大的排序算法称为可适应(adaptive)排序算法。关于可适应排序算法,请参阅 Estivill-Castro 和 Wood 合写的文章“A Survey of Adaptive Sorting algorithms”[ECW92]。

## 8.11 习题

- 8.1 用归纳法证明:插入排序总可以把一个输入数组排好序。
- 8.2 编写一个处理整数关键码的插入排序。下面是条件:输入是一个栈(不是数组),并且程序中只允许使用一定的整数及栈。结束时排序结果放在栈中,栈顶元素最小。在最差情况下,算法的执行时间为 $\Theta(n^2)$ 。
- 8.3 起泡排序的实现中有如下循环:

```
for (int j = n - 1; j > i; j--);
```

考虑一下将它换成以下语句的影响:

```
for (int j = n - 1; j > 0; j--);
```

新的实现还能正常执行吗?这种改变会影响到程序的渐近复杂性吗?这个改变对运行时间有什么影响?

- 8.4 当实现插入排序时,二分法检索可以用来查找第 $i$ 个元素在前 $i-1$ 个元素中的可能插入位置。为什么使用二分法检索不能改善插入排序的渐近运行时间?

- 8.5 图 8.5 中给出了选择排序的最少交换次数为  $\Theta(n)$ 。因为算法并不检查第  $i$  个元素是否已经在第  $i$  个位置;这就是说,那可能带来不必要的交换。
- (a)改进算法使之没有不必要的交换。
  - (b)你认为这种改进能加快速度吗?
  - (c)编写两个程序验证一下原始插入排序算法和改进算法的运行时间。哪个算法实际上更快?
- 8.6 当待排序列存在多个具有相同关键码的记录时,经过排序后,如果这些记录的相对顺序仍然保持不变,则这个排序算法称为稳定的。本章中所讲的算法中哪些是稳定的,哪些不是,并说明理由。如果稍微改变一下就可以使算法变为稳定的,应该怎样改?
- 8.7 图 8.9 给出了一个用栈来代替递归实现快速排序的优化算法。那么在最差情况下栈有多深?怎样组织递归调用的顺序可以减小栈的深度?可以减为多少?在图 8.9 算法的基础上写出所作的更改。
- 8.8 设  $L$  为一数组,  $L.length()$  给出数组中元素(记录)的数目,  $qsort(L, 0, i)$  用快速排序算法将  $L$  排序(结果仍然保留在  $L$  中)。那么下面程序段的平均渐近运行时间各为多少?
- (a) `for (i = 0; i < L.length(); i++)`  
    `qsort(L, 0, i);`
  - (b) `for (i = 0; i < L.length(); i++)`;  
    `qsort(L, 0, L.length() - 1);`
- 8.9 某个待排序的序列是一个可变长度的字符串序列,这些字符串一个接着一个地存储于惟一的字符数组中,另一个数组存储指向这个大字符串数组的索引(存储指向特定字符串的指针)。请改写快速排序算法,对这个字符串序列进行排序。函数要修改索引数组,使得第一个指针指向最小字符串的起始位置。
- 8.10 在  $1 \leq n \leq 1000$  范围内,画出  $f_1(n) = n \log n$ ,  $f_2(n) = n^{1.5}$ ,  $f_3(n) = n^2$  的曲线图,比较一下它们的增长率。典型地,插入排序的常数因子比 Shell 排序及快速排序要少(随着  $n$  的变化而变化得快)。当  $n = 1000$  时,Shell 排序要比插入排序快,那么它的常数因子与插入排序的常数因子相比,应该快多少倍呢?当  $n = 1000$  时,快速排序要比插入排序快,那么它的常数因子与插入排序的常数因子相比,应该快多少倍呢?
- 8.11 假设有一个算法 `SPLITk` 可以将一个序列  $L$  (有  $n$  个元素)分成  $k$  个子序列,每一个包含一个或者更多的元素。对于  $i < j \leq k$ ,序列  $i$  中的所有元素都小于序列  $j$  中的元素。如果  $n < k$ ,那么  $k - n$  个子序列为空,其余的长度为 1。设 `SPLITk` 的运行时间为  $O(L \text{ 的长度})$ 。再设  $k$  个子序列可以在常数时间内连接起来。考虑一下下面的程序段:

```
List SORTk(List L) {  
    List sub[k];      // To hold the sublists  
    if (L.length() > 1) {  
        SPLITk(L, sub); // SPLITk places the sublists into sub  
    }  
}
```

```

    for (i = 0; i < k; i++)
        sub[i] = SORTk(sub[i]); // Sort each sublist
    L = concatenation of k sublists in sub;
    return L;
}

```

- (a) SORTk 的最差运行时间是多少? 为什么?
- (b) SORTk 的平均运行时间为多少? 为什么?
- 8.12 下面是一个略有变化的排序问题。问题是把  $n$  个坚果放到  $n$  个对应大小的筛子中。设每个筛子都有一个对应的坚果与其大小相同。但是我们并不知道哪个坚果应该放到哪个筛子中。两个坚果或者两个筛子之间的区别难以用肉眼辨认, 因此不能用直接比较大小的方式区别它们。但是你可以比较坚果与筛子之间的大小关系, 这只需要把坚果放入筛子中即可(假设比较只需要常数时间)。坚果要么大于筛子, 要么小于筛子, 要么等于筛子。那么在最差情况下, 把坚果都归入到各自的筛子中需要的最小比较次数是多少?
- 8.13 (a) 编写一个算法尽可能快地排序 3 个元素。那么在最佳、平均、最差情况下的比较和交换次数各为多少?
- (b) 编写一个算法尽可能快地排序 5 个数字。那么在最佳、平均、最差情况下的交换与比较次数各为多少?
- (c) 编写一个算法排序 8 个数字, 算法要尽量快。那么在平均、最佳、最差情况下的比较与交换次数各为多少?
- 8.14 编写一个算法, 对一个没有重复关键码的数值在 0 到 30 000 范围内的序列排序。保持空间开销较小。
- 8.15 下面的各个操作中, 哪一个最适合先进行排序处理? 对于这些操作, 简短地描述一个实现算法, 并给出算法的渐近复杂性。
- (a) 找最小值。
- (b) 找最大值。
- (c) 计算算术平均值。
- (d) 找中间值。
- (e) 找出出现次数最多的值。
- 8.16 考虑这样一个递归的归并排序, 当子数组小于某个阈值时开始采用插入排序。如果归并排序调用了  $n$  次, 那么插入排序将被调用几次? 为什么?
- 8.17 编写一个归并排序函数(输入是一个链接数组)。
- 8.18 使用与 8.9 节相似的方法来证明: 在一个有  $n$  个值的有序数组中, 寻找给定元素的最差情况时间下限为  $\log n$ 。
- 8.19 给出一个数值 0 到 7 之间的排列, 该排列将使快速排序(8.4 节实现的算法)具有最差时间代价。

## 8.12 项目设计

- 8.1 基于本章所给快速排序 Java 代码,测试当输入数据的长度变化很大时,下面几种优化的快速排序算法的性能。对于下面的几种优化方法,可以进行某些组合,尽可能找到最快的快速排序算法。
  - (a)在选择轴值时,多看一些值。
  - (b)当子序列长度小于某个阈值时,使用插入排序方法。请测试不同的阈值。
  - (c)使用栈来消除递归,并使用内嵌来取消不必要的函数调用。
- 8.2 对于本章所述算法,写出自己的程序,并比较这些程序的运行时间。尽量让每个程序越快越好。你得到了与图 8.14 及图 8.15 相同的结果吗?如果没有,找出原因。你的结果与其他同学的结果比较呢?这说明了采用实验的方法研究算法时间有的一些什么困难?
- 8.3 使用不同的增量来研究 Shell 排序。并与 8.3 节“增量除以 2”的排序函数比较。请特别试一下“增量除以 3”的方法,该方法对长度为  $n$  的序列以  $n/3$ 、 $n/9$ ……为增量。其他增量方案也能正常运行吗?

## 第9章 文件管理和外排序

算法和数据结构的实现可以基于主存储器,也可以基于辅助存储器(例如磁盘和磁带),这会影响算法和数据结构的设计。为此,本章描述了主存储器和辅助存储器的基本差别。这些差别主要与存储介质中数据的访问速度、数据的存储量和数据的永久性有关。大多数文件处理技术都基于这样一个基本事实:访问磁盘和磁带比访问主存储器要慢得多。这种速度上的巨大差异对基于磁盘的应用程序到底有什么影响呢?这个问题导致了对基于磁盘排序的研究。

我们从描述主存储器和辅助存储器之间的显著差别开始。9.2节讨论磁盘和磁带的物理特性。9.3节介绍管理缓冲池的基本方法。在以后的章节中将多次使用缓冲池。9.4节讨论访问存储在磁盘中数据的随机访问模型。9.5节到9.8节讨论一些基本原则,这些基本原则适用于对一组因数量过大而无法放到主存储器中的记录进行排序。

### 9.1 主存储器和辅助存储器

一般说来,计算机存储设备分为主存储器(primary memory 或者 main memory)和辅助存储器(secondary storage 或者 peripheral storage)。主存储器通常指随机访问存储器(Random Access Memory, 即 RAM),辅助存储器指硬盘、软盘和磁带这样的设备。主存储器还包括高速缓存(cache)和视频存储器(video memory)。但是,由于它们的存在不会影响主存储器和辅助存储器之间的根本差别,因此我们不加考虑。

由于CPU的速度变得越来越快,每一种新型计算机配置的主存储器也更多了。既然存储器容量不断增加,是不是就不需要较慢的磁盘存储设备了呢?这种情况可能不会出现,因为用户想要存储、处理比原来更大的文件,而这种需求的增长至少和配置的主存储器容量的增长一样快。主存储器和辅助存储设备的价格都在快速下降,然而从图9.1中的价格表就可以看出来,每兆磁盘存储设备的价格比每兆主存储器的价格低两个数量级。过去每兆软盘的价格一般比每兆硬盘的价格便宜得多,但是现在每兆软盘的价格实际上更贵了。介于软盘和硬盘之间的是可移动驱动器,它们每个存储单位的价格和硬盘相当。磁带的价格比磁盘的价格便宜一个数量级。光存储设备,例如CD-ROM,以更便宜的价格提供更大的存储空间。

介质	价格	每兆价格
32MB RAM	\$ 225	\$ 7.00/MB
1.4MB 软盘	\$ 0.50	\$ 0.36/MB
2.1GB 硬盘	\$ 210	\$ 0.10/MB
1GB JAZ 盘	\$ 100	\$ 0.10/MB
2GB 盒式磁带	\$ 20	\$ 0.01/MB

图 9.1 常用的可写电子数据存储介质价格比较表。表中价格是 1997 年中期的代表价格

与主存储器相比,辅助存储设备还有至少两个其他优点。采用辅助存储设备存储的文件是永久的(persistent),也就是当电源关闭后,文件不会在磁盘或磁带中消失,这可能是非常重要的一条了。然而,用作主存储器的RAM则是易失的(volatile)——所有信息会因电源关闭而丢失。第二个优点是可以方便地把软盘、可移动驱动器、CD-ROM和磁带从一台计算机上取下来,再放到另一台计算机上使用。这为两台计算机之间传递信息提供了方便途径。

尽管辅助存储器有很低的价格以及永久存储能力和便携性,但是访问时间却很长。尽管并不是所有的磁盘访问都花费同样的时间(后面会有更多的解释),在1997年,访问存储在磁盘上的1字节数据一般需要大约10毫秒(也就是千分之10秒)。感觉上,这可能不是很慢,但是同访问主存储器中的1字节数据需要的时间相比,这就太慢了。在1997年,标准个人计算机中RAM的访问时间大约是60或70纳秒(即十亿分之60或70秒)。这样,访问磁盘中1字节数据需要的时间大约是访问主存储器中1字节数据的时间的25万倍。然而,磁盘和RAM的访问时间都在缩短,而且以大致相同的速度缩短。它们之间10年前的相对速度和今天的相对速度基本上是一样的,RAM访问时间和磁盘访问时间的差距仍然在10万到100万倍之间。

为了从直觉上理解这两个速度之间的显著差距,假设你在本书的目录中查找磁盘条目,然后再翻到相应的页,考虑一下这个过程所花费的时间。把这个时间称为“主存储器”访问时间。如果你花20秒就能完成这个访问,那么25万倍长的访问时间就是两个月。

由于访问磁盘中的数据太慢,因而需要创建有效的应用程序处理存储在磁盘中的信息。由于磁盘访问时间和主存储器访问时间之间的比率是100万比1,这使得设计基于磁盘的应用程序时遵循下面这条规则极其重要。

#### 使磁盘访问次数最少!

一般说来,有两种方法能够使磁盘访问次数最少。第一种方法是适当安排信息位置。这样,如果需要访问辅助存储器中的数据,就能以尽可能少的访问次数得到需要的数据,而且最好第一次访问就能得到。文件结构(file structure)就是用于组织存储在辅助存储器中数据的一种数据结构。文件结构的组织应当使磁盘访问次数最少。另一种减少磁盘访问次数的方法是合理组织信息,使每次磁盘访问都能得到更多的数据,从而减少将来的访问需要。也就是说,如果不难做到,准确地猜出以后需要什么信息,现在就取出这些以后需要的信息。你将会看到,从磁盘或磁带中读取几百个连续字节与读取一个字节需要的时间没有太大差别。因此,这一技术确实是可行的。

减少磁盘访问次数的另一种办法是压缩存储在磁盘中的信息。3.8节讨论了空间/时间权衡问题,即如果愿意牺牲时间,就会减少对空间的需求。然而,对于处理存储在磁盘上的数据的应用程序,空间/时间权衡原则表明,需要磁盘存储的数据越少,程序就运行得越快。这是因为同计算时间相比,从磁盘读取信息花费的时间非常多。因此,如果把解压缩数据额外增加的时间和通过减少存储需求而省下来的磁盘访问时间相比,几乎在任何情况下,前者都会比后者少。这正是文件被压缩时发生的情况,解压缩信息需要占用CPU时间,但是这段时间与减少读取字节数而省下来的时间相比要少得多。现在的文件压缩程序并没有设计成允许随机访问被压缩文件的一部分,所以在使用商用磁盘压缩工具的一般性处理中,基于磁盘的应用程序的空间/时间权衡原则并不能被方便地采用。然而,在将来,磁盘控制器可能会自动压缩和解压缩存储在磁盘中的文件。这样,利用基于磁盘的应用程序的空间/时间权衡原则既可以节省空间,也可以节省时间。今天的许多盒式磁带驱动器I/O期间都会自动压缩、解压缩信息。

## 9.2 磁盘和磁带驱动器

Java 语言程序员把存储在磁盘中可以随机访问的文件看成是一段连续的字节,而且可以把这些字节结合起来构成记录,称此为逻辑文件(logical file)。实际存储在磁盘中的物理文件(physical file)通常不是一段连续的字节,而是成块地分布在整个磁盘中。文件管理器(file manager)是操作系统的一部分,当应用程序请求从逻辑文件读数据时,它把这些逻辑位置映射为磁盘中具体的物理位置。同样,当根据文件的逻辑相对位置写入数据时,文件管理器必须把这个逻辑相对位置转换成磁盘中相应的物理位置。要想对这些操作的大致时间代价有一些了解,需要知道磁盘的物理结构和基本工作方式。

磁盘通常称为直接访问(direct access)存储设备。这表示访问文件中的任何一条记录都花费几乎相同的时间,这与磁带之类的顺序访问(sequential access)存储设备不同。顺序存储设备需要磁带阅读器从磁带的开始处处理数据,直到到达需要的位置。你将会看到,磁盘访问只是近似于直接访问。在任何一个给定时刻,总有一些记录可以比其他记录被更快地访问到。

一块硬盘由一个或多个圆形盘片(platter)组成,这些盘片从上到下排列,中间有一定的间隙,所有盘片都围着一个主轴(spindle)。盘片像电唱机中的唱片一样,以恒定的速率连续转动。盘片的每个可用表面都有一个读/写磁头(read/write head),或者称为 I/O 磁头(I/O head)。数据就是通过这些磁头读出或写入的,这有些像电唱机的活动臂从唱片中得到声音一样。与电唱机的针头不同,磁盘的读/写磁头实际上不接触盘片的表面。它与盘片表面稍微有一些距离。常规操作时,磁头与盘片的任何接触都会损伤磁盘。这个距离非常小,比一粒灰尘的高度还要小。一粒灰尘的高度和这个距离的比例就相当于跨越美国的 5000 千米航空飞行与飞机进行 1 米高的飞行的比例。然而,软盘的读/写磁头却与软盘表面接触,这也是造成它访问时间较慢的部分原因。

如图 9.2(a)所示,硬盘一般有多个盘片和多个读/写磁头。每个磁头都固定到一个回转臂的一端,回转臂的另一端与支杆相连。支杆可以把所有磁头一起向内或向外移动。当磁头在盘片上方的某个位置时,磁头就可以直接访问相应盘片上的数据。磁头在盘片的某个位置上时可以访问的所有数据就构成了一个磁道(track),即一个盘片上与主轴具有相同距离的所有数据,如图 9.2(b)所示。与主轴具有相同距离的、分布在各个盘片上的所有磁道称为一个柱面(cylinder)。因此,一个柱面就是回转臂处在某一位置上可以读取的所有数据。

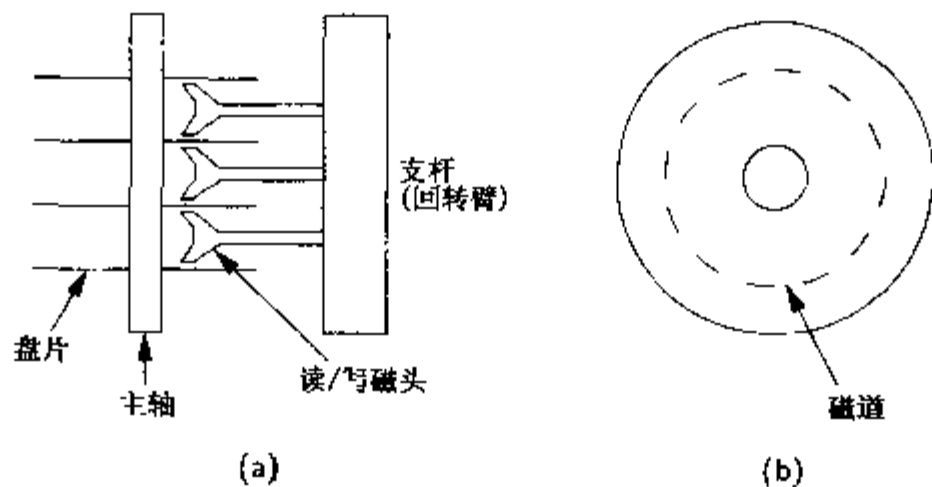


图 9.2 (a) 一个安装了一叠盘片的磁盘。(b) 一个磁盘盘片中的一个磁道



每个磁道还可以细分为多个扇区(sector)。两个相邻扇区之间有扇区间间隙(inter-sector gap),扇区间间隙内不存储数据。磁头可以通过这些间隙识别扇区结尾。每个扇区中都包含同样的数据量。由于外层磁道更长一些,同样一英寸的长度,它们比内层磁道包含的位数更少。这样,大约有一半可以使用的存储空间被浪费了,因为只有最内层的磁道以最高的数据密度存储数据。图 9.3 说明了这种安排。

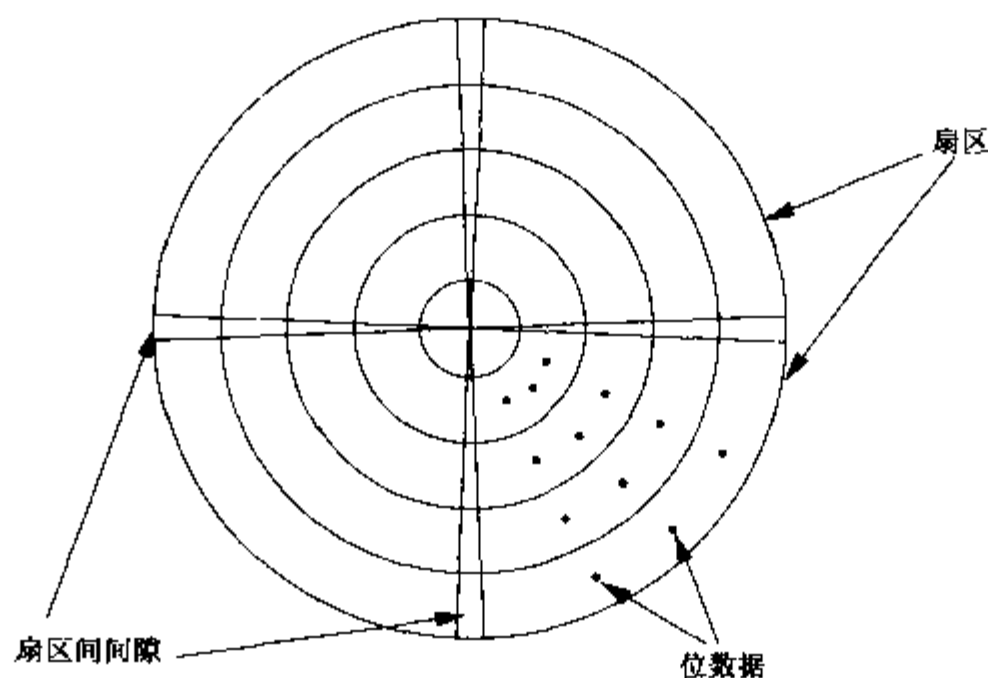


图 9.3 磁盘盘片的组织。点表示信息密度

CD-ROM(和一些软盘)与硬盘的这种物理布局不同,它们由一个单一的螺旋磁道组成。磁道内的位数据是均匀分布的,因此磁道里面的部分和外面的部分数据密度相同。为了沿着螺旋磁道以相同的速率读出数据,随着 I/O 磁头向磁盘中心移动,驱动器必须减慢磁盘的旋转速率。这种读取机制更复杂,而且更慢。

从硬盘读取一个字节或者多个字节数据可以分成三个独立的步骤。第一,移动 I/O 磁头,把它定位到包含数据的磁道上。这个移动称为寻道(seek)。第二,磁头等待包含数据的扇区旋转到磁头下面。磁盘在使用过程中总是在旋转,一般以每分钟 3600 或 7200 转旋转。等待目标扇区转到 I/O 磁头下面的时间称为旋转延迟(rotational delay 或者 rotational latency)。第三步是数据的实际传送(例如读取或者写入)。一旦数据旋转到了 I/O 磁头的下面,读取数据花费的时间相对来说比较少,仅仅相当于通过旋转盘片把数据移动到磁头下面需要的时间。实际上,磁盘的设计并不是考虑每次请求读取一个位的数据,而是读取一个扇区的数据。这样,一个扇区就是一次可以读出或者写入的最小数据量。

读取扇区的数据后,计算机需要花时间处理。在计算机进行处理时,磁盘仍然继续旋转。当一次读取多个相邻扇区的数据时,旋转就会产生问题。因为驱动器读取第一个扇区后,要处理数据,然后就会发现此时第二个扇区已经从 I/O 磁头下转过去了。必须等待盘片继续旋转,把第二个扇区送到磁头下面。

由于每个磁盘的旋转速率是固定的,计算机对每个扇区数据的处理时间也是固定的,系统设计者可以知道从第一个扇区已经读取到 I/O 磁头准备读取第二个扇区这段时间间隔,磁盘已经旋转了多远。与其让第二个逻辑扇区与第一个逻辑扇区物理上相邻,倒不如让第二个逻辑扇区与第一个逻辑扇区间隔一段距离,使得当 I/O 磁头准备读取第二个逻辑扇区的时候,它正好处在 I/O 磁头的下面。这种安排数据的方法称为交错法(interleaving),逻辑上相邻的

扇区之间的物理距离称为交错因子(interleaving factor)。图 9.4(b)是一个磁道中的扇区,扇区的交错因子是 3。扇区 2 和扇区 1 之间有扇区 4 和扇区 7 分隔。如果从扇区 1 第 1 次移动到 I/O 磁头下面开始读取数据,读取整个磁道需要 3 次磁盘旋转。第 1 次旋转读取扇区 1、扇区 2 和扇区 3。第 2 次旋转读取扇区 4、扇区 5 和扇区 6。第 3 次旋转读取扇区 7、扇区 8。

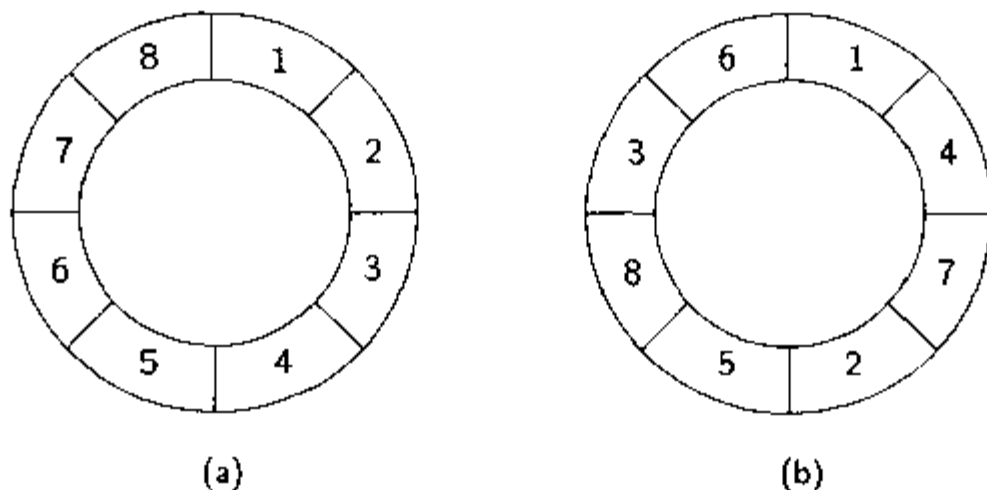


图 9.4 磁盘磁道的组织

(a)没有交错的扇区。(b)以 3 为交错因子组织的扇区

对比一下读取图 9.4(b)所示的交错扇区与图 9.4(a)所示的简单连续扇区。对于简单连续扇区的情况,首先读取第 1 个扇区,接下来是处理已经读取的数据,从而产生一些延迟。然后等待磁盘旋转,使得第 2 个扇区再次处于 I/O 磁头的下面,这需要旋转一整圈。如果没有交错,读取如图 9.4(a)所示的磁道需要磁盘旋转 8 圈。

一般说来,最好把一个文件的所有扇区都放在一起。这是由于下面两点假设:

1. 寻道时间慢(一般是 I/O 操作花费最大的部分)。
2. 如果读出了文件的一个扇区,很可能就要读出文件的下一个扇区。

假设 2 称为引用的局部性(locality of reference),这是一个在计算机科学中一再出现的概念。

多个扇区通常集结成组,称为一个簇(cluster)。组成一个簇的扇区可能是交错放置的。簇是文件分配的最小单位,因此所有文件都是一个或者几个簇的大小。簇的大小由操作系统决定。文件管理器记录每个文件是由哪些簇组成的。在 MS-DOS 系统中,磁盘中有指定的部分,称为文件分配表(File Allocation Table)。文件分配表中记录哪些扇区属于哪个文件。而 UNIX 系统则不使用簇。在 UNIX 系统中,文件分配的最小单位和读出/写入的最小单位是一个扇区,在 UNIX 术语中称为一个块(block)。UNIX 系统维护相关信息,这些信息记录文件由哪些块组成,称为索引结点(i-node)。

属于同一个文件的一组物理上相连的簇称为一个范围(extent)。在理想情况下,组成一个文件的所有簇在磁盘中是连续的(即,文件由一个范围组成)。这样,访问文件的不同部分需要的寻道时间最少。如果当文件创建时磁盘快要满了,可能没有一个足够大的范围可以容纳这个新文件。而且,如果一个文件变大了,可能没有物理上与之相连的空闲空间。因此,一个文件可能由广泛分布在磁盘中的多个范围组成。磁盘越满,磁盘中文件的变化越多,文件就越零散(结果寻道时间也就越长)。文件零散会导致性能显著下降,因为访问数据需要额外的寻道。

当文件的逻辑记录长度和扇区长度不匹配时,就会出现另一类问题。如果扇区长度不等

于多条记录的长度(或者一条记录的长度不等于多个扇区的长度),记录就不能正好放入一个(或者多个)扇区中。例如,一个扇区的长度是 2048 个字节,而一条逻辑记录的长度是 100 个字节。这样,一个扇区中只能存放 20 条记录,而剩下 48 个字节。其结果是,要么浪费多余的空间,要么允许记录跨扇区边界。如果允许一条记录跨扇区边界,读出这条记录就需要 2 次磁盘访问。如果使剩余的空间为空,这些浪费的空间就称为内部碎片(internal fragmentation)。

内部碎片也可能出现在簇边界处。当文件长度不是正好等于多个簇的长度时,在最后一个簇的结尾一定会浪费一些空间。当用文件长度对簇长度进行取模计算的结果为 1 时,就会出现最糟糕的情况(例如,一个文件的长度是 2049 个字节,而一个簇的长度是 2048 个字节)。因此,簇长度的选择就要在减少顺序处理的大文件的寻道时间(为了减少寻道时间,簇的长度大一些更好)和减少小文件的存储空间浪费(为了减少浪费的存储空间,簇的长度小一点更好)两个方面之间权衡。

每一种磁盘组织方式都需要使用一些磁盘空间管理扇区、簇等等。磁道内扇区布局如图 9.5 所示。必须存储在磁盘中的信息包括文件分配表、包含地址标识和每个扇区状态信息(是否可用)的扇区头(sector headers)和扇区之间的间隙。扇区头中还包括错误校验码,用于验证数据有没有被破坏。这就是大多数磁盘有一个“名义上的”大小,它比存储在驱动器中实际使用的数据量大的原因。这里有一个典型的例子,一个磁盘驱动器名义上的大小是 1044 MB,而实际经过格式化以后只能提供 1000 MB 的磁盘空间。差距的产生就在于组织磁盘中的信息也需要占用空间。由于碎片的原因,还会失去更多的空间。

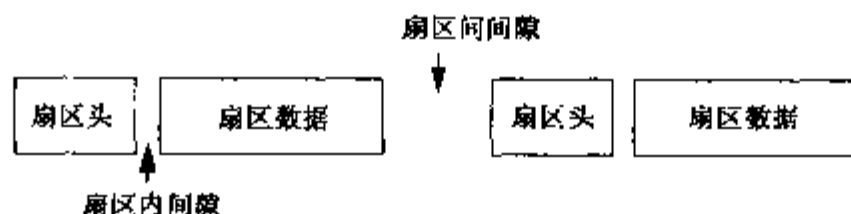


图 9.5 磁道内部扇区间隙说明。每个扇区从一个扇区头开始,扇区头中包括扇区地址和这个扇区内容的错误校验码。紧接着扇区头是一个小的扇区内间隙,其后就是扇区数据。每个扇区和下一个扇区之间通过一个较大的扇区间间隙分开

### 9.2.1 磁盘访问的代价

访问磁盘中扇区的主要代价一般是寻道时间。这里假定必须进行寻道。当顺序读取一个文件时(如果存放文件的扇区物理上是连续的),需要的寻道时间很少。然而,当随机访问一个磁盘的扇区时,当前磁道和目标磁道之间的平均距离是磁盘中磁道总数的三分之一。这个预期代价是通过计算在任意两个磁道之间寻道的所有可能情况的平均距离得到的。寻道的代价可以再细分为让回转臂移动的启动时间和跨越一个磁道需要的时间。这样,跨越  $n$  个磁道寻道的代价可以用方程  $f(n) = t * n + s$  表示,其中  $t$  是跨越 1 个磁道的时间,而  $s$  是启动时间。拥有 675 MB 存储空间的 Maxtor 磁盘就是一个典型的例子。生产厂家的规格说明标识,在一个有 612 个磁道的盘片上,  $t = 0.08 \text{ ms}$ ,  $s = 3 \text{ ms}$ 。因此,这个磁盘的平均寻道时间可以这样计算:  $0.08 * 612/3 + 3 \approx 19.3 \text{ ms}$ 。

多年来磁盘的一般旋转速度是 3600 r/min,即每 16.7 ms 转一圈。近一时期磁盘的旋转速度是 7200 r/min。当随机读取一个扇区时,可以预计磁盘需要转半圈,使得目标扇区到达

I/O磁头之下,或者对一个 3600 r/min 的磁盘来说,需要 8.3 ms。

一旦扇区到达 I/O 磁头下面,扇区中的数据就可以以扇区旋转的速度在磁头下面传送。如果要读取整个磁道,那么就需要旋转一圈(16.7ms),使得整个磁道经过磁头下面。如果只需要读取磁道的一部分,那么相应地,需要的时间也就更少。例如,如果磁道中有 32 个扇区,而只需要读出 1 个扇区,这就需要大约 0.5 ms。

**例 9.1** 假定一个磁盘总容量(名义上)有 675MB,分布在 15 个盘片上,每个盘片上有 45MB。每个盘片上有 612 个磁道,每个磁道中包含 150 个扇区,每个扇区内有 512 个字节。I/O 磁头的启动时间是 3 ms,在一次寻道过程中磁头跨越 1 个磁道的时间是 0.08 ms,假定操作系统维护的簇大小是 8 个扇区(4KB),因而每个磁道有 18 个簇(其余的空间被内部管理信息占用了)。根据这一信息可以估计出磁盘访问的代价。

假定磁道中扇区的交错因子是 3。从磁道的第一个扇区处于 I/O 磁头之下开始算起,磁盘需要转多少圈才能读取一个磁道? 答案是 3 圈,因为每转 1 圈可以读取三分之一的扇区。那么读出这个磁道需要多少时间呢? 由于需要转 3 圈,每转 1 圈需要 16.7 ms,读出一个磁道的全部时间就是 50.1 ms。

如果一个文件的长度是 128KB,分成 256 条记录,每条记录长度等于一个扇区的长度(每个扇区的长度是 512 个字节),读出这个文件要花多少时间呢? 这个文件存储在 32 个簇中,因为每个簇中正好 8 个扇区。问题的答案在很大程度上依赖于文件如何存储在磁盘中,即它们是都放在一起还是分别放在多个范围内。我们计算一下这两种情况会产生多大差异。如果文件的存储方式是填充一个磁道的所有 144 个可用扇区和相邻磁道的 112 个扇区,那么总代价就是初始第 1 个磁道的寻道时间(假设这需要三分之一磁盘宽度的随机寻道)、等待第一个扇区转到磁头下面的旋转延迟、读取时间(由于交错需要 3 圈)、相邻磁道的寻道时间、第二次等待磁头的旋转延迟和最后读取其余扇区的时间(可以通过假定旋转完整的 3 圈加以简化,这样做基本上是正确的)。这样,总共预计时间就是:

$$\begin{aligned} \text{总时间} &= (\text{初始寻道时间}) + (\text{旋转延迟} + \text{读取时间}) + (\text{第二次寻道时间}) + \\ &\quad (\text{旋转延迟} + \text{读取时间}) \\ &= (612/3 * 0.08 + 3) + ((0.5 + 3) \times 16.7) + (0.08 + 3) \\ &\quad + ((0.5 + 3) \times 16.7) = 139.3 \text{ ms} \end{aligned}$$

如果文件的簇随机分布在整个磁盘上,那么需要的时间就是随机读取 32 个簇的时间,这意味着每一次随机寻道之后都接着一次旋转延迟,接着是磁盘旋转 24/150 圈的时间(由于我们想要读取 8 个扇区,其交错因子是 3)。这就需要:

$$\begin{aligned} &32 * (612/3 * 0.08 + 3 + 16.7/2 + 24/150 * 16.7) \\ &= 32 * 30.3 = 969.6 \text{ ms} \end{aligned}$$

或者将近 1 秒。这个例子说明为什么防止磁盘文件零散很重要,以及为什么“磁盘碎片收集器”可以加速文件处理时间。当磁盘将要被填满的时候,此时每当创建或者修改一个文件,文件管理器必须搜索空闲空间,这时经常出现文件碎片。

**例 9.2** 读取一个扇区内的信息需要一次寻道,接着是旋转延迟,等待目标扇区到达 I/O 磁头下面,最后是目标扇区经过磁头,读出目标扇区内的数据。按照例 9.1 中的规格说明,就是:

$$612/3 * 0.08 + 3 + 16.7/2 + 1/150 * 16.7 = 27.8 \text{ ms}$$

有时把这个值称为磁盘的平均访问时间(average access time)。实际上,它已经过于简化了。当现在的大多数磁盘厂商计算平均访问时间的时候,他们还要考虑目标信息已经驻留在磁盘缓存(disk cache)中的可能性(磁盘缓存存储最近读取的扇区)。例如,磁盘厂商宣称的平均访问时间大约是我们计算值的一半。9.3 节讨论缓存的概念。

### 9.2.2 磁带

一般来说,磁带比硬盘更便宜。一般的价值 20 美元的 9 磁道磁带每英寸存储 6250 个位(6250 bpi)的数据,磁带总长是 2400 英尺。这样,从理论上说,一个 9 磁道磁带能够容纳大约 170 MB,平均每兆位不到 12 美分。但是现在,一个好的 9 磁道磁带驱动器的价格超过一台工作站,大概是 10 000 美元到 20 000 美元。这样,9 磁道磁带并没有被普遍用到工作站上,但是仍然广泛地应用于更大的计算设备上。盒式磁带类似于 9 磁道磁带,普遍用于工作站上和个人计算机上。一个容量为 1GB 的盒式磁带价值 20 美元,或者每兆位 1 美分。

磁带与磁盘的不同之处是磁带只能顺序访问。也就是说,如果不正转或者反转磁带,就不能从磁带的当前位置到达目标位置。这使得对于随机访问情况,磁带慢得无法接受。影响磁带性能的第二个物理特性是磁带驱动器要花费一些时间使磁带停下来。这样,就必须使用一个很大的块间间隔(interblock gap)分开数据,以便 I/O 磁头能够识别出一个间隔,从而使磁带停下来。块间间隔一般等于多条记录的长度。这样,在每两条记录之间放置一个间隔就会浪费大量空间。为了避免这种浪费,需要把多条记录组织到一个块(block)中;一个块中的记录数称为块因子(blocking factor)。磁带只能在块间隔处停止高速旋转或者反转。要找到某一条记录或者文件,需要磁带(相对地)快速到达目标块。然后 I/O 磁头必须缓慢地读取这个块,使得 CPU 接收到目标记录。这样,块的长度大一些就会节省空间,但是这是以增加处理块中单条记录需要的时间为代价的。如果块总是作为一个单位顺序处理,那么这并不是问题。

磁带的第二个性能度量是它们能够以多快的速度传送数据。一个一般的 9 磁道磁带驱动器能以每秒 200 英寸(200 ips)的速度处理数据。一个 6250 bpi 的磁带在一个 200 ips 磁带驱动器中名义上的(或者高峰)传输率是:

$$6250 \text{ bpi} \times 200 \text{ ips} = 1.25 \text{ MB/s}$$

然而,块间间隔减少了磁带中的实际数据密度,以及可能达到的最快传输率。找到目标记录的时间和处理时间进一步减小了高峰传输率。

由于磁带很便宜但又很慢,而且只适合于顺序访问,通常不用它来存储需要快速访问的数据。然而,磁带通常用于备份和归档。由于磁带不再普遍用于联机处理,本书不再对它作进一步讨论。

## 9.3 缓冲区和缓冲池

让我们再考虑一下从磁盘中读取数据的时间。如果按照例 9.1 中磁盘的规格说明,读取一个磁道数据的平均代价为  $612/3 \times 0.08 + 3 + 3.5 \times 16.7 = 77.8 \text{ ms}$ 。读取一个扇区数据要花费  $612/3 \times 0.08 + 3 + 16.7/2 + 16.7/150 = 27.8 \text{ ms}$ 。这是一个相当大的节省(稍微超过三分之一的时间),但是读取了磁道中不到 1% 的数据。如果要读取一个字节的数,大约要花费  $612/3 \times 0.08 + 3 + 16.7/2 = 27.7 \text{ ms}$ 。同读取一个扇区的信息需要的时间相

比,这在时间上的减少并不显著。由于这个原因,每当访问磁盘时,甚至当只请求一个字节的的数据时,几乎所有的磁盘驱动器都会自动读取或者写入整个扇区的数据。

一旦读取了一个扇区,它的信息就存储在主存储器中了,这称为缓冲(buffering)或缓存(caching)信息。如果下一个磁盘请求要访问同一个扇区,就不再从磁盘中读取,因为信息已经存储在主存储器中了。本章的开始说过,应该使磁盘访问次数最少。缓冲方法就是这样的一个例子:从磁盘中取出更多的信息,以满足将来的请求。如果要随机访问一个文件中的信息,那么两个连续的磁盘请求需要同一个扇区内的数据的可能性非常小。然而,实际上大多数磁盘请求的位置都接近于前一次请求的位置(至少在逻辑文件中是这样)。这说明下一次请求“命中缓存”的可能性比随机情况的机会更高。

这个原理说明了新的磁盘为什么比过去的磁盘更快的一个原因。不仅硬件比原来更快,而且现在的信息存储使用更好的算法和更大的缓存区,使得从磁盘中取信息的次数尽量少。同样的概念也用于在CPU内更快的存储器中存储部分程序,即在现代微处理器中广为使用的CPU缓存。

现在,几乎所有的操作系统都自动进行扇区级缓冲,而且磁盘驱动器的控制器硬件中通常也直接建立扇区级缓冲。大多数操作系统至少维护两个缓冲区,一个缓冲区用于输入,另一个缓冲区用于输出。考虑一下在按字节复制操作时,如果只使用一个缓冲区会怎样。包含第一个字节的扇区将被读入I/O缓冲区中。输出操作会破坏这个惟一的I/O缓冲区中的内容,以写入这个字节。然后,缓冲区需要再次从磁盘中填充以得到第二个字节,在输出时又将被破坏。对这个问题的简单解决方法就是一个缓冲区用于输入,另一个缓冲区用于输出。

一旦收到I/O请求,大多数磁盘控制器就能够独立于CPU进行操作。由于在一个I/O操作时间内CPU一般可以执行几百万条指令,这样做非常有用。最大限度地利用这种微并行机制的技术称为双缓冲(double buffering)。想像一下正在顺序处理一个文件。当正在读取第一个块时,CPU不能处理那些信息,因此这时必须等待或者找其他一些事情去做。一旦读取了第一个块,CPU就开始进行处理。与此同时,磁盘驱动器立即开始读取第二个块。如果CPU处理一个扇区的时间基本上等于磁盘控制器读取一个扇区的时间,那么就有可能保持CPU持续得到文件中的数据。同样的概念也可以用于输出,在CPU写入存储器中的一个输出缓冲区的同时,把另一个缓冲区写入磁盘中。这样,在支持双缓冲的计算机中,它至少需要使用两个输入缓冲区和两个输出缓冲区。

在存储器中缓存信息是一个非常好的想法,这个想法甚至经常推广为多缓冲区。操作系统或者应用程序可以在多个缓冲区中存储信息。存储在一个缓冲区中的信息经常称为一页(page),这些缓冲区合起来称为缓冲池(buffer pool)。缓冲池的目标是增加存储器中存储的信息量,希望对于新的信息请求,从缓冲池中得到请求信息的可能性更大,从而不必再从磁盘中读出。

只要缓冲池中存在没有使用的缓冲区,就可以根据需要从磁盘中读出新的信息,放到这个缓冲区中。当应用程序不断从磁盘中读入新的信息时,最终缓冲池中的所有缓冲区都会被填满。一旦出现这种情况,就需要作出选择,牺牲哪些缓冲区中的信息,从而为新请求的信息提供空间。

当替换缓冲池中的信息时,目标是选择一个包含“不需要的”信息的缓冲区。也就是说,这个缓冲区包含的信息被再次请求的可能性最小。有多种方法可以作出决策。一种方法是“先



进先出”(FIFO)。这种方法简单地把缓冲区排成一个队列。使用队列最前面的缓冲区存储新信息,然后把它放到队列的最后。在这种方式中,被替换的缓冲区是保存信息时间最长的缓冲区,但愿这些信息不再需要。当处理过程是基本上顺序地以稳定的速率沿着文件从前到后移动时,这是一个合理的假设。然而,许多程序反复使用某个特定的关键信息段,而信息的重要性与信息的第一次访问时间到现在有多久没什么关系,但是却与信息的最后一次访问时间到现在多近有极大的关系。

另一种方法称为“最不频繁使用”(LFU)方法。LFU 记录缓冲池中每个缓冲区的访问次数。当需要重新使用一个缓冲区存储新信息时,被访问次数最少的缓冲区被认为包含“最不重要的”信息,接下来就使用这个缓冲区。尽管直觉上看起来 LFU 很合理,它仍然有很多缺陷。首先,需要为每一个缓冲区存储和更新访问计数。其次,过去被引用多次可能与现在是否仍然会被引用无关。这样,就经常需要一些记录“到期”的时间机制。有一些缓冲区因为总是没有被替换(除非对已经读过的所有扇区维护计数,而不只是对当前在缓冲池中的扇区维护计数),会缓慢地建立很大的引用计数。引入记录“到期”的时间机制也可以避免这个问题。

第三种方法称为“最近最少使用”(LRU)方法。LRU 简单地在—个链表中保存缓冲区。每当访问了一个缓冲区中的信息,就把这个缓冲区放到链表的最前边。当需要读取新的信息时,使用链表最后面的缓冲区(最近最少使用的缓冲区),丢弃其中的“老”信息。除非通过一个应用程序信息访问模式的特别知识能够得到一种特殊目的的缓冲区管理方法,否则,这是一种很容易实现的近似于 LFU 的方法,也是一种可供选择的管理缓冲池的方法。

## 9.4 程序员的文件视图

前面已经提到,Java 程序员对随机访问文件的逻辑视图是一个单一的字节流。与文件的交互可以看作是一种通信通道,可以对这个通信通道发出下面三种指令之一:从文件的当前位置读取字节、向文件的当前位置写入字节和在文件中移动当前位置。正常情况下不需要知道字节在扇区、簇等内部的存储,操作系统会自动进行扇区级缓冲。

当处理磁盘文件中的记录时,访问顺序对 I/O 时间有很大的影响。随机访问(random access)过程对记录的处理独立于文件中记录的逻辑顺序。顺序访问(sequential access)按照文件中记录的逻辑顺序处理记录。如果磁盘文件的物理布局 and 它的逻辑顺序一致,顺序访问需要的寻道时间会更少一些。如果在磁盘中创建文件时空闲空间所占的比例很高,有可能使文件的物理布局 and 它的逻辑布局一致。

许多操作系统都支持虚拟存储(virtual memory)。利用虚拟存储技术,程序员可以假想认为主存储器比实际存在的更多。这是通过缓冲池从磁盘中读入块来实现的。磁盘中存储虚拟存储器的全部内容;根据存储器的访问需要把块读入主存储器。当然,使用虚拟存储技术的程序比不使用虚拟存储技术,而把数据存储在物理存储器中的程序慢。其好处是减少了程序员的工作,这是因为好的虚拟存储系统不需要修改程序就使人感觉到提供了更大的主存储器。图 9.6 说明了虚拟存储的概念。

下面是使用类 RandomAccessFile 从随机访问磁盘文件中访问信息的基本 Java 函数。

- RandomAccessFile(String name, String mode): 类构造函数,打开文件进行处理。
- read(byte[] b): 从文件当前位置读入一些字节。随着字节的读入,文件当前位置向

前移动。

- `write(byte[ ] b)`: 向文件当前位置写入一些字节(覆盖已经在这个位置的字节)。随着字节的写入,文件当前位置向前移动。
- `seek(long pos)`: 在文件中移动当前位置,这样就可以在文件中的另一个位置读出或者写入字节。
- `close`: 处理结束后关闭文件。

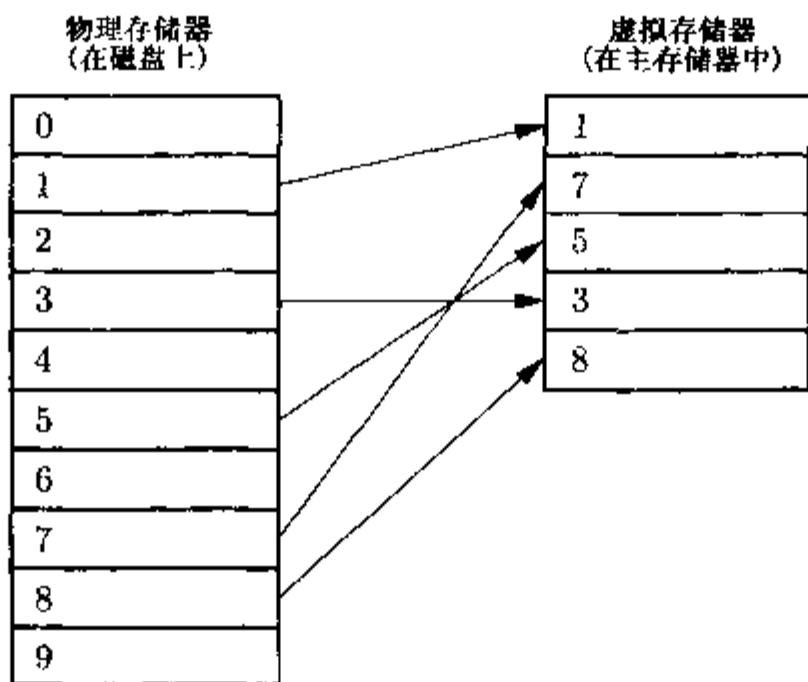


图 9.6 虚拟存储说明。全部信息都驻留在磁盘(物理存储器)上。最近访问过的那些扇区保存在主存储器(虚拟存储器)中。在这个例子中,物理存储器的扇区 1、7、5、3 和 8 当前存储在虚拟存储器中。如果接到对扇区 9 的存储器访问,就必须替换当前主存储器中的一个扇区。

## 9.5 外部排序

现在考虑一下把因数量太大而无法放到主存储器中的一组记录排序的问题。由于记录必须驻留在外部存储器中,因而相对于第 8 章中讨论的内部排序,这些排序方法称为外部排序(external sort)。

对大量记录排序是许多应用程序的核心,例如工资管理和其他大型商业数据库,从而设计出了许多外部排序算法。几年前,排序算法设计者寻求对多磁带、多磁盘等特定硬件配置进行优化使用。今天,大多数计算任务在具有强大功能的 CPU 的个人计算机和低端工作站上完成,但是这些计算机上一般只安装一块磁盘,最多不超过两个。这里提供的技术用于配备一块磁盘的优化处理。这样,我们就可以只关注外部排序中最重要的问题,而忽略不重要的、依赖于机器的细节。有些读者可能需要实现更有效的外部排序算法,而算法的实现需要利用复杂的硬件配置,那么请参考一下 9.9 节的深入学习导读。

如果需要排序的一组记录太大而不能放到主存储器中,那么只能把其中的一些记录先从磁盘中读出来,进行重排,然后再把这些记录写回磁盘。这个过程不断重复下去,直到对整个文件进行了排序,其中每条记录可能被读出多次。根据 9.2 节有关磁盘的基础知识,很显然,



外部排序算法的主要目标是尽量减少读、写磁盘的信息量。为了减少磁盘访问,甚至可以给 CPU 增加一些处理任务。

在讨论外部排序技术之前,再考虑一下从磁盘中访问信息的基本方式。从程序员的角度来看,要排序的文件是有一定顺序的、固定大小的块(block)。为了简单起见,假定每个块中都包含数目相同、大小固定的数据记录。对于有些应用程序,一条记录可能只有几个字节,其中只包含关键码和其他少量信息。而对于另一些应用程序,一条记录可能有几百个字节,而记录的关键码字段很小。假定记录不会跨越块边界,对于某些排序应用程序,这些假定可以放宽,但是忽略这些复杂性会使原理更加清楚。

9.2 节已经说明,一个扇区是 I/O 的基本单位。也就是说,所有磁盘读写都是对一个完整的扇区进行的。扇区的大小一般是 2 的幂。对于不同的计算系统和不同的磁盘大小与磁盘速度,扇区的大小可以在从 512 字节到 8K 字节的范围内选择。外部排序算法使用的块大小应当等于扇区大小或者是扇区大小的若干倍。

按照这种模型,排序算法把一块数据读入主存储器的缓冲区内,对其进行一些处理,在将来的某个时间再把它写回磁盘。从 9.1 节可以看到,从数量级上说,从磁盘中读写一个块所花费的时间是主存储器访问所花费时间的 100 万倍。根据这一事实,可以合理地认为,在主存储器中对于一个块内部的记录采用内部排序算法进行排序,例如采用快速排序算法,所花费的时间比读、写这个块所需要的时间更少。

在好的情况下,从文件中顺序读取块比随机读取块更有效。由于磁盘访问寻道时间的显著影响,顺序访问显然会更快。然而,准确地理解在什么情况下顺序文件访问实际上比随机文件访问更快,这一点很重要,因为它会影响设计外部排序算法的方法。

要使顺序访问有效,就要保持寻道时间最少。这首先要求组成一个文件的块按照一定的顺序存储在磁盘中,而且放得很近,最好填充到一些连续的磁道中。至少,组成文件的范围数目应当很少。在一般情况下,用户无法控制文件在磁盘中的布局。但是在空闲空间比例很高的情况下,按照一定顺序一次把一两个文件写入磁盘中很有可能会产生这种布局。

其次,要求在整个顺序访问过程中,磁盘的 I/O 磁头始终在这个文件上面。如果有对 I/O 磁头的争用,就不会出现这种情况。例如,在多用户的分时计算机系统中,排序进程可能会与另一个用户进程争用 I/O 磁头。甚至当排序进程完全控制 I/O 磁头时,顺序访问仍然有可能效率不高。想像一下这种情况,对于单个磁盘的所有处理都已经完成,而磁盘驱动器采用的是最通常的配置,即一组读写磁头同时在一叠盘片上移动。如果排序进程需要从一个输入文件中读出数据,或向另一个输出文件中写入数据,那么 I/O 磁头就会不停地在输入文件和输出文件之间寻道。同样,如果同时处理两个输入文件(例如在一个归并过程中),那么 I/O 磁头也会不停地在这两个输入文件之间寻道。

因此,在配备单个磁盘的系统中,对数据文件的处理通常无法达到顺序访问的最高效率。这样,如果排序算法完成较少的非顺序磁盘操作,而不是完成大量逻辑上的顺序磁盘操作,算法会更有效。实际上,这些逻辑上的顺序磁盘操作需要进行很多次寻道。

如前所述,记录的大小可能比关键码的大小大很多。例如,一个大型企业的工资管理条目可能存储几百个字节的信息,其中包括每个雇员的姓名、ID 标识号、地址和职务。排序关键码可能是 ID 标识号,只需要几个字节。最简单的排序算法可能是把这样一条记录作为一个整体进行处理,每当需要处理的时候就读取整条记录。然而,这会极大地增加需要进行的 I/O

操作数量,因为一个磁盘块中只有很少几条记录。另一种选择是进行一次关键码排序(key sort)。在这种方法中,把所有关键码存储在一个索引文件中,其中每个关键码与一个指针一起存储,这个指针标识相应记录在原始数据文件中的位置。关键码和指针的组合应当比原来记录的大小小很多;这样,索引文件就会比整个数据文件小很多。可以对索引文件排序,因为它需要的 I/O 操作更少,这是由于索引记录比完整的原始记录更小。

一旦完成了对索引文件的排序,就可以对原来文件中的记录重新排列。一般并不这样做,有两点原因。首先,按照排列的次序从记录文件中读取记录需要对每条记录随机访问,这要花费大量时间,而且只有在需要按照排列的次序查看或者处理全体记录时才有用(与之对应的是检索选定的记录)。第二,数据库系统一般允许对多个关键码进行检索。也就是说,今天可能要求按照 ID 标识号的顺序进行处理。明天,经理可能想要根据工资数排序信息。这样,对整条记录可能没有一个“单一的”排序顺序。但是,可以维护多个索引文件,每个文件使用一个排序关键码。第 11 章中会深入探讨这种方法。

## 9.6 外部排序的简单方法

如果你的操作系统支持虚拟存储,最简单的外部排序方法是把整个文件读入虚拟存储器中,然后运行一个内部排序方法,例如快速排序。如果使用这种方法,虚拟存储管理器就可以使用它的缓冲池机制控制磁盘访问。但是,这种方法并不总是可行的。一个潜在的问题是虚拟存储器的大小通常比可用磁盘空间小得多。这样,输入文件可能无法放到虚拟存储器中。如果调整内部排序方法,利用算法自己的缓冲池,并结合 9.3 节讨论过的缓冲池管理技术之一,就可以克服虚拟存储器大小的限制。

调整内部排序算法使之应用于外部排序,这种思路的更普遍的问题是这样做不可能比设计一个新的尽量减少磁盘存取的算法更有效。考虑一下简单地调整快速排序算法,使之用于外部排序的情况。快速排序从处理整个记录组开始,第一次划分把索引从两端移到内部。这可以通过有效利用缓冲池来实现。下一步就是处理每一个子记录组,接着处理子记录组的子记录组,依此类推。随着子记录组越来越小,处理很快变成对磁盘的随机访问。即使 I/O 操作可能很有效,平均情况下,快速排序处理每条记录仍然需要  $\log n$  次。很快就会看到,还有比这种方法更好的方法。

进行外部排序的一个更好的方法源于归并算法。归并算法最简单的形式是对记录顺序地完成一系列扫描。在每一趟扫描中,归并的子列越来越大。这样,第一趟扫描把长度为 1 的子列归并成长度为 2 的子列;第二趟扫描把长度为 2 的子列归并成长度为 4 的子列;依此类推。这些被排序的子列称为顺串(run)。每一趟子列归并扫描都把一个文件的内容复制到另一个文件中。下面是算法的一个梗概,图 9.7 进行了说明。

1. 把原来的文件分成两个大小相等的顺串文件(run file)。
2. 从每个顺串文件中取出一个块,读入输入缓冲区中。
3. 从每个输入缓冲区中取出第一条记录,把它们按照排好的次序写入一个顺串输出缓冲区。
4. 从每个输入缓冲区中取出第二条记录,把它们按照排好的次序写入另一个顺串输出缓冲区。

5. 在两个顺串输出缓冲区之间交替输出,重复这些步骤直到结束。当一个输入块用完时,从相应的输入文件读出第二个块。当一个顺串输出缓冲区已满时,把它写回相应的输出文件。

6. 使用原来的输出文件作为输入文件,重复第2步到第5步。在第二趟扫描中,每个输入顺串文件中的前两条记录已经排好了次序。这样,就可以把这两个顺串归并成一个长度为4个元素的顺串输出。

7. 对顺串文件的每一趟扫描产生的顺串越来越大,直到最后只剩下一个顺串。

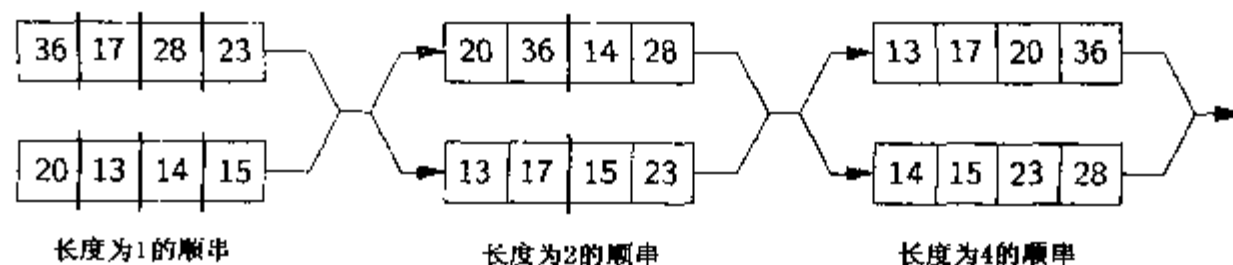


图 9.7 简单外部归并算法的说明。输入的记录平均来自两个输入文件。把来自每个输入文件的第一个顺串归并起来,放到第一个输出文件中。把来自每个输入文件的第二个顺串归并起来,放到第二个输出文件中。归并在这两个输出文件之间交替,直到输入文件为空。然后把输入文件和输出文件的位置颠倒过来,每一趟归并都使顺串长度加倍

这个算法可以方便地利用 9.3 节描述的双缓冲技术。每一趟归并都顺序地读出输入顺串文件,然后顺序地写入输出顺串文件。然而,要使顺序处理和双缓冲技术有效,需要每个文件单独使用一个 I/O 磁头。这就意味着每一个输入和输出文件都必须在一个单独的磁盘上,要使效率最高,就需要 4 个磁盘。

如果一个文件有  $n$  条记录,对这个文件进行简单的归并排序需要  $\log n$  趟扫描。这样,就需要对每条记录进行  $\log n$  次磁盘读写。仔细观察就会发现,对于小的顺串不需要使用归并排序,这样就可以显著减少归并的趟数。可以采取一种简单的改进方法,读入一块数据,在主存储器中进行排序,然后作为一个已排序的顺串输出。例如,假定块的大小是 4KB,块中每条记录占 8 个字节,其中 4 个字节是数据,4 个字节是关键码。这样,每个块包含 512 条记录。标准归并排序需要 9 趟归并才能产生 512 条记录的顺串。然而,如果把每个块作为一个处理单位,通过内部排序算法处理,一趟就能完成这个块的排序。然后可以使用标准归并算法归并这些顺串。标准归并算法处理 256K 记录需要 18 趟扫描。如果通过内部排序创建包含 512 条记录的初始顺串,就可以减少整个排序过程,这个过程包括一趟初始顺串创建和九趟扫描,这九趟扫描把所有初始顺串放到一起。归并趟数将近标准算法的一半。

可以进一步扩展这一概念。通常,可用的主存储器大小比一个块的大小更大。如果处理的初始顺串再大一些,归并排序需要的趟数就会更少一些。例如,大多数现代计算机都有从 0.5 兆到几十兆的 RAM 可供使用。如果所有这些存储器(除了一小部分用于缓冲区和局部变量)都尽可能多地用于建立初始顺串,那么只用很少的几趟扫描就可以处理非常大的文件。下一节将给出产生大顺串的技术,这些大顺串一般比主存储器大两倍左右。

另一种减少扫描趟数的方法是在每一趟扫描中多归并几个顺串。尽管标准归并算法一次归并两个顺串,但这个限制是不必要的。9.8 节讨论多路归并技术。

这些年提出了外部排序算法的各种变体。然而,大多数变体依据的都是同样的原理。一般说来,外部排序的所有好算法都基于下面两步:

1. 把文件分成大的初始顺串。
2. 把所有顺串归并到一起, 形成一个已排序的文件。

## 9.7 置换选择排序

这一节讨论的问题是怎样为一个磁盘文件创建尽可能大的初始顺串, 这里假定可供使用的 RAM 大小是固定的。如前所述, 一种简单的方法是把一个尽可能大的 RAM 分配给一个大数组, 从磁盘中读出数据并放到这个数组内, 然后使用快速排序算法为这个数组排序。这样, 如果分配给数组的可用存储器大小是  $M$  条记录, 那么就可以把输入文件分成长度为  $M$  的初始顺串。一种更好的方法是使用称为置换选择(replacement selection)的算法。在平均情况下, 这种算法可以创建长度为  $2M$  条记录的顺串。置换选择实际上是堆排序算法的一个微小变体。虽然堆排序算法比快速排序算法慢, 但是在此无关紧要, 这是因为 I/O 时间对任何外部排序算法的总运行时间都起着决定性作用。

置换选择算法把 RAM 看成一块大小为  $M$  的连续数组, 再加上输入缓冲区和输出缓冲区(如果操作系统支持双缓冲区, 可能还需要额外的 I/O 缓冲区。这是因为置换选择算法在输入、输出中都进行顺序处理)。把输入文件和输出文件都想像成记录流。置换选择算法在需要的时候从输入流中顺序地取出一条记录, 然后一次一条记录地向输出流输出顺串。通过使用缓冲区, 一次磁盘 I/O 就可以处理一个块。最初读入一块记录, 放到输入缓冲区中。置换选择算法一次从输入缓冲区中移去一条记录, 直到缓冲区为空。此时再读入下一块记录。向缓冲区输出也是类似的: 一旦填满了输出缓冲区, 就把它作为一个单位写回磁盘。这个过程如图 9.8 所示。

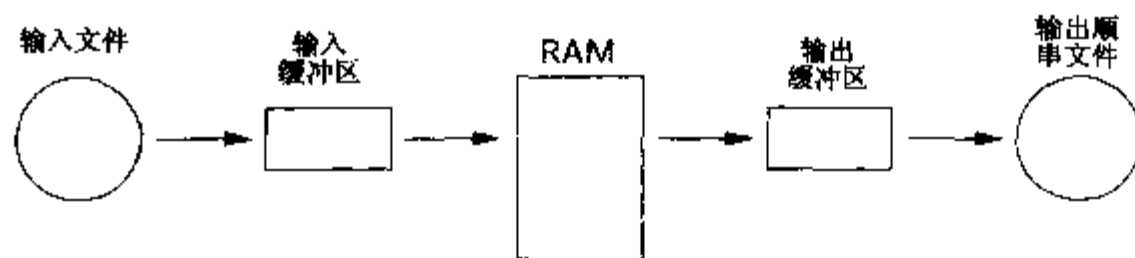


图 9.8 置换选择算法概述。按照顺序处理输入记录。开始时 RAM 中放入  $M$  条记录。随着记录的处理, 就把它写回输出缓冲区。当缓冲区填满时, 就把它写回磁盘。同时, 当置换选择需要记录时, 就从输入缓冲区中读取记录。每当这个缓冲区为空时, 就从磁盘中读取下一块记录

置换选择算法工作方式如下(假定主要处理在一个大小为  $M$  条记录的数组中完成)。

1. 从磁盘中读出数据放到数组中。设置  $LAST = M - 1$ 。
2. 建立一个最小值堆(回忆一下, 最小值堆定义为每个结点中记录的关键码值都小于其子孙结点中记录的关键码值)。
3. 重复以下步骤, 直到数组为空:
  - (a) 把具有最小关键码值的记录(根结点)送到输出缓冲区。
  - (b) 设  $R$  是输入缓冲区中的下一条记录。如果  $R$  的关键码值大于刚刚输出的关键码值:
    - i. 把  $R$  放到根结点。
    - ii. 否则使用数组中  $LAST$  位置的记录代替根结点, 然后把  $R$  放到  $LAST$  位置。设置

LAST = LAST - 1。

(c)筛出根结点,重新排列堆。

当在步骤 3(b)中的判断为真时,就把一条新记录添加到堆中,最终作为顺串的一部分输出。只要来自输入文件记录的关键码值大于输出到顺串中最后一条记录的关键码值,就可以把这些来自输入文件的记录安全地添加到堆中。关键码值较小的记录不能作为这个顺串的一部分输出,因为不能把它们排好次序。必须把这些值存放起来,将来作为另一个顺串的一部分进一步处理。然而,这里堆会缩小一个单位,堆输出的最后一条记录所占据的地方现在就成为空闲空间。这样,置换选择算法就会慢慢地缩小堆的大小,同时使用废弃的堆空间存储在下一个顺串中使用的记录。一旦完成了第一个顺串(堆为空),数组中就会填满在形成第二个顺串过程中准备处理的记录。图 9.9 说明了置换选择算法创建的顺串的一部分。

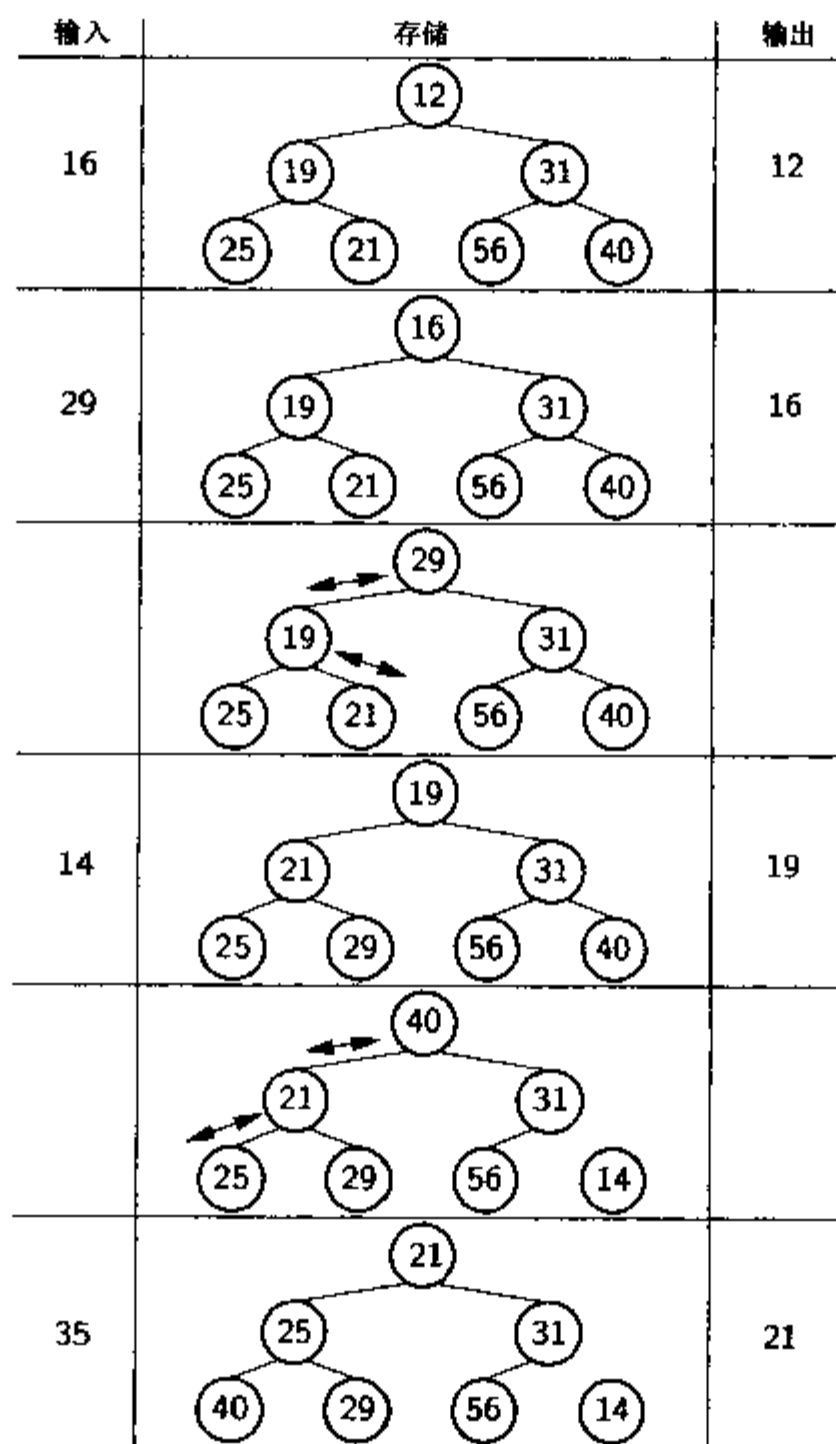


图 9.9 置换选择算法的例子。建立堆之后,输出根结点值 12,使用新来的值 16 代替。接下来输出值 16,用新来的值 29 代替。把堆重新排序,把值 19 升为根结点。接下来输出值 19。新来的值 14 在这个顺串中太小,被放到数组的最后,把值 40 移到根结点。重新排列堆的结果,把值 21 升为根结点,接下来就输出它

如果堆的大小是  $M$ ,一个顺串的最小长度就是  $M$  条记录,因为至少原来在堆中的那些

记录将成为顺串的一部分。如果碰到的情况还好(例如,输入已经被排序),那么任意长的顺串都是可能的。实际上,可以把整个文件作为一个顺串处理。如果碰到的情况不好(例如,输入是反向排序的),那么顺串的长度只能是  $M$ 。

置换选择算法产生的顺串的预期长度是多少呢?可以通过一个称为扫雪机问题(snow-plow argument)的类比进行推断。

设想在一次降雪量很大但降落很均匀的暴风雪中,扫雪机沿着一个环形道路前进。在扫雪机转了至少一圈以后,想像一下路面上雪的情况。靠近扫雪机后面的路面是空的,因为刚刚清扫过。雪覆盖得最厚的路面在扫雪机的最前面,因为这个地方被扫雪机清扫后降雪的时间最长。在任何时刻,整个路面上的雪量为  $S$ 。雪以稳定的速率在轨道上不停落下,有些雪落在扫雪机的“前面”,另一些则落在扫雪机的“后面”(在一个环形道路上,实际上所有的雪都落在扫雪机的前面,图 9.10 说明了这一思想)。在扫雪机的下一圈清扫中,道路上所有的雪  $S$  都被清除了,还加上又落下的一半。由于一切都处于稳定的状态,在一次清扫之后,在道路上仍然有雪量  $S$ ,所以在一次清扫中一定会落下  $2S$  的雪量,而且在一次清扫中清除了  $2S$  的雪量(后面留下雪量  $S$ )。

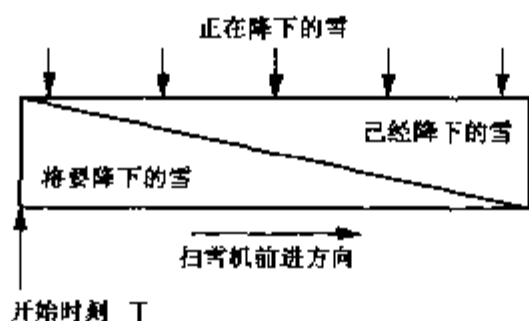


图 9.10 扫雪机类比图显示了扫雪机在环绕一圈过程中的行为。为了说明方便,使用了一个环形道路,以横截面显示。在任何时刻  $T$ ,大多数雪在扫雪机的前面。扫雪机沿着道路移动,它的前面总有同样多的雪量。随着扫雪机向前移动,在时刻  $T$  已经堆积在路面上的雪被清除得越来越少,而路面上正在降下的雪则越来越多

在置换选择算法的开始,几乎所有来自输入文件的值(即“在扫雪机前面”)都比这个顺串的最新输出的关键码值大,因为这个顺串中的初始关键码值都很小。随着对顺串的处理,最新输出的关键码值变得越来越大,从而来自输入文件的新关键码值很可能太小(即“在扫雪机的后面”);这些记录到了数组的底部。顺串的总长度预计是数组长度的两倍。当然,这要假定到来的关键码值在关键码范围内平均分布(在扫雪机问题中,假定雪在整个路面上均匀降下)。已排序的和反向排序的输入不符合这种预计,因此要改变顺串的长度。

## 9.8 多路归并

一般的外部排序算法在第二阶段归并第一阶段创建的顺串。如果使用简单的二路归并,那么  $R$  个顺串对整个文件需要  $\log R$  趟扫描。尽管  $R$  应当远远小于记录总数(由于每个初始顺串都应当包含许多记录),我们仍然希望进一步减少把顺串归并到一起需要的扫描趟数。二路归并不能充分利用可用主存储器。由于归并是作用在两个顺串上的顺序过程,因此每个顺串一次只需要有一个块的记录在主存储器中。这样,置换选择算法的堆使用的大多数空间(一般有多个块的长度)并没有在归并过程中使用。

如果一次归并多个顺串,就可以更好地利用这些空间,同时可以大大减少归并顺串需要的扫描趟数。多路归并与二路归并类似。如果有  $B$  个顺串需要归并,从每个顺串中取出一个块放在主存储器中使用,那么  $B$  路归并算法仅仅查看  $B$  个值(每个输入顺串最前面的值),并且选择最小的一个输出。把这个值从它的顺串中移出,然后重复这个过程。当任何顺串的当前块用完时,就从磁盘中读出这个顺串的下一块。图 9.11 说明了一个多路归并。

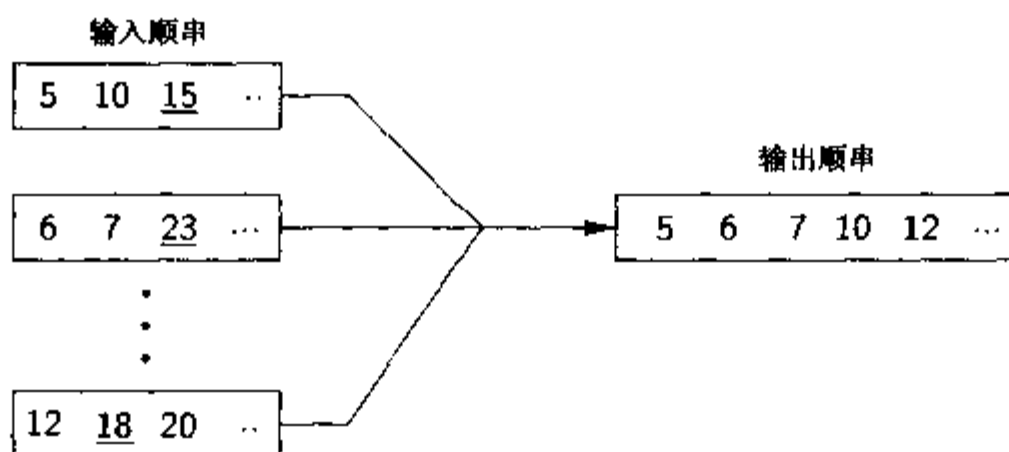


图 9.11 多路归并说明。检查每个输入顺串中的第一个值,把最小的值送到输出中。从输入中移去这个值,并且重复这个过程。在这个例子中,首先比较值 5、6 和 12。从第一个顺串中移去值 5,并且送到输出。接下来比较值 10、6 和 12。输出前 5 个值之后,每个块中的当前值就是带有下划线的那个值

从概念上讲,多路归并假定每个顺串存储在一个单独的文件中。然而,实际上这不是必需的。我们只需要知道每个顺串在单一一个文件中的位置,每当需要从某一个顺串中得到新数据时,就使用 `fseek` 把文件指针移到相应的块。自然,使用这种方法就不能对输入文件进行顺序处理了。然而,如果所有的顺串都存储在同一个磁盘中,那么处理也不可能是顺序的,因为 I/O 磁头要被两个顺串交替使用。这样,多路归并就用一趟随机访问扫描代替多趟(只是有可能)顺序扫描。如果处理不是顺序的(例如所有顺串都在同一个磁盘中的时候),这样做不会损失时间。

多路归并可以极大地减少需要的扫描趟数。如果存储器中有为每个顺串存储一个块的地方,那么一趟扫描就可以归并所有的顺串。这样,置换选择算法一趟扫描就可以建立初始顺串,多路归并一趟扫描就可以归并所有的顺串,总的代价是两趟扫描。然而,对于真正的大文件,可能由于顺串太多,从而需要放在主存储器中的块也太多。如果有地方能为一个  $B$  路归并分配  $B$  个块,而顺串的数目  $R$  比  $B$  大,那么就需要进行多趟扫描。也就是说,先归并前  $B$  个顺串,再归并  $B$  个顺串,依此类推。然后再通过下面的过程归并这些超级顺串,一次归并  $B$  个超级顺串。

一趟扫描可以归并一个多大的文件呢?假定可以为置换选择算法的堆分配  $B$  个块(导致顺串的平均长度是  $2B$  个块),接下来进行一个  $B$  路归并,在一次多路归并中,平均可以处理具有  $2B^2$  个块大小的文件;在  $k$  个  $B$  路归并中,平均可以处理具有  $2B^{k+1}$  个块大小的文件。为了理解它的增长有多快,假定有大小为 0.5MB 的工作主存可供使用,一个块的大小是 4KB,在工作主存中就会有 128 个块。平均顺串长度是 1MB(是工作主存大小的两倍)。一趟扫描可以归并 128 个顺串。这样,在只有 0.5MB 的工作主存中,平均说来,两趟扫描(一趟用于建立顺串,一趟用于进行归并)就可以处理大小为 128MB 的文件。如果工作主存的大小固定,块大一些则可以减少在一趟扫描中处理的文件的大小;块小一些或者工作主存大一些可以增加一趟扫描处理的文件大小。对于 0.5MB 的工作主存和 4KB 的块,两趟扫描就可以处理 16GB 大



小的文件,这对于大多数应用程序已经够大了。这样,对于单磁盘的外部排序来说,这是一个非常有效的算法。

图 9.12 显示了对于以下实现排序 1MB 和 4MB 的文件的运行时间比较:(1)具有两个输入文件和两个输出文件的标准归并排序;(2)具有 128KB 初始顺串的二路归并排序;(3)在 128KB 初始顺串上的 K 路归并排序。在每一种情况下,文件由 218 和 220 条记录组成,每条记录包括 4 个字节的关键码值。

算法	时间(微秒)	
	1MB	4MB
简单归并排序	123 310	543 870
128KB 的初始顺串	25 650	151 320
初始顺串为 128KB 的 K 路归并	14 340	67 670

图 9.12 三种外部排序算法的比较。这三种排序对 1MB 和 4MB 的文件中一组小记录进行,工作主存为 128KB,由大小为 4KB 的 32 个块组成。时间单位是微秒,比较在运行 Visual J++ 的 Microsoft Windows95 平台上进行

从这个试验可以看出,建立大的初始顺串可以把运行时间减少到标准归并排序的四分之一。使用多路归并可以进一步把时间减半。

总的说来,一种好的外部排序算法会尽量做好以下几个方面:

- 建立尽可能长的初始顺串。
- 在所有阶段尽可能把输入、处理和输出进行并行处理。
- 使用尽可能多的工作主存。应用更多的主存储器通常会加速处理。实际上,更多的主存储器比更快的磁盘效果更显著。对于外部排序来说,更快的 CPU 在运行时间方面不会有更大的改进。
- 如果有可能,使用多个磁盘,以便 I/O 处理有更大的并行性,并且允许顺序文件处理。

## 9.9 深入学习导读

有关文件处理的更通用的教材是 Folk 和 Zoellig 的《File Structures: A Conceptual Toolkit》[FZ92]。关于文件处理方面关键问题的高级一些的讨论是 Betty Salzberg 的《File Structures: An Analytical Approach》[Sal88]。有关外部排序方法的大量讨论可以在 Salzberg 的书中找到。本章提供的内容类似于 Salzberg 的思想。

有关磁盘模型化和度量的细节,参见 Ruemmler 和 Wilkes 的论文“An Introduction to Disk Drive Modeling”[RW94]。有关计算机硬件和组织结构的介绍,参见 Andrew S. Tanenbaum 的《Structured Computer Organization》[Tan90]。

任何一个关于 UNIX 操作系统的好的参考指南都描述了 UNIX 文件结构的内部组织。Graham Glass 的《UNIX for Programmers and Users》就是其中的一本[Gla93]。

扫雪机问题来源于 Donald E. Knuth 的《Sorting and Searching》[Knu81],其中还包括各种外部排序算法。



## 9.10 习题

- 9.1 计算机存储器的价格变化很快。对照图 9.1 中列出的各种介质,查找一下它们的当前价格。你得到的信息会改变有关磁盘处理的基本结论吗?
- 9.2 假定一个非常老的磁盘配置如下。全部存储量将近 40MB,分成 10 个盘面。每个盘面有 128 个磁道;每个磁道 64 个扇区;每个扇区 512 个字节;每个簇 16 个扇区。交错因子是 4。磁盘以 3600 rpm 转动。读/写回转臂的平均启动时间是 20ms,移动回转臂的平均时间是每个磁道 0.3 ms。现在假定磁盘中有 128KB 的文件。在一般情况下,读取文件中的所有数据要花多长时间? 假定文件的第一个磁道随机位于磁盘中的某个位置,整个文件放在一组相邻的磁道内,文件完全填满它所在的磁道。每次 I/O 磁头移到一个新的磁道,必须完成一次寻道。给出你的结论。
- 9.3 证明从磁盘中随机选择的两个磁道的平均距离是磁盘中磁道总数的三分之一。
- 9.4 假定一个磁盘配置如下:存储总量将近 1033MB,分成 16 个盘面。每个盘面有 2100 个磁道,每个磁道有 63 个扇区,每个扇区 512 字节,每个簇包括 8 个扇区。其交错因子是 3,磁盘以 7200 r/min 旋转。每个读/写回转臂的平均启动时间是 3ms,移动回转臂的平均时间是每个磁道 0.08ms。现在假定磁盘中有 128KB 的文件。在一般情况下,读取文件中的所有数据要花多少时间? 假定文件中的第一个磁道随机位于磁盘上的某个位置,整个文件放在一组相邻的磁道内,文件完全填满它所占据的磁道。每次 I/O 磁头移动到一个新的磁道,必须完成一次寻道。给出你的结论。
- 9.5 有一个 6250 bpi 的磁带驱动器,记录大小是 160 个字节,一个块间间隔是 0.3 英寸,如果要想 90% 的磁带都包含数据,需要的块因子是多少?
- 9.6 有一个数据库应用程序。假定它花费 100ms 从磁盘中读取一个块,花费 20ms 在一个块中搜索一条记录,主存储器的缓冲池中可以容纳 5 个块的数据。对记录的请求会指定哪个块中包含这条记录。如果一个块被访问,在接下来的 10 个请求中,每个请求访问这个块的可能性是 10%。下面对系统的每一种改进,会有什么样的性能提高:
  - (a) 换一个速度是原来两倍的 CPU。
  - (b) 换一个速度是原来两倍的磁盘。
  - (c) 增加足够的主存储器,使缓冲池的大小是原来的两倍。
- 9.7 一个文件中包含 100 万条记录,这些记录按照关键码值排序。每一次查询都引用记录的关键码。文件存储在磁盘扇区中,每个扇区包含 100 条记录。随机读取一个扇区的平均时间是 50ms。顺序读取扇区只需要 5ms。处理查询的“批处理”算法首先按照查询在文件中出现的次序对查询进行排序,然后顺序读取整个文件,在读取文件的时候按照次序处理所有查询。这个算法隐含说明在开始处理之前,所有查询都必须可以使用。“交互式”算法是按照查询到达的次序处理每一个查询,每次都搜索请求的扇区(除非在偶尔的情况下,一行中的两个请求使用同一个扇区)。详细说明在什么情况下批处理方法比交互式方法更有效。
- 9.8 假定使用缓冲池管理虚拟存储器。缓冲池中包含 5 个缓冲区,每个缓冲区存储一块

数据。存储器访问根据块 ID 进行。假定有下列一组存储器访问：

5 2 5 12 3 6 5 9 3 2 4 1 5 9 8 15 3 7 2 5 9 10 4 6 8 5

对于下面的每一种缓冲池替换策略,说明替换最后缓冲池中的内容。假定缓冲池初始为空。

(a)先进先出。

(b)最不经常使用(只保留当前存储区中块的计数)。

(c)最不经常使用(保留所有块的计数)。

(d)最近最少使用。

- 9.9 假设一条记录长为 32 个字节,一个块长为 1024 个字节(因此每个块有 32 条记录),工作主存是 1MB(还有用于 I/O 缓冲区、程序变量等的额外空间)。对于使用置换选择算法和一趟扫描多路归并的最大文件,预期的大小是多少?解释你是怎样得到这个结果的。
- 9.10 假定工作主存大小是 256KB,分成多个块,每个块 8192 个字节(还有其他的空间用于 I/O 缓冲区、程序变量等)。对于使用置换选择算法和两趟扫描多路归并的最大文件,预期的大小是多少?解释你是怎样得到这个结果的。
- 9.11 判断下面命题是否正确,并加以证明:给定主存空间中有  $M$  条记录大小的堆,如果文件中没有记录在  $M$  前面或者有很多更大的关键码值,置换选择算法就会完全排序一个文件。
- 9.12 假定一个数据库中包含 100 万条记录,每条记录 100 个字节长。估计一下在一个一般的工作站上排序这个数据库要花多少时间。
- 9.13 假定一个公司的计算机配置能够处理它们每个月的工资。进一步假定工资处理的瓶颈是对所有雇员记录的排序操作,这里要使用一个外部排序算法。公司的工资程序编写得很好,计划进行出租服务,为其他公司提供工资处理。总裁有一笔来自第二家公司的处理请求,其雇员数目是这家公司的 100 倍。她意识到自己的计算机在可接受的时间内不能对原来 100 倍的记录排序。描述一下,为了减少处理更大的工资数据库需要的时间,下面对计算机系统进行的改进会产生什么影响?
- (a)CPU 的速度增长 2 倍。
- (b)磁盘 I/O 时间增长 2 倍。
- (c)主存储器访问时间增长 2 倍。
- (d)主存储器大小增长 2 倍。
- 9.14 怎样扩展本章描述的外部排序算法才能处理变长记录?

## 9.11 项目设计

- 9.1 在磁盘中,图片一般作为数据一行一行地存储。考虑一下 16 色图片的情况。这样,可以使用 4 个位表示一个像素。如果允许每个像素 8 个位,就不需要解压缩像素的处理了(由于 1 个像素对应 1 个字节,在大多数机器中字节是最低级的寻址单位)。如果把两个像素压缩到一个字节中,就能够节省空间,但是必须对像素进行解压缩。对于每个像素 8 个位和压缩为每个像素 4 个位两种情况,哪种情况从磁盘中读取并

访问图像中每个像素花的时间更多?对这两种情况进行编程,并比较需要的时间。

9.2 根据 LRU 缓冲池替代策略,实现一个基于磁盘的缓冲池库。磁盘中的块从文件的开始处连续编号,第一块的编号是 0。假定块的大小是 1024 个字节,前 4 个字节用于存储对应缓冲区的块 ID。下面 4 个函数构成了缓冲池库的 ADT。

- `Object New_Block()`:请求创建一个新磁盘块。缓冲池管理器确定磁盘中的下一个空闲块,并且返回一个存储新块的缓冲区的引用(不要忘记把块 ID 放到新缓冲区的前 4 个字节中)。
- `Object Get_Block ( int ID )`:给定一个块 ID,返回包含这个块的缓冲区的引用。如果请求的块当前不在缓冲池中,就需要从磁盘中读入。
- `void Free_Block ( int ID )`:释放一个块中的信息。缓冲池管理器应当把这个块作为未用的,这样就可以服务于下一个 `New_Block` 请求。
- `void Dirty_Block ( int ID )`:认为标识的块具有修改的信息。当从缓冲池中清除“干净”的块时,不需要把它们写回磁盘。当从缓冲池中清除“脏”块时,必须把它们的内容复制到磁盘中。

9.3 根据本章描述的置换选择算法和多路归并实现一个外部排序。对于大记录文件和小记录文件两种情况测试你的程序。记录大小为多大时关键码排序最合适?

## 第 10 章 检 索

组织和检索信息是大多数计算机应用程序的核心,而检索(search)当然是所有计算任务中最频繁的了。可以抽象地把检索看成这样一个过程,这个过程确定一个具有某个值的元素是不是某个集合的成员。对检索更一般的看法是试图在一组记录中找到有某个关键码值的记录,或者找到关键码值符合某些条件的一些记录,例如关键码值在某个值的范围内。

可以像下面这样形式化地定义检索。

**定义 10.1** 假定  $k_1, k_2, \dots, k_n$  是互不相同的关键码值,我们有一个集合  $T$ , 其中有  $n$  条记录,形式如下:

$$(k_1, I_1), (k_2, I_2), \dots, (k_n, I_n)$$

其中  $I_j$  是与关键码  $k_j$  相关联的信息,  $1 \leq j \leq n$ 。给定某个关键码值  $K$ , 检索问题 (search problem) 是在  $T$  中定位记录  $(k_j, I_j)$ , 使得  $k_j = K$ 。检索就是定位关键码值  $k_j = K$  的记录的系统化方法。

检索成功就是找到至少一个关键码值为  $k_j$  的记录, 使得  $k_j = K$ 。检索失败就是找不到记录, 使得  $k_j = K$  (可能不存在这样的记录)。

精确匹配查询 (exact-match query) 是指检索关键码值与某个特定值匹配的记录。范围查询 (range query) 是指检索关键码值在某个指定值的范围内的所有记录。

可以把检索算法分成三类:

1. 顺序表和线性表方法。
2. 根据关键码值直接访问方法 (散列法)。
3. 树索引方法。

本章和下面几章依次介绍这些方法。本章考虑检索存储在线性表和顺序表中数据的方法。顺序表只是数组的另一个名称。本章中的线性表表示线性表的实现, 其中包括链表或数组。尽管在 10.3 节讨论应用于集合的特定技术, 这些方法中的大多数适用于序列 (例如, 允许重复的关键码值)。这一章的前三节讨论的技术最适合检索存储在 RAM 中的一组记录。10.4 节讨论散列技术, 这一技术把数据组织到一个表中, 根据关键码的值确定表中每一条记录的位置。

第 11 章讨论基于树的信息组织方法, 包括常用的称为 B 树的数据结构。有些程序必须组织存储在磁盘中的大量记录, 几乎所有这样的程序都使用散列技术或者 B 树的某种变体。散列方法只能用于特定的访问功能 (例如精确匹配查询), 而且一般只有不允许重复关键码值的时候才适用。当使用散列方法不合适的时候, 基于磁盘的应用程序就可以选择 B 树方法。

### 10.1 检索已排序的数组

在例 3.1 中已经给出了最简单的检索形式: 顺序检索算法。对一个未排序的线性表顺序

检索,在平均和最差的情况下需要  $\Theta(n)$  时间。对于重复检索的大量记录,顺序检索慢得难以忍受。减少检索时间的一种方法是通过排序记录进行预处理。

对于已排序的表,最常用的检索算法是 3.5 节描述的二分法检索。如果对关键码值的分布一无所知,那么二分法检索是检索一个已排序的表的最好算法。然而,有时候我们知道预期的关键码值分布。考虑一下某个人在一部很大的词典中查找一个单词的最一般的行为。大多数人当然不会采用顺序检索方法。一般说来,人们会使用二分法检索的一种改进形式,至少在他们接近要查找的词之前一直采用这种方法。检索一般不从词典的中间开始。如果要查找的词以's'开头,查找者会估计到以's'开头的条目从词典的四分之三处开始。这样,他就会先翻到词典的四分之三处,然后根据所看到的内容决定接下来向哪里翻。也就是说,人们一般根据关键码值预期分布的知识“算出”接下来向哪里翻。这种经过计算的二分法检索形式称为词典检索(dictionary search)或者插值检索(interpolation search)。

词典检索试图利用存储在表中的记录的关键码值估计分布知识。某个关键码在关键码范围内的位置被翻译成表中相应记录的估计位置,并且首先检查这个位置。随着二分法检索的进行,找到的关键码值就会排除这个位置以上或以下的所有记录,然后可以根据找到的关键码实际值计算出表中剩余范围内的新位置。接下来的检查根据新的计算做出。这个过程不断继续,直到找到需要的记录,或者表缩小到没有记录剩下。

当估计的关键码值分布符合关键码值的实际分布时,词典检索比二分法检索更有效。如果估计的分布与实际分布有显著差异,那么词典检索的效率就会非常差。例如,想像一下你在电话簿中检索名字“Young”。在一般情况下,你会到靠近电话簿的后面去找。如果找到一个以“Z”开头的名字,就会稍微向前翻一点。如果下一个找到的名字仍然以“Z”开头,你就会再向前翻一点。如果这本特别的电话簿不同寻常,几乎一半的条目都以“Z”开头,那么你就会多次向前移,每次从检索中排除一些记录。在最极端的情况下,如果关键码值分布的计算很差,词典检索的性能不会比顺序检索更好。

## 10.2 自组织线性表

尽管大多数线性表根据关键码值排列顺序,但是这并不是惟一的方法。为了快速检索,另外一种组织线性表的方法是根据估计的访问频率排列记录。假定我们知道,对于每一个关键码值  $k_i$ ,带有关键码值  $k_i$  的记录被请求的概率是  $p_i$ 。还假定线性表的组织先放请求频率最高的记录,接下来是请求频率次高的记录,依此类推。线性表的检索可以从第一个位置开始顺序进行。经过多次检索,一次检索需要的预计比较数是:

$$\bar{C}_n = 1p_1 + 2p_2 + \cdots + np_n$$

也就是说,访问第一条记录的代价是 1(因为要查看 1 个关键码值),出现这种情况的概率是  $p_1$ 。访问第二条记录的代价是 2(因为必须查看第 1 条记录和第 2 条记录的关键码值),访问它的概率是  $p_2$ ,依此类推。对于这  $n$  条记录,假定所有要检索的记录都存在,  $p_1$  到  $p_n$  的总和一定是 1。

特定的概率分布更容易计算出结果。

**例 10.1** 如果一个线性表中每一条记录被访问到的机会都相同(对一个未排序线性表的经典顺序检索),计算检索这个线性表的预计代价。设  $p_i = 1/n$ ,得到:

$$\bar{C}_n = \sum_{i=1}^n i/n = (n+1)/2$$

这个结果符合我们的预期估计,即正常的顺序检索平均要访问一半记录。如果记录真的有同样的访问概率,那么根据频率排序记录不会有什么好处。

指数概率分布则会产生不同的结果。

**例 10.2** 计算检索一个按照频率排序的线性表的预计代价,其概率定义如下:

$$p_i = \begin{cases} 1/2^i & \text{当 } 1 \leq i \leq n-1 \text{ 时} \\ 1/2^{n-1} & \text{当 } i = n \text{ 时} \end{cases}$$

那么

$$\bar{C}_n \approx \sum_{i=1}^n (i/2^i) \approx 2$$

对于这个例子,预计访问数是一个常数,这是因为访问第一条记录的概率很高,第二个则比第一个低很多,但是还是比访问第三条记录的概率高很多,依此类推。这表明,对于有些概率分布,根据频率排序线性表能够产生很有效的检索技术。

在许多检索应用程序中,实际访问模式遵循一种称为 80/20 规则(80/20 rule)的经验规律。80/20 规则表明 80% 的访问都是对 20% 的记录进行的。值 80 和 20 都是估计值:每一个应用程序都有自己的值。然而,这种性质的行为在实际应用中极其频繁地出现(这说明了磁盘和 CPU 厂商为加速对慢速存储设备中数据的访问而采用缓存技术的成功之处;参见第 9.3 节关于缓冲池的讨论)。当 80/20 规则起作用时,可以估计出一个根据访问频率排序的线性表的合理检索性能。

**例 10.3** 有些自然情况下出现的分布经常遵循一种称为 Zipf 分布的模式。这类例子包括自然语言,如英语中单词使用的频率,城市中的人口规模(例如,人口相对比例相当于“使用频率”)。Zipf 分布与式(2.9)中定义的调和级数有关。对于  $n$  条记录中的第  $i$  项,Zipf 频率定义为  $1/iH_n$ 。这样,对于各项遵循 Zipf 分布的级数,预计代价就是:

$$\bar{C}_n \approx \sum_{i=1}^n i/iH_n = n/H_n \approx n/\log_e n$$

当频率分布遵循 80/20 规则时,在一个按照频率排序的表中检索,平均需要查看十分之一的记录。

在许多应用程序中,无法事先知道哪一条记录被访问到的频率最高。如果考虑得更复杂一点,有些记录可能在一段时间内频繁地被访问,此后就极少被访问了。这样,记录的访问概率就可能随着时间变化,自组织线性表(self-organizing list)就是寻求解决这些问题的。

自组织线性表根据实际的记录访问模式在线性表中修改记录的顺序。自组织线性表使用启发式规则决定如何重新排列线性表<sup>①</sup>。这些启发式规则类似于管理缓冲池(见 9.3 节)的规则。实际上,一个缓冲池就是一种形式的自组织线性表。根据预计的访问频率重新排列缓冲池是一个很好的策略,因为一般情况下必须检索缓冲池的内容,来确定需要的信息是否在主存储器中。当缓冲区按照访问频率排序时,如果这时要读取一页新信息,线性表最后的那个缓冲区最适合重新使用。下面是管理自组织线性表的三个传统的启发式规则:

<sup>①</sup> 一个启发式规则是一个“经验性规律”,即实际上工作得很好的简单规则。

1. 保持一个按照频率排序的线性表最显然的方式是为每一条记录保存一个访问计数,而且一直按照这个顺序维护计数,这种方法称为计数方法(count)。计数方法类似于缓冲池替代策略中的最不经常使用方法(LFU)。这样,每当访问一条记录时,如果记录的访问数已经大于它前面记录的访问数,这条记录就会移到线性表的前面。很明显,计数方法将按照到现在为止实际出现的频率顺序保存记录。除了保存访问计数需要空间以外,计数方法对随着时间的推移而访问频率改变的反应也不好。在频率计数系统中,一旦一条记录被访问了很多次,不管将来的访问历史怎样,它都会一直在线性表的前面。

2. 找到一条记录时就把它放到线性表的最前面,而把其他记录后退一个位置。这种方法类似于最近最少使用(LRU)缓冲池替代策略,称为移至前端方法(move-to-front)。如果使用链表来存储记录,这种启发式规则很容易实现。如果记录存储在数组中,把一条记录从数组的后面向前移动会导致大量记录改变位置。移至前端方法是一个有效的启发式规则,这是因为当至少完成  $n$  次检索时,它需要的访问次数最多是对  $n$  条记录优化静态排序(optimal static ordering)后需要的访问次数的两倍。也就是说,如果事先知道检索序列(至少  $n$  次),而且已经按照频率顺序存储了记录,以便使得这些访问的代价最小,这个代价将至少是移至前端启发式规则需要的代价的一半(这将在 14.3 节使用均摊分析进行证明)。最后,移至前端方法对访问频率的局部变化能够很好地反应,这是因为如果一条记录在一段时间内被频繁访问,在这段时间它就会靠近线性表的前边。

3. 把找到的记录与它在线性表中的前一条记录交换位置。这种启发式规则称为转置(transpose)。无论对于基于链表实现的线性表还是对于基于数组实现的线性表,转置都是一种很好的方法。随着时间的推移,最常使用的记录将移到线性表的前面,曾经被频繁访问但是以后不再使用的记录将会慢慢落到后面。这样,它看起来好像能够很好地响应变化的访问频率。但是,有一些极端访问序列使得转置方法的效果很差。考虑一下访问线性表中最后一条记录(称它为  $X$ )的情况。然后就会把这条记录与倒数第二条记录(称它为  $Y$ )交换位置,使  $Y$  成为最后一条记录。如果现在访问  $Y$ ,它就会与  $X$  交换位置。如果访问序列不断在  $X$  和  $Y$  之间交替,检索就总会查到线性表的最后,这是因为两条记录都不能向前移动。然而,这种情况在实际中很少出现。

**例 10.4** 假定有 8 条记录,其关键码值为 0 到 7,而且最初以升序排列。现在考虑一下应用下面访问模式的结果:

5 3 5 6 4 6 5 0 3 5 6 4

如果线性表根据计数方法组织,经过这些访问,线性表的最后结果就是:

5 6 3 4 0 1 2 7

而 12 次访问的总代价将是 44 次比较(假定当一条记录的频率计数上升时,它移到线性表的前边,成为具有这个频率计数值的最后一条记录。经过前两次访问之后,5 成为第 1 条记录,而 3 成为第 2 条记录)。

如果线性表根据移至前端启发式规则组织,那么最后的线性表就是:

4 6 5 3 0 1 2 7

需要的总比较次数是 54。

最后,如果线性表根据转置启发式规则组织,那么最后的线性表就是:

0 1 5 3 6 4 2 7



需要的总比较次数是 62。

尽管在一般情况下自组织线性表不会像检索树或者已排序的线性表表现得那样好,这二者都需要  $O(\log n)$  检索时间,然而在许多情况下,自组织线性表都证明是一种很有价值的工具。它们优于排序线性表的显著之处是不需要对线性表进行排序,这意味着插入一条新记录的代价很低,当需要频繁插入记录时,这补偿了检索的高代价。自组织线性表比检索树更容易实现,而且对于小的线性表很可能更有效,另外它们不需要额外的空间。最后,对于顺序检索几乎足够快的情况,把一个未排序的线性表改变为自组织的线性表可能会极大地加速应用程序的运行,而这种修改只需稍微增加一点额外代码。

作为应用自组织线性表的一个例子,考虑一个压缩并传送消息的算法。线性表是根据移至前端规则自组织的,根据以下规则以单词和数字形式进行传送:

1. 如果单词前面已经出现,就传送这个单词在线性表中的当前位置,把这个单词移到线性表的前面。
2. 如果单词是第一次出现,就传送这个单词,把这个单词放在线性表的前面。

发送者和接收者都以同样的方式记录单词在线性表中的位置(使用移至前端规则),这样它们就会对编码重复出现单词的数字意义达成一致。例如,考虑一下下面这个要传送的例子(为了简单起见,忽略字母的大小写):

The car on the left hit the car I left.

前三个单词以前没有出现,因此必须把它们作为完整的单词发送。第四个单词是“the”的第二次出现,在这里是线性表中的第三个单词。这样,只需要传送位置值“3”。接下来的两个单词都没有出现过,因此必须作为完整的单词传送。第七个单词是“the”的第三次出现,很巧合,仍然在第三个位置。第八个单词是“car”的第二次出现,它现在在线性表的第五个位置。“I”是一个新单词,最后一个单词“left”现在在第五个位置。这样,整个传送就是:

The car on 3 left hit 3 5 I 5 .

这种压缩方法的思想类似于 Ziv-Lempel 编码算法(Ziv-Lempel coding)。Ziv-Lempel 编码是文件压缩工具中经常使用的一类编码算法。Ziv-Lempel 编码在遇到字符串的重复出现时,会使用一个指向字符串在文件中第一次出现位置的指针来代替。为了加快检索一个前面已经出现的单词需要的时间,编码存储在一个自组织的线性表中。

### 10.3 集合的检索

确定一个值是不是某个集合的元素,这是在一组记录中检索关键码的一种特殊情况。这样,本书中讨论的任何一种检索方法都可以用于检查集合中的元素。不仅如此,还可以利用这个问题的限制条件加速检索过程。

对于限定关键码范围的情况,可以采用一种简单的技术,这就是存储一个位数组,为每一个可能的元素分配一个比特位位置。对于包含在实际集合中的元素,把它们对应的位设置为 1;对于不包含在集合中的元素,把它们对应的位设置为 0。例如,考虑一下 0 到 15 之间的素数集合,图 10.1 显示了相应的位表。要确定某一个值是不是素数,只需检查对应的位。这种表示方法称为位向量(bit vector)。

如果集合的大小适合计算机的一个字长,那么通过逻辑上的位操作,就可以完成集合的



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0

图 10.1 0 ~ 15 中素数集合的位表。当且仅当位置  $i$  是素数时把这个位置的位设置为 1

并、交、差运算 集合  $A$  与  $B$  的并运算就是按位或 OR 函数(在 Java 中是 `|` 符号)。集合  $A$  与  $B$  的交运算就是按位与 AND 函数(在 Java 中是 `&` 符号)。例如,如果要计算数 0 到 15 之间奇素数集合,只需要计算表达式:

0011010100010100 & 0101010101010101

集合  $A$  与  $B$  的差运算  $A - B$  在 Java 语言中可以使用表达式  $A \& \sim B$  ( $\sim$  是非运算的符号)实现。对于不能放到计算机的一个字中的集合,可以依次对组成整个位向量的字完成对应的操作。

这种根据位向量计算集合的方法有时候可以用于文档检索(document retrieval)。考虑这样一个问题,从一组文档中挑选出包含某些选定的关键字。对于每一个关键字,文档检索系统存储一个位向量,每个文档一位。如果用户想要知道哪些文档包含某三个关键字,就把相应的三个位向量进行 AND 操作。位置上值为 1 的位就对应所需要的文档。同样,可以为每个文档存储一个位向量,标识在文档中出现的关键字。这种组织方法称为签名文件(signature file)。可以通过对签名操作找到带有所需关键字组合的文档。

## 10.4 散列方法

这一节提供一种完全不同的检索表的方法:根据关键码值直接访问记录。把关键码值映射到表中的位置来访问记录的过程称为散列(hashing)。大多数散列方法按照地址计算的顺序把记录放到表中。这样,就不按值的顺序或者频率的顺序放置记录了。把关键码值映射到位置的函数称为散列函数(hash function),通常用  $h$  来表示。存放记录的数组称为散列表(hash table),用  $T$  来表示。散列表中的一个位置称为一个槽(slot)。散列表  $T$  中槽的数目用变量  $M$  表示,槽从 0 到  $M - 1$  编号。设计散列方法的目标是使得对于任何关键码值  $K$  和某个散列函数  $h$ ,  $0 \leq h(K) \leq M$ , 而且  $T[h(K)].key() = K$ 。

散列方法一般只适用于集合,通常不适用于允许多条记录有相同关键码值的应用程序。散列方法一般不适用于范围检索。也就是说,不能很方便地找到关键码值在某个范围的所有记录(如果有),也不能找到最大或者最小关键码值的记录,或者按照关键码值的顺序访问记录。散列方法最适合回答这样的问题:“如果有,哪条记录的关键码值是  $K$ ?”对于可以把访问方式限制到这类查询的应用程序,散列方法通常是可供选择的检索方法。因为如果实现得正确,它的效率非常高。然而,在这一章就会看到,有许多实现散列的方法,很容易设计出不合适的实现。散列方法既适合基于主存储器的检索,也适合基于磁盘的检索。组织存储在磁盘上的大型数据库有两个广为使用的方法,散列方法是其中之一(另一种方法是 B 树,在第 11 章中介绍)。

作为对散列概念简单但不实际的介绍,考虑一下相对于记录数来说关键码值的范围很小的情况。例如,有  $n$  条记录,每条记录有惟一的关键码值,范围为 0 到  $n - 1$ 。在这个简单的例子中,带有关键码值  $i$  的记录可以存储在  $T[i]$  中,散列函数只是  $h(K) = K$  (实际上,在这

种情况下根本不需要把关键码值作为记录的一部分存储,因为它与索引是相同的)。要找到带有关键码值  $i$  的记录,只要简单地查看  $T[i]$ 。

一般说来,关键码值范围中的值比散列表中的槽多。考虑一个更为实际的例子,假定关键码可以取 0 到 65 535 范围内的任意值(例如,关键码是一个两字节无符号整数),而且我们预计在任何给定的时间段内存储 1000 条记录左右。在这种情况下,使用一个带有 65 536 个槽的散列表是不可行的,其中大部分都会为空。因而,必须设计一个散列函数,允许把记录存储在一个更小的表中。由于可能的关键码值的范围比表的尺寸大,至少有些槽会有多个关键码值映射到。对于一个散列函数  $h$  和两个关键码值  $k_1, k_2$ , 如果  $h(k_1) = \beta = h(k_2)$ , 其中  $\beta$  是表中的一个槽,那么就说  $k_1$  和  $k_2$  对于  $\beta$  在散列函数  $h$  下有冲突(collision)。

在一个根据散列方法组织的数据库中,找到带有关键码值  $K$  的记录包括两个过程:

1. 计算表的位置  $h(K)$ 。
2. 从槽  $h(K)$  开始,使用(如果需要)冲突解决策略找到包含关键码值  $K$  的记录。

### 10.4.1 散列函数

散列方法一般用于记录的关键码值范围很大的情况,并且把记录存储在一个槽数目相对较少的表中。当两条记录映射到表的同一个槽中时,就会发生冲突。如果选择散列函数时很细心,或者很幸运,那么冲突的实际数目就会很小。但是,即使在最好的情况下,冲突也几乎是不可避免的<sup>①</sup>。例如,考虑下一间坐满学生的教室。某些学生有同样生日的可能性是多少(即,某年的同一天,不需要同一年)? 如果有 23 个学生,那么很少可能有 2 个学生有同样的生日。尽管学生的生日可以在 365 天中的任何一天(不考虑闰年),大多数日期不会是班级学生的生日。学生越多,具有同样生日的可能性就越大。把学生根据生日映射到日期类似于使用生日作为散列函数把记录分配给表中的槽(大小是 365),这并不表示哪些学生具有同样的生日,或者在一年中的哪一天会有人有同样的生日。

考虑更为实际的情况,根据散列方法组织的数据库必须使得存储的记录相对于散列表的槽数目占很高的比例,一般要使表至少一半是满的。由于在这种情况下很可能会发生冲突,那么是否需要考虑散列函数避免冲突的能力呢? 当然需要。从技术上说,任何能够把所有可能的关键码值映射到散列表槽中的函数都是散列函数。把所有记录都映射到同一个槽中的函数也是散列函数,但是它对于记录检索没有任何帮助。

一般说来,希望选择的散列函数能够把记录以相同的概率分布到散列表的所有槽中。某一个散列函数具体做得怎么样依赖于关键码在允许的关键码值范围内的分布。在有些情况下,数据在整个范围内分布得很好。例如,如果输入是规范地从关键码范围内选择的一组随机数,而且散列函数分配关键码范围时使得散列表中的每一个槽对应同样大小的一部分范围,那么散列函数就可能把输入记录规范地分布到表中。然而,在许多应用程序中,记录的关键码值高度聚集或者分布很差。当输入的记录在整个关键码值范围分布得不好时,很难设计出一个

<sup>①</sup> 这里的例外就是完美散列(perfect hashing)。完美散列是没有散列记录冲突的系统。在为一组记录选择散列函数时,在选择散列函数之前就需要确定整个记录组。完美散列很有效,因为它总是会在散列函数计算的位置找到正要查找的记录:只需要一次访问就能找到记录。选择一个完美的散列函数代价很高,但是在需要很高的检索性能时也是值得的。一个例子就是检索 CD-ROM 中的数据。在这种情况下数据库永远也不会改变,每次检索的时间代价很高,数据库设计者可以在发行 CD-ROM 之前建立散列表。

散列函数把记录很好地分布在表中,特别是事先不知道输入分布的时候尤其如此。

有许多原因导致数据值的分布很差。自然的分布是指数形式的。例如,美国前 100 个最大城市的人口数聚集在分布的底部,有一些个别城市则在顶部(这就是 Zipf 分布的一个例子,见 10.2 节)。收集到一起的数据很可能被以某种方式扭曲。例如,某领域中的样本被舍入,比如说接近 5(即,所有的数字以 5 或 0 结尾)。如果输入是一组常用英语单词,开头字母的分布就会很差。注意在每一个例子中,关键码的高位或者低位的分布很差。

设计散列函数时,一般会遇到下面两种情况之一:

1. 对关键码值的分布一无所知。在这种情况下,希望选择的散列函数在关键码值范围内能够产生一个规范的关键码值随机分布,同时避免明显的聚集可能性,如对关键码值的高位或低位敏感的散列函数。

2. 对关键码值的分布有所了解。在这种情况下,应当使用一个依赖于分布的散列函数,避免把相关的关键码值映射到散列表的同一个槽中。例如,如果对英文单词进行散列,就不应当对第一个字符的值散列,因为这样很可能使分布不均匀。

下面是说明这些观点的几个散列函数的例子。

**例 10.5** 考虑下面这个散列函数,它把整数散列到一个有 16 个槽的表中。

```
static int h(int x) {  
    return(x % 16);  
}
```

这个散列函数返回的值只依赖于关键码的最低四位。由于这些位的分布可能很差(例如,相当比例为偶数,低位为 0),结果分布也就可能很差。这个例子表明表的大小  $M$ (一般是用作取模的值)对散列函数的好坏很关键。

**例 10.6** 有一个用于数值的很好的散列函数,称为平方取中方法(mid-square method)。平方取中方法计算关键码值的平方,对于长度为  $2r$  的表,取出结果的中间  $r$  位。由于关键码值的大多数位或者所有位都对结果有所贡献,这样做的效果很好。

**例 10.7** 下面是一个用于字符串的散列函数:

```
static int h(String x, int M) {  
    int i, sum;  
    for (sum = 0, i = 0; i < x.length(); i++)  
        sum += (int) x.charAt(i);  
    return( sum % M);  
}
```

这个函数把字符串中字母的 ASCII 值累加起来。如果  $M$  很小,它就能够平均地把字符串分布到散列表的槽中,因为它对所有的字符都给予同样的权重。这就是折叠方法(folding method)的一个例子。字符串中字符的顺序对散列函数的结果没有影响。还有一个类似的用于整数的方法,它把组成关键码值的数字累加起来,假定有足够的数字满足(1)使得一个或者两个分布不好的数字不会影响过程的结果;(2)产生一个比  $M$  还大的和。像许多散列函数一样,最后一步就是对结果使用取模操作,使用表的长度  $M$  在表的长度范围内产生一个值。如果和不够大,那么取模操作就会产生很差的分布。因为“A”的 ASCII 值是 65,所以一个大写字

母字符串的和总是在 650 到 900 之间。对于一个长度为 100 或者更小的散列表,会有一个很好的分布。对于一个大小为 1000 的散列表,分布就会极差。

**例 10.8** 下面是一个用于字符串的更好的散列函数:

```
static long ELFhash (String key, int M) {
    long h = 0;
    for( int i = 0; i < key.length(); i++ ) {
        h = &h < < 4 ) + (int) key.charAt(i);
        long g = h & 0xF0000000L;
        if (g != 0) h ^= g > > > 24;
        h &= ~g;
    }
    return h % M;
}
```

因为在 UNIX 系统 V Release 4 的 ELF 文件格式中使用,这个函数称为 ELFhash, ELF 文件格式用于存储可执行文件和目标文件。它是散列函数的典型实际应用。ELFhash 把一个固定长度的字符串作为输入。它对于长字符串和短字符串都很有效,字符串中每个字符都有同样的作用,它通过一种方式把字符的十进制值结合起来,这种方式可以产生散列表位置的平均分布。

### 10.4.2 开散列方法

尽管散列函数的目标是使冲突最少,实际上冲突还是无法避免的。这样,散列方法的实现必须包括冲突解决策略。冲突解决技术可以分为两类:开散列方法(open hashing,也称为单链方法,separate chaining)和闭散列方法(closed hashing,也称为开地址方法,open addressing)<sup>①</sup>。这两种方法的不同之处与冲突记录的存储有关,开散列方法把冲突记录存储在表外,闭散列方法把冲突记录存储在表中的另一个槽内。这一节介绍开散列方法,10.4.3 节介绍闭散列方法。

一种简单形式的开散列方法是把散列表中的每一个槽定义为一个链表的表头,散列到某一个槽的所有记录都放到这个槽的链表内。图 10.2 是一个散列表,这个表中的每一个槽存储一条记录以及一个指向链表其余部分的指针。

一个槽链接的线性表中的记录可以按照多种方式排列:按照插入次序排列、关键码值的次序排列或者访问频率的次序排列。对于检索不成功的情况,根据关键码值排序是有好处的,因为一旦在线性表中遇到一个比要检索的关键码值大的关键码,就知道应该停止检索了。如果线性表中的记录没有排序或者按照访问频率排序,那么一次不成功的检索就需要访问线性表中的每一条记录。

对于一个存储  $N$  条记录长度为  $M$  的表,散列函数(在理想情况下)将把记录在表中的  $M$  个位置平均放置,使得平均每个链表中有  $N/M$  条记录。假定表中的槽比存储的记录还多,很少会有槽包含多于一条记录。在一个链表为空或者只有一条记录的情况下,一次检索只需要访问一次链表。这样,散列方法的平均代价就是  $\Theta(1)$ 。然而,如果聚集使得许多记录散列到

<sup>①</sup> 是的,“开散列”与“开地址”的意义相反会引起混淆,但实际上就是这样的。

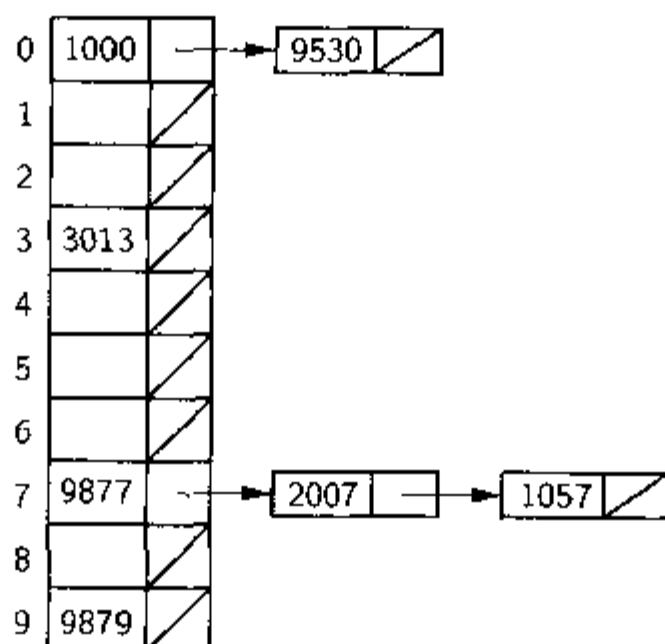


图 10.2 一个有 7 个数的开散列方法。这 7 个数存储在有 10 个槽的散列表中,使用的散列函数是  $h(K) = K \bmod 10$ 。数的插入顺序是 9877、2007、1000、9530、3013、9879 和 1057。有两个值散列到第 0 个槽,1 个值散列到第 3 个槽,3 个值散列到第 7 个槽,1 个值散列到第 9 个槽

某几个槽中,那么访问一条记录的代价就会更高,因为需要检索链表中的许多元素。

当把散列表放到主存储器中,用一个标准的存储器内部链表实现时,开散列方法最合适。在磁盘中用一种很有效的方式存储一个开散列表是很困难的,因为一个链表中的多个元素可能存储在不同的磁盘块中,这就会导致检索一个关键码值产生多次磁盘访问,从而背离了使用散列方法的目的。

可以观察到,开散列方法和分配排序(Binsort)之间有一些类似的地方。观察开散列方法的一种方式是把每一条记录放到一个盒子中。多条记录可能被散列到同一个盒子中,但是这条记录在各个盒子中的初始分配应当极大地减少检索操作访问的记录数。简单的分配排序与此类似,也可以减少每个盒子中的记录数,使这些个数很少的记录可以使用其他方式排序。

### 10.4.3 闭散列方法

闭散列方法把所有记录直接存储在散列表中。每条记录  $i$  有一个基位置(home position)就是  $h(k_i)$ ,即由散列函数计算出来的槽。如果要插入一条记录  $R$ ,而另一条记录已经占据了  $R$  的基位置,那么就把  $R$  存储在表中的其他槽内,由冲突解决策略确定应该是哪个槽。自然,检索时也要像插入时一样,遵循同样的策略,以便重复进行冲突解决过程,找出在基位置没有找到的记录。

#### 桶式散列

闭散列的一种途径是把散列表中的槽分成多个桶(bucket)。把散列表看作有  $M$  个槽的数组,分成  $B$  个桶,每个桶中有  $M/B$  个槽。散列函数把每条记录分配到某个桶的第一个槽中。如果这个槽已经被占用,那么就把这条记录在桶中向下移,直到找到一个空槽。如果一个桶被完全占满,那么就把这条记录存储在表的最后具有无限容量的溢出桶(overflow bucket)中。所有桶共享同一个溢出桶。好的实现方法使用的散列函数会把记录在各个桶之间平均分布,使得进入溢出桶的记录尽可能少。图 10.3 说明的就是桶式散列方法。

散列表		溢出	
0	1000 9530	1057	
1			
2	9877 2007		
3	3013		
4	9879		

图 10.3 一个有 7 个数的桶式散列方法。这 7 个数存储在一个包含 5 个桶的散列表中,使用的散列函数是  $h(K) = K \bmod 5$ 。每个桶包含两个槽。数值的插入顺序是 9877、2007、1000、9530、3013、9879 和 1057。这些值中有两个值散列到第 0 个桶,三个值散列到第 2 个桶,一个值散列到第 3 个桶,一个值散列到第 4 个桶。由于第二个桶不能放下 3 个值,把第 3 个值放到溢出桶中

当检索一条记录的时候,第一步就是散列关键码,确定哪个桶包含这条记录。然后就检索这个桶中的记录。如果没有找到需要的关键码值,而桶内仍然有空槽,那么就结束检索。如果桶已经满了,那么需要的记录有可能存储在溢出桶中。在这种情况下,就必须检索溢出桶,直到找到记录,或者溢出桶中所有记录都已经检查过了。如果溢出桶中的记录很多,这将是一个非常耗时的过程。

桶式散列的一个简单变体是首先把关键码值散列到它的基位置,好像没有使用桶式散列一样。如果基位置已满,那么就把记录压向桶的后面。如果到达了桶底,那么冲突解决例程就到桶的上面查找空槽。例如,假定桶中包含 8 条记录,第一个桶由第 0 到第 7 个槽组成。如果一条记录散列到第 5 个槽,冲突解决过程就会尝试按照下面的次序把记录插入表中,5、6、7、0、1、2、3,最后是 4。如果桶中所有的槽都满了,那么就把记录放到溢出桶中。这种方法的优点是减少了冲突,这是因为所有的槽都可以是基位置,而不仅仅是桶中的第一个槽。这种桶式散列方法非常适合基于磁盘的散列表。如果使桶的大小等于磁盘块的大小,就会使得一条记录与它的基位置在同一个磁盘块的机会最大。

桶式方法适用于实现基于磁盘的散列表,因为桶的大小可以设置为磁盘块的大小。每当进行检索或者插入的时候,就把整个桶读入主存储器。对于插入或者检索的所有处理都在这一次磁盘访问中,除非桶已经满了。如果桶已经满了,还要从磁盘中读出溢出桶。自然,应当使得溢出很小,从而使不必要的磁盘访问最少。

### 线性探查

现在转向“经典”形式的散列方法:不采用桶式散列方法的闭散列,它的冲突解决策略可以使用散列表中的任何槽。

在插入期间,冲突解决策略的目标就是当记录的基位置已经被占用时在散列表中找到一个空槽。可以这样把任何冲突解决方法都看成产生一组能够放置记录的散列表的槽。该组第一个槽就是关键码的基位置。如果基位置被占用,冲突解决策略就会到达此组中的下一个槽。如果这个槽也被占用了,那么就必须找到另外一个槽,依此类推。这组槽就称为冲突解决策略产生的探查序列(probe sequence)。插入过程工作如下:

```
void hashInsert( Elem R ) {           // Insert record R into T
    int home;                          // Home position for R
    int pos = home = h(R.key()); // Initial position
    for( int i = 1; T[pos] != null; i ++ ) {
        pos = ( home + p( R.key(), i )) % M; // Next probe slot
        Assert.assertFalse( T[pos].key() != R.key(),
            "Duplicates not allowed");
    }
    T[pos] = R;                        // Insert R
}
```

在这个实现中,  $p(K, i)$  是探查函数(probe function), 为关键码  $K$  的探查序列的第  $i$  个槽返回相对于基位置的偏移量。

从散列表中检索记录和插入记录使用同样的探查序列。通过这种方式, 就可以找到不在基位置的记录。下面是用 Java 语言实现的检索过程。

```
Elem hashSearch( int K ) {           // Search for the record with key K
    int home;                          // Home position for K
    int pos = home = h(k); // Initial position
    for( int i = 1; (T[pos] != null) && (T[pos].key() != K); i ++ )
        pos = ( home + p(K, i) ) % M; // Next probe position
    if (T[pos] == null) return null; // K not in hash table
    else return T[pos];               // Found it
}
```

插入和检索例程都假定每个关键码的探查序列中至少有一个槽是空的, 否则它们就会进入一个无限循环中。

关于桶式散列方法的讨论提供了一种简单的解决冲突的方法。如果记录的基位置被占用, 那么就在桶中下移, 直到找到一个空槽。这就是称为线性探查(linear probing)的冲突解决技术的一个例子。用于简单线性探查的探查函数是:

$$p(K, i) = i$$

一旦到达表的底部, 探查序列就折回到表的开始处。线性探查的优点是在探查序列回到基位置之前, 表中所有的槽都可以作为插入新记录的候选位置。

作为一种解决冲突的技术, 线性探查有一些明显的缺陷。主要问题已经在图 10.4 中说明了。这里是一个有 11 个槽的散列表, 用于存储 4 位数, 散列函数是  $h(K) = K \bmod 11$ 。在图 10.4(a) 中, 已经把 6 个数放到了表中, 还剩下 5 个空槽。

在理想情况下, 表中的每一个空槽都有同样的机会接收下一个要插入的记录。在这个例

0	1001	0	1001
1	9537	1	9537
2	3016	2	3016
3		3	
4		4	
5		5	
6		6	
7	9874	7	9874
8	2009	8	2009
9	9875	9	9875
10		10	1052

(a)
(b)

图 10.4 线性探查问题的例子

(a)插入 6 个值,顺序是 9874、2009、1001、9537、3016 和 9875,使用的散列函数是  $h(K) = K \bmod 11$ 。(b)把值 1052 添加到散列表中

子中,散列函数给予每个槽(大致上)同样的可能性,使其成为下一个关键码的基位置。然而,考虑一下如果下一条记录关键码的基位置是第 0 个槽会发生什么情况呢?线性探查会把记录放到第 3 个槽中。对于基位置在第 1 个和第 2 个槽的记录也会发生同样的情况。基位置在第 3 个槽的记录自然要放在第 3 个槽中。这样,下一条记录放到第 3 个槽中的概率就是  $4/11$ 。类似地,可以把散列到第 7、8、9 个槽中的记录放到第 10 个槽中。然而,只有散列到第 4 个槽中的记录会存储到第 4 个槽内,这种情况发生的概率是  $1/11$ 。同样,下一条记录放到第 5 个槽中的概率也是  $1/11$ ,第 6 个槽也是一样。这样,结果概率就不再相等了。

更糟糕的是,如果下一条记录放到第 10 个槽中,如图 10.4(b)所示,那么接下来的记录放到第 3 个槽中的概率就是  $8/11$ 。线性探查这种把记录聚集到一起的倾向称为基本聚集(primary clustering)。小的聚集可能汇合成大的聚集,使得问题更糟。基本聚集的问题是它会导致很长的探查序列。

### 改进的冲突解决方法

怎样才能避免基本聚集呢?一种可能的方式是仍然使用线性探查,但是以常数  $c$  而不是以 1 跳过。也就是说,探查序列中的第  $i$  个槽将是  $(h(K) + ic) \bmod M$ 。通过这种方式,基位置相邻的记录就不会进入同一个探查序列了。

在回到基位置之前,探查序列应该把散列表的所有槽都走一遍。但并不是所有的探查函数都有这个特性。例如,如果  $c = 2$ ,而且表中包含偶数个槽,那么任何基位置在偶数槽的关键码的探查序列将只走遍所有偶数槽。同样,基位置在奇数槽的关键码的探查序列将走遍所有奇数槽。这样,表的长度和线性探查常数的这种组合就把关键码值有效地分成了两个集合,这两个集合存储在散列表中两个不连贯的部分里。为了使探查序列走遍表中所有槽,常数  $c$  必须与  $M$  互素。对于一个长度  $M = 10$  的散列表,如果  $c$  是 1、3、7 或 9 中的一个,那么任何关键码的探查序列都会走遍所有的槽。当  $M = 11$  时候, $c$  取 1 到 10 之间的任意值,对任何



关键码值都会产生一个走遍所有槽的探查序列。

考虑一下这种情况,  $c = 2$ , 要插入一条记录, 它的关键码是  $k_1$ ,  $h(k_1) = 3$ 。  $k_1$  的探查序列是 3、5、7、9 等等。如果另一个关键码  $k_2$  的基位置在第 5 个槽, 那么它的探查序列就是 5、7、9 等等。  $k_1$  和  $k_2$  的探查序列链到了一起, 从而导致了聚集。也就是说, 值  $c > 1$  的线性探查不能解决基本聚集问题。我们希望找到一个探查函数, 它不会把关键码以这种方式链到一起。最好  $k_1$  的探查序列在序列的第 1 步以后就不与  $k_2$  的探查序列相同。相反, 这些探查序列应当叉开。

理想的探查函数应当在探查序列中随机地从未走过的槽中选择下一个位置, 即探查序列应当是散列表位置的一个随机排列。但是, 实际上不能随机地从探查序列中选择下一个位置, 因为在检索关键码的时候不能建立起同样的探查序列。然而, 可以做一些类似于伪随机探查(pseudo-random probing)的事情。在伪随机探查中, 探查序列中的第  $i$  个槽是  $(h(K) + r_i) \bmod M$ ,  $r_i$  是 1 到  $M - 1$  之间数的“随机”数序列。所有插入和检索都使用相同的“随机”数序列。探查函数就是:

$$p(K, i) = \text{Perm}[i - 1]$$

这里 Perm 是一个长度为  $M - 1$  的数组, 它包含值从 1 到  $M - 1$  的随机序列。

**例 10.9** 考虑一个长度为  $M = 101$  的表,  $r_1 = 2, r_2 = 5, r_3 = 32$ 。假定有两个关键码  $k_1$  和  $k_2$ ,  $h(k_1) = 30, h(k_2) = 28$ 。  $k_1$  的探查序列是 30、32、35、62。  $k_2$  的探查序列是 28、30、33、60。这样, 尽管  $k_2$  在第 2 步就会探查得到  $k_1$  的基位置, 此后这两个关键码的探查序列马上就分开了。

清除基本聚集的另一种技术称为二次探查(quadratic probing)。这里, 探查序列中的第  $i$  个值是  $(h(K) + i^2) \bmod M$ 。这样, 探查函数就是:

$$p(K, i) = i^2$$

同样, 具有不同基位置的两个关键码具有不同的探查序列。

**例 10.10** 对于一个长度  $M = 101$  的表, 假定对于关键码  $k_1$  和  $k_2$ ,  $h(k_1) = 30, h(k_2) = 29$ 。  $k_1$  的探查序列是 30、31、34、39,  $k_2$  的探查序列是 29、30、33、38。这样, 尽管  $k_2$  会在第 2 步探查  $k_1$  的基位置, 这两个关键码的探查序列此后就立即分开了。

伪随机探查和二次探查都能够消除基本聚集, 即关键码共享探查序列段的问题。然而, 如果两个关键码散列到同一个基位置, 那么它们就会具有同样的探查序列。这是因为伪随机探查和二次探查产生的探查序列只是基位置的函数, 而不是原来关键码值的函数(无论是伪随机探查还是二次探查, 函数  $p$  都不使用参数  $K$ )。如果散列函数在某一个基位置聚集, 那么伪随机探查和二次探查仍然会保持聚集。这个问题称为二级聚集(secondary clustering)。

为了避免二级聚集, 需要使探查序列是原来关键码值的函数, 而不是基位置的函数。这种方式的一种简单技术就是使探查函数的常量依赖于第二个散列函数, 再进行线性探查。这样, 探查序列的形式就是:

$$p(K, i) = i * h_2(K)$$

这种方法称为双散列方法(double hashing)。

**例 10.11** 还假定一个散列表的长度是  $M = 101$ , 有 3 个关键码  $k_1, k_2$  和  $k_3$ ,  $h(k_1) = 30, h(k_2) = 28, h(k_3) = 30, h_2(k_1) = 2, h_2(k_2) = 5, h_2(k_3) = 5$ 。那么  $k_1$  的探查序列就是 30、32、34、36 等等。  $k_2$  的探查序列就是 28、33、38、43 等等。  $k_3$  的探查序列是 30、35、40、

45 等等。这样关键码之间就会共享同一段探查序列了。当然,如果第 4 个关键码  $k_4$  有  $h(k_4) = 28, h_2(k_4) = 2$ , 那么它就会与  $k_1$  有同样的探查序列。伪随机探查和二次探查可以与双散列方法结合起来解决这个问题。

好的双散列方法实现应当保证所有探查序列常数都与表的长度  $M$  互素。这很容易做到。一种方法就是选择  $M$  是一个素数,  $h_2$  返回的值在  $1 \leq h_2(K) \leq M - 1$  范围之间。另一种方式就是对于某个值  $m$ , 设置  $M = 2^m$ , 让  $h_2$  返回一个 1 到  $2^m$  之间的奇数值。

### 闭散列方法分析

散列方法的效率怎么样呢? 可以根据完成一次操作需要访问的记录数来衡量散列方法的性能。这里关心的基本操作是插入、删除和检索。把成功的检索和不成功的检索区分开是很有用的。在删除一条记录之前必须先找到它。这样, 删除一条记录之前需要访问的记录数等于成功检索到它需要访问的记录数。要插入一条记录, 必须能够沿着记录的探查序列找到一个空槽。这等于对这条记录进行一次不成功的检索(回忆一下, 由于不允许两条记录具有同样的关键码值, 成功检索到这条记录就会产生一个错误)。

当散列表为空的时候, 插入的第一条记录总会找到空基位置。这样, 找到一个空槽只需要一次记录访问。如果所有记录都存储在它自己的基位置, 那么成功的检索只需要一次记录访问。随着表的不断填入, 把记录插入到它的基位置的概率就减小了。如果一条记录散列到一个已被占用的槽, 那么冲突解决策略必须能够找到另一个槽来存储它。如果要查找的记录没有保存在它的基位置, 也需要额外的记录访问, 这是因为要沿着记录的探查序列查找它。随着表的不断填充, 越来越多的记录有可能被放到离其基位置很远的地方。

根据这些讨论, 可以看到散列方法预计的代价是表填充程度的一个函数。定义表的负载因子(load factor)为  $\alpha = N/M$ , 其中  $N$  是表中当前的记录数。

在假定探查序列沿着散列表中的槽随机排列的情况下, 可以通过分析推知插入(或者一次不成功的检索)的预计代价估计值是  $\alpha$  的函数。发现基位置被占用的概率是  $\alpha$ 。发现基位置和探查序列中下一个槽都被占用的概率是  $\frac{N(N-1)}{M(M-1)}$ ,  $i$  次冲突的概率是:

$$\frac{N(N-1)\cdots(N-i+1)}{M(M-1)\cdots(M-i+1)}$$

如果  $N$  和  $M$  都很大, 那么这大约是  $(N/M)^i$ 。预计的探查数是 1 加上  $i$  次冲突在  $i \geq 1$  时的概率之和, 大致上是:

$$1 + \sum_{i=1}^{\infty} (N/M)^i = 1/(1-\alpha)$$

一次成功的检索(或者一次删除)的代价与插入的代价相同。然而, 插入代价的预计值不是在删除的时候, 而是在插入的时候, 依赖于  $\alpha$  的值。可以通过从 0 到  $\alpha$  的当前值的积分推导出这个代价的一个估计(实质上是所有插入代价的一个平均值), 得到结果:

$$\frac{1}{\alpha} \int_0^{\alpha} \frac{1}{1-x} dx = \frac{1}{\alpha} \log_e \frac{1}{1-\alpha}$$

有一点很重要, 要认识到这三个方程表示预计操作的代价, 使用了不合实际的假设, 假设探查序列基于散列表中槽的随机排列(这样就会避免由于聚集产生的所有代价)。这样, 这些

代价就是平均情况的下限估计。分析表明,线性探查插入和不成功检索的平均代价是 $\frac{1}{2}(1 + 1/(1 - \alpha)^2)$ ,删除和成功检索的平均代价则是 $\frac{1}{2}(1 + 1/(1 - \alpha))$ 。10.5节的参考文献中可以找到这些结果的证明。

图 10.5 显示了这四个方程的图形,帮助直观理解基于负载因子的散列方法的预计性能。两条实线显示的是“随机”探查序列在(1)插入或者不成功检索和(2)删除或者成功检索的情况下的代价。根据估计,由于这些操作一般沿着探查序列深入地向下检索。插入或者不成功检索的代价增长得很快。两个虚线显示用于线性探查的同类代价。根据估计,线性探查的代价比“随机”探查的代价增长得更快。

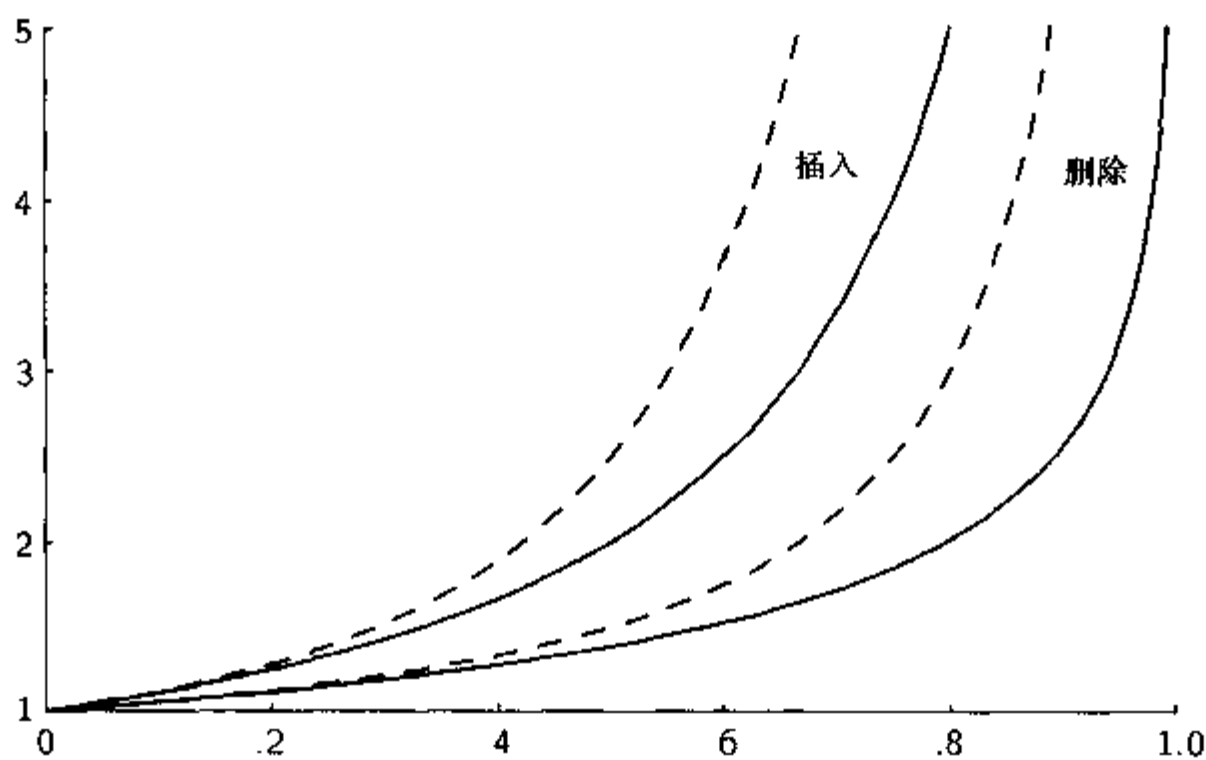


图 10.5 负载因子为  $\alpha$  的预计记录访问增长图。横轴是  $\alpha$  的值,纵轴是对散列表访问的预计数。实线表示随机探查的代价,而虚线表示线性探查的代价。最左边的两条线表示插入和不成功检索的代价;右边的两条线表示删除和成功检索的代价

从图 10.5 可以看到,散列方法的代价一般接近一条记录的访问,非常有效,比需要  $\log n$  次记录访问的二分法检索好得多。随着  $\alpha$  的增长,预计的代价也会增长。对于很小的  $\alpha$  值,预计的代价也很低,它小于 2,直到散列表将近半满。根据这个分析,实际经验是系统的设计要使散列表将近半满,因为超过这个点,性能就会急剧下降。这需要实现者对于在最大负载情况下表中可能有多少记录有所了解,从而相应地选择表的长度。

你可能会注意到,把  $\alpha$  限制到 50% 的建议与基于磁盘的空间/时间权衡原则相抵触,这个原则尽量减少磁盘空间的使用,以增加信息密度。散列方法代表一种不常见的情况,这是因为从引用的局部性中得不到什么好处。从一种意义上说,散列系统的实现者尽一切可能消除引用局部性的作用! 在一个设计得很好的散列系统中,如果有一个包含最近访问过的记录的磁盘页,下一次记录操作访问同一个磁盘页的机会并不比随机情况更好,这是因为一个好的散列实现会打破检索关键码之间的关系。散列方法并不利用引用局部性改进性能,它通过增加散列表的空间换取记录在其基位置的概率更大。这样,散列表的可用空间越多,散列方法就越有效。

根据记录的访问模式,甚至在面临冲突的情况下也有可能减少访问的预计代价。回忆一下 80/20 规则:80% 的访问集中于 20% 的数据。也就是说,对一些记录的访问更频繁一些。如果两条记录散列到同一个基位置,哪一个放到基位置最好?哪一个放到探查序列下面的一个槽中呢?答案是访问频率更高的记录应当放到基位置,因为这样可以减少记录访问总数。在理想情况下,记录应当沿着探查序列按照访问频率排序。

近似于这个目标的一种方法是每当访问记录的时候,就沿着探查序列修改记录的次序。如果要检索的记录不在它的基位置,就使用自组织线性表的启发式规则。例如,如果使用线性探查冲突解决策略,那么每当确定一条记录不在其基位置时,就可以把这条记录与其探查序列中的前一条记录交换位置。另外那条记录现在就会离其基位置更远了,希望它被访问的频率也更小了。这种方法对于这一节提供的其他冲突解决策略没有作用,这是因为为了改进对一条记录的访问而交换一对记录,可能会从探查序列中清除另一条记录。

另一种方法就是保持每一条记录的访问计数,并且周期性地重新散列整个表。记录应当按照频率高低的顺序插入散列表中,保证上一次被频繁访问的记录有更好的机会接近其基位置。

## 删除

当从散列表中删除记录的时候,有两点需要着重考虑:

1. 删除记录一定不能影响后面的检索。也就是说,检索过程必须仍然能够通过新清空的槽。这样,删除过程就不能简单地把槽标记为空,因为这样会断开探查序列后面的记录。例如,在图 10.4(a)中,关键码 1001 和 9537 都散列到第 0 个槽。按照冲突解决策略,把关键码 9537 放到第 1 个槽中。如果从表中删除了 1001,对 9537 的检索仍然需要检查第 1 个槽。

2. 不希望让散列表中的位置由于记录删除而不可再用,释放的槽应该能够为后来的插入使用。

通过在被删除记录的位置上置一个特殊标记,就可以解决这两个问题,这个标记称为墓碑(tombstone)。墓碑标志一条记录曾经占用这个槽,但是现在已经不再占用了。如果沿着探查序列检索时遇到墓碑,检索过程会继续下去。插入时若遇到一个墓碑,就可以用这个槽存储新记录。然而,为了避免插入两个相同的关键码,检索过程仍然需要沿着探查序列下去,直到找到一个真正的空位置。

墓碑的使用使得检索过程仍然能够正常工作,并且能够重新使用已删除记录的槽。然而,经过一系列交替的插入、删除操作之后,一些槽会包含墓碑。这将会增加记录本身到其基位置的平均长度,超过了墓碑不存在的情况下它所应该在的位置。一般的数据库应用程序会先把一组记录装入散列表中,然后进入一个交替的插入、删除阶段。在表中装入初始记录之后,前几次删除会加长到达记录的探查序列的平均长度(它将会增加墓碑)。随着时间的推移,平均长度会达到一个相对平衡点,这是因为插入会通过填充墓碑槽减小平均长度。例如,在开始把记录装入数据库之后,平均路径长度可能是 1.2(即,平均每一次检索会需要 0.2 次访问超出基位置)。在一系列插入、删除操作之后,由于墓碑的原因,这个平均距离可能会增加到 1.6。这看起来只是很少的增长,但是这个超出基位置的长度是删除之前超出长度的 3 倍。

这个问题有两种可能的解决方案:

1. 在删除时进行一次局部重组,试图缩小平均路径长度。例如,在删除一个关键码之后,

继续深入这个关键码的探查序列,把探查序列中后面的记录交换到当前删除记录的槽中(注意不要从探查序列中清除一个关键码)。

2. 定期重新散列整个表。也就是说,把所有记录重新插入到一个新的散列表中。这样不仅能够清除墓碑,还为把最频繁访问的记录放到它们的基位置提供了机会。

## 10.5 深入学习导读

对于各种自组织技术效率的比较,参见 Bentley 和 McGeoch “Amortized Analysis of Self-Organizing Sequential Search Heuristics” [BM85]。10.2 节文本压缩的例子来自 Bentley 等人的 “A locally Adaptive Data Compression Scheme” [BSTW86]。有关 Ziv-Lempel 编码的更多内容,参见 James A. Storer 编写的《Data Compression: Methods and Theory》[Sto88]。

有关文档检索技术的更多信息,参见 Salton 和 McGill 编写的《Introduction to Modern Information Retrieval》[SM83]。

有关完美散列的介绍和一个好的算法,参见 Fox 等人的论文 “Practical Minimal Perfect Hash Functions for Large Databases” [FHCD92]。

有关各种冲突解决策略的深入分析,参见 Knuth, Volume 3 [Knu81]。

本章提供的散列方法的模型用于固定长度的散列表,这里没有提到的一个问题是当散列表已满而且必须插入更多记录时应该怎么办。这就是动态散列方法讨论的问题。R.J. Enbody 和 H.C. Du 的 “Dynamic Hashing Schemes” [ED88] 对此有很好的介绍。

## 10.6 习题

10.1 修改 3.5 节的二分法检索例程,实现词典检索。假定关键码值在 1 到 10 000 范围之间,而且关键码值在这个范围内平均分布。

10.2 修改快速排序算法,在一个有  $n$  个数的未排序的数组中找到第  $K$  个最小值 ( $K < = n$ )。算法在平均情况下应当需要  $\Theta(n)$  时间。

10.3 为方程  $T(n) = \log_2 n$  和  $T(n) = n/\log_e n$  绘图。下面两种方法哪一种性能更好? 一个是对已排序的线性表进行二分法检索,另一个是对根据访问频率排序的线性表顺序检索,频率符合 Zipf 分布。给出随着时间变化的差异。

10.4 假定值 0 到 7 存储在一个自组织线性表中,开始按照升序存放。对于 10.2 节建议的三种自组织启发式规则,按照下面的顺序访问线性表,给出结果线性表和需要的比较总数。

3 7 7 6 7 4 6 7 6 7 4 2 4 7 6

10.5 对于三种自组织线性表启发式规则中的每一个,给出一组记录访问,它使得按照这个规则需要的比较数最多。

10.6 编写一个算法,实现频率计数自组织线性表启发式规则,假定线性表使用数组实现。特别地,编写一个函数 FreqCount,它把要检索的值作为输入,并且相应地调整线性表。如果值不在线性表中,就把它添加到线性表的最后,其频率计数是 1。

10.7 编写一个算法,实现移至前端自组织线性表启发式规则,假定线性表使用数组实

- 现。特别地,编写一个函数 MoveToFront,它把要检索的值作为输入,并且相应地调整线性表。如果值不在线性表中,就把它添加到线性表的开始位置。
- 10.8 编写一个算法,实现转置自组织线性表的启发式规则,假定线性表使用数组实现。特别地,编写一个函数 transpose,它把要检索的值作为输入,并且相应调整线性表。如果值不在线性表中,就把它添加到线性表的最后。
- 10.9 编写函数,如 10.3 节所述,对一个每个位代表集合元素的任意长的位向量计算集合的并、交和差,假定对于每一个操作,两个向量都有相同的长度。
- 10.10 计算下列情况下的概率。这些概率可以通过分析计算出来,也可以编写一个计算机程序,通过模拟产生概率。
- (a)一个组里有 23 个学生,其中两个学生有相同生日的概率是多少?
  - (b)一个组里有 100 个学生,其中三个学生有相同生日的概率是多少?
  - (c)要使得一个班中有 2 个同学有相同出生月份的概率至少是 50%,那么这个班要有多少个学生呢?
- 10.11 假定把关键码  $K$  散列到有  $n$  个槽(从 0 到  $n - 1$  编号)的散列表中。对于下面的每一个函数  $h(K)$ ,这个函数作为散列函数可以接受吗?(即,对于插入和检索,散列程序能正常工作吗?)如果可以,它是一个好的散列函数吗? 函数 Random( $n$ )返回一个 0 到  $n - 1$  之间的随机整数(包含这两个数在内)。
- (a)  $h(k) = k/n$ ,其中  $k$  和  $n$  都是整数。
  - (b)  $h(k) = 1$ 。
  - (c)  $h(k) = (k + \text{Random}(n)) \bmod n$ 。
  - (d)  $h(k) = k \bmod n$ ,其中  $n$  是一个素数。
- 10.12 假定有一个 7 个槽的散列表(槽从 0 到 6 编号)。如果使用散列函数  $h(k) = k \bmod 7$  和线性探查,用于数字线性表 3,12,9,2,给出最后的结果散列表。在插入值为 2 的关键码之后,列出每一个空槽作为下一个被填充槽的概率。
- 10.13 对下列字符串,运行 ELFhash 的结果是什么?
- (a)HELLO WORLD
  - (b)NOW HEAR THIS
  - (c)HEAR THIS NOW
- 10.14 使用闭散列,利用双散列方法解决冲突,把下面的关键码插入到一个有 13 个槽的散列表中(槽从 0 到 12 编号)。使用的散列函数  $H1$  和  $H2$  在下面给出定义。给出插入 8 个关键码以后的散列表。一定要说明如何使用  $H1$  和  $H2$  进行散列。函数 Rev( $k$ )颠倒 10 进制数的各个位的数字,例如,Rev(37) = 73;Rev(7) = 7。
- $H1(k) = k \bmod 13$ 。
- $H2(k) = (\text{Rev}(k + 1) \bmod 11)$ 。
- 关键码:2,8,31,20,19,18,53,27。
- 10.15 对于使用特殊值标记墓碑来代替删除记录的散列表,编写其删除函数的算法。修改函数 hashInsert 和 hashSearch,使其对墓碑能正常工作。
- 10.16 考虑下面数 1 到 6 的一个随机排列:

2,4,6,1,3,5

现在,考虑使用这个排列对一个长度为 7 的散列表实现伪随机探查。这个排列能解决基本聚集问题吗?当实现伪随机探查时,选择一个排列的意思是什么?

## 10.7 项目设计

- 10.1 用桶式散列实现一个存储在磁盘中的数据库。定义记录为 128 个字节长,其中 4 个字节是关键码,120 个字节是数据。其余 4 个字节留作存储必要的信息来支持散列表。散列表中的一个桶是 1024 个字节长,因此每个桶可以放 8 条记录。散列表要包含 27 个桶(总共 216 条记录的空间,槽从 0 到 215 编号)。后接一个溢出桶,在文件中的记录位置是 216。散列函数是  $R.key() \% 213$ (这意味着表中最后三个槽不作为任何记录的基位置)。冲突解决函数是在桶内折回的线性探查。例如,如果一条记录散列到第 5 个槽,冲突解决过程会尝试以 5,6,7,0,1,2,3,4 的顺序把这条记录插入表中。如果一个桶满了,就把记录放到文件最后的溢出部分。系统应当支持记录的插入、删除和检索。当进行测试的时候,考虑系统用于一次存储 100 个左右的记录。
- 10.2 实现 10.2 节描述的文本压缩系统。
- 10.3 实现三个自组织线性表的启发式规则、计数规则、移至前端规则和转置规则。对各种输入数据运行这三种启发式规则,比较它们的代价。代价的度量应当是检索线性表时需要的比较总数。使用的输入数据要使自组织线性表的比较更合理,也就是说,使数据的频率分布不均匀,这对于比较各种启发式规则很重要。一个好的方法是读入文本文件。线性表应当存储文本文件中的单个单词。从空线性表开始,就像对 10.2 节的例子所做的那样。每次在文本文件中遇到一个单词不在线性表中时,就把它添加到线性表的最后,然后进行相应的重排。
- 10.4 用线性探查实现散列方法。通过经验模拟,确定随着  $\alpha$  的增长,插入和删除的代价(即重新建立图 10.5 的虚线)。



## 第 11 章 索引技术

本章介绍的文件结构用于组织存储在磁盘中的大量记录。这种文件结构支持高效率的插入、删除和检索操作。如果所有检索形式都是“找到关键码值为  $K$  的记录”的情况,散列技术可以提供极高的性能。但是,许多应用程序需要更为广泛的检索能力。一个例子就是范围查询,它的意思是检索关键码值在某个范围内的所有记录。另一种查询可能需要按照关键码值的顺序访问所有记录。散列表无法高效率地支持这两种操作。

输入顺序文件(entry-sequenced file)按照记录进入系统的顺序把记录存储在磁盘中。输入顺序文件相当于一个磁盘中未排序的线性表,因此不支持高效率检索。一个自然的解决方法就是按照检索关键码的顺序排序记录。一般的数据库,例如一家公司的一组雇员或者顾客记录,可能包含多个检索关键码。要回答某个顾客的问题,可能需要按照顾客的名字检索。公司经常希望对大量邮件按照邮政编码的顺序排序并输出记录。政府文书工作可能需要按照社会保障号的顺序进行处理。这样,就可能没有一个惟一“正确”的存储记录的顺序。

9.5 节讨论了按照关键码排序的概念。那里创建了一个索引文件,索引文件的记录由关键码值和指向数据库主文件中完整记录的指针组成。索引文件为记录指定了一个顺序,而不需要重新排列记录实体本身。一个数据库可能有多个相关的索引文件,每个索引文件都通过一个不同的关键码字段支持对记录的高效率访问。索引文件除了允许根据多个关键码访问以外,还有助于避免重组原始记录文件的高昂代价。索引文件允许直接访问某条记录,而不管记录在数据库中的顺序。

数据库中的每条记录通常都有一个惟一标识,称为主码(primary key)。例如,一组人员记录的主码可能是每个人的社会保障号或者标识号。但是,标识号一般不便检索,因为检索者不可能知道它。然而检索者可能知道所需雇员的名字。还有,检索者可能对找到工资在某个确定范围内的所有雇员感兴趣。如果这些就是数据库的一般检索请求,那么名字和工资就应当作为单独的索引。然而,工资索引中的关键码值不可能是惟一的。像工资这样的关键码,可能多条记录有相同的关键码值,称为辅码(secondary key)。大多数检索可以使用辅码来完成。辅码索引把一个辅码值与具有这个辅码值的每一条记录的主码值关联起来。此时,可以直接检索整个数据库,找到具有此主码值的记录。也可以通过一个主码索引完成,这个主码索引带有指向磁盘中实际记录的指针。在后一种情况下,只有主码索引提供磁盘中记录的实际位置。

索引技术是一种组织大型数据库的重要技术,已经开发出了多种索引方法。10.4 节讨论了通过散列方法直接访问记录。按照关键码排序的简单线性表也可以作为一个记录文件的索引。10.1 节讨论了检索主存储器中排序线性表的技术。下面几节讨论与基于磁盘文件的线性表索引技术相关的其他一些问题。然而,排序的线性表不能很好地完成插入、删除操作。

第三种索引方法是树形索引。树一般用于组织大型数据库,这些数据库必须支持记录的插入、删除和关键码范围检索。11.2 节简要描述了 ISAM。对于必须支持记录的插入、删除的大型数据库,ISAM 是解决其存储问题的尝试性步骤。它的失败之处有助于说明树形索引技术的价值。11.3 节介绍了与树形索引相关的一些基本问题。11.4 节介绍了 2-3 树,这种平衡



树结构是 11.5 节介绍的 B 树的一种简单形式。B 树是在基于磁盘的大型数据库系统中用得最广泛的索引方法,而且已经开发了许多变体。11.5 节从讨论一种变体开始,这种变体一般简单地称为“B 树”。11.5.1 小节给出最广泛实现的变体——B<sup>+</sup> 树。

## 11.1 线性索引

线性索引(linear index)的索引文件中是一组顺序的关键码/指针对,这个文件按照关键码顺序排序,指针指向磁盘中的完整记录。根据索引文件长度的不同,可以把线性索引的索引文件存储在主存储器中,也可以存储在磁盘中。线性索引有许多优点,它提供了对变长数据库记录的便捷访问方式,这是因为索引文件中的每一项都包含一个定长的关键码字段和一个定长的指向记录(变长)开始位置的指针,如图 11.1 所示。它还能够通过线性索引对数据库记录高效率地检索和随机访问,因为它适于二分法检索。

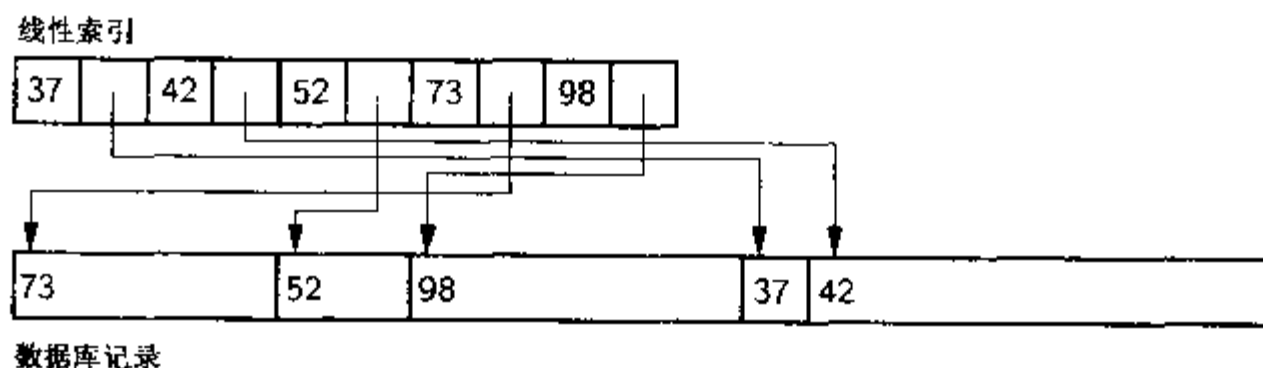


图 11.1 变长记录线性索引图示。索引文件中的每条记录都是定长的，其中包含一个指向数据库文件中相应记录开始位置的指针

线性索引也有明显的局限性。如果数据库中包含很多记录,线性索引本身可能会因太大而无法存储到主存储器中。由于检索过程中可能需要多次磁盘访问,这就使得二分法检索的代价太高。解决这个问题的一种方法是在主存储器中存储一个二级线性索引,它标识存储目标关键码的索引文件所在的磁盘块。例如,磁盘中的线性索引可能放置在多个 1024 字节的块中。如果线性索引中每个关键码/指针对需要 8 个字节,那么每个块中可以存储 128 个关键码。主存储器中的二级索引是一个表,这个表存储线性索引文件中每个块的第一个位置的关键码值。如果线性索引需要 100 个磁盘块(100KB),二级索引中就只包含 100 项。要找到包含待检索关键码值的磁盘块,首先检索这个 100 项的表,找到小于等于检索关键码值的最大值,这样就找出了索引文件中的相应块,然后把这个块读入主存储器。此时,在这个索引文件块内进行二分法检索就会得到一个指向数据库中实际记录的指针。这个过程如图 11.2 所示。一般地,二级索引存储在主存储器中。这样,通过这种方法访问数据库就需要两次磁盘读取:一次读索引文件,另一次读数据库文件,以找到实际记录。

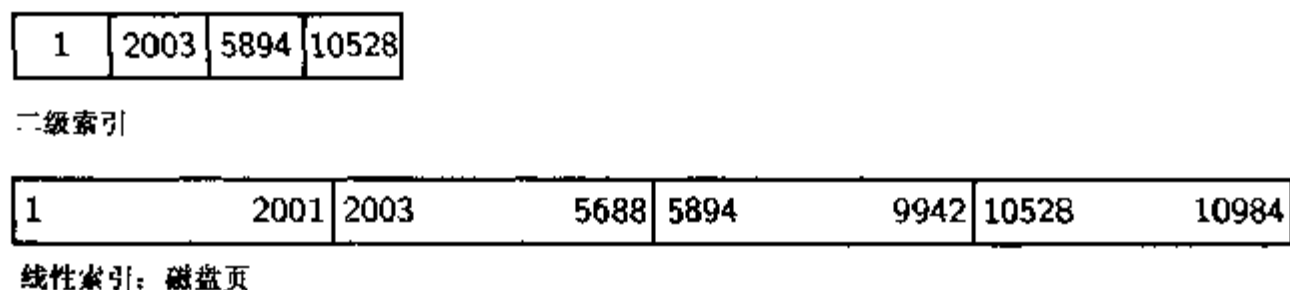


图 11.2 简单的二级线性索引。线性索引存储在磁盘中。较小的二级索引存储在主存储器中。二级索引中的每条记录存储索引文件相应磁盘块的第一个关键码值

每当向数据库中插入一条记录或者从数据库中删除一条记录时,所有相关的二级索引都必须更新。更新线性索引的代价很高,因为数组的全部内容都要移动一个位置。另一个问题是具有相同辅码值的多条记录,每一条记录都在索引中重复那个关键码值。当辅码字段的范围有限(例如,标识工作类别的字段只有很少几个可能值)时,这种重复会浪费很多的空间。

对简单排序数组的一种改进是二维数组,其中每一行对应一个辅码值。一行中包含一些主码,这些主码的记录具有标识的辅码值。图 11.3 说明了这种方法。现在没有辅码值重复了,可能产生相当大的空间节省。插入和删除的代价也减少了,因为表中只有一行需要调整。当添加一个新的辅码值时,就把一个新行添加到数组中。这可能会导致移动多条记录,但是这种情况不会频繁发生。

Jones	AA10	AB12	AB39	FF37
Smith	AX33	AX35	ZX45	
Zukowski	ZQ99			

图 11.3 二维线性索引。每一行列出与某个辅码相关的主码。在这个例子中,辅码是一个名字,主码是一个惟一的 4 字符代码

这种方法的缺陷是数组必须有一个固定的长度,从而对可能与某个辅码相关的主码数目强加一个上限。而且,对于那些相关记录的数目比数组长度小的辅码,会浪费行中剩余空间。更好的方法是有一个辅码值的一维数组,每个辅码值与一个链表相关联。如果索引存储在主存储器中,这能够工作得很好。但是当索引存储在磁盘中就不一样了,因为某个关键码的链表可能分散在多个磁盘块中。

考虑一下存储雇员记录的大型数据库。如果主码是雇员的标识号,辅码是雇员的姓名,那么名字索引表中的每一条记录都把一个名字与一个或多个标识号关联起来,标识号索引则把一个标识号与一个指向磁盘中完整记录的惟一指针关联起来。这样组织起来的辅码索引也称为倒排表(inverted list)或者倒排文件(inverted file)。它把检索过程反过来,从辅码到主码,再到实际数据记录。它也称为一个线性表,因为每个辅码值(从概念上说)有一组主码值与之关联。图 11.4 说明了这种组织方式,这里,把名字作为辅码,主码是一个 4 个字符的惟一标识符。

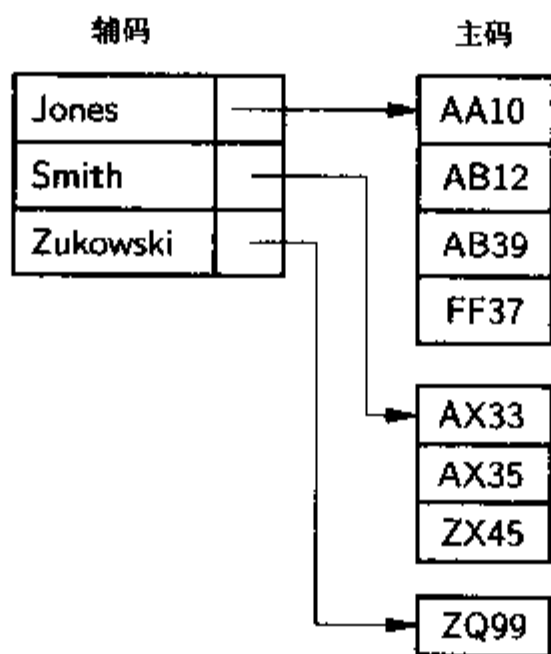


图 11.4 倒排表图示。每个辅码值存储在辅码索引表中。表中的每个辅码值都有一个指向一组主码的指针,这些主码相关的记录都具有这个辅码值

图 11.5 说明了存储倒排表的一个更好的方法。一组辅码值如前所示,与每个辅码关联的是一个指向主码数组的指针,主码数组使用链表实现。这种方法把所有辅码表的存储结合到一个数组中,可能会节省空间。主码数组中的每条记录由一个主码值和一个指向表中下一个元素的指针组成。从数组中插入、删除辅码很容易,从而使得这种方法成为实现基于磁盘的倒排文件的一种很好的方法。

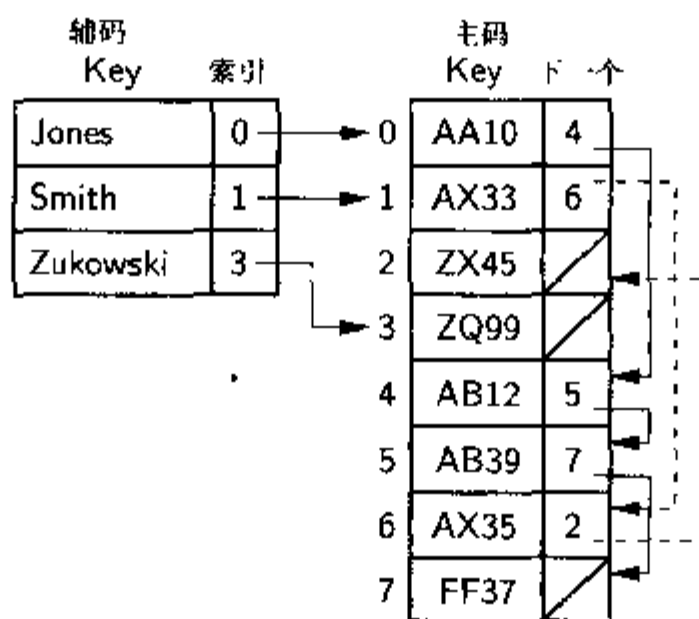


图 11.5 通过把辅码数组与主码数组结合起来实现倒排表。辅码数组中每一条记录都包含一个指向主码数组中某一条记录的指针

## 11.2 ISAM

怎样处理需要频繁更新的大型数据库呢? 线性索引的主要问题是,它是一个单一的大块,不便于更新,因为一次更新就需要改变索引中每个关键码的位置。倒排表倒是减轻了这个问题的严重性,但是倒排表只适用于辅码值较少、辅码值重复程度很高的辅码索引。如果能把线性索引分成多块,每次更新只影响一部分索引,那么线性索引作为主码索引还是很好的。这种思想会贯穿本章的其余部分,最后汇集到今天最广为使用的索引方法 B<sup>+</sup> 树中,但是首先必须从研究 ISAM 开始。这种方法是解决需要频繁更新的大型数据库的一个早期尝试。

在发明有效的树形索引方法以前,使用过各种各样的基于磁盘的索引方法。所有这些方法都很笨拙,这主要是因为没有合适的处理更新的方法。一般说来,更新会使索引降低性能,ISAM 就是这样的一个索引。在采用 B 树之前,IBM 曾经广泛使用过它。ISAM 基于线性索引的一种改进形式,如图 11.6 所示。记录按照主码值的顺序存储,磁盘文件按照磁盘中的柱面组织。在主存储器中有一个表,这个表中列出了文件内每个柱面中的最小关键码值。每个柱面中还有一个表,这个表中列出了柱面内每个块中的最小关键码值。每个柱面还有一个溢出区。当插入新记录时,就把新记录放到相应柱面的溢出区中(实际上,柱面就是作为桶使用的)。如果柱面的溢出区完全填满了,那么就使用系统级的溢出区。检索从主存储器中系统级表开始,找到相应的柱面,然后从磁盘中读入柱面的块表,查找相应块。如果在块中找到记录,那么检索就告完成,否则就检索柱面的溢出区。

在数据库最初创建之后,只要没有记录的插入、删除,访问就会很有效,只需两次磁盘读取。第一次磁盘读取从目标柱面中得到块表,第二次磁盘读取得到的相应块。在情况好的时

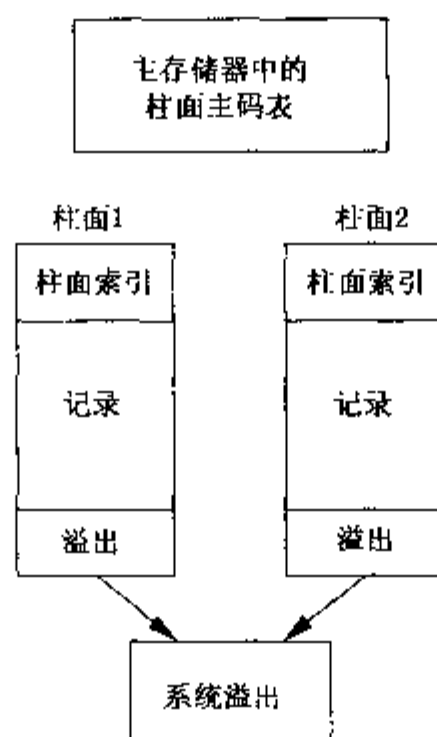


图 11.6 ISAM 索引系统说明

候,就会找到需要的记录。经过多次插入后,溢出表会变得过大。随着柱面溢出区的填满,就会导致过长的检索时间。对这个问题的“解决方法”是定期重组整个数据库,这意味着要在柱面之间重新分配记录,在每个柱面内重新排序记录,并且更新系统索引表和柱面内的块表。在 20 世纪 60 年代,这种重组对于数据库系统很普遍,而且一般在夜间完成。

### 11.3 树形索引

如果数据库是静态的,也就是说,当极少或者没有记录的插入、删除时,线性索引是很有效的。ISAM 对于有限次更新足够了,但是不适用于频繁修改。由于 ISAM 本质上是二级索引,因而对于真正的大型数据库,它也要进行分解,因为这些数据库第一级索引的柱面数太大,不能放在主存储器中。

一般的数据库应用程序具有以下特征:

1. 频繁更新大量记录。
2. 检索根据一个关键码或者多个关键码的组合进行。
3. 使用关键码范围查询。

对于这样的数据库,需要更好的数据组织方式。一种方法是使用二叉检索树(BST)存储主码索引和辅码索引。BST 可以存储重复的关键码值,可以提供有效的插入、删除以及有效的检索,而且还可以完成有效的范围查询。如果主存储器空间足够大,BST 是实现主码索引和辅码索引的一种可行的选择。

但是,BST 会变得很不平衡,即使在较好的情况下,叶结点的深度也很容易成倍变化。当树存储在主存储器中时,这可能不是一个重要问题,因为检索和更新需要的时间仍然是  $\Theta(\log n)$ 。然而,当树存储在磁盘中时,叶结点的深度就非常关键了。

每次访问一个 BST 结点 B 时,都要访问从根结点到结点 B 路径上的所有结点。必须从磁盘中得到这个路径上的每个结点,每次磁盘访问都得到一个块的信息。如果一个结点与其父结点在同一个磁盘页中,那么一旦其父结点在主存储器中,访问这个结点的代价就微忽其微了,这样,

就非常有必要把子树保存在同一个磁盘页内。但是,每次访问 BST 的结点都需要从磁盘中读入另一个页。如果 BST 访问表现出良好的引用局部性,使用一个缓冲池在主存储器中存储多个 BST 页就可以缓解磁盘访问问题,但是这并不能完全解决问题。如果 BST 不平衡,问题就更严重了,因为树中层次很深的结点可能会导致读入很多磁盘页,这样,要对基于磁盘的 BST 进行有效的检索,有两个重要问题必须提及。第一个问题是如何保持树的平衡,第二个问题是如何在磁盘页中安排结点,使得从根结点到任何叶结点的路径所经过的页最少。

需要选择一种方法,既能平衡 BST,又能很好地把 BST 结点分配到页中,使访问这个结点需要的磁盘 I/O 最少,如图 11.7 所示。然而,由于插入和删除,坚持这种方法是很困难的,特别是当进行一次更新时树仍然应当保持平衡,这样做可能需要大量重组操作。每次更新应当只影响一些磁盘页。从图 11.8 中可以看到,采用一种规则,要求 BST 是完全二叉树,会引起大量的磁盘页重组。

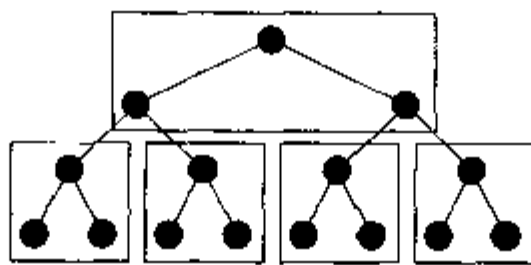


图 11.7 把 BST 分成页。BST 在磁盘页之间划分,每个磁盘页有三个结点的空间。从根结点到任何叶结点的路径都包含在两个磁盘页中

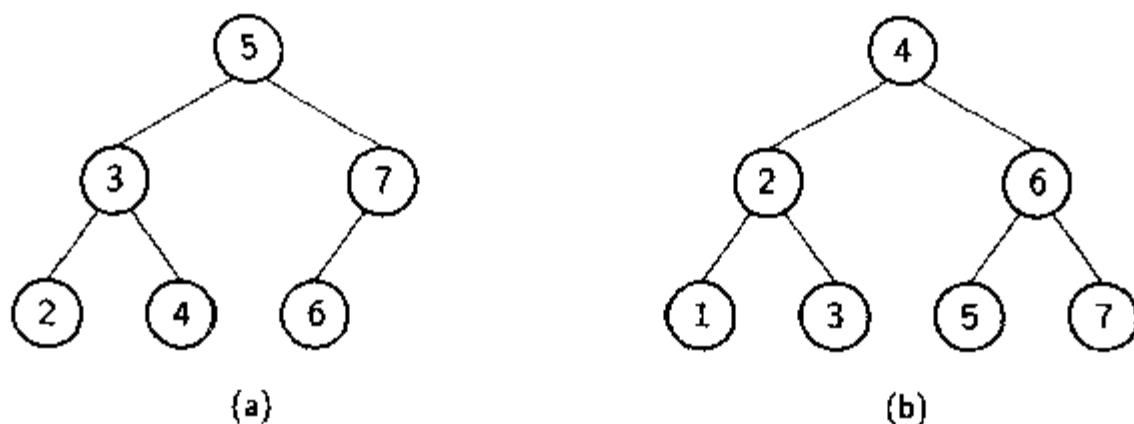


图 11.8 插入之后试图重新平衡 BST 的代价很高

(a)有 6 个结点具有完全二叉树形状的 BST。(b)把一个值为 1 的结点插入(a)的 BST 中。为了既维持完全二叉树形状,又维持 BST 的特性,需要对树进行重大重组

可以选择另外一种树形结构来解决这些问题,这种树形结构更新之后仍然能自动保持平衡,而且适于按页存储。有许多广泛使用的平衡树形数据结构,还有许多技术用于保持 BST 平衡。一个例子就是 13.2 节讨论的伸展树。作为另一种选择,11.4 节给出了 2-3 树(2-3 tree),这种树的特征是它的叶结点总是在同一层。这里优先讨论 2-3 树的主要原因是它会很自然地引出 11.5 节的 B 树,而 B 树是当前最为广泛使用的索引方法。

## 11.4 2-3 树

这一节讨论的检索树类型称为 2-3 树。2-3 树不是二叉树,但是它的形状符合下面的定义。

**定义 11.1** 一个 2-3 树的形状有下面的特征:

1. 一个结点包含一个或两个关键码。
2. 每个内部结点有 2 个子女(如果它包含一个关键码)或者 3 个子女(如果它包含两个关键码),因此得名 2-3 树。
3. 所有叶结点都在树的同一层,因此树的高度总是平衡的。

除了这些形状上的特征以外,2-3 树还有一个类似于 BST 的检索树特征。对于每个结点,左子树中所有后继结点的值都小于第一个关键码的值,而中间子树中所有结点的值都大于或等于第一个关键码的值。如果有右子树(相应地,结点存储两个关键码),那么中间子树中所有后继结点的值都小于第二个关键码的值,而右子树中所有结点的值都大于或等于第二个关键码的值。为了维持这些形状特征和检索特征,结点插入、删除时需要采取特别的操作。2-3 树有这样一个优点,它能够以相对较低的代价保持树高的平衡。

图 11.9 中是一棵 2-3 树,结点标识为有 2 个值字段的矩形框。只有两个子女结点的右边值字段为空,叶结点可能包含一个或两个关键码。图 11.10 是 2-3 树结点的 Java 类说明。

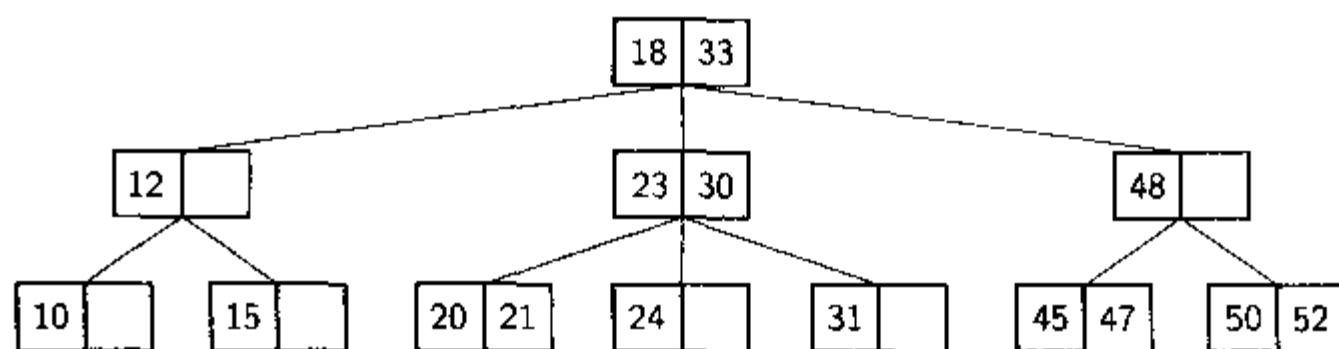


图 11.9 一棵 2-3 树

```

class TTreeNode {           // 2-3 tree node implementation
    private Elem lkey;       // The node's left key
    private Elem rkey;       // The node's right key
    private int numKeys;     // Number of key values stored
    private TTreeNode left;  // Pointer to left child
    private TTreeNode center; // Pointer to middle child
    private TTreeNode right; // Pointer to right child

    public TTreeNode() { center = left = right = null; numKeys = 0; }
    public TTreeNode(Elem l, Elem r, TTreeNode p1, TTreeNode p2, TTreeNode p3) {
        lkey = l; rkey = r;
        if (r == null) numKeys = 1; else numKeys = 2;
        left = p1; center = p2; right = p3;
    }

    // Add key and subtree to internal node
    public void addKey(Elem val, TTreeNode child) {
        Assert.notNull(numKeys==1, "Illegal addKey");
        numKeys = 2;
        if (val.key() < lkey.key()) {
            rkey = lkey; lkey = val;
            right = center; center = child;
        }
        else { rkey = val; right = child; }
    }

    public int numKeys() { return numKeys; }
    public TTreeNode lchild() { return left; }
    public TTreeNode rchild() { return right; }
}
  
```

```

public TNode cchild() { return center; }
public TNode setCenter(TNode val) { return center = val; }
public Elem lkey() { return lkey; } // Left key value
public Elem rkey() { return rkey; } // Right key value
public boolean isLeaf() { return left == null; }

public void setLkey(Elem val) { // Change left key
    Assert.notFalse(val!=null, "Can't nullify left value");
    lkey = val;
}

public void setRkey(Elem val) { // Change right key
    Assert.notFalse(numKeys!=0, "Can't set right key of empty node");
    if ((val == null)&&(numKeys == 2)) // Clear subtree
        {right = null; numKeys = 1;}
    rkey = val;
}
} // class TNode

```

图 11.10 2-3 树结点的 Java 实现

这个类说明例子不区分叶结点和内部结点,因此在空间上是低效率的,因为每个叶结点都存储三个指针。这里可以应用 5.3.1 小节的技术实现内部结点类型和叶结点类型分开。

从 2-3 树定义的规则可以推导出树的结点数和树的深度之间的关系。一棵高度为  $k$  的 2-3 树至少有  $2^{k-1}$  个叶结点,因为如果每个内部结点有 2 个子女,它就会形成完全二叉树的形状。一个高度为  $k$  的 2-3 树至多有  $3^{k-1}$  个叶结点,因为每个内部结点最多可以有 3 个子女。

在 2-3 树中检索类似于在 BST 中检索。检索从根结点开始。如果根结点不包含检索关键码  $K$ ,那么就在可能包含关键码  $K$  的子树中继续进行检索。存储在根结点中的值能够确定哪个子树需要检索,例如,如果在图 11.9 的树中从根结点开始检索值 30。由于 30 在 18 和 33 之间,故它只可能在中间子树中,检索根结点中间的子女就会得到这个关键码。如果检索值 15,那么第一步还是检索根结点。由于 15 小于 18,进入第一个(左边的)分支,在下一层,进入第二个分支,到达包含值 15 的叶结点。如果检索的关键码是 16,那么当碰到包含值 15 的叶结点时,就会发现检索的关键码不在树中。下面是一个 2-3 树检索的 Java 实现,它在功能上相当于 BST 的 findhelp 函数:

```

private Elem findhelp(TNode root, int val) {
    if (root == null) return null; //val not found
    if (val == root.lkey().key()) return root.lkey(); //val found
    if ((root.numKeys().key() == 2) && (val == root.rkey().key()))
        return root.rkey();
    if (val < root.lkey().key()) //Search left subtree
        return findhelp(root.lchild(), val);
    else if (root.numKeys() == 1) //Search center subtree
        return findhelp(root.cchild(), val);
    else if (val < root.rkey().key()) //Search center subtree
        return findhelp(root.cchild(), val);
    else return findhelp(root.rchild(), val); //Search right subtree
}

```

向一个 2-3 树插入记录类似于向一个 BST 插入记录,新记录也放到相应的叶结点中。与

BST 插入记录不同,2-3 树并不创建新的子女结点放置插入的记录。如果关键码在树中,第一步是找到将会包含这个关键码的叶结点。如果这个叶结点只包含一个值,那么不需要对树进一步修改,就可以把新关键码添加到这个结点中,如图 11.11 所示。在这个例子中插入了值 14,从根结点开始检索,到达存储值 15 的叶结点。把 14 作为左边的值添加进来(把带有关键码值 15 的记录放到右边的位置)。

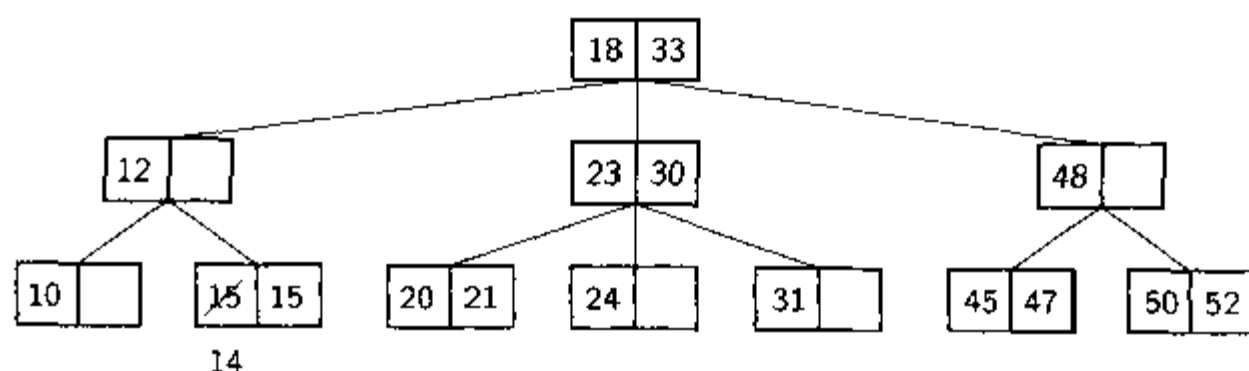


图 11.11 简单地向图 11.9 的 2-3 树中插入记录。值 14 插入树中包含 15 的叶结点。由于结点中有地方放第二个关键码,只要简单地把它添加到左边的位置,而把 15 移到右边的位置即可

如果新关键码要插入叶结点 L 中,而 L 已经包含了两个关键码,那么就需要创建更多的空间。考虑一下结点 L 中的两个关键码值和要插入的关键码值,不用关心哪两个关键码值已经在 L 中,哪一个关键码值是要插入的关键码值。第一步把 L 分裂成两个结点,这样,就必须从空闲存储区中创建一个新结点——称它为 L'。L 得到这三个关键码值中最小的一个,L' 得到其中最大的一个。中间的关键码值与指向 L' 的指针传回给父结点。这称为一次提升(promotion)。随后把提升的关键码值插入父结点中。如果父结点当前只包含一个关键码值(这样只有两个子女),那么只要简单地把提升的关键码值和指向 L' 的指针添加到父结点中。如果父结点已经满了,那么就重复进行分裂——提升过程。图 11.12 说明了一次简单的提升。图 11.13 说明了当向树中添加新的一层时,提升需要根结点分裂时会发生什么情况。在每一种情况下,所有的叶结点仍然具有同样的深度。图 11.14 给出了插入过程的 Java 实现。

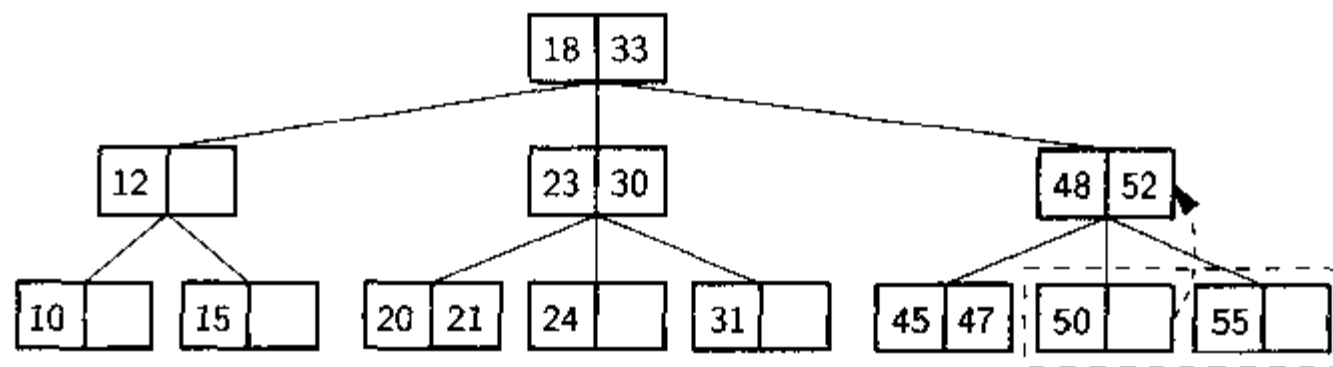


图 11.12 2-3 树的一个简单结点分裂插入过程。把值 55 添加到图 11.9 的 2-3 树中,这使得包含值 50 和 52 的结点分裂,把值 52 提升到父结点中

图 11.14 的 `inserthelp` 有两个参数。第一个参数是指向当前子树根结点的指针,第二个参数是要插入的值。返回值是一个对象数组,用于在插入期间发生分裂时返回信息。如果真的发生了这种情况,数组中的第一个对象是被提升的元素,数组中的第二个对象是新添加的子树。这些值构成了分裂过程中返回的信息(使用图 11.14 中的帮助函数 `splitnode` 实现)。



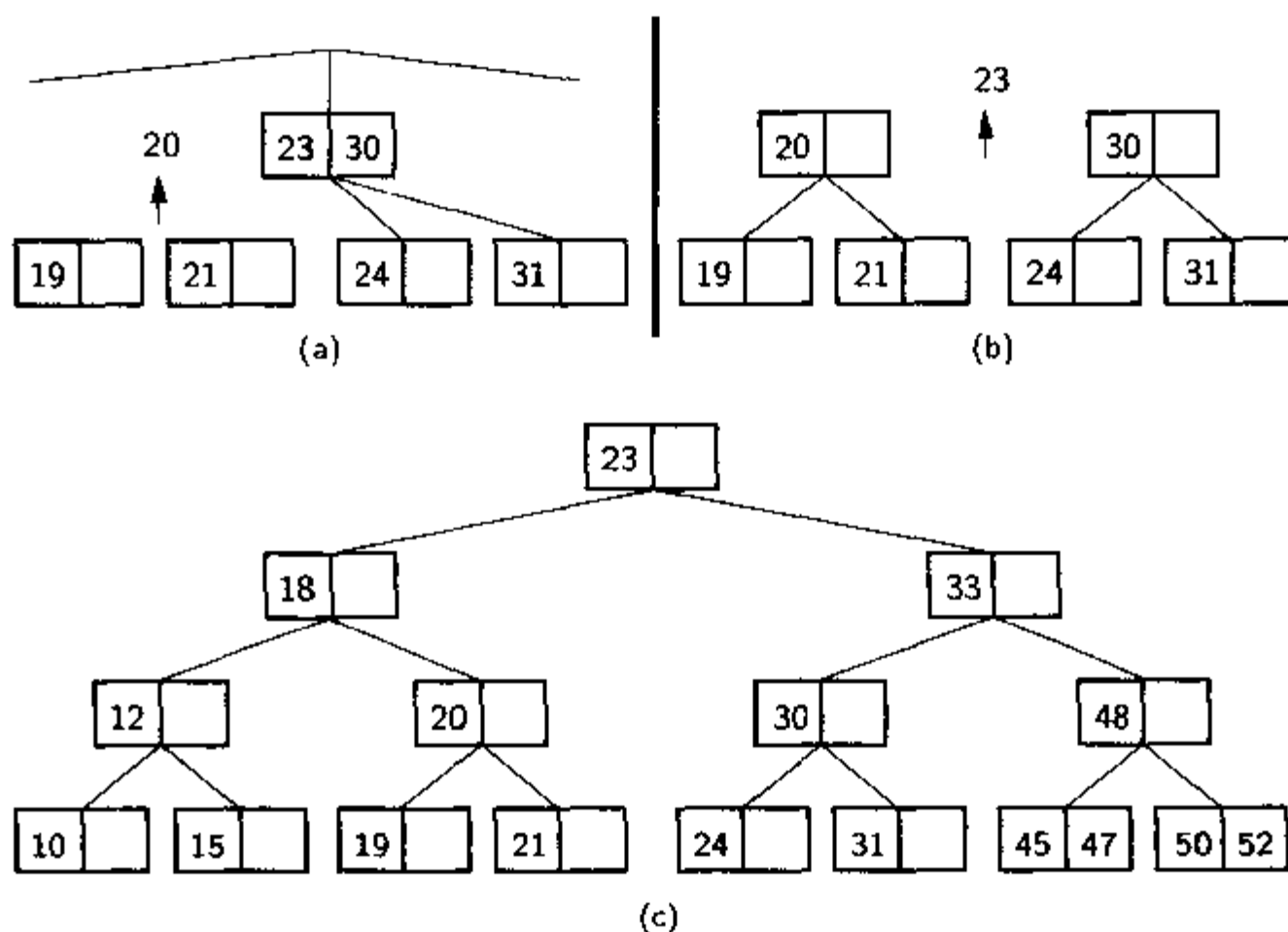


图 11.13 插入关键码导致 2-3 树的根结点分裂的例子

(a)把值 19 添加到图 11.9 的 2-3 树中,这引起了包含值 20 和 21 的结点分裂,提升值 20。(b)这又依次引起了包含值 23 和 30 的内部结点分裂,提升值 23。(c)最后,根结点分裂,被提升的值 23 成为新的根结点的左边关键码,结果使树高了一层

```
private Object[] inserthelp(TTNode rt, Elem val) { // Do insert
    if (rt == null) { // Empty tree.
        rt = new TTNode(val, null, null, null, null);
        Object[] temp = {rt}; return temp;
    }
    if (rt.isLeaf()) // At leaf node -- insert here
        if (rt.numKeys() == 1) // Easy case
            { rt.addKey(val, null); return null; }
        else return splitnode(rt, val, null);
    // Now at Internal node
    Object[] retval; // Hold result from insert
    if (val.key() < rt.lkey().key())
        retval = inserthelp(rt.lchild(), val);
    else if ((rt.numKeys() == 1) || (val.key() < rt.rkey().key()))
        retval = inserthelp(rt.cchild(), val);
    else retval = inserthelp(rt.rchild(), val);
    if (retval == null) return null; // No split of child node
    // If here, child node split
    if (rt.numKeys() == 1) { // Just add to rt
        rt.addKey((Elem)retval[0], (TTNode)retval[1]);
        return null; // No further splitting
    }
}
```

```

{
    // Split this node as well
    return splitnode(rt, (Elem)retval[0], (TNode)retval[1]);
}

// Split a node into two; return info on promoted value
Object[] splitnode(TNode rt, Elem val, TNode child) {
    Object[] temp = new Object[2]; // Store info to be returned
    if (val.key() > rt.rkey().key()) { // Val is rightmost
        temp[0] = rt.rkey(); // Promote old right key
        TNode hold = rt.rchild();
        rt.setRkey(null); // Clear it
        temp[1] = new TNode(val, null, hold, child, null);
    }
    else if (val.key() > rt.lkey().key()) { // Middle
        temp[0] = val; // Promote val
        temp[1] = new TNode(rt.rkey(), null, child, rt.rchild(), null);
        rt.setRkey(null); // Clear it
    }
    else { // Val is leftmost
        temp[0] = rt.lkey(); // Promote old left key
        temp[1] = new TNode(rt.rkey(), null, rt.cchild(),
                           rt.rchild(), null);
        rt.setCenter(child); rt.setLkey(val);
        rt.setRkey(null); // Clear it
    }
    return temp; // Return promoted element and its subtree
}

```

图 11.14 2-3 树插入例程

从 2-3 树中删除一个关键码时,有三种情况需要考虑。从包含两条记录的叶结点中清除一个关键码是最简单的情况,只是简单地清除这个关键码,没有影响到其他结点。第二种情况是叶结点只有惟一个关键码,把这个关键码清除。第三种情况是从一个内部结点中清除一个关键码。在第二种情况和第三种情况下,用另一个关键码代替被删除的关键码,占用它的位置,同时维护树的正确次序,这类似于从 BST 中清除一个结点。如果 2-3 树过于分散,那么就可能没有这样一个关键码,使得所有结点仍然保持至少一个关键码。在这种情况下,需要把兄弟结点合并到一起。2-3 树的删除操作特别复杂,这里不进一步讨论了。下一节将给出删除操作的完整讨论,在那里将把它推广到 B 树的一个变体。

2-3 树的插入例程和删除例程不会在树的底部添加新结点,然而,它们会引起叶结点的分裂或合并,而且可能会引起连锁效应,一直波及到树的根结点。如果需要,树的根结点也可能会分裂,从而创建一个新的根结点,并且使树更深一层。对于删除操作,根结点的最后两个子

女可能会合并,清除根结点,并且使树减少一层。在每一种情况下,所有叶结点都在同一层。当所有叶结点都在同一层时,就说一棵树是树高平衡的(height balanced)。由于 2-3 树是树高平衡的,而且每一个内部结点有至少 2 个子女,从而知道树的最大深度是  $\log n$ ,这样,所有 2-3 树的插入、查找和删除操作都需要  $\Theta(\log n)$  时间。

## 11.5 B 树

这一节介绍 B 树。B 树的研究通常归功于 R. Bayer 和 E. McCreight,他们在 1972 年的论文中描述了 B 树。到 1979 年,B 树几乎已经代替了除散列方法以外的所有大型文件访问方法。对于需要完成插入、删除和关键码范围检索等操作的应用程序,B 树或者 B 树的一些变体,是标准的文件组织方法。B 树涉及了当实现基于磁盘的检索树时遇到的所有问题:

1. B 树总是树高平衡的,所有叶结点都在同一层。
2. 更新和检索操作只影响一些磁盘页,因此性能很好。
3. B 树把相关的记录放在同一个磁盘块中,从而利用了访问局部性原理。
4. B 树保证树中至少有一定比例的结点是满的,这样能够改进空间利用率,同时在检索和更新操作期间减少磁盘读取数。

一个阶为  $m$  的 B 树是每个内部结点都包含多达  $m$  个分支的树。内部结点有  $m$  个子女,存储  $m-1$  个关键码值。

**定义 11.2** 一个  $m$  阶 B 树有以下特性:

- 根要么是一个叶结点,要么至少有两个子女。
- 除了根结点和叶结点以外,每个结点有  $\lceil m/2 \rceil$  到  $m$  个子女。
- 所有叶结点在树的同一层,因此树总是树高平衡的。

其实,B 树只是 2-3 树的一种推广。从另一个方面来说,2-3 树是一个 3 阶 B 树。通常,要使 B 树中一个结点的大小能够填满一个磁盘块。一个 B 树结点的实现一般允许 100 个甚至更多的子女,这样,一个 B 树结点的大小就相当于一个磁盘块的大小了,存储在树中的“指针”值实际上是包含子女结点的块号。在一般的应用程序中,使用缓冲池和页替换方法,如 LRU (见 9.3 节),来管理 B 树的块 I/O。

图 11.15 是一个 4 阶 B 树,每个结点最多有 3 个关键码,内部结点最多有 4 个子女。

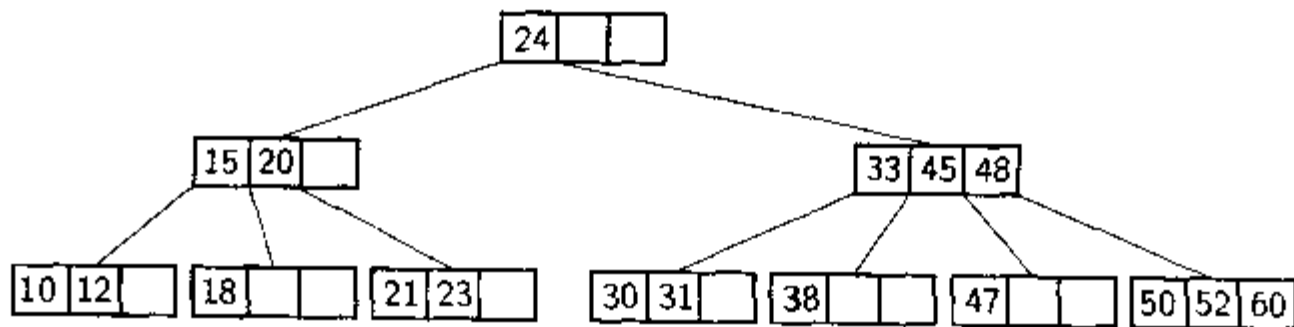


图 11.15 一个 4 阶 B 树

B 树检索是 2-3 树检索的一种推广,它是一个交替的两步过程,从 B 树的根结点开始。

1. 在当前结点中对关键码进行二分法检索。如果找到检索关键码,那么就返回这条记录。如果当前结点是叶结点,而且没有找到关键码,那么就报告检索失败。

2. 否则,沿着正确的分支重复这一过程。

例如,考虑检索图 11.15 的树中带有关键码值 47 的记录。先检查根结点,然后进入右边的分支。检查完第一层结点后,进入第三个分支来到下一层,从而到达包含关键码值 47 的叶结点。

B 树插入是 2-3 树插入的推广。第一步是找到在空间允许的情况下应当包含要插入的关键码的叶结点。如果这个结点中有地方,那么就插入关键码。如果没有,就把这个结点分裂成两个,并且把中间的关键码提升到父结点。如果父结点也已经满了,就再把它分裂,并且再次提升中间的关键码。

插入过程保证了保持所有结点至少半满。例如,当一个 4 阶 B 树的内部结点已满时,将会有 5 个子女。这个结点分裂成两个结点,每个结点包含两个关键码,这样就继续保持 B 树的特性了。

### 11.5.1 B<sup>+</sup> 树

前面一节提到 B 树广泛用于实现基于磁盘的大型系统。实际上,前面一节介绍的 B 树几乎从来都没有实现过,11.4 节介绍的 2-3 树也是一样。最普遍使用的是 B 树的一个变体,称为 B<sup>+</sup> 树。当需要更高的效率时,就需要使用一个更为复杂的变体,称为 B\* 树。

B<sup>+</sup> 树和 BST 以及 11.4 节介绍的 2-3 树最显著的差异是 B<sup>+</sup> 树只在叶结点存储记录。内部结点存储关键码值,但是这些关键码值只是占据位置,用于引导检索,这意味着内部结点在结构上与叶结点有着显著的差异。内部结点存储关键码来引导检索,把每个关键码与一个指向子女结点的指针关联起来,叶结点存储实际记录。在 B<sup>+</sup> 树纯粹作为索引的情况下,叶结点则存储关键码和指向实际记录的指针,实际记录存储在单独的磁盘文件中。根据记录大小与关键码大小的比例,  $m$  阶 B<sup>+</sup> 树中的叶结点可能存储多于或少于  $m$  条记录。要求很简单,叶结点存储足够的记录,达到至少半满。B<sup>+</sup> 树的叶结点一般链接起来,形成一个双链表。这样,通过访问链表中的所有叶结点,就可以按照排序的次序遍历全部记录。下面是 B<sup>+</sup> 树结点类的 Java 伪代码表示:

```
class Pair {           // Each BNode stores an array of Pair
    Object pointer; // BNode leaves point to Elem,
                    // BNode internal nodes point to BNode
    int key;          // The key value for this pair
} // class Pair

class BNode {          // The BNode class
    boolean isLeaf;     // True if this is a leaf
    Pair recarray[MAXREC]; // Array of key/pointer pairs
    int numrec;         // Number of pairs currently in node
    BNode leftptr, rightptr; // Each level forms a doubly-linked list
} // class BNode
```

结点的一个重要实现细节是当图 11.15 显示内部结点包含 3 个关键码和 4 个指针时,类 BNode 存储关键码/指针对。图 11.15 中的 B<sup>+</sup> 树是它的传统形式。为了简化实现,结点实

际上把一个关键码和一个指针关联起来。假定认为每个内部结点最左边的位置还有一个关键码,这个关键码值小于等于结点最左边的子树中任何可能的关键码值。一般的 B<sup>+</sup> 树实现都存储一个额外的虚记录,这条记录的关键码值小于任何可能存在的键码值。

B<sup>+</sup> 树特别适合范围查询。一旦找到了范围内的第一条记录,就顺序处理结点中的其余记录,然后继续下去。尽可能深入叶结点链表,就可以找到范围内的其余记录。

B<sup>+</sup> 树的检索必须一直到达相应的叶结点,除此以外,它与正规 B 树的检索完全一样。即使在内部结点找到了检索关键码值,这个值只是占据一个位置,并不提供对实际记录的访问。要在图 11.16 的 B<sup>+</sup> 树中找到值 33,就要从根结点开始检索。存储在根结点中的值 33 只是作为占位符,标识大于或等于 33 的关键码值可以在第二个子树中找到。从根结点的第二个子女进入第一个分支,到达包含带有关键码值 33 的实际记录(或者一个指向实际记录的指针)的叶结点。下面是一个 B<sup>+</sup> 树检索算法的伪代码段:

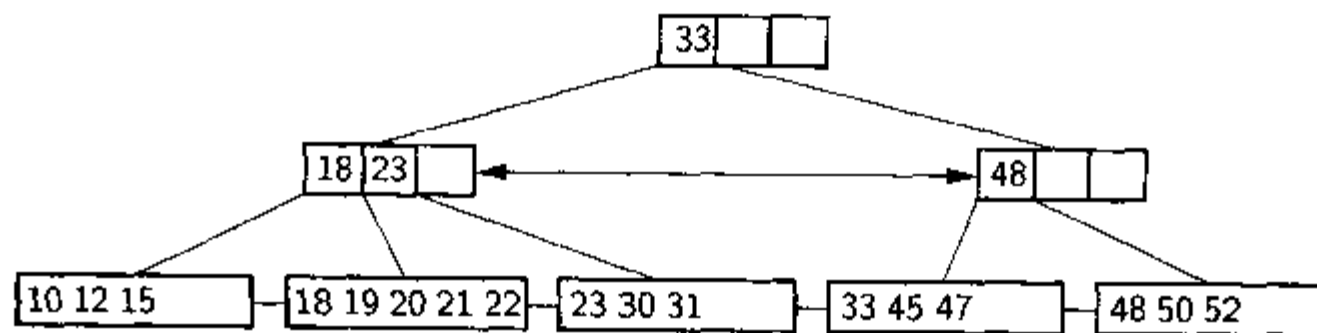


图 11.16 4 阶 B<sup>+</sup> 树的例子。内部结点必须存储 2 到 4 个子女。对于这个例子,记录的大小假定叶结点可以存储 3 到 5 条记录

```

Elem findhelp(BPNode root, int K) {
    // function binaryle(A, n, K) returns the greatest value
    // less than or equal to K in array A containing n records.
    int currec = binaryle(root.reccarray, root.numrec, K);
    if (root.isLeaf()) // If its a leaf node...
        if (root.reccarray[currec].key() == K) // Children are Elms
            return (Elem)root.reccarray[currec].pointer;
        else return null; // K not in this node
    else return findhelp((BPNode)root.reccarray[currec].pointer, K); // Process child
}
  
```

把一条记录插入 B<sup>+</sup> 树类似于把一条记录插入 B 树。首先,找到应当包含记录的叶结点。如果叶结点还没有满,那么就添加新的记录,而且没有影响到 B<sup>+</sup> 树的其他结点。如果叶结点已经满了,就把它分裂成两个(在两个结点之间平均分配记录),然后把新形成的右边结点的最小关键码值的一份拷贝提升到父结点。就像在 2-3 树中一样,提升可能会依次引起父结点分裂,最后可能会引起根结点分裂,从而引起 B<sup>+</sup> 树增加一层。B<sup>+</sup> 树的插入能够保持所有叶结点处于同样的深度。图 11.17 通过多个例子说明了插入过程。下面是 B<sup>+</sup> 树插入算法的类 Java 伪代码段:

```

// B+-tree pseudocode: insert
// putinarray(A, pos, pair) places pair in array A at position pos.
// Function splitnode(rt, pos, pair) places pair in rt at pos.
// But, in the process, rt is split into two nodes, each node
  
```

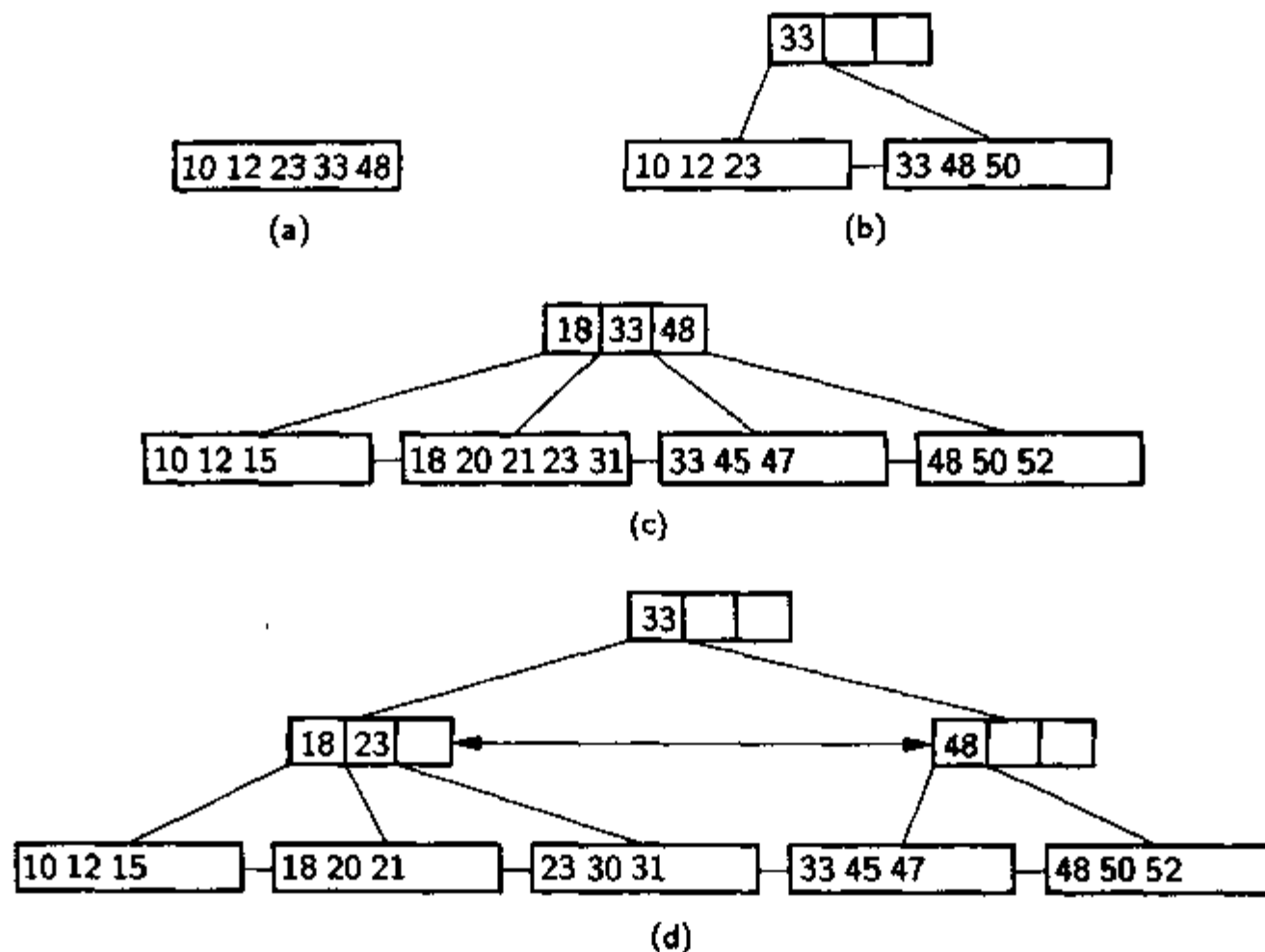
```

// taking half of the records, and the new node is returned.
Pair inserthelp(BPNode root, Elem rec) {
    Pair temp;          // Hold a key/pointer pair
    int currec = binaryle(root.reccarray, root.numrec, rec.key());
    if (root.isLeaf()) { // Leaf node - - set up values to insert
        Assert.notFalse(root.reccarray[currec].key() != rec.key(),
            "Duplicates are not allowed");
        temp = new Pair(rec.key(), rec);
    }
    else { // internal node
        temp = inserthelp((BPNode)root.reccarray[currec].pointer, rec);
        if (temp == null) return null; // Child did not split, so no
        // insert into current node

    }

    // Now, do the insert to the current node. Split if necessary
    if (! root.isFull()) putinarray(root.reccarray, currec, temp);
    else {
        BPNode tp = splitnode(root, currec, temp);
        return new Pair(tp.reccarray[0].key(), tp);
    }
    return null;
}

```

图 11.17 B<sup>+</sup>树插入的例子

(a) 一个包含 5 条记录的 B<sup>+</sup> 树。(b) 把一个关键码值为 50 的记录插入(a)中的树之后的 B<sup>+</sup> 树。叶结点分裂, 从而创建了第一个内部结点。(c) (b) 中的 B<sup>+</sup> 树经过了更多的插入。(d) 把一个关键码值为 30 的记录插入(c)中树的结果。第二个叶结点分裂, 从而依次引起内部结点分裂, 创建新的根结点

要从 B<sup>+</sup> 树中删除一条记录,首先找到包含要删除记录的叶结点。如果这个叶结点超过半满,那么只需清除记录,剩下的叶结点仍然至少半满,图 11.18 对此进行了说明。

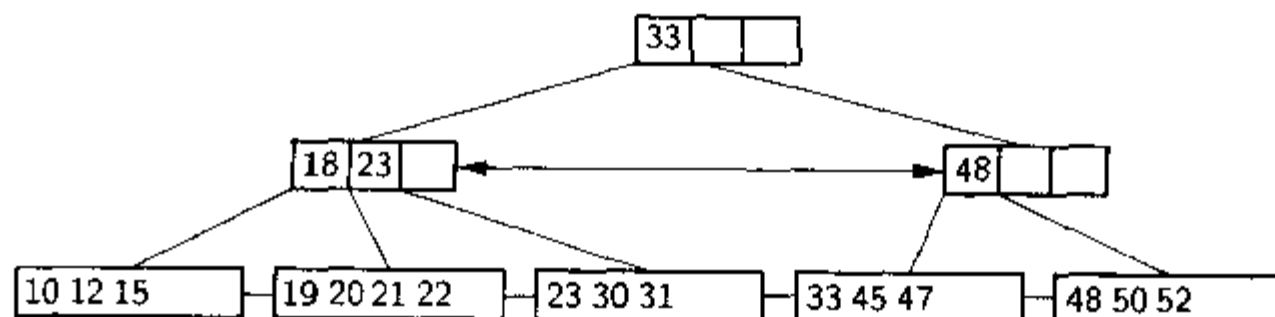


图 11.18 从 B<sup>+</sup> 树中进行简单的删除。从图 11.16 的树中清除带有关键码值 18 的记录。由于 18 只是用于在父结点中引导检索的占位符,即使树中没有关键码为 18 的记录,它也不需要改变

如果删除一条记录使结点中的记录数减少到小于最低限度(称为下溢, underflow),那么必须采取一些措施使结点足够满。第一选择是查看相邻的兄弟结点,确定是否有更多的记录,以填补这个空缺。如果兄弟结点的记录很多,那么就从兄弟结点转移过来足够的记录,使两个结点有同样多的记录,这样做是为了尽可能延迟由于删除而引起结点再一次下溢。这个过程可能需要父结点修改其占位关键码值,以反映每个结点真正的第一个关键码值。图 11.19 说明了这个过程。

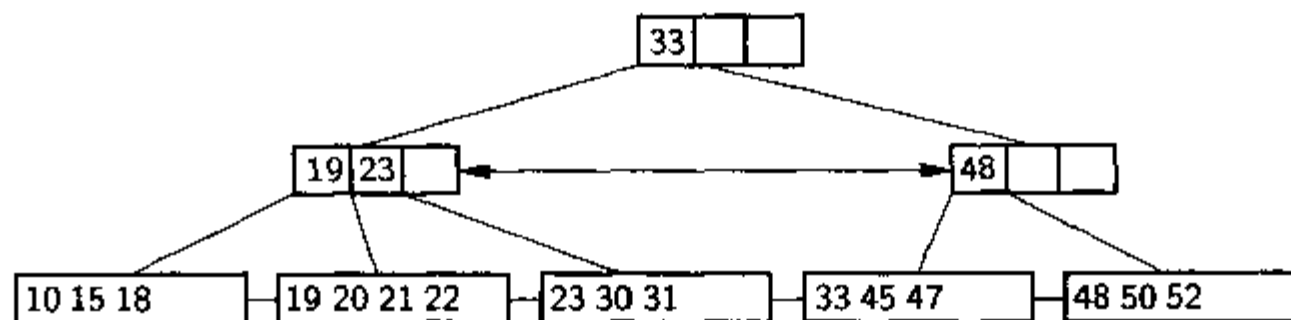


图 11.19 通过从兄弟结点借用记录的方式对图 11.16 中的 B<sup>+</sup> 树进行删除。从最左边的叶结点中删除值为 12 的关键码,使带有关键码值 18 的记录移到最左边的叶结点,占据它的位置。要正确标识子树中的关键码范围,必须更新父结点。在这个例子中,父结点最左边关键码值改为 19

如果没有一个兄弟结点可以把记录借给这条记录过少的结点(称它为 N),那么结点 N 必须把它的关键码让给一个兄弟结点,并且从树中删除结点 N。兄弟结点当然有空间这样做,因为兄弟结点最多半满(想一想它没有记录借给结点 N),而由于下溢,结点 N 的记录不足一半。这个合并过程把父结点的两个子树合成一个,可能会依次使父结点下溢。如果根结点的最后两个子女合并到一起,那么树就会减少一层。图 11.20 说明了结点合并删除过程。

图 11.21 是 B<sup>+</sup> 树删除算法的类 Java 伪代码段。

B<sup>+</sup> 树要求所有结点至少半满(除了根结点以外)。这样,存储利用率必须至少达到 50%。这对于许多实现已经可以满足要求了,但是如果结点更满,那么需要的空间就会更少(因为磁盘文件中未用的空间更少),而且处理的效率也会更高(因为每个块中的信息量更大,平均来说,需要读入主存储器的块也就更少)。由于 B<sup>+</sup> 树已经广泛使用,许多人试图改进其性能。一种实现方法是使用 B<sup>+</sup> 树的一个变体,称为 B\* 树。除了分裂、合并结点的规则不同以外,B\* 树

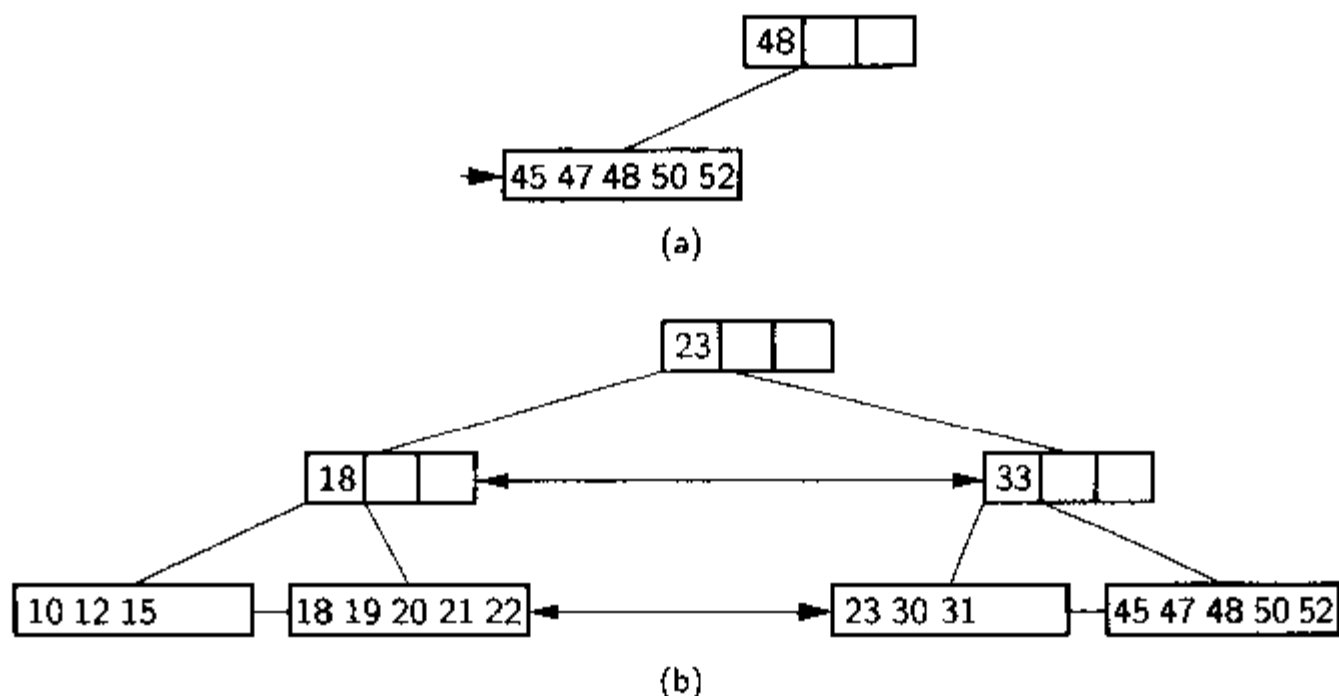


图 11.20 通过合并兄弟结点,从图 11.16 中的 B<sup>+</sup> 树删除关键码值为 33 的记录

(a)最左边的两个叶结点合并到一起,形成一个叶结点。可是,父结点现在只有一个子女。(b)由于左边的子树有一个多余的叶结点,就把这个叶结点传给右边的子树。根结点的占位关键码值和右边的内部结点的占位关键码值被更新,以反映这种改变。值 23 移到根结点,老的根结点值 33 移到最右边的内部结点

与 B<sup>+</sup> 树完全相同。B<sup>+</sup> 树在结点上溢时不把它分成两半,而是把一些记录给它相邻的兄弟结点。如果兄弟结点也满了,那么就把这两个结点分成三个。同样,当一个结点下溢时,它就与其两个兄弟结点结合,这三个结点减少为两个结点。这样,结点总是至少有三分之二的记录<sup>①</sup>。

```
// removehelp returns position of the child node adjusted, if any.
// If the child node did not underflow or change its first record,
// then return -1, to indicate the parent shouldn't be modified.
// Function removerec(A, n, c) removes the record at position c from
// array A containing n records.
// Function merge_nodes(N1, N2) merges together the record arrays of
// BPNodes N1 and N2.
// Function shuffle_nodes(N1, N2) copies records as necessary within
// BPNodes N1 and N2, so that both have equal number of records.
int removehelp(BPNode root, int K, int thispos) {
    int currec = binaryle(root.reccarray, root.numrec, K);
    if (root.isLeaf())
        if (root.reccarray[currec].key() != K) return -1; // K not found
    else { // Delete from child
        currec = removehelp((BPNode)root.reccarray[currec].pointer, K, currec);
        if (currec == -1) // Child did not underflow
            return -1;
    }
}
```

<sup>①</sup> 如果需要更高的空间利用率,这种思想还可以进一步扩展。然而,更新例程就会变得更加复杂。有一次我参与开发一个项目,实现了一个 3 ~ 4 结点的分裂与合并例程。这比 B<sup>+</sup> 树的 2 ~ 3 结点的分裂与合并例程有更好的性能。然而,分裂与合并例程如此复杂,以致于其开发者在完成之后都无法理解。





万条记录(100 个二级结点,每个结点有 100 个满的子女)。一个四层 B<sup>+</sup>树至少有 250 000 条记录,最多可以有 1 亿条记录。这样,产生一个超过四层的 B<sup>+</sup>树需要非常大的数据库。

可以使用下面的方法进一步减少 B 树需要的磁盘读取数。首先,树的上面几层可以一直存储在主存储器中。因为树分叉很快,上面两层需要的空间较少。如果 B 树只有四层深,那么到达任何给定记录的指针最多需要两次磁盘读取(第二层和第三层的叶结点)。

如前所述,应当使用缓冲池管理 B 树中的结点。一般说来,在同一时刻树中可以有多个结点在主存储器中。最直接的方法是使用一种标准方法进行结点替换,例如 LRU。然而,有时可能需要“锁定”某个结点,例如根结点。一般说来,如果缓冲池的大小适当(即至少是树的深度的两倍),那么就不需要特别的页替换技术了,因为上层结点自然会被频繁访问。

## 11.6 深入学习导读

对于本章所涉及问题的更广泛讨论,参见一般的文件处理教材。例如,Folk 和 Zoellick 编写的《File Structures : A Conceptual Toolkit》[FZ92]。特别地,Folk 和 Zoellick 对主索引和辅索引的关系给出了很好的讨论。关于 B 树的各种实现最全面的讨论是 Comer 的综述性论文 [Com79]。有关 B 树实现的深入细节也可以参见 [Sal88]。对于 B<sup>+</sup>树一类数据结构缓冲池管理策略的讨论参见 Shaffer 和 Brown [SB93]。

## 11.7 习题

- 11.1 假定一个计算机系统有 1024 个字节的磁盘块。要存储的每一条记录中 4 个字节是关键码,4 个字节是数据字段,而且,可以使用 256KB 的主存储器存储线性索引。记录已经排序,顺序地存放到磁盘文件中,每个块中的第一条记录用于线性索引。
- (a) 如果在主存储器中维护一个对文件的索引,文件中可以有多少条记录?
- (b) 如果使用如图 11.2 所示的 1024 个字节(即 128 个关键码值)的二级索引,可以存储多少条记录?
- 11.2 假定一个计算机系统有 4096 字节的磁盘块,要存储的每一条记录中 4 个字节是关键码,64 个字节是数据字段。通过线性索引访问磁盘文件中的记录。记录已经排序,顺序存放到磁盘文件中。每个块中的第一条记录用于线性索引。
- (a) 如果索引的最大空间是 2MB,线性索引可以维护多大的磁盘文件?
- (b) 如果使用如图 11.2 所示的 4096 字节(即 1024 个关键码值)的二级索引,磁盘文件中可以存储多少条记录?
- 11.3 怎样修改 3.5 节的 binary 函数,才能使它支持具有定长关键码的变长记录,关键码按照图 11.1 所示的简单线性索引进行索引。
- 11.4 假定一个数据库存储的记录由 2 个字节的整数关键码和一个变长的数据字段组成,数据字段是一个字符串。给出下面一组记录的线性索引(如图 11.1 所示):

397	Hello world!
82	XYZ

1038 This string is rather long  
 1037 This is shorter  
 42 ABC  
 2222 Hello new world!

- 11.5 下面的每一条记录都是由一个 4 位数的主码(没有重复)和一个 4 个字符的辅码(有许多重复)组成。

3456 DEER  
 2398 DEER  
 2926 DUCK  
 9737 DEER  
 7739 GOAT  
 9279 DUCK  
 1111 FROG  
 8133 DEER  
 7183 DUCK  
 7186 FROG

- (a)给出这组记录的倒排表(如图 11.4 所示)。  
 (b)给出这组记录改进的倒排表(如图 11.5 所示)。
- 11.6 在什么情况下 ISAM 的实现比 B<sup>+</sup>树的实现更有效?
- 11.7 证明  $k$  层 2-3 树的叶结点数目在  $2^{k-1}$  到  $3^{k-1}$  之间。
- 11.8 给出把值 55 和 46 插入图 11.9 的 2-3 树的结果。
- 11.9 给定一组记录,其关键码值是字母。记录按照下面的顺序插入:C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J。给出插入这些记录后的 2-3 树。
- 11.10 给定一组记录,其关键码值是字母。记录按照下面的顺序插入:C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J。当把 2-3 树改进为 2-3<sup>+</sup>树时,即内部结点只作为占位符,给出插入这些记录后的 2-3<sup>+</sup>树。假定叶结点可放多达 2 条记录。
- 11.11 给出把值 55 插入图 11.15 中 B 树的结果。
- 11.12 给出把值 1, 2, 3, 4, 5 和 6(以此顺序)插入图 11.16 中 B<sup>+</sup>树的结果。
- 11.13 给出把值 18, 19 和 20 从图 11.20b 的 B<sup>+</sup>树中删除的结果。
- 11.14 给定一组记录,其关键码值是字母。记录按照下面的顺序插入:C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J。给出插入这些记录后的 4 阶 B<sup>+</sup>树。假定叶结点可放多达 3 条记录。
- 11.15 假定有一个 B<sup>+</sup>树,它的内部结点可以存储多达 100 个子女,叶结点可以存储多达 15 条记录。对于 1, 2, 3, 4 和 5 层 B<sup>+</sup>树,能够存储的最大和最小记录数是多少?
- 11.16 假定有一个 B<sup>+</sup>树,它的内部结点可以存储多达 50 个子女,叶结点可以存储多达 50 条记录。对于 1, 2, 3, 4 和 5 层 B<sup>+</sup>树,能够存储的最大和最小记录数是多少?

## 11.8 项目设计

- 11.1 对变长记录实现一个二级线性索引,如图 11.1 和 11.2 所示。假定磁盘块的长度

是 1024 个字节。数据库文件中的记录一般在 20 到 200 个字节之间,包括 4 个字节的键码值。索引文件中的每条记录存储一个键码值和相应记录的第一个字节在数据库文件中的字节偏移量。顶层索引(存储在主存储器中)应当是一个简单的数组,存储索引文件相应页的最小键码值。

- 11.2 实现 2-3+ 树,即内部结点只作为占位符的 2-3 树。应当尽可能少地修改图 11.10 中的 2-3 树结点定义。
- 11.3 实现 11.5 节描述的 B+ 树。假定磁盘块是 1024 个字节,这样叶结点和内部结点都是 1024 个字节。记录应当存储一个 4 字节(int)键码值和 60 字节的数据字段。内部结点应当存储键码值/指针对,其中“指针”是子女结点在磁盘中的实际块号。内部结点和叶结点都需要空间存储各种信息,例如存储在那个结点的记录计数、指向同一层下一个结点的指针等等。这样,根据你的实现,叶结点将存储 15 条记录,内部结点将有地方存储大约 120 到 125 个子女。使用项目 9.2 中的缓冲池库管理对存储在磁盘中结点的访问。你的实现应当支持插入、删除、根据键码值精确检索和键码范围检索等操作。

原书空白

# 第四部分 应用与高级话题

第 12 章 线性表和数组高级技术

第 13 章 高级树形结构

第 14 章 分析技术

第 15 章 计算的限制

## 第 12 章 线性表和数组高级技术

对许多应用程序来说,用简单的线性表和数组完成任务就足够了,而其他应用程序则需要对一些操作的支持,这些操作不能使用第 4 章的标准线性表表示方法有效实现。本章介绍线性表和数组的各种高级实现,它们克服了简单的链表和连续数组表示方法的一些问题。本章介绍的一些主题还起到强调逻辑表示与物理实现分离的概念,因为有些“线性表”的实现有非常不同的内部组织方式。

12.1 节介绍跳跃表,它是存储按照关键码值排序记录的的概率数据结构。尽管它很简单,却以极高的可能性提供有效的访问和更新。12.2 节描述了一组广义表的表示方法,广义表是可以包含子表的表。12.3 节讨论稀疏矩阵的表示方法,稀疏矩阵是大多数元素值都为 0 的大型矩阵。12.4 节讨论存储管理技术,它实质上是分配大数组变长部分的一种方式。

### 12.1 跳跃表

设计跳跃表是为了解决基于数组的线性表和链表的基本问题:一次检索或更新需要的时间都是线性的。跳跃表是概率数据结构(probabilistic data structure)的一个例子,因为它随机地做出决定。

跳跃表提供了可以代替 BST 和相关树形结构的另外一种选择。BST 的主要问题是它可能很容易就会变得不平衡。第 11 章的 2-3 树可以保证不管数据值按照什么顺序插入,都能保持平衡,但是它的实现非常复杂。第 13 章介绍伸展树,它也能保证提供好的性能,但是它也为 BST 树增加了复杂性。跳跃表不能保证提供好的性能(好的性能定义为检索、插入和删除时间是  $\Theta(\log n)$ ),但是它提供好的性能的概率极高。它还比已经知道的平衡树形结构更容易实现。同样,它在实现难度和性能两方面进行了很好的权衡。

图 12.1 说明了跳跃表隐含的概念。图 12.1(a)是一个简单的链表,其结点按照结点值的顺序排列。检索排序的链表需要沿着链表一次一个结点地移动,平均需要访问  $O(n)$  个结点。考虑一下添加一个指向其他结点的指针,使得检索交替地跳过结点,如图 12.1(b)所示。把只有一个指针的结点定义为 0 级跳跃表结点,而把有两个指针的结点定义为 1 级跳跃表结点。

进行检索时,先沿着 1 级指针走,直到找到一个比检索关键码大的值,然后回到 0 级指针。如果需要,再多走一个结点,这样就会有效地把工作减少一半。可以继续以这种方式添加指针,直到像图 12.1(c)那样。对于一个有  $n$  个结点的表,第一个结点和中间结点最终会有  $\log n$  个指针。进行检索时,从最下面一行指针开始,走得尽可能远,一次跳过多条记录。然后,根据需要使步伐越来越小。通过这种安排,最差情况下的访问数是  $O(\log n)$ 。

下面是检索跳跃表的一个实现。每个跳跃表结点都包含一个名为 forward 的数组,它存储如图 12.1(c)所示的指针。位置 forward[0]存储一个 0 级指针,forward[1]存储一个 1 级指针,依此类推。跳跃表的类定义包括数据成员 level,它存储跳跃表中结点的最大级数。假定

跳跃表存储一个有 level 个指针的头结点。

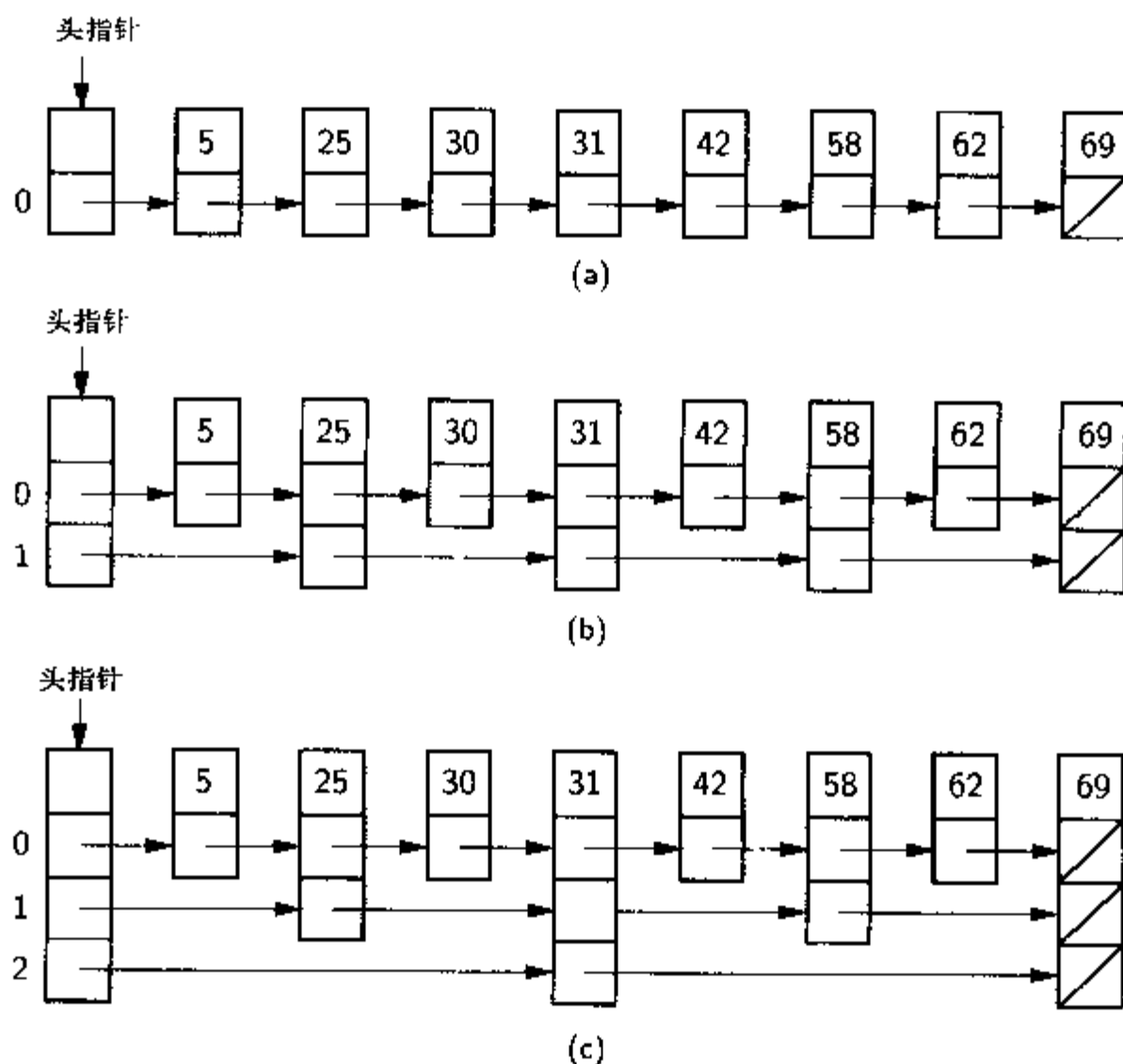


图 12.1 跳跃表概念说明

(a)一个简单的链表。(b)每隔一个结点添加一个附加指针来扩展链表。(c)理想的跳跃表,保证  $O(\log n)$  检索时间

```
public Elem search(int searchKey) { // Skiplist Search
    SkipNode x = head;                // Dummy header node
    for (int i = level; i >= 0; i--)   // For each level...
        while ((x.forward[i] != null) && // go forward
            (((Elem)x.forward[i].value()).key() < searchKey))
            x = x.forward[i];          // Go one last step
    x = x.forward[0]; // Move to actual record, if it exists
    if ((x != null) && (((Elem)x.value()).key() == searchKey))
        return (Elem)x.value();        // Got it
    else return null;                  // Its not there
}
```

在图 12.1(c) 的跳跃表中,从头结点开始,检索一个值为 62 的结点。沿着头结点在 list.level 指针进行,在这个例子中是第 2 级,这个指针指向值为 31 的结点。由于 31 小于 62,接下来从值为 31 的结点的 forward[2] 指针移到 69。由于 69 大于 62,不能到达这个结点,但是必须把当前的级数减 1。

接下来,沿着 forward[1] 到达值为 58 的结点。由于 58 小于 62,沿着 58 的 forward[1] 指针到达 69。由于 69 太大,沿着 58 的 0 级指针到达 62。由于 62 不小于 62,这样就跳出了



while 循环,向前移动一步,到达值为 62 的结点。

图 12.1(c)中理想的跳跃表组织成一半结点只有 1 个指针,四分之一的结点有 2 个指针,八分之一的结点有 3 个指针,依此类推。距离是平均划分的,实际上这是一个“完美平衡”跳跃表。在通常的插入、删除过程中维护这样一种平衡代价很大。跳跃表的关键之处是我们不需要考虑这些。每当插入一个结点,就为它指定一个级别(即指针数目)。指定的级别是随机的,使用指数分布得到,结点有一个指针的概率是 50%,有两个指针的概率是 25%,依此类推。下面的函数根据这样一种分布确定结点的级别:

```
int randomLevel() { // Pick a level on exponential distribution
    int lev;
    for (lev = 0; DSUtil.random(2) == 0; lev++); // Do nothing
    return lev;
}
```

一旦确定了结点的相应级别,下一步就是要找到在哪里插入结点,并且把它在所有相应的级别链接起来。下面是把一个新值插入跳跃表的一个实现。

```
public void insert(Elem newValue) { // Insert into skiplist
    int newLevel = randomLevel(); // Select level for new node
    if (newLevel > level) // New node will be deepest
        AdjustHead(newLevel); // Add null pointers to header
    SkipNode[] update = new SkipNode[level + 1]; // Track end of level
    SkipNode x = head; // Start at header node
    for (int i = level; i >= 0; i--) { // Search for insert position
        while ((x.forward[i] != null) &&
            (((Elem)x.forward[i].value()).key() < newValue.key()))
            x = x.forward[i];
        update[i] = x; // Keep track of end at level i
    }
    x = new SkipNode(newValue, newLevel); // Create new node
    for (int i = 0; i <= newLevel; i++) { // Splice into list
        x.forward[i] = update[i].forward[i]; // Who x points to
        update[i].forward[i] = x; // Who y points to
    }
}
```

图 12.2 说明了跳跃表的插入过程。在这个例子中,从把一个值为 10 的结点插入空跳跃表开始。假定 randomlevel 返回一个值 1(即结点在第 1 级,有 2 个指针)。由于空跳跃表没有结点,这个表的级别(即头结点的级别)必须设置为 1。插入新的结点,得到图 12.2(a)的跳跃表。

接下来插入值 20。假定这时 randomLevel 返回 0。检索过程到达值为 10 的结点,新结点就插入到它的后面,如图 12.2(b)所示。

第 3 个插入结点的值是 5,再一次假定 randomLevel 返回值 0。这样就会得到图 12.2(c)的跳跃表。

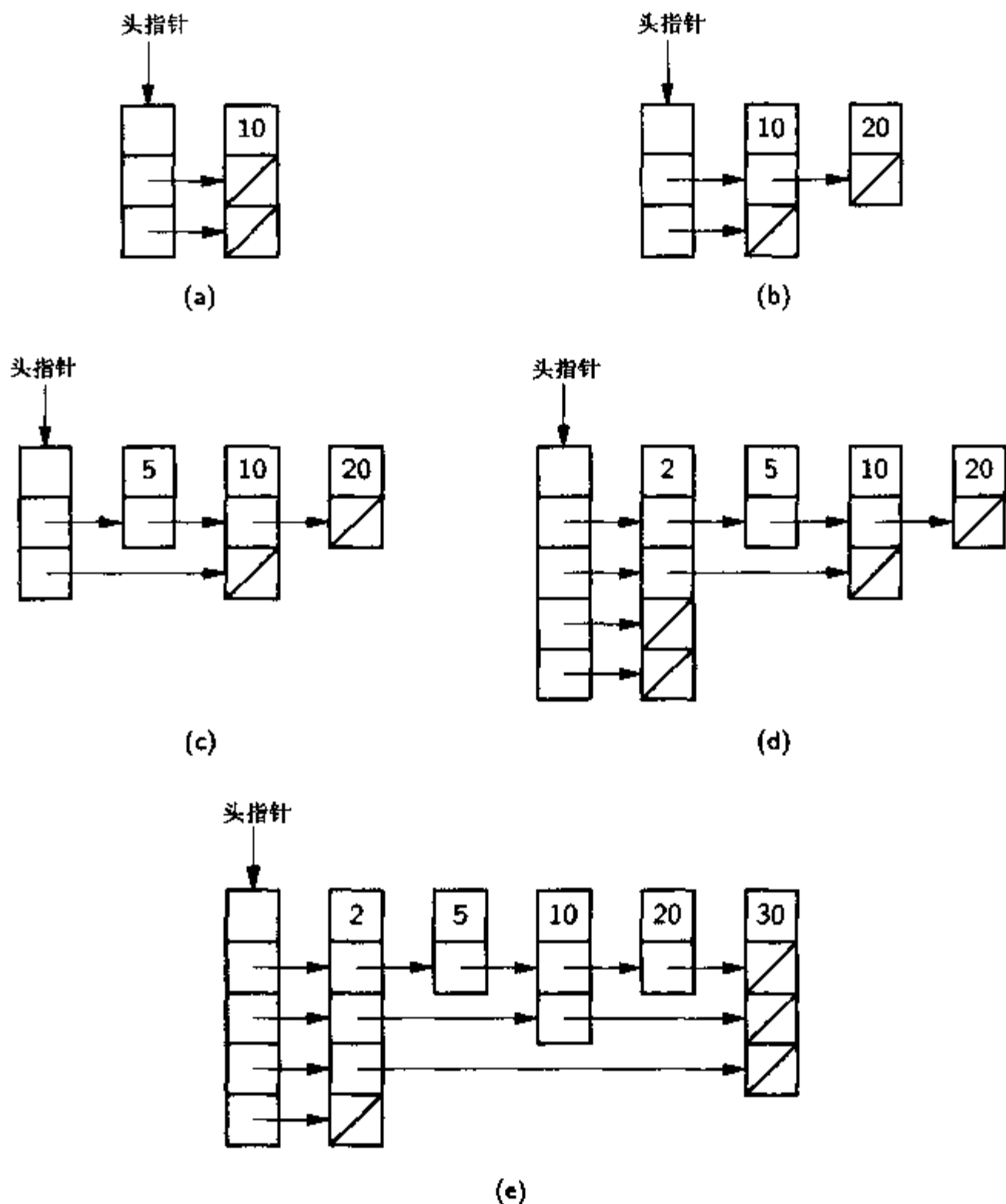


图 12.2 跳跃表插入说明

(a)在第 1 级插入初始值 10 以后的跳跃表。(b)在第 0 级插入值 20 以后的跳跃表。(c)在第 0 级插入值 5 以后的跳跃表。(d)在第 3 级插入值 2 以后的跳跃表。(e)在第 2 级插入值 30 以后的最终跳跃表

第 4 个插入结点的值是 2, 假定 `randomLevel` 返回值 3。这意味着跳跃表的级别必须增加, 引起头结点得到两个附加的 (null) 指针。此时, 新结点被添加到表的前面, 如图 12.2(d) 所示。

最后, 在第 2 级插入一个值为 30 的结点。这一次, 仔细看一看 `update` 用于什么数组。在检索新结点的相应位置时, 它存放每一级到达的最远的结点。检索过程从头结点的第 3 级开始, 进行到存储值为 2 的结点。由于这个结点的 `forward[3]` 是 null, 在这一级不能继续向前走了, 这样, `update[3]` 存储一个指向值为 2 的结点的指针。同样, 在第 2 级也不能继续向前走了, 因此 `update[2]` 也存储一个指向值为 2 的结点的指针。在第 1 级, 走到存储值为 10 的结点, 这是第 1 级所能走到的最远的地方, 因此, `update[1]` 存储一个指向值为 10 的结点的指针。最后在第 0 级, 在值为 20 的结点结束, 在这里, 可以添加值为 30 的结点。新结点的 `for-`

ward[i]指针设置为  $\text{update}[i] \rightarrow \text{forward}[i]$ , 存储在  $\text{update}[i]$  从 0 到 2 索引的结点把它们的  $\text{forward}[i]$  指针改为指向新结点。这样就在跳跃表中的所有级别插入新结点了。

remove 函数留作练习。 $\text{update}$  数组也是作为检索要删除记录的一部分建立的, 然后对于  $\text{update}$  数组标识的那些结点, 就让它们的前向指针进行调整, 绕过被删除的记录。

randomLevel 可能为一个新插入的结点产生很高的级别, 或者很低的级别。跳跃表中可能有很多结点都有多个指针, 导致不必要的插入代价, 因为没有跳过很多记录, 在检索期间产生很差的 ( $\Theta(n)$ ) 性能。相反, 级别很低的结点太多。在最差情况下, 所有的结点都在第 0 级, 相当于一个常规链表。如果这样, 检索将又需要  $\Theta(n)$  时间。然而, 性能极差的可能性非常低, 一行中 10 个结点都在第 0 级的概率只有  $1/1024$ 。像跳跃表这样的概率数据结构的理念是“不用担心, 会很好的”。只要简单地接受 randomLevel 的结果, 并且期望概率最终会按照我们的意愿工作即可。这种方法的好处是算法简单, 在一般情况下对于所有操作只需要  $\Theta(\log n)$  时间。

实际上, 跳跃表可能会比 BST 有更好的性能。由于数据插入顺序的原因, BST 可能有很差的性能。跳跃表的性能不依赖于数据插入表中的顺序。随着跳跃表中结点数目的增加, 遇到最差情况的可能性也会以指数方式递减, 这样, 跳跃表就显示出理论上的最差情况(在这种情况下, 跳跃表的所有操作都是  $\Theta(n)$ ) 和实际中的  $\Theta(\log n)$  平均情况快速增长的概率之间的距离。

## 12.2 广义表

回忆一下第 4 章, 线性表是一个形如  $(x_0, x_1, \dots, x_{n-1})$  的有限的、已排序项的序列, 其中  $n \geq 0$ 。可以用 null 或者 () 表示一个空线性表。在第 4 章, 假定线性表的所有元素都具有相同的数据类型。这一节把这个线性表的定义加以扩展, 允许元素是任意的。一般说来, 线性表的元素是以下两种类型之一:

1. 一个原子(atom)。原子是一个某种类型的数据记录, 例如一个数字、符号或者字符串。
2. 另外一个线性表, 称为一个子表(sublist)。

一个包含子表的线性表可以写成:

$$(x_1, (y_1, (a_1, a_2), y_3), (z_1, z_2), x_4)$$

在这个例子中, 线性表有 4 个元素。第 2 个元素是子表  $(y_1, (a_1, a_2), y_3)$ , 第 3 个元素是子表  $(z_1, z_2)$ 。注意, 子表  $(y_1, (a_1, a_2), y_3)$  还包含一个子表。如果表 L 有一个或多个子表, 那么就称 L 是一个广义表(multilist)。没有子表的表就称为线性表(linear list)或者链(chain)。注意, 广义表的这个定义很符合定义 2.1 中集合的定义, 其中集合的成员既可以是一个基本元素, 也可以是一个集合。

根据广义表的形式是树的形式、DAG(有向无环图)还是一般的图的形式, 可以使用各种方法限制广义表的子表。纯表(pure list)是这样一种表结构, 它的图对应于一棵树, 如图 12.3 所示。也就是说, 从根结点到任何结点只有一条路径, 这等于说没有对象可以在表中出现多于一次。在纯表中, 每对括号都对应树的一个内部结点, 表的成员对应于结点的子女, 表中的原子对应于叶结点。

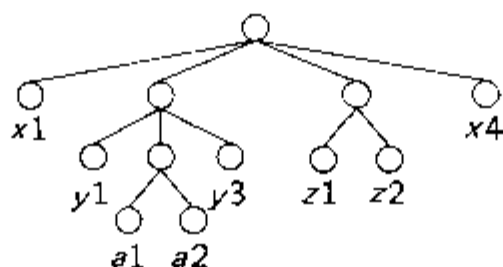


图 12.3 用树表示广义表的例子

可重入表(reentrant list)是这样一种表结构,它的图对应于一个 DAG。从根结点开始,可以通过多条路径访问一个结点。这等于说只要不形成回路,对象(包括子表)可能会在表中多次出现。所有边都向下指,从表示一个表或者子表的结点到达它的元素。图 12.4 是一个可重入表。要使用括号表示法写出这个表,可以根据需要重复结点,这样,图 12.4 中表的括号表示法可以写成:

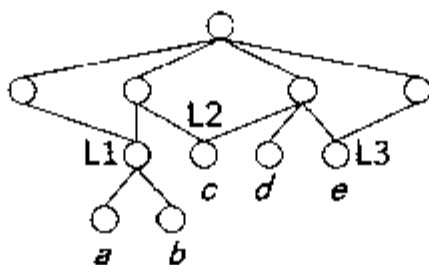
$$(((a, b)), ((a, b), c), (c, d, e), (e))$$


图 12.4 可重入广义表的例子。表结构的形状是一个 DAG(所有边都向下指)

为了简便起见,可以采用一种约定,允许对子表和原子标号,例如“L1:”。每当重复出现一个标号,就可以用对应于这个标号的元素代替。这样,图 12.4 中表的括号表示法可以写成:

$$((L1:(a, b)), (L1, L2:c), (L2, d, L3:e), (L3))$$

循环表(cyclic list)是这样一种表结构,它的图对应于任何有向图,有向图中可能包含回路。图 12.5 就是这样一个表。为了用括号表示法写出这样一个表,需要使用标号。下面是图 12.5 中表的表示:

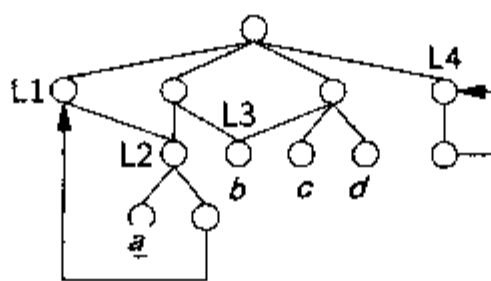
$$(L1:(L2:(a, L1)), (L2, L3:b), (L3, c, d), L4:(L4))$$


图 12.5 循环表的例子。这种结构的形状是一个有向图

广义表可以采用多种方式实现。从本书前面对线性表、树和图数据结构建议的实现中,可以熟悉广义表的大多数实现方式。

最简单的方法是使用数组表示,这对具有定长元素的链可以工作得很好,它相当于第 4 章的简单线性表,在第 4 章简要讨论了对具有变长元素的链的实现。可以把嵌套的子表看成变长元素。要使用这种方法,需要每个子表的一些开始和结束标识。可以使用 6.5 节讨论的顺序树的实现的一个变体,这没有什么奇怪的,因为纯表等价于一般的树形结构。最简单的方法

是显式地存储写出的表示法中的括号。在图 12.6 中,括号可以作为单独的符号存储,但是,像任何顺序表示法一样,对第  $n$  个子表的访问必须顺序地从表的开始处进行。

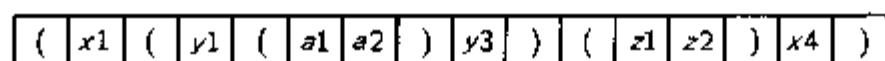


图 12.6 把图 12.3 表示成纯表

因为纯表等价于树,也可以使用链接分配方法支持对表的子女的直接访问。简单的线性表可以用链表来表示,纯表也可以用链表来表示,但是要有一些附加的标记字段来标识这个结点是原子还是子表。如果是子表,数据字段就会指向子表的第一个元素,这在图 12.7 中说明了。

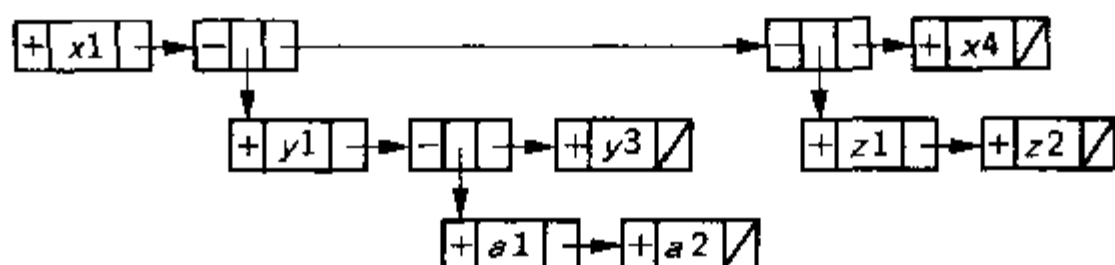


图 12.7 图 12.3 中纯表的链表表示。每个链结点中的第一个字段存储一个标记位。如果标记位存储“+”,那么数据字段存储一个原子。如果标记位存储“-”,那么数据字段就存储一个指向子表的指针

另一种方式是使用存储两个指针字段的链结点表示表中的所有元素,除了原子以外,它只包含数据,这就是程序设计语言 LISP 使用的系统,图 12.8 说明了这种方法。指针可能包含一个标记位,来标识它指向什么,被指向的对象也可能存储一个标记位,来标识它自己,标记把原子和表结点区分开。这种实现可以方便地支持可重入表和循环表,因为一个结点可以指向任何其他结点。

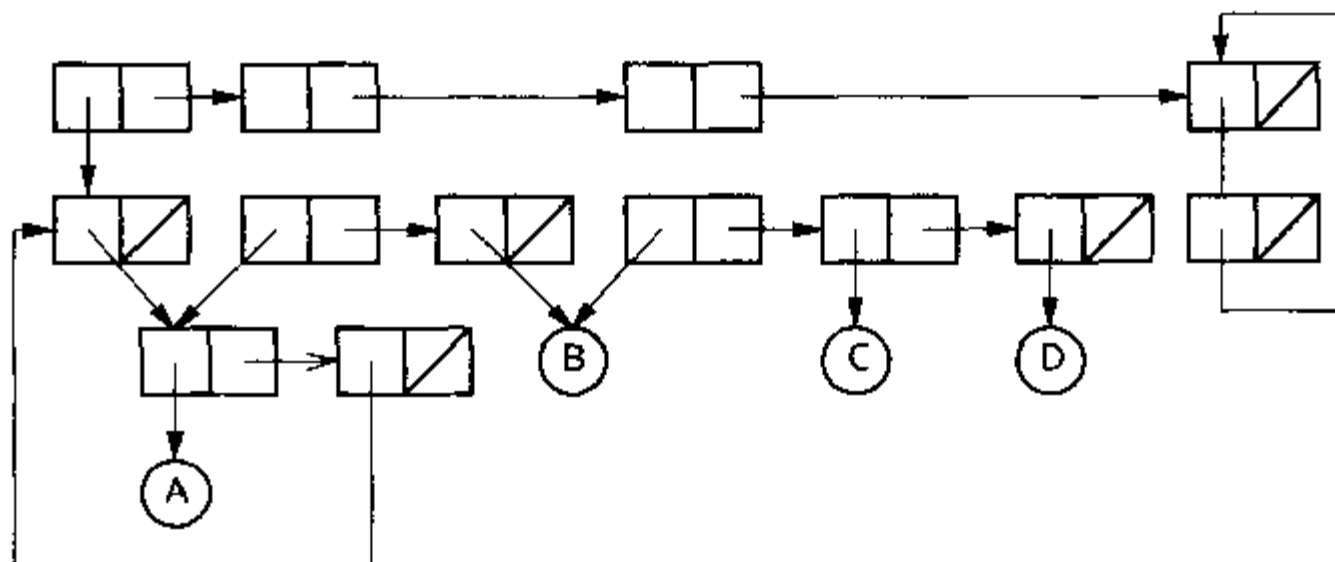


图 12.8 图 12.5 中循环广义表的类 LISP 链表表示。每个链结点存储两个指针,一个指针既可以指向一个原子,也可以指向另一个链结点。链结点用两个框表示,原子用圆圈表示

### 12.3 矩阵的表示方法

有些应用程序需要表示一个大型二维矩阵,矩阵中许多元素的值为 0。一个例子是下三

角矩阵,它源于解联立方程的系统。下三角矩阵在位置 $[r, c]$  ( $r < c$ )存储 0 值,如图 12.9(a)所示,这样,矩阵的右上三角总是 0。可以利用这个事实来节省空间。不必在一个  $n \times n$  数组中存储需要的  $n(n+1)/2$  份信息,使用一个长度为  $n(n+1)/2$  的线性表就会节省空间。只有能够找到某种方法,来定位表中对应于原来矩阵的位置 $[r, c]$ 的元素,这样做才可行。

a <sub>00</sub>	0	0	0
a <sub>10</sub>	a <sub>11</sub>	0	0
a <sub>20</sub>	a <sub>21</sub>	a <sub>22</sub>	0
a <sub>30</sub>	a <sub>31</sub>	a <sub>32</sub>	a <sub>33</sub>

(a)

a <sub>00</sub>	a <sub>01</sub>	a <sub>02</sub>	a <sub>03</sub>
0	a <sub>11</sub>	a <sub>12</sub>	a <sub>13</sub>
0	0	a <sub>22</sub>	a <sub>23</sub>
0	0	0	a <sub>33</sub>

(b)

图 12.9 三角矩阵  
(a)下三角矩阵。(b)上三角矩阵

要得到这样一个方程,注意到矩阵的第 0 行只有一个非 0 值,第 1 行有两个非 0 值,依此类推。这样,第  $r$  行前面共有  $r$  行,总共有  $\sum_{k=1}^r k = (r^2 + r)/2$  个非 0 元素。再加  $c$  到达第  $r$  行的第  $c$  个位置就会得到下面的方程,把原来矩阵中的位置 $[r, c]$ 转换为线性表中的正确位置:

$$\text{matrix}[r, c] = \text{list}[(r^2 + r)/2 + c]$$

可以使用类似的方法存储上三角矩阵,即一个在位置 $[r, c]$  ( $r > c$ )有 0 值的矩阵,如图 12.9(b)所示。在这种情况下,对于一个  $n \times n$  上三角矩阵,方程应当是:

$$\text{matrix}[r, c] = \text{list}[rn - (r^2 + r)/2 + c]$$

现在考虑更为复杂的情况,存储在一个  $n \times n$  矩阵中的大量元素值都是 0,没有规则限制哪些位置是 0,哪些位置不是 0。这种矩阵称为稀疏矩阵(sparse matrix)。一种表示稀疏矩阵的方法是把两个坐标结合成一个单一的值,用它作为散列表的关键码。这样,如果想要知道矩阵中某个位置的值,就在散列表中按照关键码检索。如果没有找到这个位置的值,就认为它是 0。如果对矩阵的所有查询都是对指定位置的访问,这是一种理想的方法。然而,如果要在指定行找到第一个非 0 元素,或者在指定列中找到当前元素下面的下一个非 0 元素,那么散列表就不那么令人满意了。

实现矩阵的另一种方法称为正交表(orthogonal list),如图 12.10 所示。这里有一组行首,每个行首包含一个指向一组矩阵记录的指针。第二个组是列首,也包含一个指向一组矩阵记录的指针。每个非 0 矩阵元素都存储指针,指向它在这一行前后的非 0 邻居,每个非 0 元素还包含指向它在列的前后的非 0 邻居的指针。这样,每个非 0 元素都存储它自己的值、它在矩阵中的位置和 4 个指针。通过遍历一行或者一列就可以找到非 0 元素。注意一个指定行的第一个非 0 元素可以出现在任何列;同样,任何行或者列中的相邻非 0 元素都可能在数组的任何(更高的)行或者列中。这样,每个非 0 元素都必须显式地存储它的行和列的位置。

要知道矩阵中某个位置是否包含非 0 元素,就要遍历相应的行或列。例如,要查找第 7 行第 1 列的元素,可以遍历第 7 行,也可以遍历第 1 列。当遍历一行或一列时,如果到达了元素的正确位置,那么它的值就不是 0。如果遇到一个更高位置的元素,那么要查找的元素就不在稀疏矩阵中。在这种情况下,元素的值就是 0。例如,当遍历图 12.10 中矩阵的第 7 行时,首先到达第 7 行第 1 列的元素。如果这就是要查找的元素,那么检索就可以停止了。如果要

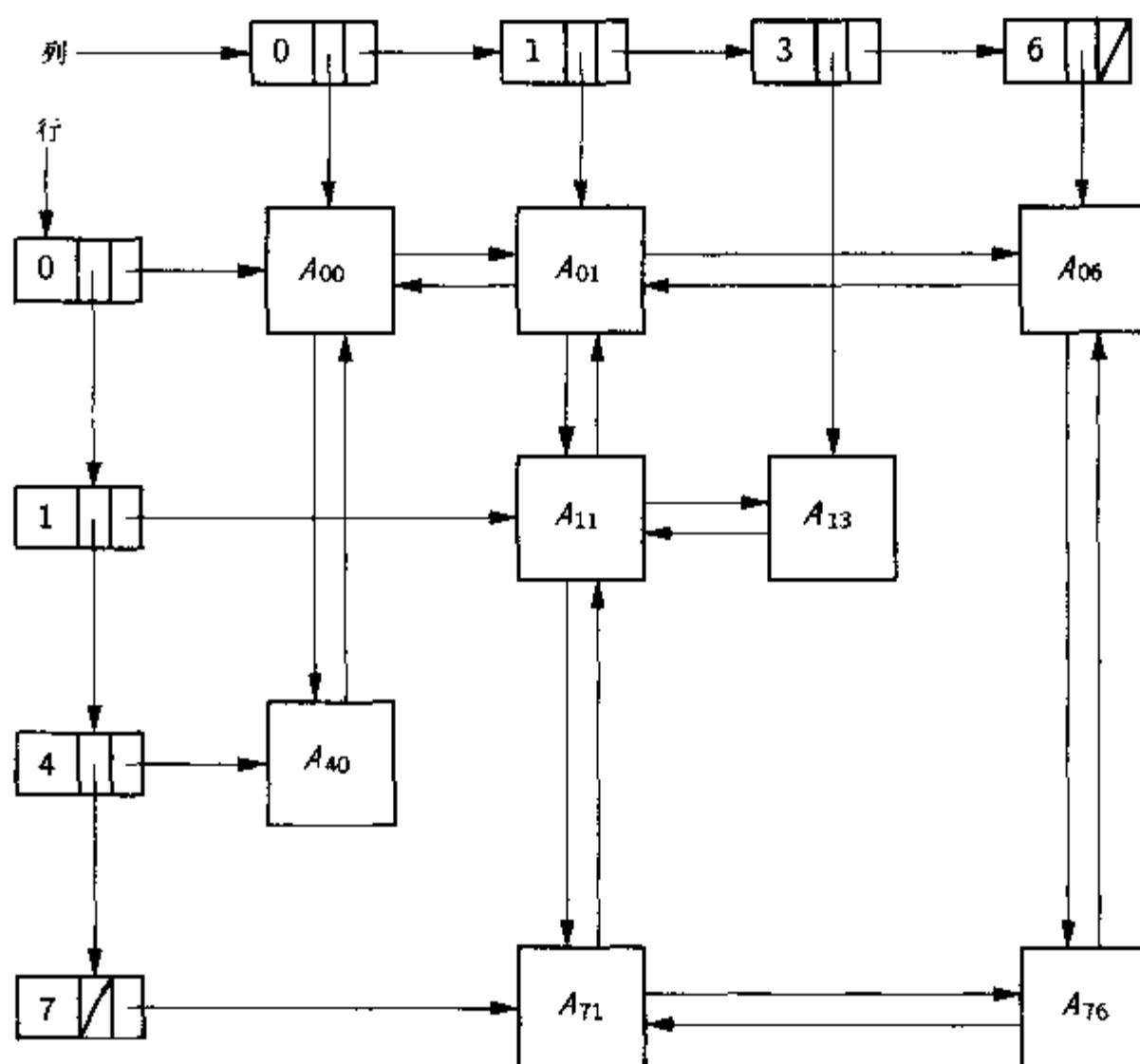


图 12.10 稀疏矩阵表示方法的一个例子

查找第 7 行第 2 列的元素,那么检索就沿着第 7 行继续进行,接下来到达第 6 列的元素。这时就知道没有存储稀疏矩阵第 7 行第 2 列的元素。

插入、删除操作可以按照类似的方式完成,都是在相应的行和列中进行元素的插入、删除。

使用稀疏矩阵表示方法存储的每个非 0 元素都比简单的  $n \times n$  矩阵中的元素占用更多的空间。什么时候稀疏矩阵表示方法比标准表示方法的空间效率更高呢?要计算出结果,就需要确定标准矩阵表示方法需要多少空间,稀疏矩阵表示方法需要多少空间。稀疏矩阵的大小依赖于非 0 元素的数目,而标准矩阵的大小是不变的。需要知道指针和数据值的(相对)大小。为了简单起见,计算忽略了行指针和列指针占用的空间。

作为一个例子,假定一个数据值使用 2 个字节,一行或者一列的索引使用 2 个字节,1 个指针使用 4 个字节。一个  $n \times m$  矩阵需要  $2nm$  个字节。稀疏矩阵中每个非 0 元素需要 22 个字节(4 个指针,2 个数组索引和 1 个数据值)。如果把  $X$  设为非 0 元素的百分比,可以计算出  $X$  值小于多少时稀疏矩阵表示方法在空间上更有效。使用方程:

$$22mnX = 2mn$$

解出  $X$ ,发现当  $X < 1/11$  时,也就是说,当少于 9% 的元素不是 0 值时,使用稀疏矩阵这种实现方法在空间上更有效。对于这两种实现,数据值、指针或矩阵索引相对大小的各种不同值都能够导致显著不同的分界点。

处理稀疏矩阵需要的时间依赖于矩阵存储的非 0 元素数。当检索一个元素时,其代价是目标元素所在的行或列中目标元素前面元素的数目。当一个矩阵存储  $n$  个非 0 元素,而另一个矩阵存储  $m$  个非 0 元素时,两个矩阵相加这样的操作的代价就是  $O(n + m)$ 。

## 12.4 存储管理

本书中描述的大多数数据结构实现,它们存储和访问的对象都是大小相同的,例如存储在一个线性表或者一个树中的整数。已经描述了一些简单方法,用于在数组或栈中存储变长记录。本节讨论存储管理技术,它用于处理变长空间请求这个一般性的问题。存储管理的基本模型是有一(大)块连续的存储位置,称之为存储池(memory pool)。一些存储空间请求定期发出,用以在存储池中得到一些空间。存储管理器必须在存储池的某个位置找到一块连续的位置,它至少是被请求的大小。响应这样的一次请求就称为一次存储分配(memory allocation)。以前分配的存储空间可能在将来的某个时间返回给存储管理器,称为存储回收(memory deallocation)。

如果所有的请求和释放都遵循一种简单的模式,例如后请求先释放(栈顺序),或者先请求先释放(队列顺序),那么存储管理就非常容易了。在这一节考虑更一般的情况,可以按照任何顺序请求和释放任意大小的块,这称为动态存储分配(dynamic storage allocation)。动态存储分配的一个例子是为编译器的运行时刻环境管理空闲存储区,例如Java语言中的系统new操作。另一个例子是在多任务操作系统中管理主存储器,这里,一个程序可能需要一定大小的空间,存储管理器必须记录哪个程序正在使用哪一块主存储器。还有一个例子是磁盘的文件管理器,当一个磁盘文件被创建、扩展或者删除时,文件管理器必须分配或回收磁盘空间。

使用这种方式管理的一块存储器或者磁盘空间,有时也称为一个堆(heap)。这里使用的术语“堆”与其他章节中讨论的堆数据结构不同,这里的“堆”表示动态存储管理方法访问的空闲存储区。

在本节其余部分,首先研究用于动态存储管理的技术,然后处理这样一个问题:当存储池中没有一个足够大的存储块以满足请求时应该做什么。

### 12.4.1 动态存储分配

为了进行动态存储分配,把存储器看成一个由一组变长块组成的数组,其中一些块是空闲的(free),一些块则是保留的(reserved)或者已分配的。空闲块链接到一起,形成一个可利用空间表,以满足将来的存储请求。图12.11说明了经过一系列存储分配和回收之后出现的情况。

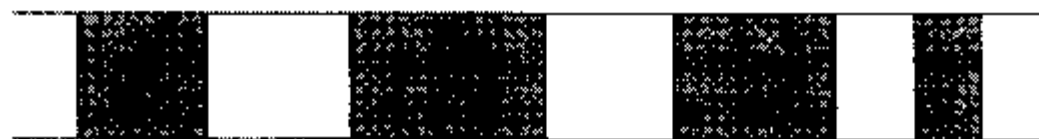


图 12.11 动态存储分配模型。存储区由一组变长的块组成,有些已经分配,有些仍然空闲。在这个例子中,阴影区域表示当前已经分配的存储区,没有阴影的区域表示没有使用的存储区,用于将来的分配

当存储管理器收到一个存储请求之后,就要在可利用空间表中找到足够大的块以满足请求;如果找不到这样的块,那么存储管理器就要求助于12.4.2小节讨论的失败处理策略(failure policy)。

如果有一个请求要得到 $m$ 个字,但是没有一个块的大小正好是 $m$ ,那么就要用一个更大



的块来满足请求。在这种情况下,一种可能的办法是把整个块都交给存储分配的请求者。当块的大小只比请求的大小稍微大一点时,这样做是合适的;否则,对于一个大小为  $k$  的块,而  $k > m$ ,存储管理器要保留  $k - m$  大小的空间,以形成一个新的空闲块,而其余的空间则用于满足请求。

在动态存储管理中会遇到两种类型的碎片。当满足存储请求时产生大量的小空闲块,其中没有一个可以用于满足一个一般的请求,从而导致外部碎片(External fragmentation)的出现。当把多于  $m$  个字的存储空间分配给要得到  $m$  个字大小的请求时,就会浪费空闲存储空间,从而产生内部碎片(Internal fragmentation),这相当于当把多个簇分配给文件时产生的内部碎片。内部碎片和外部碎片的区别如图 12.12 所示。

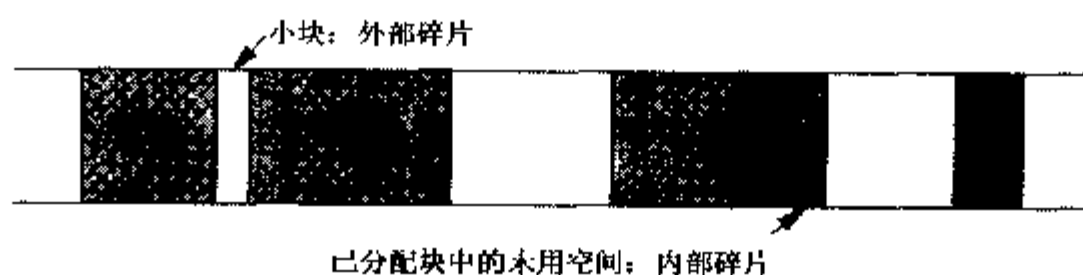


图 12.12 内部碎片和外部碎片说明

有些存储管理方法牺牲内部碎片的空间,使得存储管理更简单(可能会减少外部碎片)。例如,在以簇为单位分配文件空间的文件管理系统中不会产生外部碎片。另一个例子是这一节讨论的伙伴方法(buddy method)。

有一种方法是找到一个足够大的块满足请求,可能要把其余的空间作为空闲块,这种方法称为顺序适配(sequential fit)方法。

### 顺序适配方法

顺序适配方法试图找到一个“好”的块来满足存储请求。这里描述的三种顺序适配方法都假定空闲块组织成双链表,如图 12.13 所示。由于空闲块是空的,存储管理器可以利用它来完成自己的工作,也就是说,存储管理器暂时“借用”空闲块中的空间来维护自己的双链表,为此,每个未分配的块都必须足够大,以保存这些指针。除此之外,通常允许存储管理器向一个保留块增加一些字节的空间,用于它自己的目的,也就是说,对于一个需要  $m$  个字节空间的请求,存储管理器可能会分配稍微多于  $m$  个字节的空间,额外的字节由存储管理器自己使用,而不是供请求者使用。这里认为所有存储块都像图 12.14 那样组织,带有标记和链表指针的空间。这里,根据块的开始和最后的标记位来区别空闲块和保留块,除此之外,空闲块和保留块都在块的开始处紧接着标记位有一个长度标识器,标识这个块有多大,空闲块在块结尾处标记位前面有另外一个长度标识器。最后,空闲块有左指针和右指针,指向它在可利用空间表中的相邻块。

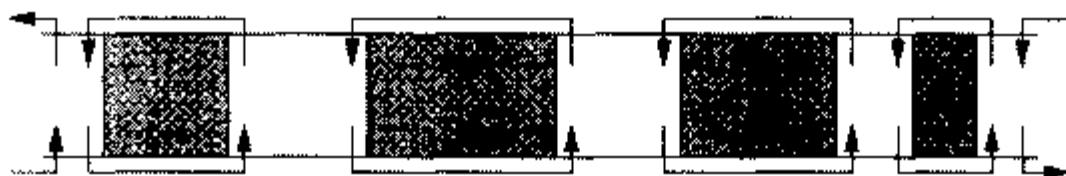


图 12.13 存储管理器看到的空闲块双链表。阴影区域代表已分配的存储区。没有阴影的区域是可利用空间表的部分

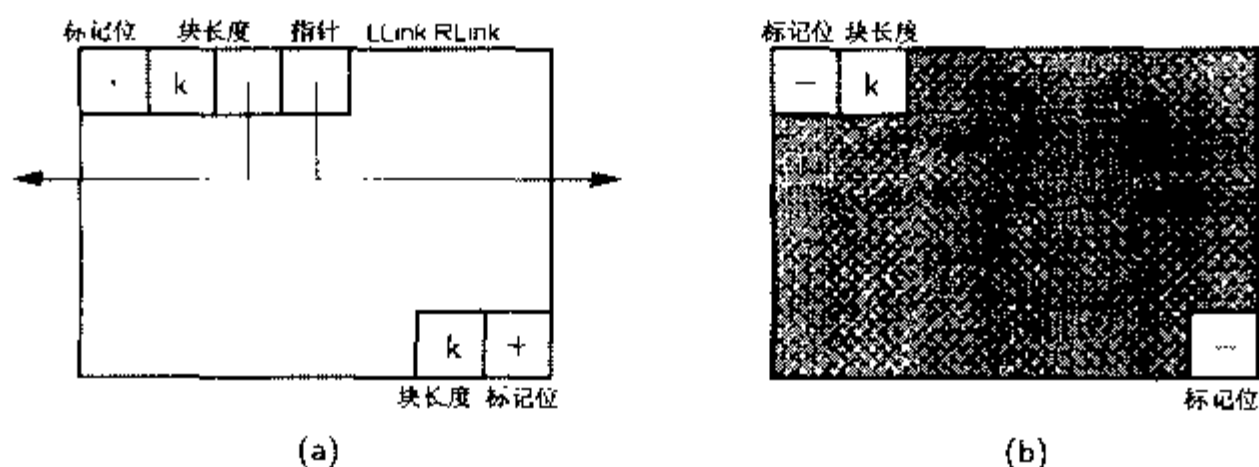


图 12.14 存储管理器看到的块。每个块中都包括一些附加信息,如可利用空间表的链指针、开始和结尾标记和一个长度字段

(a)空闲块的布局。块的开始有一个标记位字段,块长度字段和两个指向可利用空间表的指针 LLink 和 RLink。块的结尾包含第二个标记位字段和第二个块长度字段。(b)一个有  $k$  个字节的保留块。存储管理器把这  $k$  个字节和块开始处的附加标记位字段、块长度字段与块结尾处的第二个标记位字段加到一起

通过与每个块相关的信息字段,存储管理器能够根据需要分配和回收块。当一个请求到来,要求得到  $m$  个字的存储区时,存储管理器检索空闲块链表,直到找到一个“合适”的块用于分配。下面就讨论它怎样确定一个块是否合适。如果一个块正好包含  $m$  个字(加上用于标记和长度字段的的空间),那么就把它从可利用空间表中移出;如果块(大小为  $k$ )足够大,那么剩余的  $k - m$  个字就作为可利用空间表中的一个块保留在当前位置,并且设置相应的标记、长度和链指针字段。图 12.15 给出的存储管理器类说明了这个过程,为了简单起见,假定所有字段(标记、长度和指针)都是一个整型大小。

```
class MemManager implements MemManADT {
    private static final int STARTTAG = 0;    // Start tag offset
    private static final int FULLSIZE = 1;    // Size field offset
    private static final int USERSIZE = 2;    // User size offset
    private static final int LPTR = 2;        // Left freelist pointer
    private static final int RPTR = 3;        // Right freelist pointer
    private static final int DATAPOS = 3;    // Start of data
    private static final int FREE = -1;       // Tag value
    private static final int RESERVED = -2;   // Tag value
    private static final int ENDSIZE = 4;     // Size field offset
    private static final int FREEENDTAG = 5;   // Tag field offset
    private static final int RESENDTAG = 3;    // Tag field offset
    private static final int MINEXTRA = -2;    // Extra space needed
    private static final int FREEOVERHEAD = 6; // Number free fields
    private static final int RESOVERHEAD = 4;  // Number of res fields
    private static final int MINREQUEST = 2;  // Smallest data request

    int[] mempool;           // The data space
    MemHandle freelist;      // Free memory access

    MemManager(int size) {   // Constructor
```

```

    mempool = new int[size]; // Allocate space
    freelist = new MemHandle(0); // Start of freelist
    mempool[STARTTAG] = mempool[size - 1] = FREE;
    mempool[FULLSIZE] = mempool[size - 2] = size - FREEOVERHEAD;
    mempool[LPTR] = mempool[RPTR] = 0; // Circ doubly-linked list
}

public int[] getValue(MemHandle h) { // Return data for h
    int startpos = h.getPos();
    int length = mempool[startpos + USERSIZE];
    int startdata = startpos + DATAPOS;
    int[] stuff = new int[length];
    for (int i = 0; i < length; i++) stuff[i] = mempool[startdata + i];
    return stuff;
}

// Sample sequential fit implementation: First fit
private int pickFreeBlock(int length) {
    if (freelist == null) return -1; // No free block
    int freestart = freelist.getPos();
    for (int curr = freestart;;)
        if (mempool[curr + FULLSIZE] >= (length + MINEXTRA))
            return curr;
        else {
            curr = mempool[curr + RPTR];
            if (curr == freestart) return -1; // No block available
        }
}

public MemHandle request(int[] info) {
    int datasize;
    if (info.length < MINREQUEST) // Minimum necessary for
        datasize = MINREQUEST; // a sustainable block
    else datasize = info.length;
    int start = pickFreeBlock(info.length);
    if (start == -1) return null; // No block is big enough
    if (mempool[start + FULLSIZE] > (datasize + RESOVERHEAD)) {
        // Fix up the remaining free space
        int oldsize = mempool[start + FULLSIZE] - datasize - RESOVERHEAD;
        mempool[start + oldsize + FREEENDTAG] = FREE;
        mempool[start + oldsize + ENDSIZE] = oldsize;
        mempool[start + FULLSIZE] = oldsize;

        // Now, fix up the new block
    }
}

```

```

        int newstart = start + mempool[start + FULLSIZE] + FREEOVERHEAD;
        mempool[newstart + STARTTAG] = RESERVED;
        mempool[newstart + FULLSIZE] = datasize;
        mempool[newstart + USERSIZE] = info.length;
        mempool[newstart + datasize + RESENDTAG] = RESERVED;
        for (int i = 0; i < info.length; i++)
            mempool[newstart + DATAPOS + i] = info[i];
        return new MemHandle(newstart);
    }
else { // Give over the whole block, remove from freelist
    // First, adjust the freelist
    if (mempool[start + RPTR] == start)
        freelist = null; // This is the last block
    else {
        mempool[mempool[start + RPTR] + LPTR] = mempool[start + LPTR];
        mempool[mempool[start + LPTR] + RPTR] = mempool[start + RPTR];
    }
    // Now, fill in the block
    mempool[start + STARTTAG] = RESERVED;
    mempool[start + FULLSIZE] += FREEOVERHEAD - RESOVERHEAD;
    mempool[start + USERSIZE] = info.length;
    for (int i = 0; i < info.length; i++)
        mempool[start + DATAPOS + i] = info[i];
    mempool[start + mempool[start + FULLSIZE] + RESENDTAG] = RESERVED;
    return new MemHandle(start);
}
}

public void release(MemHandle h) {
    // Implementation is left as an exercise
}

// class MemManager

```

图 12.15 类 MemManager 用于说明存储管理

存储管理器存储类型为 int 的数组。它有三个基本函数,如下面的接口所定义:

```

interface MemManADT {
    public MemHandle request (int [ ] info); //Request space
    public int [ ] getValue (MemHandle h); //Retrieve data
    public void release (MemHandle h); //Release space
} //interface MemManADT

```

函数 request 以一个整型数组作为输入,把数组的内容存储到存储池中。它返回一个

MemHandle 值,使得 MemManager 类的用户能够访问已存储的数据。函数 `getValue` 返回与  $h$  关联的整型数组,函数 `release` 释放与 MemHandle  $h$  关联的整型数组占用的存储器。

MemManager 的实现与图 12.14 稍微有一些不同,保留块存储两个长度字段:一个是保留空间的实际长度,另一个是块中实际存储用户数据的空间长度(可能小一些)。

在函数 `request` 中,当一个块的大小比  $m$  大到值得保存其余部分时,就把这个块一分为二。如果一个块被分开,前面的部分仍然留在可利用空间表中,后面的部分被分配。函数 `request` 的主体用于设置存储管理器使用的各个字段。

当一个块  $F$  被释放时,必须把它合并到可利用空间表中。如果不注意合并相邻的空闲块,那么这只是简单地向空闲块双链表插入的过程。然而,有必要合并相邻的块,因为这样存储管理器就能够满足最大可能长度的请求。由于有存储在每个块最后的标记字段和长度字段,合并很容易完成,如图 12.16 所示。存储管理器首先检查块  $F$  之前的存储单元,看一看前一个块(称它为  $P$ )是不是空闲的。如果是,那么  $P$  标记位前面的单元就存储  $P$  的大小,这样就在存储器中标识了这个块的开始位置。然后, $P$  就可以简单地让它的长度扩展到包含块  $F$ 。如果块  $P$  不是空闲的,那么就只把块  $F$  添加到可利用的空间表中。最后,还要检查跟在块  $F$  后面的位。如果这个位标识下一个块是(称它为  $S$ )空闲的,那么就把  $S$  从可利用空间表中清除,并且相应地扩展  $F$  的长度。函数 `release` 的编码留作习题。

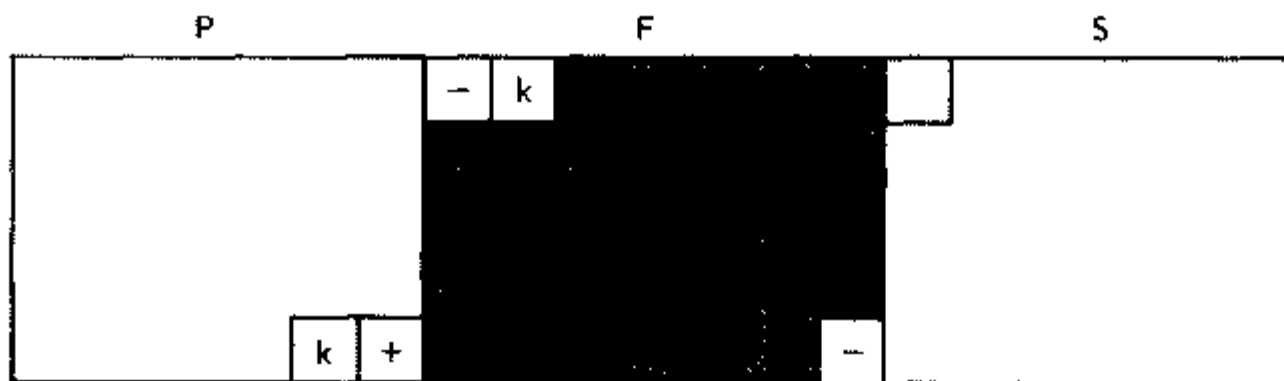


图 12.16 把块  $F$  添加到可利用空间表中。存储池中  $F$  开始处前面的字存储前一个块  $P$  的标记位。如果  $P$  是空闲的,就把  $F$  合并到  $P$  中。通过使用  $F$  的长度字段就可以找到  $F$  的开始处。同样,紧接着  $F$  结尾的字就是块  $S$  的标记字段。如果  $S$  是空闲的,就把它合并到  $P$  中

现在考虑如何选择一个“合适的”空闲块来满足存储请求。为了说明这个过程,假定可利用空间表中有 4 个块,大小为 500, 700, 650 和 900(按照这个顺序),假定有一个需要 600 个单位存储空间的请求。在这个例子中,忽略上面讨论的标记位、链指针和长度字段的额外开销。

最简单的选择一个块的方法是沿着可利用空间表向下找,直到找到一个大小至少为 600 的块。图 12.15 中的函数 `pickFreeBlock` 说明了这个过程。这个块中的剩余空间都留在可利用空间表中。如果从表的开始处进行,向下找到大小至少为 600 的第一个空闲块,就会选择长度为 700 的块。由于这种方法选择空间足够大的第一个块,就称它为首先适配(first fit)。可以改进性能的一种简单变体是,不总是从可利用空间表的表头开始,而是记住前一次检索最后到达的位置,从那里开始。当到达可利用空间表的结尾时,检索再从可利用空间表的表头开始。这一修改减少了不必要的检索数目,这些检索是上一次检索经过的一些小块。

首先适配有一个潜在的缺陷:它可能会“浪费”很大的块,从而无法满足以后的大请求。避免使用不必要的大块的策略称为最佳适配(best fit)。最佳适配查看整个表,在其中找出一个

至少和请求大小一样大的最小块(即请求“最佳”或者最近适配)。继续前面那个例子,最佳适配是大小为 650 的块,剩下大小为 50 的空间。最佳适配的缺陷是它需要检索整个表,另一个问题是最佳适配块的剩余部分可能很小,这样对将来的请求就没有什么用处了,也就是说,最佳适配有可能使外部碎片问题特别严重,而同时使得无法满足一个偶尔大请求的可能性最小。

与最佳适配相反的一种策略可能会行得通,因为它尽量使外部碎片问题的效果最小,这种方法称为最差适配(worst fit)。最差适配总是分配表中最大的块,希望块的剩余部分能用于满足将来的请求。在这个例子中,最差适配是长度为 900 的块,留下长度为 300 的空间。如果有一些不常见的大请求,这种方法满足这些请求的可能性很小。如果请求往往倾向于有相同的大小,那么这可能是一种有效的策略。像最佳适配一样,最差适配也需要为每一个存储请求检索整个表,找到最大的块,而且,也可以对可利用空间表从大到小排序空闲块。

哪一个策略最好呢?这依赖于预期的存储请求类型。如果请求的都是很大的块,那么最佳适配可能完成得很好;如果请求都是相同大小,很少有大块或者小块请求,首先适配或者最差适配可能完成得很好。但是,总是有一种请求模式,使得三种顺序适配方法中有一种完成得很好,而另外两种无法完成。例如,如果请求序列是 600, 650, 900, 500, 100, 可利用空间表包含块(500, 700, 650, 900), 首先适配可以满足所有请求,但最佳适配却不能。同样最佳适配可以满足请求序列 600, 500, 700, 900, 而首先适配却不能。

### 伙伴方法

顺序适配方法依赖于空闲块的链表,每一次存储请求查找一个合适的块时都必须检索整个表,这样,对于一个包含  $n$  个块的可利用空间表,在最差情况下查找一个合适空闲块的时间将是  $\Theta(n)$ 。合并相邻空闲块也有些复杂。最后,空闲块和保留块都需要标记字段和长度字段。空闲块中的字段不花费任何空间(由于它们存放在未用的存储器中),但是保留块中的字段却带来额外开销。

伙伴系统解决了大多数问题,它检索合适大小的块很有效,合并相邻空闲块很简单,在保留块中不需要存储标记字段或者其他信息字段。伙伴系统假定存储空间的大小是  $2^N$ , 对于某个整数  $N$ 。空闲块和保留块的大小都是  $2^k$ ,  $k \leq N$ 。在任何给定的时间,都可能有各种大小的空闲块和保留块。伙伴系统为每一种大小的空闲块都保留一个单独的列表,最多可能有  $N$  个这样的列表,因为只有  $N$  种不同的大小。

当到达一个需要  $m$  个字的请求时,首先确定  $k$  的最小值,使得  $2^k \geq m$ 。如果可利用空间表中有一个大小为  $2^k$  的块,就从可利用空间表中选择它。伙伴系统不担心内部碎片问题:大小为  $2^k$  的整个块都分配出去。

如果没有大小为  $2^k$  的块,就找下一个更大一点的块。把这个块分成两半(如果需要可以重复下去),直到创建一个大小为  $2^k$  的目标块。作为这个分割过程的副产品产生的任何其他块都放到相应的可利用空间表中。

伙伴系统的缺陷是它允许内部碎片,例如,一个 257 个字的请求需要一个大小为 512 的块。伙伴系统的主要好处是外部碎片少,检索一个大小合适的块的代价比最佳适配等方法更少,因为只需要在大小为  $2^k$  的块表中找到第一个可用的块,另外合并相邻空闲块很容易。

这种方法称为伙伴系统的原因在于合并发生的方式。任何大小为  $2^k$  的块的伙伴(buddy)是另一个同样大小的块,地址中除了第  $k$  位相反以外其余位都相同。例如,图 12.17(a)中开

始地址为 0000 长度为 8 的块有一个地址为 1000 的伙伴。同样,在图 12.17(b)中,地址为 0000 长度为 4 的块有伙伴 0100。如果按照地址值对空闲块排序,通过检索正确的块长度表就可以找到伙伴。合并只需要把结合的伙伴移到可利用空间表中,形成一个更大的块。

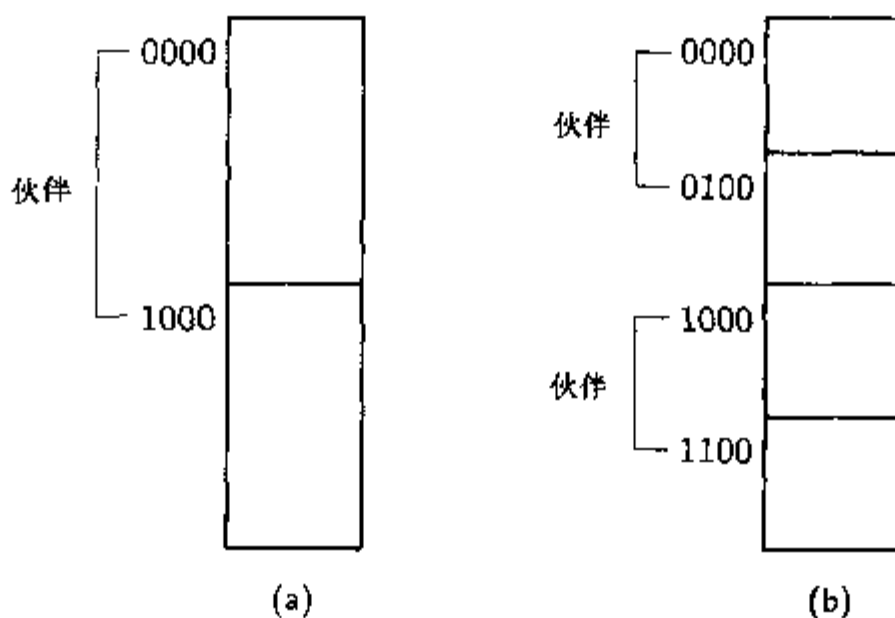


图 12.17 伙伴系统的例子

### 其他存储分配方法

除了顺序适配和伙伴方法以外,还有许多其他存储管理方法。特别是,如果应用程序非常复杂,可能需要把可用存储器分成多个存储区(memory zone),每一个存储区采用不同的存储管理方法,这种方式称为隔离式存储方法(segregated storage method)。例如,有些存储区可能使用一种简单的存储访问模式,如用于定长记录的先进先出,因此使用栈就可以对这个存储区进行简单而有效地管理。其他存储区可能需要这一节讨论的用于一般目的的存储分配方法。存储区的好处是可以对存储器的一些部分进行更有效的管理,其缺陷是如果存储区的大小选择不当,一个存储区可能已经填满,而另一个存储区却有过多的空闲存储空间。

存储管理的另一种方法是对所有存储请求都强加一个标准大小。在磁盘文件管理中已经看到这种方法的一个例子,其中所有文件都以簇为单位分配,这种方法会导致内部碎片,但是管理由簇组成的文件比管理任意大小的文件容易。以簇为单位分配的方法还可以放松存储请求需要由一个连续存储块来满足的限制。大多数磁盘文件管理器和操作系统的主存管理器都采用簇或页式系统,页的管理通常使用一个缓冲池来完成,以便在主存储器中有效分配可用的页。

#### 12.4.2 失败处理策略和无用单元收集

在处理过程中的某一时刻,存储管理器可能遇到一个无法满足的存储请求。在有些情况下,可能什么都不能做,因为没有足够的空闲存储空间来满足请求,而应用程序可能需要立即满足请求。在这种情况下,存储管理器别无选择,只能返回一个错误,从而导致应用程序执行的失败。然而,在许多情况下,除了返回错误以外还有其他选择,这些可能的选择总称为失败处理策略(failure policy)。

在有些情况下,可能有足够的空闲存储空间来满足请求,但是它分散成多个小块。当使用顺序适配存储分配方法时就可能发生这种情况,外部碎片是一些小块,收集起来就可以满足请求。在这种情况下,可以通过移动保留块压缩存储空间,以便空闲存储空间结合成一块。这种

方法的一个问题是应用程序必须能够处理这样一个事实,它的所有数据现在都已经移到了不同的地方。如果应用程序以任何方式依赖于数据的绝对位置,都会引起严重的后果。处理这个问题的-一种方法是使用句柄(handle),句柄是对存储位置的二级间接指引。存储分配例程不返回一个指向存储块的指针,但是返回一个变量的指针,这个变量则指向存储位置,这个变量就是句柄。句柄从不移动它的位置,但是可能移动块的位置,并且因此修改句柄的值。图 12.18 说明了这种方法。

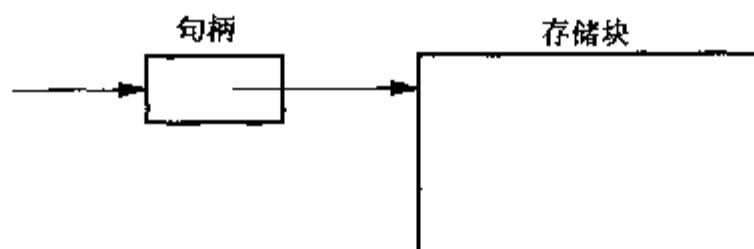


图 12.18 对动态存储管理使用句柄。存储管理器返回句柄的地址来响应一个存储请求,句柄中存储实际存储块的地址。通过这种方式,不影响应用程序就可以移动存储块(修改句柄中的地址)

在某些应用程序中采用的另一种失败处理策略是延迟存储请求,直到有足够的存储空间可用。例如,一个多任务操作系统可能要采用这样一种策略,它一直等到一个进程有足够的存储空间可用才允许这个进程运行。尽管这种延迟可能会使用户很烦恼,然而它比终止整个系统要好。这里假定其他进程最终都会停止并释放存储空间。

另一种选择就是给存储管理器分配更多的空间。在一个分区存储分配系统中,存储管理器是更大系统的一部分,这可能是一个可行的选择。在一个实现自己的存储管理器的 Java 程序中,有可能从系统级 new 操作符中得到更多的存储空间,如第 4 章中可利用空间表所做的那样。

最后-一个失败处理策略是无用单元收集(garbage collection),考虑下面一组语句:

```
p = new int [i];
q = new int [j];
p = q;
```

在 C++ 等一些语言中,这被认为是一种很坏的形式,由于第三个赋值语句,原来分配给 p 的空间丢失了,这块空间不能再被程序使用了。这种丢失的存储空间称为无用单元(garbage),也称为内存泄漏(memory leak)。当没有程序变量指向一块空间时,以后就不可能对它进行访问。注意如果另一个变量已赋值指向 p 的空间,那么重新赋值 p 就不会产生无用单元。

Java 对无用单元的处理有些不同。Java 程序使用 new 操作符分配存储空间,以后丢弃对该空间的所有指针。这样,Java 中的无用单元很普遍,实际上在正常处理期间无法避免。当 Java 消耗尽所有存储空间后,它就求助于失败处理策略,称之为无用单元收集(garbage collection),恢复无用单元占用的空间。无用单元收集包括检查所有被管理的存储池,确定哪些部分仍然被使用,哪些部分是无用单元。特别地,有一个表保存所有程序变量,从其中任何一个变量都无法到达的存储位置就认为是无用单元。当无用单元收集器执行时,所有没有使用的存储位置都放到可利用空间表中,供将来使用。这种方法的好处是能够方便地收集无用单元。但它也有缺陷,从用户的角度来看,每当进行无用单元收集时系统必须终止。例如,在 Emacs



文本编辑器中这是很明显的现象,这个编辑器一般使用 LISP 实现,LISP 也是实现无用单元收集的一种语言。用户偶尔必须等待一会儿,因为这时存储管理系统正在完成无用单元收集。

传统上有许多算法用于无用单元收集。第一个是引用计数(reference count)算法。每个存储块都包含一个计数字段。每当指针指向一个链结点,其引用计数就会加 1。无论什么时候指针不再指向这个链结点,其引用计数就会减 1。如果引用计数变为 0,那么这个链结点就被认为是无用单元,立即放回到可利用空间表中。这种方法的好处是它不需要一个显式的无用单元收集阶段,因为每当存储单元成为无用单元时就立即被放到可利用空间表中。

UNIX 文件系统就使用引用计数算法。文件可以有多个名字,称为链。文件系统为每个文件保持一个链接数目计数,每当一个文件被“删除”时,实际上就把它的链字段简单地减 1。如果还有另一个链指向这个文件,那么文件系统就不回收空间。每当链的数目减少到 0 时,就把文件的空间归还,供重新利用。

引用计数方法有许多主要缺陷。首先,必须为每一个存储对象维护一个引用计数。当对象很大时(如一个文件),这样会工作得很好。然而,在一个像 LISP 这样的系统中,其中的存储对象一般包括两个指针或一个值(一个原子),它就不会很好地工作。当存在无用单元循环引用时,就会出现另一个主要问题。考虑一下图 12.19,这里每个存储对象都被指向一次,但是对象的集合仍然是无用单元,因为没有指针指向对象集合。这样,当存储对象没有循环地链接在一起时,引用计数才能工作(如 UNIX 文件系统),其中文件只能组织成一个 DAG。



图 12.19 无用单元循环引用的例子。循环中的所有存储单元都有非 0 引用计数,因为每个单元都有一个指针指向它,但是整体上却是无用单元

无用单元收集的另外一种方法称为标记/清除(mark/sweep)策略。这里,每一个存储对象只需要一个标记位,而不是一个引用计数字段。当可利用空间表用完时,就会进入如下的无用单元收集阶段:

1. 清除所有标记位。
2. 变量表中的每一个变量沿着指针进行深度优先检索,对于 DFS 期间遇到的每一个存储单元,都把它标记位打开。
3. 通过对存储池进行“清除”,访问所有存储单元。所有未标记的单元就认为是无用单元,可以放到可利用空间表中。

标记/清除方法的好处是它比引用计数方法需要的空间更少,对于循环情况也能工作。然而,它有一个很大的缺陷,就是被“隐藏”了的、进行处理所需要的空间需求。DFS 是一个递归算法:要么必须递归实现它(在这种情况下编译器的运行系统维护一个栈),要么存储管理器维护它自己的栈。如果所有存储空间都包含在一个链表中会发生什么呢?那么递归的深度(或者栈的大小)就是存储单元的数目。但是,DFS 需要的空间必须在可以想像得到的最差时刻,即当存储空间已经用完的时候是可用的。

幸而有一种聪明的技术使得 DFS 得以完成,而不需要额外的用于栈的空间。与前面相反,它使用被遍历的结构保存栈。在遍历中每深入一步,不需要向栈中存储一个指针,而是“借用”即将深入的那个指针,把它设置为指向刚经过的结点,如图 12.20 所示。每一个借用的

指针存储一个附加位,告诉我们进入被指向的链结点的左分支还是右分支。在任何给定时刻只能从根结点深入一条路径,而且可以沿着指针的线索回来。在返回的时候(相当于弹出递归栈),设置指针指回原来的位置,以便不破坏结构。这称为 Deutsch-Schorr-Waite 无用单元收集算法。

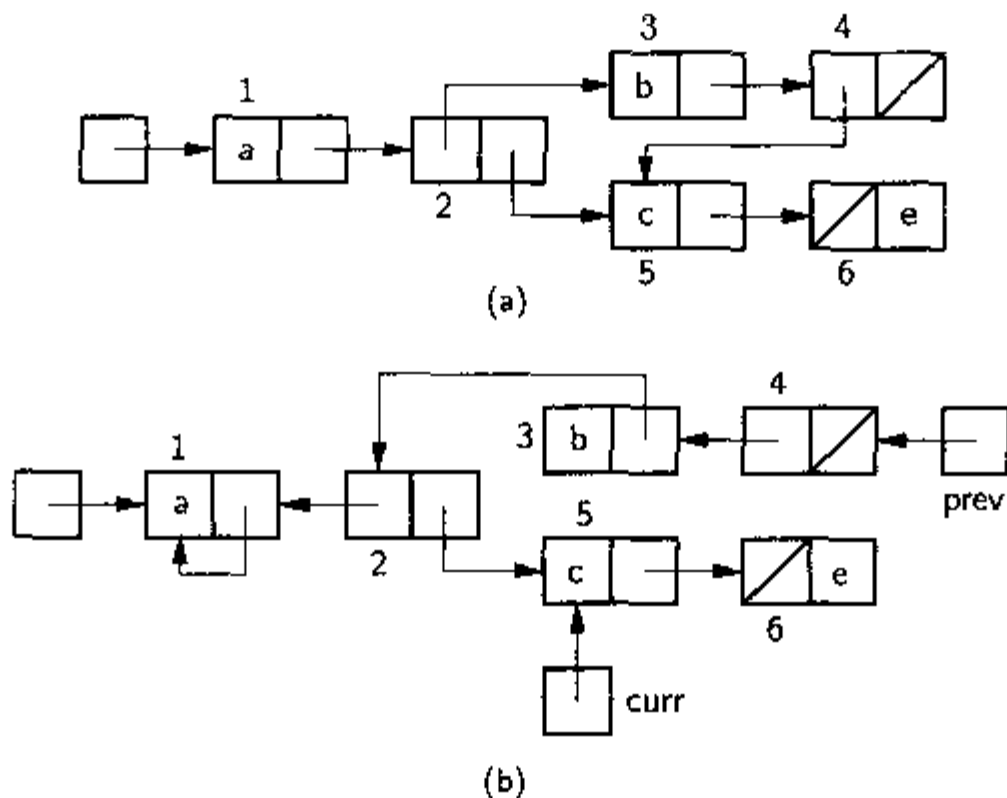


图 12.20 Deutsch-Schorr-Waite 无用单元收集算法的例子

(a)初始广义表结构。(b)当无用单元收集算法正在处理链结点 5 的时刻(a)的广义表结构。从 prev 变量到结构头结点的指针链已经被无用单元收集算法(临时)创建了

## 12.5 深入学习导读

有关跳跃表的更多信息,参见 William Pugh 的“Skip lists: A Probabilistic Alternative to Balanced Trees”[Pug90]。

有关操作系统的介绍性教材包括了与存储管理问题相关的许多主题,其中有磁盘中的文件布局 and 主存储器中的信息缓存。这里涉及的有关存储管理、缓冲池和页式分配的所有主题都与操作系统的实现相关。例如,可参见 Harvey M. Deitel 的《Operating System》[Dei90]。对于 Apple Toolbox 系统中关于存储管理的讨论,参见 Apple Computer 的《Inside Macintosh》系列 [App85]。

对于有关 LISP 的信息,参见 Friedman 和 Felleisen 的《The little Lisper》[FF89]。另一个好的 lisp 参考书是 Guy L. Steele 的《Common Lisp: The Language》[Ste84]。Emacs 是一个优秀的文本编辑器和一个全面开发的程序设计环境,有关它的信息参见 Richard M. Stallman 的《GNU Emacs Manual》[Sta94]。

## 12.6 习题

- 12.1 给出插入下列值后得到的跳跃表。在每次插入之后画出跳跃表。对于每一个值,假定其对应结点的深度在下面的表中给出。

值	深度
5	2
20	0
30	0
2	0
25	1
26	3
31	0

12.2 如果有一个从不需要修改的表,那么就可以使用一种比跳跃表更简单的方法。其思想仍然是一样的,为了对第  $i$  个元素有效地访问,向表结点添加附加的指针。怎样向单链表的每个元素中添加第二个指针,才能在  $O(\log n)$  时间访问第  $i$  个元素呢?

12.3 跳跃表结点的预期级别(即指针的平均数)是多少?

12.4 编写一个函数,从跳跃表中清除一个具有给定值的结点。

12.5 编写一个函数,在跳跃表中找到第  $i$  个结点。

12.6 对于第 12.3 节的稀疏矩阵表示,当数据值需要 8 个字节、数组索引需要 2 个字节、指针需要 4 个字节的时候,要使它比标准的二维矩阵表示方法在空间上更有效,矩阵中有多少比例的值必须是 0?

12.7 给定一个纯表的链接表示方法如下:

$$(x_1, (y_1, y_2, (z_1, z_2), y_4), (w_1, w_2), x_4)$$

编写一个逆置算法,在所有级别逆置子表(包括最高级)。对于这个例子,结果将是对应于:

$$(x_4, (w_2, w_1), (y_4, (z_2, z_1), y_2, y_1), x_1)$$

的链接表示方法。

12.8 编写一个函数,把一个元素添加到 12.3 节稀疏矩阵表示方法中的一个指定的位置。

12.9 编写一个函数,从 12.3 节稀疏矩阵表示方法中的一个指定位置删除一个元素。

12.10 编写一个函数,将一个使用 12.3 节稀疏矩阵表示方法表示的矩阵转置。

12.11 编写一个函数,将两个使用 12.3 节稀疏矩阵表示方法表示的矩阵相加。

12.12 编写一个存储管理分配和回收例程,其中所有请求和释放都遵循后请求、先释放(栈)的顺序。

12.13 编写一个存储管理分配和回收例程,其中所有请求和释放都遵循先请求、先释放(队列)的顺序。

12.14 编写一个函数,把一个块返回给顺序适配存储管理系统。

12.15 假定存储池中包含 3 块空闲存储空间。其大小是 1300, 2000 和 1000。给出符合下列情况的存储请求的例子:

- (a) 首先适配分配方法能够工作,但是最佳适配和最差适配分配方法不能完成。
- (b) 最佳适配分配方法能够工作,但是首先适配和最差适配分配方法不能完成。
- (c) 最差适配分配方法能够工作,但是首先适配和最佳适配分配方法不能完成。

## 12.7 项目设计

- 12.1 实现 12.1 节的跳跃表。跳跃表类的 ADT 应当尽可能类似于 BST 类的 ADT。
- 12.2 实现 12.3 节的稀疏矩阵表示方法,你的实现应当完成:
  - 在给定位位置向矩阵插入一个元素。
  - 从矩阵的给定位位置删除一个元素。
  - 检索矩阵中给定位位置的元素。
  - 对一个矩阵进行转置。
  - 两个矩阵相加。
- 12.3 完成图 12.15 中 MemManager 类的实现。你的实现应当支持三种顺序适配方法:首先适配、最佳适配和最差适配。测试一下你的系统,根据经验确定在什么情况下每种方式完成得最好。
- 12.4 编写一个基于 12.4.1 小节伙伴方法的存储管理系统。你的系统应当支持对指定大小块的请求和对前面请求块的释放。
- 12.5 实现图 12.20 说明的 Deutsch-Schorr-Waite 无用单元收集算法。

## 第 13 章 高级树形结构

本章介绍在特定的应用程序中使用的几种树形结构。13.1 节介绍的 Trie 结构普遍用于存储字符串,适于存储、检索字典,它还可以用于说明关键码空间分解的概念。13.2 节的伸展树是 BST 的一个变体。伸展树是自平衡检索树的一个例子,不管记录的插入顺序怎样,它都能保证很好的性能。13.3 节介绍了几种空间数据结构,用于组织带有  $x, y$  坐标的点数据。

对于每一种数据结构,这里都给出其基本操作的描述。实际实现则留在读者的项目设计中。

### 13.1 Trie 结构

BST(二叉检索树)的形状依赖于数据记录插入顺序。记录插入顺序的一种排列可能产生一个平衡树,而另一种排列则可能产生一个链表形状的、不平衡的树。原因是存储在根结点中的关键码值把关键码值范围分成两个部分:小于根结点关键码值的关键码值和大于根结点关键码值的关键码值。其结果是,BST 依赖于树中根结点关键码值和其他结点关键码值分布之间的关系,可能是平衡的,也可能是不平衡的。这样,BST 就是一种基于对象空间(object space)分解而组织的数据结构,这样说是因为关键码值范围分解是由存储在树中的对象(即数据记录中的关键码值)驱动的。

如果不希望进行对象空间分解,另外一种选择就是对树中每一个结点在关键码范围内预定义划分位置,也就是说,应当预定义根结点,把关键码范围划分成相等的两半,而不管数据记录的具体值或插入顺序。关键码值在关键码范围中小的部分的那些记录存储在左边的子树中,而关键码值在关键码范围中大的部分的那些记录存储在右边的子树中。

尽管这样的分解规则不一定会产生平衡树(如果记录在关键码值范围内的分布不好,树也可能不平衡),至少树的形状不依赖于关键码的插入顺序。进而,树的深度还受到关键码范围精度的限制,即树的深度决不会比存储一个关键码值需要的位数更大。例如,如果关键码是 0 到 1023 之间的整数,那么关键码的存储长度就是 10 个二进制位,这样,可能到了第 10 位才能确定两个关键码是否相同。在最差情况下,两个关键码将沿着树中同样的路径,直到第 10 个分支才分开,结果,树的深度不会超过 10 层。

基于关键码值范围相等子划分的分解称为关键码空间(key space)分解。在计算机图形学中,一种相关的技术称为图像空间(image space)分解,这个术语有时也应用于基于关键码空间分解的数据结构。基于关键码空间分解的任何数据结构都称为一个 Trie 结构。据说“trie”来源于“retrieval”,但是,这并不意味着这个词发音为“tree”,从而与单词“tree”的正常使用混淆。“Trie”实际上发音为“try”。

像 B<sup>+</sup> 树一样,基于关键码空间分解的树结构只在叶结点存储数据记录。内部结点只是作为占位符占据位置并引导检索过程。

图 13.1 说明了 Trie 结构的概念。为了确定关键码值范围的中间值,必须给关键码值强

加上下限。由于这个例子中插入的最大值是 120, 假定范围从 0 到 127 (因为 128 是大于 120 的 2 的幂中的最小值)。关键码的二进制值确定在检索过程中的给定点选择左边的分支还是右边的分支, 最重要的位在根结点确定分支方向。图 13.1 给出了一个二叉 Trie 结构(binary trie), 这样说是因为在这个例子中 Trie 结构是一个二叉树。

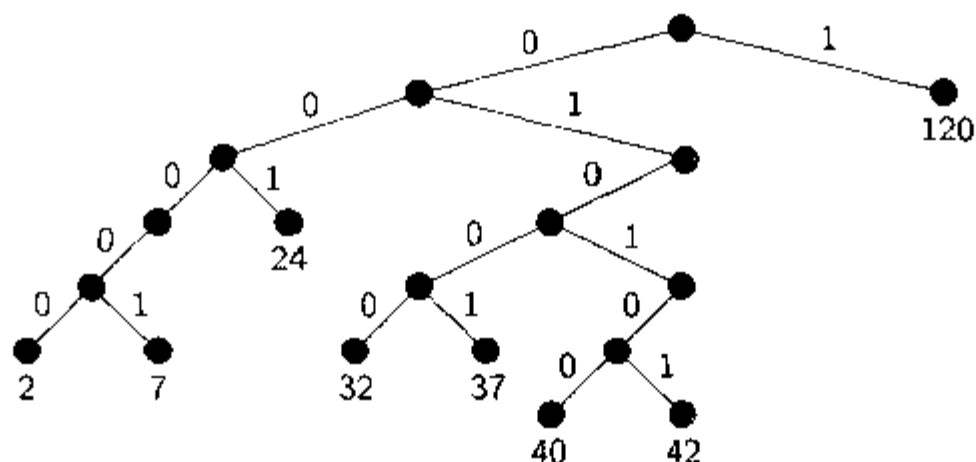


图 13.1 值( 2,7,24,31,37,40,42,120 )集合的二叉 Trie 结构,所有数据值都存储在叶结点。边使用用于确定每一个结点分支方向的二进制位标号。关键码值的二进制形式确定到达记录的路径,假定每个关键码都表示成一个 7 位的值,这个值表示 0 到 127 之间的一个数

5.4 节的 Huffman 编码树是二叉 Trie 结构的另一个例子。Huffman 树中的所有数据值都在叶结点, 每一个分支都把可能的字母编码范围分成两半。Huffman 编码实际上来源于 Trie 结构中的字母位置。

尽管这些是二叉 Trie 结构的例子, 实际上 Trie 结构可以用任何分支因子建立。通常, 分支因子通过使用的字母表来确定。对于二进制数, 它的字母表是  $\{0, 1\}$ , 结果就是一个二叉 Trie 结构。其他字母表会产生其他分支因子。

一个 Trie 结构的应用是存储字典中的单词。这样一个 Trie 结构称为字母表 Trie 结构(alphabet trie)。为了简单起见, 下面的例子忽略字母大小写。字母表包括 26 个标准英文字母和一个特殊符号(\$), 这个符号表示字符串的结束。这样, 每个结点的分支因子(高达)27。一旦建立了字母表 Trie 结构, 就可以用它确定一个单词是否在字典中。考虑一下在图 13.2 的字母表 Trie 结构中检索一个单词, 被检索单词的第一个字母确定从根结点进入哪一个分支, 第二个字母确定在第二层进入哪一个分支, 依此类推。只有能够到达一个单词的字母才能在分支中显示出来。在图 13.2(b)中, Trie 结构的叶结点存储实际单词的一份拷贝, 而在图 13.2(a)中, 单词是根据与每个分支相关的字母建立的。

实现字母表 Trie 结构中结点的一种方式是把这个结点作为一个按照字母索引的、有 27 个指针的数组。由于大多数结点的分支只能到达字母表中一小部分可能的字母, 另外一种实现是使用一个指针链表指向子女结点, 如图 6.9 所示。

图 13.2(b)字母表 Trie 结构中叶结点的深度与 Trie 结构中的结点数目没有关系, 但是, 结点的深度却依赖于把这个结点的单词与其他结点的单词区分开所需要的字符数。例如, 如果单词“anteater”和单词“antelope”都存储在 Trie 结构中, 那么直到第 5 个字母才能区分开这两个单词, 这样, 这些单词的存储至少需要 5 层深。一般说来, 字母表 Trie 结构中结点深度的限制因素是存储单词的长度。

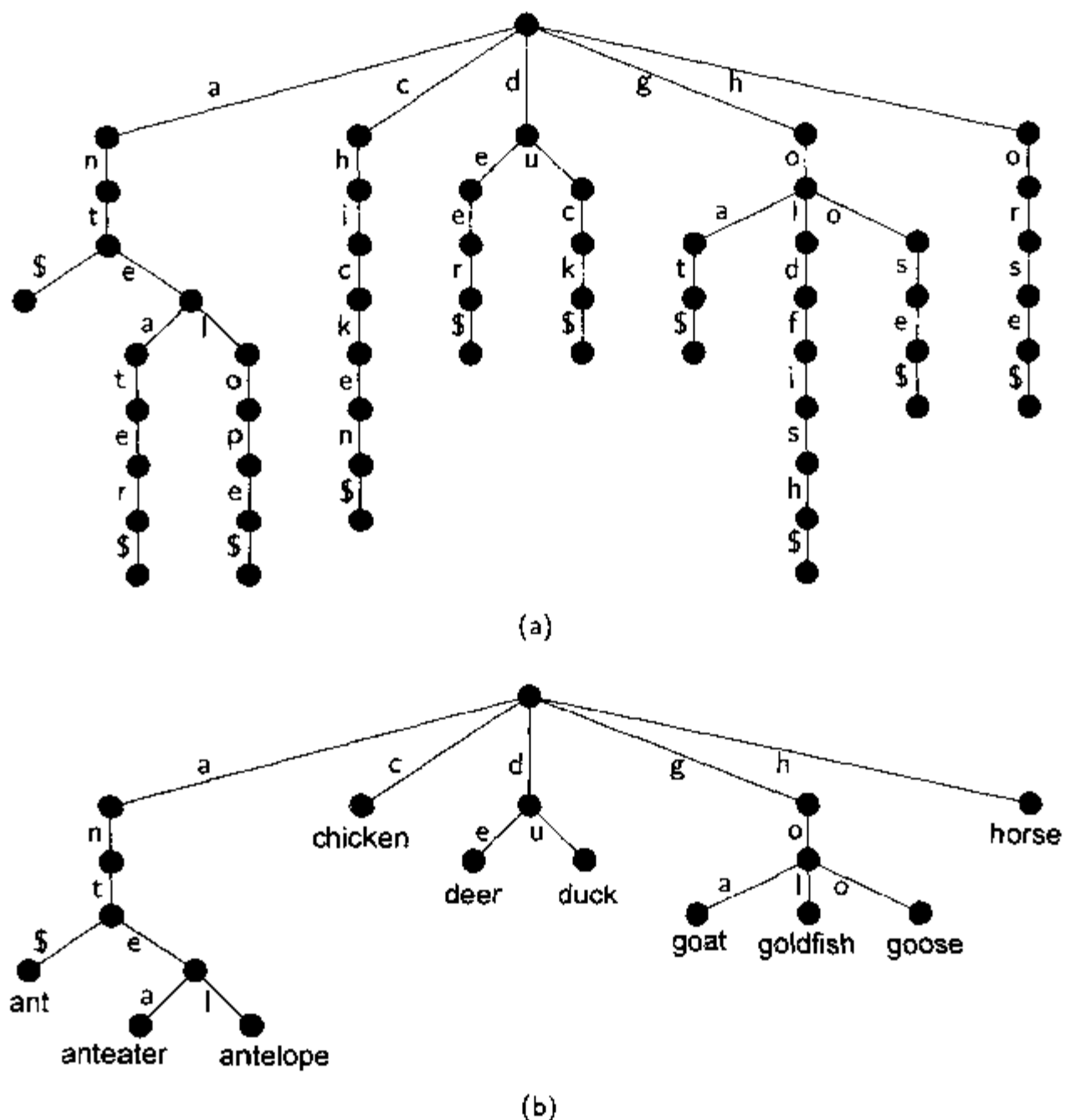


图 13.2 10 个单词的字母表 Trie 结构的两个变体

(a)每个结点包含对应于一个字母的一组链,而且这组单词中的每个字母都有一个对应的链。“\$”用于标识单词结束。内部结点用于引导检索,并且一个字母一个字母地拼出单词。不需要显式地存储单词。(b)这里对 Trie 结构进行足够的扩展,以分辨出两个单词。Trie 结构的每个叶结点存储一个完全的单词;内部结点只用于引导检索

如果大量使用某个前缀,就会导致平衡性差和结块等现象。例如,一个存储英语中常用单词的字母表 Trie 结构在树的“th”分支有许多单词,但是在“zq”分支就没有多少单词。

通过把一个 Trie 结构的字母表用等价的二进制编码代替,就可以把任何一个多路分支 Trie 结构用一个二叉 Trie 结构代替。另外,还可以使用 6.3.4 小节把一般树结构转换成二叉树结构的技术把一个一般的 Trie 结构转换成一个二叉 Trie 结构,而不必改变字母表。

图 13.1 和图 13.2 的 Trie 结构的实现可能非常低效,因为某些关键码集合可能产生大量只有一个子女的结点。D. Morrison 发明了 Trie 结构实现的一种变体,称为 PATRICIA,它表示“Practical Algorithm To Retrieve Information Coded In Alphanumeric”。如果字母表是二进制字母表, PATRICIA Trie 结构(以后称为 PAT Trie 结构)是一个完全二叉树,它把数据记录存储在叶结点,而使用内部结点存储关键码位模式中的位置,在以后的查询中通过这个位置决定进入哪个分支。图 13.3 中显示了对应于图 13.1 中值的 PAT Trie 结构。

例如,当在图 13.3 的 PAT Trie 结构中检索值 7(二进制值为 0000111)时,根结点标识首先检查位置 0 的位(最左边的位)。由于值 7 的第 0 位是 0,进入左边分支。在第 1 层,分支依

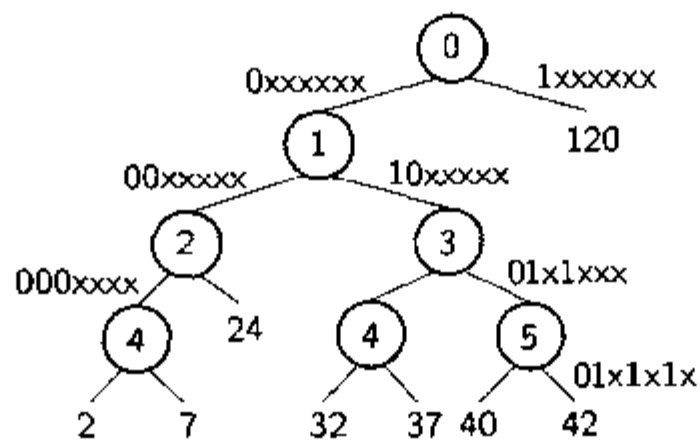


图 13.3 值( 2,7,24,31,37,40,42,120 )的 PAT Trie 结构 所有数据值都存储在叶结点中,而内部结点存储用于确定分支方向的位的位置。假定每一个关键码都用一个 7 位二进制值表示,这个值表示 0 到 127 之间的一个数。PAT Trie 结构中的一些分支已经被标号,以标识子树中所有值的二进制表示。例如,标号为 0 结点的左子树的所有值必须是值 0xxxxxx(其中 x 表示即可以为 0 也可以为 1 的位)。标号为 3 结点的右子树的所有结点一定有值 01x1xxx

赖于第 1 位的值,这个值也是 0。在第 2 层,分支依赖于第 2 位的值,这个值还是 0。在第 3 层,存储在结点中的索引是 4,这表示接下来检查关键码的第 4 位(第 3 位的值无关紧要,因为子树中的所有关键码值在第 3 位的位置都具有相同的值)。对于关键码值 7,这个位的值是 1,因此进入最右边分支。由于这样就进入了叶结点,把检索关键码与存储在那个结点中的关键码比较一下,如果它们匹配,就找到了需要的记录。

在检索过程中,每一个内部结点中只有关键码的 1 位参与比较。这是很重要的,因为检索关键码可能很大。PAT Trie 结构的检索只需要一次完全关键码比较,这只有在到达叶结点时才会发生。

## 13.2 伸展树

我们已经多次看到,BST 非常有可能变得不平衡,使得检索操作和更新操作的代价非常大。这个问题的一种解决方法是采用其他树形结构,例如 2-3 树;另一种选择是以某种方式修改 BST 的访问函数,以保证树能很好地工作。这是一个很有吸引力的想法,它对于堆结构确实做得很好,堆的访问函数使堆结构保持完全二叉树形状。但是,要求 BST 总是保持完全二叉树形状需要在更新操作时对树做极大的修改,这已在 11.3 节讨论过。改进 BST 性能的另一种方法是不必总是使树保持平衡,而是在每次访问时尽量使 BST 更加平衡。这种折衷的一个例子就是伸展树(Splay Tree)。

伸展树本身实际上不是一个数据结构,而是改进 BST 性能的一组规则。每当执行检索、插入、删除操作时,这些规则就控制对 BST 的修改。它们的目的是对完成一组操作需要的时间提供保证,从而避免标准 BST 操作在最差情况下的线性时间性能。伸展树不能保证每一个单个操作都是有效率的,但是,伸展树的访问规则保证对于一个有  $n$  个结点的树, $m$  个操作,当  $m \geq n$  时的代价为  $O(m \log n)$  时间。这样,一次插入或检索操作可能花费  $O(n)$  时间。然而, $m$  个这样的操作能够保证总共需要  $O(m \log n)$  时间,每次访问操作的平均代价为  $O(\log n)$ 。对于任何检索树结构,这是一个非常需要的性能保证。



伸展树访问函数的操作类似于 10.2 节自组织线性表的移至前端规则,还类似于 6.2 节管理父指针树的路径压缩技术。这些访问函数总体上趋向于使树更加平衡,但是单独一次访问不一定会使树更加平衡。

每当访问一个结点 S 时(例如,当 S 被插入、删除或者是检索目标时),伸展树就完成一次称为展开(splaying)的过程,展开处理把结点 S 移到 BST 的根结点。当要删除结点 S 时,展开过程把结点 S 的父结点移到根结点。结点 S 的一次展开包括一组旋转(rotation)。一次旋转调整结点 S 相对于其父结点和祖父结点的位置,把它移到树中的更高层。旋转有三种类型。

只有当结点 S 是根结点的子女时,才完成单旋转(single rotation)。单旋转在图 13.4 中说明,它基本上在保持 BST 特性的情况下把结点 S 和它的父结点交换位置。

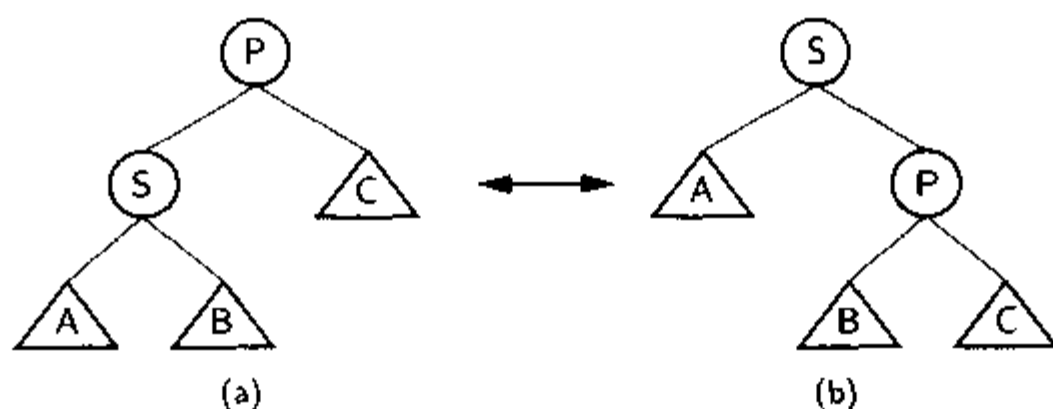


图 13.4 伸展树的单旋转。只有当被展开的结点是根结点的子女时才会发生这种旋转。这里,结点 S 被提升到根结点,与结点 P 一起旋转。由于结点 S 的值小于结点 P 的值,结点 P 必须成为结点 S 的右子女。子树 A、B 和 C 的位置相应地进行改变,以保持 BST 的特性,但是这些子树的内容仍然不变

(a)初始的树以结点 P 作为父结点。(b)旋转发生以后的树。完成两次单旋转就可以把树恢复到原来的形状。

同样,如果(b)是树的初始形状(即结点 S 是根结点,而结点 P 是它的右子女),那么(a)就是一次单旋转的结果

双旋转(double rotation)有两种类型。双旋转涉及到结点 S、结点 S 的父结点(称为 P)和结点 S 的祖父结点(称为 G)。双旋转的结果是把 S 在树中向上移两层。

第一种双旋转称为之字形旋转(zigzag rotation)。当出现以下两种情况之一时,就会发生之字形旋转:

1. 结点 S 是结点 P 的左子女,结点 P 是结点 G 的右子女。
2. 结点 S 是结点 P 的右子女,结点 P 是结点 G 的左子女。

也就是说,当结点 G、结点 P、结点 S 形成一个之字形时就进行之字形旋转。图 13.5 说明了之字形旋转。

另一种双旋转称为一字形旋转(zigzig rotation)。当出现以下两种情况之一时,就会发生一字形旋转:

1. 结点 S 是结点 P 的左子女,结点 P 是结点 G 的左子女。
2. 结点 S 是结点 P 的右子女,结点 P 是结点 G 的右子女。

这样,在之字形旋转不合适的环境下就会发生一字形旋转。图 13.6 说明了一字形旋转。

之字形旋转趋向于使树更平衡,因为它使子树 B 和 C 上升一层,而使子树 D 下降一层,结果经常是使树的高度减 1。一字形提升一般不会降低树的高度,它只是把新访问的记录向根结点移动。

展开结点 S 包括一系列双旋转,直到结点 S 到达根结点或根结点的子女。然而,如果需

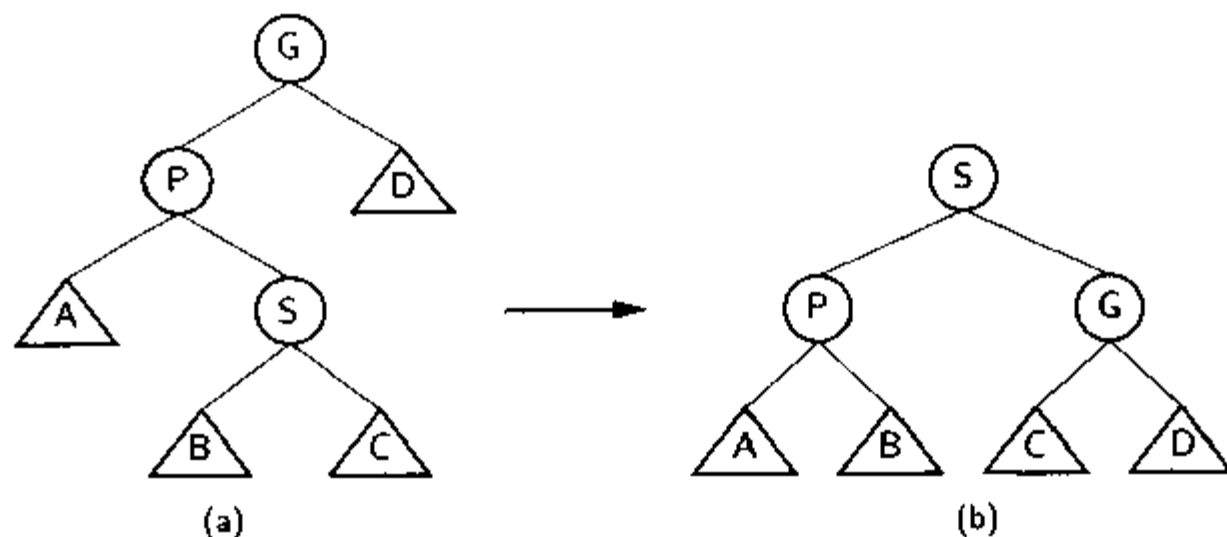


图 13.5 伸展树的之字形旋转

(a)初始结点 S、结点 P、结点 G 为之字形的树。(b)旋转发生之后的树。子树 A、B、C 和 D 的位置进行了相应改变,以保持 BST 的特性

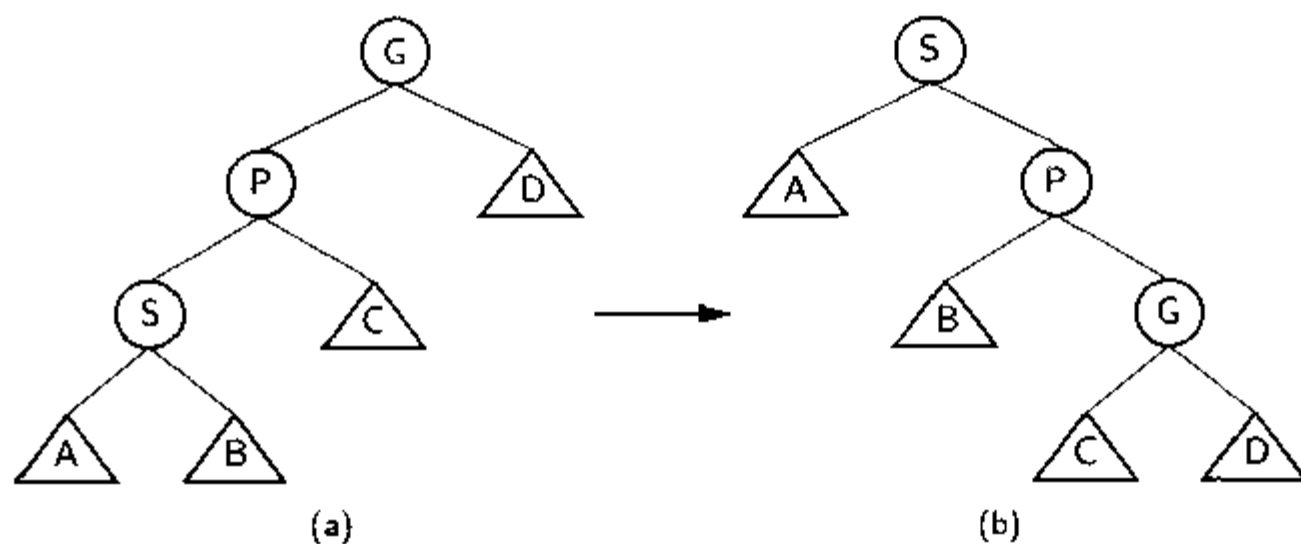


图 13.6 伸展树的一字形旋转

(a)初始结点 S、结点 P、结点 C 为一字形的树。(b)旋转发生之后的树。子树 A、B、C 和 D 的位置进行了相应改变,以保持 BST 的特性

要,一次单旋转也会使结点 S 成为根结点,这个过程趋向于使树重新平衡。在任何情况下,都会使访问最频繁的结点靠近树的顶层,从而减少访问代价。证明伸展树确实能够保证  $O(m \log n)$  时间超出了本书范围,更多的信息参见 13.4 节中的参考书目。

为了说明伸展树是如何进行操作的,考虑一下在图 13.7(a)的伸展树中检索值 89。在伸展树中检索值 89 与在 BST 中检索完全一样,然而,一旦找到了这个值,就把它通过展开移到根结点。在这个例子中需要三次旋转,第一次是一个之字形旋转,其结果如图 13.7(b)所示;第二次是一个之字形旋转,其结果如图 13.7(c)所示;最后一步是一个单旋转,其结果是图 13.7(d)中的树。注意展开过程使得树的层次变浅了。

除了结点一插入就展开到根结点以外,伸展树的插入操作与 BST 相同。同样,除了被删除结点的父结点展开到根结点以外,从伸展树中删除结点与从对应的 BST 中删除是一样的。

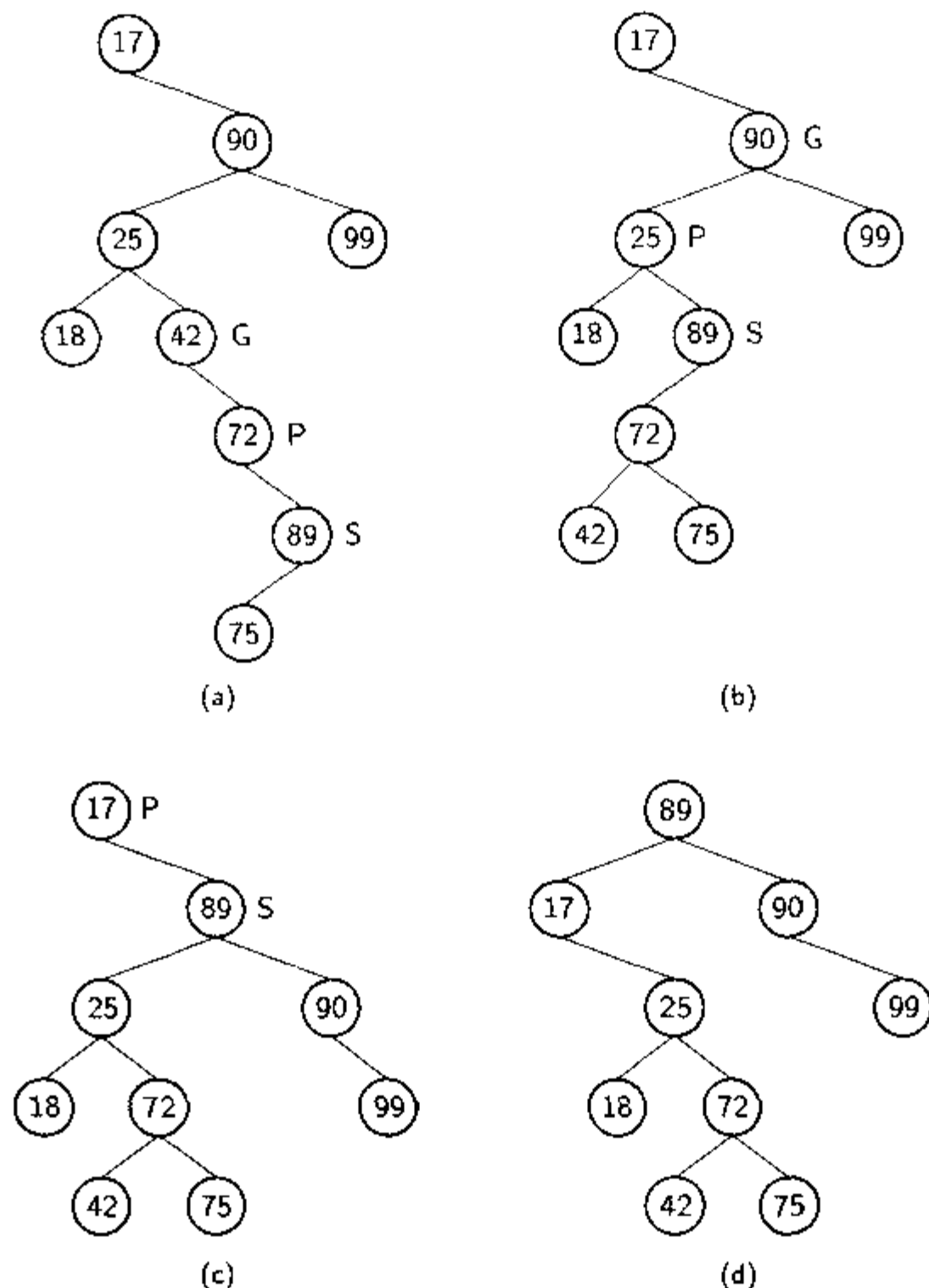


图 13.7 在伸展树中完成一次检索以后再展开的例子。在找到值为 89 的结点以后,通过完成三次旋转,这个结点就展开到根结点了

(a)初始的伸展树。(b)在(a)的树中对值为 89 的结点完成一次一字形旋转之后的结果。(c)在(b)的树中对值为 89 的结点完成一次之字形旋转之后的结果。(d)在(c)的树中对值为 89 的结点完成一次单旋转之后的结果

### 13.3 空间数据结构

到现在为止讨论的所有检索树 BST、伸展树、B 树和 Trie 结构,都被设计为用于检索一个一维关键码。典型的例子是整数关键码,它的一维范围可以直观地看作是一个数轴。有些数据库需要支持多个关键码,即可以根据多个关键码中的任何一个检索记录。一般说来,每个这样的关键码都有自己的索引或散列表,任何给定的检索查询都相应地检索这些独立索引中的一个或几个。

多维关键码检索提供了一个非常不同的概念。想像一下有个城市记录数据库,其中每个

城市都有一个名字和一个  $x, y$  坐标。BST 或者伸展树为基于城市名字的检索提供了很好的性能,可以用两个分开的 BST 来索引  $x$  坐标和  $y$  坐标。利用这些数据结构可以完成插入、删除城市操作,按照名字或者一个坐标定位城市的操作。然而,在一个二维空间检索两个坐标中的一个并不是一种自然的、看待检索的方式。另外一种选择是把  $x, y$  坐标结合成一个单一的关键词,就是说结合这两个坐标,根据结果关键词在 BST 中索引城市。这就允许根据坐标进行检索,但是不会考虑到有效的二维范围查询(range queries),例如检索在一个指定点的给定范围内的所有城市。问题是 BST 只对一维关键词工作得很好,而坐标是二维关键词,其中的一维并不比另一维更重要。

多维范围查询是空间应用程序(spatial application)的重要特性。由于一对坐标给出空间的一个位置,就把它称为一个空间属性(spatial attribute)。要有效地实现空间应用程序,就需要使用空间数据结构(spatial data structure)。空间数据结构存储根据位置组织的数据对象,是地理信息系统、计算机图形学、机器人学和许多其他应用中使用的一类重要数据结构。

这一节为存储二维或者更多维点数据提供了两个空间数据结构。它们是  $k$ -d 树和 PR 四分树(PR quadtree)。 $k$ -d 树是 BST 向多维的自然扩展,它是一个二叉树,其分支决策在关键词的各个维之间交替。像 BST 一样, $k$ -d 树使用对象空间分解。PR 四分树使用关键词空间分解,因此是一种 Trie 形式的结构,它只有在一维关键词的情况下才是一个二叉树(在这种情况下,它是一个有二进制字母表的 Trie 结构)。对于  $d$  维关键词,它有  $2^d$  个分支。这样,在二维情况下,PR 四分树有 4 个分支(因此得名“四分树”),实际上在每一个分支空间都被划分成 4 个大小相等的部分。

### 13.3.1 $k$ -d 树

$k$ -d 树是对 BST 的改进,从而能够对多维关键词进行更有效的处理。 $k$ -d 树不同于 BST 的地方在于  $k$ -d 树的每一层都根据这一层的某个特定检索关键词做出分支决策,这个检索关键词就称为识别器(discriminator)。对于  $k$  维关键词,在第  $i$  层把识别器定义为  $i \bmod k$ 。例如,假定存储的数据按照  $x, y$  坐标组织。在这里  $k$  是 2(有两个坐标), $x$  坐标字段指定为关键词 0,而  $y$  坐标指定为关键词 1。在每一层,识别器都在  $x, y$  之间交替,这样,第 0 层的一个结点  $N$ (根结点)把  $x$  值小于  $N_x$  的结点放到它的左子树中(因为  $x$  是检索关键词 0,而  $0 \bmod 2 = 0$ ),右子树中将包含  $x$  值大于  $N_x$  的结点。第 1 层的一个结点  $M$  将把  $y$  值小于  $M_y$  的结点放到它的左子树中,而对  $M_x$  和结点  $M$  的后继结点的  $x$  值则没有限制,因为分支决策只根据  $y$  坐标作出。图 13.8 给出了一个例子,说明一组二维点怎样存储在一个  $k$ -d 树中。

在图 13.8 中,包含点的区域限制到  $100 \times 100$  的方块中(这个划分方式是任意选择的),而每一个内部结点都对检索空间进行划分。每个划分用一条线显示,竖线用于使用  $x$  作为识别器的结点,横线用于使用  $y$  作为识别器的结点。根结点把空间划分成两部分,它的子女进一步把空间划分成更小的部分。子女的划分线不会穿过根结点的划分线,这样, $k$ -d 树中的每个结点都有助于把空间分解为矩形,显示结点可能落到的各个子树的范围。

除了树的每一层与某个识别器相关以外,在  $k$ -d 树中检索一个指定  $x, y$  坐标的记录就像在 BST 中检索一样。考虑一下在  $k$ -d 树中检索一个位于点  $P = (69, 50)$  的记录。首先把  $P$  与存储在根结点的点(图 13.8 中是记录 A)比较。如果  $P$  的位置匹配记录 A 的位置,那么检索就成功了。在这个例子中位置不匹配(A 的位置  $(40, 45)$  不是  $(69, 50)$ ),检索必须进入更深

的层。把 A 的  $x$  值与 P 的  $x$  值比较,确定进入哪个分支。由于 A 的  $x$  值 40 小于 P 的  $x$  值 69,进入右边的子树( $x$  值大于或等于 40 的所有城市都在右子树中),A 的  $y$  值在这一层不影响进入哪一条路径的决策。在第二层,P 不匹配记录 C 的位置,因此必须选择另一个分支。然而,在这一层根据 P 和 C 的  $y$  值决定进入哪一个分支(因为  $1 \bmod 2 = 1$ ,对应于  $y$  坐标)。由于 C 的  $y$  值 10 小于 P 的  $y$  值 50,进入右边分支。在这一点,P 与位置 D 比较。匹配正确,从而检索成功。就像在 BST 中一样,如果检索过程到达一个 null 指针,那么检索的点就不包含在树中。下面是 k-d 树检索的一个实现,相当于 BST 类中的 findhelp 函数。其中变量  $D$  存储关键码的维数。

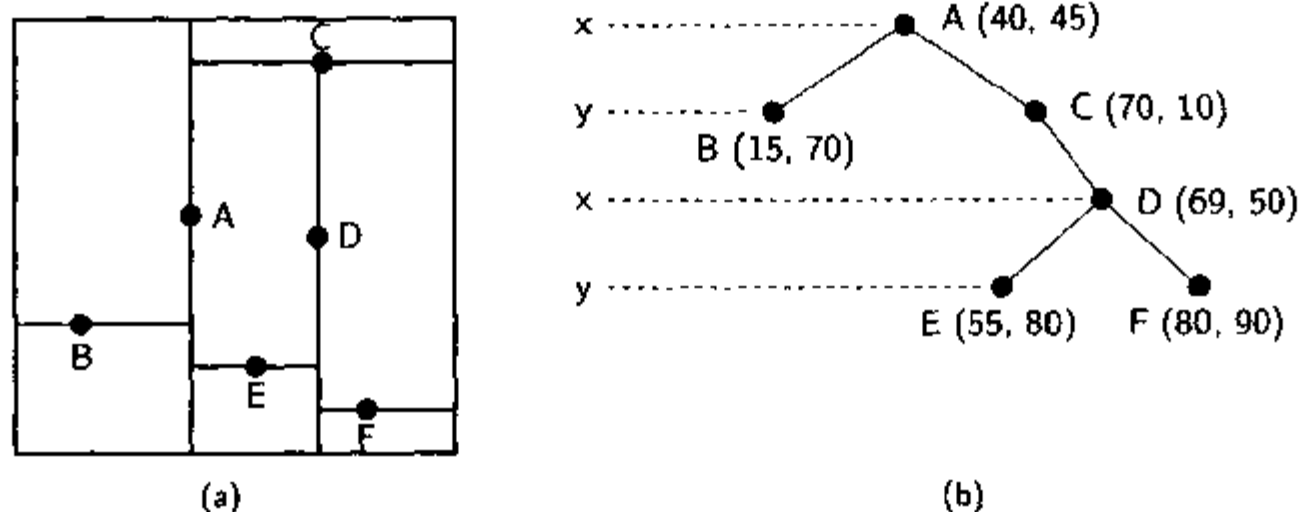


图 13.8 k-d 树的例子

(a)k-d 树对一个包含 7 个数据点的  $100 \times 100$  单元区域的分解。(b)区域(a)的 k-d 树

```
private DElem findhelp(BinNode rt, int[] key, int level) {
    if (rt == null) return null;
    DElem it = (DElem)rt.element();
    if (it.equalKey(key)) return (DElem)rt.element();
    if (it.key[level] > key[level])
        return findhelp(rt.left(), key, (level + 1) % D);
    else
        return findhelp(rt.right(), key, (level + 1) % D);
}
```

向 k-d 树插入新结点类似于向 BST 插入新结点。首先进行 k-d 树的检索过程,直到找到一个 null 指针,标识插入新结点的合适位置。例如,要在图 13.8 的 k-d 树中位置 (10, 50) 插入一条记录,首先需要进行一次检索,到达包含记录 B 的结点。在这一点,新记录插入到 B 的左子树中。

从 k-d 树中删除结点类似于从 BST 中删除结点,但是稍微困难一些。就像从 BST 中删除一样,第一步是找到要删除的结点(称它为 N),然后需要找到结点 N 的一个子女,在树中用它代替结点 N。如果结点 N 没有子女,那么用一个 null 指针代替结点 N;如果结点 N 有一个子女,而这个子女还有自己的子女,就不能像在 BST 中那样使结点 N 的父结点指向结点 N 的子女。这样做会改变子树中所有结点的层次,从而用于检索的识别器也就会改变,结果是这个子树不再是一个 k-d 树了,因为一个结点的子女现在可能在识别器方面破坏了 BST 特性。

类似于 BST 的删除,存储在结点 N 中的记录要么被结点 N 的右子树中具有结点 N 识别

器的最小值记录代替,要么被结点 N 的左子树中具有结点 N 识别器的最大值记录代替。假定结点 N 在奇数层,因而  $y$  是识别器,那么结点 N 就被它的右子树中最小  $y$  值(称它为  $Y_{\min}$ )记录代替。问题是  $Y_{\min}$  不一定是最左边的结点,而在 BST 中则是最左边的结点。这样,就必须使用一个改进的检索过程在左子树中找到最小  $y$  值。下面实现了这个查找最小值的检索:

```
private DElem findmin(BinNode rt, int descrim, int level) {
    DElem temp1, temp2;
    if (rt == null) return null;
    temp1 = findmin(rt.left(), descrim, (level - 1) % D);
    if (descrim != level) {
        temp2 = findmin(rt.right(), descrim, (level + 1) % D);
        if ((temp1 == null) || ((temp2 != null) &&
            (temp1.key(descrim) > temp2.key(descrim))))
            temp1 = temp2;
    } // Now, temp1 has the smaller value
    if ((temp1 == null) || (temp1.key(descrim) >
        ((DElem)rt.element()).key(descrim)))
        return (DElem)rt.element();
    else
        return temp1;
}
```

对删除过程的递归调用就会从树中清除  $Y_{\min}$ 。最后,  $Y_{\min}$  的记录代替了结点 N 的记录。

只有右子树存在的时候,才可以用右子树中的最小值结点代替要删除的结点。如果右子树不存在,那么就要在左子树中找到一个合适的替代者。然而,使用左子树中相应识别器具有最大值的记录代替结点 N 并不令人满意,因为这个新值可能重复。如果是这样,那么在结点 N 左子树中对于识别器将具有相等的值,这将违反 k-d 树的排序规则。幸好,对这个问题有一个简单的解决方法。首先移动结点 N 的左子树,使其成为右子树(即,简单交换结点 N 的左右子女指针值),这时,就可进行正常的删除过程,使用包含当前结点 N 的右子树中识别器最小值记录,代替要删除的结点 N 的记录。

假定要找出在一个指定点 P 的某个距离  $d$  范围内的所有记录。这里使用欧氏距离(Euclidian distance),即如果:

$$\sqrt{(P_x - N_x)^2 + (P_y - N_y)^2} \leq d$$

就定义点 P 在点 N 的距离  $d$  之内。

如果检索过程到达一个结点,这个结点识别器的关键码值比检索关键码中相应的值多出  $d$ ,那么右子树中的任何记录都不可能在检索关键码距离  $d$  之内,因为这一维中的所有关键码值都太大了。类似地,如果当前结点识别器关键码值比检索关键码值小  $d$ ,那么左子树中也没有记录会在范围内。在这些情况下,不需要检索有问题的子树,从而节省了大量时间。一般说来,在范围查询期间必须访问的结点数与在查询半径内的数据记录数有关。

例如,假如要进行一次检索,在图 13.8 的 k-d 树中找到与点(25,65)距离小于 25 个单位的所有城市。检索从根结点开始,根结点中包含记录 A。由于(40,45)距离检索点正好是 25

个单位,应当把它报告出来。然后,检索过程确定要进入树的哪一个分支。检索范围扩展到 A 的(垂直)划分线的左边和右边,所以树的两个分支都要检索,先处理左边的子树。这里,检查记录 B,发现它在检索范围内。由于存储记录 B 的结点没有子女,左子树的处理就完成了。现在开始处理 A 的右子树。检查记录 C 的坐标,发现它不在范围内,这样就不报告它。然而,即使记录 C 不在范围内,C 的子树中的城市也有可能在范围内。因为记录 C 在第一层,这一层的识别器是  $y$  坐标。由于  $65 - 25 > 10$ ,记录 C 左子树中的记录(即大于 C 的记录)不可能在检索范围内,这样,就不需要检索 C 的左子树(如果有)了。但是,C 的右子树中的城市可能在检索范围内,这样,检索进行到包含记录 D 的结点。同样,D 在检索范围之外。由于  $25 + 25 \times 69$ ,记录 D 的右子树中没有记录可能在检索范围内,这样,就只需要检索记录 D 的左子树了。这就需把记录 E 的坐标与检索范围比较。记录 E 在检索范围之外,处理结束。下面是区域检索函数的一个实现:

```
private void rshelp(BinNode rt, int[] point, int radius, int lev) {
    if (rt == null) return;
    if (!inCircle(point, radius, ((DElem)rt.element()).coord()))
        System.out.println(rt.element());
    if (((DElem)rt.element()).key(lev) > (point[lev] - radius))
        rshelp(rt.left(), point, radius, (lev+1) % D);
    if (((DElem)rt.element()).key(lev) < (point[lev] + radius))
        rshelp(rt.right(), point, radius, (lev+1) % D);
}
```

当访问一个结点时,使用函数 `InCircle` 检查结点记录和查询点之间的欧氏距离。简单地检查  $x$  坐标之间和  $y$  坐标之间的差是否小于查询距离是不够的,因为记录仍然可能在查询范围之外,如图 13.9 所示。

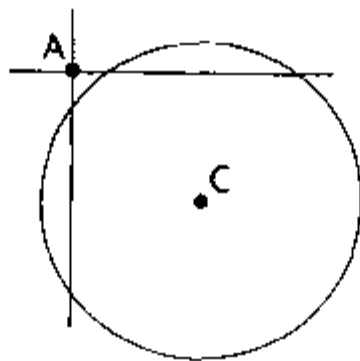


图 13.9 函数 `InCircle` 必须检查记录和查询点之间的欧氏距离。有可能记录 A 的  $x$  坐标和  $y$  坐标都在查询点 C 的查询距离之内,而记录 A 本身却在查询范围之外

### 13.3.2 PR 四分树

点-区域四分树(Point-Region Quadtree)(以后称为 PR 四分树)是一个树形结构,其结点要么正好有 4 个子女,要么是 1 个叶结点(即它在形状上是一个完全四叉[4-ary]树)。PR 四分树代表二维平面上的一组数据点,它把包含这些数据点的区域四等分,每个小区域再四等分,依次下去,直到叶结点不再包含多于一个点。也就是说,如果一个区域包含 0 个或者 1 个数据点,那么就用由一个叶结点组成的 PR 四分树表示它。如果区域包含多于一个数据点,那么就

把这个区域四等分。相应的 PR 四分树就包含一个内部结点和四个子树,每个子树代表区域的四分之一,可能会继续对其四等分。PR 四分树的每个内部结点代表对二维区域的一个划分。区域的四个部分(或者等价地,相应子树)被依次称为 NW、NE、SW 和 SE。对每个包含多个点的部分继续进行四等分,直到 PR 四分树的每个叶结点最多包含一个点。

例如,考虑图 13.10(a)的区域和图 13.10(b)对应的 PR 四分树。分解过程要求有一个固定的关键码范围,在这个例子中,区域被假定是  $128 \times 128$  大小。PR 四分树的内部结点只用于标识区域的分解,不存储数据记录。

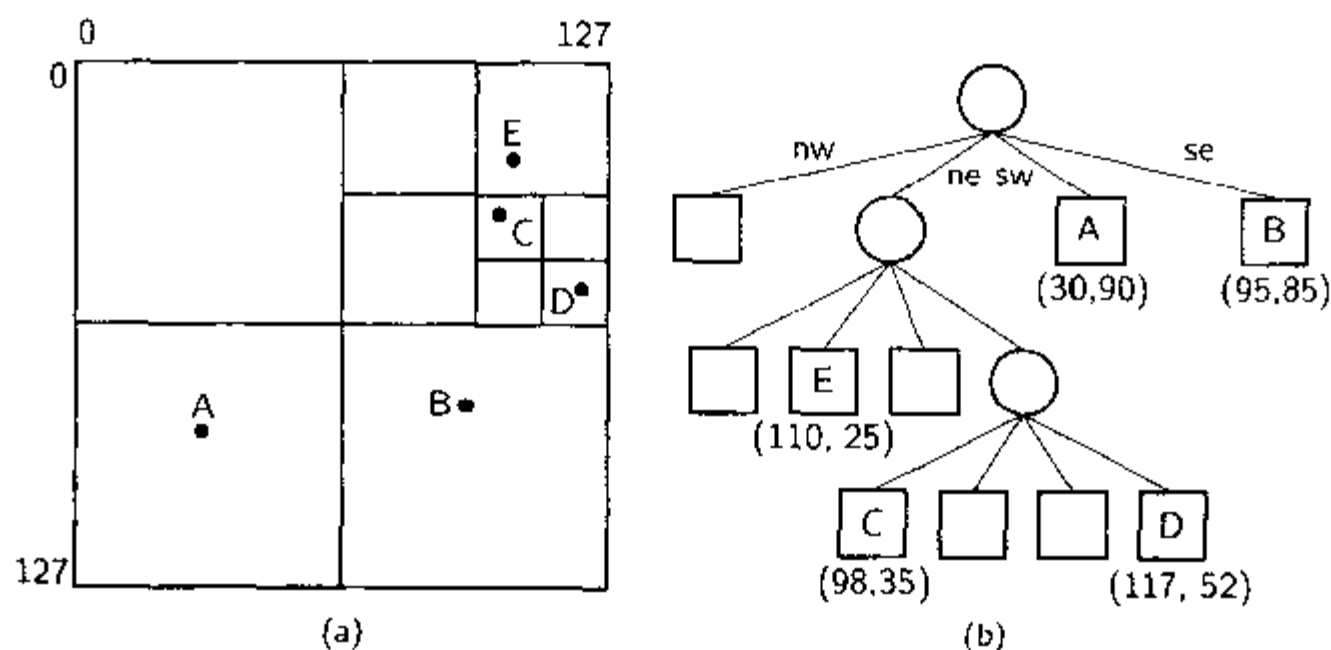


图 13.10 一个 PR 四分树的例子

(a)数据点的图。把区域定义成正方形,原点在左上角,边长为 1024。

(b)(a)中点的 PR 四分树。(a)还显示了 PR 四分树对这个区域进行的块分解

在 PR 四分树中检索一个匹配点  $Q$  的记录很直接。从根结点开始,不断进入包含  $Q$  的部分,直到到达叶结点。如果根结点就是叶结点,那么只要检查结点的数据记录是否匹配点  $Q$ 。如果根结点是内部结点,继续进行下去,到达包含检索坐标的子女。例如, NW 部分包含的点的  $x$  值和  $y$  值都在 0 到 63 之间。NE 部分包含点的  $x$  值在 64 到 127 之间,而  $y$  值在 0 到 63 之间。如果根结点的子女是叶结点,那么就检查这个子女,看看是否能够找到  $Q$ 。如果这个叶结点是另一个内部结点,就继续对树进行检索,直到找到一个叶结点。如果这个叶结点存储一条记录,这条记录的位置与点  $Q$  匹配,那么检索就成功了,否则点  $Q$  就不在树中。

要把记录  $P$  插入 PR 四分树中,首先要找到包含  $P$  的位置的叶结点。如果这个叶结点是空的,那么就把  $P$  存储在这个叶结点中;如果这个叶结点已经包含  $P$  (或者一个具有  $P$  的坐标的记录),那么就要报告记录重复;如果叶结点已经包含另一条记录,那么就必须继续分解这个结点,直到已存在的记录和  $P$  分别进入不同的结点。图 13.11 显示了这样一次插入过程的例子。

要删除一条记录  $P$ ,首先要找到 PR 四分树中包含记录  $P$  的结点  $N$ ,然后把结点  $N$  改为空。接下来看一看结点  $N$  的三个兄弟结点,如果它们中只包含一个点,那么必须把结点  $N$  和它的兄弟结点结合起来,形成一个结点  $N'$ 。这个合并过程持续下去,直到到达某一层,在这一层中至少有两个点包含在结点  $N'$  和它的兄弟结点表示的子树中。例如,如果要从图 13.11(b)表示的 PR 四分树中删除点  $C$ ,结果结点必须与它的兄弟结点合并,这个更大的结点再与其兄弟结点合并,并把 PR 四分树恢复到图 13.11(a)的分解。



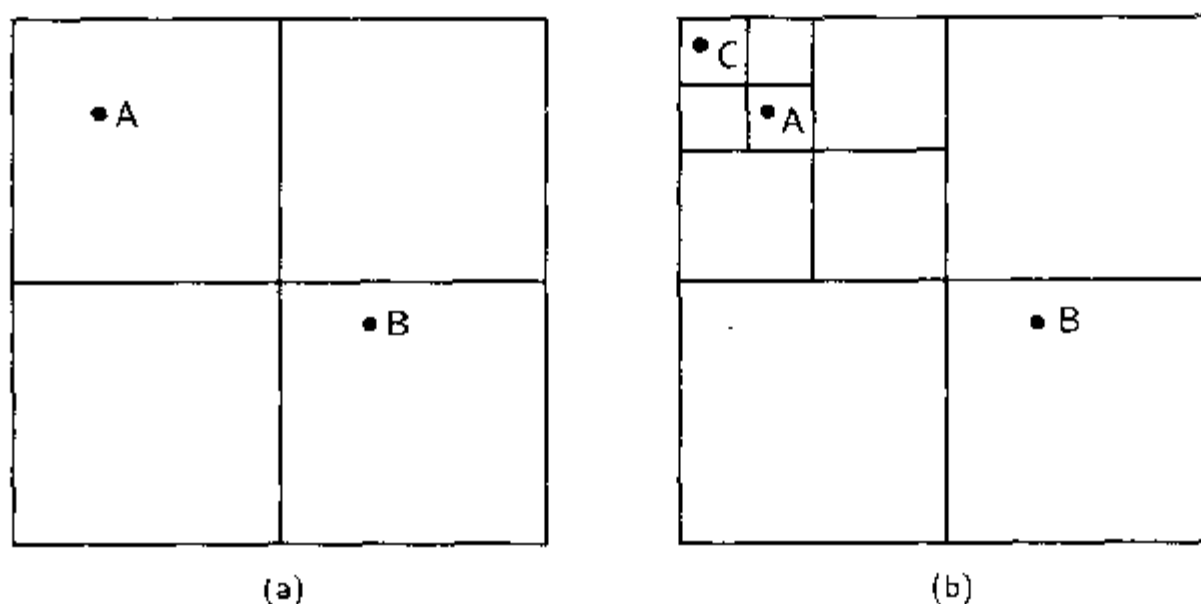


图 13.11 PR 四分树插入的例子

(a)初始 PR 四分树包含两个数据点。(b)插入点 C 后的结果。包含点 A 的块必须被分成 4 个子块。如果只进行一次分解,点 A 和点 C 将仍然在同一个块中,因此需要第二次分解,把它们分开

对 PR 四分树进行区域检索很容易。要找到以查询点  $Q$  为中心,半径为  $r$  的范围内的所有点,从根结点开始进行。如果根结点是一个空叶结点,那么就找不到数据点;如果根结点是包含一个数据记录的叶结点,那么就检查这个数据点的位置,确定它是否在范围内;如果根结点是一个内部结点,那么就递归地完成这个过程,但是只对那些包含检索范围某些部分的子树进行。

### 13.3.3 其他空间数据结构

k-d 树和 PR 四分树之间的区别说明了实现空间数据结构遇到的许多问题。k-d 树提供了区域的对象空间分解,而 PR 四分树提供了关键码空间分解;k-d 树在所有结点中都存储记录,而 PR 四分树只在叶结点中存储记录。最后,这两种树形结构在结构上是完全不同的。k-d 树是二叉树,而 PR 四分树有  $2^d$  个分支(在二维情况下,  $2^2 = 4$ )。考虑一下把这种概念扩展到三维的情况。一个三维 k-d 树将把  $x$  坐标、 $y$  坐标和  $z$  坐标交替作为识别器,对应于 PR 四分树的三维情况有  $2^3$  即 8 个分支。这样一棵树称为八分树(octree)。

根据这些思想,也可设计出一个基于二维关键码空间分解的二叉 Trie 结构,或者一个与二维情况等价的、使用对象空间分解的四叉树。二分树(Bintree)是一个类似于 k-d 树,但在每一层交替识别器的二叉 Trie 结构。图 13.8 中的点的二分树如图 13.12 所示。另外,还可以使用以数据点为中心的四路空间分解,这样的分解得到的树称为点四分树(Point quadtree)。图 13.10 中数据点的点四分树如图 13.13 所示。

本节只是简单地涉及了空间数据结构领域。现在已经发明了多种有趣的空间数据结构,其中许多都有变体和不同实现。有些最有趣的开发是与基于磁盘的应用程序采用空间数据结构有关的。重要的是要注意,所有这些基于磁盘的实现都归结为 B 树或散列方法的变体。

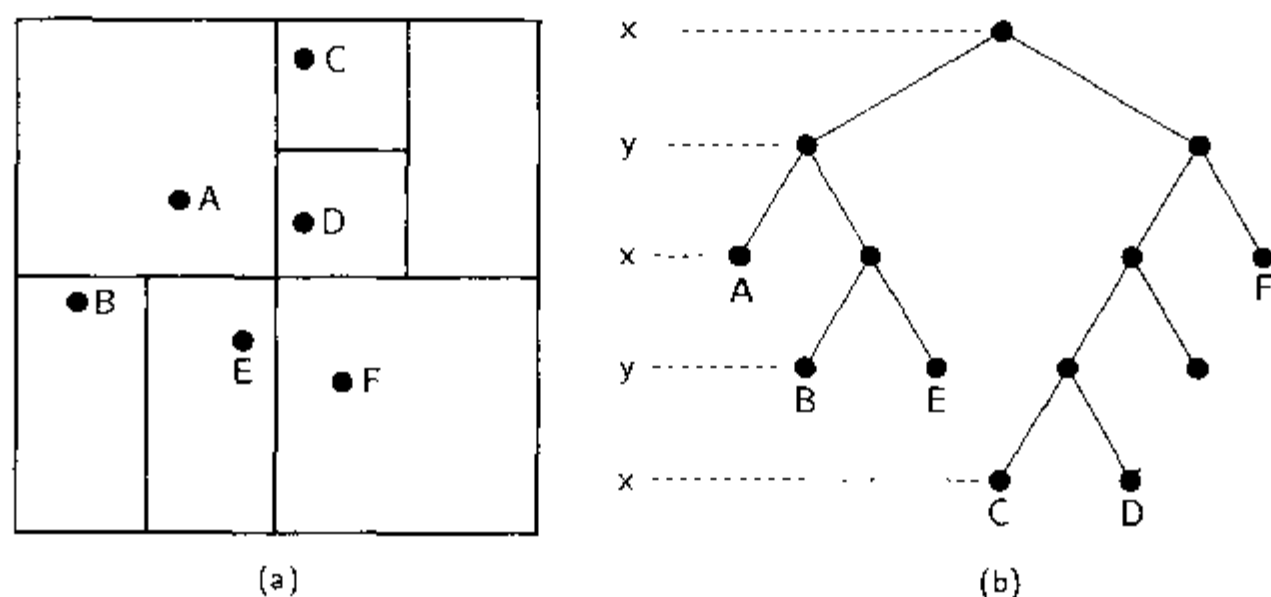


图 13.12 一个二分树的例子。二分树是使用关键码空间分解和识别器在各个维之间交替的一个二叉树,把它与图 13.8 的 k-d 树进行比较

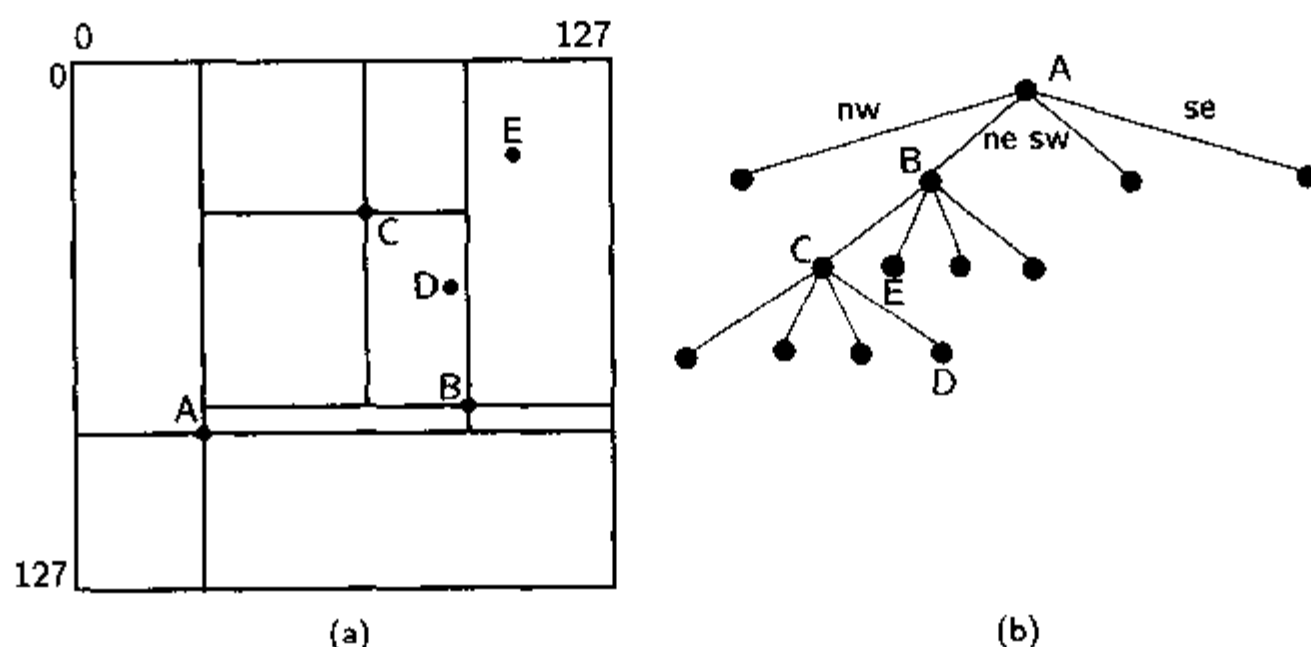


图 13.13 一个点四分树的例子,使用对象空间分解的四叉树。把它与图 13.10 中的 PR 四分树比较一下

## 13.4 深入学习导读

在 Frakes 和 Baeza-Yates 等的《Information Retrieval: Data Structure Algorithms》中讨论了 PATRICIA Trie 结构和其他 Trie 结构的实现[FBY92]。

有关伸展树的进一步阅读,参见 Sleator 和 Tarjan 的“Self-adjusting Binary Search”[ST85]。伸展树的一种替代选择是 AVL 树。AVL 树使用另一组旋转,保证 BST 在所有的時候都是平衡的。有关 AVL 树的讨论参见 Knuth[Knu73]。

空间数据结构领域非常丰富,而且发展很快。要得到这方面的好的介绍,参见 Hanan Samet 的两本书,《Applications of Spatial Data Structures》和《Design and Analysis of Spatial Data Structures》[Sam90a, Sam90b]。有关 PR 四分树的最好的参考书也是[Sam90b]。k-d 树是由 John Louis Bentley 发明的。有关 k-d 树的进一步信息,除了[Sam90b]以外,还可以参见[Ben75]。

有关两路与多路分支的相对空间需求的讨论,参见 Shaffer、Juvvadi 和 Heath 的“A gener-

alized Comparison of Quadtree and Bintree Storage Requirements” [SJH93]。

## 13.5 习题

- 13.1 对于下面一组值:42,12,100,10,50,31,7,11,99,给出它们的二叉 Trie 结构(如图 13.1 所示)。
- 13.2 对于下面一组值:42,12,100,10,50,31,7,11,99,给出它们的 PAT Trie 结构(如图 13.3 所示)。
- 13.3 编写 Trie 结构的插入例程。
- 13.4 编写 Trie 结构的删除例程。
- 13.5 给出在图 13.7(d)的伸展树中检索值 75 得到的伸展树。
- 13.6 给出在图 13.7(d)的伸展树中检索值 18 得到的伸展树。
- 13.7 给出图 13.10 中点的 k-d 树,这些点按照字母顺序插入。
- 13.8 给出图 13.8 中点的 PR 四分树,这些点按照字母顺序插入。
- 13.9 当在一个 PR 四分树中完成一次区域检索时,只需要检索一个内部结点的这样一些子树,这些子树对应的区域在查询范围之内。通过把查询范围中  $x$ 、 $y$  的范围与对应子树区域的  $x$ 、 $y$  范围比较,就可以很容易地计算出哪些子树的区域在查询范围内。然而,如图 13.9 所示, $x$ 、 $y$  的范围可能重叠,而查询范围与区域实际上却不交错。编写一个函数,准确地确定一个范围是否与一个区域交错。
- 13.10 给出图 13.10 中点的二分树,这些点按照字母顺序插入。
- 13.11 给出图 13.8 中点的点四分树,这些点按照字母顺序插入。

## 13.6 项目设计

- 13.1 使用 Trie 数据结构设计一个程序,对变长字符串排序,程序所做的工作与所有字符串中字母总数要成比例。有些字符串可能很长,而大多数则很短。
- 13.2 一个字符串的后缀字符串(suffix string)集合包括这个字符串本身,没有第 1 个字符的字母串,没有前 2 个字符的字符串,等等。例如,“HELLO”的后缀字符串集合是:

| HELLO,ELLO,LLO,L,O,O |

一个后缀树(suffix tree)是对于一个给定字符串,包含它的所有后缀字符串的 PAT Trie 结构。后缀树的好处是它允许使用通配符对字符串进行检索,例如,检索关键词“TH\*”表示找到所有以“TH”作为前两个字符的字符串,这可以很容易地使用一个正规 Trie 结构实现。在一个正规 Trie 结构中检索“\*TH”效率很低,但是在一个后缀树中却是高效的。对于一个词典中的单词或者短语实现后缀树。

- 13.3 改进 5.5 节的 BST 类,使其使用伸展树的旋转。你的新实现不应当改变原来 BST 类的 ADT(抽象数据类型)。
- 13.4 通过输入大量数据,比较标准 BST 的实现与伸展树的实现。在什么情况下伸展树实际上能够节省时间?

- 13.5 使用 k-d 树实现一个城市数据库,每个数据库记录都包含城市的名字(一个任意长度的字符串)和使用整数  $x$  和  $y$  表示的城市坐标。你的数据库应当允许记录的插入,根据记录名字或坐标删除,以及根据名字或坐标检索,还应当支持区域查询,即请求打印在一个指定点的给定距离范围内的所有记录。
- 13.6 使用 PR 四分树实现一个城市数据库,每个数据库记录都包含城市的名字(一个任意长度的字符串)和使用整数  $x$  和  $y$  表示的城市坐标。你的数据库应当允许记录的插入,根据记录的名字或坐标删除,以及根据名字或坐标检索,还应当支持区域查询,即请求打印在一个指定点给定距离范围内的所有记录。
- 13.7 使用二叉树实现一个城市数据库,每个数据库记录都包含城市的名字(一个任意长度的字符串)和使用整数  $x$  和  $y$  表示的城市坐标。你的数据库应当允许记录的插入、根据记录的名字或坐标删除,以及根据名字或者坐标检索,还应当支持区域查询,即请求打印在一个指定点给定距离范围内的所有记录。
- 13.8 使用点四分树实现一个城市数据库,每个数据库记录都包含城市的名字(一个任意长度的字符串)和使用整数  $x$  和  $y$  表示的城市坐标。你的数据库应当允许记录的插入、根据记录的名字或坐标删除,以及根据名字或坐标检索,还应当支持区域查询,即请求打印在一个指定点给定距离范围内的所有记录。
- 13.9 使用 PR 四分树实现问题 6.5 的一个有效的解决方法,即在一个 PR 四分树中存储点的集合。对于每一个点,使用 PR 四分树找出在距离  $D$  之内的应当相等的那些点。这个解决方法的渐进复杂性是什么?
- 13.10 选择这一章描述的任何两个点表示方法(即从 k-d 树、PR 四分树、二叉树和点四分树中选择两个)。实现你的两种选择,并通过大量数据集合对其进行比较,描述一下哪种方法更容易实现,哪种方法在空间上更有效,哪种方法在时间上更有效。

## 第 14 章 分析技术

本书包括许多算法的时间需求、数据结构的空间需求的渐进分析的例子。它经常只是简单地给出一个方程,对问题中的算法或数据结构的行为建模,然后,如果方程中包含递归或累加,就推导出方程的一个闭合形式解。有时候一个分析证明起来更困难,需要聪明的深入洞察才能推导出正确的模型,例如分析置换选择(9.7 节)平均长度的扫雪机类比。

来源于扫雪机类比的方程很简单。在其他情况下,开发模型是很直接的,但是分析结果方程却不是,一个例子就是快速排序的平均情况分析。8.4 节给出的方程只对关键位置列出了所有可能的情况,把对快速排序递归调用对应的代价累加起来。然而,对于结果递归关系推导出闭合形式的解法并不容易。

许多迭代算法需要计算出一个总和,以确定循环的代价,14.1 节提供了找到求和的闭合形式解法的技术。许多基于递归的算法的时间需求可以通过递归关系很好地进行建模,14.2 节给出了解决递归问题技术的一个简要介绍。

14.3 节介绍了均摊分析(amortized analysis),均摊分析处理这组操作的代价。可能这一组操作中单个操作的代价很高,但其余操作的代价得到了限制,以致于整个这组操作可以有效地完成。均摊分析已经成功地用于分析本书提供的许多算法,包括这组 UNION/FIND 操作的代价(6.2 节),一组伸展树操作的代价(13.2 节)和一组自组织线性表操作的代价(10.2 节)。14.3 节将详细讨论这个主题。

### 14.1 求和技术

本节提供一些基本技术,推导出求和的闭合形式解(也称为“解决”求和问题)。第一种方法是“猜试法”,适用于闭合形式解是多项式表达式的求和。

**例 14.1** 考虑一下这个熟悉的求和  $\sum_{i=0}^n i$ ,很明显,它小于  $\sum_{i=0}^n n$ ,其和是  $n^2 + n$ 。这样,有理由猜测这个求和的闭合形式解是一个如  $c_1 n^2 + c_2 n + c_3$  的多项式形式,其中  $c_1, c_2, c_3$  为 3 个常数。如果是这种情况,就可以对简单的情况进行求和,得到系数。对于这个例子,把  $n$  代入 0、1、2 得到 3 个联立方程。由于  $n = 0$  的求和是 0,  $c_3$  一定是 0。对于  $n = 1$  和  $n = 2$ ,得到下面两个方程:

$$c_1 + c_2 = 1$$

$$4c_1 + 2c_2 = 3$$

从而得到  $c_1 = 1/2, c_2 = 1/2$ 。如果求和的闭合形式解是一个多项式,那么它只能是:

$$1/2 n^2 + 1/2 n + 0$$

一般写成:

$$\frac{n(n+1)}{2}$$

既然我们已经有了一个候选的闭合形式解,就可以使用数学归纳法来验证它是否正确。

在这个例子中它确实是正确的,如例 2.2 所示。

只要答案是一个多项式表达式,“猜试”方法就可以使用。特别是,类似推理可以用于解决  $\sum_{i=1}^n i^2$ , 或者更一般的对于任何正整数  $c$  的  $\sum_{i=1}^n i^c$ 。

解决求和问题更一般的方法称为移位方法(shifting method),移位方法从求和的变差中减去求和,变差的选择应当能够使得大多数项消去。

**例 14.2** 移位相加解决求和问题的第一个例子是:

$$F(n) = \sum_{i=0}^n ar^i = a + ar + ar^2 + \cdots + ar^n$$

这称为几何级数。我们的目标是为  $F(n)$  找到某个变差,使得从一个减去另一个能够得到一个易于操作的方程。由于求和的各项之间的差异是因数  $r$ ,如果把整个表达式都乘以  $r$ ,就可以移项。

$$rF(n) = r \sum_{i=0}^n ar^i = ar + ar^2 + ar^3 + \cdots + ar^{n+1}$$

从一个方程减去另一个方程,如下所示:

$$\begin{aligned} F(n) - rF(n) &= a + ar + ar^2 + ar^3 + \cdots + ar^n \\ &\quad - (ar + ar^2 + ar^3 + \cdots + ar^n) - ar^{n+1} \end{aligned}$$

结果只留下最后一项:

$$\begin{aligned} F(n) - rF(n) &= \sum_{i=0}^n ar^i - r \sum_{i=0}^n ar^i \\ (1-r)F(n) &= a - ar^{n+1} \end{aligned}$$

这样,我们就得到结果:

$$F(n) = \frac{a - ar^{n+1}}{1-r}$$

其中  $r \neq 1$ 。

**例 14.3** 移位方法的第 2 个例子是解决:

$$F(n) = \sum_{i=1}^n i2^i = 1 \cdot 2^1 + 2 \cdot 2^2 + 3 \cdot 2^3 + \cdots + n \cdot 2^n$$

通过乘 2 就可以把项消去:

$$2F(n) = 2 \sum_{i=1}^n i2^i = 1 \cdot 2^2 + 2 \cdot 2^3 + 3 \cdot 2^4 + \cdots + (n-1) \cdot 2^n + n \cdot 2^{n+1}$$

$2F(n)$  的第  $i$  项是  $i \cdot 2^{i+1}$ , 而  $F(n)$  的第  $i+1$  项是  $(i+1) \cdot 2^{i+1}$ , 从一个表达式减去另一个表达式就可以得到  $2^i$  的和与一些消不去的项。

$$\begin{aligned} 2F(n) - F(n) &= 2 \sum_{i=1}^n i2^i - \sum_{i=1}^n i2^i \\ &= \sum_{i=1}^n i2^{i+1} - \sum_{i=1}^n i2^i \end{aligned}$$

在第二个求和中移位第  $i$  个值,用  $i+1$  代替为  $i$ :

$$= n2^{n+1} + \sum_{i=0}^{n-1} i2^{i+1} - \sum_{i=0}^{n-1} (i+1)2^{i+1}$$

把第二个求和分成两个部分:

$$= n2^{n+1} + \sum_{i=0}^{n-1} i2^{i+1} - \sum_{i=0}^{n-1} i2^{i+1} - \sum_{i=0}^{n-1} 2^{i+1}$$

消去相似的项:

$$= n2^{n+1} - \sum_{i=0}^{n-1} 2^{i+1}$$

再在求和中移位第  $i$  个值,用  $i$  代替  $i+1$ :

$$= n2^{n+1} - \sum_{i=1}^n 2^i$$

用我们已经知道的一个解代替求和:

$$= n2^{n+1} - (2^{n+1} - 2)$$

最后,重新组织方程:

$$= (n-1)2^{n+1} + 2$$

## 14.2 递归关系

递归关系经常用于对递归函数的代价建模。例如,标准的归并排序(8.5节)对一个长度为  $n$  的表进行处理,把它分成两半,对每一半完成归并排序,最后在第  $n$  步把两个子表合到一起。可以把它的代价建模为:

$$T(n) = 2T(n/2) + n$$

也就是说,在输入长度为  $n$  的情况下,算法的代价是输入长度为  $n/2$  时的代价的两倍(对归并排序的递归调用)加上  $n$ (把两个子表合在一起的时间)。

有许多处理递归关系的方法,这里简单地考虑三种方法。第一种方法是估计技术:猜测递归的上下限,使用归纳方法证明上下限,然后根据需要进行收缩。第二种方法是扩展递归,把它转换成求和,然后使用求和技术。第三种方法是当递归是一种特定形式时,利用已经证明的定理。特别是,典型的分治算法,如归并排序,会产生一种符合我们已有解的模式。

### 14.2.1 估计上下限

解决递归问题的第一种方法是猜测答案,然后试着证明它正确。如果给出了一个正确的上下限估计,经过归纳证明很容易就可以验证事实。如果证明成功,那么就试着收缩上下限;如果证明失败,那么就放松限制再试着证明;一旦上下限符合要求,就完成了任务。当只是查找渐进复杂性时,这是一种很有用的技术;当寻找一种精确的闭合形式解时(即寻找表达式的常量时),这个方法就不合适了。

**例 14.4** 使用猜测技术找到归并排序的渐进限制,其运行时间用下面的方程描述:

$$T(n) = 2T(n/2) + n; \quad T(2) = 1$$

可以从猜测这个递归有一个上限  $O(n^2)$  开始,更准确地说,假定:

$$T(n) \leq n^2$$

通过归纳证明这个猜测是正确的。在这个证明中,为了使计算更简便,假定  $n$  是 2 的乘方。对于基本情况,  $T(2) = 1 \leq 2^2$ 。对于归纳步骤,我们要证明对于所有的  $n = 2^N$ ,  $N \geq 1$ ,  $T(n) \leq n^2$  能够得到  $T(2n) \leq (2n)^2$ 。归纳假设是:

$$T(i) \leq i^2 \quad \text{当 } i \leq n \text{ 时}$$

接下来

$$T(2n) = 2T(n) + 2n \leq 2n^2 + 2n \leq 4n^2 \leq (2n)^2$$

这就是我们要证明的。这样,  $T(n)$  就在  $O(n^2)$  之中了。

$O(n^2)$  是一个好的估计吗? 在倒数第二步我们从  $n^2 + 2n$  到达了更大的  $4n^2$ 。这表明  $O(n^2)$  是一个很高的估计。如果我们的猜测更小一些, 例如对于某个常数  $c$ ,  $T(n) \leq cn$ , 很明显, 这样做不行, 因为  $c2n = 2cn$ , 没有为额外的代价  $n$  留下余地, 使这两块合在一起。这样, 真正的代价一定在  $cn$  和  $n^2$  之间。

现在试一试  $T(n) \leq n \log n$ 。对于基本情况, 递归定义设置  $T(2) = 1 \leq (2 \cdot \log 2) = 2$ 。假定(归纳假设)  $T(n) \leq n \log n$ , 那么:

$$T(2n) = 2T(n) + 2n \leq 2n \log n + 2n \leq 2n(\log n + 1) \leq 2n \log 2n$$

就是我们想要证明的。类似地, 我们可以证明  $T(n)$  在  $\Omega(n \log n)$  中。这样,  $T(n)$  也是  $\Theta(n \log n)$ 。

### 14.2.2 扩展递归

如果只需要一个近似的答案, 上下限估计是有效的。要找到精确的答案, 就需要更精确的技术, 其中一种技术就是扩展(expanding)递归。在这种方法中, 方程右边较小的项被依次根据定义代替, 这就是扩展步。这些项被再次扩展, 依次下去, 直到没有递归结果的一个完整系列, 这样就会得到一个求和问题, 然后就可以使用解决求和问题的技术了。

**例 14.5** 为下面的递归关系找到解:

$$T(n) = 2T(n/2) + 5n^2; T(1) = 7$$

为了简单起见, 我们假定  $n$  是 2 的乘方, 因此我们把它重新写为  $n = 2^k$ 。递归关系可以像下面这样扩展:

$$\begin{aligned} T(n) &= 2T(n/2) + 5n^2 \\ &= 2(2T(n/4) + 5(n/2)^2) + 5n^2 \\ &= 2(2(2T(n/8) + 5(n/4)^2) + 5(n/2)^2) + 5n^2 \\ &= 2^k T(1) + 2^{k-1} \cdot 5 \left( \frac{n}{2^{k-1}} \right)^2 + \cdots + 2 \cdot 5 \left( \frac{n}{2} \right)^2 + 5n^2 \end{aligned}$$

最后这个表达式可以使用如下的求和很好地表示:

$$\begin{aligned} 7n + 5 \sum_{i=0}^{k-1} n^2 / 2^i \\ = 7n + 5n^2 \sum_{i=0}^{k-1} 1/2^i \end{aligned}$$

根据方程(2.5), 我们有:

$$\begin{aligned} &= 7n + 5n^2(2 - 1/2^{k-1}) \\ &= 7n + 5n^2(2 - 2/n) \\ &= 7n + 10n^2 - 10n \\ &= 10n^2 - 3n \end{aligned}$$

这就是  $n$  是 2 的乘方时递归问题的精确解答。



### 14.2.3 分治法递归

解决递归的第三种方法是利用已经知道的描述一类递归问题解法的定理。一个有用的例子是给出一类已知的称为分治法(divide and conquer)递归的解答的定理。这类问题具有形式:

$$T(n) = aT(n/b) + cn^k; \quad T(1) = c$$

其中  $a, b, c$  和  $k$  都是常数。一般说来,这个递归描述了大小为  $n$  的问题分解成  $a$  个大小为  $n/b$  的子问题,而  $cn^k$  是合并各个部分解需要的工作量。归并排序就是分治法的一个例子,而且它的递归符合这种形式,二分法检索也是这样的例子。我们使用扩展递归的方法为分治递归推导出一般形式的解法,假定  $n = b^m$ 。

$$\begin{aligned} T(n) &= a(aT(n/b^2) + c(n/b)^k) + cn^k \\ &= a^m T(1) + a^{m-1}c(n/b^{m-1})^k + \cdots + ac(n/b)^k + cn^k \\ &= c \sum_{i=0}^m a^{m-i} b^{ik} \\ &= ca^m \sum_{i=0}^m (b^k/a)^i \end{aligned}$$

注意:

$$a^m = a^{\log_b n} = n^{\log_b a} \quad (14.1)$$

这个求和是一个几何级数,它的求和依赖于比率  $r = b^k/a$ 。有如下三种情况。

1.  $r < 1$ 。根据方程(2.4):

$$\sum_{i=0}^m r^i < 1/(1-r) \quad \text{常数}$$

这样:

$$T(n) = \Theta(a^m) = \Theta(n^{\log_b a})$$

2.  $r = 1$ 。由于  $r = b^k/a$ , 可知  $a = b^k$ , 从对数的定义立即可以知道  $k = \log_b a$ , 从方程(14.1)注意到  $m = \log_b n$ , 这样:

$$\sum_{i=0}^m r = m + 1 = \log_b n + 1$$

由于  $a^m = n \log_b a = n^k$ , 有:

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^k \log n)$$

3.  $r > 1$ 。根据方程(2.6):

$$\sum_{i=0}^m r = \frac{r^{m+1} - 1}{r - 1} = \Theta(r^m)$$

这样:

$$T(n) = \Theta(a^m r^m) = \Theta(a^m (b^k/a)^m) = \Theta(b^{km}) = \Theta(n^k)$$

可以把上面的推导概括为下面的定理。

**定理 14.1**

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & a > b^k \\ \Theta(n^k \log n) & a = b^k \\ \Theta(n^k) & a < b^k \end{cases}$$

每当合适的时候就可以应用这个定理,而不需要重新推导递归问题的解法。例如,应用这个定理可以解决:

$$T(n) = 3T(n/5) + 8n^2$$

由于  $a = 3, b = 5, c = 8$  和  $k = 2$ , 并且  $3 < 5^2$ 。应用定理的第三种情况,  $T(n) = \Theta(n^2)$ 。

作为另一个例子,使用定理解决归并排序的递归关系:

$$T(n) = 2T(n/2) + n; \quad T(1) = 1$$

由于  $a = 2, b = 2, c = 1$  和  $k = 1$ , 并且  $2 = 2^1$ 。应用定理的第二种情况,  $T(n) = \Theta(n \log n)$ 。

#### 14.2.4 快速排序平均情况分析

在 8.4 节,我们确定快速排序的平均情况分析具有以下递归关系:

$$T(n) = cn + \frac{1}{n} \sum_{k=0}^{n-1} [T(k) + T(n-1-k)], \quad T(0) = T(1) = c$$

$cn$  项是 `findpivot` 和 `partition` 步骤的上限。通过观察发现每个元素  $k$  都有同样的可能成为划分元素,从而得到这个方程。我们还发现两个递归项  $T(k)$  和  $T(n-1-k)$  是相等的,因为一个是从  $T(0)$  累加到  $T(n-1)$ ,另一个是从  $T(n-1)$  累加到  $T(0)$ ,从而可以简化这个递归关系式,得到:

$$T(n) = cn + \frac{2}{n} \sum_{k=0}^{n-1} T(k)$$

求和的移位方法提供了闭合形式解。把两边都乘以  $n$ ,从  $nT(n+1)$  的公式中减去结果:

$$nT(n) = cn^2 + 2 \sum_{k=1}^{n-1} T(k)$$

$$(n+1)T(n+1) = c(n+1)^2 + 2 \sum_{k=1}^n T(k)$$

从两边减去  $nT(n)$ ,得到:

$$(n+1)T(n+1) - nT(n) = c(n+1)^2 - cn^2 + 2T(n)$$

$$(n+1)T(n+1) - nT(n) = c(2n+1) + 2T(n)$$

$$(n+1)T(n+1) = c(2n+1) + (n+2)T(n)$$

$$T(n+1) = \frac{c(2n+1)}{n+1} + \frac{n+2}{n+1}T(n)$$

注意到  $\frac{c(2n+1)}{n+1} < 2c$ , 扩展递归关系,可以得到:

$$\begin{aligned} T(n+1) &\leq 2c + \frac{n+2}{n+1}T(n) \\ &= 2c + \frac{n+2}{n+1} \left( 2c + \frac{n+1}{n}T(n-1) \right) \\ &= 2c + \frac{n+2}{n+1} \left( 2c + \frac{n+1}{n} \left( 2c + \frac{n}{n-1}T(n-2) \right) \right) \\ &= 2c + \frac{n+2}{n+1} \left( 2c + \cdots + \frac{4}{3} \left( 2c + \frac{3}{2}T(1) \right) \right) \\ &= 2c \left( 1 + \frac{n+2}{n+1} + \frac{n+2}{n+1} \frac{n+1}{n} + \cdots + \frac{n+2}{n+1} \frac{n+1}{n} \cdots \frac{3}{2} \right) \\ &= 2c \left( 1 + (n+2) \left( \frac{1}{n+1} + \frac{1}{n} + \cdots + \frac{1}{2} \right) \right) \end{aligned}$$

$$= 2c + 2c(n+2)(H_{n+1} - 1)$$

其中  $H_{n+1}$  是调和级数。根据方程(2.10),  $H_{n+1} = \Theta(\log n)$ , 因此这个求和是  $\Theta(n \log n)$ 。

### 14.3 均摊分析

这一节给出均摊分析(amortized analysis)的概念;均摊分析是对一组操作的分析。特别是,均摊分析允许处理这样一种情况, $n$  个操作在最差情况下的代价小于任何一个操作在最差情况下的代价的  $n$  倍。均摊分析并不是把注意力集中到每个操作的单独代价上,然后再把它们加到一起,而是看到整个一组操作的代价,再把整体的代价分摊到每一个单独的操作上去。

均摊分析的一个简单例子是对一个未排序的数组进行一组顺序检索。对于  $n$  次随机检索,每次检索在平均情况下的代价是  $n/2$ , 因此对于这一组检索预期估计的总代价是  $n^2/2$ 。但是,在最差情况下,所有检索都到达数组的最后一项,这种情况下,每一次的检索代价是  $n$ , 总的最差情况下的代价是  $n^2$ 。把这种情况与另外一组  $n$  次检索的代价比较一下,这组检索对于数组中的每一项正好检索一次。在这种情况下,有些检索的代价一定很高,还有一些检索的代价则一定很低。在最好、平均和最差情况下,这个问题的检索总数一定是  $\sum_{i=1}^n i \approx n^2/2$ 。这只是代价较悲观的分析的一半数值。在更悲观的分析中,组内的每一个操作的代价都处于最差情况。

作为均摊分析的另一个例子,考虑一下为一个二进制计数器加 1 的过程。算法要把低位(最右边)向高位(最左边)移动,把 1 变成 0,直到碰到第 1 个 0,把这个 0 变成 1,这样加 1 操作就完成了。下面是实现加 1 操作的 Java 代码,假定一个长度为  $n$  的二进制数存储在一个长度为  $n$  的数组中。

```
for (i = 0; ((i < A.length) && (A[i] == 1)); i++)
    A[i] = 0;
if (i < A.length)
    A[i] = 1;
```

如果我们从 0 到  $2^n - 1$  开始计数(需要一个至少  $n$  位的计数器),按照被处理的位的数目,加 1 操作的平均代价是多少? 简单的最差情况分析认为,如果所有  $n$  位都是 1(除了高位以外),那么  $n$  位都需要处理,这样,如果有  $2^n$  次加 1,那么代价就是  $n2^n$ 。然而,这就太高了,因为很少有这么多位需要处理,实际上,有一半的时间低位是 0,因此只有这些位需要处理。有四分之一的的时间低两位是 01,因此只有低两位需要处理。看待这个问题的另一种方式是低位总是要翻转,其左边的位有一半时间需要翻转,下一位有四分之一的的时间需要翻转,依此类推。我们可以通过求和得到它(从右到左计算位的代价):

$$\sum_{i=0}^{n-1} \frac{1}{2^i} < 2$$

也就是说,每次加 1 操作翻转位的平均数是 2,对于一组  $2^n$  次加 1 操作,总代价只有  $2 \cdot 2^n$ 。

通过栈数据结构的一个简单变体,可以说明均摊分析的一个有用的概念。简单修改 pop 函数,使用它带有第 2 个参数  $k$ ,标识完成  $k$  次弹出操作。这个修改过的 pop 函数称为 multi-pop。如下所示:

```
void multipop( int k ) { ... } //pop k elements from stack
```

如果栈中有  $n$  个元素, `multipop` 的“局部”最差情况分析是  $\Theta(n)$ 。这样, 如果调用 `push` 操作  $m_1$  次, 调用 `multipop`  $m_2$  次, 那么对于这组操作, 简单的最差情况代价是  $m_1 + m_2 \cdot n = m_1 + m_2 \cdot m_1$ 。这个分析极其悲观, 显然不可能每次调用 `multipop` 都弹出  $m_1$  个元素。注意力集中于单个操作的分析不能处理这种全局性的限制, 因此我们转向均摊分析, 对全部操作进行建模。

对于这个问题, 均摊分析的关键在于势 (potential) 的概念。在任何一个给定的时间, 栈中都有·一定数目的元素, `multipop` 操作的代价不会多于元素的数目。每一次调用 `push` 函数都把另一个元素放到栈中, 只需要一次 `multipop` 操作就可以把它们全都清除掉。这样, 每次调用 `push` 函数都会使得栈的势增加 1 个元素。对 `multipop` 所有调用的代价决不会超过栈的总势(对 `multipop` 本身每次调用相关的一个固定时间代价除外)。

对于任何一组 `push` 和 `multipop` 操作的均摊代价是三种代价的和。首先, 每一次 `push` 操作都要花费固定时间; 其次, 每一次 `multipop` 操作都要花费额外的固定时间, 不管这次调用弹出多少项; 最后, 计算所有 `multipop` 操作扩展的势的总数, 它最多为  $m_1$ , 即 `push` 操作的数目。因此总代价可以表达为:

$$m_1 + (m_2 + m_1) = \Theta(m_1 + m_2)$$

最后一个例子使用均摊分析证明下面二者之间的关系, 一个是 10.2 节的移至前端自组织线性表启发式规则的代价, 另一个是线性表的最优静态排序的代价。

回忆一下, 对于一组检索操作, 当线性表的记录按照访问频率排序时, 就会得到一个静态线性表的最小代价。如果不允许记录的位置改变, 这就是对记录最好的排序, 因为最常访问的记录在最前面(因而代价最小), 接下来是访问频率稍低的记录, 依此类推。

**定理 14.2** 对于·一个长度为  $n$ , 且使用移至前端启发式规则的自组织线性表进行  $n$  次或更多次检索的任意一组检索序列  $S$  需要的比较总数, 决不会超过当把这组检索序列  $S$  应用于以最优静态顺序排序的线性表时需要的比较总数的两倍。

**证明:** 与线性表中每条记录的检索关键码的每一次比较既可能成功, 也可能失败。对于  $m$  次检索, 无论是自组织线性表还是静态线性表, 必须有  $m$  次成功的比较。自组织线性表中不成功比较的总数是所有互不相同的关键码之间两两比较次数的总和。

考虑一下某一对关键码  $A$  和  $B$ 。对于任意检索序列  $S$ ,  $A$  和  $B$  之间(不成功)比较的总数等于只检索  $A$  或  $B$  的  $S$  的子序列需要的  $A$  和  $B$  之间的比较数, 把这个序列称为  $S_{AB}$ 。也就是说, 包含对其他关键码的检索不影响  $A$  和  $B$  的相对位置, 因此不会影响对  $A$  和  $B$  之间不成功比较总代价的相对贡献。

移至前端启发式规则对子序列  $S_{AB}$  进行的  $A$  和  $B$  之间不成功比较的数目, 最多是当把  $S_{AB}$  应用于线性表的最优静态顺序时需要的  $A$  和  $B$  之间的不成功比较数的两倍。要明白这一点, 假定  $S_{AB}$  包含  $i$  个  $A$ ,  $j$  个  $B$ , 其中  $i \leq j$ 。在最优静态顺序下, 需要  $i$  次不成功的比较, 因为在线性表中  $B$  一定出现在  $A$  的前面(它的访问频率更高)。每当请求顺序从  $A$  变到  $B$  或者从  $B$  变到  $A$  时, 移至前端方法都会产生一次不成功的比较。这些可能的改变的总数是  $2i$ , 因为每一次改变都要涉及一个  $A$ , 而每一个  $A$  最多是两次改变的一部分。

由于对于任何给定的关键码对, 移至前端方法需要的不成功比较的总数最多是最优静态

顺序的两倍,对于所有关键码对,移至前端方法需要的不成功比较的总数也最多是两倍多。因为这两种方法的成功比较数是相同的,移至前端方法需要的比较总数小于最优静态顺序方法需要的比较总数的两倍。

## 14.4 深入学习导读

解决递归关系的一本很好的介绍性书籍是 Fred S. Roberts 所著的《Applied Combinatorics》[Rob84]。更高级的内容参见 Graham, Knuth 和 Patashnik 的《Concrete Mathematics》[GKP89]。

有关完成均摊分析的各种方法, Cormen、Leiserson 和 Rivest 在《Introduction to Algorithms》[CLR90]中提供了很好的讨论。有关伸展树在  $m > n$  时需要  $m \log n$  时间对  $n$  个结点完成一组  $m$  个操作的均摊分析,参见 Sleator 和 Tarjan 的“Self-Adjusting Binary Search Trees”[ST85]。定理 14.2 的证明来自 Bentley 和 McGeoch 的“Amortized Analysis of Self-Organizing Sequential Search Heuristics”[BM85]。

## 14.5 习题

14.1 使用猜测多项式并推导出系数的技术解决求和问题:  $\sum_{i=1}^n i^2$

14.2 使用猜测多项式并推导出系数的技术解决求和问题:  $\sum_{i=1}^n i^3$

14.3 使用猜测多项式并推导出系数的技术解决求和问题:  $\sum_{i=n}^b i^2$

14.4 使用移位方法解决求和问题:  $\sum_{i=1}^n i$

14.5 使用移位方法解决求和问题:  $\sum_{i=1}^n 2^i$

14.6 使用移位方法解决求和问题:  $\sum_{i=1}^n i2^{n-i}$

14.7 证明 2.4 节的函数 TOH 需要的移动数是  $2^n - 1$ 。

14.8 给出并证明递归关系  $T(n) = T(n-1) + c$ ,  $T(1) = c$  的闭合形式解。

14.9 利用归纳法证明递归关系:

$$T(n) = 2(T)(n/2) + n; \quad T(2) = 1$$

的闭合形式解是  $\Omega(n \log n)$ 。

14.10 以渐近形式,而不是以精确形式,找到递归关系的解:

$$T(n) = T(n/2) + \sqrt{n}; \quad T(1) = 1$$

14.11 通过扩展递归关系找到下式的精确的闭合形式解:

$$T(n) = 2T(n/2) + n; \quad T(2) = 2$$

14.12 使用定理 14.1 证明二分法检索需要  $\Theta(\log n)$  时间。

- 14.13 回忆一下,当一个散列表超过半满时,它的性能就会急剧下降。对这个问题的一种解决方法是把散列表中的所有元素插入到一个原来是原来两倍大小的新散列表中。假定插入一个散列表的(预计)平均情况代价是  $\Theta(1)$ ,证明当采用重新插入策略时,插入的平均情况代价仍然是  $\Theta(1)$ 。
- 14.14 两个  $n \times n$  矩阵相乘的标准算法需要  $\Theta(n^3)$  时间,通过以各种方式对乘法进行重新布置和分组可能会做得更好一些。这种做法的一个例子是 Strassen 的矩阵相乘算法。假定  $n$  是 2 的乘方,下面的  $A$  和  $B$  是  $n \times n$  数组,  $A_{ij}$  和  $B_{ij}$  指大小为  $n/2 \times n/2$  的数组。Strassen 的算法按照特定的顺序把子数组相乘,如下面的方程所示:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} s_1 + s_2 - s_4 + s_6 & s_4 + s_5 \\ s_6 + s_7 & s_2 - s_3 + s_5 - s_7 \end{bmatrix}$$

也就是说,通过一组  $n/2 \times n/2$  数组的矩阵相乘和相加,得到一个  $n \times n$  数组的相乘结果。子数组之间的相乘也可以使用 Strassen 的算法,两个子数组之间的相加需要  $\Theta(n^2)$  时间。子因子定义如下:

$$\begin{aligned} s_1 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\ s_2 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\ s_3 &= (A_{11} - A_{21}) \cdot (B_{11} + B_{12}) \\ s_4 &= (A_{11} + A_{12}) \cdot B_{22} \\ s_5 &= A_{11} \cdot (B_{12} - B_{22}) \\ s_6 &= A_{22} \cdot (B_{21} - B_{11}) \\ s_7 &= (A_{21} + A_{22}) \cdot B_{11} \end{aligned}$$

- (a)说明 Strassen 的算法是正确的。
- (b)Strassen 算法需要多少个子数组相乘,多少个子数组相加? 如果以同样的方式定义子数组,正常的矩阵相乘需要多少次相乘、相加? 说明 Strassen 算法和正常矩阵相乘算法的递归关系。
- (c)为你推导的 Strassen 算法递归关系给出闭合形式解(使用定理 14.1)。
- (d)针对 Strassen 算法的实用性,给出你的观点。
- 14.15 有一个带有  $N$  个结点的 2-3 树,证明  $M$  次插入额外结点需要  $O(M + N)$  次结点分离。
- 14.16 实现一种长度不确定的基于数组的线性表的一种方法是,让数组可以增长与收缩,这称为动态数组(dynamic array)(Java 的 vector 类可以实现一个动态数组)。
- (a)如果数组的初始长度为 0,每当数组中元素的数目超过数组的大小时,就把数组的大小变成 2 倍,那么把元素插入线性表的均摊代价是多少?
- (b)考虑一下每当数组不到一半时就把数组长度减半的下溢策略。举一个例子说明这种策略会导致很差的均摊代价。
- (c)给出一个比(b)中建议更好的下溢策略,它的均摊分析显示删除需要  $O(n)$  时间。
- 14.17 回忆一下,如果无向图中的两个顶点有一个路径相连,那么它们就在同一个互连

子图中。在无向图中找到一个互连子图有一个很好的算法,它从第一个顶点调用一个 DFS 开始,DFS 到达的所有顶点都在同一个互连子图中,因此把它们标记起来。然后查看一遍顶点的 mark 数组,直到找到一个未标记的顶点  $i$ 。再对  $i$  调用 DFS,从  $i$  可以到达的所有顶点都在第二个互连子图中,再查看顶点的 mark 数组,直到所有顶点都在某个互连子图中。下面是算法的一个框架:

```
for (i = 0; i < G.n(); i++) // Assume n vertices in the graph
    G.setMark(i, 0); // Vertices start in no component
int compcount = 1; // Counter for current component
for (i = 0; i < G.n(); i++)
    if (G.getMark(i) == 0) // Start a new component
        DFS_component(G, i, compcount);
        compcount++;
}

void DFS_component(Graph G, int v, int compcount) {
    G.setMark(v, compcount);
    for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
        if (G.getMark(G.v2(w)) == 0)
            DFS_component(G, G.v2(w));
}
```

使用均摊分析中势的概念说明为什么这个算法的总代价是  $\Theta(|V| + |E|)$ ? (注意由于这个算法不允许一组任意的 DFS 操作,而只是固定为从每个顶点简单调用一次 DFS,故而这不是一个真正的均摊分析。)

- 14.18 给出一个类似于定理 14.2 使用的证明,说明这样一个事实,对于一个使用访问计数方法、长度为  $n$  的自组织线性表,任意一组  $n$  次或多于  $n$  次的检索序列  $S$  需要的比较总数,不会超过当把序列  $S$  应用于以优化静态顺序存储的线性表需要的比较总数的两倍。

## 14.6 项目设计

- 14.1 使用路径压缩和重量权衡合并两种规则实现 6.2 节的 UNION/FIND 算法。计算一下各组等价情况需要的结点访问总数,以确定算法的实际性能是否符合预期的代价  $\Theta(n \log^* n)$ 。
- 14.2 实现标准的  $\Theta(n^3)$  矩阵相乘算法和 Strassen 的矩阵相乘算法(见习题 14.14)。通过经验测试,试估计两种算法运行时间方程的常数因子。 $n$  应当有多大才能使 Strassen 的算法比标准算法更有效?

## 第 15 章 计算的限制

### 15.1 简介

本书中包含数据结构的许多例子,这些数据结构的例子用于解决大量问题,其中有许多高效的算法。一般说来,我们的检索算法争取在最差情况下达到  $O(\log n)$ ,而排序算法争取达到  $O(n \log n)$ 。有一些算法——例如每对顶点间最短路径的算法——其渐近复杂性更高,Floyd 的每对顶点间最短路径算法是  $O(n^3)$ 。

我们能够有效地解决许多问题,部分原因在于使用了有效的算法。对于某些知道算法的问题,很有可能编写出一个低效的算法来“解决”这个问题。例如,考虑一个算法,它测试其输入的每种可能排列,直到找到一个提供已排序线性表的正确排列。这个算法的运行时间可能高得无法接受,因为它与排列数成正比。对于  $n$  个输入,排列数是  $n!$ 。解决最小支撑树问题时,如果我们测试边的每一种可能子集,看一看哪一个能形成最小支撑树,对于一个边数为  $|E|$  的图,工作量将与  $2^{|E|}$  成正比。可是对于这些问题,我们都有更聪明的算法,可以(相对来说)更快地找到答案。

但是,实际生活中有许多计算问题必须使用可能很耗费时间的算法来解决。一个简单的例子是汉诺塔问题,它需要  $2^n$  次移动才能解决一个  $n$  层的塔。任何一个计算机程序都不可能用少于  $\Omega(2^n)$  的时间解决这个问题,因为许多移动必须打印出来。

除了这些解法必须花费很长时间运行以外,还有许多问题,我们根本不知道它们是否有有效的算法。对于这样的问题,我们知道的最好的算法都运行得很慢,但是可能还有更好的算法有待于发现。然而,尽管运行时间很长的问题很不如意,不能解决的问题则更糟糕!这样的问题确实存在,本章就研究这样一些问题。本章对代价很高和无法解决的问题的理论给出了一个简要介绍。

### 15.2 归约

我们从一个重要的概念开始,来理解问题之间的关系,这个概念就是归约(reduction)。归约使得我们可以通过一个问题解决另外一个问题。同样重要的是,当我们希望理解一个问题的难度时,归约便于我们对问题代价的上下限做出相关的表述(相对于算法和程序)。

由于本章广泛讨论问题的概念,我们从简化问题描述的表示法开始。在整个一章中,把问题定义为输入和输出之间的一个映射。问题的名字全部使用大写字母给出,这样,排序问题(Sorting)的完整定义如下。

**定义 15.1** 排序问题:

输入:一组整数  $x_0, x_1, x_2, \dots, x_{n-1}$ 。

输出:这组整数的一个排列  $y_0, y_1, y_2, \dots, y_{n-1}$ ,使得每当  $i < j$  时,  $y_i \leq y_j$ 。



一旦你已经买来或者编写出一个程序解决问题,例如排序问题,你就可能把它作为一个工具解决一个不同的问题,在软件工程中这称为软件重用(software reuse)。为了说明这一点,我们考虑另外一个问题。

### 定义 15.2 配对问题:

输入:两组整数  $X = (x_0, x_1, \dots, x_{n-1})$  和  $Y = (y_0, y_1, \dots, y_{n-1})$ 。

输出:两组数的元素配对,  $x$  中的最小值配对  $y$  中的最小值,  $x$  中的次小值配对  $y$  中的次小值,依此类推。

图 15.1 说明了配对问题(Pairing)。解决配对问题的一种方法是使用一个已存在的排序程序,首先排序两组整数,然后根据它们的排序位置进行配对。从技术上说,配对问题被归约到排序问题,因为可以用排序问题解决配对问题。

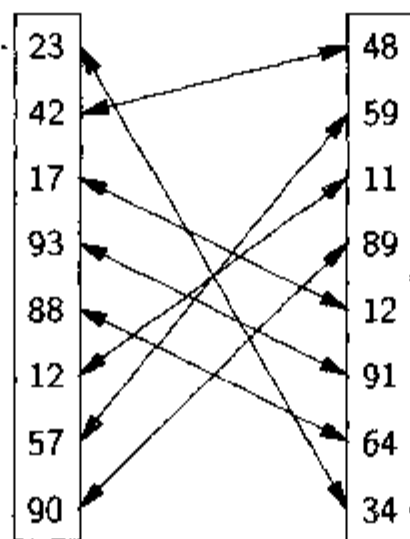


图 15.1 配对问题图示。这两个表中的数搭配起来,使得两个表中的最小值配成一对,两个表中的次小值配成一对,依此类推

归约是一个三步过程。第一步把一个配对问题的实例转化成两个排序问题的实例。转换步骤没什么意思,它只是得到每一个序列,并且把序列赋给要传递给排序问题的数组。第二步是排序这两个数组(即把排序应用于每一个数组)。第三步是把排序问题的输出转化为配对问题的输出。这可以通过配对每个数组的第一个元素、第二个元素等完成。

从配对问题到排序问题的归约有助于建立配对问题代价的一个上限。根据渐近表示法,假定可以找到一种方法“足够快地”把配对问题的输入转化为排序问题的输入,还能找到第二种方法“足够快地”把排序问题的结果转换回配对问题的结果,那么配对问题的渐近代价就不会比排序问题的渐近代价高。在这种情况下,从配对问题向排序问题的转换需要做的工作极少,而且从排序问题的结果转换回配对问题的结果要做的工作也极少,因而这种解法的主要代价是完成排序操作。这样,配对问题的一个上限就是  $O(n \log n)$ 。

除了解决新问题(并且为其建立一个上限)以外,归约还有另外一个用途。如果可以“足够快地”把排序问题转化为配对问题,那么配对问题的最小代价又是什么呢?从 8.9 节我们知道,排序问题的代价是  $\Omega(n \log n)$ 。假定在  $O(n)$  时间可以完成配对问题,那么创建排序算法的一种方法就是把排序问题转换成配对问题,为配对问题运行算法,最后把结果转换回用于排序的结果。假定可以在排序问题和配对问题之间“足够快地”来回进行转换,那么这样就会为排序问题产生一个  $O(n)$  算法!由于这与我们知道的排序问题的下限矛盾,推理中惟一有问

题的地方就是配对问题可能在  $O(n)$  时间完成的初始假设。实际上,我们知道配对问题的代价至少要和排序问题一样高,因此它本身必须有下限  $\Omega(n \log n)$ 。

为了完成关于这个配对问题下限的证明,现在需要找到一种方法把排序问题归约到配对问题。这很容易完成。举一个排序问题实例(即一个有  $n$  个元素的数组  $A$ )。并且如此生成第二个数组  $B$ :对于  $0 \leq i < n$ ,在  $i$  处赋给  $i$  值。把这两个数组传递给配对问题。取出配对问题的结果集,使用配对问题中来自  $B$  的一半值辨别  $A$  的一半应当在已排序的数组中占据哪些位置,即,现在可以使用  $B$  数组中对应的值作为排序关键码重新排列  $A$  数组中记录的顺序,并且运行一个简单的  $\Theta(n)$  的 Binsort(分配排序)。排序问题向配对问题的转换可以在  $O(n)$  时间完成,同样配对问题输出的转换也可以在  $O(n)$  时间转换成正确的排序问题的输出。这样,这个“排序算法”的代价主要是配对问题的代价。

考虑一下,如果有两个问题,可以找到从一个问题到另一个问题的合适归约。第一个问题的任意一个输入实例,把它命名为  $I$ ,然后把  $I$  转换成一个解,称之为  $SOL$ 。第二个问题的任意一个输入实例,称之为  $I'$ ,然后把  $I'$  转换成一个解,称之为  $SOL'$ 。可以把归约正式定义为一个三步的过程:

1. 把第一个问题的任意实例转化成第二个问题的一个实例。也就是说,必须有一个变换把第一个问题的任意实例  $I$  转换成第二个问题的实例  $I'$ 。

2. 对第二个问题的实例  $I'$  应用一个算法,产生一个结果  $SOL'$ 。

3. 把  $SOL'$  转换成  $I$  的解答,要使归约可以接受, $SOL$  实际上必须是  $I$  的正确解答。

有一点很重要,注意归约过程本身不能给出解决任何一个问题的算法。它只是在我们知道第二个问题解法的情况下,给出一种解决第一个问题的方法。更重要的是,对于本章其余部分要讨论的主题,归约给出一种通过一个问题理解另一个问题上下限的方式。特别是,给定一个高效的转换,第一个问题的上限至多是第二个问题的上限。反过来,第二个问题的下限至少是第一个问题的下限。

作为归约的第二个例子,考虑一下两个  $n$  位数相乘的问题。标准的乘法方法是把第一个数的最后一位与第二个数相乘(代价为  $\Theta(n)$  时间),把第一个数的倒数第二位与第二个数相乘(再花费  $\Theta(n)$  时间),对于第一个数中  $n$  位的每一位都依此类推。最后,把中间结果加起来。把长度为  $M$  和  $N$  的两个数相加可以很容易在  $\Theta(M+N)$  时间完成。由于第一个数的每一位都与第二个数的每一位相乘,这个算法需要  $\Theta(n^2)$  时间。已经知道有更快但更复杂的渐近算法,但是没有  $O(n)$  一样快的算法。

接下来我们问这样一个问题:对一个  $n$  位数平方也像两个  $n$  位数相乘一样难吗? 我们可能希望这个特定的例子将可能有一个比一般的相乘问题所需要的算法更快的算法。然而,一个简单的归约证明表明,平方与相乘一样难。

归约的关键是下面的公式:

$$X \times Y = \frac{(X+Y)^2 - (X-Y)^2}{4}$$

这个公式的重要性是它允许我们把相乘的任意一个实例转换成一组操作,其中包括三次加减法(每一次都可以在线性时间完成)、两次平方以及一次除以 4。除以 4 可以在线性时间完成(只需要简单转换成二进制,移动两位,再转移回来)。

这个归约表明,如果能找到平方的线性时间算法,就可以使用它建立乘法的线性时间

算法。

使用归约的另一个例子是利用对数完成乘法。乘法比加法更难,因为两个  $n$  位数相乘的代价是  $O(n^2)$ ,而两个  $n$  位数加法的代价是  $O(n)$ 。回忆一下 2.3 节,对数的一个特性是:

$$\log nm = \log n + \log m$$

这样,如果取对数和它的逆运算代价不高,就可以先把两个操作数取对数,相加,再对和进行逆运算,从而把乘法运算归约成加法运算。

在正常情况下,取对数运算及其逆运算的代价很高,因此这种归约不可行。然而,这种归约正是计算尺工作的基本原理。计算尺使用对数刻度度量两个数的长度,实际上是自动进行对数运算。然后,把这两个长度加起来,对和进行反对数运算,从另一个对数刻度上读出结果。通常认为代价非常高的部分(完成对数和反对数运算)在这里代价并不高,因为它是计算尺的物理部分。这样,通过归约为加法,整个乘法过程就可以以很低的代价完成。

作为归约的另外一个例子,考虑两个  $n \times n$  矩阵的乘法。对于这个问题,假定存储在矩阵中的值是简单的整数,两个简单整数之间相乘花费常数时间。两个矩阵相乘的标准算法是把第一个矩阵第一行的每一个元素与第二个矩阵第一列的对应元素相乘,然后把所有这些数加起来,这要花费  $\Theta(n)$  时间。这  $n^2$  个元素中的每一个都采用同样的方式计算,总共需要  $\Theta(n^3)$  时间。已经知道有更快的算法,但是没有一个像  $\Theta(n^2)$  一样快。

现在,考虑两个对称矩阵(symmetric matrix)相乘的情况。对称矩阵是其第  $ij$  项等于其  $ji$  项的矩阵,也就是说,矩阵的右上三角是其左下三角的一个映像。对于这种受限情况,是否可以利用它比一般情况更快地进行两个对称矩阵的相乘呢? 答案是否定的,从下面的归约可以看出来。假定给定两个  $n \times n$  矩阵  $A$  和  $B$ ,可以从任意一个矩阵  $A$  建立一个如下的  $2n \times 2n$  对称矩阵:

$$\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$$

这里 0 表示由零值组成的  $n \times n$  矩阵,  $A$  是原来的矩阵,  $A^T$  代表矩阵  $A$  的转置<sup>①</sup>,结果矩阵现在是对称的。可以采用同样的方式把矩阵  $B$  转换成一个对称矩阵。如果对称矩阵可以“更快地”进行相乘(特别地,如果它们可以在  $\Theta(n^2)$  时间相乘),那么就可以利用下面的式子得到任意两个  $n \times n$  矩阵在  $\Theta(n^2)$  时间相乘的结果:

$$\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & A^T B^T \end{bmatrix}$$

在上面的公式中,  $AB$  是矩阵  $A$  和矩阵  $B$  相乘的结果。

### 15.3 难解问题

本节讨论一些真正“难解”的问题。可以有多种方式认为一个问题为难解的。首先,可能对理解问题本身的定义有困难;其次,可能在找到或者理解解决问题的算法上有困难。当一个计算机理论研究者使用“难解”这个词时,它不同于上面这两种一般意义上的困难。贯穿这一节,“难解”表示问题最众所周知的算法的运行时间开销很大。难解问题的一个例子是汉诺塔

① 转置操作把原来矩阵中  $ij$  位置的项放到其转置矩阵的  $ji$  位置。

问题。理解这个问题和它的解法很简单,编写一个程序解决这个问题也很简单。但是,对于任何一个极大的值  $n$ ,要花费非常长的运行时间。试一试编写一个程序,解决一个 30 层的汉诺塔问题!

汉诺塔问题要花费指数运行时间,即它的运行时间是  $\Theta(2^n)$ ,这与一个花费  $\Theta(n \log n)$  时间或者  $\Theta(n^2)$  时间的算法截然不同,它与一个花费  $\Theta(n^4)$  的算法也大不一样。这些算法的运行时间都是多项式运行时间的例子,因为这些方程中所有项的指数都是常数。回忆一下第 3 章,如果我们买一台新型计算机,它的运行速度是原来运行速度的两倍,对于复杂性大小为  $\Theta(n^4)$  的算法,在给定时间内我们可以解决问题的规模比在低速机器上增长  $\sqrt[4]{2}$ 。也就是说,即使它非常小,但是有一个乘数因子增长。对于任何以多项式表示运行时间的算法,这都是正确的。

考虑一下如果你买一台比原来快两倍的计算机,试图在给定时间内解决一个更大的汉诺塔问题会发生什么情况。由于汉诺塔问题的复杂性是  $\Theta(2^n)$ ,我们解决这个问题只多一层!没有乘数因子,这对于任何指数算法都是正确的:处理能力的常数倍增长导致解决问题能力的定量增加。

在本章的其余部分,难解(hard)的算法定义为以指数时间运行的算法,即对于某个常数  $c > 1$ ,以  $\Omega(c^n)$  时间运行的算法,下一节将给出难解问题的定义。

### 15.3.1 NP 完全性

想像一下这样一台神奇的计算机,它解决问题的方式是从一个问题的所有可能答案中猜测出一个正确的答案。可以用另一种方式看待这个问题,想像一下可以同时测试所有可能答案的一台超级并行计算机。当然,这台计算机可以做普通计算机能做到的任何事,它还能比普通计算机更快地解决一些问题。考虑这样一些问题,给出解的某个猜测,检查猜测的解,看看它是否能够在多项式时间完成。即使可能解的数目是指数级的,任何给定的猜测可以在多项式时间检查出来(相当于在多项式时间同时检查出所有可能的解),这样问题就可以在多项式时间解决。相反,如果不能通过猜测出正确答案并检查它的方式在多项式时间得到问题的答案,通过其他方式也不可能在多项式时间做到。

“猜测”问题的正确答案,或者并行检查所有可能的答案以确定哪一个正确,这种思想称为非确定性(nondeterminism)。如果一个算法采用这种方式工作,这个算法就称为非确定性算法(nondeterministic algorithm)。算法在一个非确定性机器上以多项式时间运行的任何问题都有一个特殊的名字:它是一个 NP 问题。这样,NP 问题就是能在一个非确定性机器上以多项式时间解决的那些问题。

当然,并不是在常规机器上需要指数运行时间的所有问题都在 NP 中。例如,汉诺塔问题就不在 NP 中,因为它对于  $n$  层需要  $O(2^n)$  步打印出正确的移动。一个非确定性机器不能在多项式时间“猜测”并打印出正确答案。

另外,考虑一个通常称为货郎担问题(Traveling Salesman Problem)的问题。

#### 定义 15.3 货郎担问题(1)

输入:一个完全有向图  $G$ ,图中每个边都有赋值的距离。

输出:包括每一个结点的最短简单回路。

图 15.2 说明了这个问题。这里显示了 5 个顶点,还有边和每对边的相关代价(为了简单

起见,假定两个方向的代价相同,尽管这不是必需的)。如果货郎以 ABCDEA 的顺序访问城市,他所走过的总距离为 13;一个更好的路径是 ABDCEA,其代价是 11;这个图的最佳路径是 ABEDCA 其代价是 9。

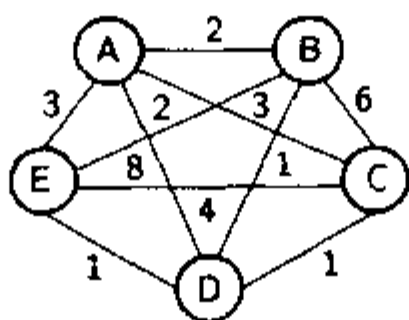


图 15.2 货郎担问题图示。图中显示了 5 个顶点和每对城市之间的边。问题是访问所有的城市,每个城市只访问一次,再回到起始城市,而使总代价最少

不能用一台非确定性计算机在多项式时间解决这个问题。问题是给出一个候选回路,尽管可以很快检测出答案是否是一个符合相应形式的回路,但是没有一种简单方法知道它实际上就是最短的回路。然而,可以解决这个问题一个变体,它们的形式是决策问题(decision problem)。一个决策问题就是其答案要么为“是”要么为“否”的问题。货郎担问题的决策形式如下:

#### 定义 15.4 货郎担问题(2)

输入:一个完全有向图  $G$ ,图中每条边都有作为权重的距离,还有一个整数  $K$ 。

输出:如果有一个简单回路,其总距离  $\leq K$ ,并包含  $G$  中的每一个顶点,那么就输出 YES,否则输出 NO。

可以使用一台非确定性计算机在多项式时间解决这个问题。非确定性算法只是并行地检查图中边的所有可能子集。如果边的任何一个子集是一个总长度小于或者等于  $K$  的符合条件的回路,答案就是 YES,否则答案就是 NO。注意只需要某些子集符合要求,有多少个子集失败并不重要。检查算法可以像下面这样在多项式时间内运行:把一个子集的各个边的距离相加,并验证这些边形成一个恰好访问每个结点一次的回路。但是,一共有  $|E|!$  个子集需要检查,因此这个算法不能在一个常规计算机上转换成一个多项式时间算法。尽管这个问题已经被许多计算机科学家广泛研究了许多年,世界上还是没有人能够找到一个多项式时间算法,在一台常规计算机上解决货郎担问题。

大量问题都具有这个特性:我们知道有效的非确定性算法,但是不知道是否存在有效的确定性算法。同时,不能证明这些问题中的任何一个不存在有效的确定性算法。这类问题称为 NP 完全性(NP-complete)问题。有关 NP 完全性问题最奇怪、也最有吸引力的地方是如果有人对其中的一个问题找到了在常规计算机上以多项式时间运行的答案,那么经过一系列归约, NP 中的所有其他问题也都可以常规计算机上经过多项式时间得到解决。

一个问题  $X$  定义为 NP 完全性问题,如果:

1.  $X$  在 NP 中,而且
2. NP 中的每一个其他问题都可以在多项式时间归约到  $X$ 。

第二个要求看起来似乎不可能,但是实际上有几百个这样的问题,其中包括货郎担问题。另外一个这样的问题称为  $K$  团( $K$ -CLIQUE)问题。 $K$  团问题问,任意给定一个无向图  $G$ ,是否存在一个至少有  $k$  个顶点的完全子图。没有人知道  $K$  团问题有没有多项式时间解法,但是如

果为 K 团问题或者货郎担问题找到这样一个算法,那么就可以修改这个解法,在多项式时间解决另一个问题,而且还可以解决 NP 中的其他所有问题。

知道一个问题 P1 是 NP 完全性问题,在理论上的一个主要好处是,可以使用它来说明另一个问题 P2 是 NP 完全性问题,这可以通过找到一个从 P1 到 P2 的多项式时间归约来完成。由于我们已经知道 NP 中的所有问题都可以在多项式时间归约到 P1(通过 NP 完全性定义),现在就可以知道通过归约到 P1 再归约到 P2 的简单算法,也可以把所有问题都归约到 P2。

知道一个问题是 NP 完全性问题有一个极为实际的好处。如果对于任何一个 NP 完全性问题找到了多项式时间解法,就可以对所有这些问题找到多项式时间解法。其含义是:

1. 由于没有人找到这样一个解法,它一定是非常困难的,或者是不可能做到的。
2. 为一个 NP 完全性问题寻找多项式时间解法的努力,可以认为扩展到所有 NP 完全性问题。

NP 完全性对于一般程序员有什么实际意义呢?如果你的老板要求你提供一个快速算法解决一个问题,如果你说你所能做到的最好方法是一个指数时间算法,她一定会很不高兴。但是如果你发现这个问题是一个 NP 完全性问题,而她仍然会不高兴,可她至少不会迁怒于你。通过表明你的问题是一个 NP 完全性问题,你实际上是在说世界上最有才华的计算机科学家花费了将近 30 年的时间尝试为你的问题寻找一个多项式时间算法,但都失败了。

在常规计算机上使用多项式时间可以解决的问题称为 P 问题。很显然,只要简单地忽略使用非确定性能力,P 问题中的所有问题在一台非确定性计算机上都可以在多项式时间解决。NP 中的有些问题是 NP 完全性的。由于所有可在多项式时间解决的问题都可在指数时间解决,可以认为所有能够在指数时间或者更多时间解决的问题构成一个更大的问题类,这样,我们就可以根据图 15.3 看待指数时间或者更大的问题类了。

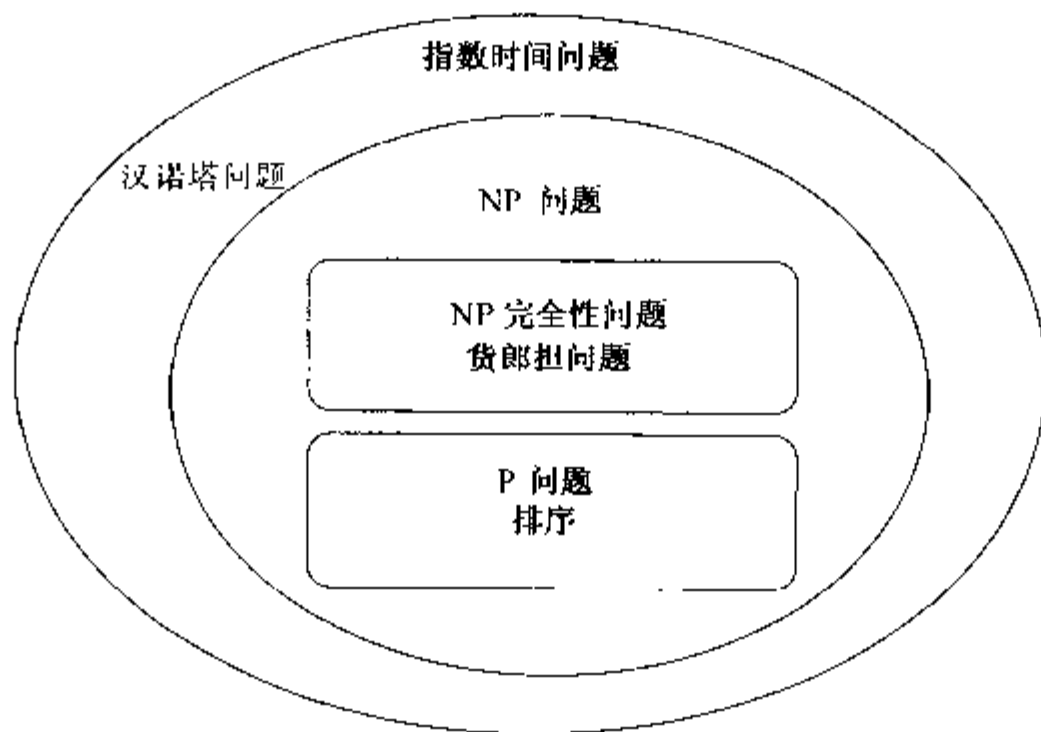


图 15.3 我们关于需要指数时间或更少时间问题的知识。其中一些问题可以在一个非确定性计算机上在多项式时间内解决。在这些问题上,有些知道是 NP 完全性的,有些则可以在常规计算机上以多项式时间解决

是否  $P = NP$  是理论计算机科学中没有回答的最重要的问题。如果它们是相等的,就意味着货郎担问题和其相关的问题有多项式时间算法。由于已经知道货郎担问题是 NP 完全性

的,如果发现了这个问题的一个多项式时间算法,那么 NP 中的所有问题都可以在多项式时间得到解决。相反,如果能够证明货郎担问题有一个指数时间下限,我们就会知道  $P \neq NP$ 。

### 15.3.2 绕过 NP 完全性问题

但是,发现你的问题是 NP 完全性问题并不意味着就可以忘记它。不管问题的复杂性怎样,货郎都要找到一条合理的路径去叫卖。如果遇到一个必须解决的 NP 完全性问题时怎么办呢?

有许多技术可以尝试。一种方法是只运行这个程序的一个小实例。对于有些问题,这样做无法接受,例如,货郎担问题随着城市数的增加,运算复杂性增长得非常快,以致于在问题的规模超过 20 个城市时就不能在一台现代计算机上解决它了。然而, NP 中的其他一些问题尽管需要指数运行时间,却不是增长得非常快,不影响当问题的规模不太大却很有用时给予解决。这样的一个例子是背包问题(KNAPSACK problem)。给定一组物体,每一个物体都有给定的大小和给定的值。对于一个大小为  $k$  的背包,是否有这些物体的一个子集,使得各个物体大小的总和小于等于  $k$ ,而各个物体值的总和大于等于  $v$ ? 尽管这个问题是 NP 完全性问题,因而最有名的解法也需要指数运行时间,但是当  $k$  和  $v$  在几千左右的时候,在物体的数目很多的时候它仍然是可以解决的。

处理 NP 完全性问题的第二种方法是解决这个问题的不太困难的一个特定实例。例如,图论中的许多问题是 NP 完全性问题,但是同样的问题在具有特定限制的图中却不算太困难。例如顶点覆盖问题(VERTEX COVER problem)问是否有一个数目为  $k$  或者少于  $k$  的顶点组成的子集,使得图中的每一条边至少有一个端点在这个子集中。一般说来,这是一个 NP 完全性问题,但是对于对偶图(其顶点可以分成两个子集,每个子集内部的两个顶点之间没有边)有一个多项式时间解法。

第三种方法是找到问题的一个近似解。有许多找近似解的方法,一种方法是使用启发式规则来解决问题,即基于一种并不是总能给出正确答案的经验性的算法。例如,可以使用启发式规则这样近似地解决货郎担问题,从任意一个城市开始,然后总是找到下一个距离最短的没有访问过的城市。这样很少能给出最短路径,但是这个解已经足够好了。还有许多其他启发式规则为货郎担问题给出更好的解。对于一些问题,近似算法能够给出可保证的性能,答案可能就在接近最佳可行答案的一定范围内。

## 15.4 不可解问题

每天都有一些专业程序员编写一些陷入死循环的程序。当然,当一个程序处于死循环时,你无法确切知道它只是一个运行得很慢的程序,还是一个处于死循环的程序。在等待“足够的时间”之后,你就会终止它。如果你的编译器能够查看这个程序,并且在你运行它之前就告诉你这个程序可能会进入死循环,这该有多好啊!另外,给定一个程序和某个输入,如果不用实际运行这个程序就知道程序的执行对于这个输入是否会导致死循环,这将是非常有用的。

但是,停机问题(halting problem),正如它的名字一样,是无法解决的。没有一个计算机程序能够肯定地确定另外一个计算机程序是否会对所有输入停机,也没有一个计算机程序能够肯定地确定另外一个计算机程序是否会对一个指定的输入停机。怎么会这样呢?程序员经常



查看程序,确定程序是否会停机,当然这是可编程的。有些人认为任何程序都是可以分析的,作为对这些人的警告,在继续阅读之前看一看下面的代码段:

```
while (n > 1)
    if (ODD(n))
        n = 3 * n + 1;
    else
        n = n / 2;
```

这是一段有名的代码。通过这段代码赋给  $n$  的值序列有时称为输入值  $n$  的 Collatz 序列 (Collatz sequence)。这段代码对所有输入值  $n$  会停止吗? 没有人知道答案。已经试过的每一个输入都会停机,但是它对所有输入都会停机吗? 对于这段代码,由于不知道它是否会停机,所以也不知道它的运行时间的上限。至于下限,我们可以很容易给出  $\Omega(\log n)$  (见习题 3.11)。

我个人相信,某一天会有一个很聪明的人彻底分析清楚这个程序,并且彻底证明这段代码对于所有的输入值  $n$  都停机。完成这件事可能会为我们带来新的技术,使我们普遍提高分析程序的能力。可是,来自可计算性(computability)——研究计算机不可能做什么的计算机科学分支——的证明使我们相信总能找到另一个我们不能分析的程序,这是停机问题不可解的一个结果。

### 15.4.1 不可数性

在证明停机问题不可解之前,首先证明并不是所有的函数都是可以编程的,这是因为程序的数目比所有可能的函数的数目小得多。

**定义 15.5** 如果一个集合中的每一个元素都可以惟一对应于一个正整数,那么就称这个集合是可数的(countable)。如果不可能把一个集合中的每一个元素都惟一对应于一个正整数,那么就称这个集合是不可数的(uncountable)。

要理解“对应于一个正整数”意味着什么,想像一下有一行无限数目的桶,标号为 1、2、3……对于某个集合,把集合中的元素放到桶中,每个桶中最多一个元素。如果能够找到一种方式把集合中的所有元素都对应地放到桶中,那么这个集合就是可数的。例如,考虑正偶数集合 2、4、6……我们可以把一个整数  $i$  对应到第  $i/2$  个桶中(或者,如果不介意跳过一些桶,那么就把  $i$  对应到第  $i$  个桶中),这样,正偶数集合就是可数的。这没有什么奇怪的,即使看起来好像正偶数比正整数“更少”。有趣的是,实际上正整数并不比正偶数更多,因为可以简单地通过把整数  $i$  分配给正偶数  $2i$ ,使得惟一地把每一个正整数对应到正偶数。

程序的数量是可数的还是不可数的呢? 可以把一个程序简单地看作一个字符串(包括特殊标记、空格和行分隔符)。假定一个程序中能够出现的不同字符数目是  $P$  (在 ASCII 字符集中,  $P$  必须小于 128,但是实际数目并不重要)。如果字符串数是可数的,那么程序数当然也是可数的。我们可以按照如下方法把字符串对应到桶中。把空字符串对应到第一个桶,现在,取出所有由一个字符组成的字符串,并把它们按照字母表或者 ASCII 码顺序对应到接下来的  $P$  个桶中。接下来,取出所有由两个字符组成的字符串,再按照 ASCII 码顺序从左到右对应到接下来的  $P^2$  个桶中。同样把三个字符的字符串对应到桶中,然后是四个字符的字符串,依此



类推。通过这种方式,任何给定长度的字符串都可以对应到某个桶中。

经过这个过程,任何有限长度的字符串最终都会分配到某个桶中。这样,由于任何程序只不过是有限长度的字符串,也就要分配到某个桶中。因为所有的程序都可以对应到某个桶中,所以所有程序的集合是可数的。自然,桶中大多数字符串并不是合法程序,但是这没有关系,重要的是对应于程序的字符串也在桶中。

现在考虑所有可能的函数的数量。为了简单起见,假定所有函数都有一个正整数输入,而产生一个正整数输出,这样的函数称为整数函数(integer function)。一个函数只是从输入值到输出值的映射。当然,并不是所有的计算机程序都把整数作为输入值,产生整数输出值。然而,计算机读入或者写出的一切,实质上都是一系列数字,只是可以把这些数字解释成字母或者其他东西。任何有用的计算机程序的输入和输出都可以编码为整数值,因此我们这个简单计算机输入输出模型很普遍,可以覆盖所有可能的计算机程序。

我们现在希望看到是否可以把所有整数函数对应到桶的无限集合中。如果能够做到,那么函数的数量就是可数的,也就可能把每个函数对应到一个程序。如果整数函数的集合不能对应到桶的集合,那么整数函数就一定没有到程序的对应。

把每一个整数函数想像成一个表,它有两列,无限多个行。第 1 列列出从 1 开始的正整数,第 2 列是当给定第 1 列的值作为输入时输出的函数。这样,表就为每个函数明确地描述了从输入到输出的映射,把它称为函数表(function table)。

接下来,我们试一试把函数表对应到桶中。要进行对应,必须对函数排序,但是选择什么顺序并不重要。例如,第 1 个桶中存放总是返回 1 的函数,而不管这个函数的输入值,第 2 个桶中存放返回其输入的函数,第 3 个桶中存放把输入变成两倍再加 5 的函数,第 4 个桶中存放的函数看不出输入和输出之间的简单关系。这 4 个函数对应到前 4 个桶中,如图 15.4 所示。

1		2		3		4		5
$x$	$F_1(x)$	$x$	$F_2(x)$	$x$	$F_3(x)$	$x$	$F_4(x)$	
1	1	1	1	1	7	1	15	
2	1	2	2	2	9	2	1	
3	1	3	3	3	11	3	7	
4	1	4	4	4	13	4	13	
5	1	5	5	5	15	5	2	
6	1	6	6	6	17	6	7	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

图 15.4 把函数对应到桶中

能把每一个函数都对应到桶中吗? 答案是否定的,因为总有办法创建一个新函数,这个函数无法对应到任何桶中。假如有人提出一种把函数对应到桶中的方式,并且声称可以包括所有函数,我们可以按照下面的方法建立一个新函数,这个函数不能对应到任何桶中。从第 1 个桶中的函数得到输入值 1 的输出值,把这个值称为  $F_1(1)$ ,把它加 1,并把结果作为新函数输入值 1 的输出值。不管分配给新函数的其余值是多少,它一定不同于表中的第 1 个函数,因为这两个函数对于输入值 1 会得到不同的输出值。现在从表中的第 2 个函数得到输入值 2 的输出值(称之为  $F_2(2)$ ),这样,我们的新函数一定不同于第 2 个函数,因为它们至少对于输入值 2 有不同的输出值。继续下去,对于所有的输入值  $i$ ,  $F_{\text{neu}}(i) = F_i(i) + 1$ ,这样,新函数至少在第  $i$  个位置不同于某

个函数  $F_i$ 。由于新函数不同于每一个其他函数,它一定不在表中。不管我们怎样把函数对应到桶中,这都是正确的,因此整数函数集合是不可数的。它的重要性是并不是所有函数都可以对应于一个程序,一定有函数对应不到任何程序。图 15.5 说明了这个问题。

1		2		3		4		5	
$x$	$F_1(x)$	$x$	$F_2(x)$	$x$	$F_3(x)$	$x$	$F_4(x)$	$x$	$F_{new}(x)$
1	①	1	1	1	7	1	15	1	2
2	1	2	②	2	9	2	1	2	3
3	1	3	3	3	⑪	3	7	3	12
4	1	4	4	4	13	4	⑬	4	14
5	1	5	5	5	15	5	2	5	→
6	1	6	6	6	17	6	7	6	→
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

图 15.5 整数函数不可数图示

### 15.4.2 停机问题的不可解性

尽管知道存在一些函数无法通过计算机程序计算出来,这在理论上是很有趣的,但这是否表明存在一些有用的问题但无法计算出来呢? 现在我们证明停机问题不可能通过任何计算机程序解决,证明采用反证法。

首先假定有一个名为 `halt` 的函数可以解决停机问题。很显然,不可能编写出一些不存在的东西。如果确实存在解决停机问题的函数,这里就是它的一个合理框架。函数 `halt` 有两个输入:一个代表 Java 程序或函数源代码的字符串,和另一个代表输入的字符串,我们希望能够确定输入的程序或函数是否能够停下来。如果输入的程序或函数能够对于给定的输入停下来,函数 `halt` 就返回 `true`,否则就返回 `false`。

```
bool halt(char[] prog, char[] input)
{
    Code to solve halting problem
    if (prog does halt on input) then
        return ( true );
    else
        return ( false );
}
```

现在我们考查两个简单函数,由于这里给出了完整的 Java 代码,它们显然存在:

```
boolean selfhalt (char[] prog) {
    // Return TRUE if program halts when given itself as input.
    if (halt(prog, prog))
        return ( true );
    else
        return ( false );
}
```

```
void contrary ( char[] prog ) {  
    if (selfhalt(prog))  
        while(true); // Go into an infinite loop
```

当函数 `contrary` 运行时其本身会发生什么情况呢? 一种可能性是对 `selfhalt` 的调用返回 `true`, 也就是说, `selfhalt` 表明 `contrary` 在其自身上运行时将会停机。在这种情况下, `contrary` 进入一个无限循环(从而不会停下来)。另一方面, 如果 `selfhalt` 返回 `false`, 那么 `halt` 就已经表明了 `contrary` 不会在其自身上停下来, 而 `contrary` 返回了, 也就是说它停下来了。这样一来 `contrary` 就与 `halt` 所说的矛盾。

`contrary` 的动作在逻辑上与 `halt` 能够正确解决停机问题的假设不一致, 我们并没有做出其他假设导致这种不一致, 这样, 通过反证法就证明了 `halt` 不能正确解决停机问题, 从而没有程序能够解决停机问题。

既然已经证明了停机问题不可解, 我们使用归约方法还可以证明其他问题也不可解, 其策略是假设一个计算机程序能解决要正在考虑的问题, 然后使用这个程序解决另一个已经知道不可解的问题。

例如, 考虑停机问题的下面一个变体。给定一个计算机程序, 当输入为空字符串时它会停下来吗(即没有输入时它会停下来吗)? 要证明这个问题是不可解的, 我们利用一个用于可计算性证明的标准技术: 使用一个计算机程序修改另一个计算机程序。

假定有一个函数 `Ehalt`, 它在没有输入时确定一个给定的程序是否会停机。回忆一下我们对停机问题的证明涉及这样一个函数, 它有两个参数, 一个是代表程序的字符串, 另一个是代表输入的字符串。考虑另一个函数 `combine`, 它把一个程序 `P` 和一个输入字符串 `I` 作为参数。函数 `combine` 修改程序 `P`, 把 `I` 作为它的一个静态变量 `S` 存储, 并且进一步修改对 `P` 中输入函数的所有调用, 使得它们从 `S` 中得到输入。调用结果程序 `P'`, 无需多想, 任何一个一般的编译器只要加以修改就可以取得计算机程序和输入字符串, 产生一个用这种方式修改的新计算机程序。现在, 把 `P'` 输入 `Ehalt`。如果 `Ehalt` 说 `P'` 会停下来, 那么我们就知道 `P` 对于输入 `I` 将会停下来, 也就是说, 我们现在有一个解决原来停机问题的步骤。我们所做的惟一假设就是 `Ehalt` 存在, 因此, 确定一个程序在没有输入的时候是否会停机就是不可解的。

### 15.4.3 确定程序行为是不可解的

我们希望用计算机解决的许多问题是不可解的, 这些问题中有许多与程序的行为有关。例如, 证明一个程序是“正确”的, 也就是说, 证明一个程序能够完成某一项功能是一个与程序行为有关的证明, 这样, 能够做的事就极其有限了。特别是, 不能可靠地确定某一个程序是否能完成某一项功能, 也不可能确定某个程序中的某一行代码是否已经执行过。

这并不意味着不能编写一个计算机程序处理一种特殊情况, 可能对于大多数程序, 我们感兴趣的是检测。例如, 有些 C 编译器将检查 `while` 循环的控制表达式是不是一个取值为 `false` 的常量表达式。如果是, 编译器将发出警告, `while` 循环的代码永远不会被执行。然而, 不可能编写出一个检查所有输入程序的计算机程序, 使得当这个输入程序被给予一个指定的输入值时检查一行指定的代码是否会执行。

另一个不可解的问题是一个程序中是否包含计算机病毒。“包含计算机病毒”是一种行

为,因此,不可能肯定地确定任意一个程序是否包含计算机病毒。然而,有许多好的启发式规则可以用于确定一个程序是否有可能包含计算机病毒,通常也能确定一个程序是否已经包含了某些计算机病毒,至少是现在已经知道的那些病毒。实际的病毒检测程序做得很好,但是心怀恶意的人总能开发出病毒检测程序不能够识别出来的新病毒。

## 15.5 深入学习导读

有关 NP 完全性理论的经典教材是 Michael R. Garey 和 David S. Johnston 的《Computers and Intractability: A Guide to the Theory of NP-completeness》[GJ79]。Lawler 等人编辑的《The Traveling Salesman Problem》讨论了许多方法,以对这个特定的 NP 完全性问题找到一个在合理时间内解决问题的可接受的解法[LLKS85]。

有关 Collatz 函数的更多信息,参见 B. Hayes 的“On the ups and downs of hailstone numbers”,《科学美国人》1984 年 1 月的“Computer Recreations”,和《美国数学月刊》1985 年 1 月的 J. C. Lagarias 的“The  $3x + 1$  problem and its generalizations”。

对于可计算性和不可解问题领域的介绍,参见 James L. Hein 的《Discrete Structure, Logic, and Computability》(Hei95)。

## 15.6 习题

- 15.1 考虑这样一个算法,在一个数组中找到最大元素:首先排序数组,然后选择最后(最大的)一项,从而找到最大元素。对于这个在一个序列中找到最大元素的问题,这个归约在问题的上下限方面告诉了我们什么(如果有)?为什么不能把排序问题归约为找到最大元素问题?
- 15.2 使用归约证明一个  $n \times n$  矩阵的平方和两个  $n \times n$  矩阵相乘的代价一样高。
- 15.3 使用归约证明两个上三角  $n \times n$  矩阵相乘与两个任意  $n \times n$  矩阵相乘的代价一样高。
- 15.4 (a)说明为什么利用从 1 到  $n$  的所有值相乘来计算  $n$  的阶乘是一个指数时间算法。  
(b)说明为什么利用 Stirling 公式(见 2.2 节)近似计算  $n$  的阶乘是一个多项式时间算法。
- 15.5 图  $G$  的一个汉密尔顿回路(Hamiltonian cycle)是一个从起始顶点出发,访问图中每个顶点一次,并回到起始顶点的回路。汉密尔顿回路问题问图  $G$  中是否包含一个汉密尔顿回路。假定汉密尔顿回路问题是 NP 完全性问题,证明货郎担问题也是 NP 完全性问题。
- 15.6 假定顶点覆盖问题是 NP 完全性问题,找到一个从顶点覆盖问题到 K 团问题的多项式时间归约,从而证明 K 团问题也是 NP 完全性问题。
- 15.7 假定 K 团问题是 NP 完全性问题,找到一个从 K 团问题到顶点覆盖问题的多项式时间归约,从而证明顶点覆盖问题也是 NP 完全性问题。
- 15.8 证明实数集合是不可数的,使用的方法类似于 15.4.1 小节证明整数函数集合不可

数的方法。

- 15.9 下面是背包问题的另一种形式,我们称它为精确背包(EXACT KNAPSACK)问题。给定一组物体,每一个都有给定的整数大小,背包的大小为整数  $k$ ,有没有一个物体的子集,正好可以放到背包中?假定精确背包问题是 NP 完全性问题,使用归约证明背包问题是 NP 完全性问题。
- 15.10 使用类似于 15.4.2 小节给出的归约方法证明,确定一个程序是否会打印某些输出的问题是不可解的。
- 15.11 使用类似于 15.4.2 小节给出的归约方法证明,确定一个程序是否会在程序中执行某一条语句的问题是不可解的。
- 15.12 使用类似于 15.4.2 小节给出的归约方法证明,确定两个程序是否会对同样的输入停机的问题是不可解的。
- 15.13 使用类似于 15.4.2 小节给出的归约方法证明,确定是否有某个输入会使两个程序都停机的的问题是不可解的。

## 15.7 项目设计

- 15.1 实现顶点覆盖问题,即给定图  $G$  和整数  $K$ ,回答是否有一个大小为  $K$  或者更小的顶点覆盖。开始先使用穷举算法检查大小为  $K$  的所有可能的顶点集合,以找到一个可接受的顶点覆盖,并且对一组输入的图度量运行时间。然后使用你能够想到的任何启发式规则试着减少运行时间。接下来,通过找到形成顶点覆盖的最小顶点集合来试着找到这个问题的近似解。
- 15.2 实现背包问题。对一组输入,度量其运行时间。这个问题的最大实际输入大小是多少?
- 15.3 实现货郎担问题的一个近似解法,即给定一个图  $G$  和所有边的距离代价,找到访问  $G$  中所有顶点的最小回路。试一试各种启发式规则,以对大量的输入图找到最好的近似。
- 15.4 编写一个程序,给定一个正整数  $n$  作为输入,打印出这个数的 Collatz 序列。对于具有很长 Collatz 序列的整数,你可以发现什么?对于各组整数的 Collatz 序列长度,你可以发现什么?

## 附录 A C 和 Pascal 程序员的 Java 导引

本附录为 C 和 Pascal 程序员提供 Java 语言的简单引导性介绍。这个导引并不是教你如何使用 Java 语言进行编程,它只是提供足够的背景知识,使你能理解本书的代码例子。

导引由第 4 章的 3 个例子组成。每一节都从代码开始,接着是要介绍的新 Java 语法的描述。一旦你理解了这些例子中包含的 Java 语法,就应当能理解本书中使用的所有 Java 代码了。

第一个例子提供了 List 接口,一个 Java 接口可以用于描述一个 ADT,这里不提供线性表接口的实现。第二个例子是线性表接口基于数组的部分实现。第三个例子中包含 Link 类和从链表实现中选择出来的一些成员函数。

在开始介绍例子代码之前,我们首先从 Java 和类的一些术语开始。Java 是一个面向对象程序设计语言,熟悉 C++ 语言的读者会发现 Java 很容易学习,因为 Java 语言是比 C++ 更纯的面向对象程序设计语言,Java 还是一个非常小的语言,它的大多数语法来源于 C,尽管 Java 语言决不是 C 语言的一个超集或子集。C 程序员读 Java 程序最需要学习的是类的语法。

从表面上看,一个类就像 C 语言中的一个结构或者 Pascal 语言中的一条记录,然而,类的内涵要比它们丰富得多。类中包含成员(member),有两种类型成员:数据成员(变量)和成员函数。成员函数完成与类相关的操作。Java 中的每个函数都是某个类的成员,其中没有与任何类都无关的自由函数。Java 程序只是一组类的集合,其中某个类的成员函数是“主”例程。本书中包括的几乎所有代码例子都是类或者类成员函数的实现。

### A.1 例 1:线性表的接口

```
1  public interface List :           // List ADT
2      public void clear();           // Remove all Objects
3      public void insert(Object item); // Insert Object at curr
4      public void append(Object item); // Insert Object at tail
5      public Object remove();         // Remove/return Object
6      public void setFirst();         // Set current to first pos
7      public void next();             // Move current to next pos
8      public void prev();             // Move current to prev pos
9      public int length();            // Return current length
10     public void setPos(int pos);     // Set current to pos
11     public void setValue(Object val); // Set current Object
12     public Object currValue();       // Return value of Object
13     public boolean isEmpty();        // true if list is empty
14     public boolean is IList();       // True if within list
15     public void print();             // Print list elements
16     : // interface List
```

我们的第一个例子给出了接口。接口并不实际负责 Java 类的实现,而是标识声明要实现这个接口的类必须具有的一些函数及其相关参数类型。这里用接口定义线性表的 ADT。

第 1 行标识这是一个接口,它的名字是 List,这个接口是公有的,也就是说,Java 程序的所有部分都知道 List 接口的存在。第 1 行还说明了 Java 注释的语法。// 符号右边的任何文字都是注释,编译器会自动忽略。

第 2 行到第 15 行给出了一组成员函数,声明实现 List 接口的任何类都需要实现这些成员函数。这些成员函数中的每一个都声明为公有的,意即 List 对象的所有用户都可以使用它们。第 2 行的函数 clear 说明把 void 作为返回类型,这表示它不会返回任何值。函数 clear 也没有任何输入参数。

函数 Insert 也没有返回值,但是它有一个参数,这个参数的类型是 Object。Object 是 Java 语言中最一般的类型,这意味着这个参数实际上可以是任何类,因为所有类都是 Object 类的子类,并且一个实际参数的类型可以是一个正规参数类型的子类。

第 13 行的函数 isEmpty 有一个类型为 boolean 类型的返回值,这个类型是 Java 的布尔类型。

## A.2 例 2:基于数组的线性表实现

```

1  class AList implements List {                               // Array-based list
2      private static final int defaultSize = 10;
3      private int msize;                                       // Maximum size of list
4      private int numInList;                                    // Actual number in list
5      private int curr;                                         // Position of current Object
6      private Object[] listArray;                              // Array holding list Objects
7
8      AList() { setup(defaultSize); }                           // Constructor: fixed size
9
10     AList(int sz) { setup(sz); }                               // Constructor: default size
11
12     private void setup(int sz) {                               // Do actual initialization
13         msize = sz;
14         numInList = curr = 0;
15         listArray = new Object[sz];                           // Create listArray
16     }
17
18     public void clear()                                         // Remove all Objects from list
19     { numInList = curr = 0; }                                   // reinitialize values
20
21     public void insert(Object it) {                             // Insert Object
22         Assert.notFalse(numInList < msize, "List is full");
23         Assert.notFalse((curr >= 0) && (curr <= numInList),
24             "Bad value for curr");

```

```

25     for (int i=numInList; i>curr; i--)      // Shift Objects
26         listArray[i] = listArray[i-1];
27     listArray[curr] = it;
28     numInList++;                          // Increment current size
29     !
30
31     public Object currValue() {             // Return current Object
32         Assert.assertFalse(isInList(), "No current element");
33         return listArray[curr];
34     !
35
36     public void print() {                   // Print all list's Objects
37         if (isEmpty()) System.out.println("");
38         else {
39             System.out.print("( ");
40             for (setFirst(); isInList(); next())
41                 System.out.print(currValue() + " ");
42             System.out.println(")");
43         }
44     }
45     { // class AList

```

这个例子有选择地提供了基于数组的线性表的部分实现。它的形式是名为 AList 的 Java 类。

第 1 行表明定义了一个名为 AList 的 Java 类,这个类实现了名为 List 的接口。这里,“实现”的意思是 AList 中包含了 List 接口标识的函数,而且函数的参数和返回类型都是正确的。

第 2 行到第 6 行声明了这个类的私有数据成员。私有数据成员由定义的类使用,而不能由其他类使用。这样,这些内容就被保护起来了,从而防止了程序中其他部分的不适当访问。

第 2 行给出了一个常量值的声明。关键字 static 表明在所有 AList 对象中,这个变量只有惟一一份副本。在一般情况下,对于数据变量,每个 AList 对象都有它自己的副本。但是,成员 defaultSize 只创建一次,由各个 AList 对象变量共享。关键字 final 表明这个变量的值以后在程序中不能被修改。

第 3 行到第 6 行声明了这个类的 4 个常规数据成员。对于这里的每一个数据成员,每个 AList 对象都有它自己的副本。前 3 个数据成员是整型量,第 4 个数据成员是 1 个 Object 类型的数组。注意,第 6 行的变量声明并不为 listArray 实际声明任何空间,实际工作在后面进行。

第 8 行和第 10 行声明这个类的两个构造函数(constructor)。你可以分辨出它们是构造函数,因为它们的名字和类的名字一样,而且没有返回类型。当创建一个对象的新实例时,就调用这个对象的构造函数。可以使用 new 操作符在空闲存储区动态创建一个对象(我们将深入研究一个这样的例子)。在这个例子中,每当一个 AList 对象创建时,构造函数就为数组分配空间。

这里的构造函数有两个版本。这是一个重载(overloading)的例子,可以通过这种方式重



载任何一个成员函数,只要各成员函数的参数形式互不相同。只有各成员函数的参数形式互不相同,编译器才能区分出一次函数调用引用的是哪一个版本。

构造函数的一个版本有一个整型参数,而另一个则没有。对于有参数的版本,参数标识要分配数组的大小;没有参数的版本把数组大小设为 defaultSize。这两个版本的函数都调用一个私有成员函数 setup,这个函数的函数体在第 12 行到第 16 行。只有通过 AList 类的其他成员函数才能访问 setup 函数,它完成初始化 AList 对象的实际工作。

第 15 行利用 new 操作符为 listArray 数组分配空间,这一行表示将给数组 sz 个 Object 类型的引用。引用类似于 C 语言或者 Pascal 语言中的指针。注意,这一行并不实际分配一组 Object 类型的变量,而是一组指向 Object 类型变量的指针。最初,这些指针被设置为 null。当线性表被填入数组元素,这些指针将指向实际数据。通过使类型为 Object,我们使用了可能的最一般的类型。实际数据元素可能是 Object 类型的任何一个子类,实际上是任何类。

函数 setup 的第 13 行把一个值赋给变量 msize,变量 msize 没有在这个函数中声明。这样,它就去引用 AList 类的数据成员 msize。

函数 insert 的第 22 行和第 23 行表明了调用某个类的一个成员函数的语法。正常的语法是“object.funcname(params)”,其中 object 是类的某个变量。第 22 行和第 23 行给出了这个一般语法的一个特殊情况。Java 没有独立于某个类的成员函数的函数概念,这里,类是 Assert,其成员函数是 notFalse。Assert 只是类,不是这个类的对象的一个实例(也不是一个变量)。可能有人会要求为类 Assert 分配一个变量,再去调用它的一个成员函数,这样做是很不明智的。因此,Java 允许程序员声明类型为 static 的成员函数,表示可以使用类名访问它,而不需要这个类的一个变量。

从第 8 行到第 10 行对 setup 的调用看起来违反了刚刚讨论的调用类成员函数的语法。就像第 13 行对变量 msize 的使用被认为是引用类 AList 的成员一样,第 8 行和第 10 行对 setup 的引用也是一样。

从第 36 行到第 44 行的 print 函数说明了编写输出部分的语法。第 37 行、第 41 行和第 42 行对函数 System.out.print() 和 System.out.println() 的调用是从某个类中调用静态成员函数的进一步的例子(这里是 Java 标准包的 out 类)。这些函数有一个类型为 String 的参数, System.out.println() 把一个新行字符放到字符串的最后。第 41 行表明了类型转换和字符串连接的使用。函数 currValue 返回对一个 Object 类型对象的引用,它认为对象有某个字符串表示。编译器为没有定义的对象给予一个缺省的字符串表示,第 41 行的“+”表示将把一块空间连接到 currValue 产生的字符串中。

### A.3 例 3:链表的实现

```

1  class Link {                                // A singly linked list node
2      private Object element;                  // Object for this node
3      private Link next;                      // Pointer to next node in list
4      Link(Object it, Link nextval)           // Constructor 1
5          { element = it; next = nextval; }    // Given Object
6      Link(Link nextval) { next = nextval; }   // Constructor 2

```

```

7      Link next() { return next; }
8      Link setNext(Link nextval) { return next = nextval; }
9      Object element() { return element; }
10     Object setElement(Object it) { return element = it; }
11     ; // class Link
12
13     //////////////////////////////////////
14
15     public class LList implements List { // Linked list class
16         private Link head;                // Pointer to list header
17         private Link tail;                // Pointer to last Object
18         protected Link curr;             // position of current Object
19
20         LList(int sz) { setup(); }         // Constructor -- Ignore sz
21         LList() { setup(); }              // Constructor
22
23         private void setup()
24         { tail = head = curr = new Link(null); } // Create header
25
26         public void clear() {              // Remove all Objects
27             head.setNext(null);
28             curr = tail = head;            // Reinitialize
29         }
30
31         // Insert Object at current position
32         public void insert(Object it) {
33             Assert.notNull(curr, "No current element");
34             curr.setNext(new Link(it, curr.next()));
35             if (tail == curr)              // Appended new Object
36                 tail = curr.next();
37         }
38
39         public Object remove() {           // Remove and return Object
40             if (! isInList()) return null;
41             Object it = curr.next().element(); // Remember value
42             if (tail == curr.next()) tail = curr; // Set tail
43             curr.setNext(curr.next().next()); // Remove from list
44             return it;                     // Return value
45         }
46     } // class LList

```

这个例子说明了两个类：Link 类用于链表的结点，LList 类的一部分用于实现链表。这个例子只提供了 Java 语法的一些新实例。

类 `LList` 被定义为公有的,而类 `Link` 却不是。这里的想法是链表的使用者需要访问类 `LList`,而类 `Link` 只是类 `LList` 的协助者,不需要是公有的。根据如何具体把类组织到文件中,可以采用 Java 的包机制对外部用户隐藏 `Link` 类的存在,隐藏的具体技术细节超出了本导引的范围。

第 18 行是一个声明为 `protected` 的数据成员。受保护成员的状态是公有和私有之间的一个成员状态,特别地,只有类 `LList` 派生的成员才可以使用这个变量。变量 `curr` 被声明为受保护成员,因为第 7 章图的邻接表表示法的实现使用了链表类的一个扩展,名为 `GraphList`。

第 24 行是典型的使用 `new` 操作符为对象进行空间分配。再次注意,声明某个类的一个变量并不为这种类型实际分配空间。变量只是对对象的一个引用。对 `new` 的一次调用将得到需要的空间。在第 24 行,创建了一个新的 `Link` 对象(使用第 6 行的 `Link` 类构造函数),三个名为 `tail`、`head`、`curr` 的引用都被赋值指向这个新的 `Link` 对象。

第 27 行是对一个对象的成员函数的典型访问。这里调用了 `Link` 类的成员函数 `setNext`,其语法是 `object.funcname(params)`。这里,对象是名为 `head` 的 `Link` 变量。这个例子中的参数是“空”指针值。

第 41 行也是对类成员的一个调用,但这里是一组调用。`curr.next().element()` 中有“.”操作符的两个实例,它从左到右进行处理。这样,激活 `curr.next()` 返回一个 `Link` 类的引用,然后调用这个被返回的 `Link` 类引用的 `element` 函数。

## 参 考 文 献

- [Aad79] James L. Adams. *Conceptual Blockbusting*. Norton, New York, second edition, 1979.
- [AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.
- [AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.
- [App85] Apple Computer, Inc. *Inside Macintosh*, volume I. Addison-Wesley, Reading, MA, 1985.
- [BB88] G. Brassard and P. Bratley. *Algorithmics: Theory and Practice*. Prentice Hall, Englewood Cliffs, NJ, 1988.
- [Ben75] John Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509-517, September 1975. ISSN: 0001-0782.
- [Ben82] John Louis Bentley. *Writing Efficient Programs*. Prentice Hall, Englewood Cliffs, NJ, 1982.
- [Ben84] John Louis Bentley. Programming pearls: The back of the envelope. *Communications of the ACM*, 27(3):180-184, March 1984.
- [Ben85] John Louis Bentley. Programming pearls: Thanks, heaps. *Communications of the ACM*, 28(3):245-250, March 1985.
- [Ben86a] John Bentley. *Programming Pearls*. Addison-Wesley, Reading, MA, 1986.
- [Ben86b] John Louis Bentley. Programming pearls: The envelope is back. *Communications of the ACM*, 29(3):176-182, March 1986.
- [Ben88] John Bentley. *More Programming Pearls: Confessions of a Coder*. Addison-Wesley, Reading, MA, 1988.
- [BM85] John Louis Bentley and Catherine C. McGeoch. Amortized analysis of self-organizing sequential search heuristics. *Communications of the ACM*, 28(4):404-411, April 1985.
- [Bro75] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA, 1975.
- [BSTW86] John Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320-330, April 1986.
- [CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.
- [Com79] Douglas Comer. The ubiquitous b-tree. *Computing Surveys*, 11(2):121-137, June 1979.

- 1979.
- [Dei90] Harvey M. Deitel. *Operating Systems*. Addison-Wesley, Reading, MA, second edition, 1990.
- [ECW92] Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *Computing Surveys*, 24(4):441-476, December 1992.
- [ED88] R. J. Enbody and H. C. Du. Dynamic hashing schemes. *Computing Surveys*, 20(2):85-113, June 1988.
- [Epp95] Susanna S. Epp. *Discrete Mathematics with Applications*. Wadsworth Publishing Company, Belmont, CA, second edition, 1995.
- [FBY92] W. B. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures & Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1992.
- [FF89] Daniel P. Friedman and Matthias Felleisen. *The Little LISP*. Macmillan Publishing Company, New York, 1989.
- [FHCD92] Edward A. Fox, Lenwood S. Heath, Q. F. Chen, and Amjad M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105-121, January 1992.
- [Fla96] David Flanagan. *Java in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, 1996.
- [FZ92] M. J. Folk and B. Zoellick. *File Structures: A Conceptual Toolkit*. Addison-Wesley, Reading, MA, second edition, 1992.
- [GI91] Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *Computing Surveys*, 23(3):319-344, September 1991.
- [GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.
- [GKP89] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, MA, 1989.
- [Gla93] Graham Glass. *UNIX for Programmers and Users*. Prentice Hall, Englewood Cliffs, NJ, 1993.
- [Gle92] James Gleick. *Genius: The Life and Science of Richard Feynman*. Vintage, New York, 1992.
- [Hei95] James L. Hein. *Discrete Structures, Logic, and Computability*. Jones and Bartlett, Sudbury, MA, 1995.
- [Jay90] Julian Jaynes. *The Origin of Consciousness in the Breakdown of the Bicameral Mind*. Houghton Mifflin, Boston, MA, 1990.
- [Knu73] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, MA, second edition, 1973.
- [Knu81] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA, second edition, 1981.
- [Knu94] Donald E. Knuth. *The Stanford GraphBase*. Addison-Wesley, Reading, MA, 1994.

- 
- [KP78] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, New York, second edition, 1978.
- [Lam93] Leslie Lamport. How to write a proof. Technical Report 94, DEC Systems Research Center, February 1993.
- [LLKS85] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley & Sons, New York, 1985.
- [LP81] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, Englewood Cliffs, NJ, 1981.
- [Man89] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, Reading, MA, 1989.
- [Pó157] George Pólya. *How To Solve It*. Princeton University Press, Princeton, NJ, second edition, 1957.
- [Pug90] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668-676, June 1990.
- [Raw92] Gregory J. E. Rawlins. *Compared to What? An Introduction to the Analysis of Algorithms*. Computer Science Press, New York, 1992.
- [Rob84] Fred S. Roberts. *Applied Combinatorics*. Prentice Hall, Englewood Cliffs, NJ, 1984.
- [Rob86] Eric S. Roberts. *Thinking Recursively*. John Wiley & Sons, New York, 1986.
- [RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17-28, March 1994.
- [Sal88] Betty Salzberg. *File Structures: An Analytic Approach*. Prentice Hall, Englewood Cliffs, NJ, 1988.
- [Sam90a] Hanan Samet. *Applications of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.
- [Sam90b] Hanan Samet. *Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.
- [SB93] Clifford A. Shaffer and Patrick R. Brown. A paging scheme for pointer-based quadrees. In D. Abel and B-C. Ooi, editors, *Advances in Spatial Databases*, pages 89-104, Springer Verlag, Berlin, June 1993.
- [Sed80] Robert Sedgewick. *Quicksort*. Garland Publishing, Inc. New York, 1980.
- [Sed88] Robert Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, second edition, 1988.
- [Sel95] Kevin Self. Technically speaking. *IEEE Spectrum*, 32(2):59, February 1995.
- [SJH93] Clifford A. Shaffer, Ramana Juvvadi, and Lenwood S. Heath. A generalized comparison of quadtree and bintree storage requirements. *Image and Vision Computing*, 11(7):402-412, September 1993.
- [SM83] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw Hill, New York, 1983.
- [Sol90] Daniel Solow. *How to Read and Do Proofs*. John Wiley & Wiley, New York, second

- edition, 1990.
- [ST85] D. D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652-686, 1985.
- [Sta94] Richard M. Stallman. *GNU Emacs Manual*. Free Software Foundation, Cambridge, MA, tenth edition, July 1994.
- [Ste84] Guy L. Steele. *Common Lisp: The Language*. Digital Press, Bedford, MA, 1984.
- [Sto88] James A. Storer. *Data Compression: Methods and Theory*. Computer Science Press, Rockville, MD, 1988.
- [Tan90] Andrew S. Tanenbaum. *Structured Computer Organization*, Prentice Hall, Englewood Cliffs, NJ, third edition, 1990.
- [Tar75] Robert E. Tarjan. Efficiency of a good but not linear set merging algorithm. *Journal of the ACM*, 22(2):215-225, April 1975.
- [TRE88] Pete Thomas, Hugh Robinson, and Judy Emms. *Abstract Data Types: Their Specification, Representation, and Use*. Clarendon Press, Oxford, 1988.
- [Wel88] Dominic Welsh. *Codes and Cryptography*. Oxford University Press, Oxford, 1988.
- [WN96] Patrick Henry Winston and Sundar Narasimhan. *On to Java*. Addison-Wesley, Reading, MA, 1996.

[ G e n e r a l   I n f o r m a t i o n ]

书名= 数据结构与算法分析 ( J a v a 版 )

作者=

页数= 3 2 2

S S 号= 0

出版日期=

V s s 号= 6 1 0 7 9 8 0 3



封面页  
书名页  
版权页  
前言页  
目录页

第一部分 预备知识

第 1 章 数据结构和算法

- 1 . 1 数据结构的原则
  - 1 . 1 . 1 学习数据结构的必要性
  - 1 . 1 . 2 代价与效益
  - 1 . 1 . 3 本书的目的
- 1 . 2 抽象数据类型和数据结构
- 1 . 3 问题、算法和程序
- 1 . 4 算法的效率
- 1 . 5 深入学习导读
- 1 . 6 习题

第 2 章 数学预备知识

- 2 . 1 集合
- 2 . 2 常用数学术语
- 2 . 3 对数
- 2 . 4 递归
- 2 . 5 级数求和与递归
- 2 . 6 数学证明方法
  - 2 . 6 . 1 反证法
  - 2 . 6 . 2 数学归纳法
- 2 . 7 评估
- 2 . 8 深入学习导读
- 2 . 9 习题

第 3 章 算法分析

- 3 . 1 概述
- 3 . 2 最佳、最差和平均情况
- 3 . 3 换一台更快的计算机，还是换一种更快的算法
- 3 . 4 渐进分析
  - 3 . 4 . 1 上限
  - 3 . 4 . 2 下限
  - 3 . 4 . 3 表示法
  - 3 . 4 . 4 化简法则
- 3 . 5 程序运行时间的计算
- 3 . 6 问题的分析
- 3 . 7 多参数问题
- 3 . 8 空间代价
- 3 . 9 实际操作中的一些因素
- 3 . 1 0 深入学习导读
- 3 . 1 1 习题
- 3 . 1 2 项目设计

第二部分 基本数据结构

第 4 章 线性表、栈和队列

- 4 . 1 线性表
  - 4 . 1 . 1 顺序表的表示法
  - 4 . 1 . 2 链表
  - 4 . 1 . 3 线性表实现方法的比较
  - 4 . 1 . 4 元素的表示
  - 4 . 1 . 5 双链表
  - 4 . 1 . 6 循环链表
- 4 . 2 栈
  - 4 . 2 . 1 顺序栈
  - 4 . 2 . 2 链式栈
  - 4 . 2 . 3 顺序栈与链式栈的比较
  - 4 . 2 . 4 递归的实现
- 4 . 3 队列

		4 . 3 . 1	顺序队列
		4 . 3 . 2	链式队列
		4 . 3 . 3	顺序队列与链式队列的比较
	4 . 4		习题
	4 . 5		项目设计
第 5 章	二叉树		
	5 . 1		定义及主要特性
		5 . 1 . 1	满二叉树定理
		5 . 1 . 2	二叉树的抽象数据类型
	5 . 2		周游二叉树
	5 . 3		二叉树的实现
		5 . 3 . 1	使用指针实现二叉树
		5 . 3 . 2	空间开销
		5 . 3 . 3	使用数组实现完全二叉树
	5 . 4		H u f f m a n 编码树
		5 . 4 . 1	建立 H u f f m a n 编码树
		5 . 4 . 2	H u f f m a n 编码及其用法
	5 . 5		二叉检索树
	5 . 6		堆与优先队列
	5 . 7		深入学习导读
	5 . 8		习题
	5 . 9		项目设计
第 6 章	树		
	6 . 1		树的定义与术语
		6 . 1 . 1	树结点的 A D T ( 抽象数据类型 )
		6 . 1 . 2	树的周游
	6 . 2		父指针表示法
	6 . 3		树的实现
		6 . 3 . 1	子结点表表示法
		6 . 3 . 2	左子结点 / 右兄弟结点表示法
		6 . 3 . 3	动态结点表示法
		6 . 3 . 4	动态 “ 左子结点 / 右兄弟结点 ” 表示法
	6 . 4		K 叉树
	6 . 5		树的顺序表示法
	6 . 6		深入学习导读
	6 . 7		习题
	6 . 8		项目设计
第 7 章	图		
	7 . 1		术语和表示法
	7 . 2		图的实现
	7 . 3		图的周游
		7 . 3 . 1	深度优先搜索
		7 . 3 . 2	广度优先搜索
		7 . 3 . 3	拓扑排序
	7 . 4		最短路径问题
		7 . 4 . 1	单源最短路径
		7 . 4 . 2	每对顶点间的最短路径
	7 . 5		最小支撑树
		7 . 5 . 1	P r i m 算法
		7 . 5 . 2	K r u s k a l 算法
	7 . 6		深入学习导读
	7 . 7		习题
	7 . 8		项目设计
第三部分	排序和检索		
第 8 章	内排序		
	8 . 1		排序的术语及记号
	8 . 2		三种代价为 $(n^2)$ 的排序方法
		8 . 2 . 1	插入排序
		8 . 2 . 2	起泡排序
		8 . 2 . 3	选择排序

	8 . 2 . 4	交换排序算法的时间代价
	8 . 3	S h e l l 排序
	8 . 4	快速排序
	8 . 5	归并排序
	8 . 6	堆排序
	8 . 7	分配排序和基数排序
	8 . 8	对各种排序算法的实验比较
	8 . 9	排序问题的下限
	8 . 1 0	深入学习导读
	8 . 1 1	习题
	8 . 1 2	项目设计
第 9 章		文件管理和外排序
	9 . 1	主存储器和辅助存储器
	9 . 2	磁盘和磁带驱动器
	9 . 2 . 1	磁盘访问的代价
	9 . 2 . 2	磁带
	9 . 3	缓冲区和缓冲池
	9 . 4	程序员的文件视图
	9 . 5	外部排序
	9 . 6	外部排序的简单方法
	9 . 7	置换选择排序
	9 . 8	多路归并
	9 . 9	深入学习导读
	9 . 1 0	习题
	9 . 1 1	项目设计
第 1 0 章		检索
	1 0 . 1	检索已排序的数组
	1 0 . 2	自组织线性表
	1 0 . 3	集合的检索
	1 0 . 4	散列方法
	1 0 . 4 . 1	散列函数
	1 0 . 4 . 2	开散列方法
	1 0 . 4 . 3	闭散列方法
	1 0 . 5	深入学习导读
	1 0 . 6	习题
	1 0 . 7	项目设计
第 1 1 章		索引技术
	1 1 . 1	线性索引
	1 1 . 2	I S A M
	1 1 . 3	树形索引
	1 1 . 4	2 - 3 树
	1 1 . 5	B 树
	1 1 . 5 . 1	B + 树
	1 1 . 5 . 2	B 树分析
	1 1 . 6	深入学习导读
	1 1 . 7	习题
	1 1 . 8	项目设计
第四部分		应用与高级话题
第 1 2 章		线性表和数组高级技术
	1 2 . 1	跳跃表
	1 2 . 2	广义表
	1 2 . 3	矩阵的表示方法
	1 2 . 4	存储管理
	1 2 . 4 . 1	动态存储分配
	1 2 . 4 . 2	失败处理策略和无用单元收集
	1 2 . 5	深入学习导读
	1 2 . 6	习题
	1 2 . 7	项目设计
第 1 3 章		高级树形结构
	1 3 . 1	T r i e 结构

	1 3 . 2	伸展树
	1 3 . 3	空间数据结构
	1 3 . 3 . 1	k - d 树
	1 3 . 3 . 2	P R 四分树
	1 3 . 3 . 3	其他空间数据结构
	1 3 . 4	深入学习导读
	1 3 . 5	习题
	1 3 . 6	项目设计
第 1 4 章		分析技术
	1 4 . 1	求和技术
	1 4 . 2	递归关系
	1 4 . 2 . 1	估计上下限
	1 4 . 2 . 2	扩展递归
	1 4 . 2 . 3	分治法递归
	1 4 . 2 . 4	快速排序平均情况分析
	1 4 . 3	均摊分析
	1 4 . 4	深入学习导读
	1 4 . 5	习题
	1 4 . 6	项目设计
第 1 5 章		计算的限制
	1 5 . 1	简介
	1 5 . 2	归约
	1 5 . 3	难解问题
	1 5 . 3 . 1	N P 完全性
	1 5 . 3 . 2	绕过 N P 完全性问题
	1 5 . 4	不可解问题
	1 5 . 4 . 1	不可数性
	1 5 . 4 . 2	停机问题的不可解性
	1 5 . 4 . 3	确定程序行为是不可解的
	1 5 . 5	深入学习导读
	1 5 . 6	习题
	1 5 . 7	项目设计
附录 A		C 和 P a s c a l 程序员的 J a v a 导引
	A . 1	例 1 : 线性表的接口
	A . 2	例 2 : 基于数组的线性表实现
	A . 3	例 3 : 链表的实现
参考文献		
附录页		