普通高等教育“十一五”国家级规划教材

高等学校计算机科学与技术专业系列教材

数据结构与算法

张铭　王腾蛟　赵海燕　编著

高等教育出版社
Higher Education Press

高等学校计算机科学与技术专业系列教材

依据教育部高等学校计算机科学与技术教学指导委员会编制的高等学校计算机科学与技术专业规范和IEEE/ACM制订的CC2001－CC2005有关教程编写。

数据结构与算法

本书特色：

- 系统介绍各种基本数据结构的概念、原理和操作方法，并从算法的角度讨论排序和检索算法。
- 以问题求解为主线，从问题抽象、数据抽象和算法抽象的角度组织数据结构与算法的设计。
- 注重将理论内容结合应用背景，加强上机动手实践能力训练，着重培养学生的抽象思维能力与问题建模能力，使其学会如何根据实际问题对数据结构和算法进行取舍，在时间复杂度和空间复杂度之间进行平衡。
- 提供PPT电子教案、网络课程、立体化教学网站等教学资源，与本书配套的实习课教材《数据结构与算法实验教程》将于2009年出版。

ISBN 978-7-04-023961-4
9 787040 239614 >
定价 31.00 元

普通高等教育“十一五”国家级规划教材

高等学校计算机科学与技术专业系列教材

数据结构与算法

张　铭　王腾蛟　赵海燕　编著

高等教育出版社

内容提要

本书是普通高等教育“十一五”国家级规划教材，也是北京市精品课程主讲教材。本书按照 IEEE/ACM CC2005 和教育部教指委关于“计算机科学与技术专业规范”（CCC2005）的要求编写，力求使学生较全面地理解数据结构的概念、掌握各种数据结构与算法的实现方式，同时比较不同数据结构和算法的特点，重点强调实践教学和学生动手能力的培养。

本书的内容涉及基本数据结构、排序、索引、检索、高级数据结构等内容，借助抽象数据类型，从逻辑结构的角度系统地介绍线性表、字符串、二叉树、树和图等各种基本数据结构；从算法的角度系统地介绍各类排序、检索和索引算法；从应用的角度介绍一些更复杂的数据结构与算法分析技术。本书采用能够更自然体现抽象数据类型概念的 C++ 语言作为算法描述语言，注意对每一种数据结构的不同存储方法及相关算法进行比较分析。很多算法使用了参数化的模板，从而提高了算法中数据类型的通用性，支持高效的代码重用。

本书概念清楚，逻辑性强，内容新颖，可作为普通高校计算机科学与技术专业学生的教材和参考书，也可作为参加计算机科学与技术学科硕士/博士生、软件工程硕士生入学考试的考试参考书，还可供计算机应用技术和电子学等理科专业的学生参考。

图书在版编目（CIP）数据

数据结构与算法/张铭，王腾蛟，赵海燕编著. —北京：高等教育出版社，2008.6
ISBN 978-7-04-023961-4

Ⅰ.数… Ⅱ.①张…②王…③赵… Ⅲ.①数据结构-高等学校-教材②算法分析-高等学校-教材 Ⅳ.TP311.12

中国版本图书馆CIP数据核字（2008）第059328号

策划编辑 倪文慧 **责任编辑** 彭立辉 **封面设计** 于文燕 **责任绘图** 尹 莉
版式设计 马敬茹 **责任校对** 王 超 **责任印制** 韩 刚

出版发行	高等教育出版社	**购书热线**	010-58581118
社　　址	北京市西城区德外大街4号	**免费咨询**	800-810-0598
邮政编码	100120	**网　　址**	http://www.hep.edu.cn
总　　机	010-58581000		http://www.hep.com.cn
		网上订购	http://www.landraco.com
经　　销	蓝色畅想图书发行有限公司		http://www.landraco.com.cn
印　　刷	北京鑫丰华彩印有限公司	**畅想教育**	http://www.widedu.com
开　　本	787×1092　1/16	**版　　次**	2008年6月第1版
印　　张	25	**印　　次**	2008年6月第1次印刷
字　　数	560 000	**定　　价**	31.00元

本书如有缺页、倒页、脱页等质量问题，请到所购图书销售部门联系调换。

版权所有　侵权必究

物料号　23961-00

高等学校计算机科学与技术系列
教材编审委员会

主　任:李　未
副主任:傅育熙　王志英　齐治昌　陈　平　蒋宗礼　马殿富
委　员:(按姓氏笔画为序)
王　戟(国防科学技术大学)
宁　洪(国防科学技术大学)
刘　强(清华大学)
孙吉贵(吉林大学)
庄越挺(浙江大学)
何炎祥(武汉大学)
何钦铭(浙江大学)
张晨曦(同济大学)
李宣东(南京大学)
李晓明(北京大学)
陈　钟(北京大学)
陈道蓄(南京大学)
周立柱(清华大学)
周傲英(华东师范大学)
孟祥旭(山东大学)
岳丽华(中国科学技术大学)
罗军舟(东南大学)
姚淑珍(北京航空航天大学)
胡事民(清华大学)
骆　斌(南京大学)
徐宝文(东南大学)
黄虎杰(哈尔滨工业大学)
蒋建伟(上海交通大学)
廖明宏(哈尔滨工业大学)
熊　璋(北京航空航天大学)
樊晓桠(西北工业大学)

序

计算机和通信技术的迅猛发展，不仅形成了融合度最高、潜力最大、增长最快的信息产业，而且成为推动全球经济快速增长和全面变革的关键因素。进入21世纪，我国的信息产业虽然已取得了长足的发展，但与发达国家相比，还有不小的差距。国家信息化的发展和信息产业国际竞争能力的提高，迫切需要高素质、创新型的计算机专业人才。

高素质计算机专业人才的培养离不开高质量的计算机教育。我们的专业虽然机会多，处于非常有利的条件，但是我们同样面临着一件事，就是从规模发展向质量提高的转变。怎么提高质量？专业素质的教育和应用素质的训练非常重要。尤其是我国高等教育进入大众化发展阶段，社会对计算机专业人才呈现出了多样化的需求。而与此同时，计算机学科的发展已极大地突破了原有的学科体系框架，形成了在“计算机科学与技术”之下向多个专业方向发展的新格局。在这种背景下，教育部高等学校计算机科学与技术教学指导委员会编制了《高等学校计算机科学与技术专业发展战略研究报告暨专业规范（试行）》（以下简称“专业规范”）。专业规范按照“培养规格分类”的指导思想，提出了3种类型、4个方向，即科学型（计算机科学方向）、工程型（计算机工程方向、软件工程方向）、应用型（信息技术方向）的计算机专业发展建议，体现了社会对不同人才类型的需求，对于指导我国计算机教学改革与建设，规范计算机教学工作，促进计算机教学质量的提高都具有重要的意义。

高水平的教材是一流教育质量的重要保证。为了配合专业规范的试行，便于广大高等学校教师按照新的专业规范组织实施教学，高等教育出版社在大力支持专业规范研究与起草工作的同时，还邀请规范起草小组的有关专家成立“高等学校计算机科学与技术系列教材编审委员会”，组织规划了结合计算机专业规范、面向全国高等学校计算机专业本科生的“高等学校计算机科学与技术系列教材”。令人高兴的是，一批有创新、改革精神，且有丰富教学经验的高等学校教师投身到新体系计算机专业教材的编写中来，他们用自己创造性的思维、辛勤的汗水诠释专业规范的思想，把新的课程体系和教学内容生动地传达给师生，并进行着有意义的教学实践。

“高等学校计算机科学与技术系列教材”以专业规范和CC2001－CC2005有关教程为依据，以强化基础、突出实践、注重创新为原则，体现了学科课程体系和教学内容改革的新成果。

此外，这一系列教材还配有丰富的教学辅助资源，并与现代教育技术手段相结合，充分发挥网络平台的作用，使教材更有利于广大教师和学生使用。目前，这一系列教材有不少选题已列入普通高等教育“十一五”国家级规划教材，希望这些教材的出版能够对新形势下我国高等学校计算机专业课程改革与建设起到积极的推动作用，使我国高等学校的计算机专业教学质量再上一个台阶。

中国科学院院士

2006—2010 年教育部高等学校计算机科学与技术教学指导委员会主任

二〇〇七年十一月

前　言

计算机科学已经深入应用到各个领域，不仅有效地解决了各种工程和科学计算中的数值计算问题，而且有效地解决了许多文本处理、信息检索、数据库管理、图像识别、人工智能等非数值的数据处理问题。

数据结构是计算机学科的一个重要分支研究领域，它是算法分析与设计、操作系统、软件工程、数据库概论、编译技术、计算机图形学、人机交互等专业基础课和专业课程的先行课程。其他计算机科学领域及有关的应用软件都会涉及各种数据结构。例如，语言编译要使用栈、散列表及语法树；操作系统中要用队列、存储管理表及目录树；数据库系统运用线性表、多链表及索引树等进行数据管理；而在人工智能领域，按求解问题性质的差异将涉及各种不同的数据结构，例如广义表、集合、搜索树及各种有向图等。

美国 IEEE 和 ACM 的教学计划 CC2005 把“算法与数据结构”列入计算机以及信息技术相关学科专业的本科必修基础课程。国内外许多科技人员对学习、研究数据结构都非常重视。2006 年，教育部高等学校计算机科学与技术教学指导委员会（以下简称教育部计算机教指委）发布的《高等学校计算机科学与技术专业发展战略研究报告暨专业规范（试行）》（以下简称“专业规范”）明确强调了实践教学和学生动手能力培养的重要性。

在 IEEE/ACM CC2005 和教育部计算机教指委的专业规范精神指导下，结合作者十多年来任教“数据结构与算法”课程所积累的丰富教学经验和教学素材，写成本教材和配套的《数据结构与算法实验教程》这两本国家“十一五”规划立项教材。

“数据结构与算法”是一门理论和实际紧密结合的重要的计算机类基础课程。本教材融入新的数据结构和算法思想，注意基本概念的引入和阐述，对主要数据结构及其相关算法分析技术进行了比较深入的讨论。适应分层次、多种规格培养人才的需求，加强与应用背景的结合，加强实践能力训练，指导学生学会利用数据结构和算法的理论和技术解决具体的应用问题。通过学习，学生能更有效地组织数据、设计高效的算法、完成高质量的程序，以满足错综复杂的实际应用需要，为将来从事计算机相关的学习、研究和开发工作打下扎实的基础。

为了给学生布置比较综合的索引和检索上机实习，本教材把多维数组、广义表、AVL 树等内容编排到第 12 章“高级数据结构”中。这样，可以留出一个月左右的时间，让学生合作编写学期大综合实习题。此外，教师还可以利用这段时间，讲授红黑树、字符树、后缀树、伸展树等非常实用的高级数据结构技术。本教材建议学时为 48 学时，那些较为复杂的内容可以根据专业选讲或安排能力强的学生自学。内容提要及学时分配如下：

1. 概论(2 学时)

数据结构定义(逻辑结构、存储结构、运算),抽象数据类型,算法及其算法度量和评价(大 O 表示法及其运算规则)。

2. 线性表、栈与队列(2 ~4 学时)

线性表(向量、链表)、栈与队列(顺序、链接)、栈的应用。

选讲递归到非递归的转换机制和方法。

3. 字符串(4 学时)

字符串抽象数据类型、存储表示和类定义、字符串的运算、字符串的模式匹配。

4. 二叉树 (6 ~8 学时)

二叉树的概念及性质、二叉树的抽象数据类型、二叉树的周游、二叉树的存储实现、二叉检索树、堆与优先队列、Huffman 树。

选讲非递归深度优先周游二叉树和穿线二叉树。

5. 树(4 学时)

树的概念、森林与二叉树的等价转换、树的抽象数据类型、树的周游、树的链式存储、树的顺序存储。

6. 图 (6 学时)

图的基本概念、图的抽象数据类型、图的存储结构、图的周游(深度优先、搜索、广度优先、拓扑排序)、最短路径问题、最小生成树(Prim 算法、Kruskal 算法)。

7. 内排序(6 学时)

排序问题的基本概念、插入排序(直接插入、Shell 排序)、选择排序(直接选择、堆排序)、交换排序(冒泡排序、快速排序)、分配排序(桶式排序、基数排序)。

选讲索引排序、排序算法的理论及实验时间代价、排序问题的下限。

8. 文件管理和外排序(2 学时)

介绍外排序的特点、二路外排序、置换选择排序。

9. 检索(4 学时)

检索的基本概念、基于线性表的检索、基于集合的检索和散列方法。

10. 索引技术(2 ~4 学时)

倒排索引、$B+$ 树等动态索引组织。

选讲红黑树索引、位图索引。

11. 高级数据结构(0 ~4 学时)

选讲广义表、字符 Trie、后缀树、AVL 树。

与本书相关的《数据结构与算法》教材(许卓群、杨冬青、唐世渭、张铭,2004 版)和配套的《数据结构与算法——学习指导与习题解析》自 2004 年出版发行以来,受到了众多读者的欢迎。本教材继承了这两本相关教材的基本知识体系,扩展讨论了红黑树、后缀树等实用数据结构。在教材的组织方式上,注重从问题求解的角度出发,讨论相关的基本理论、数据和算法抽象、数据结

构与算法设计以及在 C++ 程序设计语言中的实现。

本书第 8、10、11、12 章由张铭教授编写，第 5、6、7、9 章由王腾蛟副教授编写，第 1、2、3、4 章由赵海燕副教授编写。其中，第 8、10、12 章是在 2004 版《数据结构与算法》教材中张铭所编写部分的基础上修订而成，5～12 章是在张铭和助教编写的算法代码基础上调整而成，其他章节都是作者新编写的内容。本教材采用了 2004 版《数据结构与算法》的部分习题，作者在此对许卓群教授、杨冬青教授、唐世渭教授表示衷心致谢。张铭对全书在文字风格、内容、例题选择、算法实现等方面进行了细致的修改和统稿。赵海燕和王腾蛟也对部分书稿进行了交叉审改。作者在 2007 年 11 月赠阅了 50 多本预印本，以广泛征求同行专家和读者的反馈意见。

北京大学李晓明教授非常支持本教材的写作和出版，给予作者很多帮助。北京工业大学蒋宗礼教授、北京理工大学陈朔鹰教授、北京大学谢柏青教授等专家仔细审查了本书的预印版，并提出了很好的修改意见。高等教育出版社为本书的出版提供了有力的支持和帮助。本书的很多内容曾经由我们三位作者在 2004—2007 年度给北京大学信息学院讲授的“数据结构与算法”课程中使用，感谢听课的 2003—2006 级信息学院学生以及参与课程辅导的历届研究生助教。2006 级冯熙铉、陈醒、钱昊、李昂、贾由、姚金宇、王子琪、陈学轩、丁羽、袁文清、黄柏彤、金鑫、李义春、周金果、吴轲、吴迪、张晓东、付炳楠、韩薇、田峰、黄鹏、王昊等同学，仔细阅读了本书的预印版，验证了书稿的算法并调试了代码。他们的建议和意见都弥足珍贵，作者在此表示真挚的感谢。

尽管本书经过了作者的反复讨论和推敲，并经历了 4 年的教学检验，但限于水平，难免有不妥之处，希望读者不吝赐教。本教材的配套教学网站为 http://db.pku.edu.cn/mzhang/ds/（课程主网站，通过教育网访问）或 http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/（北京大学精品课程汇报网站，本课程被评为 2008 年“北京市精品课程”）。教材中只给出了最主要的算法和代码，完整的实现方法可参考教学网站上提供的与本书配套的代码包。读者也可访问中国高校计算机课程网下载有关资源，网址为 http://computer.cncourse.com/computer/coursecenter/indexdefault。

张　铭　王腾蛟　赵海燕

2008 年 1 月于北京大学

目　录

第 1 章　概论 …… 1
1.1　问题求解 …… 1
1.1.1　问题描述:股市的传言 …… 2
1.1.2　问题分析和抽象 …… 2
1.1.3　数据结构和算法设计 …… 3
1.2　数据结构 …… 6
1.2.1　数据的逻辑结构 …… 6
1.2.2　数据的存储结构 …… 7
1.2.3　抽象数据类型 …… 9
1.3　算法 …… 11
1.3.1　算法的概念 …… 11
1.3.2　算法设计 …… 12
1.4　算法分析 …… 14
1.4.1　渐进分析方法 …… 15
1.4.2　最佳、最差和平均情况 …… 19
1.4.3　时间和空间的折衷 …… 21
1.4.4　求解问题时数据结构的选择和评价 …… 22
本章小结 …… 22
习题 …… 23
上机题 …… 24
第 2 章　线性表 …… 26
2.1　线性表的概念 …… 26
2.1.1　线性表的抽象数据类型 …… 26
2.1.2　线性表的存储结构 …… 28
2.1.3　线性表运算分类 …… 29
2.2　顺序表 …… 29
2.2.1　顺序表的类定义 …… 29
2.2.2　顺序表的运算实现 …… 31
2.3　链表 …… 34
2.3.1　单链表 …… 34
2.3.2　双链表 …… 40
2.3.3　循环链表 …… 42
2.4　线性表实现方法的比较 …… 44
本章小结 …… 44
习题 …… 45
上机题 …… 45
第 3 章　栈与队列 …… 46
3.1　栈 …… 46
3.1.1　栈的抽象数据类型 …… 47
3.1.2　顺序栈 …… 47
3.1.3　链式栈 …… 50
3.1.4　表达式求值 …… 52
3.1.5　栈与递归 …… 58
3.2　队列 …… 68
3.2.1　队列的抽象数据类型 …… 68
3.2.2　顺序队列 …… 69
3.2.3　链式队列 …… 72
3.3　栈与队列的深入讨论 …… 74
3.3.1　顺序栈与链式栈的比较 …… 74
3.3.2　顺序队列与链式队列的比较 …… 75
3.3.3　限制存取点的表 …… 75
本章小结 …… 76
习题 …… 77
上机题 …… 77
第 4 章　字符串 …… 79
4.1　字符串的基本概念 …… 79

4.1.1 字符编码 …… 79
4.1.2 字符的编码顺序 …… 80
4.1.3 字符串抽象数据类型 …… 81
4.2 字符串的存储结构和实现 …… 82
4.2.1 字符串的顺序存储 …… 82
4.2.2 字符串类 class String 的存储结构 …… 84
4.2.3 字符串运算的实现 …… 87
4.3 字符串的模式匹配 …… 89
4.3.1 朴素的模式匹配算法 …… 90
4.3.2 字符串的特征向量 …… 93
4.3.3 KMP 模式匹配算法 …… 96
本章小结 …… 97
习题 …… 98
上机题 …… 99
第5章 二叉树 …… 100
5.1 二叉树的概念 …… 100
5.1.1 二叉树的定义和基本术语 …… 100
5.1.2 满二叉树、完全二叉树、扩充二叉树 …… 101
5.1.3 二叉树的主要性质 …… 102
5.2 二叉树的周游 …… 104
5.2.1 二叉树的抽象数据类型 …… 104
5.2.2 深度优先周游二叉树 …… 106
5.2.3 广度优先周游二叉树 …… 111
5.3 二叉树的存储结构 …… 112
5.3.1 二叉树的链式存储结构 …… 112
5.3.2 完全二叉树的顺序存储结构 …… 115
5.4 二叉搜索树 …… 116
5.5 堆与优先队列 …… 120
5.5.1 堆的定义及其实现 …… 120
5.5.2 优先队列 …… 126
5.6 Huffman 树及其应用 …… 127
5.6.1 Huffman 树 …… 127
5.6.2 Huffman 编码 …… 129
本章小结 …… 131
习题 …… 131
上机题 …… 133
第6章 树 …… 135
6.1 树的定义和基本术语 …… 135
6.1.1 树和森林 …… 135
6.1.2 森林与二叉树的等价转换 …… 137
6.1.3 树的抽象数据类型 …… 139
6.1.4 树的周游 …… 140
6.2 树的链式存储结构 …… 143
6.2.1 “子结点表”表示方法 …… 143
6.2.2 静态“左子/右兄”表示法 …… 144
6.2.3 动态表示法 …… 145
6.2.4 动态“左子/右兄”二叉链表表示法 …… 145
6.2.5 父指针表示法和在并查集中的应用 …… 148
6.3 树的顺序存储结构 …… 153
6.3.1 带右链的先根次序表示 …… 153
6.3.2 带双标记的先根次序表示 …… 154
6.3.3 带度数的后根次序表示 …… 155
6.3.4 带双标记的层次次序表示 …… 156
6.4 *K* 叉树 …… 157
本章小结 …… 158
习题 …… 158
上机题 …… 159
第7章 图 …… 161
7.1 图的定义和基本术语 …… 161
7.2 图的抽象数据类型 …… 165
7.3 图的存储结构 …… 166

7.3.1 相邻矩阵 …… 166
7.3.2 邻接表 …… 169
7.3.3 十字链表 …… 173
7.4 图的周游 …… 174
7.4.1 深度优先周游 …… 175
7.4.2 广度优先周游 …… 176
7.4.3 拓扑排序 …… 177
7.5 最短路径 …… 179
7.5.1 单源最短路径 …… 180
7.5.2 每对顶点之间的最短路径 …… 182
7.6 最小生成树 …… 184
7.6.1 Prim 算法 …… 185
7.6.2 Kruskal 算法 …… 187
本章小结 …… 189
习题 …… 190
上机题 …… 192
第 8 章 内排序 …… 194
8.1 排序问题的基本概念 …… 194
8.2 插入排序 …… 195
8.2.1 直接插入排序 …… 196
8.2.2 Shell 排序 …… 198
8.3 选择排序 …… 200
8.3.1 直接选择排序 …… 200
8.3.2 堆排序 …… 202
8.4 交换排序 …… 204
8.4.1 冒泡排序 …… 204
8.4.2 快速排序 …… 206
8.5 归并排序 …… 212
8.6 分配排序和索引排序 …… 215
8.6.1 桶式排序 …… 216
8.6.2 基数排序 …… 217
8.6.3 索引排序 …… 226
8.7 排序算法的时间代价 …… 228
8.7.1 简单排序算法的时间代价 …… 228
8.7.2 排序算法的理论和实验时间 …… 229
8.7.3 排序问题的下限 …… 232
本章小结 …… 234
习题 …… 236
上机题 …… 240
第 9 章 文件管理和外排序 …… 241
9.1 主存储器和外存储器 …… 241
9.2 文件的组织和管理 …… 242
9.2.1 文件组织 …… 243
9.2.2 C++ 的流文件 …… 244
9.3 外排序 …… 245
9.3.1 置换选择排序 …… 245
9.3.2 二路外排序 …… 248
9.3.3 多路归并——选择树 …… 250
本章小结 …… 257
习题 …… 258
上机题 …… 259
第 10 章 检索 …… 260
10.1 基于线性表的检索 …… 261
10.1.1 顺序检索 …… 261
10.1.2 二分检索 …… 263
10.1.3 分块检索 …… 266
10.2 集合的检索 …… 268
10.2.1 集合的数学特性 …… 268
10.2.2 计算机中的集合 …… 269
10.3 散列方法 …… 274
10.3.1 散列函数 …… 276
10.3.2 开散列方法(拉链法) …… 280
10.3.3 闭散列方法(开地址法) …… 282
10.3.4 闭散列表的算法 …… 285
10.3.5 散列方法的效率分析 …… 290
10.3.6 散列方法的应用 …… 292
本章小结 …… 293

习题 …… 294
上机题 …… 297
第 11 章 索引技术 …… 300
11.1 线性索引 …… 301
11.2 静态索引——多分树 …… 302
11.3 倒排索引 …… 303
11.3.1 基于属性的倒排 …… 303
11.3.2 对正文文件的倒排 …… 305
11.4 动态索引 …… 308
11.4.1 *B* 树 …… 308
11.4.2 *B*+树 …… 314
11.4.3 *B* 树的性能分析 …… 317
11.4.4 动态索引和静态索引性能的比较 …… 319
11.5 位索引技术 …… 320
11.5.1 位图索引 …… 320
11.5.2 签名文件 …… 321
11.6 红黑树 …… 322
11.6.1 红黑树的定义 …… 322
11.6.2 红黑树的相关性质 …… 323
11.6.3 插入结点算法 …… 323
11.6.4 删除结点算法 …… 325
本章小结 …… 328
习题 …… 330
上机题 …… 334
第 12 章 高级数据结构 …… 335
12.1 多维数组 …… 335
12.1.1 多维数组的存储 …… 335
12.1.2 特殊矩阵 …… 336
12.1.3 稀疏矩阵 …… 337
12.2 广义表和存储管理 …… 339
12.2.1 广义表的定义和存储结构 …… 339
12.2.2 可利用空间表 …… 342
12.2.3 存储的动态分配和回收 …… 344
12.2.4 失败处理策略和无用单元回收 …… 347
12.3 Trie 结构和 Patricia 树 …… 348
12.4 改进的二叉搜索树 …… 352
12.4.1 最佳二叉搜索树 …… 353
12.4.2 平衡的二叉搜索树 …… 361
12.4.3 伸展树 …… 369
本章小结 …… 374
习题 …… 375
上机题 …… 376
参考文献 …… 382

第1章 概 论

在信息技术日益渗透到各行各业的今天，计算机已成为解决各类实际问题的主要工具。利用计算机来解决各种实际问题不可避免地涉及众多的知识和学科，而计算机学科便是其中的核心学科。

作为计算机学科一个重要的分支，数据结构与算法的研究涉及构筑计算机求解问题过程的两大基石：刻画实际问题中信息及其关系的数据结构和描述问题解决方案的逻辑抽象的算法。数据结构与算法呈相互依赖的关系，只有恰当地确立了问题的结构，才能选择和设计合适的解决方法。可以说，数据结构与算法的相关知识是有效使用计算机的基本前提。所有计算机系统软件和应用软件都要用到各种类型的数据结构以及一些典型的算法设计和分析技巧。在诸如图形、数据库、数值分析等专门领域中，解决问题的能力依赖于其中的算法与数据结构的最新进展。“数据结构与算法”课程是“算法分析与设计”、“操作系统”、“软件工程”、“数据库概论”、“编译技术”、“计算机图形学”、“人机交互”等专业基础课和专业课程的先修课程。

本章由一个实例开始阐述从问题求解到程序设计的过程，并概述其中涉及的基本概念。在阐述数据结构和算法的同时，揭示数据结构与算法之间的密切关系。

1.1 问题求解

学习过计算引论的人大多都听说过 Niklaus Wirth 的著名公式“程序 = 数据结构 + 算法”。该公式在阐明算法、数据结构与程序设计的重要性的同时，也从一个方面阐述了数据结构与算法的密切关系。

何谓程序，算法以及数据结构又是什么？作为计算机科学的核心课程，“数据结构与算法”正是为了解答这些问题而设立的。简单来讲，程序是计算机“指令”的某种组合，用来控制计算机的工作流程，完成一定的逻辑功能，从而实现某种任务。算法是程序的逻辑抽象，是解决某类客观问题的过程。数据结构是现实世界的数据及其间关系的反映，可以从逻辑结构和物理结构两个层面去刻画：客观事物自身所具有的结构特点称为逻辑结构，例如家族谱系即为一个天然的树形逻辑结构；而逻辑结构在计算机中的具体实现则称为物理结构，例如用于表示树形逻辑结构的指针或数组。

数据结构与算法呈相互依赖的紧密关系。如果把数据结构喻为建筑工程中的建筑设计图，那么算法就是工程中的施工流程图。因此，任何讲解数据结构的书都不可能撇开算法而独立地

介绍数据结构，反之亦然。

人们利用计算机的目的是解决实际的应用问题。在明确所要解决的问题的基础上，经过对问题的深入分析和抽象，为其在计算机中建立一个确切的模型，然后确定恰当的数据结构表示该模型，再在此基础上设计合适的算法，最后根据设计的数据结构和算法进行相应的程序设计来模拟和解决实际问题。

计算机求解问题的核心是算法设计，而算法设计又高度依赖于数据结构，数据结构的选择则取决于问题本身的需求。下面通过一个简单的实例来说明问题的求解过程以及如何进行相应的算法设计和数据结构设计。

1.1.1 问题描述：股市的传言

股票市场对各种传言高度敏感，股票经纪人会据此进行股票的抛售和购进。一般情况下，经纪人只信任那些“来源可靠”的信息。例如，来自同一部门某位同事的消息。两个人之间的消息传递是有方向的，且消息传递所需要的时间因人而异。表1.1所示为5个经纪人之间消息传递的示例。其中，如果消息传递到B_1，则B_1可把消息传递到其他3个人B_4、B_2、B_5，所用时间单位分别为4、8、3，可用二元组（对方代号，所需时间单位）来表示消息接收方的信息。

表1.1 经纪人之间消息传递的关系

经纪人代号	消息可传递的人数	消息接收方信息
B_1	3	(4,4) (2,8) (5,3)
B_2	1	(5,8)
B_3	4	(1,6) (4,10) (2,7) (5,2)
B_4	0	
B_5	2	(2,5) (1,5)

为此，某交易人员打算设计一个可以使假情报在部门的各经纪人之间快速传播的系统，以便从中谋利。时间就是金钱，为了赢得时间，要求这些假消息传播的速度越快越好。作为一个假消息的制造者来说，需要在各部门中物色一个合适的人选s，然后把消息传给s，由s开始传播直到该部门所有的人都知道此消息为止。假设一个人收到消息后便立即将消息发送出去。问题的关键是找出能在最短时间内让所有人都收到消息的消息发布源s。

1.1.2 问题分析和抽象

为了完成上述制造和传播假消息的任务，首先需要分析经纪人之间的关系：明确一个部门中哪些经纪人之间可以相互传递消息；哪些经纪人之间没有关系，无法把消息送达；或者哪些经纪

人只接收消息而不传播消息，应该避免选择他作为发布源。例如，表 1.1 中若把消息首先送达 B_4，则无法传播，因为 B_4 可传递的人数为 0。

如果每个经纪人用一个顶点表示，两个可传递消息的经纪人之间用一条带权有向边来表示消息传递的方向和所需时间，则各个经纪人之间的关系可用一个称做有向带权图的数据结构来表示，如图 1.1 所示。

图 1.1 中的顶点 B_i 表示第 i 个经纪人。从顶点 B_i 到 B_j 之间存在的有向边 $<B_i,B_j>$ 表示代号为 i 的经纪人可传递消息给代号为 j 的经纪人，边上标注的数字是传递消息所需时间。从 B_i 到 B_j 所需的时间与 B_j 到 B_i 的时间不尽相同。

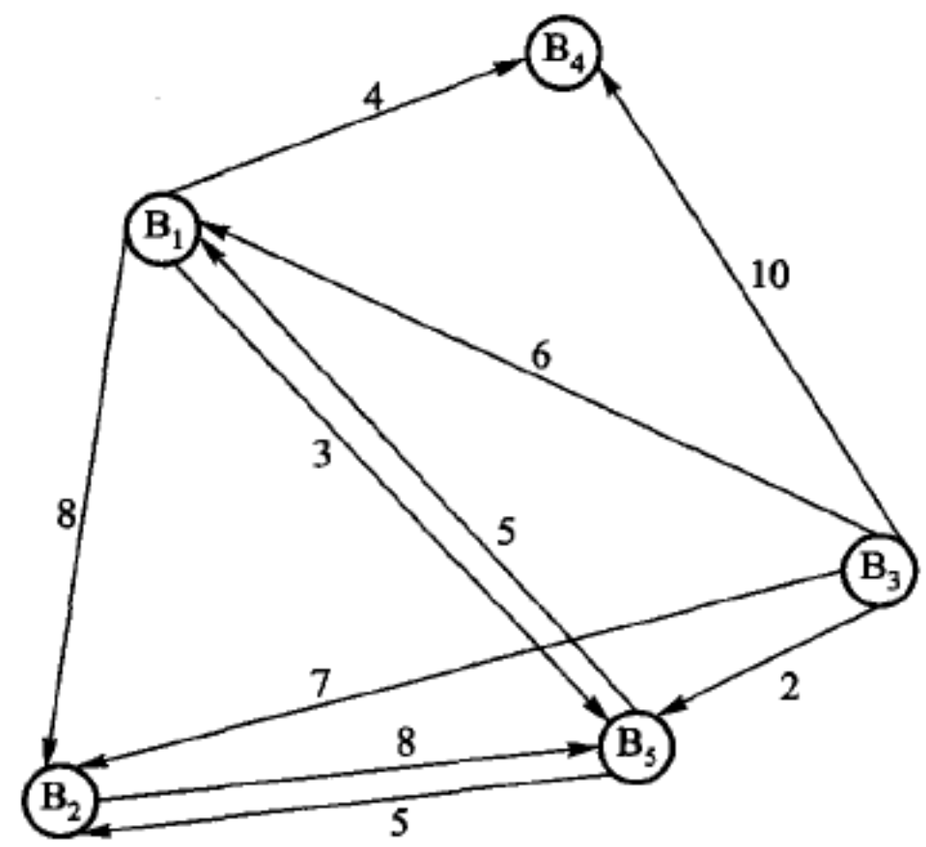


图 1.1 经纪人间的消息传递图

那么，上述假消息传播的问题可归结为在一个有向图中寻找这样的一个顶点：从它出发能够以最短的时间把消息传递到所有其他顶点。显然，在图 1.1 这样的带权有向图中，从一个顶点到另一顶点的传递有可能经过别的顶点，例如，B_5 到 B_4 需经过 B_1 作为中介。顶点序列 $B_5B_1B_4$ 称做从顶点 B_5 到 B_4 的一条路径。从一个顶点到另一个顶点可能存在多条路径，每条路径的边数不尽相同。有向带权图中两个顶点间的路径长度定义为路径上各边的权值之和。令各顶点间长度最短的那条路径为这两个顶点间的最短路径。因此，上述问题的一个简单的解决办法是先求出图中各个顶点之间的长度最短的路径（简称最短路径）；然后，再对每个顶点 v 求出它到其他各顶点的最短路径中的最大值，即从顶点 v 出发传递消息到其他各点最终所需的时间；最后，选取这些最大值中取值最小的那个顶点，即为问题所要寻找的开始顶点。其中，最关键的一步是如何求取图中各个顶点之间的最短路径。

这样，借助于图这种数据结构，股票市场假消息传播的实际问题就归结成有向图的最短路径问题这样更一般的抽象问题。

1.1.3 数据结构和算法设计

问题明确之后，接下来的任务是设计适当的算法和数据结构来描述解决方案。根据问题分析阶段的讨论可知，若从顶点 v_i 到 v_j 之间存在有向边 $<v_i,v_j>$，则意味着消息可从顶点 i 传递到顶点 j；若一个顶点没有到其他顶点的边，则表示消息从此顶点传播不出去，图 1.1 中的 B_4 即为这样的顶点；反之，若一个顶点没有来自其他顶点的边，则意味着别处的消息传播不过来，例如图 1.1 中的 B_3，只能作为消息源。

如果一个有向图不连通，即某些顶点对之间没有路径可通，则表示消息无法从一个顶点出发传播到所有其他顶点，也就不能完成前述的任务。因此，为了实施消息的传播，有向图中应该至

少存在一个顶点，从该顶点出发可到达所有其他顶点。在这样的图中，要寻找某一个顶点，且从该顶点出发到其他各顶点的最短路径的最大者在同类中最小，可采用的步骤如下：

(1) 从图中选择一个顶点，求其到其他各点的最短路径，并找出其中的最大值，若有不可达者，则用一个很大的数∞表示相应的最短路径。

(2) 重复第(1)步，直到所有的顶点都如此处理过。

(3) 求出这些最大值中的最小者。

求解结果有两种可能：其一，能够找到具有最小值的顶点，任务圆满完成；其二，所有的最大值都为∞，则说明此图中不存在这样一个顶点从它出发可以把消息顺利传播到其他各个顶点，该问题无解。这样，问题的关键就是如何计算图中各个顶点间的最短路径。本书第7章将详细讨论有关最短路径的 Dijkstra 算法和 Floyd 算法。

一般来说，解决一个问题可能存在多种方案，例如可以采用不同的存储方案，采用不同的算法等。此时的关键是找到一个能充分利用计算机资源的解决方案，这就涉及数据结构的选择问题。

计算机存储图结构通常采用相邻矩阵(adjacency matrix)和邻接表(adjacency list)两种方式，每种方案各有其适用的场景。鉴于本题需要计算任意两个顶点之间的最短路径，因此选择相邻矩阵作为图的存储方式，选择 Floyd 算法来计算各个顶点间的最短路径。

用相邻矩阵表示图需要保存一个 $n \times n$ 的矩阵，其中位于第 i 行第 j 列的元素表示从顶点 i 到顶点 j 的边的权值。对应图 1.1 的相邻矩阵如图 1.2 所示。

	B_1	B_2	B_3	B_4	B_5
B_1	0	8	∞	4	3
B_2	∞	0	∞	∞	8
B_3	6	7	0	10	2
B_4	∞	∞	∞	0	∞
B_5	5	5	∞	∞	0

图 1.2 经纪人间消息传递的相邻矩阵

假设共有 MAX 个经纪人，两人之间若不存在路径用 INFINITY 来表示。算法 1.1 是采用 Floyd 算法来求解该问题的一种 C++语言的实现。

【算法 1.1】 股市传言。

```
#include <iostream>
#include <string>

using namespace std;

#define MAX 102
#define INFINITY 65536

int BrokerToPassRomour (Graph& G ) {
    int ** D;                                   // 传入 Floyd 算法的参数，存放求出的各顶点
                                                // 间的最短路径
    int max[G. VerticesNum( )];                 // 记录每个顶点到其他顶点的最短路径的最
                                                // 大值
    int min =0;                                 // 记录 max 数组中的最小值
```

```
    int pos = 0;                                    // 记录 max 数组中最小值的位置
    for ( int i = 0; i < G. VerticesNum( ) ; i ++ )  // 初始化 max 数组
        max[ i ] = -1 ;
    Floyd( G, D) ;                                  // 调用 Floyd 算法求每个顶点之间的最短路径
    for ( i = 0; i < G. VerticesNum( ) ;i ++ ) {     // 找出每行的最大值,存放到 max 数组
        for ( int j = 0; j < G. VerticesNum( ) ; j ++ ) {
            if ( D[ i ][ j ] > max[ i ] )
                max[ i ] = D[ i ][ j ] ;
        }
    }
    min = max[ 0 ] ;
    for ( i = 1 ; i < G. VerticesNum( ) ; i ++ )     // 查找 max 数组的最小值及其位置
        if ( max[ i ] < min ) {
            min = max[ i ] ;
            pos = i;
        }
    if ( min == INFINITY) {
        cout << "此图不连通" << endl;
        return -1;
    }
    return pos;                                     // 返回最合适的开始顶点
}
```

【算法 1.1 结束】

其中,调用的函数 Floyd(G,D)可参考本书第 7 章的 Floyd 算法,第 1 个参数 G 表示一个图,其成员函数 VerticesNum( )返回图中的顶点数目,第 2 个参数 D 是一个相邻矩阵。针对一个给定的图,用 Floyd 算法计算出图中各个顶点间的最短路径并存储在参数 D 中返回。本例采用 Floyd 算法后求得的矩阵如图 1.3 所示。

其中,每一行代表从 $B_i(i=1\cdots5)$ 出发到其他各点的最短路径长度。例如,从顶点 B_1、B_2、B_4、B_5 出发到其他各顶点的最短路径中最大值均为∞,而从 B_3 出发的最长的最短路径为 10,这些最大值存放在 max 数组中。其中,最短者为 10,因此 B_3 就是所寻找的起始顶点。

	B_1	B_2	B_3	B_4	B_5
B_1	0	8	∞	4	3
B_2	13	0	∞	17	8
B_3	6	7	0	10	2
B_4	∞	∞	∞	0	∞
B_5	5	5	∞	9	0

图 1.3 经纪人间消息传递的相邻矩阵

从以上股市传言问题的求解过程,可以看出利用计算机求解问题:首先,需要仔细分析问题的要求,明确所要完成的任务需求;然后,对问题进行分析和抽象,建立问题的抽象模型;在此基础上,确定该模型所采用的数据结构,设计算法来求解该问题;最后,选定某种程序设计语言来编码实现。整个过程即是一个所谓的问题抽象、数据抽

象和算法抽象的过程。

1.2 数据结构

数据结构(data structure)描述的是按照一定逻辑关系组织起来的待处理数据元素的表示及相关操作,涉及数据的逻辑结构、数据的存储结构和数据的运算。

1.2.1 数据的逻辑结构

数据的逻辑结构(logical structure)是从具体问题抽象出来的数学模型,反映了事物的组成结构及事物之间的逻辑关系。例如,1.1 节中经纪人之间的消息传递路径可以用一个有向图来表示。

从集合论的观点来看,数据的逻辑结构可以用一个二元组 $B=(K,R)$ 来表示。其中,K 是数据结点(node)组成的有穷集合,每一个结点代表一个数据或一组有明确结构的数据;关系集 R 是定义在集合 K 上的一组二元关系(binary relation),其中每个关系 $r(r\in R)$ 都是 K 上的二元关系,用来描述数据结点之间的逻辑关系。K 上的二元组是 K 中元素的有序对,记为 $<k,k'>(k,k'\in K)$。K 上的一个关系即为 K 上的一些二元组所组成的集合,K 上不同的二元组集合构成不同的关系。

若 $r\in R$,有 $k,k'\in K$, $<k,k'>\in r$,则称 k 为 k' 在关系 r 上的前驱,k' 为 k 在关系 r 上的后继。没有前驱的结点称为开始结点,没有后继的结点称为终止结点。

例如,一个教学计划包含许多课程,在这些课程之间,有些必须按规定的先后次序开设,有些则没有次序的要求。此时,每门课程都可作为一个数据结点,而主修某个专业所需要学习的全部课程组成一个课程结点集 K。课程之间的各类关系构成了关系集合 R,如先修关系 r 等。每一个关系可以用课程的关系元组(有向图中两个顶点间的有向边)给出。例如,计算机专业的课程中具有先修关系的 <程序设计,数据结构>。

二元组 B 中的结点具有何种结构?结点之间的关系具有哪些表现形态和限制?下面将就此予以简单的概述。

1. 结点的类型

数据结构重点研究的是数据间的结构关系,因此把组成结构的那些元素都视为结点。结点是数据结构中数据的基本单位。结点的数据类型既可以是基本数据类型,也可以是复合数据类型。

基本数据类型是指数据结点所包含的元素被作为整体看待和处理的类型。类似于常见的程序设计语言中提供的基本数据类型,例如整数类型(integer)、实数类型(real)、布尔类型(bool)、字符类型(char)和指针类型(pointer)等。

复合类型是由基本数据类型组合而成的数据结构类型，不同的组合方式形成不同的类型。像程序语言中的数组类型、结构（记录）类型、对象类型等都属于复合数据类型。复合数据类型本身，又可以参与定义结构更为复杂的结点类型。

总而言之，结点的类型不限于基本数据类型，可以根据应用的需要来灵活定义。

2. 结构的分类

讨论逻辑结构(K,R)的结构分类，一般以关系集 R 的分类为主。在本书讨论的数据结构中，R 中一般只包含一个关系。假定关系集 R 仅含一个关系 r，即，$R=\{r\}$，此时记(K,R)为(K,r)。根据 r 所刻画的数据结构的性质来对逻辑结构进行分类。

线性结构（linear structure）中的关系 r 称为线性关系，也称前驱关系。结点集合 K 中的每个结点在关系 r 上最多只有一个前驱结点和一个后继结点。具体而言，结点集合 K 在关系 r 上存在一个唯一的开始结点，它没有前驱，但有唯一的后继；也存在一个唯一的终止结点，具有一个唯一的前驱而没有后继；其他结点均为内部结点，在关系 r 上既有一个唯一的前驱，也有一个唯一的后继。线性结构在程序设计中应用最多，例如数组、链表、栈和队列等。

树形结构（tree structure）简称树结构或层次结构，其关系 r 一般称为层次关系。结点集合 K 中有且仅有一个结点 $k_0 \in K$，在关系 r 中没有前驱，该结点称为树根（root）。除结点 k_0 外，K 中所有结点都有且仅有一个前驱，但后继的数目不加限制。同时，K 中除结点 k_0 外的任何结点 $k \in K$，都存在一个结点序列 $k_0,k_1,\cdots,k_s$，使得 k_0 是树根，且 $k_s=k$，有序对 $<k_{i-1},k_i> \in r(1 \leqslant i \leqslant s)$，这样的结点序列称为从根到结点 k 的一条路径（path）。树形结构存在很多形态，如二叉树结构、有序树结构等，它们都有着各自独特的应用。UNIX 和 DOS 中的文件系统就是一个典型的树结构。

图结构（graph structure）有时也称为网络结构。图结构的关系 r 对集合 K 中结点的前驱和后继数目不加任何约束。例如，交通网络就是一个非常复杂的典型图结构。

从数学上看，树形结构和图结构的基本区别就是“每个结点是否仅仅从属于一个直接前驱”。而线性结构和树形结构的基本区别是“每个结点是否仅有一个直接后继”。

以上这种结构分类揭示出枯燥数据之间的相互关系，给出了关系本身的一般性质。这对于理解数据结构以及正确设计算法都很重要。后续各章节将具体讨论各种数据结构及其相关的应用。

1.2.2 数据的存储结构

数据的存储结构所要解决的问题是各种逻辑结构在计算机中的物理存储表示。计算机的主存储器具有“空间相邻”和“随机访问”的特点：基本的存储单元是字节，用非负整数对存储地址编码，提供对存储空间上相邻的单元集合的随机访问；计算机指令具有按地址随机访问存储空间内任意单元的能力，访问不同地址所需的访问时间基本相同。

对于逻辑结构(K,r)而言，其数据的存储结构就是建立一种由逻辑结构到物理存储空间的映射。一方面，需要建立一个从结点集合 K 到存储器 M 的映射：$K \rightarrow M$，其中每一个结点 $k \in K$ 都

对应一个唯一的连续存储区域 $c \in M$。另一方面，还要把每一个关系元组 $<k_i,k_j> \in r$（其中 $k_i,k_j \in K$ 是结点）映射为相应的存储单元的地址间的关系（顺序关系或指针的地址指向关系）。

同一种逻辑结构可以采用不同的表示方式，即采用不同的映射关系来建立数据的逻辑结构到存储结构的转换。例如，线性结构既可以采用顺序的方式来存储，形成顺序表；也可以采用链接的方式，得到链表。

常用的基本存储映射方法有顺序方法、链接方法、索引方法和散列方法。

1. 顺序方法

顺序存储方法把一组结点存放在一片地址相邻的存储单元中，结点间的逻辑关系用存储单元间的自然顺序关系来表达。顺序存储法为使用整数编码访问数据结点提供了便利。程序设计语言内提供的数组是顺序方法的一个具体实例。

顺序存储结构通常也被称为紧凑存储结构。其紧凑性是指其存储空间除了存储数据本身之外，没有存储其他附加的信息。紧凑性可以用“存储密度”来度量：所存储的“数据”占用的存储空间和该结构（包括附加信息）占用的整个存储空间大小之比。显然，存储密度太小的存储结构的空间效率比较低。

除线性结构外，对于部分非线性数据结构也可以采用顺序存储方法。例如，树形结构也可采用顺序方式来存储，前提是同时存储一些附加信息来表示结点之间的逻辑关系。

2. 链接方法

链接方法是在结点的存储结构中附加指针域来存储结点间的逻辑关系。链接方法中数据结点由两部分组成：数据域存放结点本身的数据，指针域存放指向其后继结点的指针。

根据应用的需要，结点的指针域也可存储多个指针来表达一个结点同时链接多个结点的情况。例如，非线性结构中一个结点可能会有多个后继结点的情况。

链接方法适用于那些需要经常增删结点而动态变化的数据结构。

链接方法的缺陷在于：为了访问结点集 K 中的某个结点，必须使用指向该结点的指针。当不知道结点指针时，为了在结点集 K 中寻找某个符合条件的结点，就要从链头开始沿着链接结点的链索，通过逐个结点的比较来搜索。

3. 索引方法

索引法是顺序存储的一种推广，通过建造一个由整数域 Z 映射到存储地址域 D 的索引函数 $Y:Z \rightarrow D$，把整数索引值 z 映射到结点的存储地址 $d \in D$，从而形成一个存储了一组指针的索引表，每个指针指向存储区域的一个数据结点。索引表的存储空间是附加在结点存储空间之外的，每一个元素就是指向相应数据结点的指针，即结点存储单元的开始地址。

作为一种存储机制，索引的主要作用是提高检索的效率。当数据量很大时，对数据的检索可能涉及大量读/写磁盘的操作，会影响效率。通过索引则可以降低读/写的数据量，根据检索码确定了被检索数据的存储地址之后再进行相应的读/写。

一般而言，索引函数并非数组那样的简单线性函数。在数据结点长度不等的情况下，索引函数就不是线性函数。索引方法的一个应用就是处理结点大小不等的数据结构。图 1.4 所示的就

是数据结点长度不等的情况。

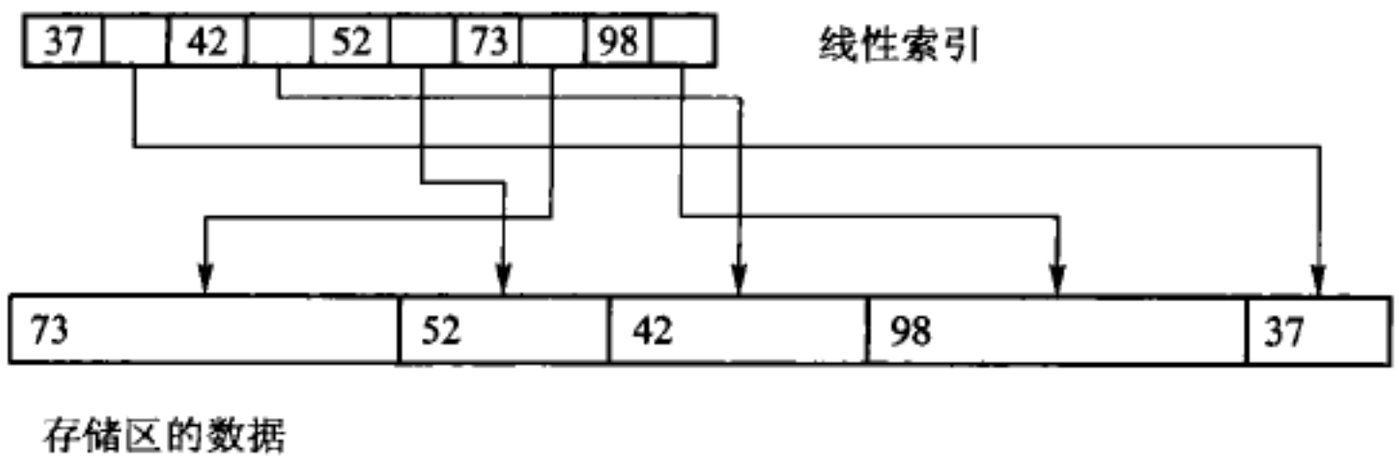


图 1.4 索引示例

4. 散列方法

作为索引法的一种延伸和扩展,散列法利用散列函数进行索引值的计算,然后通过索引表求出结点的指针地址。其主要思想是根据结点的关键码值来确定其存储地址,利用一种称为散列函数(hash functions)的关系把关键码的值映射到存储空间的地址,然后把结点存入此存储单元中。

散列技术的关键问题是如何选择和设计恰当的散列函数、构造散列表、研究构造散列表时的碰撞解决方案等。

一个散列函数把一个给定的关键码映射为一个小于 K 的非负整数,K 的大小取决于具体的应用。散列函数应该满足一些重要的散列性质:散列函数计算出的地址尽可能均匀地分布在构造的散列表地址空间,散列函数的计算应该简单化,以便提高地址计算速度。

在一个具体的应用中,可以根据需要选用以上 4 种存储方法或其组合。例如,树形结构的子结点表表示方法就是顺序和链接的结合。另外,一个逻辑结构可以有多种不同的存储方案,因此在选择存储方法时,还要综合考虑定义在其上的运算及其算法的实现。

1.2.3 抽象数据类型

抽象意味着一个概念可能有多种实现方式。无论在数学领域还是在程序设计领域,抽象的能力都源于这样的一个事实:一旦一个抽象问题得到解决,则很多同类的问题便可迎刃而解。用户自定义数据类型通常即是对多种可能的结构和实现的一种抽象。

抽象数据类型(abstract data type, ADT)的概念是逐步形成的。20 世纪 60 年代末到 70 年代,人们将用户自定义的类型称做抽象数据类型。传统的算法语言没有提供用户自定义类型的机制,20 世纪 60 年代后期,为了实现算法细节和数据内部结构的隐蔽,在 Simula 67 语言中引入了类,随后出现了模块的概念。模块分成模块式和模块体,模块式定义外部可见的运算接口,模块体则定义对外不可见的私有数据和运算。通过接口和实现的分离,模块提供了用户自定义类型的手段,达到数据抽象、信息隐蔽的目的。在这个意义下,模块所定义的数据类型称做抽象数据类型。实际上,模块提供了一种抽象数据类型实现的手段。

作为描述数据结构的一种理论根据，抽象数据类型可以看做是定义了一组操作的一个抽象模型。其特点是将数据和操作封装在一起，使得用户程序只能通过抽象数据类型所定义的操作来访问其中的数据，从而实现信息隐藏，也在很大程度上可以使人们独立于程序的实现细节来理解数据结构的主要性质和约束条件，从更高的抽象级别去讨论和使用数据结构和算法。例如，整数与整数上的加、减、乘、除四则运算可以定义为一个抽象数据类型。

一般而言，抽象数据类型可以用三元组表示：

$$(D,R,P)$$

也即

```
ADT 抽象数据类型名 {
    数据对象 D: <数据对象的定义>
    数据关系 R: <数据关系的定义>
    基本操作 P: <基本操作的定义>
} ADT 抽象数据类型名
```

其中，D 是数据对象，R 是 D 上的关系集，D 和 R 二者定义了数据的模型，是对数据的抽象；P 是对 D 的基本操作集，定义了该数据模型的功用，是对数据操作的抽象。

在面向对象语言（例如 Java 和 C++）中，一个操作集合经常用接口来给出而不需要透露这些操作实现的细节。C++用类（包括模板类）的说明来表示 ADT，用类的实现来实现 ADT。因此，C++中实现的类相当于数据的存储结构及其在存储结构上实现的对数据的操作。

ADT 和类的概念实际上反映了程序或软件设计的两层抽象：在概念层（或称为抽象层）上描述问题的 ADT；在实现层上描述问题的类。此外，C++中的类只是一个由用户定义的普通类型，可用它来定义变量（称为对象或类的实例）。因此，抽象数据类型可看成是数据的逻辑结构以及在逻辑结构上定义的抽象操作。本书在引入和介绍每种数据结构时都将采用代码 1.2 所示的模板类来描述。

【代码 1.2】 抽象数据类型定义模板。

```
template <class Type>                    // 模板参数为类型 Type
class className {
private:                                 // 数据结构的取值类型和取值空间
    Type dataList;                       // 定义数据及其存储方式
    ...
public:                                  // 运算集
    methodName();                        // 定义对数据的操作
    ...
};
```

【代码 1.2 结束】

1.3 算　法

算法的研究可以追溯到公元前300多年,算法的中文名称出自周髀算经,而英文名称 Algorithm 则源于9世纪波斯数学家比阿勒·霍瓦里松的名字 al - Khwarizmi,他首先在数学上提出了算法这个概念。算法的第一次编写是 Ada Byron 于1842年为巴贝奇分析机编写求解伯努利方程的程序,因此 Ada Byron 被大多数人认为是世界上第一位程序员。由于查尔斯·巴贝奇(Charles Babbage)未能完成他的巴贝奇分析机,这个算法最终未能在巴贝奇分析机上执行。20世纪的英国数学家图灵(Turing)提出了著名的图灵论题,并提出一种假想的计算机的抽象模型,这个模型被称为图灵机。图灵机的出现解决了算法定义的难题,对算法的发展起到了重要的作用,使得大多数算法都可以转换成程序交给计算机执行,原来认为靠人力难以完成的算法也变得可行。由此也拉开了算法研究和实用的帷幕。

从应用范围来看,算法可分成数值算法和非数值算法两大类。从工作方式看,算法可分成串行算法和并行算法两大类。早在20世纪70年代,D. E. Knuth 就指出,计算机科学就是研究算法的学问。

1.3.1 算法的概念

简单来说,算法是对特定问题求解过程的描述,是指令的有限序列,即为解决某一特定问题而采取的具体而有限的操作步骤。程序是算法的一种实现,计算机按照程序逐步执行算法,实现对问题的求解。

求最大公因子的辗转相除法以及求解联立线性方程组的主元素消除法都是算法的典型例子。求解一个问题通常用该问题的输入数据类型和该问题所要求解的结果(算法的输出数据)所应遵循的性质来描述。以求最大公因子算法为例,它的输入是整数类型,输入数据是任意给定的两个正整数 n 和 m,而算法的输出则是既能整除 n 又能整除 m 的所有公因子中的最大的非负整数。对于求解联立线性方程组,它的输入数据是方程的系数矩阵和方程等式右侧的常数向量,而其输出结果数据是方程变元的一组取值,它们代入方程应该满足所给的等式。

算法一般具有以下性质:

(1) 算法的通用性:算法通用性的含义是,对于那些符合输入数据类型的任意输入数据,都能根据算法进行问题求解,并保证计算结果的正确性。

(2) 算法的有效性:算法是有限条指令组成的指令序列,其中每一条指令都必须能够被人或机器所确切执行。指令的类型应该明确规定,仅限于若干明确无误的指令动作,是一个有限的指令集。例如,辗转相除法,其基本动作(指令)只包括整数相除求余数和商数以及判断余数是否为0等几个指令。这些基本指令不仅能够被准确无误地执行,而且每一步的执行结果具有确定

的类型。算法的有效性不仅指出了每一步指令能够被有效执行,而且也规定了指令的执行结果,其结果应具有确定的数据类型,是可预期的。

(3) 算法的确定性:算法每执行一步之后,关于下一步怎么执行,应该有明确的指示。下一步动作可以是条件判断、分支指令、顺序执行一条指令或者指示整个算法的结束等。算法的确定性就是要保证每一步之后都有关于下一步动作的指令,不能缺乏下一步指令(即"被锁住"状态)或仅含有模糊不清的指令。

(4) 算法的有穷性:算法的执行必须在有限步内结束。换言之,算法不能含有死循环。注意,算法由有限条指令所组成这一事实本身并不能保证算法执行的有穷性,设计算法时,应该关注算法的结束条件。

总之,了解这些基本性质有助于人们更确切地理解算法的概念。

1.3.2 算法设计

算法设计与算法分析是计算机科学的核心问题。算法设计是求解问题时必须要考虑的,其任务是对各类具体问题设计求解的方法和过程。常用的算法设计方法有穷举法(enumeration)、回溯法(backtrack)、分治法(divide and conquer)、递归法(recursion)、贪心法(greedy)和动态规划法(dynamic programming)等。下面分别就这些算法设计方法予以简单介绍。

1. 穷举法

穷举法也称枚举法,其基本思想是将问题空间中的所有求解对象一一列举出来,然后逐一加以分析、处理,并验证结果是否满足给定的条件。穷举完所有对象后,问题将最终得以解决。穷举法具有以下一些特点:

(1) 对象应该是有限的,有明显的穷举范围。

(2) 有穷举规则,可按某种规则列举对象。

(3) 一时找不出解决问题的更好途径。

一般而言,对一个问题空间进行全局的穷举,往往很费时间,效率上难以满足要求,但在问题的局部采用穷举法常很有效。

例如,C/C++语言中常用的分支语句 switch 就是一种穷举。

2. 回溯法

回溯法也称试探法,该方法的基本思想是将问题的候选解按某种顺序逐一枚举和检验,来寻找一个满足预定条件的解。当发现当前候选解不可能是解时,就退回到上一步重新选择下一个候选解。如果当前候选解还不满足问题规模要求,但是满足所有其他要求,那么继续扩大当前候选解的规模,并继续试探。如果当前候选解满足包括问题规模在内的所有要求时,该候选解就是问题的一个解。在回溯法中,放弃当前候选解,寻找下一个候选解的过程称为回溯。扩大当前候选解的规模,以继续试探的过程称为向前试探。

第 5、6、7 章将要介绍的深度优先搜索就是典型的回溯法实例。

3. 分治法和递归法

分治法的设计思想很朴素，其要点是在遇到一个难以直接解决的大问题时，将其分割成一些规模较小的子问题，以便各个击破，分而治之，然后把各个子问题的解合并起来，得出整个问题的解。分治法是一种自顶向下的设计方法。

如果原问题可分割成若干个子问题，这些子问题都可解，并且可利用这些子问题的解求出原问题的解，那么这种分治法就是可行的。由分治法产生的子问题往往是原问题的较小模式，在这种情况下，反复应用分治手段，可以使问题的规模不断缩小，最终缩小到很容易直接求出其解的规模，这自然会导致递归过程的产生。分治与递归如同一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。

本书第 8、9 章将要介绍的快速排序、归并排序、二分检索等都是分治法的应用实例。

4. 贪心法和动态规划法

贪心法的基本思路是从问题的初始状态出发，依据某种贪心标准，通过若干次的贪心选择而得出最优值（或较优解）。贪心法并不是从整体上考虑问题，它所做出的选择只是在某种意义上的局部最优解，寄希望于由局部的最优解构建最终的全局最优解。选择能产生问题最优解的最优度量标准是使用贪心法的核心问题。

比贪心法更为一般的动态规划法通常也用于求解具有某种最优性质的问题。当一个问题的解可以看成一系列判定的结果时，可以利用动态规划方法设计其求解算法。在这类问题中，可能会有许多可行解，希望从中找到具有最优值的解。动态规划法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。与分治法不同的是，适合于用动态规划求解的问题，经分解得到的子问题往往不是相互独立的。若用分治法来解这类问题，则分解得到的子问题数目太多，有些子问题被重复计算了很多次。如果能够保存已解决的子问题答案，而在需要时利用这些已求得的答案，就可以避免大量的重复计算，节省时间。一般用一个表来记录所有已解的子问题的答案。不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中，这就是动态规划法的基本思路。

第 7 章介绍的最小生成树的 Prim 算法和 Kruskal 算法、求最短路径的 Dijkstra 算法和 Floyd 算法等都是动态规划的实例。其中，Floyd 算法是比较典型的动态规划，而其他 3 种算法是贪心算法。

可以选择和组合这些基本算法设计方法，根据问题的资源约束和要求，设计出相关数据结构和算法来求解问题。运用算法的常见方式有以下 4 种：

（1）算法的组合。

（2）经典算法的变形与推广。

（3）研究困难问题的特殊情况。

(4) 探索新的算法。

1.4 算法分析

一般而言,同一类问题总是存在着多种解决方案,即针对一个问题可以有多个算法。这些算法也许各有其优缺点以及适用的场景。那么如何在多个算法中按照某种原则进行取舍?这就需要用到算法分析技术来评价算法的效率。

算法分析的任务是利用数学工具,对每一个具体算法讨论其各种复杂度,以探讨某种具体算法适用于哪类问题,或某类问题宜采用哪种算法。其目的在于从解决同一问题的不同算法中选择出比较合适的一种,或者是对原有的算法进行改造、加工,使其更优。

为比较不同算法的效率,Juris Hartmanishe 和 Richars E. Stearns 提出了一种称为计算复杂性(computational complexity)的方法来度量算法的难度。计算复杂性分析显示了实施一个算法所需的努力或代价。算法的复杂性是度量算法效率、评价算法优劣的重要依据。对于一个算法而言,其复杂性的高低体现在运行该算法所需要的计算机资源的多少上,所需的资源越多表明该算法的复杂性越高;反之,该算法的复杂性越低。计算机资源中最重要的是时间资源(即处理器)和空间资源(即存储器),因而算法的复杂性通常体现为时间复杂性和空间复杂性两个指标。因此,本书通过时间和空间两个方面来分析和评价算法的效率。

不言而喻,对于任意给定的问题,设计出复杂性尽可能低的算法是人们所追求的一个重要目标;另一方面,当给定的问题已有多种算法时,选择其中复杂性最低者也是在选用算法时应遵循的一个重要准则。因此,算法的复杂性分析对算法的设计或选用有着重要的指导意义和实用价值。

对于算法的复杂性,首先需要明确两个问题:

(1) 怎样表达一个算法的复杂性。

(2) 怎样计算一个算法的复杂性。

影响程序运行时间的因素有很多。例如,一个运行在 Cray 巨型机上的算法,如果放在 PC 上肯定会慢很多。另一方面,一个效率较差的算法运行在 Cray 上可能比一个效率极好的算法运行在 PC 上还要快许多。由此可见,一个算法的运行时间是和系统相关的。另外,即使运行在同样机器上的算法,如果所采用的程序语言不同,其运行时间也可能有所不同。采用编译性的语言往往要比解释性的语言快许多。例如,采用 C 语言编写的程序比用 LISP 语言编写的采用同样算法的程序快将近 20 倍。

综上所述,评估算法的效率不能采用诸如微秒、纳秒这样的真实的时间单位。换言之,对于算法分析来说,最重要的不是具体的运行时间,而是算法与输入规模之间的关系,即用一定"规模(size)"的数据作为输入时程序运行所需的"基本操作(basic operation)"数来描述时间效率。例如,如果数据规模 n 和运行时间 t 间存在一个线性的关系,$t_1 = cn_1$,那么数据规模增加到原来

的 5 倍后，运行时间也相应地增加同样的倍数，即 $n_2 = 5n_1$，则 $t_2 = 5t_1$。同理，如果 $t_1 = \log_2 n$，则 n 的规模加倍意味着时间增加一个单位，即，$n_2 = 2n$，$t_2 = \log_2(2n) = \log_2 n + 1 = t_1 + 1$。此处的“规模”和“基本操作”的关系视具体的算法而定。“规模”一般指输入量的数目，例如在排序问题中，问题的规模即为待排序元素的个数；而完成一个“基本操作”所需的时间应该与被操作的具体数值无关，例如两个整数相加的操作时间与这两个整数的具体取值没有关系。

1.4.1 渐进分析方法

一般情况下，用来表示数据规模和时间关系的函数都相当复杂。计算这样的函数时通常只考虑大的数据，而那些不能显著改变函数量级的部分都可以忽略掉，其结果就是原函数的一个近似值，这个近似值在数据规模很大时会足够接近原值。这种方法称为渐进算法分析（asymptotic algorithm analysis）方法，简称渐进分析。例如，下面的这个函数：

$$f(n) = n^2 + 100n + \log_{10} n + 1\,000$$

在 n 的取值很小时，例如为 1 时，最后的常数项 1 000 对函数值的贡献最大；当 $n = 10$ 时，第 2 项（$100n$）和最后的 1 000 具有相同的贡献；当 n 达到 100 时，前两项的贡献相同；但当 n 大于 100 后，第 2 项的贡献就小于第 1 项，而且 n 越大，第 2 项对函数值而言就越微不足道。由于第 1 项是二次增长的，当 n 取值很大时，函数的取值主要依赖于第 1 项的贡献。

渐进分析是对资源开销的一种不精确的估计，它提供的是对算法所需资源开销进行评估的简单化模型，以便把注意力集中在最重要的部分。应该注意的是，并非所有的情况下都可以忽略常数部分。当算法要解决的问题规模很小时，各项系数和常数项就会起到举足轻重的作用，如同上面的函数所示。因此，用于上万个数的排序算法也许并不适用于仅对 10 个数的排序。

1. 大 O 表示法

渐进分析最常用的表示方法是估计函数的增长趋势，采用由 Paul Bachmann 于 1894 年引入的大 O 表示法。假设 f 和 g 为从自然数到非负实数集的两个函数。

定义 1：如果存在正数 c 和 N，使得对任意的 $n \geqslant N$，都有 $f(n) \leqslant cg(n)$，则称 $f(n)$ 在集合 $O(g(n))$ 中，或简称 $f(n)$ 是 $O(g(n))$ 的。

该定义说明了函数 f 和 g 之间的关系，既可以表达成函数 $g(n)$ 是函数 $f(n)$ 取值的上限（upper bound），也可以说函数 f 的增长最终至多趋同于 g 的增长。

因此，大 O 表示法提供了一种表达函数增长率上限的方法。换言之，若某种算法在 $O(g(n))$ 中只是表明了该算法最多会差到何种程度。当然，一个函数增长率的上限可能不止一个。例如，一个在集合 $O(n)$ 中的函数也一定在集合 $O(n^2)$ 中，同时也在集合 $O(n^3)$ 中。大 O 表示法给出了所有上限中最小的那个上限。

下面列出了大 O 表示法所具有的某些有益特性，可在计算算法效率时使用。

（1）如果函数 $f(n)$ 是 $O(g(n))$ 的，$g(n)$ 是 $O(h(n))$ 的，则 $f(n)$ 是 $O(h(n))$ 的。

（2）如果函数 $f(n)$ 是 $O(h(n))$ 的，$g(n)$ 是 $O(h(n))$ 的，则 $f(n) + g(n)$ 是 $O(h(n))$ 的。

(3) 函数 an^k 是 $O(n^k)$ 的,符号 a 表示不依赖于 n 的任意常数。

(4) 若 $f(n)=cg(n)$,则 $f(n)$ 是 $O(g(n))$ 的。

(5) 对于任何正数 a 和 b,且 $b\neq 1$,函数 $\log_a n$ 是 $O(\log_b n)$ 的。即任何对数函数无论底数为何值,都具有相同的增长率。

(6) 对任何正数 $a\neq 1$,都有 $\log_a n$ 是 $O(\log_2 n)$ 的。本书把 $\log_2 n$ 简写为 $\log n$。

常见的上限 $g(n)$ 有以下若干种,表达式中的符号 a 是不依赖于 n 的任意常数,$\text{rate}_{n\to\infty} f(n)$ 表示 n 趋于无限大时函数 $f(n)$ 的极限:

(1) $g(n)=1$, 常数函数,不依赖于数据规模 n。

(2) $g(n)=\log n$,对数函数,它比线性函数 n 增长慢。

(3) $g(n)=n$,线性增长,随着问题规模 n 而增长。例如:

$$\text{rate}_{n\to\infty}(n\text{ 个 }a\text{ 相加})=\text{rate}_{n\to\infty}(n\times a)=O(n)$$

(4) $g(n)=n^2$,二阶增长,例如:

$$\text{rate}_{n\to\infty}(1+2+3+\cdots+n)=\text{rate}_{n\to\infty}(n\times(n+1)/2)=O(n^2)$$

(5) $g(n)=n\log(n)$,其增长率的阶数低于二阶,但高于一阶线性。例如:

$$\text{rate}_{n\to\infty}\sum_{i=1}^{\log n}\sum_{j=1}^{n}a=O(n\log(n))$$

这种 $n\log(n)$ 型的渐进式一般出现在树形数据结构的算法中。

(6) $g(n)=a^n$,指数增长,随问题规模 n 快速增长。

这种指数型的渐进式往往出现在递归定义的函数计算中。值得注意的是,指数增长的渐进式比任何高次的多项式函数如 n^3、n^4 等都增长得更快。

这些渐进式的函数曲线,在 n 很大的时候其大小差别非常大。举例来说,令时间单位为 1 μs,在 n 等于 1 000 时,对于线性增长率,时间开销 $T(n)=n$ 为 1 000 μs,即 1 ms;而对二次增长率 $T(n)=n^2$,其时间开销是 10^6 μs,即 1 s;对于三次增长率的 $T(n)=n^3$ 而言,其时间开销就是 1 000 s(约 16 分钟)。最后,对于指数型的增长率 $T(n)=2^n$ 所花费的时间约为 10^{286} 年,而迄今地球的年龄还不超过 10^{10} 年。具有指数增长率的算法简称指数爆炸型算法,在应用时需要谨慎使用。

表 1.2 给出了各个时间量级在现实世界所对应的物体速度。同时,图 1.5 给出了常用函数的增长趋势。

表 1.2 各个量级的实例

m/s	英制度量衡	现实世界的例子
10^{-11}	1.2 英寸/世纪	钟乳石的生长速度
10^{-10}	1.2 英寸/十年	大陆板块的漂移速度
10^{-9}	1.2 英寸/年	指甲的生长速度

续表

m/s	英制度量衡	现实世界的例子
10^{-8}	1 英尺/年	头发的生长速度
10^{-7}	1 英尺/月	杂草的生长速度
10^{-6}	3.4 英寸/天	冰河的流动速度
10^{-5}	1.4 英寸/时	表的分针转动速度
10^{-4}	1.2 英尺/时	胃肠的蠕动速度
10^{-3}	2 英寸/分	蜗牛的速度
10^{-2}	2 英尺/分	蚂蚁的速度
10^{-1}	20 英尺/分	巨龟的速度
1	2.2 英里/时	人类的散步速度
10^{1}	22 英里/时	人类疾跑速度
10^{2}	220 英里/时	螺旋桨飞机的速度
10^{3}	37 英里/分	喷气式飞机的速度
10^{4}	370 英里/分	宇宙飞船的速度
10^{5}	3 700 英里/分	流星撞击地球的速度
10^{6}	620 英里/秒	地球自转速度
10^{7}	6 200 英里/秒	卫星从洛杉矶到纽约的速度
10^{8}	62 000 英里/秒	光速的三分之一

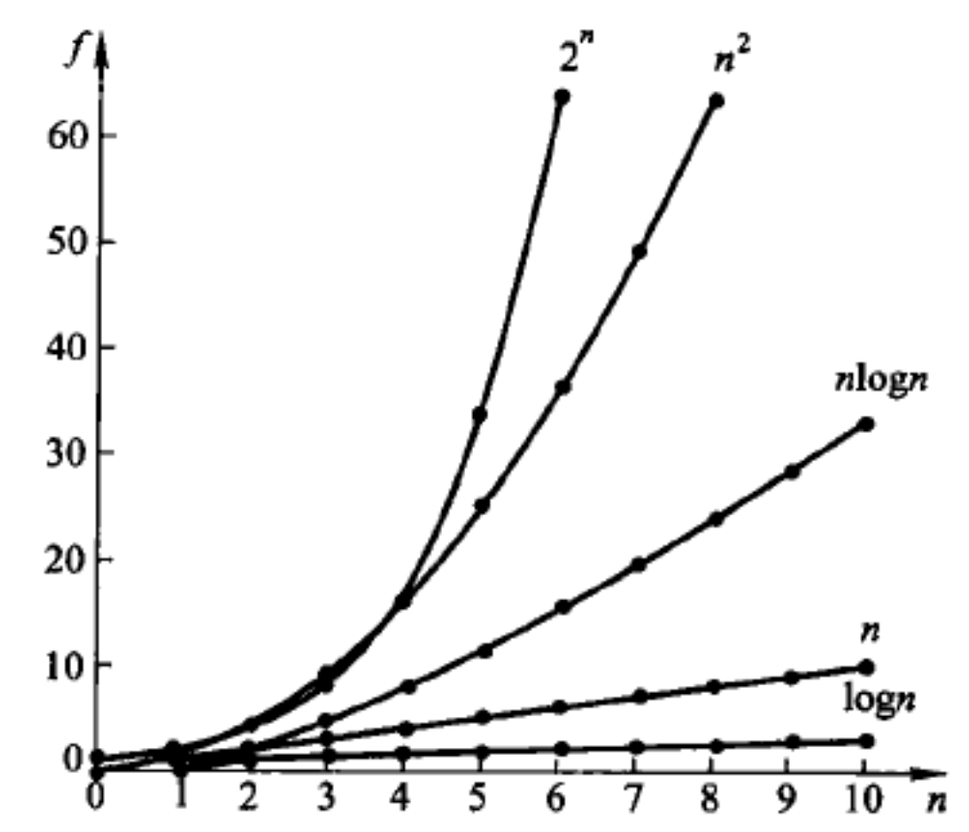


图 1.5 常用函数的增长趋势

2. Ω 表示法

大 O 表示法给出了函数增值率的上限。与此相对，还有 Ω 表示法，Ω 读做“欧米伽”(Omega)。其定义如下：

定义2：如果存在正数 c 和 N，使得对所有的 $n \geqslant N$，都有 $f(n) \geqslant cg(n)$，则称 $f(n)$ 在集合 $\Omega(g(n))$ 中，或简称 $f(n)$ 是 $\Omega(g(n))$ 的。

此定义说明了 $cg(n)$ 是函数 $f(n)$ 取值的下限(lower bound)，也可以说函数 $f(n)$ 的增长最终至少是趋同于函数 $g(n)$ 的增长。

大 O 表示法和 Ω 表示法的唯一区别在于不等式的方向不同。正如大 O 表示法，Ω 表示法是在函数增值率的所有下限中那个最大的下限。

3. Θ 表示法

大 O 表示法和 Ω 表示法描述了某一函数增长的上限和下限。当上、下限相同时，则可用 Θ(读做“希塔”)表示法。如果一个函数既在集合 $O(g(n))$ 中又在集合 $\Omega(g(n))$ 中，则称其为 $\Theta(g(n))$。即存在正常数 c_1、c_2 以及正整数 N，使得对于任意的正整数 $n > N$ 有下列两不等式同时成立：

$$c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$$

此定义具体给出了下限估计和上限估计两种估计式。例如：

$$f(n) = 100 \times n^2 + 5 \times n + 500$$

令 $g(n) = n^2$，此时存在常数 $c_1 = 100$，$c_2 = 105$，$N = 10$，当 $n > N$ 时，

$$c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$$

成立。因此，可以说 $f(n)$ 为 $\Theta(n^2)$。

4. 渐进分析的实例

通过对算法所需的时间和空间代价的估算，渐进分析方法可以衡量算法的复杂度。实际上，大多数情况下，算法的时间复杂度是根据算法执行过程中需要实施的赋值、比较等基本运算数目的多少来衡量的。

先从一个简单的例子开始分析。下面是一段对数组中的各个元素求和的代码：

```
for (i = sum = 0; i < n; i ++)
    sum += a[i];
```

其中，主要的操作为赋值运算，因此该算法的时间代价主要体现在赋值操作的数目上。在循环开始之前有两次赋值，分别对 i 和对 sum 进行；循环进行了 n 次，每次循环中执行两次赋值，分别对 sum 和 i 进行更新操作。总共有 $2+2n$ 次赋值操作，其渐进复杂度为 $O(n)$。

在循环中嵌套循环，其复杂度会相应增大。例如，下面这段代码依次求出给定数组的所有子数组中各元素之和：

```
for (i = 0; i < n; i ++) {
    for (j = 1, sum = a[0]; j <= i; j ++)
        sum += a[j];
```

```
        cout << "sum for subarray 0 through" + i + " is " + sum << endl;
    }
```

循环开始前，有一次对 i 的赋值操作。之后，外层循环共进行 n 次，每个循环中包含一个内层循环以及对 i、j、sum 分别进行赋值的操作；每个内层循环执行 2 个赋值操作，分别更新 sum 和 j；共执行 i 次（$i=1,2,\cdots,n-1$）。因此，整个程序总共执行的赋值操作为

$$1+3n+\sum_{i=1}^{n-1}2i=1+3n+2(1+2+\cdots+n-1)=1+3n+n(n-1)$$
$$=O(n)+O(n^2)=O(n^2)$$

一般情况下，循环中嵌套循环会增加算法的复杂度，但也并非总是如此。例如，在上面的实例中，如果只对每个子数组的前 5 个元素求和，则相应的代码可采用下面的方式：

```
for ( i=4; i<n; i++ )
    for (j=i-3,sum=a[i-4]; j <= i; j++)
        sum += a[j];
```

此时，外层循环进行 $n-4$ 次。对每个 i 而言，内层循环只执行 4 次，每次的操作次数与 i 的大小无关：8 次赋值操作。外加对 i 的初始化，整个代码总共进行 $O(1)+8(n-4)=O(n)$ 次赋值操作。尽管存在嵌套循环，但算法的整体时间复杂度依然呈线性增长。

另外，对于排序算法而言，其主要时间开销体现在比较和交换（或移动）等操作上，具体可参考第 8 章的各算法分析。

1.4.2 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，其时间复杂度也不同。换言之，算法的渐进分析往往无法独立于输入数据的状态而进行。这是因为算法实际执行的操作往往依赖于算法中分支条件的走向，而这些分支走向又取决于输入数据的取值。为此，对算法进行渐进分析时会根据输入数据的取值分情况进行，以便确切了解各种算法所适用的情况以及能否在规定的响应时间内完成。

例如，求一个数组的所有有序子数组中最长的一个。在数组[1,8,1,2,5,0,11,9]中，这个最长的有序子数组为[1,2,5]，长度为 3。可用以下代码实现：

```
for( i=0,length=1; i<n-1;i++ ) {
  for( j1=j2=k=i;k<n-1 && a[k]<a[k+1]; k++,j2++);
  if (length<j2-j1-1)
      length=j2-j1+1;
}
```

这段代码的时间代价和数组 a 中元素的实际取值状态相关。根据这些元素的初始状态可以分成以下几种情况：

（1）如果数组 a 的所有元素是以降序方式输入的，那么外层循环执行 $n-1$ 次，每次内层循

环只执行1次，整个的时间开销为 $O(n)$。

(2) 如果数组 a 的所有元素是以升序方式输入的，那么外层循环执行 $n-1$ 次，对每个 $i(i=0,1,\cdots,n-1)$，内层循环需要执行 $(n-1-i)$ 次，整个的时间开销为 $O(n^2)$。

(3) 在大多数情况(即平均情况)下，数组的元素是无序的，既不按升序也不按降序输入。下面介绍一下如何计算平均情况下的算法复杂度。

一般而言，计算平均情况的复杂度应该考虑算法的所有输入情况，确定针对每种输入情况算法所需的操作数目。在简单情况下，如果每种输入出现的概率相同，可把针对每种输入的操作数目依次相加，再除以输入的总数目来得到平均的开销。但是，每种输入的出现概率并非总是相同，此时分析平均情况的复杂度就需要把每种输入出现的概率作为权值加以考虑，即。

$$C_{\text{avg}} = \sum_i p(\text{input}_i)\,\text{steps}(\text{input}_i)$$

此处，假设可以事先得知输入的概率分布情况。$p(\text{input}_i)$ 为第 i 种输入的出现概率，steps (input_i) 为算法处理第 i 种输入时所需的操作或步骤数。当然，所有的概率均为非负数，且满足 $\sum_i p(\text{input}_i) = 1$。

尽管平均情况的复杂度是算法在输入规模为 n 时的典型表现，但平均情况的分析并不总是可行的，因为这需要了解算法的实际输入在所有可能的输入集合中的分布状况。例如，上面例子中获取数组中元素的平均分布情况并不是很容易。

以从一个规模为 n 的一维数组中找出一个给定的 K 值为例(假设该数组中有且仅有一个元素的值为 K)，顺序检索法将从第一个元素开始，依次检查每一个元素，直到找到 K 为止。

最佳情况下，数组中第1个元素就是 K，此时只要检查一个元素即可。

最差情况下，K 是数组的最后一个元素，此时算法需要检查数组中所有的 n 个元素才能找到。

平均情况下，如果 K 出现在数组中每个位置上的概率相等，即 K 出现在每个单元中的概率均为 $\frac{1}{n}$，平均需要

$$\frac{1+2+\cdots+n}{n} = \frac{n+1}{2}$$

次比较才能找到。

概率不相等时，例如出现在第1个位置的概率为1/2，第2个位置上的概率为1/4，而出现在其他位置的概率相等，即

$$\frac{1-1/2-1/4}{n-2} = \frac{1}{4(n-2)}$$

则查找 K 平均需要

$$\frac{1}{2}+\frac{2}{4}+\frac{3+\cdots+n}{4(n-2)} = 1+\frac{n(n+1)-6}{8(n-2)} = 1+\frac{n+3}{8}$$

次比较，比等概率的$\frac{n+1}{2}$快将近4倍。

由此可见，第1种情况是最佳的，所需的步骤最少；第2种情况所需的步骤最多，是最差的情形；第3种则介于最佳和最差之间。那么分析一种算法时，应该研究最佳、最差还是平均情况？一般而言，最佳情况发生的概率太小，并不能作为算法性能的代表，但它可以帮助算法设计者或使用者了解某个算法在何种情况下适用。而最差情况可以让人了解一个算法至少能做多快，这点在实时系统中尤其重要。例如，在空运处理系统中，一个绝大部分情况下能管理n架飞机的算法，如果不能在规定时间内管理n架飞机，则该算法是不可接受的。此外，对于多数算法而言，最坏情况和平均情况的时间开销的公式虽然不同，但是往往只是常数因子大小的区别，或者常数项大小的区别。

上面3种情况的复杂度都相对比较简单，可以得到很精确的函数关系，但复杂度与数据规模之间的关系并非总是一目了然。尤其是平均情况的复杂性分析往往需要复杂的计算，此时前面介绍的大O、Ω和Θ等渐进分析法便可以派上用场。

1.4.3 时间和空间的折衷

对于空间开销，也可以采用类似的渐进分析方法。但是，很多常见算法所使用的数据结构是静态的存储结构。所谓静态是指算法所使用的存储空间在算法执行过程中并不发生变化。一旦确定了输入数据和问题规模，其数据结构大小也就确定下来。对于这种静态数据结构，空间开销的估算比较容易，往往与所涉及的问题规模成正比（空间开销为线性增长），或者不随问题的规模而增大（空间开销为常数）。本书后续章节对这类使用静态数据结构的算法一般将仅限于讨论其时间代价。

当然，也经常会遇到一些使用动态数据结构的算法，它们的存储空间是变化的，在算法运行过程中有时会有数量级的增大或缩小。对于这种情况，空间开销的分析和估计是十分必要的。

在算法设计分析中，还涉及一个“时空资源的折衷原理”。这个原理可以简述如下：对于同一个问题求解，一般会存在多种算法。而这些算法在时间和空间开销上的优劣往往表现出“时空折衷”的性质。所谓“时空折衷”是指，为了改善一个算法的时间开销，往往可以增大空间开销为代价；反之亦然。例如，为了从公司黄页数据表中快速查询公司电话（假设每个公司的名称、地址和电话都存储在该数据表中），可以附加一个散列式索引表，其散列函数可以将公司名称快速变换为索引值，通过这个索引值可以读出一个指针，指向存储该公司电话的存储单元。这种散列法附加了索引表的空间开销，但节省了时间，用散列函数的简单计算代替了原本耗时的在黄页中逐项进行公司名称的比较操作，时间开销从线性增长改善为常数时间，而空间开销方面则增添了一个线性增长率的存储空间。在设计算法时经常采用这种以空间开销换取时间开销的办法。为此，就需要对原有存储结构做出修改，或者想出全新的、更适合逻辑数据结构使用的存储方案。当然，有时也可以牺牲计算机的运行时间，通过增大时间开销来换取存储空间的节省，例如树的

顺序存储。

1.4.4 求解问题时数据结构的选择和评价

如同1.1节所述，利用计算机求解一个给定问题时，一个关键任务是确立问题模型所采用的数据结构。一旦确定了数据结构，算法的设计便水到渠成。

根据应用的具体情况，设计或选择数据结构可以遵循以下一些原则：

(1) 仔细分析所要解决的问题，特别是求解问题所涉及的数据类型和数据间的逻辑关系。

(2) 在算法设计之前往往先进行数据结构的初步设计。

(3) 注意数据结构的可扩展性，包括考虑当输入数据的规模发生改变时，数据结构是否能够适应。同时，数据结构应该适应求解问题的演变和扩展。

(4) 数据结构的设计和选择也要比较算法的时空开销的优劣。

数据结构的选择和评价是复杂的，需要考虑的因素也不限于上述的几条，一般要具体情况具体分析。本书会针对具体的问题进一步阐述和检验这些原则。

本章小结

本章介绍了“数据结构与算法”以及本书要用到的一些预备知识和相应的设计原则。从求解问题的角度开始引入数据结构的基本概念、算法的概念和基本性质、算法设计的基本方法以及衡量算法效率的方法。

利用计算机求解问题的过程是一个问题抽象、数据抽象和算法抽象的过程。首先，从分析问题的要求、明确需求开始；然后，对问题进行分析和抽象，建立问题的抽象模型；在此基础上，确定该模型所采用的数据结构，设计求解该问题的算法；最后，选定某种程序设计语言来编码实现。

数据结构描述的是按照一定逻辑关系组织起来的待处理数据元素的表示及相关操作，涉及数据的逻辑结构、数据的存储结构和数据的运算。

数据的逻辑结构是从具体问题抽象出来的数学模型，反映了事物的组成结构及事物之间的逻辑关系。数据结构重点研究的是数据间的结构关系，组成结构的所有元素都看做结点。结点是数据结构中数据的基本单位。结点的数据类型既可以是基本数据类型，也可以是复合数据类型。根据结点间关系 r 所刻画的数据结构的性质可把逻辑结构分为线性结构、树形结构和图结构三大类。

数据的存储结构所要解决的问题是各种逻辑结构在计算机中的物理存储表示，即从逻辑结构到存储结构的一种映射。同一种逻辑结构可以采用不同的映射关系来建立到存储结构的转换，从而形成不同的存储表示方式。常用的基本存储映射方法有顺序方法、链接方法、索引方法和散列方法。

作为描述数据结构的一种理论根据，抽象数据类型 ADT 可以看做是定义了一组操作的一个抽象模型，在很大程度上可以帮助人们独立于程序的实现细节来理解数据结构的主要性质和约束条件，从而达到数据抽象、信息隐蔽的目的。

算法是对特定问题求解过程的描述，是指令的有限序列。程序是算法的一种实现，计算机按照程序逐步执行算法，实现对问题的求解。算法应具备通用性、有效性、确定性、有穷性等基本性质。

算法设计与算法分析是计算机科学的核心问题。算法设计的任务是对各类具体问题设计良好的算法并研究设计算法的规律和方法。常用的算法设计方法有穷举法、回溯法、分治法、递归法、贪心法和动态规划法等。算法分析的任务是利用数学工具，对每一个具体算法讨论其各种复杂度，以探讨各种具体算法的具体适用场景和应用。其目的在于从解决同一问题的不同算法中选择出比较合适的一种，或者是对原有的算法进行改造、加工，使其更优。

针对同一个问题一般会存在多种算法，这些算法在时间和空间开销上的优劣往往表现出“时空折衷”的性质。需要根据应用的具体情况，在设计或选择数据结构时进行适当的权衡。

习　题

1. 设计一个算法，从大到小依次输出顺序读入的 3 个整数 X、Y 和 Z 的值。
2. 试举例说明何为数据结构。
3. 数据的存储方式有哪些？简要介绍每一种存储方法的特点。
4. 分析下面程序段中循环语句的执行次数。

```
int i = 0; s = 0; n = 100;
do {
    i = i + 1;
    s = s + 10 * i;
} while (!((i < n) && (s < n)));
```

5. 阅读下列算法，指出算法 A 的功能和时间复杂度，其中 h、g 分别为单循环链表中两个结点指针。

```
void B(int * s, int *q) {
    int * p;
    p = s;
    while (p->next != q)
        p = p->next;
    p->next = s;
}
void A (int* h, int* g) {
    B(h,g);
    B(g,h);
}
```

6. 设 n 是偶数，计算运行下列程序段后 m 的值并给出该程序段的时间复杂度。

```
m = 0;
for (i = 1; i <= n; i++)
    for (j = 2 * i; j <= n; j++)
        m = m + 1;
```

7. 有下列运行时间函数：

(1) $T_1(n) = 1\ 000$；　(2) $T_2(n) = n^2 + 1\ 000n$；　(3) $T_3(n) = 3n^3 + 100n^2 + n + 1$；

分别写出相应的大 O 表示的运算时间。

8. 试给出下面两个算法的时间复杂度。

```
(1) for(i = 1; i <= n; i++)
        x = x + 1;
(2) for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            x = x + 1;
```

9. 试给出一个求两个整数的最大公因数的算法，并分析其时间和空间复杂度。

10. 假设计算机的速度为每秒处理 10^{10} 个操作，已知 6 个算法的时间复杂度如下，要求在一个小时之内完成计算。试问对每个算法来说可处理的数据规模为多少？

(1) n^2

(2) n^3

(3) $100n^2$

(4) $n\log_{10}n$

(5) 2^n

(6) 2^{2^n}

11. 设有函数 $f(n)$ 和 $g(n)$，且 $f(n)$ 在集合 $O(g(n))$ 中，下面说法中哪些正确，哪些不正确？对正确的说法给出其证明，对不正确的说法给出其反例：

(1) $\log_2 f(n)$ 是 $O(\log_2 g(n))$

(2) $2^{f(n)}$ 是 $O(2^{g(n)})$

(3) $f(n)^2$ 是 $O(g(n)^2)$

12. 如果一个算法的时间复杂度可表示成下面的公式，试计算其复杂度。

(1) $T(n) = T(\lceil n/2 \rceil) + 1$

(2) $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$

(3) $T(n)2 = 2T(\lfloor n/2 \rfloor) + n$

上 机 题

1. 设 m、n 均为自然数，m 可表示为一些不超过 n 的自然数之和，试编写函数 $f(m,n)$ 计算这种表示方式的数目。例如，$f(5,3) = 5$，有 5 种表示方式：3 + 2、3 + 1 + 1、2 + 2 + 1、2 + 1 + 1 + 1、1 + 1 + 1 + 1 + 1。

2. 试用 C++ 的类声明定义“复数”的抽象数据类型。要求：

(1) 在复数内部用浮点数定义其实部和虚部。

(2) 实现3个构造函数:默认的构造函数没有参数;第二个构造函数将双精度浮点数赋给复数的实部,虚部置为0;第三个构造函数将两个双精度浮点数分别赋给复数的实部和虚部。

(3) 定义获取和修改复数的实部和虚部以及+、-、*、/等运算的成员函数。

(4) 定义重载的流函数来输出一个复数。

第2章 线 性 表

线性结构是最简单且应用最广泛的一种数据结构，其基本特点是结构中的各元素之间满足线性关系。所谓线性关系是指数据元素之间存在着一对一的关系，即存在唯一的开始元素和唯一的终止元素，除此之外的其他元素均有且仅有一个唯一的直接前驱元素和一个唯一的直接后继元素。按此关系可以把结构中的所有元素排成一个线性序列，故称为线性表。26 个英文字母组成的字母表(A,B,C,…,Z)就是一个线性表，表中每个元素是由单个字母字符组成的数据项；一副扑克牌的点数(2,3,…,10,J,Q,K,A)也是一个线性表，其中数据元素是每张牌的点数。

线性结构具有如下的结构特点：

(1) 均匀性：虽然不同线性表的数据元素可以是各种各样的，但在同一线性表中的各数据元素必定具有相同的数据类型和长度。

(2) 有序性：各数据元素在线性表中都有自己的位置，且数据元素之间的相对位置是线性的。

根据应用的特点，线性结构在不同的应用中会采取不同的存储结构和实现方法。因此，线性结构在不同场合具有不同的称谓，例如顺序表、链表、串、栈、顺序文件等；结构中的元素也相应地采取不同的称呼，例如表目、结点、或记录等。一般情况下，这些称谓均可作为同义词来互通使用。

根据结构中元素的复杂程度，线性结构可分为简单线性结构和高级线性结构。简单线性结构包括通常使用的顺序表、队列、散列表等，而多维数组、广义表则属于高级线性结构。

本章将首先从线性表的基本概念、存储表示、基本运算及其实现等方面讨论线性表，本书其他章节将陆续介绍字符串、栈和队列、文件以及广义表。

2.1 线性表的概念

2.1.1 线性表的抽象数据类型

线性表(linear list)是由称为元素(element)的数据项组成的一种有限且有序的序列，这些元素也可称为结点或表目。

若用第 1 章中介绍的二元组 $B=(K,R)$ 来表示线性表，则有

$K = \{k_0, k_1, \cdots, k_{n-1}\}$

$R = \{r\}$

$r = \{ < k_i, k_{i+1} > \mid 0 \leqslant i \leqslant n-2\}$

其中，结点集合 K 中的元素个数 n 称为线性表的长度，当 $n=0$ 时，线性表被称为空表，当 $n>0$ 时称为非空表。k_0 称为开始结点或表首，k_{n-1} 称为终止结点或表尾。

线性关系 r 刻画元素间的前驱或后继关系：k_i 称为 k_{i+1} 的前驱，k_{i+1} 称为 k_i 的后继。因此，线性关系也称为前驱关系或后继关系。直观地讲，线性表的所有结点可以按关系 r 排列成一个线性序列，结点 k_0 是结点 k_1 的前驱，结点 k_1 是结点 k_2 的前驱，……，结点 k_{n-2} 是结点 k_{n-1} 的前驱等。对于后继而言，也同样成立。

从线性表的定义可知其逻辑特征：结点集合 K 中的每个结点在关系 r 上最多只有一个前驱结点和一个后继结点。线性表的关系 r 具有反对称性和传递性。

空表意味着线性表中不包含任何元素。而一个非空线性表 $(k_0, k_1, \cdots, k_{n-1})$ 中的数据元素 k_i $(0 \leqslant i \leqslant n-1)$ 是一个抽象的符号，在不同情况下具有不同的含义。虽然概念上并不反对线性表具有不同类型的数据元素（例如广义表），但本章讨论的简单线性表约定所有元素都具有相同的数据类型，并需事先给出其类型定义、元素或是简单数据类型或是复合结构类型。

在线性表上可以实施的操作（或运算）依赖于具体的应用，但一般不外乎两大类，一类是对整个表的操作，另一类是对表中元素的操作。前者用于创建或置空一个线性表、合并两个线性表、判断线性表是否为空或为满；后者则用于查找线性表中满足一定条件的元素、在线性表中插入或删除指定的元素等。

作为一个抽象数据类型而言，线性表由其数据元素及实施在其上的运算集两部分构成。根据抽象和封装的原则，对线性表的操作只可通过其提供的运算集合中的运算来进行。

下面用 C++类模板的方法，给出线性表类（名字为 list，模板参数为元素类型 T）的一个抽象数据类型说明。其中，每一个运算都用函数接口指出其输入/输出参数及其返回值类型。

【代码 2.1】 线性表的抽象数据类型定义。

```
template <class T>
class List {
    void clear();                                  // 置空线性表
    bool isEmpty();                                // 线性表为空时，返回 true
    bool append(const T value);                    // 在表尾添加一个元素 value，表的长度增 1
    bool insert(const int p, const T value);       // 在位置 p 上插入一个元素 value，表的长度增 1
    bool delete(const int p);                      // 删除位置 p 上的元素，表的长度减 1
    bool getValue(const int p, T& value);          // 把位置 p 的元素值返回到变量 value 中
    bool setValue(const int p, const T value);     // 用 value 修改位置 p 的元素值
    bool getPos (int &p, const T value);           // 把值为 value 的元素所在位置返回到变量 p 中
};
```

【代码 2.1 结束】

线性表抽象数据类型并不是唯一的。针对具体应用的不同要求，线性表的抽象数据类型可以适当增删某些运算，并且可由这些基本的运算来构建更加复杂的运算。

在有些应用中，线性表位置(position)的获取非常重要，可以增加以下这些获取位置的运算函数：

```
bool setPos(int pos);          // 设置当前下标
bool setStart();               // 把当前下标移动到表头
bool setEnd();                 // 把当前下标移动到表尾
bool prev();                   // 当前下标向前左移一位
bool next();                   // 当前下标向右右移一位
```

当前位置的获取有助于对线性表元素依次进行处理，例如下面的代码段中，每个元素依次被传给函数 DoSomething 进行相应的处理。

```
List <T> MyList;
bool b = MyList.setStart();
while (b) {
    DoSomething(Mylist.getValue());
    b = MyList.next();
}
```

线性表运算的具体实现与它在计算机中的物理存储结构密切相关，运算的效率也与存储结构相关。下面介绍顺序和链接这两种常用的线性表存储结构。

2.1.2 线性表的存储结构

线性表的存储结构是指为它所开辟的计算机存储空间以及所采用的程序实现方法，本质上是逻辑结构到存储空间的映射。不仅要为结点集合到存储器单元建立一个映射，同时还要为元素之间的线性关系到相应的存储单元地址间的关系建立映射。

线性表的存储结构主要有两类：

(1) 定长的顺序存储结构，简称顺序表。程序中通过创建数组来建立这种存储结构，其主要特点是为线性表分配一块连续的存储空间，元素顺序地存储在这些地址连续的空间中，以“物理位置相邻”来表示线性表中数据元素之间的关系。定长存储结构的不足之处是限制了线性表的长度变化。

(2) 变长的线性存储结构，也称链接式存储结构，简称链表。链接式存储结构使用指针来表示元素之间的线性关系，利用线性表的前驱和后继关系将各个元素用指针链接起来。变长的线性存储结构对线性表长度不加限制，在需要加入新元素时，可以方便地申请空间并链接到合适的位置上。

线性表的这两种存储结构及其运算的具体实现将在 2.2 节和 2.3 节中详细介绍。

2.1.3 线性表运算分类

按照特性,线性表的运算可以划分为以下 5 类:

(1) 创建线性表的一个实例。

(2) 线性表的析构函数 ~list( ),消除线性表实例并释放所占空间。

(3) 获取有关当前线性表的信息,包括由内容寻找位置、由位置读取元素内容等,不改变线性表的内容。

(4) 访问线性表并改变线性表的内容或结构;例如更新指定元素内容、添加元素、删除元素、清空线性表等。

(5) 辅助管理操作,例如求表的当前长度等。

2.2 顺 序 表

按顺序方式存储的线性表称为顺序表(array - based list),又称为向量(vector),通过创建数组来建立。顺序表中的每个元素按其顺序有唯一的索引值,又称下标值,可以用来方便地访问元素内容。

2.2.1 顺序表的类定义

顺序表的示意图如图 2.1 所示,其中图 2.1(a)为线性表的逻辑结构,图 2.1(b)为线性表的顺序存储结构。

数据元素	k_0	k_1	…	k_i	…	k_{n-1}	…	…
逻辑地址	0	1	…	i	…	$n-1$	…	maxSize-1

(a) 线性表的逻辑结构

数据元素	k_0	k_1	…	k_i	…	k_{n-1}	…	…
存储地址	b	$b+L$	…	$b+i*L$	…	$b+(n-1)*L$	…	$b+(\text{maxSize}-1)*L$

(b) 线性表的顺序存储结构

图 2.1 顺序表的示意图

假设每个元素占用 L 个存储单元,并以第一个单元的存储地址代表数据元素位置。设顺序表的开始结点 k_0 的存储位置记为 $b=\text{loc}(k_0)$,称为顺序表的首地址或基地址,下标为 i 的元素 k_i 的存储位置则为

$$\text{loc}(k_i)=b+i\times L$$

由上述公式可知，每个元素的存储位置都与起始位置相差一个与位序成正比的常数。只要确定了基地址，线性表中任一元素的地址都可以方便地计算出来，从而达到随机存取的效果，因此顺序表是一种随机存取的存储结构。顺序表中两个物理位置相邻的元素在逻辑上互为前驱和后继，因此元素间的物理相邻关系表示了它们逻辑上的相邻关系。

根据2.1.1节给出的线性表抽象数据类型，算法2.2用C++类描述线性表的一种顺序实现。根据顺序表的特点，函数getValue()的实现非常简单。

【代码2.2】 一种顺序表的类定义。

```
template <class T>                          // 假定顺序表的元素类型为T
class arrList : public List<T> {            // 顺序表，向量
private:                                    // 线性表的取值类型和取值空间
    T *aList;                               // 私有变量，存储顺序表的实例
    int maxSize;                            // 私有变量，顺序表实例的最大长度
    int curLen;                             // 私有变量，顺序表实例的当前长度
    int position;                           // 私有变量，当前处理位置
public:                                     // 顺序表的运算集
    arrList(const int size) {               // 创建一个新的顺序表，参数为表实例的最大长度
        maxSize = size;
        aList = new T[maxSize];
        curLen = position = 0;
    }
    ~arrList() {                            // 析构函数，用于消除该表实例
        delete [] aList;
    }
    void clear() {                          // 将顺序表存储的内容清除，成为空表
        delete [] aList;
        curLen = position = 0;
        aList = new T[maxSize];
    }
    int length();                           // 返回此顺序表的当前实际长度
    bool append(const T value);             // 在表尾添加一个元素value，表的长度增1
    bool insert(const int p, const T value); // 在位置p上插入一个元素value，表的长度增1
    bool delete(const int p);               // 删除位置p上的元素，表的长度减1
    bool setValue(const int p, const T value); // 用value修改位置p的元素值
    bool getValue(const int p, T& value);   // 把位置p的元素值返回到变量value中
    bool getPos(int & p, const T value);    // 查找值为value的元素，并返回第1次出现的位置
};
```

【代码2.2结束】

2.2.2 顺序表的运算实现

下面介绍基于数组的顺序表检索、插入和删除等运算。

1. 顺序表的检索

顺序表上的检索运算可以分为按位置的查找和按内容的查找两类。前者在顺序表中的实现非常简单,根据 2.2.1 节所示的计算公式可以直接计算其存储地址,可在常数时间内存取该元素。后者是指在顺序表中寻找满足某种条件,诸如值为 value 的元素的所在位置。该运算的基本算法是将 value 与顺序表中的元素依次进行比较,如果某次比较相等,则该元素即为所要查找的元素,并返回 true,否则继续比较,直到所有元素都与 value 进行过比较但不匹配,表明此时顺序表不存在元素值为 value 的元素,返回 false。

【算法 2.3】 在顺序表查找值为 value 的元素的位置。

```
// 在表中查找值为 value 的元素,
// 成功则返回 true,并将该元素所在的下标记录在参数 p 中,失败则返回 false
template <class T>                          // 假定顺序表的元素类型为 T
bool arrList<T> :: getPos (int &p,const T value) {
    int i;                                  // 元素下标
    for (i=0; i<n; i++)                     // 依次比较
        if (value == aList[i]) {            // 下标为 i 的元素与 value 相等
            p=i;                            // 将下标由参数 p 返回
            return true;
        }
    return false;                           // 顺序表没有元素值为 value 的元素
}
```

【算法 2.3 结束】

此算法的执行时间主要体现在元素的比较次数中,即 for 循环上。最好的情况是第 1 个元素的值即为 value,此时只需比较一次即可;最差的情况则是表中没有值为 value 的元素,这时需要比较 n 次;一般情况下,假设 value 出现在每个位置的概率相同,均为 $p=1/n$,则平均的比较时间为

$$\sum_{i=1}^{n} p \times i = \frac{1}{n}(1+2+\cdots+n) = \frac{n+1}{2}$$

即等概率情况下,检索算法需要寻找表中一半的元素,平均的时间开销为 $O(n)$。

2. 顺序表的插入

插入操作将改变顺序表的内容,因此必须满足一定的限制条件:除了涉及被更新的那个元素之外,其他线性关系的相对顺序应该保持不变。为此,需要对顺序表实施一系列的元素移动操作来维护逻辑的和存储的线性关系。

另外，由于顺序表是长度固定的线性结构，因此对其进行插入运算时还需要检查顺序表中实际元素的个数，以免因为插入而发生溢出现象（即超过所允许的最大长度 maxSize 的值）。算法 2.4 在顺序表的位置 p 上插入一个值为 value 的元素，图 2.2 所示为相应的示意图。

aList	k_0	k_1	…	k_p	…	…	k_{n-1}	…	…
	0	1	…	p	…	…	$n-1$	…	maxSize−1

aList	k_0	k_1	…	value	k_p	…	…	k_{n-1}	…	…
	0	1	…	p	$p+1$	…	…	n	…	maxSize−1

图 2.2 顺序表元素插入示意图

【算法 2.4】 在顺序表的某位置上插入元素。

```
// 设元素的类型为 T，aList 是存储顺序表的数组，maxSize 是其最大长度；
// p 为新元素 value 的插入位置，插入成功则返回 true，否则返回 false
template <class T>                          // 假定顺序表的元素类型为 T
bool arrList<T> :: insert (const int p,const T value) {
    int i;
    if (curLen >= maxSize) {                // 检查顺序表是否溢出
        cout << "The list is overflow" << endl;
        return false;
    }
    if (p<0 || p>curLen) {                  // 检查插入位置是否合法
        cout << "Insertion point is illegal" << endl;
        return false;
    }
    for (i=curLen; i>p; i--)
        aList[i] = aList[i-1];              // 从表尾 curLen -1 起向右移动直到 p
    aList[p] = value;                       // 位置 p 处插入新元素
    curLen++;                               // 表的实际长度增 1
    return true;
}
```

【算法 2.4 结束】

此算法的执行时间主要消耗在元素的移动操作上。最好的情况下，插入位置为当前线性表的最后，此时移动次数为 0；最差情况下插入位置为线性表的最前面，此时所有 n 个元素均需移动；一般情况下，插入位置为 i 时需要移动其后的 $n-i$ 个元素，假设在各个位置插入的概率相等，均为 $p=1/(n+1)$，则平均移动元素次数为

$$\sum_{i=0}^{n} p \times (n-i) = \frac{1}{n+1}\sum_{i=0}^{n}(n-i) = \frac{n}{2}$$

即等概率的情况下插入算法平均需要移动顺序表中的一半元素，总时间开销与表中元素个数成正比，为 $O(n)$。

3. 顺序表的删除

删除运算需要检查事先检查顺序表是否为空表，只在非空表时才能进行删除。算法 2.5 将删除顺序表中位置 p 上的元素，位置 p + 1 至 curLen − 1 的元素顺序左移，即图 2.3 中的元素 k_{p+1} 至元素 k_{n-1} 逐次顺序左移一位。

aList	k_0	k_1	k_2	…	k_p…	…	k_{n-1}	…	…
	0	1	2	…	p	…	$n-1$	…	maxSize−1

aList	k_0	k_1	k_2	…	k_{p+1}	…	k_{n-1}	…	…
	0	1	2	…	p	…	$n-2$	…	maxSize−1

图 2.3 顺序表元素删除示意图

【算法 2.5】 删除顺序表中给定位置上的元素。

```
// 设元素的类型为 T;aList 是存储顺序表的数组; p 为即将删除元素的位置
// 删除成功则返回 true,否则返回 false
template < class T >                          // 顺序表的元素类型为 T
bool arrList < T > :: delete( const int p) {
    int i;
    if ( curLen < = 0 ) {                     // 检查顺序表是否为空
        cout << " No element to delete \n" << endl;
        return false ;
    }
        if ( p <0 || p > curLen − 1 ) {       // 检查删除位置是否合法
        cout << "deletion is illegal\n" << endl;
        return false ;
    }
    for ( i = p; i < curLen − 1 ; i ++ )
        aList[ i ] = aList[ i + 1 ];          // 从位置 p 开始每个元素左移直到 curLen
    curLen − − ;                              // 表的实际长度减 1
    return true;
}
```

【算法 2.5 结束】

删除操作的时间代价分析与插入操作相似，其平均代价为 $O(n)$。

总之，顺序表按位置读取元素非常方便，其时间代价为 $O(1)$，但插入、删除以及按内容查找操作需要线性的时间代价 $O(n)$。

2.3 链 表

尽管顺序表是一种非常有用的数据结构，但其至少存在以下两方面的局限：

(1) 改变顺序表的大小需要重新创建一个新的顺序表并把原有的数据都复制过去。

(2) 因为逻辑关系是通过物理位置的相邻来表示的，增删元素平均需要移动一半的元素。为了克服顺序表无法改变长度的缺点，并满足许多应用经常插入新结点或删除结点的需要，产生了链表(linked list)这样的数据结构。

链表可以看成一组既存储数据又存储相互连接信息的结点集合。这样，各结点不必如顺序表那样存放在连续的存储空间中，可以散放在存储空间的各处，而由称为指针的域来按照线性表的后继关系链接结点。链表的特点是可以动态地申请内存空间，根据线性表元素的数目动态地改变其所需要的存储空间。在插入元素时申请新的存储空间，删除元素时释放其占有的存储空间。

链接存储是最常用的存储方法之一，它不仅可以用来表示线性表，也常用于其他非线性的数据结构。例如，随后几章要讨论的树结构和图结构，其中很多应用都使用结点的链接方式。本章主要讨论几种用于线性表的链接存储结构：单链表、双链表、循环链表等，统称为链表。本节在讨论链表的同时，还涉及了动态存储管理的基本概念和方法。注意，链接存储结构中所涉及的指针变量以及它们涉及的申请和释放计算机内存空间的方法。

2.3.1 单链表

在链表结构中，如果每个结点只包含指向其后继的指针，则称为单链表(singly linked list)。图 2.4 所示为一个单链表的存储示例。

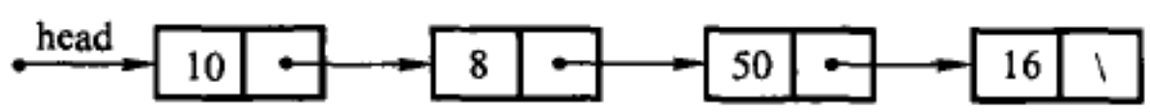


图 2.4 单链表示例

由图 2.4 可知，单链表是通过指针把它的一串内存结点链接成一个链，因此这些结点在内存空间中地址不必两两相邻。单链表的存储结点由两部分组成：一部分存放对用户而言非常重要的结点数据，称做 data 域，如图 2.4 中的 10、8、50 和 16；另一部分作为辅助成分，存放指向后继结点的指针域 next，如图 2.4 中的结点 10 中存放其后继结点 8 的地址。对于没有后继结点的终止结点而言，其 next 域值为空指针 NULL（在图中用“\”表示），如图 2.4 中结点 16 的 next 域。

【代码 2.6】 单链表的结点定义。

```
template <class T> class Link {
```

```
public:
    T   data;                                          // 用于保存结点元素的内容
    Link <T>    * next;                                // 指向后继结点的指针

    Link(const T info,const Link <T> * nextValue = NULL) {   // 具有两个参数的 Link 构造函数
        data = info;
        next = nextValue;
    }
    Link(const Link <T> *  nextValue) {                // 具有一个参数的 Link 构造函数
        next = nextValue;
    }
};
```

【代码 2.6 结束】

由于单链表中的结点是一个独立的对象，故将其定义为一个独立的类，以便于复用。代码 2.6 给出了单链表结点类型 Link 的定义。注意，Link 是由其自身来定义的，因为其中的 next 域指向正在定义的类型本身，这种类型称为自引用型。鉴于第 3 章的栈和队列也要用到 Link 类，故在此将其数据成员声明为公有的。

用一个指向表首的变量 head 存放指向单链表开始结点的指针。由于单链表中各个结点的存储地址并不连续，因此访问任何结点都只能从头指针开始沿着结点的 next 域来进行。例如，在图 2.4 中，head -> next -> data = 8，而 head -> next -> next -> data = 50。一个线性表的元素数目越多，则此链越长。为了加速对链表尾端的访问，特别是针对尾附函数 append( )这样在表尾直接添加元素的操作需求，往往会使用另一个表尾变量 tail 来存放指向单链表尾结点的指针，以方便操作，如图 2.5 所示。对于链表的访问只可通过头、尾指针来进行。增加了尾指针后，append( )即可在常数时间内完成。

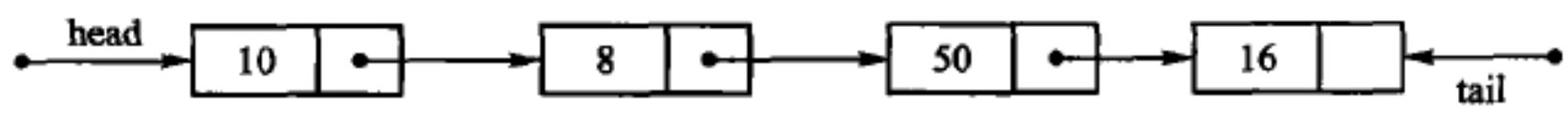


图 2.5　具有头、尾两指针的单链表

代码 2.7 给出了这种具有头和尾结点的单链表的抽象数据类型。根据链表的实际应用特点，在其中增加了一个返回链表长度的函数 length( )和一个返回指向给定结点指针的私有函数 setPos(const int p)。在某些定义了当前位置的应用中，可以使用函数 setPos( )把某个指定位置设置成当前位置。

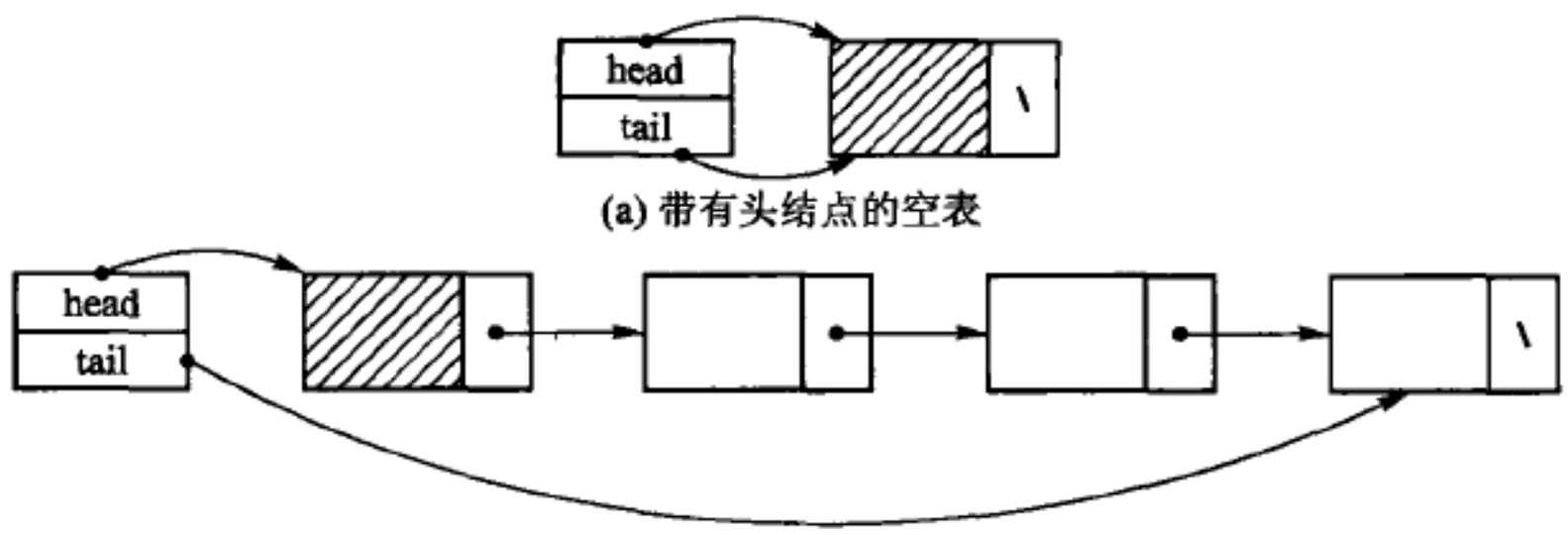
【代码 2.7】　单链表的类型定义。

```
template <class T> class lnkList : public List <T> {
private:
    Link <T>  * head, * tail;              // 单链表的头、尾指针
    Link <T>  * setPos(const int p);       // 返回线性表指向第 p 个元素的指针值
```

```
public:
    lnkList(int s);                          // 构造函数
    ~lnkList();                              // 析构函数
    bool isEmpty();                          // 判断链表是否为空
    void clear();                            // 将链表存储的内容清除,成为空表
    int length();                            // 返回此链表的当前实际长度
    bool append(cosnt T value);              // 在表尾添加一个元素 value,表的长度增 1
    bool insert(cosnt int p,cosnt T value);  // 在位置 p 上插入一个元素 value,表的长度增 1
    bool delete(cosnt int p);                // 删除位置 p 上的元素,表的长度减 1
    bool getValue(cosnt int p,T& value);     // 返回位置 p 的元素值
    bool getPos(int &p,const T value);       // 查找值为 value 的元素,并返回第 1 次出现的位置
}
```

【代码 2.7 结束】

为了操作便利,在实际组织单链表时,通常会引入一个称为头结点(header node)的特殊结点作为表头,以避免对空表的处理,从而避免对边界情况做特别考虑。作为表中的一个虚结点,这个头结点的值被忽略,不被看做表中的实际元素。图 2.6(a)所示为一个带有头结点的空表,图 2.6(b)则为一个典型的带有头结点的单链表。



(a) 带有头结点的空表

(b) 典型的带有头结点的单链表

图 2.6 引入头结点的单链表

引入头结点有利于对空链表进行操作或在链表头进行操作等特殊情况的处理,尤其是插入和删除操作对边界的处理。否则,在表为空时进行插入操作或者删除表中唯一的元素时,均需要特殊的边界处理。

算法 2.8 给出了这种带有头结点的单链表的构造函数和析构函数。为了与顺序表的接口保持一致,链表的构造函数引入了一个表示表的最小长度的参数。这个参数对链表是不必要的,因此在实现中被忽略掉。

【算法 2.8】 带有头结点的单链表构造函数与析构函数。

```
template <class T>
lnkList:: lnkList(int defSize) {
```

```
        head = tail = new Link < T > ;          // 此处采用 class Link 的构造函数来初始化 head 和 tail
}
template  < class T >
lnkList : :  ~ lnkList( )  {
        Link < T >  * tmp;
        while ( head ! =  NULL)  {
            tmp = head;
            head = head -> next;
            delete tmp;
        }
}
```

【算法 2.8 结束】

下面仍以检索、插入和删除这 3 个最常用的运算为例来介绍单链表的实现方法。

1. 链表的检索

由于地址空间的不连续，单链表无法像顺序表那样直接通过结点位置来定位其地址，而是需要从头指针 head 所指的首结点开始沿 next 域，逐个结点地进行访问。

根据链表的特点，只要有指向一个结点的指针，便可通过该指针访问此结点。换言之，按位置检索只要返回指向该位置的指针即可。算法 2.9 给出了在链表中查找第 i 个结点的代码，返回值为指向找到的结点的指针。其中，链表中第 i 个结点是按照 C/C ++ 的数组下标编号规则，从 0 到 n - 1，头结点的编号为 - 1。当单链表实际长度小于给定的 i 时，返回 NULL，当 i 为 - 1 时返回指向头结点的指针。

【算法 2.9】 寻找链表的第 i 个结点。

```
Tem plate  < class T >                                  // 线性表的元素类型为 T
Link < T > *  lnkList  < T > : : setPos( int i)  {
        int count = 0;
        if ( i  ==  - 1)                                // i 为 - 1 则定位到头结点
            return head;

        Link < T >  * p = new Link < T > ( head -> next) ;  // 循链定位，若 i 为 0 则定位到第一个结点
        while ( p ! =  NULL && count < i)  {
            p = p ->  next;
            count + + ;
        } ;
        return p;      // 指向第 i 结点，i = 0，1，…，当链表中结点数小于 i 时返回 NULL
}
```

【算法 2.9 结束】

由算法 2.9 可知，在链表上基于位置检索需要从链表头开始循链移动，直到找到第 i 个位

置，所以平均需要 $O(n)$ 的时间。

按内容查找的方法与此类似，也是循链比较，只是比较的对象为数据域而已。因此，无论是按内容还是按位置来检索链表，都只能采取顺序访问的方式，无法进行随机存取。

2. 链表的插入和删除

由于单链表结点之间的前驱后继关系由指针来表示，因此在插入或删除结点时，维护结点之间的逻辑关系只需要改变相关结点的 next 域，而不必像顺序表那样移动一系列存储单元的内容。

在单链表中插入和删除结点还牵涉到存储管理的问题。在单链表中插入一个新结点时，可以使用 new 命令为新结点开辟存储空间。与 new 对应，可使用 delete 命令释放从单链表中删除的结点所占用的空间，否则这些被占用的空间会变成存储空间中无法利用的垃圾，影响再利用。new 和 delete 是 C++程序语言为动态存储管理提供的两个重要命令，C 语言标准函数库中提供的 malloc( )和 free( )函数具有同样的功能。

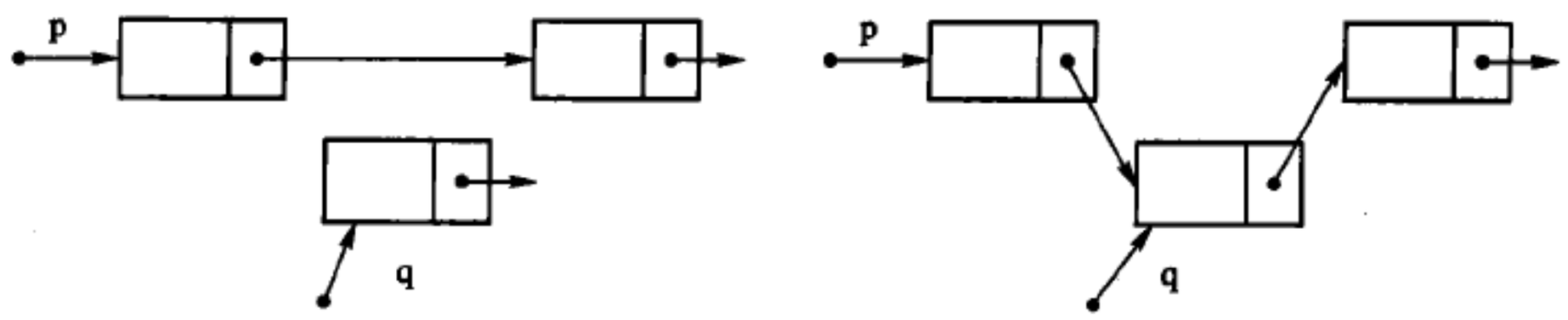
向链表中插入一个新元素的操作，具体包括创建一个新结点（并赋值）和修改相关结点的链接信息以维护原有的前驱后继关系。相关结点涉及新插入结点的前驱结点以及新结点自身，需要把新结点的指针域设置为其前驱的指针值，而前驱结点的指针应该指向新结点。由于单链表没有指向前驱的指针，因此在第 i 个位置插入结点时，必须先获得位置 i－1 的指针，以便保持插入后链接正确。对于 n 个结点的线性表，插入点可以有 n＋1 个：i＝0 表示插入在表头，setPos(i－1)将返回头结点 head，新结点直接链接到 head 后，成为表中第一个结点；i＝n 表示在表尾插入，与 append( ) 功能相同。算法 2.10 给出了单链表插入运算的一种实现方法，相应的插入过程如图 2.7 所示。

【算法 2.10】 插入单链表的第 i 个结点。

```
template <class T>                                  // 线性表的元素类型为 T
bool lnkList <T> :: insert(const int i,const T value) {
        Link <T> *p, *q;

        if ((p = setPos(i -1)) == NULL) {    // p 是第 i 个结点的前驱
                cout << " 非法插入点" <<endl;
                return false;
        }
        q = new Link <T> (value,p -> next);
        p -> next = q;
        if (p == tail)                              // 插入点在链尾,插入结点成为新的链尾
                tail = q;
        return true;
}
```

【算法 2.10 结束】



(a) 准备新结点，定位待插入点　　(b) 成功插入后指针链接情况

图 2.7　插入算法示意图

与插入操作相同，从单链表中删除一个结点也需要修改被删结点前驱的指针域来维护结点间的线性关系，同时要释放被删结点所占用的内存，以免“丢失”。算法 2.11 给出了删除单链表第 i 个结点的代码，相应的操作过程如图 2.8 所示。

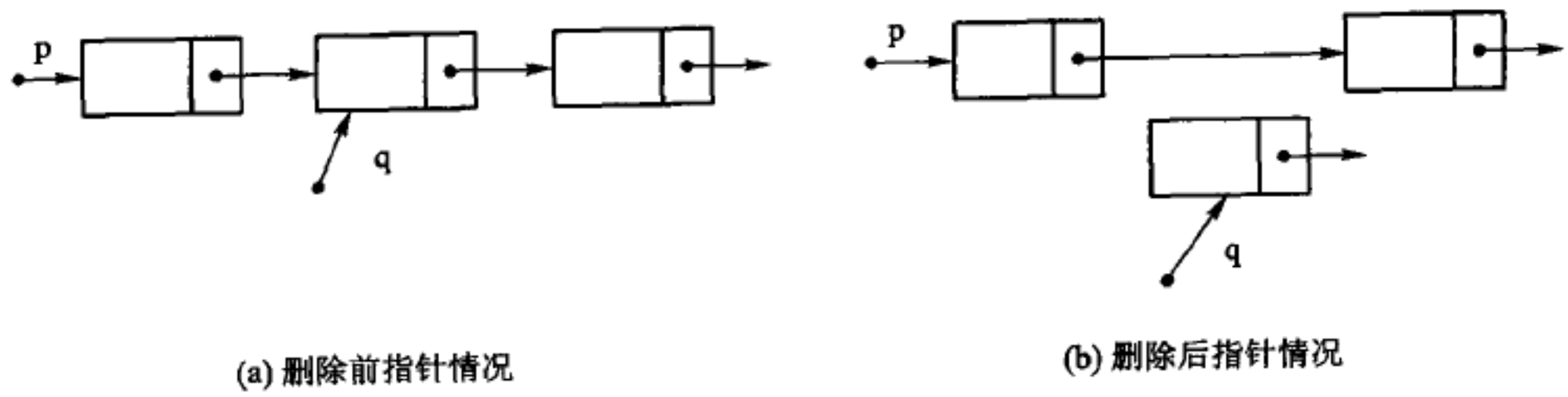
【算法 2.11】　单链表的删除算法。

```
template <class T>                                  // 线性表的元素类型为 T
bool lnkList <T> :: delete(const int i) {
    Link <T>  *p, *q;
    if ((p = setPos(i - 1)) == NULL || p == tail) {  // 待删结点不存在，即给定的 i 大于当前链中
                                                     // 元素个数
        cout << " 非法删除点 " << endl;
        return false;
    }
    q = p -> next;                                   // q 是真正待删结点
    if (q == tail) {                                 // 待删结点为尾结点，则修改尾指针
        tail = p;
        p -> next = NULL;
        delete q;
    }
    else if (q != NULL) {                            // 删除结点 q 并修改链指针
        p -> next = q -> next;
        delete q;
    }
    return true;
}
```

【算法 2.11 结束】

从算法 2.10 和算法 2.11 可以看到，尽管插入（删除）操作本身可在常数时间内完成结点的创建（释放）和链接信息的修改，但在位置 i 进行插入删除操作时，需要先定位到位置 i - 1 的结点。而定位操作的平均时间代价为 $O(n)$，这也是在某些应用中需要在抽象数据类型中增加一个表示当前位置的成员的原因，因为通常插入和删除均在当前位置上进行。对于前面定义线性

表抽象数据类型时所讨论的位置运算问题，具体到用单链表方式实现时，通常保持当前位置的前一个指针以方便操作。



(a) 删除前指针情况　　(b) 删除后指针情况

图 2.8　删除示意图

2.3.2　双链表

单链表的主要不足之处在于其指针域仅指向其后继结点，因此从一个结点不能有效地找到其前驱，而必须从表首开始顺着 next 域逐一查找。这对于长链表而言，代价相当可观。为此，引入双链表(double linked list)结构，其基本思路是在每个结点中再增加一个指向前驱的指针，形成如图 2.9 所示的双链表，其中带有斜线阴影的结点表示头结点。

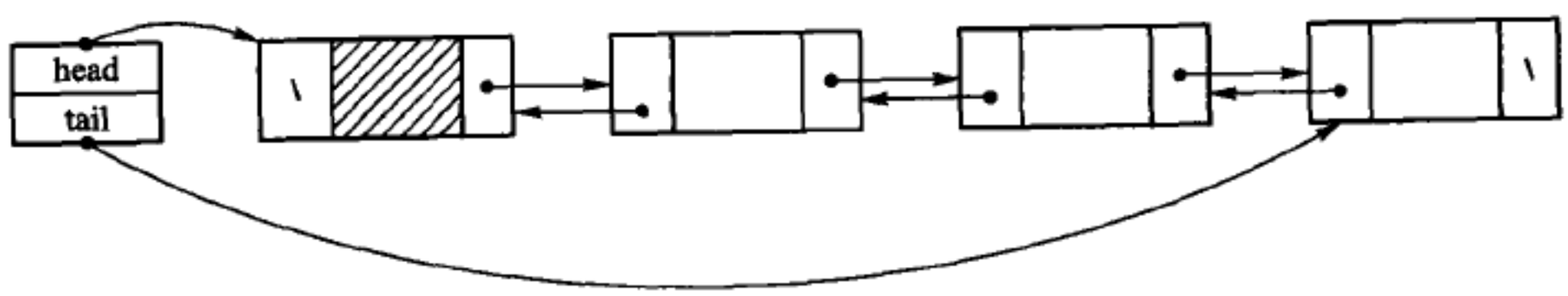


图 2.9　加入表头结点的双链表

图 2.9 中变量 head 用于指向表首结点，变量 tail 用于指向表尾结点。图 2.10 给出了相应的结点图示，其中 next 表示指向后继的指针，prev 表示指向前驱的指针。线性表的链式实现究竟采用单链表还是双链表对 List 类的用户而言应该是透明的。下面代码 2.12 给出了使用双链表的一个 Link 类的实现。

图 2.10　双链表的结点

【代码 2.12】　双链表的结点定义和实现。

```
template <class T> class Link {
public:
    T  data;                          // 用于保存结点元素的内容
    Link<T> *next;                    // 指向后继结点的指针
    Link<T> *prev;                    // 指向前驱结点的指针
```

```
    Link(const T info,Link<T> * preValue = NULL,Link<T> * nextValue = NULL) {
                                                        // 构造函数:值和前后指针
        data = info;
        next = nextValue;
        prev = preValue;
    }
    Link(Link<T> * preValue = NULL,Link<T> * nextValue = NULL){ // 给定前后指针的构造函数
        next = nextValue;
        prev = preValue;
    }
}
```

【代码 2.12 结束】

与单链表上的对应操作相比,双链表要稍微复杂些,因为双链表需要维护两个链。下面以插入和删除两个操作为例进行讨论。

与单链表不同,若要删除双链表中的一个结点,则不仅要修改该结点前驱的 next 域,还要修改该结点后继的 prev 域。例如,如果要删除指针变量 p 所指的结点,需要通过下述的操作来维护前驱和后继两条链。

```
p -> prev -> next = p -> next;
p -> next -> prev = p -> prev;
```

然后把变量 p 的前驱和后继置空,再释放 p 所指空间即可。

```
p -> next = NULL;
p -> prev = NULL;
delete p;
```

删除操作的示意图如图 2.11 所示。

同样,双链表中要在 p 所指结点后插入一个新结点 q,具体操作步骤如下:

(1) 执行 new q 开辟结点空间。

(2) 填写新结点的数据域信息。

(3) 填写新结点在链表中的链接关系,即

```
q -> prev = p;
q -> next = p -> next;
```

(4) 修改 p 所指结点及其后继结点在新结点插入后的链接信息,即把新结点的地址填入原 p 所指结点的 next 域以及新结点后继的 prev 域回指新结点本身,即

```
p -> next = q;
q -> next -> prev = q;
```

插入过程的示意图如图 2.12 所示。

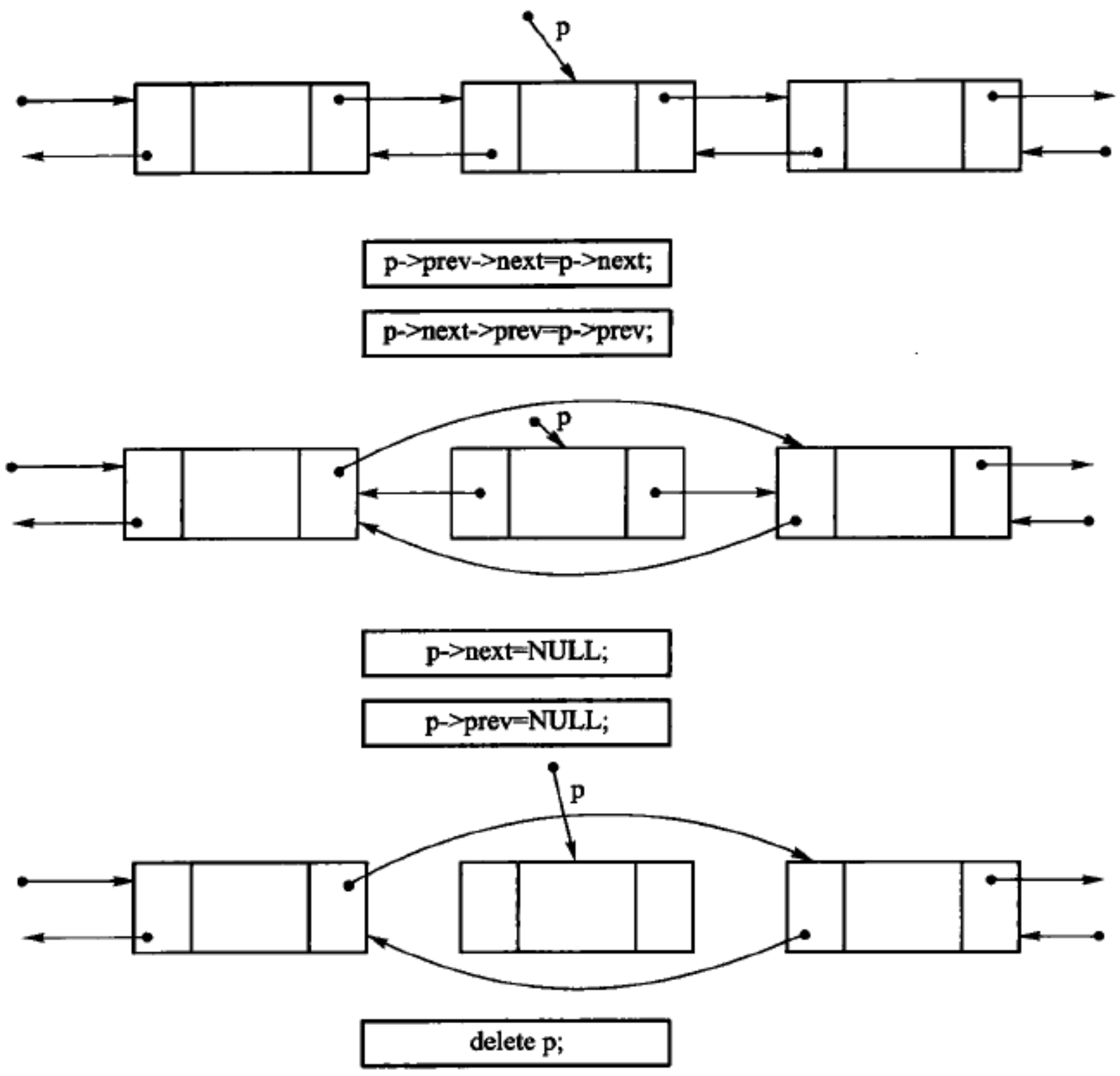


图 2.11 双链表的删除操作示意

尽管双链表的空间开销比单链表稍多,但可在 $O(1)$ 时间内找到给定元素的前驱。

2.3.3 循环链表

某些情况需要把结点组成环形链表。例如,多个进程在一段时间内访问同样的资源,为了保证每一个进程可以公平地分享这个资源,可以把这些进程组织在如图 2.13 所示的称为循环链表(circularly linked list)的结构中。可以通过指针 current 访问该结构,current 指向的结点即为将要激活的进程。随着 current 指针的移动,可以激活每一个进程。

只要对原有的单链表形式稍做改变,即把最后一个结点的指针设置成指向表首的结点,就可以形成一个如图 2.14 所示的循环链表。不增加额外的存储花销,就能给很多操作带来方便,因为从循环表中任一结点出发都能访问到表中其他任意结点。

实现时,循环链表往往只需要指向表尾的指针 tail,而无须指向头结点的 head 指针,因为 tail 的后继即为 head 所指结点。这样,头尾结点都可在常数时间内找到。类似地,将双链表的首尾

结点链接起来就可以得到如图 2.15 所示的循环双链表。

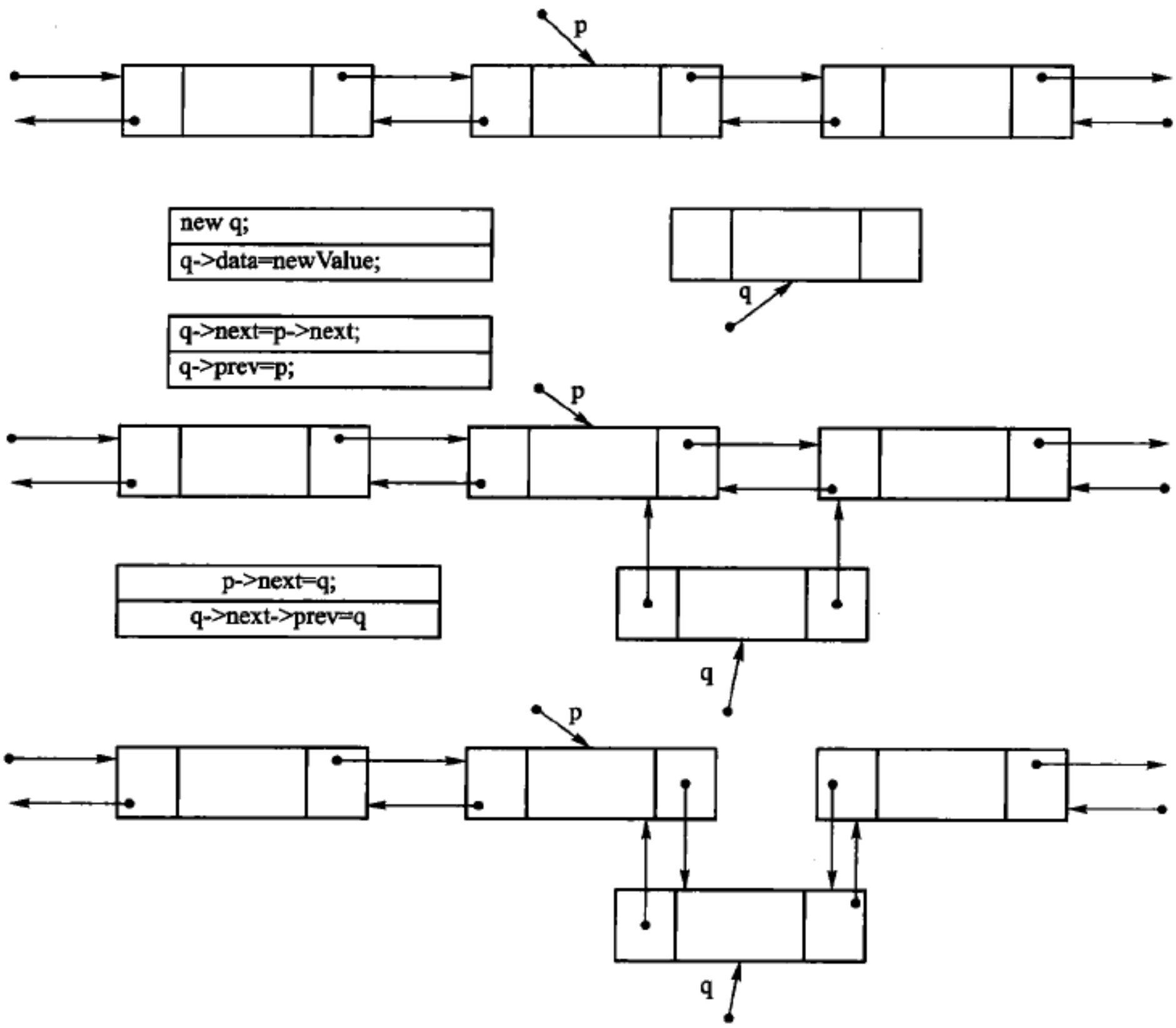


图 2.12　双链表插入操作示意

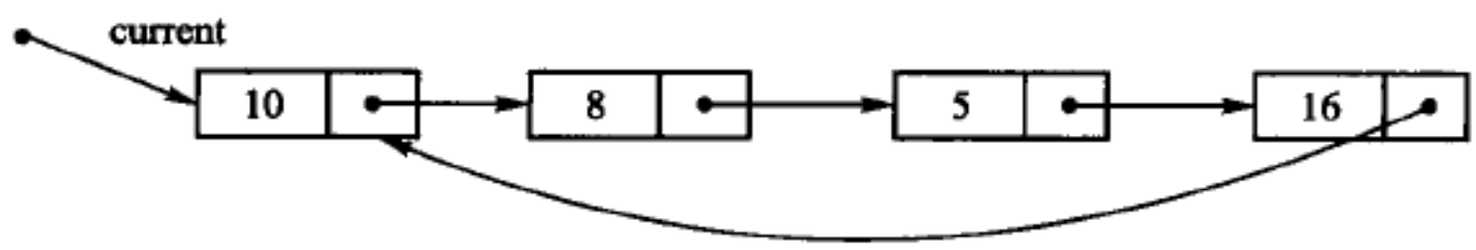


图 2.13　循环链表示意图

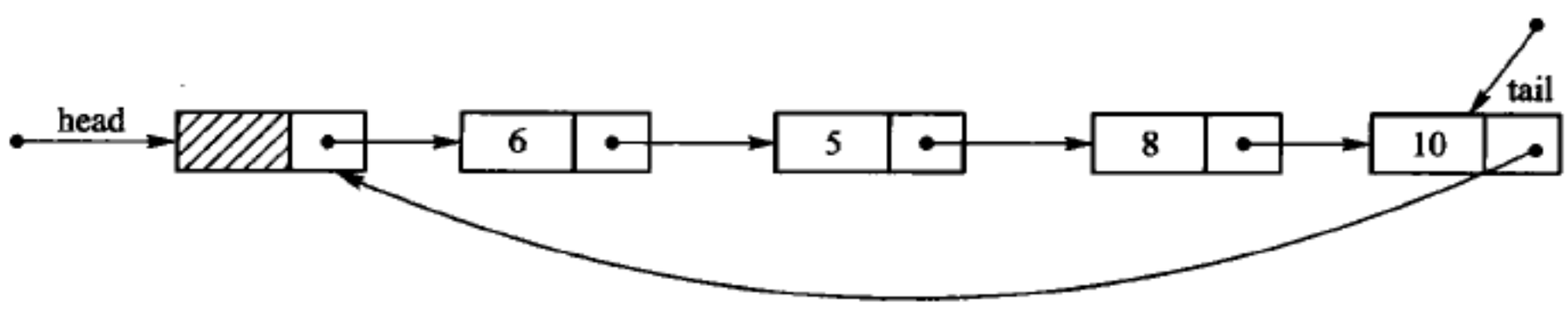


图 2.14　带头结点的循环链表示意图

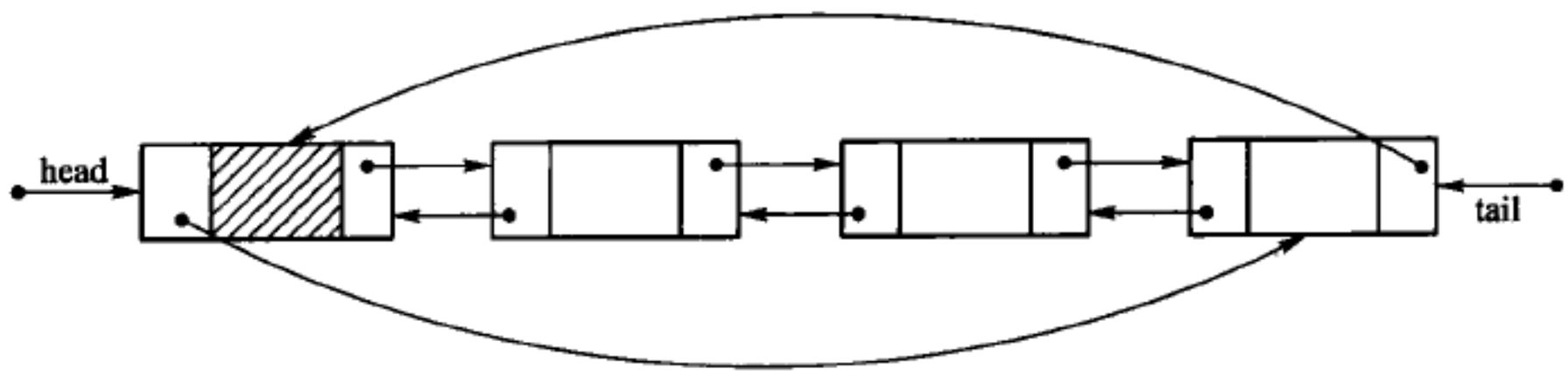


图 2.15 循环双链表示意图

2.4 线性表实现方法的比较

顺序表是最简单的数据组织方法，因此大多数程序设计语言都提供了数组这种机制来实现顺序表。除具有易用、空间开销较小等特点之外，顺序表还提供了对任意元素进行随机访问的能力，适合于二分搜索及快速排序等运算，是存储静态数据的理想选择。

除了适用于频繁插入或删除内部元素的线性表之外，链表还适合于管理那些长度经常变化或事先无法确定长度的线性表。

作为基本的数据结构，线性表有着广泛的应用。例如，存储管理本质上就是利用线性表管理可利用空间。而且，线性表还可以作为基本成分构建复杂的数据结构，例如散列方法就是把顺序表和链表结合起来的一种数据结构。在实际应用中，具体采用何种实现方法，取决于应用线性表数据元素的统计特征和操作特点。下面列出几个在取舍时值得注意的因素。

1. 不要使用顺序表的场合

线性表中经常要插入/删除内部元素时，不宜使用顺序表，因为顺序表的插入/删除操作平均情况下需要移动表中一半的元素。此外，无法确定线性表长度的最大值时，也不宜采用顺序表。

2. 不要使用链表的场合

经常对线性表进行按位置的访问，而且按位读操作比插入/删除操作频繁时，不宜使用链表，因为顺链表扫描浏览比按顺序表下标读元素要费时。此外，指针本身的存储开销也需要考虑，如果与结点内容所占空间相比，指针所占的比例较大时（超过 1:1），应该慎重选择。

本章小结

本章在介绍线性表基本概念和抽象数据类型的基础上，描述了线性表的两种实现方法。

线性结构是最简单且最常用的一种数据结构，其基本特点是结构中的元素之间满足线性关系，按这个关系可以把所有元素排成一个线性序列。线性表是由若干数据元素组成的有限序列，是一种比较简单的数据结构。线性表通常有顺序和链式两种存储方式，在两种存储结构上线性

表各运算的实现效率各有千秋。

顺序表是组织数据的最简单方法，具有易用、空间开销较小以及对元素随机访问等特点，是存储静态数据的理想选择。而链表不仅可以适用于那些频繁增删结点的应用，还适用于处理事先无法确定长度的线性表。

习 题

1. 设顺序表 a 中的数据元素递增有序，试设计一个算法，将 x 插入到顺序表的适当位置，以保持该表的有序性。

2. 设 $A=(a_1,\cdots,a_m)$ 和 $B=(b_1,\cdots,b_n)$ 均为顺序表，A' 和 B' 分别为 A 和 B 中除去最大共同前缀后的子表。若 $A'=B'=$ 空表，则 $A=B$；若 $A'=$ 空表，而 $B'\neq$ 空表，或者两者均不为空表，且 A' 的表首元素小于 B' 的表首元素，则 $A<B$；否则 $A>B$。试写一个比较 A、B 大小的算法。

3. 设线性表中的数据元素以值递增排列，并以单链表作为存储结构。设计一个高效的算法，删除表中所有值大于 min 且小于 max 的元素，同时释放被删除结点的空间，并分析算法的时间复杂度。

4. 假设有两个按元素值递增有序排列的线性表 A 和 B，均以单链表作为其存储结构，试设计一个算法将 A 和 B 归并成一个按元素值递减有序排列的线性表 C，并要求利用原表（即表 A 和表 B）的结点空间构造表 C。

5. 已知由一个链表表示的线性表中含有三类字符的数据元素（字母字符、数字字符和其他字符），试设计一个算法将该线性链表分割为 3 个循环链表，其中每个循环链表中均只含一类字符。

6. 试设计一个算法，删除一个顺序表中从第 i 个元素开始的 k 个元素。

7. 已知线性表中元素以值递增有序排列，并以单链表作为存储结构。试设计一个算法，删除表中值相同的多余元素，使得操作后表中的所有元素值均不相同，同时释放被删除的结点空间。

8. 已知两个元素按值递增有序排列的线性表 A 和 B，且同一表中的元素值各不相同。试构造一个线性表 C，其元素为 A 和 B 中元素的交集，且表 C 中的元素也按值递增有序排列。设计对 A、B、C 都是顺序表时的算法。

9. 要求同第 8 题。设计当 A、B、C 都是单链表时的算法。

10. 设表达式以字符形式已存入数组 $E[n]$ 中，“#”为表达式的结束符，试设计判断表达式中括号“(”和“)”是否配对的算法。

上 机 题

1. 试设计一个非递归算法在 $O(n)$ 时间内将一个含有 n 个元素的单链表逆置，要求其辅助空间为常量。

2. 给定一个单向链表，试设计一个既节省时间又节省空间的算法来找出该链表的倒数第 m 个元素。实现这个算法，并对可能出现的特殊情况做相应处理。自行设计链表的数据结构。（“倒数第 m 个元素”的含义：当 $m=0$ 时，链表的最后一个元素将被返回）

3. 设有 n 个人围坐在一个圆桌周围，现从第 s 个人开始报数，数到第 m 的人出列，然后从出列的下一个人重新开始报数，数到第 m 的人又出列，如此反复直到所有的人全部出列为止。Josephus 问题是：对于任意给定的 n、s、m，求出按出列次序得到的 n 个人员的序列。试在计算机上模拟 Josephus 问题的求解过程。

第3章 栈与队列

栈(stack)和队列(queue)作为两种重要的线性结构,在计算机科学中具有非常广泛的应用,从简单的表达式计算到编译器对程序语法的检查,再到操作系统对各种设备的管理等都会涉及。

从逻辑结构来说,栈和队列都是典型的线性结构。与线性表不同的是,栈和队列上的操作比较特殊,受到一定的限制,仅允许在线性表的一端或两端进行。因此,栈和队列常被称为操作受限的线性表,或者限制存取点的线性表。

栈和队列在日常生活中均有许多实例,例如叠放在一起的一摞盘子可看成栈的一个实例,而在银行排队取款的一行人则构成一个典型的队列。一般而言,栈的特点是后进先出,常用来处理具有递归结构的数据;而队列的特点则是先进先出,在实际中体现出公平的原则,可以用来暂时存放需要按照一定次序依次处理但尚未处理的元素。

本章将对栈和队列这两种结构及其典型应用进行介绍。

3.1 栈

栈是一种限定仅在一端进行插入和删除的线性表,无论是往栈中插入元素还是删除栈中的元素,或者读取栈中的元素,都只能固定在线性表的一端进行。通常,栈的这一端被称为栈顶(top),与此相对,栈的另一端叫做栈底(bottom)。由此可知,最后插入栈中的元素是最先被删除或读取的元素,而最先压入的元素则被放在栈的底部,要到最后才能取出。换言之,栈的修改是按后进先出的原则进行。因此,通常栈被称为后进先出(last in first out)表,简称 LIFO 表,如图 3.1 所示。例如,碗橱里的一叠盘子、大型火车站用于调整调度火车头方向的调度栈结构等都可视为栈的模型。

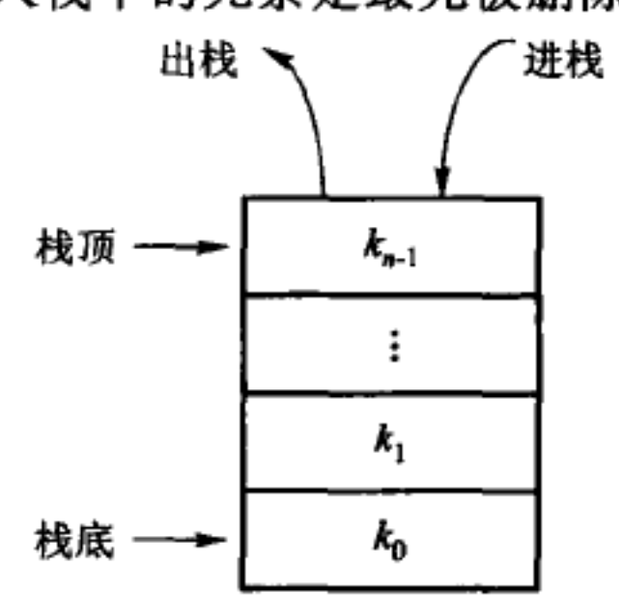


图 3.1 栈的示意图

尽管操作受限降低了栈的灵活性,但也正因为如此而使得栈更有效且更容易实现。栈的应用非常广泛,并因此形成了栈的一些特殊术语。习惯上称往栈中插入元素为 push 操作,简称为压栈或入栈;删除栈顶元素被称为 pop 操作,简称为出栈或弹出。

如同线性表可以为空表一样,没有元素的栈称为空栈。例如,刚建立的栈一般是空栈,随着栈中所有元素的删除,栈也会变成空栈。

3.1.1 栈的抽象数据类型

基于栈的特性，定义在栈的抽象数据类型中的运算包括进栈 push、出栈 pop、读栈顶 top 等常用操作以及判断栈是否为空的 isEmpty 和栈是否已满的 isFull 等边界判断操作。根据抽象和封装的原则，只可通过其抽象数据类型定义的运算来对栈进行操作。

下面的 C++ 类模板给出了栈类（名字为 Stack，模板参数为元素类型 T）的一个抽象数据类型定义。

【代码 3.1】 栈的抽象数据类型定义。

```
template <class T>                      // 栈的元素类型为 T
class Stack {
public:                                 // 栈的运算集
        void clear();                   // 变为空栈
        bool push(const T item);        // item 入栈，成功则返回真，否则返回假
        bool pop(T & item);             // 返回栈顶内容并弹出，成功返回真，否则返回假
        bool top(T & item);             // 返回栈顶内容但不弹出，成功返回真，否则返回假
        bool isEmpty();                 // 若栈已空返回真
        bool isFull();                  // 若栈已满返回真
};
```

【代码 3.1 结束】

上述定义的这个抽象数据类型并不是唯一的。针对具体应用的不同要求，操作函数可以适当增删。例如，栈的链表实现中并不需要判断栈是否为满，某些实现中也许会把读取栈顶元素的 top 操作和出栈操作 pop 合二为一。

栈的实现与其存储结构相关，下面介绍顺序栈和链式栈两种存储结构及相应的栈操作。

3.1.2 顺序栈

采用顺序存储结构的栈称为顺序栈（array - based stack），需要一块连续的区域来存储栈中的元素，因此需要事先知道或估算栈的大小。

顺序栈本质上是简化的顺序表。对元素数目为 n 的栈，首先需要确定数组的哪一端表示栈顶。如果把数组的第 0 个位置作为栈顶，按照栈的定义，所有的插入和删除操作都在第 0 个位置上进行，即意味着每次的 push 或 pop 操作都需要把当前栈的所有元素在数组中后移或前移一个位置，时间代价为 $O(n)$。反之，如果把最后一个元素的位置 $n-1$ 作为栈顶，那么只需将新元素添加在表尾，出栈操作也只需删除表尾元素，每次操作的时间代价仅为 $O(1)$。图 3.2 所示为按后一种方案实现的栈，其中 t 表示栈顶。

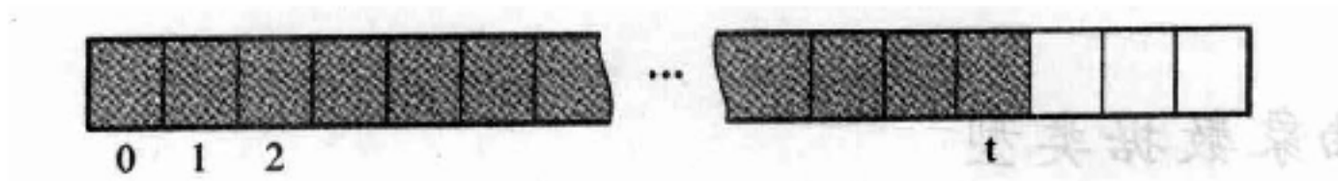


图 3.2 栈的顺序存储结构示意

顺序栈实现时，用一个整型变量 top（通常称为栈顶指针）来指示当前栈顶位置，同时也可以表示当前栈中元素的个数。代码 3.2 给出了一个顺序栈类及其部分成员函数的实现方法。

【代码 3.2】 栈的顺序实现。

```
template <class T>
class arrStack: public Stack <T> {
private:                                    // 栈的顺序存储
        int     mSize;                      // 栈中最多可存放的元素个数
        int     top;                        // 栈顶位置，应小于 mSize
        T      *st;                         // 存放栈元素的数组
public:                                     // 栈的运算的顺序实现
        arrStack(int size) {                // 创建一个给定长度的顺序栈实例
                mSize = size;
                top = -1;
                st = new T[mSize];
        }
        arrStack() {                        // 创建一个顺序栈的实例
                top = -1;
        }
        ~arrStack() {                       // 析构函数
                delete [] st;
        }
        void clear() {                      // 清空栈内容
                top = -1;
        }
        bool push(const T item) {           // 入栈操作的顺序实现
                if (top == mSize-1) {       // 栈已满
                        cout << "栈满溢出" << endl;
                        return false;
                }
                else {                      // 新元素入栈并修改栈顶指针
                        st[++top] = item;
                        return true;
                }
        }
```

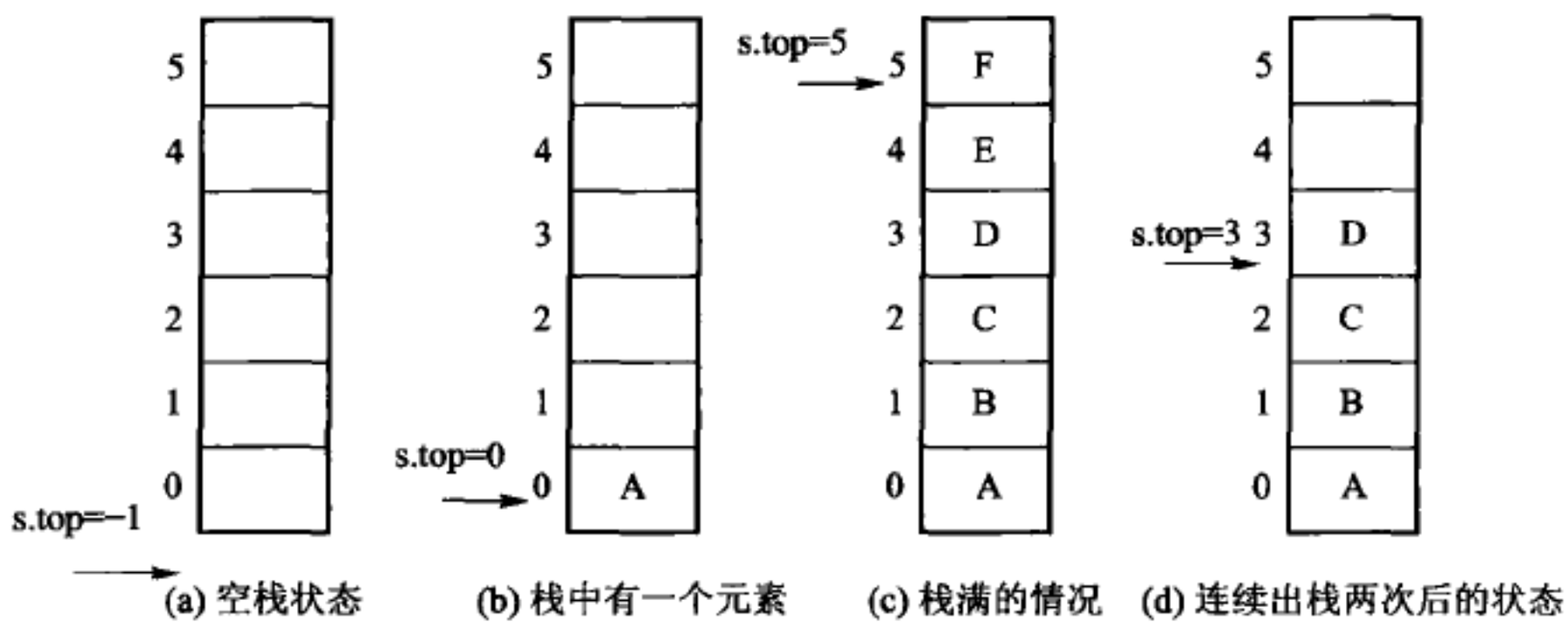
```
    bool pop(T & item) {                    // 出栈的顺序实现
        if (top == -1) {                    // 栈为空
            cout << "栈为空,不能执行出栈操作" << endl;
            return false;
        }
        else {
            item = st[top --];              // 返回栈顶元素并修改栈顶指针
            return true;
        }
    }
    bool top(T & item) {                    // 返回栈顶内容,但不弹出
        if (top == -1) {                    // 栈空
            cout << " 栈为空,不能读取栈顶元素" << endl;
            return false;
        }
        else {
            item = st[top];
            return true;
        }
    }
}
```

【代码 3.2 结束】

top 可以有两种定义方式。一种是将其设置为栈中第一个空闲位置,即空栈的 top 为 0;另一种则把 top 定义为栈中最上面的那个元素的位置,而非第一个空闲位置,此时空栈的 top 应该初始化为 -1 或任何非自然数。在本书中采取后者来表示栈顶。因为前者会浪费数组的一个位置。这样,push 和 pop 操作只在 top 所指示的数组位置插入或删除一个元素。因为 top 表示栈中最上面元素的位置,所以 push 一个元素时首先把 top 加 1,然后把新元素插入到新的栈顶位置。同样,pop 先删除栈顶元素,再把 top 减 1。

图 3.3 所示为 mSize 取值为 6 的顺序栈 s 中数据元素和栈顶指针的变化。其中,图 3.3(a) 表示空栈状态,此时 s.top = -1;图 3.3(b) 表示栈中具有一个元素的情况,此时栈顶 top 所指为栈顶元素所在的位置 0;图 3.3(c) 则表示在进行若干次进栈操作之后栈满的情况,此时 top = mSize -1; 图 3.3(d) 则是在图 3.3(c) 基础上连续出栈两次后的状态。

栈中元素是动态变化的,当栈中已有 mSize 个元素时,进栈操作会产生上溢出(overflow)。相应的,在空栈上进行出栈操作则会造成下溢出(underflow)。为了避免溢出,在对栈进行 push 和 pop 操作之前需要检查栈是否已满或是否已空。



(a) 空栈状态 (b) 栈中有一个元素 (c) 栈满的情况 (d) 连续出栈两次后的状态

图 3.3 顺序栈的存储结构

【算法 3.3】 改进的进栈操作。

```
template <class T>
bool arrStack<T> :: push(const T item) {
    if (top == mSize -1) {
        T *newSt = new T [mSize * 2];
        for (i = 0; i <= top; i++)
            newSt[i] = st[i];
        delete [] st;                    // 释放原栈
        st = newSt;
        mSize *= 2;
    }
    st[++top] = item;
    return true;
}
```

【算法 3.3 结束】

如果出现上溢时仍希望对顺序栈执行进栈操作,可以考虑对当前的顺序栈进行适当的扩容。例如,像算法 3.3 那样,申请一个扩大一倍的新数组,把顺序栈的原有内容顺序移动到新的数组中,再按正常的方式来执行进栈操作。

3.1.3 链式栈

链式栈(linked stack)本质上是简化的链表。需要注意的是为了方便存取,栈顶元素应该设置为链表头。图 3.4 所示为链式栈的一个简单示意图。

代码 3.4 是链式栈的一个简单实现。其中,数据成员 top 是一个指向链式栈的首结点(栈顶)的指针,链表的结点类型采用第 2 章中已定义过的 Link 类模板。进栈操作 push 在链表头插

入元素，出栈操作 pop 删除链头元素并释放空间。显而易见，push 和 pop 的时间代价均为 $O(1)$。

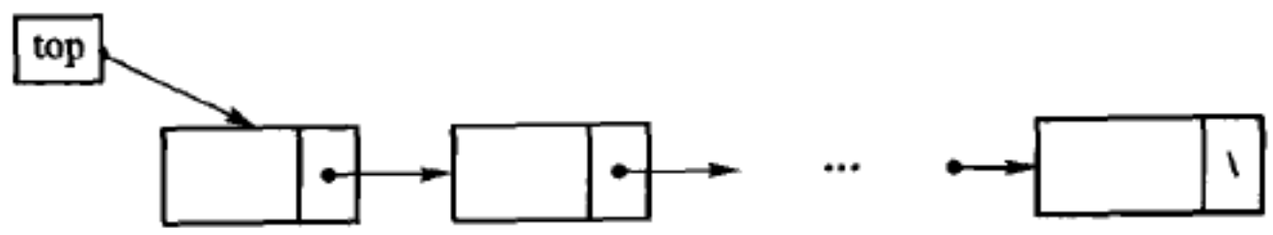


图 3.4 链式栈示意

【代码 3.4】 栈的链式实现。

```
template <class T>
class lnkStack : public Stack <T> {
private:                                    // 栈的链式存储
    Link <T> *   top;                       // 指向栈顶的指针
    int  size;                              // 存放元素的个数
public:                                     // 栈运算的链式实现
    lnkStack(int defSize) {                 // 构造函数
        top = NULL;
        size = 0;
    }
    ~lnkStack() {                           // 析构函数
        clear();
    }
    void clear() {                          // 清空栈内容
        while (top != NULL) {
            Link <T> * tmp = top;
            top = top -> next;
            delete tmp;
        }
        size = 0;
    }
    bool push(const T item) {               // 入栈操作的链式实现
        Link <T> * tmp = new Link <T> (item, top);
        top = tmp;
        size ++;
        return true;
    }
    bool pop(T & item) {                    // 出栈的链式实现
        Link <T> * tmp;
        if (size == 0) {
```

```
            cout << "栈为空,不能执行出栈操作" << endl;
            return false;
        }
        item = top -> data;
        tmp = top -> next;
        delete top;
        top = tmp;
        size -- ;
        return true;
    }
    bool top(T & item) {                    // 返回栈顶内容,但不弹出
        if (size == 0) {
            cout << "栈为空,不能读取栈顶元素" << endl;
            return false;
        }
        item = top -> data;
        return true;
    }
}
```

【代码3.4结束】

3.1.4 表达式求值

栈的应用非常广泛,只要满足后进先出的特性都可以使用栈结构。例如,记录网页访问历史、保存文本编辑器中的undo序列、编译栈中函数调用地址和参数的保存、二叉树的深度优先周游等。下面通过一些简单的实例来具体说明如何使用栈解决实际问题。

表达式求值是程序设计语言编译器中的一个最基本的问题。表达式是由常量、变量、运算符、函数调用等按一定规则组合而成的。为了讨论方便,在不失一般性的前提下,下面以简化的整型四则运算表达式为例来说明栈在表达式计算中的作用。

1. 表达式的定义

(1) 基本符号集:由数字0~9、加减乘除符号以及括号这16个基本符号组成,各符号间用逗号“,”隔开:

{0,1,…,9,+,-,*,/,(,)}

为了跟C++的表达式一致,这里用“*”代替数学表达式的“×”,用“/”代替数学表达式的“÷”。

(2) 语法成分集:共有以下5个语法成分:

{<表达式>,<项>,<因子>,<常数>,<数字>}

(3) 语法公式集:语法公式又称为产生式规则,用于定义语法成分。为了简便,在不影响通用性的前提下,下面给出的表达式的语法公式进行了简化处理:

<表达式> ::= <项> + <项> | <项> - <项> | <项>
<项> ::= <因子> * <因子> | <因子> / <因子> | <因子>
<因子> ::= <常数> | (<表达式>)
<常数> ::= <数字> | <数字><常数>
<数字> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

在语法公式中,符号"::="是规则定义符,其左部是一个需要定义的语法成分,右部则是定义该语法成分的语法规则,包含若干基本符号或若干语法成分作为其组成成分。其中,连接符"|"用于分隔其他语法成分,并不作为语法成分,其含义是"或者"的意思。例如,语法公式"<数字> ::=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9"表示数字是在符号0~9这10个数字中任选其一。

公式右部常由若干语法成分顺序拼接而成。例如,语法公式

<项> ::= <因子> * <因子> | <因子> / <因子> | <因子>,

说明项是或由两个因子中间插上一个运算符"*"或"/"所组成,或者是一个单独因子。

语法公式可以递归定义。例如,公式

<常数> ::= <数字> |<数字> <常数>

表示常数是由若干数字顺序排列而成。而公式

<因子> ::= <常数> |(<表达式>)

的右部出现被圆括号括起来的<表达式>成分,因为前面"语法公式集"中<表达式>的定义右部出现<项>,而<项>的定义右部又出现<因子>,所以形成递归定义,即所涉及的几个语法成分之间互相引用。这种递归定义的语法公式,通过将其左部不断代入到右部相同语法成分的相应位置,就可以组成复杂的结构。

上述公式中的运算符都是放在两个运算对象中间,故称为中缀表达法。中缀表达的算术表达式在具体求值时,一般需按照如下计算次序:

① 先执行括号内的计算,后执行括号外的计算。在具有多层括号时,按层次反复地去括号,左右括号必须配对。

② 在无括号或同层括号时,先乘(*)、除(/),后加(+)、减(-)。

③ 在同一个层次,若有多个乘除(*、/)或加减(+,-)的运算,则按自左至右顺序执行。

括号可以改变运算符的优先级。例如,表达式"23+(34*45)/(5+6+7)"的计算过程是:

① 34 * 45 = 1 530;

② 5 + 6 + 7 = 18;

③ 1 530 / 18 = 85;

④ 23 + 85 = 108。

为了符合上述语法公式,该公式中34*45采用了括号的形式,与通用的"23+34*45/(5+

6 + 7)"形式的语义相同。

(4) 后缀表达式:又称逆波兰表示法,不含括号,运算符放在两个参与运算的语法成分的后面。按照后缀表达式求值时,所有的求值计算皆按运算符出现的顺序,严格从左向右进行。下面是与上述中缀表达式对应的后缀表达式的语法公式:

<表达式> ::= <项> <项> + | <项> <项> - | <项>

<项> ::= <因子> <因子> * | <因子> <因子> / | <因子>

<因子> ::= <常数>

其他两个语法公式和中缀式一样:

<常数> ::= <数字> | <数字> <常数>

<数字> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

例如,中缀表达式"23 + (34 * 45)/(5 + 6 + 7)"对应的等价后缀表达式为:

23 34 45 * 5 6 + 7 + / +

通过观察可以看出,该后缀式中所有操作数的出现次序与其在等价的中缀表达式中的出现次序完全相同,不同之处仅在于运算符按照实际计算的顺序出现在对应操作数的后面,而且消除了括号。

2. 中缀表达式到后缀表达式的转换

下面讨论如何把中缀表达式转换成等价的后缀表达式,这是栈的一个典型应用场景。在高级程序设计语言的编译软件中,就使用类似的转换算法对算术公式进行变换,最终将其转换为计算机可以直接执行的机器指令序列。

中缀表达式的运算次序受到运算符优先级和括号的影响。因此,将中缀表达式转换成等价的后缀表达式的关键在于如何恰当地去掉中缀表达式中的括号,然后在必要时按先乘除后加减的优先规则调换运算符的先后次序。在去括号的过程中用栈来存储有关的元素。

实现这个转换的基本思路如下:从左至右顺序扫描中缀表达式,用栈来存放表达式中的操作数、开括号以及在这个开括号后面的其他暂时不能确定计算次序的内容。

如果中缀表达式以字符串 InfixExp 表示,转换后的后缀表达式用字符串 PostfixExp 表示,则该转换算法的输入即为 InfixExp,输出便为 PostfixExp。在从左至右顺序扫描中缀表达式时:

(1) 当输入的是操作数时,直接输出到后缀表达式 PostfixExp 序列中。

(2) 当遇到开括号时,将其入栈。

(3) 当输入遇到闭括号时,先判断栈是否为空,若为空,则表示括号不匹配,应作为错误异常处理,清栈退出。若非空,则把栈中元素依次弹出,直到遇到第一个开括号为止,将弹出的元素输出到后缀表达式序列中。由于后缀表达式不需要括号,因此弹出的开括号不放到输出序列中,若没有遇到开括号,说明括号不匹配,做异常处理,清栈退出。

(4) 当输入为运算符(四则运算 + - * / 之一)时:

- 循环,当(栈非空 && 栈顶不是开括号 && 栈顶运算符的优先级不低于输入的运算符的优先级)时,反复操作将栈顶元素弹出,放到后缀表达式序列中。

• 将输入的运算符压入栈内。

(5) 最后,当中缀表达式的符号全部扫描完毕时,若栈内仍有元素,则将其全部依次弹出,放在后缀表达式序列的尾部。若在弹出的元素中遇到开括号,则说明括号不匹配,做异常处理,清栈退出。

这样,通过对中缀表达式进行一次扫描就可将其转换成等价的后缀表达式。上面仅给出了算法的梗概和思路,其程序实现涉及字符符号读入、语法检查以及语法错误处理等细节,有兴趣的读者可作为练习给出具体的算法。

3. 后缀表达式求值

由于按照运算符出现的次序进行,后缀表达式的计算相对比较简单。下面讨论后缀表达式求值的算法。假设后缀表达式以“ = ”等号作为结束输入的标志。计算过程使用一个栈来存放操作数,自左至右顺序扫描后缀表达式,依次分析该序列中的符号。

(1) 如果遇到一个操作数,则压入栈中。

(2) 如果遇到一个运算符,就从栈中弹出两个操作数,按照运算符对这两个操作数进行相应的运算,然后将计算结果压入栈中。

如此继续,直到遇到符号“ = ”,此时栈顶的值就是输入表达式的值。例如,后缀表达式

23 34 45 * 5 6 + 7 + / +

的实际求值过程如图 3.5 所示。自左至右扫描,遇到操作数 23、34、45 便将其压入栈中,遇到第一个运算符“ * ”时,取栈顶两次得到两个操作数 45 和 34,进行“ * ”运算,其运算结果 1 530 压回栈中,即后缀式的“34 45 * ”被其乘运算结果 1 530 所替换。再继续往右扫描,把遇到的操作数 5、6 分别压入栈中,当遇到运算符“ + ”时,同样从栈中弹出两个操作数,“5 6 + ”被它的运算结果 11 所替代。然后,把遇到的操作数 7 压入栈中,当再遇到加号时,从栈中弹出 7 和 11,“11 7 + ”

步骤	待处理的后缀表达式 (左) (右)	栈的状态 (底)				(顶)	进行的算术运算
1	23 34 45 * 5 6 + 7 + / +						
2	34 45 * 5 6 + 7 + / +	23					
3	45 * 5 6 + 7 + / +	23	34				
4	* 5 6 + 7 + / +	23	34	45			
5	5 6 + 7 + / +	23	1 530				34*45=1 530入栈
6	6 + 7 + / +	23	1 530	5			
7	+ 7 + / +	23	1 530	5	6		
8	7 + / +	23	1 530	11			5+6=11入栈
9	+ / +	23	1 530	11	7		
10	/ +	23	1 530	18			11+7=18入栈
11	+	23	85				1 530/18=85入栈
12		108					23+85=108入栈

图 3.5 后缀表达式的计算过程

计算结果18压入栈顶。其后，遇到运算符“/”时，同样从栈中弹出18、1 530，注意由于先进后出，先弹出的为第2操作数，后弹出的为第1操作数，“1 530/18”结果85压入栈中。最后，遇到运算符“+”时，弹出85和23，“23 + 85 = 108”就是表达式的最终求值结果。

下面给出这一算法的程序实现方法。假设输入的后缀表达式符合语法规范。

【算法3.5】 后缀表达式的求值。

```
// Class Declaration 类的说明
class Calculator {
private:
        Stack < double > s;                                  // 此栈用于压入保存操作数
        bool GetTwoOperands(double& opd1,double& opd2);      // 从栈顶弹出两个操作数 opd1 和 opd2
        void Compute(char op);                               // 取两个操作数,并按 op 对两个操作数
                                                             // 进行计算
public:
        Calculator(void){} ;                                 // 创建计算器实例,开辟一个空栈
        void Run(void);                                      // 读入后缀表达式,遇到符号“=”时,
                                                             // 求值计算结束
        void Clear(void);                                    // 计算器的清除,为下一次计算做准备
};

// 计算器类 class Calculator 中部分成员函数的程序实现
bool Calculator::GetTwoOperands(double& opd1,double& opd2) {
        if (s.isEmpty()) {
                cerr << "Missing operand!" << endl;
                return false;
        }
        s.pop(&opd1);                                        // 右操作数
        if (s.isEmpty()) {
                cerr << "Missing operand!" << endl;
                return false;
        }
        s.pop(&opd2);                                        // 左操作数
        return true;
}

// 调用 GetTwoOperands,并按 op 运算对两个操作数进行计算
void Calculator::Compute(char op) {
        bool result;
        double operand1,operand2;
```

```
        result = GetTwoOperands(operand1,operand2);          // 从栈中连续弹出两个操作数
        if (result == true)                                  // 根据操作符对操作数进行运算
            switch(op) {
                case'+ ': s.push(operand2 + operand1);
                          break;
                case'- ': s.push(operand2 - operand1);
                          break;
                case'* ': s.push(operand2 * operand1);
                          break;
                case'/ ': if (operand1 == 0.0) {
                              cerr << "Divide by 0!" << endl;
                              s.clear();
                          }
                          else
                              s.push(operand2 / operand1);
                          break;
            }
        else
            s.clear();
    }

    // 读入表达式。遇到操作数则入栈,遇到操作符则从从栈中连续弹出两个操作数计算,遇"="结束
    void Calculator::Run(void) {
        char c;
        double newOperand,res;
        while (cin >> c,c != '=') {                          // 根据输入数据进行判断
            switch(c) {
                case'+ ':
                case'- ':
                case'* ':
                case'/':
                        Compute(c);                          // 遇到操作符调用 Compute 计算
                        break;
                 default :                                   // 遇到操作数则入栈
                        cin.putback(c);
                        cin >> newOperand;
                        s.push(newOperand);
                        break;
            }
```

```
        }
        if (s.pop(&res))
            cout << res << endl;                    // 打印出求值的最后结果
    }
```

【算法 3.5 结束】

其中，为了示意，在处理除法运算时，算法 3.5 中采用了 if(operand1 == 0.0)的形式来判断除数是否为 0。在实际编写程序时，浮点数是否为 0 不能直接与 0 进行相等比较，而是要通过某个很小的阈值来判断，例如采用 if(abs(operand1) <1E-7)判断 operand1 的绝对值是否小于设定的值。

3.1.5 栈与递归

由于符合人类自顶向下抽象描述问题的思维方式，递归成为数学和计算机科学的基本概念，是解决复杂问题的一个有力手段。许多程序设计语言都支持递归，这些支持本质上是通过栈来实现的。

本节将以阶乘函数的计算为例，分析函数的递归调用在程序运行阶段的工作过程。在此基础上，以背包问题为例说明递归算法到非递归的一种机械转换方法。

1. 递归的定义

以阶乘函数为例来说明递归的定义。阶乘 $n!$ 的递归定义如下：

$$n! = \begin{cases} 1, 当\ n \leqslant 0 \\ n \times (n-1)!, 当\ n > 0 \end{cases}$$

为了定义整数 n 的阶乘，必须先定义 $(n-1)$ 的阶乘，而为了定义 $(n-1)$ 的阶乘，又需先定义 $(n-2)$ 的阶乘，如此直到 0 为止，因为此时阶乘定义为 1。这种用自身的简单情况来直接或间接地定义自己的方式称为递归定义。

可以看出，一个递归定义由两部分组成。其一为递归基础，也称递归出口，是递归定义的最基本情况，也是保证递归结束的前提；其二则为递归规则，确定了由简单情况构筑复杂情况需遵循的规则。上面的阶乘定义中递归出口即为 $n \leqslant 0$，此时阶乘定义为 1；递归规则为 $n \times (n-1)!$，即 n 的阶乘由 $(n-1)$ 的阶乘来构筑。这个递归定义可在 C++语言中由下面的这个递归函数来实现：

【算法 3.6】 阶乘函数。

```
long factorial(long n) {
    if (n <= 0)
        return 1;
    return n * factorial(n-1);
}
```

【算法 3.6 结束】

2. 递归函数的实现

大多数程序设计语言运行环境所提供的函数调用机制是由底层的编译栈支持的。编译栈中的“运行时环境”指的是目标计算机上用来管理存储器并保存执行过程所需信息的寄存器及存储器的结构。

在非递归调用的情况下，数据区的分配可以在程序运行前进行，直到整个程序运行结束再释放，这种分配称为静态分配。采用静态分配时，函数的调用和返回处理比较简单，不需要每次分配和释放被调用函数的数据区。在递归调用的情况下，被调函数的局部变量不能静态地分配某些固定单元，而必须每调用一次就分配一份，以存放当前所使用的数据，当返回时随即释放。这种只有在执行调用时才能进行的存储分配称为“动态分配”，此时需要在内存中开辟一个称为运行栈(runtime stack)的足够大的动态区。

用作动态数据分配的存储区可按多种方式组织。典型的组织如图 3.6 所示，将存储器分为栈(stack)区域和堆(heap)区域，栈区域用于分配具有后进先出 LIFO 特征中的数据(如函数的调用)，而堆区域则用于不符合 LIFO 的数据(如指针的分配)的动态分配。

运行栈中元素的类型(即被调函数需要的数据区类型)涉及动态存储分配中的一个重要概念：函数活动记录(activation record)。当调用或激活一个函数时，相应的活动记录包含了为该函数的局部数据所分配的存储空间。通常，活动记录至少应包括如图 3.7 所示的几部分内容。

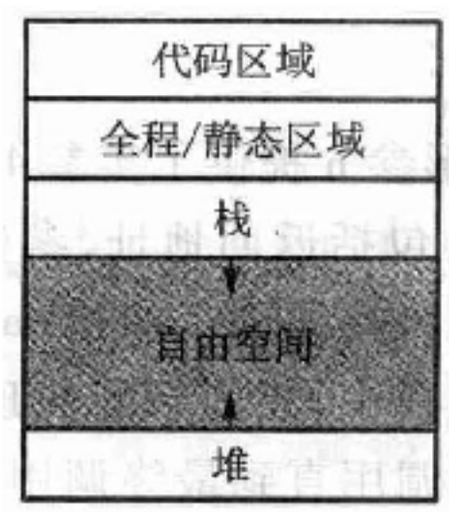


图 3.6 运行时存储器的组织形式

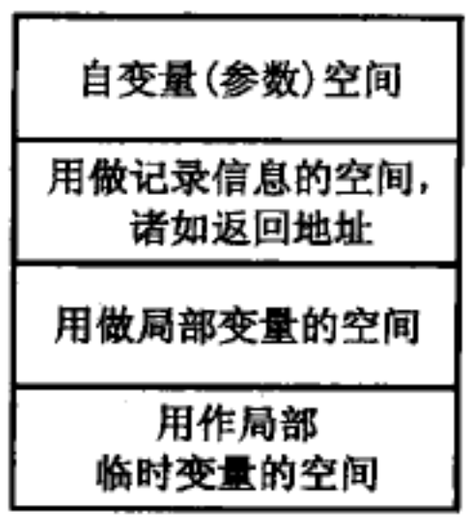


图 3.7 活动记录的内容

每次调用一个函数时，执行进栈操作，把被调函数所对应的活动记录分配在栈的顶部；而在每次从函数返回时，执行出栈操作，释放本次的活动记录，恢复到上次调用所分配的数据区中。因此，被调函数中变量地址全部采用相对于栈顶的相对地址来表示。因为运行栈中存放的是被调函数的活动记录，所以运行栈又称为活动记录栈(stack of activation record)。同时，由于运行栈按照函数的调用序列来组织，因此也称为调用栈(call stack)。

一个函数在运行栈中可以有若干不同的活动记录，每个代表了一个不同的调用。对于递归函数来说，递归深度就决定了其在运行栈中活动记录的数目。当函数进行递归调用时，函数体的同一个局部变量在不同的递归层次被分配给不同的存储空间，放在运行栈的不同位置。

概括来讲，函数调用可以分解成以下 3 步来实现：

(1) 调用函数发送调用信息，包括调用方要传送给被调方的信息，如传给形式参数(简称形

参)的实在参数(简称实参)的值、函数返回地址等。

(2) 分配被调方需要的局部数据区,用来存放被调方定义的局部变量、形参变量(存放实参的值)、返回地址等,并接收调用方传送来的调用信息。

(3) 调用方暂停,把计算控制转移到被调方,即自动转移到被调函数的程序入口。

当被调方结束运行,返回到调用方时,其返回处理一般也分解为3步进行:

(1) 传送返回信息,包括被调方要传回给调用方的信息,诸如计算结果等。

(2) 释放分配给被调方的数据区。

(3) 按返回地址把控制转回调用方。

设要计算4的阶乘,在C/C++语言中可以设计一个主程序main(参考算法3.7)来调用算法3.6定义的阶乘函数。

【算法3.7】 计算阶乘的主程序。

```
#include <iostream>
void main() {
    long x;
    cin >> x;
    cout << factorial(4) << endl;
}
```

【算法3.7结束】

主程序通过factorial(4)这个语句向阶乘函数facrotial()的形参n提供了实参4。通过调用,建立阶乘函数factorial()的一个活动记录,把当前的必要信息,包括返回地址、参数(此时传入4)、局部变量等存入栈中,如图3.8(a)所示。在计算factorial(4)时,调用了factorial(3),此时需要为新的被调函数建立活动记录(此时传入的参数为3)并压入栈中,成为新的栈顶,依次类推,factorial(3)调用又引起factorial(2)的调用,栈顶再次更新,依次调用直到最终调用factorial(0),此时factorial(0)的活动记录成为新的栈顶。由于factorial(0)满足递归的出口条件,可以直接得到结果,执行结束后,其活动记录从栈顶弹出,并将计算结果和控制权返回给其调用方factorial(1)。factorial(1)根据factorial(0)的返回结果1可以计算出1! = 1,执行结束后,也从栈顶弹出其活动记录,继续将控制权转移给它的调用方factorial(2),如图3.8(b)所示,按进栈顺序的反序依次从栈中删除每个活动记录,把计算结果和控制权逐层上移,最后factorial(4)把控制连同计算结果24返回给调用它的main()函数。这样,当在main()中执行cout语句时,只在运行时环境中保留了main()和全局/静态区域的活动记录。

3. 递归算法到非递归的转换

递归的算法具有可读性强、结构简练、正确性易证明等优点,但是在时空的开销上相对较大。为提高算法的时空效率,尤其是在某些对响应时间很敏感的实时应用环境下,或在不支持递归的程序环境中,必须将递归算法转化为非递归算法,问题才能得到有效解决。

把一个递归算法转化为相应的非递归算法的方法很多。本节重点介绍一种利用栈进行转换

的方法，以进一步揭示递归的本质以及栈与递归的内在联系。

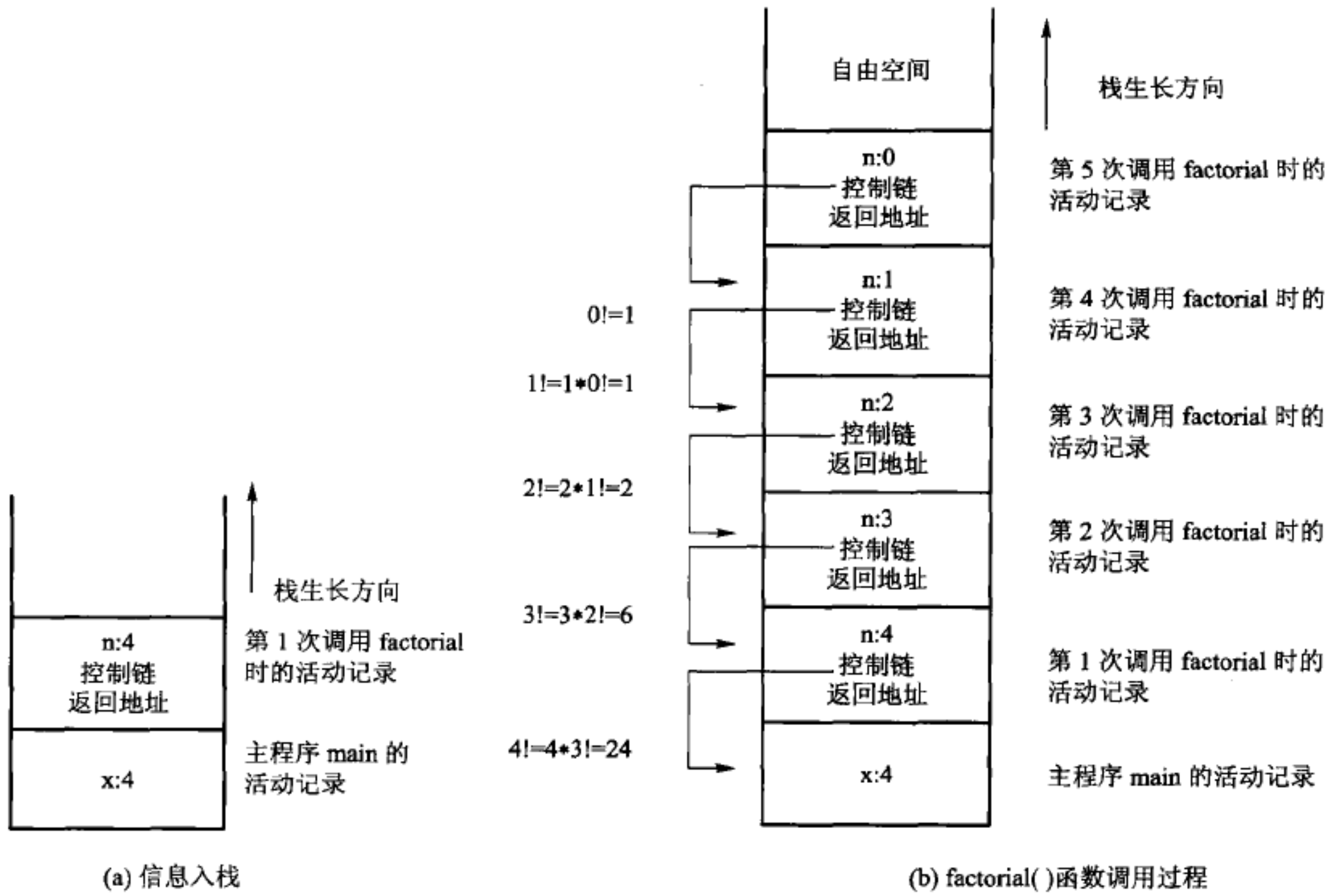


图 3.8 递归计算 factorial(4)时，内部栈的变化情况示意

递归有很多分类方法。例如，根据递归所处的位置可分为尾部递归和非尾部递归，所谓尾部递归是指递归函数中最后一个操作是一个递归调用，之后不再有其他语句；与尾部递归相对的是非尾部递归。此外，根据递归的调用方式可分为直接递归和间接递归，根据有无嵌套还可分为嵌套递归和无嵌套递归等。

根据是否需要回溯，还可以把递归分成简单递归和复杂递归两种。简单递归一般可以根据递归式来找出其递推公式。例如，阶乘函数这样简单的递归基本上可以用循环迭代的方式来取代递归，循环结束条件通常比较容易确定。阶乘的迭代实现如算法 3.8 所示。

【算法 3.8】 阶乘的迭代实现。

```
long factorial(long n) {
    long m = 1;
    long i ;
    if ( n > 0)
        for ( i = 1; i <= n; i ++ )
            m = m * i ;
```

```
        return m ;
    }
```

【算法 3.8 结束】

而复杂递归的循环结束条件就不那么容易确定，需要对整个递归程序进行分析。一般情况下，可模拟编译系统处理递归的机制，使用栈等数据结构保存回溯点来求解。

以阶乘函数为例，其递归出口为 n <=0 时返回结果 1，在 n >0 的情况下均需按照一个递归规则来统一处理：求其前驱 n - 1 的阶乘。因此，在利用栈将其转换成非递归的过程中，遇到不等于 0 的参数 n，则按递归规则将其压栈，并将其值减 1；当遇到 0 时则停止递归，将其结果返回给上层调用函数。算法 3.9 是根据此转换过程得到的非递归算法的模拟实现，其中，由于递归出口的返回值为 1，因此将中间变量 m 的初值置为 1，以简化出口的处理。

【算法 3.9】 阶乘的一种非递归实现。

```
long factorial( long n) {
    Stack < long > s;
    long tmp;
    long m = 1 ;
    while ( n >0)                    // 不满足递归出口
        s. push( n -- ) ;            // 按递归规则把相应数据压栈
    while ( s. pop( &tmp) )          // 满足递归出口，开始进行返回处理
        m  *= tmp;
    return m ;
}
```

【算法 3.9 结束】

通过上面介绍的递归函数的实现机制，可以看出系统处理递归的简单准则：当遇到递归规则（涉及递归调用）时进行压栈操作，把递归函数的相关信息保存在栈中；当遇到递归出口时则进行出栈的操作，把结束了的被调递归函数的结果等信息返回上一级的调用函数。

下面介绍模拟编译系统的递归机制，机械地把一个包含 t 个递归规则的复杂递归算法转换为非递归算法的过程。首先设置一个工作栈，每遇到一个递归规则都进行压栈操作，遇到递归出口则进行出栈操作，并根据其对应的递归规则来进行相应的递归返回处理。共有 t 种不同的返回处理情况，外加整个递归程序的入口和出口处理，总共 t + 2 种不同的处理情况。具体的模拟过程可分成以下几个步骤：

（1）设置一个工作栈 S 保存当前工作记录：定义栈的数据成员，把在函数中出现的所有参数、局部变量、返回语句标号等信息保存起来，类似于系统运行栈的活动记录。

（2）设置 t + 2 个语句标号：label0 标示第一个可执行语句；labelt + 1 则设置在函数体结束处，作为每一个递归出口都需执行的语句；labeli（$1 \leqslant i \leqslant t$）为第 i 个递归返回处理的语句。

（3）增加非递归入口：作为递归函数的入口，进行压栈操作，采用形如 S. push（t + 1，p_1，p_2，…，p_m，q_1，q_2，…，q_n）；的语句，其中，t + 1 为递归出口处理语句的标号，p_1，…，p_m 表示递归函

数的参数，$q_1, q_2, \cdots, q_n$ 表示该函数的局部变量。

(4) 替换第 i(i = 1, …, t)个递归规则：若函数体中第 i(i = 1, …, t)个递归调用语句形如 rf($a_1, a_2, \cdots, a_m$)；则替换为

```
S. push(i,a1,…,am); // 实参进栈
goto label0;
```

并且增加标号为 i 的语句：

```
labeli: S. pop(&x);
// 根据栈顶的工作记录 S. top (&x)的取值进行相应的返回处理
```

(5) 所有递归出口处增加语句 goto labelt + 1；以进行递归结束处的相关处理。

(6) 标号为 t + 1 的语句是一个形如

```
switch (S. top(&x). rd) {
    case 0 : goto label0;
             break;
    case 1 : goto label1;
             break;
    ...
    case t + 1 : S. pop(&item)
             // 返回处理
             break;
    default : break;
}
```

的分支语句，根据不同的递归规则进行相应的处理，其中 rd 表示返回地址。

(7) 改写循环和嵌套中的递归，使其都在同层次，不再嵌套，有时需要把有返回值的函数改写成 void 型函数，把返回值作为函数的参数，然后应用第(4)步解决。

(8) 优化处理：分析程序结构，消除其中冗余的进栈、出栈操作，并消除 goto 语句。

下面以经典的背包问题为例来说明这个转换过程。考虑简化了的背包问题：设有一个背包可以放入的物品重量为 s，现有 n 件物品，重量分别为 $w_0, w_1, \cdots, w_{n-1}$，问能否从这 n 件物品中选择若干件放入背包，使其重量之和正好为 s。如果存在一种符合上述要求的选择，则称此背包问题有解，否则无解。

如果用 knap(s,n) 表示上述背包的解，其返回值应为布尔值。背包问题如果有解，其选择只有两种可能：选择一组物品中包含 w_{n-1}，此时 knap(s,n)的解就是 knap($s - w_{n-1}$, n − 1)的解；选择中不包含 w_{n-1}，这样 knap(s,n)的解就是 knap(s,n − 1)的解。即

$$\text{knap}(s,n) = \begin{cases} \text{true}, & \text{当 } s = 0 \\ \text{false}, & \text{当 } s < 0 \text{ 或 } s > 0 \text{ 且 } n < 1 \\ \text{knap}(s - w_{n-1}, n-1) \parallel \text{knap}(s, n-1), & \text{当 } s > 0 \text{ 且 } n \geqslant 1 \end{cases}$$

根据定义可以编写出相应的递归算法。设给定的 n 件物品的重量存放在一个正整数的数组

w 中，w[i]表示第 i 个物品的重量，算法 3.10 给出了背包问题的递归解法。当背包问题有解时，算法 3.10 还会输出被选中的一组物品的重量。

【算法 3.10】 背包问题的递归算法。

```
bool knap(int s,int n) {
    if (s == 0)
      return true;
    if ((s<0) || (s>0 && n <1))
      return false;
    if (knap (s-w[n-1],n-1)) {
      cout << w[n-1];
      return true;
    }
    else return knap(s,n-1);
}
```

【算法 3.10 结束】

可以看到，算法 3.10 中递归的出口有两个：其一，当背包的承重量为 0 时，背包问题有解，只要不选择任何物品即可；其二，当背包承重量为负，或者虽然背包承重量大于 0 但物品个数小于 1 时，背包问题无解。当满足这两个出口之一的条件时，背包问题便结束递归。尚未达到递归出口时，背包问题的递归规则也有两种：

规则 1：若 w[n-1]包含在解中，求解 knap(s-w[n-1],n-1)。

规则 2：若 w[n-1]不包含在解中，求解 knap(s,n-1)。

首先从设计栈开始，栈中每个结点应该包含以下 4 个域：参数 s 和 n，返回地址 rd 和结果单元 k。由于 knap 算法中有两处递归规则，因此加上递归出口返回地址共有 3 种情况：

(1) 计算 knap(s,n)完毕返回到调用本函数的其他函数。

(2) 计算 knap(s-w[n-1],n-1)完毕，返回到本调用函数继续计算。

(3) 计算 knap(s,n-1)完毕，返回到本调用函数继续计算。

为区分这 3 种返回情况，rd 分别用 0、1、2 表示。因此，栈结点可声明如下：

```
enum rdType {0,1,2};
public class knapNode {
    int       s,n;          // 背包的承重量和物品的数目
    rdType    rd;           // 返回地址
    bool      k;            // 结果单元
}
```

并引入两个与栈中结点类型相同的变量 tmp 和 x 作为进出栈的缓冲：

```
knapNode  tmp, x;
```

定义一个栈变量：

```
    Stack < knapNode > stack;
```

【算法 3.11】 背包问题的非递归实现。

```
    bool nonRecKnap(int s,int n) {
        tmp.s = s, tmp.n = n, tmp.rd = 0;          // 非递归调用入口
        stack.push(tmp);
      label0:                                      // 递归调用入口
        stack.pop(&tmp);                           // 查看栈顶元素并分情况处理
        if (tmp.s == 0) {                          // 若满足递归出口条件
            tmp.k = true;                          // 修改栈顶的结果单元 k
            stack.push(tmp);
            goto label3;                           // 转向递归出口处理
        }
        if (tmp.s <0) || (tmp.s >0 && tmp.n <1) {
            tmp.k = false;                         // 修改栈顶的结果单元 k
            stack.push(tmp);
            goto label3;
        }
        stack.push(tmp);                           // 尚未满足递归出口
        x.s = tmp.s - w[tmp.n - 1];                // 按照规则 1 进行压栈处理
        x.n = tmp.n - 1;
        x.rd = 1;
        stack.push(x);
        goto label0;
      label1:                                      // 规则 1 对应的返回处理
        stack.pop(&x);                             // 查看栈顶,根据其内容分情况处理
        if (tmp.k == true) {                       // 若某层的结果单元为 true
            x.k = true;                            // 把 true 结果上传给调用方
            stack.push(x);
            cout << w[x.n - 1] << endl;            // 并输出对应的物品
            goto label3;
        }
        stack.push(x);                             // 若某层的结果单元为 false
        tmp.s = x.s;                               // 当前物品的选择不合适,回溯,调用规则 2
        tmp.n = x.n - 1;                           // 按照规则 2 进行压栈处理
        tmp.rd = 2;
        stack.push(tmp);
        goto label0;
      label2:                                      // 规则 2 对应的返回处理
```

```
    stack.pop(&x);
    x.k = tmp.k;                          // 结果单元 k 的内容上传给调用方
    stack.push(x);
label3:                                   // 递归出口处理
    stack.pop(&tmp);
    switch (tmp.rd) {
        case 0: return tmp.k;             // 算法结束并返回结果
        case 1: goto label1;              // 转向规则 1 的返回处理处
        case 2: goto label2;              // 转向规则 2 的返回处理处
    }
}
```

【算法 3.11 结束】

而后,转换按下面方式进行:

(1) 凡调用语句 knap(s,n)均代换成

① tmp.s = s;

tmp.n = n;

tmp.rd = 返回地址编号

② stack.push(tmp);

③ 转向递归入口

(2) 将调用出口的返回处理统一成:

① stack.pop(&tmp);

② 分以下情况执行:

ⅰ) tmp.rd = 0 时,算法结束。

ⅱ) tmp.rd = 1 时,转规则 1 返回处理。

ⅲ) tmp.rd = 2 时,转规则 2 返回处理。

这样转换之后便可得到一个如算法 3.11 所示的功能相同的非递归算法。该算法执行结束时,背包问题的解在 tmp.k 中,若其值为 true,则具体选中的 w[i]已在算法中输出。

可以看出,算法 3.11 中有多处弹出栈顶元素、修改其中的结果单元 k 后又重新压栈的操作。仔细分析这些操作之后,会发现算法 3.11 还有进一步优化的余地。首先,从计算过程来看,一旦在某层调用中相应栈顶元素的 k 值为 true,则此背包问题的解为真,并且将 true 逐层向外传递,将各层的 k 都置成 true,因此可采用一个全程变量 k 作为结果单元,将其初始状态设置为 false。一旦置成 true 后就不再变化,从而可省去栈中各结点的 k 数据域及其多次的重复赋值和进栈、出栈操作。其次,从调用过程来看,每递归进入一层,参数 n 就减 1,栈的指针 top 就加 1,只要在计算前将 top 置为 -1,则递归过程总有栈顶元素的 n 数据域满足 st[top].n + t = n_0,st[top].n < l 的条件就等价于 t > n_0 - 1,如果栈顶指针可访问,就可省去栈中的数据域 n,用 t > n_0 - 1 替换算法中 tmp.n < 1;最后,消除算法中的 goto 语句,便得到更为简洁优化的算法 3.12。改进后的版本

中栈的结点类型替换成下述的形式：

```
public class knapNode {
    int        s;                    // 背包的承重量
    rdType     rd;                   // 返回地址
}
```

【算法 3.12】 背包问题的非递归实现的优化版

```
bool nonRecKnapOpt(int s,int n) {
  int  t,n0 = n;
  bool k = false;

  tmp.s = s,tmp.rd = 0;
  stack.push(tmp);

  while (! stack.isEmpty()) {
      t = stack.top;
      stack.top(&tmp);
      while (tmp.s >= 0) && (tmp.s <= 0 || n0 > t) {     // 处理栈顶元素,以判断是否满足
                                                          // 递归出口条件
          if (tmp.s == 0 ) {
              k = true;
              break;
          } else {                                  // 尚未达递归出口前,按规则 1 进行压栈操作
              x.s = tmp.s - w[n0 - 1 - t];
              x.rd = 1;
              stack.push(x);
          }
          t = stack.top;
          stack.top(&tmp);
      }
      while (! stack.isEmpty()) {                   // 返回处理
          stack.pop(&tmp);
          t = stack.top;
          if (tmp.rd == 0)                          // 算法结束
              return k;
          if (tmp.rd == 1)                          // 从规则 1 返回
              if (k == true)                        // 结果为真则打印对应的物品
                  cout << w[n0 - 1 - t] << endl;
              else {                                // 否则回溯,采用规则 2 进栈
                  stack.top(&x);
```

```
                    tmp. s = x. s;
                    tmp. rd = 2;
                    stack. push( tmp) ;
                    break;
                }
            }
        }
    }
```

【算法 3.12 结束】

3.2 队 列

与栈类似,队列(queue)也是一种限制访问点的线性表。队列的元素只能从表的一端插入,另一端删除。按照习惯,通常会把只允许删除的一端称为队列的头,简称队头(front),把删除操作本身称为出队(dequeue);而称表的另一端为队列的尾,简称队尾(rear),这一端只能进行插入操作,称为入队(enqueue)。如同现实生活中的队列一样,在没有人为破坏的前提下,队列是按照到达的顺序来释放元素的,即先进入队列中的元素会先离开,因此队列通常也被称为先进先出(first in first out)表,简称 FIFO 表。图 3.9 所示为队列的一个示例。售票窗口排队等待买票的人即为队列的一个日常实例。

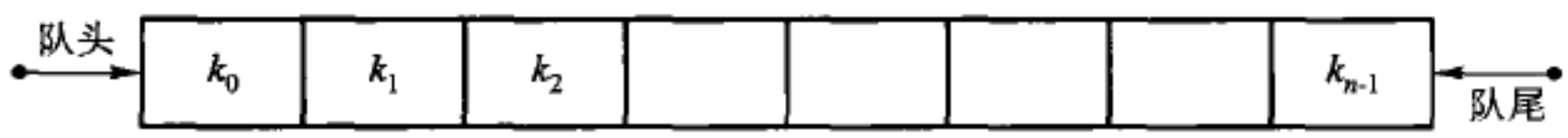


图 3.9 队列的示例

3.2.1 队列的抽象数据类型

根据数据抽象和封装的原则,对队列的操作只能通过队列的抽象数据类型所定义的运算来进行。代码 3.13 给出了一个采用 C++类模板的队列抽象数据类型定义,包括队列的入队、出队、查看队头等常用运算。

【代码 3.13】 队列的抽象数据类型定义。

```
// 栈的元素类型为 T
template <class T>
class Queue {
public:                          // 队列的运算集
    void clear( );               // 变为空队列
```

```
    bool enQueue(const T item); // 将 item 插入队尾,成功则返回真,否则返回假
    bool deQueue(T& item);      // 返回队头元素并将其从队列中删除,成功则返回真
    bool getFront(T& item);     // 返回队头元素,但不删除,成功则返回真
    bool isEmpty();             // 返回真,若队列已空
    bool isFull();              // 返回真,若队列已满
};
```

【代码 3.13 结束】

根据具体应用的不同需要,可以适当增删抽象数据类型中所定义的运算。例如,链式队列并不需要判断队列是否满。队列抽象数据类型中运算的实现方法依赖于队列的存储结构。下面介绍常用的顺序队列和链式队列两种存储结构。

3.2.2 顺序队列

用顺序存储结构来实现队列就形成了顺序队列(array - based queue)。与顺序表一样,顺序队列需要分配一块连续的区域来存储队列的元素,需要事先知道或估算队列的大小。与栈类似,顺序队列也存在溢出问题,队列满时入队会产生上溢,而当队列为空时出队则会产生下溢。在队列出现上溢出时,如果需要,也可以考虑对队列进行适当的扩容。

如何有效地实现顺序队列,需要一些灵活的变通。如果只是沿用顺序表的实现方法,就很难取得良好的效率。假设队列中有 n 个元素,顺序表的实现需要把所有元素都存储在数组的前 n 个位置上。如果选择把队列的尾部元素放在位置 0,则出队列操作的时间代价为 $O(1)$,因为队列最前面的元素是数组最后面的元素,但是此时入队列操作的时间代价为 $O(n)$,因为必须把队列中当前元素都向后移动一个位置。反之,如果把队列尾部放在位置 $n-1$,则入队列操作只需在最后添加即可,时间代价仅为 $O(1)$;而出队列操作需要移动剩余 $n-1$ 个的元素以确保它们在表的前 $n-1$ 个位置,其代价为 $O(n)$。

如果在保证队列元素连续性的同时允许队列的首尾位置在数组中移动,则可以得到更加有效的实现方法,如图 3.10 所示。在经过多次的入队和出队操作之后,队头元素由 12 变成了 8,而队尾元素也变成了新入队的 16。即随着出队操作的执行,表示队头的 front 值不断后移,同时随着入队操作的执行,表示队尾的 rear 值也在增加。

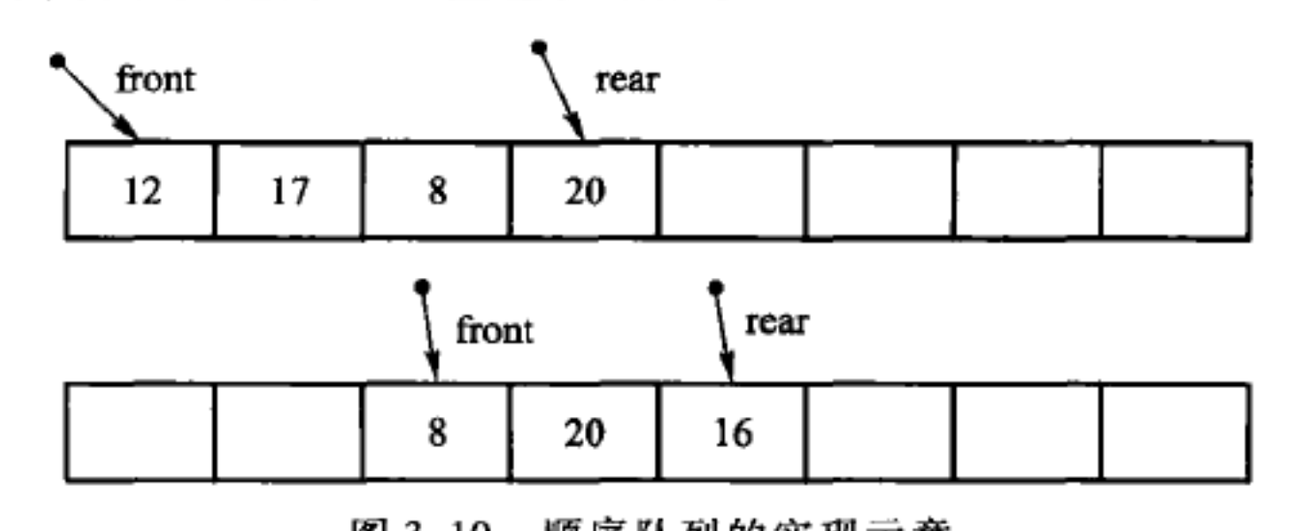
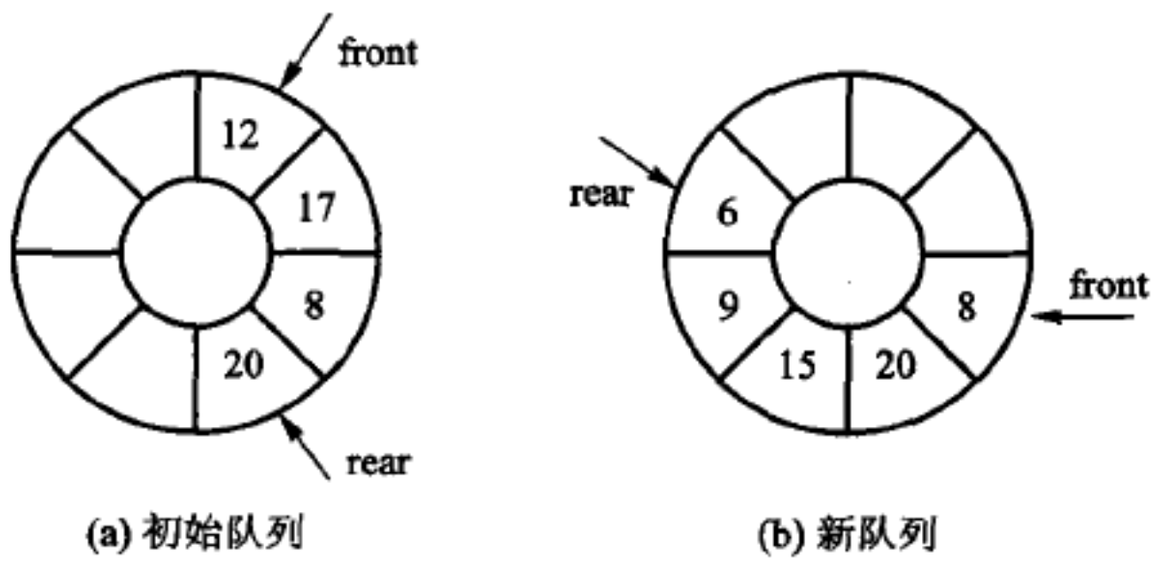


图 3.10　顺序队列的实现示意

随着时间的推移,整个队列会向数组的尾部移动,一旦到达数组的最末端,即 rear 等于 mSize - 1,即使数组的前端可能还有空闲的位置,再进行入队操作也会发生溢出。这种数组实际上尚有空闲位置而发生上溢的现象称为"假溢出"。解决假溢出的方法便是采用循环的方式来组织存放队列元素的数组,在逻辑上将数组看成一个环,即把数组中下标编号最低的位置(0 位置)看成是编号最高的位置(mSize - 1)的直接后继,这可以通过取模运算实现。即数组位置 x 的后继位置为(x + 1)% mSize,这样就形成了循环队列,也称环形队列。图 3.11 所示为一个环形队列的示例。起初,队首存放在数组中编号较低的位置,队尾则存放在数组中编号较高的位置,沿顺时针方向存放队列。这样,入队操作增加 rear 的值,出队操作增加 front 的值。



(a) 初始队列 (b) 新队列

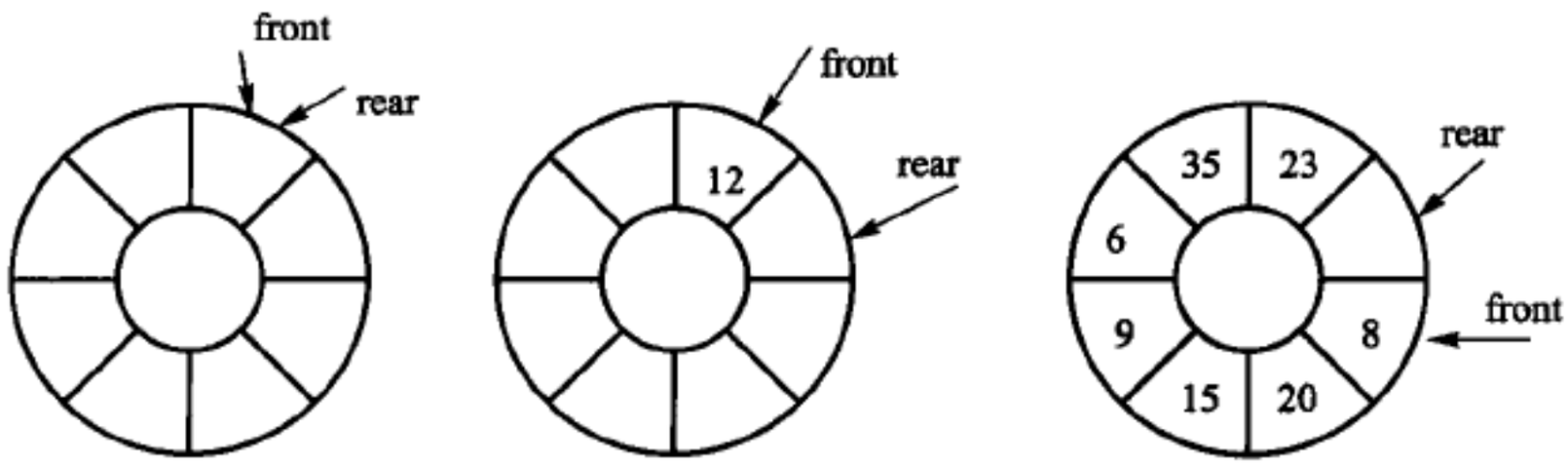
图 3.11 顺序队列的一种实现方式

初始时队列中有 4 个元素 12、17、8 和 20,如图 3.11(a)所示。经过两次出队操作和三次入队操作之后形成了包括元素 8、20、15、9、6 的新队列,如图 3.11(b)所示。

按图 3.11 所示的方式组织顺序队列时,用 front 存储队列头元素的位置,rear 存储队尾元素的位置。若 front 和 rear 的位置相同,则表明队列中只有一个元素。下面介绍如何表示一个空队列,如何表示队列已被元素填满。

首先,忽略队首 front 的实际位置和其内容时,队列中可能没有元素(空队列)、有一个元素、有两个元素等。如果数组有 n 个位置,则队列中最多有 n 个元素。因此,队列有 n + 1 种不同的状态。如果把队首 front 的位置固定下来,则 rear 应该有 n + 1 种不同的取值来区分这 n + 1 种状态,但实际上 rear 只有 n 种可能的取值,除非有表示空队列的特殊情形。换言之,如果用位置 0 ~ n - 1 间的相对取值来表示 front 和 rear,则 n + 1 种状态中必有两种不能区分。因此,需寻求其他途径来区分队列的空与满。

一种方法是记录队列中元素的个数,或者用至少一个布尔变量来指示队列是否为空。此方法需要每次执行入队或出队操作时设置这些变量。另一种方法,也是顺序队列通常采用的方法,是把存储 n 个元素的数组的大小设置为 n + 1,即牺牲一个元素的空间来简化操作的实现和提高操作的效率。图 3.12 所示为一个如此实现的循环队列,8 个元素大小的数组在放入 7 个元素后即称满,如若再要插入就会发生溢出。其中,图 3.12 (a) 表示队列的空状态,此时 front = rear;图 3.12(b)表示队列的一般状态,此时 rear = (front + 1) % (n + 1);图 3.12(c)则表示队列的满状态,此时(rear + 1) % (n + 1) = front。



(a) 队列为空的状态　　(b) 队列的一般状态　　(c) 队列的满状态

图 3.12　循环队列的实现示例

代码 3.14 给出了一个顺序队列的实现方法。

【代码 3.14】 队列的顺序实现。

```
template <class T>
class arrQueue: public Queue<T> {
private:
    int  mSize;                          // 存放队列的数组的大小
    int  front;                          // 表示队头所在位置的下标
    int  rear;                           // 表示队尾所在位置的下标
    T    *qu;                            // 存放类型为 T 的队列元素的数组
public:                                  // 队列的运算集
    arrQueue(int size) {                 // 创建队列的实例
        mSize = size + 1;                // 浪费一个存储空间,以区别队列空和队列满
        qu = new T [mSize];
        front = rear = 0;
    }
    ~arrQueue() {                        // 消除该实例,并释放其空间
        delete [] qu;
    }
    void clear() {                       // 清空队列
        front = rear;
    }
    bool enQueue(const T item) {         // item 入队,插入队尾
        if (((rear + 1) % mSize) == front) {
            cout << "队列已满,溢出" << endl;
            return false;
        }
        qu[rear] = item;
        rear = (rear + 1) % mSize;       // 循环后继
```

```
        return true;
    }
    bool deQueue(T& item) {                    // 返回队头元素并从队列中删除
        if ( front == rear) {
            cout << "队列为空" << endl;
            return false;
        }
        item = qu[front];
        front = (front + 1) % mSize;
        return true;
    }
    bool getFront(T& item) {                   // 返回队头元素,但不删除
        if (front == rear) {
            cout << "队列为空" << endl;
            return false;
        }
        item = qu[front];
        return true;
    }
}
```

【代码 3.14 结束】

3.2.3 链式队列

链式队列(linked queue)是队列的链式实现,是对链表的简化。代码 3.15 给出了链式队列的实现方法,其中链接的方向从队列的前端指向队列的尾端。成员 front 和 rear 是分别指向队首和队尾的指针,结点类型采用第 2 章中已定义过的 Link 类模板;同时,为了方便统计队列中元素的个数,增加成员 size 来表示队列中当前元素的个数。

【代码 3.15】 队列的链式实现。

```
template <class T>
class lnkQueue: public Queue <T> {
private:
    int     size;                      // 队列中当前元素的个数
    Link <T> * front;                  // 表示队头的指针
    Link <T> * rear;                   // 表示队尾的指针
public:                                // 队列的运算集
    lnkQueue(int size) {               // 创建队列的实例
```

```
        size = 0;
        front = rear = NULL;
    }
    ~lnkQueue() {                    // 消除该实例,并释放其空间
        clear();
    }
    void clear() {                   // 清空队列
        while (front != NULL) {
            rear = front;
            front = front -> next;
            delete rear;
        }
        rear = NULL;
        size = 0;
    }
    bool enQueue(const T item) {     // item 入队,插入队尾
        if (rear == NULL) {          // 空队列
            front = rear = new Link<T> (item, NULL);
        }
        else {                       // 添加新的元素
            rear -> next = new Link<T> (item, NULL);
            rear = rear -> next;
        }
        size ++;
        return true;
    }
    bool deQueue(T& item) {          // 返回队头元素并从队列中删除
        Link<T> * tmp;
        if (size == 0) {             // 队列为空,没有元素可出队
            cout << "队列为空" << endl;
            return false;
        }
        &item = front -> data;
        tmp = front;
        front = front -> next;
        delete tmp;
        if (front == NULL)
            rear = NULL;
```

```
        size --;
        return true;
    }
    bool getFront(T& item) {            // 返回队头元素,但不删除
        if (size == 0) {                // 队列为空,无法获取队头
            cout << "队列为空" << endl;
            return false;
        }
        item = front -> data;
        return true;
    }
}
```

【代码 3.15 结束】

从代码 3.15 可以看出,链式队列的各成员函数是对链表相应函数的修改。本质上,入队操作 enQueue 只是简单地把新元素放到链表的尾部,并修改 rear 使其指向新的结点;出队操作 deQueue 只是简单地删除表中最前面的那个结点,并修改 front 指针使其指向新的队首。为了减少访问队首结点的时间代价,这种实现方法中没有链表中的专用表头虚结点,因此 enQueue 操作中必须单独处理插入到空队列的情况,deQueue 操作也需考虑出队后导致空队列的情况。

3.3 栈与队列的深入讨论

3.3.1 顺序栈与链式栈的比较

栈的应用非常广泛,只要满足后进先出的特性都可以使用栈结构。例如,函数调用和递归实现、深度优先周游、表达式转换和求值等。

1. 顺序栈与链式栈的比较

顺序栈和链式栈的基本操作都只需要常数时间,因此二者在时间效率上难分伯仲。从空间角度来看,初始时顺序栈必须说明一个固定的长度,当栈不够满时,势必浪费一些空间;链式栈的长度可根据需要而增减,但每个元素都需要一个指针域,从而产生结构性开销。

在栈的实际应用中,有时需要访问栈的内部元素。顺序栈可以根据元素与栈顶的相对位置快速定位并读取内部元素,而链式栈则需要沿着指针链遍历才能访问内部元素。

鉴于上述原因,顺序栈在实际中更为常用一些。

2. 多栈管理

在编译系统等计算机系统软件中,往往同时使用和管理多个栈,这时可充分利用顺序栈单向

延伸的特性。例如,可以使用一个数组来存储两个栈,让它们共享同一存储空间。把数组的两端设置为两者各自的栈底,从两端开始向中间迎面延伸(见图 3.13),这样浪费的空间相对少一些。在两个栈的空间需求恰好相反,即一个栈增长另一个收缩时,这种方法很奏效。反之,如果两个栈同时增长,数组中间的空间会很快用完。

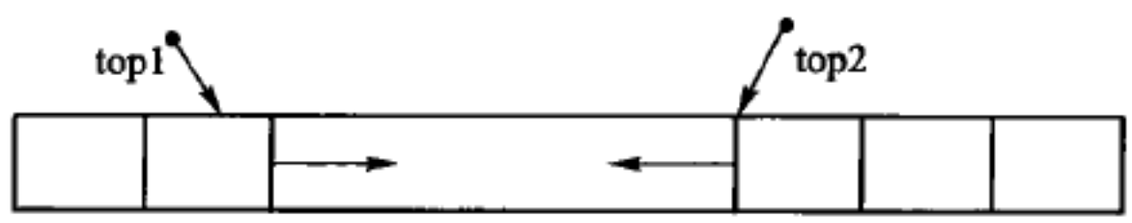


图 3.13 存储在同一个数组中的两个迎面增长的栈

3.3.2 顺序队列与链式队列的比较

由于存储空间固定,顺序队列无法满足队列规模变化很大且最大规模无法预测的情况,而链式队列则可以轻松应对这种类型的应用。另一方面,顺序队列在存取访问上很简单,可以适用那些对队列内部元素(不仅限于队列的两端)有访问需求的应用。如果可以预测浪涌统计特性就可以大致估计缓冲队列的长度。

顺序队列和链式队列中常用的入队和出队操作都需要常数时间,二者在时间效率上没有优劣之分。在空间代价上与栈的情况类似。只是顺序队列不像顺序栈那样,不能在一个数组中存储两个队列,除非总有数据项从一个队列转入另一个队列。

队列通常作为消息或数据缓冲器在很多领域得到广泛应用,但要满足先来先服务特性。例如,计算机的硬件设备之间需要队列作为其数据通信的缓冲。操作系统中使用队列对内存、打印机等各种资源进行管理,而且根据不同优先级别的服务请求,按优先类别把服务请求分别组织成多个不同的队列。

另外,在后续章节中将会看到,队列是广度优先搜索中采用的主要中间数据结构。

3.3.3 限制存取点的表

利用栈和队列的思想可以设计出一些变种的栈或队列结构。虽然它们的应用没有栈和队列那样广泛,但在一些特定情况下具有很好的应用价值。

(1) 双端队列:限制插入和删除在线性表的两端进行。栈和队列都是特殊的双端队列,对其存取设置了限制。

(2) 双栈:两个底部相连的栈。双栈是一种添加限制的双端队列,它规定从 end1 插入的元素只能从 end1 端删除,而从 end2 插入的元素只能从 end2 端删除。

(3) 超队列:一种删除受限的双端队列,删除只允许在一端进行,而插入可在两端进行。

(4) 超栈:一种插入受限的双端队列,插入只限制在一端而删除允许在两端进行。

由此可见，这几种限制存取点的表都是某种受限的双端队列。STL 提供的基本序列中就有双端队列 deque，栈和队列都是通过 deque 实现的。

本章小结

栈是一种限制访问端口的线性表，常被称为后进先出(LIFO)表。其元素的插入和删除都只能在表的一端进行，该端称为栈的栈顶，另一端则叫做栈底。栈的特点是每次取出(并被删除)的元素总是当前栈内元素中最后压入的，而最先压入的元素则被放在栈的底部，要到最后才能取出。

栈的运算包括压栈、出栈、返回栈顶元素、判断栈是否为空等运算，运算的时间代价均为常数时间。

栈的实现通常有顺序栈和链式栈两种。顺序栈用数组实现；而链式栈则用单链表方式存储，其中的指针方向是从栈顶向下链接。在实际中，顺序栈比链式栈使用得更为广泛。

栈是一种很重要、应用非常广泛的数据结构。常见的应用包括表达式转换和求值、函数调用和递归实现、深度优先搜索等。栈的一个很重要的应用在于对函数调用机制和递归实现的支持。本章通过递归函数的实现机制和递归算法到非递归算法的转换揭示了递归的本质以及栈与递归的内在联系。

队列也是一种限制访问端口的特殊线性表。其元素的删除只限于队列头一端进行；而元素的插入则被限制于队列尾一端。队列的特点是新来的成员总是加入到队的末尾，而每次取出的元素总是来自队列的前端。队列通常称为先进先出(FIFO)表。

队列的运算主要包括进队列、出队列、取队头元素、判断队列是否为空、判断队列是否已满等运算，运算的时间代价均为常数时间。

队列的存储结构和实现方法主要分为顺序和链式两种。顺序队列是比较普遍的一种队列实现方式，采用环状顺序表来存放队列元素，并用两个变量分别指向队列的头和尾，往队列中加入元素或从队列中取出元素时分别改变队列头和尾这两个变量的计数。链式队列一般用单链表方式存储队列，其中链接指针的方向是从队列的前端向尾端链接。

顺序队列采用固定的存储空间，一方面在读访问时非常简单，在特殊应用中需要访问内部元素时也很方便；但另一方面在某些无法事先估计其大小的情况下会出现溢出现象；链式队列可以通过动态申请空间来满足队列长度的大幅变动。

队列也是一种应用很广泛的数据结构。常见的应用包括广度优先搜索、消息缓冲器、操作系统的各种资源管理等，只要符合先进先出特性的应用均可运用队列。

习 题

1. 如果允许在循环队列的两端都可以进行插入和删除操作。要求：

(1) 写出循环队列的类型定义。

(2) 写出“从队尾删除”和“从队头插入”的算法。

2. 如果用一个循环数组 q[0..m-1]表示队列时，该队列只有一个队列头指针 front，不设队列尾指针 rear，而设置计数器 count 用以记录队列中结点的个数。试编写算法实现队列的 3 个基本运算：判空、入队、出队。

3. 试按以下的要求把栈 S 中的元素逆置：

(1) 使用额外的两个栈。

(2) 使用额外的一个队列。

(3) 使用额外的一个栈，外加一些非数组的变量。

4. 试给出用栈所定义的队列。

5. 假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素结点(注意不设头指针)，试编写相应的队列初始化、入队列和出队列的算法。

6. 试在一个长度为 n 的数组中实现两个栈，使得二者在元素的总数目为 n 之前都不溢出，并且保证 push 和 pop 操作的时间代价为 $O(1)$。

7. 编号为 1、2、3、4、5 的 5 辆列车顺序开进栈式结构的站台，试问开出车站的顺序有多少种可能，并予以解释。

8. 证明：从初始输入序列 $1,2,\cdots,n$，可以利用一个栈得到输出序列 $p_1,p_2,\cdots,p_n$($p_1,p_2,\cdots,p_n$ 是 $1,2,\cdots,n$ 的一种排列)的充分必要条件是不存在下标 i、j、k，满足 $i<j<k$ 同时 $p_j<p_k<p_i$。

9. 试利用栈计算后缀表达式 12 8 9 * +，并明确写出每个步骤以及每个步骤的栈的状态。

10. 试利用栈将中缀表达式 a*(b*c-d)+e 转换成后缀式。试写出每个步骤以及每个步骤的栈的状态。

11. 试写出一个按照下面定义计算整数 n 和 m 的最大公因子 $GCD(n,m)$ 的递归算法：

$$GCD(n,m)=\begin{cases} m & \text{当 } m\leqslant n \text{ 且 } n \bmod m=0 \\ GCD(m,n) & \text{当 } n<m \\ GCD(m,n \bmod m) & \text{否则} \end{cases}$$

12. 试写出计算下面序列前 n 项之和的递归算法，并给出相应的使用栈的非递归算法。

$$1+\frac{1}{2}-\frac{1}{3}+\frac{1}{4}-\frac{1}{5}\cdots$$

上 机 题

1. 利用两个栈 s1、s2 模拟一个队列时，如何用栈的运算来实现该队列的运算？

(1) enqueue：插入一个元素。

(2) dequeue：删除一个元素。

(3) queue_empty：判定队列为空。

2. 双端队列 deque 是一种插入和删除操作在线性表的两端进行的数据结构，试给出利用数组实现的 deque

两端的插入、删除操作，要求这4个操作的时间代价均为常数。

3. 试利用非数组变量，按下述条件各设计一个相应的算法以使队列中的元素有序：

(1) 使用两个辅助的队列。

(2) 使用一个辅助的队列。

4. 试按下述条件各设计一个算法把栈 s1 中的元素转移到栈 s2 中，并保持栈中元素原来的顺序：

(1) 使用一个辅助栈。

(2) 只使用一些辅助的非数组变量。

5. Fibonacci 序列 0,1,1,2,3,5,8,13,21,34,…，其中每个元素是前两个元素之和，可递归定义为

$$fib(n)=\begin{cases} n, & n=0,1 \\ fib(n-2)+fib(n-1), & n\geqslant 2 \end{cases}$$

试设计一个计算 $fib(n)$ 的递归过程，并利用栈来模拟递归调用，将递归过程改写成一个非递归过程。

6. 已知 Ackermann 函数定义如下：

$$Ack(m,n)=\begin{cases} n+1 & m=0 \\ Ack(m-1,1) & m\neq 0, n=0 \\ Ack(m-1,Ack(m,n-1)) & m\neq 0, n\neq 0 \end{cases}$$

(1) 写出 $Ack(2,1)$ 的计算过程。

(2) 写出计算 $Ack(m,n)$ 的非递归算法。

第4章 字 符 串

随着计算机技术的发展，诸如文本处理、生物计算中的 DNA 信息提取等非数值应用越来越广泛，这类问题大量使用字符串这种数据结构。

字符串(string)是零个或多个字符的顺序排列所组成的复合数据结构，是一种特殊的线性表，其基本组成元素是单个字符。大多数程序设计语言都支持字符串这种数据类型。

本章在阐述字符串的基本概念基础上，重点讨论在非数值处理中得到广泛应用的字符串模式匹配问题及其相关算法。

4.1 字符串的基本概念

字符串是组成元素(结点)为单个字符的线性表，简称“串”。一个字符串可以是一个单词、一个句子、一篇文章、或者一个文件的内容等。

字符串所包含的字符个数称为串的长度。长度为零的串称为空串，不包含任何字符内容。

尽管字符串是一种特殊的线性表，但其逻辑结构与线性表相比存在一定的区别：

(1) 字符串的数据对象约束为字符集。

(2) 线性表的基本操作大多以“单个元素”为操作对象，而字符串的基本操作通常以“串的整体”作为操作对象。

4.1.1 字符编码

字符串是程序中最常用的数据类型。按照计算机的用法，一个字符串就是一个字符的序列。例如：

Twinkle twinkle little star

00101001100

Hello world

2008 Beijing Olympic

都是字符串的例子。

组成字符串的字符可以是数字、文本字符或者特定的符号，只要它们存在于允许的字符集中。在进一步讨论字符串这种数据类型之前，先来认识一下组成字符串的基本单位——字符

(char)。通常,计算机只能识别0、1组成的字节,因此实际字符集的"字符"在内存中需要用"字节"来表示,此即所谓的字符编码。

在C/C++等程序设计语言中,字符类型是单字节的基本数据类型,采用ASCII(american standard code for information interchange,美国标准信息交换码)编码标准对字符进行编码,每个字符用一个字节表示,其中低7位表示字符,最高位均为0。ASCII编码集中包含128个字符,编号为第0~32号及第127号的34个字符为控制字符或通信专用字符,其余编号处于第33~126号之间的94个字符为通用字符。控制符包括LF(换行)、CR(回车)、FF(换页)、DEL(删除)、BEL(振铃)等,通信专用字符则有SOH(文头)、EOT(文尾)、ACK(确认)等;通用字符包括常用的52个大小写字母、10个数字以及若干标点符号和运算符号,其中大写字母的编号范围是65~90,小写字母的编号范围是97~122,10个数字的编号范围是48~57。

制定于20世纪60年代的ASCII编码标准没有考虑日后容纳中文、阿拉伯文等多种国际语言文字符号的统一编码。在发展过程中,不同的语言形成了各自独立的字符编码系统,这些编码系统有可能使用相同的编号来表示各自不同的字符。例如,作为东方语言代表的中文、日文和韩文,尽管均采用两个字节的编码方式,但在编码上存在冲突。每种语言都存在多种编码方式,例如中文的GB2312—80(中文简体国标码,包括6 763个汉字)、BIG5(中文繁体编码,容纳13 053个汉字),日文的S-JIS(日文系统常采用的编码)、JIS以及韩文Wansung码等。这就导致了不同语言系统之间进行文档交流非常困难,计算机看到的只是编码后的数字,而两种编码可能使用相同的数字代表两个不同的字符。判断文档涉及的编码模式需要额外的语言处理工具。

为了解决这些互通问题,达到信息共享的目的,产生了"通用文字符号编码标准"UNICODE。简单地说,UNICODE标准是一种可伸缩的编码,既可以容纳多种语言文字的大编码集(使用4个字节表示单个符号,以便容纳大量的多国文字符号),同时又可以缩减,允许用单字节表示常用的ASCII符号,用双字节表示中文符号等。目前,UNICODE为双字节编码制定了UTF8编码表(Charset UTF-8),同时还含有4字节的UTF-16,后者的扩充编码集可以容纳1 024×1 024个字符的大字符集,几乎可以容纳世界上各种各样语言的全部文字符号。这样,UNICODE可以确保在多种语言中都使用的字符具有相同的数字编码。

鉴于字符的编码并不改变字符串概念和操作的本质,本书将限于使用ASCII编码,基本不涉及多国语言混排的问题。

4.1.2 字符的编码顺序

为了便于进行字符串的比较和运算,字符编码表一般遵循约定俗成的"偏序编码规则"。这里所说的字符偏序是指根据字符的自然含义,某些字符间具有一定的次序,例如数字符号0,1,2,…,9的大小次序。为了符合其自然次序,在对某些字符进行编码时可以使其在字符集中的编码顺序和其自然次序一致。一般而言,令函数encode(x)为符号x到其对应的ASCII编码(序号)的映射函数,则有

encode('0') + 1 = encode('1'),

encode('1') + 1 = encode('2'),

…,

encode('8') + 1 = encode('9'),

即 10 个数字符号的编码是连续递增的。

对于字母符号 A,B,C,…,Z 也遵循类似的偏序规则,例如 encode('A') + 1 等于 encode('B')。类似的,a,b,…,z 也有 encode('a') + 1 等于 encode('b')。

任意两个字符 ch1 和 ch2 均可用其编码值来直接比较大小,例如 ch1 < ch2。对于字母而言,这种大小次序的含义就是按照字典编目次序 ch1 在 ch2 之前。

两个字符串如果按其构成的字符顺序比较大小,得到的结果如下:

"monday" < "sunday" < "tuesday"

"123" < "1234" < "23"

注意:这种次序仅仅是字符串的字典次序,与其在日常生活中人们对其理解的次序不同。例如,星期一在星期二之前,整数 123 比 23 要大等。

实际上,以上讨论的比较次序并不限于 ASCII 编码,也适合 GB2312 等其他语言文字在排序方面的需要。

4.1.3 字符串抽象数据类型

如上所述,字符串在逻辑结构上很简单,属于简单的线性表结构。如果线性表的元素为一个确定的字符集中按照某种确定方式进行编码的字符,则构成一个字符串。

在 C/C++ 中,用双引号括起来的一串文字即为一个字符串常量,可以初始化或赋值给某个字符串变量。例如"Hello world!"为一个字符串常量,它可以初始化一个字符串变量 s = "Hello world!"。空字符串表示为""。与空串不同," "则表示包含一个空格字符的字符串。

假设 s_1、s_2是两个字符串,其中 $s_1 = a_0a_1a_2\cdots a_{n-1}$,$s_2 = b_0b_1b_2\cdots b_{m-1}$,$0 \leqslant m \leqslant n$,且 a_i 和 b_j 均为字符集中的字符,如果存在整数 $i(0 \leqslant i \leqslant n-m)$,使得对任意 $j = 0,1,\cdots,m-1$,都有 $b_j = a_{i+j}$同时成立,则称字符串 s_2 是字符串 s_1 的子串,或称 s_1 包含 s_2。

空字符串是所有字符串的子串,任何串都是其自身的子串。如果字符串 str 的一个子串非空且不为 str 自身,则称其为 str 的一个真子串。例如,"quick"、"jump"、"fox"、"brown dog"均为字符串"The quick brown dog jumps over the lazy fox"的真子串。

常用的字符串操作包括:在字符串的末尾添加字符;把两个或更多串连接在一起;将一个串复制到另一个串中;在字符串中搜索一个给定字符或子串等。

代码 4.1 给出了字符串的一种抽象数据类型,其中定义了常用的字符串运算,包括串拼接、串复制、抽取子串等。

【代码 4.1】 字符串的抽象数据类型定义

```
class string {
private:                                        // 字符串的存储结构在具体实现时定义
    ...                                         // 字符串的数据表示,例如 char * str;
    ...                                         // 串的当前长度,例如 int size;
public:                                         // 字符串的运算集
    string();                                   // 创建一个空字符串
    string(char *s);                            // 创建一个初值为 s 的字符串
    ~string();                                  // 消除该串实例
    int length();                               // 返回串的长度
    int isEmpty();                              // 判断串是否为空串
    void clear();                               // 把串清空
    string append(const char c);                // 在串尾添加字符
    string concatenate(const char *s);          // 把串 s 连接在本串后面
    string copy(const char * s);                // 将一个串 s 复制到本串
    string insert(const char c, const int index);  // 往串中给定位置插入一个字符
    int find(const char c, const int start);    // 从位置 start 开始搜索串寻找一个给定字符
    string substr(const int s, const int len);  // 从位置 s 开始提取一个长度为 len 的子串
}
```

【代码4.1结束】

根据具体应用的需要可以增删抽象数据类型的操作。

4.2 字符串的存储结构和实现

选择字符串的存储结构时要考虑字符串的变长特点,同时需结合具体的应用分析各种存储方案的利弊,再进行合适的选择。

某些应用中的字符串长度变化非常显著,短如单词,长为文件。从统计分布来看,字符串长度分布的方差很大。对于这种情况,用静态长度的向量作为存储结构是不恰当的。字符串的变长特点是无法回避的,字符串的拼接、查找、置换和模式匹配等操作本身都涉及变长的字符串操作。这些字符串操作时间开销大,必须精心设计算法,选择恰当的串存储结构。

本节将讨论若干字符串的存储方案,重点讨论程序执行过程中的字符串变长存储问题。

4.2.1 字符串的顺序存储

对于字符串的长度分布变化不大的情况,字符串采用顺序存储比较合适。字符串的顺序表示就是把串中的字符顺序地存储在一组地址连续的存储单元中。一般的顺序存储方案使用类型

为 char 的一维定长数组，以便于编程和节省空间。

顺序存储的字符串适合访问字符串中单个字符或连续的一组字符，但进行插入和删除（增减字符）操作就不是很方便，需要移动插入或删除点后面的所有字符。另外，存储字符串的数组都是静态定长的，程序运行中一旦产生更长的字符串，就会造成数组溢出。这给编写程序和调试程序带来不便。

C/C++语言的标准字符串是采用顺序存储方案的典型代表。其特点是在程序中采用

```
char s[M];
```

的形式定义字符串变量，其中的 M 是整型常数，表示字符数组的长度。标准字符串需要在其末尾带一个结束标记'\0'来表示串的结束，因此字符串的长度不能超过最大长度 M-1。例如：

```
char  s1[12] = "Hello world" ;
char  s2[8] = "2008" ;
char  s3[6];
```

这 3 个数组 s1、s2、s3 在 C++中的存储示意如图 4.1 所示。

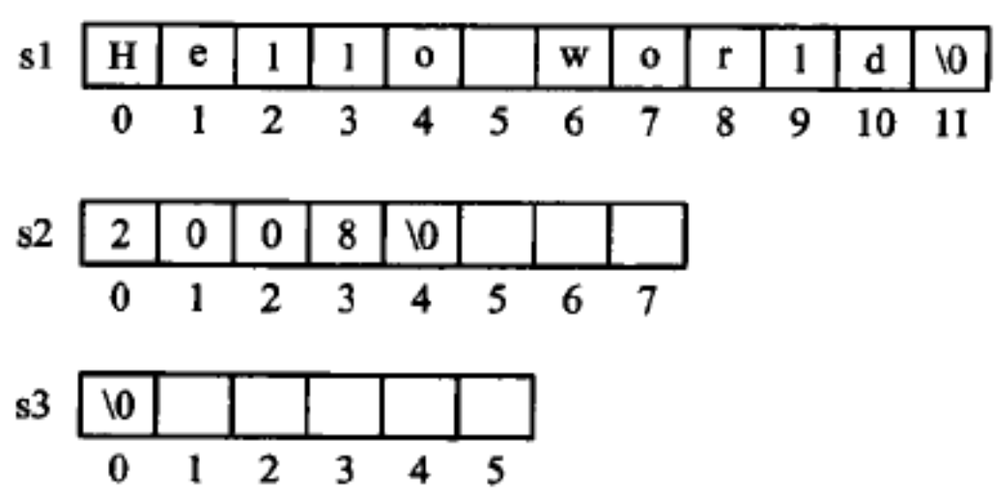


图 4.1 C++标准字符串的变量说明示意图

作为字符串结束标记，'\0'是 ASCII 码中 8 位全 0 码，又称为 NULL 符。这个 NULL 符专门用做结束标记，不作它用。对于字符串常数，C/C++也是采用这种存储方式。在说明字符串变量时，字符串可以使用字符串常数作为初值，也可以不给初值。图 4.1 中的 s3 没有给出初值，意味着它存储空字符串。

C++中的字符数组是用字符指针定义的，指向字符数组的始址。赋值语句 s1 = s2 不能被理解为字符串 s2 的内容复制到字符串 s1。这是在字符串处理中非常容易犯的错误。

标准库 <string.h> 提供了若干处理字符串的常用函数，表 4.1 列示了常用的几个。

表 4.1 标准串函数

函 数 名	函数功能说明
size_t strlen(char * s)	求字符串 s 的当前长度，不计结束符。空字符串的长度为 0
char * strcpy(char * s1, const char * s2)	将字符串 s2 复制到 s1，并返回一个指针，指向 s1 的开始
char * strcat(char * s1, const char * s2)	将字符串 s2 拼接到 s1 尾部

续表

函 数 名	函数功能说明
int strcmp(const char * s1, const char * s2)	比较字符串 s2 和 s1,若 s1 和 s2 全同则返回 0;s1 大于 s2 则返回正数; s1 小于 s2 时返回负数
char * strchr(char * s, char c)	将返回指针定位到字符串 s 中第一次出现字符 c 的位置。若字符串 s 中没有字符 c,则返回空指针
char * strrchr(char * s, char c)	从字符串的尾部逆向定位字符,将指针定位到串尾部第一次出现字符 c 的位置,没有则返回空指针

例如,对于图 4.1 中的字符串 s1 调用定位函数 strchr(s1,'o')和 strrchr(s1,'o')将分别得到串 s1 中字符'o'的第一次和倒数第一次出现位置,如图 4.2 所示。

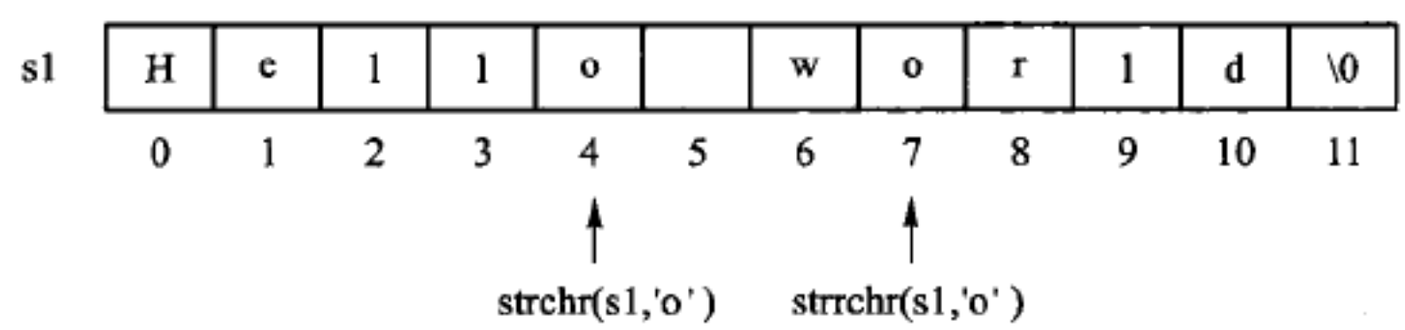


图 4.2 在字符串 s1 中寻找并定位给定字符'o'的位置

此外,输入/输出的处理也是标准串函数库中的主要组成部分。例如;

```
cin  >>  s1;        // 从标准输入中读取字符串到串变量 s1 中
cout  <<  s1;       // 把字符串变量 s1 的内容输出到标准输出
```

字符串的顺序存储方式简单易实现,C++的标准串及其相关的标准函数提供了若干实用的函数以方便字符串的处理,但终归难以避免其静态定长的局限。而在实际应用中,大多的字符串变量具有动态变化的长度。为此,下面介绍能够适应动态变化的字符串存储结构。

4.2.2 字符串类 class String 的存储结构

字符串的长度动态变化,且不适合采用链式的变长结构,因为每一个链指针比一个字符所占的存储空间还大。此外,由于字符串处理的应用极为广泛,这就要求为它提供大量的方法(函数和运算符),设计恰当的存储结构,以方便程序设计,减少编程错误,提高应用程序的效率。本节将简单介绍支持字符串动态变化的 C++ String 类。

在 C++语言中,类 String 是一个面向对象的串,它封装了串的属性,并提供了一系列的方法来访问这些属性。

代码 4.2 中的 basic_string 类定义了一个类模板,根据应用的需要确定串中的字符类型。字符类型 charT 可以是宽或窄字符。最常用的字符是 ASCII 集合中的字符,在处理一些 ASCII 码

不易表示的其他语言时，也可以使用包含在另一个字符集合中的其他字符。其中，第二个参数和第三个参数各有默认值 char _ traits 和 allocator，char _ traits 和 allocator 是它们自己的类模板，模板参数为 charT。

与字符类型相关的是字符特性（character traits），被封装在 char _ traits 类中。该类说明了类型和用来控制字符类型的函数，函数定义了有关字符的比较、等价、分配等。类 char _ traits 的实际定义完全依赖于串所使用的字符类型，即每个字符类型都有其自身的类定义。这些函数和类型可以被其他类使用，如 string 类和 iostream 类。比较两个串是否相等时，是通过一个串的字符与另一个的字符做比较，调用了对那些特殊字符类型定义的类 char _ traits 的比较操作。因此，字符特性对于使用的具体字符类型必须进行具体化。

Allocator 类封装了有关使用内存模式特性的信息，栈和堆、段内存模型都将影响指针的大小和行为。

【代码 4.2】 String 类模板定义。

```
// basic _ string 模板及其功能都定义在名字空间 std 中，由 < string > 给出
namespace std {
    template < class charT,
        class traits = char _ traits < charT > ,
        class Allocator = allocator < charT > >
    class basic _ string {
        public:
            ...
    }
}
typedef basic _ string < char > String;
```

【代码 4.2 结束】

采用“String s;”的形式来说明 String 类的变量。String 类将采用一种动态变长的存储结构，字符串不再以字符数组（char s[M]）的形式出现，摆脱了静态定长数组的束缚。String 类封装了串的属性并提供了一系列允许访问这些属性的方法。属性或类的数据成员包括：

- 字符序列。
- 字符序列的大小或长度。
- 序列中字符的类型。
- 字符的特点。
- 一个字符的大小。
- 分配运算符（allocator）。
- 迭代子（iterators）。

方法包括：

- 查找。

- 赋值。
- 拼接。

表4.2列出了String类的部分成员函数和运算符，并概述了它们的功能。从表中可以看到，有两个与字符串长度有关的方法：length()和size()。表面上，似乎length()返回字符序列的长度，size()返回字符串对象的大小。但实际上，二者都返回序列中字符的数目。max_size()返回字符串对象中可以放置的最大字符数，其计算是依赖于内存模式的。具体的内容可以参见C++的手册或相关书籍。

表4.2 C++串类部分操作列表

操作类别	方法	描述
子字符串	substr()	返回一个字符串的子串
复制/交换	swap()	交换两个字符串的内容
	copy()	将一个字符串复制到另一个字符串中
赋值	assign()	把一个字符串、一个字符、一个子字符串赋值给另一个字符串中
	=	把一个字符串或一个字符赋值给另一个字符串中
插入/追加	insert()	在给定位置插入一个字符、多个字符或字符串
	+=	将一个字符或字符串追加到另一个字符串后
	append()	将一个或多个字符或串追加在另一个字符串后
拼接	+	通过将一个字符串放置在另一个字符串后面来构建新串
查询	find()	找到并返回一个子序列的开始位置
替换/清除	replace()	替换一个指定字符或一个字符串的字串
	clear()	清除字符串中的所有字符
统计	size()	返回字符串中字符的数目
	length()	返回size()
	max_size()	返回字符串允许的最大长度

本节通过几个例子来简单讨论String类的存储结构和动态管理字符串存储空间的特点。通过String常用的操作：创建字符串String::String(char *s)、赋值运算符String String::operator =(String& s)等操作来说明字符串存储空间是如何进行动态管理的。

String类对字符串的存储依然采用字符指针char *str的方式，同时保存当前字符串的长度。在构造一个字符串时根据串的初值分配相应大小的数组空间。String类的存储结构声明如下：

```
private:                    // 具体实现的字符串存储结构
    char *str;              // 字符串的数据表示
    int size;               // 串的当前长度
```

例如，下面的变量说明语句：

```
String s1 = "Hello";
```

隐含地调用构造函数 String::String(char * s),其中参数 s 所对应的实参为 s1 = "Hello"。此时,构造函数在动态存储区开辟一个长度为 6 的字符数组,并将初始值"Hello"顺序存入,且在末尾添加结束符,如图 4.3 所示。

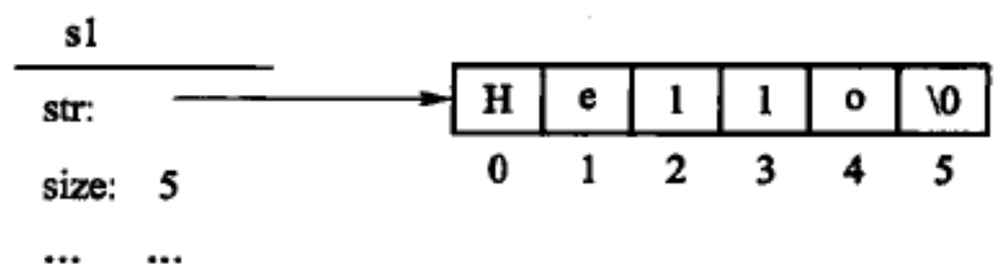


图 4.3 创建字符串的示意图

如果在上面例子的基础上,定义了另一个字符串变量:

```
String s2 = "Hello world";
```

并通过赋值语句:"s1 = s2"把字符串 s2 的内容赋值给串 s1。此时,构造函数先创建字符串实例 s2,其初值为"Hello world",长度为 11,然后在执行赋值时把实参串 s2 的值赋给 s1。为了要把 s2 这个较长的字符串复制到 s1 中,必须释放 s1 的原有空间(delete [ ] s1. str)(图 4.4 中以灰色区域格来表示),然后在动态存储区开辟新的存储数组,把 s1 的内容复制到新数组中。即赋值语句之前 s1 的 str 指针指向数组存储空间"Hello",执行赋值语句之后,该数组空间被释放,str 指向一个新的数组空间并被赋值为"Hello world "。这样,String 类根据字符串的长度,动态地调整分配给串的空间,实现了对串的变长管理。

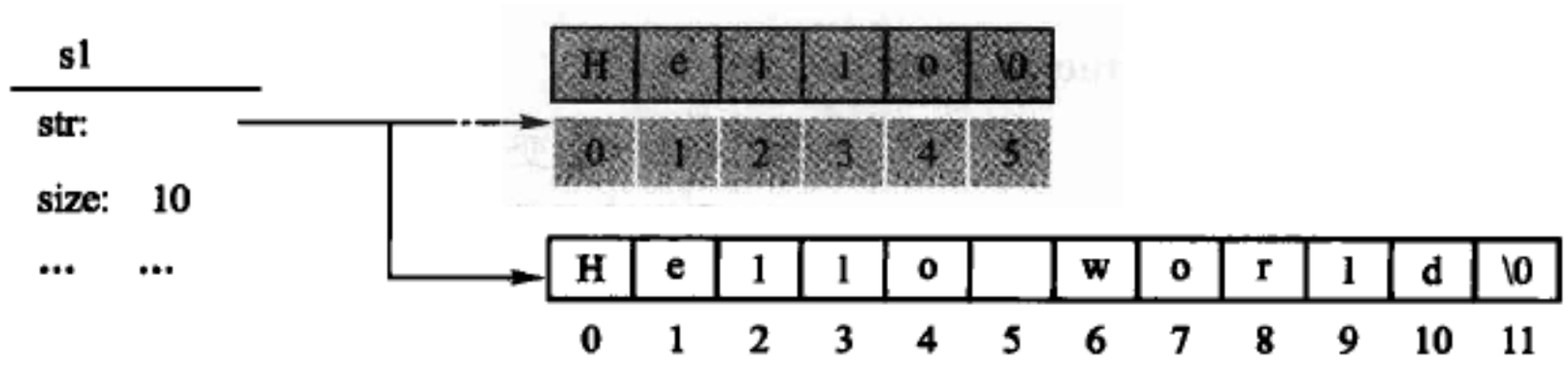


图 4.4 赋值运算示意图

4.2.3 字符串运算的实现

确定了存储结构后,字符串运算的实现相对就简单了。本节将介绍几个重要的运算。

1. C++标准串运算的实现

C++标准字符串函数库所提供的字符串操作的实现比较简单,在此仅以字符串比较运算为例进行说明。更为详尽的库函数可参看 C/C++的标准库手册。

【算法 4.3】 C++标准串比较运算的实现。

```
int strcmp(const char * s1, const char * s2) {
    int i = 0;
    while (s2[i] != '\0' || s1[i] != '\0') {
```

```
        if (s1[i] > s2[i])
            return 1;
        else if (s1[i] < s2[i])
            return -1;
        i++;
    }
    if (s1[i] =='\0'&& s2[i] !='\0')
        return -1;
    else if s2[i] =='\0'&& s1[i] !='\0')
        return 1;
    return 0;
}
```

【算法4.3结束】

需要注意的是,这个函数的返回值(0,正,负)对应于(等于,大于,小于),这与C/C++语言中通常的大小比较习惯(0和非0)不一致。

2. String串运算的实现

为了解C++ String类如何实施字符串存储空间的动态管理,本节以构造函数String::String(char *s)和抽取子串String String::Substr(int pos, int n)为例来说明。需要引入<string.h>库函数。

【算法4.4】 构造函数constructor。

```
String::String(char *s) {                    // 参数s为变长串
    size = strlen(s);                        // 用标准函数strlen(s)求串长度
    str = new char [size + 1];               // 开辟向量空间,结束符'\0'也算一个字符,size + 1
    assert(str !='\0');                      // 当开辟动态区域不成功时,运行异常退出
    strcpy(str, s);                          // 将初值s复制到str所指的存储空间
}
```

【算法4.4结束】

【算法4.5】 抽取子串函数

```
String String::Substr(int pos, int n) {      // 从本串的pos开始抽取长度为n的子串
    int i;
    int left = size - pos;                   // 自下标pos向右计数到串尾查看剩余长度
    String tmp;
    char *p, *q;
                                             // 以下7行语句,全是为取出子字符串做准备工作
    if (pos >= size)                         // 若下标pos值超过本串实际字符串长,则返回空串
        return NULL;
    if (n > left)                            // 若n超过自pos以右的子字符串长度,则把n变小
```

```
        n = left;
delete [ ] tmp. str;                    // 释放原来的存储空间
tmp. str = new char[ n + 1 ];
assert( tmp. str ! = NULL) ;            // 若开辟动态存储空间失败,则退出
p = tmp. str;                           // p 指向暂无内容的字符串
q = &str[ pos ];                        // q 指向本实例串 str 数组下标 pos 处
for (i = 0; i < n; i ++ )
        * p ++ = * q ++ ;               // 将 q 所指的内容赋值给 p,并同时后移
* p ='\0';                              // 循环结束后,让 temp. str 的结尾为'\0'
tmp. size = n;
return tmp;
    }
```

【算法 4.5 结束】

4.3 字符串的模式匹配

字符串模式匹配是一种常用的运算。所谓模式匹配(pattern matching),简单地说就是在文本(text)中寻找一个给定的模式(pattern),通常文本都很大,而模式则比较短小。典型的例子包括文本编辑和 DNA 分析。例如,在编辑文本时人们经常会使用"替换"命令来对文中的某个字符串或语句进行替换,此时便需先找到要被替换的内容,再进行修改或替换。这个要查找的内容就是模式,在正文中查找被替换内容的过程就是一个字符串模式匹配的过程。在文本编辑器中,模式通常为一个单词,长度在 10 个字符左右,而文本的长度则从几百字到上百万字不等。在生物信息中,DNA 信息一般由 A、C、G、T 这 4 个符号组成,基因一般也就是几百个字符的长度,而人类染色体的长度却有 30 亿之多。显而易见,这些应用对匹配算法的效率要求很高。

模式匹配有精确匹配和近似匹配两类。

1. 精确匹配(exact matching)

如果在目标 T 中至少一处存在模式 P,则称匹配成功,否则即使目标与模式只有一个字符不同也不能称为匹配成功,即匹配失败。例如,从文本 T 中搜索某个给定单词的匹配即属精确匹配的范畴。想要找的词可以是单选的,例如"set" ;也可以是多选的,例如模式"s? t" 表示任何以字符 s 打头、以字符 t 结尾、中间夹一个字符的串都可与之匹配,诸如"sat"、"set"、"sit"都可以,模式中的符号"?"称为通配符,可匹配任意单个字符。复杂的模式可以采用正则表达式来表示,但本节要讨论的模式不涉及通配符和正则表达式。

2. 近似匹配(approximate matching)

如果模式 P 与目标 T(或其子串)存在某种程度的相似,则认为匹配成功。常用的衡量字符串相似度的方法是根据一个串转换成另一个串所需的基本操作数目来确定的。基本操作由字符

串的插入、删除和替换来组成。限于篇幅，本节将不涉及近似匹配的内容。

本节所讨论的模式匹配只限于精确匹配中的单选情况。给定一个由字符或符号组成的字符串目标对象 T 和一个字符串模式 P，模式匹配的目的是在目标字符串 T 中搜索与模式 P 完全相同的子串，并返回 T 和 P 匹配的第一个子串（简称为"配串"）的首字符位置。

如果采用如下约定：对于目标文本 T 和模式 P，用 T_j 或 P_j 表示其在位置 j 上的字符，$T(i\cdots j)$ 或 $P(i\cdots j)$ 表示其中始于位置 i 终于位置 j 的一个子串。P 和 T 的第一个字符都从位置 0 开始。目标 T 的长度用 $|T|$ 表示，目标 P 的长度用 $|P|$ 表示。另外，a^n 表示由 n 个字符 a 组成的字符串。那么，模式匹配问题可归结为：

为使模式 P 与目标 T 匹配，在目标 T 中必须存在一个 j

$$T:\ T_0\ T_1\cdots\quad T_j\ T_{j+1}\ T_{j+2}\quad\cdots\quad T_{j+m-2}T_{j+m-1}\cdots T_{n-1}$$
$$P:\qquad\qquad P_0\ P_1\ P_2\quad\cdots\qquad P_{m-2}\ P_{m-1}$$

使得下面的条件满足

$$P_0\ P_1\ P_2\cdots\ P_{m-2}\ P_{m-1}=T_j\ T_{j+1}\ T_{j+2}\cdots\ T_{j+m-2}T_{j+m-1}$$

否则，则称模式匹配失败。

4.3.1 朴素的模式匹配算法

模式匹配的一个简单方法就是把模式 P 的字符依次与目标 T 的相应字符进行比较。首先，从首字符开始，依次将两个字符串对应位置上的字符进行比较，如图 4.5 的步骤(1)所示。当某次比较失败（称为一次"失配"）时，则把模式 P 对于 T 向右移动一个字符位置，重新开始下一趟匹配，如图 4.5 步骤(2)所示。如此不断重复，直到某趟配串成功返回，如图 4.5 的步骤(3)所示；或者比较到目标串的结束也没有出现"配串"的情况，则匹配失败，如图 4.5 的步骤(4)所示。

图 4.5 所示的这种模式匹配方法称为朴素的匹配算法，其实现代码在算法 4.6 中给出。

函数 int NaiveStrMatching(String T, String P)从目标 T 的首位置开始模式匹配，用模式 P 匹配 T，寻找首个 P 子串并返回其下标位置。若整个匹配过程失败（模板 P 移动到目标 T 尾部位置），则返回负值。

【算法 4.6】 朴素的字符串模式匹配算法

```
#include "String. h"
#include <assert. h>
int NaiveStrMatching (const String & T, const String & P) {
    int i = 0;                          // 模式的下标变量
    int j = 0;                          // 目标的下标变量
    int pLen = P. length( );            // 模式的长度
    int tLen = T. length( );            // 目标的长度
```

```
    if (tLen < pLen)                        // 如果目标比模式短,匹配无法成功
        return (-1);
    while(i < pLen && j < tLen) {           // 反复比较对应字符来开始匹配
        if (T[j] == P[i])
            i++, j++;
        else {
            j = j - i + 1;
            i = 0;
        }
    }
    if (i >= pLen)
            return (j - pLen + 1);
        else return (-1);
}
```

【算法 4.6 结束】

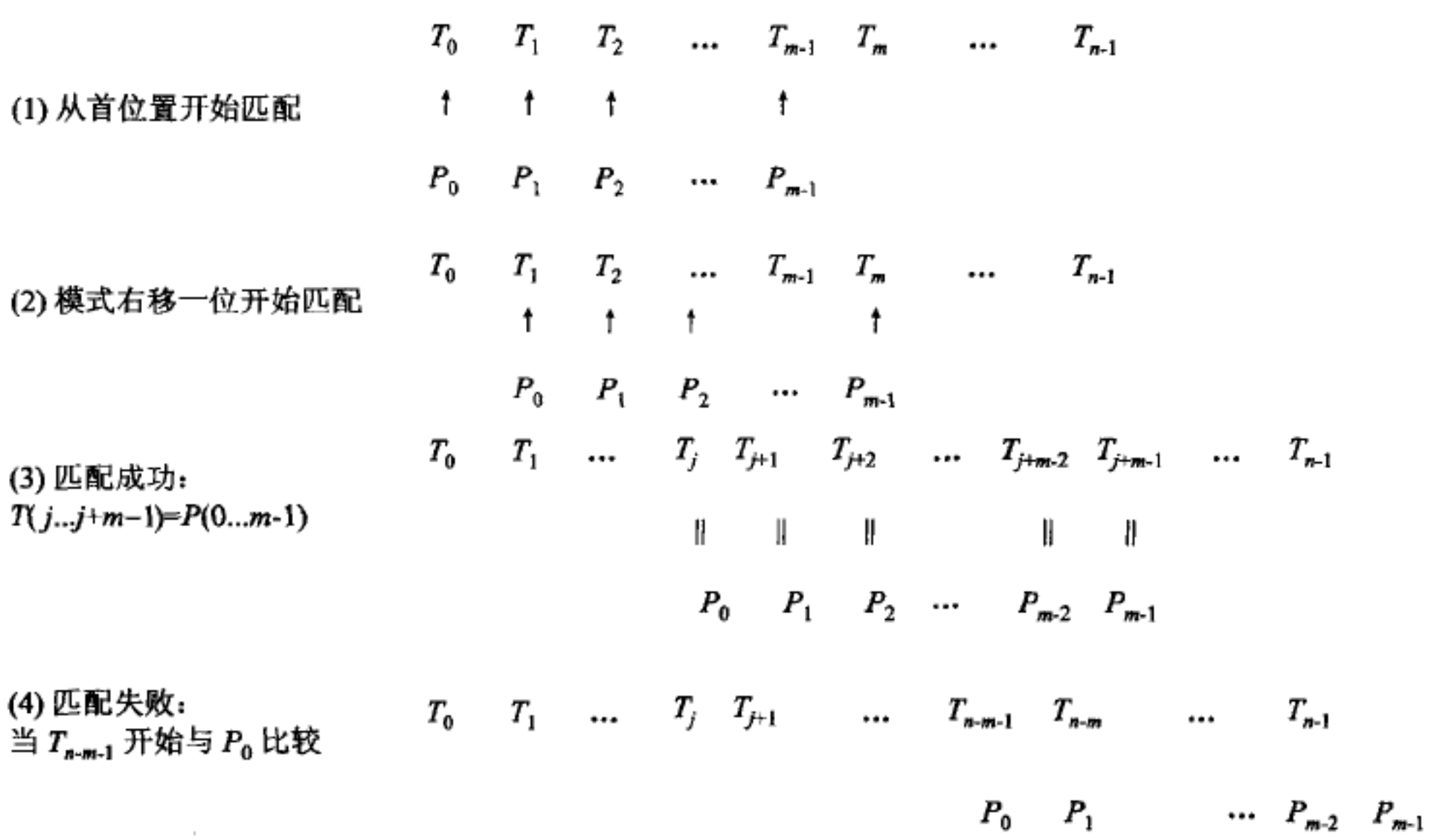


图 4.5 朴素的匹配方法示意

图 4.6 是一个采用朴素匹配算法的例子,其中目标为字符串"abacaabaccabacabaa",模式为"abacab",匹配过程中若模式的字符与当前目标字符不等则用下画线及黑体标示。匹配算法伊始,第一趟匹配中,目标中的字符 'a'与模式中的字符 'b'不匹配,此时把模式右移到下一个字符开始第二趟匹配,发现对应的字符 'b'与 'a'也不同,继续右移模式重新开始扫描,如此直到第 11 趟匹配时在目标的子串"abaab"处匹配,此时返回该子串在目标中的首位置 10,作为模式匹配成

功的结果。假如模式为"aaaabb",则会发现匹配到目标串的结束处也没有找到配串,此时则匹配失败,返回一个负数。

算法4.6的思想很简单,下面分情况讨论一下时间效率问题。

朴素匹配算法的最佳情况如图4.7所示,目标首位置开始的子串便是所要找的配串,只要通过|P|次的比较即可完成匹配,最佳情况的时间代价为 $O(|P|)$,与模式的长度成正比。

```
   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
T  a b a c a a b a c c a b a c a b a a
1  a b a c a b
2    a b a c a b
3      a b a c a b
4        a b a c a b
5          a b a c a b
6            a b a c a b
7              a b a c a b
8                a b a c a b
9                  a b a c a b
10                   a b a c a b
11                     a b a c a b
```

图4.6 朴素匹配的示例

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
a a a a a a a a a a a a a a a a a a a b
a a a a a
```

图4.7 朴素匹配的最佳情况示例

如果最终匹配失败,没有找到配串,其最佳的情况则如图4.8所示,每次都是在模式的首字符比较时不等,就把模式右移一位,直到在目标的 $|T|-|P|$ 位置处依然与模式的首字符不匹配,则宣告失败,整个过程中至少比较了 $(|T|-|P|+1)$ 次,所以其时间代价为 $O(|T|-|P|+1)$。

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
a a a a a a a a a a a a a a a a a a a b
h h h h b
  h h h h b
    h h h h b
      h h h h b
        ...
          h h h h b
```

图4.8 匹配失败的最佳情况示例

朴素匹配最差的情况需要 $O(|T||P|)$ 的代价,例如目标串 T 形如 a^n,而模式 P 则形如 $a^{m-1}b$,每趟都是模式的最后一个字符处不匹配,即每趟都需要进行 $|P|$ 次比较,再把模式右移一位,再次从模式头比较,最多需要 $(|T|-|P|+1)$ 趟,因此总共需要比较 $|P|(|T|-|P|+1)$ 次。图4.9所示为这种最差情况的一个示例。此类情况很少出现在文本中,但却经常出现在DNA信息和图像信息中。

朴素匹配算法的平均代价同时依赖于目标和模式中字符的分布概率。例如,假设在一个字符串中只允许使用2种字符,那么使用任一字符的概率为1/2。对于第 j 趟扫描来说,仅比较一次的概率为1/2,比较两次的概率则为 $1/2\times1/2=1/4$。对于长度为 m 的模式而言,比较 m 次的

概率为 $1/2 \times \cdots \times 1/2 = 2^{-m}$，因此平均情况下，第 j 趟扫描的比较次数为 $\sum_{k=1}^{|P|} \frac{k}{2^k} < 2$。因此平均比较次数为 $2(|T| - |P| + 1) < 2|T|$ 次。

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
a a a a a a a a a a a a a a a a a a a b
a a a a b
  a a a a b
    a a a a b
      a a a a b
                ...
                          a a a a b
```

图 4.9 朴素匹配最差情况示例

如果运用马尔科夫链(Markov Chain)的有关理论，可以估算一个更好的比较次数：$2^{|P|+1} - 2$。在通用的情况下，如果字符串中允许使用的字符数目为 $|A|$，则平均比较次数为 $(|A|^{|P|+1} - |A|) / (|A| - 1)$。

由此可见，尽管朴素算法的思想简单易懂，但其效率较低。从前述的几个示例中可以看出，一旦某趟匹配中发生失配，无论模式的具体情况如何，都采用模式右移一位开始下一趟的匹配。这可能会导致很多冗余的比较，因为目标 T 中的字符会多次与模式中的字符进行比较，造成目标的回溯。

例如，图 4.6 所示的模式匹配中，模式 P = "abacab" 与目标串 T = "abacaabaccabacabaa" 的第 1 趟比较中，发现 $P_5 \neq T_5$ 时，把模式右移一位，开始第 2 趟匹配，使得 P_0 与 T_1 进行比较。其实之前的匹配中，T_1 已与 P_1 比较过而且相等。由于 P_1 与 P_0 不同，因此可知 P_0 与 T_1 肯定不等，根本用不着重新比较。换言之，利用模式本身的构成特征以及上趟匹配的比较结果即可得知 P_0 与 T_1 肯定不等。

在图 4.6 所示的例子中，第 1 趟匹配后，如果模式仅右移一位，下趟匹配中一定有冗余的比较，诸如 P_0 与 T_1 的比较就不必要。一般来说，右移的位数越多效率就越好，但前提是要保证不能丢失 P 和 T 的任何可匹配子串。

Knuth、Morris、Pratt 等人发现，其实每次右移的位数存在且与目标串无关，仅依赖于模式本身，因此他们对朴素的模式匹配算法进行了改进。改进后的算法(简称 KMP 算法)的基本思想为：预先处理模式本身，分析其字符分布状况，并为模式中的每一个字符计算失配时应该右移的位数。此即为所谓的字符串的特征向量。

4.3.2 字符串的特征向量

在介绍如何求字符串的特征向量之前，再回顾一下朴素匹配算法，如图 4.10 所示。在 P 与

目标 T 的匹配过程中，当某次比较出现 $P_i \neq T_j$ 时，意味着在此前的匹配历史中有下述匹配的子串满足：

$$P(0\cdots i-1)=T(j-i\cdots j-1) \tag{1}$$

即在 $P_i \neq T_j$ 之前，P 从首字符开始的子串 $P(0\cdots i-1)$ 已与目标 T 中的子串 $T(j-i\cdots j-1)$ 相等，如果

$$P(0\cdots i-2)=P(1\cdots i-1) \tag{2}$$

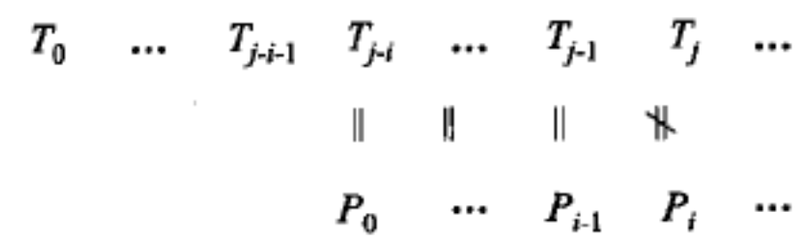


图 4.10 $P_i \neq T_j$ 时的状态

成立，即表示模式右移一位后，$P(0\cdots i-2)$ 与 $T(j-i+1\cdots j-1)$ 必相等，因为之前的(1)式可以推导出 $P(1\cdots i-1)=T(j-i+1\cdots j-1)$，带入(2)式即可。所以，此时直接比较 P_{i-1} 与 T_j 即可。由此可知，利用之前的比较结果减少了在目标串上的回溯。同理，如果(2)不满足，可以继续查看 $P(0\cdots i-3)$ 与 $P(2\cdots i-1)$ 是否相等，若相等，则把模式右移 2 位，因为此时 $P(0\cdots i-3)$ 与 $T(j-i+2\cdots j-1)$ 必相同。依次类推，必可找到某个 k 值，使得

$$P(0\cdots i-k-1)=P(k\cdots i-1) \tag{3}$$

成立，此时把模式右移 k 位来开始下一趟匹配，既不会丢失配串，又不致于造成目标的回溯。

由此可见，根据模式本身的特征，就可以确定在某趟发生失配时模式应该右移的位数，在失配的 T_j 处开始下趟匹配。这样既保证不丢失配串，又不需要在目标串回溯，从而消除冗余的比较。

通常，把 P 的前 k 个字符组成的子串 $P(0\cdots k-1)$ 称为 P 的前缀子串(也称"首串")，把 P 在 i 之前的 k 个字符组成的子串 $P(i-k\cdots i-1)$ 称为 i 位置的后缀子串(也称"尾串")。对于模式位置 i 上的字符 P_i 而言，一旦与目标 T_j 失配时，模式的下标从位置 i 移动到位置 k，P_k 直接与 T_j 比较，而跳过了 $P(0\cdots k-1)$。如果把目标 T_j 看做是不动的，则模式向右滑动了 $i-k$ 位。把对应于模式中每个字符 P_i 的这个 k 值称其为特征数 n_i，即 $P(0\cdots i-1)$ 中最大相同前缀子串和后缀子串。所有的特征数组成了模式的一个特征向量，表示了模式 P 的字符分布特征。

下面的公式给出了模式 P 的特征向量 next 的计算方法：

$$\text{next}[i]=\begin{cases}-1, & \text{对于 } i=0\\ \max\{k:0<k<i \quad \&\& \quad P(0\cdots k-1)=P(i-k\cdots i-1)\}, & \text{如果 } k \text{ 存在}\\ 0, & \text{否则}\end{cases}$$

由于计算 next[i]时充分利用了位置 i 之前的各个字符的 next 值，若用 n_i 表示 next[i]，特征数 $n_i(-1\leqslant n_i<i)$ 可以采用递归的方式来定义：

(1) $n_0=-1$；对于 $i>0$ 的 n_i，假定已知前一位置的特征数 $n_{i-1}=k$。

(2) 如果 $i>0$ 且 $P_i=P_k$ 则 $n_i=k+1$。

(3) 当 $P_i \neq P_k$ 且 $k \neq 0$ 时，则令 $k=n_k$，循环此步直到条件不满足(变为 $P_i=P_k$ 或 $k=0$)。

(4) 当 $P_i \neq P_k$ 且 $k=0$ 时，则 $n_i=0$。

根据定义，可以很容易地计算给定模式的特征向量。例如，字符串"abcdaabcab"的 next 数组为{-1, 0, 0, 0, 0, 1, 1, 2, 3, 1}。

这样，求出的 next 数组还可进一步优化。例如，若求出的 next[i]$=k$，当匹配时发现 $P_i \neq T_j$

时，按照特征向量的意义，此时该把模式右移 $i-k$ 位，即用 P_k 与 T_j 进行比较。如果 $P_i \neq P_k$，则 T_j 也不同于 P_k，还需再右移，用 $P_{next[k]}$ 来与 T_j 进行比较。这样，模式串"abcdaabcab"可以优化的下标位置为4、6、7、9，优化后的 next = {-1, 0, 0, 0, -1, 1, 0, 0, 3, 0}。

算法4.7是计算优化的字符串特征向量的一个具体实现。首先设置 next[0] = -1，在计算 next[i]时充分利用位置 i 之前各个字符的 next 值。外层循环对每个字符进行一次，每次进入内层循环计算 next[i]时，若上一轮 i'(i' = i-1)分析所得到的最大相同前缀子串 P(0…k'-1)和后缀子串 P(i'-k'…i'-1)即 P(i-k'-1…i-2)的长度为 k'，如果 P[i'](即 P[i-1]) = P[k']，则新的首尾子串长度可以增加1，即 i'和 k'可以同时增1，把新的 k 值(k'+1)赋给 next[i]。如果 P[i'] ≠ P[k']，则 i 位置的首尾子串不能直接利用 i-1 位置。此时，把 P 当作目标，而 P(0…k')当作模式，在 P[k']失配可以用 next[k']来与目标进行匹配，即令 k" = next[k']，再判断 P[i'] 和 P[k"]是否相等，直到找到合适的长度为 k 的首尾子串。如果 P[i'] = P[k]，内层循环结束，i 和 k 同时增1。如果 P[i]≠P[k]，则 next[i] = k；否则，next[i]需要优化为 next[k]。

【算法4.7】 计算字符串特征向量(优化版)。

```
int *findNext(String P) {
    int i = 0;
    int k = -1;
    int m = P.length();                 // m 为字符串 P 的长度
    assert(m > 0);                      // 若 m = 0，退出
    int *next = new int[m];             // 动态存储区开辟整数数组
    assert(next != 0);                  // 若开辟存储区域失败，退出
    next[0] = -1;
    while (i < m) {                     // 计算 i = 1…m-1 的 next 值
        while (k >= 0 && P[i] != P[k])      // 求最大首尾子串
            k = next[k];
        i++;
        k++;
        if (i == m) break;
        if (P[i] == P[k])
            next[i] = next[k];          //  P[i]和 P[k]相等，优化
        else next[i] = k;               // 不需要优化，就是位置 i 的首尾子串长度
    }
    return next;
}
```

【算法4.7结束】

图4.11给出了按照优化后的算法4.7求解字符串"abcdaabcab"的 next 数组的过程。根据计算情况顺序产生各行，第1行表示了计算开始前的初值，i = 0, k = -1, next 数组中只有首元素赋了-1，最后一行则给出了最终的计算结果：next = {-1, 0, 0, 0, -1, 1, 0, 0, 3, 0}。每

行中，第1列的箭头表示此时在外层循环，没有箭头意味着在内层循环中；其他各列分别记录了在求值过程中的 i 和 k 的值、next 数组的变化情况，模式中的下画线表示了当前状态下需要比较的两个字符。

	i	k	next[]	P
->	0	-1	{-1}	abcdaabcab
->	1	0	{-1,0}	abcdaabcab
		-1		
->	2	0	{-1,0,0}	abcdaabcab
		-1		
->	3	0	{-1,0,0,0}	abcdaabcab
		-1		
->	4	0	{-1,0,0,0,-1}	abcdaabcab
->	5	1	{-1,0,0,0,-1,1}	abcdaabcab
		0		
->	6	1	{-1,0,0,0,-1,1,0}	abcdaabcab
->	7	2	{-1,0,0,0,-1,1,0,0}	abcdaabcab
->	8	3	{-1,0,0,0,-1,1,0,0,3}	abcdaabcab
		0		
->	9	1	{-1,0,0,0,-1,1,0,0,3,0}	abcdaabcab

图 4.11 计算 next 数组示例

4.3.3 KMP 模式匹配算法

求得了模式的特征向量之后，基于特征分析的快速模式匹配算法（即 KMP 模式匹配算法）与算法 4.6 给出的朴素匹配方法很类似，只是在每趟匹配过程中如果发现某次失配时，不再单纯地把模式右移一位，而是根据当前字符的特征数来决定模式右移的位数。其具体实现方法可参见算法 4.8。

算法 4.8 的复杂性主要体现在 while 循环语句中。由于 j 只增不减的特性，因此循环体中的 j++；语句的执行次数最多为|T|次，所以与此在同一语句中的运算 i++ 也不会超过|T|次。i 的初值为 0，其间能使之减少的语句只有 i=N[i]。因为 N[i]<i，所以每执行一次该语句 i 值至少减 1，但是一旦使 i=-1，则下一步 if 语句的条件必成立，进而执行 i++，j++；，故循环体中 i=N[i] 的执行次数不会超过 i++，j++；语句的执行次数加 1。因此，整个循环体的执行次数至多为 2|T|+1 次，即其时间代价与目标串的长度成线性关系。

【算法 4.8】 KMP 模式匹配算法的实现。

```
int KMPStrMatching(const String & T, const String & P, int * N) {
    int i=0;                                  // 模式的下标变量
    int j=0;                                  // 目标的下标变量
    int pLen=P. length ( );                   // 模式的长度
    int tLen=T. length( );                    // 目标的长度
    if (tLen<pLen)                            // 如果目标比模式短，匹配无法成功
```

```
        return ( -1);
    while(i < pLen && j < tLen){                    // 反复比较对应字符来开始匹配
        if(i == -1 || T[j] == P[i])
            i++, j++;
        else i = N[i];
    }
    if (i >= pLen)
        return (j - pLen + 1);
    else return ( -1);
}
```

【算法 4.8 结束】

由于计算 next 数组的算法本身采用的也是 KMP 模式匹配，因此其时间代价也与模式长度成正比，为 $O(|P|)$。所以，整个 KMP 匹配的时间代价为 $O(|P|+|T|)$。在模式长度远小于目标长度时，基本上为 $O(|T|)$。此外，对于同一个模式 P，由于特征向量是预先计算的，其计算 next 数组的结果可以在 KMP 算法中被多次使用。

图 4.12 是对 KMP 快速匹配算法的一个详细示例。其中，模式为之前在求解 next 数组时所采用的"abcdaabcab"，目标串为"abcddabcababcdaabcababcdaabcabaa"。按精简版的特征向量求解，得到的模式特征值为{-1, 0, 0, 0, -1, 1, 0, 0, 3, 0}。在匹配的第 1 趟比较中，比较了 4 次之后，第 5 次比较时发现 $P_4 \neq T_4$，此时，由于 P_4 的特征值为 -1，因此把模式右移 4 - (-1) = 5 位，开始第 2 趟匹配，使得 P_0 与 T_5 进行比较（此时，由于特征值为 -1，所以目标也右移 1 位），在进行到 9 次比较时发现 $P_3 \neq T_8$，此时再根据 P_3 的特征值 0，把模式右移 3 - 0 = 3 位，开始第 3 趟匹配，在进行到 12 次比较时发现 $P_2 \neq T_{10}$，再根据 P_2 的特征值 0，把模式右移 2 - 0 = 2 位，开始第 4 趟匹配，这样在比较了 22 次之后完成匹配。

```
a b c d d a b c a b a b c d a a b c a b a b c d a a b c a b a
a b c d a a b c a b
  × i=4;j=4;j=N[j]=-1;
    a b c d a a b c a b
            × i=8;j=3;j=N[j]=0;
            a b c d a a b c a b
                ×
                a b c d a a b c a b 匹配成功,i=19,j=10
```

图 4.12 KMP 匹配示例

本 章 小 结

字符串是由零个或多个字符顺序排列组成的有限序列。它是一种特殊的线性表，其特殊性

主要体现在组成表的每个元素均为一个字符以及与此相应的一些特殊操作。一个字符串中所包含的字符的个数为串的长度，长度为零的字符串称为空字符串。主字符串 S_1 中若干个连续字符组成的子序列 S_2 被称为 S_1 的子串，而 S_1 被称为 S_2 的主串。

选择字符串的存储结构时要考虑字符串的变长特点。

顺序存储使用类型为 char 的一维定长数组，访问字符串中的单个字符或连续的一组字符比较容易，但进行插入和删除（增减字符）操作就不太方便，因为需要移动相关的字符。为了更方便地处理字符串，避免静态定长字符串数组的问题，有些字符串类的存储结构和实现方案提供了动态的字符串存储空间管理。

常用的字符串运算包括字符串的复制、求长度、比较、拼接、抽子串、寻找字符等。根据存储结构的不同，这些运算的实现也有所不同。

模式匹配是一个比较复杂的串操作，是子串（模式）在主串（目标串）中的定位操作。常用的模式匹配算法有朴素的原始匹配算法和经过优化改进的无回溯算法。

朴素的模式匹配比较直观，易于理解，但由于回溯而使复杂度提高，其时间代价与目标串长度和模式串长度的乘积成正比。

无回溯的模式匹配中最具代表性的是 KMP 算法。它对模式本身的字符分布特征进行分析，生成模式的特征向量，并在匹配的过程中利用模式的特征向量，以提高模式匹配的效率，其时间代价是目标串长度的线性函数，计算模式特征向量的时间也与模式本身的长度成正比。

习 题

1. 已知：s = "(xyz) + *"，t = "(x + z) * y"。试利用连接、抽子串和置换等基本运算，将 s 转化为 t。

2. 设 s1、s2 为字符串，试给出使 s1 + s2 = s2 + s1 成立的所有可能的条件（+ 为连接运算）。

3. 编写一个程序，统计在输入字符串中各个不同字符出现的频度并将结果存入文件，字符串中的合法字符为 A ~ Z 这 26 个字母和 0 ~ 9 这 10 个数字。

4. S 是一个存放在数组中长为 n 的字符串，试编写一个程序将 S 按下述要求改造之后输出：

(1) 将 S 的所有偶数位上的字符按照其原来的下标从大到小的次序放在 S 的后半部分。

(2) 将 S 的所有奇数位上的字符按照其原来的下标从小到大的次序放在 S 的前半部分。

例如，S = "ABCDEFGHIJKL"，改造后的 S 为"ACEGIKLJHFDB"。

5. 设计算法判断一个字符串 s 是否对称，若对称则返回 1，否则返回 0。例如，s 为"abba"返回 1，s 为"abab"返回 0。

6. 编写算法，求所有包含在字符串 s 中而不包含在字符串 t 中的字符（s 中重复的字符只选一个）构成的新串 r 以及 r 中每个字符在 s 中第一次出现的位置。

7. 设计一个线性算法来判断串 T 是否是另一个串 T'的循环反转，例如 arc 和 car 便互为循环反转。

8. 编写算法，从字符串 s 中删除所有和字符串 t 相同的子串。

9. 求字符串"BAAABBBAA"的特征向量，并与目标串"BAAABBBCDDDCCHHHHBBBAAABBBAADD"进行匹配，试画出匹配的过程示意图。

上 机 题

1. 给出函数 atoi(x)的实现方法,其功能是将字符串 x 转换为整数,字符串 x 由 0~9 十个数字和表示负数的"-"符号组成,返回值为整型数值。

2. 设计算法来统计一个输入字符串中所包含的整数个数,并输出这些数。假设输入的字符串既有数字又有非数字字符,例如 ak123x456 17960? 302gef4563,其中连续数字作为一个整体看成整数,例如其中的 123、456 等。

3. 编写算法,对给定的字符串 str,返回其最长重复子串及其下标位置。例如,str = "abcdacdac",则子串"cdac"是 str 的最长重复子串,下标为 2。

4. 字符串综合练习:编写一个简单行编辑程序,对文本文件进行插入、删除等修改操作。可以是类似于 UNIX Vi 或 DOS Edlin 的简单行编辑,要求实现以下功能:

(1) 行插入。

(2) 行删除。

(3) 改变当前行指针。

(4) 对于超过一屏的长文件,进行分页显示。

(5) 基于模式匹配算法进行查找和替换。

要求和提示:

- 必须实现查找字符串的操作(用 KMP 或其他模式匹配算法),不允许用编程环境所提供的查找算法(可以用函数重载)。
- 有能力的同学可以支持"*"、"?"等通配符。
- 本题不要求做图形界面,建议大部分同学实现普通的字符界面编辑器,注意界面简单友好。也允许实现为 Word 或 UltraEdit 那样的全屏幕编辑程序。
- 允许使用编程环境提供的图形包、字符串类(例如 MFC、CEditView)。
- 可以研究网上开源代码包,但最好不要直接采用,可以在详细说明自己引用了哪些包中哪些代码段的情况下,进行局部引用。
- 统计自己编写的代码行数,开源代码包、编程环境生成的代码框架等不能计算在内。

第5章 二 叉 树

前面的章节主要介绍了线性结构。从本章开始将介绍非线性结构。在非线性结构中，至少存在一个数据元素可以有两个或者两个以上的直接前驱或直接后继。

本章介绍树形结构的一个重要特例——二叉树，涉及二叉树的相关概念与存储表示，并给出其运算具体实现的详细描述，最后介绍堆、Huffman 树等典型应用。

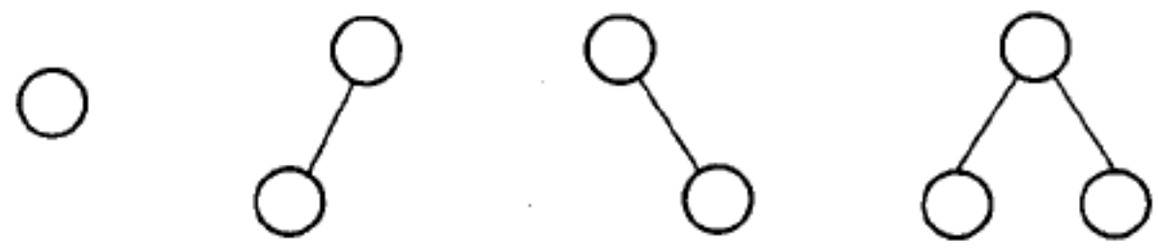
5.1 二叉树的概念

二叉树是一种特殊的树形结构，许多概念与树的相关概念类似。

5.1.1 二叉树的定义和基本术语

二叉树(binary tree)由结点的有限集合构成，这个有限集合或者为空集(empty)，或者为由一个根结点(root)及两棵互不相交、分别称做这个根的左子树(left subtree)和右子树(right subtree)的二叉树组成的集合。

上述二叉树的递归定义刻画了二叉树的固有特性。由二叉树的定义可以看出，二叉树既可以是一棵空树，也可以是由一个根结点和分别为其左子树和右子树的互不相交的二叉树组成(其左、右子树也可以为空)。由此，二叉树的 5 种基本形态如图 5.1 所示。



(a) 空二叉树　(b) 仅有根的二叉树　(c) 右子树为空的二叉树　(d) 左子树为空的二叉树　(e) 左、右子树均非空的二叉树

图 5.1　二叉树的 5 种基本形态

值得注意的是，二叉树的子树有左、右之分，其次序不能颠倒。例如，图 5.1 中的(c)和(d)就分别表示两棵不同的二叉树。

二叉树的结点与表的元素类似，可以表示任何一种数据类型。

二叉树是由唯一的起始结点引出的结点集合。这个起始结点称为根(root)。二叉树中的任

何非根结点都有且仅有一个前驱结点，称为该结点的父结点（或称为双亲，parent）。根结点即为二叉树中唯一没有父结点的结点。二叉树中的任何结点最多可能有两个后继结点，分别称为左子结点（或左孩子、左子女，left child）和右子结点（或右孩子，右子女，right child）。具有相同父结点的结点之间互称兄弟结点（sibling）。二叉树中一个结点的子树数目称为该结点的度（degree）。没有子结点的结点称为叶结点（leaf，也称树叶或终端结点），叶结点的度为0。除叶结点以外的那些非终端结点称为内部结点（或分支结点，internal node）。父结点 k 与子结点 k' 之间存在一条有向连线 $<k, k'>$，称做边（edge）。

若二叉树中存在结点序列 $\{k_0, k_1, \cdots, k_s\}$，使得 $<k_0, k_1>$，$<k_1, k_2>$，…，$<k_{s-1}, k_s>$ 都是二叉树中的边，则称从结点 k_0 到结点 k_s 存在一条路径（path），该路径所经历的边的个数称为这条路径的路径长度（path length）。若有一条由 k 到达 k_s 的路径，则称 k 是 k_s 的祖先（ancestor），k_s 是 k 的子孙（descendant）。

切断一个结点与其父结点的连接，则该结点与其子孙构成的树就称为以该结点为根的子树（subtree）。从根结点到某个结点的路径长度称为结点的层数（level），根结点为第0层，非根结点的层数是其父结点的层数加1。

除了严格区分左右子结点外，其他二叉树概念在树形结构中也适用。

5.1.2 满二叉树、完全二叉树、扩充二叉树

满二叉树和完全二叉树是两种特殊形态的二叉树。

如果一棵二叉树的任何结点，或者是树叶，或者左右子树均非空，则这棵二叉树称做满二叉树（full binary tree）。图5.2(a)就是一棵满二叉树的示例。

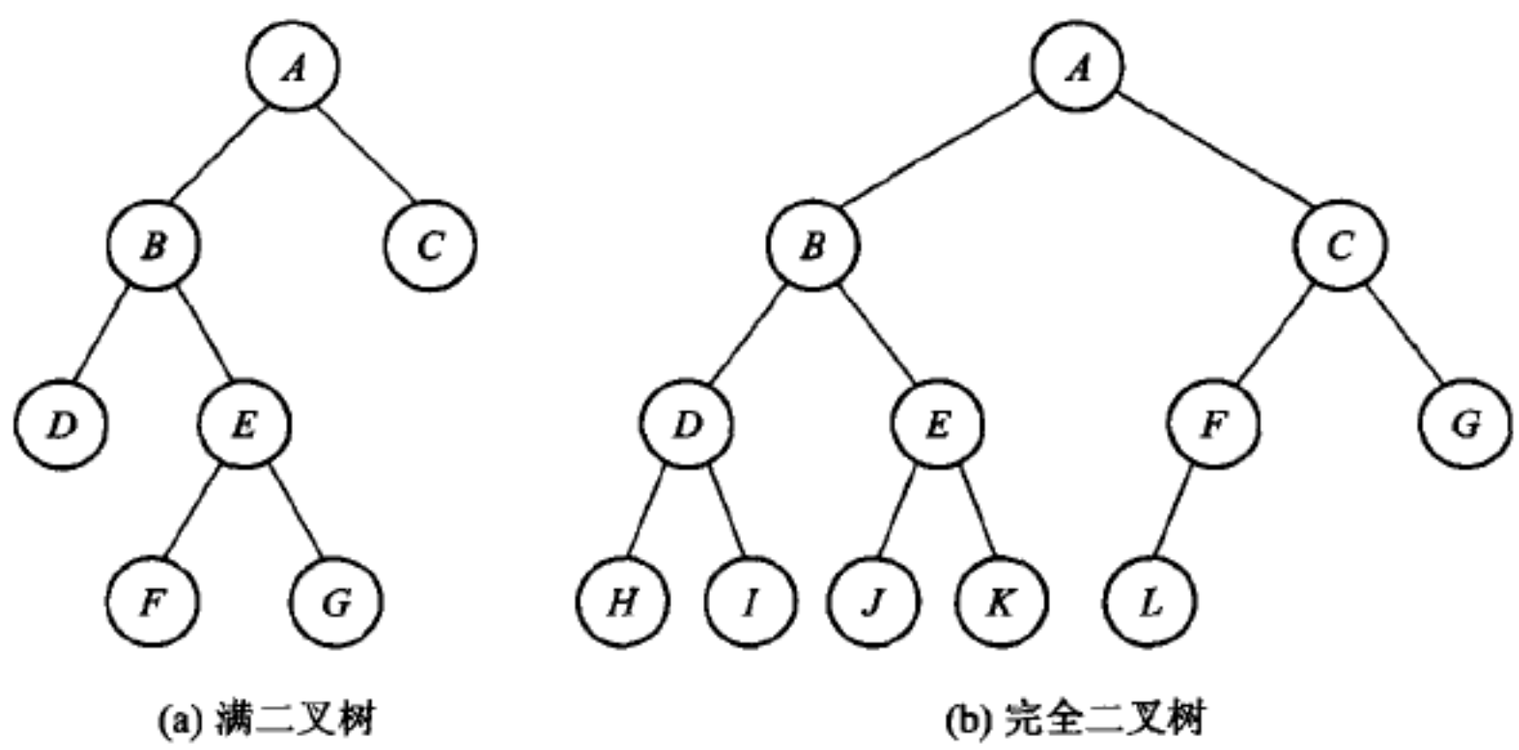


图5.2 满二叉树和完全二叉树示例

如果一棵二叉树最多只有最下面的两层结点度数可以小于2，并且最下面一层的结点都集中在该层最左边的连续位置上，则此二叉树称做完全二叉树（complete binary tree）。图5.2(b)

就是一棵完全二叉树，显然这棵完全二叉树不是满的。

完全二叉树的特点是其叶结点只可能在层次最大的两层出现。完全二叉树中由根结点到各个结点的路径长度总和在具有同样结点个数的二叉树中达到了最小，即任意一棵二叉树中根结点到各结点的最长路径一定不短于结点数目相同的完全二叉树中的路径长度。另外，完全二叉树还有几个重要性质将在下一节中进行介绍。在许多算法中都会用到完全二叉树的概念。

在二叉树中出现空子树的位置增加空树叶，所形成的二叉树称为扩充二叉树(extended binary tree)。构造一棵扩充二叉树只需要在原二叉树中度数为1的分支结点下增加一个空树叶，在原二叉树的树叶下面增加两个新的空树叶即可。扩充二叉树是满二叉树，新增空树叶(以下称为外部结点)的个数等于原二叉树的结点(以下称为内部结点)个数加1。

图5.3所示为一棵二叉搜索树（二叉搜索树将在第5.4节中介绍）的扩充二叉树。为了区别于原来二叉树中的结点，图中用小方框表示空树叶。

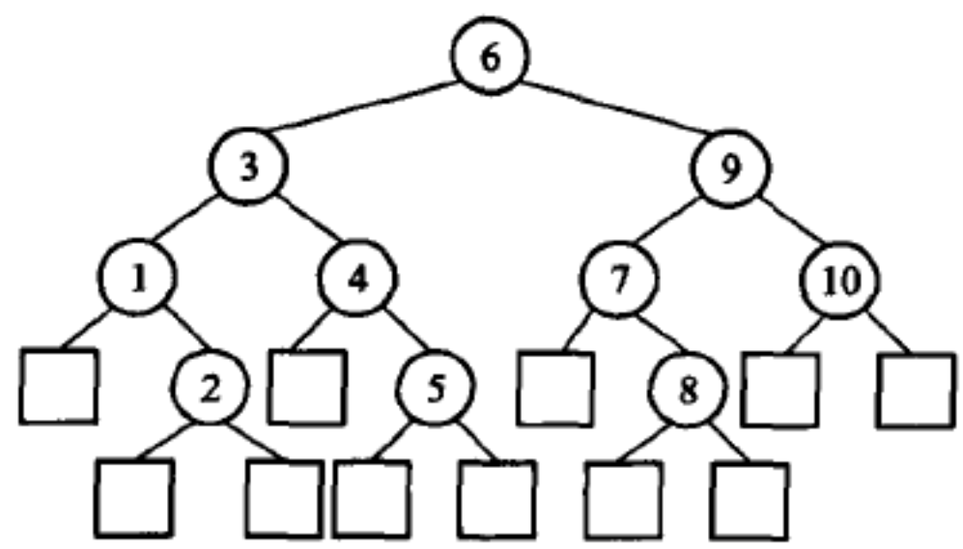


图5.3 扩充二叉树

从扩充的二叉树的根到每个外部结点的路径长度之和称为外部路径长度(E)。扩充的二叉树里从根到每个内部结点的路径长度之和称为内部路径长度(I)。

例如，图5.3中扩充二叉树的外部路径长度E和内部路径长度I分别是

$$E=3+4+4+3+4+4+3+4+4+3+3=39$$

$$I=0+1+2+3+2+3+1+2+3+2=19$$

外部路径长度E和内部路径长度I满足：$E=I+2n$，其中n是内部结点个数。

证明：对内部结点数目进行归纳。

当$n=1$时，$I=0$且$E=2$，故$E=I+2n$成立。

若对于有n个内部结点的扩充二叉树此等式已成立，即$E_n=I_n+2n$，现在考虑有$n+1$个内部结点的扩充二叉树。如果删去一个分支结点，该分支结点到根结点的路径长度是K，可使之成为有n个内部结点的扩充二叉树。由于删去了一个路径长度为K的内部结点，因此内部路径长度变为$I_n=I_{n+1}-K$；由于减少了两个路径长度为$K+1$的外部结点，增加了一个路径长度为K的外部结点，外部路径长度变为$E_n=E_{n+1}-2(K+1)+K=E_{n+1}-K-2$。由前两个等式，有$E_{n+1}=I_n+2n+K+2=I_{n+1}+2(n+1)$，因此$E=I+2n$。

5.1.3 二叉树的主要性质

性质1. 在二叉树中，第i层上最多有2^i个结点($i\geq 0$)。

证明：利用数学归纳法。

当$i=0$时，$2^0=1$，只有一个根结点，正确。

现在假设对所有的$j(1 \leqslant j < i)$命题成立，即第j层上至多有2^j个结点。下面证明当$j=i$时结论也成立。由归纳假设，第$i-1$层上最多有2^{i-1}个结点。由于二叉树每个结点的度数最大为2，因此第i层上的最大结点数为第$i-1$层上的最大结点数的2倍，即2^i个。

性质2. 深度为k的二叉树至多有$2^{k+1}-1$个结点($k \geqslant 0$)。

其中深度(depth)定义为二叉树中层数最大的叶结点的层数。

证明：由性质1可知，第i层的最大结点数为2^i，因此

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

性质3. 任何一棵二叉树，若其终端结点数为n_0，度为2的结点数为n_2，则$n_0 = n_2 + 1$。

证明：设n_1为二叉树中度为1的结点数。该二叉树的结点总数n为度分别为0、1、2的结点数之和，即

$$n = n_0 + n_1 + n_2 \tag{5.1}$$

除根结点外，其余结点都有一条边进入，设边数为e，有$n = e + 1$。由于这些边是由度为1或2的结点射出的，因此又有$e = n_1 + 2n_2$，于是得

$$n = e + 1 = n_1 + 2n_2 + 1 \tag{5.2}$$

由公式5.1和5.2得

$$n_0 + n_1 + n_2 = n_1 + 2n_2 + 1，即\ n_0 = n_2 + 1$$

性质4. 满二叉树定理：非空满二叉树树叶数目等于其分支结点数加1。

证明：满二叉树定理由性质3可直接推出。

性质5. 满二叉树定理推论：一个非空二叉树的空子树数目等于其结点数加1。

证明：设二叉树为T，将其所有空子树换为树叶，记新的扩充满二叉树为T'。所有原来T的结点现在是T'的分支结点。根据满二叉树定理，新添加的树叶数目等于T结点个数加1。而每个新添加的树叶对应于T的一个空子树。因此，T中空子树的数目等于T中的结点个数加1。

性质6. 有n个结点($n>0$)的完全二叉树的高度为$\lceil \log_2(n+1) \rceil$，深度为$\lceil \log_2(n+1) \rceil - 1$。

其中，二叉树的高度(height)定义为二叉树中层数最大的叶结点的层数加1。

证明：假设高度为h，则根据性质2和完全二叉树的定义，有

$$2^{h-1} - 1 < n \leqslant 2^h - 1 \quad 或 \quad 2^{h-1} < n + 1 \leqslant 2^h$$

对不等式中各项取对数，于是得到$h - 1 < \log_2(n+1) \leqslant h$。因为$h$为整数，所以$h = \lceil \log_2(n+1) \rceil$。

性质7. 对于具有n个结点的完全二叉树，结点按层次由左到右编号，则对任一结点i($0 \leqslant i \leqslant n-1$)有：

(1) 如果$i=0$，则结点i是二叉树的根结点；若$i>0$，则其父结点编号是$\lfloor (i-1)/2 \rfloor$。

(2) 当$2i+1 \leqslant n-1$时，结点i的左子结点是$2i+1$，否则结点i没有左子结点。
当$2i+2 \leqslant n-1$时，结点i的右子结点是$2i+2$，否则结点i没有右子结点。

(3) 当i为偶数且$0 < i < n$时，结点i的左兄弟是结点$i-1$，否则结点i没有左兄弟。

当 i 为奇数且 $i+1<n$ 时，结点 i 的右兄弟是结点 $i+1$，否则结点 i 没有右兄弟。

证明：这里证明(2)，(1) 和(3)即可由结论(2)推得。对于 $i=0$，由完全二叉树的定义，其左孩子的编号是1，如果 $1>n-1$，即不存在编号为1的结点，此时结点 i 没有左子结点。其右子结点的编号只能是2，如果 $2>n-1$，此时结点 i 没有右子结点。

对于 $i>0$ 分两种情况讨论：

(1) 设第 j 层的第一个结点编号为 i(此时有 $i=2^j-1$)，则其左子结点必为第 $j+1$ 层的第一个结点，其编号为 $2^{j+1}-1=2i+1$。如果 $2i+1>n-1$，那么 i 没有左子结点；其右子结点必为第 $j+1$ 层第二个结点，其编号为 $2i+2$。

(2) 假设第 j 层的某个结点编号为 i，若它有左子结点，那么它的左子结点必然是第 $j+1$ 层中的第 $2[i-(2^j-1)]+1$ 个，其左子结点的编号为 $(2^{j+1}-1)+2[i-(2^j-1)]=2i+1$；如果结点 i 有右子结点，那么其右子结点的编号必定是 $2i+2$。

5.2 二叉树的周游

上面介绍了二叉树的逻辑结构。本节将讨论二叉树的抽象数据类型和二叉树的几种周游算法。

5.2.1 二叉树的抽象数据类型

一般情况下，需要在二叉树的各个结点中存储必要的信息，对二叉树的操作和运算也主要集中在访问二叉树的结点信息上。例如，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。从二叉树的应用角度来看，有时还需要遍历二叉树的每个结点。在二叉树的抽象数据类型中，定义了二叉树基本操作的集合，在具体应用中可以以此为基础进行扩充。

为了强调抽象数据类型与存储无关，在此并没有具体规定该抽象数据类型的存储方式。二叉树结点的各种存储方式将在5.3节进行讨论。

代码5.1和代码5.2分别给出了二叉树结点和二叉树抽象数据类型的定义。类型模板T表示结点数据的数据类型。二叉树结点类中声明代码5.2的二叉树类为其友元类，以便于二叉树访问结点类的私有成员。在5.3节的代码5.8将给出部分成员函数的具体实现方法。

【代码5.1】 二叉树结点的抽象数据类型。

```
template <class T>
class BinaryTreeNode {
friend class BinaryTree<T>;          // 声明二叉树类为结点类的友元类，以便访问私有数据成员
private:
    T info;                                          // 二叉树结点数据域
```

```
public:
    BinaryTreeNode();                                          // 默认构造函数
    BinaryTreeNode(const T& ele);                              // 给定数据的构造函数
    BinaryTreeNode(const T& ele,BinaryTreeNode<T> *l, BinaryTreeNode<T> *r);
                                                               // 子树构造结点
    T value() const;                                           // 返回当前结点的数据
    BinaryTreeNode<T> * leftchild() const;                     // 返回当前结点的左子树
    BinaryTreeNode<T> * rightchild() const;                    // 返回当前结点的右子树
    void setLeftchild(BinaryTreeNode<T> *);                    // 设置当前结点的左子树
    void setRightchild(BinaryTreeNode<T> *);                   // 设置当前结点的右子树
    void setValue(const T& val);                               // 设置当前结点的数据域
    bool isLeaf() const;                                       // 判断当前结点是否为叶结点
    BinaryTreeNode<T> & operator = (const BinaryTreeNode<T> & Node);    // 重载赋值操作符
};
```

【代码 5.1 结束】

【代码 5.2】 二叉树的抽象数据类型。

```
template <class T>
class BinaryTree {
private:
    BinaryTreeNode<T> * root;                                  // 二叉树根结点
public:
    BinaryTree() {root = NULL;};                               // 构造函数
    ~BinaryTree() {DeleteBinaryTree(root);};                   // 析构函数
    bool isEmpty() const;                                      // 判定二叉树是否为空树
    BinaryTreeNode<T> * Root() {return root;};                 // 返回二叉树根结点
    BinaryTreeNode<T> * Parent(BinaryTreeNode<T> * current);   // 返回当前结点的父结点
    BinaryTreeNode<T> * LeftSibling(BinaryTreeNode<T> * current);  // 返回当前结点的左兄弟
    BinaryTreeNode<T> * RightSibling(BinaryTreeNode<T> * current); // 返回当前结点的右兄弟
    void CreateTree(const T& info, BinaryTree<T> & leftTree, BinaryTree<T> & rightTree);
                                                               // 构造新树
    void PreOrder(BinaryTreeNode<T> * root);                   // 前序周游给定二叉树
    void InOrder(BinaryTreeNode<T> * root);                    // 中序周游给定二叉树
    void PostOrder(BinaryTreeNode<T> * root);                  // 后序周游给定二叉树
    void LevelOrder(BinaryTreeNode<T> * root);                 // 按层次周游给定二叉树
    void DeleteBinaryTree(BinaryTreeNode<T> * root);           // 删除给定的二叉树
};
```

【代码 5.2 结束】

二叉树的某些应用中，经常需要在树中查找某些具有一定特征的结点，或者对树中的各个结

点做相应的处理，这就要求按照一定顺序访问二叉树中的结点。

所谓二叉树的周游（或称遍历，traversal）是指按照一定顺序依次访问树中所有的结点，并使得每个结点仅被访问一次。这里所说的“访问”是指操作，可以理解成对二叉树结点数据成员的处理，例如输出、修改结点的信息等。

对于线性结构来说，周游是一个很容易的问题。但是二叉树是一种非线性结构，每个结点可以有一个以上的直接后继，因此需要把二叉树中的结点转化为顺序结构才能周游整个二叉树。周游一棵二叉树的过程实际上就是把二叉树的结点放入一个线性序列的过程，即对二叉树进行线性化。

在此介绍两类周游二叉树的方法：深度优先周游和广度优先周游。

5.2.2 深度优先周游二叉树

深度优先周游算法的思想是尽可能地沿分支结点向深度方向进行周游。对于二叉树来说，深度优先周游即先沿着分支结点向左下降，当遇到左子树为空的时候，返回到上面最近的且其右子树尚未访问到的分支结点，转向该分支结点的右子结点，然后再尽可能地沿着左链前进。重复执行上述过程，直到周游完所有的结点为止。

深度优先周游二叉树时，结点既可以在向下周游之前访问，也可以在从子树返回之后访问。根据结点的访问时间，可以定义不同的二叉树深度优先周游算法。

二叉树的逻辑结构表明，二叉树由3个基本单元组成：根结点、左子树和右子树，如图5.4所示。

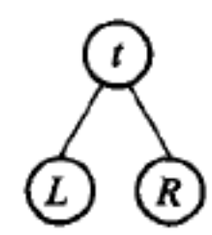


图5.4 二叉树的基本结构

根据二叉树的递归定义，可以考虑用递归算法来实现二叉树的周游。

把对一棵二叉树的周游过程转化为3个部分：处理当前结点，处理左子树和处理右子树。如果用 L、t 和 R 分别表示周游二叉树的左子树、访问当前结点和周游二叉树的右子树，那么可以形成6种周游二叉树的方案：tLR、LtR、LRt，tRL、RtL、RLt。通常，对子树总是按照先左后右的顺序进行处理，tLR、LtR 和 LRt 是常用的深度优先周游二叉树的方案。

基于二叉树的递归定义，这3种深度优先周游的递归定义如下：

（1）前序法（tLR 次序，preorder traversal）。其递归定义是：

- 访问根结点。
- 按前序周游左子树。
- 按前序周游右子树。

（2）中序法（LtR 次序，inorder traversal）。其递归定义是：

- 按中序周游左子树。
- 访问根结点。
- 按中序周游右子树。

(3) 后序法(*LRt* 次序,postorder traversal)。其递归定义是:

- 按后序周游左子树。
- 按后序周游右子树。
- 访问根结点。

前序(中序、后序)周游二叉树得到二叉树结点的一个有序序列,称为二叉树的前序(中序、后序)序列。对于如图 5.5 所示的二叉树前序序列是 *ABDEGCFHI*;中序序列是 *DBGEACHFI*;后序序列是 *DGEBHIFCA*。

二叉树这种非线性结构经过周游转化成线性序列,从而可以指定某种周游次序下某结点的前驱和后继结点。对于图 5.5 所示二叉树中的结点 *E*,在前序序列中它的前驱是结点 *D*,后继是结点 *G*;中序序列中 *E* 的前驱是结点 *G*,后继是结点 *A*;*E* 在后序序列中的前驱是结点 *G*,后继是结点 *B*。

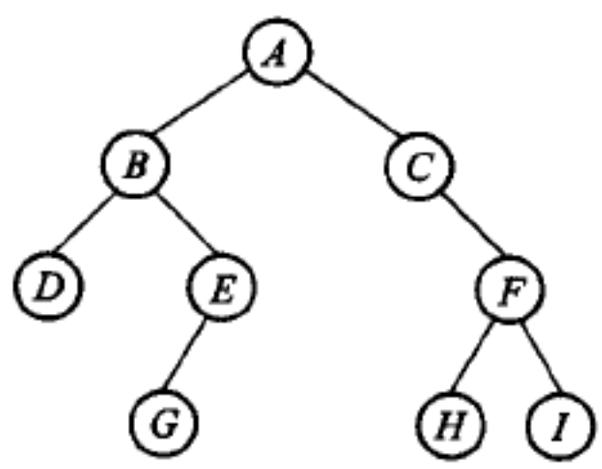


图 5.5 二叉树示例

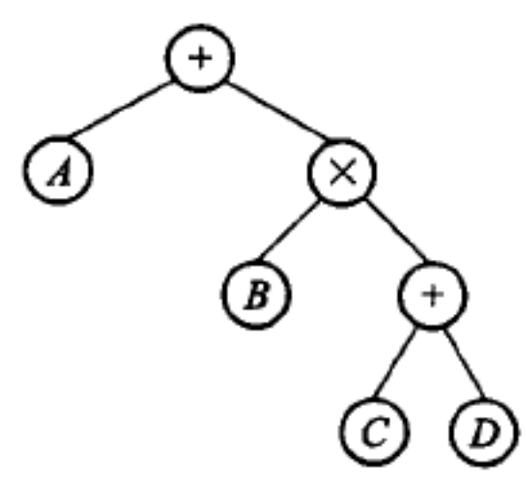


图 5.6 表达式树

二叉树的周游算法与表达式的"前缀"和"后缀"表示法之间有着密切的联系。例如,图 5.6 是表达式 $A+B\times(C+D)$ 的二叉树表示。按照前序方式周游,运算符出现在前面,参与运算的对象紧跟在其后,这样就形成了前缀表达式(波兰式):

$+A\times B+CD$

按照中序方式周游,得到的结果是去掉括号的中缀表达式:

$A+B\times C+D$

下面是后序方式周游的结果,得到的是后缀表达式(逆波兰式):

$ABCD+\times+$

1. 用递归实现二叉树的深度优先周游

按照二叉树周游的递归定义,算法 5.3 给出了深度优先周游的 3 种方式——前序、中序和后序周游二叉树的递归实现。算法的实现与存储结构无关。

在算法 5.3 中,函数 Visit 是对数据元素操作的相应函数,读者可以自行定义。

【算法 5.3】 深度优先周游二叉树或其子树。

```
template < class T >
void BinaryTree < T > ::PreOrder ( BinaryTreeNode < T >  * root)  {        // 前序周游二叉树或其子树
    if ( root ! = NULL) {
```

```
            Visit( root -> value( ) ) ;                          // 访问当前结点
            PreOrder( root -> leftchild( ) ) ;                   // 前序周游左子树
            PreOrder( root -> rightchild( ) ) ;                  // 前序周游右子树
        }
    }

    template < class T >
    void BinaryTree < T > :: InOrder ( BinaryTreeNode < T >  * root ) {      // 中序周游二叉树或其子树
        if ( root ! = NULL) {
            InOrder ( root -> leftchild( ) ) ;                   // 中序周游左子树
            Visit( root -> value( ) ) ;                          // 访问当前结点
            InOrder( root -> rightchild( ) ) ;                   // 中序周游右子树
        }
    }

    template < class T >
    void BinaryTree < T > :: PostOrder ( BinaryTreeNode < T >  * root) {     // 后序周游二叉树或其子树
        if ( root ! = NULL) {
            PostOrder( root -> leftchild( ) ) ;                  // 后序周游左子树
            PostOrder( root -> rightchild( ) ) ;                 // 后序周游右子树
            Visit( root -> value( ) ) ;                          // 访问当前结点
        }
    }
```

【算法 5.3 结束】

2. 深度优先周游二叉树的非递归算法

算法 5.3 中的 3 种周游算法都是递归算法。递归算法虽然简洁,但是有时程序设计不允许用递归,这时就存在如何把递归程序转化成等价的非递归算法的问题。解决这个问题的关键是设置一个栈结构。仿照递归算法执行过程中编译栈的工作原理,可以写出非递归周游二叉树的算法。

前序周游一棵二叉树的递归运行过程如下:首先访问其根结点,然后向左下降进入根的左子树,在左子树中又是按照前序先访问根结点,然后向左下降进入其左子树,直到周游完最底层的左子树,然后向上继续前序周游其根结点的右子树。

设计前序周游二叉树的非递归算法时,借助于一个栈,用于在向左下降的过程中记录留待以后访问的右子树的根结点,以便周游完一个结点的左子树后能顺利地转移到这个结点的右子树,继续周游下去。

非递归前序周游算法的主要思想是:每遇到一个结点,先访问该结点,并把该结点的非空右子结点推入栈中,然后周游其左子树;左子树周游不下去时,从栈顶弹出待访问的结点,继续周游。算法执行过程中,只有非空结点入栈。为了算法简洁,最开始压入一个空指针作为监视哨;

当这个空指针被弹出来时，则周游结束。

【算法 5.4】 非递归前序周游二叉树或其子树。

```
template < class T >
void BinaryTree < T > : : PreOrderWithoutRecursion( BinaryTreeNode < T >  * root) {
    using std : : stack ;                                   // 使用 STL 中的栈
    stack < BinaryTreeNode < T > *  > aStack ;
    BinaryTreeNode < T >  * pointer = root ;
    aStack. push( NULL) ;                                   // 栈底监视哨
    while ( pointer) {                                      // 或者！ aStack. empty( )
        Visit( pointer -> value( ) ) ;                      // 访问当前结点
        if ( pointer -> rightchild( ) ! = NULL)             // 非空右孩子入栈
            aStack. push( pointer -> rightchild( ) ) ;
        if ( pointer -> leftchild( ) ! = NULL)
            pointer = pointer -> leftchild( ) ;             // 左路下降
        else {                                              // 左子树访问完毕，转向访问右子树
            pointer = aStack. top( ) ;                      // 获得栈顶元素
            aStack. pop( ) ;                                // 栈顶元素退栈
        }
    }
}
```

【算法 5.4 结束】

非递归中序周游二叉树算法的主要思想是：每遇到一个结点就把它压入栈，然后去周游其左子树；周游完左子树后，从栈顶弹出并访问这个结点，然后按照其右链接指示的地址再去周游该结点的右子树。算法 5.5 是中序周游二叉树的非递归算法。

【算法 5.5】 非递归中序周游二叉树或其子树。

```
template < class T >
void BinaryTree < T > : : InOrderWithoutRecursion( BinaryTreeNode < T >  * root) {
    using std : : stack ;                                   // 使用 STL 中的栈
    stack < BinaryTreeNode < T > *  > aStack ;
    BinaryTreeNode < T >  * pointer = root ;
    while ( ! aStack. empty( ) || pointer) {
        if ( pointer) {
            aStack. push( pointer) ;                        // 当前指针入栈
            pointer = pointer -> leftchild( ) ;             // 左路下降
        }
        else {                                              // 左子树访问完毕，转向访问右子树
            pointer = aStack. top( ) ;                      // 获得栈顶元素
```

```
                aStack. pop( ) ;                    // 栈顶元素退栈
                Visit( pointer -> value( ) ) ;      // 访问当前结点
                pointer = pointer -> rightchild( ) ;  // 指针指向右孩子
            }
        }
    }
```

【算法 5.5 结束】

后序周游二叉树时最先处理左子树，然后是右子树，最后才访问当前结点。算法 5.6 给出了非递归实现二叉树后序周游的算法。在非递归的后序周游算法中，每遇到一个结点，先把它压入栈中，去周游它的左子树；周游完它的左子树后，应继续周游该结点的右子树；周游完右子树之后，才从栈顶弹出该结点并访问它。由于访问某个结点前需要知道是否已经访问该结点的右子树，因此需要给栈中的每个元素加一个标志位 tag。标志位用枚举类型 Tags 表示：Left 表示已进入该结点的左子树；Right 表示已进入该结点的右子树。

【算法 5.6】 非递归后序周游二叉树或其子树。

```
enum Tags{Left,Right} ;                     // 定义枚举类型标志位
template < class T >
class StackElement {                        // 栈元素的定义
public:
    BinaryTreeNode < T > * pointer;         // 指向二叉树结点的指针
    Tags tag;                               // 标志位
} ;
template < class T >
void BinaryTree < T > ::PostOrderWithoutRecursion( BinaryTreeNode < T > * root) {
    using std::stack;                       // 使用 STL 的栈
    StackElement < T > element;
    stack < StackElement < T > > aStack;
    BinaryTreeNode < T > * pointer;
    if ( root == NULL)                      // 如果是空树则返回
        return;
    else pointer = root;
    while ( ! aStack. empty( ) || pointer) {
        while ( pointer ! = NULL) {         // 如果当前指针非空则压栈并下降到最左子结点
            element. pointer = pointer;
            element. tag = Left;            // 置标志位为 Left，表示进入左子树
            aStack. push( element) ;
            pointer = pointer -> leftchild( ) ;
        }
```

```
            element = aStack. top( ) ;              // 获得栈顶元素
            aStack. pop( ) ;                        // 栈顶元素退栈
            pointer = element. pointer;
            if ( element. tag = = Left) {           // 如果从左子树返回
                element. tag = Right;               // 置标志位为 Right, 表示进入右子树
                aStack. push( element) ;
                pointer = pointer -> rightchild( ) ;
            }
            else {                                  // 如果从右子树返回
                Visit( pointer -> value( ) ) ;      // 访问当前结点
                pointer = NULL;                     // 置 point 指针为空,以继续弹栈
            }
        }
    }
```

【算法 5.6 结束】

不管采用哪种周游方式,对于有 n 个结点的二叉树,周游完树的所有元素都需要 $O(n)$ 时间。只要对每个结点的处理(函数 Visit 的执行)时间是一个常数,那么周游二叉树就可以在线性时间内完成。所需要的辅助空间为周游过程中栈的最大容量,即树的高度。最坏情况下,具有 n 个结点的二叉树高度为 n,所需要的空间复杂度为 $O(n)$。

5.2.3 广度优先周游二叉树

二叉树的周游除了按上述的前序、中序和后序进行外,还可以从上到下、从左到右地按层次进行。下面讨论按层次(即广度优先)周游二叉树的算法。

对二叉树按层次周游的过程是:首先访问第 0 层,即根结点所在的层,然后从左到右依次访问第一层两个结点,依次类推,当第 i 层的所有结点访问完之后,再从左至右依次访问第 $i+1$ 层的各个结点。对于图 5.5 中的二叉树进行广度优先周游的结果为:

ABCDEFGHI

根据层次周游二叉树的性质,这里需要使用一个队列作为辅助的存储结构。层次周游过程的实现就是从根结点开始逐层逐个地访问各个结点。在周游开始的时候,首先将根结点放入队列;然后每次从队列中取出队头元素进行处理,每处理一个结点时,按从左至右的顺序把它的所有子结点放入队列。这样,上层结点总是排在下一层结点的前面,从而实现了二叉树的广度优先周游。

算法 5.7 给出了利用队列实现广度优先周游二叉树。初始化时,根结点插入到空队列,周游的每一步,算法都将从队列头上删除一个结点,并且将其子结点插入队尾。

【算法 5.7】 广度周游二叉树及其子树。

```
template < class T >
void BinaryTree < T > : : LevelOrder( BinaryTreeNode < T >  * root) {
    using std : : queue;                                    // 使用 STL 的队列
    queue < BinaryTreeNode < T > * > aQueue;
    BinaryTreeNode < T >  * pointer = root;
    if ( pointer)
        aQueue. push( pointer) ;                            // 根结点入队列
    while ( ! aQueue. empty( ) ) {                          // 队列非空
        pointer = aQueue. front( ) ;                        // 获得队列首结点
        aQueue. pop( ) ;                                    // 当前结点出队列
        Visit( pointer -> value( ) ) ;                      // 访问当前结点
        if ( pointer -> leftchild( )  ! = NULL)
            aQueue. push( pointer -> leftchild( ) ) ;       // 左子树进队列
        if ( pointer -> rightchild( ) ! = NULL)
            aQueue. push( pointer -> rightchild( ) ) ;      // 右子树进队列
    }
}
```

【算法 5.7 结束】

广度优先周游一棵具有 n 个结点的二叉树，其时间复杂度也是 $O(n)$。队列所需要的最大存储空间由二叉树中具有最多结点数目的那一层上的结点个数来决定。因此，周游一棵满的完全二叉树所需要的队列空间最大，最大长度是 $(n+1)/2$。

5.3 二叉树的存储结构

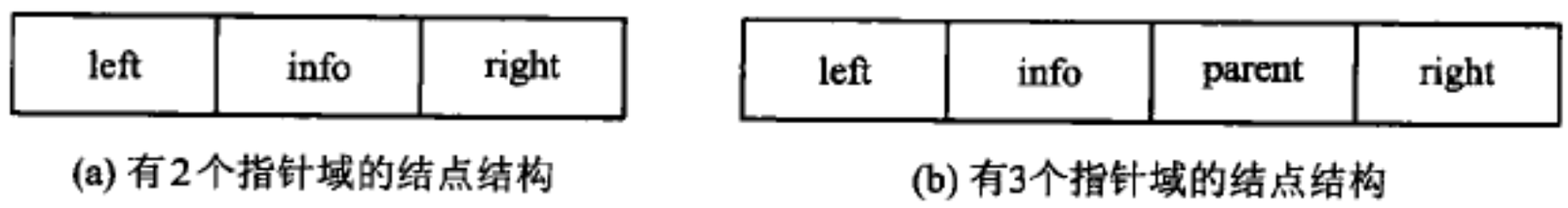
前面几节详细地讨论了二叉树的逻辑结构和抽象数据类型，它们不依赖于二叉树的物理存储方式。二叉树的存储实现方法有多种，既可以用链接存储结构实现，又可以用顺序存储结构实现。这些方法各有其特点和适用范围，在应用中要根据情况决定采用哪种方法。下面详细介绍二叉树的这两种存储方式。

5.3.1 二叉树的链式存储结构

由二叉树的定义可知，二叉树是一种非线性结构，其中的每一个结点由一个数据元素和指向其左右子树的分支组成。

根据二叉树的这种逻辑结构，存储二叉树自然地可以使用链式存储方式。所谓链式存储方式，是指二叉树的各结点随机地存储在内存空间，结点之间的关系用指针表示。

在用链接方式存储二叉树时，对于每一个结点，除了存储结点本身的信息外，还要设置两个指向左右子结点的指针。即表示二叉树的链表的结点包含 3 个域：数据域和左、右指针域。用 info 域存储结点的数据元素，另外再设置两个指针域 left 和 right，分别指向结点的左子结点和右子结点。当结点的某个子结点为空时，相应的指针为空指针。其结点结构如图 5.7(a)所示。有时为了便于查找二叉树中某个结点的父结点，还可以在结点结构中加入一个指向其父结点的指针域，如图 5.7(b)所示。



(a) 有2个指针域的结点结构　　(b) 有3个指针域的结点结构

图 5.7　二叉树结点的存储结构

利用这两种结点结构所得到的二叉树的存储结构分别称为二叉链表（也称“left - right 存储法”）和三叉链表。在使用二叉链表存储的二叉树中，如果要查找某个结点的父结点，则需要从根结点出发依次巡查。而在用三叉链表表示的二叉树中就很容易实现，只需要顺着 parent 指针就能直接找到该结点的父结点。

二叉链表表示法是二叉树最常用的存储结构。如果用一个头指针指向二叉树的根结点，那么图 5.5 中二叉树的二叉链表存储结构就可以表示为如图 5.8 所示的形式。

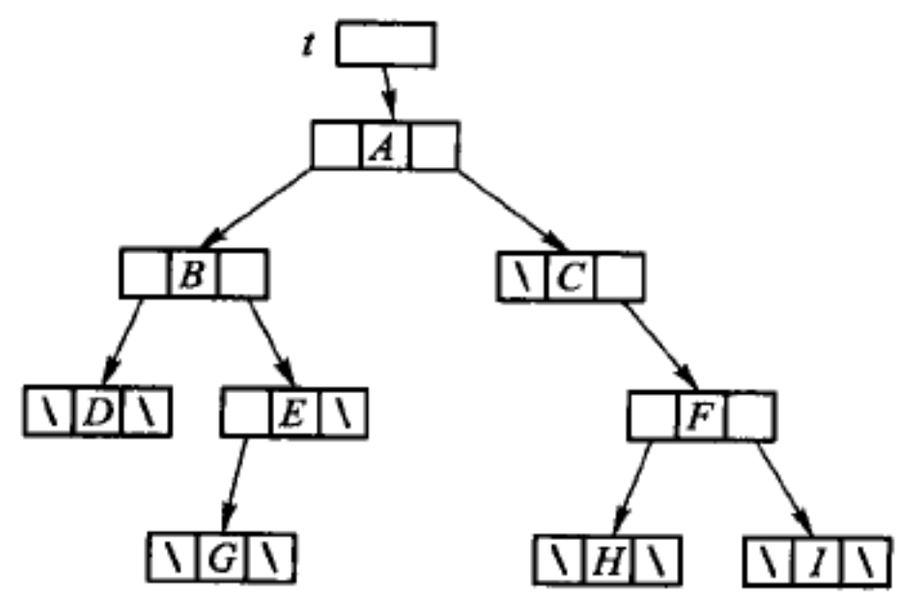


图 5.8　图 5.5 中二叉树的二叉链表存储结构

很容易看出，在含有 n 个结点的二叉链表中有 $n+1$ 个空链域。可以利用这些空链域存储其他的有用信息，形成其他的链式存储结构，例如习题 22 中的穿线二叉树。

第 5.2 节代码 5.1 和代码 5.2 给出的二叉树结点和二叉树的抽象数据类型，其实现本质上可以看做二叉链表表示形式。实现二叉树的二叉链表，需要扩展代码 5.1 的二叉树结点抽象数据类型 BinaryTreeNode。每个结点元素除拥有数据域 info 以外，还要增加两个指针 left 和 right 分别指向左右子结点结构。代码 5.8 给出了代码 5.2 定义的二叉树类 BinaryTree 中部分成员函数的具体实现。

【代码 5.8】　二叉树部分成员函数的实现。

```
// 用二叉链表实现二叉树需要在代码 5.1 的 BinaryTreeNode 类中增加两个私有数据成员
private:
    BinaryTreeNode < T >  * left;                    // 指向左子树的指针
    BinaryTreeNode < T >  * right;                   // 指向右子树的指针

template < class T >
```

```
bool BinaryTree < T > :: isEmpty( ) const {                    // 判定二叉树是否为空树
        return ( root ! = NULL ? false : true) ;
}

template < class T >
BinaryTreeNode < T > *  BinaryTree < T > ::Parent( BinaryTreeNode < T >  * current) {
        using std::stack;                                      // 使用 STL 中的栈
        stack < BinaryTreeNode < T > *  > aStack;
        BinaryTreeNode < T >  * pointer = root;
        if( root ! = NULL && current ! = NULL) {
                while ( ! aStack. empty( ) || pointer) {
                        if ( pointer ! = NULL) {
                                if ( current == pointer -> leftchild( ) || current == pointer -> rightchild( ) )
                                        return pointer;        // 如果 pointer 的孩子是 current 则返回 parent
                                aStack. push( pointer) ;       // 当前指针入栈
                                pointer = pointer -> leftchild( ) ;  // 当前指针指向左孩子
                        }
                        else {                                 // 左子树访问完毕,访问右子树
                                pointer = aStack. top( ) ;     // 获得栈顶元素
                                aStack. pop( ) ;               // 栈顶元素退栈
                                pointer = pointer -> rightchild( ) ;// 当前指针指向右孩子
                        }
                }
        }
}

// 创建一棵新树,参数 info 为根结点元素,leftTree 和 rightTree 是不同的两棵树
template < class T >
void BinaryTree < T > :: CreateTree ( const T& info, BinaryTree < T > & leftTree, BinaryTree < T > & right-
Tree) {
        root = new BinaryTreeNode < T > ( info, leftTree. root, rightTree. root) ;      // 创建新树
        leftTree. root = rightTree. root = NULL;          // 原来两棵子树的根结点置为空,避免非法访问
}
template < class T >
void BinaryTree < T > :: DeleteBinaryTree( BinaryTreeNode < T >  * root) {    // 后序周游删除二叉树
        if ( root ! = NULL) {
                DeleteBinaryTree( root -> left) ;                              // 递归删除左子树
                DeleteBinaryTree( root -> right) ;                             // 递归删除右子树
```

```
        delete root;                                    // 删除根结点
    }
}
```

【代码 5.8 结束】

5.3.2 完全二叉树的顺序存储结构

二叉树的顺序存储是指按照一定次序，用一组地址连续的存储单元存储二叉树上的各个结点元素。由于二叉树是一种非线性结构，因此必须将二叉树的结点排成一个线性序列，使得通过结点在这个序列中的相对位置就能够确定结点之间的逻辑关系。但通常情况下，只通过相应位置不足以刻画整个树形结构。

而对于一棵具有 n 个结点的完全二叉树，可以从根结点起自上而下、从左至右地把所有的结点编号，得到一个足以反映整个二叉树结构的线性序列。采用这种方式，线性序列里存储的结点就是按照层次周游二叉树得到的排列。事实上，这就是完全二叉树的顺序表示法。

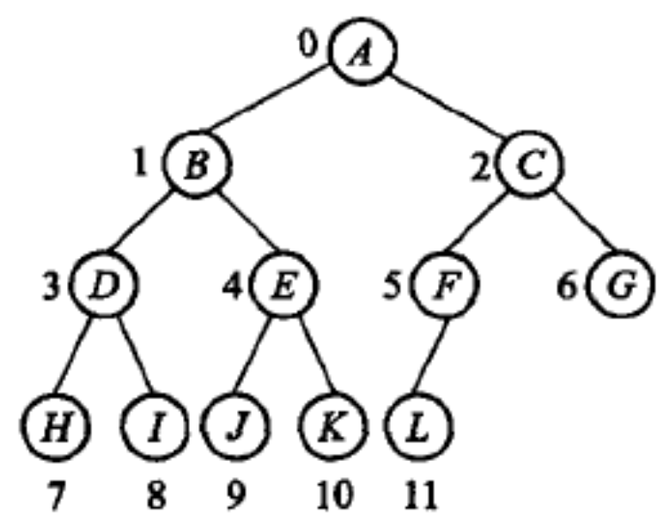


图 5.9 完全二叉树的结点编号

像图 5.9 所示的那样，按层次顺序将一棵 n 个结点的完全二叉树中的所有结点从 0 到 $n-1$ 编号。可以将这棵二叉树编号为 i 的结点元素存储在一维数组下标为 i 的分量中。例如，图 5.9 所示的二叉树的顺序存储结构如图 5.10 所示。

在二叉树的这种表示方法中，各结点之间的逻辑关系是隐含表示的。在完全二叉树中，除了最下面一层结点外，各层都充满了结点，且每一层结点的个数恰好是上一层结点个数的两倍。完全二叉树的顺序存储结构中数组下标的关系仍然满足 5.1.3 节中介绍的完全二叉树的性质，因此根据标志结点元素存储位置的数组下标可以方便地计算出其左右子结点及其父结点的存储地址。对于任何一个二叉树结点，如果它存储在数组的第 i 个位置，那么它的左右子结点分别存放在 $2i+1$ 和 $2i+2$ 的位置，其父结点的下标为 $\lfloor (i-1)/2 \rfloor$。

A	B	C	D	E	F	G	H	I	J	K	L
0	1	2	3	4	5	6	7	8	9	10	11

图 5.10 完全二叉树的顺序存储结构

对于完全二叉树这种特殊情况，结点的层次序列就足以反映整个二叉树的结构。顺序方式是完全二叉树最简单、最节省空间的存储方式。与完全二叉树相关的大多数算法都使用这种顺序存储方式，例如 5.5 节中的堆以及 8.3.2 节的堆排序。第 6 章介绍了一般情况下树的顺序存储，由于二叉树与树能够进行等价转换，因此一般情况下的二叉树也可以采用类似的顺序存储结构方法。

在不同的二叉树的存储结构中，不仅空间开销有差异，实现二叉树的操作方法也不同。因此

在具体应用中采取什么存储结构，除了根据二叉树的形态之外，还应该考虑时间、空间复杂度和算法的简洁性。

5.4 二叉搜索树

在实际应用中，经常会碰到这样的操作：在一组记录中检索一个记录，向其中插入一个记录或者删除一个记录等。

对于一个无序线性表，插入记录时只需要把该记录放在表的末端，但在表中查找一个特定记录的检索时间却相对较慢。对于有序线性表，如果用二分查找法检索特定的记录，检索效率较高，但是遇到动态增减变化的情况时（如插入或删除一个元素）需要移动大量的元素。

很多应用都需要一种插入、删除和检索记录效率都较高的数据组织方法。本节介绍的二叉搜索树（binary search tree，BST，也可称为二叉排序树、二叉查找树等），就是这样一种高效的数据结构。

二叉搜索树是一类满足以下属性的特殊二叉树：二叉搜索树中的每个非空结点表示一个记录；若某结点左子树不为空，则左子树上所有结点的值均小于该结点的关键码值；若其右子树不为空，则右子树上所有结点的值均大于该结点的关键码值。二叉搜索树可以是一棵空树，任何结点的左右子树都是二叉搜索树。按照中序周游整个二叉树可得到一个由小到大的有序排列。

二叉搜索树的定义表明树中各结点的关键码必须是唯一的。例如，对于关键码集合 $K=\{50,19,35,55,20,5,100,52,88,53,92\}$，二叉搜索树如图 5.11 所示。

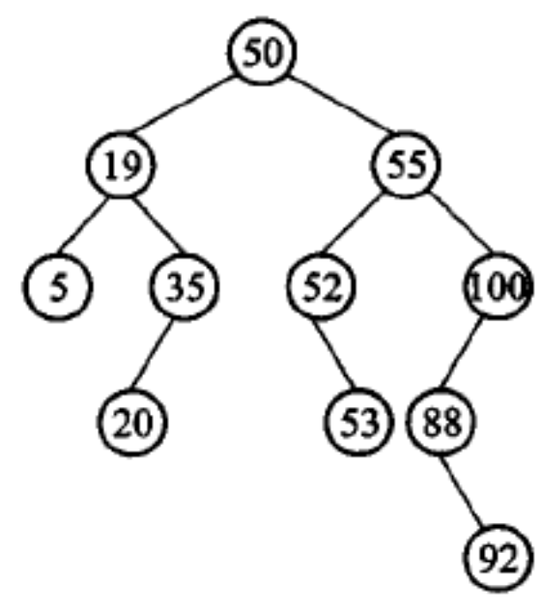


图 5.11 二叉搜索树

二叉搜索树是一种动态查找表，既可以在查找过程中插入表中不存在的某个数据元素，也可以从中删除已经存在的某个元素。下面介绍二叉搜索树的检索、插入和删除算法。

二叉搜索树的检索过程比较容易。假设要在二叉搜索树中检索关键码 key，则从根结点开始，如果根结点存储的值为 key，则返回检索结果，检索结束；如果不是，则必须检索树的更深层。由二叉搜索树的定义可知，树中的数据元素是按照一定的大小顺序排列的，因此继续检索时只需要查找两棵子树之一，这也是二叉搜索树的高效率所在。将给定值 key 与根结点的关键码比较，如果 key 小于根结点的值，则只需要检索左子树；如果 key 大于根结点的值，则只检索右子树。这个过程一直持续到 key 被匹配成功或者遇到叶结点为止。如果遇到叶结点仍没有发现 key，则说明 key 不在这棵二叉搜索树中。

二叉搜索树的插入算法，需要运用检索方法，查找待插入关键码是否在树中；如果存在则不允许插入重复关键码；如果直到找到叶结点还没有发现重复关键码，则把新结点插入到待插入方向作为新的叶结点。对于给定的关键码集合，可以从一个空的二叉搜索树开始，按照检索路径搜索这棵二叉树，将关键码逐个插入到相应的叶结点位置，从而动态生成二叉搜索树。图 5.11 所

示的二叉搜索树就是按照给定的关键码顺序动态插入而得到的。

二叉搜索树插入操作具体是这样的:将待插入结点的关键码与根结点的关键码相比较,若待插入的关键码小于根结点的关键码,则进入左子树,否则进入右子树。按照同样的方式沿检索路径直到叶结点,确定插入位置,把待插入结点作为一个新叶结点插入到二叉搜索树中。算法 5.9 给出插入结点的算法,其中的两个指针参数 root 和 newpointer 分别指向二叉搜索树的根和待插入的新结点。

【算法 5.9】 二叉搜索树的结点插入算法。

```
template < class T >
void BinarySearchTree < T > : : InsertNode ( BinaryTreeNode < T >  * root , BinaryTreeNode < T >  * new-
pointer) {
    BinaryTreeNode < T >  * pointer = NULL;
    if( root == NULL) {                                  // 如果是空树
        Initialize( newpointer) ;                        // 则用指针 newpointer 作为树根
        return;
    }
    else pointer = root;
    while ( pointer ! = NULL) {
        if ( newpointer -> value( ) == pointer -> value( ) ) // 如果存在相等的元素则不用插入
            return ;
        else if ( newpointer -> value( ) < pointer -> value( ) ) {
                                                         // 如果待插入结点小于 pointer 的关键码值
            if ( pointer -> leftchild( ) == NULL) {      // 如果 pointer 没有左孩子
                pointer -> left = newpointer;            // newpointer 作为 pointer 的左子树
                return;
            }
            else pointer = pointer -> leftchild( ) ;     // 向左下降
        }
        else {                                           // 若待插入结点大于 pointer 的关键码值
            if ( pointer -> rightchild( ) == NULL) {     // 如果 pointer 没有右孩子
                pointer -> right = newpointer;           // newpointer 作为 pointer 的右子树
                return;
            }
            else pointer = pointer -> rightchild( ) ;    // 向右下降
        }
    }
}
```

【算法 5.9 结束】

对二叉搜索树的检索,每一次只需与结点的一棵子树做比较。而在执行插入操作时,也不必

像在有序线性表中插入元素那样要移动大量的数据，只需改动某个结点的空指针插入一个叶结点即可。与查找结点的操作一样，插入一个新结点操作的时间复杂度是根到插入位置的路径长度，因此在树形比较平衡时二叉搜索树的效率相当高。

由于二叉搜索树要求关键码之间满足一定的大小关系，这就使得从树中删除一个结点的算法比较复杂。从二叉搜索树中删除一个结点时，要保持二叉搜索树的性质，就不能在二叉搜索树中留下一个空位置，因此需要用另一个结点来填充这个位置并且保持性质。设 pointer、temppointer 是指针变量，其中 pointer 表示要删除的结点。首先，找到待删除的结点 pointer，删除该结点的过程如下：若结点 pointer 没有左子树，则用 pointer 右子树的根代替被删除的结点 pointer；若结点 pointer 有左子树，则在左子树里找到按中序周游的最后一个结点 temppointer，把 temppointer 的右指针置成指向 pointer 的右子树的根，然后用结点 pointer 左子树的根代替被删除的结点 pointer。

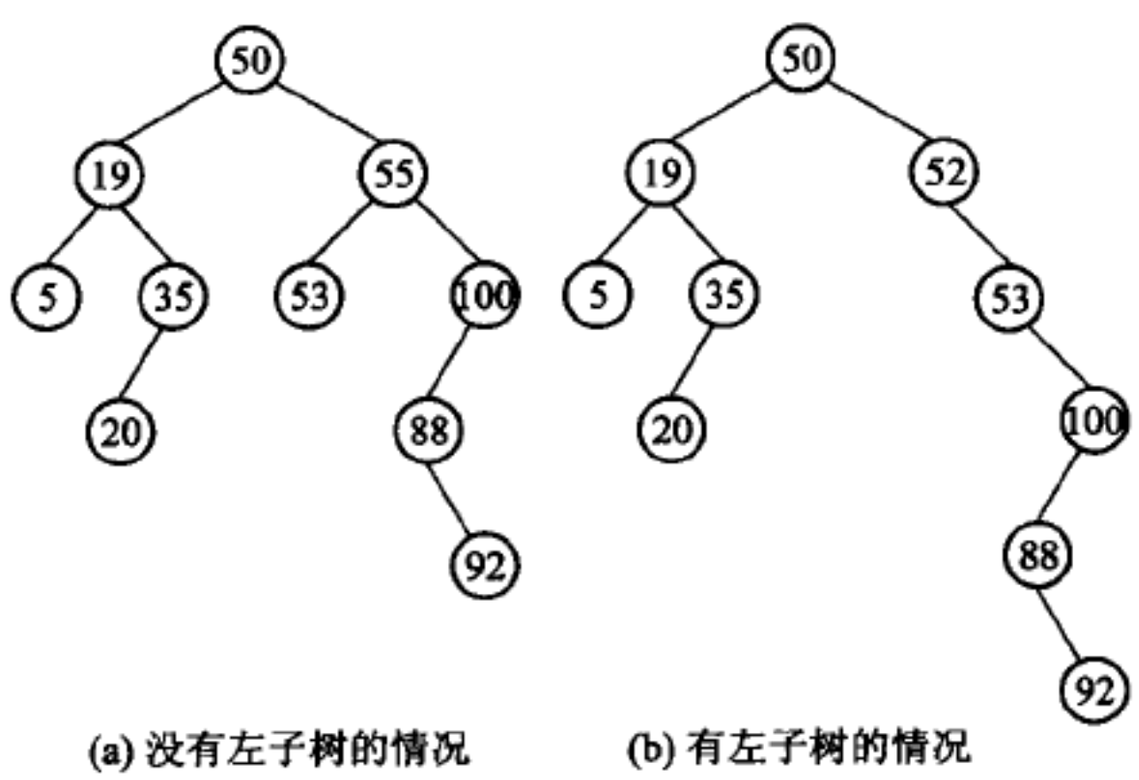
图 5.12(a)和(b)分别表示从图 5.11 所示的二叉搜索树中删除结点 52 和 55 以后得到的二叉搜索树。

从图 5.12(b)可以看到，将结点 pointer 的右子树下降为 temppointer 的右子树后，二叉搜索树的高度可能会增加。为了避免这种情况，也可以对删除方法作其他的规定，只要删除操作使得最后形成的二叉树仍然满足二叉搜索树的性质即可。

改进的二叉搜索树结点删除算法的思想为：若结点 pointer 没有左子树，则用 pointer 右子树的根代替被删除的结点 pointer；若结点 pointer 有左子树，则在左子树中找到按中序周游的最后一个结点 temppointer(即左子树中的最大结点)并将其从二叉搜索树中删除。由于 temppointer 没有右子树，因此删除该结点只需用 temppointer 的左子树代替 temppointer，然后用 temppointer 结点代替待删除的结点 pointer。

图 5.13 表示按这种规定从图 5.11 所示的二叉搜索树中删除结点 55 后得到的二叉搜索树。

算法 5.10 给出了改进的二叉搜索树结点删除算法。



(a) 没有左子树的情况　　(b) 有左子树的情况

图 5.12　二叉搜索树的删除示例

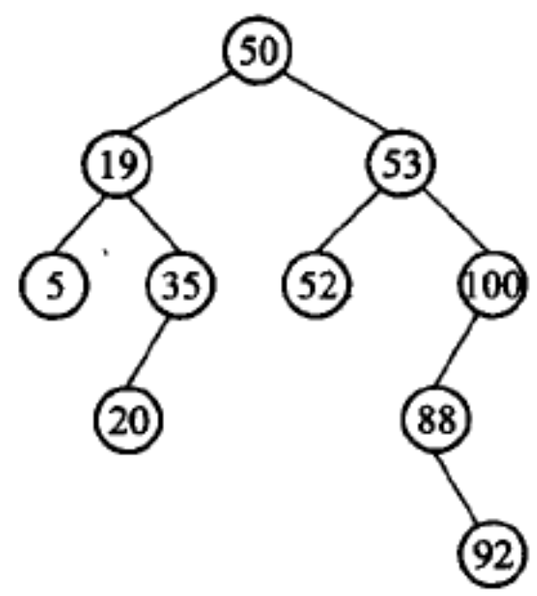


图 5.13　改进的二叉搜索树的删除

【算法 5.10】 改进的二叉搜索树的结点删除。

```
template <class T>
void BinarySearchTree<T>::DeleteNodeEx(BinaryTreeNode<T> *pointer) {
    if ( pointer == NULL )                                  // 若待删除结点不存在则返回
        return;
    BinaryTreeNode<T> *temppointer;                         // 用于保存替换结点
    BinaryTreeNode<T> *tempparent = NULL;                   // 用于保存替换结点的父结点
    BinaryTreeNode<T> *parent = Parent(pointer);            // 用于保存待删除结点的父结点
    if ( pointer -> leftchild() == NULL)                    // 如果待删除结点的左子树为空
        temppointer = pointer -> rightchild();              // 替换结点赋值为其右子树的根
    else {                                                  // 待删除结点左子树不空,在左子树中
                                                            // 寻找最大结点作为替换结点
        temppointer = pointer -> leftchild();
        while (temppointer -> rightchild() != NULL) {       // 寻找左子树中的最大结点
            tempparent = temppointer;
            temppointer = temppointer -> rightchild();      // 向右下降
        }
        if (tempparent == NULL)                             // 如果替换结点就是被删结点的左子结点
            pointer -> left = temppointer -> leftchild();   // 替换结点左子树挂接到被删结点的
                                                            // 左子树
        else tempparent -> right = temppointer -> leftchild(); // 替换结点的左子树作为其父结点右
                                                            // 子树
        temppointer -> left = pointer -> leftchild();       // 继承 pointer 的左子树
        temppointer -> right = pointer -> rightchild();     // 继承 pointer 的右子树
    }
    // 下面用替换结点代替待删除结点
    if (parent == NULL)
        root = temppointer;
    else if ( parent -> leftchild() == pointer )
        parent -> left = temppointer;
    else parent -> right = temppointer;
    delete pointer;                                         // 删除该结点
    pointer = NULL;
    return;
}
```

【算法 5.10 结束】

5.5 堆与优先队列

在现实应用中，经常会遇到频繁地在一组对象中查找最大值或最小值的情况。为了达到这个目的，可以每次都先排序，然后从已排序的序列中找到其最大值或最小值。这种方法虽然可行，但时间开销却比较大。对于这种特殊的应用，有没有一种结构能够提供较高的效率？本节将介绍这种数据结构——堆。

堆可分为最小堆和最大堆。本节主要讨论最小堆，最大堆具有类似的性质。

5.5.1 堆的定义及其实现

最小堆（min－heap，最小值堆）是关键码序列$\{K_0, K_1, \cdots, K_{n-1}\}$，它具有如下特性：

$K_i \leqslant K_{2i+1}$，

$K_i \leqslant K_{2i+2}(i=0,1,\cdots,\lfloor n/2 \rfloor -1)$

从逻辑的角度来讲，堆是一种树形结构，而且是一种特殊的完全二叉树。此完全二叉树的每个结点对应于序列中的一个关键码，根结点对应于关键码K_0，按层次从左至右依次类推。说其特殊，主要是因为堆序只是局部有序，即最小堆对应的完全二叉树中所有内部结点的值均不大于其左右子结点的关键码值，且一个结点与其兄弟之间没有必然的联系。最小堆不像二叉搜索树那样实现了关键码的完全排序。相比较而言，只有当结点之间是父子关系时，才可以确定这两个结点关键码的大小关系。

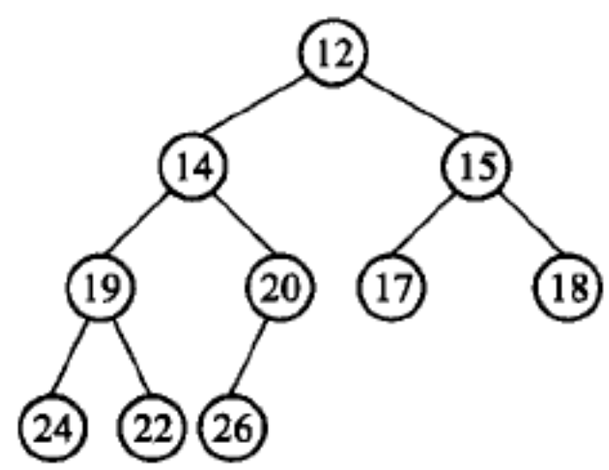


图 5.14 最小堆对应的完全二叉树

例如，关键码序列$K=\{12,14,15,19,20,17,18,24,22,26\}$所对应的最小堆形成的完全二叉树形式如图 5.14 所示。

根据最小堆的定义，堆对应的完全二叉树的根结点具有最小关键码值，即堆所对应序列的第一个元素具有最小值。

在 5.3.2 节已经介绍过用顺序存储结构实现完全二叉树的有效方法，因此可以把最小堆的关键码存储在一维数组中。只需要计算简单的代数表达式，就能够非常容易地查找某个结点的父结点和子结点，既避免了使用指针来保持结构，又能有效地执行相应操作。当然，由于顺序存储的空间分配要求，必须预先知道堆的大小。

比较复杂的堆操作是插入和删除；在此先考虑插入操作的实现。首先，新添的元素加入末尾。为了保持最小堆的性质，需要沿着其祖先的路径，自下而上依次比较和交换该结点与父结点的位置，直到重新满足堆的性质为止。这样做会出现两种情况，要么新结点升到最小堆的顶端，要么到某一位置时发现父结点比新插入的结点关键码值小。例如，在图 5.14 所示的最小堆中插

入元素 13，其过程可以用图 5.15 来表示。

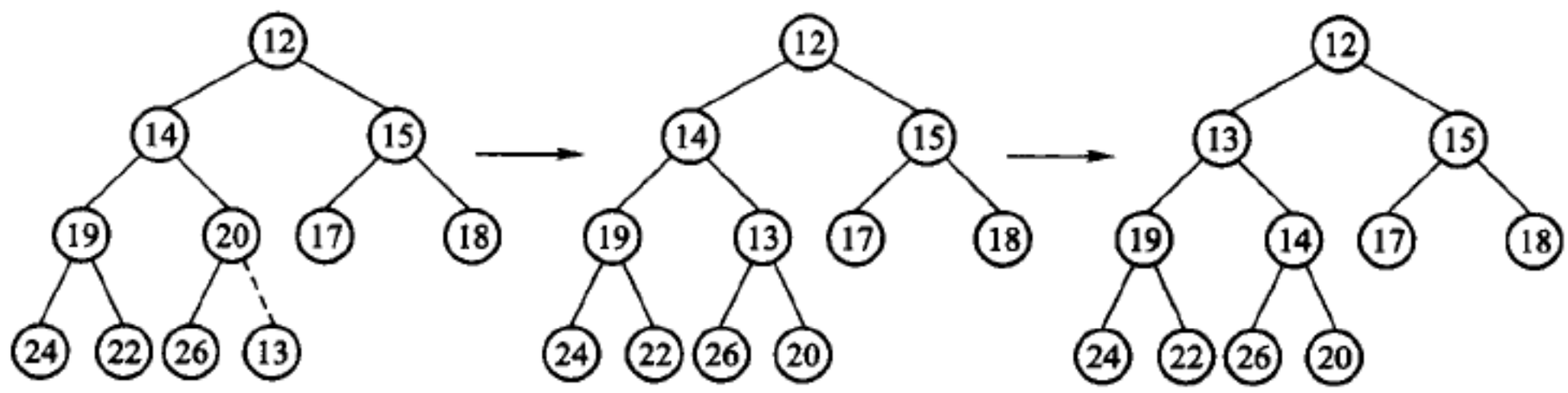


图 5.15 在最小堆 5.14 中插入元素 13

在插入的过程中总是自下而上逐渐上升，最后停在满足最小堆性质的位置，这个过程通常被称为“筛选”。

删除操作的处理与插入时方向相反。考虑到删除某个位置的元素后形成了一个空的位置，首先把最末端的结点填入这个位置。同理，这样做也可能导致破坏最小堆的堆序特性。末端元素需要与被删位置的子结点比较交换，直到过滤到该结点小于最小子结点的正确位置为止。例如，在图 5.14 所示的最小堆中删除元素 14，其过程可以用图 5.16 表示。

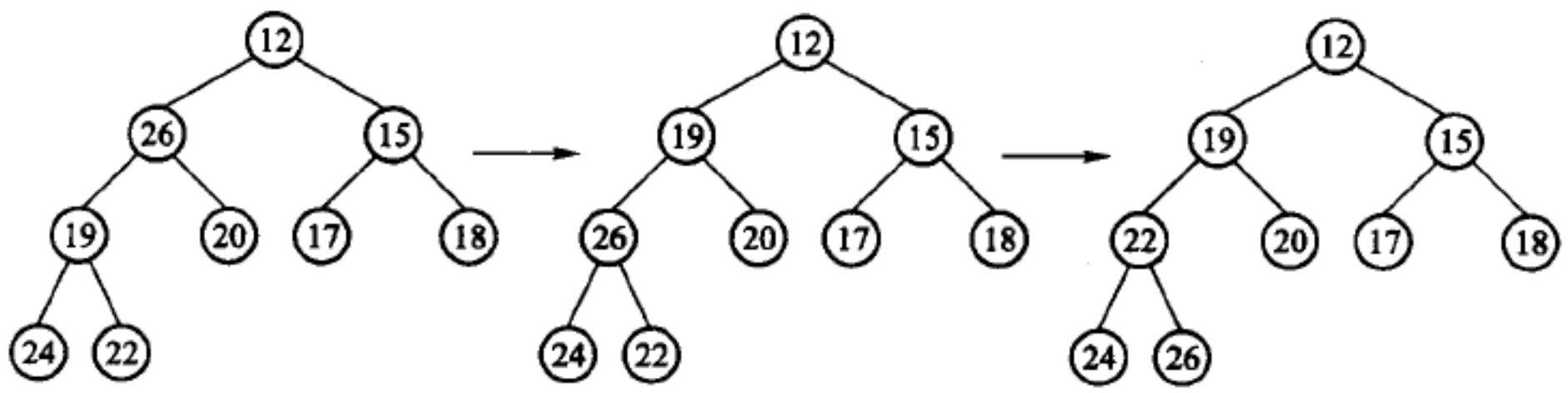


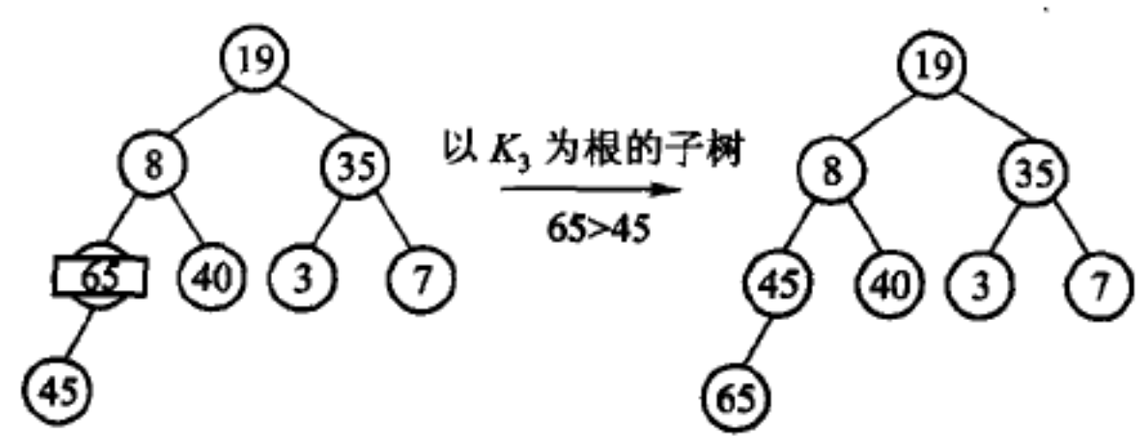
图 5.16 在最小堆 5.14 中删除元素 14

前面介绍了基于最小堆的两个基本操作——插入和删除，下面就介绍一下建立最小堆的方法，用数组作为堆的存储方式。

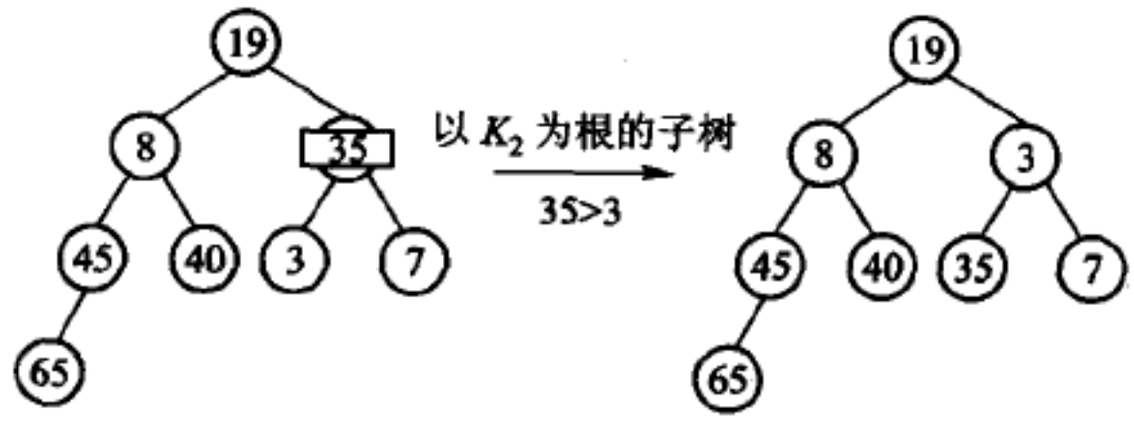
首先，将所有关键码放到一维数组中，此时形成的完全二叉树并不具备最小堆的特性，但是仅包含叶子结点的子树已经是堆，即在有 n 个结点的完全二叉树中，当 $i > \lfloor n/2 \rfloor - 1$ 时，以关键码 K_i 为根的子树已经是堆。这时，从含有内部结点数最少的子树（这种子树在完全二叉树的倒数第二层，此时 $i = \lfloor n/2 \rfloor - 1$）开始，从右至左依次进行调整。对这一层调整完成之后，继续对上一层进行同样的工作，直到整个过程到达树根时，整棵完全二叉树就成为一个堆。

对当前结点 K_i，调整过程是从 K_i 开始向下筛选的过程：如果 $K_i \leqslant K_{2i+1}$ 且 $K_i \leqslant K_{2i+2}$，则以 K_i 为根的子树就已经是堆，不需要调整结点的位置；否则将 K_i 的值与其子结点中关键码小的一个进行交换（不妨假定 $K_{2i+1} < K_{2i+2}$），交换后继续对以 K_{2i+1} 为根的子树进行重建，这样在最坏情况下向下调整到树叶，然后依次调整以 $K_{i-1}, K_{i-2}, \cdots, K_0$ 为根的子树。完成了以 K_0 为根的树的筛选，就完成了建堆的过程。

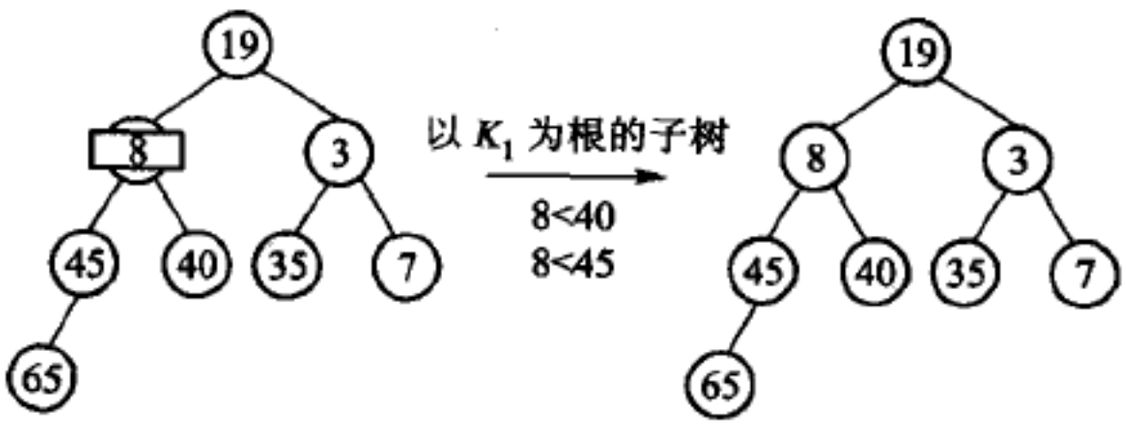
图 5.17 描述了对于关键码集合 $K=\{19,8,35,65,40,3,7,45\}$，用筛选法建堆的过程，其中 $n=8$，$\lfloor n/2 \rfloor -1=3$，所以从 $K_3=65$ 开始调整。



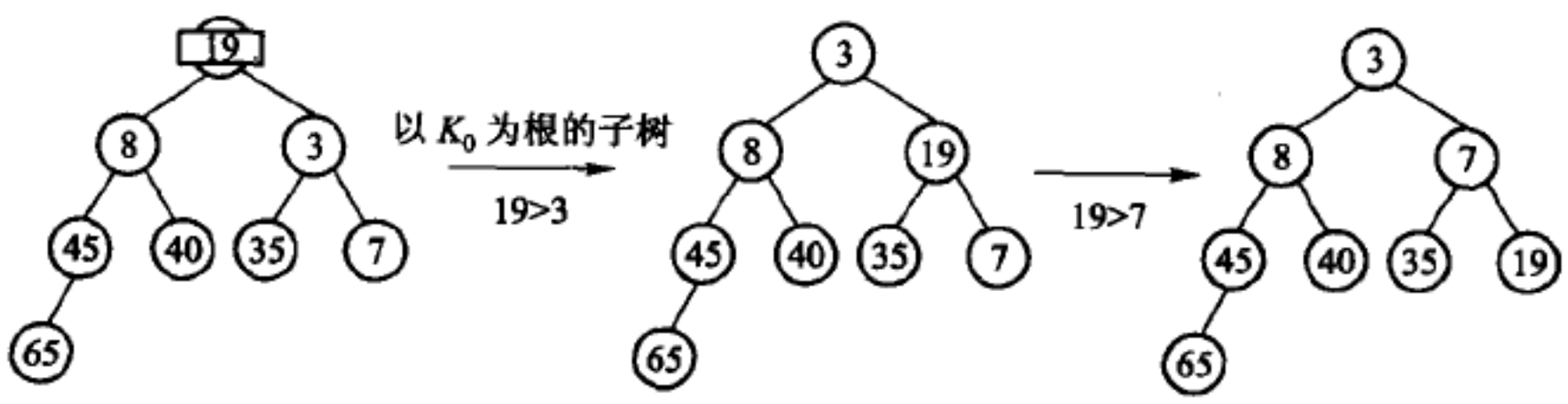
(a) 无序状态并对以 K_3 为根的子树进行调整



(b) 对以 K_2 为根的子树进行调整



(c) 对以 K_1 为根的子树无须进行调整



(d) 对以 K_0 为根的子树进行调整

图 5.17 建堆过程示例

代码 5.11 给出了堆的类定义和部分成员函数的实现。其中包括最小堆的初始化以及在最小堆中插入和删除元素等算法。

【代码 5.11】 堆的类定义和筛选法。

```
template <class T>
class MinHeap {                                    // 最小堆类定义
```

```
private:
    T *heapArray;                                   // 存放堆数据的数组
    int CurrentSize;                                // 当前堆中的元素数目
    int MaxSize;                                    // 最大元素数目
    void swap(int pos_x, int pos_y);                // 交换位置 x 和 y 的元素
    void BuildHeap();                               // 建堆
public:
    MinHeap(const int n);                           // 构造函数,参数 n 为堆的最大元素数目
    virtual ~MinHeap(){delete []heapArray;};        // 析构函数
    bool isEmpty();                                 // 判断堆是否为空
    bool isLeaf(int pos) const;                     // 判断是否为叶结点
    int LeftChild(int pos) const;                   // 返回左孩子位置
    int RightChild(int pos) const;                  // 返回右孩子位置
    int Parent(int pos) const;                      // 返回父结点位置
    bool Remove(int pos, T& node);                  // 删除给定下标的元素
    bool Insert(const T& newNode);                  // 向堆中插入新元素 newNode
    T& RemoveMin();                                 // 从堆顶删除最小值
    void SiftUp(int position);                      // 从 position 开始向上调整
    void SiftDown(int left);                        // 从 left 开始向下筛选
};

template <class T>
MinHeap<T>::MinHeap(const int n) {
    if (n <=0)
        return;
    CurrentSize =0;
    MaxSize = n;                                    // 最大元素数目赋值为 n
    heapArray = new T[MaxSize];                     // 创建堆空间
    // 此处进行堆元素的赋值工作
    BuildHeap();
}
template <class T>
bool MinHeap<T>::isLeaf(int pos) const {            // 判断是否为叶结点
    return (pos >= CurrentSize/2) && (pos < CurrentSize);
}
template <class T>
void MinHeap<T>::BuildHeap() {                      // 建堆
    for (int i = CurrentSize/2 -1; i >=0; i--)
```

```
            SiftDown(i);
}
template < class T >
int MinHeap < T > ::LeftChild(int pos) const {          // 返回左孩子位置
    return 2 * pos + 1;
}
template < class T >
int MinHeap < T > ::RightChild(int pos) const {         // 返回右孩子位置
    return 2 * pos + 2;
}
template < class T >
int MinHeap < T > ::Parent(int pos) const {             // 返回父结点位置
    return (pos - 1)/2;
}
template < class T >
bool MinHeap < T > ::Insert(const T& newNode) {         // 向堆中插入新元素 newNode
    if (CurrentSize == MaxSize)                         // 如果堆已满则返回 FALSE
        return FALSE;
    heapArray[CurrentSize] = newNode;                   // 新元素 newNode 放到堆的末尾
    SiftUp(CurrentSize);                                // 向上调整
    CurrentSize ++;                                     // 堆的当前元素数加 1
    return TRUE;
}
template < class T >
T& MinHeap < T > ::RemoveMin() {                        // 从堆顶删除最小值
    if (CurrentSize == 0) {                             // 判断堆非空
        cout << "Can't Delete";
        exit(1);
    }
    else {
        swap(0, --CurrentSize);                         // 交换堆顶和堆末尾的元素
        if (CurrentSize > 1)                            // 若当前元素数大于 1 则从堆顶开始向下筛选
            SiftDown(0);
        return heapArray[CurrentSize];
    }
}
template < class T >
bool MinHeap < T > ::Remove(int pos, T& node) {         // 删除给定下标的元素
```

```
    if ((pos < 0) || (pos >= CurrentSize))
        return false;
    node = heapArray[pos];                          // 记录删除的元素
    heapArray[pos] = heapArray[--CurrentSize];      // 用最后的元素值替代被删除元素
    if (heapArray[Parent(pos)] > heapArray[pos])// 当前元素小于父结点,需要上升调整
        SiftUp(pos);
    else SiftDown(pos);                             // 当前元素大于父结点,向下筛选
    return true;
}
template <class T>
void MinHeap<T>::SiftUp(int position) {             // 从 position 开始向上调整
    int temppos = position;
    T temp = heapArray[temppos];
    while ((temppos > 0) && (heapArray[Parent(temppos)] > temp)) {
        heapArray[temppos] = heapArray[Parent(temppos)];
        temppos = Parent(temppos);
    }
    heapArray[temppos] = temp;
}
template <class T>
void MinHeap<T>::SiftDown(int left) {               // 从 left 开始向下筛选
    int i = left;                                   // 标识父结点
    int j = LeftChild(i);                           // 用于记录关键值较小的子结点
    T temp = heapArray[i];                          // 保存父结点
    while (j < CurrentSize) {                       // 过筛
        if ((j < CurrentSize - 1) && (heapArray[j] > heapArray[j+1]))     // 若有右子结点
                                                                          // 且小于左子结点
            j++;                                    // j 指向右子结点
        if (temp > heapArray[j]) {                  // 如果父结点大于子结点的值则交换位置
            heapArray[i] = heapArray[j];
            i = j;
            j = LeftChild(j);
        }
        else break;                                 // 堆序性满足时则跳出
    }
    heapArray[i] = temp;
}
```

【代码 5.11 结束】

从上面函数的定义可知，SiftDown( )函数中执行 while 循环的次数不会超过子树的高度，对于由 n 个数据元素组成的堆，SiftDown( )函数的时间复杂度是 $O(\log n)$。而建堆要不断地向下调整，大约 $n/2$ 次调用 SiftDown( )操作，因此建堆的时间复杂度的上界是 $O(n\log n)$。这只是一个粗略的分析，下面讨论建堆的时间开销。

对于 n 个结点的堆，其对应的完全二叉树的层数为 $\log n$。设 i 表示二叉树的层编号，则第 i 层上的结点数最多为 $2^i(i\geqslant 0)$。建堆的过程中，对每一个非叶子结点都调用了一次 SiftDown( )调整算法，而每个数据元素最多向下调整到最底层，即第 i 层上的结点向下调整到最底层的调整次数为 $\log n - i$。因此，建堆的计算时间为

$$\sum_{i=0}^{\log n} 2^i \cdot (\log n - i) \tag{5.3}$$

令 $j = \log n - i$，代入 5.3 式得

$$\sum_{i=0}^{\log n} 2^i \cdot (\log n - i) = \sum_{j=0}^{\log n} 2^{\log n - j} \cdot j = \sum_{j=0}^{\log n} n \cdot \frac{j}{2^j} < 2n \tag{5.4}$$

因此，建堆算法的时间复杂度是 $O(n)$，这就说明可以在线性时间内把一个无序的序列转化成堆序。

但是，最小堆只适合于查找最小值，查找任意值的效率不高。

类似的，也可以定义最大堆。第 8.3.2 节利用最大堆实现了堆排序。

5.5.2 优先队列

优先队列(priority queue)是一种有用的数据结构。它是 0 个或多个元素的集合，每个元素都有一个关键码值，执行的操作有查找、插入和删除等。优先队列的主要特点是支持从一个集合中快速地查找并移出具有最大值或最小值的元素。最小优先队列，适合查找和删除最小元素；最大优先队列，适合查找和删除最大元素。

优先队列的应用很广泛。例如，计算机的操作系统用一个优先队列来实现等候进程的调度管理。在一系列等待执行的进程中，每一个进程可以用一个数值来表示它的优先级，优先级越高，这个值就越小。优先级高的进程应该最先获得处理器。另一个例子是打印机的输出任务队列。对于先后到达的打印几百页和只有几页的任务，一个合理的方法是先打印页数少的任务，后打印页数多的任务，这样做就是按照文件的大小来排列打印任务的优先顺序。

当然，优先序列可以用多种方法实现，例如采用无序线性表或者用有序线性表。虽然都可以查找到最小元素，但是删除或插入的时间代价很大。堆是一种很好的优先队列的实现方法。一旦最小堆建成之后，其堆顶元素就满足关键码最小的要求，为快速查找和删除优先级最高的元素创造条件。进行插入、删除等操作时，也能有效地保持堆的性质，从而确保优先队列的高效性。

5.6 Huffman 树及其应用

Huffman 树又称为最优树，是一种带权路径长度最短的树，有着广泛的应用。

5.6.1 Huffman 树

在 5.1.2 节中介绍了扩充二叉树以及外部路径长度的概念。所谓外部路径长度就是从扩充二叉树的根结点到每一个外部结点的路径长度之和。从这个定义可以得出，在外部结点数相同的情况下，完全二叉树能够使得外部路径长度最小。

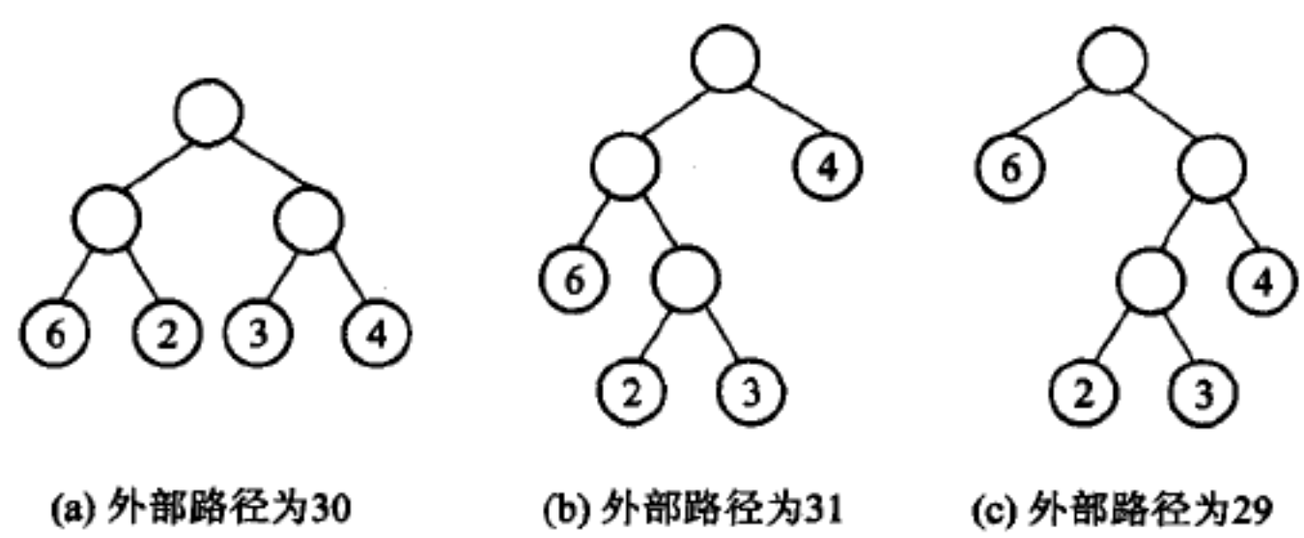
下面把上述定义推广到带权结点的情况。结点的带权路径长度是指从根结点到该结点的路径长度与结点权值的乘积。带权外部路径长度（wighted external path）就是外部结点的带权路径长度之和。

假设有 n 个权值分别为 $w_0, w_1, \cdots, w_{n-1}$（$n \geq 2$）的结点，求带权外部路径长度就是要构造一棵具有 n 个外部结点的扩充二叉树，每一个外部结点 k_i 取 w_i 作为它的权，l_i 表示该外部结点的路径长度，则带权外部路径长度可记作

$$\sum_{i=0}^{n-1} w_i l_i$$

其中，带权外部路径长度最小的二叉树称为 Huffman 树。

例如，图 5.18 中表示了 3 棵具有 4 个外部结点的二叉树，各外部结点的权值分别为 6、2、3、4。



(a) 外部路径为30　　(b) 外部路径为31　　(c) 外部路径为29

图 5.18　具有不同带权外部路径长度的二叉树

它们的带权外部路径长度分别为：

(a) $6 \times 2 + 2 \times 2 + 3 \times 2 + 4 \times 2 = 30$

(b) $6 \times 2 + 2 \times 3 + 3 \times 3 + 4 \times 1 = 31$

(c) $6 \times 1 + 2 \times 3 + 3 \times 3 + 4 \times 2 = 29$

其中，5.18(c)中所示的二叉树外部带权路径长度最小。可以验证，它就是一棵 Huffman 树，

也就是说，这棵树在所有的具有6、2、3、4权值的叶结点的二叉树中带权外部路径长度最小。

可以直观地看出，权值越大的叶离根结点越近，二叉树的带权外部路径长度就越小。在实际应用中，带权外部路径长度小的扩充二叉树应用比较广泛。那么，如何构造一棵Huffman树呢？Huffman最早给出了求具有最小带权外部路径（minimum wighted external path）长度的扩充二叉树的方法。描述如下：

（1）对于给定的 n 个权值 $w_0, w_1, \cdots, w_{n-1}$（$n \geqslant 2$），构成 n 棵二叉树的集合 $T = \{T_0, T_1, T_2, \cdots, T_{n-1}\}$，使得每一棵扩充二叉树只具有一个带权为 w_i 的根结点。

（2）构造一棵新的扩充二叉树，在集合 T 中找出两个权值最小的树作为新树根结点的左右子树，把新树根结点的权值赋为其左右子树根结点的和。

（3）在集合 T 中删除这两棵树，并把得到的新扩充二叉树加入到集合中。

（4）重复步骤（2）、（3）的操作，直到集合 T 中只含有一棵树为止。

例如，图5.19所示为图5.18（c）的Huffman树的构造过程。其中，每一个结点中标注的是所赋的权值。

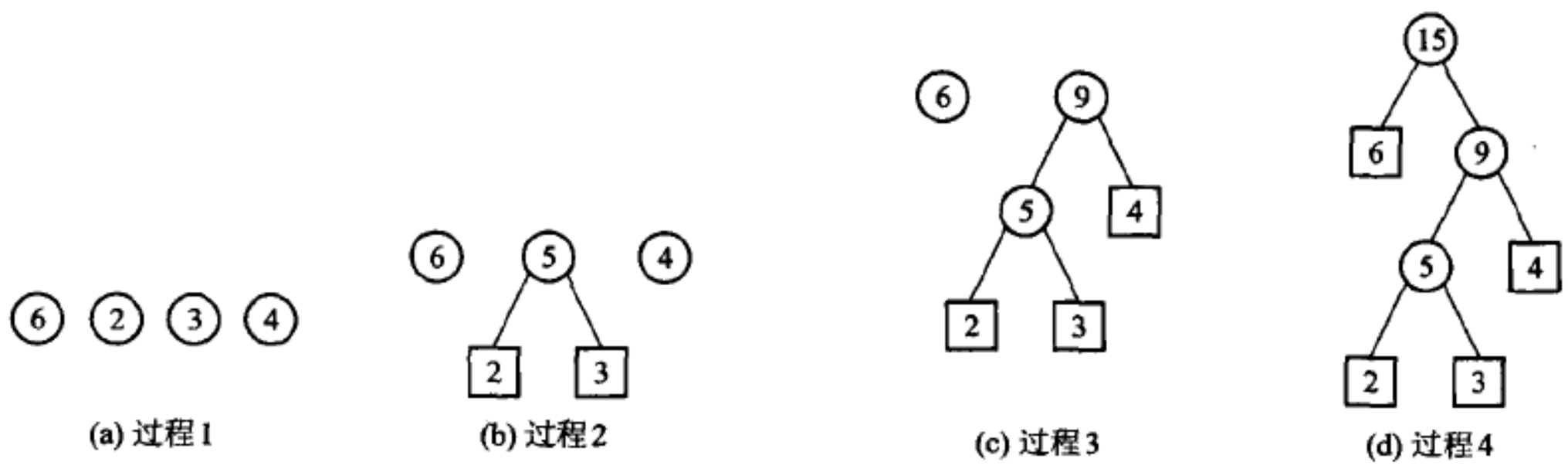


图5.19 Huffman树的构造过程

代码5.12给出了Huffman树的类定义，构造函数实现了Huffman树的构造。其中类HuffmanTreeNode继承自BinaryTreeNode类，只是增加了父指针成员变量。算法使用最小堆选出权值最小的两个结点。

【代码5.12】 Huffman树的类定义。

```
template < class T >
class HuffmanTree  {
private:
    HuffmanTreeNode < T >  * root;                    // Huffman 树的根结点
    void MergeTree ( HuffmanTreeNode < T > &ht1, HuffmanTreeNode < T > &ht2, HuffmanTreeNode
< T >  * parent);    // 把以 ht1 和 ht2 为根的两棵 Huffman 树合并成一棵以 parent 为根的二叉树
    void DeleteTree( HuffmanTreeNode < T >  * root);    // 删除 Huffman 树或其子树
public:
    HuffmanTree( T weight[ ],int n);                    // 构造 Huffman 树，参数 weight 为权值数
```

```
                                                        // 组,n 为数组长度
    virtual ~HuffmanTree() {DeleteTree(root);};         // 析构函数
};

template < class T >
HuffmanTree < T > ::HuffmanTree(T weight[ ], int n) {
    MinHeap < HuffmanTreeNode < T > > heap(n);          // 最小值堆
    HuffmanTreeNode < T > * parent, firstchild, secondchild;
    HuffmanTreeNode < T > * NodeList = new HuffmanTreeNode < T > [n];
    for (int i = 0;i < n;i ++) {                        // 初始化
        NodeList[i]. info = weight[i];
        NodeList[i]. parent = NodeList[i]. left = NodeList[i]. right = NULL;
        heap. Insert(NodeList[i]);                      // 向堆中添加元素
    }
    for (i = 0;i < n - 1;i ++) {                        // 通过 n - 1 次合并建立 Huffman 树
        parent = new HuffmanTreeNode < T >;             // 申请一个分支结点
        firstchild = heap. RemoveMin();                 // 选择权值最小的结点
        secondchild = heap. RemoveMin();                // 选择权值次小的结点
        MergeTree(firstchild,secondchild,parent);       // 将权值最小的两棵树合并到 parent 树
        heap. Insert( * parent);                        // 把 parent 插入到堆中
        root = parent;                                  // Huffman 树的根结点赋为 parent
    }
    delete [ ]NodeList;
}
```

【代码 5.12 结束】

如果把 K 叉树定义为结点的有限集合,它或者为空集,或者由一个根和 K 个有序且不相交的 K 叉树组成,则 Huffman 算法可以推广到 K 叉树。

5.6.2 Huffman 编码

在数据通信系统中,电文传送是经常遇到的问题。传送电文时需要将字符转换成二进制组成的字符串。例如,假设要传送由 a、b、c、d 四个字符组成的电文 abbaaadc,可以将 a、b、c、d 分别编码为 00、01、10、11,则上述电文就转换成了 0001010000001110 进行传送,其总长度为 16,对方可以对接收到的电文按每两个二进制位进行译码。

但是,在传送电文时,总是希望传输信息的总长度尽可能短。这一问题转化成如何对上述字符集设计一套二进制编码,使得采用这种编码进行通信时,信息传输量最小。如果对每一个字符用不同长度的二进制编码,令电文中出现次数多的字符使用尽可能短的编码,则传送数据时总长

度便可以达到最小。例如，同样对电文 abbaaadc，将 a、b、c、d 分别编码为 0、1、01、10，那么传送的电文就转换成 0110001001，总长度只有 10 位。但是，这样的电文无法译码，仅 011 就可以译码为 abb、cb。为了在字符间省去不必要的间隔符号，要求所给出的每一个字符的编码在系统中都能唯一确定。即任一字符的二进制编码不能解释为其他字符编码的前缀，这种编码称为前缀码。

Huffman 树的一个重要应用就是解决数据通信中的二进制编码问题。

假设电文中的字符集为 $D=\{d_0,\cdots,d_{n-1}\}$，D 中各字符出现的频率为 $W=\{w_0,\cdots,w_{n-1}\}$。要对 D 中的字符进行二进制编码，使得通信编码总长最短，即电文总长度 $\sum_{i=0}^{n-1} w_i l_i$ 最小，其中 l_i 为第 i 个字符的二进制编码长度。由此可见，设计电文总长度最短的编码问题就转化成了设计字符出现频率作为外部结点权值的 Huffman 树的问题。

Huffman 编码过程如下：用 $d_0,d_1,\cdots,d_{n-1}$ 作为外部结点构造具有最小带权外部路径长度的扩充二叉树，把从每个结点引向其左子结点的边标上号码 0，从每个结点引向其右子结点的边标上号码 1。从根结点到每个叶结点路径上的编号连接起来就是这个外部结点所代表字符的编码。得到的二进制前缀码称做 Huffman 编码。

例如，在一个数据通信系统中使用的字符是 a、b、c、d、e、f、g，对应的频率分别为 15、2、6、5、20、10、18。利用 Huffman 算法构造以各外部结点频率为权值的 Huffman 树，并把二叉树中每一条到左子树的边标记为 0，到右子树的边标记为 1，得到的最优二叉树如图 5.20 所示。

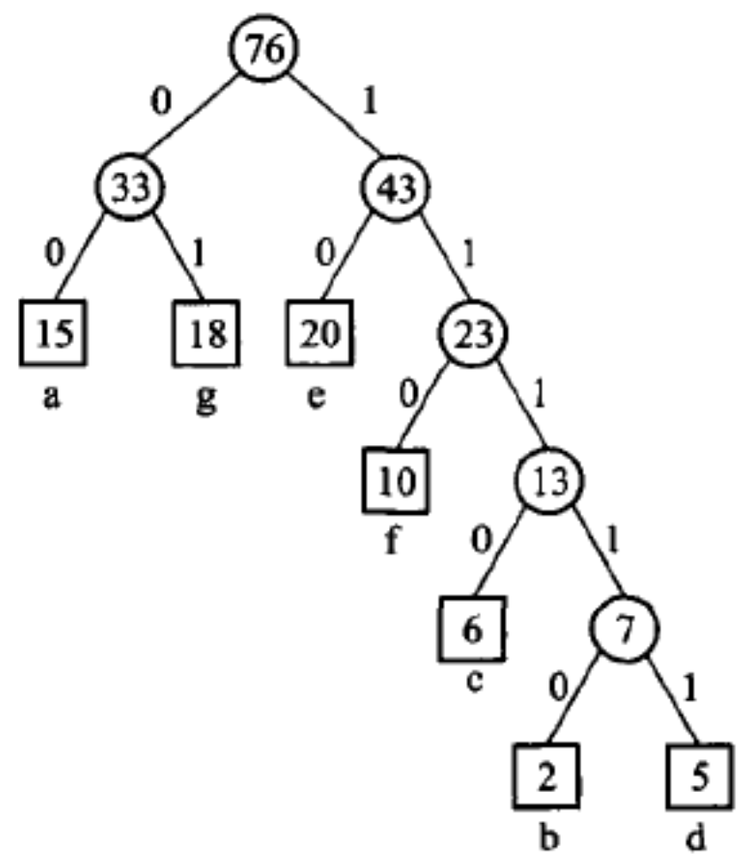


图 5.20 Huffman 编码示例

各字符的二进制编码为：

a:00	b:11110	c:1110	d:11111
e:10	f:110	g:01	

给出了各个字符的 Huffman 编码后，将字符串信息用这种编码来表示就变得很容易。此时，只需要简单地把字母用对应的二进制代码替换即可，这可以通过查表来完成。

用 Huffman 算法构造出的这棵扩充二叉树同时也可以用来译码。译码的含义是分解电文中的字符串，从左到右逐位判别代码串，直至确定每一个字母。由于 Huffman 编码是一种前缀码，因此译码过程不会产成混淆。这一译码过程可用 Huffman 树编码过程的逆过程来实现：从二叉树的根开始，把二进制编码每一位的值与 Huffman 树边上标记的 0、1 相匹配，确定选择左分支还是右分支，直至确定一条到达树叶的路径。一旦到达树叶，就译出一个字符，然后继续用这棵二叉树译出其他二进制编码。

要证明 Huffman 编码可以使得信息传输总量最小，只要证明所构造的二叉树的带权外部路径长度最小即可。严格的证明需要用到图论中的 Huffman 定理，读者可以参考相关的文献，在此不再赘述。从 Huffman 树的构建过程也可以看出，结点的权值越大，它到根结点的路径就越短；权值越小，结点就会放到离根结点较远的位置上。

本章小结

本章介绍了二叉树的概念。与线性表、栈、队列一样，二叉树也是一种重要的数据结构，而且应用十分广泛。

在二叉树中，每个结点最多有两个子结点。由二叉树的逻辑结构给出了二叉树的递归定义：二叉树若不为空，则由根结点和两棵互不相交的子树构成，其左右子树也是二叉树。随后给出了几种特殊的二叉树结构，例如满二叉树、完全二叉树和扩充二叉树。

二叉树周游是指按照一定顺序依次访问树中的所有结点，且使得每一个结点仅被访问一次。根据访问结点和左右子树的次序不同，二叉树的周游算法可以分为深度优先周游和广度优先周游。深度优先周游算法有前序、中序和后序 3 种。在此给出了二叉树深度优先周游的递归和非递归实现。

二叉树的抽象数据类型的各个成员函数的实现需要参考具体的存储结构和应用。二叉树有两种主要的实现方式：链式存储结构和顺序存储结构。用指针实现二叉树是常用的方法，其中每个结点包含数据域和左、右指针域，分别用来存储结点本身的信息和指向左、右子结点的指针。这种存储方式称为二叉链表。顺序存储方式主要介绍了用数组实现完全二叉树，即一棵具有 n 个结点的完全二叉树按层次顺序把所有结点从 0 到 $n-1$ 编号，按编号的次序将所有结点元素存储在一维数组中。

二叉搜索树(BST)是一种重要的索引结构，可以用于快速检索。BST 每个结点关键码的值大于其左子树上所有结点的关键码值，而小于其右子树上所有结点的值，且其左右子树分别为一棵二叉搜索树。因此，按中序周游二叉搜索树可以得到关键码的正序序列。在二叉搜索树中进行检索可以得到较高的效率，动态插入和删除结点时需要保持二叉搜索树的性质。

堆是一种特殊的完全二叉树，常用于实现优先队列。其特殊性主要体现为局部有序性，即最小堆对应的完全二叉树中所有内部结点的值均不大于其左右子结点的关键码值，而一个结点与其兄弟之间没有必然的联系。堆的插入和删除操作必须保持最小堆的堆序性质不变，因此在堆中插入和删除一个元素时需要调整父子结点的位置关系。建堆过程是一个从下至上不断筛选的过程。

结点的带权路径长度是指从根结点到该结点的路径长度与结点权值的乘积。Huffman 树采用贪心法构造带权外部路径长度最小的二叉树，在信息编码中有广泛的应用。

习　题

1. 对于 3 个结点 A、B、C，有多少棵不同的二叉树？试将其画出来。
2. 分别按前序、后序、中序列出图 5.21 所示二叉树的结点。

3. 以下命题是否为真？若真请证明之。

一棵二叉树的所有终端结点(叶结点)，在前序序列、中序序列以及后序序列中都按相同的相对位置出现。

4. 找出所有这样的二叉树，其结点在下列两种次序之下恰好都以同样的顺序出现：

(1) 前序和中序。

(2) 前序和后序。

(3) 中序和后序。

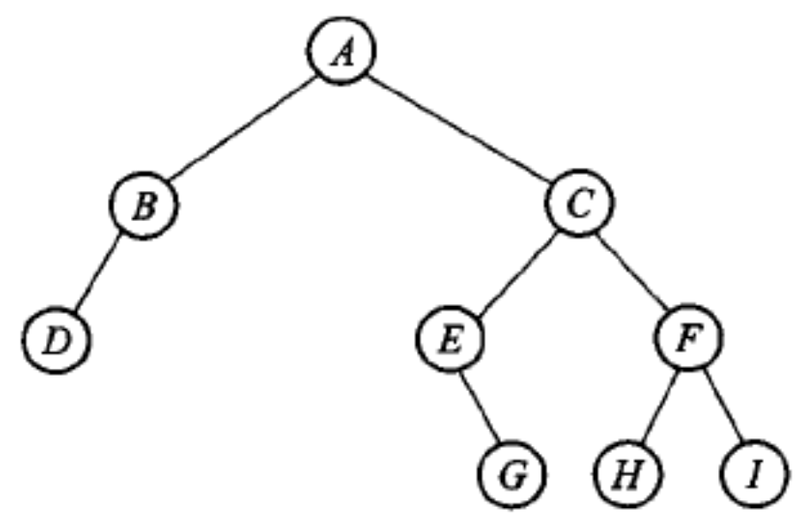


图 5.21 习题 2 的图例

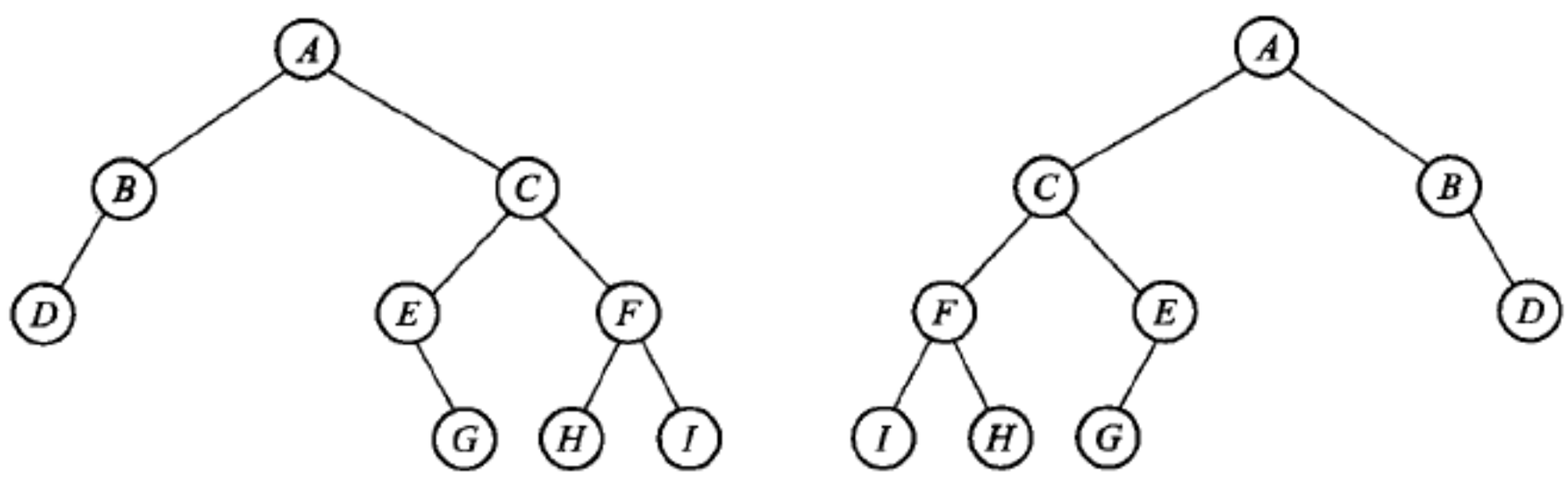
5. 写一个递归函数计算二叉树的叶结点个数。

6. 写一个递归函数计算二叉树的高度(只有一个根结点的二叉树高为 1)。

7. 设计一个镜面影射算法，将一棵二叉树的每个结点的左、右子结点交换位置。例如，将图 5.22(a)所示的二叉树转化为图 5.22 (b)。

8. 给定结点类型为 BinaryTreeNode 的 3 个指针 p、q、rt，假设 rt 为根结点，求距离结点 p 和结点 q 最近的这两个结点的共同祖先结点。

9. 画出图 5.22(a)所示二叉树的二叉链表存储表示图。



(a) 原二叉树　　(b) 镜面映射后的新二叉树

图 5.22 习题 7 的图例

10. 对于 3 个关键码值 A、B、C，有多少个不同的二叉搜索树？试将其画出来。

11. 试证明：二叉搜索树结点的中序序列就是二叉搜索树结点按关键码值排序的序列。

12. 写出从二叉搜索树中删除一个关键码的递归算法。

13. 给出关键码序列{wxw, wxz, wzw, wzy, wzz, yyw, yyx, zww, zwx, zwy, zyw, zyx, zyy, zyz}。从空二叉搜索树(BST)开始，按照上述关键码出现的顺序依次插入，画出插入所有结点后的 BST。

14. 编写一个递归函数 search()，传入参数为一棵二叉树(不是二叉搜索树 BST)和一个值 K，如果值 K 出现在树中则返回 true，否则返回 false。相应的，写出一个等价的非递归函数 search()。

15. 编写一个递归函数 smallcount()，传入一棵二叉搜索树(BST)的根和值 K，返回值小于或等于 K 的结点数目。函数 smallcount()应尽可能少地访问 BST 的结点。相应的，写出一个等价的非递归函数 smallcount()。

16. 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。

17. 编写一个函数 IsBST，传入参数为一棵二叉树，如果该二叉树是 BST 则返回 true，否则返回 false。

18. 编写一个函数 IsMinHeap()，传入参数为一个数组，如果该数组中的值满足最小堆的定义则返回 true，否

则返回 false。

19. 初始关键码序列为{E,D,X,K,H,L,M,C,P},试给出用筛选法所建的最小堆,并写出其相应的序列。在建堆过程中,移位次数是多少?

提示:一次移位就是对关键码的一次赋值操作。例如,交换 Arr[i]与 Arr[j]的操作是 3 次移位:tmp = A[i],A[i] = A[j],A[j] = tmp。不考虑下标变量的赋值(例如 i = 100 不算一次赋值)。

20. 对于给出的一组权 $W = \{1,4,9,16,25,36,49,64,81,100\}$,构造具有最小带权外部路径长度的扩充二叉树,并求出它的带权外部路径长度。

21. 给定一组权 $W_0, W_1, \cdots, W_{n-1}$,说明怎样构造一个具有最小带权外部路径长度的扩充的 k 叉树。试对权集 1,4,9,16,25,36,49,64,81,100 来具体构造一个这样的扩充三叉树。

22. 任何包括 n 个结点的二叉树的二叉链表表示中,$2n$ 个指针中都只有 $n-1$ 个用来指示结点的左右子结点,而另外 $n+1$ 个为空。可以用指向结点按某种次序周游的前驱和后继的指针来代替这些空的指针。这种附加的指针称做"线索",加进了线索的二叉链表称做穿线二叉树(thread binary tree)。由于有了线索的存在而使得周游二叉树和找结点在指定次序下的前驱、后继的算法变得很简单。例如,图 5.23 就是一个中序穿线树。

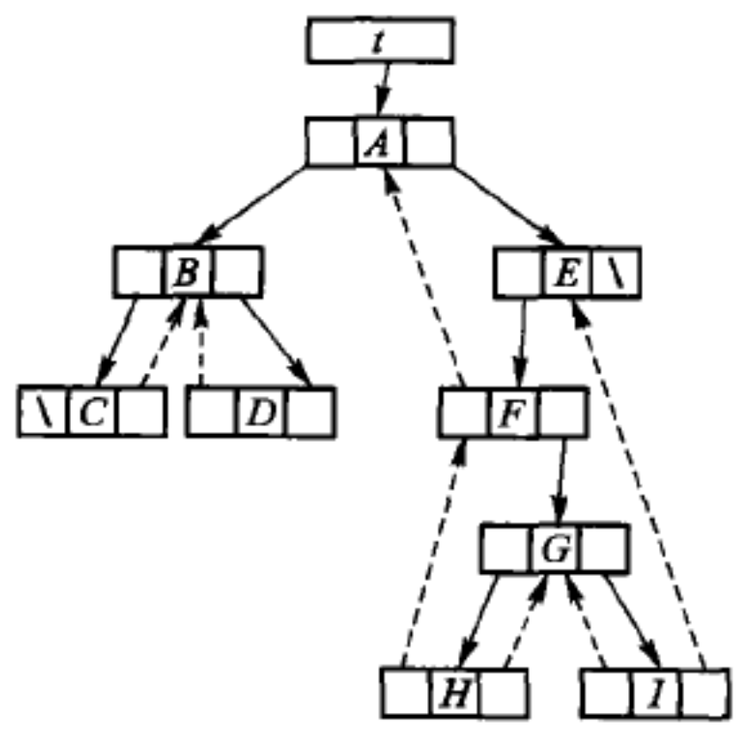


图 5.23 中序穿线二叉树

在穿线树中如果结点有左子树,那么它的左指针 left 就指向其左孩子,否则 left 指向其前驱结点;如果结点有右子树,那么它的右指针 right 就指向其右孩子,否则 right 就指向其后继结点。为区分指针和线索,需要在每个结点中增加两个标志位,分别指示左右指针位置中存的是指针还是线索。以下给出了穿线二叉树结点的结构。

left	ltag	info	rtag	right

试给出将二叉树以中序遍历使其变成穿线树的线索化算法。

上 机 题

1. 二叉树两个结点间的最小距离。

定义二叉树两个结点间的最小距离为:这两个结点的最近公共祖先结点分别到这两个结点的路径长度之和。试设计一种方法,找出给定二叉树中任意两个结点之间的最小距离,可以考虑以图形显示。

2. 表达式二叉树。

表达式可以用表达式二叉树来表示。对于简单的四则运算表达式,请实现以下功能:

(1) 对于任意给出的前缀表达式(不带括号)、中缀表达式(可以带括号)或后缀表达式(不带括号),能够在计算机内部构造出一棵表达式二叉树,并且以图示显示出来(字符图或图形的形式)。

(2) 对于构造好的内部表达式二叉树,按照用户的要求,输出相应的前缀表达式(不带括号)、中缀表达式(可以带括号,但不允许冗余括号)或后缀表达式(不带括号)。

提示:所谓中缀表达式中的冗余括号,就是去掉该括号不影响表达式的计算顺序。例如,"(c + b) + a"中的

括号是冗余的,可以表示成不冗余的"c + b + a"。

3. 逻辑表达式推演

一个逻辑表达式如果对于其变元的任一种取值都为真,则称为重言式;反之,如果对于其变元的任一种取值都为假,则称矛盾式;否则,既非重言式也非矛盾式。编写程序通过真值表判断一个逻辑表达式属于哪一类。

(1) 逻辑运算符包括"|"、"&"、"~",分别标识逻辑或、与、非,表达式包含括号。

(2) 如果是既非重言式,也非矛盾式,试按照用户给定变元的取值(可选)显示逻辑表达式的值。

(3) 按照用户的要求可以打印真值表。

4. Huffman 编码

完成根据代码 5.12 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。

5. 优先队列

利用最大堆实现一个优先队列。对于队列的操作应该至少支持下列几种指令:

```
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。需要建立一种机制来获取所需对象在堆中的位置,还需要对堆的实现进行修改,以存储对象在数组中的位置,以便在辅助数组结构中记录针对堆中对象的修改。

第6章　树

树形结构是结点之间以分支关系定义的层次结构。作为一种重要的非线性结构，树形结构中一个结点至多有一个前驱结点，但可以有多个后继结点。树形结构是客观世界中很多事物与关系的抽象模型，例如人类社会的亲属关系可以按层次表示成家谱树；公司、学校等社会机构也按层次化的方式组织。在计算机科学领域中，树形结构也无处不在，例如在编译程序中，用树表示源程序的句法结构；在数据库系统中，用树组织信息；在文件系统中，用树结构表示目录等。

不同于第5章二叉树只有左右两个子结点，树结点可以包括任意数目的子结点。本章将详细讨论树和森林的概念、存储结构以及树的相关操作等内容。

6.1　树的定义和基本术语

6.1.1　树和森林

树的定义如下：

树(tree)是包括 n 个结点的有限集合 $T(n\geqslant 1)$，使得：

(1) 有且仅有一个特定的称为根(root)的结点。

(2) 除根以外的其他结点被分成 m 个($m\geqslant 0$)不相交的有限集合 $T_1,T_2,\cdots,T_m$，而每一个集合又都是树。其中，树 $T_1,T_2,\cdots,T_m$ 称做这个根的子树(subtree)。

这个定义是递归的，即在树的定义中又用到了树的概念。一棵树可以分成几个分支(称做子树，subtree)，而每个分支也都是一棵树。图6.1就是一棵树的示例，这棵树由根结点 A 和分别以结点 B、C 为根的子树组成。

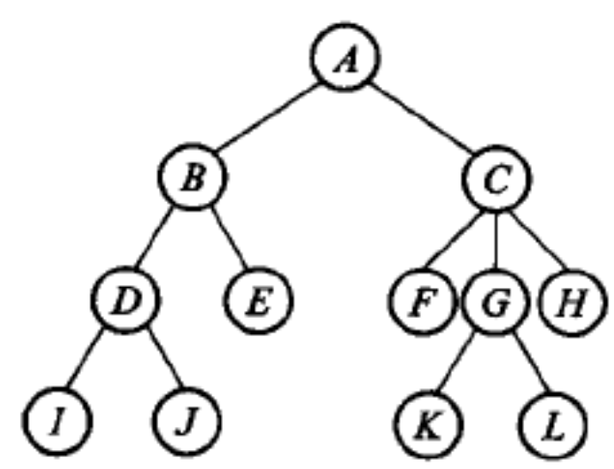


图6.1　树形表示法

树的逻辑结构可以这样描述：树是包含 n 个结点的有穷集合 $K(n>0)$，且在 K 上定义了一个满足以下条件的二元关系 $R=\{r\}$：

(1) 有且仅有一个结点 $k_0\in K$，它对于关系 r 来说没有前驱。结点 k_0 称做树的根。

(2) 除结点 k_0 外，K 中的每个结点对于关系 r 来说都有且仅有一个前驱。

(3) 除结点 k_0 外的任何结点 $k \in K$,都存在一个结点序列 $k_0, k_1, \cdots, k_s$,使得 k_0 就是树根,且 $k_s = k$,其中有序对 $<k_{i-1}, k_i> \in r(1 \leqslant i \leqslant s)$。这样的结点序列称为从根 k_0 到结点 k 的一条路径。

从图 6.1 所表示的树中抽象出来的逻辑结构是:

结点集合 $K = \{A, B, C, D, E, F, G, H, I, J, K, L\}$

K 上的关系 $r = \{<A,B>, <A,C>, <B,D>,$
$<B,E>, <C,F>, <C,G>, <C,H>,$
$<D,I>, <D,J>, <G,K>, <G,L>\}$

除了定义的角度不同外,以上给出的树的逻辑结构描述和树的递归定义是等价的。

树的基本术语,例如父结点、子结点、根、树叶、子孙、路径、层次等,都类似于二叉树中的相关概念。在一棵树中,若存在结点 k 指向结点 k' 的连线,则称 k 是 k' 的父结点,而 k' 则是 k 的子结点,有向连线 $<k, k'>$ 称做边。同一个父结点的子结点之间互称兄弟。树中没有父结点的结点称为根;没有子结点的结点称为树叶。结点的子树数目称为结点的度,树的度是树中各结点度的最大值,二叉树的度是 2。若树中存在结点序列 $k_0, k_1, \cdots, k_s$,使得 $<k_0, k_1>, <k_1, k_2>, \cdots, <k_{s-1}, k_s>$ 都是树中的边,则称从结点 k_0 到结点 k_s 存在一条路径。若有一条由 k 到达 k_s 的路径,则称 k 是 k_s 的祖先,k_s 是 k 的子孙。结点的层数同样是由树根开始定义的,根结点为第 0 层,非根结点的层数是其父结点的层数加 1。

自然界中树的子结点次序没有必要加以区别,称为无序树。但计算机的存储是有序的,为方便计算机处理,往往把子结点按从左到右的次序顺序编号,即把树作为有序树(ordered tree)看待。

注意:度为 2 的有序树并不是二叉树,因为有序树中在第一子结点被删除后,第二子结点自然顶替成为第一子结点。因此,度为 2 并且严格区分左右两个子结点的有序树才是二叉树。

森林(forest)是零棵或多棵不相交的树的集合(通常是有序集合)。对于树中的每个结点,其子树组成的集合就是森林;而加入一个结点作为根,森林就可以转化成一棵树。

树形结构在客观世界中是大量存在的,有多种逻辑表示方法,例如树形表示法、凹入表示法、文氏图表示法、嵌套括号表示法等。

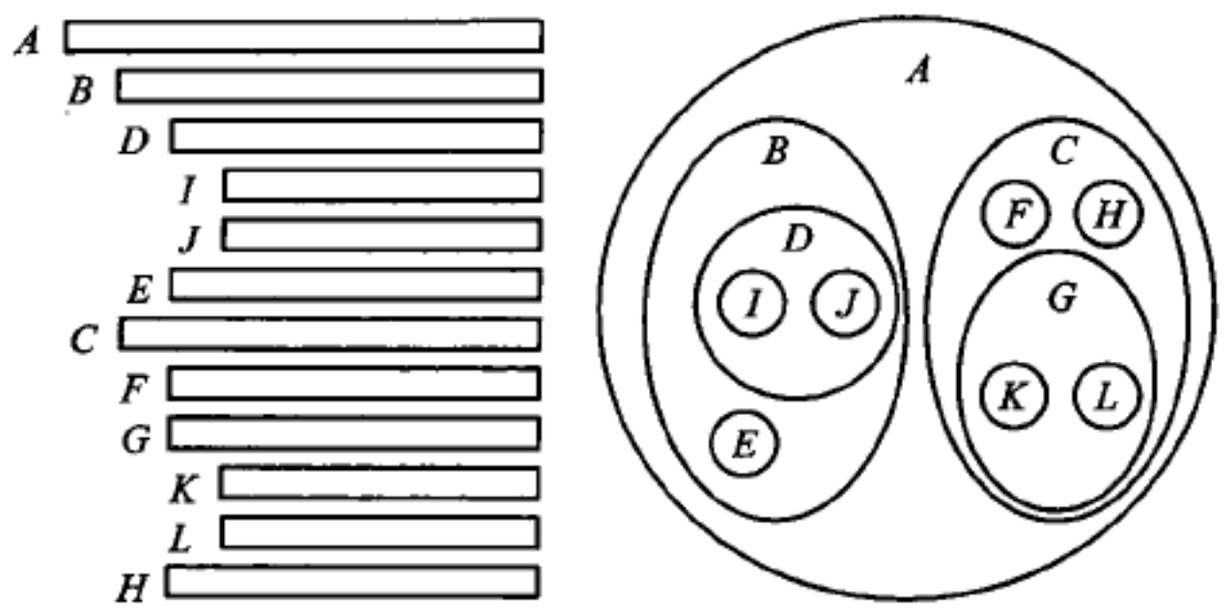
树形表示法其实是根在上的倒挂树,用圆圈结点表示树的数据元素,用连接两个结点的边表示数据元素的关系。尽管树中的边没有标明方向,但是一般默认上面的结点表示前驱,下面的数据元素表示该结点的后继。图 6.1 就用树形表示法表示了树的层次结构。

凹入表示法表示出根结点及各个子树的层次关系,线条表示各结点。树根结点对应的线条最长,各子结点对应的线条短于其父结点并置于父结点线条的下方。各兄弟结点对应的线条长度相同。这种表示法类似于图书的目录表。图 6.2(a)用凹入表示法表示图 6.1 中的树。

文氏图表示法用嵌套的圆表示树。每棵树对应于一个圆,树的根结点及其子树画在同一个圆内,同一个根结点的任意两棵子树对应的圆互不相交。图 6.2(b)是图 6.1 中树的文氏图表示法。

嵌套括号表示法是类似于广义表的一种表示方法。括号表示层次关系,每棵树对应于一个以根结点作为表名的表,根结点写在左边,表由子树森林对应的表组成,各个子树对应的表用逗

号隔开。图 6.2(c)是图 6.1 中树的嵌套括号表示法。



(a) 凹入表表示法　　(b) 文氏图表示法

(A(B(D(I,J),E), C(F,G(K,L),H)))

(c) 嵌套括号表示法

图 6.2 树形结构的各种表示法

树的表示方法的多样性,说明了树在现实世界中的重要性。树结构可以用于对真实世界的很多问题进行数学建模,因此成为计算机科学中常用的问题求解工具。

6.1.2 森林与二叉树的等价转换

树与二叉树、森林与二叉树之间可以相互转化,而且这种转换是一一对应的。树和森林转化成二叉树后,森林或树的相关操作都可以转换成对二叉树的操作。

树和森林到二叉树的转换过程可用连线、切线、旋转"三步曲"来说明:

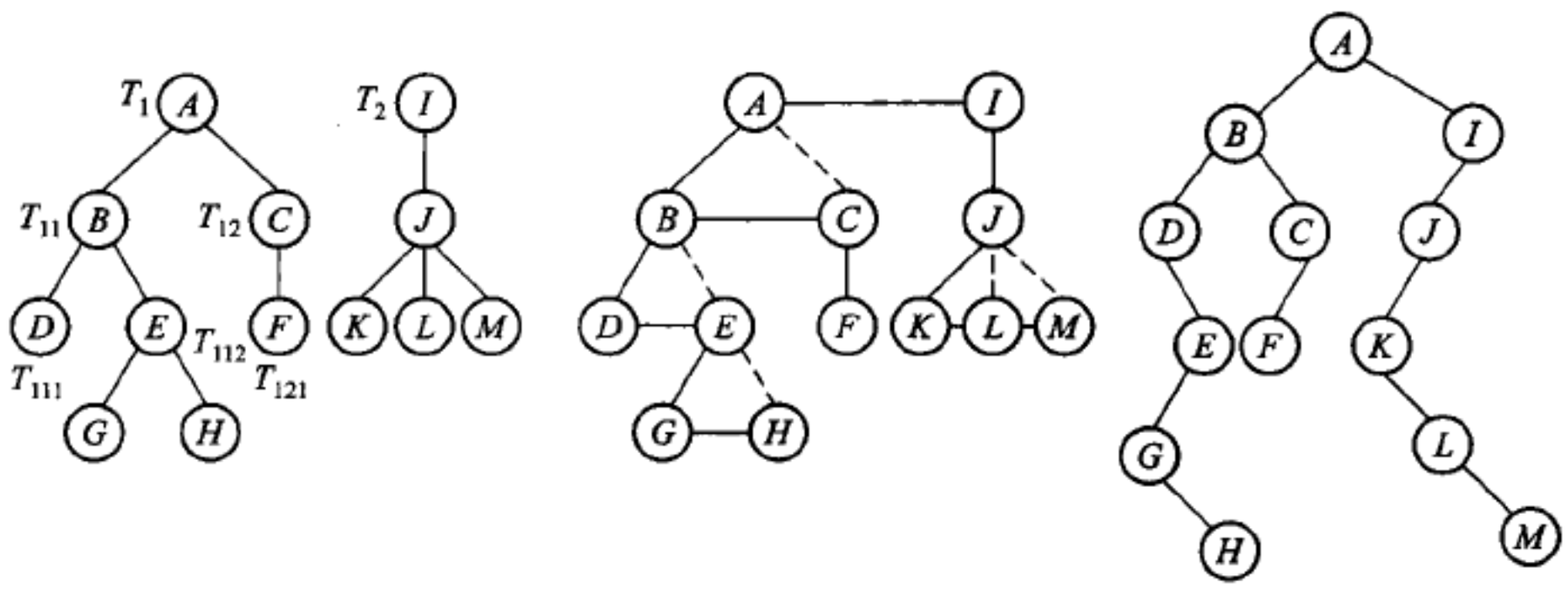
- 连线:将兄弟结点用线连接起来。
- 切线:保留父结点与其第一个子结点的连线,将父结点到其他子结点的连线切掉。
- 旋转:以根为轴,平面向下顺时针方向旋转一定的角度。旋转只是为了调整画面,使得转化后的二叉树看起来比较规整。

可以证明,树和森林做这样的转换所构成的二叉树是唯一的。图 6.3 给出了将森林转换为二叉树的转换过程示意图。图 6.3(a)是由 T_1 和 T_2 两棵树构成的森林,经过连线、切线处理得到图 6.3 (b),旋转操作结束就得到图 6.3(c)所示的二叉树。

在对应的二叉树里,一个结点的左子结点是它在原树(或森林)中的第一个子结点,右子结点是它在原来树(或森林)中的下一个兄弟。森林或树转化成二叉树的形式定义如下:

设有序集合 $F=\{T_1,T_2,\cdots,T_n\}$ 表示由树 $T_1,T_2,\cdots,T_n$ 组成的森林,则森林 F 可以按如下规则递归转换成二叉树 $B(F)$:

(1) 若 F 为空,即 $n=0$,则 $B(F)$ 为空。



(a) 由T_1、T_2构成的森林　(b) 经连线、切线处理后的二叉树　(c) 进行旋转后的二叉树

图 6.3 森林转换为二叉树

(2) 若 F 非空，即 $n>0$，则 $B(F)$ 的根是森林中第一棵树 T_1 的根 W_1，$B(F)$ 的左子树是树 T_1 中根结点 W_1 的子树森林 $F=\{T_{11},T_{12},\cdots,T_{1m}\}$ 转换成的二叉树 $B(T_{11},T_{12},\cdots,T_{1m})$；$B(F)$ 的右子树是从森林 $F'=\{T_2,\cdots,T_n\}$ 转换而成的二叉树。

例如，对于图 6.3(a)的森林 $F=\{T_1,T_2\}$，取 T_1 的根 A 作 $B(F)$ 的根，$B(F)$ 的左子树为 $B(T_{11},T_{12})$，再取 T_{11} 的根 B 为 $B(F)$ 的左子树的根。结点 B 的左子树不为空，取其子树 T_{111} 的根 D 作为结点 B 的左子树的根。D 的左子树为空，D 的右子树为 $B(T_{112})$，于是取 T_{112} 的根结点 E 作 D 的右子树的根，依次进行处理。B 的右子树为 $B(T_{12})$，取 T_{12} 的根结点 C 作结点 B 的右子树的根。C 的左子树为 $B(T_{121})$，于是取 T_{121} 的根 F 作 C 的左子树的根，C 的右子树为空。类似地构造 $B(F)$ 的右子树。构造完成之后，就得到了如图 6.3(c)所示的二叉树形式。

通过转换过程可以看出，森林中的任意一个结点都对应于二叉树中的一个结点。森林中某结点的第一个子结点在二叉树中是相应结点的左子结点，森林中某结点的右兄弟结点在二叉树中是相应结点的右子结点。也就是说，在二叉树中，左分支上的各结点在原来的树中是父子关系，而右分支上的各结点在原来的树或森林中是兄弟关系。对于单棵树而言，由于树的根结点没有兄弟，因此变换后的二叉树的根结点的右子结点必然为空。

反过来，自然也可以把二叉树转换为树或森林：形象地说是上面三步的逆操作。

- 旋转：以根为轴，平面逆时针方向旋转。
- 补线：若结点 x 是父结点 y 的左子结点，则把 x 的右子结点、右子结点的右子结点，依次类推，直到最右子结点，用连线与 y 连起来。
- 删线：去掉所有父结点到右子结点的连线。

图 6.4(a)所示的二叉树用这样的方法处理就转换成了图 6.4(b)所示的森林。

设 B 是一棵二叉树，root 是 B 的根，B_L 是 root 的左子树，B_R 是 root 的右子树，则对应于二叉树 B 的森林或树 $F(B)$ 的形式定义如下：

(1) 若 B 为空，则 $F(B)$ 是空的森林。

(2) 若 B 不为空，则 $F(B)$ 是一棵树 T_1 加上森林 $F(B_R)$，其中树 T_1 的根为 root，root 的子树

为 $F(B_L)$。

可以按照此定义确定从图 6.4(a)的二叉树到图 6.4(b)的森林的转换。树 T_1 是由根结点 A 以及二叉树中由 B、C 和 D 组成的左子树 $F(B_L)$ 构成。结点 A 作为树 T_1 的根结点，继而将结点 B、C 和 D 组成的二叉树 B_L 转化成 A 的子树森林 $F(B_L)$。结点 A 的第一棵子树是结点 B，A 的第二棵和第三棵子树分别是结点 C 和 D。同理，也可以把以结点 E 为根的二叉树 B_R 转化成森林 $F(B_R)$。这样就构成了如图 6.4(b)所示的森林。

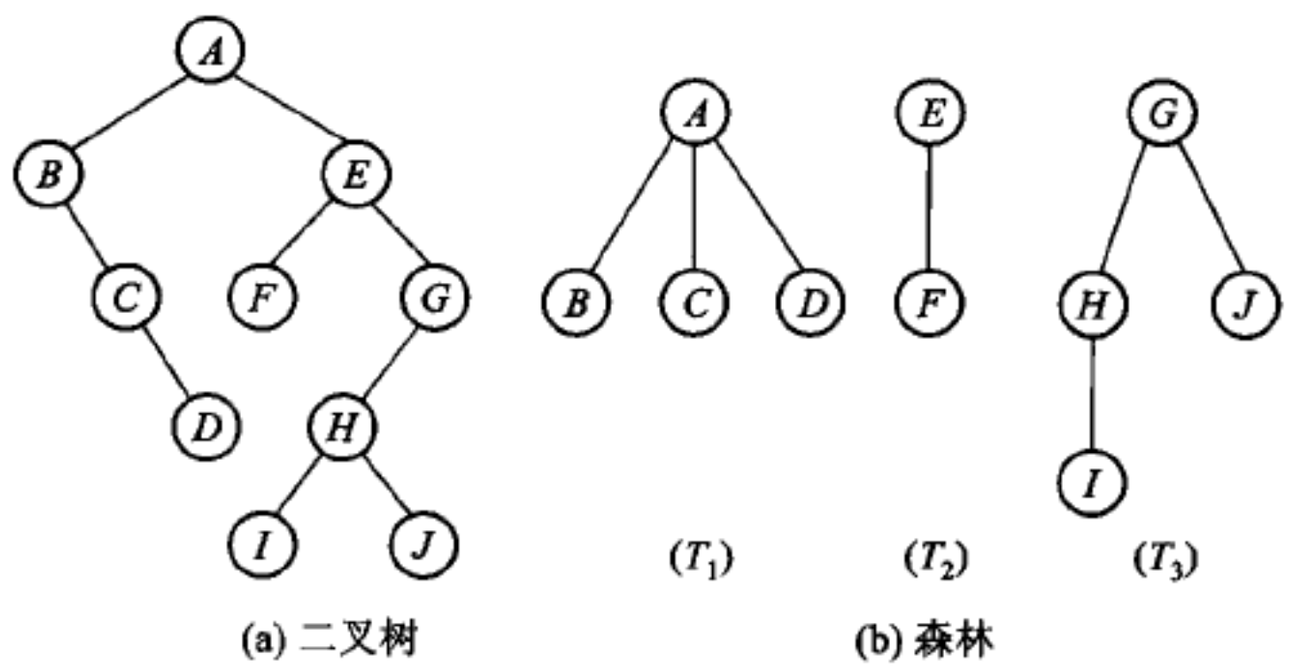


图 6.4 二叉树转换为森林

6.1.3 树的抽象数据类型

类似于二叉树，树的抽象数据类型包括结点类和树类，分别由代码 6.1 和代码 6.2 给出。抽象数据类型中没有给出类中成员函数的具体实现。后面的几节将针对具体问题中树的实现方法给出其中一些重要公有函数的实现，完整的实现代码可参考前言提到的课程网站。

【代码 6.1】 树结点的抽象数据类型。

```
template < class T >
class TreeNode  {
public:
    TreeNode(const T& value);                          // 拷贝构造函数
    virtual ~TreeNode() {};                            // 析构函数
    bool isLeaf();                                     // 判断当前结点是否为叶结点
    T Value();                                         // 返回结点的值
    TreeNode < T > * LeftMostChild();                  // 返回第一个左子结点
    TreeNode < T > * RightSibling();                   // 返回右兄弟
    void setValue(const T& value);                     // 设置当前结点的值
    void setChild(TreeNode < T > * pointer);           // 设置左子结点
    void setSibling(TreeNode < T > * pointer);         // 设置右兄弟
    void InsertFirst(TreeNode < T > * node);           // 以第一个左子结点身份插入结点
```

```
        void InsertNext(TreeNode<T> * node);                    // 以右兄弟的身份插入结点
    };
```

【代码6.1结束】

【代码6.2】 树的抽象数据类型。

```
    template<class T>
    class Tree  {
    public:
        Tree();                                                 // 构造函数
        virtual ~Tree();                                        // 析构函数
        TreeNode<T> * getRoot();                                // 返回树中的根结点
        void CreateRoot(const T& rootValue);                    // 创建值为 rootValue 的根结点
        bool isEmpty();                                         // 判断是否为空树
        TreeNode<T> * Parent(TreeNode<T> * current);            // 返回当前结点的父结点
        TreeNode<T> * PrevSibling(TreeNode<T> * current);       //返回当前结点的前一个兄弟
        void DeleteSubTree(TreeNode<T> * subroot);              // 删除以 subroot 为根的子树
        void RootFirstTraverse(TreeNode<T> * root);             // 先根深度优先周游树
        void RootLastTraverse(TreeNode<T> * root);              // 后根深度优先周游树
        void WidthTraverse(TreeNode<T> * root);                 // 广度优先周游树
    };
```

【代码6.2结束】

6.1.4 树的周游

由树和森林的定义可以引出两种周游树或森林的方法,既可以按深度的方向周游树和森林,也可以按广度的方向周游树和森林。

1. 深度优先周游树

对于树或森林,一个结点可能具有多于两个子树,因此不能像二叉树的中序周游法那样给出树的中根次序周游方式,但是仍然可以考虑树或森林的前序和后序周游算法。

仿照周游二叉树深度优先算法的前序法和后序法可以类似地定义周游树和森林的先根次序和后根次序周游。

先根次序周游森林:

(1) 访问森林中第一棵树的根结点。

(2) 在先根次序下周游第一棵树根结点的子树森林。

(3) 在先根次序下周游其他的树构成的森林。

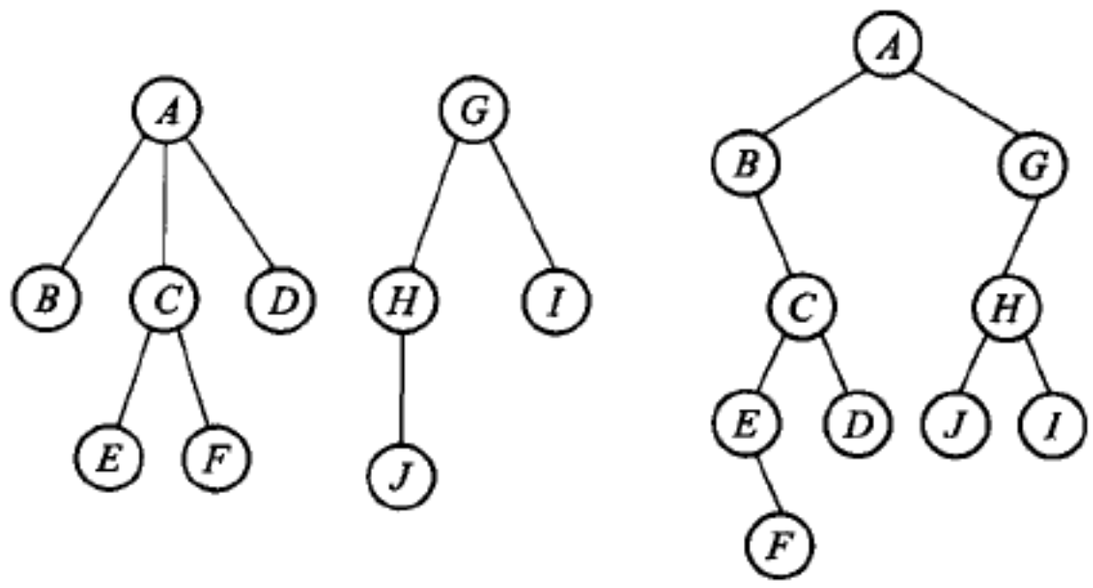
后根次序周游森林:

(1) 在后根次序下周游第一棵树根结点的子树森林。

(2) 访问森林中第一棵树的根结点。

(3) 在后根次序下周游其他树构成的森林。

例如,图 6.5 (a)所示的森林,按先根次序周游得到的结点序列是



(a) 森林　　(b) 森林对应的二叉树

图 6.5　森林和对应的二叉树

ABCEFDGHJI

注意:按先根次序周游森林的序列正好等同于其对应二叉树的前序序列。

按后根次序周游得到的结点序列是

BEFCDAJHIG

注意:后根次序周游得到的排列正好是该森林对应的二叉树在中序次序周游下的结点序列。事实上,由森林与二叉树的转化关系可知,森林 F 的第一棵树 T_1 的诸子树 $T_{11},T_{12},\cdots,T_{1m}$ 对应 $B(F)$ 的左子树,而 F 中其余的树 $T_2,\cdots,T_n$ 对应于 $B(F)$ 的右子树。森林的后根次序周游过程是,先周游 $\{T_{11},T_{12},\cdots,T_{1m}\}$,再访问 T_1 的根结点,最后周游 $\{T_2,\cdots,T_n\}$,这恰好是二叉树 $B(F)$ 的中序周游过程。

图 6.5(b)是图 6.5(a)中的森林所对应的二叉树,若分别按照前序和中序周游这棵二叉树,可以分别得到与上述先根和后根周游树相同的遍历序列。

算法 6.3 和算法 6.4 分别给出了先根和后根深度优先周游树或森林的递归算法。

【算法 6.3】 先根深度优先周游树或森林。

```
template < class T >
void Tree < T > ::RootFirstTraverse(TreeNode < T > * root) {
    while (root ! = NULL) {
        Visit(root -> Value());                          // 访问当前结点
        RootFirstTraverse(root -> LeftMostChild());      // 周游第一棵树树根的子树
        root = root -> RightSibling();                   // 周游其他的树
    }
}
```

【算法 6.3 结束】

【算法 6.4】 后根深度优先周游树或森林。

```
template < class T >
void Tree < T > ::RootLastTraverse( TreeNode < T > * root) {
    while ( root ! = NULL) {
        RootLastTraverse( root -> LeftMostChild( ) ) ;        // 周游第一棵树树根的子树
        Visit( root -> Value( ) ) ;                           // 访问当前结点
        root = root -> RightSibling( ) ;                      // 周游其他的树
    }
}
```

【算法 6.4 结束】

2. 广度优先周游

广度优先(breadth - first)周游也称"宽度优先周游",或者"层次周游",从树的第0层(根结点)开始,自上至下逐层周游;在同一层中,则按照从左到右的顺序对结点逐一访问。

按广度的方向周游如图 6.5(a)所示的森林,得到的结点序列是

AGBCDHIEFJ

按广度优先周游图 6.5 (a)中的第一棵树,得到的结点序列是

ABCDEF

注意:图 6.5(a)中森林的层次序列与图 6.5 (b)所示二叉树的层次次序不同。按广度方向周游森林对应的二叉树时,不是简单地按算法 5.7 那样按照离二叉树根从近到远地遍历,而是沿着二叉树右链访问的过程,左指针起到承上启下的作用。

算法 6.5 给出的是广度优先周游树或森林的算法。

【算法 6.5】广度优先周游树或森林。

```
template < class T >
void Tree < T > ::WidthTraverse( TreeNode < T > * root) {
    using std::queue;                                         // 使用 STL 队列
    queue < TreeNode < T > * > aQueue;
    TreeNode < T > * pointer = root;
    while ( pointer ! = NULL) {
        aQueue. push( pointer) ;                              // 当前结点进入队列
        pointer = pointer -> RightSibling( ) ;                // pointer 指向当前结点的右兄弟
    }
    while ( ! aQueue. empty( ) ) {
        pointer = aQueue. front( ) ;                          // 获得队首元素
        aQueue. pop( ) ;                                      // 当前结点出队列
        Visit( pointer -> Value( ) ) ;                        // 访问当前结点
        pointer = pointer -> LeftMostChild( ) ;               // pointer 指向当前结点的最左子结点
```

```
            while (pointer != NULL) {                    // 当前结点的子结点进队列
                aQueue.push(pointer);
                pointer = pointer -> RightSibling();
            }
        }
    }
```

【算法 6.5 结束】

6.2 树的链式存储结构

在计算机中，树有多种存储方式，一类是链式结构，另一类是顺序结构。但是，无论在应用中采取何种方式，都要求树的存储结构不但能存储各结点本身的数据信息，还要求能准确反映树中各结点之间的逻辑关系。本节将介绍树的几种链式存储方式。

6.2.1 “子结点表”表示方法

“子结点表”(list of children)表示方法，就是指每个分支结点的子结点按照从左至右的顺序形成一个链表存储在该分支结点中，其主体是一个存储了树中各结点信息的数组。数组中的每个元素包括 3 个域，分别用来存放结点信息的值、其父结点指针以及指向其子结点表的指针。其形式如图 6.6 所示，这是图 6.1 树的“子结点表”实现。最左侧的一列标明结点在数组中的下标；

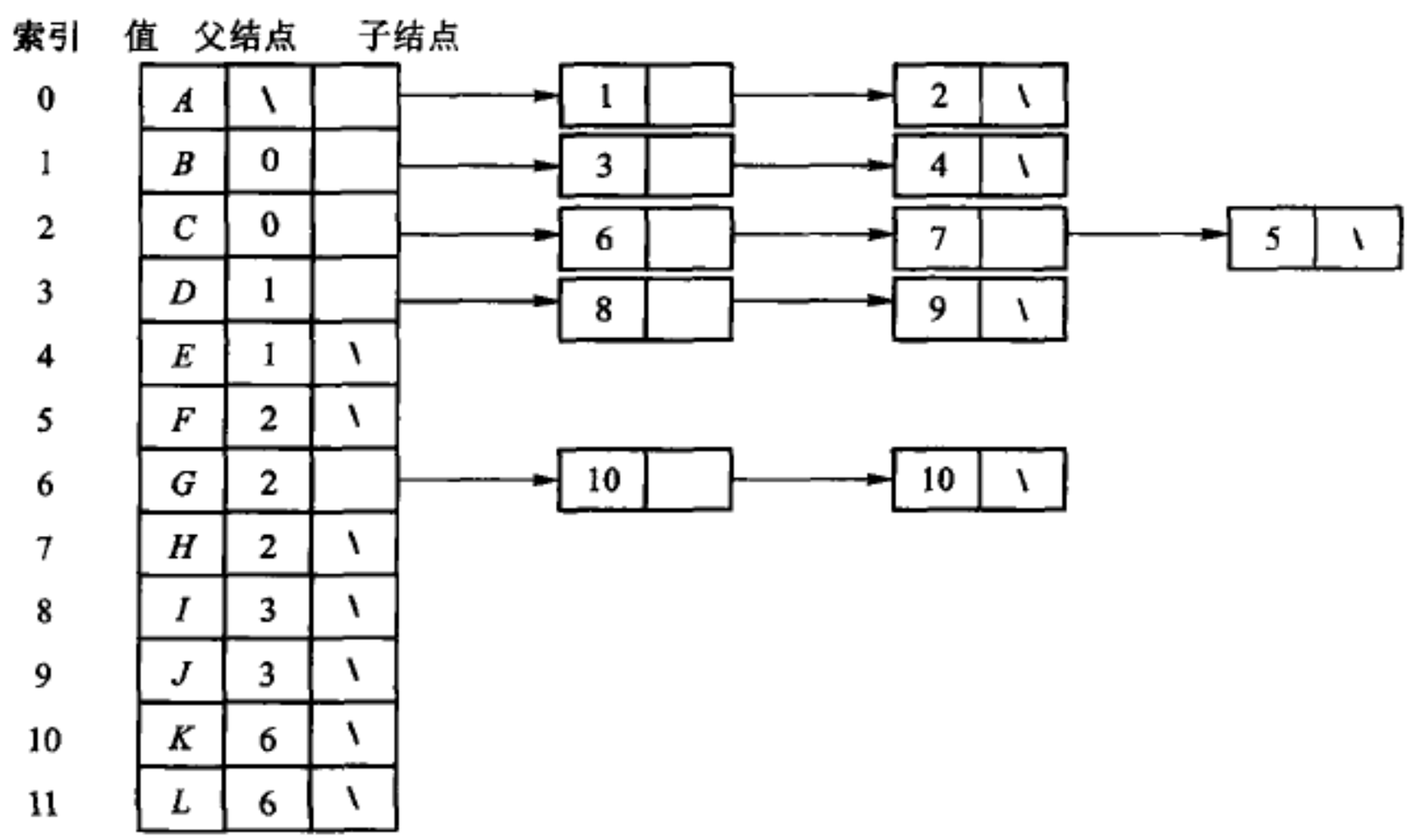


图 6.6 以“子结点表”表示法实现图 6.1 中的树

值列存储结点信息;父结点列存储该结点的父指针,为了简明,这些指针实际上使用数组的下标值;最后一列用于存储指向子结点链表的指针,链表中子结点的顺序由左至右,且每个表项均存储着指向下一个子结点的指针。

在"子结点表"表示法中,最左子结点可由子结点链表的第一个表项直接找到,顺链可以找到所有子结点。

6.2.2 静态"左子/右兄"表示法

下面介绍上述表示方法的一个改进——静态"左子/右兄"(也称"左子结点/右兄弟结点",left child/right sibling)表示法,使得访问结点的右侧兄弟结点更加方便。

图6.7所示为静态"左子/右兄"的一个实例,该方法仍然使用数组存储树中的各结点。每个结点元素包括4个域,分别用于存储结点的值、指向其父结点以及指向最左子结点和右侧兄弟结点的指针。这种表示法比"子结点表"表示法的空间效率更高,而且每个结点的存储空间大小固定。

如果两棵树存储在同一个数组中,那么把其中一棵树添加为另一棵树的子树的操作非常简单。把图6.7中以A为根结点的树变成H的最左子树,其合并过程如图6.8所示。结点数组中有3个指针被调整:H的左子结点指针指向A,A的父指针指向H,A的右侧兄弟结点指针指向J。

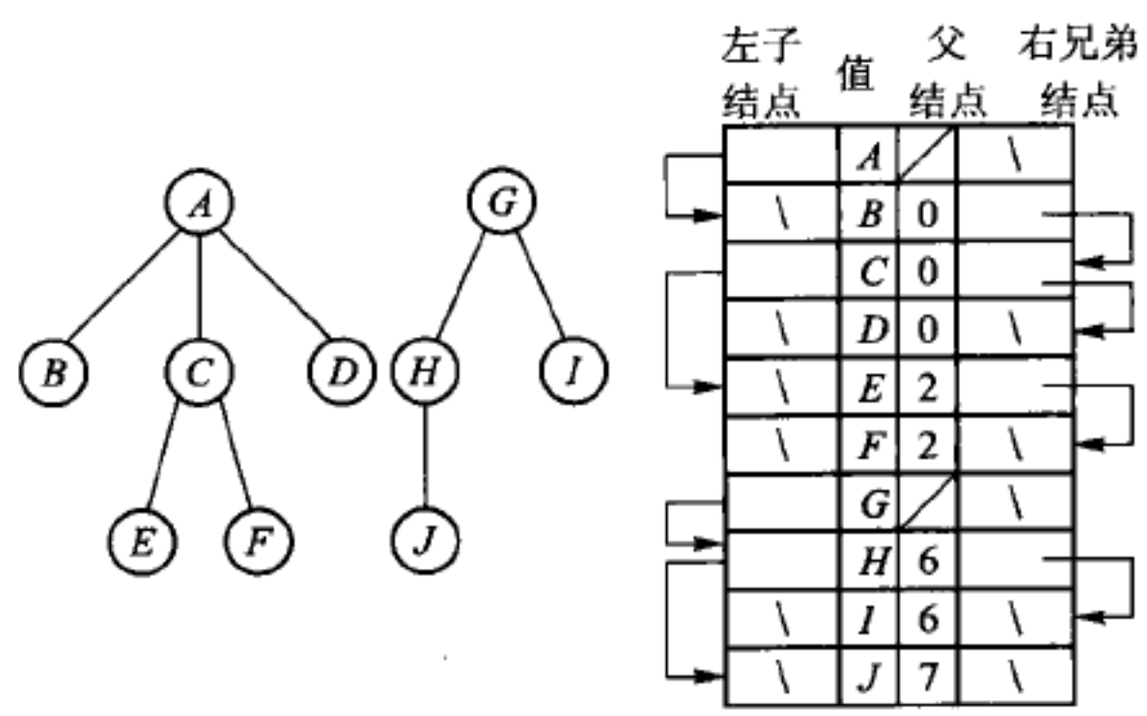


图6.7 "左子/右兄"表示两棵树

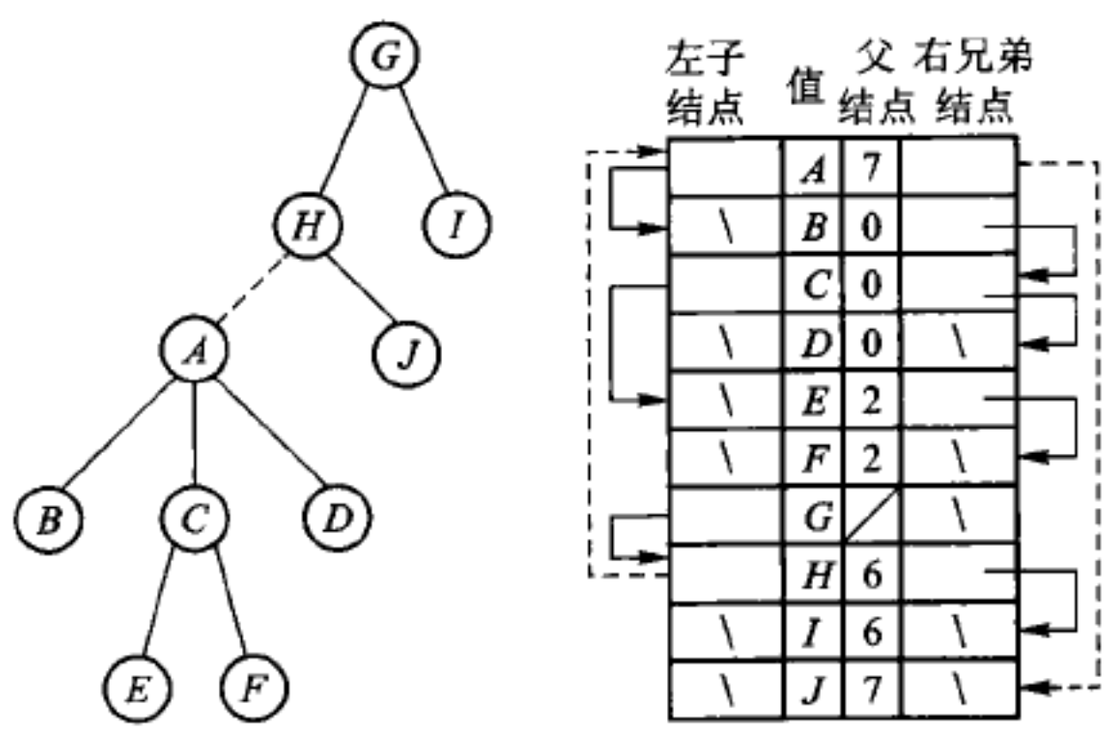
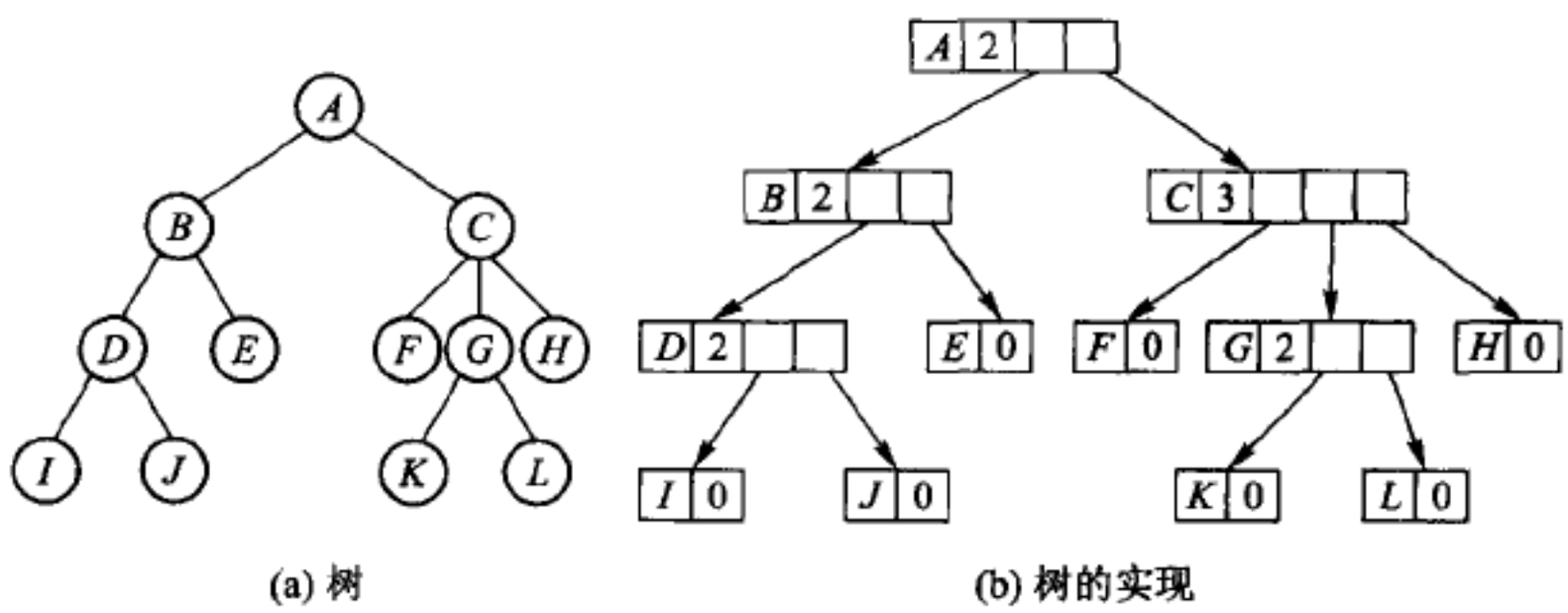


图6.8 使用静态"左子/右兄"实现对树的归并

6.2.3 动态表示法

由于树中的结点可以有任意的度数,并且各结点的子结点数随时可能发生变化。树结点的动态性给树的实现带来了困难。当然,可以采取一种不太灵活的方式,即限定树的度数,在实现树的时候只需要给树的每一个结点分配确定数目的指针域即可。很明显,这种存储方法对度数小的结点会浪费存储空间,而超过度的限制时增加结点会非常不方便。另外一种实现方法是为每个结点分配可变的存储空间,每一个结点都存储一个子结点指针表,其子结点的数目也存储在该结点中。使用这样的存储方式可以避免上述不足,即使子结点数目发生变化,也只需给该结点重新分配一个大小合适的存储空间即可。由此可见,这种途径更具有灵活性。如图 6.9 所示,每个结点存储一个子结点指针链表,还可以存储其子结点的数目。这种表示法本质上与"子结点表"表示法相同,但是它可以动态地分配结点空间,而不是把所有结点分配在同一个数组中。



(a) 树　　(b) 树的实现

图 6.9 树的动态表示法

6.2.4 动态"左子/右兄"二叉链表表示法

6.1.2 小节介绍了森林和树的等价转换,6.2.2 小节的静态"左子/右兄"可以看成是这种方法的静态链表实现。

可以直接采用 5.3.1 节介绍的方法实现动态的"左子/右兄"二叉链表表示法(dynamic"Left Child/Right Sibling",往往简称为"左子/右兄"表示法,或二叉链表表示法)。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,根的右链就是森林中每棵树的根结点。这种方法应用最广泛。

代码 6.6 和代码 6.7 给出相应的树结点和树的抽象数据类型中部分成员函数的具体实现。

【代码 6.6】 树结点的抽象数据类型中的关键实现细节。

```
// 用二叉链表实现树要在代码 6.1 的 TreeNode 类中增加以下私有数据成员
private:
```

```
        T m_Value;                                          // 树结点的值
        TreeNode<T> * pChild;                               // 左子结点指针
        TreeNode<T> * pSibling;                             // 右兄弟指针

// 部分成员函数的具体实现
template<class T>
bool TreeNode<T>::isLeaf() {                                // 判断当前结点是否为叶结点
        if (pChild == NULL)                                 // 如果结点是叶结点返回 true
                return true;
        return false;
}
template<class T>
void TreeNode<T>::setValue(const T& value) {                // 设置结点的值
        m_Value = value;
}
template<class T>
void TreeNode<T>::InsertFirst(TreeNode<T> * node) {         // 以第一个子结点的身份插入结点
        if (pChild == NULL)                                 // 如果没有子结点
                pChild = node;                              // 将 node 置为子结点
        else {                                              // 如果有子结点
                node -> setSibling(pChild);                 // 将原来子结点置为 node 的右兄弟
                pChild = node;                              // 将 node 置为子结点
        }
}
```

【代码 6.6 结束】

【代码 6.7】 树的抽象数据类型的部分实现。

```
// 在代码 6.2 的 Tree 类中增加私有数据成员
private:
        TreeNode<T> * root;                                 // 树根结点

// 部分成员函数的具体实现
template<class T>
TreeNode<T> * Tree<T>::Parent(TreeNode<T> * current) {
        using std::queue;                                   // 使用 STL 队列
        queue<TreeNode<T> *> aQueue;
        TreeNode<T> * pointer = root;
        TreeNode<T> * upperlevelpointer = NULL;             // 用于记录父结点
```

```
    if (current != NULL && pointer != current) {
        while (pointer != NULL) {                          // 森林中所有根结点进队列
            if (current == pointer)                        // 根的父结点指针为空,返回 NULL
                return NULL;
            aQueue.push(pointer);                          // 当前结点进队列
            pointer = pointer -> RightSibling();           // 当前指针指向右兄弟
        }
        while (! aQueue.empty()) {
            pointer = aQueue.front();                      // 取队列首结点指针
            aQueue.pop();                                  // 当前元素出队列
            upperlevelpointer = pointer;                   // 指向上一层的结点
            pointer = pointer -> LeftMostChild();          // pointer 指向当前结点的最左子结点
            while (pointer) {                              // 当前结点的子结点进队列
                if (current == pointer)
                    return upperlevelpointer;              // 返回父结点指针
                else {
                    aQueue.push(pointer);
                    pointer = pointer -> RightSibling();
                }
            }
        }
    }
    return NULL;
}

template < class T >
void Tree < T > ::DestroyNodes(TreeNode < T > * root) {    // 删除以 root 为根的子树的所有结点
    if (root != NULL) {
        DestroyNodes(root -> LeftMostChild());             // 递归删除第一子树
        DestroyNodes(root -> RightSibling());              // 递归删除其他子树
        delete root;                                       // 删除根结点
    }
}

template < class T >
void Tree < T > ::DeleteSubTree(TreeNode < T > * subroot) { // 删除以 subroot 为根的子树的所有
                                                            // 结点
    TreeNode < T > * pointer;
```

```
if ( subroot == NULL) return;                          // 若待删除的子树为空则返回
pointer = Parent (subroot);                            // 找 subroot 的父结点
if ( pointer == NULL)                                  // subroot 就是森林第一个树根的情况
        root = subroot -> RightSibling( );             // 将 root 设置为 subroot 的右兄弟
else if ( pointer -> LeftMostChild( ) == subroot)      // subroot 为最左子结点的情况
        pointer -> setChild( subroot -> RightSibling( ) );  // pointer 的左子结点置为 subroot 的右
                                                       // 兄弟
else {                                                 // subroot 有左兄弟的情况
        pointer = pointer -> LeftMostChild( );         // 下降到最左兄弟
        while ( pointer -> RightSibling( ) ! = subroot)  // 顺右链找到直接左兄弟
                pointer = pointer -> RightSibling( );
                pointer -> setSibling( subroot -> RightSibling( ) );   // subroot 的前后兄弟挂接,subroot 脱链
        }
        subroot -> setSibling( NULL);
        DestroyNodes( subroot);                        // 删除以 subroot 为根的子树的所有结点
}
```

【代码6.7结束】

6.2.5 父指针表示法和在并查集中的应用

1. 父指针表示法

在某些应用中,只需要知道父结点情况,因此每个结点只需要保存一个指向其父结点的指针域,这种实现被称为父指针(parent pointer)表示法。

为了简明,用数组存储树的所有结点,同时在每个结点中附设一个指针指示其父结点的位置。例如,图6.10就展示了图6.1所示的树的父指针表示法的存储结构。

结点索引	0	1	2	3	4	5	6	7	8	9	10	11
值	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*
父结点索引	\	0	0	1	1	2	2	2	3	3	6	6

图6.10 父指针表示法

由于树中每一个结点的父指针是唯一的,因此父指针表示法可以唯一地表示任何一棵树。在这种表示方法下,寻找一个结点的父结点只需要$O(1)$时间。在树中可以从一个结点出发找出一条向上延伸到达其祖先的路径,即从一个结点到其父亲结点,再到其祖父结点等。求祖先路径所需要的时间正比于路径上的结点个数,因此父指针表示法对于求树根结点的运算非常方便。

合并两棵树的操作也非常简单,只需要将一棵树的树根表示为另一棵树的子结点,即设置父指针值即可。

用父指针表示求结点的子结点和兄弟结点比较麻烦,需要查询整个结构。另外,这种存储结构中没有表示出结点之间的左右次序。父指针表示法适合于无序树的情况,而且只适合于查询

结点的根和合并树等操作，其主要优点是节省存储空间且操作便捷。

2. 等价类和并查集

父指针表示法的一个重要应用是实现并查集(union/find)。并查集是一种特殊的集合，由一些不相交的子集构成。并查集的基本操作是：

(1) find：判断两个结点是否在同一个集合中。

(2) union：归并两个集合。

像栈、队列一样，并查集也是一种重要的抽象数据类型，可以用于求解等价类问题。

在离散数学中已经学习了等价关系和等价类的概念：假定一个具有 n 个元素的集合 S，另有一个定义在集合 S 上的关系集合 R。x、y、z 表示集合中的元素。若关系 R 是等价关系，当且仅当如下条件为真时成立：

(a) 对于所有的 x，有 $(x,x) \in R$(即关系是自反的)。

(b) 当且仅当 $(x,y) \in R$ 时 $(y,x) \in R$(即关系是对称的)。

(c) 若 $(x,y) \in R$ 且 $(y,z) \in R$，则有 $(x,z) \in R$(即关系是传递的)。

如果 $(x,y) \in R$，则元素 x 和 y 是等价的。等价类是指相互等价的元素所组成的最大集合。所谓最大，就是指不存在类以外的元素，与类内部的元素等价。由 $x \in S$ 生成的一个 R 等价类，用数学语言表示为 $[x]_R = \{y \mid y \in S \wedge xRy\}$。$R$ 可将 S 划分成为 r 个不相交的划分 $S_1, S_2, \cdots, S_r$，这些集合的并为 S。

假定集合 S 有 n 个元素，k 个形如 (x,y) 的等价偶对就确定了一个等价关系 R。下面给出划分等价类的过程：

初始情况 S 中的每个元素都属于一个独立的等价类 $S_1, S_2, \cdots, S_n$。依次读入 k 个偶对，对每个读入的偶对 (x,y)，判断 x 和 y 是否属于相同的子集：假设 $x \in S_i, y \in S_j$，如果 $S_i \neq S_j$，则将集合 S_i 置入集合 S_j 并将集合 S_i 置为空(或者将 S_j 置入 S_i 并将 S_j 置为空)。最后，$S_1, S_2, \cdots, S_n$ 中所有的非空子集就是由等价关系 R 确定的等价类。

从上述操作可见，划分等价类需要对集合进行 3 种操作：

(1) 给每一个元素构造一个独立的集合。

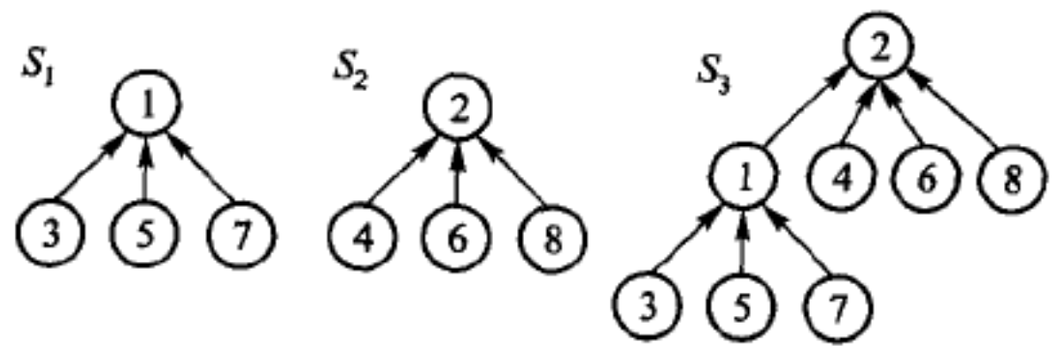
(2) 判定某个元素所在的子集，目的是为了确定两个元素是否在同一个集合之中。即搜索包含该元素的等价类，对于同一个集合中的元素返回相同的结果，否则返回不同的结果。

(3) 归并两个集合为一个集合。

用父指针表示的树形结构实现的并查集可以很容易地解决等价类问题。约定森林 $F = \{T_1, T_2, \cdots, T_r\}$ 表示集合 S，森林中的每一棵树 T_i 表示集合 S 的一个子集，树中的结点表示集合 S 中的一个元素。树中的每一个非根结点都指向其父结点，用根结点作为集合的标识符。

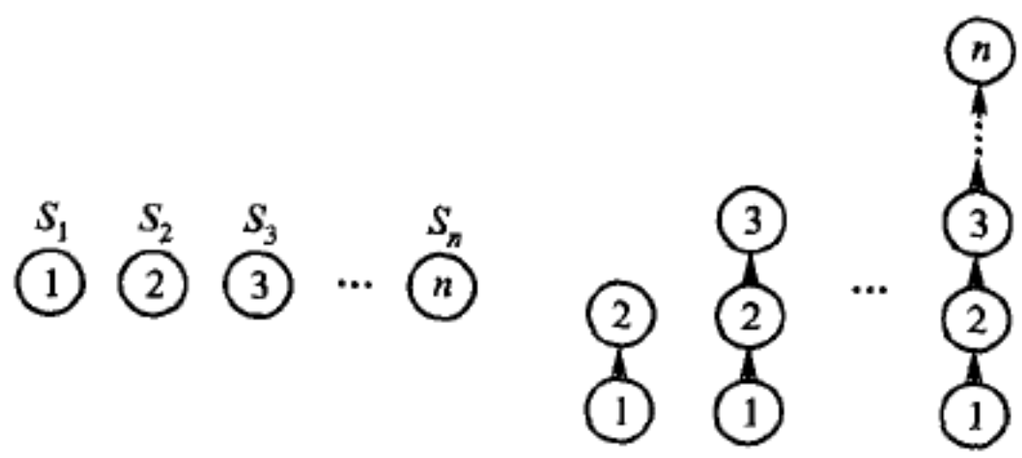
由于用根结点表示子集的类别，因此“查找”某一个元素所属的集合时，只要从该结点出发，沿父链域找到树的根结点即可。实现集合的“并”操作只要将一棵子树的根指向另一棵子树的根即可。例如，图 6.11(a)和(b)中的两棵树分别表示子集 $S_1 = \{1,3,5,7\}$ 和 $S_2 = \{2,4,6,8\}$。图 6.11 中的(c)就实现了 $S_3 = S_1 \cup S_2$。

每次合并前都需要进行两次查找，查找所需要的时间由树的高度决定，合并所需的时间为 $O(1)$。容易看出，在最坏情况下，合并可能使 n 个结点的树退化成一条链。下面就是一个例子，将6.12(a)表示的子集合并时，每次都把集合元素多的根结点指向含元素少的根结点，经过 $n-1$ 次合并操作后得到的树的高度就为 n，如图6.12(b)所示。



(a) 子集 S_1 (b) 子集 S_2 (c) 并集 S_3

图6.11 集合的表示方法



(a) n 个集合 (b) 合并操作

图6.12 合并操作的一个极端情况

为了防止树退化为单链，应该让每个结点到其相应根结点的距离尽可能小，可以做如下两种改进。

第一种改进的方法是合并操作根据子集中包含元素的数目，令含元素少的子集的树根指向含元素多的子集的根，称做"重量权衡合并规则"(weighted union rule)。把小树合并到大树中，可以把树的整体深度限制在 $O(\log n)$，每次 find 操作只需要 $O(\log n)$ 时间。

代码6.8是树的父指针表示与并查(union/find)算法实现(只列出部分成员函数的实现)。类 ParTreeNode 和类 ParTree 分别描述了树结点和树的定义。成员函数 Different()检查两个结点是否在不同的集合中，函数 Union()用于归并两个集合，函数 Find()用于找到目标结点的根结点。如果一个等价对中的两个元素在同一个等价类中，则不需要做变动；否则两个等价类可以用 Union()函数归并。

【代码6.8】 树的父指针表示与 union/find 算法实现。

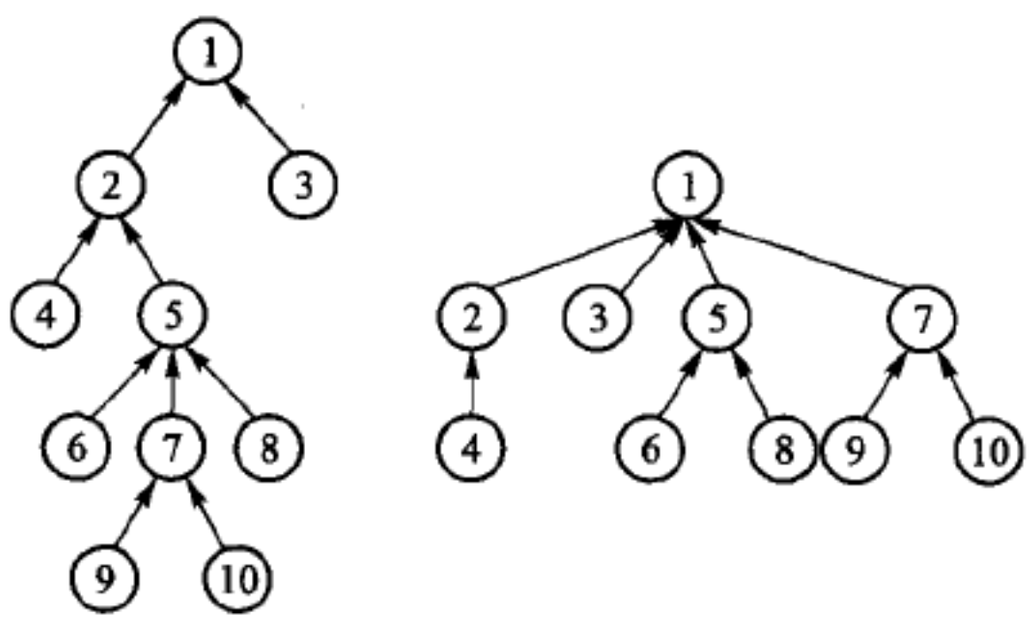
```
template < class T >
class ParTreeNode {                                    // 树结点类定义
private:
    T                    value;                        // 结点的值
```

```
    ParTreeNode <T>    * parent;                    // 父结点指针
    int                nCount;                      // 子树元素数目
public:
    ParTreeNode();                                  // 构造函数
    virtual ~ParTreeNode(){};                       // 析构函数
    T       getValue();                             // 返回结点的值
    void    setValue(const T& val);                 // 设置结点的值
    ParTreeNode <T> * getParent();                  // 返回父结点指针
    void    setParent(ParTreeNode <T> * par);       // 设置父结点指针
    int     getCount();                             // 返回结点数目
    void    setCount(const int count);              // 设置结点数目
};
template <class T>
class ParTree {                                     // 树类定义
public:
    ParTreeNode <T> * array;                        // 存储树结点的数组
    int    Size;                                    // 数组大小
    ParTreeNode <T> * Find(ParTreeNode <T> * node)const;    // 查找 node 结点的根结点
    ParTree(const int size);                        // 构造函数
    virtual ~ParTree();                             // 析构函数
    void Union(int i,int j);                        // 把下标为 i、j 的结点合并成一棵子树
    bool Different(int i,int j);                    // 判断结点 i 和 j 是否有相同的根结点
};
template <class T>
ParTreeNode <T> * ParTree <T> ::Find(ParTreeNode <T> * node) const {   // 返回 node 结点的根结点
    ParTreeNode <T> * pointer = node;
    while (pointer -> getParent() != NULL)
        pointer = pointer -> getParent();
    return pointer;
}
template <class T>
bool ParTree <T> ::Different(int i,int j) {         // 判断结点 i 和 j 是否有相同的根结点
    ParTreeNode <T> * pointeri = Find(&array[i]);   // 找到结点 i 的根
    ParTreeNode <T> * pointerj = Find(&array[j]);   // 找到结点 j 的根
    return pointeri != pointerj;                    // 若结点 i 和 j 的根结点不相同返回 true
}
template <class T>
void ParTree <T> ::Union(int i,int j)   {
```

```
ParTreeNode <T> * pointeri = Find(&array[i]);   // 找到结点 i 的根
ParTreeNode <T> * pointerj = Find(&array[j]);   // 找到结点 j 的根
if (pointeri != pointerj) {
    if (pointeri -> getCount() >= pointerj -> getCount()) {
        pointerj -> setParent(pointeri);        // 把结点 i 设置为 j 的父结点
        pointeri -> setCount(pointeri -> getCount() + pointerj -> getCount()); // 设置结点数目
    }
    else {
        pointeri -> setParent(pointerj);        // 把结点 j 设置为 i 的父结点
        pointerj -> setCount(pointeri -> getCount() + pointerj -> getCount()); // 设置结点数目
    }
}
```

【代码 6.8 结束】

加速并查集运算的另一个方法是采用路径压缩(path compression)技术。在执行 find 操作时,把结点到树根的路径上所有结点的父指针都指向根结点。例如,图 6.13(a)所示为依次合并集合{7,9,10}、{5,6,8}、{2,4}和{1,3}后所形成的树。图 6.13(b)所示为查找结点 7 的根时,对结点 7 到根所涉及的结点进行压缩路径之后的树的形态。路径压缩可以缩短结点与根之间的路径。



(a) 路径压缩之前　　(b) 路径压缩之后

图 6.13　路径压缩示例

下面提供了一种实现路径压缩的递归算法。函数 FindPC 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。算法 6.9 给出了带路径压缩的 FindPC 算法实现。

【算法 6.9】 带路径压缩的 find 算法。

```
template <class T>
ParTreeNode <T> * ParTree <T> ::FindPC(ParTreeNode <T> * node) const {
    if (node -> getParent() == NULL)
        return node;
```

```
        node -> setParent( FindPC( node -> getParent( ) ) ) ;
        return node -> getParent( ) ;
    }
```

【算法 6.9 结束】

路径压缩大大加速了 find 运算。在执行 union 时总是将小树并到大树上，而且在执行 find 时实行路径压缩，此时可以证明 n 次 find 至多需要 $O(n\alpha(n))$ 时间。其中，$\alpha(n)$ 是单变量 Ackermann( ) 函数的逆，它是一个增长速度比 $\log n$ 慢得多但又不是常数的函数。在实际应用中，$\alpha(n)$ 往往小于 4。因此，执行一串交错的合并和查找操作所需要的时间几乎与合并和查找的次数成线性关系。上述路径压缩算法是目前最有效的并查集实现方法。

6.3 树的顺序存储结构

顺序存储表示法存储树要求把树中的结点按照一定的顺序存储到一片连续的存储单元中。为了能够显示出树的逻辑结构，需要在结点表中包含足够的结构信息。

下面介绍与树和森林的遍历次序相关的几种典型的顺序存储结构。以图 6.5(a) 中所示的森林为例进行介绍。

6.3.1 带右链的先根次序表示

图 6.5(a) 所示的森林的先根次序序列是 *ABCEFDGHJI*。

在先根序列中，任何结点的子树的所有结点都直接跟在该结点之后，任何一个分支结点后面紧跟的是其第一个子结点。

在“带右链的先根次序表示”中，每个结点包括结点本身数据以及附加的两个表示结构信息的 ltag 和 rlink 域。结点的形式为：

ltag	info	rlink

其中，info 是结点的数据；rlink 是右指针，指向结点的下一个兄弟，即树或森林所对应的二叉树中结点的右子结点；ltag 是一个标记位，当结点是叶结点，即对应的二叉树中没有左子结点时，ltag 为 1，否则为 0。图 6.5(a) 森林的右链先根次序如图 6.14 所示。

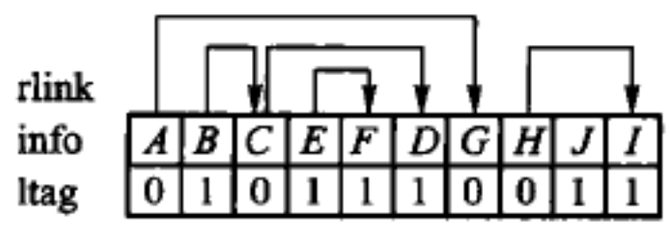


图 6.14 带右链的先根次序表示法

为了节省存储单元，“带右链的先根次序表示”用 ltag 代替了 llink。从结点的次序和 ltag 的值完全可以推知 llink。ltag 为 0 的结点有左子结点，其 llink 指向存储区中该结点的下一个结点；ltag 为 1 的结点没有左子结点，

其 llink 为空。

6.3.2 带双标记的先根次序表示

事实上,“带右链的先根次序表示”中 rlink 也有冗余,可以把 rlink 指针替换为一个标志位 rtag,成为“带双标记的先根次序表示”。其中,每个结点包括结点本身数据以及两个标志位 ltag 和 rtag,其结点的形式为:

ltag	info	rtag

图 6.5(a)森林的双标记位先根次序可表示为如图 6.15 所示的形式。

由结点的先根次序以及 ltag、rtag 两个标志位,就可以确定树“左子/右兄”链表中结点的 llink 和 rlink 值。其中,llink 的确定与带右链的先根次序表示法相同。

由先根次序的性质可知,任何结点的子树都直接排在该结点的后面,接着再排该结点的下一个兄弟结点。任何一棵子树,在先根序列中排在最后的一定是没有子结点的叶结点,该叶结点的 ltag 为 1。确定 rlink 的过程就需要利用这个性质。当一个结点 x 的 rtag 为 1 时,它的 rlink 显然应为空;当一个结点 x 的 rtag 为 0 时,它的 rlink 应指向结点序列中排在以 x 为根的子树后紧接着的那个结点 y。由于树的先根次序中子树结点是嵌套出现的,在顺序搜索的过程中需要嵌套处理 x 的所有子树,因此确定 y 的过程要用到栈结构。扫描到一个 rtag 为 0 的结点就将其进栈;扫描到一个 ltag 为 1 的结点就从栈顶弹出一个结点并为其设置 rlink。

rtag	0	0	0	0	1	1	1	0	1	1
info	*A*	*B*	*C*	*E*	*F*	*D*	*G*	*H*	*J*	*I*
ltag	0	1	0	1	1	1	0	0	1	1

图 6.15 带双标记的先根次序表示法

算法 6.10 给出从带双标记位的先根次序表示导出各结点的 llink 和 rlink 的算法。

【算法 6.10】带双标记位先根次序树构造算法。

```
template < class T >
class DualTagTreeNode  {                                  // 双标记位先根次序树结点类
public:
      T      info;                                        // 结点数据信息
      int    ltag;                                        // 左标记
      int    rtag;                                        // 右标记
      DualTagTreeNode( );                                 // 构造函数
      virtual  ~DualTagTreeNode( );                       // 析构函数
};
template  < class T >
Tree < T > :: Tree( DualTagTreeNode < T >  * nodeArray, int count)  {
      // 利用带双标记位的先根次序表示构造左子结点右兄弟表示的树
```

```
using std::stack;                                   // 使用 STL 中的栈
stack < TreeNode < T > *  > aStack;
TreeNode < T >  * pointer = new TreeNode < T > ;    // 准备建立根结点
root = pointer;
for ( int i = 0; i < count - 1; i ++ ) {            // 处理一个结点
    pointer -> setValue( nodeArray[ i ]. info) ;    // 结点赋值
    if ( nodeArray[ i ]. rtag == 0)                 // 若右标记为 0 则将结点压栈
        aStack. push( pointer) ;
    else pointer -> setSibling( NULL) ;             // 若右标记为 1 则将兄弟指针设为空
    TreeNode < T >  * temppointer = new TreeNode < T > ;
    if ( nodeArray[ i ]. ltag == 0)                 // 如果左标记为 0 则设置一个子结点结点
        pointer -> setChild( temppointer) ;
    else {                                          // 若左标记为 1
        pointer -> setChild( NULL) ;                // 子结点指针设为空
        pointer = aStack. top( ) ;                  // 取栈顶元素
        aStack. pop( ) ;
        pointer -> setSibling( temppointer) ;       // 为其设置一个兄弟结点
    }
    pointer = temppointer;
}
// 处理最后一个结点
pointer -> setValue( nodeArray[ count - 1 ]. info) ;
pointer -> setChild( NULL) ;
pointer -> setSibling( NULL) ;
}
```

【算法 6.10 结束】

6.3.3 带度数的后根次序表示

图 6.5(a)所示的森林的后根次序序列是 *BEFCDAJHIG*。

在此后根次序序列中,任何一棵子树的所有结点都聚集在一起,并且以该子树的根作为最后一个结点。

在"带度数的后根次序表示"中,每个结点有两个域,info 用于存放结点的数据,degree 用于存储结点的度数。其结点的形式为:

info	degree

图6.5(a)森林的带度数后根次序可表示为如图6.16所示的形式。

这种表示法不包括指针，但它仍能反映树的结构。由于在后序序列中每一棵子树的结点都存储在一起，且每一棵子树在后序序列中的最后一个位置存储这棵子树的根结点，因此将带度数的后根次序表示转化成森林的逻辑结构时，只需要从左至右进行扫描。度数为零的结点是叶子结点(也可以看做是一棵子树的根)，当遇到度数非零(设为 k)的结点时，则排在该结点之前且离它最近的 k 个子树的根就是该结点的 k 个子结点。例如，上述森林中的结点 C 的度数为2，那么排在 C 之前的两个结点 E 和 F 就是结点 C 子树的根结点，从而结点 C、E 以及结点 F 就构成了一棵子树，结点 C 是这棵子树的根；同理，A 的度数为3，则排在 A 之前且离 A 最近的三棵子树的根结点就是 A 的子结点，即 A 的子结点是 B、C 和 D，从而又形成了一棵树。如此进行下去，就可以得到如图6.5(a)中所示的森林。带度数的后根次序表示实现树时也需要用到栈。

info	*B*	*E*	*F*	*C*	*D*	*A*	*J*	*H*	*I*	*G*
degree	0	0	0	2	0	3	0	1	0	2

图6.16 带度数的后根次序表示法

6.3.4 带双标记的层次次序表示

图6.5(a)所示的森林的层次次序序列为 *AGBCDHIEFJ*。

这种层次次序的序列也同样表现了森林的一些结构信息：森林中处于同一层的结点在序列中都聚集在一起，任何一个结点只要它有右兄弟，那么在序列中它的右兄弟就排在该结点后面。

类似于带双标记的先根次序表示，引入带双标记的层次次序表示法，其结点的形式为：

ltag	info	rtag

其中，info 是结点的数据；当结点没有左子结点时标记位 ltag 为1，否则为0；当结点没有下一个兄弟时标记位 rtag 为1，否则为0。

图6.17所示为图6.5(a)的带双标记位的层次次序表示。

ltag	0	0	1	0	1	0	1	1	1	1
info	*A*	*G*	*B*	*C*	*D*	*H*	*I*	*E*	*F*	*J*
rtag	0	1	0	0	1	0	1	0	1	1

图6.17 带双标记的层次次序表示法

由层次次序的性质可知，任何结点的子结点都排在该结点的所有兄弟结点后面。并且任何结点的最后一个兄弟结点都不会再有下一个兄弟，即最后一个兄弟的 rtag 为1。由结点的层次次序以及 ltag、rtag 两个标志位，就可以确定树的左子/右兄链表中结点的 llink 和 rlink 值。rlink 的值很容易确定：结点的 rtag 为1，则 rlink 为空；结点的 rtag 为0，则 rlink 指向该结点紧邻的下一个结点。

下面用 ltag 和 rtag 的值来确定 llink。当一个结点 x 的 ltag 为0时，其 llink 不空，应该指向结点序列中排在结点 x 的兄弟结点后面的那个结点 y。由于跟层次序列相关，因此需要使用队列确定这个结点的子结点。顺序扫描树的层次序列，若结点的 ltag 值为1时，则置其 llink 为空；否则把该结点 x 放入队列。如果遇到 rtag 为1的结点，这时该结点的兄弟结点链已经扫描完毕，下

一个将要扫描的结点 y 就是结点 x 的最左子结点，此时取出队列的第一个元素（即先前保存的 x），将其 llink 指向 y 即可。

只要把算法 6.10 中的栈修改为队列，就可以同样地从带双标记位的先根次序表示导出各结点的 llink 和 rlink，顺序扫描树的层次序列。rlink 很好确定，如果结点的 rtag 值为 0，那么其后的结点就是右兄弟，否则 rlink 为空。若结点的 ltag 值为 1，则置其 llink 为空；当一个结点 x 的 ltag 为 0 时，其 llink 不空，应该指向结点序列中排在结点 x 的兄弟结点链之后的那个结点 y，因此 x 结点应该进入队列。遇到 rtag 为 1 的结点后，这时 x 的兄弟结点链已经扫描完毕，下一个将要扫描的结点 y 就是结点 x 的最左子结点，此时取出队列的第一个元素（即先前保存的 x），将其 llink 指向 y 即可。

由于避免了存储指针，因此树的顺序表示方法可以把树压缩存储在磁盘上，以便日后能够方便地把树结构恢复为内存的链式存储形式，并进行各种操作。树形结构还有其他的顺序表示形式，而且很多顺序存储方法既适应于树/森林，也适应于二叉树。

6.4 K 叉 树

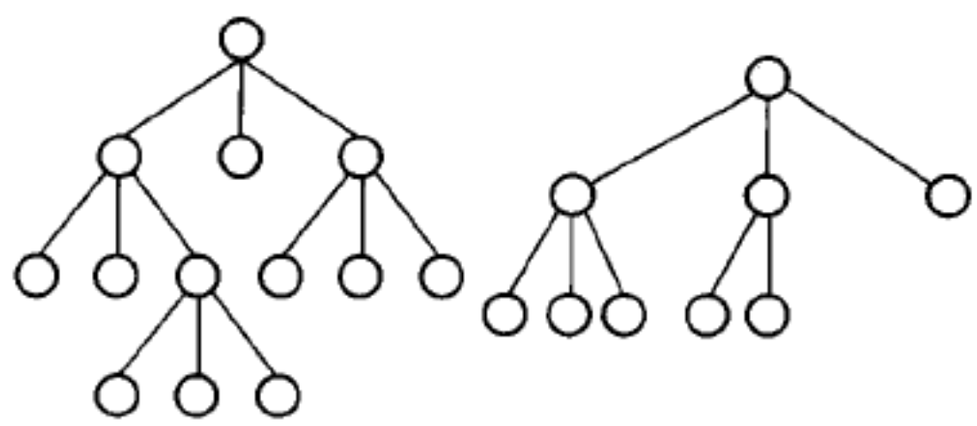
前面讨论的树是各结点有任意度数的树。在一般的情况下，允许一棵树的结点有不同的度。本节简单介绍一种特殊的树——K 叉树。K 叉树 T 是具有下列性质的有限结点集：

（1）集合可以为空。

（2）非空集合是由一个根结点 root 及 K 棵互不相交的 K 叉树构成。也就是说，其余结点被划分成 $T_0, T_1, \cdots, T_{K-1}$（$K \geq 1$）个子集，每个子集都是 K 叉树，使得 $T = \{R, T_0, T_1, \cdots, T_{K-1}\}$。

二叉树的许多性质可以推广到 K 叉树。在考虑 K 叉树中叶子结点数目与分支结点数目之间的关系时，同样可推出类似的定理。K 叉树的各分支结点都有 K 个子结点，子结点的数目是确定的，因此实现 K 叉树比较容易，可以使用前面介绍的链式或顺序存储方式。

满 K 叉树和完全 K 叉树与满二叉树和完全二叉树类似。图 6.18 中给出了 $K=3$ 时满 K 叉树和完全 K 叉树的表示。实际上大多数 K 叉树的应用采用的是完全 K 叉树或者满 K 叉树。对于完全 K 叉树，可以类似于完全二叉树，按编号顺序存储在一个数组中。



(a) 满 3 叉树　　(b) 完全 3 叉树

图 6.18　满 3 叉树与完全 3 叉树

K 叉树的应用也十分广泛。在计算机图形学中,4 叉树和 8 叉树有非常重要的应用。

本章小结

本章介绍了树和森林的概念、术语、表示方法及重要的算法。

对于树的概念,本书给出了两种等价的定义方式:

(1) 从递归来看,树可以看做是由根结点和子树构成,而每一棵子树也是树。

(2) 从二元组的逻辑结构来,树是结点的有穷集合以及在该集合上定义的一个二元关系。

在树的基本概念中,还给出了树及森林的相关概念,例如树叶、根结点、边、层数、度数、有序树等。这些概念基本上沿用二叉树一章中的相关术语。

树与二叉树、森林与二叉树之间存在一一对应的关系。本书介绍了树与二叉树和森林与二叉树相互转化的方法。

树和森林的周游是十分重要的运算。

按深度优先周游树和森林包括先根次序和后根次序周游。按先根次序周游森林等同于按前序法周游树或森林对应的二叉树,按后根次序周游森林等同于按中序法周游对应的二叉树。

广度优先周游树,是指从树的第零层(根结点)开始,自上至下逐层周游;在同一层中,按照从左到右的顺序对结点逐一进行访问,直到访问完最下一层的所有结点。

本章重点介绍了树的存储结构。联系到具体的应用,可以选择不同的存储方式。

常用的链式存储结构有"子结点表"、静态"左子/右兄"、动态表示法、"左子/右兄"二叉链表表示法和父指针表示法等。"左子/右兄"二叉链表是采用得最多的表示法。父指针表示法可以方便地实现并查集,常用于维护由一些不相交子集构成的集合,判断两个结点是否在同一个集合中以及归并两个集合。

顺序存储表示法把树中结点按照一定的顺序存储到一片连续的存储单元中。"带双标记的先根次序表示"、"带度数的后根次序表示"和"带双标记的层次次序表示"都是与树/森林遍历次序相关的顺序存储结构,根据存储方法的性质加入了必要的附加信息。

在本章的最后还介绍了 K 叉树的概念及性质。K 叉树是由一个根结点及 K 棵互不相交的 K 叉树构成。二叉树的许多性质可以推广到 K 叉树,通常用数组实现完全 K 叉树。

习 题

1. 给出一棵树的逻辑结构 $T=(K,R)$,其中

$K=\{A, B, C, D, E, F, G, H, I, J, K, L, M\}$

$R=\{<A, B>, <A, C>, <A, D>, <B, E>, <B, F>, <B, G>, <C, H>, <D, I>, <D, J>,$

$<F, K>$, $<F, L>$, $<I, J>$}

试用树形表示法画出此树，并按照递归定义将树划分为子树，指出哪些结点是根，哪些结点是树叶，确定各结点的层数、度数，并将该树转化为对应的二叉树。

2. 3 个结点可以构造出多少个不同的有序树？试将其画出来。

3. 将图 6.19 中的二叉树转换为对应的森林，分别按照先根次序、后根次序列出其结点。

4. 编写出一个算法来判断两棵树是否相同。尽可能提高算法效率，并分析算法的运行时间代价。

5. 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。尽可能提高算法效率，分析算法的运行时间代价。

6. 编写一个函数，以一棵树为输入，返回树的结点数目。

7. 分析"子结点表"表示法、"左子/右兄"二叉链表表示法的结构性开销比率。相比之下它们各自的空间效率如何？

8. 使用重量权衡合并规则与路径压缩，对下列从 0 ~ 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)

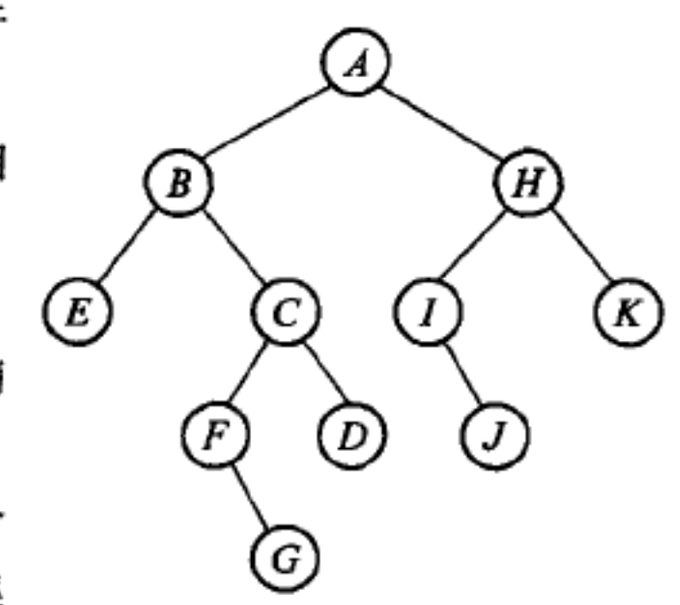


图 6.19 习题 3 的图示

9. 编写函数，以一棵树的根为输入，返回一棵对应的二叉树。

10. 一个高度为 L 的 K 叉树，如果按层次顺序对树中所有结点从 1 开始编号，问：

(1) 编号为 N 的结点的父结点(若存在)编号是多少？

(2) 编号为 N 的结点的第 i 个子结点(若存在)编号是多少？

(3) 编号为 N 的结点有右兄弟的条件是什么，其编号是多少？

11. 画出图 6.20 所示二叉树对应森林的左子/右兄法存储表示、带右链的先根次序存储表示、带左链的层次次序存储表示和带度数的后根次序存储表示。

12. 设计一个算法，将一个用带度数的后根次序法表示的森林转换为左子/右兄法表示。

13. 设计一个算法，将一个用带右链的先根次序法表示的森林转换为用带度数的后根次序法表示。

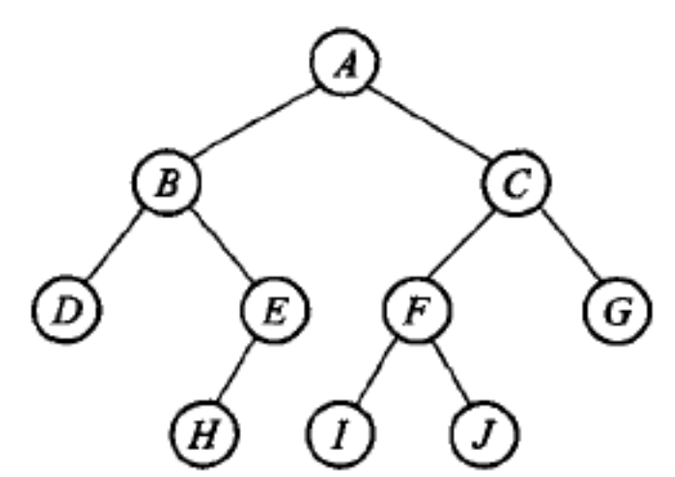


图 6.20 习题 10 的图示

上 机 题

1. 使用带子结点指针数组的树的链接表示法，实现 6.1.3 节中定义的树类。此实现必须能支持结点的子结点数目的变化。在生成时，结点应该只被分配只够存储其初始子结点集的空间。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。

2. 使用父指针表示法，实现 6.1.3 节中定义的树类。

3. 若用大写字母标识树的结点，则可用带标号的广义表形式表示一棵树，其语法图如图 6.21 所示。

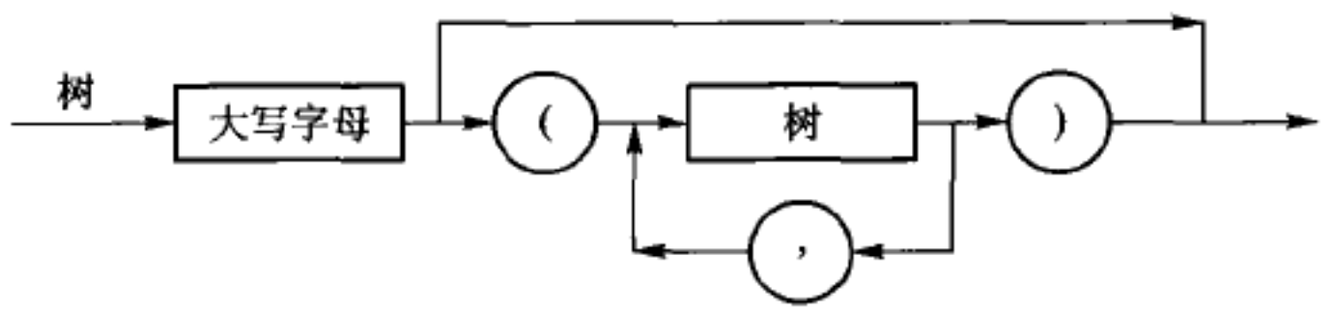


图 6.21 上机题 3 的图示

编写递归算法：

(1) 将这种表示方法的字符序列转化成树的左子/右兄表示法。

(2) 将用左子/右兄表示法表示的树转化为这种表示法的字符序列。

4. 另一种表示二维数组的方法是 4 叉树表示法，其中一个结点的 4 个儿子分别表示数组的 NE、SE、SW、NW，4 个分块如图 6.22 所示。这种表示二维数组的 4 叉树称为金字塔。

(1) 设计一个基于金字塔表示法的矩阵乘法算法。

(2) 对于给定的 $2^p \times 2^p$ 矩阵，设计一个构造表示这个矩阵的二叉树算法。并将这个算法推广到可表示任意 $M \times N$ 矩阵的情形，算法效率如何？

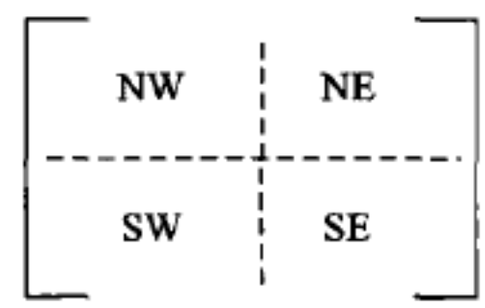


图 6.22 上机题 4 的图示

第7章　图

图(graph)是一种比线性结构和树形结构更为复杂的数据结构。在这种非线性结构中,结点之间的关系可以是任意的。

一般来讲,数据的逻辑结构可表示为结点的有穷集合 K 和 K 上的一个关系 r,如果对 K 中结点相对于关系 r 的前驱、后继个数加以限制,则可以分别定义线性结构、树形结构和图结构,即

- 线性结构:唯一前驱、唯一后继,反映一种线性关系。
- 树形结构:唯一前驱、多个后继,反映一种层次关系。
- 图结构:不限制前驱的个数,也不限制后继的个数,反映一种网状关系。

此时不难发现,线性表和树可看做是受限图。

图被广泛地应用于模拟真实世界或者抽象问题。其应用已经渗透到数学、物理学、化学、计算机科学、逻辑学乃至社会科学等诸多领域中。

7.1　图的定义和基本术语

考虑这样一个问题:要在几个城市之间建立起通信网络,使得每两个城市都有直接或者间接的通信线路。可以用一种自然、直观的描述方法来描述这个问题:用一个顶点代表一个城市;用顶点之间的连线代表相应两个城市之间的通信线路;在连线旁边附加一个数值表示该通信线路的造价。图7.1所示为5个城市之间的一个通信网络。由于资金有限,希望建立一个花费最小的通信网络。这样的问题就转化为在图7.1中找到一个连通所有顶点且边的权值之和最小的图。这样,一个实际问题就被表达成一种明晰的结构。

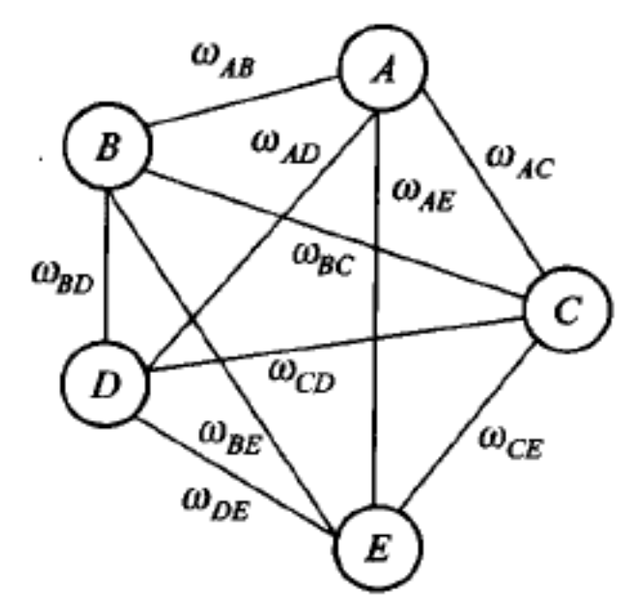


图7.1　用图描述通信网络

下面详细介绍图的定义和基本术语。

简单地说,图由表示数据元素的集合 V 和表示数据之间关系的集合 E 组成,记为 $G = <V, E>$。在图中,数据元素通常称做顶点(vertex),V 就是顶点的有穷非空集合;顶点的序偶,称之为边(edge),E 是边的集合。

若代表一条边的顶点序偶是无序的(即该边无方向),则称此图为无向图。无序的偶对用圆括号表示。设 $v_1, v_2 \in V$,若 $(v_1, v_2) \in E$,则在无向图 G 中 v_1 和 v_2 间有一条边,且 (v_1, v_2) 和 (v_2, v_1) 是同一条边。

若代表一条边的顶点序偶是有序的(即边有方向),则称此图为有向图。有序偶对用尖括号表示。若 $<v_1,v_2> \in E$,则 $<v_1,v_2>$ 表示有向图 G 中从 v_1 到 v_2 的一条边。有向图的边也可以称为弧(Arc),v_1 称为弧尾(tail)或边的始点,v_2 称为弧头(head)或边的终点。注意,$<v_1,v_2>$ 与 $<v_2,v_1>$ 是不同的两条弧。

例如,图 7.2(a)所示为无向图,用集合表示如下:

$G_1 = <V(G_1),E(G_1)>$

$V(G_1) = \{v_0,v_1,v_2,v_3,v_4\}$

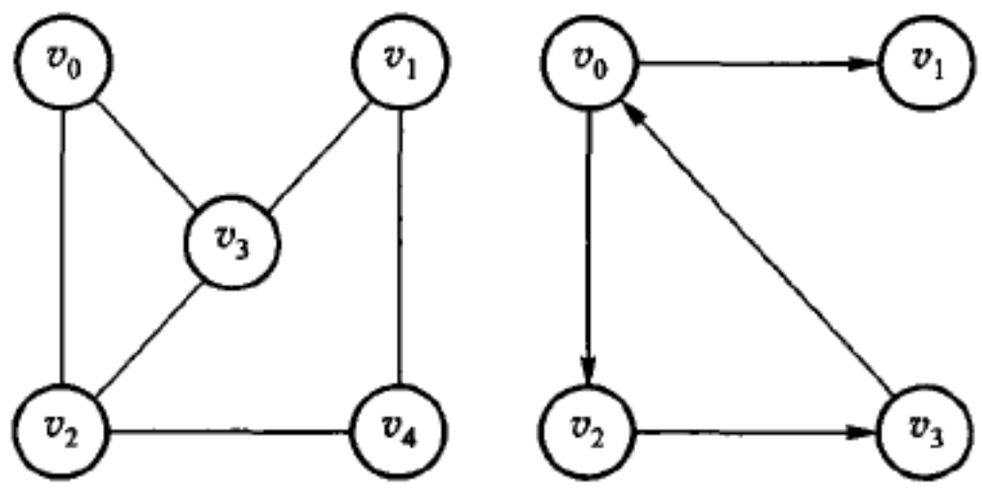
$E(G_1) = \{(v_0,v_2),(v_0,v_3),(v_1,v_3),(v_1,v_4),(v_2,v_3),(v_2,v_4)\}$

图 7.2(b)所示为有向图,用集合表示如下:

$G_2 = <V(G_2),E(G_2)>$

$V(G_2) = \{v_0,v_1,v_2,v_3\}$

$E(G_2) = \{<v_0,v_1>,<v_0,v_2>,<v_2,v_3>,<v_3,v_0>\}$



(a) 无向图G_1 (b) 有向图G_2

图 7.2 图的示例

有时图的边或弧附带权(weight),可以用于表示从一个顶点到另一个顶点的距离、代价或耗费等。每条边或弧都带权的图称为带权图(weighted graph)。

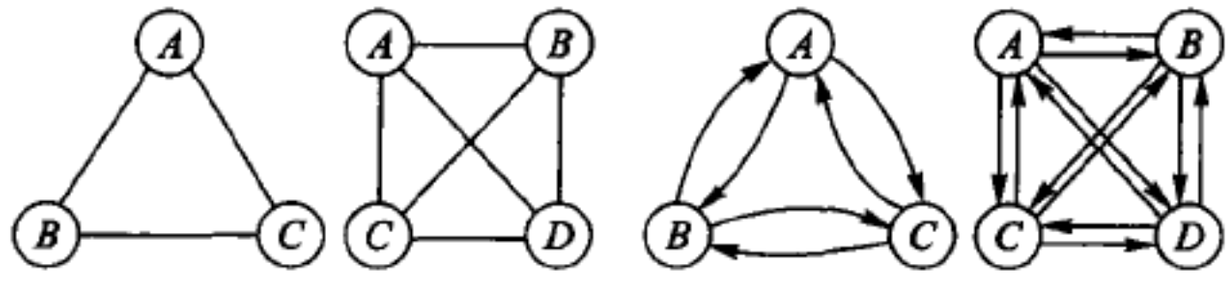
在下面的讨论中,将对图做一些限制:图中两个顶点之间的边不能多于一条,也就是说,无向图中不能有两条不同的(v_1,v_2)边,有向图也不能有 $<v_1,v_2>$ 出现两次的情况。另外,不考虑图中顶点到其自身的边,即若(v_1,v_2)或 $<v_1,v_2>$ 是图 G 的一条边或弧,则 $v_1 \neq v_2$。

通常用 n 表示图中顶点的数目,用 e 表示边或弧的数目。无向图中 e 的取值范围是 $0 \sim n(n-1)/2$;有向图中 e 的取值范围是 $0 \sim n(n-1)$。边数相对较少的图称为稀疏图(sparse graph),边数相对较多的图称为稠密图(dense graph)。任何两个顶点间都有边相关联的图称为完全图(complete graph),完全图显然具有最大的边数。图 7.3 所示为具有 3 个和 4 个顶点的完全图。

设 $G = <V,E>$ 是一个图,若 E' 是 E 的子集,V' 是 V 的子集,且 E' 中的边仅与 V' 中顶点相关联,则图 $G' = (V',E')$ 称为图 G 的子图(subgraph)。图 7.4 所示为图 7.2 中 G_1 和 G_2 的若干子图。

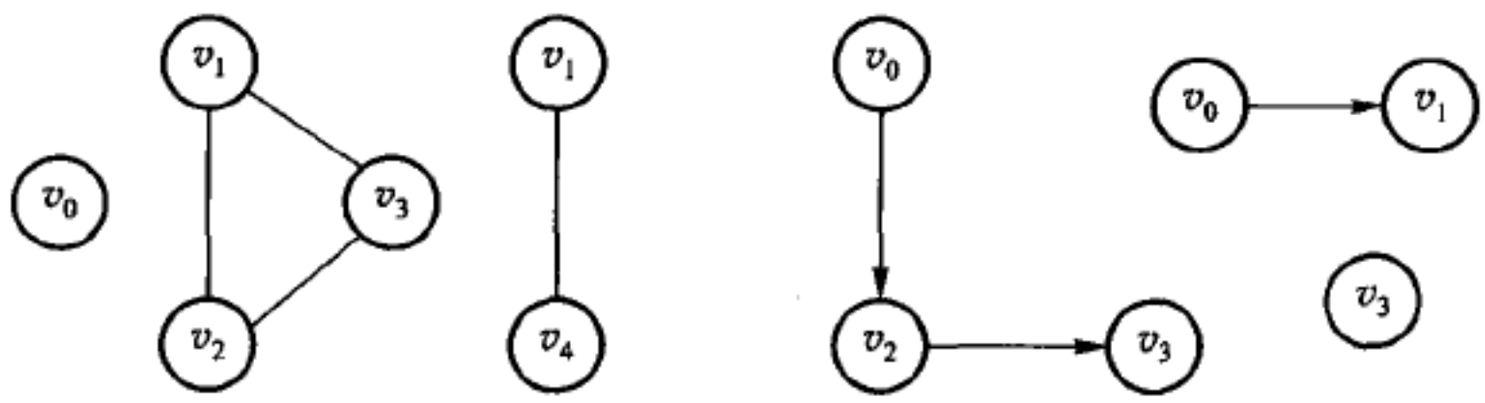
一条边所连接的两个顶点是相邻的(adjacent),被称为相邻顶点或者邻接点(neighbors),即如果边$(v_1,v_2) \in E$,那么 v_1 与 v_2 相邻接。连接一对邻接点 v_1 和 v_2 的边(v_1,v_2)被称为与顶点 v_1 和 v_2 相关联(incident)的边,或者称边(v_1,v_2)依附(incident)于顶点 v_1 和 v_2。在图 7.2 的无向图 G_1

中，与顶点 v_0 相关联的边有 (v_0, v_2) 和 (v_0, v_3)。若 $<v_1, v_2>$ 是有向图的一条边，则称顶点 v_1 邻接到顶点 v_2，顶点 v_2 邻接自顶点 v_1，而弧 $<v_1, v_2>$ 是与顶点 v_1 和 v_2 相关联的边。在图 7.2 的有向图 G_2 中，与顶点 v_1 相关联的边只有 $<v_0, v_1>$。



(a) 无向完全图　　(b) 有向完全图

图 7.3　完全图



(a)无向图G_1的若干子图　　(b)有向图G_2的若干子图

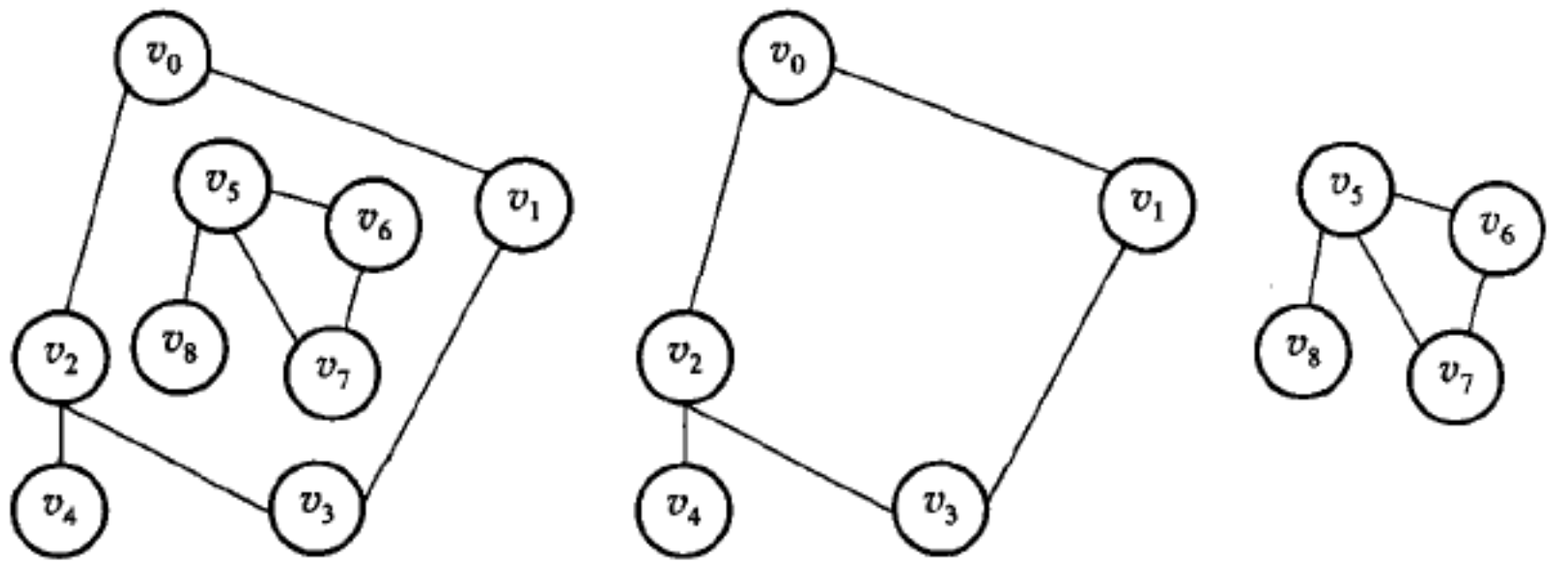
图 7.4　图 7.2 的若干子图

无向图中顶点 v 的度(degree) 是与该顶点相关联的边的数目，记为 $D(v)$。如果 G 是一个有向图，则把以顶点 v 为终点的弧的数目称为 v 的入度(in degree)，记为 $ID(v)$；把以顶点 v 为始点的弧的数目称为 v 的出度(out degree)，记为 $OD(v)$，并把出度为 0 的顶点称为终端顶点(terminal) 或叶子。有向图中顶点 v 的度(degree) 定义为 $D(v) = ID(v) + OD(v)$。例如，在图 7.2 的无向图 G_1 中，顶点 v_1 的度为 2。在图 7.2 的有向图 G_2 中，顶点 v_1 的入度为 1，出度为 0，度为 1。一般来说，如果图 G 中有 n 个顶点 $\{v_0, v_1, \cdots, v_n\}$，$e$ 条边，$D(v_i)$ 为顶点 v_i 的度数，显然有 $e = \frac{1}{2}\sum_{i=0}^{n-1} D(v_i)$。

无向图 $G = <V, E>$ 中从顶点 v_p 到顶点 v_q 的路径(path)是一个顶点序列 $(v_p = v_{i0}, v_{i1}, v_{i2}, \cdots, v_{im} = v_q)$，其中 $(v_{ij-1}, v_{ij}) \in E, 1 \leqslant j \leqslant m$。若 G 是有向图，则路径也是有向的，顶点序列应满足 $<v_{ij-1}, v_{ij}> \in E, 1 \leqslant j \leqslant m$。路径长度(length)定义为路径上的边(或弧)的数目。第一个顶点和最后一个顶点相同的路径称为回路或环(cycle)。序列中顶点不重复出现的路径称为简单路径(simple path)。除了第一个顶点和最后一个顶点外，其余顶点不重复的回路，称为简单回路(simple cycle)。不带回路的图称为无环图(acyclic graph)。不带回路的有向图称为有向无环图(directed acyclic graph, DAG)。一个有向图中，若存在一个顶点 v_0，从此顶点有路径可以到达图中其他所有顶点，则称此有向图为有根的图，v_0 称做图的根。

例如，图 7.2(a)的无向图 G_1 中三条边 (v_0, v_2)，(v_2, v_3)，(v_3, v_1) 构成一条由 v_0 到 v_1 的路径，简写成 (v_0, v_2, v_3, v_1)，这是一条简单路径；而在图 7.2(b)的有向图 G_2 中，(v_0, v_2, v_3) 是一条简单有向路径。

在无向图中，如果从顶点 v_i 到顶点 v_j 有路径，则称 v_i 和 v_j 是连通的(connected)。如果对于图中的任意两个顶点 $v_i, v_j \in V$，v_i 和 v_j 都是连通的，则称无向图 G 为连通图。连通分量(connected component)定义为无向图中的极大连通子图。图 7.2 中 G_1 是一个连通图。图 7.5(a)所示无向图 G_3 是非连通图，图 7.5(b)所示为 G_3 的两个连通分量。



(a) 非连通的无向图G_3　　(b) 非连通无向图G_3的连通分量

图 7.5　非连通无向图的连通分量示意图

对于有向图 $G = <V, E>$，若 G 中任意两个顶点 v_i 和 v_j $(v_i \neq v_j)$，都有一条从 v_i 到 v_j 的有向路径，同时还有一条从 v_j 到 v_i 的有向路径，则称有向图 G 是强连通图。例如，图 7.2(b)中的 G_2 不是强连通的，因为从 v_1 到 v_3 不存在一条路径。有向图强连通的极大子图称为该有向图的强连通分支或者强连通分量。例如，图 7.6 所示为图 7.2(b)中 G_2 的两个强连通分量。

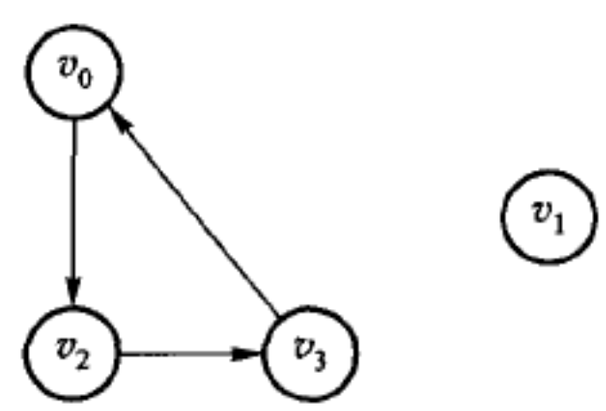
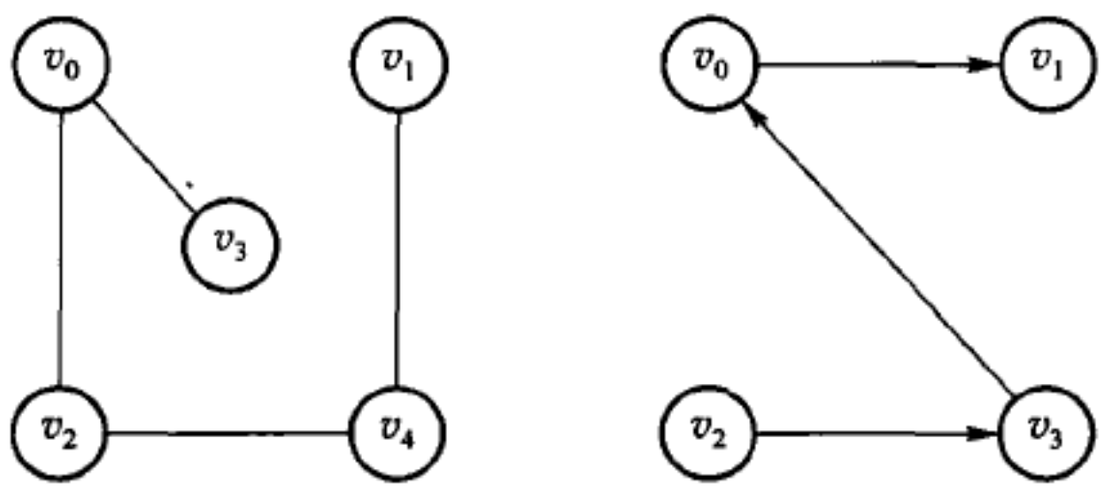


图 7.6　有向图 G_2 的两个强连通分量

一个连通图的生成树是含有该连通图全部顶点的一个极小连通子图。若连通图 G 的顶点个数为 n，则 G 的生成树的边数为 $n-1$。但是有 $n-1$ 条边的图不一定是生成树。如果无向图 G 的一个生成树 G' 上添加一条边，则 G' 中一定有环，因为依附于这条边的两个顶点有另一条路径。相反，如果 G' 的边数小于 $n-1$，则 G' 一定不连通。图 7.7 所示为图 7.2 所示无向图 G_1 和有向图 G_2 的一棵生成树。



(a) 无向图G_1的生成树　　(b) 有向图G_2的生成树

图 7.7　图 7.2 中无向图和有向图的生成树示例

自由树(free tree)是不带简单回路的无向图，它是连通的，并且具有 $n-1$ 条边。网络(net-

work)是带权的连通图,例如图 7.8 所示的 G_4是一个网络。

如果一个有向图只有一个顶点的入度为 0,其余顶点的入度均为 1,则称为有向树。一个有向图的生成森林由若干棵有向树组成,这些树的并集包含了原图的所有顶点,各有向树的弧不相交。图 7.9 所示为有向图生成森林的示例。

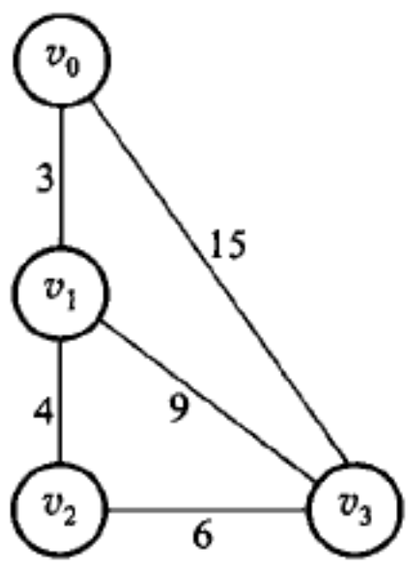
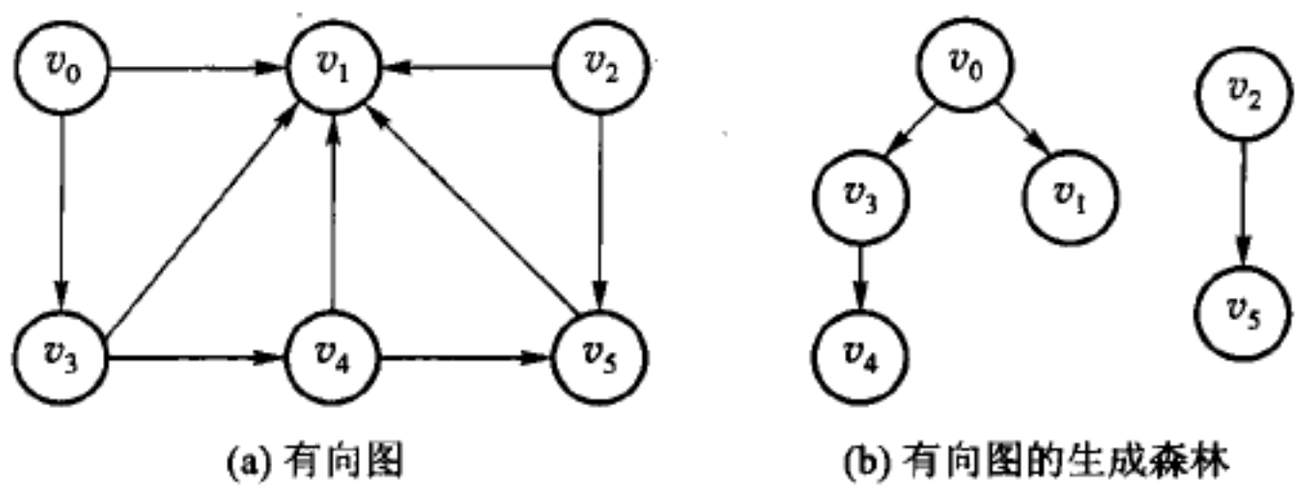


图 7.8 网络实例 G_4



(a) 有向图　　(b) 有向图的生成森林

图 7.9 有向图及其生成森林

7.2 图的抽象数据类型

与其他结构类似,图的基本操作主要是插入、删除和查找等。下面给出的图抽象数据类型定义中列出的成员函数是针对图的基本操作。需要说明的是,与无序树一样,图的顶点之间没有次序,从图的逻辑结构定义来看,任何一个顶点都可以看做是图的第一个顶点;任一个顶点的邻接点之间也是没有次序的。但是对于具体的图,一旦根据指定的存储结构建立完成之后,图中每一个顶点的所有邻接点之间就有了先后次序。因此,下面用到的顶点或边的次序是在具体的存储结构下的次序。

图的抽象数据类型由代码 7.1 定义。但是图的抽象数据类型列出的只是图的一部分操作,在后面的讲述过程中,将不断增加相应的操作。7.3 节将根据图的不同存储结构,以不同的方式来实现图的基类。

【代码 7.1】 图的抽象数据类型。

```
class Graph  {                                          // 图的抽象数据类型
public:
    int VerticesNum();                                  // 返回图的顶点个数
    int EdgesNum();                                     // 返回图的边数
    Edge FirstEdge(int oneVertex);                      // 返回依附于顶点 oneVertex 的第一条边
    Edge NextEdge(Edge preEdge);                        // 返回与 preEdge 有相同顶点的下一
条边
    bool setEdge(int fromVertex,int toVertex,int weight);  // 添加边
    bool delEdge(int fromVertex,int toVertex);          // 删除边
```

```
        bool IsEdge(Edge oneEdge);                    // oneEdge 是否是边
        int FromVertex(Edge oneEdge);                 // 返回 oneEdge 的始点
        int ToVertex(Edge oneEdge);                   // 返回 oneEdge 的终点
        int Weight(Edge oneEdge);                     // 返回 oneEdge 的权
    };
```

【代码 7.1 结束】

7.3 图的存储结构

图的存储结构应该根据具体问题的要求来设计。图的常用存储结构有相邻矩阵、邻接表和十字链表等。

7.3.1 相邻矩阵

图的相邻矩阵(adjacency matrix)或邻接矩阵表示顶点之间的邻接关系,即表示顶点之间有边或没有边的情况。设 $G=<V,E>$ 是一个有 n 个顶点的图,则图的相邻矩阵是一个二维数组 $A[n,n]$,定义如下:

$$A[i,j]=\begin{cases}1,若(v_i,v_j)\in E 或 <v_i,v_j>\in E\\0,若(v_i,v_j)\notin E 或 <v_i,v_j>\notin E\end{cases}$$

图 7.10(a)和(b)给出了无向图 G_1 和有向图 G_2 的相邻矩阵。

无向图的相邻矩阵是对称的。顶点 i 的度是第 i 行的元素或第 i 列的元素累加之和,即

$$D(v_i)=\sum_{i=0}^{n-1}A[i,j]=\sum_{j=0}^{n-1}A[i,j]$$

有向图的相邻矩阵不一定是对称的。因为如果相邻矩阵第 i 行第 j 列为 1,则表示有一条从顶点 v_i 到顶点 v_j 的弧,而此时不一定存在由顶点 v_j 到顶点 v_i 的弧。第 j 列的所有元素之和为顶点 v_j 的入度 $ID(v_j)$;第 i 行的元素之和是顶点 v_i 的出度 $OD(v_i)$,即

$$\begin{bmatrix}0&0&1&1&0\\0&0&0&1&1\\1&0&0&1&1\\1&1&1&0&0\\0&1&1&0&0\end{bmatrix}$$

(a) 无向图G_1 的相邻矩阵

$$\begin{bmatrix}0&1&1&0\\0&0&0&0\\0&0&0&1\\1&0&0&0\end{bmatrix}$$

(b) 有向图G_2 的相邻矩阵

图 7.10 图的相邻矩阵表示

$$ID(v_j)=\sum_{j=0}^{n-1}A[i,j],\qquad OD(v_i)=\sum_{i=0}^{n-1}A[i,j]$$

网络(带权图)的相邻矩阵定义如下:

$$A[i,j]=\begin{cases}w[i,j],若\ i\neq j,(v_i,v_j)\in E 或 <v_i,v_j>\in E\\\infty,若\ i\neq j 且(v_i,v_j)\notin E 或 <v_i,v_j>\notin E\\0,若\ i=j\end{cases}$$

如图 7.11 表示图 7.8 所示的网络 G_4 的相邻矩阵。

$$\begin{bmatrix} 0 & 3 & \infty & 15 \\ 3 & 0 & 4 & 9 \\ \infty & 4 & 0 & 6 \\ 15 & 9 & 6 & 0 \end{bmatrix}$$

图 7.11 带权图 G_4 及其相邻矩阵

对于带权的有向图，第 i 行中所有 $0 < A[i,j] < \infty$ 的元素个数之和就是顶点 v_i 的出度，第 j 列中所有 $0 < A[i,j] < \infty$ 的元素数目之和为顶点 v_j 的入度。

先把与存储结构无关的图的数据特征和相关操作实现为一个基类，代码 7.2 只给出了部分成员函数的实现。然后，代码 7.3 给出相邻矩阵实现的图类和部分成员函数，没有考虑无向图对称边的情况。对于 n 个顶点的图，相邻矩阵的空间代价都为 $O(n^2)$，与边数无关。

【代码 7.2】 图的基类。

```
class Edge{                                    // 边类
public :
    int from, to, weight;                      // from 是边的始点，to 是终点，weight 是边的权
    Edge( ){                                   // 缺省构造函数
        from = -1;to = -1;weight = 0;
    }
    Edge( int f,int t,int w){                  // 给定参数的构造函数
        from = f;to = t;weight = w;
    }
};
class Graph{
public :
    int numVertex;                             // 图中顶点的个数
    int numEdge;                               // 图中边的条数
    int * Mark;                                // 标记图的顶点是否被访问过
    int * Indegree;                            // 存放图中顶点的入度
    Graph( int numVert){                       // 图的构造函数
        numVertex = numVert;
        numEdge = 0;
        Indegree = new int[ numVertex ];
        Mark = new int[ numVertex ];
        for( int i = 0; i < numVertex; i ++ ){
            Mark[ i ] = UNVISITED;             // 标志位设为 UNVISITED
            Indegree[ i ] = 0;                 // 入度设为 0
        }
    }
    ~Graph( ){                                 // 析构函数
        delete [ ] Mark;                       // 释放 Mark 数组
        delete [ ] Indegree;                   // 释放 Indegree 数组
```

```
        }
        int VerticesNum() {                                  // 返回图中顶点的个数
            return numVertex;
        }
        bool IsEdge(Edge oneEdge) {                          // oneEdge 是否是边
            if(oneEdge.weight > 0 && oneEdge.weight < INFINITY && oneEdge.to >=0)
                return true;
            else return false;
        }
    };
```

【代码 7.2 结束】

代码 7.3 是图的相邻矩阵表示法的具体实现。

【代码 7.3】 用相邻矩阵表示图。

```
class Graphm: public Graph  {
private:
    int **matrix;                                            // 指向相邻矩阵的指针
public:
    Graphm(int numVert):Graph(numVert)  {                    // 构造函数
        int i,j;
        matrix = (int **)new int *[numVertex];               // 申请 matrix 数组行向量数组
        for(i=0;i < numVertex;i++)                           // 申请 matrix 数组行的存储空间
            matrix[i] = new int[numVertex];
        for(i=0;i < numVertex;i++)                           // 相邻矩阵的所有元素都初始化为 0
            for(j =0;j < numVertex;j++)
                matrix[i][j] =0;
    }
    ~Graphm()  {                                             // 析构函数
      for(int i=0;i < numVertex;i++)
          delete [] matrix[i];                               // 释放每个 matrix[i]行申请的空间
      delete [] matrix;                                      // 释放 matrix 指针指向的行向量空间
    }
    Edge FirstEdge(int oneVertex)  {                         // 返回依附于顶点 oneVertex 的第一条边
        Edge myEdge;
        myEdge.from = oneVertex;                             // 将顶点 oneVertex 作为边的始点
        for(int i=0; i < numVertex; i++) {                   // 找第一个 matrix[oneVertex][i]不为 0 的 i
            if(matrix[oneVertex][i] !=0) {
                myEdge.to = i;
                myEdge.weight = matrix[oneVertex][i];
```

```
                break;                                  // 找到第一条边就跳出循环
            }
        }
        return myEdge;
    }
    Edge NextEdge(Edge preEdge)  {                      // 返回与 preEdge 有相同顶点的下一条边
        Edge myEdge;
        myEdge.from = preEdge.from;                     // 边的始点与 preEdge 的始点相同
        if(preEdge.to < numVertex) {    // 如果 preEdge.to + 1 >= numVertex,将不存在下一条边
            for(int i = preEdge.to + 1;i < numVertex;i++) {   // 找下一个 matrix[oneVertex][i]
                                                              // 不为 0 的 i
                if(matrix[preEdge.from][i] != 0) {
                    myEdge.to = i;
                    myEdge.weight = matrix[preEdge.from][i];
                    break;                          // 找到下一条边就跳出循环
                }
            }
        }
        return myEdge;
    }
    void setEdge(int from,int to,int weight)  {     // 为图设置一条边
        if(matrix[from][to] <= 0) {                 // 如果原边不存在则边数和终点入度加 1
            numEdge++;
            Indegree[to]++;
        }
        matrix[from][to] = weight;                  // 设置边的权为 weight
    }
    void delEdge(int from,int to)  {                // 删除图的一条边
        if(matrix[from][to] > 0) {                  // 如果原边存在,则边数和终点入度减 1
            numEdge--;
            Indegree[to]--;
        }
        matrix[from][to] = 0;                       // 边权重修改为 0
    }
};
```

【代码 7.3 结束】

7.3.2 邻接表

当图中的边数较少时,相邻矩阵就会出现大量的 0 元素,存储这些 0 元素将耗费大量的存储

空间。对于稀疏图,可以采用邻接表存储法。

邻接表(adjacency list)表示法是一种链式存储结构,由一个顺序存储的顶点表和 n 个链接存储的边表组成。顶点表目有两个域:顶点数据域和指向此顶点边表指针域。边表把依附于同一个顶点 v_i 的边(即相邻矩阵中同一行的非0元素)组织成一个单链表。边表中的每一个表目都代表一条边,由两个主要的域组成:与顶点 v_i 邻接的另一顶点的序号、指向边表中下一个边表的目的指针。

顶点结点和边(或弧)结点的结构如下:

顶点结点

data	firstarc

边(或弧)结点

adjvex	nextarc	info

假设 (v_i,v_j) 是无向图中的一条边,在顶点 v_i 的边表里存储有边 (v_i,v_j) 对应的边结点,在顶点 v_j 的边表里还需存放 (v_j,v_i) 对应的边结点。因此,对于无向图,同一条边在邻接表中出现两次。图7.12给出了无向图 G_1 的邻接表表示。

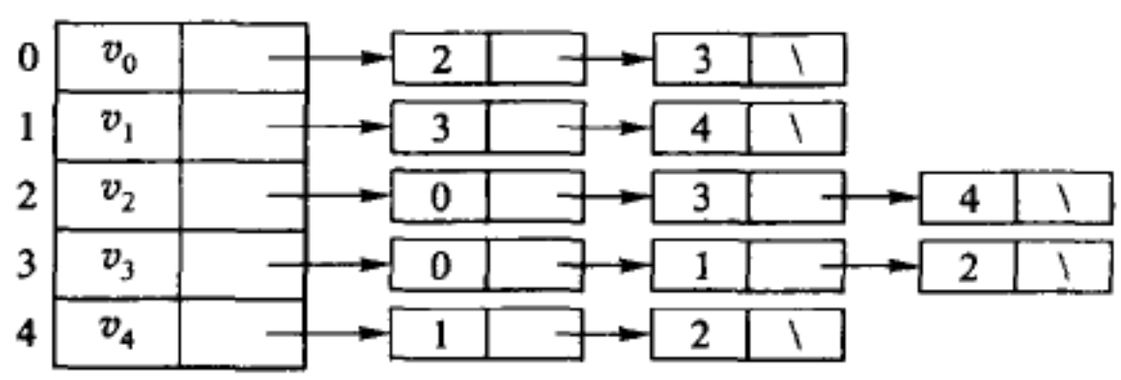
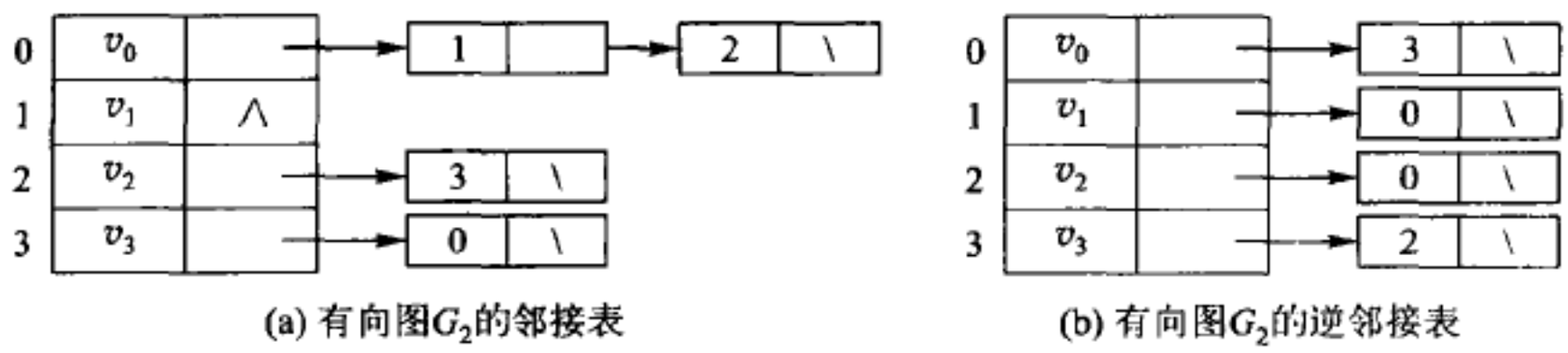


图7.12 无向图 G_1 的邻接表表示

图7.13(a)所示为有向图 G_2 的邻接表表示。对于有向图,一条弧在邻接表中只出现一次。顶点 v_i 的边表的表目个数是该顶点的出度,因此邻接表也称为出边表;要知道顶点 v_i 的入度,必须遍历整个邻接表,查看有多少个边表目中的顶点(即弧头顶点)为 v_i。为了便于确定顶点结点的入度,可以建立有向图的逆邻接表,如图7.13(b)所示。在有向图的逆邻接表中,顶点 v_i 的边表中表示的是以该顶点为终点的边,因此逆邻接表也称为入边表。逆邻接表中顶点 v_i 的边表的表目个数是该顶点的入度。



(a) 有向图 G_2 的邻接表　　(b) 有向图 G_2 的逆邻接表

图7.13 有向图 G_2 的邻接表和逆邻接表表示

对于带权图,每一个边结点需要设置一个域来保存权值信息。图7.14所示为带权图 G_4 的邻接表表示。

若图中有 n 个顶点和 e 条边,如果该图是无向图,则需要用到 n 个顶点结点和 $2e$ 个边结点;

若是有向图时，不考虑逆邻接表，只需要 n 个顶点结点和 e 个边结点。当边数 e 很小时，可以节省大量的存储空间。边表中表目顺序往往按照顶点编号从小到大排列。

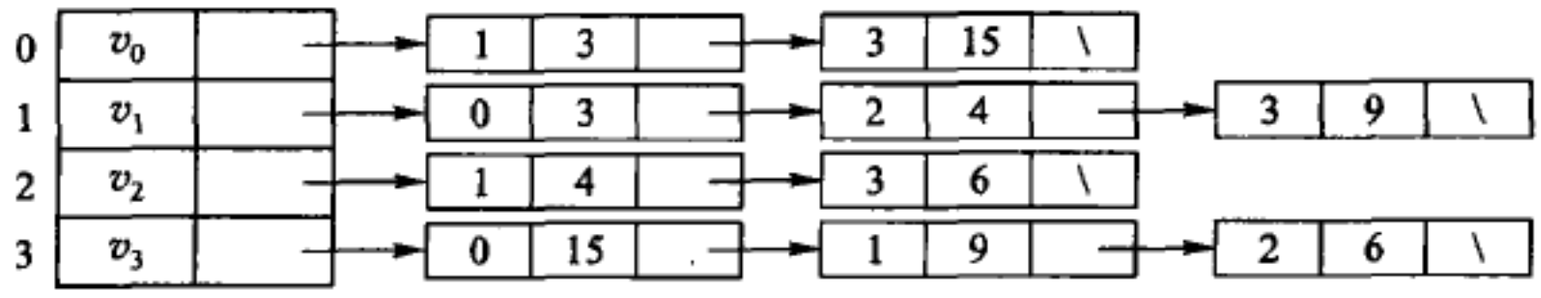


图 7.14 带权图 G_4 的邻接表表示

代码 7.4 给出的是图的邻接表表示法的类定义和实现，没有考虑无向图对称边的情况。

【代码 7.4】 用邻接表存储图。

```
struct listUnit{                                        // 邻接表中边结点的结构体定义
    int vertex;                                         // 边的终点
    int weight;                                         // 边的权
};
template < class Elem >
class Link{                                             // 链表元素类
public :
    Elem element;                                       // 表目的数据
    Link  * next;                                       // 指向下一个链表元素的指针
    Link( const Elem& elemval, Link  * nextval = NULL){ // 构造函数
        element = elemval;
        next = nextval;
    }
    Link( Link  * nextval = NULL){                      // 构造函数
        next = nextval;
    }
};
template < class Elem >
class LList{                                            // 链表类
public :
    Link < Elem >  * head;                              // 定义一个头指针以方便操作
    LList( ){                                           // 构造函数
        head = new Link < Elem > ( );
    }
};
class Graphl: public Graph{
private:
    LList < listUnit >  * graList;                      // 保存所有边表的数组
```

```
public:
    GraphI (int numVert):Graph(numVert){                    // 构造函数
        graList = new LList < listUnit > [numVertex];
    }
    Edge FirstEdge(int oneVertex){                          // 返回依附于顶点 oneVertex 的第一条边
        Edge myEdge;
        myEdge.from = oneVertex;                            // 将顶点 oneVertex 作为边的始点
        Link < listUnit >  * temp = graList[oneVertex].head;     // temp 指向边表的第一个元素
        if(temp -> next ! = NULL){                          // 如果顶点 oneVertex 边表非空
            myEdge.to = temp -> next -> element.vertex;
            myEdge.weight = temp -> next -> element.weight;
        }
        return myEdge;
    }
    Edge NextEdge(Edge preEdge){                            // 返回与 preEdge 有相同顶点的下一条边
        Edge myEdge;
        myEdge.from = preEdge.from;                         // 边的始点与 preEdge 的始点相同
        Link < listUnit >  * temp = graList[preEdge.from].head;  // temp 指向边表的第一个元素
        while (temp -> next ! = NULL && temp -> next -> element.vertex  <= preEdge.to)
            temp = temp -> next;                            // 确定 preEdge 的位置
        if (temp -> next ! = NULL){                         // 如果 preEdge 的下一条边存在
            myEdge.to = temp -> next -> element.vertex;
            myEdge.weight = temp -> next -> element.weight;
        }
        return myEdge;
    }
    void setEdge(int from,int to,int weight){               // 为图设置一条边
        Link < listUnit >  * temp = graList[from].head;     // temp 指向边表的第一个元素
        while (temp -> next ! = NULL && temp -> next -> element.vertex  <  to)
            temp = temp -> next;                            // 确定边(from,to)在边表中的位置
        if(temp -> next == NULL){                           // 如果边(from,to)不存在且是最后一条边
            temp -> next = new Link < listUnit > ;          // 在边表最后加入边(from,to)
            temp -> next -> element.vertex = to;
            temp -> next -> element.weight = weight;
            numEdge ++ ;                                    // 边数加 1
            Indegree[to] ++ ;                               // 终点的入度加 1
            return;
    }
```

```
        if(temp->next->element.vertex == to){                    // 如果边(from,to)在边表中已存在
            temp->next->element.weight = weight;                 // 只需要改变边的权值
            return;
        }
        if(temp->next->element.vertex > to){                     // 边(from,to)不存在且后面还有边
            Link<listUnit> *other = temp->next;                  // 临时存放指向后面链表元素的指针
            temp->next = new Link<listUnit>;                     // 在边表中插入边(from,to)
            temp->next->element.vertex = to;
            temp->next->element.weight = weight;
            temp->next->next = other;                            // 链接后面的链表元素
            numEdge++;                                           // 边数加 1
            Indegree[to]++;                                      // 终点的入度加 1
            return;
        }
    }
    void delEdge(int from,int to){                               // 删掉图的一条边
        Link<listUnit> *temp = graList[from].head;               // temp 指向边表的头结点,temp→next 为
                                                                 // 第一个元素
        while(temp->next != NULL && temp->next->element.vertex < to)
            temp = temp->next;                                   // 确定边(from,to)在边表中的位置,为了删除方便,
                                                                 // temp 指向的是被删除元素的前驱
        if(temp->next == NULL)return;                            // temp 的后继为空,则已经走到表尾,边不存在,返回
        if(temp->next->element.vertex > to)return;               // 边表的出边是按照编号排序的,边的终
                                                                 // 点标号如果超出参数 to,则该边不存
                                                                 // 在,返回
        if(temp->next->element.vertex == to){                    // 边(from,to)在边表中存在
            Link<listUnit> *other = temp->next->next;
            delete temp->next;                                   // 从边表中删除
            temp->next = other;                                  // 重新挂接其他链表元素
            numEdge--;                                           // 边数减 1
            Indegree[to]--;                                      // 终点的入度减 1
            return;
        }
    }
};
```

【代码 7.4 结束】

7.3.3 十字链表

十字链表(orthogonal list)是有向图的另一种链式存储结构,可以看成是邻接表和逆邻接表

的结合。表中对应于有向图的每一条弧有一个表目，共有5个域：头（headvex）和尾（tailvex）分别表示弧头（终点）和弧尾（始点）顶点序号；tailnextarc 链接指针指向下一条顶点以 tailvex 为弧尾的弧；headnextarc 指针指向下一条以顶点 headvex 为弧头的弧；此外，还有一个表示弧权值等信息的 info 域。顶点表目由3个域组成：data 域存放顶点的相关信息；firstinarc 链接指针指向第一条以该顶点为终点的弧；firstoutarc 链接指针指向第一条以该顶点为始点的弧。所有的顶点也可以放在顺序存储结构中。

下面给出十字链表的顶点结点和弧结点表目的结构表示：

data	firstinarc
	firstoutarc

顶点结点

tailvex	tailnextarc	headvex	headnextarc	info

弧（有向边）结点

图7.15给出了图7.2(b)所示的有向图 G_2 的十字链表表示，其中省略了弧的 info 域：

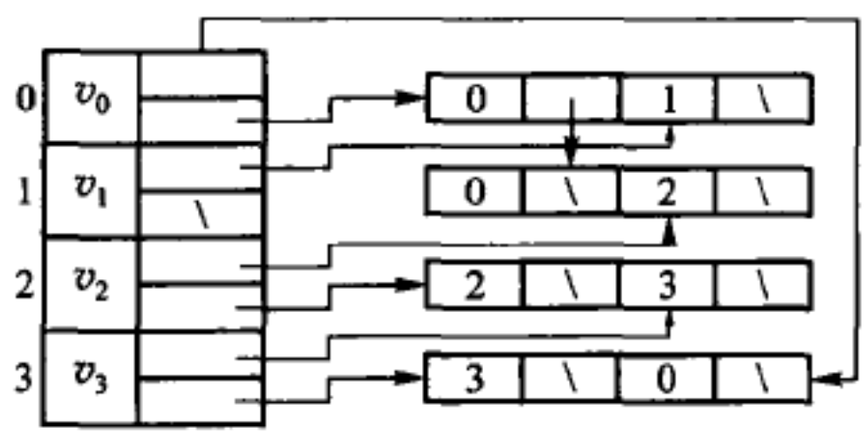


图7.15 有向图的十字链表示例

在十字链表中，很容易找到以 v_i 为始点和终点的弧。从顶点结点 v_i 的 firstoutarc 出发，由 tailnextarc 域链接起来的链表，正好是原来的邻接表结构。统计这个链表中的表目个数，可以得到顶点 v_i 的出度。如果从顶点结点 v_i 的 firstinarc 出发，由 headnextarc 域链接起来的链表，恰好是原来的逆邻接表结构。统计这个链表中的表目个数，可以求出顶点 v_i 的入度。

7.4 图的周游

图的周游（traversing graph）是指从图中的某一个顶点出发，按照一定的策略访问图中的每一个顶点，使得每一个顶点被访问且只被访问一次。图的周游算法是求解图的连通性问题、拓扑排序和关键路径等问题的基础。

图的周游比树的周游复杂，因为图中每一个顶点都可能和其他顶点相邻接。某一个顶点被访问后，还可能经过其他路径又回到这个顶点。为了避免重复访问同一个顶点，在周游图的过程中应该记下顶点是否已经被访问，若遇到已访问的顶点则不再访问。

深度优先周游和广度优先周游是两种基本的周游方法，对有向图和无向图都适用。

7.4.1 深度优先周游

图的深度优先搜索(depth-first search,DFS)类似于树的先根次序周游。深度优先周游方法的特点是尽可能先对纵深方向进行搜索。

对于给定的图 $G = <V,E>$,初始状态是 V 中的所有顶点都未被访问。首先,选取一个顶点开始搜索。设 $v_0 \in V$ 是源点,访问顶点 v_0并标记为 VISITED,然后访问 v_0邻接到的未被访问过的顶点 v_1,再从 v_1出发递归地按照深度优先的方式周游。当遇到一个所有邻接顶点都被访问过的顶点 w 时,则回到已访问顶点序列中最后一个拥有未被访问的相邻顶点的顶点 u,再从 u 出发递归地按照深度优先的方式周游。重复上述过程直至从 v_0出发的所有边都已检测过为止。此时,图中所有由源点有路径可达的顶点都已被访问过。若图 G 是连通图,则周游过程结束;否则,选择一个尚未访问的顶点作为新的源点进行深度优先搜索,直至图中所有顶点均被访问。事实上,深度优先搜索的结果是沿着图的某一分支搜索,直到它的末端,然后回溯,沿着另一分支进行同样的搜索,依次类推。

以图 7.16 所示的有向图 G 为例,深度优先搜索的周游过程如下:假设从顶点 v_0出发进行搜索,首先将顶点 v_0的标记位设置为 VISITED,然后从 v_0的邻接表中取出仅有的邻接点 v_1。因为 v_1从未曾访问,将顶点 v_1的标志位设置为 VISITED,然后从顶点 v_1的邻接表中选取一个邻接点。假设在顶点 v_1的邻接表中 v_2排在 v_3之前,于是先访问 v_2,将顶点 v_2的标志位设置为 VISITED。因为 v_2的邻接表中除了已被访问的顶点 v_0外,没有其他的邻接点,所以搜索过程回退到顶点 v_1。找到下一个邻接顶点 v_3,将顶点 v_3的标志位设置为 VISITED。在查找顶点 v_3的邻接表时,发现 v_3的两个邻接点 v_0和 v_2都已经被访问过,于是回退到顶点 v_1,此时也出现了相同的情况,最后回退到顶点 v_0。要完成图的周游,还有其他顶点没有被访问过,因此还得继续找一个 UNVISITED 顶点继续深度优先搜索。例如,此时首先访问 v_4,继续深度搜索,依次访问顶点 v_5和 v_6。

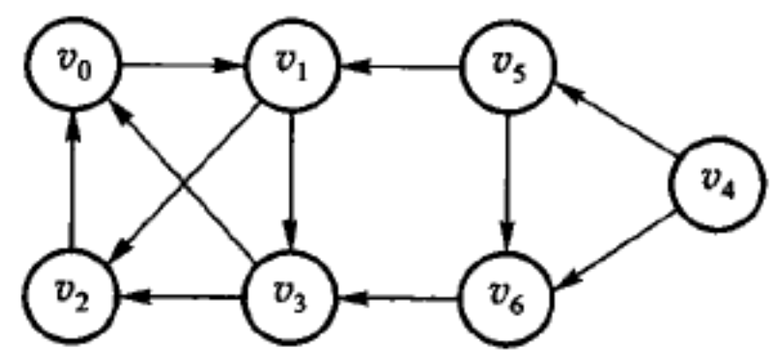


图 7.16 有向图 G

对于确定了的存储结构和源点,深度优先周游产生的深度优先序列是唯一的。

算法 7.5 是图的深度优先周游算法的递归实现。

【算法 7.5】 图的深度优先周游(DFS)算法。

```
void DFS(Graph& G, int v){                          // 深度优先搜索的递规实现
    G.Mark[v] = VISITED;                            // 将标记位设置为 VISITED(被访问过)
    Visit(G,v);                                     // 访问顶点 v
    for (Edge e = G.FirstEdge(v);G.IsEdge(e);e = G.NextEdge(e))
        if(G. Mark[G.ToVertex(e)] == UNVISITED)
            DFS(G, G.ToVertex(e));
```

【算法 7.5 结束】

周游图的过程实质上是搜索每个顶点的邻接点的过程，时间主要耗费在从该顶点出发搜索它的所有邻接点上。分析上述算法，对于具有 n 个顶点和 e 条边的无向图或有向图，深度优先周游算法对图中每个顶点至多调用一次 DFS( ) 函数。用相邻矩阵表示图时，共需检查 n^2 个矩阵元素，所需时间为 $O(n^2)$；用邻接表表示图时，找邻接点需将邻接表中所有边结点检查一遍，需要时间 $O(e)$，对应的深度优先搜索算法的时间复杂度为 $O(n+e)$。

7.4.2 广度优先周游

系统地访问图的所有顶点的另一个方法是广度优先搜索(breadth-first search，BFS)。其周游的过程是：从图中的某个顶点 v 出发，访问并标记了顶点 v 之后，横向搜索 v 的所有邻接点 u_1，u_2，…，u_t。在依次访问 v 的各个未被访问的邻接点之后，再从这些邻接点出发，依次访问与它们邻接的所有未曾访问过的顶点。重复上述过程直至图中所有与源点 v 有路径相通的顶点都被访问过为止。若 G 是连通图，则周游完成；否则，在图 G 中选一个尚未访问的顶点作为新源点继续广度优先搜索。

例如，对于图 7.16 所示的有向图 G，首先访问 v_0 和 v_0 的邻接点 v_1，然后访问顶点 v_1 的所有邻接点 v_2 和 v_3。由于顶点 v_2 和 v_3 的所有邻接点都已经被访问过，选择未被访问过的顶点 v_4 继续广度优先搜索，即访问顶点 v_4 和 v_4 的邻接点 v_5、v_6。

广度优先搜索的过程类似于树的按层次次序周游。可以使用 FIFO 队列保存已访问过的顶点，从而使得先访问的顶点的邻接点在下一轮被优先访问到。在搜索过程中，每访问到一个顶点后将其入队，当队头元素出队时将其未被访问的邻接点入队，每个顶点只入队一次。

算法 7.6 是图的广度优先周游算法的实现。

【算法 7.6】 图的广度优先周游(BFS)算法。

```
void BFS(Graph& G, int v){
    using std::queue;                          // 使用 STL 中的队列
    queue<int> Q;
    Visit(G,v);                                // 访问顶点 v
    G.Mark[v] = VISITED;                       // 将标记位设置为 VISITED
    Q.push(v);                                 // 顶点 v 入队列
    while (! Q.empty()){                       // 如果队列非空
        int u = Q.front();                     // 获得队列头部元素
        Q.pop();                               // 队列头部元素出队
        // 与该顶点邻接的所有未访问过的顶点入队
        for(Edge e = G.FirstEdge(u);G.IsEdge(e);e = G.NextEdge(e))
            if(G. Mark[G.ToVertex(e)] == UNVISITED){
```

```
                Visit(G, G.ToVertex(e));
                G.Mark[G.ToVertex(e)] = VISITED;
                Q.push(G.ToVertex(e));
            }
        }
    }
```

【算法 7.6 结束】

广度优先搜索实质上与深度优先相同，只是访问顺序不同而已，二者的时间复杂度也相同。

7.4.3 拓扑排序

有向图的边可以看做顶点之间制约关系的描述。在工程实践中，有些工程的进行经常受到一定条件的约束，例如一个工程项目通常由若干个子工程组成，某些子工程完成之后另一些子工程才能开始。

一个无环的有向图称为有向无环图(directed acyclic graph，DAG)。有向无环图常用来描述一个过程或一个系统的进行过程。对于有向无环图 $G = <V,E>$，如果顶点序列满足：存在顶点 v_i 到 v_j 的一条路径，那么在序列中顶点 v_i 必在顶点 v_j 之前，顶点集合 V 的这种线性序列称做一个拓扑序列(topological order)。根据有向图建立拓扑序列的过程称为拓扑排序(topological sorting)。对有向无环图的顶点进行拓扑排序，对应于实际问题就是对各项子工程排出一个线性顺序关系。如果条件限制这些工作必须串行，就应该按拓扑次序安排依次执行。拓扑排序可以解决先决条件问题，即以某种线性顺序来组织多项任务，以便能够在满足先决条件的情况下逐个完成各项任务。

例如，一个计算机专业的学生需要学习的一系列课程如表 7.1 所示。其中，有些课程是基础课，独立于其他课程，例如“高等数学”、“程序设计基础”；而另外一些课程必须学完作为它的基础的先修课程后才能开始，例如学习“数据结构”就先要学习“程序设计基础”和“离散数学”。

表 7.1 计算机专业的课程

课程代号	课程名称	先修课程
c_0	高等数学	无
c_1	程序设计基础	无
c_2	离散数学	c_0、c_1
c_3	数据结构	c_1、c_2
c_4	算法语言	c_1
c_5	编译原理	c_3、c_4

续表

课程代号	课程名称	先修课程
c_6	操作系统	c_3、c_8
c_7	普通物理	c_0
c_8	计算机原理	c_7

这些先决条件定义了课程之间的先后关系，这种关系可以用图7.17所示的有向无环图表示。图中顶点表示课程，有向边 $<c_i,c_j>$ 表示课程 c_i 是课程 c_j 的先修课程。

图7.17所示的有向无环图进行拓扑排序可以得到拓扑序列($c_0,c_1,c_2,c_3,c_4,c_5,c_7,c_8,c_6$)，也可以得到($c_0,c_7,c_8,c_1,c_4,c_2,c_3,c_6,c_5$)等。当然，一个有向无环图顶点的拓扑序列不是唯一的，对图7.17所示有向无环图也可以构造其他的拓扑有序序列。这样，学生就可以按照某一个拓扑排序的顺序来安排学习计划。

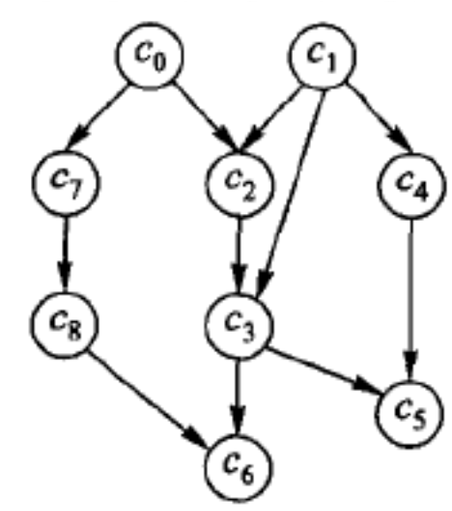


图7.17 表示课程优先关系的有向无环图

进行有向图的拓扑排序方法如下：

(1) 从有向图中选出一个没有前驱(入度为0)的顶点并输出。

(2) 删除图中该顶点和所有以它为起点的弧。

不断重复上述两个步骤，会出现两种情形：要么有向图中顶点全部被输出，要么当前图中不存在没有前驱的顶点。当图中的顶点全部输出时，就完成了有向无环图的拓扑排序；当图中还有顶点没有输出时，说明有向图中含有环。可见，拓扑排序可以检查有向图是否存在环。

以图7.17为例，c_0和c_1没有前驱，可以选其中任何一个输出。假设先输出c_0，删除顶点c_0和弧$<c_0,c_2>$、$<c_0,c_7>$之后，顶点c_1和c_7没有前驱，则不妨从中选择c_1输出，然后删去顶点c_1和弧$<c_1,c_2>$、$<c_1,c_3>$、$<c_1,c_4>$。依次类推，就可以得到有向无环图的拓扑有序序列。

下面用邻接表作为有向图的存储结构来实现有向图的拓扑排序。每个顶点中加入一个存放该顶点的入度的域(indegree)。这样，检查顶点数组就可以方便地找出入度为0的顶点，即没有前驱的顶点。删除该顶点及以它为尾的弧，即将边表中所有弧头顶点的入度减1。

为了减少查找度为0的顶点的次数，可把入度为0的顶点构造成一个队列，使得每次查找入度为0的顶点时只要从队列中取出第一个顶点即可，而不必检查整个顶点表。删除入度为0的顶点，如果此时某个顶点的入度减为0，就将其插入队列。

算法7.7是图拓扑排序的队列实现。

【算法7.7】 用队列实现的图拓扑排序。

```
void TopsortbyQueue(Graph& G){
    for(int i=0;i < G.VerticesNum();i++)                    // 初始化 Mark 数组
        G.Mark[i]=UNVISITED;
    using std::queue;                                       // 使用 STL 中的队列
```

```
queue < int > Q;
for( i =0; i < G. VerticesNum( ); i ++ )                    // 入度为 0 的顶点入队
    if( G. Indegree[ i ] ==0)
        Q. push( i );
while ( ! Q. empty( ) ){                                    // 如果队列非空
    int v = Q. front( );                                    // 获得队列顶部元素
    Q. pop( );                                              // 队列顶部元素出队
    Visit( G,v);                                            // 访问顶点 v
    G. Mark[ v ] = VISITED;                                 // 将标记位设置为 VISITED
    for( Edge e = G. FirstEdge( v ) ;G. IsEdge( e ) ;e = G. NextEdge( e ) ){
        G. Indegree[ G. ToVertex( e ) ] -- ;                // 与该顶点相邻的顶点入度减 1
        if( G. Indegree[ G. ToVertex( e ) ] ==0)            // 如果顶点入度减为 0 则入队
            Q. push( G. ToVertex( e ) );
    }
}
for( i =0; i < G. VerticesNum( ); i ++ )                    // 利用标记位可以判断图中是否有环
    if( G. Mark[ i ] == UNVISITED) {
        cout < < "此图有环!";
        break;
    }
}
```

【算法 7.7 结束】

对于有 n 个顶点和 e 条边的图，若其存储结构用邻接表表示，实现拓扑排序开始时建立入度为 0 的顶点队列需要检查所有顶点一次，所需的时间 $O(n)$，然后排序中每个顶点输出一次，更新顶点的入度需要检查每条边总计 e 次，因此执行时间为 $O(n+e)$。

7.5 最短路径

在实际生活中，经常会碰到这样的问题：一位旅客要从 A 城市到达 B 城市，希望选择一条路程最短的路线。对这样的问题进行建模需要用带权图，用顶点表示各个城市，边表示城市之间的交通线路，线路长度为边的权值。此时，旅客的目标转化为从这个带权图中找到 A 城市和 B 城市之间路径上权值之和最小的方案。图 7.18 所示为最短路径的一个示例，由 A 到 B 的最短路径即为 ACB。

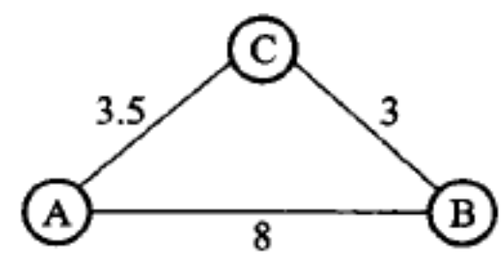


图 7.18 最短路径的示例

一般称路径上的第一个顶点为源点(source)，最后一个顶点为汇点(sink)，或终点(destination)。本节将讨论单源最短路径和所有顶点对之间的最短路径两类问题。

7.5.1 单源最短路径

给定一个带权图 $G = <V,E>$，其中每条边(v_i,v_j)上的权 $W[v_i,v_j]$是一个非负实数。另外，给定 V 中的一个顶点 s 充当源点。现在要计算从源点 s 到所有其他各顶点的最短路径，这个问题通常称为单源最短路径(single-source shortest paths)问题。图 7.19 所示为单源最短路径的一个示例。

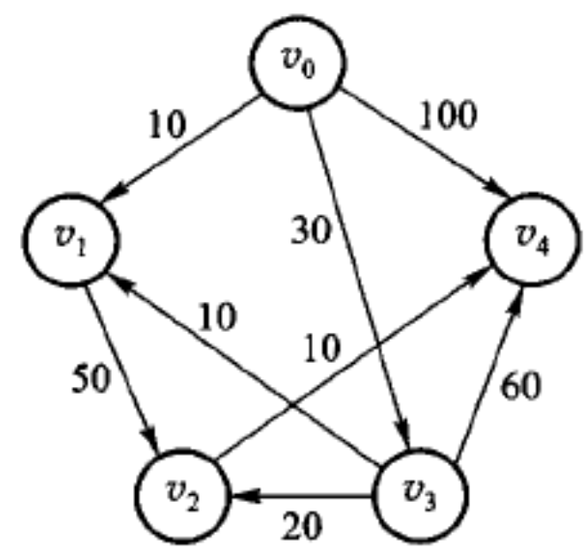


图 7.19 单源最短路径的示例

解决单源最短路径问题的一个常用算法是 Dijkstra 算法，它是由 E. W. Dijkstra 提出的一种按路径长度递增的次序产生到各顶点最短路径的贪心算法。其基本思想是：把图的顶点集合化分成两个集合 S 和 $V-S$。第一个集合 S 表示最短距离已经确定的顶点集，其余的顶点放在另一个集合 $V-S$ 中。初始时，集合 S 只包含源点，即 $S=\{s\}$，此时只有源点到自己的最短距离是已知的。设 v 是 V 中的某个顶点，把从源点 s 到顶点 v 且中间只经过集合 S 中顶点的路径称为从源点到 v 的特殊路径，并用数组 D 来记录当前所找到的从源点 s 到每个顶点的最短特殊路径长度。D 的初始状态为：如果从源点 s 到顶点 v 有弧，则 $D[v]$记为弧的权值；否则将 $D[v]$置为无穷大。Dijkstra 算法每次从尚未确定最短路径长度的集合 $V-S$ 中取出一个最短特殊路径长度最小的顶点 u，将 u 加入集合 S，同时修改数组 D 中由 s 可达的最短路径长度：若加进 u 做中间顶点，使得 v_i的最短特殊路径长度变短，则修改 v_i的距离值(即当 $D[u]+W[u,v_i]<D[v_i]$时，令 $D[v_i]=D[u]+W[u,v_i]$)。然后，重复上述操作，一旦 S 包含了所有 V 中的顶点，D 中各顶点的距离值就记录了从源点 s 到该顶点的最短路径长度。

距离数组中还可以设立一个域来记录从源点到顶点 v 的最短路径上 v 前面经过的一个顶点，这样就可以推导出最短路径。初始时，对所有的 $v \neq s$，均设置其前一个顶点为 s。在用 Dijkstra 算法更新最短路径长度时，只要 $D[u]+W[u,v] < D[v]$，就设置 v 的前一个顶点为 u，否则不做修改。当 Dijkstra 算法终止时就可以找到源点 s 到顶点 v 的最短路径上每个顶点的前一个顶点，从而找到源点 s 到顶点 v 的最短路径。

例如，对于图 7.19 所示的有向图，应用 Dijkstra 算法计算从源点 v_0到其他顶点间的最短路径长度的过程如表 7.2 所示。

表 7.2 用 Dijkstra 算法的处理过程，源顶点为 v_0

迭代步数	S	v_0	v_1	v_2	v_3	v_4
初始状态	$\{v_0\}$	Length:0 pre:0	length:**10** pre:0	length:∞ pre:0	length:30 pre:0	length:100 pre:0
1	$\{v_0,v_1\}$	Length:0 pre:0	length:10 pre:0	length:60 pre:1	length:**30** pre:0	length:100 pre:0
2	$\{v_0,v_1,v_3\}$	Length:0 pre:0	length:10 pre:0	length:**50** pre:3	length:30 pre:0	length:90 pre:3

续表

迭代步数	S	v_0	v_1	v_2	v_3	v_4
3	$\{v_0,v_1,v_3,v_2\}$	length:0 pre:0	length:10 pre:0	length:50 pre:3	length:30 pre:0	length:**60** pre:2
4	$\{v_0,v_1,v_3,v_2,v_4\}$	length:0 pre:0	length:10 pre:0	length:50 pre:3	length:30 pre:0	length:60 pre:2

Dijkstra 算法是用贪心算法设计的一个典型的例子,它每次从 $V-S$ 中选择具有最短特殊路径长度的顶点 u,确定从源点到 u 的最短路径长度。贪心法可以认为是动态规划法的特例,这种算法为什么得到的就是最短路径呢?事实上,如果存在一条从源点到 u 且长度比 $D[u]$ 更短的路径,假设这条路径经过第二个集合的某些顶点(不妨设为 x)才到达 u。在这条路径上分别用 $d(s,x)$、$d(x,u)$ 和 $d(s,u)$ 表示源点 s 到顶点 x、顶点 x 到顶点 u 和源点 s 到顶点 u 的距离,则有 $d(s,u)=d(s,x)+d(x,u)<D[u]$,且 $D[x]\leqslant d(s,x)$。由边的权值的非负性,推得 $D[x]<D[u]$。这与是第二个集合中距离值最小的顶点矛盾,所以每次加入第一个集合的 u 的距离值就是从 s 到 u 的最短路径长度。

算法 7.8 是具体实现的 Dijkstra 算法,其中 Dist 类在算法中可以直接比较大小。

【算法 7.8】 Dijkstra 算法。

```
class Dist{                                  // Dist 类,Dijkstra 和 Floyd 算法用于保存最短路径信息
public :
    int index;                               // 顶点的索引值,仅 Dijkstra 算法用到
    int length;                              // 当前最短路径长度
    int pre;                                 // 路径最后经过的顶点
};
void Dijkstra(Graph& G, int s, Dist * &D){          // s 是源点
    D = new Dist[G. VerticesNum()];                 // 数组 D 记录当前找到的最短特殊路径长度
    for(int i=0; i < G. VerticesNum(); i++){        // 初始化 Mark 数组、D 数组
        G. Mark[i] = UNVISITED;
        D[i]. index = i;
        D[i]. length = INFINITE;
        D[i]. pre = s;
    }
    D[s]. length = 0;                               // 源点到自身的路径长度设置为 0
    MinHeap < Dist > H(G. EdgesNum());              // 最小值堆用于找出最短路径
    H. Insert(D[s]);
    for (i=0; i < G. VerticesNum(); i++){
        bool FOUND = false;
        Dist d;
        while (!H. isEmpty()){
            d = H. RemoveMin();                     // 获得到 s 路径长度最小的顶点
```

```
            if(G. Mark[d. index] == UNVISITED){ // 如果未访问过则跳出循环
                FOUND = true;
                break;
            }
        }
        if(!FOUND)                          // 若没有符合条件的最短路径则跳出本次循环
            break;
        int v = d. index;
        G. Mark[v] = VISITED;               // 将该顶点的标记位设置为 VISITED
        // 加入 v 以后需要刷新 D 中 v 的邻接点的最短路径长度
        for(Edge e = G. FirstEdge(v);G. IsEdge(e);e = G. NextEdge(e))
            if(D[G. ToVertex(e)]. length > (D[v]. length + G. Weight(e))){
                D[G. ToVertex(e)]. length = D[v]. length + G. Weight(e);
                D[G. ToVertex(e)]. pre = v;
                H. Insert(D[G. ToVertex(e)]);
            }
    }
}
```

【算法 7.8 结束】

对于 n 个顶点 e 条边的图，图中的任何一条边都可能在最短路径中出现，因此最短路径算法对每条边至少都要检查一次。算法 7.8 采用最小堆来选择权值最小的边，因此每次改变最短特殊路径长度时需要对堆进行一次重排，此时的时间复杂度为 $O((n+e)\log e)$，适合于稀疏图。如果像算法 7.10 介绍的 Prim 算法那样，通过直接比较 D 数组元素，确定代价最小的边就需要总时间 $O(n^2)$；取出最短特殊路径长度最小的顶点后，修改最短特殊路径长度共需要时间 $O(e)$，因此共需要花费 $O(n^2)$ 的时间，这种方法适合于稠密图。

7.5.2 每对顶点之间的最短路径

给定一个带权图 $G=<V,E>$，其中每条边 (v_i,v_j) 上的权 $W[v_i,v_j]$ 是一个非负实数。要求计算对任意的顶点有序对 $<v_i,v_j>$ 找出从顶点 v_i 到顶点 v_j 的最短路径。这个问题通常称为带权图的所有顶点对之间的最短路径(all-pairs shortest paths)问题。

解决这一问题可以每次以一个顶点为源点，重复执行 Dijkstra 算法 n 次，这样就可以求得所有的顶点对之间的最短路径及其最短路径长度，其时间复杂度为 $O(n^3)$。

下面介绍求所有顶点对之间最短路径的比较直接的 Floyd 算法，这是一个典型的动态规划法，先自底向上分别求解子问题的解，然后由这些子问题的解得到原问题的解。这个算法的时间复杂度也是 $O(n^3)$，但形式上比较简单。

Floyd 算法用相邻矩阵 adj 来表示带权有向图，该算法的基本思想是：初始化 $adj^{(0)}$ 为相邻矩阵 adj，在矩阵 $adj^{(0)}$ 上做 n 次迭代，递归地产生一个矩阵序列 $adj^{(1)},\cdots,adj^{(k)},\cdots,adj^{(n)}$。其中，经过第 k 次迭代，$adj^{(k)}[i,j]$ 的值等于从顶点 v_i 到顶点 v_j 路径上所经过的顶点序号不大于 k 的最短路径长度。由于进行第 k 次迭代时已求得矩阵 $adj^{(k-1)}$，那么从顶点 v_i 到顶点 v_j 中间顶点的序号不大于 k 的最短路径有两种情况：一种是中间不经过顶点 v_k，此时有 $adj^{(k)}[i,j]=adj^{(k-1)}[i,j]$；另一种是中间经过顶点 v_k，此时 $adj^{(k)}[i,j] < adj^{(k-1)}[i,j]$，这条由顶点 v_i 经过 v_k 到顶点 v_j 的中间顶点序号不大于 k 的最短路径由两段组成：一段是从顶点 v_i 到顶点 v_k 的中间顶点序号不大于 $k-1$ 的最短路径，另一段是从顶点 v_k 到顶点 v_j 的中间顶点序号不大于 $k-1$ 的最短路径，路径长度应为这两段长度之和，用下面的公式计算：$adj^{(k)}[i,j]=adj^{(k-1)}[i,k]+adj^{(k-1)}[k,j]$。综合这两种情况有

$$adj^{(k)}[i,j]=\min\{adj^{(k-1)}[i,j],adj^{(k-1)}[i,k]+adj^{(k-1)}[k,j]\}$$

这样 $adj^{(n)}[i,j]$ 等于从顶点 v_i 到顶点 v_j 中间顶点序号不大于 n 的最短路径长度，也就是所要求的从顶点 v_i 到顶点 v_j 的最短路径。

有时，除了计算出一个带权图中从任意一个顶点到其他顶点的最短路径长度，还需要确定相应的最短路径。为此，可以设置一个 $n\times n$ 的矩阵 $path$，$path[i,j]$ 是由顶点 v_i 到顶点 v_j 的最短路径上排在顶点 v_j 前面的那个顶点，即当 k 在 Floyd 算法中使得 $adj^{(k)}[i,j]$ 达到最小值，就置 $path[i,j]=k$。如果当前没有最短路径，就将 $path[i,j]$ 置为 -1。

例如，图 7.20 所示的有向带权图，矩阵 $adj^{(0)}$ 的初值以及每次迭代的结果如图 7.21 所示。

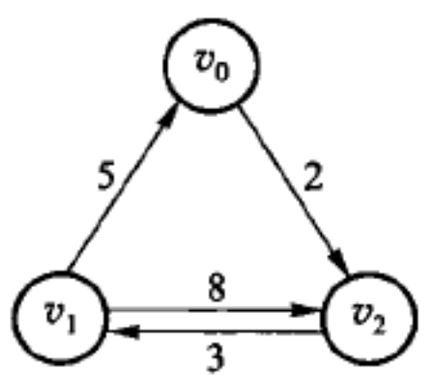


图 7.20 每对顶点间的最短路径的示例

$$adj^{(0)}=\begin{bmatrix}0&\infty&2\\5&0&8\\\infty&3&0\end{bmatrix}\qquad path=\begin{bmatrix}0&-1&0\\1&1&1\\-1&2&2\end{bmatrix}$$

$$adj^{(1)}=\begin{bmatrix}0&\infty&2\\5&0&7\\\infty&3&0\end{bmatrix}\qquad path=\begin{bmatrix}0&-1&0\\1&1&0\\-1&2&2\end{bmatrix}$$

$$adj^{(2)}=\begin{bmatrix}0&\infty&2\\5&0&7\\8&3&0\end{bmatrix}\qquad path=\begin{bmatrix}0&-1&0\\1&1&0\\1&2&2\end{bmatrix}$$

$$adj^{(3)}=\begin{bmatrix}0&5&2\\5&0&7\\8&3&0\end{bmatrix}\qquad path=\begin{bmatrix}0&2&0\\1&1&0\\1&2&2\end{bmatrix}$$

图 7.21 每对顶点间的最短路径的 Floyd 算法迭代过程

算法 7.9 是求每对顶点之间最短距离的 Floyd 算法。

【算法 7.9】 Floyd 算法。

```
void Floyd(Graph& G, Dist * * &D){
    int i,j,v;
    D = new Dist * [G. VerticesNum()];                    // 为数组 D 申请空间
```

```
for( i = 0; i < G. VerticesNum( ) ;i ++ )
D[ i] = new Dist[ G. VerticesNum( ) ] ;
for( i = 0;i < G. VerticesNum( ) ;i ++ )                    // 初始化数组 D
    for( j = 0;j < G. VerticesNum( ) ;j ++ ) {
        if( i == j) {
            D[ i] [ j]. length = 0;
            D[ i] [ j]. pre = i;
        }
        else {
            D[ i] [ j]. length = INFINITE;
            D[ i] [ j]. pre = -1;
        }
    }
for( v = 0;v < G. VerticesNum( ) ;v ++ )
    for( Edge e = G. FirstEdge( v) ; G. IsEdge( e) ;e = G. NextEdge( e) ) {
        D[ v] [ G. ToVertex( e) ]. length = G. Weight( e) ;
        D[ v] [ G. ToVertex( e) ]. pre = v;
    }
// 顶点 i 到顶点 j 的路径经过顶点 v 如果变短,则更新路径长度
for( v = 0;v < G. VerticesNum( ) ;v ++ )
    for( i = 0;i < G. VerticesNum( ) ;i ++ )
        for( j = 0;j < G. VerticesNum( ) ;j ++ )
            if( D [ i] [ j]. length > ( D[ i] [ v]. length + D[ v] [ j]. length) ) {
                D[ i] [ j]. length = D[ i] [ v]. length + D[ v] [ j]. length;
                D[ i] [ j]. pre = D[ v] [ j]. pre;
            }
}
```

【算法 7.9 结束】

7.6 最小生成树

回到本章开始提到的例子。假设要在 n 个城市之间建立通信网络,要在最节省经费的情况下建立这个通信网络,则联通 n 个城市只需要 $n-1$ 条线路。那么,如何在这些可能的线路中选择 $n-1$ 条使得总花费最小就是本节将要介绍的最小生成树问题。

图 G 的生成树是一棵包含 G 的所有顶点的树,树上所有权值总和表示代价,在 G 的所有生成树中,代价最小的生成树称为图 G 的最小生成树(minimum-cost spanning tree,MST)。

构造最小生成树有多种算法。本节中要介绍的构造最小生成树的 Prim 算法和 Kruskal 算法

都是贪心算法，都利用了下面的最小生成树的性质。

设 $G = <V,E>$ 是一个连通的带权图，其中每条边 (v_i,v_j) 上带有权 $W(v_i,v_j)$。集合 U 是顶点集 V 的一个非空真子集。构建生成树时需要一条边连通顶点集合 U 和 $V-U$。如果 $(u,v) \in E$，其中 $u \in U, v \in V-U$，且边 (u,v) 是符合条件的权值 $W(u,v)$ 最小的，那么一定存在一棵包含边 (u,v) 的最小生成树。这条性质也称为 MST 性质。

MST 性质可以用反证法证明如下：假设图 G 的任何一棵最小生成树都不包含边 (u,v)，显然当把边 (u,v) 加入到 G 的一棵最小生成树 T 中时，由生成树的定义，将产生一个含有边 (u,v) 的回路。由于 T 是生成树，T 中必存在不同于边 (u,v) 的另一条边 (u',v')，使得 $u' \in U, v' \in V-U$，且 u 和 u' 之间，v 和 v' 之间均有路径相连通，如图 7.22 所示。将边 (u',v') 删除，这样就消除了上述回路，并得到了另一棵生成树 T'。由于 $W(u,v) \leqslant W(u',v')$，因此生成树 T' 的耗费不大于树 T 的耗费。于是 T' 是一棵包含边 (u,v) 的最小生成树，这与假设矛盾。

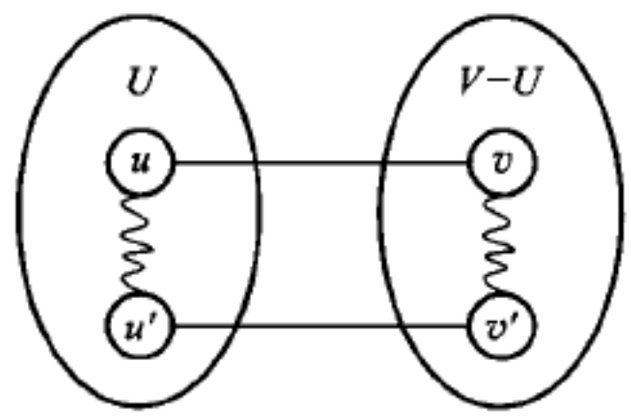


图 7.22 含 (u,v) 的回路

7.6.1 Prim 算法

设 $G = <V,E>$ 是一个连通的带权图，其中 V 是顶点的集合，E 是边的集合，TE 为最小生成树的边的集合。则 Prim 算法通过以下步骤得到最小生成树：

(1) 初始状态：$U = \{u_0\}$，$TE = \{\}$。其中，u_0 是顶点集合 V 中的某一个顶点。

(2) 在所有 $u \in U, v \in V-U$ 的边 $(u,v) \in E$ 中找一条权值最小的边 (u_0,v_0)，将这条边加进集合 TE 中，同时将此边的另一顶点 v_0 并入 U。这一步骤的作用是在边集 E 里找一条两个顶点分别在集合 U 和 $V-U$ 中且权值最小的边，把这条边添加到边集 TE 中，并把这条边上不在 U 中的那个顶点加入到 U 中。

(3) 如果 $U = V$，则算法结束；否则重复步骤(2)。

算法结束时，TE 中包含了 G 中的 $n-1$ 条边。经过上述步骤选取到的所有边恰好就构成了图 G 的一棵最小生成树。

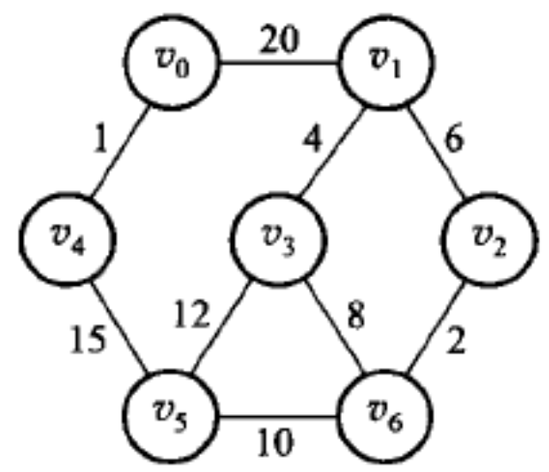
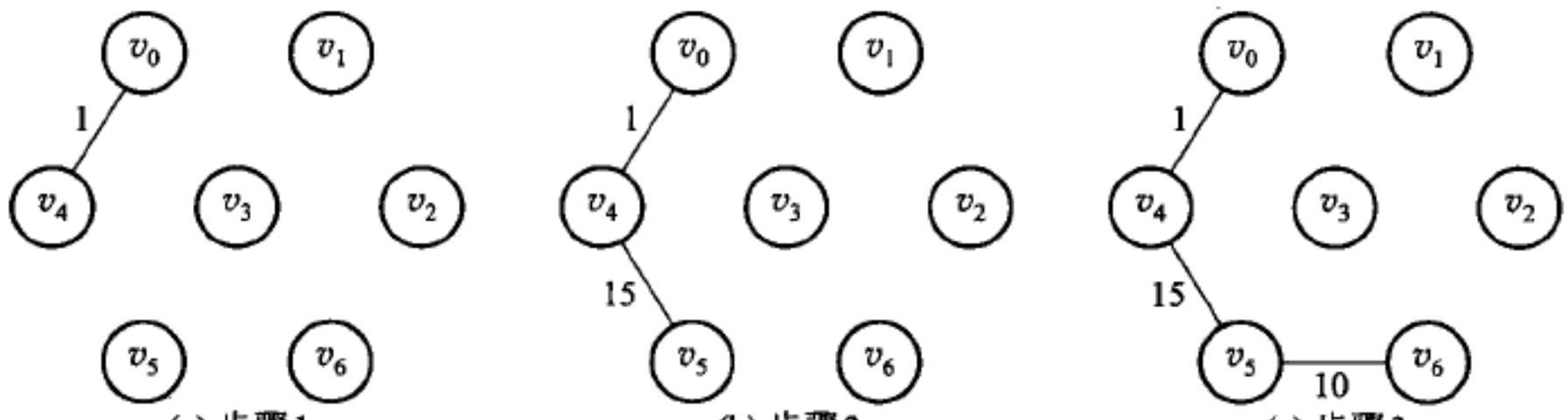
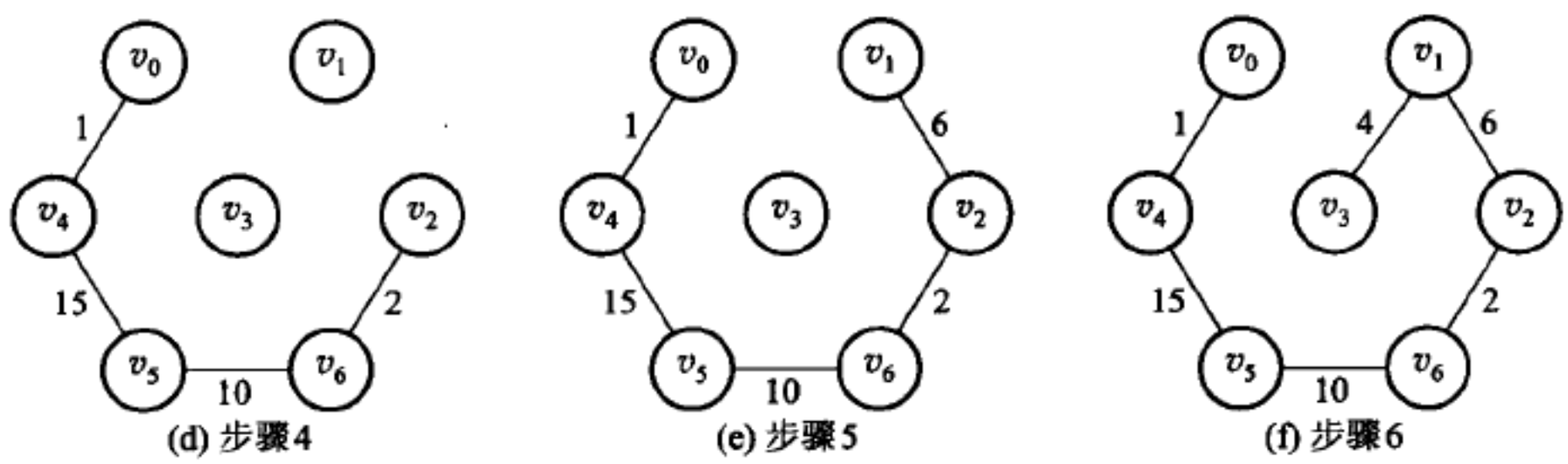


图 7.23 带权图

例如，对于图 7.23 所示的带权图，按照 Prim 算法选取边的过程如图 7.24 所示。



(a) 步骤1 (b) 步骤2 (c) 步骤3



(d) 步骤4 (e) 步骤5 (f) 步骤6

图 7.24 Prim 算法构造图 7.23 中带权图的最小生成树的步骤

Prim 算法的实现如算法 7.10 所示。

【算法 7.10】 Prim 算法。

```
void Prim(Graph& G, int s, Edge * &MST){   // s 是开始顶点,数组 MST 用于保存最小生成树的边
    int MSTtag = 0;                                  // 最小生成树的边计数
    MST = new Edge[G.VerticesNum() - 1];             // 为数组 MST 申请空间
    Dist *D;
    D = new Dist[G. VerticesNum()];                  // 为数组 D 申请空间
    for(int i = 0; i < G.VerticesNum(); i++){        // 初始化 Mark 数组、D 数组
        G.Mark[i] = UNVISITED;
        D[i].index = i;
        D[i].length = INFINITE;
        D[i].pre = s;
    }
    D[s].length = 0;
    G.Mark[s] = VISITED;                     // 开始顶点标记设置为 VISITED
    int v = s;
    for(i = 0; i < G.VerticesNum() - 1; i++){
    if(D[v] == INFINITY) return;             // 非连通,有不可达顶点
    // 因为 v 的加入,需要刷新与 v 相邻接的顶点的 D 值
    for(Edge e = G.FirstEdge(v); G.IsEdge(e); e = G.NextEdge(e))
        if(G. Mark[G.ToVertex(e)] != VISITED &&(D[G.ToVertex(e)].length > e.weight)){
            D[G.ToVertex(e)].length = e.weight;
            D[G.ToVertex(e)].pre = v;
        }
    v = minVertex(G, D);                             // 在 D 数组中找最小值记为 v
    G.Mark[v] = VISITED;                             // 标记访问过
    Edge edge(D[v].pre, D[v].index, D[v].length);    // 保存边
    AddEdgetoMST(edge, MST, MSTtag++);               // 将边 edge 加到 MST 中
}
```

```
}
int minVertex(Graph& G, Dist * & D) {                    // 在 Dist 数组中找最小值
    int i, v;
    for(i=0; i < G.VerticesNum(); i++)
        if(G.Mark[i] == UNVISITED) {
            v = i;                                       // 让 v 为随意一个未访问的顶点
            break;
        }
    for(i=0; i < G.VerticesNum(); i++)
        if((G.Mark[i] == UNVISITED)&&(D[i] < D[v]))
            v = i;                                       // 保存当前发现的具有最小距离的顶点
            return v;
    }
```

【算法 7.10 结束】

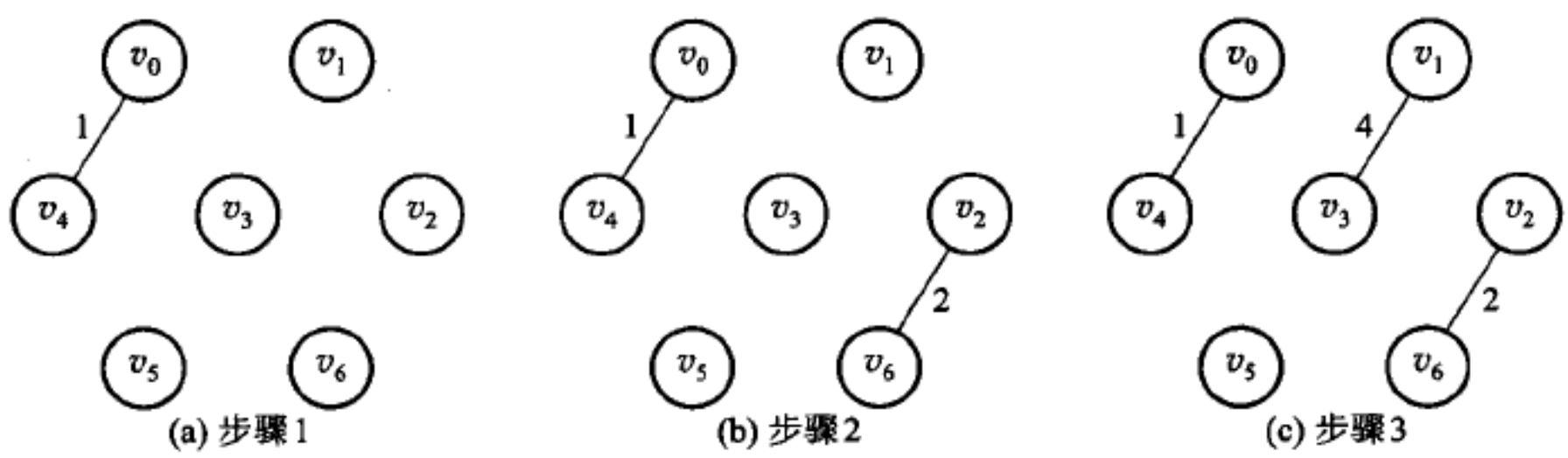
可以看出，Prim 算法非常类似于 Dijkstra 算法，但 Prim 算法中的距离值不需要累积，直接采用离集合最近的边距。Prim 算法的时间复杂度也与 Dijkstra 算法相同。本算法通过直接比较 D 数组元素，确定代价最小的边就需要总时间 $O(n^2)$；取出权最小的顶点后，修改 D 数组共需要时间 $O(e)$，因此共需要花费 $O(n^2)$ 的时间，算法 7.10 适合于稠密图。

7.6.2 Kruskal 算法

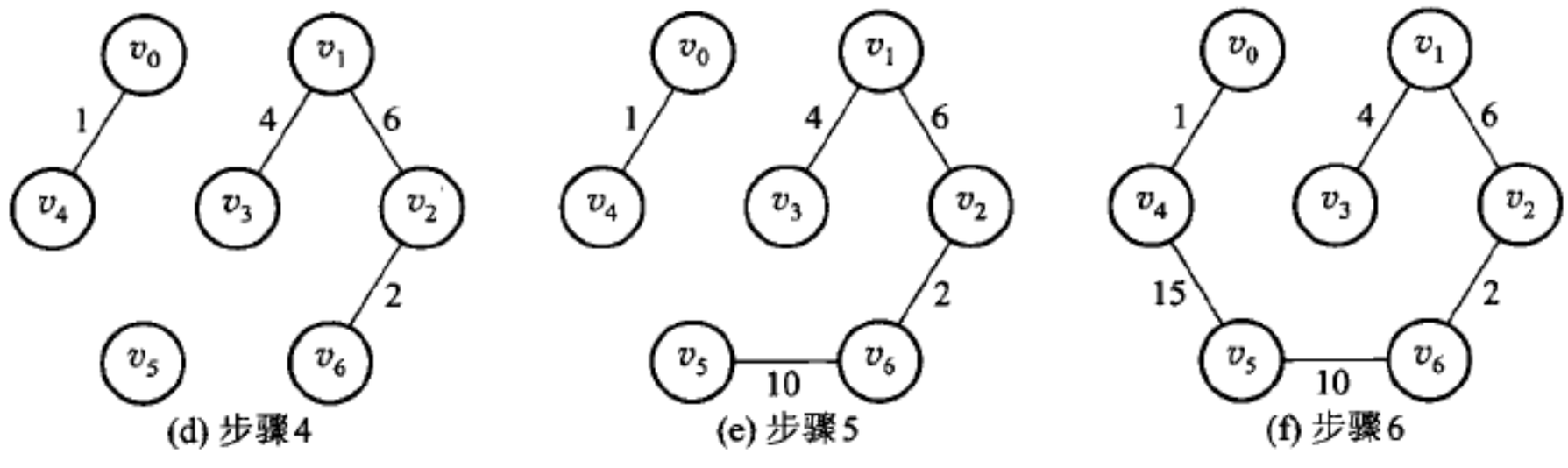
构造最小生成树的另一个常用算法是 Kruskal 算法。Kruskal 算法使用的贪心准则是从剩下的边中选择不会产生环路且具有最小权值的边加入到生成树的边集中。

给定含有 n 个顶点和 e 条边的无向连通带权图 $G = <V,E>$，Kruskal 算法的构造思想是：首先将 G 中的 n 个顶点看成是独立的 n 个连通分量，这时的状态是有 n 个顶点而无边的森林，可以记为 $T = <V,\{\}>$。然后，在 E 中选择代价最小的边，如果该边依附于两个不同的连通分支，那么将这条边加入到 T 中，否则舍去这条边而选择下一条代价最小的边。依次类推，直到 T 中所有顶点都在同一个连通分量中为止，此时就得到图 G 的一棵最小生成树。

例如，对图 7.23 所示的带权图，按照 Kruskal 算法构造最小生成树的过程如图 7.25 所示。



(a) 步骤1　(b) 步骤2　(c) 步骤3



(d) 步骤4 (e) 步骤5 (f) 步骤6

图 7.25 Kruskal 算法构造图 7.23 中带权图的最小生成树的步骤

在 Kruskal 算法中,把连通分量中的顶点作为集合元素,采用第 6 章的并查集的 Find 算法确定边的两个关联顶点所属的连通分支,采用 Union 算法合并两个连通分量。

在 Kruskal 算法中,需要按边权值递增的顺序依次查看边,可以把按边的权值组织优先队列,权值越小,优先级就越高。用最小堆来实现这个优先队列。

算法 7.11 给出了 Kruskal 算法的代码实现。

【算法 7.11】 Kruskal 算法。

```
void Kruskal(Graph& G, Edge * &MST){                    // 数组 MST 用于保存最小生成树的边
    ParTree < int > A(G. VerticesNum());                // 等价类
    MinHeap < Edge > H(G. EdgesNum());                  // 最小堆
    MST = new Edge[G. VerticesNum() - 1];               // 为数组 MST 申请空间
    int MSTtag = 0;                                     // 最小生成树的边计数
    bool heapEmpty;
    for(int i = 0; i < G. VerticesNum(); i++)           // 将图的所有边插入最小堆 H 中
        for(Edge e = G. FirstEdge(i); G. IsEdge(e); e = G. NextEdge(e))
            if(G. FromVertex(e) < G. ToVertex(e))       // 对于无向图,防止重复插入边
                H. Insert(e);
    int EquNum = G. VerticesNum();                      // 开始 n 个顶点分别作为一个等价类
    while (EquNum > 1){                                 // 当等价类的个数大于 1 时合并等价类
        heapEmpty = H. isEmpty();
        if(! heapEmpty)
            Edge e = H. RemoveMin();                    // 获得一条权最小的边
        if(heapEmpty || e. weight == INFINITY){
            cout << "不存在最小生成树." << endl;
            delete[] MST;                               // 释放空间
            MST = NULL;                                 // MST 赋为空
            return;
        }
        int from = G. FromVertex(e);                    // 记录该条边的信息
        int to = G. ToVertex(e);
        if(A. Different(from, to)){                     // 边 e 的两个顶点不在一个等价类
            A. Union(from, to);                 // 将边 e 的两个顶点所在的等价类合并为一个
```

```
                AddEdgetoMST(e,MST,MSTtag++);        // 将边 e 加到 MST
                EquNum--;                            // 等价类的个数减 1
            }
        }
    }
```

【算法 7.11 结束】

Kruskal 算法的时间复杂度为 $O(e\log e)$。这个算法的时间复杂度主要取决于边数，因此 Kruskal 算法适合于构造稀疏图的最小生成树。

本 章 小 结

本章介绍了图的基本概念以及图的重要运算，重点是图的存储结构、图的周游、拓扑排序、最短路径、最小生成树等相关算法。

首先，介绍了图的相关定义和抽象数据类型。图可以分为无向图和有向图，这两种图中含有一些对应的概念，例如边和弧、度和出度/入度、连通分量和强连通分量等。在图的定义中还给出了子图、带权图等概念。

图有 3 种主要存储结构：相邻矩阵表示法、邻接表表示法和十字链表表示法。相邻矩阵用数组表示顶点间相邻关系的矩阵。图的邻接表由顶点表和边表组成，把依附于同一个顶点的边（或弧）存放在同一个边表中。十字链表是有向图的另一种链式存储结构，可以看做有向图的邻接表和逆邻接表的结合。相邻矩阵表示法与邻接表表示法有以下差异：

（1）相邻矩阵适合存储稠密图，邻接表适合存储稀疏图。

（2）求无向图顶点的度时采用这两种存储结构都比较容易，求有向图顶点的出度时采用相邻矩阵更方便。

（3）判断是否是图中的边，相邻矩阵更容易。

（4）求边数 e，相邻矩阵耗时为 $O(n^2)$，与 e 无关；邻接表的耗时为 $O(n+e)$。

图的周游是指从图中的某一个顶点出发，按照一定的策略访问图中的每一个顶点，使得每一个顶点被访问且只被访问一次。图的深度优先搜索（DFS）类似于树的先根次序周游，采用的搜索方法的特点是尽可能先对纵深方向进行搜索。广度优先搜索（BFS）的基本思想是“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问。

对一个有向无环图进行拓扑排序，是将图中所有顶点排成一个线性序列，使得弧尾在弧头之前出现。算法的实现思想是：反复选择入度为 0 的顶点输出并删除该顶点涉及的所有边。

单源最短路径和所有顶点对之间的最短路径是两类在带权图上寻找最短路径的问题。解决单源最短路径问题的常用算法是 Dijkstra 算法，求所有顶点对之间的最短路径使用 Floyd 算法，这两种算法都是动态规划算法。其中，Floyd 算法是比较典型的动态规划，而 Dijkstra 算法是贪心算法。

构造最小生成树的算法有 Prim 算法和 Kruskal 算法。Prim 算法与 Dijkstra 算法类似，具体操

作是从图中的一个顶点开始，把这个顶点包括在 MST 中，然后反复寻找一个顶点已在 MST 中而另一个顶点还不在 MST 中的最小权边，把新边和新结点加入到生成树中。Prim 算法适合于稠密图。构造最小生成树的另一个常用算法是 Kruskal 算法，其构造思想是将 G 中的 n 个顶点看成是独立的 n 个连通分量，然后在边集中选择代价最小且依附于不同连通分支的边加入到生成树中。Kruskal 算法较适合于稀疏图。

习 题

1. 对于图 7.26 所示的带权有向图：

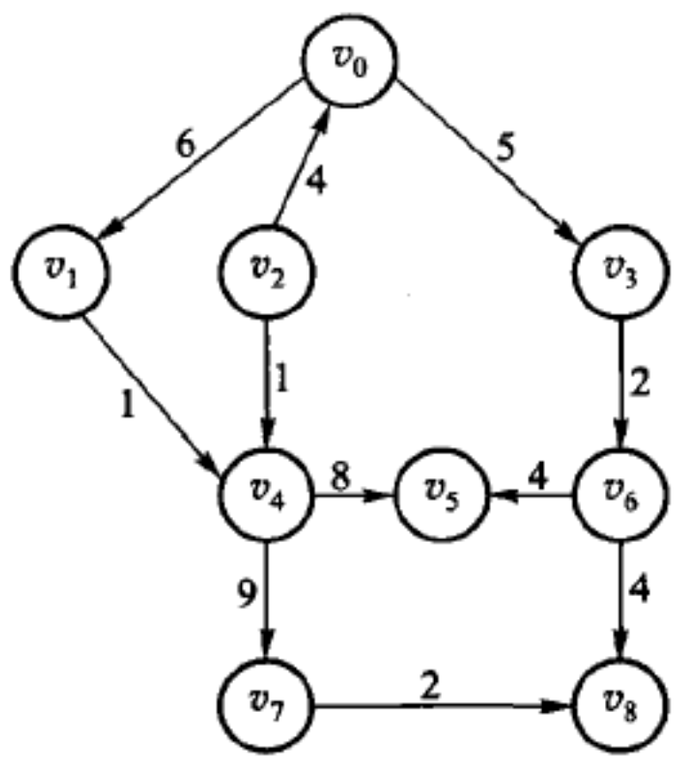


图 7.26 习题 1 的图例

(1) 写出其相邻矩阵。

(2) 画出其邻接表表示。

(3) 计算每个顶点的入度和出度。

(4) 如果每个指针需要 4 个字节，每个顶点的标号需要 2 个字节，每条边的权需要 2 个字节，则此图采用哪种表示法所需要的空间较少？

2. 对于图 7.27 所示的有向图，从顶点 v_1 出发分别画出其深度优先搜索(DFS)和广度优先搜索(BFS)的生成森林。

3. 求图 7.28 所示的有向图中从顶点 v_4 到其他各顶点的全部最短路径及长度。

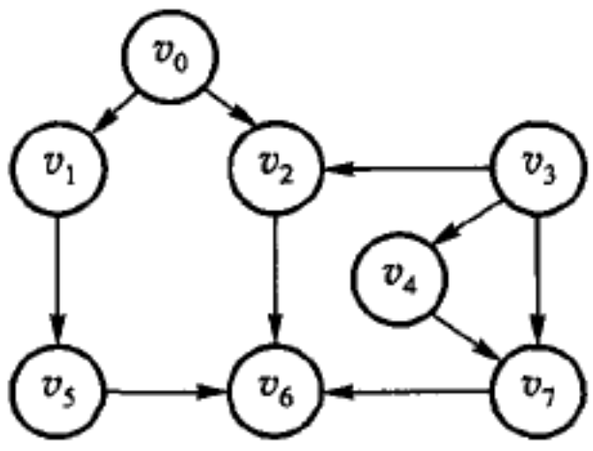


图 7.27 习题 2 的图例

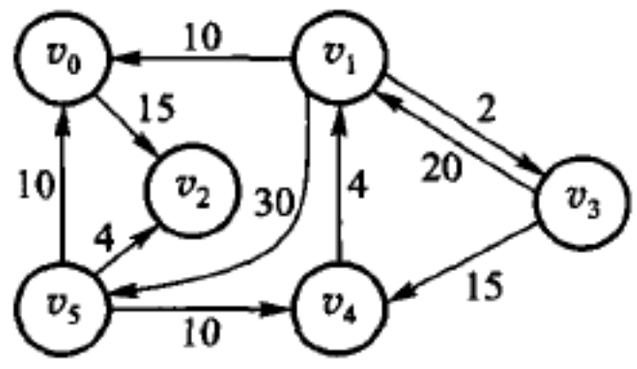


图 7.28 习题 3 的图例

4. 拓扑排序的结果不是唯一的，对于图 7.29 所示有向图的顶点进行拓扑排序，能够得到多少个不同的拓扑序列？试输出得到的所有拓扑序列。

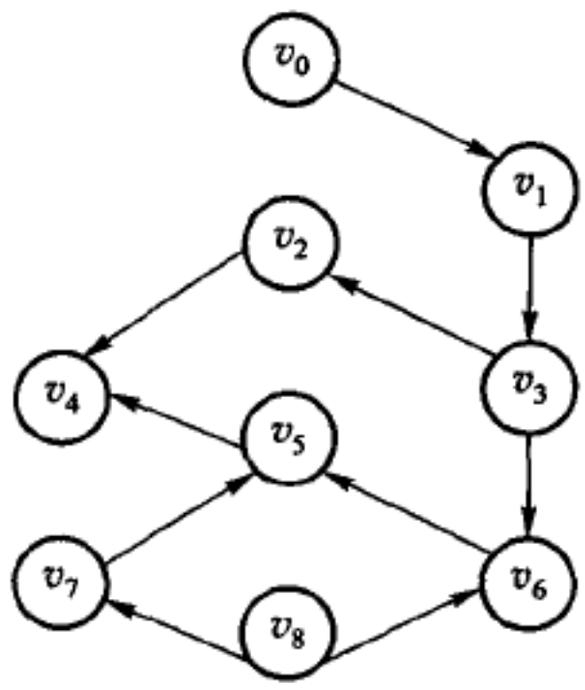


图 7.29　习题 4 的图例

5. 证明：只要适当排列顶点的次序，就能够使有向无环图的相邻矩阵中对角线以下元素全为 0。

6. 证明：

（1）一个没有简单回路的连通无向图 $G = <V,E>$ 有 $n-1$ 条边。

（2）一个有 $n-1$ 条边的无环图一定是连通的。

7. 有向图 $G = <V,E>$ 的转置是图 $G^T = <V,E^T>$，其中边 $<u,v> \in E^T$，当且仅当 $<v,u> \in E$，即 G^T 就是逆转 G 中所有的方向而得到的图。试按照相邻矩阵和邻接表两种表示法写出从 G 计算 G^T 的有效算法，并确定算法的时间复杂度。

8. 有向图 $G = <V,E>$ 的平方图是图 $G^2 = <V,E^2>$，其中边 $<u,v> \in E^2$，当且仅当存在一个顶点 $x \in V$，使得 $<u,x> \in E$ 且 $<x,v> \in E$，即当图 G 中存在一条从顶点 u 到顶点 v 的长度为 2 的路径时，$<u,v> \in E^2$。试按照相邻矩阵和邻接表两种表示法写出从 G 产生 G^2 的有效算法，并确定算法的时间复杂度。

9. 写出一个算法确定一个有 n 个顶点 e 条边的图（有向图或者无向图）是否包含回路，所设计的算法的时间复杂度应该是 $O(n+e)$。

10. 说明并验证你所认为的每对顶点间最短路径问题的最大可能下限。

11. 证明：如果图 G 所有边的权均不相等时，它只存在一棵最小生成树（MST）。

12. Dijkstra 最短路径算法是否给出一棵生成树？是否给出一棵最小生成树（MST）？并证明你的结论。

13. 设计算法找图（有向图或无向图）的所有连通分量（对于有向图则是强连通分量）。提示：第一个连通分量的所有顶点使用第一分量的标记，第二个连通分量的所有顶点使用第二分量的标记，依次类推。

14. 证明：对于一个无向图 $G = <V, E>$，若 G 中各顶点的度均大于或等于 2，则 G 中必有回路。

15. 设有一个含 n 个顶点的有向无环连通图 G，试问 G 有多少条边？

16. 在有向图，源是一个入度为 0 的顶点，试证明每个有向无环图（DAG）至少有一个源。

17. 什么样的有向无环图（DAG）具有唯一的拓扑排序？

18. 采用相邻矩阵表示一个有向图 G，写出一个算法确定 G 是否含有一个漏（即入度为 $n-1$，出度为 0 的顶点），要求该算法的时间复杂度是 $O(n^2)$。

19. 对于一个具有 n 个顶点和 e 条边的有向图 $G = <V, E>$，证明：求其强连通分量的算法所需的时间复杂度是 $O(n+e)$。

20. 设计一个算法，在图（有向图或者无向图）的邻接表表示的基础上实现边的插入和删除。

21. 只要图中不存在权值为负数的边，Dijkstra 算法就可以使用。如果一个图存在权值为负数的边，Dijkstra 算法是否继续可用？如果你认为 Dijkstra 算法不可用，是否有办法改进 Dijkstra 算法使其可用？试证明你的结论，如果你改进了 Dijkstra 算法，也请证明所做改进的正确性。

22. 设一个带权有向图 $G = <V, E>$，v 是 G 的一个顶点，v 的偏心距定义为：max{从 u 到 v 的最短路径的长度}，其中 $u \in V$。所谓最短路径的长度，不是指边数，而是指路径上的边所带的权的总和。

将 G 中偏心距最小的顶点称为 G 的中心，试设计一个算法求带权有向图的中心，并确定算法的时间复杂度。

23. 每棵树是一个有向无环图（DAG），但并不是所有 DAG 就是树，试设计一个程序以判断 DAG 是否是树，并确定程序所采用算法的时间复杂度。

24. 设计算法找有向图的根。以一个有向图作为输入，如果它有根，则输出它的所有的根；如果没有，则输出空。试分析算法的时间复杂度。

25. 设计算法找有向无环图（DAG）每对顶点间的“最长简单路径”（所谓“最长简单路径”，是指该简单路径包含的边数最多）。以一个有向无环图作为输入，对于每对顶点，如果它们之间存在简单路径，则输出其中路径长度最长（边数最多）的简单路径，否则输出空。试分析算法的时间复杂度。

26. 设计一个程序，输入一个图（有向图或者是无向图）$G = <V, E>$ 以及一对顶点 v_i、$v_j \in V$，输出的结果是：如果从 v_i 到 v_j 存在一条简单路径，则输出从 v_i 到 v_j 的所有简单路径；如果不存在，则输出空。

上 机 题

1. 套汇是指利用汇率差异将一个单位的货币转换为大于一个单位的同种货币。例如，假设 1 美元兑换 7.51 人民币，1 元人民币兑换 0.07 英镑，1 英镑兑换 2.03 美元，那么如果一个人拿 1 美元先兑换成人民币，再把人民币兑换成英镑，最后把英镑兑换成美元，则他最后能够得到 $1 \times 7.51 \times 0.07 \times 2.03 = 1.07$ 美元，从而获得 $1.07 - 1 = 0.07$ 美元的利润，这就是套汇。假设有 n 种货币 $v_1, v_2, \cdots, v_n$ 和有关汇率的 $n \times n$ 矩阵，其中 $A[i,j]$ 是一单位货币 v_i 兑换成货币 v_j 的单位数，要求设计一个程序判断是否存在一个货币序列 $v_{i1}, v_{i2}, \cdots, v_{ik}$ 使得 $A[i1, i2] \times A[i2, i3] \times \cdots \times A[ik, i1] > 1$，如果存在，则输出所有这样的货币序列；如果不存在，则输出空，并确定算法的时间复杂度。

2. 有向无环图在工程计划和管理系统中具有广泛的应用，是描述一项工程或系统进行过程的有效工具。如果用一个带权的有向无环图表示这种结构，其中顶点表示事件，有向边表示活动，弧上的权值表示活动进行的时间。这样的有向图称做 AOE（activity on edge）网络。

利用 AOE 网络可以对工程进行估算，分析完成整个工程至少需要的时间以及哪些活动是影响工程进度的关键等问题。

完成工程的最短时间是从开始点到结束点的最长路径长度，具有最长路径长度的路径称为关键路径（critical path），关键路径上的活动称为关键活动。如果不按期完成关键活动就会影响整个工程的完成时间的活动，提前完成那些不在关键路径上的活动也不能加快整个工程的进度。因此，问题的关键就是分析哪些活动是关键活动，以提高关键活动的效率，从而减少完成整个工程的期限。

用 $e(i)$ 表示活动 a_i 的最早发生时间，即从开始结点 v_1 到结点 v_i 的最长路径长度；用 $l(i)$ 表示活动 a_i 的最迟发生时间，即在保证结点 v_n 在 $e(n)$ 时刻发生的前提下，活动 a_i 允许发生的最迟时间。可见，$\Delta t = l(i) - e(i)$ 表示完成事件 v_i 的时间余量。所谓关键活动就是那些 $l(i) = e(i)$ 的活动。

为了求 AOE 网络中活动的 $l(i)$ 和 $e(i)$，定义 $ee(i)$ 为事件 v_i 的最早发生时间以及 $le(i)$ 为事件 v_i 允许发生的最晚时间。如果 AOE 网络中的活动 a_i 由弧 $<j,k>$ 给出，其时间用权值 $W(<v_j,v_k>)$ 表示，则有下列关系

$e(i)=ee(j)$

$l(i)=le(k)-W(<j,k>)$

计算事件 v_i 的最早发生时间 $ee(i)$ 和最迟开始时间 $le(i)$ 的过程如下：

(1) 令 $ee(0)=0$，然后向前推进求 $ee(j)$：

$ee(j)=\max\{ee(i)+W(<v_i,v_j>)\}$，其中 i 包括所有使 $<v_i,v_j>$ 是边的 i，$1\leqslant j\leqslant n$。

(2) 令 $le(n)=ee(n)$，然后向后退求 $le(i)$：

$le(i)=\min\{le(j)-W(<v_i,v_j>)\}$，其中 j 包括所有使 $<v_i,v_j>$ 是边的 j，$0\leqslant i\leqslant n-2$。

在上述两个步骤中，第(1)步求 $ee(j)$ 必须在结点 v_j 的所有前驱结点的最早发生时间都已求得的前提下进行；第(2)步求 $le(i)$ 必须在结点 v_i 的所有后继结点的最晚发生时间都已求得的前提下进行，因此结点序列 v_1，v_2，…，v_n 必须是一个拓扑序列。

对于每个活动 $a_k=<v_i,v_j>$，$e(k)=ee(i)$，$l(k)=le(j)$。如果 $l(k)-e(k)$ 等于边 a_k 的权，那么 a_k 是关键活动。

了解了上面的知识后，试设计一个以文件方式存储事件结点网络(例如图 7.30 所示)的格式，从而实现从一个文件中读入一个事件结点网络，并求出各活动可能的最早开始时间和允许的最晚完成时间、整个工程的最短完成时间、哪些活动是关键活动、提高哪些活动的速度能够使整个工程提前完成等，并将求出的结果写入到另一个文件。

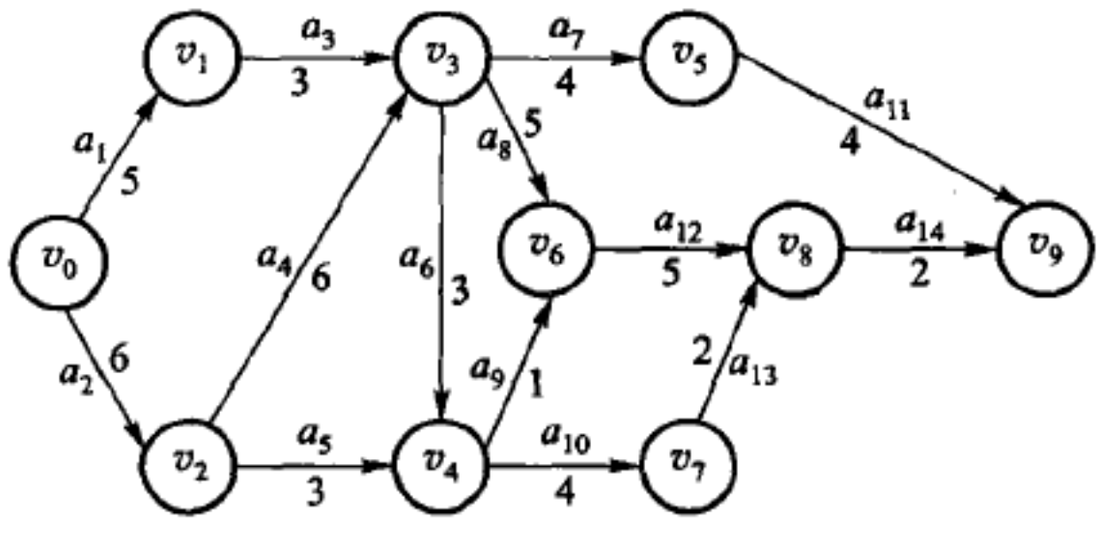


图 7.30 上机题 2 的图例

第8章 内 排 序

在日常生活中，经常需要对收集到的各种数据信息进行处理，这些数据处理中经常用到的核心运算就是排序(sorting)。例如，图书管理员将书籍按照编号排序放置在书架上，方便读者查找；打开计算机的资源管理器，可以选择按名称、大小、类型等来排列图标；搜索引擎把与检索词最相关的页面排在前面返回给用户。由于排序运算的广泛性和重要性，人们在长期的实践中不断开发出各种各样的排序算法，并被广泛应用在很多领域。

排序主要分为两类：内排序和外排序。如果待排序的记录个数较少，整个排序过程中所有的记录都可以直接存放在内存中，这样的排序叫做内排序(internal sorting)。如果待排序记录数量太大，内存无法容纳所有记录，在排序过程中还需要访问外存，这样的排序叫做外排序(external sorting)，将在第9章详细介绍。

本章主要介绍几种比较经典而又常用的内排序算法，包括算法思想、算法源码以及详细的代价分析和试验比较，以便于读者将来在具体应用中根据各算法特性而选取适合的排序方法。本章还将证明基于比较的排序问题，时间代价为 $\Theta(n \log n)$。

8.1 排序问题的基本概念

由于讨论的是内排序，因此本章在大部分情况下都是讨论基于顺序存储的排序，即输入数据存储在数组中。讨论排序时，习惯上用术语“记录”(record)或“元素”(element)来代替本书前7章中的术语“结点”(node)。记录是进行排序的基本单位，把所有这些待排记录称为“序列”(sequence)，而不用术语“线性表”。所谓排序就是将序列中的记录(即线性表中的结点)按照特定的顺序排列起来。

每一个记录内都有一个排序码(sort key)域作为排序运算的依据。为了简化讨论，本章假设排序码为整数，其实也可以是字符串、组合类型或记录中多个域的组合。注意，排序码不一定是关键码(key)，关键码是唯一确定记录的一个或多个域。如果排序码不是关键码，就可能有多个记录具有同一个排序码，因此排序结果可能不唯一。为了简便，很多教材一般用关键码作为排序码，也有的教材统一称为“关键码”。

考虑到C++的习惯，本章讨论的大部分序列的下标都从0开始。给定一个序列 $R=\{r_0, r_1, \cdots, r_{n-1}\}$，其排序码分别为 $k=\{k_0, k_1, \cdots, k_{n-1}\}$，排序后形成一个新的有序序列 $R'=\{r'_0,$

$r'_1,\cdots,r'_{n-1}\}$，对应于排序码 $k'=\{k'_0,k'_1,\cdots,k'_{n-1}\}$，其中 $k'_0\leqslant k'_1\leqslant\cdots\leqslant k'_{n-1}$ 或者 $k'_0\geqslant k'_1\geqslant\cdots\geqslant k'_{n-1}$，前者称为不减序列，后者称为不增序列。如果没有重复排序码，即 $k'_0<k'_1<\cdots<k'_{n-1}$ 或者 $k'_0>k'_1>\cdots>k'_{n-1}$，则前者称为升序，后者称为降序。在要求不太严格的情况下，也称不减序列为升序，不增序列为降序。排序的作用就是将一组记录按照排序码的值不减（升序）或者不增（降序）的顺序重新排列成一个有序序列。本章中的排序算法一般是对记录数组进行不减或升序排序。如果待排序的序列正好符合排序要求，则称“正序”序列；如果把待排序的序列逆转过来，正好符合排序要求，则称“逆序”序列。

如果存在多个具有相同排序码的记录，经过排序后这些记录的相对次序仍然保持不变，这种排序算法称为“稳定的”（stable），否则称为“不稳定的”。在有些应用领域中，可能要求尽量不要改变具有相同排序码的记录的原始输入顺序，这样就需要采用那些稳定的排序算法。例如，在8.6.2节可以看到分配排序低位优先法 LSD，第一趟分配和收集以后的其他排序步骤，都要求采用稳定排序算法。

排序算法如此之多，那么在特定的环境下该如何选择排序算法？评价一种排序算法的好坏主要通过空间代价和时间代价两方面来衡量，尤其是时间代价。如果有特殊的空间限制，则要注意采用辅助空间较小的算法。

排序算法的时间代价一般通过记录的总比较和总移动次数来衡量，一次移动就是一次记录赋值，一次 swap 交换为三次移动。记录的数量、排序码和记录的大小以及输入记录的原始有序程度都会影响到排序算法的相对运行时间。一般而言，所需时间越短的算法越好。但是，估算排序算法的时间代价时，需要分别考虑3种情况下的代价：最小时间代价（最佳情况）、最大时间代价（最差情况）以及平均时间代价（平均情况）。当序列中不含重复记录时，一个长度为 n 的序列共有 $n!$ 种排列。假定每种排列的出现都是等概率的，序列所有可能排列的平均运行时间就是排序算法的平均时间代价。

根据不同策略，排序算法可以有不同的分类法。例如，把插入排序、直接选择排序和冒泡排序归为简单排序，把快速排序和归并排序归为分治排序。本章按照以下的分类：把直接插入排序、Shell 排序归为插入排序类，把冒泡排序和快速排序归为交换排序，而把直接选择排序、堆排序以及第9章的选择树归并排序划为选择排序，分配排序中介绍桶排序、基数排序以及索引排序。

8.2 插入排序

插入排序（insert sorting）的算法思想十分简单，就是对待排序的记录逐个进行处理，每个新记录与同组那些已排好序的记录进行比较，然后插入到适当的位置。插入排序算法的关键在于如何将一个新记录 r_i 插入到已排序序列 R' 中，这涉及两个方面：找到序列中应插入的位置以及如何移动序列中那些已排好序的元素以便插入新记录。

插入排序的变种很多，例如本章上机题1中的二分插入排序和交换插入排序。本节主要介

绍基本的插入排序以及基于插入排序的 Shell 排序。

8.2.1 直接插入排序

直接插入排序通过线性搜索来确定待插入记录的位置。如果前面已经有若干个记录排成了不减序列，则对已排序记录按照从大到小依次逐个与新记录进行比较，直到找到第一个不大于新记录的值，这就是新记录应该插入的位置；依次把新记录插入到逐步扩大的已排序子序列中，直到最后完全排好序。

对待排序序列 $R=\{r_0,r_1,\cdots,r_{n-1}\}$，插入排序算法 8.1 将按照下述步骤对 R 进行排序，排完序后的有序序列为 $R'=\{r'_0,r'_1,\cdots,r'_{n-1}\}$：

(1) 首先记 R'为只含一个记录 r_0的序列，即 $R'=\{r_0\}$，此时 $r'_0=r_0$。

(2) 将记录 r_1与序列 R'中的 r'_0进行比较，将 r_1插入到序列中适当的位置。例如，如果 $r_1<r'_0$，就将 r_1插到 r'_0前面，此时 $r'_0=r_1,r'_1=r_0,R'=\{r_1,r_0\}$；否则 $r_1\geqslant r'_0$，r_1和 r'_0的相对位置保持不变，此时$r'_0=r_0,r'_1=r_1,R'=\{r_0,r_1\}$。

(3) 插入 r_i时$(1<i<n)$，$\{r'_0,r'_1,\cdots,r'_{i-1}\}$已经有序，将 r_i暂时保存起来，腾出该记录所占的数组元素位置，从右向左依次将 r_i与前面 j 位置的记录 r'_j比较。如果 $r_i<r'_j$，则 r'_j向后移动一位；如果 $r_i\geqslant r'_j$，则找到了正确位置，直接将 r_i放到 $j+1$ 的位置上，$\{r'_0,r'_1,\cdots,r'_j,r_i,r'_{j+1},\cdots,r'_{i-1}\}$。最后，得到一个已经排好序的 R'。

【算法 8.1】 插入排序算法。

```
template <class Record>
void InsertSort(Record Array[], int n){        // 直接插入排序,Array[]为待排序数组,n 为数组长度
    Record TempRecord;                         // 临时变量
    int i, j;
    for(i =1; i < n; i++){                     // 依次插入第 i 个记录
        TempRecord = Array[i];                 // 先把记录 Array[i]保留在临时记录变量中
        j=i-1;                                 // 内存循环变量从 j 从 i-1 开始
        // 往前寻找记录 i 的正确位置
        while (j> =0 && TempRecord < Array[j]){
            Array[j+1] = Array[j];             // 将那些大于记录 i 的记录后移
            j=j- 1;                            // 下标 j 前移
        }
        Array[j+1] = TempRecord;               // 此时 j+1 就是记录 i 的正确位置,回填
    }
}
```

【算法 8.1 结束】

图 8.1 所示为一个插入算法的例子。图中每一行的数据表示插入了第 i 个记录后的序列。

实下画线表示即将插入的第 $i+1$ 个记录，虚下画线表示第 i 个记录插入后的新位置。箭头连线从下一个待插入元素位置指向最终插入位置。

直接插入排序算法是稳定的，因为每次插入只与临近记录逐个比较，直到找到第一个不大于新记录的值停止。也就是说，如果前面有等于新记录排序码值的记录，该记录仍然会在前方位置。可以看出，图 8.1 中 $i=4$ 的那一行，没有改变两个 34(34 和 34′)的原始顺序，直到排序结束二者的相对顺序也没有改变。

下标：	0	1	2	3	4	5	6	7
待排：	45 ‖	34	78	12	34'	32	29	64
i=1	34	45 ‖	78	12	34'	32	29	64
i=2	34	45	78 ‖	12	34'	32	29	64
i=3	12	34	45	78 ‖	34'	32	29	64
i=4	12	34	34'	45	78 ‖	32	29	64
i=5	12	32	34	34'	45	78 ‖	29	64
i=6	12	29	32	34	34'	45	78 ‖	64
i=7	12	29	32	34	34'	45	64	78 ‖

图 8.1 插入排序

现在来分析直接插入排序算法的运行代价。算法用到了一个辅助存放待插入记录的临时变量，因此空间代价为一个记录大小，即 $\Theta(1)$。

时间代价的分析复杂一些，直接插入排序的程序体由两重循环组成。外层循环迭代 $n-1$ 次；内层循环的开始和结束有两个记录的移动操作，迭代次数则是根据实际情况来定的，依赖于在第 i 个记录前的 $i-1$ 个记录中有多少个小于第 i 个记录。

数组正序排列时，每次第 i 个记录一进入内层循环就退出，迭代次数为 0，比较次数为 $n-1$；当前记录保存在临时变量中 $n-1$ 次，回填 $n-1$ 次，移动次数共为 $2(n-1)$。因此，最佳情况下直接插入排序的最小时间代价为 $\Theta(n)$。

相反，最差的情况下，数组本身恰巧是按照降序排列，即正好与需要排列的顺序相反(逆序)。每次在外层的第 i 次迭代中内层都需要进行 i 次循环，比较 i 次；当前记录保存在临时变量中发生 1 次移动，回填时发生 1 次移动，前面序列 Array[0]…Array[$i-1$]顺序向后移动 i 次，总共移动 $i+2$ 次。总比较次数为：

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

总移动次数为：

$$\sum_{i=1}^{n-1}(i+2) = n(n-1)/2 + 2(n-1) = (n-1)(n+4)/2 = \Theta(n^2)$$

因此，最差情况下直接插入排序的最大时间代价为 $\Theta(n^2)$。

下面估计直接插入排序的平均时间代价。考虑序列 $R=\{r_0,r_1,\cdots,r_{n-1}\}$ 的一个排列 $P=\{p_0,p_1,\cdots,p_{n-1}\}$。如果 P 中两个元素 p_i 和 p_j 满足 $p_i>p_j$ 但 $i<j$，那么 (p_i,p_j) 称为一个逆置(inversion)。P 的全逆置序列为 $P'=\{p_{n-1},p_{n-2},\cdots,p_0\}$。当 R 中不含重复元素时，P 共有 $n!$ 种排列。如果将 P 中的元素两两组成对，则共有 $n(n-1)/2$ 个元素对，每个元素对都可能构成 P 或者 P' 的一个逆置，因此根据每个元素对出现的几率相等的原则，平均有一半的逆置出现在 P 中，即 P 中平均有 $n(n-1)/4$ 个逆置。

当处理第 i 个记录时，内层循环的迭代次数依赖于该记录前面比它大的记录个数，即逆置的数目。例如，在图 8.1 中处理到 $i=3$ 时，记录 12 前面有 3 个比它大的数，存在 3 个逆置。逆置的数目决定了比较及移动的次数，因此计算平均时间代价也就是要计算整个数组中平均有多少个逆置。按照上面的推理，算法 8.1 的平均时间代价为 $n(n-1)/4$，即 $\Theta(n^2)$。

8.2.2 Shell 排序

8.2.1 节已经介绍过，在初始序列为正序时，插入排序算法的最佳时间代价为 $\Theta(n)$。另外，对于短序列的情况，插入排序也比较有效。Shell 排序(Shell sorting)有效地利用了插入排序的这两个性质。

Shell 排序是以他的发明者 D. L. Shell 的名字来命名的。与直接插入排序不同的是，Shell 排序不是着眼于相邻记录之间的比较，而是对那些不相邻的记录进行比较和移动。Shell 排序通过分组来进行排序：先将待排序序列分为若干个子序列，而且要保证子序列中的记录在原始数组中不相邻，且间距相同，分别对这些小子序列进行插入排序；然后，减少记录间的间距，减少小序列个数，将原始序列分为更大、更有序的子序列，分别进行插入排序；重复进行下去，直到最后间距减少为 1(整个序列比较接近于正序状态)，然后对整个序列进行插入排序。

由于 Shell 排序按照不断缩小的增量来将原始序列分成若干个子序列，因此有时也称为缩小增量排序法(diminishing increment sorting)。为了方便讨论，采取“增量每次除以 2 递减”的方法来选择增量，即增量序列为 $(2^k, 2^{k-1},\cdots,2,1)$，并且假设数组长度 n 为 2 的整数幂次方。例如，如图 8.2 所示，当 $n=8$ 时，具体实现步骤如下：

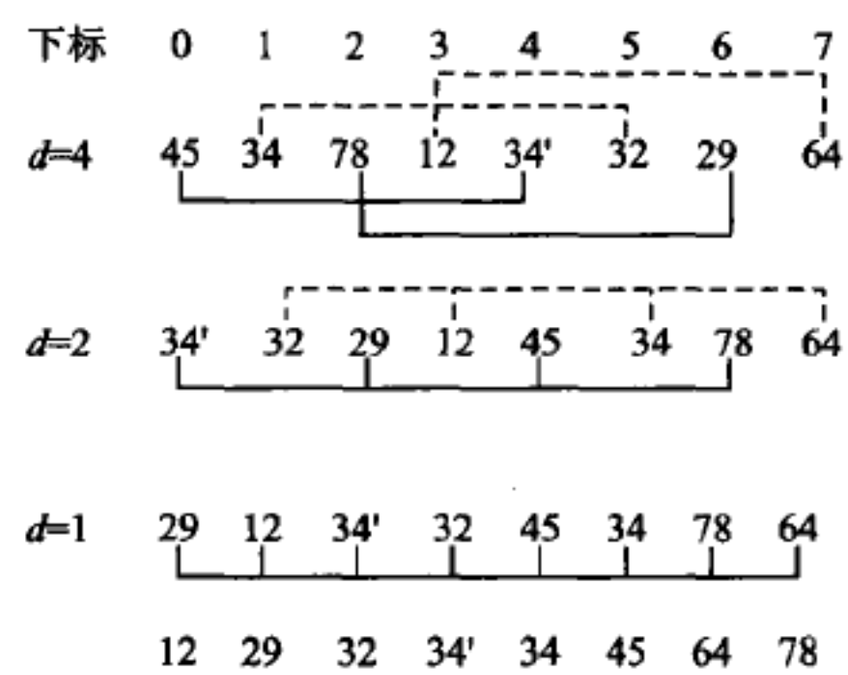


图 8.2 Shell 排序

(1) 首先，将图 8.2 中第一行所示的原始序列分为 $n/2$ 即 4 个子序列，每个子序列包含 2 个记录，记录下标间距

$d=4$。第一个子序列中记录下标为0和4,即(45,34′),第二个子序列为1和5,即(34,32),第三个为2和6,即(78,29),第四个为3和7,即(12 ,64)。分别对4个子序列进行插入排序,排序后的结果如图8.2中第二行所示。

(2) 再将序列分为 $n/4$ 即2个子序列,每个子序列包含4个记录,记录下标间距 $d=2$。第一个子序列中记录的下标为0、2、4、6,即(34′,29,45,78),第二个子序列为1、3、5、7,即(32,12,34,64)。此时,两个子序列中都只含1个逆置,已经基本上有序了,然后分别对两个子序列进行插入排序,排序后结果如图8.2中第三行所示。

(3) 最后对整个序列进行插入排序,事实上此时整个序列中只含4个逆置,远小于平均逆置数 $n(n-1)/4=14$,排序后的结果如图8.2中第四行所示。

算法8.2给出了"增量每次除以2递减"Shell排序的具体实现代码。其中,ModInsSort增量插入排序每次比较Array[j]与Array[j-delta],通过交换一步步地把当前待插入记录移动到位。插入算法的处理细节与算法8.1略有不同。&Array[i]表示ShellSort()函数传递给ModInsSort()的是Array中第i位元素的地址(0≤i<delta),即各个子序列的首地址,整个待处理数组长度为n-i,子序列中有(n-i)/delta个元素。读者可以在算法8.1的基础上改写增量插入排序函数ModInsSort(),这样得到的Shell排序算法实验性能稍微快一些。

【算法8.2】 "增量每次除以2递减"的Shell排序。

```
template <class Record>
void ShellSort(Record Array[], int n){          // Shell 排序,Array[]为待排序数组,n 为数组长度
    int i,delta;
    for(delta=n/2; delta>0; delta /= 2)         // 增量 delta 每次除以 2 递减
        for(i =0;i<delta;i++)                   // 分别对 delta 个子序列进行插入排序
            ModInsSort(&Array[i], n-i, delta); // "&"传 Array[i]的地址,待处理数组长度为 n-i
    // 如果增量序列不能保证最后一个 delta 间距为 1,可以安排下面这个扫尾性质的插入排序
    // ModInsSort(Array, n, 1);
}

template <class Record>
void ModInsSort(Record Array[], int n, int delta){// 修改的插入排序算法,参数 delta 表示当前的增量
    int i, j;
    for(i =delta;i<n;i+=delta)                  // 对子序列中第 i 个记录,寻找合适的插入位置
        for(j =i;j>=delta;j-=delta){            // j 以 dealta 为步长向前寻找逆置对进行调整
            if(Array[j] < Array[j-delta])       // Array[j] < Array[j-delta],则二者为逆置对
                swap(Array, j,j-delta);         // 交换 Array[j]和 Array[j-delta]
            else break;
        }
}
```

【算法8.2结束】

算法 8.2“增量每次除以 2 递减”，因此可以保证最后一次的间距为 1，可以对整个序列进行插入排序。如果采用其他增量序列，一定要注意人为地增加一个间距为 1 的增量元素。例如，把算法 8.2 最后的注释行变为有效算法行，安排扫尾插入排序，对整个数组的全体元素进行排序。

Shell 排序子序列互相交错，而且跨度比较大，它是一种不稳定的排序算法。图 8.5 中的 34 和 34'的排序结果就是不稳定的。

由于 Shell 排序中用到了直接插入排序，其中 swap( )函数需要用到一个临时变量，因此 Shell 排序的空间代价为 $\Theta(1)$。注意，这里所采用的插入排序算法是每次发现逆序对就交换；而算法 8.1 是找到待插入的位置，该位置后的元素依次后移，新元素一次入位。算法 8.1 在数据规模很大时比较有优势。在 Shell 排序中要处理很多小子串，算法 8.1 的方法在这里没有明显的优势。为了强调 Shell 的分组思想，算法 8.2 采用了交换逆序对这种简洁的写法。

Shell 排序的效率比直接插入排序要高。事实上，整个排序过程中，第一轮循环中逆置个数比较多，最后几轮循环中子序列都是基本有序的。因此，效率比起直接对整个数组进行插入排序要好。但是，当选取“增量每次除以 2 递减”时，效率仍然为 $\Theta(n^2)$，并没有多大效果。问题就在于这样选取的增量之间并不互质，因此间距为 2^{k-1} 的子序列都是由那些间距为 2^k 的子序列组成的，而实际上上一轮循环中这些子序列都已经排过序，导致后面的处理效率不高。最坏情况下，所有比较大的记录的下标都是奇数，而比较小的记录的下标都是偶数。由于每次分出的子序列要么下标都是奇数，要么下标都是偶数，因此每次对子序列排序后，那些大数仍然在奇数位置上，小数仍然都在偶数位置上，整个序列仍然没有达到基本有序状态，导致最后一次对整个序列进行直接插入排序时效率大幅降低。

Hibbard 提出了一种新的增量序列 $\{2^k-1, 2^{k-1}-1, \cdots, 7, 3, 1\}$，推理证明这种选取增量序列的 Shell 排序的效率可以达到 $\Theta(n^{3/2})$，但模拟实验表明甚至可以达到 $\Theta(n^{5/4})$，只是尚未得到理论上的证明。“增量每次除以 3 递减”的 Shell 排序的效率也是 $\Theta(n^{3/2})$。事实上，选取其他增量序列还可以更进一步减少时间代价，甚至有的增量序列可以达到 $\Theta(n^{7/6})$，很接近 $\Theta(n\log n)$，比直接插入排序都要快得多。

8.3 选择排序

选择排序(selection sorting)的算法思想是逐个找出第 i 小的记录，并将其放到数组的第 i 个位置。选择排序的关键在于如何从剩余的未排序记录中找出最小(或最大)的那个记录。本节将介绍简单的线性查找方法——直接选择排序，基于二叉树堆进行选择的方法——堆排序。

8.3.1 直接选择排序

下面介绍一种直接选择排序，如算法 8.3 所示。其算法思想是逐个找出第 i 小的记录，并将这个记录与数组的第 i 个位置的记录交换，第 i 小的记录一次交换到位。

【算法 8.3】 直接选择排序

```
template <class Record>
void SelectSort(Record Array[], int n){        // 直接选择排序,Array[]为待排序数组,n 为数组长度
    int i, j, Smallest;
    for(i=0; i<n-1; i++){                      // 依次选出第 i 小的记录,即剩余记录中最小的那个
        Smallest=i;                            // 首先假设记录 i 就是最小的
        for(j=i+1;j<n;j++)                     // 开始向后扫描所有剩余记录
            if(Array[j] < Array[Smallest])
                Smallest=j;                    // 如果发现更小的记录,记录它的位置
        swap(Array, i, Smallest);              // 交换 Array[i]和 Array[Smallest],第 i 小的记录到位
    }
}
```

【算法 8.3 结束】

在第 i 次循环中,首先记录当前位置 i,由于 i 之前的记录已经是已排序的,因此只需要从 i 开始查找。由于选择出第 i 小的记录 Array[Smallest]后,该记录将马上与 Array[i]进行交换。可能存在某个 $i<j<$Smallest,且 Array[i] == Array[j];将 Array[i]与 Array[Smallest]交换后,Array[i]就排到 Array[j]之后,Array[i]与 Array[j]就可能失去稳定性(如果将来的交换中,二者相对顺序不变)。因此,算法 8.3 的直接选择排序是不稳定的。

图 8.3 给出了一个选择排序的例子。图中的每行表示第 i 次选择($0\leqslant i<n-1$)后的序列,带箭头的连线连接下一轮准备交换的两个记录。第一轮循环中,首先下标 Smallest=0,从数组第一个记录向后扫描,发现 34 比 45 小,于是 Smallest=1,继续向后扫描,发现 12 又比 29 小,于是

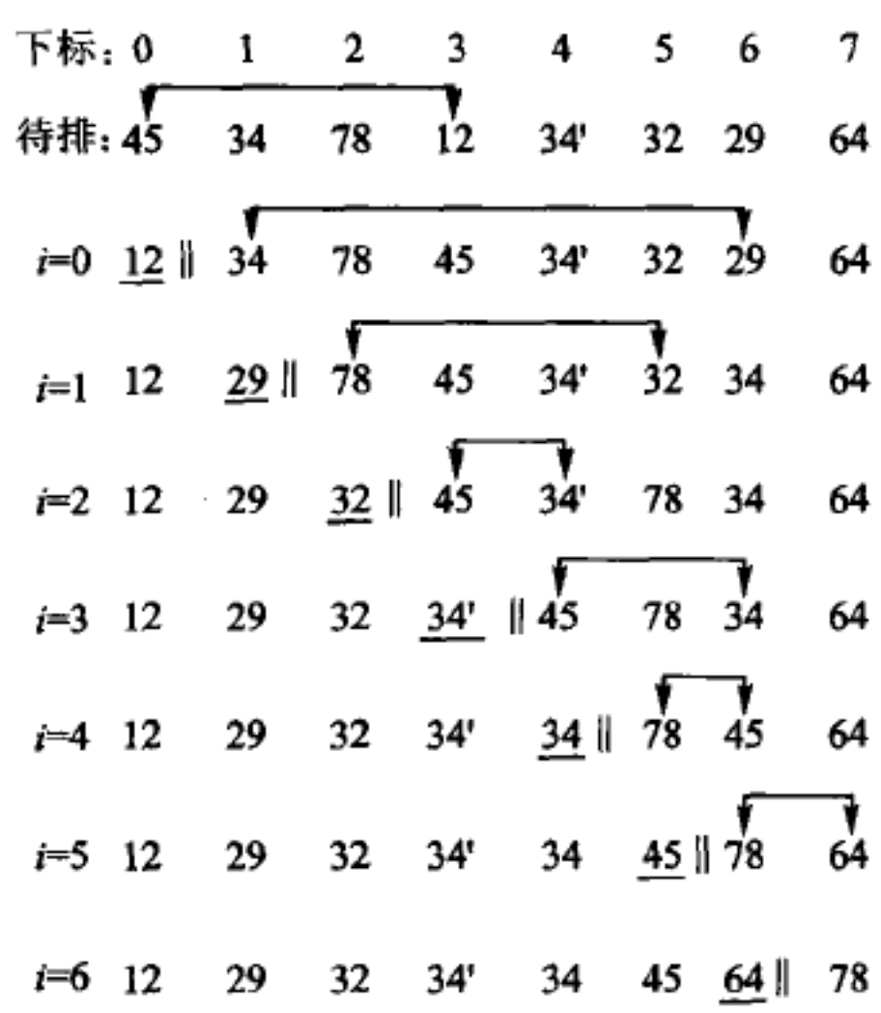


图 8.3 选择排序

Smallest = 3，再向后扫描发现前面已没有比 12 更小的数，于是 12 就是最小记录。将它与数组中第 1 个记录 45 交换，得到如图第二行（$i=0$）所示的序列。类似的，第二轮从剩余的 7 个记录中找到最小的 29，与记录 34 交换，得到第 3 行（$i=1$）序列；依次类推，直到最后未排序序列中只剩下 1 个记录，这时整个数组都已经排好序。

从图 8.3 可以看出，记录交换的跨度比较大。排序前，34 在 34'之前，34 在第二轮就被交换到了数组下标为 6 的位置。34'所处的下标为 4，由于算法是从前向后扫描，在第 4 轮（$i=3$）它先于 34 被选择出来，因此排序的结果是（34'，34），不稳定。

直接选择排序外循环迭代 $n-1$ 次，第 i 次循环内循环迭代（$n-1-i$）次，需要进行（$n-1-i$）次比较，1 次交换（3 次移动）。因此，总的比较次数为：

$$\sum_{i=0}^{n-2}(n-1-i) = n(n-1)/2 = \Theta(n^2)$$

整个排序过程最多需要 $n-1$ 次交换。在数据规模不大而每个数据记录结构很庞大的情况下，直接选择排序还是比较合适的。通常，移动大记录的单位时间开销比较大，此时直接选择排序可以最少的移动操作节省排序时间。另外，还可以通过 8.6.3 节介绍的索引排序等方法减少移动或交换大记录所花费的时间。

直接选择排序总的时间代价仍为 $\Theta(n^2)$。算法不依赖于原始数组的输入顺序，因此最大、最小、平均时间代价均为 $\Theta(n^2)$。

由于算法 8.3 中用到了交换操作，需要用到一个临时记录，因此空间代价为 $\Theta(1)$。

8.3.2 堆排序

在介绍直接选择排序时，采用的是直接从剩余记录中线性查找最大记录的方法，没有充分利用前一轮查找所能得到的信息。本节介绍的堆排序（heapsort），利用堆数据结构来保持剩余记录相对大小的信息，因而是更有效的选择排序。

堆有两种：最大堆和最小堆。这里介绍基于最大堆的排序算法。最大堆满足以下条件：堆中每个父结点中的数据项都要大于或等于其子结点中的数据项。因此，最大堆的堆顶记录就是整个堆中的最大记录。次大记录则存放在堆顶的左或右子结点中，堆排序正是利用这一点来查找剩余记录中的最大记录。

堆排序主要包括两个步骤：

（1）对所有记录建立最大堆，这个过程类似于第 5.5 节介绍过的最小堆。

（2）取出堆顶的最大记录与数组末端的记录交换，最大记录放在下标 $n-1$ 的位置，原数组末端元素临时处于根结点；将根元素向下调整到合适的位置，即剩下的 $n-1$ 个记录重新调整为堆，再取新堆顶最大的记录，与数组第 $n-2$ 位交换；……；不断重复这一操作，直到堆为空。这时数组正好是按从小到大排序。

例如，待排序数组为{45，29，32，12，34，34'，78}，堆排序的具体过程如图 8.4 所示。因

为建堆过程前面已介绍过，因此直接给出一个已转化为堆的数组{78,34,45,12,29,34',32}，如图 8.4(a)所示。将堆顶最大记录 78 删除到数组末端后，序列变为{32, 34, 45, 12, 29, 34'|78}，竖线后面的数字表示已排好序，待调整结构如图 8.4(b)所示。重新建堆后得到序列{45, 34, 34',12,29,32|78}，如图 8.4(c)所示。再次删掉堆顶的 45，并从根结点向下调整为新堆，得到序列{34, 32, 34', 12, 29 | 45, 78}，如图 8.4(d)所示。类似地重复操作，直到堆为空。

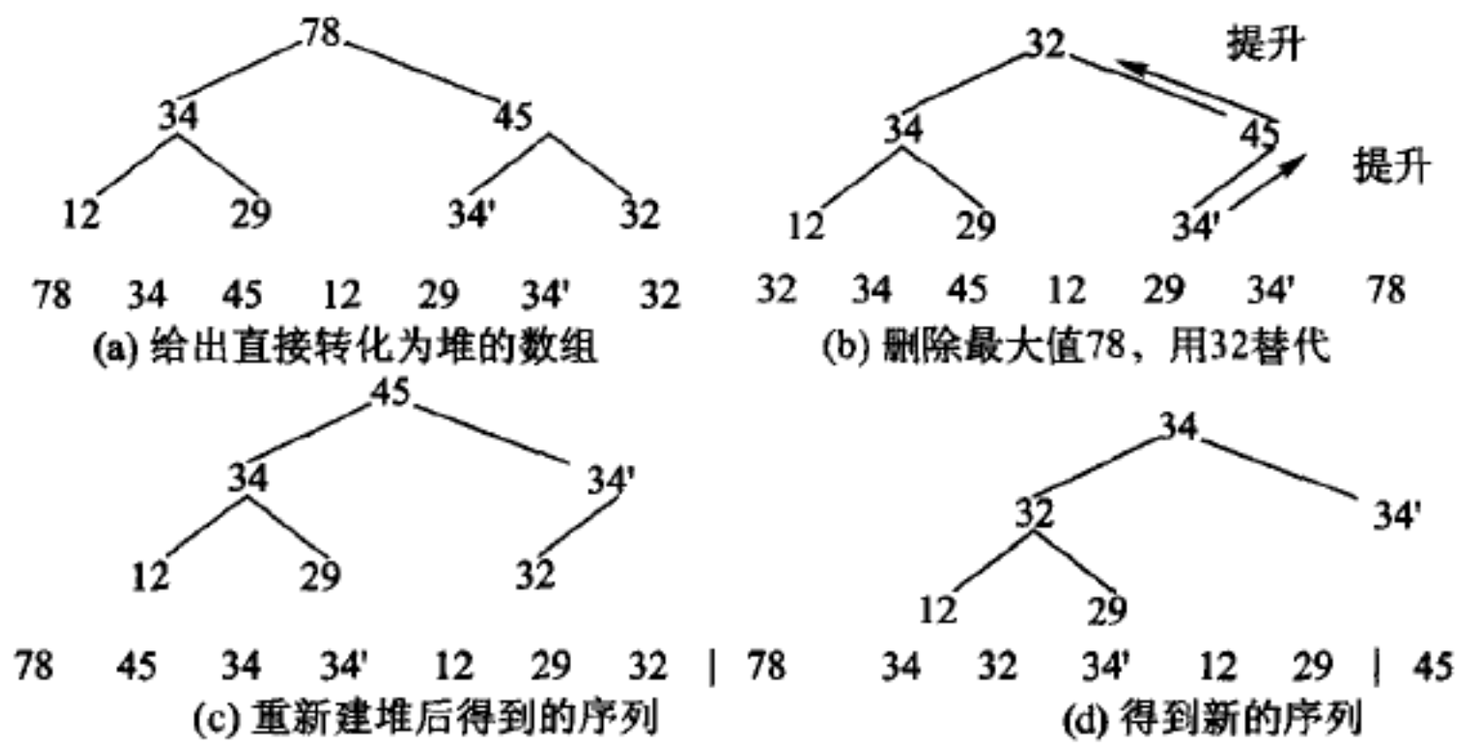


图 8.4 堆排序

堆排序是不稳定的，因为在建堆过程中，二叉树的父子结点之间的移动不能保证两个重复记录一定能保持原始输入顺序。例如，图 8.4(c)之后，第三个出堆的就是 34，最终排序结果为{12, 29, 32, 34', 34, 45, 78}，算法不稳定。算法 8.4 给出了堆排序的类模板定义和具体实现代码，其中函数初始化堆和 RemoveMax()的具体实现可参考 5.5.1 节。

【算法 8.4】 堆排序。

```
template <class Record>
void sort(Record Array[ ],int n)
{
    int i;
    MaxHeap<Record> max_heap = MaxHeap<Record>(Array,n,n);        // 建堆
    for(i=0; i < n-1; i++)                    // 依次找出剩余记录中的最大记录，即堆顶
        max_heap.RemoveMax();                 // 算法操作 n-1 次，最小元素不需要出堆
}
```

【算法 8.4 结束】

第 5 章已经介绍过建堆函数时间代价为 $\Theta(n)$，而每次删除堆顶重新建堆需要 $\Theta(\log n)$的时间，由于一共需要进行 n 次删除堆顶操作，因此堆排序的总时间代价为 $\Theta(n)+\Theta(n\log n)=\Theta(n\log n)$。理论上，堆排序最佳、最差、平均情况下的时间代价均为 $\Theta(n\log n)$。

由于堆排序中用到了一个临时记录变量，因此算法空间代价为 $\Theta(1)$。

堆排序的优点在于建堆很快，只需要 $\Theta(n)$时间，而且只需要一次建堆就可以反复使用，因此 n 越大效率越高。在后面第 9 章中，还将进一步详细介绍堆排序在外排序中的应用。

8.4 交换排序

交换排序的基本思想是：两两比较待排序记录的关键码，发现记录逆置则进行交换，直到没有逆置对为止。冒泡排序和快速排序是典型的交换排序算法。

8.4.1 冒泡排序

冒泡排序（bubble sorting，也称为起泡排序）的算法思想就是不停地比较相邻的记录，如果不满足排序要求，就交换相邻记录，直到所有的记录都已经排好序为止。对于长度为 n 的待排序记录数组 $R=\{r_0,r_1,\cdots,r_{n-1}\}$，冒泡排序将按照下述步骤对 R 进行排序：

（1）从数组末端开始，不断比较相邻记录，不满足排序要求就交换，例如首先比较倒数第一个记录 r_{n-1} 和倒数第二个记录 r_{n-2}，如果 $r_{n-1}<r_{n-2}$，则两者进行交换。然后，类似地依次对 r_{n-2} 和 r_{n-3}，r_{n-3} 和 r_{n-4}，…，r_2 和 r_1，r_1 和 r_0 进行比较处理。这样比较完一轮后，就会发现最小的那个记录已经被推到数组的最左端 r_0 位置上，就好像一个气泡从水底慢慢地冒上来，最后浮出水面。这就是冒泡排序的命名由来。

（2）开始第二轮冒泡过程，由于 r_0 已经是最小的记录，因此第二次冒泡只需要对 r_{n-1} 到 r_1 进行比较。第二次冒泡完成后，次小的记录就被推到 r_1 上。这时，r_0，r_1 已排好序，第三轮冒泡只需要从 r_{n-1} 比较到 r_2。依次类推，直到数组中所有记录都已经排好序为止。

冒泡排序算法还可以按照不断找出第 i 大的记录来进行：即第一次冒泡找出最大的记录放到数组最末端，第二次找出次大，……。算法的具体实现取决于编程人员的个人喜好。

图8.5所示为一个冒泡排序的例子。图中的每行表示第 i 次冒泡（$0\leqslant i<n-1$）时的序列，箭头连线表示尾部记录在本轮排序将调整到箭头所指记录之前。带下画线的数字表示该次冒泡中找到

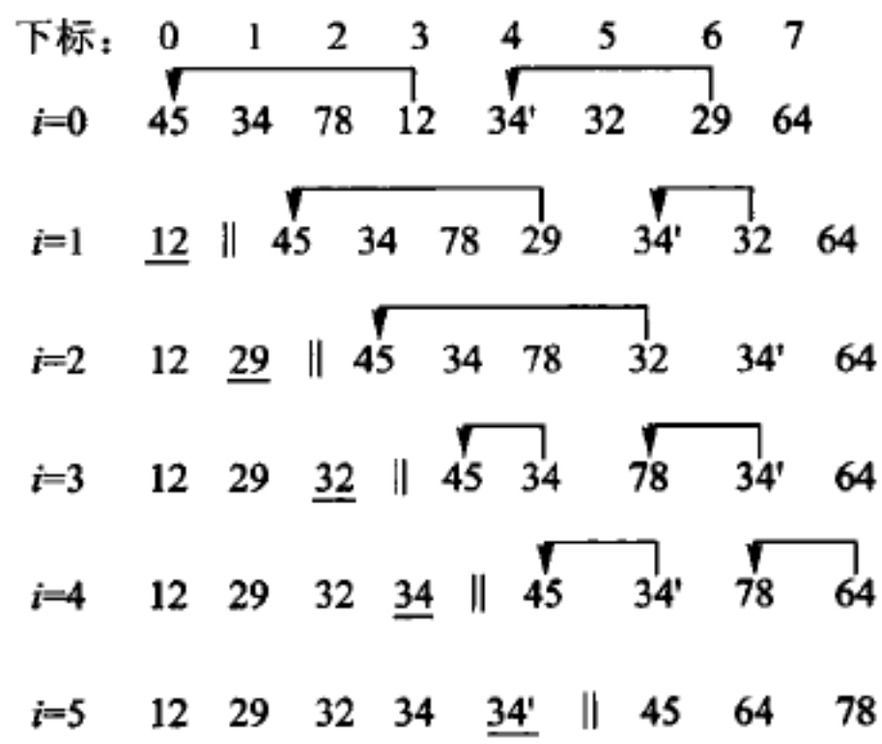


图8.5 冒泡排序

的第 i 小的记录。在第一次冒泡($i=0$)过程中,首先 29 越过 32、34',左移了两位,遇到 12 停下来,紧接着 12 越过 78、34、45,到达数组最左端,变成了第 2 行,等待 $i=1$ 这一次的排序。第二次冒泡($i=1$)时 12 已经是最小的数,不需要参与冒泡。$i=5$ 那次冒泡,没有发生一次交换操作,算法 8.5 中的变量 NoSwap 为真,整个数组事实上已经是有序的,继续冒泡已经没什么意义,退出算法。

与插入排序算法不同的是,冒泡排序在对第 i 个记录进行排序时,不需要改动前面已排好序的记录。算法 8.5 给出了冒泡排序的具体实现代码。

【算法 8.5】 冒泡排序。

```
template <class Record>
void BubbleSort(Record Array[], int n){   // 优化的冒泡排序,Array[]为待排序数组,n 为数组长度
    bool NoSwap;                          // 是否发生了交换的标志
    int i, j;
    for(i=0; i < n-1; i++){
        NoSwap = true;                    // 标志初始为真
        for(j=n-1; j > i; j--)
            if(Array[j] < Array[j-1]){// 判断(Array[j-1],Array[j])是否逆置
                swap(Array, j, j-1);      // 交换逆置对 Array[j],Array[j-1]
                NoSwap = false;           // 发生了交换,标志变为假
            }
        if(NoSwap)                        // 如果没发生过交换,表示已排好序,结束算法
            return;
    }
}
```

【算法 8.5 结束】

冒泡排序是稳定的。在图 8.5 中,$i=3$ 那次冒泡过程,首先右边的 34'越过 78,然后遇到左边 34,停下来;接着左边 34 越过 45,到达终点,结果如 $i=4$ 行所示。可以看出,冒泡排序并没有改变这两个 34 的原始顺序。

冒泡排序算法中用到了交换操作,需要用到一个临时记录,因此算法空间代价为 $\Theta(1)$。

现在来分析冒泡排序的时间代价。最佳情况发生在输入数组为正序时,只运行第一轮循环进行 $n-1$ 次比较后就发现没有发生过交换,结束排序,因此最小时间代价为 $\Theta(n)$。

最差情况下外层循环最多可能发生 $n-1$ 次,第 i 次内层 for 循环中共需要比较 $n-i$ 次,最多交换 $n-i$ 次。因此,最差情况下,总比较次数和交换次数都是:

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

平均交换次数为最多次数的一半,$\sum_{i=1}^{n-1}(n-i)/2 = n(n-1)/4 = \Theta(n^2)$,也为 $\Theta(n^2)$ 次。因此,冒泡排序的最大、平均时间代价均为 $\Theta(n^2)$。

8.4.2 快速排序

很自然地，人们会觉得短数组比长数组更好处理，因此可能会想到将原始数组分为若干个子部分然后分别进行排序。快速排序就是基于分治法的排序算法。分治法(divide and conquer)的关键是分、治、合，将给定问题分成若干个子问题(分)，再对每个子问题求解(治)，最后将所有子问题的解合并成一个综合的解(合)，得到原始问题的解。

1962 年，伦敦 Elliot Brothers Ltd 公司的 Tony Hoare 发明了快速排序(quicksort)方法。实际上快速排序名副其实，它几乎是最快的排序算法，被评选为 20 世纪十大算法之一。在 8.7.3 节将看到，其平均时间代价为 $\Theta(n \log n)$。快速排序之所以很快，主要因为它在对数组进行分组时不是随便划分而是尽量将原数组划分为两半。快速排序的算法思想主要是：

(1) 从待排序序列 S 中任意选择一个记录 k 作为轴值(pivot)。

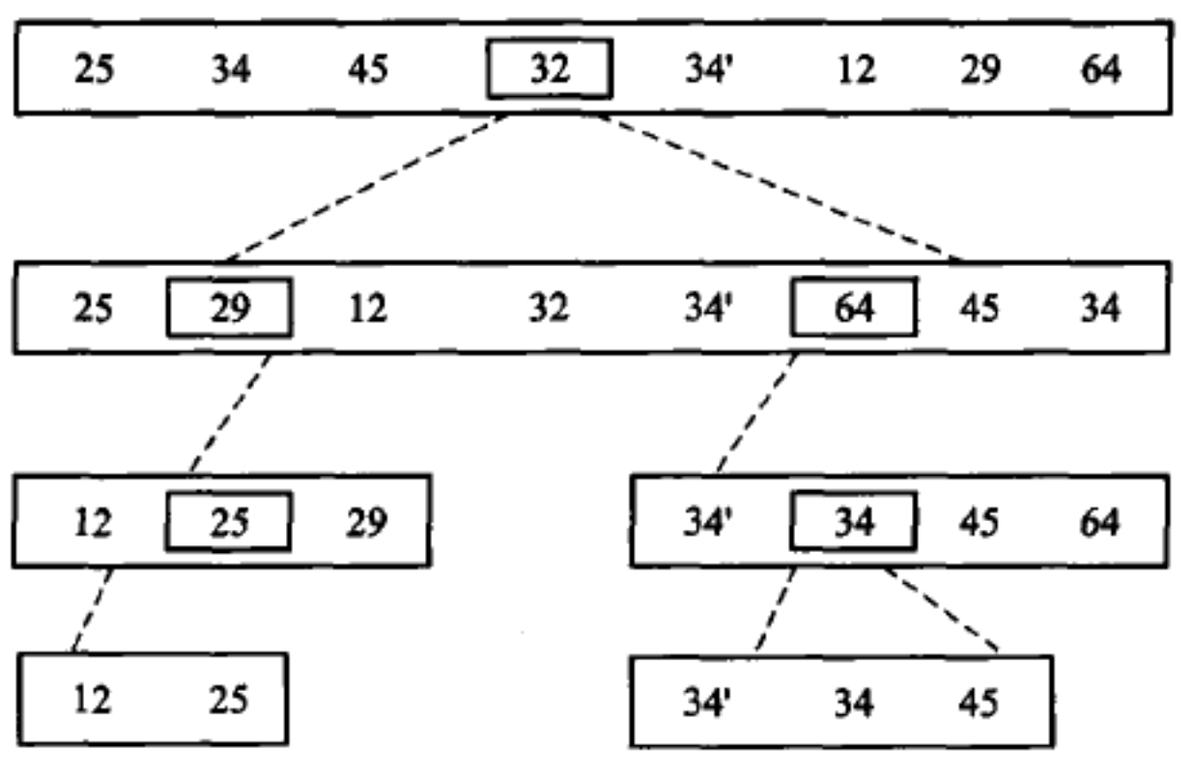
(2) 将剩余的记录分割(partition)成左子序列 L 和右子序列 R。

(3) L 中所有记录都小于或等于 k，R 中记录都大于等于 k，因此 k 正好位于正确的位置。

(4) 对子序列 L 和 R 递归进行快速排序，直到子序列中只含有 0 或 1 个元素，退出递归。

步骤(2)、(3)是“分”、“治”过程，可以同时实现，完成序列的分割。轴值 k 已经到位，而左序列 L 的记录再也不会跑到 k 的右边，反之亦然，因此不需要明显的“合”过程。

快速排序实现起来比较复杂，下面通过图 8.6 所示的例子先演示一下具体排序过程，其中带方框的数字为轴值。首先选择 32 作为轴值，分割后得到 $L=(25,29,12)$，$R=(34',64,45,34)$；对 L，选择轴值 29，分为(25,12)和一个空集合；对 R，选择轴值 64，分为(34,34', 45)和一个空集；然后继续分割下去，不断向下递归，直到集合中只剩下 0 或者 1 个元素，返回到上一层。如果将轴值看成子根，分割后的 L 和 R 分别看做为轴值的左右子结点，则整幅图可以看成一棵二叉树。对二叉树进行中序遍历，收集轴值以及单个元素的叶结点，就可以得到最终的排序结果。



最终排序结果： 12 25 29 32 34' 34 45 64

图 8.6 快速排序图示

下面讨论快速排序的具体实现方法。第一个问题就是如何选择轴值 pivot，这对快速排序算法的时间性能影响很大，轴值的选择应尽量使得序列可以据此划分为均匀的两半。从图 8.6 可以看到，最糟糕的情况莫过于轴值恰好是第一个或者最后一个记录，这样分出的两个子序列中就会有一个为空，从而使分治法根本起不到作用。

最简单的办法就是选择第一个记录（下标为 start）或者最后一个记录（下标为 end）作为轴值。但这样的弊端在于：当原始输入数组恰巧是正序或者恰好是逆序时，每次分割都会将剩余记录全都分到一个序列中，而另一个序列却为空。可以选取中间点（start + end）/2 的中间点作为轴值，这种轴值在输入数据为正序或逆序时可以平分序列，实验效果非常好。

解决了轴值的选择问题，下一步就该考虑如何实现最关键的分割过程。最简单的分割方法就是 l、r 下标分别从序列的左端、右端向序列中间扫描；左边越过那些小于等于 pivot 的值，停在第一个大于 pivot 的值 k_l；右边越过那些大于等于 pivot 的值，停在第一个小于 pivot 的值 k_r；交换逆置记录 k_l 和 k_r；从交换后的位置，继续从左右向中间扫描，发现并交换逆置记录对，直到 l、r 交叉而整个序列扫描完毕。这种方法适用于需要处理 l、r 边界，轴值最后定位等情况。

一种改进的方法就是从序列两端交替检查空闲位置，将逆置记录移动到空闲位置上，而不是直接交换逆置记录。不同的分割方法所分出来的子序列不同。图 8.7 是第二种分割方法的图示。

下标	0	1	2	3	4	5	6	7
待分割	25	34	45	32	34'	12	29	64
第1行	25	34	45	64	34'	12	29	32
	l							r
第2行	25	34	45	64	34'	12	29	32
		l						r
第3行	25	34	45	64	34'	12	29	34
		l					r	
第4行	25	29	45	64	34'	12	29	34
			l				r	
第5行	25	29	45	64	34'	12	45	34
			l			r		
第6行	25	29	12	64	34'	12	45	34
				l		r		
第7行	25	29	12	64	34'	64	45	34
				lr				
结果	25	29	12	32	34'	64	45	34

图 8.7 分割过程

初始序列如图 8.7 第一行所示，轴值为 32，用下画线标注。分割前先将轴值与最后一个记录交换，并将轴值保存到一个临时变量中，此时序列中的最后一个位置就是空闲位置，用方框表

示，如图中第二行所示，其中 l、r 分别用来扫描整个序列的左右指针，初始时分别位于序列两端。首先 l 指针从左边开始扫描，找到 34 大于轴值，停下，如图中第二行所示。将这个逆置记录放到空闲位置 r 上，这时 34 原来所在位置 l 就空闲了，可以用来存放下一个逆置记录，然后从 r 前面的记录开始寻找小于轴值的逆置记录，找到 29，停下，如图中第三行所示。将 29 放到空闲位置 l 上，这时 29 原来所在位置 r 变为空闲。不断循环下去，直到 l、r 相遇，表示整个序列已经扫描完毕，此时的空闲位置正好就是轴值的正确位置，将轴值 32 回填回去，如图中最后一行所示。此时，可以看到轴值 32 左边的记录都小于它，右边的记录都大于它，分割过程完毕。

分割完毕后，分别对轴值左边和右边的子序列递归进行快速排序。算法 8.6 给出了具体实现的代码。其中函数 Sort( ) 的参数 l 与 r 分别是待排序子序列左、右两端记录在原始数组中的下标。第一次调用时左端为 0，右端为 $n-1$。每一次分割后，左边的记录都小于或等于轴值，右边的记录都大于等于轴值。由于快速排序是不稳定的，因此等于轴值的情况可以有多种处理方法。算法 8.6 中对等于轴值的不同处理情况，在注释中予以了说明。

【算法 8.6】 快速排序。

```
template < class Record >
void QuickSort( Record Array[ ], int left, int right) {   // Array[ ]为待排序数组，left、right 分别为数组
                                                          // 两端
    if( right <= left) return;                            // 如果子序列中只有 0 或 1 个记录，就不需排序
    int pivot = SelectPivot( left, right);                // 选择轴值
    swap( Array, pivot, right);                           // 分割前先将轴值放到数组末端
    pivot = Partition( Array, left, right);               // 分割后轴值已到达正确位置
    QuickSort( Array, left, pivot - 1);                   // 对轴值左边的子序列进行递归快速排序
    QuickSort( Array, pivot + 1, right);                  // 对轴值右边的子序列进行递归快速排序
}

int SelectPivot( int left, int right) {          // 选择轴值，参数 left、right 分别表示序列的左右端下标
    return( left + right)/2;                              // 选择中间记录作为轴值
}

template < class Record >
int Partition( Record Array[ ], int left, int right) {    // 分割函数，分割后轴值已到达正确位置
    int l = left;                                         // l 为左指针，r 为右指针
    int r = right;
    Record TempRecord = Array[ r];                        // 将轴值存放在临时变量中
    while( l != r) {                                      // 开始分割，l、r 不断向中间移动，直到相遇
        // l 指针向右移动，越过那些小于等于轴值的记录，直到找到一个大于轴值的记录
        while ( Array[ l] <= TempRecord && r > l)// " <= "也可以改写为" < "，但增加记录移动
            l++;
```

```
        if(l < r){                          // 若 l、r 尚未相遇,将逆置元素换到右边的空位
            Array[r] = Array[l];
            r--;                            // r 指针向左移动一步
        }
        // r 指针向左移动,越过那些大于等于轴值的记录,直到找到一个小于轴值的记录
        while(Array[r] >= TempRecord && r > l)// // ">="也可以改写为">",但增加记录移动
            r--;
        if(l < r){                          // 若 l、r 尚未相遇,将逆置元素换到左边的空位
            Array[l] = Array[r];
            l++;                            // l 指针向右移动一步
    }
    }     // end while
    Array[l] = TempRecord;                  // 把轴值回填到分界位置 l 上
    return l;                               // 返回分界位置 l
}
```

【算法 8.6 结束】

图 8.6 和图 8.7 都是选择位于数组中间的值 Array[(left + right)/2]当作轴值,采用算法 8.6 的分割方法进行快速排序。由于记录移动的跨度比较大,因此快速排序是不稳定的。初始序列 34 排在 34'之前,一次分割后 34 就排在 34'之后,而且最终排序结果也不稳定。由于快速排序不稳定,因此遇到那些等于轴值的关键码时,处理的原则是尽量减少记录的移动。

从算法可以看出,快速排序的比较次数比移动次数要多很多,因此主要考虑比较次数。由于快速排序是递归排序,因此可以用递归推理来计算它的时间代价。假设对长度为 n 的序列进行快速排序所需的时间为 $T(n)$,当序列中只含有 0 或 1 个记录时,QuickSort()函数直接返回,时间代价为常数,在这里用 1 来表示,因此 $T(0)=T(1)=1$。选择轴值所用的时间也为常数,分割所需时间随 n 线性增长,设为 cn,分割后两个子序列的长度分别为 i 和 $n-i-1$,对它们进行快速排序所需时间分别为 $T(i)$ 和 $T(n-1-i)$,因此

$$T(n)=T(i)+T(n-1-i)+c\cdot n \tag{8.1}$$

最差情况下,每次分割都将记录全部分到一个子序列中,而另一个子序列为空,即 $i=0$ 或者 $n-1-i=0$,这种情况下,下次处理的子序列长度只比上次的少 1。因此,如果忽略掉常数时间 $T(0)$,有

$$\begin{aligned}
T(n)&=T(n-1)+c\cdot n\\
T(n-1)&=T(n-2)+c(n-1)\\
T(n-2)&=T(n-3)+c(n-2)\\
&\vdots\\
T(2)&=T(1)+c(2)
\end{aligned}$$

因此,总的时间代价为

$$T(n)=T(1)+c\sum_{i=2}^{n}i=\Theta(n^2) \tag{8.2}$$

可以看到，快速排序的最大时间代价为 $\Theta(n^2)$，看上去似乎并不比前面介绍的几种排序算法快。而快速排序最差情况下需要 $\Theta(n)$ 次分割，每次分割时递归算法都需要用到编译栈中一个临时记录来存储轴值，因此最大空间代价为 $\Theta(n)$。

在最好的情况下，每次分割都恰好将记录分为两个长度相等的子序列。这种情况下，

$$T(n) = 2T(n/2) + cn$$

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + c$$

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + c$$

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + c$$

$$\vdots$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + c$$

而算法共要进行 $\log n$ 次分割，所以

$$\frac{T(n)}{n} = \frac{T(1)}{1} + c\log n$$

$$T(n) = c\,n\log n + n = \Theta(n\log n)$$

因此，快速排序的最小时间代价为 $\Theta(n\log n)$。

平均情况应考虑到所有可能的情况，对各种情况所耗费的时间求和，然后除以总情况数。假设在每次分割时，轴值处于结果数组中各位置的概率是一样的。也就是说，轴值将数组分成长度为 0 和 $n-1$，1 和 $n-2$，…的子序列的概率是相等的，都为 $1/n$。在这种假设下，$T(i)$ 和 $T(n-1-i)$ 的平均值均为

$$T(i) = T(n-1-i) = \frac{1}{n}\sum_{k=0}^{n-1} T(k) \tag{8.3}$$

代入 8.1 式，得到

$$T(n) = cn + \frac{1}{n}\sum_{k=0}^{n-1}(T(k) + T(n-1-k)) = cn + \frac{2}{n}\sum_{k=0}^{n-1} T(k) \tag{8.4}$$

公式 8.4 两边同时乘 n，得到

$$nT(n) = cn^2 + 2\sum_{k=0}^{n-1} T(k) \tag{8.5}$$

以 $n-1$ 代入公式 8.5，得到

$$(n-1)T(n-1) = c(n-1)^2 + 2\sum_{k=0}^{n-2} T(k) \tag{8.6}$$

由公式 8.5 与公式 8.6 相减得到

$$nT(n) - (n-1)T(n-1) = c(2n-1) + 2T(n-1)$$

即

$$nT(n) = (n+1)T(n-1) + 2cn - c$$

忽略常数 $-c$,有

$$nT(n) = (n+1)T(n-1) + 2cn$$

可以得到

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$

$$\vdots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

所以

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\sum_{i=3}^{n+1}\frac{1}{i} = \Theta(\log n)$$

$$T(n) = \Theta(n\log n) \tag{8.7}$$

因此,快速排序平均时间代价为 $\Theta(n \log n)$,平均需要 $\Theta(\log n)$次分割,平均空间代价为 $\Theta(\log n)$。

事实上,快速排序还有改进的余地。当快速排序的子数组小于某个长度时,不必继续递归。这样,那些子数组中的数值是无序的;但是整块地看待这些子数组时,可以发现它们一块块地有序,即左边子数组的排序码都要小于右边数组的排序码。整个序列已经基本有序时,这样的待排序数组正适合使用插入排序,可以最后对整个数组进行一次插入排序。根据表 8.4 的实验,最好的组合方式是当 n(子数组的长度)小于等于 28 时就使用插入排序。阈值与算法、机器的软/硬件条件有关,在不同的环境下,得到过 9、16、28 等最佳阈值。

算法 8.7 给出了优化的快速排序的类模板定义以及具体的实现代码。其中,函数 SelectPivot()和 Partition()的具体实现与前面的快速排序算法中相同。

【算法 8.7】 优化的快速排序。

```
#define THRESHOLD 28
template < class Record >
void ModQuickSort(Record Array[ ], int left, int right){      // 优化的快速排序,
    if (right - left + 1 > THRESHOLD){                        // 处理长度大于阈值(28 最佳)的长子串
        int pivot = SelectPivot(left, right);                 // 选择轴值
        swap(Array, pivot, right);                            // 将轴值放在数组末端
        pivot = Partition(Array, left, right);                // 分割后轴值已到达正确位置
        ModQuickSort(Array, left, pivot - 1);        // 对轴值左边的子序列进行递归快速排序
        ModQuickSort(Array, pivot + 1, right);       // 对轴值右边的子序列进行递归快速排序
    }
}
template < class Record >
```

```
void Quicksort(Record * Array, int n){
    ModQuickSort(Array, 0, n-1);          // 调用优化的递归快排,不处理小子串
    InsertSort(Array, n);                 // 最后这个序列进行扫尾插入排序
}
```

【算法 8.7 结束】

实验证明,优化后的快速排序的确更快,但算法 8.7 并没有从数量级上改进原来的算法,因此时间代价仍为 $\Theta(n \log n)$,平均空间代价仍然为 $\Theta(\log n)$。

当然,还可以使用栈来消除递归调用。但是,由于目前编译器的优化工作做的越来越出色,因此这种优化措施的实际效果并不是很明显。考虑到算法的可读性,一般情况不建议读者做消除递归的优化。

随机化的快速排序(randomized quicksort,RQS)的主要思想是随机地选择一个数作为轴值。同前面介绍的快速排序一样,平均复杂度仍为 $\Theta(n \log n)$。但是实际应用中,随机化的快速排序却往往表现得比普通快速排序更好。

这就涉及一个所谓样本的随机性问题。快速排序的平均代价分析的前提假设是随机输入数据,即均匀分布。对快速排序的所有应用场合而言,这是对的;但对某一个系统而言,输入数据往往呈某种特定的分布,此时平均代价分析就不正确。也就是说,所有快速排序程序的平均运行时间是 $O(n \log n)$,但你的系统里却不一定是这样。简而言之,"输入随机"这个条件对快速排序这个抽象的算法成立,但是对于某一个特定的快速排序实现就不一定成立。

随机化的快速排序正好弥补了这个问题。通过随机选轴值,随机化的快速排序保证了输入数据在任何情况下都是随机的。因此,随机化的快速排序时间代价就是快速排序的平均复杂度 $\Theta(n\log n)$。随机化的快速排序最大程度地避免了快速排序陷入"坏数据"(bad data)中。

另一个改进是所谓的 Median-of-3 partition:随机找出 3 个元素,并取其中间的元素作为轴值。此算法的实验效果比随机化的快速排序还好。

8.5 归并排序

归并排序(merge sorting)简单地将原始序列划分为两个子序列,然后分别对每个子序列递归排序,最后再将有序子序列合并。归并排序的主要步骤为:

(1) 将序列划分为两个子序列。

(2) 分别对两个子序列递归进行归并排序。

(3) 将这两个已排好序的子序列合并为一个有序序列,即归并过程。

归并排序也是一种基于分治法的排序。快速排序侧重于"分"(含"治"),即用轴值分割子序列的过程,没有明显的"合"。归并排序的"分"很简单,它侧重于"治"和"合",即对两个有序子序列进行归并的过程:每次比较子序列头,取出较小的进入结果序列;其后次小记录顶上来,继

续比较两个子序列头，取出较小的进入结果序列；重复上述过程，直到其中一个子序列为空，剩余子序列中的记录就可以直接进入结果序列。

为了便于理解，首先看一下如图 8.8 所示的归并排序示例。序列中共有 8 个记录，首先不断向下将原始序列划分为越来越多的子序列，直到子序列长度为 1，停止划分，此时共有 8 个长度为 1 的子序列。然后，进行第一轮归并，将 4 对长度为 1 的子序列归并为 4 个长度为 2 的子序列，再归并出 2 个长度为 4 的子序列，最后再归并成一个长度为 8 的序列，得到最终结果。

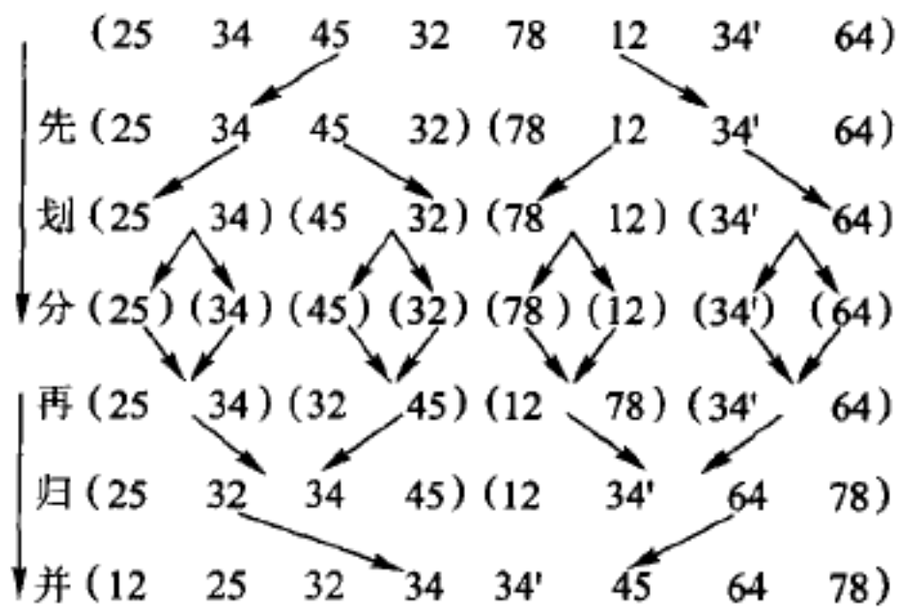


图 8.8 归并排序

归并过程最关键，图 8.8 在合并(25,34)和(32,45)时，首先对两个序列的最小记录 25 和 32 进行比较，发现 25 小于 32；因为 25 在它本身所在序列中就是最小的，而它又小于另一个序列的最小值，因此此时 25 就是合并序列中的最小值。从第一个序列中取出 25 放到合并序列中第一个位置；此时，次小记录 34 就成为该序列的新的最小记录，34 与 32 进行比较，发现 32 较小，类似地将 32 放到合并序列中第二个位置。继续用这种方法，不断比较两个数组中未被处理序列最前端的记录，并将较小的记录依次放入合并序列中。

算法 8.8 给出了两路归并排序的类模板定义以及具体的实现代码。

【算法 8.8】 两路归并排序。

```
template <class Record>
void MergeSort(Record Array[], Record TempArray[], int left, int right){   // Array 为待排序数组
                                                                           // left、right 为两端下标
    int middle;
    if(left < right){                                   // 如果序列中只有 0 或 1 个记录，就不用排序
        middle = (left + right)/2;                      // 从中间划分为两个子序列
        MergeSort(Array,TempArray,left,middle);         // 对左边一半进行递归
        MergeSort(Array, TempArray,middle + 1,right);   // 对右边一半进行递归
        Merge(Array, TempArray,left,right,middle);      // 进行归并
    }
}
```

```
// 两个有序子序列都从左向右扫描,归并到新数组
template <class Record>
void Merge(Record Array[], Record TempArray[], int left, int right, int middle){
    int i, j, index1, index2;
    for(j = left; j <= right; j++)                          // 将数组暂存入临时数组
        TempArray[j] = Array[j];
    index1 = left;                                          // 左边子序列的起始位置
    index2 = middle + 1;                                    // 右边子序列的起始位置
    i = left;                                               // 从左开始归并
    while (index1 <= middle && index2 <= right){
        // 取较小者插入合并数组中
        if(TempArray[index1] <= TempArray[index2])          // 为保证稳定性,相等时左边优先
            Array[i++] = TempArray[index1++];
        else Array[i++] = TempArray[index2++];
    }
    while (index1 <= middle)                                // 只剩左序列,可以直接复制
        Array[i++] = TempArray[index1++];
    while (index2 <= right)                 // 与上个循环互斥,直接复制剩余的右序列
        Array[i++] = TempArray[index2++];
}
```

【算法 8.8 结束】

从算法 8.8 中可以看出,归并排序所需时间主要包括划分时间、两个子序列的排序时间以及归并时间。划分时间为常数,可以忽略。归并时间随着数组长度 n 线性增长,暂且记为 n。因此,对一个长度为 n 的数组进行归并排序的时间代价为

$$T(n) = 2T(n/2) + c \cdot n \tag{8.8}$$

当数组长度为 1 时,函数直接返回,$T(1) = 1$。因此,与前面的快速排序类似,归并排序的总时间代价也为 $\Theta(n \log n)$。由于归并排序不依赖于原始数组的输入情况,每次划分时两个子序列的长度都是基本一样的,因此归并排序的最大、最小以及平均时间代价均为 $\Theta(n \log n)$。

归并过程需要用到一个临时数组来保存原记录数组,因此归并排序的空间代价为 $\Theta(n)$。

归并排序也有改进的余地。下面是 R. Sedgewick 发明的一个优化归并排序方法,在把数组暂时复制到临时数组时,将第二个子数组中元素的顺序颠倒了一下。这样,两个子数组从两端开始处理,向中间推进,使得这两个子数组的两端互相成为另一个数组的"监视哨",从而不用像算法 8.8 那样需要检查子序列结束的情况。另外,当子数组小于某个长度(例如 28)时,也不继续递归,而是对整个序列使用插入排序。

算法 8.9 给出了优化的两路归并排序的类模板定义以及具体的实现代码。

【算法 8.9】 优化的两路归并排序。

```
#define THRESHOLD 28
```

```
template <class Record>
void ModMergeSort(Record Array[], Record TempArray[], int left, int right){   // Array 为待排序数
                                                                               // 组,left、right 两端
    int middle;
    if(right - left + 1 > THRESHOLD){          // 如果序列长度大于阈值(28 最佳),递归进行归并
        middle = (left + right)/2;             // 从中间划分为两个子序列
        ModMergeSort(Array, TempArray ,left,middle);          // 对左边一半进行递归
        ModMergeSort(Array, TempArray ,middle + 1,right);     // 对右边一半进行递归
        ModMerge(Array, TempArray ,left,right,middle);        // 进行归并
    }
    elseInsertSort(&Array[left],right - left + 1);        // 小长度子序列进行插入排序,
                                                          // "&"传 Array[left]的地址
}

// 优化的 Sedgwick 两个有序子序列归并,右子序列逆置了,都从两端向中间扫描,归并到新数组
template <class Record>
void ModMerge(Record Array[],Record TempArray[],int left,int right,int middle){     // 归并过程
    int index1,index2;      // 两个子序列的起始位置
    int i,j,k ;
    for(i = left; i <= middle; i++)                // 复制左边的子序列
        TempArray[i] = Array[i];
    for(j = 1; j <= right - middle; j++)           // 复制右边的子序列,但顺序颠倒过来
        TempArray[right - j + 1] = Array[j + middle];
    // 开始归并,取较小者插入合并数组中
    for(index1 = left, index2 = right, k = left;k <= right;k++)
        if(TempArray[index1] <= TempArray[index2])     // 为保证稳定性,相等时左边优先
            Array[k] = TempArray[index1 ++];
            else Array[k] = TempArray[index2 --];
    }
```

【算法 8.9 结束】

实验证明,优化后的算法的确比原算法快一些。但是,这种优化并没有从数量级上改善原来的算法,因此时间代价仍为 $\Theta(n \log n)$,空间代价仍为 $\Theta(n)$。

8.6 分配排序和索引排序

本章要介绍的最后一种排序算法是分配排序(distribution sorting),这种排序算法的唯一特征是不需要进行关键码之间的比较。但是需要事先知道记录序列的一些具体情况。

如果事先知道序列中记录的排序码都位于某个小区间段内，就可以用下面介绍的桶式排序。当排序码值域 m 很大时，可以拆分为多个部分来进行比较，这就是基于桶式排序的基数排序(radix sorting)。为减少记录的移动，可以采取基于静态链的基数排序，这本质上是一种索引排序(index sorting)或称地址排序(address sorting)。索引排序后的结果应该可以在 $\Theta(n)$ 的线性时间内整理为按照数组下标为序的有序序列。

8.6.1 桶式排序

桶式排序(bucket sorting，或 bin sorting)是一种十分简单的分配排序，其示意图如图 8.9 所示。为了简化讨论，在这一节中，将记录的排序码值看成记录的值，且值的类型为整型。

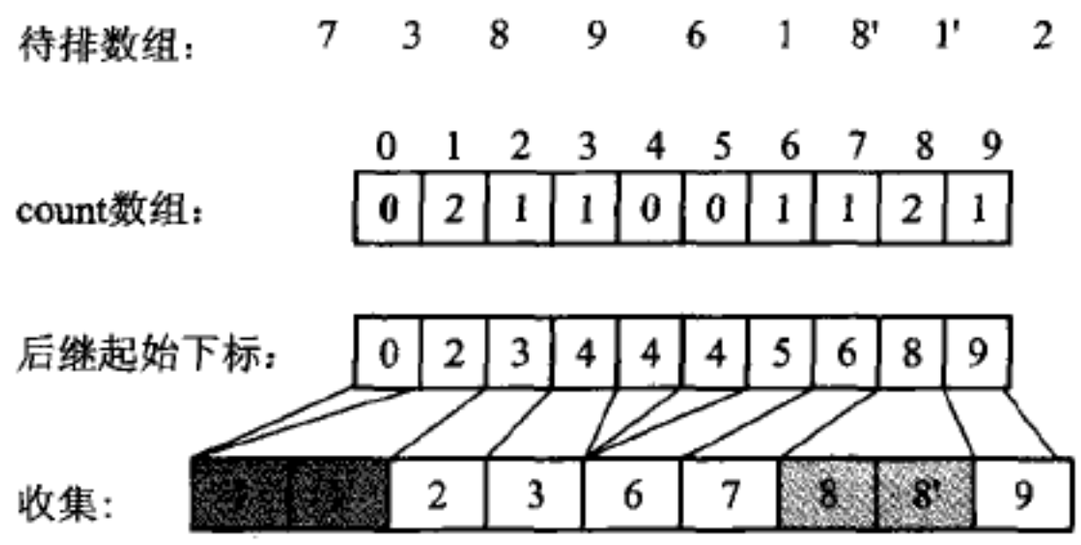


图 8.9 桶式排序示意图

假如知道某个长度为 n 的序列中所有记录的值都在 $0 \sim m-1$ 之间，如果事先知道每个桶中会有多少个元素，就可以使用一个长度为 n 的辅助数组。例如，如果第一轮第 0 个桶将含有 3 个记录，第 1 个桶将接收 5 个记录，就可以预先将前 3 个位置空出来留给第 0 个桶使用，把接着的 5 个位置空出来留给第 1 个桶使用。因此，还需要用 m 个计数器分别来统计 $0,1,2,\cdots,m-1$ 这些数出现的个数，将具有相同值的记录都分配到同一个桶中，然后依次按照编号从桶中取出记录，组成一个有序序列。

例如当 $m=10$ 时，来看这样一个序列{7, 3, 8, 9, 6, 1, 8', 1', 2}，事先已经知道序列中的数都是 0 ~ 9 之间的，可以扫描一遍序列，统计出 0,1,2,…9 这 10 个数出现的次数分别为 0、2、1、1、0、0、1、1、2、1，存放在 count 数组中。根据 count[i] = count[i-1] + count[i] 累积计算出新的 count 数组，可以算出每个元素 i(i∈[0,10))在结果数组中的结束下标位置依次为 0、2、3、4、4、4、5、6、8、9。count[i]记录了 i+1 的开始位置，也可以说是元素 i 的结束位置，即元素 i 应该从 Array[count[i]-1]往前追溯，存放 count[i] - count[i-1]个。最后，输出排序结果应该从初始序列的尾部开始往前扫描，以保证算法的稳定性。假设元素 i(例如 8)出现两次，i 和 i'，这样能够把原始序列中出现在较后的 i'先安排在 Array[count[i]-1](8'在结果数组下标 7 的位置)，后面再遇见 i 时，则安排在 Array[count[i]-2](8 安排在下标 6)。最后，得到的序列应该是{1, 1',2,3,6,7,8,8',9}。

算法 8.10 给出了桶式排序的算法。算法针对记录值域为[0, max)的整数序列进行排序。

【算法 8.10】 桶式排序。

```
template <class Record>
void BucketSort(Record Array[], int n, int max){        // Array 数组长度为 n,所有记录都位于
                                                        // 区间[0,max)上
    Record * TempArray = new Record[n];                 // 临时数组
    int * count = new int[max];                         // 桶容量计数器,count[i]存储了
                                                        // 第 i 个桶中的记录数
    int i;
    for(i =0; i < n; i++)                               // 把序列复制到临时数组
        TempArray[i] = Array[i];
    for(i =0; i < max; i++)                             // 所有计数器初始都为 0
        count[i] =0;
    for(i =0; i < n; i++)                               // 统计每个取值出现的次数
        count[Array[i]] ++;
    for(i =1; i < max; i++)                             // 统计小于等于 i 的元素个数
        count[i] = count[i-1] + count [i];              // count[i]记录了 i+1 的开始位置
    for(i =n-1;i >=0; i--)                              // 从尾部开始按顺序输出,保证排序的稳定性
        Array[--count[TempArray[i]]] = TempArray[i];    // TempArray[i]放到
                                                        // Array[count[TempArray[i] -1]中
}
```

【算法 8.10 结束】

对于一个长度为 n 且值域区间长度为 m 的数组,例如$[0,m)$,桶式排序首先需要扫描一遍序列,以便进行统计计数,然后再统计小于等于 $i(i\in[0,m))$的个数,输出有序序列时共循环 n 次,因此总的时间代价为 $\Theta(m+n)$。由于算法中需要用到 m 个计数器,还需要一个长度为 n 的临时数组,因此空间代价为 $\Theta(m+n)$。

当 m 为 $\Theta(n)$数量级时,时间代价为 $\Theta(\Theta(n)+n)$,还是 $\Theta(n)$。此时,桶式排序的效率相对于前面那些排序算法是一个飞跃。但是,当 m 为更高数量级,例如 $\Theta(n\log n)$或 $\Theta(n^2)$时,时间代价就由 m 来决定,也变成 $\Theta(n\log n)$或 $\Theta(n^2)$。此时,桶式排序相比于前面那些排序就没有什么优势,不仅如此,桶式排序还需要额外的 $\Theta(m+n)$的空间代价。因此,桶式排序只对于 m 较小时才具有实际意义。

8.6.2 基数排序

当值域 m 很大时,可以对桶式排序做一些改进:将排序码拆分为多个部分来进行比较。例如,如果要对 0 ~ 9999 之间的整数进行排序,可以先按千、百、十、个位拆分。这种将排序码按照其进制的基数进行拆分排序的方法就是基数排序,这是分配排序的一种特例。

假设一个长度为 n 的序列 $R=\{r_0,r_1,\cdots,r_{n-1}\}$，记录的排序码 K 包含 d 个子排序码 $K=(k_{d-1},k_{d-2},\cdots,k_1,k_0)$，称序列 R 对排序码 $(k_{d-1},k_{d-2},\cdots,k_1,k_0)$ 有序就是对于任意的两个记录 $R_i,R_j(0\leqslant i<j\leqslant n-1)$ 都满足

$$(k_{i,d-1},k_{i,d-2},\cdots,k_{i,1},k_{i,0})\leqslant(k_{j,d-1},k_{j,d-2},\cdots,k_{j,1},k_{j,0})$$

其中 k_{d-1} 称为最高位排序码，k_0 称为最低位排序码。

这 d 个排序码可以是不同类型。例如，按照桥牌规则，扑克牌的排序码就是（花色，面值）的组合值，花色按黑桃（S）>红心（H）>梅花（D）>方片（C）的顺序，面值按 A>K>Q>J>10>9…>3>2 的顺序。

人们对整数的运算是十进制的，因此为方便讨论，本章例子都选取 10 作为基数。例如，4 位数拆分为 $K=(k_3,k_2,k_1,k_0)$，$0\leqslant k_i\leqslant 9(i=3,2,1,0)$，其中 k_3 为最高位千位，k_0 为最低位个位。其实，对于由 0、1 代码组成的整数，还可以看做二进制、四进制、八进制、十六进制，而采用基数 $r=2$、4、8、16。对于字符串，则可以采用基数 $r=26$（英文字母个数）。

按照排序码的先后顺序，分配排序分为两种：高位优先法和低位优先法。

（1）高位优先法（most significant digit first，MSD）分配排序。

先对高位 k_{d-1} 排序，将序列分成若干子序列，每个子序列具有相同 k_{d-1} 值；然后，对每个子序列按次高位 k_{d-2} 排序，分成更小的子序列；依次重复，直到对 k_1 排序后，分成的极小子序列，每个子序列内部含有相同的排序码 $(k_{d-1},k_{d-2},\cdots,k_1)$。最后，对每个子序列用最低位 k_0 排序，将所有子序列依次连接在一起，成为一个有序序列。这是一个分、分、…、分、收的过程，是一个递归分治问题。上面的 4 位数排序例子就是按照高位优先法来排序的。

（2）低位优先法（least significant digit first，LSD）分配排序。

从最低位 k_0 开始排序，对于排好的序列再用次低位 k_1 排序，依次重复，直至对最高位 k_{d-1} 排好序后，整个序列成为有序的。这是一个分、收；分、收；…；分、收的过程。

例如，对扑克牌这样的组合排序码，对扑克牌 S3 HJ C8 H9 S9 D3 CA D7 排序。

（1）MSD 方法：先按花色分 4 堆，各堆中再按牌点分，最后把 4 堆合并。

先按花色：C8 CA D3 D7 HJ H9 S3 S9

再按面值：C8 CA D3 D7 H9 HJ S3 S9

（2）LSD 方法：先按牌点分成顺序的 13 堆，再依次从这 13 堆中顺序取出 4 种花色。

先按面值：S3 D3 D7 C8 H9 S9 HJ CA

再按花色：C8 CA D3 D7 H9 HJ S3 S9

在人们看起来似乎觉得 MSD 比较直观，一般都采用 MSD 方法。但是计算机常用 LSD 进行分配排序，因为它速度较快，便于统一处理；而 MSD 在分配之后还要处理各子集的排序问题，这通常是一个递归过程，计算机处理比较复杂。本节的算法都是讨论 LSD 方法。

在讨论基数排序的具体实现前，先看一下如图 8.10 所示的两位数基数排序。图 8.10(a) 按照高位优先来排序，先按照十位将数据分到 10 个桶中，每格为一个桶，可以看到十位为 0 和 1 的桶都为空，十位为 2 的桶中含有两个记录，…，十位为 9 的桶中有 0 个记录。然后，分别对每个桶

再按个位进行分配，例如对十位为 2 的那个桶，会进一步分为 10 个更小的子桶，其中个位为 2 和 6 的桶中各有 1 个记录，其他数字的桶都为空。最后，一共有 100 个子桶，分别对应 0 ~ 99 之间的所有数据，然后再将每个子桶中的记录按顺序收集起来。

图 8.10(b)按照低位优先来排序，先按照个位将数组分到 10 个桶中，每格为一个桶，可以看到个位为 0 的桶为空，个位为 1 的桶中含有两个记录，……个位为 9 的桶中有 1 个记录，然后将这些桶中的记录依次收集起来。第二次再按照十位进行分配，分为 10 个桶，例如对十位为 5 的那个桶，收集后得到(53,59)，然后再将每个桶中的记录依次收集起来。

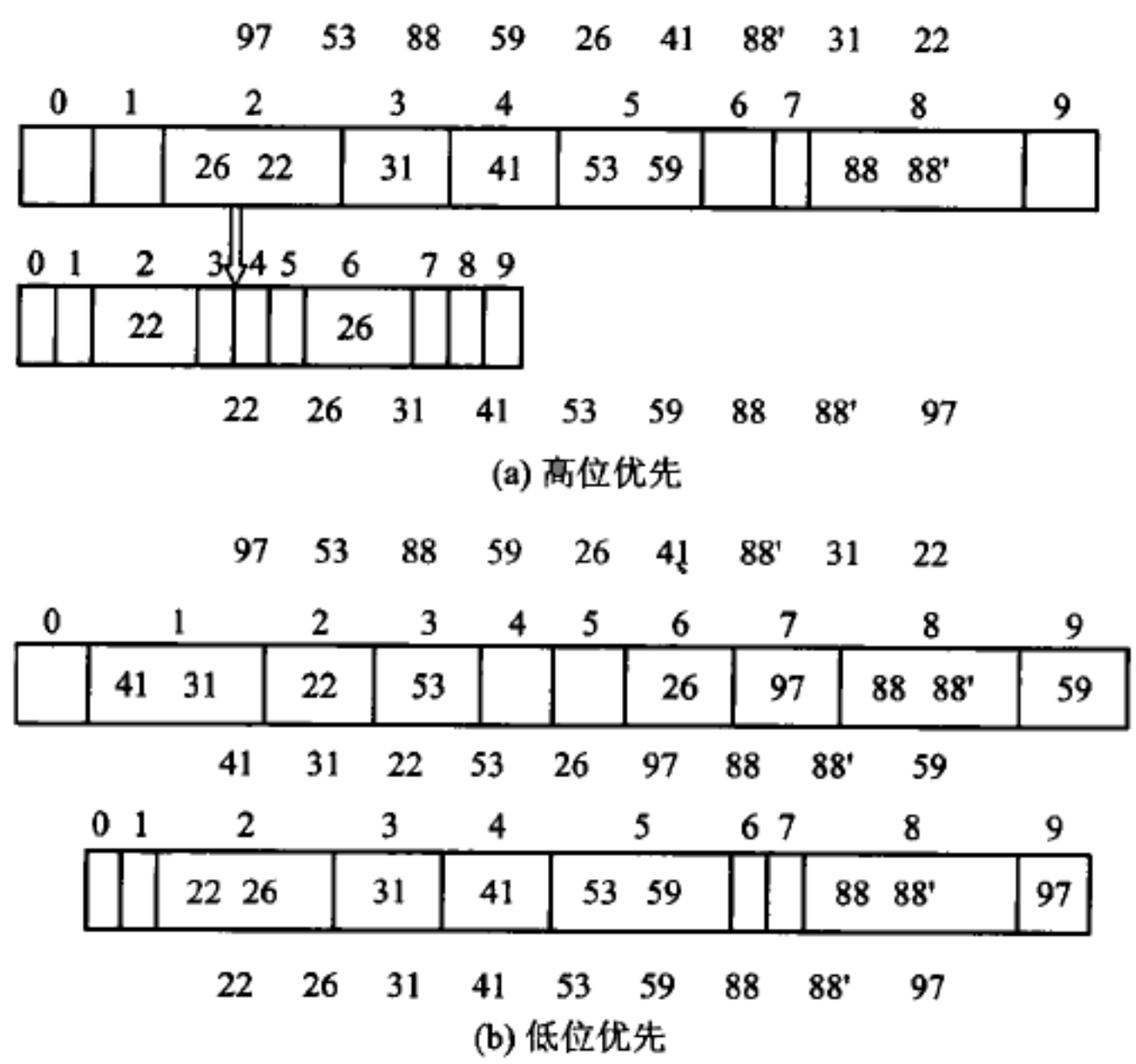


图 8.10 基数排序

在具体实现基数排序时，数据项的存储方式有两种：顺序存储和链式存储。

1. 顺序存储

假设原始输入数组 S 的长度为 n，基数为 r，排序码个数为 d。最简单的实现方法就是：用队列数组来表示桶，每次把 S 中所有记录分配到 r 个队列(桶)中；然后，进行收集，将 r 个队列(桶)中的记录收集回 S 中。这样的分配和收集一共需要进行 d 次。

这种实现方法的时间和空间代价很大。因为需要用一个数组 S 存放待排序的所有记录，再用 r 个数组存储分配时的 r 个队列(即桶)，这样每分配一次需要移动 n 个记录，收集起来又要移动 n 个记录，d 遍分配和收集共需 $2dn$ 次移动，时间代价很高。另一方面，由于每个队列的大小也不好估计，每个队列中都可能有 0 ~ n 个元素，若都按 n 分配，则共需 rn 个结点空间，显然非常浪费。

实际上可以 d 次调用桶式排序来实现，下面的算法 8.11 实现了这个思想。

【算法 8.11】 顺序方法的基数排序(d 次桶式排序)。

```
template <class Record>
void RadixSort(Record Array[], int n, int d, int r){   // n 为数组长度,d 为排序码个数,r 为基
    Record * TempArray = new Record[n];                 // 辅助排序的临时数组
    int * count = new int[r];                           // 桶容量计数器,count[i]存储了第 i 个桶中的记录数
    int i, j, k;
    int Radix = 1;                                      // 模保持进位,用来取 Array[j]的第 i 位排
    序码
    for (i = 1; i <= d; i++){                           // 分别对第 i 个排序码进行分配
        for(j = 0; j < r; j++)
            count[j] = 0;                               // 初始计数器均为 0
        for(j = 0; j < n; j++){                         // 统计每个桶中的记录数
            k = (Array[j] / Radix) % r;                 // 取 Array[j]的第 i 位排序码
            count[k] ++;                                // 相应计数器加 1
        }
        for(j = 1; j < r; j++)                          // 将 TempArray 中的位置依次分配给 r 个桶
            count[j] = count[j - 1] + count[j];
        // count[i]记录了 i + 1 的开始位置,元素 i 应该从 Array[count[i] - 1]往前追溯,
        // 存放 count[i] - count[i - 1]个
        for(j = n - 1; j >= 0; j--){                    // 从数组尾部,把记录收集到相应桶
            k = (Array[j] / Radix) % r;                 // 取 Array[j]的第 i 位排序码,Array[j]应该
                                                        // 放到 count[k] - 1 处
            count[k] --;                                // 使用了第 k 个桶的一个位置,桶剩余量的计
                                                        // 数器减 1
            TempArray[count[k]] = Array[j];             // Array[j]放到 TempArray 中,下标位置在上
                                                        // 面已经被减 1
        }
        for(j = 0; j < n; j++)                          // 将临时数组中的内容复制到 Array 中
            Array[j] = TempArray[j];
        Radix *= r;                                     // 向左进一位,修改模 Radix
    }
    delete[]TempArray;                                  // 释放临时数组空间
    delete[]count;                                      // 释放桶计数器空间
}
```

【算法 8.11 结束】

图 8.11 中给出了一个基数排序的例子。初始数组内容为{97,53,88,59,26,41,88',31,22}，$n=9$，$d=2$，$r=10$。第一趟按个位进行分配，如图 8.11(a)所示。首先，统计 count 数组，然

后将临时数组中的位置按 count 数组的值依次分配给 r 个桶，最后进行收集，第一趟收集后的结果为{41,31,22,53,26,97,88,88',59}。图 8.11(b)类似地按十位进行第二趟分配和收集，得到最后的排序结果{22,26,31,41,52,59,88,88',97}。

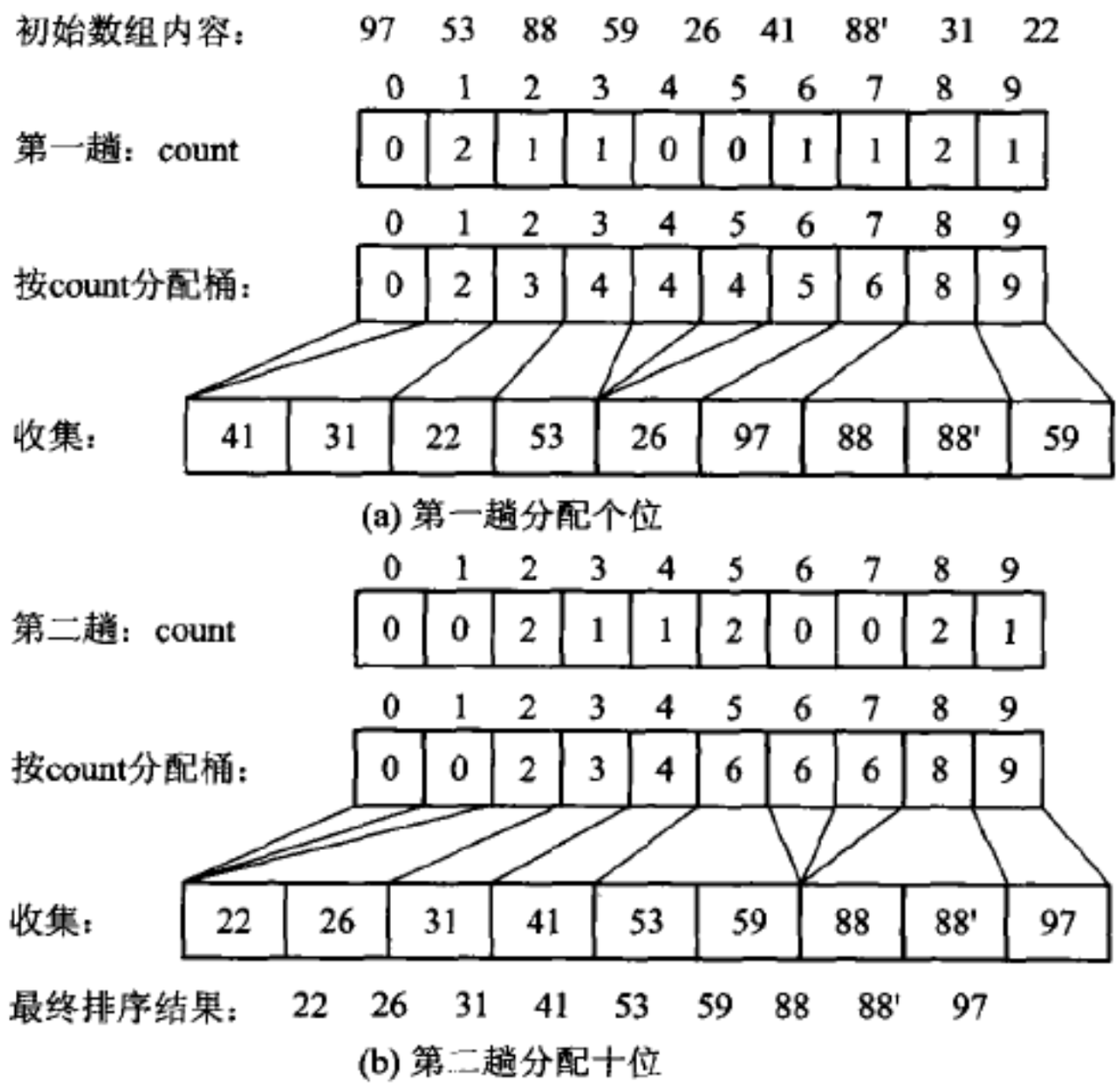


图 8.11 基于顺序存储和桶排序的基数排序

现在来分析顺序基数排序的算法代价。算法需要用一个长度为 n 的临时数组 TempArray 来存放待排序的记录，用一个长度为 r 的计数器数组来统计每个桶中的记录个数，因此空间代价为 $\Theta(n+r)$，当 n 较大时，显然也很浪费空间，但是比起前面的 $r \cdot n$ 要好得多。前面已经介绍过，桶式排序的时间代价为 $\Theta(m+n)$，而基数排序相当于作了 d 次桶式排序，每次 $m=r$，因此顺序基数排序的时间代价为 $\Theta(d(n+r))$。

2. 链式存储(静态链)

前面讨论了一个简单的顺序存储实现方法，其空间代价非常高，主要是因为每个桶中的记录个数未定，如果都按照最大记录数 n 来分配，就需要分配 $r \cdot n$ 个记录空间，而事实上只用到其中的 n 个记录空间，这样会浪费很多空间。而另一方面，分配和收集时都需要移动所有记录，导致时间代价也较高。

这可以通过链式存储的 LSD 来改进，在原数组中给每个记录设置整数 next 域，该域指向的是同一个桶中下一个元素的下标，这 r 个桶组成 r 个静态队列。LSD 分配第 i 位时，按照某记录排序码 i 位的值分配到相应的队列，实际上只需要修改该记录的 next 域；收集时，只需要利用 next 域依次把各队列首尾相连。

n 个待排序记录都有 next 段，再加上 r 个桶的头尾指针，因此空间代价为 $\Theta(n+r)$。时间方面，根据链式存储的优点，分配和收集时不需要移动记录本身，只需要修改记录的 next 指针；处理一位时，扫描一遍待排数组进行分配的时间为 $\Theta(n)$，收集 r 个队列的时间为 $\Theta(n+r)$，因此 d 遍分配和收集的时间代价为 $\Theta(d(n+r))$。

下面给出基于静态链的具体实现代码，如算法 8.13 所示。首先，在代码 8.12 中给出静态链结点和队列的类定义。在实现时，静态链结点类 Node 实例化 Record 模板类。

【代码 8.12】 静态队列定义。

```
//结点类
class Node{
public:
    int key;                    // 结点的排序码值
    int next;                   // 下一个结点在数组中的下标
};

// 静态队列类
class StaticQueue{
public:
    int head;
    int tail;
};
```

【代码 8.12 结束】

【算法 8.13】 基于静态链的基数排序。

```
// 静态链实现的基数排序,n 为数组长度,d 为排序码个数,r 为基数
template <class Record>
void RadixSort(Record *Array, int n, int d, int r){
    int i, first=0;                          // first 指向静态链中第一个记录
    StaticQueue *queue=new StaticQueue[r];   // 存放 r 个桶的静态队列
    for(i=0; i < n-1; i++)                   // 初始化建链,相邻记录的静态指针链接为单链表
        Array[i].next=i+1;                   // 第 i 个记录的静态指针域设为 i+1,
                                             // 即下一个元素的下标
    Array[n-1].next=-1;                      // 链尾 next 为空
    for(i=0; i < d; i++){                    // 对第 i 个排序码进行分配和收集,一共 d 趟
        Distribute(Array, first, i, r, queue);
        Collect(Array, first, r, queue);
    }
    delete[] queue;
    AddrSort(Array, n, first);       // 线性时间整理静态链表,使得数组按下标有序
```

```
}

// 分配过程,A 中存放待排序记录,first 为静态链中的第一个记录,i 为第 i 个排序码,r 为基数
template <class Record>
void Distribute(Record *Array, int first, int i, int r, StaticQueue *queue){
    int j, k, a, curr = first;
    for(j = 0; j < r; j++)                          // 初始化 r 个队列
        queue[j].head = -1;
    while(curr != -1){                              // 对整个静态链进行分配
        k = Array[curr].key;                        // 取第 i 位排序码数字 k
        for(a = 0; a < i; a++)
            k = k / r;
        k = k % r;
        // 把 Array[curr]分配到第 k 个桶中
        if(queue[k].head == -1)                     // 如果桶为空,Array[curr]就是第一个记录
            queue[k].head = curr;
        else Array[queue[k].tail].next = curr;      // 否则加到桶的尾部
        queue[k].tail = curr;                       // 当前记录的下标 curr 被标记为该桶的尾部
        curr = Array[curr].next;                    // 静态指针 curr 移动一位,继续分配下一个记录
    }
}

// 收集过程,Array 中存放待排序记录,first 为静态链中的第一个记录,r 为基数
template <class Record>
void Collect(Record *Array, int& first, int r, StaticQueue *queue){
    int last, k = 0;                                // 已收集到的最后一个记录
    while(queue[k].head == -1)                      // 找到第一个非空队列
        k++;
    first = queue[k].head;
    last = queue[k].tail;
    while (k < r-1){                                // 继续收集下一个非空队列,r-1 是最后一个
                                                    // 找下一个非空队列
        k++;
        while(k < r-1 && queue[k].head == -1)       // 当前队列 k 为空,而且还不是最后
            k++;                                    // 试探下一个队列
        if(queue[k].head != -1){                    // 将这个非空序列与上一个非空序列连接起来
            Array[last].next = queue[k].head;
            last = queue[k].tail;                   // 此时最后一个记录为这个序列的尾部记录
```

```
        }
    }
    Array[last].next = -1;                          // 收集完毕
}

template <class Record>
void AddrSort(Record *Array, int n, int first){     // 线性时间整理静态链表,使得数组按下标有序
    int i, j;
    j = first;                                      // j 待处理数据下标,第一次为 first
    Record TempRec;
    for(i=0; i < n-1; i++){                         // 循环 n-1 次,每次处理第 i 大记录
        TempRec = Array[j];                         // 暂存第 i 大的记录 Array[j]
        Array[j] = Array[i];                        // 当前下标 i 的数据存放到 j 位置
        Array[i] = TempRec;                         // 第 i 大记录入位
        Array[i].next = j;                          // 第 i 大的记录的 next 链要保留调换轨迹 j
        j = TempRec.next;                           // j 移动到下一位
        while (j <= i)                              // j 比 i 小,则是轨迹,顺链找到数据
            j = Array[j].next;
    }
}
```

【算法 8.13 结束】

图 8.12 给出了一个基于静态链的基数排序的例子,初始序列为{97,53,88,59,26,41,88',31,22},$n=9$,$d=2$,$r=10$,如图 8.12(a)所示。整个排序过程分别如下:第一趟分配后静态链数组 queue 的结果如图 8.12(b)所示,第一趟收集结果如图 8.12(c)所示。类似的,第二趟分配和收集结果分别如图 8.12(d)和图 8.12(e)所示。两趟分配和收集后,排序过程结束,数组中静态链 next 索引依次为{-1,3,6,2,7,1,0,5,4},第一个下标 first 为 8。再调用函数 AddrSort()顺链整理,按照原数组下标{8,4,7,5,1,3,2,6,0}输出结果{22,26,31,41,53,59,88,88',97}。

AddrSort()函数的代价是 $\Theta(n)$。i 循环 n 次,关键是 while 循环的次数。当发觉第 i 个关键码不在正确位置,则需要调整,而关键码不在原处是因为该关键码曾被置换到后面去了,需要根据 next 链的调换轨迹向后追踪。由于总的置换次数是 $\Theta(n)$,故跟踪次数也为$\Theta(n)$。

根据前面的分析,算法 8.11 和 8.13 的时间代价都是 $\Theta(d(n+r))$。当 r 远小于 n 时,例如整数的基数 $r=10$,字符串的基数 $r=26$,均为常数,这种情况下,可以忽略掉 r,时间代价为 $\Theta(d\cdot n)$,而 d 取决于数字的位数或者字符串的长度。如果 d 相对于数组长度 n 很小的时候,时间代价就为 $\Theta(n)$了。也就是说,当 d 远小于 n,r 远小于 n 或者与 n 相同数量级时,基数排序的时间代价为 $\Theta(n)$。这样看来,基数排序似乎是最快的排序算法。那么基数排序到底是不是最快的排序算法呢?

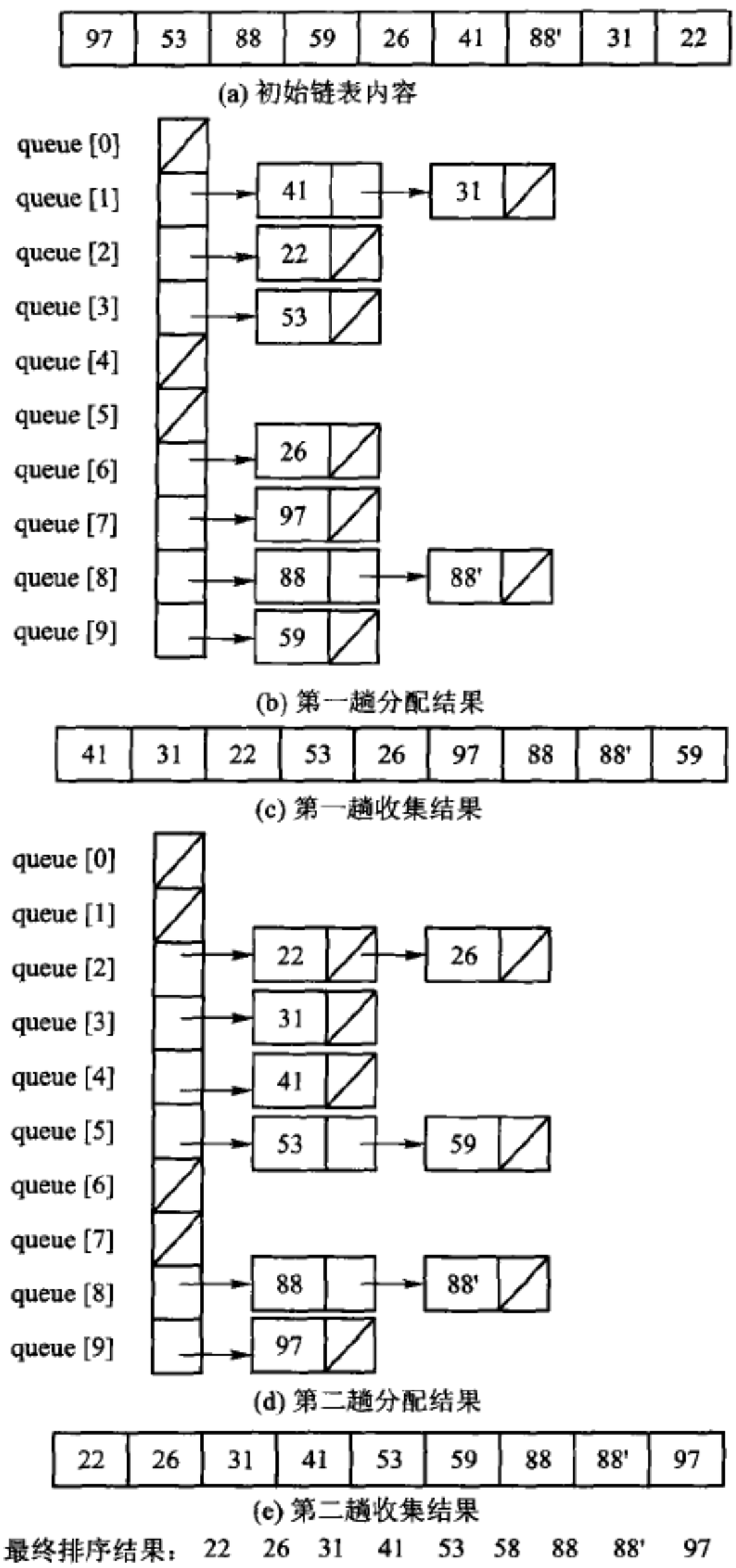


图 8.12 基于静态链的基数排序

现在考虑 n 个互不相同的关键码，此时需要 n 个不同的编码来表示它们。也就是说，$d >= \log_r n$，即 k 在 $\Omega(\log n)$ 中。因此，对 n 个不同的关键码值进行基数排序需要耗用 $\Omega(n \log n)$ 的时间代价。

实际应用中，常用的 32 位计算机，其 int 类型所占的 32 个二进制位数几乎都比 $\log n$ 大。因

此，比较两个较长整数的时间代价为 $\Omega(\log n)$，但是实际上人们认为整数的比较运算在常数时间内完成。

基数排序必须对关键码值进行 $d \approx \log_r n$ 次算术运算。因此，处理 n 个不同关键码时，基数排序实际的时间代价为 $\Theta(n \log n)$。

但有时情况也不容乐观，如果 n 个记录的排序码均不重复，即排序码个数为 n，如果以 r 为基数，则排序码会被拆分为 $\log_r n$ 位，即 $d = \log_r n$，这时 d 就不能再当作常数来处理，此时基数排序的时间代价就变成 $\Theta(n \log_r n)$。例如，如果按照二进制来选择基数 $r = 2$ 来对整数进行基数排序，那么时间代价就会变成 $\Theta(n \log n)$。也就是说，基数排序实质上并不是 $\Theta(n)$ 的算法，与堆排序、快速排序、归并排序同属 $\Theta(n\log n)$ 复杂度量级。

其实，算法 8.13 中取排序码的第 i 位的如下代码还有改进的余地。

```
// 取排序码的第 i 位
for(int a = 0; a < i; a++)
  k = k / r;
k = k % r;
```

对正整数，而且采用 2 的 g 次幂 $r = 2^g$ 为基数，可以利用 C/C++ 的位操作，换成下面的代码

```
k = (((k &(0xFF << (i * g))) >> (i * g)))% r;
```

其中，g 是相应的 bit 位数。先把 0xFF 左移 i * g 位，然后与 k 进行按位与运算，消除了 k 的右边 i * g 位。接着，再把运算结果右移 i * g 位，至此达到了"for(int a = 0; a < i; a++)k = k / r;"的效果，把基数 r 下 k 的第 i 位调整到最右端。最后，模 r 取余数，真正取出基数 r 下 k 的第 i 位。

在表 8.4 同样的测试环境下，改用位运算，1M 数据 r = 16，g = 4 链式基数排序速度为 0.5532，比表中的 0.6570 快出很多。

8.6.3 索引排序

当记录很大时，需要减少移动（或交换）记录的次数，以降低总的排序时间。为此，可以让数组中的每一个元素存储指向该元素记录的指针，在需要移动记录时，只移动指针值（或索引地址）而不移动记录本身。这种排序称为"索引排序"，或称"地址排序"。8.6.2 节中的静态链基数排序实际上也是一种索引排序。

图 8.13(a)是待排序记录（只标出了关键码）及其索引，关键码值为 42 的记录处于关键码值为 5 的记录之前；图 8.13(b)是排序后的情况，关键码值为 5 的记录排在关键码值为 42 的记录之前。在图 8.13(b)中，原始待排数组尚未整理，记录的序被保持在索引数组中。

索引排序虽然需要一些空间来存放指针，但换来了更高的效率，这是典型的用空间换取时间的技巧。需要说明的是，索引排序并不要求在待排序数组中就给出附加空间。可以通过申请新数组，通过下标来进行对应。排序后，需要根据辅助的索引数组来调整原始的待排数组，使其按照下标有序地存放记录。这些整理都应该在 $O(n)$ 时间之内完成。

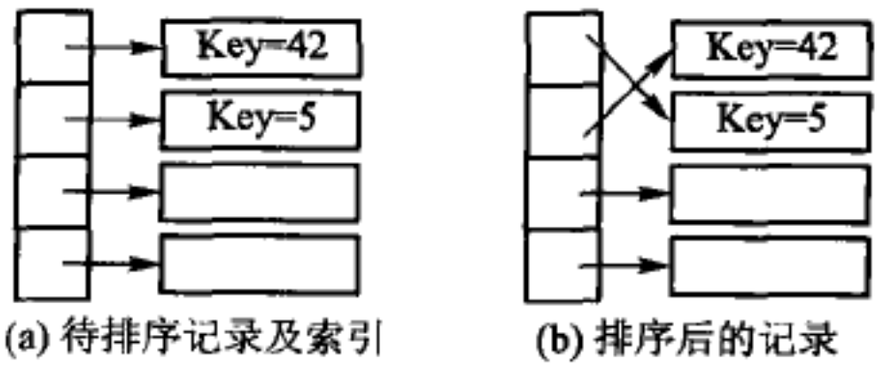


图 8.13 索引排序示意图

除 8.6.2 节链式基数排序那样的 next 链索引结果外，还有表 8.1 所示的两种主要索引形式。索引 1 中的下标 IndexArray1[i]存放的是 Array[i]中数据在排序后应该处于的位置，即结果数组中 Array[IndexArray1[i]]应该等于 Array[i]；索引 2 的结果下标 IndexArray2[i]存放的是 Array[i]中应该摆放的数据位置，即结果数组中 Array[i]应该等于 Array[IndexArray2[i]]。

表 8.1 两种索引排序

原数组下标 i	0	1	2	3	4	5	6	7
记录数组 Array[i]的排序码	29	25	34	64	34'	12	32	45
索引 1 下标 IndexArray1[i]	2	1	4	7	5	0	3	6
索引 2 下标 IndexArray2[i]	5	1	0	6	2	4	7	3
整理后结果数组	12	25	29	32	34	34'	45	64

上述这两种索引排序的处理方法非常相似，本节只讨论后者，读者可自行补充前者的相关算法。

对第 2 种索引 IndexArray2，可以观察到，下标 0 应该放入目前处于下标 5 的排序码 12(的记录)，(5,12) - >(0,12)；同理，(4,34') - >(5,34')；(2,34) - >(4,34)，(0,29) - >(2,29)。这样，下标{0,5,4,1}涉及的元素{29,12,34',34}进行了一轮循环调整后，都已到位。类似地，顺着{1}和{3,6,7}两条索引链，进行两轮循环调整，整个数组就调整完毕。

算法 8.14 实现图 8.13 示意的索引排序算法，核心排序算法是直接插入排序。算法 8.15 从 $0 \sim n-1$ 扫描数组，如果索引值与当前位置不符合，就顺着索引链进行循环调整，直到找到等于当前下标值的索引位置为止。每一轮循环所涉及的记录都调整到位，其索引下标都改为所在位置的下标，即满足“索引值等于下标值”，因此不会参与其他的循环调整。

【算法 8.14】 采用简单插入的索引排序。

```
// Array[ ]为待排序数组,IndexArray[ ]为索引数组,n 为数组长度
template < class Record >
void IndexSort( Record Array[ ], int IndexArray[ ], int n) {
    int i,j;
    for( i =0; i < n; i ++ ) IndexArray[ i ] = i;   // 初始化索引下标
    // 简单插入排序的地址排序过程
    for( i =1; i < n; i ++ )                        // 依次插入第 i 个记录
```

```
        for(j = i; j > 0; j--)                  // 依次比较,发现逆置就交换
            if(Array[IndexArray[j]] < Array[IndexArray[j-1]])
                swap(IndexArray, j, j-1);       // 交换 Array[IndexArray[j]]和 Array[IndexArray[j-1]]
            else break;                         // 此时 i 前面记录已排序
    AdjustRecord(Array, IndexArray, n);
}
```

【算法 8.14 结束】

【算法 8.15】 利用索引数字进行循环调整的算法。

```
// 根据 IndexArray 整理 Array,IndexArray[ ]为索引数组,n 为数组长度
template < class Record >
void AdjustRecord(Record Array[ ],int IndexArray[ ], int n){
    Record TempRec;                         // 只需要一个临时存储空间
    int i, j;
    for(i=0; i < n; i++){                   // 循环 n-1 次,每次处理数组中第 i 个记录
        j=i;                                // j 为临时变量,是循环链中的当前元素
        TempRec = Array[i];                 // 暂存 i 下标中目前的记录
        while(IndexArray[j] != i){          // 如果循环链中索引下标还不是 i,则顺链循环调整
            int k = IndexArray[j];          // k 为链接 j 指向的下标
            Array[j] = Array[k];            // 把 k 下标中的值复制到 j 位,第 j 大元素正确归位
            IndexArray[j] = j;              // 因为是正确归位,索引 j 就是自身
            j = k;                          // j 换到循环链中的下一个,继续处理
        }
        Array[j] = TempRec;     // 最后找到的 j,其索引下标值为 i,因此第 i 大元素正确入位
        IndexArray[j] = j;                  // 因为是正确归位,索引 j 就是自身
    }
}
```

【算法 8.15 结束】

在调整算法 8.15 中,i 下标是无回溯的 for 循环,while 循环中的调整处理也都一次到位,每个元素最多参与一轮调整。因此,整个调整算法的时间代价为 $\Theta(n)$。显然,空间代价为 $\Theta(1)$。

8.7 排序算法的时间代价

8.7.1 简单排序算法的时间代价

直接插入、直接选择和冒泡这 3 种排序算法思想简单且易于实现。表 8.2 分别给出了这 3

种排序算法的时间代价。可以看出，这 3 种排序算法的时间代价都很大，在平均和最差情况下的时间代价均为 $\Theta(n^2)$，不适于对大规模的记录文件进行排序。

关键原因在于插入和冒泡算法都有一个共同缺点：只对相邻的两个记录进行比较和移动。因此，记录只能一步步地向目标位置移动，效率低下。直接选择排序虽然没有这样，但是在选择第 i 小的记录时也是在剩下的记录中逐个地进行线性比较，虽然改进了冒泡排序的交换次数，但比较次数仍然没有降低，因此总体效率依旧很低。

前面 8.2.1 节分析过插入排序的时间代价主要由记录移动到目标位置所需要的步长之和来决定，而步长又由序列中的逆置数决定。由于一个长度为 n 的序列平均有 $n(n-1)/4$ 对逆置，因此任何一种只对相邻记录进行比较的排序算法的平均时间代价都是 $\Theta(n^2)$。

事实上，这些算法之所以效率太低，是因为它们在排序过程中的比较或者移动都是一步步进行的，实际上做了很多重复的操作。因此，如果想彻底从数量级上提高算法效率，就需要摆脱这种逐个对每个记录一步步操作的算法思想。例如，Shell 排序就跨越相邻元素进行分区插入排序而达到了 $\Theta(n^{1.5})$ 的效率。

表 8.2 三种简单排序算法的时间代价

时间代价	直接插入排序			直接选择排序			冒泡排序		
	比较	移动	总代价	比较	移动	总代价	比较	移动	总代价
最佳情况	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n)$	0	$\Theta(n)$
平均情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
最差情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$

8.7.2 排序算法的理论和实验时间

本章介绍了很多种排序。从运行速度上，$\Theta(n\log n)$ 显然要比几种简单的 $\Theta(n^2)$ 算法要好。但是在时间代价同是 $\Theta(n\log n)$ 的排序算法中，排序算法的选择还要受到待排序数组规模、记录大小以及其他特定应用因素的影响。

本章在介绍每种算法的时候，大都对算法的最佳、最差及平均情况进行了分析。表 8.3 总结出这些算法的时间和空间代价，其中 Shell 排序中给出了增量每次除以 3 时的算法开销。

表 8.3 排序算法的理论性能

算法	最大时间	平均时间	最小时间	辅助空间代价	稳定性
直接插入排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$	稳定
冒泡排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$	稳定

续表

算 法	最大时间	平均时间	最小时间	辅助空间代价	稳 定 性
直接选择排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	不稳定
Shell 排序(3)	$\Theta(n^{3/2})$	$\Theta(n^{3/2})$	$\Theta(n^{3/2})$	$\Theta(1)$	不稳定
快速排序	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(\log n)$	不稳定
归并排序	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$	稳定
堆排序	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(1)$	不稳定
桶式排序	$\Theta(n+m)$	$\Theta(n+m)$	$\Theta(n+m)$	$\Theta(n+m)$	稳定
基数排序	$\Theta(d \cdot (n+r))$	$\Theta(d \cdot (n+r))$	$\Theta(d \cdot (n+r))$	$\Theta(n+r)$	稳定

为了更好地了解这些算法的实际性能，下面通过实验来比较算法的性能。利用随机函数来生成一个长度为 n 的整形数据序列，随机函数如 8.16 所示。

【代码 8.16】 随机函数。

```
inline void Randomize() { srand(1); }                  // 设置随机种子
inline int Random(int n) { return rand()%(n); }        // 返回一个 0 ~ n 之间的随机整数值
ELEM *sortarray = new ELEM[n];
for(int i=0; i < n; i++)                               // 产生随机数组
    sortarray[i] = Random(RANGE);                      // RANGE 可以为任意的常数。表 8.4 是把
                                                       // RANGE 定义为 32003 得到的测试数据。
```

【代码 8.16 结束】

对于小规模的数据，需要产生若干个这种规模的随机数组来测试，取其平均时间。代码 8.17 是用于时间测试的函数，首先调用 Settime()启动计时器 tstart 以记录起始的系统时间；关键代码段结束后调用 Gettime()，就会调用系统时钟函数 clock()获得结束时的系统时间；结束时间减去存储在计时器 tstart 中的起始时间，就是关键代码的耗时。

【代码 8.17】 时间测试函数定义。

```
#include <time.h>
#define CLOCKS_PER_SEC 1000          // ANSI 的时钟宏，每秒 1000 次
clock_t tstart = 0;                  // 开始计时时的初值
void Settime() { tstart = clock(); }  // 初始化计时器
// 计算从上次设置 Settime()后流逝的时间
double Gettime() {return(clock() - tstart)/(double)CLOCKS_PER_SEC; }
```

【代码 8.17 结束】

表 8.4 给出了排序算法的测试运行数据，测试数据都是 16 位正整数。分别给出了各种算法在 6 种不同数组规模下随机数据的排序时间，以秒为单位。另外，还给出了对 10 K 的正序和逆序数组进行排序的时间。测试硬件环境为：Intel T2400 1.83 GHz 双核 CPU，1 GB 内存。软件环

境为:Windows XP 操作系统,Visual C++6.0 编译器。

表 8.4 排序算法的实验性能

数组规模	10	100	1 K	10 K	100 K	1 M	10 k 正序	10 k 逆序
直接插入排序	0.00000047	0.000020	0.001782	0.1752	17.917	_____	0.00011	0.35094
选择排序	0.00000110	0.000041	0.002922	0.2778	36.500	_____	0.27781	0.29109
冒泡排序	0.00000160	0.000156	0.015620	1.5617	207.69	_____	0.00006	2.44840
Shell 排序(2)	0.00000156	0.000036	0.000640	0.0109	0.1907	3.0579	0.00156	0.00312
Shell 排序(3)	0.00000078	0.000016	0.000281	0.0038	0.0579	0.8204	0.00125	0.00687
堆排序	0.00000204	0.000027	0.000344	0.0042	0.0532	0.6891	0.00406	0.00375
快速排序	0.00000169	0.000021	0.000266	0.0030	0.0375	0.4782	0.00190	0.00199
优化快排/16	0.00000172	0.000020	0.000265	0.0020	0.0235	0.3610	0.00082	0.00088
优化快排/28	0.00000062	0.000011	0.000141	0.0018	0.0235	0.2594	0.00063	0.00063
归并排序	0.00000219	0.000028	0.000375	0.0045	0.0532	0.5969	0.00364	0.00360
优化归并/16	0.00000063	0.000014	0.000188	0.0030	0.0375	0.4157	0.00203	0.00265
优化归并/28	0.00000062	0.000013	0.000204	0.0027	0.0360	0.4156	0.00172	0.00265
顺序基数/8	0.00000610	0.000049	0.000469	0.0048	0.0481	0.4813	0.00484	0.00469
顺序基数/16	0.00000485	0.000034	0.000329	0.0032	0.0324	0.3266	0.00328	0.00313
链式基数/2	0.00002578	0.000233	0.002297	0.0234	0.2409	3.4844	0.02246	0.02281
链式基数/4	0.00000922	0.000075	0.000719	0.0075	0.0773	1.3750	0.00719	0.00719
链式基数/8	0.00000704	0.000048	0.000466	0.0049	0.0502	0.9953	0.00469	0.00469
链式基数/16	0.00000516	0.000030	0.000266	0.0028	0.0295	0.6570	0.00281	0.00281
链式基数/32	0.00000500	0.000027	0.000235	0.0028	0.0297	0.5406	0.00263	0.00266

从表 8.4 可以看出 $\Theta(n\log n)$ 的几种排序算法的确比插入、选择、冒泡这 3 种 $\Theta(n^2)$ 的算法要快很多,尤其在数据规模比较大的时候。这 3 种简单排序算法中,插入排序的试验时间最好,而冒泡排序的性能最差。因此,插入排序经常被用来与其他排序方法结合使用。例如,本章所讨论的 Shell 排序,另外优化快速排序、优化归并排序都在底层用到了插入排序。

表 8.4 没有给出 $\Theta(n^2)$ 的算法在 1 M 规模下的运行时间,因为这些算法的时间都是随着 n 呈多项式级增长的,相对于后面的那些算法时间相差太大。

在数据规模超过 1 K 时,Shell 排序明显优于 3 个 $\Theta(n^2)$ 的算法。增量为 3 的 Shell 排序比增量为 2 的快很多,二者只有在 10 K 正序时速度比较接近。

值得注意的是,快速排序的确是最快的。比同级别的归并排序和堆排序都要快很多,比其他基数排序也要快,尤其是数组规模较大时更为明显。快速排序采用阈值 28 的优化效果非常好,在实践中应用十分广泛。

归并排序的优化效果对阈值没有快速排序那么敏感,阈值 16 和 28 的优化效果比较接近。

当基数 r 逐渐增大,基数排序时间代价 $\Theta(n\log_r n)$ 也逐渐减小。由于实验记录为整数,因此

顺序基数排序比链式要快。如果记录是比较大的组合类型,应该采用链式存储以减少排序过程中的记录移动,从而提高速度。对于16位整数的情况,可以看到基数为16和32时差别不大。在实际应用中,可以通过调整 r 的值来获得较好的性能。

选择排序、归并排序、基数排序对数据分布情况不敏感。插入排序、冒泡排序、Shell排序都是正序比逆序要快。而且,在正序或基本有序等特殊情况下,插入排序、冒泡排序等简单的算法是最好的。快速排序的正序和逆序速度一样,都比随机分布要快,这是因为算法选取中间点作为轴值,在输入数据为正序或逆序正好平分序列。堆排序逆序比正序稍微快一些,这是因为算法采用最大堆、逆序时已经自然成堆,建堆速度比较快。

ANSI C中提供了qsort快速排序。STL根据不同的需求,提供了不同的排序函数:sort对给定区间所有元素进行排序,stable_sort对给定区间所有元素进行稳定排序,partial_sort对给定区间所有元素部分排序,nth_element找出给定区间的某个位置对应的元素,等等。

STL sort实际上是一种基于快速排序的混合算法,stable_sort采用归并排序,partial_sort采用堆排序。STL中的sort()函数类似于算法8.7,首先采用快速排序,其轴值为三点选中,当长度小于阈值($n<20$)时停止递归,留下分块有序的子序列最后进行整个序列的插入排序。为了避免出现性能退化为 $O(n^2)$ 的情况,STL引入了一个递归计数。当递归深度超过一定阈值(STL设置的阈值为 $2\log n$)时,算法转入到代价为 $O(n\log n)$ 的堆排序partial_sort()函数。在最坏情况下算法也能达到接近 $O(n\log n)$ 的性能,最坏情况下的性能比堆排序要差一些,但是比快速排序要好得多。而STL sort的平均性能与快速排序差不多。

8.7.3 排序问题的下限

直接插入排序、直接交换排序、冒泡排序等算法的平均时间代价是 $\Theta(n^2)$,堆排序、快速排序、归并排序是更加有效的 $\Theta(n\log n)$ 的算法。在某些特殊情况下,基数排序的时间代价接近 $\Theta(n)$,但是从实验数据中也可看到,基数排序在大部分时候远比 $\Theta(n\log n)$ 的快速排序慢得多。快速排序几乎是所有这些算法中最快的排序,那么还有没有更快的排序?

这一小节主要探讨这个问题,分析排序问题本身的时间代价的上限和下限。所谓排序问题的下限,就是解决排序问题所能达到的最佳可能效率,即使尚未设计出算法。而上限则是指已知的最快算法所达到的最佳渐近效率。如果问题的上限与下限相同,那么从渐近分析的意义上说,不可能有更有效的算法,因此没必要再去花费大量时间精力去试图改进。

任何算法的时间代价都不可能小于它的I/O时间,因此没有哪种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。实际上,排序的时间只可能在 $\Omega(n)\sim O(n\log n)$ 之间。

已经知道所有基于比较的排序算法的上限为 $O(n\log n)$,那么它的下限能不否降到 $\Omega(n)$?

首先,来看基于比较的排序到底需要多少次比较。可以用二叉判定树(decision tree)来分析这个问题。判定树每个结点列举当前状态下可能的排列集合,而边表示记录之间的比较,即一个

判断。每个叶结点为一种排列情况，因此一共有 $n!$ 个叶结点。

例如，在图 8.14 中，用判定树来模拟 3 个记录 A、B、C 之间的排序过程。为了简便，假设这 3 个记录互不相等。最初，因为尚未经过任何比较，因此根结点包括这 3 个记录所有可能出现的排列，一共有 $3!=6$ 种可能的排列。

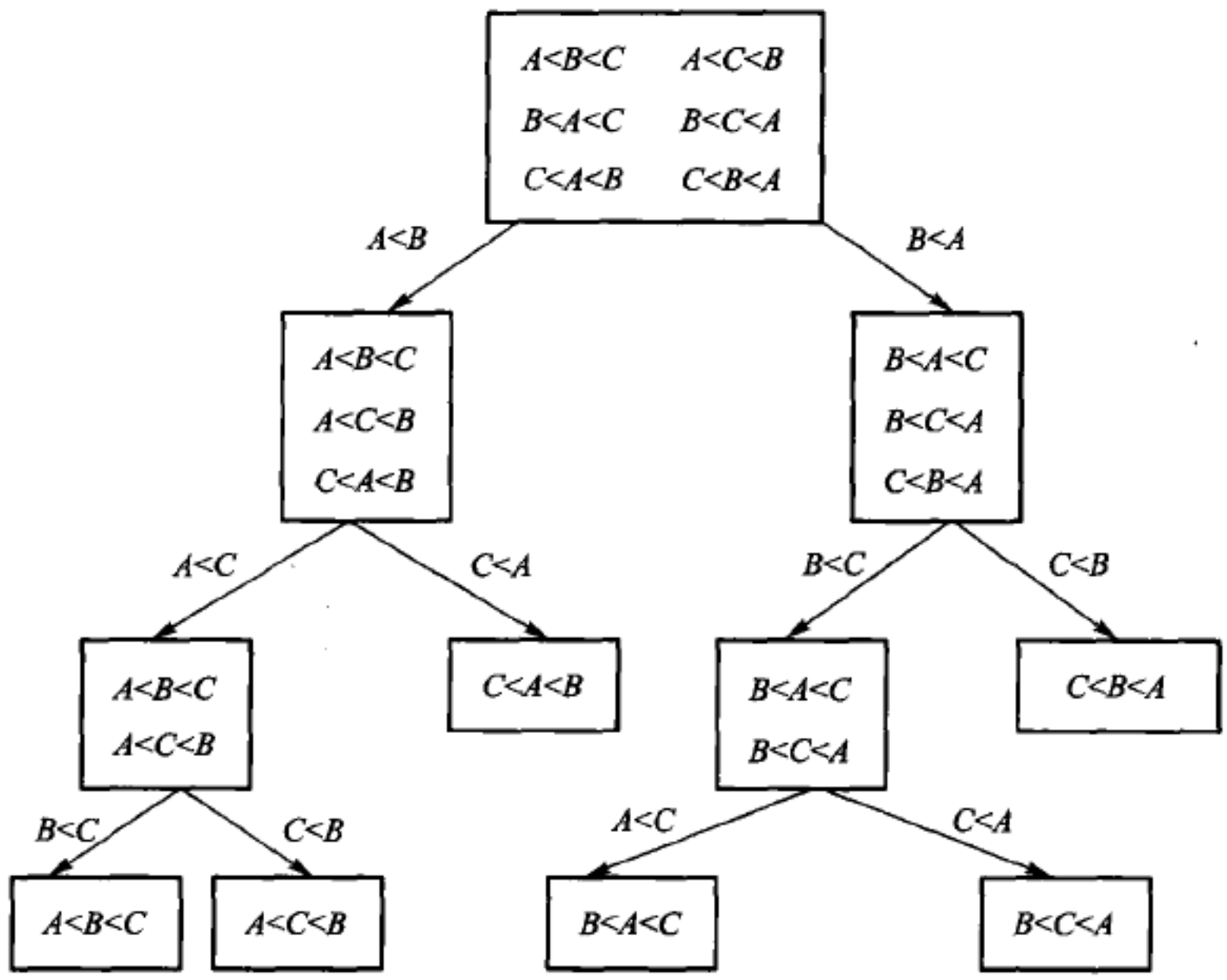


图 8.14 用判定树来模拟基于比较的排序

首先来比较 A 和 B，有两种可能情况。如果 $A<B$，则出现根的左子结点所示的那 3 种排列；如果 $B<A$，就有右子结点所示的另 3 种排列。以左子结点的情况为例，继续比较 A 和 C。如果 $A>C$，此时只可能有一种排列情况，$C<A<B$，到达叶结点，此时排序结果已经确定，排序过程结束；如果 $A<C$，则可能有两种排列 $\{A<B<C, A<C<B\}$，因此还需进一步比较 B 和 C。如果 $B<C$，则只有一种情况满足条件，$A<B<C$，排序结束；如果 $C<B$，则只有 $A<C<B$ 这种情况满足条件，排序结束。

因此，可以看到，基于比较的排序都可以用判定树来表示。不同的排序策略所得到的判定树也各不相同。而判定树中叶结点的最大深度（从根结点到叶结点的最大步长距离）就是排序算法在最差情况下需要的最大比较次数，叶结点的最小深度就是最佳情况下的最小比较次数。例如，在图 8.14 中，最差情况下需要进行三次比较。平均比较次数就是判定树中叶结点的平均深度。

根据二叉树的性质，知道深度为 d 的二叉树最多含有 $L=2^d$ 个叶结点。因此，含有 L 个叶结点的二叉树深度为 $d=\log L$。由于 n 个非重复记录一共有 $n!$ 种排列情况，即对 n 个记录进行基于比较的排序，判定树一共有 $n!$ 个叶结点。因此，这棵判定树的深度为 $\log(n!)$。也就是说，基

于比较的排序算法在最差情况下需要 $\log(n!)$ 次比较。根据以下推理，得到一个结论：基于比较的排序算法需要的比较次数为 $\Omega(n \log n)$。

$$\log(n!) = \log(n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1)$$

$$\log(n!) = \log n + \log(n-1) + \log(n-2) + \cdots + \log 2 + \log 1$$

$$\log(n!) \geqslant \log n + \log(n-1) + \log(n-2) + \cdots + \log(n/2)$$

$$\log(n!) \geqslant \log(n/2) + \log(n/2) + \log(n/2) + \cdots + \log(n/2)$$

$$\log(n!) \geqslant \frac{n}{2}\log(n/2) = \frac{n}{2}(\log n - 1)$$

因此

$$\log(n!) = \Omega(n \log n)$$

既然已经知道在最差情况下任何基于比较的排序算法都需要 $\Omega(n \log n)$ 次比较，因此最差情况时间代价就是 $\Omega(n \log n)$。由于所有基于比较的排序算法都需要 $\Omega(n \log n)$ 的时间开销，因此排序问题本身需要的运行时间为 $\Omega(n \log n)$。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。

因此，最后结论就是任何一种基于比较的排序算法至少需要 $\Theta(n \log n)$ 时间。也就是说，已经没必要也无法再设计出更快的基于比较的排序算法。当然，也可以像基数排序那样考虑不通过比较来实现排序，但实验证明，基数排序的效率反倒不如 $\Theta(n \log n)$ 的快速排序。

本章小结

其实所谓排序就是将线性表序列中的记录按照特定的顺序排列起来。如果存在多个具有相同排序码的记录，经过排序后这些记录的相对次序仍然保持不变，则这种排序算法称为“稳定的”，否则称为“不稳定的”。

作为最常用的算法，排序、检索和索引历来是数据结构讨论的重点问题。检索是直接面向用户的，往往要求比较好的性能，即以较快的检索速度把最相关的检索结果呈现给用户。为了提高检索速度，往往需要预先索引，而索引可能要求事先对数据进行排序。而且为了把最相关的检索结果呈现给用户，也需要把检索结果按照相关性排序。

排序算法最能够体现算法设计和算法分析的魅力，其速度要求非常高，其中内排序主要考虑的是怎样减少关键码之间的比较次数和记录移动次数来提高排序速度；而外排序则考虑外存的特性，尽量减少访外操作，以提高排序速度。内排序算法是《数据结构与算法》课程讨论的重点内容。

本章介绍了一些经典的内排序算法，这些重要排序算法的思想可以总结如下：

(1) 直接插入排序：依次对第 i 个记录进行排序，将当前无序区的第一个记录 Array[i] 插入到有序区 Array[0..i-1] 的合适位置。

(2) Shell 排序:按照某种增量序列,依次操作。在处理第 i 个增量 d 时,把整个待排数组 Array 中距离 d 倍数的那些记录放在同组子序列中,组内进行插入排序;逐渐扩大小序列的规模,而减少小序列个数;最后,所有序列都合并在一个大序列中进行一趟插入排序,从而完成排序。注意,如果最后一个增量不是 1,则需要对整个数列进行一次插入排序。

(3) 直接选择排序:依次从剩余记录中找出第 i 小的记录,直接交换到正确位置(本趟待排子序列的最前端)。

(4) 堆排序:对所有记录建立最大堆(也可以用最小堆),取出堆顶的最大记录,重新建堆。不断重复这一操作,直到堆为空,得到完整的有序序列。

(5) 冒泡排序:依次从剩余记录中找出第 i 小的记录,通过不断比较和交换相邻记录来寻找最小记录,每趟都把找到的最小记录放在本趟待排子序列的最前端。选择排序可以看做是对冒泡排序的一种改进。

(6) 快速排序:首先从待排序序列 Array 中任意选择一个记录 k 作为轴值,然后将剩余的记录分割成左子序列 L 和右子序列 R。L 中包含所有小于或等于轴值 k 的记录,R 中包含所有大于轴值 k 的记录,k 正好到位。原问题分解为子序列 L 和 R 这两个规模更小的同结构子问题,对 L 和 R 子序列递归进行快速排序,直到子序列中只含有 0 或 1 个元素时退出递归。

(7) 归并排序:将序列划分为两个子序列,对子序列递归;反复比较两个有序子序列的最小记录,取较小者出列进入结果序列,合并为一个有序序列。

(8) 桶式排序:扫描一遍序列进行统计计数,然后再统计小于等于编号 i 的个数,给各桶预留适当的空间;从序列尾部往前再一次进行扫描,将具有相同值的记录都分配到同一个桶中;最后,依次按照编号从桶中取出记录,组成一个有序序列。

(9) 基数排序:将排序码拆分为 d 个子码,按照低位优先的顺序进行 d 趟桶式排序。有顺序存储和静态链式存储两种实现方法。

插入排序、冒泡排序、归并排序、分配排序(桶式、基数)都是稳定的排序算法。若要证明一个排序算法是稳定的,则需证明:对于任意两个关键码相同的记录,例如 r_i、r_j,其关键码 $K_i, = K_j$,其序号 $i<j$;排序后,它们的新位置为 i'、j',记录为 r_i'、r_j',其中 $r_i = r_i'$, $r_j = r_j'$,而其相对序号仍保持 $i'<j'$。可以举反例来证明一个排序算法是不稳定的,直接选择排序、堆排序、Shell 排序、快速排序都是不稳定的排序算法。

时间代价在排序算法中居于非常重要的地位,它是衡量排序算法的最重要的指标。排序总时间代价是比较和移动次数之和。根据应用需要,可以分别考察最小、平均、最大时间代价。对于记录比较大的情况,可以采取减少移动记录的策略,例如索引排序法。

直接插入、直接选择、冒泡这 3 种排序算法的时间代价都很大,在平均和最差情况下的时间代价均为 $\Theta(n^2)$。三者之中,插入排序的试验时间最好,而冒泡排序的性能最差。因此,插入排序经常被用来与其他排序方法结合使用。

快速排序、归并排序、堆排序、基数排序都是 $\Theta(n \log n)$ 量级的排序。快速排序的确是最快的,比同级别的排序算法都要快一些,尤其是数组规模较大时更为明显。因此,快速排序在实践

中应用十分广泛。

本章证明了所有基于比较的排序算法的时间代价都是 $\Theta(n \log n)$，这也是排序问题的时间代价。

空间代价考虑排序算法所需要的额外空间，包括编译栈消耗的辅助记录空间。直接插入排序、直接选择排序、冒泡排序、Shell 排序、堆排序的空间代价为 $\Theta(1)$，快速排序的空间代价为 $\Theta(\log n)$，归并排序的空间代价为 $\Theta(n)$，桶式排序的空间代价为 $\Theta(n+m)$，基数排序的空间代价为 $\Theta(n+r)$。如果有特殊的空间限制，则要注意采用辅助空间较小的算法。

在实际应用中，可以根据排序规模（n 的大小）、稳定性要求、待排数组有序状况、排序码的组合情况等来选择合适的排序算法，或者对这些排序算法进行适当的组合。可以遵循以下一些原则：

（1）当待排序的关键码序列基本有序时，直接插入排序最快，冒泡排序速度也较快。

（2）归并排序对待排序关键码的初始排列不敏感，因此排序速度比较稳定。

（3）若待排序的记录个数 n 较小时，可采用直接插入或直接选择排序。

（4）若 n 较大时，则应采用时间代价为 $O(n \log n)$ 的快速排序、堆排序、归并排序或基数排序方法。

（5）当 n 较大，且输入顺序比较随机（即杂乱无序）时，如果没有稳定性要求，则采用快速排序效果最好。

（6）当 n 很大且关键码位数较少时，采用静态链的基数排序效果比较好。

习 题

1. 试构造 5 个整数元素进行排序且最多只用 7 次比较的算法思想。

2. 写一个处理整数排序码的插入排序算法。算法要满足以下条件：输入数据存放于栈中（不是存放在数组中），并且程序中只许用一定的整数及栈。算法结束时排序结果放在栈中，栈顶元素最小。最差情况下，算法的执行时间为 $O(n^2)$。

3. 试给出采用增量序列 $\{2^k-1, 2^{k-1}-1, \cdots, 7, 3, 1\}$ 的 Shell 排序的时间代价为 $\Theta(n^{3/2})$ 的推理过程。

4. 采用增量序列 $\{2^k-1, 2^{k-1}-1, \cdots, 7, 3, 1\}$ 重新对图 8.5 中的例子进行排序，试写出排序各阶段时的数组状态，与原来的排序过程进行比较。

5. 直接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，即有可能带来不必要的交换。

（1）改进算法使之没有不必要的交换。

（2）这种改进能加快速度吗？试说明理由。

（3）编程验证一下原始排序算法和改进算法的运行时间，哪个算法实际上更快？

6. 考虑这样一种计数排序：对每个待排序记录，扫描整个序列统计比它小的记录个数 c，则 c 就是它在序列中的正确位置。值得注意的是，计数排序只适用于不存在重复记录的情况。编程实现这种算法，并分析其时间代价，与直接选择排序进行比较。

7. 在堆排序中采用的堆是基于二叉树的，因此时间代价为 $\Theta(n \log n)$。如果采用基于三叉树的堆来实现堆排序，时间代价是否会变成 $\Theta(n \log_3 n)$？试编写出基于三叉树的堆排序算法，并分析时间代价。

8. 冒泡排序函数中有如下循环

```
for(int j=n-1; j>=i; j--);
```

考虑一下将其换成以下语句的影响：

```
for(int j=n-1; j>0; j--)
```

新的算法是否还能正常执行，这种改变是否会影响到程序的近似复杂性，此改变对时间代价有何影响？

9. 冒泡排序的一个变种就是所谓的鸡尾酒混合排序：与本章中介绍的冒泡排序一样，它对数组总共也要进行 $n-1$ 趟排序。但是，相邻两趟的冒泡方向是相反的，例如第一趟排序将最大的记录起跑到数组的底部，而在第二趟排序中将最小的记录冒泡到数组的顶部。试编程实现该算法。

10. 在冒泡排序过程中，什么情况下排序码会朝向与排序相反的方向移动，试举例说明。在快速排序过程中有这种现象吗？

提示：排序相反的方向是某个元素从它现在的位置向其最终的位置移动的方向。例如，该元素原来在位置3，最终排序位置是5。那么如果它移动到位置4，则与排序方向同向移动；如果它移动到位置2，则与排序方向反向移动。如果移到了位置7，这次移动还算同向移动，但下次该排序方向就是从7向位置5的移动。

11. 通常，快速排序是将数组划分成两部分，当数组中有大量重复记录时，试将快速排序改为三分数组（将数组划分成大于，等于和小于轴值三部分），并讨论算法对包含 d 个不同值的长度为 N 的数组的效率。

12. 本章中算法8.6中给出的快速排序是通过递归来实现的。

(1) 试用栈来代替递归，减少函数调用次数，优化该算法。

(2) 在最差情况下栈有多深？

(3) 怎样组织递归调用的顺序可以让栈的深度减到最小，可以减为多少？

(4) 可以用队列，而不用栈来消除快速排序中的递归吗？

13. 一个非零数列中有些为正数，有些为负数。试编写一个算法，重新排列这些非零实数，使得负数排在前面，正数排在后面。

提示：参考快速排序的分割算法。

14. 设有一个仅由红、白、蓝3种颜色的条块（每个色块除了颜色外，其他性质相同）组成的序列，各种色块的个数是随机的，但3种颜色色块的总数为 n。试编写一个时间复杂度为 $\Theta(n)$ 的算法，使用辅助空间最少的算法，使得这些条块按照红、白、蓝的顺序排好，即排成荷兰国旗图案（整个图案由三大色块组成，第一块是红色，第二块是白色，第三块是蓝色）。试分析你的算法的时间和空间代价。

15. 定义"中位数"长度为 n 的序列的第 $\lceil (n+1)/2 \rceil$ 个（取上整）或第 $\lfloor (n+1)/2 \rfloor$ 个（取下整）（若对该序列排序，就是排在 $\lceil (n+1)/2 \rceil$ 或第 $\lfloor (n+1)/2 \rfloor$ 位置的值。注意，当 n 是偶数时中位数有两个）。试设计一个 $\Theta(n)$ 的算法求给定序列的中位数。

16. 设 A、B 是长为 n 的数表，已经按照非降顺序排好。如果将这 $2n$ 个数全体排序，则处于第 n 个位置的数称为中位数。设计一个最坏情况下 $O(\log n)$ 时间的算法求 A 和 B 的中位数。

(1) 描述算法的设计思想。

(2) 证明算法的时间复杂性。

17. 为了从由 n 个元素构成的未排序序列 $S=\{s_1, s_2, \cdots, s_n\}$ 中选择出第 k 个最大的元素，考虑以下算法：

(a) 如果 $n \leqslant 5$，对 S 排序且直接选出第 k 个最大元素。

(b) 否则 $n>5$:把序列 S 分割成长度为5的子序列。一般而言,将有$\lfloor n/5 \rfloor$个长度为5的子序列,一个长度为$(n\%5)$的子序列。

(c) 对长度为5的子序列用任意一种方法进行排序。

(d) 对子序列排序后从每个子序列中选取一个中位数,形成一个含$\lfloor n/5 \rfloor$个中位数的序列 $M=\{m_1,m_2,\cdots,m_{n/5}\}$。

(e) 递归重复这一选择过程,找出序列 M 中的中位数,设 m 是序列 M 的中位数。

(f) 将序列 S 分割成3个子序列$\{L, E, G\}$,使得序列 L 中的元素都比 m 小,E 中的元素都等于 m,G 中的元素都比 m 大。

(g) 如果 $k\leqslant|L|$,递归重复这个过程选择 L 中的第 k 个最大元素;如果$|L|<k<=|L|+|E|$,结果就是 m;否则,选择 G 中的第$(k-(|L|+|E|))$个最大元素。

(1) 此算法的运行时间是多少?

(2) 如果用这个算法选择轴值,试证明快速排序的最差运行时间是 $\Theta(n\log n)$。

18. 本章中介绍的是两路归并算法,如果是4路或者8路归并,应该如何改写?试分析改写后的算法代价,并与原来的算法进行比较。

19. 有人说:"由于归并排序的时间代价为 $\Theta(n\log n)$,插入排序的时间代价为 $\Theta(n^2)$,因此当对256个数字进行排序时,归并排序的速度将是插入排序的64倍!"试问这种说法是否正确?并说明理由。

20. 本章中算法8.8中给出的归并排序是通过递归来实现的。

(1) 试用栈来代替递归,编写一个非递归的归并排序算法。

(2) 不用栈、也不用队列,编写一个非递归的归并排序算法。

21. 本章中介绍的归并函数中,需要用到一个长度为 n 的辅助数组来保存合并后的记录。试对该函数进行改进,在不增加算法的时间代价前提下尽量减小空间代价。

22. 编写一种对扑克牌进行排序的基数排序。花色为高位排序码,面值为低位排序码,大小顺序为:方块<梅花<红桃<黑桃,2<3<4<5<6<7<8<9<10<J<Q<K<A。

23. 本章介绍了很多种排序算法,分别列出哪些算法具有以下特性:

(1) 不需要访问那些已排好序的记录。

(2) 在整个排序过程中,算法每次都要访问序列中的所有记录。

24. 下面的各个操作中,哪一个最适于先进行排序处理?对于这些操作,分别简短地描述一个实现算法,并给出算法的渐近复杂性。

(1) 找最小值。

(2) 找最大值。

(3) 计算算术平均值。

(4) 找中位数。

(5) 找出现次数最多的值。

25. 在下列情况下,你会选择哪种排序算法?并说明理由。

(1) 需要对1000个数字进行排序,程序只需运行一次。

(2) 只需要对50个数字进行排序,程序只需运行一次。

(3) 在编写一个很庞大的程序中需要编写一个对5个记录进行排序的函数,而且该函数将被反复调用很多次。

(4) 需要对1000个大型的记录进行排序,记录本身存储在外存中,在内存中只保存了所有记录的排序码。排序码之间的比较非常快,但移动代价很大,因为一旦移动一个排序码,相应的外存中的记录也要移动,将涉及上百个磁盘块的移动。

(5) 在图书馆里计算机类书籍区一共有12列书架,书架上的书本来都是按照编目号排列好的,但其中有些书被读者放错了地方,但通常不会偏离超过一个书架。试设计一个算法将这些书重新放回正确的位置。

(6) 需要将500张随机排列的图书卡片按照字母顺序排好序。

(7) 已知一个包含了5000个单词的列表已按字母顺序排好序,需要再进行一次检查,确保所有单词已经排好序。

26. 某整型数组 A 的10个元素值依次为{6,2,9,7,3,8,4,5,0,1}用下列各排序方法,将 A 中的元素由小到大排序。

(1) 取第一个元素值6作为分割数,试写出快速排序第一次分隔后 A 中的结果。

(2) 用堆排序,试写出将第一个选出的数据放在 A 的最后位置上,将 A 调整成堆后的 A 中结果。

(3) 用基数为3的基数排序法,试写出第一次分配和收集后 A 中的结果。

27. 已知一组元素的排序码为(46,74,16,53,14,26,40,53',86,65,27,34)。

(1) 利用直接插入排序的方法写出每次向前面有序表插入一个元素后的排列结果。

(2) 利用直接选择排序方法写出每次选择和交换后的排列结果。

(3) 利用最大堆排序的方法写出在构成初始堆和利用堆排序的过程中,每次过筛(代码5.11中的函数Sift-Down)运算后的排列结果,并画出初始堆所对应的完全二叉树。

(4) 采用快速排序,每次都取子序列的最左元素为轴值,写出每一层划分后的排列结果,并画出由此快速排序得到如图8.6所示的二叉搜索树。

(5) 利用归并排序的方法写出每一趟二路归并排序后的结果。

28. 如果只想在一个有 n 个元素的任意序列中得到其中最小的第 $k(k << n)$ 个元素之前的部分排序序列,而且这个 n 是比较小的值。采用什么排序方法能得到比用堆排序的比较次数更少?例如,有这样一个序列{503,017,512,908,170,897,275,653,612,154,509,612*,677,765,094},要得到其第4个元素之前的部分有序序列{017,094,154,170},用所选择的排序算法(非堆排序)实现时,要执行多少次比较?若用堆排序算法,要执行多少次比较?

29. 设 $S_1, S_2, \cdots S_k$ 是 k 个集合,每个集合均由 n 个实数所组成。试设计一个算法将所有的和 $x_1 + x_2 + \cdots + x_k$(其中,$x_i \in S_i$, $i = 1,2,\cdots,k$)从小到大排序,并分析算法的时间代价。

30. 设 k 为给定常数,$A_1 = <x_1, x_2, \cdots, x_k>$, $A_2 = <y_1, y_2, \cdots, y_k>$ 是 k 维正整数序列,如果存在 $x_1, x_2, \cdots, x_k$ 的排列 $x_1', x_2', \cdots, x_k'$ 满足 $x_1' < y_1, x_2' < y_2, \cdots, x_k' < y_k$,则称 A_1 嵌入 A_2,记作 $A_1 < A_2$。现有 n 个 k 维序列 $A_1, A_2, \cdots, A_k$ 构成集合 B。

(1) 说明 B 上的嵌入关系有哪些性质。

(2) 设计一个算法求使得 j 最大的 $A_{i1}, A_{i2}, \cdots, A_{ij}$,并且 $A_{i1} < A_{i2} < \cdots < A_{ij}$。

(3) 说明算法的正确性并分析算法在最坏情况下的复杂性。

31. 像静态链基数排序那样,数组元素增加一个next域逻辑链表,对这个静态链表实现插入排序(只修改next静态指针,不移动元素),则称为"表插入排序"。

(1) 实现这个表插入排序算法,算法执行后,这个长度为 n 的数组由其内部的静态链链指针next(数组内部加入一个下标域)而串成一个已排序序列(不减的顺序)。

index		1	2	3	4	5	6	7	8
key		49	38	65	97	76	13	27	52
next	6	8	1	5	0	4	7	2	3

(2) 设计一个只使用一个辅助结点的算法，在 $\Theta(n)$ 时间内，把这个由静态指针 next 串成的有序数组，整理成为从数组的 $0 \sim n-1$ 位置存放的不减的序列。

32. 考虑由 3 个不同记录组成的序列 $\{A, B, C\}$，试分别画出表示下列排序算法的二叉判定树：直接插入排序、直接选择排序、冒泡排序。

33. 设计一个算法，将数组 $A(0..n-1)$ 中的元素循环右移 k 位，假设原数组序列为 $a_0, a_1, \cdots, a_{n-2}, a_{n-1}$；移动后的序列为 $a_{n-k}, a_{n-k+1}, \cdots, a_0, a_1, \cdots, a_{n-k-1}$。要求只用一个元素大小的附加存储，元素移动或交换次数为 $O(n)$。

上 机 题

1. 编程实现下述算法，并与教材上的算法进行时间效率比较，然后用理论分析支持你的试验结论。

(1)教材中是用插入排序算法线性搜索来寻找正确位置的。在插入第 i 个记录时，前面的记录已经有序，因此可以考虑用二分法查找第 i 个记录的正确位置。这种算法称为二分法插入排序(binary insert sorting)。

(2)交换式插入排序(swap insert sorting)改写教材中每一次插入时的操作，当插入第 i 个记录时，如果该记录比它前面的那个记录小向，则交换该记录与其前面的那个记录，即将该记录向前移动一个位置，然后继续向前比较，直到遇到一个小于或等于它的记录，本次插入才完成。

2. 奇偶交换排序是另一种交换排序。它的第一趟对序列中的所有奇数项 i 进行扫描，第二趟对序列中的所有偶数项 i 进行扫描。若 A[i] > A[i+1]，则交换它们。第三趟有对所有的奇数项，第四趟对所有的偶数项，…，如此反复，直到整个序列全部排好序为止。

(1)这种排序方法结束的条件是什么？

(2)写出奇偶交换排序的算法，并实验测试其时间代价。

(3)当待排序排序码序列的初始排列是从小到大有序，或从大到小有序时，在奇偶交换排序过程中的排序码比较次数是多少？分析奇偶交换排序的平均时间代价。

3. 改写算法 8.8 中的快速排序算法，采用一种不同的轴值选择方法：选择序列中所有记录的平均值作为轴值。试设计并实现一个 MeanPivotQuickSort 类，使其能把要排列的序列分割成比平均值大和比平均值小的元素。注意：需要修改原来的分割函数，因为现在轴值可能不是序列中的元素。试编程比较两种算法的实际运行时间。

4. 试写一组英文单词按字典序排序的基数排序算法。单词区分大小写，设最大的单词有 n 个字母。

本章中介绍的排序算法大都是基于数组实现的，试采用链式存储来改写本章的排序算法，并试验比较算法的数组实现和链式实现效率。此时，可以发现大部分算法都能用链表来实现，但有些特殊算法不适于链式存储，试分析为什么这些算法不适合链式存储。

第9章　文件管理和外排序

前面章节讨论的数据结构基本上都存储在计算机内存中，也称为内部数据结构。但是，内存是一种有限的存储资源，一些大规模的数据不能完全存放于内存，而且存放在内存的数据结构随着程序的结束而消失。为了克服这些问题，需要研究存放在外部存储器上的数据结构。

对于存储在外部存储器上待排序的大文件，排序的过程需要多次进行内存和外存之间的数据交换才能得以进行。本章将主要介绍文件管理和外部排序的相关内容。在介绍具体内容之前，首先需要了解一下计算机存储设备的特点。

9.1　主存储器和外存储器

计算机存储器是用来存储程序和数据的部件，主要有两种：主存储器（primary memory 或 main memory，简称内存或主存）和外存储器（peripheral storage 或 secondary storage，简称外存）。常见的主存储器包括随机访问存储器（random access memory，即 RAM）、高速缓存（cache）和视频存储器（video memory）；外存储器则指硬盘、软盘和磁带等设备。主存储器和外存储器都是计算机系统不可或缺的组成部分，同时它们也有很多不同之处。

内存在主板上，由一系列连续编址的内存单元组成，其物理实质是一组或多组具备数据输入/输出和数据存储功能的集成电路。内存中存放的是当前正在使用（即执行中）的数据和程序，并且只用于暂时存放程序和数据。一旦断电，存放在内存中的程序和数据就会丢失。CPU 直接与内存沟通，对存储在内存地址的数据进行存取时，所需要的时间可以看做是一个很小的常数。内存具有访问速度快、造价高、存储容量小和断电后丢失数据等特点。

随着计算机运算速度的提高，计算机所配置的内存容量也不断增加。与此同时，用户数据量的急剧膨胀则需要更加廉价的外存。外存具有价格低、容量大等特点。外存种类很多，前面提到的磁盘是随机存取设备，磁带是顺序存取设备。

外存和内存相比最主要的差别就是存取速度。现在一般的内存存取时间的单位是 ns（$1\ ns = 10^{-9}\ s$），而外存一次存取时间则以 ms（$1\ ms = 10^{-3}\ s$）或 s 为数量级。与内存的速度相比，外存的存取速度就显得太慢。因此，牵扯到外存的计算机程序应当尽量减少外存的存取次数，从而减少程序的执行时间。

外存存取速度慢的一个原因是每次对外存的存取都要分成定位和实际存取两个阶段。以磁盘的一次数据存取为例，操作系统给磁盘发出存取指令后，磁盘驱动器需要把磁头移动到数据所在的磁道，

然后等待磁盘旋转到数据所在的扇区,数据存取位置移动到磁头下的时候才能进行读/写操作。磁盘执行读/写前的定位阶段平均需要耗时几毫秒或几十毫秒,与数据读/写速度相比要慢得多。

为了解决存取速度的问题,外存上的数据通常采用分块存取的方式。外存空间被划分成长度固定的存储块,称为页(page),每一页包含一定数量的数据单元。作为外存的基本存储单位,数据以页块为单位进行存取,这样可以减少外存的定位次数,从而减少外存读/写的时间耗费。

在实际应用中,顺序存取外存数据的情况非常广泛,缓冲方式特别适合需要顺序存取外存的情况。一次外存存取把一个或几个页块的内容读入内存的缓冲区,增加即将存取数据在内存中的可能性,而避免临时读取外存数据,能够大幅提高效率。

此外,外存与内存相比还有两大优点:信息不易失和便携性。其中,外存中信息的非易失性指存储在外存上的信息在断电以后不会被清除掉,是永久保存的;外存的便携性是指像软盘、CD-ROM、移动硬盘、U盘等外存设备可以方便携带,使得信息传递更加方便。

9.2 文件的组织和管理

文件(file)是存储在外存上的数据结构,是由大量性质相同的记录(record)组成的集合。所谓记录,就是具有独立逻辑意义的数据块,是文件的基本数据单位。最简单的记录可以是字符或者二进制序列,复杂的记录通常可以由若干字段或域(field)的数据项组成。

按照记录的类型,文件可以分成两种:操作系统的文件和数据库文件。操作系统的文件是一组连续的字符序列,这种序列没有明显的结构。用户也可以将文件中的信息划分成若干个逻辑记录,以便存取和使用。数据库文件是有结构的记录集合,其中每一条记录都由一个或多个数据项组成,而每个数据项是不可再分的基本数据单元。表9.1所示为一个数据库文件,每个学生的信息组成一个记录,每一条记录由6个数据项构成。

表9.1 学生登记表

姓名	学号	性别	出生年月	所在院系	入学时间
贾由	00646125	男	1988.5	元培	2006.9.1
王子琪	00648139	女	1988.2	信科	2006.9.1
吴珂	00648230	男	1988.3	信科	2006.9.1
陈醒	00648308	男	1989.7	信科	2006.9.1
…	…	…	…	…	…

按照记录信息长度,文件可以分成定长文件和不定长文件。如果文件中每一条记录均含有相同的信息长度,则这种文件称为定长文件。若文件中的记录不是相等长度的,则称为不定长文件。通常,定长文件处理起来比较方便。

按照关键码(key)的个数,文件可以分成单关键码文件和多关键码文件。单关键码文件是指文件的记录中只有一个标识关键码。而在多关键码记录文件中,记录除了有一个主关键码以外还允许有若干个次关键码。

文件的操作往往以记录为单位进行,基本操作包括顺序读取记录、添加新记录、通过一系列写入记录的操作建立文件等。此外,还有与记录内容有关的操作,例如按照检索符合某些条件的修改、删除相应记录等。

文件的操作有实时和批量两种处理方式。实时操作要求有较短的应答响应时间,在接受指令后尽可能快地完成检索或修改任务。而批量的文件处理则允许较长反馈时间,用户可以根据需求选择不同的文件处理方式。

9.2.1 文件组织

文件系统是计算机操作系统的重要组成部分,可完成文件的组织和管理。从用户的观点出发,需要一种清晰而使用方便的逻辑文件结构,即文件的逻辑组织。从系统实现的观点出发,需要一种在外存上有效存储的物理文件结构,即文件的物理组织。操作系统以文件方式组织数据,完成文件的逻辑结构和外存的物理结构之间的转化,而用户可以从这一过程中解放出来,能够按照逻辑组织有效利用存储空间并减少文件的存取时间。

文件逻辑组织有以下 3 种形式:顺序结构的定长记录、顺序结构的变长记录和按关键码存取的记录。

考虑到实际中的不同需要,文件的物理结构可以有多种多样的组织方式。下面将介绍几种常见的物理组织结构:

(1) 顺序文件:计算机中最早使用的一种文件结构。顺序文件根据文件中记录的逻辑顺序,依次把记录分配到连续的物理块中,即用物理上的顺序存储来实现文件的逻辑次序。在这种情况下,顺序文件中物理记录的顺序和逻辑记录的顺序是一致的。

顺序文件按照记录的顺序逐个存取,新插入记录也只能加在文件的末尾。顺序文件连续存取速度快,适合于顺序存取、批量修改等情况,但不适用于要求即时响应的应用。

磁带就是一种典型的顺序存储设备。对于文件数据量特别大而记录变化不大的应用,通常采用磁带进行批量处理。

(2) 散列文件:应用散列技术进行文件的组织。其基本方法是:存放散列文件中的记录时,根据记录的关键码值经过散列函数确定该记录的地址,从而实现直接存取。散列文件的记录之间不是连续存储的。

散列文件这种组织方式的优点是文件适合随机存取,记录间不需要进行排序,并且记录的插入、删除等操作方便。但是,这种组织方式不能顺序存取文件而只能利用关键码随机存取。结构化的顺序文件不适合用散列文件存取。

(3) 索引文件:包括文件数据和索引表两部分。其中,文件中的数据还可以按照顺序文件的

组织方式进行存储。

索引表是系统自动生成的。不论文件中的记录是否按照关键码有序排列，索引表中的索引项总是按照关键码顺序排列。如果数据文件也是按照关键码有序排列，那么这种索引文件称为索引顺序文件，否则称为索引非顺序文件。

索引文件的检索方式可以分成两步执行：先查找索引表，如果在索引表中存在这条记录，则根据索引项的指示存取外存中的这条记录，否则不存在该记录。当文件记录数很大时，可以建立索引表的索引，称为多级索引。第11章将详细讨论B/B+树等索引文件结构。

(4) 倒排文件：根据关键码以外的数据项（即"属性"）建立的索引，每个索引项包括一个属性值和具有该属性值的各记录的地址。倒排索引有利于根据属性来查找记录。带有倒排索引的文件称为倒排索引文件，简称倒排文件。第11章将详细讨论倒排索引文件结构。

9.2.2 C++的流文件

C++语言以"流"的概念直接支持二进制序列文件结构。本教材算法中都是用流操作处理数据对象的格式化输入/输出。C++流的另一个主要用途是作为文件存取机制。

文件流是以外存文件为输入/输出对象的数据流。输入文件流是从外存文件流向内存的数据，输出文件流是从内存流向外存文件的数据。每一个文件流都有一个内存缓冲区与之对应。

文件流与文件不是同一个概念，文件流不是由若干个文件组成的流。文件流本身不是文件，而只是以文件为输入/输出对象的流。程序员若要对磁盘文件进行输入/输出，必须通过流文件来实现。

C++提供了多种机制操作二进制文件。在C++的I/O类库中定义了几种文件类，专门用于对磁盘文件的输入/输出操作。标准输入/输出流类包括istream、ostream和iostream类，其中istream是通用输入流和其他输入流的基类，支持输入；ostream是通用输出流和其他输出流的基类，支持输出；iostream是通用输入/输出流和其他输入/输出流的基类，支持输入/输出。此外，还有3个用于文件操作的文件类：

(1) ifstream类：从istream类派生，用来支持从磁盘文件的输入。

(2) ofstream类：从ostream类派生，用来支持向磁盘文件的输出。

(3) fstream类：从iostream类派生，用来支持对磁盘文件的输入/输出。

其中，最常用的一种机制是fstream类。下面是fstream类的一些主要成员函数，利用这些函数可以操作可随机存取的磁盘文件中的信息。这些函数构成了文件的ADT。

```
#include <fstream.h>                                  // fstream = ifstream + ofstream
void fstream::open(char * name, openmode mode);       // 打开文件进行处理
fstream::read(char * ptr, int numbytes);              // 从文件当前位置读入一些字节
fstream::write(char * ptr, int numbtyes);             // 向文件当前位置写入一些字节
// seekg 和 seekp:在文件中移动当前位置,以便在文件中的任何位置读出或写入字节
fstream::seekg(int pos);                              // 输入时用于设置读取位置
```

```
fstream::seekg(int pos, ios::curr);
fstream::seekp(int pos);                    // 设置输出时的写入位置
fstream::seekp(int pos, ios::end);
void fstream::close( );                     // 处理结束后关闭文件
```

C++以流对象进行输入/输出。首先,必须定义针对磁盘文件的输入(或输出)文件流类的对象,通过文件流对象将数据从内存输出到磁盘文件或者将数据从磁盘文件输入到内存。cin 和 cout 是标准设备的输入/输出流对象,事先已在 iostream.h 中定义,不需要用户自行定义。

文件常见的 3 个基本操作:把文件指针设置到指定位置、从文件的当前位置读取字节、向文件中的当前位置写入字节,都是围绕文件指针进行的。程序员对磁盘文件进行输入/输出时,都需要管理标志文件当前位置的文件指针。

关于 C++流文件的相关知识读者可以阅读有关 C++语言的教材,这里不再赘述。

9.3 外 排 序

第 8 章介绍的内排序算法都假定被排序的数据完全存放在内存中。如果数据存放在外存文件中,则需要考虑外存的特点。采用外存文件排序技术,简称外排序(external sort)。

外排序处理的数据对象非常多,而且在内存不能一次处理完成。因此,需要根据内存的大小,将外存中的数据文件划分成若干段,每次把其中一段读入内存并用内排序方法进行排序。这些已排序的段或有序的子文件称为顺串或归并段(run),顺串需要重新写回外存等待将来处理,让出的内存空间继续处理其他文件段。外排序通常由两个相对独立的阶段组成:

(1) 文件形成尽可能长的初始顺串。

(2) 逐趟归并顺串,最后形成对整个数据文件的排序文件。

外排序所需要的时间由三部分组成:内部排序所需要的时间、外存信息读/写所需要的时间和内部归并所需要的时间。外排序必须不断地在内存与外存之间传送数据,而外存的读/写速度与内存相比要慢得多。因此,减少外存信息的读/写次数是提高外部排序效率的关键。

对同一个文件而言,进行外排序所需读/写外存的次数与归并趟数有关。假设有 m 个初始顺串,每次对 k 个顺串进行归并,归并趟数为 $[\log_k m]$。此时,为了减少归并趟数,可以从两个方面着手:一是减少初始顺串的个数 m,二是增加归并的顺串数量 k。

本节介绍几种外排序的方法。其中,"置换选择"算法在扫描一遍的前提下,能够生成更长的初始顺串,从而减少了初始顺串的个数。"多路归并"方法通过增加归并顺串的个数来减少对数据的扫描趟数。

9.3.1 置换选择排序

置换选择排序(replacement selection sorting)是在堆排序的基础上演化而来的。为了提高处

理速度,需要借助一个输入缓冲区和一个输出缓冲区。算法的处理过程为:从输入文件读取一定数量的记录进入输入缓冲区;然后,向内存工作区放入待排序记录并进行排序;记录被处理后,写到输出缓冲区;当输出缓冲区写满时,把整个缓冲区写回到外存文件。当输入缓冲区为空时,再次从外存文件中读取下一块记录。此工作过程如图 9.1 所示。

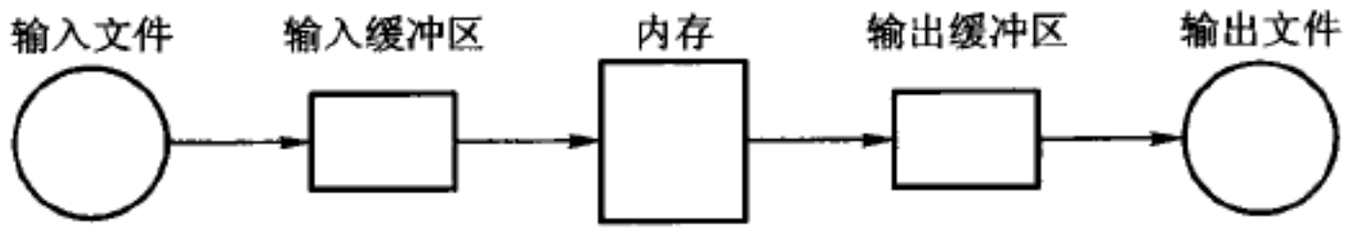
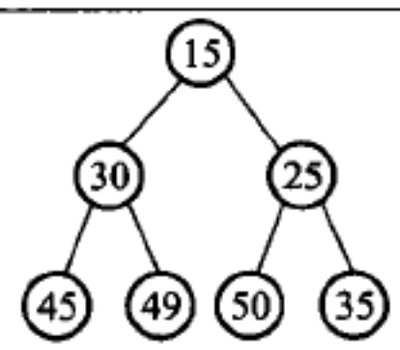
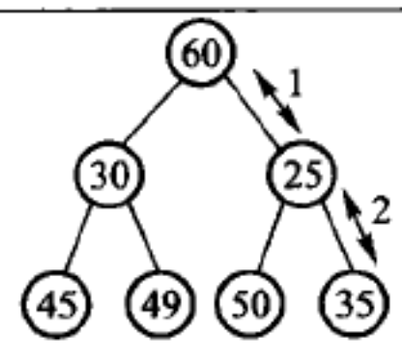
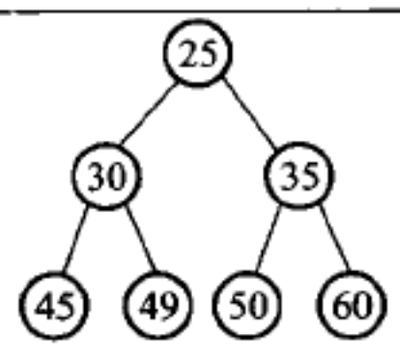


图 9.1 置换选择算法流程

置换选择方法的核心部分是在内存中使用大小等于记录个数 M 的最小堆来进行排序。初始时,首先从输入文件缓冲区中读入 M 个记录到内存工作区,建立大小为 M 的最小堆。当前最小堆中的堆顶元素就是第一个顺串的第一个记录,输出堆顶记录到输出缓冲区中。然后,从输入缓冲区中输入下一个记录。若该记录的关键码值不小于刚输出记录的关键码值,则将其作为堆顶记录,并调整当前堆;若该记录的关键码值小于刚输出的关键码值,由于希望得到更长的顺串,则用当前堆的堆底记录取代堆顶位置,将新输入的记录存放在原堆堆底位置,暂时不做处理,当前堆的体积减少一个记录并调整为一个新的最小堆。重复上述步骤,直到内存中未经堆处理的记录数达到 M 时,将这些记录重新建立成最小堆。此时,得到的顺串具有最长的长度,继而开始创建下一个初始顺串。当输出缓冲区满时,将输出缓冲区顺串写入选定的输出文件中。直到输入文件输入完毕。

例如,关键码分别为 50、49、35、45、30、25、15、60、16、27、1、…。假设内存中最多可以容纳 7 个记录,则置换选择排序的过程如图 9.2 所示。

输入缓冲区	内　　存	输出缓冲区
60、16、27、1…		15
16、27、1…		15
16、27、1…		15、25

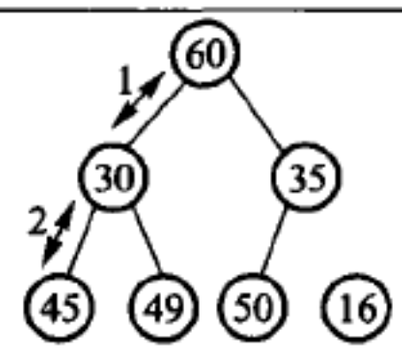
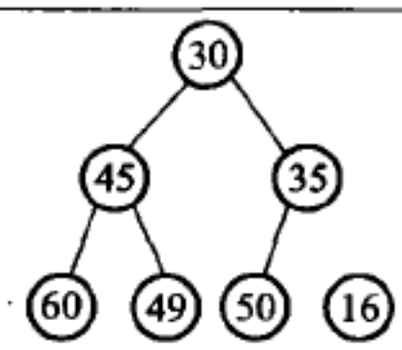
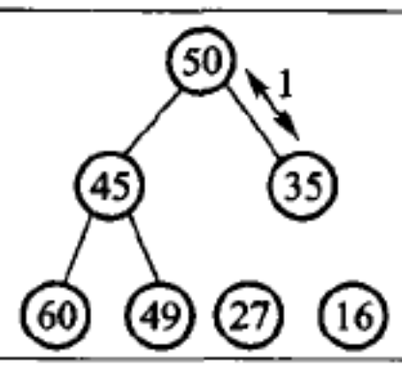
输入缓冲区	内　　存	输出缓冲区
27、1…		15、25
27、1…		15、25、30
1…		15、25、30
…	…	…

图 9.2　置换选择示例

令初始时最小堆的大小为内存工作区的大小 M，数组下标范围从 0 ~ M - 1，堆的规模大小记为 LAST。缓冲区中待输入的记录关键码记为 R。置换选择产生一个顺串的算法如下：

1. 初始化堆

(1) 从磁盘读 M 个记录放到数组 RAM 中。

(2) 设置堆尾指标 LAST = M - 1。

(3) 建立一个最小堆。

2. 重复以下步骤，直到堆为空(即 LAST < 0)

(1) 把具有最小关键码值的记录(根结点)送到输出缓冲区。

(2) 设 R 是输入缓冲区中的下一条记录，判断 R 的关键码值是否大于刚输出的关键码值。

- 如果是，则把 R 放到根结点。
- 否则：

(a) 使用数组中 LAST 位置的记录代替根结点。

(b) 把 R 放到 LAST 位置。

(c) 设置 LAST = LAST - 1。

(3) 重新排列堆，筛出根结点。

算法 9.1 是置换选择算法的实现。

【算法 9.1】　置换选择算法。

```
// 参数 A 是从外存读入数据后所存放的数组，n 是数组元素的个数，in 和 out 是输入/输出文件名
```

```
template < class T >
void replacementSelection(T *A, int n, const char *in, const char *out)
    {
        T mval;                                  // 存放最小堆的最小值
        T r;                                     // 存放从输入缓冲区中读入的元素
        FILE *inputFile;                         // 输入文件句柄
        FILE *outputFile;                        // 输出文件句柄
        Buffer < T > input;                      // 输入缓冲区
        Buffer < T > output;                     // 输出缓冲区
        initFiles(inputFile, outputFile, in, out);   // 初始化输入/输出文件
        initMinHeapArry(inputFile, n, A);        // 从输入文件读入 n 个数据初始化堆数组 A
        MinHeap < T > H(A, n);                   // 建立最小堆
        initInputBuffer(input, inputFile);       // 初始化输入缓冲区,读入一部分数据
        for (int last = (n - 1); last > = 0;) {  // 堆不为空
            mval = H. heapArray[0];              // 获得堆的最小值
            sendToOutputBuffer(input, output, inputFile, outputFile, mval);  // 把 mval 送到输出缓冲区
            input. read(r);                      // 从输入缓冲区读入一个记录 r
            if (! less(r,mval)) {                // 如果 r 值大于等于输出值,r 放到堆的根结点
                H. heapArray[0] = r;
            }
            else {                               // r 不能入堆
                H. heapArray[0] = H. heapArray[last];  // 用堆中 last 位置的记录代替根结点
                H. heapArray[last] = r;          // 把 r 放到 last 位置
                H. setSize(last);                // 堆规模减小 1
                last - - ;                       // 设置 LAST = LAST - 1
            }
            if (last ! = 0) {                    // 重新排列堆
                H. SiftDown(0);                  // 从根结点开始向下筛选
            }
        }
        endUp(output, inputFile, outputFile);    // 处理输出缓冲区,关闭输入/输出文件
    }
```

【算法 9.1 结束】

置换选择排序算法得到的顺串长度并不相等。平均情况下,置换选择排序算法可以形成长度为 2M 的顺串。

9.3.2 二路外排序

本节开始就介绍了外排序通常采用两个阶段的处理方法,即首先把数据文件划分成若干段,

用有效的内排序方法对文件的各段进行初始排序以形成顺串；然后，把这些顺串逐趟合并，直至变为一个顺串为止。外排序本质上基于归并的思想。

最简单的二路归并方法可以用第 8.5 节中的归并排序。外排序实现顺串的两两归并时，用两个输入流读取数据，用一个输出数据流建立归并后的文件，即把内存空间分成 3 个页块，其中两个用于作为输入缓冲区，另一个页块作为输出缓冲区。此时，需要多次进行外存的读/写操作。

下面用一个例子说明二路归并的过程。假设一个含有 3000 个记录的文件，首先通过 10 次内部排序形成了 10 个初始顺串 $R_1, R_2, \cdots, R_{10}$，其中每一个初始顺串的长度都含有 300 个记录。经过第一阶段的内排序过程之后，这 10 个顺串均被放到了外存中。

首先把顺串 R_1 和 R_2 的第一个页块读入到输入缓冲区，进行归并，结果放入到输出缓冲区，输出缓冲区满时写入磁盘。当一个输入缓冲区为空时，便把该顺串的下一页块读入缓冲区，继续归并，直到顺串 R_1 与 R_2 的归并完成为止，此时 R_1 和 R_2 的归并结果就形成了一个新的顺串 R_1'。然后，依次归并 R_3 和 R_4，R_5 和 R_6，…，最后归并 R_9 和 R_{10}。这样就完成了第一趟归并，得到 5 个顺串 $R_1', R_2', \cdots, R_5'$，其中每一个新的顺串含有 600 个记录。接着，再对这 5 个顺串进行二路归并，R_1'和 R_2'得到顺串 R''_1，R_3'和 R_4'得到顺串 R''_2，R_5'轮空一次，得到第二趟归并结果。如此进行下去，共需要 4 趟二路归并把整个文件排序完毕。其过程如图 9.3 所示。

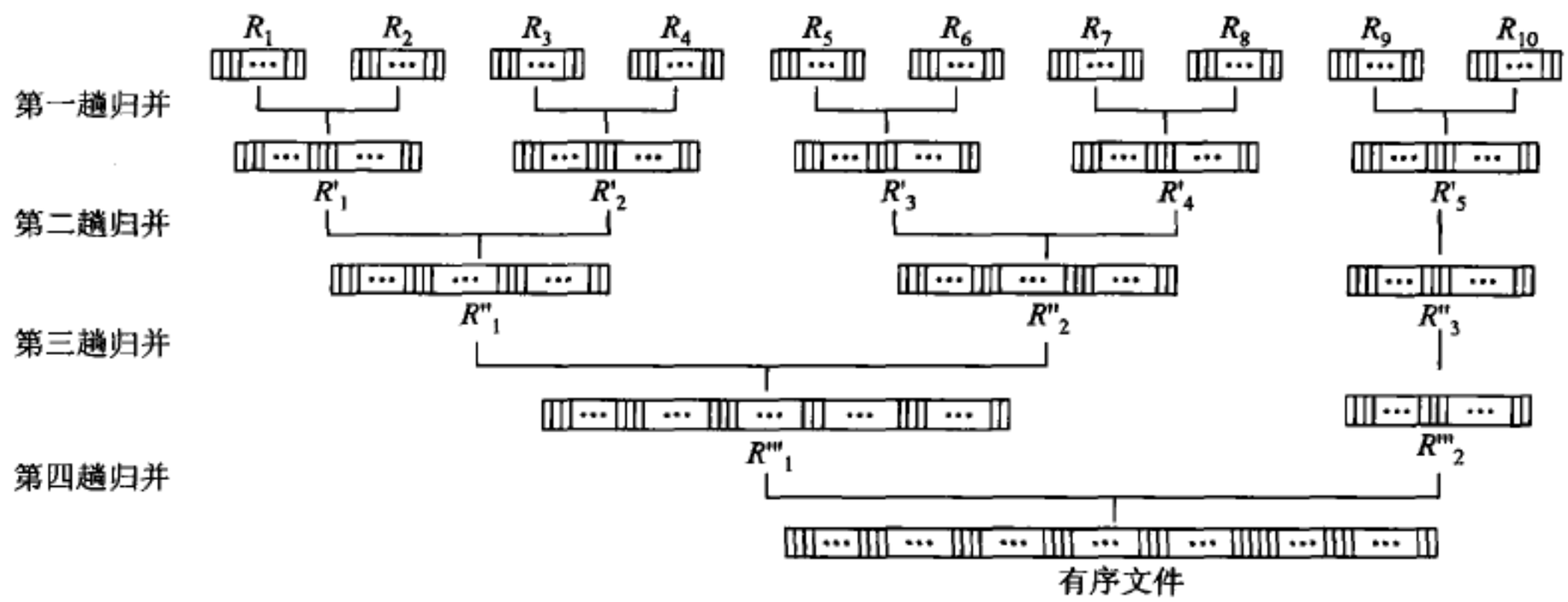


图 9.3 二路归并过程

回顾上述二路归并的过程，每一趟归并的过程都需要把外存文件扫描一遍，此时就要以物理块（页块）为单位对外存文件进行读/写多次。例如，假设每个物理页块可以容纳 100 个记录，第一趟归并顺串需要把外存文件扫描一遍，需要 30 次读/写操作；第二趟归并 R_5' 不参与，共进行 24 次读/写；第三趟归并 R''_3（原 R_5'）不参与，共进行 24 次读/写；第四趟归并，所有记录都扫描一遍，共进行 30 次读/写。4 趟归并，分别需要 108 次读/写操作。

文件长度不变，初始顺串长度增加 1 倍，则可以减少 1 趟归并，只需要 78 次读/写。因此，为一个待排文件创建尽可能大的初始顺串，可以大幅减少扫描遍数和外存读/写次数。

归并顺序的安排也能影响读/写次数，把初始顺串长度作为权，其实质就是 Huffman 树最优

化问题。

9.3.3 多路归并——选择树

k 路归并是每次将 k 个顺串合并成一个排好序的顺串。一般情况下，对 m 个初始顺串进行 k 路归并时归并趟数为 $\lceil \log_k m \rceil$。增加每次归并的顺串数量 k 可以减少归并趟数。

在 $k(k>2)$ 路归并中，为在 k 个初始顺串中的第一个记录中找出具有最小关键码值的记录，需要经过 $k-1$ 次两两比较。若得到含有 u 个记录的归并段，则需要进行 $(k-1)\times(u-1)$ 次比较。在对记录总数为 n 的文件进行外排序时，内部归并过程进行的总比较次数是 $\lceil \log_k m \rceil \cdot (k-1)\cdot(n-1)$，其中 m 表示初始顺串的个数。而

$$\lceil \log_k m \rceil \cdot (k-1)\cdot(n-1) = \lceil \log_2 m/\log_2 k \rceil \cdot (k-1)\cdot(n-1)$$

若 m 不变，随着 k 的增大，$(k-1)/\log_2 k$ 也将增大，即内部归并的时间将随之增大，某种程度上削减了由于增大 k 而减少归并趟数带来的好处。下面介绍的选择树方法可减少从 k 路归并中找出最小关键码值所需的比较次数。选择树从 k 个关键码中找出最小关键码的值所需的比较次数为树高 $\lceil \log_2 k \rceil$，这就使得内部归并过程进行的总比较次数变为

$$\lceil \log_k m \rceil \cdot \lceil \log_2 k \rceil \cdot (n-1) = \left\lceil \frac{\log_2 m}{\log_2 k} \right\rceil \cdot \lceil \log_2 k \rceil \cdot (n-1) = \lceil \log_2 m \rceil \cdot (n-1)$$

此时，内部比较次数与 k 无关，不随 k 的增加而增加。

选择树是完全二叉树，通常采用顺序存储。选择树也称为竞赛树，它反映了一系列“淘汰赛”的结果，有两种类型：赢者树和败者树。

1. 赢者树

对于 n 名选手，赢者树是一棵含有 n 个外部结点，$n-1$ 个内部结点的完全二叉树，其中每个内部结点记录了其左右子结点相应赛局的赢家。

可以通过比较两个选手的分数确定一场比赛的赢家。从树的最底层开始，每两个树叶之间进行比赛，输了的选手被淘汰，赢了的可以继续向上进行竞赛，树根记录了整个比赛的胜者。在最小赢者树中，分数小的选手获胜；在最大赢者树中，分数大的选手获胜。这里以最小赢者树为例来介绍赢者树。

假设用完全二叉树的公式化描述方法来定义赢者树，采用数组作为存储结构。选手或叶结点用数组 $L[1\cdots n]$ 表示，内部结点用数组 $B[1\cdots n-1]$ 表示。数组 B 中实际存放的是数组 L 的索引。例如，图 9.4 所示为有 5 个选手的赢者树。

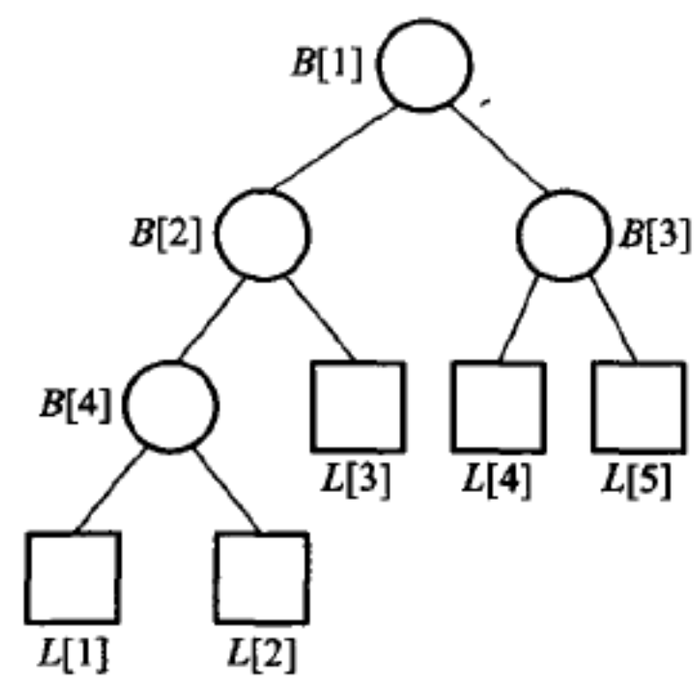


图 9.4 含有 5 个选手的赢者树

在这棵赢者树中，$L[1]$ 和 $L[2]$ 竞赛的优胜者进入到 $B[4]$，下一次与 $L[3]$ 比赛，赢家进入 $B[2]$；$L[4]$ 和 $L[5]$ 竞赛

的优胜者进入到 $B[3]$；$B[2]$ 与 $B[3]$ 比赛，最后的赢家进入 $B[1]$，这也是整个比赛的赢者。

赢者树的一个优点是，如果一个选手 $L[i]$ 的分数值改变了，可以很容易地修改这棵赢者树。此时，只需要沿着从 $L[i]$ 到根结点的路径修改二叉树，而不必改变其他比赛的结果。为了使得这个过程更高效，修改过程中如果沿到树根结点路径上的某一场比赛结果没有改变，则不需要再进行其祖先结点的比赛。这样的修改次数在 $0 \sim \log_2^n$ 之间。

为了实现赢者树的操作，必须能够确定每一个外部结点 $L[i]$ 所对应的内部结点 $B[p]$。当外部结点的数目为 n 时，最底层最左端的内部结点编号为 2^s，其中 $s = \lfloor \log_2(n-1) \rfloor$。因此，最底层的内部结点数为 $n-2^s$，最底层的外部结点数为内部结点数的 2 倍，记为 $LowExt$，$LowExt = 2 \times (n-2^s)$。令 $offset$ 代表最底层外部结点之上的所有结点数目，$offset = 2^{s+1}-1$。对于任何一个外部结点 $L[i]$，其内部结点 $B[p]$ 满足以下公式：

$$p = \begin{cases} (i + offset)/2, & i \leqslant LowExt \\ (i - LowExt + n - 1)/2 & i > LowExt \end{cases}$$

从图 9.5 中可以方便地看到赢者树中各结点与数组 L 和 B 的对应关系。其中 $n=5$，$s=2$；最底层最左端的内部结点为 $B[2^s]=B[4]$，该层的内部结点数为 $n-2^s=1$ 个；$LowExt = 2 \times (n-2^s) = 2$，最底层外部结点数为 2；倒数第 2 层最左端的外部结点为 $L[LowExt+1]$，即 $L[3]$。

赢者树所支持的主要操作包括：初始化赢者树、返回最终赢者、从选手 i 到树根的路径上重新进行比赛。代码 9.2 给出了赢者树的类定义，其具体实现可以参考败者树。

【代码 9.2】 赢者树的类定义。

```
template < class T >
class WinnerTree {
private:
    int MaxSize;                                    // 最大选手数
    int n;                                          // 当前选手数
    int LowExt;                                     // 最底层外部结点数
    int offset;                                     // 最底层外部结点之上的结点总数
    int * B;                                        // 赢者树数组,存储数组 L 的索引
    T * L;                                          // 叶结点数组
    // 在内部结点 B[p]处开始从右分支向上比赛
    void Play(int p, int lc, int rc, int( * winner)(T A[ ], int b, int c));
public:
    WinnerTree(int Treesize = MAX);                 // 构造函数
    ~WinnerTree() {delete [ ] B;}                   // 析构函数
    void Initialize(T A[ ], int size,int ( * winner)(T A[ ], int b, int c));  // 初始化赢者树
    int Winner();                                   // 返回最终胜者的索引
    void RePlay(int i, int( * winner)(T A[ ], int b, int c));  // 外部 L[i]改变后重构赢者树
};
```

【代码9.2结束】

利用赢者树进行 k 路归并时，首先产生一个具有 k 个选手的赢者树，其中第 i 个选手就是第 i 个顺串的当前元素。找出最小关键码值后，修改该顺串的当前值，赢者树重新调整继续选最小。图9.5所示为8路归并的赢者树。其中，叶结点存放各路初始顺串在归并过程中当前记录的关键码值，各个内部结点保存具有左、右子结点赢者（具有较小关键码值）所对应的外部数组 L 的索引下标。赢者树的根结点包含树中最终赢者的 L 索引（指向最小关键码值）。

图9.5所示的例子给出了8个顺串中每个顺串的前3个关键码。经过赢者树的比较可知 $L[3]$ 中的关键码值最小，此时该记录即为下一个要输出的记录。输出该记录后，顺串3的下一个关键码为11的记录作为顺串3的当前记录进入选择树。此时，需要重构选择树，需要沿着从结点 $L[3]$ 到根结点的路径重新进行比赛。重构后的赢者树如图9.6所示。

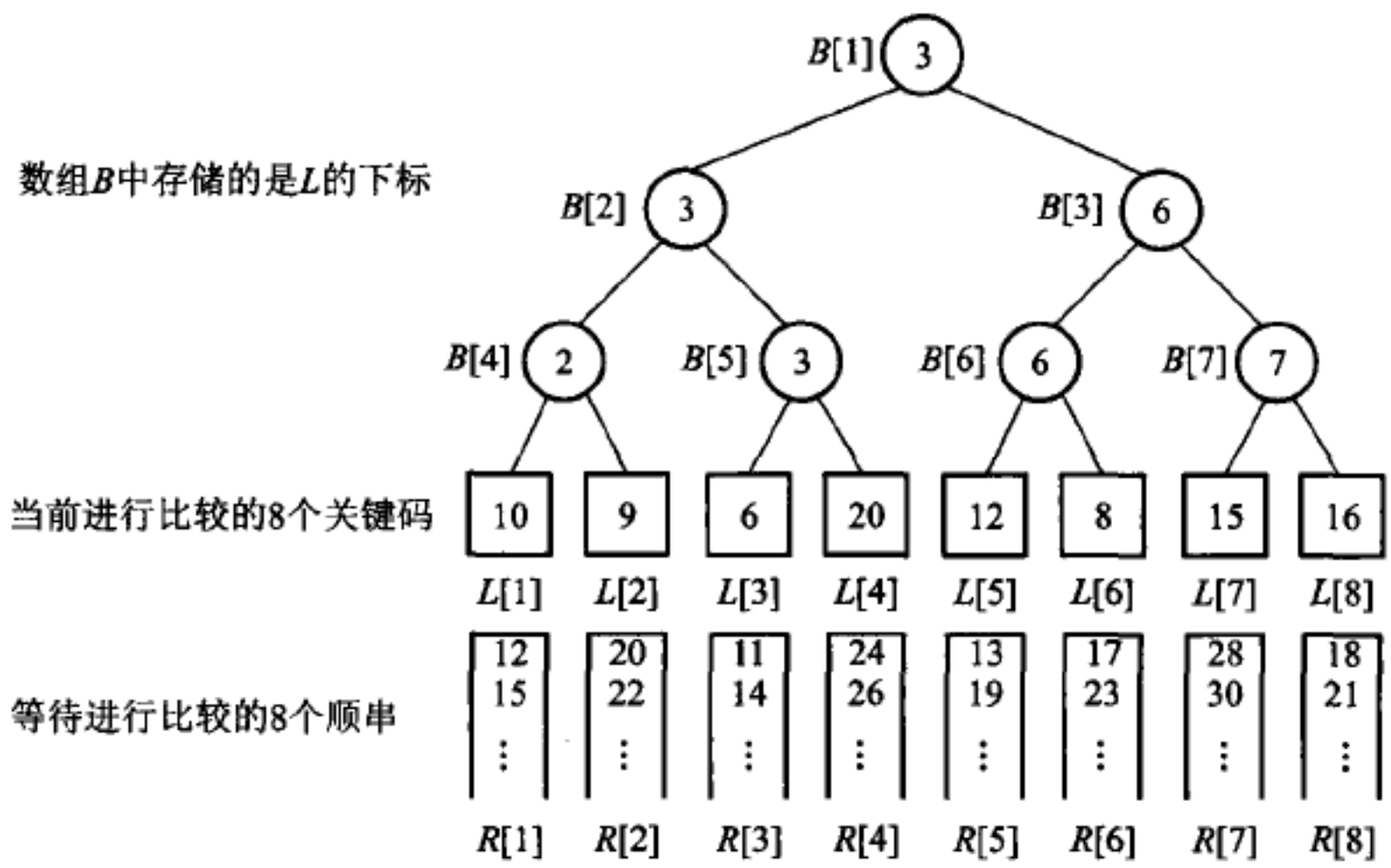


图9.5 8路归并的赢者树

2. 败者树

败者树是赢者树的一种变体。在败者树中，用父结点记录其左、右子结点进行比赛的败者，而让获胜者去参加更高阶段的比赛。另外，根结点处加入一个结点来记录整个比赛的赢者。

采用败者树是为了简化重构过程。败者树中各个分支结点记录了比赛失败者而不是胜利者的索引，当用一个新的关键码代替原来赢者的位置时，就减少了从该位置到败者树的根结点的路径上确定各个索引的工作量。假设 $L[i]$ 的关键码值最小，当用新的关键码代替它重构选择树时，由于赢者树中记录了比赛的胜者，因此路径上的所有比赛都需要找到相应的兄弟结点进行比较，而败者树的分支结点由于记录了比赛的败者，只需要往根的方向与上层父结点所索引的关键码进行比较即可。图9.7所示为与图9.5的输入顺串相同的8路归并败者树。

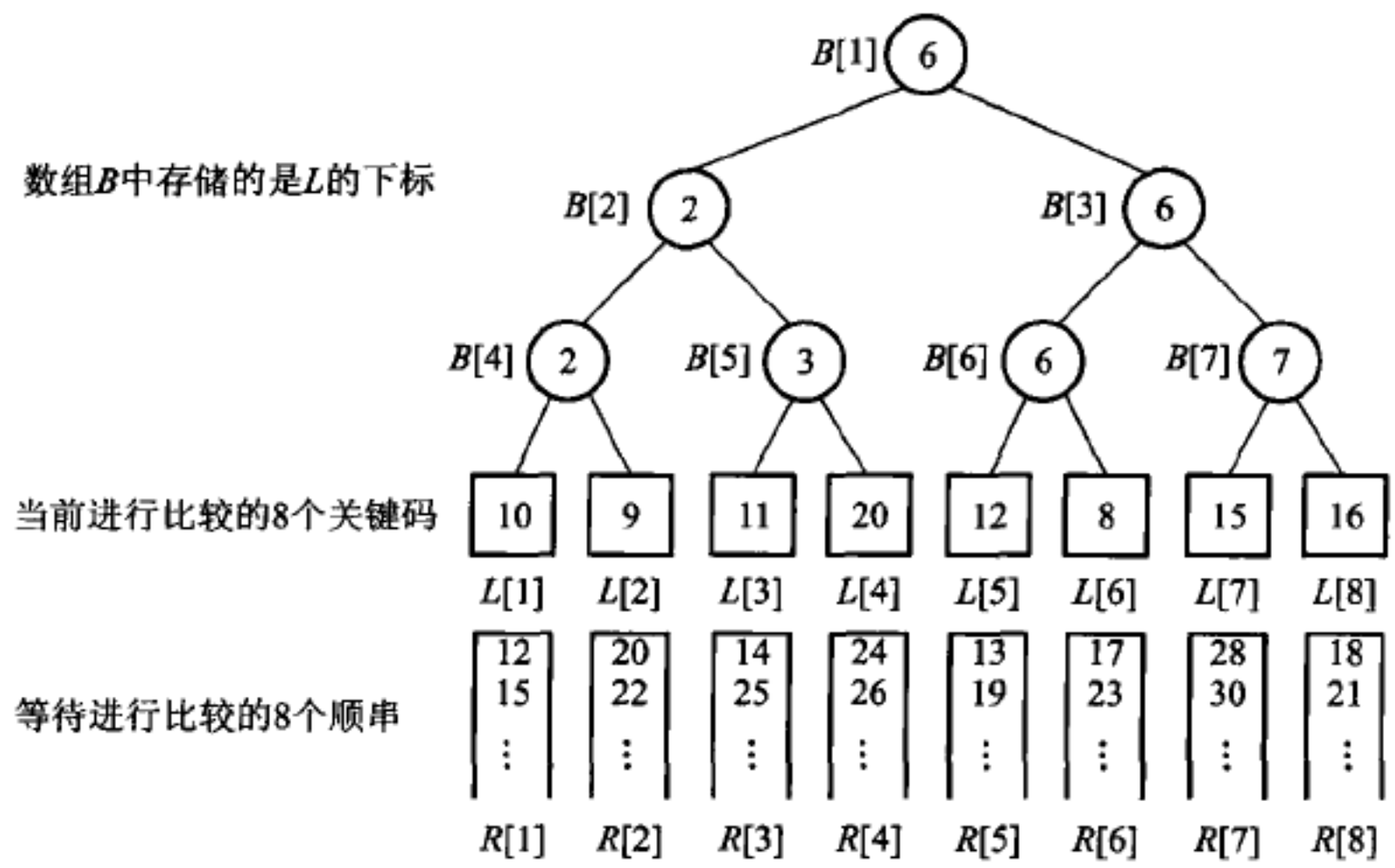


图 9.6 图 9.5 重构后的 8 路归并赢者树

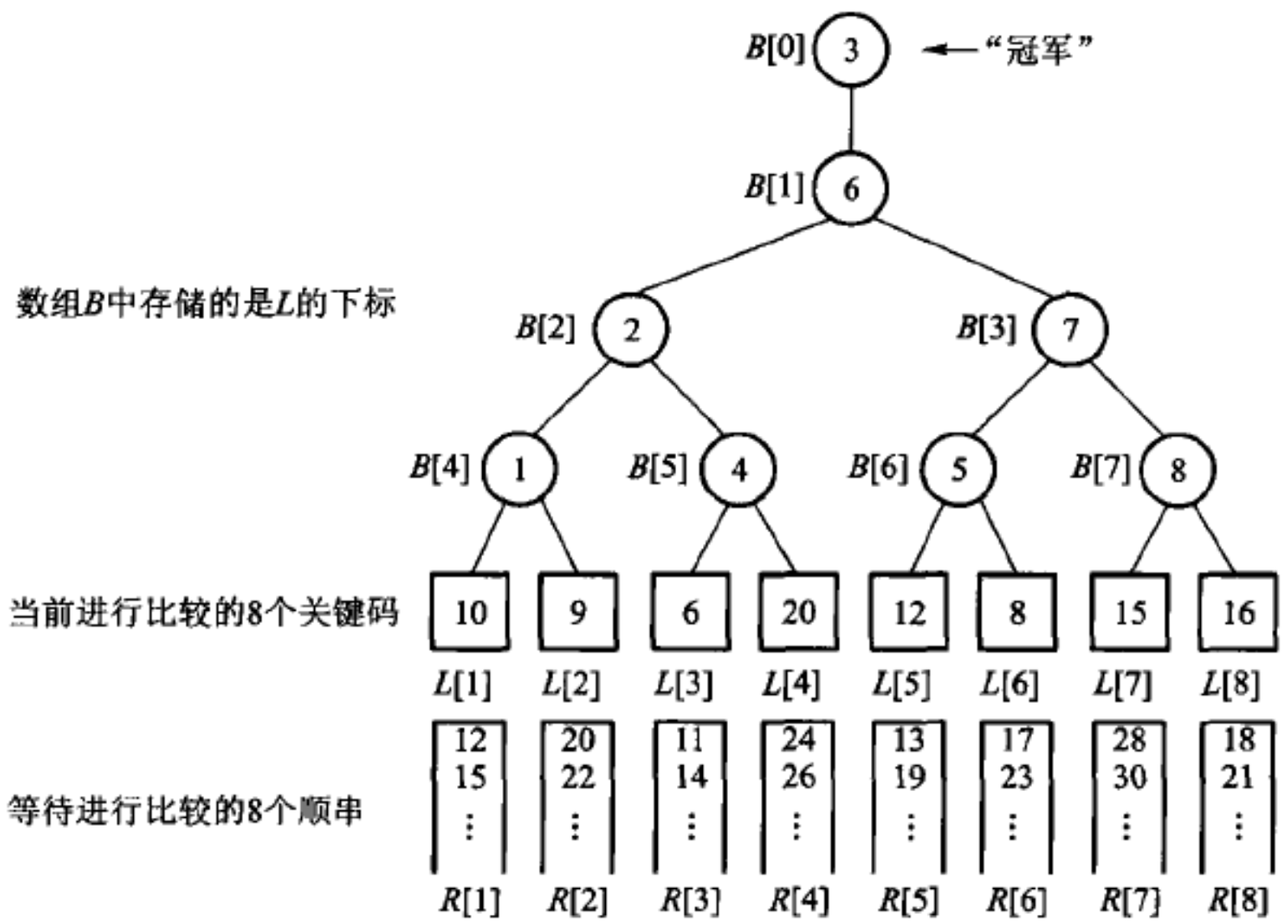


图 9.7 8 路归并的败者树示例

在图 9.7 中，方形外部结点表示当前归并段中参与归并选择的记录关键码。图中加入结点 $B[0]$ 来记录整个比赛的冠军，指示最小关键码为第三个顺串中的当前记录。

当某一个选手 $L[i]$ 的分数改变时，也需要沿着从 $L[i]$ 到根结点的路径修改败者树，而不必改变其他比赛的结果。重构过程如下：将新进入选择树的结点与其父结点进行比赛，把败者的下标存放在父结点中，而赢者再与上一级的父结点进行比较；比赛沿着到根结点的路径不断进行，

直到结点 $B[1]$处，把败者的索引放在结点$B[1]$，把赢者的索引放到结点 $B[0]$。

图 9.7 中 $B[0]$结点所指的顺串 3 中的记录 6 输出后，下一个记录 11 进入该败者树，引起调整。结点 $L[3]$（关键码值为 11）与 $B[5]$所指的结点 $L[4]$（关键码值为 20）进行比赛，败者的索引 4 放在结点 $B[5]$中，而赢者继续与结点 $B[2]$（关键码值为 9）进行比赛，败者的下标 3 放在结点 $B[2]$中，而赢方（关键码 9）继续与结点 $B[1]$（关键码值为 8）比赛，败者的索引 2 放在结点 $B[1]$中，而赢者的索引 6 放到结点 $B[0]$中。重构的结果如图 9.8 所示。

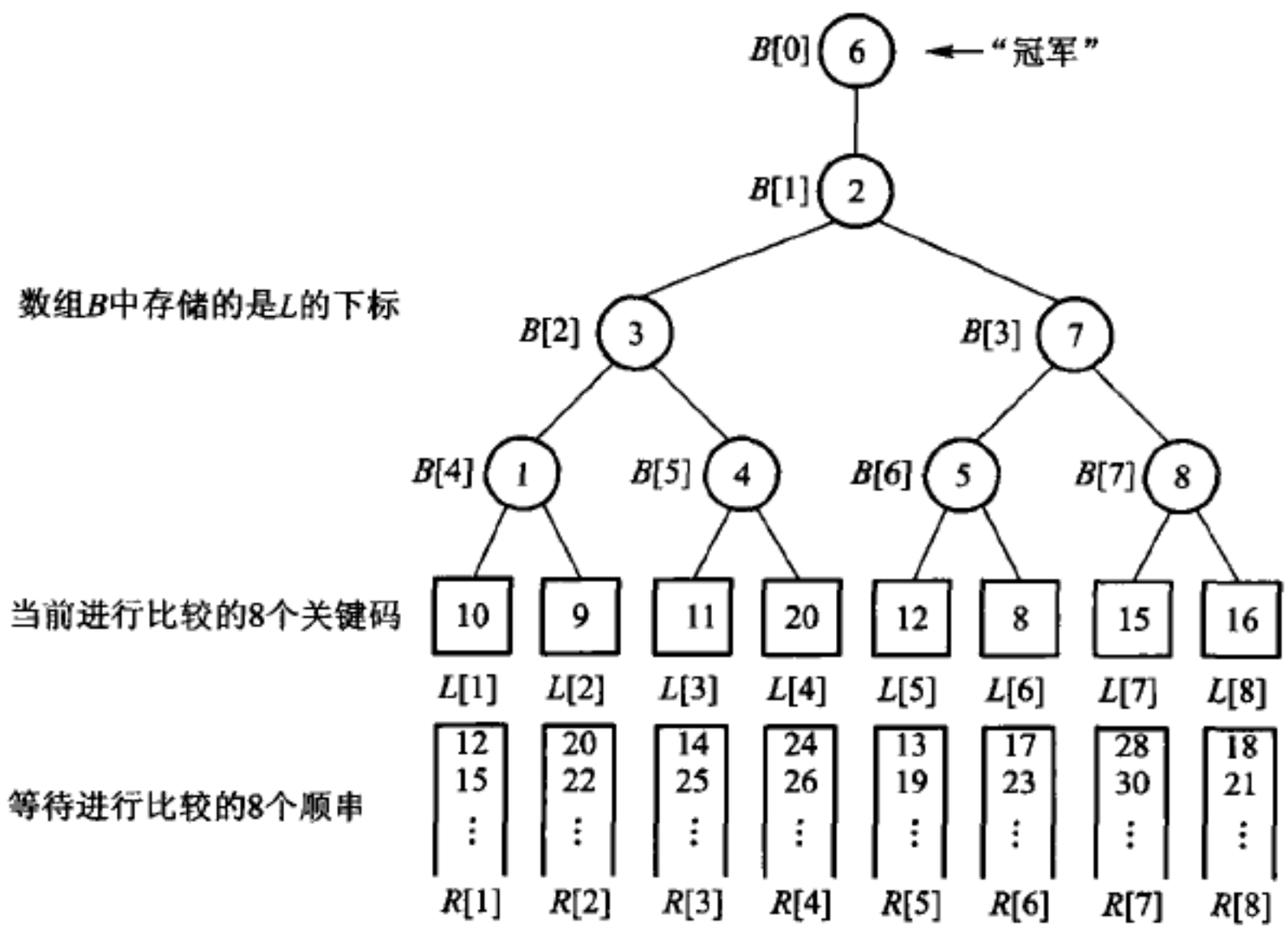


图 9.8 图 9.7 重构后的 8 路归并败者树

败者树的关键操作包括败者树的初始化、返回最终冠军、从选手 i 到树根的路径上重新比赛。败者树的类定义及其实现如代码 9.3 所示。

【代码 9.3】 败者树的类定义及其成员函数的实现。

```
template < class T >
class LoserTree{
private:
    int MaxSize;                                    // 最大选手数
    int n;                                          // 当前选手数
    int LowExt;                                     // 最底层外部结点数
    int offset;                                     // 最底层外部结点之上的结点总数
    int * B;                                        // 败者树数组,存储数组 L 的索引
    T * L;                                          // 叶结点数组
    // 在内部结点 B[p]处开始从右分支向上比赛
    void Play(int p, int lc, int rc, int( * winner)(T A[ ], int b, int c), int( * loser)(T A[ ], int b,
```

```
        int c));
    public:
        LoserTree(int Treesize = MAX);                          // 构造函数
        ~LoserTree(){delete [] B;}                              // 析构函数
        void Initialize(T A[], int size,int (*winner)(T A[], int b, int c), int(*loser)(T A[], int b,
        int c));  // 初始化
        int Winner();                                           // 返回最终赢者索引
        // 位置 i 的选手改变后重构败者树
        void RePlay(int i, int(*winner)(T A[], int b, int c), int (*loser)(T A[], int b, int c));
    };
    template <class T>
    int LoserTree<T>::Winner() {                                // 返回最终赢者的索引 B[0]
        return (n)? B[0]:0;
    }
    // 初始化败者树
    template <class T>
    void LoserTree<T>::Initialize(T A[], int size, int(*winner)(T A[], int b, int c), int(*loser)(T
    A[], int b, int c))
    {
        int i,s;
        if (size > MaxSize || size < 2) {                       // 判断能否处理 size 个选手的数组 A[]
            cout << "Bad Input!" << endl << endl;
            return;
        }
        n = size;                                               // 当前选手数 n 赋值为 size
        L = A;
        for (s = 1; 2*s <= n-1; s += s);                        // 计算 s = 2^log(n-1)
        LowExt = 2*(n-s);                                       // 计算最底层外部结点数
        offset = 2*s-1;                                         // 计算最底层外部结点之上的结点总数
        for (i = 2; i <= LowExt; i += 2)                        // 最底层外部结点的比赛
            Play((offset+i)/2, i-1, i, winner, loser);
        // 处理其余外部结点
        if (n%2) {                                              // n 为奇数,内部结点和外部结点比一次
            Play(n/2, B[(n-1)/2], LowExt+1, winner, loser);// 暂存父结点的左胜者与外部右子结点比
            i = LowExt+3;                                       // i 定位到下一个外部右子结点
        }
        else i = LowExt+2;                                      // i 定位到下一个外部右子结点
        for (; i <= n; i += 2)                                  // 剩余外部结点的比赛
```

```
        Play((i - LowExt + n - 1)/2, i - 1, i, winner, loser);
    }
    // 在内部结点 B[p]处开始从右分支向上比赛
    template < class T >
    void LoserTree < T > ::Play(int p, int lc, int rc, int( * winner)(T A[ ], int b, int c), int( * loser)(T A[ ], int b, int c))
    {
        B[p] = loser(L, lc, rc);                    // 败者索引放在 B[p]中
        int temp1, temp2;
        temp1 = winner(L, lc, rc);                  // p 处的赢者索引
        while (p > 1 && p%2) {                      // B[p]为右孩子,沿路径继续向上比赛
            temp2 = winner(L, temp1, B[p/2]);       // 赢者与父结点比较的赢者索引
            B[p/2] = loser(L, temp1, B[p/2]);       // 败者索引放入 B[p/2]
            temp1 = temp2;                          // 赢者放入 temp1
            p/ = 2;                                 // p 向上指向父结点
        }
        B[p/2] = temp1;          // B[p]是左孩子或者 p = 1,将赢者索引放到 B[p]的父结点
    }
    // 位置 i 的选手改变后重构败者树
    template < class T >
    void LoserTree < T > ::RePlay(int i, int ( * winner)(T A[ ], int b, int c), int ( * loser)(T A[ ], int b, int c)) {
        int p;                                      // 用于计算父结点索引的临时变量
        if (i < = 0 || i > n) {                     // 判断 i 是否越界
            cout < < "Out of Bounds!" < < endl;
            return;
        }
        // 确定父结点的位置
        if (i < = LowExt)
            p = (i + offset)/2;
        else p = (i - LowExt + n - 1)/2;
        B[0] = winner(L, i, B[p]);                  // B[0]中始终保存赢者的索引
        B[p] = loser(L, i, B[p]);                   // B[p]中保存败者的索引
        for (; (p/2) > = 1; p/ = 2) {               // 沿路径向上比赛
            int temp;                               // 临时存放赢者的索引
            temp = winner(L,B[p/2], B[0]);          // 赢者索引暂存 temp
            B[p/2] = loser(L,B[p/2], B[0]);         // 败者索引放入 B[p]的父结点
            B[0] = temp;                            // 赢者索引放到 B[0]
```

```
        }
    }
```

【代码 9.3 结束】

其中，成员函数 Play( )的作用是索引为 lc、rc 的两个子结点进行比赛，产生的败者索引放到父结点 $B[p]$中。此时，如果到达根结点(即 $p=1$)，则比赛已进行完毕，将赢者索引直接放到其父结点 $B[0]$即可；如果内部结点 $B[p]$处于败者树的左子树，则将赢者索引放到 $B[p]$的父结点 $B[p/2]$中，以便等待右子树比赛的结果；否则结点 $B[p]$处的赢者应该与相应左分支的赢者向上继续进行比赛，直到出现上述两种情况之一为止。

败者树的初始化过程(initialize)的设计思想是：首先，最底层的外部结点从左至右依次两两进行比赛，比赛的结果记录在败者树的内部结点中。其余外部结点比赛时会遇到以下两种情况：如果外部结点总数为奇数，那么倒数第二层最左端的外部结点应该与作为其左兄弟的内部结点进行比赛，此时剩下尚未比赛的外部结点中第一个处于右子树位置的索引值为 $LowExt+3$；否则，倒数第二层第一个处于右子树位置的外部结点索引为 $LowExt+2$。最后，完成其余外部结点的比赛。

当位置 i 的外部结点改变后需要调用成员函数 RePlay( )来重构败者树。由于败者树内部结点记录了比赛的败者索引，因此当某个结点的值改变时，只需要与其父结点记录的败者进行比较即可。重构败者树就是沿着到根结点的路径逐层向上进行比赛的过程，直到到达根结点为止。

利用败者树实现多路归并排序算法，首先初始化一棵败者树，取得最终赢者的索引。把赢者送入输出缓冲区中，如果缓冲区已满，就把输出缓冲区的数据传输到磁盘文件中。然后，再从输入缓冲区中读入一个新的关键码，重新从叶结点到根结点调整败者树选择下一个赢者；如果输入缓冲区为空，则从输入文件读取一批数据进输入缓冲区；如果对应输入文件已经没有数据(本顺串数据处理完毕)，就在新的竞赛者位置置入一个较大的数。

本节一开始就讨论到，如果每一轮归并时直接两两比较所有顺串的首结点从而找到其最小值，则找到每一个最小值的时间是 $O(k)$，产生一个大小为 n 的顺串的时间复杂度是 $O(k\cdot n)$。采用败者树对 k 个顺串进行归并，首先初始化包含 k 个选手的败者树，需要时间 $O(k)$，把最小值输出到缓冲区后，读入一个新值并重构败者树的时间为 $O(\log k)$，此时产生一个大小为 n 的顺串的总时间缩短为 $O(k+n\cdot\log k)$，近似于 $O(n\cdot\log k)$。显然，败者树归并效率更高。

本章小结

本章讨论了文件管理和外排序的相关内容。

首先介绍了内存和外存的特点：内存具有存取速度快、造价高、存储容量小和断电后数据丢失等特点；而外存的特点有读/写速度慢、造价低廉、存储容量大，但信息不易失等。

文件在外存上的组织方式称为文件的物理结构，几种常见的物理组织结构有逻辑文件、散列

文件、索引文件、倒排文件。此外，还介绍了 C++ 语言中流的概念和有关的操作。

处理数据量大的文件时，待排记录存储在外存中，不能一次读入内存，需要采用外排序技术。外排序由两个相对独立的阶段组成：根据内存的大小将外存中的数据文件排序成顺串，然后对这些顺串进行逐趟归并，直到最后整个数据文件排列有序为止。

外排序算法是本章的重点，9.3 节中介绍了置换选择算法、二路归并算法和选择树排序算法。

置换选择排序的目标是生成最长的顺串，其核心思想是利用最小堆（或最大堆）对数据进行处理。

二路外排序的归并是逐步把顺串两两合并，直至变为一个顺串为止，从而形成了已排序的文件。

在 K 路归并中，最直接的方法就是进行 $K-1$ 次关键码两两比较来找出最小的记录。为了提高效率，采用选择树的方法来实现 K 路归并。选择树是完全二叉树，有两种类型：赢者树和败者树。利用赢者树进行归并时，将两个子结点中的赢者（关键码值较小者）上升到父结点，赢者树的根结点是树中的最终赢者的索引；利用败者树进行归并时，将两个子结点中的败者（关键码值较大者）上升到父结点，赢者进入高一层的比赛，同时加进一个结点以代表比赛的全局赢者的索引值。此外，本节还详细介绍了败者树算法。

习　　题

1. 假设一个记录长 32 B，一个块长 1 024 B（因此每个块有 32 个记录），工作内存是 1 MB（还有用于 I/O 缓冲区、程序变量等的其他存储空间）。使用置换选择和多路归并，其中归并算法只允许扫描两遍。预计能得到的文件最长为多少？试解释是怎样得到这个结果的。

2. 假设原有一个计算机系统，能够处理一个大公司所有雇员记录的排序操作。由于公司合并，其雇员数目扩大为原来的 100 倍。计算机在可接受的时间内不能对原来 100 倍的记录进行外排序。下面考虑用这几种独立的方案对计算机系统进行升级，它们是否能够满足新的外排序要求？

（1）CPU 的速度增长两倍。

（2）磁盘 I/O 时间缩短至原来的一半。

（3）内存存取时间缩短至原来的一半。

（4）内存大小增长两倍。

3. 如果某个文件经内排序得到 80 个初始归并段，试问：

（1）若使用多路归并执行 3 趟完成排序，那么应取的归并路数至少应为多少？

（2）如果操作系统要求一个程序同时可用的输入/输出文件的总数不超过 15 个，则按多路归并至少需要几趟可以完成排序？如果限定这个趟数，可取的最低路数是多少？

4. 有 8 个顺串，每个顺串的第一个记录的关键码分别为 14、22、24、15、16、11、100、18，而第二个记录的关键码分别为 26、38、30、26、50、28、110、40。试画出对顺串开始 8 路归并时的败者树。从败者树输出一个全局赢者（并有相应的一个记录进入败者树）后需对败者树进行重构，试画出输出第一个全局赢者并进行重构后的败者树。

5. 算法 9.3 是一种败者树的实现方法。此外，还可以用另一种方法来初始化败者树：先构造一棵赢者树，然后把这棵赢者树映射成它所对应的败者树。这一种方法只需要修改败者树的两个成员函数 Initialize( ) 和 Play

()。其中,Play()用于在 p 处的比较,赢者索引放在 B[p]中,若 B[p]是右孩子,则持续向上比赛;Initialize 先按选手索引顺序调用 Play(),生成赢者树之后,再映射成败者树。请读者实现这种方法。

6. 设输入文件包含下列记录(仅列出其关键码):14、22、7、24、15、16、11、100、10、9、20、12、90、17、13、18、26、38、30、25、50、28、110、21、40、…,采用外部结点数为 8 的败者树方法对该文件生成初始顺串。试画出最初产生的败者树和最终输出的所有初始顺串。

上 机 题

1. 在磁盘中,图片一般作为数据一行一行地存储。16 色图片可以用 4 位表示一个像素。如果允许每个像素 8 位,则不需要解压缩像素的处理(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。对比每个像素 8 位和压缩为每个像素 4 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所花的时间更多?对这两种情况进行编程,并比较需要的时间。

2. 根据本章描述的置换选择方法和多路归并算法实现一个外部排序。针对大记录文件和小记录文件两种情况测试你的程序,考虑记录为多大时,应该首先采用关键码排序,然后再根据排序结果重排原数据文件。

第10章 检 索

检索(search)是各种数据结构中必不可少的运算,也是许多计算机应用程序的核心功能。在数据处理中,经常涉及信息存储和信息检索,即对所存储的数据进行快速有效的检索操作。

所谓检索,就是在一组记录集合中找到关键码值等于给定值的某个记录,或者找到属性值符合特定条件的某些记录。例如,一个关于驾驶证记录的数据库,它对颁发的每一张驾驶证都有一条记录。给定一个驾驶证号码,就可以查到与之相关的信息;给定一个姓名,就可能找到多条同名的驾驶证记录。

可以形式化地定义基于关键码的检索。假定 $k_1,k_2,\cdots,k_n$ 是互不相同的关键码值,有一个包含 n 条记录的集合 C,形式如下:

$$(k_1,R_1),(k_2,R_2),\cdots,(k_n,R_n)$$

其中 R_j 是与关键码 k_j 相关联的信息,$1\leqslant j\leqslant n$。给定某个关键码值 K,检索问题(search problem)就是在 C 中定位记录(k_j,R_j),使得 $k_j=K$。检索就是定位关键码值 $k_j=K$ 的记录的系统过程。

检索成功就是找到至少有一个关键码值为 k_j 的记录,使得 $k_j=K$。检索失败就是找不到记录,使得 $k_j=K$(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范围内的所有记录。

可以把检索算法分成4类:

(1) 基于线性表的检索。例如,顺序检索、二分检索。

(2) 根据关键码值直接访问。例如,根据数组下标的直接检索、散列检索等。

(3) 树索引方法。例如,二叉搜索树、字符树、B树等。

(4) 基于属性的检索。例如,倒排表、倒排文件等。

顺序检索、根据数组下标的直接检索等方法比较简单。

当涉及的数据量相当大时,检索的效率就显得格外重要。在实际应用中,为了提高检索效率,往往需要对数据进行特殊的存储处理。

最常见的方法是对数据进行预排序和建立索引。排序算法本身比较费时,但由于这只是预处理(在检索之前已经完成),因此检索时可以针对有序结构采取相应的有效算法(如二分检索)。在数据预处理时也可以牺牲一定的空间来为之建立索引,检索时充分利用辅助索引信息,这样可以换取大量的检索时间,从而提高检索效率。

散列是一种非常高效的检索方法。散列技术把数据组织到一个表中,根据关键码的值来确定表中每个记录的位置,它不适合进行范围查询,一般也不允许出现重复关键码。当散列方法不

适合于基于磁盘的应用程序时，可以选择第 11 章将要介绍的 B 树方法。

树形索引在达到比较好的检索效率的同时，又保证了较好的插入删除效率，其中二叉搜索树主要用于内存数据的索引，字符树用于组织字典，B 树用于组织大型磁盘文件。

基于属性的检索，主要采用倒排技术实现。

为了衡量一个检索算法的效率，需要在时间和空间两方面进行权衡。检索运算的主要操作是关键码值的比较，通常把检索过程中对关键码需要执行的平均比较次数，称为平均检索长度(average search length)，它是衡量检索算法优劣的时间标准。显然，平均检索长度是存储结构中元素总数 n 的函数，定义为：

$$ASL = \sum_{i=1}^{n} P_i C_i$$

其中，P_i 为检索第 i 个元素(即给定值 K 与存储结构中第 i 个元素的关键码值相等)的概率，C_i 为找到第 i 个元素所需的关键码值与给定值的比较次数。

假设线性表为(a,b,c)，检索 a、b、c 的概率分别为 0.4、0.1、0.5，则顺序检索算法的平均检索长度为 $0.4\times1+0.1\times2+0.5\times3=2.1$，即平均需要比较 2.1 次关键码值才能找到待查元素。

“检索第 i 个元素的概率 P”可理解为：在很多次的检索中，检索第 i 个元素的次数占总次数的比例为 P。显然，若已知各个元素检索概率的分布情况，则 ASL 可准确地反映检索算法的平均时间性能。

另外，衡量一个检索算法还要考虑算法所需的存储量、算法的复杂性等因素。

本章先讨论检索线性表以及散列表的方法，后面几章将依次介绍其他方法。第 10.1 节讨论基于线性表的检索，适应于检索存储在链表或数组中的序列；第 10.2 节讨论集合；第 10.3 节讨论散列技术。采用本章前两节所讨论的技术时，记录可以与关键码一起存储在内存中。散列技术把(关键码、记录地址)对组织到一个表中，根据关键码值快速地确定对应的记录位置。

10.1 基于线性表的检索

基于线性表的检索是最简单的检索：数据对象存储于数组或链表等线性结构中，检索算法根据给定值 K，在线性表中进行检索，直到找到 K 在线性表中的存放位置并读取其相应的结点属性值，或者确定在表中找不到 K 为止。

10.1.1 顺序检索

顺序检索(sequential search)是一种最基本的检索方法，其算法思想是：针对线性表里的所有记录，逐个进行关键码和给定值的比较，若某个记录的关键码和给定值相等，则检索成功，找到所查记录；反之，检索失败。表中各数据元素可以任意排列。

【代码 10.1】 顺序检索的存储结构类型定义。

```
template <class T>
class Item {
private:
    T Key;                          // 关键码域
    // ...                          // 其他域
public:
    Item(T value):key(value) {}     // 获取关键码值;
    T getKey() {
        return Key;
    }
    void setKey(const T &K) {       // 设置关键码
        Key = K;
    }
};
vector<Item<T> * > dataList;
```

【代码 10.1 结束】

下面介绍一种设置了“监视哨”的简化顺序检索算法。注意,在本节采用的存储结构中,位置 0 用来做监视哨,位置 1 到 length 用来存储实际元素,检索成功时返回元素位置,检索失败时统一返回 0。

【算法 10.2】 顺序检索算法。

```
template <class T>
int SeqSearch (vector<Item<T> * >& dataList, int length, const T &K) {
    int i = length;
    dataList[0] -> setKey (K);      // 将第 0 个元素设为待检索值
    while (dataList[i] -> getKey() != K)
        i--;                        // 从后往前逐个比较
    return i;                       // 返回元素位置
}
```

【算法 10.2 结束】

监视哨 dataList[0]的作用是保证 while 循环一定终止。因为检索不成功时,i = 0,此时 dataList[0] -> getKey() == k,使得循环终止条件“被迫”成立。因此,不必将“i > 0”写入循环检查条件,从而使其得到简化,达到节省检索时间的目的。监视哨方法利用了一个重要的“加速”原理:当程序的一个循环需要测试两个或多个条件时,应该尽量减少测试条件。

由于检索在线性表的实际应用中使用非常频繁,因此其时间性能就显得十分重要。在顺序检索中,通常假设对于所有 i,$P_i = 1/n$ (称为“等概率假设”),并在此假设下确定检索算法的平均检索长度为:

$$ASL_{succ} = \sum_{i=1}^{n} P_i \cdot C_i = \sum_{i=1}^{n} \frac{1}{n} \cdot i = \frac{1}{n} \cdot \sum_{i=1}^{n} i = \frac{n+1}{2}$$

也就是说,成功检索的平均比较次数约为线性表长度的一半。

如果检索的关键码不在线性表中,则需要进行 $n+1$ 次比较才能确定检索失败(这里指的是加了监视哨的情况)。假设被检索的关键码在线性表中(即检索成功)的概率为 p,不在线性表中(即检索失败)的概率为 $q=(1-p)$。那么,把成功和失败的检索都考虑在内时的平均比较次数为

$$\begin{aligned} ASL &= p \cdot \frac{n+1}{2} + q \cdot (n+1) \\ &= p\frac{n+1}{2} + (1-p)(n+1) \\ &= (n+1)\left(1-\frac{p}{2}\right) \end{aligned}$$

可以看出,$(n+1)/2 < ASL < (n+1)$。

顺序检索与其他检索方法相比的优势在于,对表的特性没有要求,数据元素可以任意排列。插入元素可以直接加到表尾,时间代价为 $\Theta(1)$。

但顺序检索的平均检索长度较大,在平均和最差的情况下的时间代价都是 $\Theta(n)$。当数据规模较大时,检索效率比较低。

为了加快检索,可以根据对线性表记录的预期访问或者根据当前的访问情况进行调整。例如,可以根据预期的访问概率从高到低排序记录;采用调换(transpose)方法,把找到的记录与它在线性表中的前一个记录交换位置。调换方法对链表和数组都适用。随着时间的推移,最常使用的记录将移动到线性表的前面。曾经被频繁访问但以后不再使用的记录将会慢慢地落在后面。但是,下面这个极端的访问序列使得调换方法失效。假设 X 是线性表中最后一个记录,Y 是线性表中倒数第二个记录,考虑访问序列:X、Y、X、Y、X、Y……访问序列不断地在 X 和 Y 之间交替,检索就会不断地访问整个线性表,而 X 和 Y 的位置都不能移至前端。当然,这种极端情况很少在实际中出现。

对于链表实现的线性表,还可以采用移至前端(move-to-front)方法:每当找到一个记录时就把它放到线性表的最前端,而把其他记录后退一个位置。移至前端方法对访问频率的局部变化能够很好地响应,这表现于如果一个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面。

在紫光等中文输入法中,就应用了这些线性表调整技术,把当前常用的字和词调整到前端,有效地提高了用户录入汉字的速度。

10.1.2 二分检索

在一般情况下,一个关键码的所有可能值之间存在某种次序关系。例如,若码值为整数或实

数,则数值的大小是一种次序关系;若码值为字符串,则字典序是一种次序关系。对于任何一个线性表,若其中的所有数据元素按关键码值的某种次序进行不增或不降的排列(不严格的情况下,也称为关键码值递增或递减的次序),则称为有序表。因此,有序表是一种特殊的线性表。

在有序表的存储表示下,检索是否可以用效率更高的方法来实现呢?例如,查英语字典的过程,字典按照从字符 a 到 z 有序排列。查字典时,大体上翻一下,b 打头就靠前翻,z 打头就翻到后面。如果当前页的单词排在待查单词的前面,则往后翻;如果当前页的首单词排在待查单词的后面,则往前翻。重复这个过程,直到待查单词位于当前页的首末单词范围内,最后在当前页查找,再顺序地从前向后搜索直至找到待查单词。

这种查字典的方法,体现了"先确定待查目标所在的范围,然后逐步缩小范围直到找到或找不到该目标为止"这样的思想。下面的"二分检索",其基本思想也是逐步缩小范围,以尽快确定目标。

很显然,由于有序表中所有元素按不增或不降的次序排列,若将表中任一元素 dataList[i]的关键码值 Key 与给定值 K 比较,可根据 3 种比较结果区分出 3 种情况(以递增为例):

(1) Key = K,检索成功,dataList[i]即为待查元素。

(2) Key > K,说明待查元素若在表中,且一定排在 dataList[i]之前。

(3) Key < K,说明待查元素若在表中,且一定排在 dataList[i]之后。

因此,在一次比较之后,若没有找到待查元素(即情况(1)未出现),则可根据比较结果缩小进一步检索的区间。

受此启发,在有序表的存储表示下,人们可采用二分法检索(binary search)。二分检索法的基本思想是:每次将待查区间中间位置上的数据元素的关键码值与给定值 K 比较,若不等则缩小检索区间并在新的区间内重复上述过程,直到检索成功或检索区间长度为 0(检索不成功)为止。

在算法 10.3 的二分检索中,为了与顺序检索法统一,位置 0 不存储有效元素(表示检索失败)。二分检索是比较典型的分治算法。很显然,算法 10.3 也可以改用递归法来实现。

【算法 10.3】 二分检索算法。

```
template <class T>
int BinSearch (vector<Item<T> * >& dataList,int length, const T &K) {
    int low = 1, high = length, mid;                    // low:high 分别记录数组首尾位置
    while (low <= high) {
        mid = (low + high) / 2;
        if (K < dataList[mid] ->getKey())
            high = mid - 1;                             // 右缩检索区间
        else if (K > dataList[mid] ->getKey())
            low = mid + 1;                              // 左缩检索区间
        else return mid;                                // 检索成功,返回元素位置
    }
```

```
        return 0;                                    // 检索失败,返回0
    }
```

【算法 10.3 结束】

现以有序表(15,17,18,22,35,51,60,88,93)为例说明用二分检索算法查找 K = 18 的过程(这里假定表中各元素只含关键码)。首先,取整个有序表为检索区间,通过分别置 low、high 为 1、9 来完成,如图 10.1 所示。

因区间长度大于 0,取区间中间位置 mid = (1 + 9)/2 = 5,将 mid 位置上元素的关键码值 35 与 K = 18 比较。因 18 < 35,将区间缩小为[1,4]。注意,此新区间与原区间[1,9]的差别仅在于上界不同,通过修改上界 high = mid - 1 = 5 - 1 = 4 完成缩小区间的工作,如图 10.1(b)所示。

由于未满足循环终止条件,因此重复上述过程。此时,区间中点为 mid = (1 + 4)/2 = 2,比较结果 18 > 17 表明区间应改为[3,4],这一修改由下界的修改 low = mid + 1 = 2 + 1 = 3 完成,如图 10.1(c)所示。

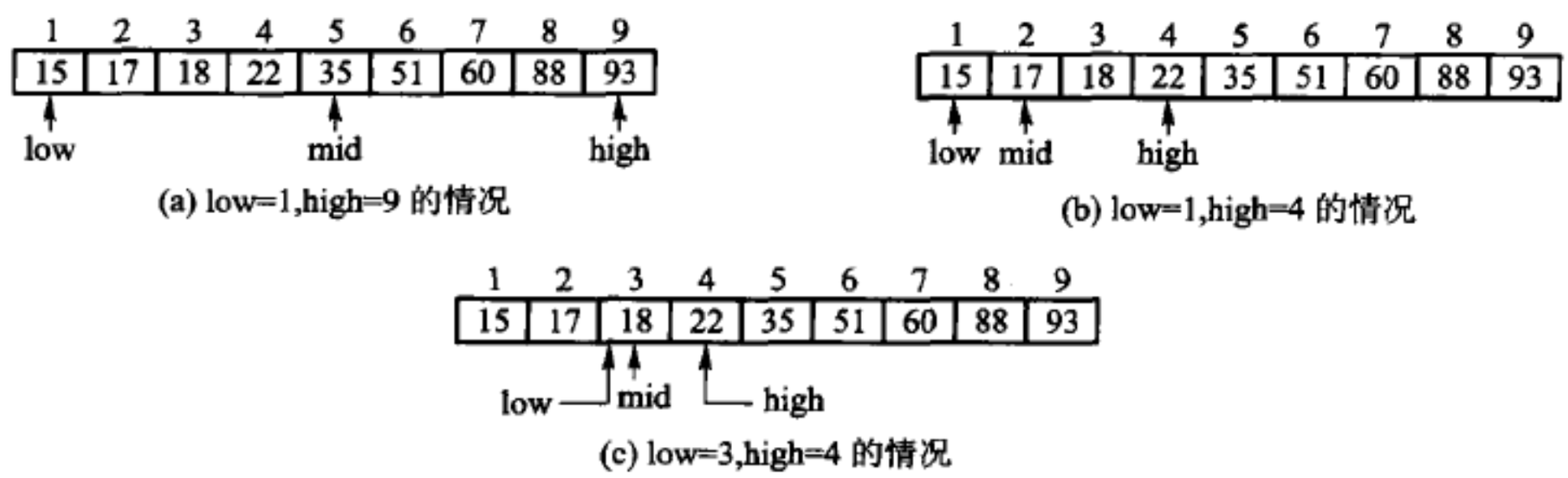


图 10.1 二分检索 K = 18 的过程

再次进行比较时,区间中点为mid = (3 + 4)/2 = 3,比较结果表明dataList[mid]正是待查元素,检索成功,返回结果为 mid = 3。

可以画一棵二分检索的决策树来描述检索过程,该决策树还可以用于分析检索性能,如图 10.2 所示(注意,圆圈外标注的是元素下标)。可以看出,这是一棵 BST(二叉搜索树)。在搜索 18 时,正好走了一条从根结点到与该给定值对应结点的路径,所以 BST 中关键码与给定值的比较次数正好等于该值对应结点所在的层数加 1,因此二分检索到要求元素所需的比较次数最多不超过这棵树的层数加 1(即 BST 的高度)。在本例中,决策树的层数为$\lceil \log_2(n+1) \rceil - 1 = 3$。注意,由于对同一长度的有序数组,采用相同的二分检索算法,因此所对应的决策树的形状是固定的。也就是说,不管数组中元素的数值是什么,数组下标位置的 BST 都保持不变。

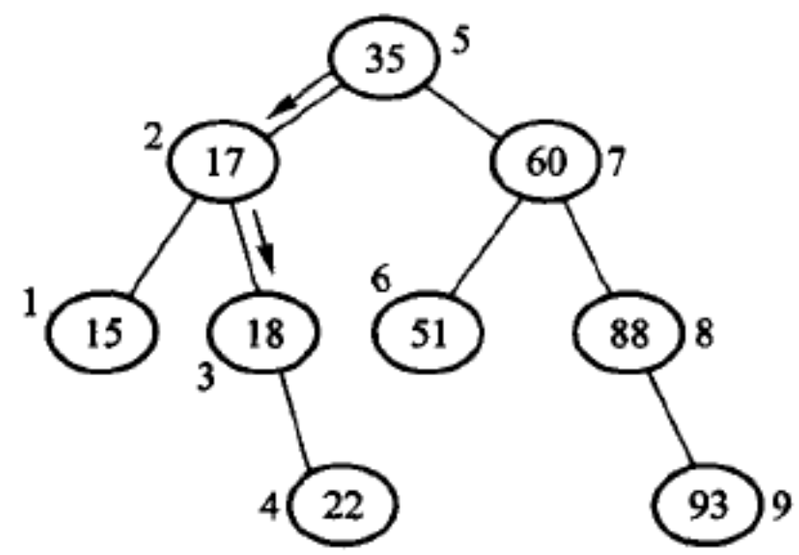


图 10.2 二分检索的决策树

还可以从代数的角度进行分析：只比较1次就成功检索的结点数为 $1=2^0$；只比较2次就成功检索的结点数为 $2=2^1$；…；比较 j 次才成功检索的结点个数为 2^{j-1}。若表中结点总数恰好为

$$n=2^0+2^1+\cdots+2^{j-1}=2^j-1$$

则最大检索长度为 j；若 $2^j-1<n\leqslant 2^{j+1}-1$，则最大检索长度为 $j+1$。总的来说，最大检索长度为 $\lceil \log_2(n+1)\rceil$。

设检索每个结点的概率相同，则平均检索长度（在假设表中结点数 $n=2^j-1$ 的情况下）为：

$$\begin{aligned} ASL &= \frac{1}{n}\left(\sum_{i=1}^{j} i\cdot 2^{i-1}\right) \\ &\approx \frac{n+1}{n}\log_2(n+1)-1 \end{aligned}$$

其推导过程为

$$\begin{aligned} ASL &= \frac{1}{n}\sum_{i=1}^{j} i\cdot 2^{i-1} \\ &= \frac{1}{n}(1+2+2^2+\cdots+2^{j-1} \\ &\qquad +2+2^2+\cdots+2^{j-1} \\ &\qquad\qquad +2^2+\cdots+2^{j-1} \\ &\qquad\qquad\qquad \cdots \\ &\qquad\qquad\qquad\qquad +2^{j-1}) \\ &= \frac{1}{n}\sum_{i=1}^{j}\sum_{m=i}^{j} 2^{m-1} \\ &= \frac{1}{n}\sum_{i=1}^{j}(2^j-2^{j-1}) \\ &= \frac{1}{n}(j\cdot 2^j-2^j+1) \\ &\approx \frac{n+1}{n}\log_2(n+1)-1 \end{aligned}$$

当 n 较大时可得：$ASL\approx\log_2(n+1)-1$。

可以看出，二分检索的平均检索长度与最大检索长度相近，因而效率较高。但它要求被检索序列事先按关键码的次序（递增或递减）排列，而排序本身是一种很费时的操作；另外，二分检索只适用于顺序存储结构，而在顺序结构中插入和删除都比较困难。因此，二分检索特别适用于那种一经建立就很少改动、而又需要经常检索的线性表。

10.1.3 分块检索

如果线性表既希望有较快的检索速度又需要动态变化，则可以采用分块检索（blocking

search)的方法。分块检索又称为索引检索,它是一种性能介于顺序检索和二分检索之间的检索算法。

分块检索方法把线性表分成若干块,在每一块中结点的存放次序是任意的,但是块与块之间必须保持关键码值递增(或者递减)的顺序。例如,递增的分块线性表,在第一块中任一结点的关键码都小于第二块中所有结点的关键码,第二块中任一结点的关键码都小于第三块中所有结点的关键码,依次类推。

另外,还需要建立一个索引表,由"(每块中最大的关键码值,块的起始位置)"这样的二元组构成一个索引表。由于表是分块有序的,因此索引表是一个递增(递减)有序表。检索时,首先用待检索的关键码在索引中查找,确定具有该关键码的结点应该在哪一块中,在索引中检索的方法既可以采用二分法,也可以采用顺序检索;然后,再到相应的块中顺序检索,便可以得到检索的结果。

图 10.3 中的线性表被分成三块:(22,12,13,9,8),(33,42,44,24,48),(60,80,74,49,86,53),并且其中数据元素按块有序(但每一块中的数据元素不要求有序)。例如,第一块中的最大关键码为 22,第二块中的最小关键码为 24,而 22 <24,因此第一块中的任一关键码小于第二块中的所有关键码。对于线性表中的每一块,索引表中有相应的一个"索引项"。每个索引项至少有两个域:块内最大关键码和块起始位置。图 10.3 中的顺序表分成三块,索引表就相应地有 3 个索引项。每一项给出对应块的最大关键码和起始位置。例如,第 2 项指出了第 2 块的最大关键码 48 和其中第 1 个数据元素在顺序表中的位置 6。总之,按上述方式组织起来的存储结构称为索引表。

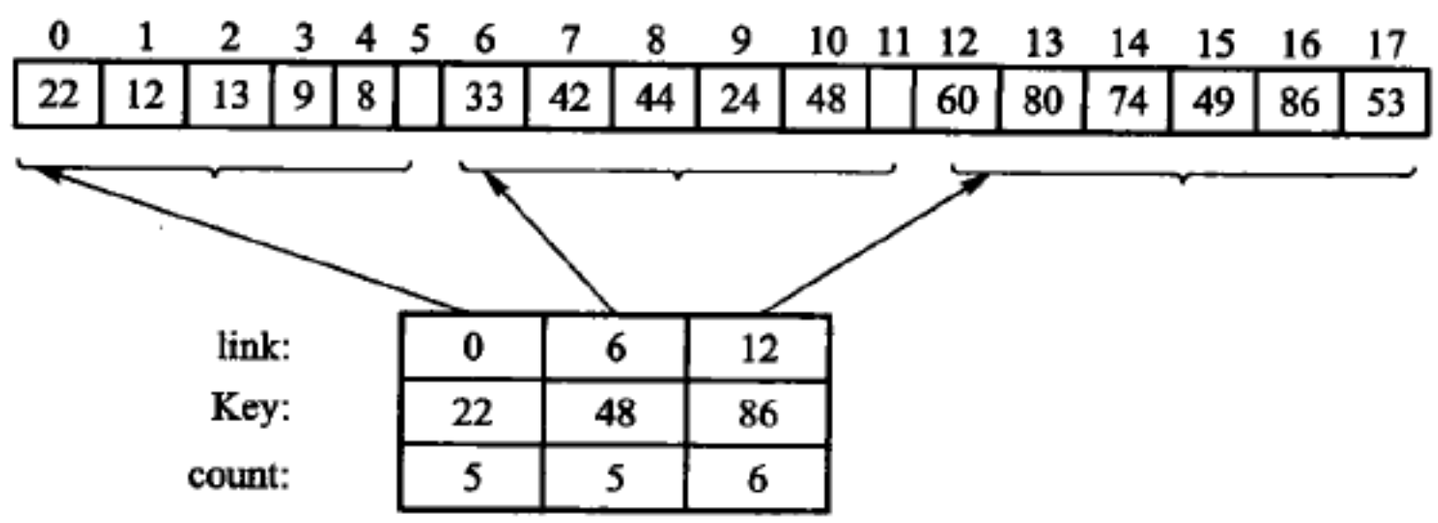


图 10.3 分块检索的存储表示

索引表上的检索分两个阶段:确定待查元素所在的块;在块内检索待查的元素。以图 10.3 所示的索引表为例:假如给定 $K=24$,应先将 K 与索引表中的各个块内最大关键码比较,得知待查元素只能在第二块,然后在第二块中进行检索。第二阶段只能采用顺序检索法,而第一阶段既可以采用顺序检索法,也可以采取二分检索法。

多数情况下,数据不需要完全填满每个块,第 b 块中结点个数小于等于 s,因此索引表还需要记录每块当前记录的数目。

分块检索的平均检索长度等于两个阶段各自的检索长度之和。设线性表中共有 n 个数据元

素，将表均分成 b 块，每块含 s 个元素（$n = s \times b$）。

如果第一阶段以二分检索来确定块，则在等概率假定下，分块检索的平均检索长度约为：

$$ASL_{bs} \approx \left[\log_2\left(\frac{n}{s}+1\right)-1\right]+\frac{s+1}{2}=\log_2\left(\frac{n}{s}+1\right)+\frac{s-1}{2}$$

若第一阶段采用顺序检索，则在等概率假定下，分块检索的平均检索长度为

$$ASL=\frac{b+1}{2}+\frac{s+1}{2}=\frac{n+s^2}{2s}+1$$

显然，当 s 取$\sqrt{n}$时，顺序分块检索的 ASL 达到最小值$\sqrt{n}+1$。如果要检索的线性表中有10 000个结点，把它分成 100 个块，每块中 100 个结点，分块检索时平均需要做 101 次比较，而顺序检索平均约做5 000次比较，二分法检索则最多需要 14 次比较。可见，分块检索的时间性能高于顺序检索而低于二分检索。

如果数据块（子表）存放在外存时，还会受到页块大小的制约，此时往往以外存一次 I/O 读取的数据（一页）作为一块。分块检索的主要代价是增加一个辅助索引数组的存储空间和将初始线性表分块排序的运算。另外，当大量地插入删除运算使块中结点数分布很不均匀时，检索速度将会下降。

分块检索的优点是：在线性表中插入或删除一个结点时，只要找到该结点应属于的块，然后在块内进行插入和删除运算即可。由于块内结点的存放是任意的，因此插入或删除比较容易，不需要移动大量的结点。插入可以在块尾进行；如果待删除的记录不是块中最后一个记录时，可以将本块内最后一个记录移入被删除记录的位置。因此，在某些情况下分块检索是一种比较容易实现的静态查找。

总之，线性表检索的上述 3 种不同实现各有优缺点。其中，顺序检索效率最低但限制最少；二分检索效率最高但限制较多。而分块检索则介于上述二者之间，在实际应用中，可根据线性表的具体情况进行选择，需要综合考虑检索效率、插入删除频率等。

10.2 集合的检索

集合是由若干个确定的、相异的对象构成。这些对象称为元素，一个集合中不包含两个完全相同的元素。在问题求解中，集合是十分有用的工具。

10.2.1 集合的数学特性

数学上的集合（set），其元素可以是原子（属初等类型，不可再分解），也可以是一个集合，而且集合的元素个数可以是有限的也可以是无限的。表示一个集合构成的方法通常是将成员放在

一对花括号中，元素之间用逗号隔开，例如{1, 4}就表示由元素 1 和 4 两个整数构成的集合。{1,4}和{4,1}代表的是同一个集合，因为集合中元素的次序是无关紧要的。

当集合中元素数目较多，难以一一枚举时，一般采用以下形式来表示：

{x|x 应满足的条件 }

其中，“x 应满足的条件”是一个谓词，严格地定义了属于该集合中的元素。例如：

{x|x 是大写英文字母 }

表示集合{A,B, …, Z}。

元素个数为零的集合称为“空集”，一般用 ϕ 来表示。

最基本的关系是成员关系，若 x 是集合 A 的元素，则称“x 属于 A”，记作 $x \in A$。

设有两个集合 A 和 B，如果集合 A 的每个元素也都是集合 B 的元素，称集合 A 被集合 B 包含，记作 $A \subseteq B$（或 $B \supseteq A$），也称 A 是 B 的子集（subset），或称 B 是 A 的超集（superset）。例如，若 $A = \{1,2,3,a,b\}$，$B = \{1,b\}$，则 B 是 A 的子集；但由于 A 中的元素 2、3、a 不是集合 B 中的元素，因此集合 A 不是集合 B 的子集。每个集合都是其本身的子集合，空集是任何集合的子集。

如果 A、B 两个集合互相包含，则称这两个集合相等，记作 $A = B$。集合 A 是集合 B 的一个真子集（或真超集），必须满足 $A \subseteq B$（或 $B \supseteq A$），并且 $A \neq B$。

集合最基本的运算是并、交、差。由至少属于集合 A 和集合 B 之一的一切元素组成的集合，称为 A 和 B 的并集，记作 $A \cup B$。由集合 A 和集合 B 的所有共同元素所组成的集合，称为 A 和 B 的交集，记作 $A \cap B$。由所有属于 A 但不属于 B 的元素的全体所组成的集合，称为 A 和 B 的差集，记作 $A - B$。

例如，若 $A = \{1,2,a,b,c\}$，$B = \{1,3,b,d\}$，则

$A \cup B = \{1,2,3,a,b,c,d\}$

$A \cap B = \{1,b\}$

$A - B = \{2,a,c\}$

一个集合 A 的所有子集合的全体也是一个集合，称为 A 的幂集，记作 $P(A)$。

例如，设 $A = \{a,b,c\}$，则 $P(A) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}$，其中 $\emptyset$ 表示空集合“{ }”。

10.2.2 计算机中的集合

计算机所支持的集合的基类型（basetype），一般是有限、顺序类型。被定义的集合类型称为与基类型相联系的集合类型。集合类型的值集是其基类型值集的幂集。集合类型的每个值是幂集的一个子集。

与集合有关的运算如表 10.1 所示。

表 10.1 与集合有关的运算

	运算名称	数学运算符号	计算机运算符号
算术运算	并	∪	+、|、OR
	交	∩	*、&、AND
	差	-	-
	相等	=	==
	不等	≠	! =
逻辑运算	包含于	⊆	<=
	包含	⊇	>=
	真包含于	⊂	<
	真包含	⊃	>
	属于	∈	IN、at

注：Pascal 语言的集合类型没有“>”、“<”运算。

要判断某一元素是否在数组中，即集合中的“∈”运算，是在一组记录中检索关键码的一种特殊情况。本书所讨论的所有检索方法都可以完成这个任务。

在集合的计算机实现中，可以用自然数来表示有限集合，规定全集为$\{0,1,\cdots,n-1\}$，即在$0\sim n-1$范围之间变化的整数集，其中n取某个较小的固定值。其他基类型的集合可以与整数集建立一一对应的关系。例如，字符集合{'a', 'b',…, 'z'}中所对应的26个小写英文字母可以一一对应为到自然数集合$\{0,1,\cdots,25\}$。还可用建立映射表来为复杂的基类型建立与整数集的对应关系。

可以设置一个 bool 型的数组来表示有限集合，元素在集合中出现，则相应位置为1，否则为0。C++标准模板库 STL 中可以通过 vector<bool>来支持位集合。

由于 C++提供了很强的移位“<<”、“>>”、按位与“&”、按位并“|”、按位异或“^”等操作，因此还可以用位向量来实现集合。即采用一个位数组(bit arrary)，为每一个可能的集合元素(对应到$0\sim n-1$的自然数集合)分配一个比特位(bit)位置。如果元素确实包含在实际集合中，就把它对应的位设置为1；如果元素不包含在集合中，就把它对应的位设置为0。Pascal 语言能够直接支持集合类型，其集合类型是用位向量数组来实现的。STL 也使用位向量数组实现了集合 bitset。

例如，对于小写字母字符型集合{'a'~'z'}，集合型变量为 chset = {'a','c','h','i','j','m','n', 't','v','w','y'}，对应于变量 chset 的位数组如图 10.4 所示。

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
1	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	0	0	1	0	1	1	0	1	0
a		c					h	i	j			m	n						t		v	w		y	

图 10.4 对应于集合变量 chset 的位数组

这种表示方法很节省空间，而且对于“属于”、“并”、“交”和“差”（“at”、“|”、“&”和“-”）操作十分方便。集合比数组的操作更加便捷，例如对于数组的插入和删除，都有大量的数据移动；而集合类型的“并”、“交”和“差”运算只需要修改相应的比特位标记。要确定某个元素是否在集合中，只需要直接检查对应的位标志。这种表示方法称为位向量（bit vector）或者位图（bitmap）。

如果集合大小在计算机的一个字长范围内，而且高级语言支持按位操作，就可以通过逻辑的位操作而完成集合的并、交、差运算。例如，在 C++语言中，集合 *A* 和 *B* 的并运算就是“*A* | *B*”（按位或），集合的交运算就是“*A* & *B*”（按位与），集合 *A* 与 *B* 的差运算可以使用表达式“*A*&~ *B*”（~是非运算的符号）实现。

例 10.1 计算数字 0～15 之间奇素数集合。

假设位向量的从右到左表示自然数，则素数｛13，11，7，5，3，2｝表示为 0010100010101100，奇数｛15，13，11，9，7，5，3，1｝表示为 1010101010101010。

计算表达式 0010100010101100 & 1010101010101010，得到结果“0010100010101000”，表示 0～15之间的奇素数集合为｛13，11，7，5，3｝。

集合的抽象数据类型可以如代码 10.4 这样定义。

【代码 10.4】 集合 mySet 的 ADT。

```
template < size_t N >                                   // N 为集合的全集元素个数

class mySet {
public:
    mySet( ) ;                                          // 构造函数
    mySet( ulong X) ;
    mySet < N > & set( ) ;                              // 设置元素属性
    mySet < N > & set( size_t P, bool X = true) ;
    mySet < N > & reset( ) ;                            // 把集合设置为空
    mySet < N > & reset( size_t P) ;                    // 删除元素 P
    bool at( size_t P) const;                           // 属于运算
    size_t count( ) const;                              // 集合中元素个数
    bool none( ) const;                                 // 判断是否空集
```

```
    bool operator==(const mySet<N>& R) const;          // 等于
    bool operator!=(const mySet<N>& R) const;          // 不等
    bool operator<=(const mySet<N>& R) const;                    // 包含于
    bool operator< (const mySet<N>& R) const;                    // 真包含于
    bool operator>=(const mySet<N>& R) const;                    // 包含
    bool operator> (const mySet<N>& R) const;                    // 真包含

    friend mySet<N> operator&(const mySet<N>& L, const mySet<N>& R);  // 交
    friend mySet<N> operator|(const mySet<N>& L, const mySet<N>& R);  // 并
    friend mySet<N> operator-(const mySet<N>& L, const mySet<N>& R);  // 差
    friend mySet<N> operator^(const mySet<N>& L, const mySet<N>& R);  // 异或
};
```

【代码 10.4 结束】

图 10.4 中的集合元素有 26 个,因此集合可以用一个无符号数来表示,其相应的操作可以通过位运算来完成。如果集合元素个数超过无符号数的字节数,则可以采用一个无符号数组来联合表示。而且,为了运算方便,算法中集合元素的表示法与例 10.1 相同,高位元素在前,低位元素在后。下面定义的数组 *A* 作为集合的私有成员。

```
typedef unsigned long ulong;
enum {
    NB = 8 * sizeof(ulong),          // unsigned long 数据类型的位的数目
    LI = N==0? 0:(N-1)/NB            // 数组最后一个元素的下标
};
ulong A[LI+1];                       // 存放位向量的数组
```

算法 10.5 实现了修改集合元素的操作(插入和删除元素)。如果 P 位为真,P 位处于数组中位置 A[P/NB],而且处于该数组中的第 P % NB 个 bit 位。(ulong)1 << (P% NB)设置二进制数右数第 P 位值为 1,其他位为 0。例如,在微机中无符号长整型为 4 个字节,因此 NB = 32 位。对于全集元素个数 N = 40 的集合,集合{35, 30,9, 7, 5, 3, 1}表示为

```
                                        0000  1000
0100  0000  0000  0000  0000  0010  1010  1010
```

用 2 个 ulong 来实现,实际表示为

```
0000  0000  0000  0000  0000  0000  0000  1000
0100  0000  0000  0000  0000  0010  1010  1010
```

【算法 10.5】 设置集合元素。

```
template<size_t N>
mySet<N>& mySet<N>::set(size_t P, bool X) {
    if (X)                                  // X 为真,位向量中相应值设为 1
        A[P/NB] |= (ulong)1 << (P % NB);    // P 对应的元素进行按位或运算
```

```
        else A[P/NB] &= ~((ulong)1 << (P% NB));      //X 为假,位向量中相应值设为 0
            return (*this);
    }
```

【算法 10.5 结束】

算法 10.6 给出了集合的“交”运算“&”,它是通过“赋值交”来实现的。

【算法 10.6】 集合的交运算“&”。

```
template < size_t N >
mySet < N > & mySet < N > ::operator& = (const mySet < N > & R) {      // 赋值交
    for (int i = LI; i >= 0; i--)                                      // 从低位到高位
        A[i] &= R.A[i];                                                // 以 ulong 元素为单位按位交
    return (*this);
}
template < size_t N >
mySet < N > operator&(const mySet < N > & L, const mySet < N > & R) { //交
    return (mySet < N > (L) & = R);
}
```

【算法 10.6 结束】

进行集合的交运算时,因为每一个元素的标志位都要参与,所以算法代价为 $\Theta(N)$。但是,由于采用了位向量,实际运算实际是 $N/(8*\text{sizeof(ulong)})$ 次无符号数的按位交操作,时间代价还是 $\Theta(N)$ 量级。

集合包含运算“>”可以通过集合的并运算“&”来实现。如果集合 A“真包含”集合 B,那么 $A\&B == B$,且 $A != B$。

【算法 10.7】 集合包含运算“>”。

```
template < size_t N >
bool mySet < N > ::operator > (const mySet < N > & R) const {          // 真包含运算
    return (((mySet(*this) &= R) == R) && (*this != R));
}
```

【算法 10.7 结束】

STL 中的 bitset 和 vector < bool > 都基于位向量支持集合操作。bitset 提供了更加丰富的位运算接口,但是集合的长度需要在编译时决定;而 vector < bool > 没有对位运算提供特别支持,但是它提供了更加灵活的访问,集合的大小不需要在编译时确定。

有些应用中的集合运算主要是集合查询和集合合并,例如第 6 章介绍的合并等价类。如果采用位向量数组来实现集合的合并,理论上算法代价为 $\Theta(N)$。对于这种情况,可以采用第 6 章中介绍的父指针表示法实现的并查集。采用了“重量权衡合并”法则的并查运算,使得集合合并运算接近于 $\Theta(1)$ 时间。

在文档信息检索(document retrieval)中,有一种签名文件(signature file)技术就是根据位向

量来计算待检索的文档集合。签名文件是一个二维数组 M,其每一列代表一个关键词,每一行代表一个文档(也可以反过来)。数组中的元素 $M[i,j]$ 存储一个位标志,标志为"1"则表示第 i 个文档中存在关键词 j(为"0"则不存在)。例如,用户想检索出包含某 3 个关键词的文档;可以把这 3 个关键词所表示的 3 个列取出来,对 3 个位向量进行"并"操作,把那些标志为 1 的位所对应所有文档号返回给用户即可。

10.3 散列方法

前面所介绍的检索,基本上都是基于关键码比较的检索。例如,顺序检索和分块检索依赖于"等于"("==")或者"不等于"("!=")的判断,而二分检索和树形检索(BST,B 树等)依赖于"大于"(">")、"等于"("==")、"小于"("<")这 3 种判断。这些检索方法的平均检索长度都与 n 有关。检索是直接面向用户的操作,当问题规模 n 很大时,上述检索的时间效率可能使得用户无法忍受。

最理想的情况是,根据关键码值,直接找到记录的存储地址,而不需要把待查关键码与候选记录集合的某些记录进行逐个比较。例如,读取指定下标的数组元素,就是根据数组的起始存储地址以及数组下标值而直接计算出来的,所花费的时间是 $\Theta(1)$,与数组元素的个数 n 无关。受此启发,计算机科学家发明了散列方法。

散列方法的主要思想是根据结点的关键码值来确定其存储地址。以关键码值 K 为自变量,通过一定的函数关系 $h(K)$(称为散列函数)计算出对应的函数值,把这个值解释为结点的存储地址,将结点的关键码和属性数据一起存入到此存储单元中。检索时,用同样的方法计算地址,然后到相应的单元中取出要找的结点,实质上是根据关键码找到其存储在表中的属性数据。通过散列方法可以对结点进行快速检索。散列(hash,也称"哈希"、"杂凑")是一种重要的存储方式,也是一种常见的检索方法。

按散列存储方式构造的存储结构称为散列表(hash table)。散列表中的一个位置称为一个槽(slot)。散列技术的核心是散列函数(hash function)。

对任意给定的动态查找表 DL,如果选定了某个"理想的"散列函数 h 及相应的散列表 HT,则对 DL 中的每个数据元素 X,函数值 $h(X.Key)$ 就是 X 在散列表 HT 中的存储位置。插入(或建表)时数据元素 X 将被安置在该位置上,并且检索 X 时也到该位置上去查找。由散列函数决定的存储位置称为散列地址。

因此,散列的核心就是:由散列函数决定关键码值($X.Key$)与散列地址 $h(X.Key)$ 之间的对应关系,通过这种关系来组织存储并进行检索。

一般情况下,散列表的存储空间是一个一维数组 $HT[M]$,散列地址是数组的下标。设计散列方法的目标,就是设计某个散列函数 h,$0 \le h(K) < M$,把对应于关键码值 K 的"(关键码、属性)"结点存储在 $HT[h(K)]$ 中。为了简便,在本节的图例中都没有明显地给出散列表中关键码

所对应的属性域。

例 10.2 已知线性表关键码值集合为：

S = { and, array, begin, do, else, end, for, go, if, repeat, then, until, while, with }

可设散列表为：char HT[26][8]。

如果散列函数 $h(K)$ 的值取为关键码 K 的第一个字母在字母表{a, b, c, …, z}中的序号，即 $h(K) = K[0] - 'a'$，那么(and, array)、(else, end)、(while, with)这三对关键码会发生冲突。可以这样定义新的散列函数，以解决上述“冲突”：散列函数的值为 K 中首尾字母在字母表中序号的平均值，即 $h(K) = (K[0] + K[\text{last}] - 2 \times 'a')/2$。例 10.2 中关键码对应的散列表如图 10.5 所示。

由上面的讨论可以看出：在一般情况下，散列表的空间必须比结点的集合大，此时虽然浪费了一定的空间，但换取的是检索效率。设散列表的空间大小为 M，填入表中的结点数为 N，则称 $\alpha = N/M$ 为散列表的负载因子(load factor，又称“装填因子”)。建立散列表时，若关键码与散列地址是一对一的关系，则在检索时只需根据散列函数对给定值进行某种运算，即可得到待查结点的存储位置。但是，散列函数可能对于不相等的关键码计算出相同的散列地址，该现象称为冲突(collision)，发生冲突的两个关键码称为该散列函数的同义词。在实际应用中，很少存在不产生冲突的散列函数，必须考虑发生冲突时的处理办法。

散列地址	关键码
0	
1	and
2	
3	end
4	else
5	
6	if
7	begin
8	do
9	
10	go
11	for
12	array

散列地址	关键码
13	while
14	with
15	until
16	then
17	
18	repeat
19	
20	
21	
22	
23	
24	
25	

图 10.5 例 10.2 中关键码对应的散列表

因此，采用散列技术时需要考虑的两个首要问题如下：

(1) 如何构造(选择)使结点“分布均匀”的散列函数。

(2) 一旦发生冲突，用什么方法来解决。

当然，还需考虑散列表本身的组织方法，下面分别进行讨论。

10.3.1 散列函数

本节讨论几种散列函数。在以下的讨论中,假设处理的是值为整型的关键码,否则总可以建立一种关键码与正整数之间的一一对应关系,从而把该关键码的检索转化为对应正整数的检索;同时,进一步假定散列函数的值落在 0 ~ M - 1 之间。散列函数的选取原则是:运算尽可能简单;函数的值域必须在散列表的范围内;尽可能使得结点均匀分布,即尽量让不同的关键码具有不同的散列函数值。需要考虑各种因素:关键码长度、散列表大小、关键码分布情况、记录的检索频率等。下面介绍几种常用的散列函数构造方法。

1. 除余法(modulo division method)

顾名思义,除余法就是用关键码 x 除以 M(往往取散列表长度),并取余数作为散列地址。除余法几乎是最简单的散列方法,散列函数为:

$$h(x) = x\% M$$

值得注意的是,在很多情况下,挑选 M 值时要格外小心。例如,把 M 设置为偶数,这意味着如果 x 是偶数,那么 $h(x)$ 也是偶数;x 是奇数,$h(x)$ 也是奇数。如果所有可能的关键码是等概率的,就没有问题。但是,如果偶数关键码比奇数关键码出现的概率大,那么函数 $h(x) = x\% M$ 就不能均匀分布这些关键码的散列值。

类似的,常有人把 M 设置为 2 的幂。此时,$h(x) = x\% 2^k$ 仅仅是 x (用二进制表示)最右边的 k 个位(bit)。若把 M 设置为 10 的幂,此时 $h(x) = x\% 10^k$ 仅仅是 x(用十进制表示)最右边的 k 个十进制位(digital)。虽然这两个散列函数易于计算,但它们不依赖于 x 的全部比特位,并不合乎要求。

因此,通常选择一个质数作为 M 值,那么 $x\% M$ 就依赖于 x 的所有位,而不仅仅是最右边 k 个低位(k 是某个比较小的常数),这就增大了均匀分布的可能性。

除余法的实现相当简单;而且 M 不必是一个常数,它的值可以在程序运行时确定,这就是除余法的优点所在。不论何种情况,除余法的运行时间显然是一个常数。

但除余法也有一个潜在的缺点,即连续的关键码映射成连续的散列值。虽然这能保证连续的关键码不发生冲突,但也意味着要占据连续的数组单元,在某些实现方法中这可能导致检索性能的降低。

2. 乘余取整法(multiplicative method)

使用此方法时,先让关键码 K 乘上一个常数 A $(0 < A < 1)$,提取乘积的小数部分。然后,再用整数 n 乘以这个值,对结果向下取整,将其作为散列的地址。散列函数为:

$$hash(K) = \lfloor n \times (A \times K\% 1) \rfloor$$

其中,"$A \times K\% 1$"表示取 $A \times K$ 小数部分,即

$$A \times K\% 1 = A \times K - \lfloor A \times K \rfloor$$

例 10.3 设关键码 $K = 123456$, $n = 10000$,且取 $A = (\sqrt{5} - 1)/2 = 0.6180339$,则

$$hash(K) = \lfloor 10000 \times (0.6180339 \times K \ \% \ 1) \rfloor$$

因此有

$$\begin{aligned} hash(123456) &= \lfloor 10000 \times (0.6180339 \times 123456 \ \% \ 1) \rfloor \\ &= \lfloor 10000 \times (76300.0041151 \cdots \% 1) \rfloor \\ &= \lfloor 10000 \times 0.0041151 \cdots \rfloor = 41 \end{aligned}$$

此方法的优点是对 n 的选择无关紧要，若地址空间为 p 位，就取 $n = 2^p$。所求出的散列地址正好是计算出来的 $A \times K\%1 = A \times K - \lfloor A \times K \rfloor$ 值的小数点后最左 p 位值。

Knuth 对常数 A 做了仔细的研究，他认为 A 可以取任何值，最佳的选择与待散列的数据特征有关。一般情况下，取黄金分割 $A = (\sqrt{5} - 1)/2$ 最理想。

3. 平方取中法(mid - square method)

由于整数相除的运行速度通常比相乘要慢，因此有意识地避免使用除余法运算可以提高散列算法的运行时间。平方取中法的具体实现是：先求关键码的平方值以扩大相近数的差别，然后根据表长度取中间的几位数(往往取二进制的比特位)作为散列函数值。因为一个乘积的中间几位数与乘数的每一数位都相关，所以由此产生的散列地址比较均匀。

例 10.4　一组二进制关键码(00000100,00000110,000001010,000001001,000000111)

平方结果为　(00010000,00100100,01100010,01010001,00110001)

若表长为 4 个二进制位，则可取中间 4 位作为散列地址：

(0100,1001,1000,0100,1100)

4. 数字分析法(digit analysis method)

设有 n 个 d 位数，每一位可能有 r 种不同的符号。这 r 种不同的符号在各位上出现的频率不一定相同，可能在某些位上分布均匀些，每种符号出现的几率均等；在某些位上分布不均匀，只有某几种符号经常出现。可根据散列表的大小，选取其中各种符号分布均匀的若干位作为散列地址。

计算各位数字中符号分布的均匀度 λ_k 的公式：

$$\lambda_k = \sum_{i=1}^{r} (\alpha_i^k - n/r)^2$$

其中，α_i^k 表示第 i 个符号在第 k 位上出现的次数，n/r 表示各种符号在 n 个数中均匀出现的期望值。

计算出的 λ_k 值越小，表明在该位（第 k 位)各种符号分布得越均匀。

例 10.5　有以下一组关键码，对其各位编号如下：

9	9	2	1	4	8
9	9	1	2	6	9
9	9	0	5	2	7
9	9	1	6	3	0
9	9	1	8	0	5

9 9 1 5 5 8

9 9 2 0 4 7

9 9 0 0 0 1

① ② ③ ④ ⑤ ⑥

其中：

① 位，仅9出现8次，$\lambda_1=(8-8/10)^2\times1+(0-8/10)^2\times9=57.6$。

② 位，仅9出现8次，$\lambda_2=(8-8/10)^2\times1+(0-8/10)^2\times9=57.6$。

③ 位，0和2各出现两次，1出现4次，$\lambda_3=(2-8/10)^2\times2+(4-8/10)^2\times1+(0-8/10)^2\times7=17.6$。

④ 位，0和5各出现两次，1、2、6、8各出现1次。

⑤ 位，0和4各出现两次，2、3、5、6各出现1次。

⑥ 位，7和8各出现两次，0、1、5、9各出现1次。

$$\lambda_4=\lambda_5=\lambda_6=(2-8/10)^2\times2+(1-8/10)^2\times4+(0-8/10)^2\times4=5.6$$

若散列表地址范围有3位十进制数字，取各关键码的④、⑤、⑥位作为记录的散列地址。也可以把第①、②、③和第⑤位相加，舍去进位，变成一位数，与第④、⑥位合起来作为散列地址。此外，也可以用其他方法。

显然，数字分析法仅适用于事先明确知道表中所有关键码每一位数值的分布情况，它完全依赖于关键码集合。如果换一个关键码集合，选择哪几位数据要重新决定。

5. 基数转换法(radix transformation)

将关键码值看成另一种进制的数再转换成原来进制的数，然后选其中几位作为散列地址。例如，对于十进制关键码值210485，先把它看成是十三进制的数，并转换为十进制数：

$$210485_{13}=2\times13^5+1\times13^4+0\times13^3+4\times13^2+8\times13^1+5\times13^0=771932_{10}$$

然后，从中选取几位作为散列地址。假设散列表长度是10 000，则可取低4位1932作为散列地址。

通常，要求两个基数互素，且新基数比原基数大。

6. 折叠法(folding method)

有时关键码所含的位数很多，采用平方取中法计算太复杂，则可将关键码分割成位数相同的几部分(最后一部分的位数可以不同)，然后取这几部分的叠加和(舍去进位)作为散列地址，这种方法称为折叠法。

例10.6 每一种西文图书都有一个国际标准图书编号(ISBN码)，它是一个10位的十进制数字。若以ISBN作为关键码建立一个散列表，当馆藏书种类不到10 000时，可采用此法构造一个4位数的散列函数。如果一本书的编号为0-442-20586-4，则移位叠加和分界叠加如图10.6所示。

```
        5864                5864
        4220                0224
+         04        +         04
[1]0088                     6092
h(K) = 0088         h(K) = 6092
```

(a) 移位叠加　　(b) 分界叠加

图 10.6　移位叠加和分界叠加

打个形象的比喻，分界叠加法就是把数字顺序写到比较透明的纸上，从低位开始把纸折起来，每个间隔中数字位数一样，然后按照某种策略计算这些数字子串的和。移位叠加就是依照折缝把纸裁开，然后把这些数字按位相加。分界叠加时，奇数面（正面）的数字串不变，偶数面（反面）的数字串把反写的数字改写为正写，然后数字子串按位相加。

7. ELFhash 字符串散列函数

ELFhash 函数在 UNIX 系统 V 版本 4 中的 ELF 格式（executable and linking format，可执行链接格式）中会用到。ELF 文件格式用于存储可执行文件与目标文件，ELFhash 函数是对字符串的散列。

结果 h 先初始化为 0。循环地将一个字符串数组中的每个元素依次处理：上次结果 h 左移 4 位，与当前元素相加，组成一个新长整数 h；取 h 的高 4 位存入 g 中；如果 g 不为零，把 g 右移 24 位，即把 g 的高 4 移动到低 4 位，然后与 h 进行按位异或操作；g 取反，然后跟 h 进行按位与运算，得到新的 h。最后得到的长整数 h 对散列表长取余，得到相应的散列数值，即在散列表中的位置。

ELFhash 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，它巧妙地对字符的 ASCII 编码值进行计算，ELFhash( ) 函数能够比较均匀地把字符串分布在散列表中。ELFhash 的实现代码见算法 10.8。

【算法 10.8】 ELFhash 字符串散列函数。

```
int ELFhash(char * K) {
    unsigned long g, h = 0;
    while ( * K) {
        h = (h << 4) + * K ++ ;
        g = h & 0xF0000000L;
        if(g)
            h^ = g >> 24;
        h& = ~g;
    }
    return h % M;
}
```

【算法 10.8 结束】

ELFhash 运算使得散列的结果受到字符串中每个字符的影响，算法本身没有必要记忆和理

解。ELFhash 同样适合于中文编码或中英文混合编码。一个汉字由两个最高位为 1 的字节组成,而英文字母、下画线等其他符号的最高位是 0。对 ELFhash 算法来说,都统一作为字符串处理输入,返回一个散列地址。

在实际应用中,应根据关键码的特点选用适当的散列函数。有人曾用"轮盘赌"(roulette wheel selection)的统计方法对它们进行模拟分析。一个轮盘由绿色 0 和红黑颜色交叉的 1 ~ 36 个共 37 个数字格组成。抛球员将球抛入轮盘上,同时将轮盘旋转,轮盘停止时,球落入哪一格数字中,压这一格的就赢钱。如果轮盘设计得很均匀,每个数字出现的概率应该是 1/37,约为 0.027。

用"轮盘赌"方法模拟的结论是:平方取中法最接近于"随机化"。若关键码不是整数而是字符串时,可以把每个字符串转换成整数,再应用平方取中法。

10.3.2 开散列方法(拉链法)

尽管散列函数的目标是尽可能均匀分布使得冲突最少,但实际上冲突是无法避免的。因此,必须研究冲突解决策略。

冲突解决技术可以分为两类:开散列方法(open hashing,也称为拉链法,separate chaining)和闭散列方法(closed hashing,也称为开地址方法,open addressing)。这两种方法的不同之处在于:开散列法把发生冲突的关键码存储在散列表主表之外,而闭散列法把发生冲突的关键码存储在表中另一个槽内。本节将介绍开散列方法,第 10.3.3 节介绍闭散列方法。

1. 拉链法

开散列方法的一种简单形式是把散列表中的每个槽定义为一个链表的表头。散列到一个特定槽的所有记录都放到这个槽的链表中。图 10.7 说明了一个开散列的散列表,这个表中每一个槽存储一个记录和一个指向链表其余部分的指针。这 7 个数存储在有 11 个槽的散列表中,使用的散列函数是 $h(K) = K\%11$。数的插入顺序是 77、7、110、95、14、75 和 62。有 2 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。

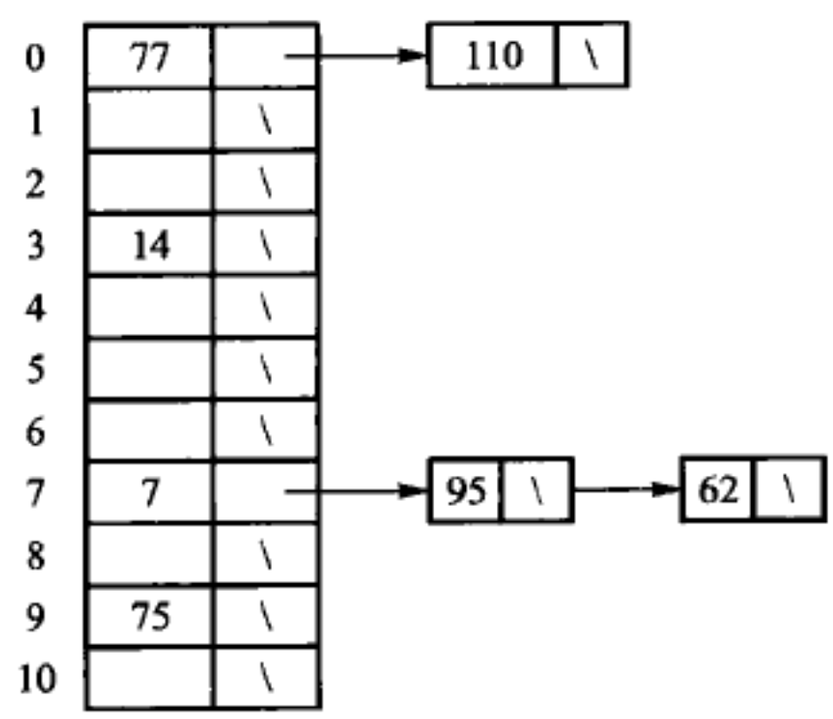


图 10.7 开散列方法的图示

一个开地址的槽链接的线性表(同义词表)中的同义词记录可以按照多种方式排列:根据输入顺序、值的顺序或者访问频率的顺序。对于检索不成功的情况,根据值排列顺序最为方便,因为一旦在线性表中遇到一个比待检索的关键码大的值,就应该停止检索。如果线性表中的记录没有排序,那么一次不成功的检索就需要访问同义词表中的每一个记录。

给定一个大小为 M、存储 N 个记录的表,在理想情况下散列函数将把 N 个记录在表中 M 个位置平均放置,使得每一个同义词表中平均有 N/M 个记录。假定表中的槽数比存储的记录数多,可以期望很少会有包含多于一个记录的槽。这样,散列方法的平均代价就是 $\Theta(1)$。但是,如果冲突使得许多记录集中分布到有限的几个槽中,那么访问一个记录的代价就会更高,因为必须检索同义词表中的许多元素。

如果整个散列表存储在内存中,用拉链法比较容易实现。但是,如果整个散列表存储在磁盘中,用每个同义词存储在一个新地址的拉链法就不太合适。因为一个同义词表中的元素可能存储在不同的磁盘页中,这就会导致在检索一个特定关键码值时引起多次磁盘访问,从而增加了检索时间。下面介绍的桶式散列方法比较适合于存储于磁盘的散列表。

2. 桶式散列

桶式散列(bucket hashing)方法的基本思想是把一个文件的记录分为若干存储桶,每个存储桶包含一个或多个页块,一个存储桶内的各页块用指针连接起来,每个页块包含若干记录。散列函数 h 把关键码值 K 转换为存储桶号,即 $h(K)$ 表示具有关键码值 K 的记录所在的存储桶号。

图 10.8 所示为一个具有 B 个存储桶的散列文件组织。有一个存储桶目录表,存放 B 个指针,每个存储桶一个,每个指针就是所对应存储桶的第一个页块的地址。

有些存储桶仅由一个页块组成,如图 10.8 中的 1 号存储桶。有的存储桶由多个页块组成,每一个页块的块头上有一个指向下一个页块的指针,例如如图 10.8 中的第 $B-1$ 号存储桶由 b_4、b_5、b_6 三个页块组成,每个存储桶中最后一个页块的头上为空指针。

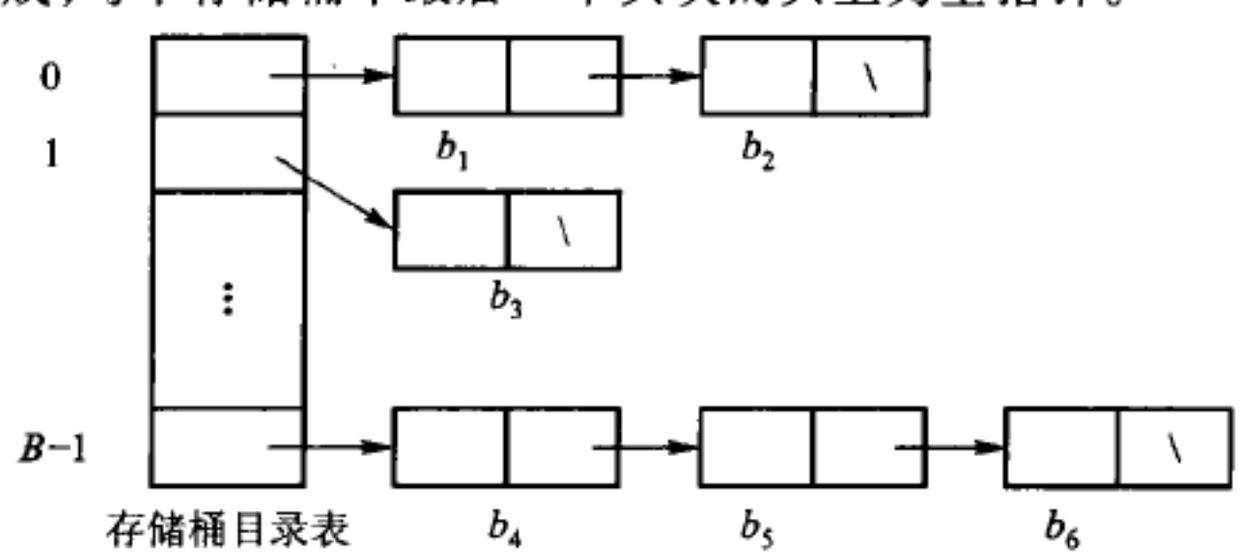


图 10.8 散列文件组织

如果 B 很小,存储桶目录表可放在内存。如果 B 较大,要存放好多页块,则存储桶目录表就放到外存上。当一个散列函数值 i 被计算出来时,首先调存储桶目录表中包含第 i 个存储桶目录的页块进入内存,从中查到第 i 个存储桶的第一个页块的地址,然后再根据这个地址调入相应的页块。

下面分析一下桶式散列的磁盘访问性能。要完成一个检索，为了调存储桶目录表进入内存（假定目录表不在内存）需进行一次访问外存磁盘块的操作（简称为“访外”），同时为了寻找要求的记录必须逐个检查一下桶内各页块，如果能找到要求的记录，则平均访外次数约为桶内页块数的一半。对于修改、插入、删除、修改等其他运算尚需另一次访外，用于重新写回外存。

最理想的状况是每个桶仅由一个页块组成，这样只需访外二次（对检索）或三次（对其他运算），但这要求存储桶的个数大致等于记录存放所需的页块数（即文件所含记录总数除以每页块能容纳的记录数），并且要求散列函数值分布均匀，这种理想状况往往是很难实现的。尤其当文件不断增长时，桶内的页块数也随之增多，由于分布不均匀，有些桶内页块数可能过多，严重影响检索效率。因此，在必要时需对文件进行重新组织，改变散列函数并增加存储桶目录表的长度。

10.3.3 闭散列方法（开地址法）

闭散列方法把所有记录直接存储在散列表中。每个记录关键码 K 有一个由散列函数计算出来的基位置（home position），即 $h(K)$。如果要插入一个关键码 K，而另一个记录的关键码已经占据了 K 的基位置（发生碰撞），则把 K 存储在表中的其他地址内，由冲突解决策略确定是哪个地址。

闭散列表解决冲突的基本思想是：当发生冲突时，使用某种方法为关键码 K 生成一个散列地址序列 $d_0, d_1, d_2, \cdots, d_i, \cdots, d_{m-1}(0 < m \leqslant M)$，称为“探查序列”。其中，$d_0 = h(K)$ 为 K 的基地址；所有 $d_i = (d_0 + p(K,i)) \% M(1 < i < M)$ 是后继散列地址，$p(K,i)$ 是探查函数。当插入 K 时，若基地址上的结点已被别的数据元素占用，则按上述地址序列依次探查，将找到的第一个开放的空闲位置 d_i 作为 K 的存储位置；若所有后继散列地址都不空闲，则说明该闭散列表已满，报告溢出。相应地，检索 K 时，依次查找按同值的后继地址序列，检索成功时返回该位置 d_i；如果沿着探查序列检索时，遇到了开放的空闲地址，则说明表中没有待查的关键码。删除 K 时，也按同值的后继地址序列依次查找，查找到某个位置 d_i 具有该 K 值，则删除该位置 d_i 上的数据元素（删除操作实际上只是对该结点加以删除标记）；如果遇到了开放的空闲地址，则说明表中没有待删除的关键码。因此，对于闭散列表来说，构造后继散列地址序列的方法，也就是处理冲突的方法。

形成探查的方法不同，所得到的解决冲突的方法也不同。下面是几种常见的构造方法。

1. 线性探查法（linear probing）

将散列表看成是一个环形表，若在基地址 d（即 $h(K) = d$）发生冲突，则依次探查下述地址单元：$d+1, d+2, \cdots, M-1, 0, 1, \cdots, d-1$ 直到找到一个空闲地址或查找到关键码为 K 的结点为止。当然，若沿着该探查序列检索一遍之后，又回到了地址 d，则无论是做插入操作还是做检索操作，都会失败。

用于简单线性探查的探查函数如下：

$$p(K,i) = i \qquad (1 < i < M)$$

例 10.7 已知一组关键码为(26,36,41,38,44,15,68,12,06,51,25),散列表长度 $M=15$,用线性探查法解决冲突构造这组关键码的散列表。

利用除余法构造散列函数,选取小于 M 的最大质数 $P=13$,则散列函数为:$h(K)=K\%13$。按顺序插入各个结点:

26: $h(26)=0$, 36: $h(36)=10$,

41: $h(41)=2$, 38: $h(38)=12$,

44: $h(44)=5$

插入 15 时,其散列地址为 2,由于 2 已被关键码为 41 的元素占用,因此需要进行探查。按顺序探查法,显然 3 为开放的空闲地址,因此可将其放在 3 单元。类似地,68 和 12 可分别放在 4 和 13 单元中。依次类推,可建立如图 10.9 所示的散列表。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
26	25	41	15	68	44	6				36		38	12	51

图 10.9 线性探查示例

散列基地址不同的结点,可能争夺相同的后继散列地址。这种现象称为"聚集"(clustering,或称为"堆积"),也称为"基本聚集"(primary clustering)。原因在于散列函数选择不当或负载因子过大。

理想情况下,表中的每个空槽都应该有相同的机会接收下一个要插入的记录。在这个例子中,下一条记录放在第 11 个槽中的概率是 2/15,放到第 7 个槽中的概率是 11/15,概率不再相等。小的聚集可能汇合成大的聚集,导致很长的探查序列。

可以改进线性探查来缓解基本聚集。每次跳过常数 c 个而不是 1 个槽。也就是说,探查序列中的第 i 个槽将是 $(h(K)+i\cdot c)\%M$,探查函数是 $p(K,i)=i\cdot c$。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。

一个好的探查函数应该能够探查散列表中的所有槽。但是某些线性探查函数达不到这个要求。例如,如果 $c=2$,而且表中槽的数目为偶数,对于任何基位置在偶数槽的关键码,它的探查序列只能走遍所有偶数槽;同样,对于基位置在奇数槽的关键码,它的探查序列只能走遍所有奇数槽。如果基地址为偶数的记录比基地址为奇数的记录多,那么偶数地址发生冲突的可能性更高,性能更差。

为了使探查序列走遍表中所有的槽,必须使常数 c 与表长 M 互素。对于一个 $M=10$ 的散列表,如果 c 取 1、3、7 或 9,那么任何关键码的探查序列都能走遍所有的槽。当 $M=11$ 时,c 取 1 ~ 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。

事实上,改进的线性探查还是不能很好地解决聚集问题。例如,$c=2$,要插入关键码 k_1 和 k_2,$h(k_1)=3$,$h(k_2)=5$。k_1 的探查序列是 3、5、7、9、…,k_2 的探查序列就是 5、7、9、…。k_1 和 k_2 的探查序列还是纠缠在一起,从而导致了聚集。

通常希望找到一个探查函数,即使两个不同关键码的基地址或者中间的探查序列偶然会合,最终的探查序列也能够岔开。

2. 二次探查法(quadratic probing)

二次探查法的基本思想是:生成的后继散列地址不是连续的,而是跳跃式的,以便为后续数据元素留下空间从而减少聚集。二次探查法的探查序列依次为:1^2, -1^2, 2^2, -2^2, …,也就是说,发生冲突时将同义词来回散列在第一个地址的两端。求下一个开放地址的公式为:

$$d_{2i-1} = (d + i^2) \% M$$

$$d_{2i} = (d - i^2) \% M$$

二次探查函数为:

$$p(K, 2i-1) = i^2$$

$$p(K, 2i) = -i^2 \qquad (0 < i < M/2)$$

例 10.8 对于一个长度 $M = 13$ 的散列表,假定对于关键码 k_1 和 k_2, $h(k_1) = 3$, $h(k_2) = 2$。k_1 的探查序列是 3、4、2、7……,k_2 的探查序列是 2、3、1、6……这样,尽管 k_2 会在第 2 步探查 k_1 的基地址,但此后这两个关键码的探查序列也会立即分开。

二次探查法的缺点是不易探查到整个闭散列表的所有位置,即上述后继散列地址可能难以包括闭散列表的所有存储位置。但是实际应用中,如果探查序列需要查找 6、7 个槽,该散列表的性能就会非常糟糕,可能需要采取重新调整散列表的措施,如设计新散列函数、申请更大的空间(降低负载因子)、重新组织散列表。

3. 随机探查法(pseudo - random probing)

理想的探查函数应当在探查序列中随机地从未访问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,实际上不能真正随机地从探查序列中选择一个位置,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,可以设置伪随机探查序列,序列中第 i 个槽是$(h(K) + r_i) \% M$,其中 r_i 是 $1 \sim M-1$ 之间数的"伪随机"数序列。所有插入和检索都使用相同的"伪随机"数。探查函数将为

$$p(K, i) = perm[i-1] \qquad (1 < i < M)$$

这里 $perm$ 是一个长度为 $M-1$ 的数组,它包含值从 $1 \sim M-1$ 的随机序列①。这里"伪随机"数序列需要避免产生 0 或者 M,因为那将导致对基地址的探查,多做一次无效操作。

例 10.9 考虑一个长度为 $M = 13$ 的表,伪随机数组前 3 个值为 $r_1 = 2$, $r_2 = 3$, $r_3 = 7$。假定有两个关键码 k_1 和 k_2, $h(k_1) = 4$, $h(k_2) = 2$。k_1 的探查序列是 4、6、7、11……,k_2 的探查序列是 2、4、5、9……这样,尽管 k_2 会把 k_1 的基地址作为第二个选择来探查,这两个关键码的探查序列此后就立即分开。

---

① 产生 n 个数的伪随机排列的函数可以是:

```
void permute(int *array, int n) {
    for (int i=1; i <= n; i ++)
        swap(array[i-1], array[Random(i)]);
}
```

4. 双散列探查法(double hashing)

伪随机探查和二次探查都能消除基本聚集——即基地址不同的关键码,其探查序列的某些段重叠在一起的问题。但是,如果两个关键码散列到同一个基地址,那么采用这两种方法仍然得到同样的探查序列,会产生聚集。这是因为伪随机探查和二次探查产生的探查序列只是基地址的函数,而不是原来关键码值的函数。这个问题称为二级聚集(secondary clustering)。

为了避免二级聚集,需要使得探查序列是原来关键码值的函数,而不是基位置的函数。双散列探查法采用第二个散列函数的值作为线性探查的常数项,形成探查序列。

双散列探查法使用两个散列函数 h_1 和 h_2,其中 h_1 与前面的 h 一样,以关键码为自变量,产生 $0 \sim M-1$ 之间的一个数作为散列地址,h_2 也以关键码为自变量产生一个 $1 \sim M-1$ 之间的、并尽量与 M 互素的数作为对散列地址的补偿。若在地址 $h_1(K)=d$ 发生冲突,则再计算$h_2(K)$,得到的探查序列为:

$$(d+h_2(K))\ \%\ M,(d+2h_2(K))\ \%M,(d+3h_2(K))\ \%\ M\ ,\cdots$$

由此可得双散列函数探查法求下一个开放地址的公式为:

$$d_i=(d+ih_2(K))\ \%\ M$$

探查函数为:

$$p(K,i)=i\cdot h_2(K) \qquad (1<i<M)$$

定义 $h_2(K)$的方法较多,但无论采用什么方法,都应该尽量使 $h_2(K)$与 M 互素,才能使发生冲突的同义词地址均匀地分布在整个表中,否则可能造成同义词地址的循环计算,从而在存储区并未放满时就产生溢出。

一种方法就是选择 M 为一个素数,h_2 返回的值在 $1\leqslant h_2(K)\leqslant M-1$ 范围之间。另一种方法设置 $M=2^m$(2 的幂),让 h_2 返回一个 1 到 2^m-1 之间的奇数值。若 M 是素数,$h_1(K)=K\ \%\ M$,则可以定义 $h_2(K)=K\bmod(M-2)+1$,或者 $h_2(K)=[K/M]\bmod(M-2)+1$。若 M 是任意数,$h_1(K)=K\ \%p$(p 是小于 M 的最大素数),可以定义 $h_2(K)=K\ \%q+1$(q 是小于 p 的最大素数)。注意,尽管最后一个方案并不能保障 $h_2(K)$与 M 互素,但还是常被采用。

用双散列函数探查法得到的探查序列是跳跃式散列在整个存储区域里的,而不是像线性探查法那样探查一个顺序的地址序列。双散列方法的优点是不易产生聚集,缺点是计算量稍微大一些。

10.3.4 闭散列表的算法

字典(dictionary)是一种特殊的集合,其元素是(关键码,属性值)二元组。关键码必须是互不相同的(在同一个字典之内)。字典在某些语言中可能称为联合内存(associative memories)或联合数组(associative arrays)。字典的主要操作是依据关键码来存储和析取值,可以用于组织简单的数据库,提供存储、查找和删除记录的功能。字典的实现方式有很多,例如有序线性表、字符树等。用散列方法实现的字典具有非常高效的检索性能。

算法 10.9 定义了散列表 ADT。为方便读者了解散列表存储表、散列函数和探查函数的作用,此处给出了定义在 private 中的相关内容。在实际应用中,用户可以根据具体情况选择它的散列函数和探查函数。类 hashdict 是一个具有 4 个参数的模板,第一个参数指明关键码的类型,第二个参数指示存储在 hashdict 中的数据记录或元素的类型,第三个参数是在关键码与元素进行比较的类,第 4 个参数是在关键码与关键码直接进行比较的类。

【算法 10.9】 散列表 hashdict ADT。

```
template <class Key, class T >
class hashdict {
private:
    T* HT;                                  // 散列表
    int M;                                  // 散列表长度
    int current;                            // 现有元素数目
    int p(Key K, int i) const               // 探查函数
    int h(Key x) const                      // 散列函数
public :
    hashdict(int sz, T elem)                // 构造函数, elem 用来定义空槽
    ~hashdict() { delete HT; }
    bool hashInsert(const T&);              // 插入元素
    bool hashSearch(const Key&, T&) const;  // 检索关键码
    T hashDelete(const Key K);              // 删除关键码
    int size()                              // 散列表中现有元素数
};
```

【算法 10.9 结束】

本节所讨论的算法基本上都是针对闭散列方法的。散列表的检索过程与建表过程类似。本节讨论的插入和检索算法都假定每个关键码的探查序列中至少有一个槽是空的,否则它们在检索不成功时就会进入无限循环。实际应用中,可以返回插入或检索算法的长度,根据检索长度的统计数据决定是否需要重新调整散列表。

插入算法思想如下:假设给定的值为 K,根据所设置的散列函数 h,计算出散列地址 $h(K)$;若表中该地址对应的空间未被占用,则把待插入记录填入该地址;如果该地址中的值与 K 相等,则报告"散列表中已有此记录";否则,按设置的处理冲突方法查找探查序列的下一个地址,如此反复下去,直到某个地址空间未被占用(可以插入)或者关键码比较相等(有重复记录,不需要插入)为止。

闭散列的插入算法见算法 10.10。

【算法 10.10】 散列表插入算法。

```
// 将数据元素 elem 插入到散列表 HT
template <class Key, class T >
```

```
bool hashdict < Key, T > ::hashInsert( const T& elem ) {
    int home;                                          // home 存储基位置
    int i = 0;                                         // 探查序列编号
    int pos = home = h( getKey( elem ) );              // 初始化探查序列
    while ( HT[ pos ] ! = EMPTY, ) {                   // 顺着非空探查序列先查找 elem
        if ( getKey( HT[ pos ] ) == getKey( elem ) )   // 存在重复关键码
            return false;                              // 发现重复，插入失败，返回 false
        i ++ ;
        pos = ( home + p( getKey( elem ), i ) ) % M;   // 继续向前探查
    }
    // 遇到探查序列的尾部空槽，可以插入新元素
    HT[ pos ] = elem;                                  // 插入元素 elem
    return true;                                       // 插入成功，返回 true
}
```

【算法 10.10 结束】

其中，$p(K,i)$是探查函数(probe function)，它表示的是关键码 K 的第 i 个探查位置对于基地址的偏移量。

对散列表检索的过程，与插入过程类似，它们所采用的探查序列也相同。假设给定的值为 K，根据建表时设置的散列函数 h，计算出散列地址 $h(K)$；若表中该地址对应的空间未被占用，则检索失败；否则将该地址中的值与 K 比较，若相等则检索成功；否则按建表时设置的处理冲突方法查找探查序列的下一个地址，如此反复下去，直到某个地址空间未被占用(检索失败)或者关键码比较相等(检索成功)为止。闭散列的检索算法见算法 10.11。

【算法 10.11】 散列表检索算法。

```
// 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。
template < class Key, class T >                        // 检索关键码值为 k 的记录
bool hashdict < Key, T > ::hashSearch( const Key& K, T& elem ) const {
    int home = h( K );                                 // home 保存 K 的基地址
    int i = 0;                                         // 探查序列编号
    int pos = home;                                    // 探查序列的初始位置
    while ( HT[ pos ] ! = EMPTY ) {                    // 顺着非空探查序列查找 K
        if ( getKey( HT[ pos ] ) == K ) ) {            // 发现目标
            elem = HT[ pos ];
            return true;
        }
        i ++ ;
        pos = ( home + p( K,i ) ) % M;
    }
```

```
        return false;                                    // 达到空槽位置,检索失败
    }
```

【算法 10.11 结束】

上面的插入和检索算法,都没有考虑到散列表的删除操作。实际应用中,建表的过程就是反复地把元素插入散列表的过程,表建好后的主要操作就是检索。但是,也有些应用,可能需要动态地插入和删除某些元素。删除闭散列表中元素时,不能简单地把表中的相应位置标为空,因为这个位置很可能位于一些探查序列中。简单地删除,将使得这些探查序列被打断。当检索探查序列中处于这个位置之后的其他记录时,就可能出现漏检现象。因为算法 10.11 是从基地址开始检查整个探查序列,直到某个地址空间未被占用则检索成功;一旦探查到空槽,就会报告检索失败。

例如,在图 10.9 中,关键码 41 和 15 都散列到第 2 个槽。根据冲突解决策略,把关键码 15 放到第 3 个槽中。如果从表中删除 41,对 15 的检索必须仍然探查第 2 个槽;此时第 2 个槽的原值 41 已经被删除,失去了对关键码 15 的探查导引作用,从而导致检索失败,也就是 15 被漏检了。

另一方面,删除后释放的槽应该能够为将来的插入使用。有人可能想到把探查序列之后的其他记录逐步前移,或者把该探查序列中最后的记录填入到刚刚被删除的单元。这些方法都是不可取的,因为一个单元可能不只处于一个探查序列中(考虑前面讨论过的"聚集"现象),这可能会挪动其他同义词表中的记录,而造成其他混乱。

例如,对于一个长度 $M=13$ 的散列表,假定对于关键码 k_1 和 k_2,$h(k_1)=2$,$h(k_2)=6$。k_1 的二次探查序列是 2、3、1、6、11,k_2 的二次探查序列是 6、7、5、10、2。假设后一序列都被 k_2 的同义词占用,然后插入两个 k_1 的同义词,分别将占用 3 和 1。此时,如图 10.10 所示,如果按上述方法删除位置 6 的元素(用该序列的最后位置 2 的元素替换之),位置 2 设为空,那么以后将会检索不到 k_1 的同义词,可事实上它们还存放在位置 3 和 1 上。

0	1	2	3	4	5	6	7	8	9	10	11	12
	k_1	k_2	k_1		k_2	k_2	k_2			k_2		

图 10.10 散列表中的同义词

可以设置一个特殊的标记位,来记录散列表中的单元状态:空单元、单元被占用、已删除。一般来说,不能把空单元、已删除这两种状态用特殊的值标记,以区别于"单元被占用"状态,因为必须严格区别"空单元"和"已删除"这两种状态。可以用墓碑(tombstone)来表示被删除状态,标志一个记录曾经占用这个槽,但是现在已经不再占用。

加入删除操作后,闭散列的插入操作都需要进行修改,而检索操作几乎可以不修改。检索时,如果遇到墓碑,检索过程会顺着探查序列继续进行,直到检索到关键码或者遇到空单元为止。此时,读者可以把墓碑看成是不等于任何关键码的特殊值,对于检索算法没有影响。

有人可能认为:在插入时,如果遇到标志为墓碑的槽,就可以把新记录存储在该槽中。但是,为

了避免插入两个相同的关键码,仍然需要沿着探查序列进行检索过程。如果在探查序列中发现了相同的关键码,则插入失败。否则,找到一个真正的空位置时,也就是整个探查序列都探测完毕时,可以在第一个墓碑处插入新关键码。下面是考虑了墓碑问题之后的散列表删除和插入算法。

【算法 10.12】 考虑墓碑问题的散列表删除算法。

```
// 删除算法:在检索函数的基础上进行了修改。
// 若检索失败,则无值可删;检索成功则将其设为墓碑。
template <class Key, class T>
T hashdict<Key,T>::hashDelete(const Key K) {
    int home = h(K)                                  // home 记录基位置
    int i = 0;
    T temp;
    int pos = home;                                  // 探查序列的初始位置
    while (HT[pos] != EMPTY) {                       // 顺着非空探查序列查找 K
        if (getKey(HT[pos]) == K) {                  // 找到待删除元素
            temp = HT[pos];
            HT[pos] = TOMB;                          // 设置墓碑
            return temp;                             // 返回目标
        }
        i++;
        pos = (home + p(K, i)) % M;
    }
    return EMPTY;                                    // 达到空槽位置,检索失败
}
```

【算法 10.12 结束】

【算法 10.13】 考虑墓碑问题的散列表插入算法。

```
// 将数据元素 elem 插入到散列表 HT
template <class Key, class T>
bool hashdict<Key, T>::hashInsert(const T &elem) {
    int home = h(getKey(elem));                      // home 记录基位置
    int i = 0, insplace;
    bool tomb_pos = false;
    int pos = home;                                  // 探查序列的初始位置
    while ((HT[pos] != EMPTY) {                      // 顺着非空探查序列先查找 elem
        if (getKey(HT[pos]) == getKey(elem))         // 存在重复关键码
            return false;                            // 不允许重复关键码,返回 false
        if ((HT[pos] == TOMB) && ! tomb_pos ) {
            insplace = pos;                          // insplace 记录第一个墓碑的位置
            tomb_pos = true;
```

```
            }
            i++;
            pos = (home + p(getKey(elem), i)) % M;
        }
    // 遇到探查序列的尾部空槽,可以插入新元素
    if (! tomb_pos)
            insplace = pos;                          // 如果没有墓碑,则 insplace 位于空槽位置
    HT[insplace] = elem;                             // 在待插入位置 insplace 插入 elem
    return true;
    }
```

【算法 10.13 结束】

10.3.5 散列方法的效率分析

根据完成一次插入、删除或检索操作所需要的记录访问次数来衡量散列方法的性能。由于散列表的插入和删除操作都是基于检索进行的,在删除一条记录之前必须先找到该记录,因此删除一条记录之前需要的访问数等于成功检索到它需要的访问数;而插入一条记录时,必须找到探查序列的尾部(对于不考虑删除的情况,是尾部的空槽;对于考虑删除的情况,也要找到尾部,才能确定是否有重复记录),这等于对这条记录进行一次不成功的检索。因此,散列表的效率问题实质上还是平均检索长度的问题,而且需要区别对待成功的检索与不成功的检索。

当散列表比较空的时候,所插入的记录比较容易插入到其空闲的基地址。如果散列表中的记录比较多,插入记录时,很可能要靠冲突解决策略来寻找探查序列中合适的另一个槽。而且,检索记录时,很多时候需要沿着探查序列逐个查找。随着散列表记录不断增加,越来越多的记录有可能放到离其基地址更远的地方。

根据上述讨论,可以看到散列方法预期的代价与负载因子 $\alpha = N/M$ 有关。其中,M 是散列表存储空间大小,N 是表中当前的记录数目。

假定探查序列是散列表中槽的随机排列,下面的分析说明插入(或者一次不成功的检索)的预期代价是 α 的函数。在这些冲突处理策略中,双散列探查法的探查序列最接近于随机分布。注意,这里假设的是最简单的情况,因此所估计出来的代价实际上是双散列探查法平均情况的下限估计。

基地址被占用的可能性是 α,基地址和探查序列中下一个槽都被占用的可能性是 $\frac{N(N-1)}{M(M-1)}$。发生第 i 次冲突的可能性是 $\frac{N(N-1)\cdots(N-i+1)}{M(M-1)\cdots(M-i+1)}$。

如果 N 和 M 都很大,则可以近似地表达为 $(N/M)^i$。探查次数的期望值是 1 加上每个第 i 次 $(i \geq 1)$ 冲突的概率之和,即

$$1 + \sum_{i=1}^{\infty} (N/M)^i = 1/(1-a)$$

一次成功检索(或删除)的代价与当时插入那个关键码的代价相同。由于随着散列表中记录的不断增加,α 值也不断增大,因此可以根据从 0 到 α 的当前值的积分推导出检索操作的平均代价(实质上是对所有插入代价的一个平均值):$\frac{1}{a}\int_0^a \frac{1}{1-x}dx = \frac{1}{a}\ln\frac{1}{1-a}$。

从表 10.2 和图 10.11 可以看出,开散列方法的效率最高,实际系统中使用的散列大多都是开散列。开散列方法非常简单、易于实现,它不会产生聚集现象(聚集导致更大的平均检索长度),删除也极为方便。大部分数据结构教材用比较多的篇幅来讨论闭散列方法,是因为闭散列需要考虑的因素更多,因而更需要精心设计。闭散列在某些受限制的系统中(例如不能使用堆栈分配新空间)有独到的用途,并且经过精心设计的闭散列的效率比开散列稳定。在图 10.11 中,实线显示的是删除或成功检索的时间代价,虚线显示的是插入或不成功检索情况下的时间代价。

表 10.2 几种冲突解决策略平均检索长度的比较

编号	冲突解决策略	平均检索长度	
		成功检索(删除)	不成功检索(插入)
1	开散列法	$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}$
2	双散列探查法	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$
3	线性探查法	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$

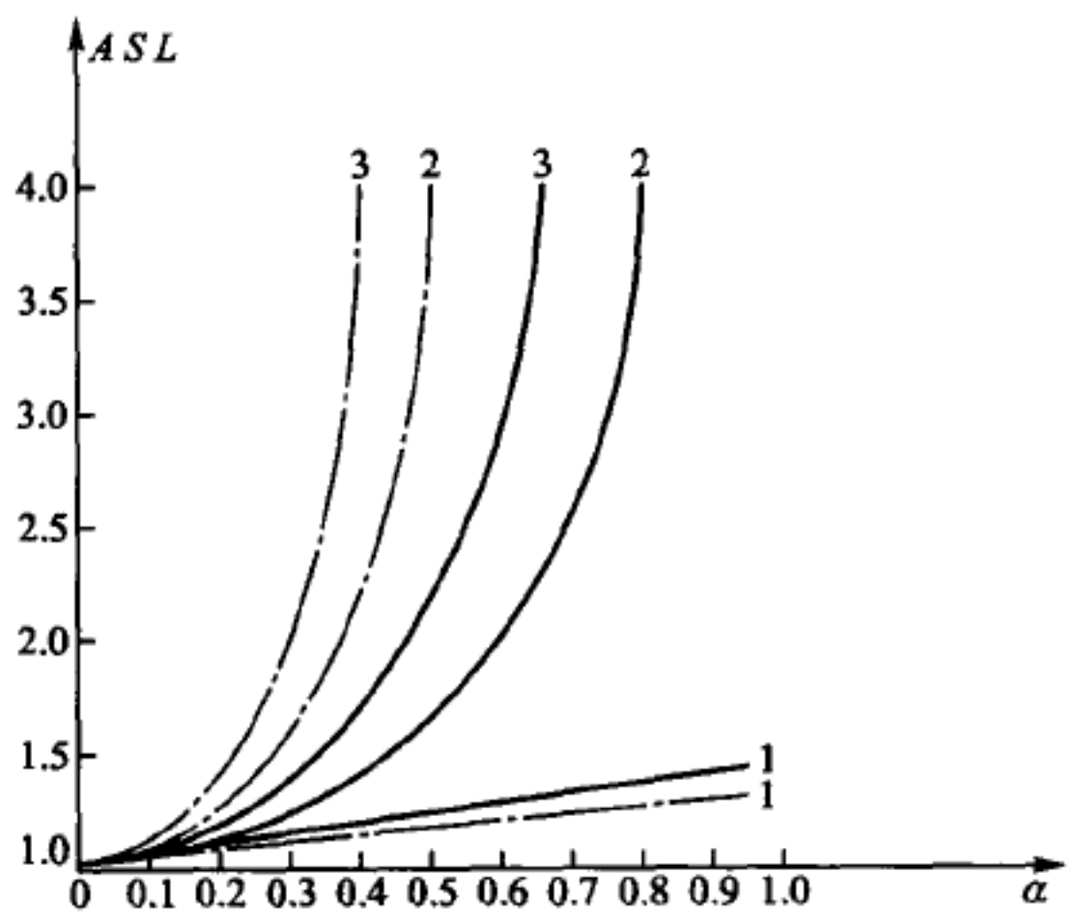


图 10.11 用几种不同方法解决碰撞时散列表的平均检索长度

当负载因子 $\alpha<0.5$，即散列表将近半满时，大部分情况下检索长度小于2。实际经验也表明0.5是一个阈值，负载因子超过0.5，散列表的性能就会急剧下降。因此，需要知道在最大负载情况下表中可能有多少条记录，而选择散列表的长度。

如果大量散列表动态性比较强，即有大量的插入删除操作，那么散列表的性能可能会下降。在这种情况下，可以保持每一条记录的访问计数。当负载因子超过阈值（假设为0.5）时重新散列整个表，或者周期性地重散列。把记录按照访问频率从高到低的顺序插入散列表中，保证上一个周期中被频繁访问的记录更有机会接近基位置。

总之，散列法的重要特征是平均检索长度不依赖于散列表中结点的个数（前面介绍的其他几种检索方法的平均检索长度都是表中结点个数的函数）。散列法的平均检索长度不随表目数量的增加而增加，而是随负载因子的增大而增加。如果安排得好，平均检索长度可以小于1.5。

10.3.6 散列方法的应用

散列方法的检索效率与数据规模无关，平均检索长度接近于1.5，因此它成为一种很受欢迎的高效检索方法。

搜索引擎中关键词字典、域名服务器DNS中域名与IP地址的对应（例如，db.pku.edu.cn与162.105.203.98，www.google.com与216.239.53.99）、操作系统中命令路径下的所有可执行程序名、编译系统中的符号表等，都采用了散列技术以提高查找速度。

UNIX的C Shell运用了一个内部的可执行程序名散列表，来加速在path变量的目录集合中搜寻指令的执行速度。用户注册时，环境变量path所涉及的路径下所有的可执行程序都被散列到命令表。当用户新加入一个可执行的程序（例如mycomm）到path变量的目录中时，如果不将这个内部散列表更新，就只能在该程序所在的目录下执行。也就是说，如果改变工作目录到别的目录中，不带相对或者绝对路径，想要用程序名mycomm直接执行那个新程序，系统会报告“mycomm: Command not found.”。此时，只要用指令rehash将内部散列表更新，便可以搜寻到该新增的程序。相关的指令还有hashstat和unhash。hashstat显示内部散列表的统计信息，指令unhash来解除对path路径的散列，使得系统每执行一个指令便到path变量的目录集合中逐一去尝试执行。

20世纪90年代，Ronald L. Rivest开发出的文档信息摘要MD5（message-digest algorithm 5）是一种高效安全的散列算法。其作用是让文件的大容量信息用散列方法“压缩”成唯一的128位信息摘要。在以后传播这个文件的过程中，如果其内容发生了改变（包括人为修改或者下载过程中的传输错误等），那么对这个文件重新计算MD5就会发现信息摘要不相同，因此可以判定所得到的是一个不正确的文件，达到所谓数字签名效果。MD5还广泛用于加密和解密技术，例如在UNIX系统中用户的密码就是以MD5（或其他类似的算法）经加密后存储在文件系统中。当用户登录时，系统把用户输入的密码计算成MD5值，然后再与保存在文件系统中的MD5值进

行比较，进而确定输入的密码是否正确。通过这样的步骤，系统在并不知道用户密码值的情况下就可以确定用户登录系统的合法性。这不但可以避免用户的密码被具有系统管理员权限的用户知道，而且还在一定程度上增加了密码被破解的难度，这也是 UNIX 系统比一般操作系统更为坚固一个重要原因。

本章小结

本章介绍了一些检索的基本知识，侧重于基于线性表的检索和基于散列表的检索。基于线性表的检索介绍了顺序检索和二分检索。基于散列表的检索是本章的重点，也是难点。散列函数和冲突解决技术是散列算法的核心。

检索成功就是至少找到一个关键码值为 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）。

平均检索长度定义为：

$$ASL = \sum_{i=1}^{n} P_i C_i$$

其中，n 是存储结构中记录的总数，P_i 为检索第 i 个元素的概率，C_i 为找到第 i 个元素所需的关键码值与给定值的比较次数。

基于线性表的检索是最简单的检索：数据对象存储于数组或链表等线性结构中，检索算法根据给定值 K 在线性表中进行检索，直到找到 K 在线性表中的存放位置或确定在表中找不到 K 为止。

顺序检索对表的特性没有要求，数据元素可以任意排列，但顺序检索的平均检索长度较大，在平均和最差的情况下的时间代价都是 $\Theta(n)$。

二分检索是针对有序表的检索。二分检索法的效率可以通过二分检索的决策树进行衡量。其平均检索长度与最大检索长度相近，在平均和最差情况下的时间代价都是 $\Theta(\log_2 n)$，效率较高。二分检索特别适用于那种一经建立就很少改动、而又需要经常检索的线性表。

分块检索又称为索引检索，它把线性表分成若干块，在每一块中结点的存放是任意的，但是块与块之间必须保持关键码值递增（或者递减）的顺序。其性能介于顺序检索和二分检索之间。

集合是由若干个确定的、相异的对象构成的，最基本的运算是“属于 ∈，at”、“并 ∪”、“交 ∩”、“差 -”。在集合的计算机实现中，可以用自然数来表示有限集合，把其他基类型的集合与整数集建立一一对应的关系，本章用位向量实现了一个集合类。在问题求解中，集合是十分有用的工具。

散列是本章的重点和难点，其核心思想是：由散列函数决定关键码值（*X. Key*）与散列地址（散列槽）h（*X. Key*）之间的对应关系，通过这种关系来实现组织散列表的存储并进行检索。常见的散列函数构造方法有除余法、平方取中法、基数转换法、折叠法、ELFhash 等。

散列函数对于不相等的关键码计算出相同散列地址时，就会发生冲突。发生冲突的两个关键码称为该散列函数的同义词。解决冲突的常用方法为开散列方法和闭散列方法。

开散列方法的一种简单形式是把散列表中的每个槽定义为一个链表的表头。散列到一个特定槽的所有记录都放到这个槽的链表中。

闭散列方法为关键码 K 生成一个散列地址序列 $d_0, d_1, d_2, \cdots, d_i, \cdots, d_{m-1}$。其中，$d_0 = h(K)$ 称为 K 的基地址；所有 $d_i(0 < i < m)$ 是后继散列地址。插入元素时，如果基地址已被占用，则按照后继散列地址序列依次探查，直到找到未被占用的地址。一个好的探查函数应该满足两个条件：一是能够探查散列表中的所有槽，二是即使两个不同关键码的基地址或者中间的探查序列偶然会合，最终的探查序列也能够岔开。常用的探查序列构造方法有：线性探查法、二次探查法、随机探查法、双散列探查法等。

散列地址不同的结点，争夺同一后继散列地址的现象称为“聚集”（或称为“堆积”）。基地址不同的关键码，其探查序列的某些段重叠在一起所产生的聚集称为“基本聚集”。两个不同的关键码散列到同一个基地址，如果其探查序列也相同，这样的聚集称为二级聚集。聚集产生的原因在于散列函数和探查序列选择不当或负载因子 $\alpha = N/M$ 过大，其中 M 是散列表存储空间大小，N 是表中当前的记录数目。

线性探查法会产生聚集的问题。伪随机探查和二次探查都能消除基本聚集的问题，但采用这两种方法仍会产生二级聚集。这是因为伪随机探查和二次探查产生的探查序列只是基地址的函数，而不是原来关键码值的函数。双散列探查法利用第二个散列函数作为常数，每次跳过常数项，做线性探查，对二级聚集问题解决得比较好，但缺点是计算量要大一些。

散列方法的效率与负载因子密切相关。当负载因子 $\alpha < 0.5$，即散列表将近半满时，大部分情况下检索长度小于2。当负载因子超过阈值（假设为0.5）或者插入删除操作太多时，需要重新散列整个表。把记录按照访问频率从高到低的顺序插入散列表中，保证上一个周期中被频繁访问的记录更有机会接近基位置。

开散列方法的效率最好，实际系统中使用的散列大多都是开散列。开散列方法非常简单，且易于实现，删除也极为方便。经过精心设计的闭散列的效率比开散列稳定。

习 题

1. 在包含 n 个关键码的线性表中进行顺序检索，若检索第 i 个关键码的概率为 p_i，p_i 分布如下：

$$p_1 = \frac{1}{2}, p_2 = \frac{1}{4}, \cdots, p_{n-1} = \frac{1}{2^{n-1}}, p_n = \frac{1}{2^n}$$

求成功检索的平均检索长度。

注释：本题中的线性表按照访问概率排序：访问第一个记录的概率很高，访问第二个记录的概率则比第一个低很多，但是还是比访问第三个记录的概率高，依次类推。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。

2. 对如下数组，分析用二分法检索88的每一步，写出每一步时的 first、mid 和 last 元素：

8　13　17　26　44　56　88　97

3. 考虑用双向链表来实现一个有序表，使得能在这个表中进行正向和反向顺序检索。若指针 p 总是指向最后成功检索到的结点，检索可以从 p 指示的结点出发沿任一方向进行。

(1) 根据这种情况编写一个函数 search(head,p,K)，检索具有关键码 K 的结点，并相应地修改 p。

(2) 如图 10.12 所示，对于双向链表，假设有序表中元素个数为 n，试给出检索成功和检索不成功时的平均检索长度。

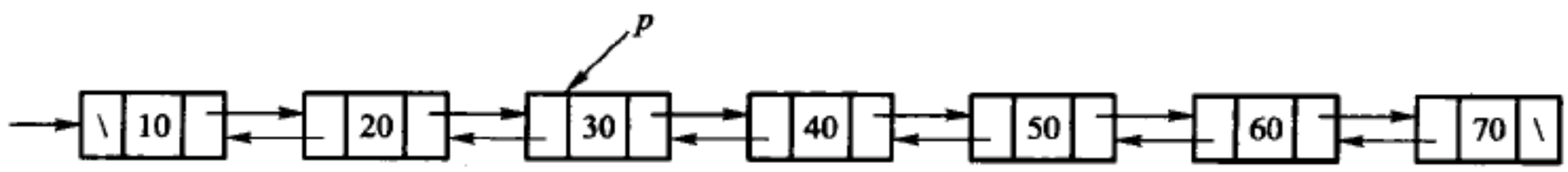


图 10.12　双向有序表

4. 假定查找有序表 A[25]中每一个元素的概率相等，试分别求出进行顺序、二分和分块(假定被分为 5 块，每块 5 个元素)查找每一个元素时的平均查找长度。

5. 编写程序，创建一个值域在[1,20000]内，长度为 10000 的数组。分别用顺序检索法、二分检索法，在[1,20000]内随机生成 10000 个值域范围内的数，并相应地进行 10000 次检索。要求程序输出如下结果：

(1) 检索总次数。

(2) 检索成功的次数。

(3) 检索成功的百分比。

(4) 每次检索进行的平均比较次数。

(5) 程序运行完毕后，画一个关于程序复杂性的分析图，采用大 O 法表示各种算法，并且用程序的实际运行效率来验证。

6. 需要确定某长度为 n 的数组中是否存在 m 个数的和等于给定值 K。例如，如果数组为 8、4、1、6，而 $m=2$，$K=10$，显然 $4+6=10$。数组中的数可能被使用两次。

(1) 试给出 $m=2$ 时时间复杂度为 $O(n^2)$ 的算法。

(2) 试给出 $m=2$ 时时间复杂度为 $O(n\log n)$ 的算法。

(3) 试给出 $m=3$ 时时间复杂度为 $O(n^2)$ 的算法。

(4) 试给出 $m=4$ 时时间复杂度为 $O(n^2\log n)$ 的算法。

提示：可以计算出两个数字和的所有可能值并进行排序，然后再利用前面的算法。

7. 假设讨论的集合是全集合 $\{0,1,2,3,\cdots,n\}$ 的一个子集合，且 n 是一个不大的整数。试定义出该集合的抽象数据类型(ADT)，实现集合 ADT 中的基本操作，并且采用集合 ADT 编写一个打印有穷集合中所有元素的算法。

8. 两个数集合 $S1$、$S2$，大小分别为 n、m，$m=O(\log n)$，试借助于排序来求 $S1$ 与 $S2$ 的交集。

9. 根据教材中定义的集合 ADT，编写一个解决背包问题的算法。

背包问题：一个背包可以放入的物品重量为 S，现有 n 件物品，重量分别 $W_1,W_2,\cdots,W_n$，其利润是 $p_1,p_2,\cdots,p_n$。编写算法判断是否可以 n 件物品中选择 $m(m\leqslant n)$ 件，其重量之和不超过 T，而其总价值最大。

提示：背包问题可以简单描述为：从第 i 到第 n 件物品中选取一种设为第 j 件，若这件可供选入(条件是它尚未被选，且被选入后背包被超重)，则选入并且继续在第 $j+1$ 件到第 n 件中选取。若第 j 件不可选入，且 $j=n$，即后面没有可选择的物品，则将前面的一组解作为装入背包的一种可能。可以用一个集合表示这些物品。

10. 液晶显示往往使用七段数码管，如图 10.13 所示的 8 字就是当所有这 7 段数码管都发光时组成。其他

0~9这10个数字中除8以外的其他数字都是由这7段中的某些段发光而形成的。现在需要找到一种排列这10个数字的方法,使得相邻两个数字由一个数字加几笔或减几笔组成,不能又加又减。

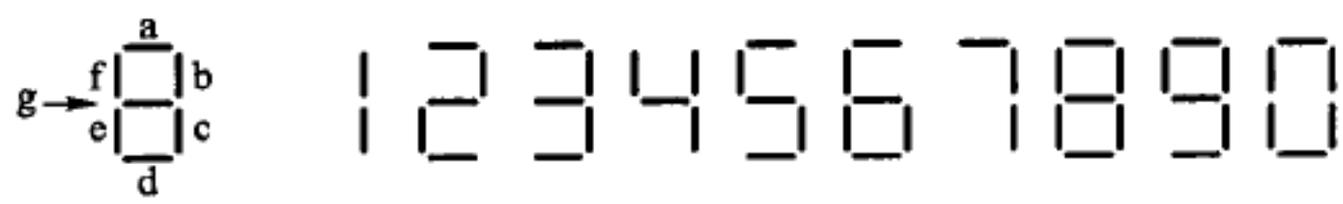


图 10.13 液晶七段数码管示意图

例如,2865937014 就是一个排列,试编程序找出所有这些排列。

注意:每个排列的逆置都是另外一个合法的排列,需要排除本质相同的解。

提示:先将问题用数学模型来表示。用七段管来表示 0~9 中的每一个数,每个数可以用 'a'~'g'的一个子集合来表示。例如,数字 8 是全集['a'..'g'],数字 3 是集合['a'..'d','g'],对于数字 1,它既可以是['e','f'],也可以是['b','c'],先选择['b','c'](注意,不同的选择所得到的答案不一样)。把它们存放于集合数组 S 中。

S[0] = ['a'..'f'] S[1] = ['b','c']

S[2] = ['a','b','d','e','g'] S[3] = ['a'..'d','g']

S[4] = ['b','c','f','g'] S[5] = ['a','c','d','f','g']

S[6] = ['a','c'..'g'] S[7] = ['a'..'c']

S[8] = ['a'..'g'] S[9] = ['a'..'d','f','g']

如果两个数字之间可以互相转化,那么它们相应的集合之间一定存在包含关系,这样,要处理的问题就转化为判断集合之间的包含关系。如果把数字作为结点,能够用边连接相邻的数字(某个数的笔画集合是另一个的子集),则构成一个无向图。现在要找一条路径经过所有的结点一次,这就是图论中的"哈密顿路径"问题。

11. 假定一个待散列存储的线性表为(32, 75, 29, 63, 48, 94, 25, 46, 18, 70),散列地址空间为 HT[23](下标位置为 0, 1,…, 22),若采用除留余数法构造散列函数和线性探查法处理冲突,试求出每一个元素的散列地址,画出最后得到的散列表,求出平均查找长度。

12. 设散列表为 HT[19],对 10、100、32、45、58、126、3、29、200、400、0 建立一个散列表。

(1) 散列函数为直接对 19 求余,给出此时的散列表和冲突次数。

(2) 散列函数为将各个位置上的数字相加后对 19 求余,给出此时的散列表和冲突次数。

(3) 试设计一个不会造成冲突的散列函数。

(4) 如果散列表大小为 17,上面 3 个问题的答案又是什么?

13. 设散列表为 HT[13],散列函数为 $h(K) = K \% 13$。用闭散列法解决冲突,对下列关键码序列 12、23、45、57、20、03、78、31、15、36 造表。

(1) 采用线性探查法寻找下一个空位,画出相应的散列表,并计算等概率下成功检索的平均检索长度和不成功检索的平均检索长度。

(2) 采用双散列法寻找下一个空位,再散列函数为 $rh(K) = (7 * K) \% 11 + 1$,寻找下一个空位的公式为 $h_i = (h_{i-1} + rh(K)) \% 13$, $h_1 = h(K)$。画出相应的散列表,并计算等概率下成功检索的平均检索长度。

14. 考虑下面 1~6 的一个排列 2、4、6、1、3、5。

(1) 在一个长度为 7 的散列表中使用这个排列实现伪随机探查,试分析将产生什么结果。

(2) 这个排列能解决基本聚集问题吗?

(3) 在伪随机探查中,对于长度为 M 的散列表,为什么要选择 $1 \sim M - 1$ 的一个"随机"排列?

15. 有一个 7 个槽的散列表(槽从 0 到 6 编号),使用散列函数 $h(k) = k \% 7$ 和线性探查,依次插入 3、12、9、2。

(1) 显示最后的结果散列表。

(2) 在插入值 2 的关键码之后,每一个空槽作为下一个被填充槽的概率是多少?

16. 给定关键码序列(26,25,20,33,21,24,45,204,42,38,29,31),用散列法进行存储,规定负载因子$\alpha = 0.6$。

(1) 试给出除余法的散列函数。

(2) 用双散列探查法解决碰撞,试画出插入所有的关键码后得到的散列表,并指出发生碰撞的次数。

17. 假定把关键码 K 散列到有 n 个槽(从 $0 \sim n-1$ 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?(即插入和检索操作能正常工作吗?)如果可以,它是一个好的散列函数吗?

注:函数 Random(n)返回一个 $0 \sim n-1$ 之间的随机整数(包含这两个数在内)。

(1) $h(k) = k / n$,其中 k 和 n 都是整数。

(2) $h(k) = 1$。

(3) $h(k) = (k + \text{Random}(n)) \% n$。

(4) $h(k) = k \% n$,其中 n 是一个素数。

18. 试设计一个简单、容易计算的散列函数,将 3 个字母的字符串映射为 $0 \sim n-1$ 之间的整数(包括 0 和 $n-1$)。并给出 $n = 11$、13、17、19 时,如下字符串的散列值:

PAL LAP PAM MAP PAT PET SET SAT TAT BAT

要求冲突尽可能的少。

19. 上机做一个小试验,测试对下列串运行 ELFhash 的结果是什么?

(1) HELLO WORLD。

(2) WORLD, HELLO!。

(3) Data Structure。

(4) Data Structure and Algorithms。

上 机 题

1. 在编译系统中,采用"符号表"来保存源代码中的描述符信息。编译器在分析阶段收集信息放入符号表,在综合阶段使用符号表中的信息生成目标代码。

这种符号表是一种有重复关键码的字典。字典记录以描述符作为关键码,属性值包括描述符类型(int、float 等)、用途(函数名、变量名、标号)、描述符内存地址、描述符作用域等。因为在不同的程序块中可以定义同名的描述符,所以符号表中必然存在多个记录具有相同的关键码,查找结构应该是最新插入的元素。

编写程序对读入的 C++程序建立这样的符号表字典。在程序块结尾处需要删除本块定义的描述符,最终所有元素都被删除掉。试比较下面这两种实现方案的效果。

(1) 采用线性表实现符号表字典。

(2) 采用散列方法实现符号表字典。

2. 实现一个文档检索系统。该系统能够按照给定的关键码检索文档,支持"与"、"或"、"差"等基本逻辑操作。

3. 使用桶式散列实现一个存储在磁盘中的数据库。

定义一个长 128 B 的,其中 4 B 是关键码,120 B 是数据,其余 4 B 预留用来存储必要的信息以支持散列表。

散列表中的一个桶是 1 024 B,因此桶内可以放 8 个记录。散列表要包含 27 个桶(总共 216 个记录的空间,槽从位置 0 ~ 215 编号索引)。后接一个溢出桶,在文件中的记录位置 216。

散列函数是 $K(R)\%197$。(这意味着表中最后 9 个槽不作为任何记录的基位置)

冲突解决方案是在桶内回转的线性探查。例如,如果一个记录散列到第 5 个槽,冲突解决过程会尝试以 5、6、7、0、1、2、3、4 的顺序把这个记录插入表中。如果一个桶满了,就把这个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。当进行测试时,考虑系统的设计用于一次存储 100 个左右的记录。

4. 设计一个简单的职工管理系统,对单位职工进行管理,包括插入、删除、查找、排序等功能。职工对象包括姓名、性别、出生年月、学历、职务、住址、电话等信息。系统功能要求如下:

(1) 新增一名职工:将新增职工对象按姓名以字典方式插入到职工管理文件中。

(2) 删除一名职工:从职工管理文件中删除符合某些条件的职工。

(3) 查询:从职工管理系统中查询符合条件的某些职工。

(4) 修改:检索某个职工对象,对某些属性进行修改。

(5) 排序:按某种需要对职工对象文件进行排序。

5. 封锁管理子系统的模拟实现。

封锁管理子系统通过加锁来控制用户对系统资源的并发使用。用户使用系统资源前必须申请封锁,即给出申请封锁的对象资源号、封锁方式、用户名。其中,资源号是取值为 1 ~ 999999 的正整数,封锁方式为 S(共享)或 X(排他)。两种封锁方式的相容矩阵如图 10.14 所示。

子系统受理封锁请求,根据所保存的封锁状态信息决定该请求是否能够获得封锁,进行相应处理,并向用户反馈处理结果。如果获得封锁,则赋给该请求一个批准号,可以使用该资源;否则需要进入等待队列(赋给该请求一个等待号)。

用户结束对某资源的使用后,应释放封锁(给出封锁对象的资源号、封锁批准号)。系统受理解锁请求,做解除封锁处理,并检查正在等待封锁该对象的封锁请求,决定能否使他们获得封锁。

系统中的资源对象很多,仅需对当前有封锁要求的对象保存封锁状况信息。处理加锁、解锁请求时必须能迅速找到有关对象的封锁状况信息,以进行相应的处理。为满足此要求,可采用散列表加链表的数据结构,如图 10.15 所示。

	S	X
S	OK	No
X	No	No

图 10.14 封锁方式的相容矩阵

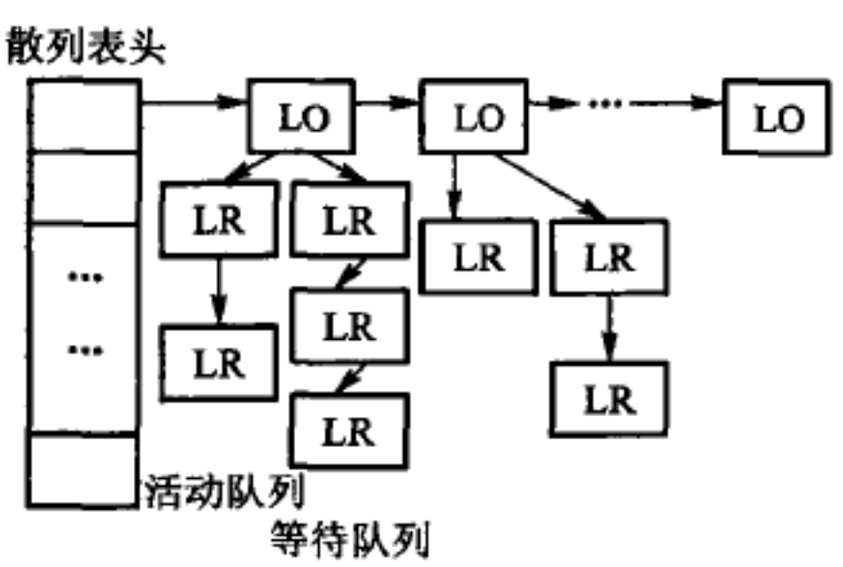


图 10.15 封锁管理子系统示意图

其中散列表的元素对应为封锁对象,以对象的资源号为散列函数的自变量(即关键码值)。散列表中元素仅为一个指向封锁对象链表的指针。LO 为封锁对象结点,对应于同一散列地址的封锁对象链接到一个链表中。LR 为封锁请求结点,每个封锁对象结点带两个封锁请求队列:活动队列中为当前持有对该对象的封锁请求,等待队列中为正在等待对该对象进行封锁的封锁请求。LO 结点和 LR 结点均向子系统自己管理的可利用空间表

申请。可以参考第 12.2.2 节关于可利用空间表的讨论。

试设计并实现散列表、LO 链表、LR 活动队列、LR 等待队列、可利用空间表的结构和基本运算。在此基础上使封锁管理子系统能提供以下功能：

(1) 受理用户从终端输入的封锁请求进行相应处理，并给出反馈信息。

(2) 受理用户从终端输入的解锁请求，并进行相应处理。

(3) 显示封锁状况信息(用户从终端输入资源号，子系统显示该对象的 LR 活动队列和 LR 等待队列)。

第11章 索引技术

许多计算机应用程序都以大型数据库为中心，这些数据库因太大而只能存放在外存中。这一类型的应用程序一方面需要具备高效检索记录的能力，另一方面，也必须有插入、更新、删除记录的能力，因此必须采用索引技术以组织这些存储在外存中的大量记录，并支持高效的操作。

索引相当于图书馆的卡片目录。有了卡片目录之后，就不需要在整个书库中搜索某一本书，而可以在目录中检索到该书的位置，然后在书库中找到它。同样，有了索引之后，就不需要在整个文件或者数据库中搜索某一个特定值，而可以通过索引确定该值的位置，然后直接访问数据库的记录。

在介绍索引文件结构之前，先了解以下这些基本术语：

• 输入顺序文件(entry - sequenced file)按照记录进入系统的顺序存储记录。输入顺序文件的结构相当于一个磁盘中未排序的线性表，因此不支持高效检索。有人可能考虑按照检索关键码的顺序存储记录，但是人们往往需要检索多个关键码，而记录在磁盘中的排列顺序只有一种。因此，事实上，仅仅改变记录在磁盘中的顺序，无法适应所有的检索需要。

• 主码(primary key，即主关键码)是数据库中的每条记录的唯一标识。例如，高校教师信息的主码可能是职工工资号。本章简称为关键码，为方便讨论大多采用整数类型。

• 辅码(secondary key)是数据库中可以出现重复值的码。例如，教师所在院系可以是高校教师信息的辅码。大多数检索都是利用辅码来完成的。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。

• 索引(indexing)是把一个关键码与其数据记录位置相关联的过程，往往是记录在主文件中的信息。索引文件(index file)则是用于记录这种“关键码——记录位置”联系的文件组织结构，其(key, pointer)二元组将每个关键码和一个指针关联，该指针指向主要数据库文件(也称为“主文件”)中的完整记录。索引文件并不需要重新排列记录在磁盘中的顺序。一个数据库可能有多个相关的索引文件，每个索引文件往往支持一个主码或辅码域，可以通过该索引文件高效访问主文件中的相应记录。索引技术是组织大型数据库的一种重要技术。

当数据库文件中的记录不按照关键码的顺序排列时(例如按照加入的顺序排列)，需要对每一个记录建立一个索引项，这样建立的索引被称为稠密索引(dense index)。如果记录在磁盘中是按照关键码的顺序存放，则可以把记录分成多个组(块)，对一组记录建立一个索引项，这种索引称为稀疏索引(spare index)。稀疏索引项的指针指向这一组记录在磁盘中的起始位置。

数据结构设计的重要目标之一是提高操作速度，对数据库而言，主要是检索速度。实际上索

引是为检索服务的，而排序又是为索引服务的。10.3 节所讨论的散列方法，其实是对关键码的索引，与关键码对应的记录数据可能存放在其他地方；局部平衡的红黑树、平衡的 AVL 树、自组织的伸展树等二叉搜索树具有良好的检索性能，非常适合于基于内存的索引。

本章首先介绍几种重要的线性索引、多分树静态索引、B/ B^+ 树动态索引等基于磁盘的索引文件组织方式，并对其性能进行比较和分析；然后，介绍适用于低基数数据域的位图索引技术以及文本信息检索中的签名文件索引；最后，将介绍一种降低了平衡要求，检索、插入和删除效率都比较高的常用二叉搜索树索引结构——红黑树。

11.1 线性索引

线性索引(linear index)的索引文件被组织成一组简单的(key, pointer)对的序列。线性索引文件按照关键码的顺序进行排序；文件中的指针则指向存储在磁盘上的文件记录起始位置或者主索引中主码的起始位置，如图 11.1 所示。只要内存空间允许，把索引存储在内存中能大幅提高检索速度。

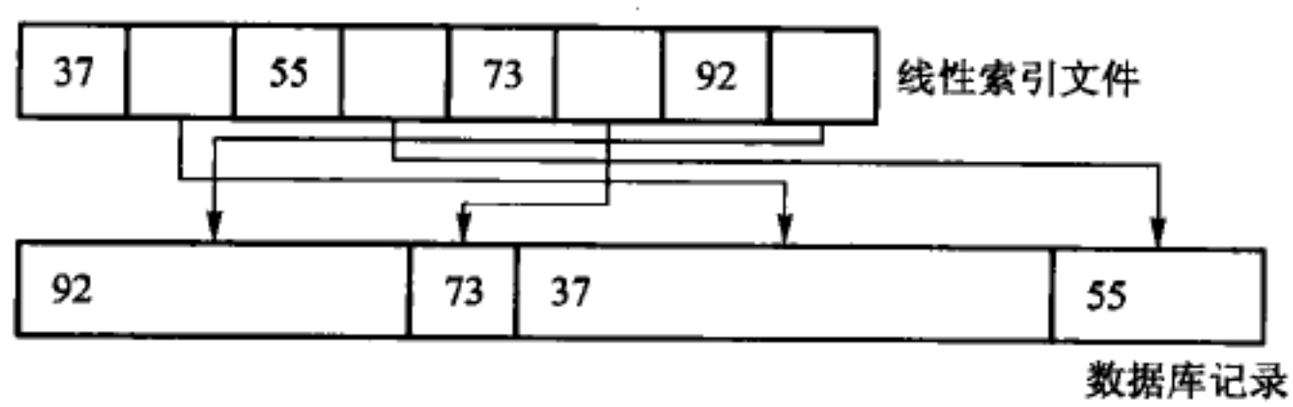


图 11.1 线性索引，图中的数据库记录不等长

线性索引有很多优点，它为访问变长数据库记录提供了一种简单而有效的处理方式。首先，索引文件中的关键码记录和指针记录都是定长的，而且索引中的关键码是有序的，可以很容易地访问索引文件中的记录。其次，使用线性索引可以对数据进行高效检索和随机访问，因为对线性索引可以使用二分检索(binary search)，而二分检索的时间复杂度是 $O(\log n)$。第三，在一些需要对数据进行顺序处理的应用中，线性索引文件也具有较好的效果，例如比较操作和批处理操作。第四，相对于后面将要介绍的其他类型的索引文件来说，线性索引也是一种节省空间的索引方法。

线性索引的主要问题是：如果存在大量的数据库记录，那么索引文件可能会因为太大而无法整个存储到内存中，这样在一次检索过程中有可能多次访问磁盘，从而影响检索的效率。为此，引入二级线性索引来解决这个问题。

二级线性索引文件中同样是(key, pointer)序列。每一条二级线性索引记录都对应于一个线性索引文件的磁盘块，关键码值与对应的线性索引文件的磁盘块中第一条记录(从物理位置上看的第一条)的关键码的值相同，而指针则指向相应线性索引文件的磁盘块的起始位置。

例如，如果磁盘块的大小是 1 024 B，线性索引中每个(key, pointer)记录需要 8 个字节，那么每个磁盘块可以存储 128 条这样的记录。假设线性文件索引中包含 10 000 条记录，那么该线性索引占用 79 个磁盘块，相应的，二级线性索引文件中有 79 项记录。如图 11.2 所示，二级线性索引记录中的关键码与相应磁盘块中第一条记录的关键码的值相同，指针则指向相应磁盘块的起始位置。

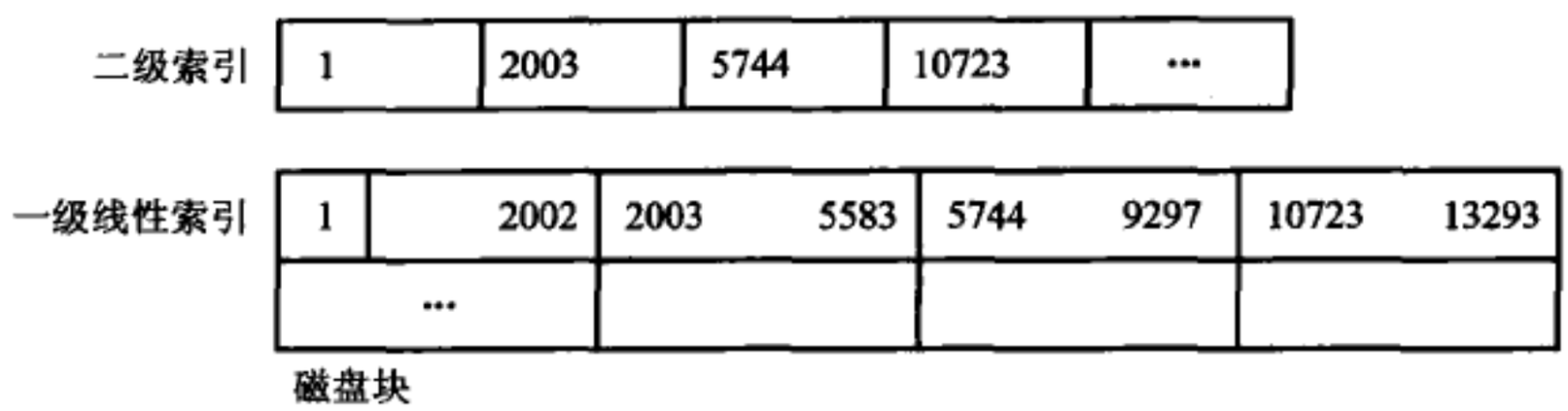


图 11.2 二级索引文件

在检索时，线性索引文件并不被读入内存，被读入内存的是二级线性索引文件。在上例中，如果检索关键码为 2555 的记录，首先在内存中的二级线性索引文件中找到关键码的值小于等于 2555 的最大关键码所在的记录——关键码为 2003 的记录；再根据记录中的指针找到其对应线性索引文件的磁盘块，并把该块读入内存；然后，再按照二分法对该块进行检索，找到记录在磁盘上的位置；最后，把所需记录读入，完成检索操作。由于二级索引往往存储在内存中，因此通常只需要访问两次磁盘即可，一次读入线性索引文件，一次读入数据库记录。

线性索引的另一个问题是：一旦在数据库中插入或者删除了记录，就必须更新线性索引。这是因为，线性索引是顺序存储的，一旦插入或者删除记录，那么插入点(删除点)后的所有记录都必须移动位置。而且，如果有二级线性索引，所有与其相关的二级线性索引也都必须更新。因此，线性索引的插入和删除操作代价巨大。

11.2 静态索引——多分树

静态索引在文件创建、初始装入记录时生成，一旦生成就固定下来，在系统运行(例如插入和删除记录)过程中索引结构并不改变，只有当文件再组织时才允许改变索引结构。

组织索引一般不用二叉树而采用多分树(multi - way tree)，这样能大幅减少访问外存的次数。图 11.3 所示的树原来是一棵较大的二叉树，图中所示有六层，若每存取一个结点需访问一次外存，则检索这棵二叉树需要访问 6 次外存索引(还需要 1 次访问外存数据块的操作)；若把结点按图中的虚框进行组合，则每个组合块包含 7 个结点，把这样的组合块作为新的结点，则构成了多分树——每个结点有 8 个分支的八分树，检索这棵多分树只需访问两次外存索引。

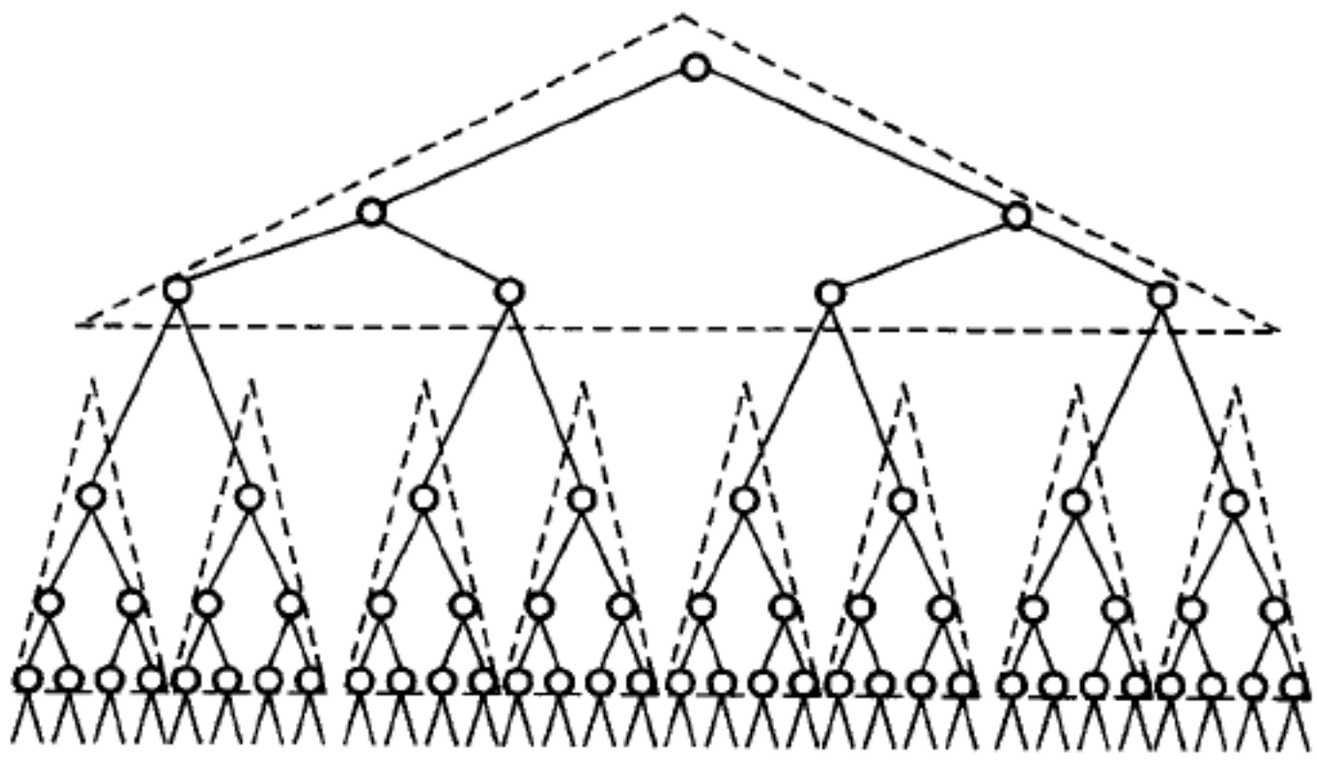


图 11.3 多分树

当然,对多分树来说,如果每个结点更大一些,即带有更多的分支,例如 128 个分支,就能以更少的外存访问次数来完成检索。但是,结点一大,就需要较大的缓冲区,而且读入一个结点也需要较多时间,因此结点的大小要适宜。一个结点最好能够放在一个磁盘块中。

多分树的叶结点存放数据记录,这些存放数据记录的结点合在一起,称为"数据基本区";多分树的非叶结点存放其各子树结点中最大(或最小)的关键码,这些结点合在一起,称为"索引区"。在运行过程中,如果数据基本区中应该插入某记录的结点已满,就发生溢出,此时把溢出的记录存放到另开辟的溢出区中,而不改变索引的结构。

ISAM(indexed sequential access method——索引顺序存取方法)是多分树的应用。ISAM 是解决需要频繁更新的大型数据库的一个早期尝试,采用多级索引:主索引、柱面索引、磁道索引。若文件很大,占用很多柱面,此时主索引也可以包含很多层。在采用基于 B^+ 树的 VSAM 技术之前的 20 世纪 60 年代,IBM 公司曾经广泛地采用 ISAM 技术,较好地解决了大型数据库频繁更新数据的问题。

11.3 倒排索引

在实际应用中,不仅需要按关键码值检索记录并进而找到记录中的其他数据项值,而且经常需要按照记录中的其他数据项,即属性值来检索记录,这样就需要按属性值建立索引。这种索引表中的每一项都包括一个属性值和具有该属性值的各记录地址。由于这种索引不是由记录来确定属性值,而是由属性值来确定记录的位置,因而称为倒排索引(inverted index)。带有倒排索引的文件称为倒排索引文件,简称倒排文件(inverted file)。

11.3.1 基于属性的倒排

在实际应用中,经常要求检索结构中某个或若干个属性满足一定条件的结点,把这类检索简

称为基于属性的检索。

例如,表11.1是某高校教师登记表,每位教员的数据是表中的一个记录,以职工号(EMP#)为主关键码,另外有姓名(Name)、系(Department)、职称(Profession)、专长(Specialty),地址(Address)等属性。关键码EMP#可以唯一标识一个教员,而其他的属性则不能,学校里可能有同名教员,此时就会有两个或更多的教员记录的Name数据域具有相同值。计算机系有几十名教员,也就是说有几十个记录的Department域值是"计算机"。

表11.1 数据库主表

Emp#	Name	Departement	Profession	Speciality	Address
119	王卓	数学	讲师	代数	B102
132	王亮	数学	助教	几何	E220
155	李宇	数学	教授	代数	C105
208	赵亮	物理	助教	力学	C211
211	张伟	物理	讲师	原子物理	D508
310	周兵	计算机	讲师	英语	B423
330	孙丽	计算机	教授	软件	A108
341	何江	计算机	助教	计算机	F406
421	刘阳	外语	助教	英语	E310
455	刘珍	外语	讲师	法语	A225
…					

对上述教师登记表可能提出这样的检索要求:列出职称为"讲师"的所有职工号,列出计算机系擅长"英语"的教员名单等,这些要求都是基于属性的检索。用前面讨论过的检索技术,也可以处理这些要求,但是代价比较大。处理前一个检索需要顺序扫描所有记录,把Profession域值为"讲师"的那些记录的EMP#域值记下来,处理后一个检索也需要顺序读取所有记录,对每个记录先检查其Department数据域,对那些值为"计算机"的记录再看它的Specialty域值是否为"英语",若是则记下Name域值,这样处理的效率很低。

为有效地处理基于属性的检索,考虑使用特殊的存储形式:倒排表(inverted list)。倒排表是基于属性的倒排,即在保留原表的同时,对于感兴趣的属性(即可以用来作为检索参数的属性)的每个值都建立一个称做倒排表的线性表,并存放与此属性值相对应的所有关键码值。

例如,对于上述教师登记表可以分别建立关于系(Depatment)、职称(Profession)、专长(Speciality)的倒排表,如表11.2所示。

表 11.2 数据库倒排表

Department list	EMP#
数学	0155,0132,0119
物理	0208,0211
计算机	0330,0310,0341
外语	0421,0455
Profession list	**EMP#**
教授	0155,0330
讲师	0211,0119,0455,0310
助教	0421,0208,0132,0341
Specialty list	**EMP#**
代数	0155,0119
几何	0132
力学	0208
原子物理	0211
软件	0330,0341
英语	0421,0310
法语	0455

有了倒排表,再来处理上面提出的检索要求,效率就会比较高。对于前一个要求只要查 Profession 属性为"讲师"的倒排表,得到职工号集合{0211,0119,0455,0310};对于第二个要求,首先通过 Department 倒排表得到属性为"计算机"的职工号集合{0330,0310, 0341},通过 Specialty 倒排表得到属性为"英语"的职工号集合{0421,0310},两个集合的交集是{0310},按交集的关键码值检索主表,得到相应的 Name 域值。

由此可见,采用倒排文件,可以先对相应的倒排表进行交、并运算,然后再按结果集合中的主码在主文件中的地址指针(为了提高可读性,表 11.2 中用主码代替指针)去存取记录,这样就大幅减少了记录的存取量,提高了检索速度。

倒排表不能代替原来的主表,而只作为主表的辅助表,需要与主表并存。因为若取消了主表,那么当提出"打印出职工号为 0330 的教员的一切有关信息"这样的要求时就不好处理。

在倒排表系统中,进行插入和删除时,不但要改变原主表,而且要改变倒排表的表目。例如,删除关键码为 0132 的记录,就牵涉在 Department 值为"数学"的倒排表里,在 Profession 值为"助教"的倒排表里,在 Specialty 值为"几何"的倒排表里都要删掉 0132 这个表目。

倒排表系统能够对于基于属性的检索进行高效的处理。但是,保存倒排表需要一定的存储代价;完成倒排表集合的运算需要较大的内存空间;更新主文件时,维护倒排索引的工作量也比较大。

11.3.2 对正文文件的倒排

随着电子数据的高速增长,各种电子格式的文件数量不断增加。对正文文件的索引技术也受到了越来越多的关注。为了更好地利用越来越多的文本数据,对文本数据的检索和管理已经

越来越重要。

正文索引(text indexing)处理的就是"建立一个数据结构以提供对文本内容的快速检索"。

建立正文索引有两种方法:词索引(word index)和全文索引(full - text index)。词索引的基本思想是:把正文看做由符号和词所组成的集合,从正文中抽取出关键词,然后用这些关键词组成一些适合快速检索的数据结构。这项技术适用于多种文本类型,特别是那些可以很容易就解析成一组词的集合文本,例如英文的文本。但是,这项技术不适合生物学数据和一些东方语言书写的文本。对建立了词索引的正文进行查询,查询词需要被限制在词索引的关键词范围内。

全文索引的基本思想是,把正文看做一个长字符串。在数据结构中记录子字符串的开始位置,这样查询就可以针对正文中的任何子字符串。这种方法使得人们可以对每一个字符建立索引,从而使查询词不再限于关键词。但是,建立全文索引通常需要更大的空间,在很多应用中受到了限制。

倒排文件是使用得最广泛的词索引。一个倒排文件就是一个排序关键词的列表,其中每个关键词指向一个倒排表(posting list),指向该关键词出现的文档集合以及在该文档中的位置。对于词 t_i,它的 posting list 可表示为 $\{(d_{i1},f_{i1},a_{i1}*)+(d_{i2},f_{i2},a_{i2}*)+\cdots\}$,其中 d_{i1}、d_{i2} 为文档标号,f_{i1}、f_{i2} 为词 t_i 在文档中的出现次数(依此对 posting list 中出现的文档进行排序),$a_{i1}*$、$a_{i2}*$ 为词 t_i 在文档中出现信息(位置、权重等)的罗列。

为节省存储空间、提高查询响应时间,可把 posting list 的出现信息 f_{i1}、f_{i2}(频率信息)去掉,简单表示成 $\{(d_{i1},a_{i1}^*)+(d_{i2},a_{i2}^*)+\cdots\}$。甚至还可以把位置信息去掉,而只保留该关键词所出现的文档编号。

表 11.3 是 6 个文档的内容,表 11.4 是相应的倒排文件内容,记录了关键词在文档中出现的单词位置,实际应用中可以记录关键词在文件中的字节位置,以便于快速定位。

表 11.3 原文件内容

文档编号	文本内容
1	Pease porridge hot, please porridge cold,
2	Pease porridge in the pot,
3	Nine days old.
4	Some like it hot, some like it cold,
5	Some like it in the pot,
6	Nine days old.

表 11.4 正文倒排文件的词索引

编号	词语	(文档编号,位置)
1	cold	(1,6) (4,8)
2	days	(3,2) (6,2)

续表

编　号	词　语	(文档编号,位置)
3	hot	(1,3) (4,4)
4	in	(2,3) (5,4)
5	it	(4,3) (4,7) (5,3)
6	like	(4,2) (4,6) (5,2)
7	nine	(3,1) (6,1)
8	old	(3,3) (6,3)
9	pease	(1,1) (1,4)
10	porridge	(1,2) (1,5) (2,2)
11	pot	(2,5) (5,6)
12	some	(4,1) (4,5) (5,1)
13	the	(2,4) (5,5)

建立一个正文倒排文件通常需要以下几个步骤:

(1) 对文档集中的所有文件都进行分割处理,把正文分成多条记录。如何切分正文记录取决于程序的需要。一条记录可以是正文中的一个定长块、一个段落或者一个章节。如果正文本身就是一组文档,那么一条记录还可以是一篇文档。

(2) 给每条记录赋一组关键词。这些关键词以人工或者自动的方式从记录中抽取出来。通常,还需要确定一组停用词(Stopword),例如一些助词和介词,这些词不会被索引。在英语等西文中还经常用到"抽词干"(Stemming)技术,例如 computer、computing、computed 等词,都处理为只剩下"comp"词干。抽词干技术使得人们可以使用一个关键词来表示一组语义上相近的索引和检索项。在汉语等东方语言中,因为词与词之间没有空格分割,要把词从句子中划分出来还需要进行切词(segmentation)处理,这需要自然语言理解技术(natural language processing,是人工智能的一个研究分支)的支持。

(3) 得到各个关键词的集合,对于每一个关键词,得到其倒排表,然后把所有倒排表存入文件,并记录每个倒排表在索引文件中的开始位置以及每个表的大小。建立起倒排文件之后,对关键词的检索通常被分成两步:

第一步,在倒排文件中检索关键词。

第二步,如果找到了关键词,则获取对应的倒排表,并获取倒排表中的记录。

为了在索引文件中高效检索关键词,通常使用另一个索引结构进一步对关键词表进行有序索引。对关键词表进行索引的常用技术有散列和 B^+ 树等,如果关键词表可以全部放入内存,也可以用二叉搜索树建立索引。

虽然倒排索引支持高效的检索，而且被广泛地用于很多文本数据库系统中，但是倒排索引所支持的检索类型还是非常有限的。首先，只有在索引文件中的关键词才可以用作检索词；其次，某些应用在倒排文件中的索引效率并不高，例如对短语、习惯用语和完整字符串的检索；第三，倒排文件需要的空间代价往往很高。一般情况下，如果把文本中每个词都抽出来做关键词，一个未压缩的倒排文件的大小往往是原始文件的50% ~300%。

11.4 动态索引

动态索引结构在文件创建、初始装入记录时生成，在系统运行过程中插入或删除记录时，索引结构本身也可能发生改变，以保持较好的检索性能。

11.4.1 *B* 树

二叉搜索树 BST 不能保证平衡，有可能退化为线性表，从而降低检索效率。外存索引结构的平衡性能非常重要，因为退化的树结构将导致树形索引的失败，而面临巨大的访外开销。由11.2 节中多分树的讨论可以看出，外存索引树结点在不超过磁盘页块大小限制的前提下，应该尽量增加每个结点内关键码的数目。

第12章将要介绍的 AVL 树使得 BST 始终保持 $\Theta(\log n)$ 级的平衡状态，但其频繁的插入删除调整对于外存树索引来说，也需要很大的 I/O 访问量，即磁盘访外开销。

怎样保持多分树高度上的平衡性，不让它退化为线性？怎样使得多分树结点中关键码尽可能多，而插入删除操作也不需要太多调整树结构的动作？

1970 年 R. Bayer 和 E. Mccreight 提出了一种 *B* 树（balanced tree，国内也有人称“B - 树”，其中“ - ”是英文的连字符，不要说成“*B* 减树”），它是一种平衡的多分树，适用于组织动态的索引结构。

定义一个 m 阶的 *B* 树需满足下列条件：

（1）每个结点至多有 m 个子结点。

（2）除根结点和叶结点外，其他每个结点至少有$\lceil m/2 \rceil$个子结点。

（3）根结点至少有两个子结点（例外情况：根可以为空，或者独根）。

（4）所有的叶结点在同一层，可以有$\lceil m/2 \rceil - 1$ 到 $m-1$ 个关键码。

（5）有 k 个子结点的结点（分支结点）恰好包含 $k-1$ 个关键码。

最底层的叶结点的指针指向空（对应于失败检索），空指针数目正好等于树中所包含关键码的数加1。

在 *B* 树中，每个结点内部的关键码从小到大排列。一个包含 j 个关键码，$j+1$ 个指针的 *B* 树结点如图11.4 所示。这 j 个关键码正好划分了 $j+1$ 个子结点的关键码值域。其中 K_i 是关键码

值，$K_1 < K_2 < \cdots < K_j$，P_i 是指向包括 $K_i \sim K_{i+1}$ 之间的关键码的子树的指针。事实上，对应于每个关键码还有一个隐含域，存储该关键码所代表的记录信息，一般情况下是指向该记录在主文件中位置的指针。

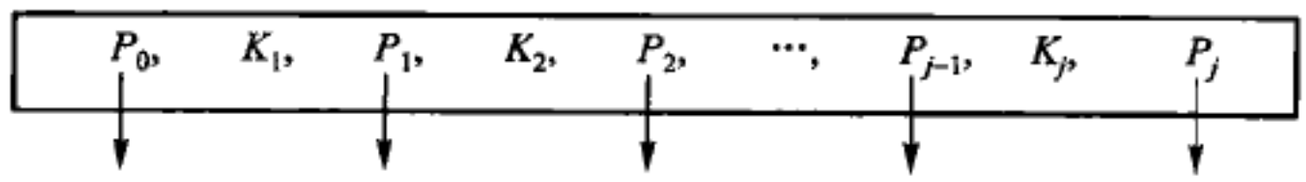


图 11.4 B 树结点的一般形式

类似于二叉树中序周游那样来访问定义 B 树的周游：

- "0"递归周游 P_0 所指向的子树。
- "1"访问 K_1。
- "1′"递归周游 P_1 所指向的子树。

…

- "j"访问 K_j。
- "j′"递归周游 P_i 所指向的子树。

此时，可以得到 B 树中所有关键码的递增序列（假设关键码不重复），此序列中的相邻关键码按照大小顺序形成了前驱后继关系。

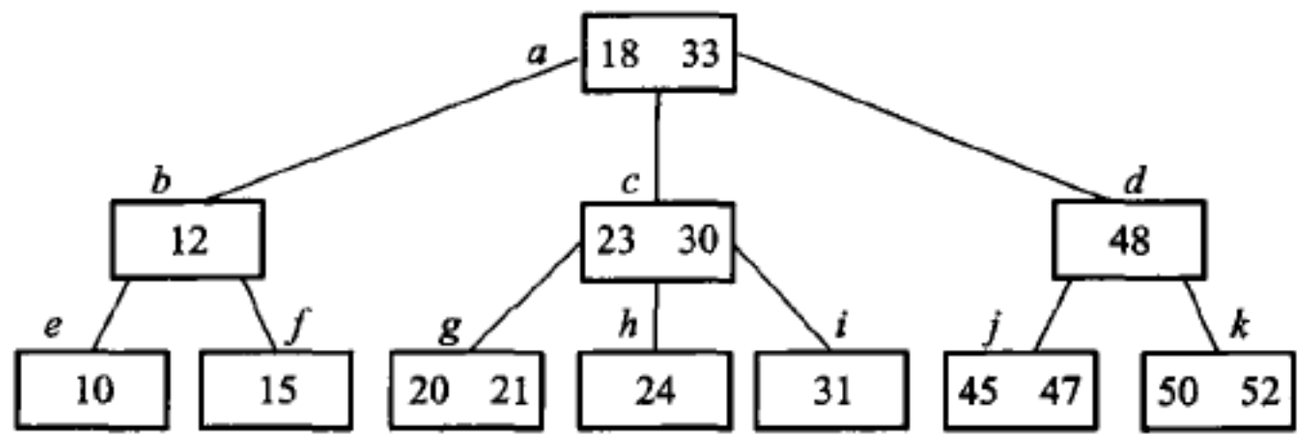
由 11.4.3 节的 B 树性能分析可知，实际应用中 B 树的阶非常高，可以到达 100 左右。为了画图方便，本章给出的是阶比较小的示例。

图 11.5(a)所示为一棵 3 阶的 B 树，每个结点的子结点的个数在 2($\lceil 3/2 \rceil$)和 3 之间，因此每个结点可包含 1 或 2 个关键码；所有的叶结点都在第 2 层（第 0 层为根），每个结点中关键码按字典顺序排列。根结点中，18 和 33 之间的指针，指向的是大于 18 小于 33 的那些关键码所在的 c 子树。有些教材像图 11.5(b)那样画出叶结点指向的空结点。B 树的每个关键码，本身都隐含着指向该记录在主文件中存储位置的指针，如图 11.5(c)所示。这在 11.4.3 节计算 B 树的存储效率时非常重要。因为 B 树只是一种索引结构，索引结构本质上索引(key，pointer)，高效的索引结构能够基于关键码 key 快速地定位(key，pointer)对，然后根据指针 pointer 检索出对应于该关键码的记录值。

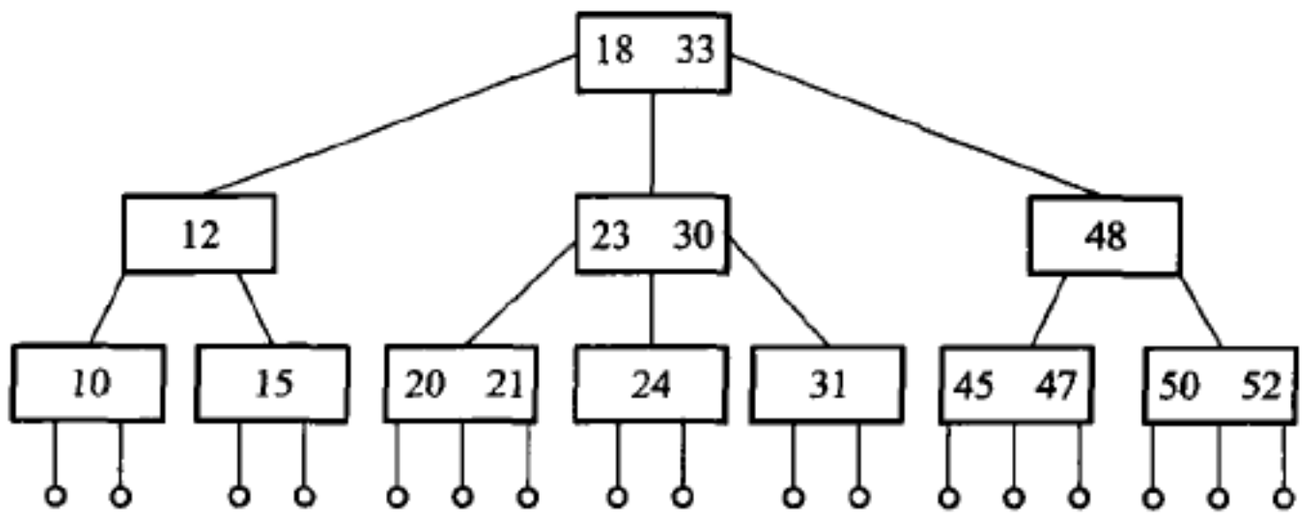
1. B 树的检索

在 B 树里检索给定的关键码是交替的两步过程。首先把根结点读出来，在根结点所包含的关键码 $K_1, \cdots, K_j$ 中查找给定的关键码值（当结点包含的关键码不多时，就用顺序检索；当结点包含的关键码数目较多时，可以用二分检索），若找到则检索成功。否则，一定可以确定待查关键码值是在某个 K_i 和 K_{i+1} 之间（因为在结点内部的关键码是排序的），于是取 P_i 所指向的结点继续查找，直到找到；或 P_i 为空指针，表示检索失败。

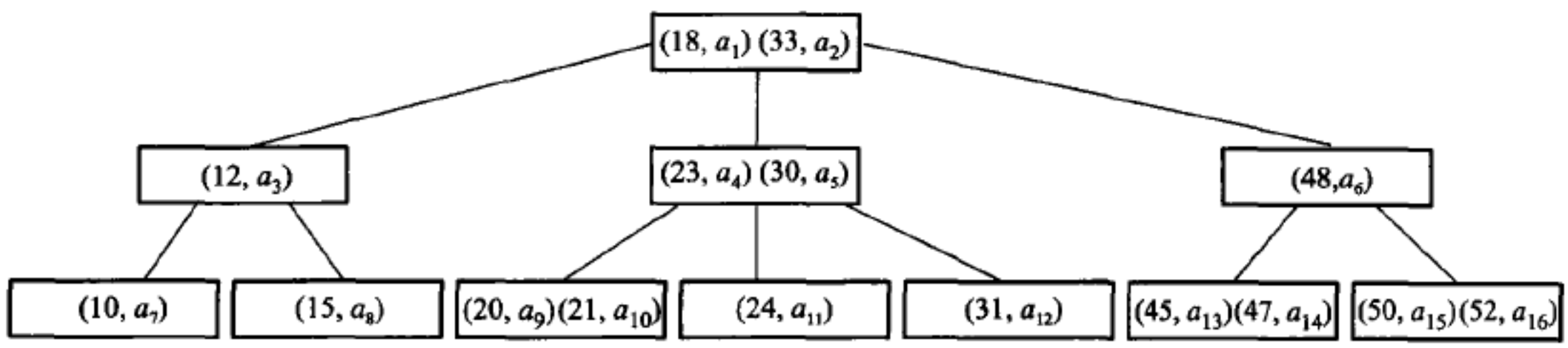
设 B 树的高度为 h（独根树高为 1），则在自顶向下检索到叶结点的过程中可能需要进行 h 次读索引盘，再加上根据关键码中隐含指针找到该记录主文件位置的 1 次读数据盘，最多需要



(a) 3 阶 B 树



(b) 画出外部空结点的 3 阶 B 树



(c) B 树的隐含指针，其中 a_1~a_{16} 代表关键码所在主文件中具体磁盘块的索引地址

图 11.5 3 阶 B 树

$h+1$ 次访问外存(简称“访外”)。

图 11.5 中检索关键码 31,需要 3 次访问索引树结点(a、c、i),外加 1 次读关键码 31 所在主文件的数据盘 a_{12},共 4 次访外。而检索关键码 12,需要 2 次访问索引树结点(a、b),外加 1 次读关键码 12 所在主文件的数据盘 a_3,共 3 次访外。

2. B 树的插入

在 B 树中插入一个关键码的方法也很简单。插入可能导致 B 树朝着根的方向生长。首先,检索此关键码是否在 B 树中,若在则拒绝插入;若不在,则最后找到的叶结点即为待插入位置。如果插入后该叶结点中关键码个数小于 m,则插入完成。否则,关键码个数等于 m,产生上溢出。这种情况下,取这 m 个关键码的中位数为分界码,叶结点分裂为分别具有$\lceil (m-1)/2 \rceil$和$\lfloor (m-1)/2 \rfloor$个关键码的两个结点;分界码插入到父结点中;如果父结点没有发生上溢出,则插入操作结束;否则该分裂过程继续往根的方向传递,有可能一直分裂到根。

若向图 11.5 的 B 树插入关键码 14，则直接插入到 15 所在的叶结点 f，只需要 1 次写操作（从根开始检索到结点 f，需要 3 次读盘操作）。继续插入关键码 55，因为要插入的结点 k 已包含 2 个关键码，插入新结点将导致溢出。在这种情况下，要把结点 k 分裂为两个（新申请一个结点 k'），并把（50，52，55）的中位关键码 52 拿出来插到结点的父结点 d 中，结果如图 11.6 所示。

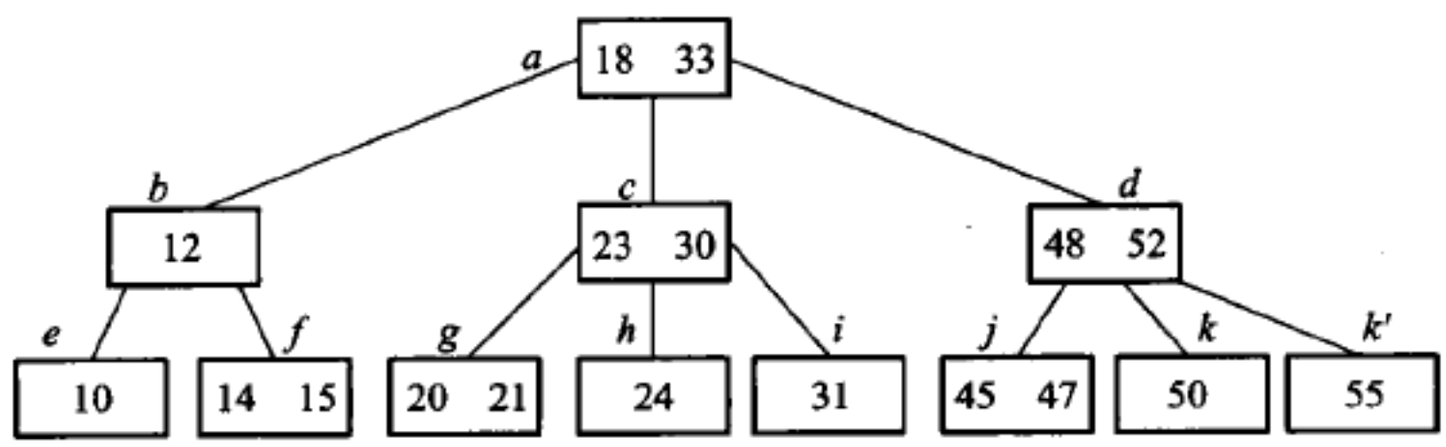


图 11.6 在图 11.5 的 3 阶 B 树中插入关键码 14、55 后的结果

父结点也可能是满的，此时就需要再分裂，再往上插。最坏的情况，这个过程可能一直传到根，如果需要分裂根，由于根是没有父结点的，这就需要建立一个新的根结点，此时整个 B 树就增加了一层。图 11.7 就是根结点分裂的示例。在图 11.6 的 B 树继续插入关键码 19 后，结点 g 分裂为两个结点 g、g'，（19、20、21）的中位关键码 20 上传，与父结点 c 一起继续分裂为 c、c' 两个结点，中位关键码 23 上传到根，引起根结点分裂为 a、a' 两个新结点，并且中位 23 成为新树根 t 中的关键码。

图 11.7 的插入过程有 10 次对 B 树的访外操作，其中读盘 3 次（a、c、g），写盘 7 次（g、g'、c、c'、a、a'、t）。这里不考虑对主数据文件的访外操作，也不考虑申请新磁盘块的开销。如果图 11.6 和图 11.7 是两次临近的操作，也就是说最近访问过的磁盘块还存放在内存缓冲区，那么读根结点 a 的操作就不需要访外，只需要对 c、g 访外。这里没有考虑子结点有父指针信息的情况，即不考虑父结点磁盘块发生变化所造成的父指针修改。

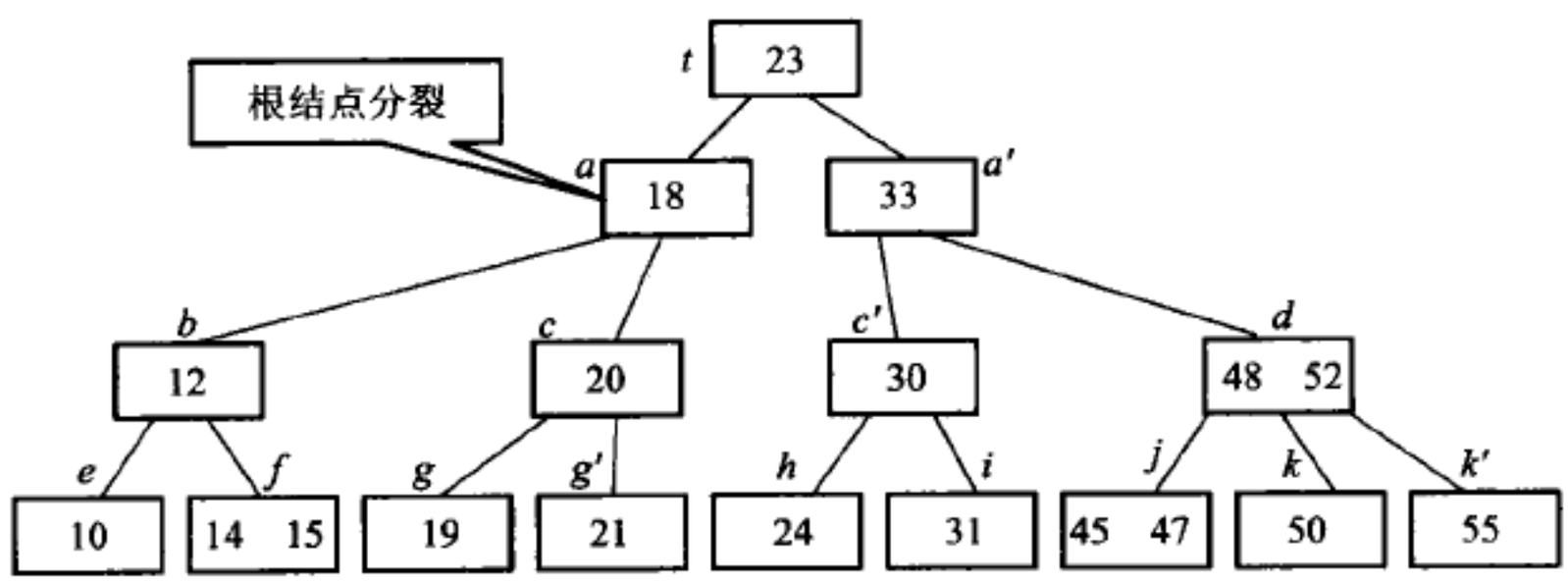


图 11.7 在图 11.5 的 3 阶 B 树中插入关键码 19 后导致根分裂

假设内存工作区足够大，使得在向下检索时读入的结点在插入后向上分裂时不必再从磁盘读入。B 树插入操作的读盘次数与检索相同。如果不分裂，直接把新关键码所插入到的结点写到磁盘，最少写盘次数为 1 次。一次插入操作最多读/写次数可以如下：假设总共 h 层，每层都需

要分裂(包括根),每分裂一个非根结点要向磁盘写出 2 个结点,分裂根结点(最后一次)要写出 3 个结点,因此

$$
\begin{aligned}
\text{插入操作最多访外次数} &= \text{找插入结点向下读盘次数} \\
&\quad + \text{分裂非根结点时写盘次数} \\
&\quad + \text{分裂根结点时写盘次数} \\
&= h + 2(h-1) + 3 \\
&= 3h + 1。
\end{aligned}
$$

根据上面的公式计算,图 11.7 根也产生分裂,可以计算出访外为:$3 \times 3 + 1 = 10$ 次。

根结点允许的子结点数目范围在 $2 \sim m$ 之间,而内部结点的子结点数目在$\lceil m/2 \rceil \sim m$ 之间是因为插入算法分裂到根结点时,会分为 2 个子结点并产生一个新根。如果要求与其他内部结点具有同样的出度,就需要对 B 树进行比较大的调整,这就违背了 B 树操作的局部调整原则。而且,实际运行时,根结点往往在内存中保存(当然最终它还是要写回外存,因此也有 m 的上限),不需要遵守内部结点的子结点数目下限$\lceil m/2 \rceil$的约束;大部分内部结点在外存中,需要保证存储块至少半满以折衷存储效率和操作效率,因此限制在$\lceil m/2 \rceil \sim m$。所以,根结点的子结点数据范围在 $2 \sim m$ 之间。根结点不可能只有一个子结点,这样太浪费存储空间,而且会相应地增加检索时间。

B 树插入时,如果发生上溢出则分裂,而不是送给邻居一些关键码。这是由于插入操作可能发生在创建阶段,分裂的结点很快就会增加新的关键码;而送关键码涉及读和修改邻居结点,访外开销大,且操作麻烦。

3. B 树的删除

删除的过程与插入类似,但比插入稍微复杂些。如果删除的关键码不在叶结点层,则需要先把此关键码与其 B 树中的后继对换位置,然后再删除该关键码。在 B 树中,一个关键码的后继就是比它大的那些关键码中值最小的那一个。非叶层待删关键码的后继关键码一定处于叶层,即待删关键码的右指针所指子树中最左结点的最小关键码。如果删除的关键码在叶结点层,则直接从叶结点中删除,这可能导致此结点所包含的关键码的个数小于$\lceil m/2 \rceil - 1$,产生下溢出。

这种情况下,考察该结点的直接左兄弟或右兄弟(习惯上先左后右),从兄弟结点借若干个关键码到该结点中。实际上,往往是被删除结点和被借结点的关键码,还有父结点中的分界码一起排列,取中位数为新分界码,尽量使得两个新兄弟结点所含关键码个数基本相同(这样可以延迟下一次删除引起的下溢出)。当在兄弟结点的关键码个数也很少,刚好等于$\lceil m/2 \rceil - 1$ 时,不能借关键码。这种情况下,要把删除关键码的结点,它的左/右兄弟(先左后右)结点及父结点中的分界关键码合并为一个结点。合并的情况导致父结点中减少了一个关键码,这样有可能导致进一步的合并。合并如果一直传递到根结点,将导致根结点与两个子结点进行合并,形成新的根结点,从而使整个树减少了一层。

删除过程中,一般情况下,出现下溢出时先看是否能借关键码,借不了再考虑合并。如果阶

比较小，例如阶为3（关键码数在1～2个），阶为4（关键码数在1～3个），此时删除可能会出现空结点的情况，可以先考虑是否能合并，再考虑借关键码，这样有可能降低树的高度。实际应用中，B 树的阶很大，不会出现删除一个关键码使得结点变空的情况。

对图11.8(a)所示的5阶 B 树，删除关键码120。首先，120与结点 h 中它的后继134交换，在叶结点 h 中删除120后关键码数下溢出。向左邻居 g 借关键码，两个兄弟和父结点 c 中的临时分界码134一起排序，取(108,110,115,118,134,146)的中位数118成为新分界（第二个中位数，使得左结点比右结点多一个关键码），形成如图11.8(b)所示的 B 树。

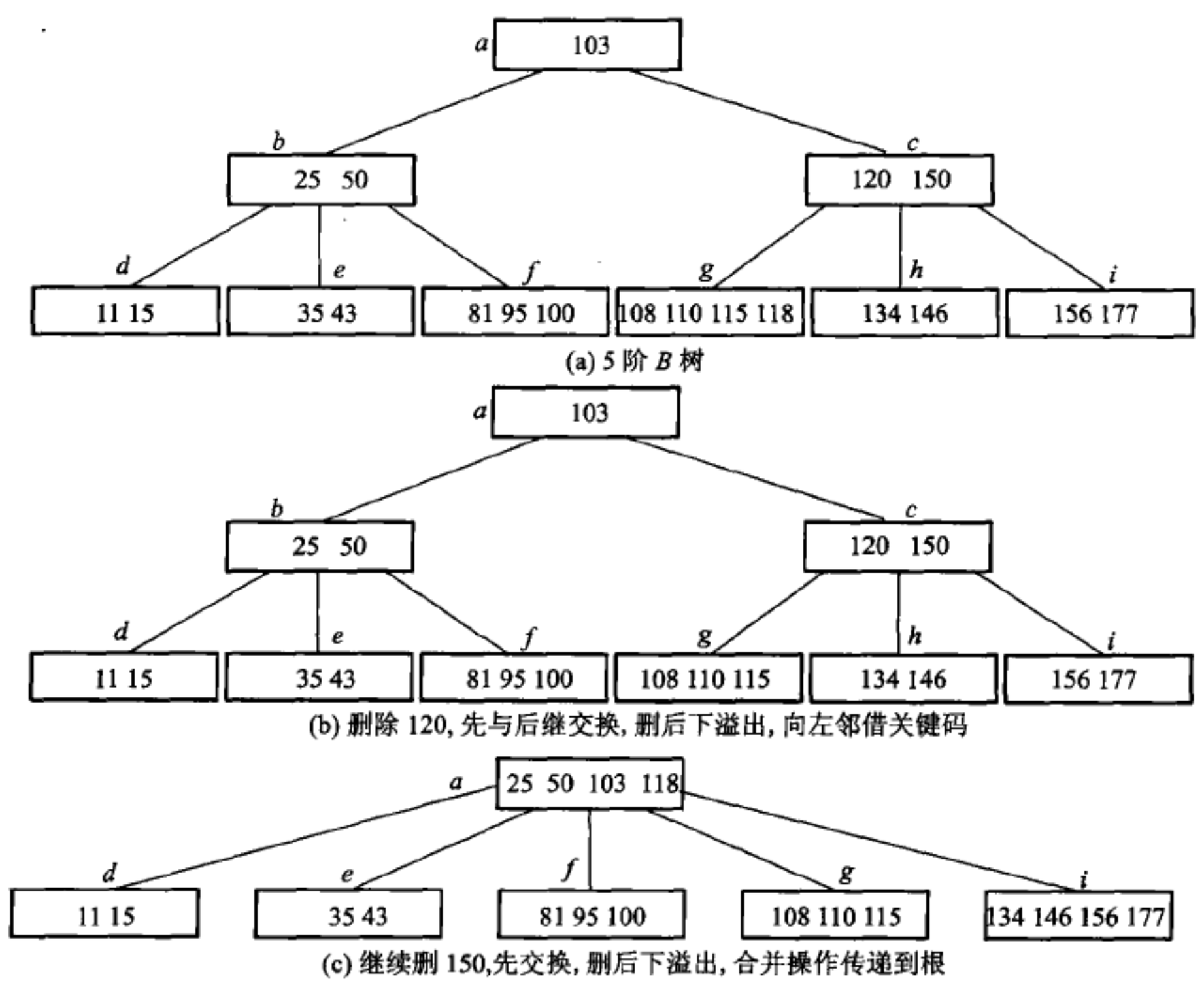


图11.8 5阶 B 树，连续删除关键码120、150示意图6

继续删除150，150与后继156交换，换到叶结点 i 中。删除叶结点 i 中的150后，只剩下177一个关键码，产生下溢出。此时其左兄弟 h 无关键码可借，因此 h、156、i 合并为新 i 结点，释放空结点 h。合并后，父结点 c 也下溢出，b、103、c 继续合并为新根结点，释放空结点 b、c，产生如图11.8(c)所示的 B 树。

上述两个连续删除的操作，其访外次数是一起考虑的。删除120，读盘（a、c、h、g），写盘 h、g、c。继续删除150，可以认为相关的 a、c、h 已经在内存，只要新读入 i 即可；合并时写出 i，读入 b，写出 a。

4. *B* 树的性质

如果 *B* 树的阶为 3,则称为"2 - 3 树"。反过来,*B* 树也可以看做 2 - 3 树的一种推广。*B* 树有如下性质:

(1) *B* 树总是树高平衡的,所有的叶结点都在同一层。

(2) 更新和检索操作只影响一些磁盘块,因此性能很好。

(3) *B* 树把相关的记录(即关键码具有接近的值)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。这样既能够保证一定的空间利用率,又使得插入和删除不必像静态多分树那样频繁地修改结点,从而总体上减少了磁盘读取数。

11.4.2 *B*+ 树

在实际应用中产生了一些 *B* 树的变形。例如,每个结点的"充满度"可以不同;树在不同层上可有不同的阶;每个关键码的长度可变;把关键码全部顺序存储在叶结点上,而索引部分是每个结点内最大(或最小)关键码的复写。下面介绍一下应用得很广泛的 *B*+ 树。

m 阶 *B* + 树的结构定义如下:

(1) 每个结点至多有 *m* 个子结点。

(2) 每个结点(除根外)至少有$\lceil m/2 \rceil$个子结点。

(3) 根结点至少有两个子结点(例外情况:根可以为空,或者独根)。

(4) 所有的叶结点在同一层,可以有$\lceil m/2 \rceil$到 *m* 个关键码,叶结点包含全部关键码以及相应的记录信息(例如该记录的主文件中位置指针),叶结点之间可以用双链表顺序链接。

(5) 有 *k* 个子结点的结点(分支结点)必有 *k* 个关键码。

图 11.9 所示为一个 3 阶($m = 3$)*B* + 树的例子,其所有的关键码均出现在叶结点上,上面各层结点中的关键码均是下一层相应结点中最大关键码的复写(当然,也可采用"最小关键码复写"原则,本教材以"最大关键码复写"为例)。由图中可以看出,*B* + 树的构造是由下而上的,*m* 限定了结点的大小,自底向上地把每个结点的最大关键码或最小关键码复写到上一层结点中。在图中,也可以不画叶结点的链接。

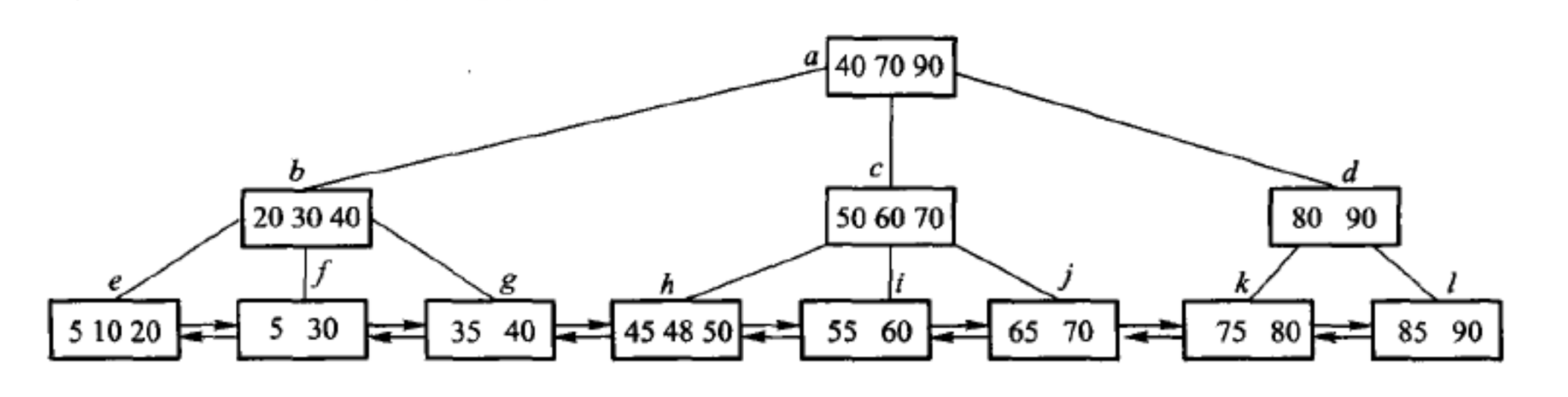


图 11.9 3 阶 *B* + 树

$B+$树的检索、插入、删除过程基本上与 B 树类似，也有一些差别。

1. B+树的检索

在 $B+$树中有两个头指针，一个指向 $B+$树的根结点，另一个指向具有最小关键码的叶结点。可以对 $B+$树进行随机检索和顺序检索两种形式的访问。

随机检索过程从根结点开始，自顶向下，若在上层已找到待查的关键码并不停止，而是继续沿指针向下一直查到叶结点层的这个关键码。事实上，对应于叶结点的每个关键码还有一个隐含的指针，指向该记录在主文件中的位置（其中叶结点的阶可以也与内部结点不一样）。随机检索适合于查找任意关键码，也是插入和删除操作的基础。

顺序检索适合于进行范围查询，可以从叶头结点开始，也可以根据随机检索查找到的范围内的第一个叶结点，通过叶结点链，顺序读取范围内的其余数据。

2. $B+$树的插入

当插入时，一个全满的 m 个关键码的结点再加一个新关键码，就要分裂为两个结点，分别具有$\lceil (m+1)/2 \rceil$和$\lfloor (m+1)/2 \rfloor$个关键码，并要保证上层结点中有这两个结点的最大关键码复写。

图 11.10 是在图 11.9 的 3 阶 $B+$树中插入关键码 15 后的结果。沿 a、b、e 查找，找到叶结点；在叶结点 e 中插入 15，发生上溢出，(5,10,15,20)分裂为两个结点 e 和 e'；左结点 e 的新索引码 10 上传到父结点 b，(10,20,30,40)分裂为 b 和 b'；左结点 b 的新索引码 20 上传到父结点 a，(20,40,70,90)分裂为两个结点 a 和 a'，并产生两个新索引码 40 和 90 作为新根结点 t 的关键码，树增高一层。整个过程读盘 3 次(a、b、e)，写盘 7 次(e、e'、b、b'、a、a'、t)。这里没有考虑子结点有父指针信息的情况，即不考虑父结点磁盘块发生变化所造成的父指针修改，也没有考虑叶结点链接变化所需的访外。

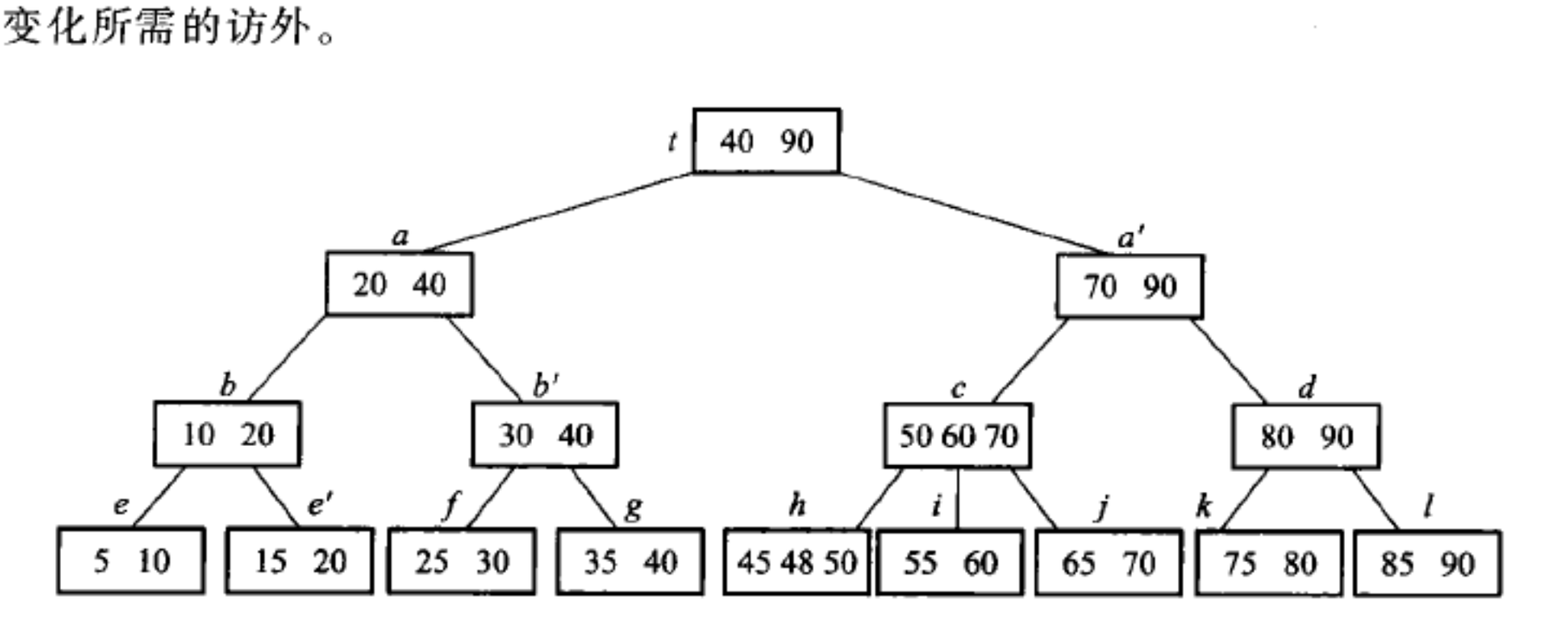


图 11.10 在图 11.9 的 3 阶 $B+$树中插入 15 后，树增高一层

注意：如果新插入关键码大于根中最大的分界码，就应该从根结点一直顺最大分界码所指向的指针，找到树中最右的结点，将新关键码插入到该结点中。如果取最小值作为分界码，则是把小于根中最大值的新关键码插入到最左结点。

3. $B+$树的删除

当删除时，关键码在叶结点层删除后，把真正的最大关键码替代其父结点中相应的分界关键码值。如果不影响检索，也可以保留原来的分界复本。删除造成下溢出（关键码数小于$\lceil m/2 \rceil$）时，其处理过程与B树类似，即首先考虑借兄弟结点的关键码，借不了则需要与左（或右）兄弟结点合并。

图11.11所示为在图11.9的3阶$B+$树中删除关键码75后的结果。沿a、d、k查找，找到叶结点；在k中删去75，发生下溢出，剩余关键码80与右邻l结点合并为新k(80,85,90)；父结点d中原分界码80删除，d结点下溢出，借左邻c的关键码，c和d的关键码平分；父结点a中的分界码70修改为60。整个过程读盘5次（a、d、k、l、c），写盘4次（k、d、c、a），释放了一个空磁盘块l。

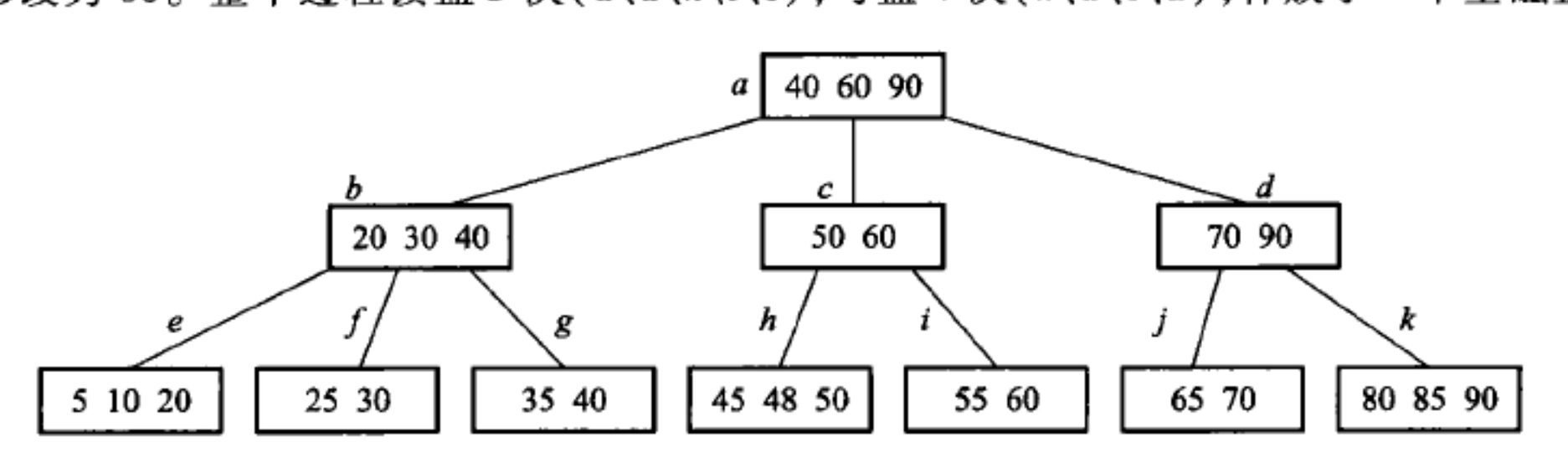


图11.11 从图11.9的3阶$B+$树中删除75后

4. 混合型$B+$树

可以看到，$B+$树的分支结点实际上只起到了引导向叶方向检索的作用，叶结点中才真正存放数据记录在主文件中的地址，内部分支结点与叶结点可以不同阶。图11.12所示为一种混合型$B+$树：分支层采用图11.5的B树，阶为4（2～4个子结点，1～3个关键码），分界码是下层子结点最小关键码的复写；叶层采用图11.9所示的$B+$树形式，阶为5（3～5个关键码）。如果只有一个叶结点，则不需要建立内部分支层。

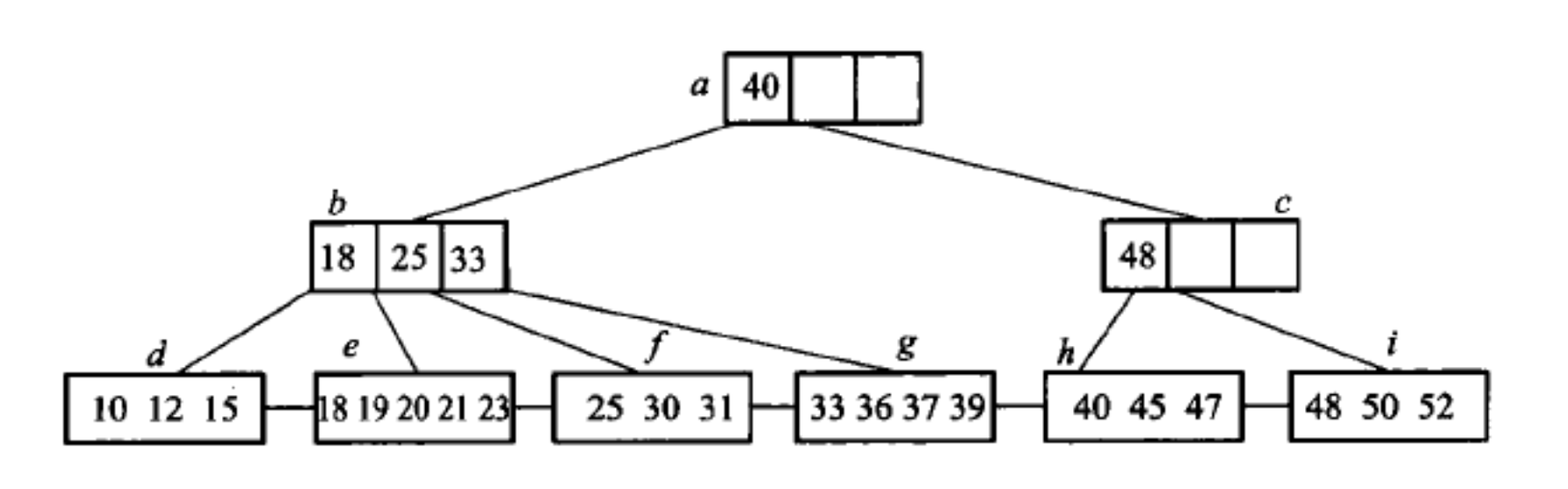


图11.12 混合型$B+$树，内部结点阶为4，叶结点阶5

这种$B+$树的检索也必须到达相应的叶结点。例如，在图11.12所示的$B+$树中查找值40，首先找到根结点a开始检索，存储在a中的值40只是作为占位符，表示大于或等于40的关键码值可以在c子树中找到；$40<48$，因此进入c的第一个分支h，找到关键码值40及其主文件索引指针。

插入过程类似于B树插入过程。首先，找到应当包含记录的叶结点L。如果叶结点L还没

满，就把新记录添加进去，不影响到 $B+$ 树的其他结点。如果叶结点已经满了，就将其分裂成两个结点（在两个结点之间平均分配记录），然后把两个叶结点（包括新结点）中最小关键码的那两个复本写入父结点替代原来那一个复本。这也可能会引一系列的分裂，甚至传递到根结点，从而引起 $B+$ 树增加一层。注意，内部结点的分裂可参照 11.4.1 节中的 B 树那样进行处理。

在图 11.12 中插入关键码 22，首先查找 a、b、e；将 22 插入 e 后，e 分裂为两个结点 e 和 e'；右叶结点 e' 的最小关键码 21 复写到 b，引起 b 的分裂，(18,21,25,33) 分成 (18,21) 和 (33)，中位数 25 上传到根结点 a。插入的结果如图 11.13 所示。

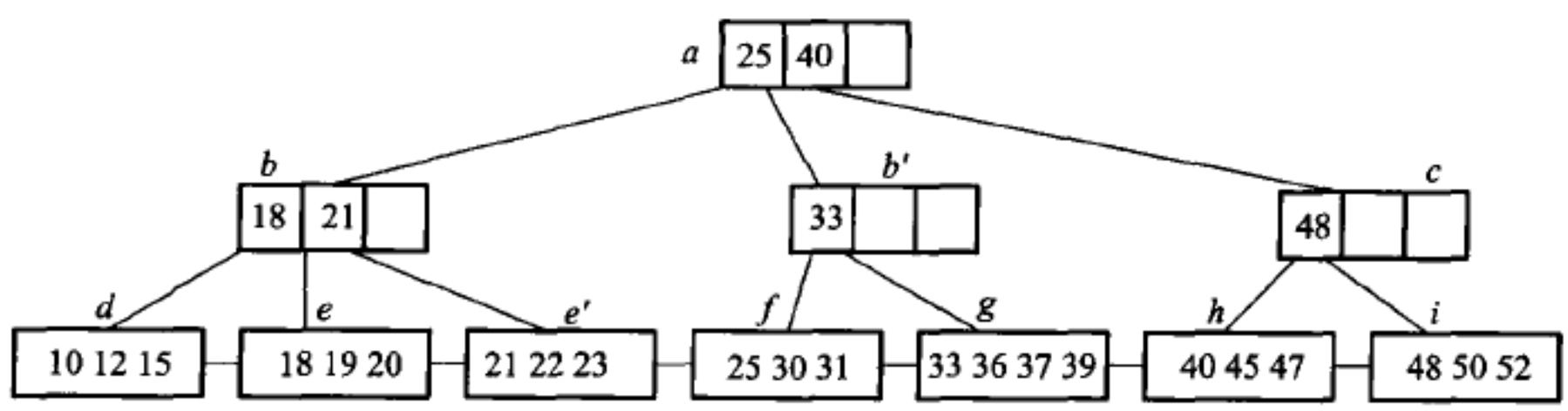


图 11.13 在图 11.12 混合型 $B+$ 树（内部结点阶为 4，叶结点阶 5）中，插入 22 后的结果

要从 $B+$ 树中删除关键码 K，首先找到包含 K 的叶结点 L。如果叶结点 L 超过半满，则只需直接删除 K（可以不修改父结点中的复写关键码，因为目前不影响检索）。如果删除后使得叶结点中的记录数目下溢出，则需要向兄弟叶结点借关键码或与兄弟合并。

在插入了 22 之后的图 11.13 中删除关键码 40，叶结点 h 下溢出，其右邻 i 没有关键码可借（不能向堂兄弟 g 借关键码），因此 h 和 i 结点合并，形成新叶结点 h。父结点 c 需要删除一个分界码，也发生下溢出，b' 和 c 合并，父结点 a 中的分界码 40 修改为 45 并下移。形成如图 11.14 所示的结果。

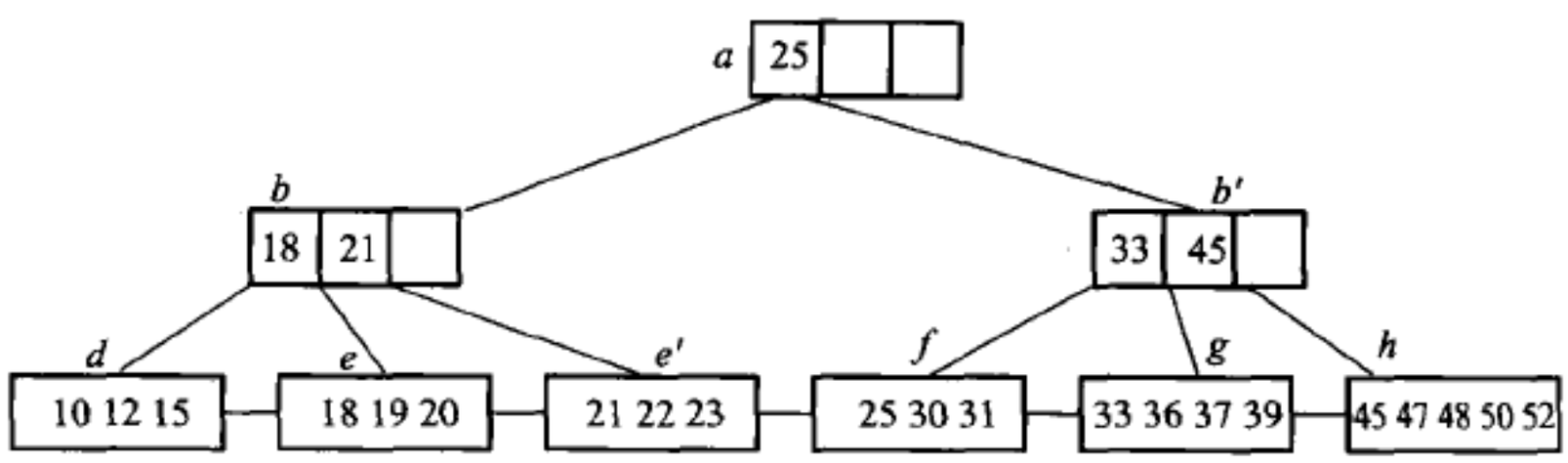


图 11.14 在图 11.13 混合型 $B+$ 树（内部结点阶为 4，叶结点阶 5）中，删除 40 后的结果

11.4.3 B 树的性能分析

下面来讨论一下 B 树的检索效率及应用情况。

1. B 树的层数及 $B/B+$ 树索引的文件规模

设 B 树包含 N 个关键码，那么对应于图 11.5(b) 中那些小圆圈的扩展外部空结点（即失败

检索)就有 $N+1$ 个。设外部结点都在第 I 层(根为第0层),因为根至少有两个子结点,因此第一层至少有两个结点。

除根和树叶外,其他结点至少有$\lceil m/2 \rceil$个子结点。因此,第二层至少有$2\lceil m/2 \rceil$个结点,第三层至少有$2\lceil m/2 \rceil^2$个结点……,第 I 层(图11.5(b)那样扩展的外部结点层)至少有$2\lceil m/2 \rceil^{I-1}$个结点,于是有$N+1 \geqslant 2\lceil m/2 \rceil^{I-1}$。即

$$I \leqslant 1 + \log_{\lceil m/2 \rceil}\left(\frac{N+1}{2}\right)$$

这意味着若 $N=1999998$,$m=199$,则 I 至多等于4。因为对于一次检索至多进行 I 次存取,因此这个公式保证了 B 树的检索效率是相当高的。

下面再从另一个角度来估计 $B+$ 树的索引规模。假定磁盘块大小为4 096个字节,且整数型关键码值占4个字节,指针占8个字节。不考虑存储块块头信息所占的空间,满足$4m+8(m+1) \leqslant 4096$的最大整数值 m 就是 $B+$ 树的阶,$m=340$。假若磁盘块的平均充满度介于最大和最小中间,即平均有255个指针。对于3层的 $B+$ 树,根结点有255个子结点,有$255^2=65025$个叶结点;每个叶结点也有255个索引记录,即可以有255^3约16.6×10^6个指向记录的指针。也就是说,记录数小于等于16.6 MB的文件都可以被3层的 $B+$ 树容纳,3次访外就可以获得关键码在主文件的地址。实际上,对 $B+$ 树的访问还可以减少到一次磁盘I/O访外操作。因为 $B+$ 树树根通常处于内存中,而且现在系统内存空间都比较大,$B+$ 树的第二层结点完全保存在缓冲区中也是合理的。这样,只需要读一次叶结点磁盘块。

2. B 树进行插入时需分裂的结点个数的上限

设 p 是内部结点个数,N 是 B 树包括的关键码个数。当根包括一个关键码,除根外的所有内部结点都包括$\lceil m/2 \rceil-1$个关键码时 B 树所包括的总关键码个数最少,因此有$N \geqslant 1+(\lceil m/2 \rceil-1)(p-1)$,可得$p-1 \leqslant \frac{N-1}{\lceil m/2 \rceil-1}$。现在考虑最坏的情况,即从第二个结点开始,每个结点均是经过分裂而得到的,也就是说插入 N 个关键码的过程中进行了 $p-1$ 次分裂,那么每插入一个关键码平均分裂的结点个数为

$$s=\frac{p-1}{N} \leqslant \frac{N-1}{(\lceil m/2 \rceil-1)\cdot N} \leqslant \frac{1}{\lceil m/2 \rceil-1}$$

3. $B/B+$ 树的存储效率

$B+$ 树内部分支结点(非叶层)中的关键码不需要像 B 树那样带隐含指针。下面的分析表明,在磁盘页块容量一致、页块指针大小相同的情况下,索引同一个主文件,B 树比 $B+$ 树高,即 $B+$ 树索引效率更高。

设一个主文件有 N 个记录。假设关键码所占字节数与指针相同,一个磁盘页块可以存 m 个(关键码,子结点页块地址)二元对。$B+$ 树最大充盈度为100%,$B+$ 树最小充盈度为50%,假设 $B+$ 树平均充盈度为$(1+0.5)/2=75\%$,即平均每个结点有$0.75m$个子结点,则 $B+$ 树的高度为$\lceil \log_{0.75m} N \rceil$。

由于 B 树结点除关键码和子结点页块指针外，还有指向本关键码在主文件中页块地址的隐含指针，即存储（关键码，隐含指针，子结点页块指针）三元组，因此一个 B 树结点最多容纳 $2m/3$ 个这样的三元组，即相应的 B 树为 $0.67m$ 阶。假设 B 树平均充盈度也是 75%，则 B 树每个结点平均有 $0.5m$ 个子结点，B 树的高度为 $\lceil \log_{0.5m} N \rceil$。

4. $B/B+$ 树的应用

B 树只适用随机检索，不适于顺序检索；而 $B+$ 树把所有的关键码都存在叶结点上，可以把 $B+$ 树的叶层形成顺序链表，为顺序检索提供了方便。$B+$ 树这种索引顺序文件既能进行顺序存取、又能进行随机存取，在实际应用中基本上都是采用 $B+$ 树。

虚拟存储存取方法（Virtual Storage Access Method，VSAM）是 $B+$ 树应用的一个典型例子，它是一种索引顺序文件的组织方式，这种文件组织的实现使用了 IBM370 系列的操作系统的分页功能。这种存取方法与存储设备无关，与柱面、磁道等物理存储单位没有必然的联系，它的存储单位是“逻辑”的，即“虚拟”的。

11.4.4 动态索引和静态索引性能的比较

下面比较一下基于多分树的静态索引结构和基于 B 树及 $B+$ 树的动态索引结构。动态索引结构的优点是：能保持较高的检索效率，查找一个后插入的记录和查找一个原记录具有相同的速度；动态地分配和释放存储，使得保持平均 75% 的存储利用率；而且动态索引结构不需要进行文件再组织。因此，基于 $B+$ 树的 VSAM 通常被作为大型索引顺序文件的标准组织方式。而静态索引结构在多次插入、删除后，一方面溢出区中记录越来越多，溢出区拉链越来越长，大幅降低了检索效率；另一方面，有的数据区有很多空单元处于无用状态，严重影响空间利用率，这样发展到一定的时候就需要进行文件再组织。但是，动态索引存在下列问题：

（1）要考虑并行策略。

当某个用户在查找数据时，可能会有另一个用户在插入数据而修改了索引，以致使第一个用户找不到数据，而静态索引在修改时只要锁住正在修改的结点，其余的均不会改变，不存在并行策略问题。

（2）辅助索引维护困难。

对主文件的动态索引会给属性项索引（辅助索引，例如倒排索引）带来不便，因为索引的动态改变会造成关键码（连同其记录）改变其地址，这就要引起辅助索引指针的改变。而静态索引中记录是不移动的，当插入新记录时，可采用溢出区拉链的方法，因此辅助索引可以直接用其地址作为指针。

（3）索引层数多。

因为每个结点都可能要进行分裂，所以必须在上层中保留其指针，指针所占据的空间使得结点中所含关键码数目减少，增加了索引层数，影响了效率。而静态索引的指针不发生变化，因此许多指针可以省略，使得一个结点可以包含较多的关键码，从而减少了索引层数。

由于上述几个问题的存在，因此很多情况下静态索引在性能上超过了动态索引。但是，静态索引需要文件再组织，非常麻烦，并且开销很大。因此，动态索引的应用更广泛。尤其当系统根本没有时间进行文件再组织时，就更需要使用动态索引结构。

11.5 位索引技术

B/B+树对低基数的数据域几乎毫无价值。例如，性别属性只有男、女取值，这种情况不适合建立 *B/B*+树索引。

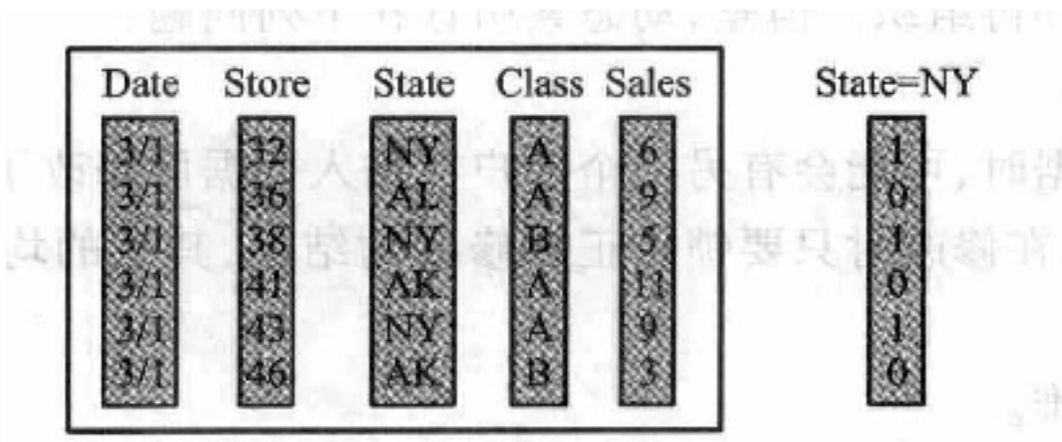
位图映射(bit map)索引对重要的数据域索引其所有可能取值对应的记录位向量。计算机可以迅速地执行位图映射操作，是一种非常高效的索引，对于低基数列特别适用。

签名文件(signature file)被广泛地应用于文本信息检索中，其核心思想是把文本的特征用签名(signature)位串来表示。签名文件比原文件小得多，因此可以提供更快速的检索。

11.5.1 位图索引

假设数据库的记录数为 n，其数据域 x 的一个位图索引是长度为 n 的位向量集合。每一个位向量对应于域 x 可能出现的值。如果第 i 个记录的数据域 x 值为 v，那么对应于值 v 的位向量在位置 i 上的取值为1；否则，该向量的位置 i 上的取值为0。

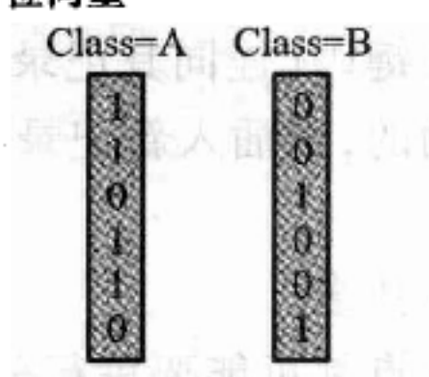
图11.15(a)所示为某连锁公司的百货销售数据库记录以及数据域 State = NY(纽约州)对应的位向量。图11.15(b)所示为域 State 的位向量集合，图11.15(c)所示为数据域 Class 的位向量集合。



(a) 百货销售数据库记录以及 State=NY (纽约州)的位向量

State=AK	State=AL	…	State=NY
0	0		1
0	1		0
0	0		1
1	0		0
0	0		1
1	0		0

(b) 数据域 State 的位向量集合



(c) 数据域 Class 的位向量集合

图11.15 百货销售数据库记录的位图索引示意

查询纽约州销售等级为 A 的记录，位向量 101010 和 110110 进行按位与运算，得到 100010，第 1 个和第 5 个为满足查询要求的销售记录。

位图索引也能够帮助回答范围查询，此时只需根据要求获取范围内的位向量，并进行相应的位运算即可。

对于具有规范结构的文件，也可以方便地找到指定的第 i 个记录。例如，假设文件的记录结构是(F,G)，其中数据域 F 为整型，G 为字符串型。一个文件有编号为 1 ~6 的记录，其值依次为(30,foo)、(30,bar)、(40,baz)、(50,foo)、(40,bar)、(30,baz)。

第一个数据域 F 的位图索引有 3 个值，对应于 3 个位向量，每个长度为 6。30 所对应的向量为 110001，因为第 1、第 2 和第 6 个记录其数据域 F 取值为 30；F = 40 的向量为 001010，F = 50 的向量为 000100。

如表 11.5 所示，数据域 G 的位图索引也有 3 个位向量，因为有 3 个不同的字符串出现。

表 11.5 文件中 G 数据域的示意位图索引

值	向　量
bar	010010
baz	001001
foo	100100

把数据库的一个记录看成一行，某个属性在各记录中的取值看成列，位图索引实质上是按列存储数据。一个位图索引需要的比特位总数是记录数和该数据域取值数的乘积。列数据比行数据更易进行压缩，可节省 50% 的磁盘空间，其索引空间比 $B+$ 树小。

当数据记录发生改变时，相应的位图向量都需要修改。尤其是增加或删除记录时，位向量长度也要有相应的改变，这给位图索引的管理带来了很大的麻烦。相比之下，$B+$ 树的修改是非常便利的。因此，很多情况下人们都选择 $B+$ 树，即使它比压缩位图占用更多的空间。

11.5.2 签名文件

签名文件(signature file)被广泛地应用于文本信息检索中。

最简单的签名文件采用二维数组实现。二维数组 M 的每一列代表一个关键词，每一行代表一个文档(也可以反过来)。数组中的元素 $M[i,j]$ 存储一个位标志，标志为“1”则表示第 i 个文档中存在关键词 j(为“0”则不存在)。例如，用户想检索出包含某 3 个关键词的文档；可以把这 3 个关键词所表示的 3 个列取出来，对 3 个位向量进行“并”操作，把那些位标志为 1 的文档号返回给用户即可。

目前，主流的签名文件技术，通过散列函数及重叠编码(superimposed coding)为每个文档产生一个称为签名(signature)的位串，然后顺序存入一个单独的文件(签名文件)中。首先，可以采

用停用词表(stop list)来忽略普通的词;同时,对英文词进行抽词干(stemming)处理,对中文进行切词(chinese segmentation)处理;最后,采用 hash 数值函数产生签名位串。保留词在文档中的位置信息,将每个词的签名串联起来就得到该文档的签名,这样可以产生无重叠编码的签名文件。

此外,人们还提出了具有更高检索速度的二级签名文件、签名树以及基于签名的分割等。

签名文件方法的优点在于实现简单,插入操作效率高,能够处理词的部分查询,能够支持不断增长的文件以及拼写错误的容错性,而且该方法易于并行化。其主要缺点就是对大文件的操作时间较长,不易处理日益增长的词汇表;签名集合的设计需要巨大的开销。

11.6 红 黑 树

红黑树(red - black tree)是一种扩充的二叉搜索树 BST,树中的每一个结点的颜色是黑色或红色。它利用对树中结点红黑着色的要求达到局部平衡,其插入和删除算法性能好、易实现,同时检索效率也比较高。

11.6.1 红黑树的定义

满足下列条件的二叉搜索树是红黑树:

(1) 每个结点要么是红色,要么是黑色(后面将对颜色进行说明)。

(2) 根结点永远是黑色的。

(3) 所有的扩充外部叶结点都是空结点,并且是黑色的。

(4) 如果一个结点是红色的,那么它的两个子结点都是黑色的(不允许两个连续的红色结点)。

(5) 结点到其子孙外部结点的每条简单路径都包含相同数目的黑色结点。

结点 X 的阶(rank,也称"黑色高度"):从该结点到其子树中任意外部结点的任一条路径上的黑色结点数量(不包括 X 结点,包括叶结点)。外部结点的阶是零,根的阶称为该树的阶。

在图11.16所示的红黑树中,黑方块是外部结点,黑圈是黑色结点(粗线是指向黑色结点的

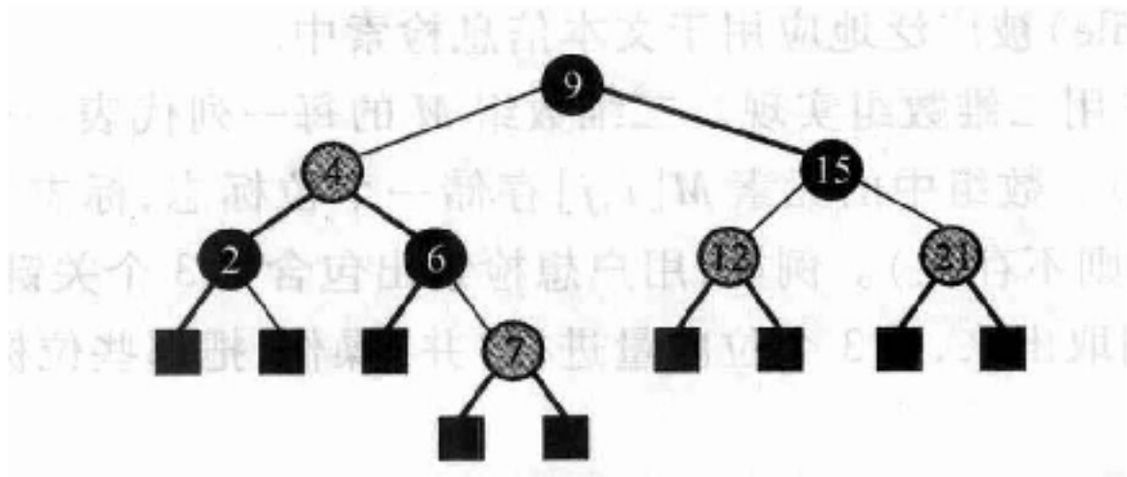


图 11.16 红黑树示意图

指针),浅色圈代表红色结点(细线是指向红色结点的指针)。该树中没有双红结点,根结点是黑色的。任何一条根结点到外部结点的简单路径上,都有两个黑结点,因此该树的阶为 2。根的左子结点 4 的阶是 2,右子结点 15 的阶是 1。

11.6.2 红黑树的相关性质

红黑树具有以下性质:

(1) 红黑树是满二叉树(空叶结点也看做结点)。

(2) 阶为 n 的红黑树,从根结点到叶结点的简单路径最短长度是 n,最长长度是 $2n$;或者说该树的树高(即根到最深的叶结点所经过的结点数)最小是 $n+1$,最大是 $2n+1$。

(3) 阶为 n 的红黑树,其含有的内部结点最少时是一棵完全满二叉树,此时内部结点数是 2^n-1。

(4) 含有 n 个内部结点的红黑树树高最大是 $2\log_2(n+1)+1$。

性质(1)从定义即可得出。

性质(2)容易证明:根据红黑树定义条件(4),结点到其子孙结点的每条简单路径中不可能有两个连续的红色结点。根据定义条件(5),最短的简单路径中全是黑色结点,此时路径长度是 n;由于最长的简单路径是红黑交替的,且根结点和叶结点必是黑色的,因此最多有 n 个红色结点,$n+1$ 个黑色结点(含根结点),路径长度是 $2n$。

性质(3)证明:当红黑树含有的内部结点最少时,则均为黑色结点;根据定义条件(4)和性质(1)易知,此时红黑树是一棵完全满二叉树,且树高是 n;又根据完全满二叉树性质可知,所有结点总数是 $2^{n+1}-1$,叶结点数是内部结点数加 1,可算得内部结点数是 2^n-1。

性质(4)证明:设红黑树的阶为 x,红黑树的树高是 h,内部结点数是 n。由性质(2)得 $h\leqslant 2x+1$,则 $x\geqslant(h-1)/2$;由性质(3)得 $n\geqslant 2^x-1$;从而得 $n\geqslant 2^{(h-1)/2}-1$,可得出 $h\leqslant 2\log_2(n+1)+1$。

虽然在定义红黑树时,将外部结点包括进来非常方便,但在执行过程中往往用空指针而不用物理结点来描述这些外部结点。由于指针颜色与结点颜色是紧密联系的,因此对于每个结点,需要存储的只是该结点的颜色或指向其子结点的指针颜色。

可以使用对普通 BST 的检索算法来完成对红黑树的检索,由性质(4)可知其检索长度为 $O(\log_2 n)$。

11.6.3 插入结点算法

与二叉树检索树(BST)的插入算法类似,红黑树中插入新记录时,首先检索到该记录所在的位置,时间代价为 $O(\log_2 n)$。然后,插入记录,把记录着色为红色。不同的是,红黑树还要检查

插入记录是否影响树的平衡。如果新增结点的父结点是黑色，则算法结束；否则，根据红黑树定义条件(3)。此时，红黑树不平衡，需要调整，调整方法分情况讨论如下：

情况1：新增结点的叔父结点是黑色。如图11.17所示，设新增结点是X(红色)，父结点是A(红色)，祖父结点是B(黑色)，叔父结点是C(黑色)。

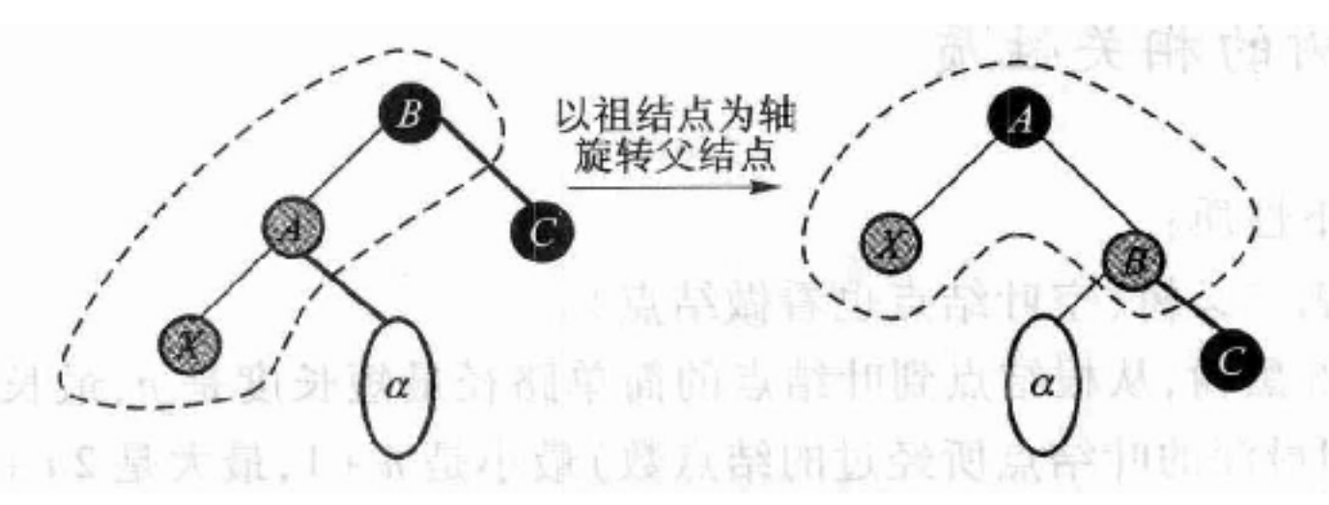


图11.17 父结点是红色，叔父结点是黑色的情况，旋转解决红红冲突

注意：图11.17中新增结点X是A结点的左子结点。若X是A的右子结点，可以把X结点和其父结点A换位，则A结点就成为X结点的左子结点，此时即可类似地处理，只是X和A符号换位而已。

在图11.17中，A结点是父结点的左子结点，如果是右子结点，则需要向左旋转。方法类似，但此时需要保证X结点是A结点的右子结点，而不是左子结点。

实际上，有4种子结构需要调整，如图11.18所示。其实质是取双红结点、祖父结点这三者的中位数作为新的子根结点，并且着色为黑色，子根的两个子结点都着色为红色。每个结点的阶都保持原值，调整完成。图11.17所示为图11.18 LL型的例子。

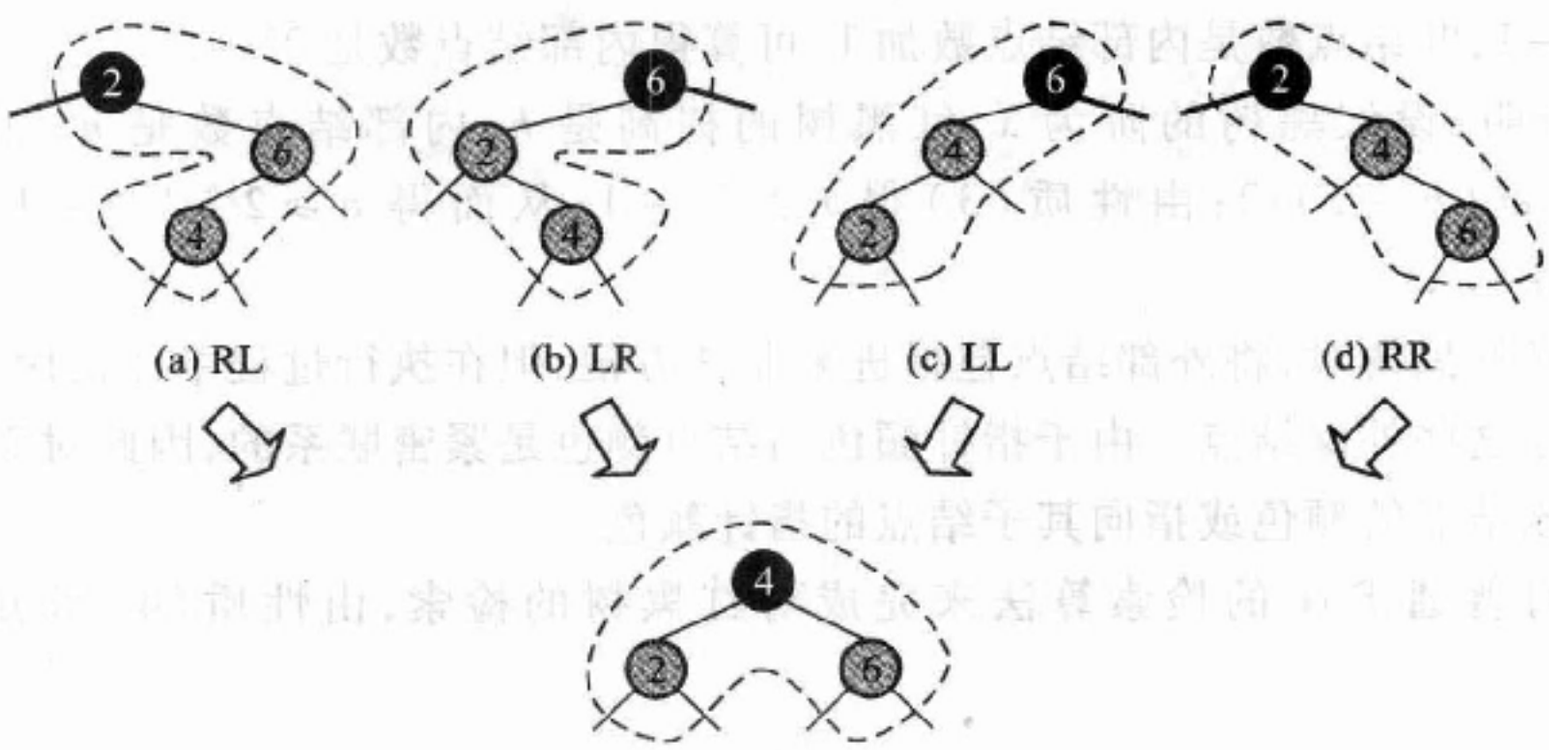


图11.18 叔父结点为黑色，进行重构调整每个结点的阶都保持原值，调整完成

情况2：新增结点的叔父结点也是红色。如图11.19所示，设新增结点是X(红色)，父结点是A(红色)，祖父结点是B(黑色)，叔父结点是C(红色)。其中，X是A的左子结点或右子结点对这种情况没有影响。

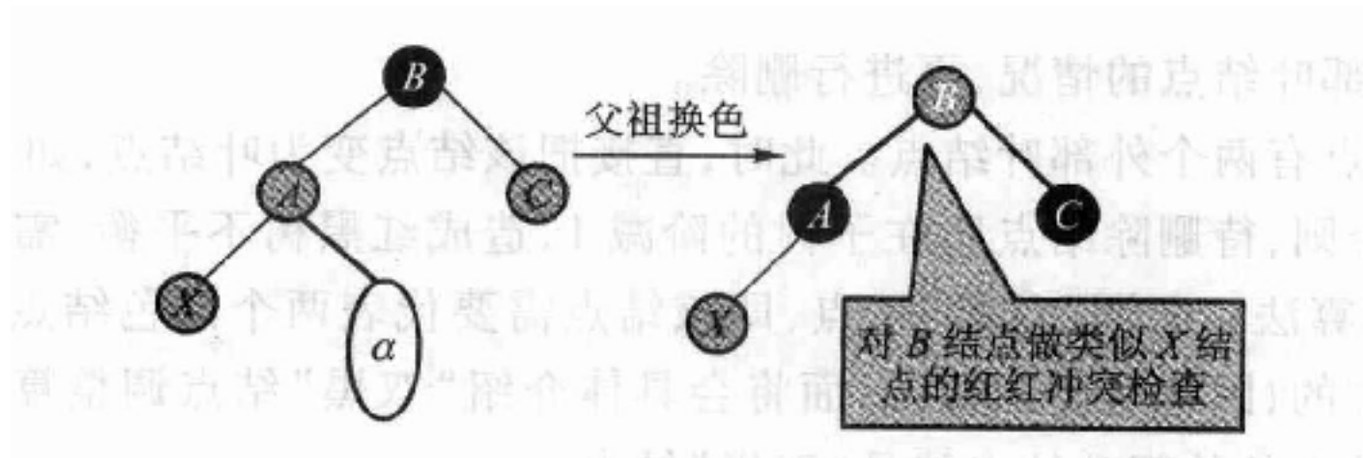


图 11.19 父结点、叔父结点均为红色，则父和祖换色，然后递归处理方法

此时，把 X 的父结点 A 和叔父结点 C 都变成黑色，其祖父结点 B 变成红色。这时只是解决了 A 结点以下的红红冲突，但有可能会引起 B 结点与其父结点的冲突，因此要像处理 X 结点一样，对 B 结点进行递归调整。最糟糕的情况会引起根结点变成红色，此时只需把根结点的着色改成黑色即可，但会引起树的阶增加 1。

图 11.20 所示为在红黑树中插入一个关键码之后引起的红红冲突调整示例。

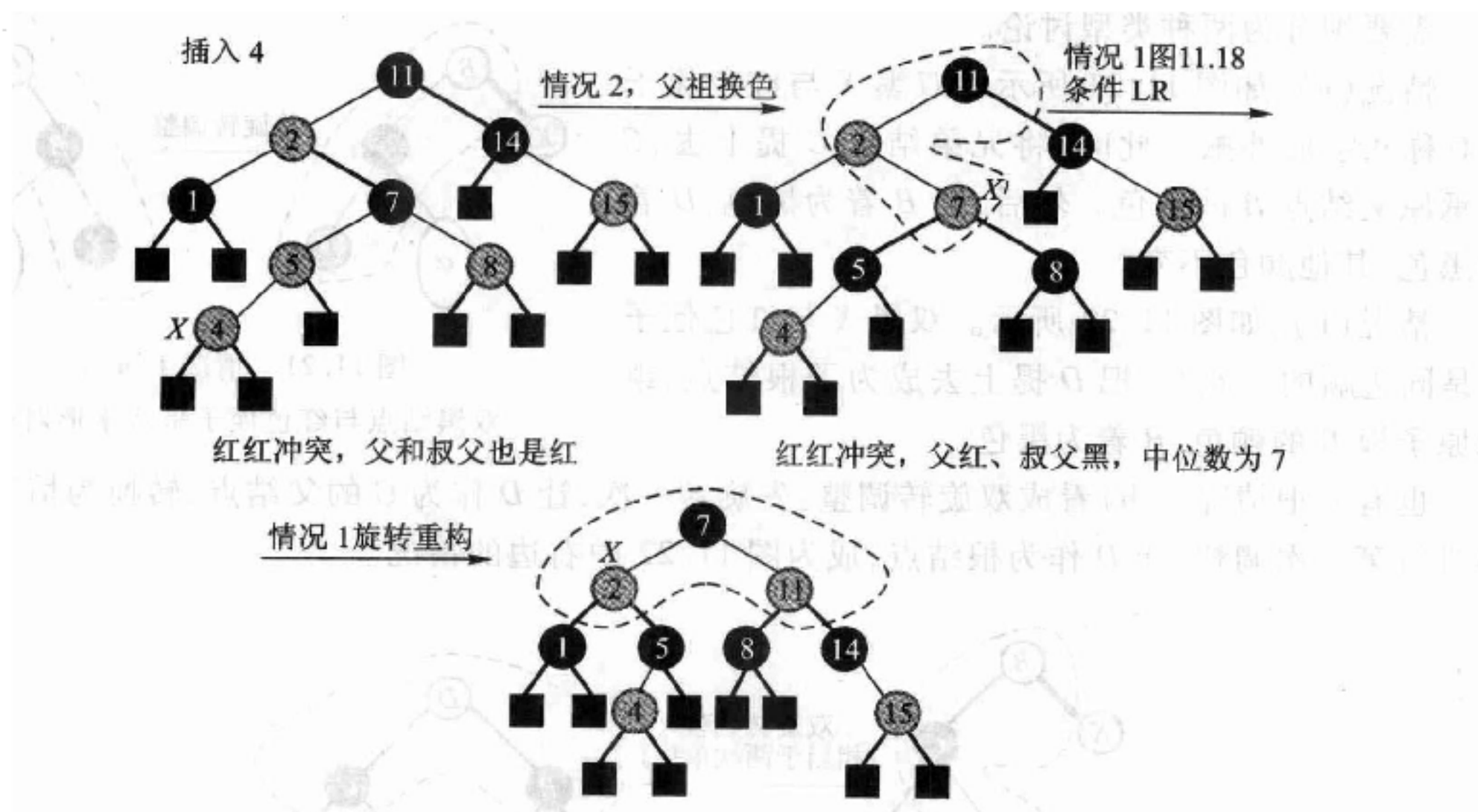


图 11.20 红黑树插入示例

11.6.4 删除结点算法

红黑树的删除操作比插入操作复杂一些，基本删除算法与 BST 类似，但需要检查红黑平衡性。首先，找到待删除的结点，如果该结点有一个以上的空指针（有一个或两个外部叶结点），则直接删除；如果有两个非叶子结点，则在右子树中找到最小值结点（其后继结点）与该结点进行值交换（结点位置的着色不变），然后再对交换后的位置调用删除结点算法，直到转换位置成为

具有一个或两个外部叶结点的情况，再进行删除。

(1) 待删除结点有两个外部叶结点。此时，直接把该结点变为叶结点，如果待删除结点是红色，则算法结束。否则，待删除结点所在子树的阶减1，造成红黑树不平衡，需要调整，该算法称为“双黑”结点调整算法。所谓“双黑”结点，即该结点需要代表两个黑色结点，才能维持树的平衡，事实上是不允许的，因此需要调整(后面将会具体介绍“双黑”结点调整算法)。删除上述黑结点后，替代被删结点的外部叶结点就是“双黑”结点。

(2) 待删除结点只有一个外部叶结点，可知其非空子结点肯定为红色，待删结点肯定为黑色，此时只需把其子结点提升到删除结点位置，颜色变为黑色即可。

上述删除算法把待删除结点已删掉，此时只需考虑解决“双黑”结点现象即可，这也是删除算法中最复杂的地方。下面分情况讨论，注意所讨论情况是针对“双黑”结点是左子结点的情况；若是右子结点，可参照左子结点的处理方法，注意左右对称地修改。

情况1：双黑结点的兄弟结点是黑色，且子结点有红色。

需要细分为两种类型讨论。

情况(a)，如图11.21所示。双黑 X 与红色侄子 D 对称八字形外撇。此时，将兄弟结点 C 提上去，C 继承原父结点 B 的着色。然后，把 B 着为黑色，D 着为黑色，其他颜色不变。

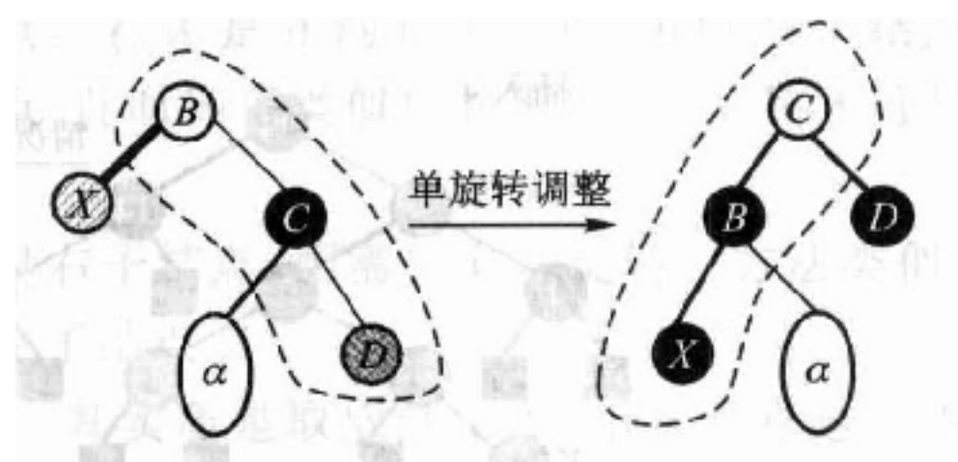


图11.21 情况1(a)：双黑结点与红色侄子呈八字形对称

情况(b)，如图11.22所示。双黑 X 与红色侄子 D 是同边顺的。此时，把 D 提上去成为子根结点，继承原子根 B 的颜色，B 着为黑色。

也有人把情况1(b)看成双旋转调整：先旋转一次，让 D 作为 C 的父结点，转换为情况1(a)；再进行第二次调整，让 D 作为根结点，成为图11.22中右边的情况。

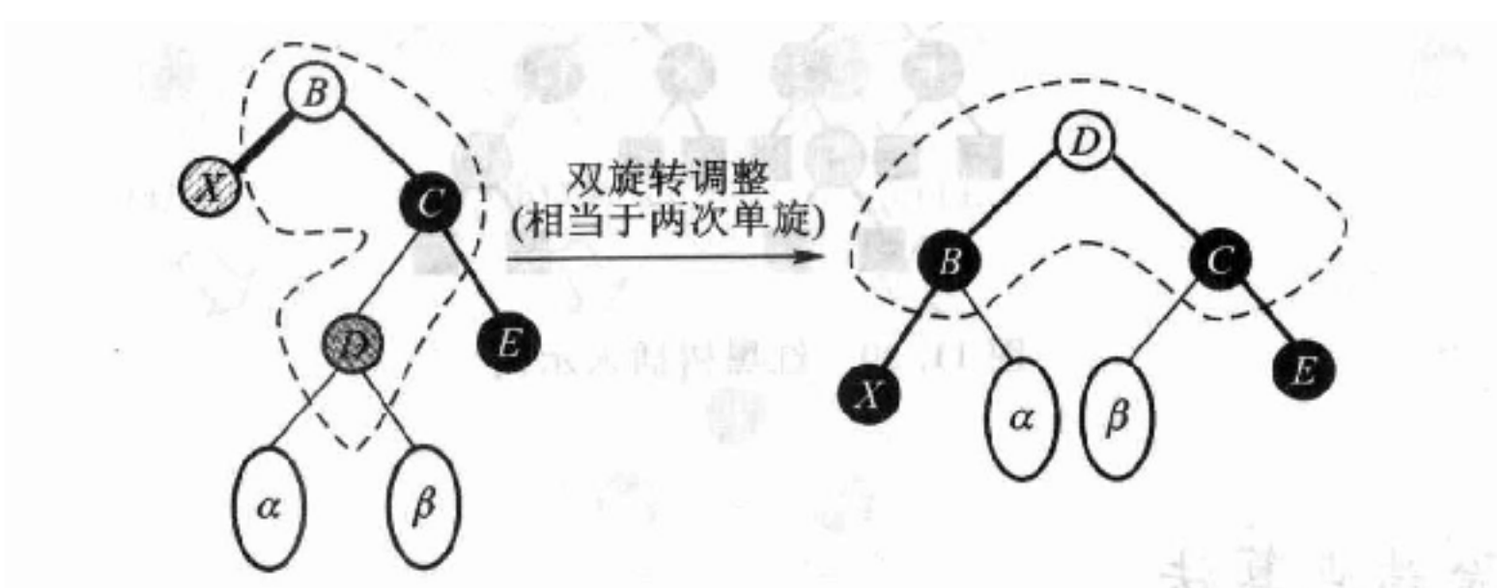


图11.22 情况1(b)：双黑结点与红色侄子同边顺

情况2：双黑结点的兄弟结点是黑色，并且兄弟结点有两个黑色子结点。

如图11.23所示，双黑结点的兄弟结点 C 是黑色，且有两个黑色子结点 D 和 E。进行换色处理，把 C 着红色，B 着黑色，如果 B 原为红色，则算法结束。否则，对 B 继续做双黑调整。

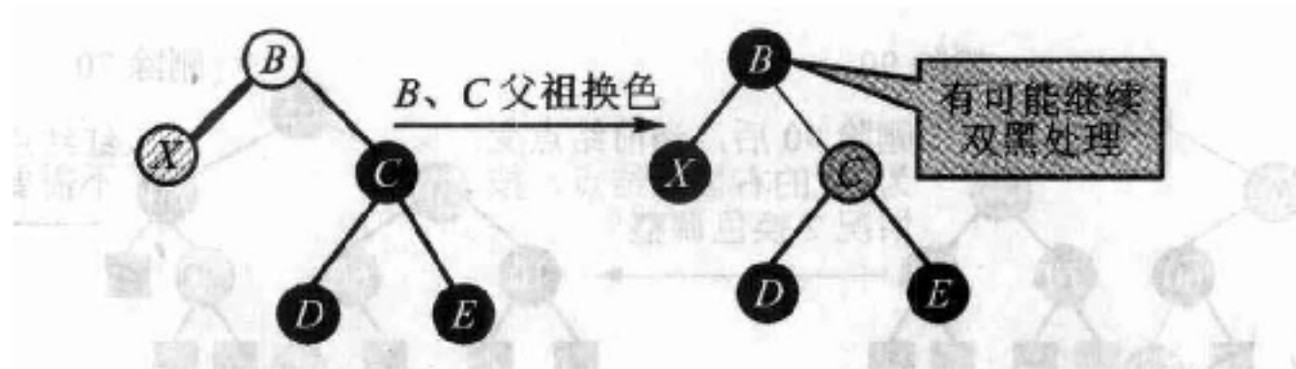


图 11.23 情况 2 双黑的兄弟结点为黑色，且兄弟结点有两个黑色子结点

情况 3：双黑结点的兄弟结点是红色。

如图 11.24 所示，双黑的父结点肯定为黑色（否则与红色子结点 C 冲突），双黑的兄弟结点 C 的子结点 α 和 β 肯定都是黑色。因此，可做旋转，此时 X 结点仍是双黑结点，只是转化为前面两种情况之一继续处理。

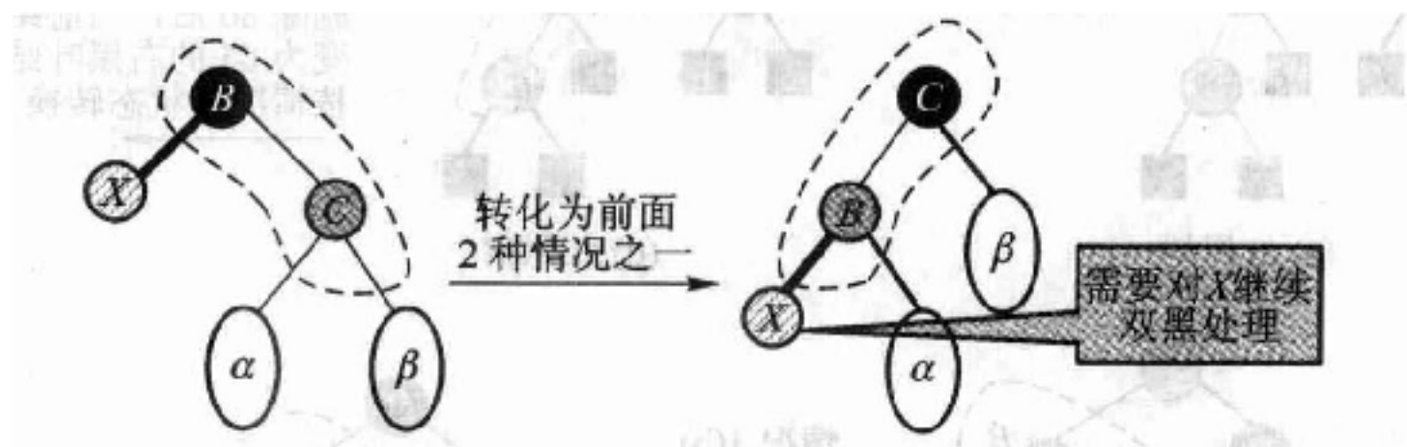


图 11.24 情况 3 双黑的兄弟结点为红色

以上各种情况讨论了删除结点所作的操作，重点在于对删除黑色结点时的调整操作，由以上步骤可知，调整“双黑”结点最多有三步旋转操作，其他操作则是换色、递归处理，相对简单。

图 11.25 所示为在红黑树中连续删除 3 个关键码之后引起的调整示例，其调整运用到了双黑冲突处理策略。

从图 11.25(a)的红黑树中删除 90，结点 80 的右子结点——黑的空叶结点变成当前结点，其兄弟 70 是带两个黑子女的黑结点，按情况 2 进行“双黑”调整，对 70 和 80 进行换色处理。由于 80 原来是红色的，因此改变颜色后使树重新恢复了图 11.25(b)所示的平衡状态。

继续从图 11.25(b)中将 70 删除，就得到了图 11.25(c)。由于删除的是红色结点，删除后树仍然是平衡的。

从图 11.25(c)中将 80 删除，得到图 11.25(d)。这次删除的是非根黑色结点，破坏了红黑树平衡，替代 80 的是结点 X，X 结点符合情况 3，进行双黑调整，得到图 11.25(e)。调整后的当前结点还是 X，符合情况 1(b)，进行双黑调整，得到图 11.25(f)。

上述调整过程中，需要注意图的双黑结点是右子结点的情况，与前面情况 1 至情况 3 的图例是对称的。

上面介绍的插入和删除操作，为了重新恢复红黑树的平衡需要自底向根的方向调整，其平均和最差检索时间都是 $O(\log_2 n)$。因此，红黑树的结点往往由数据、左指针、右指针、颜色和父指针 组成。如果不给出父指针，就可以借用堆栈，从根结点至插入/删除结点路径上所遇到的每个

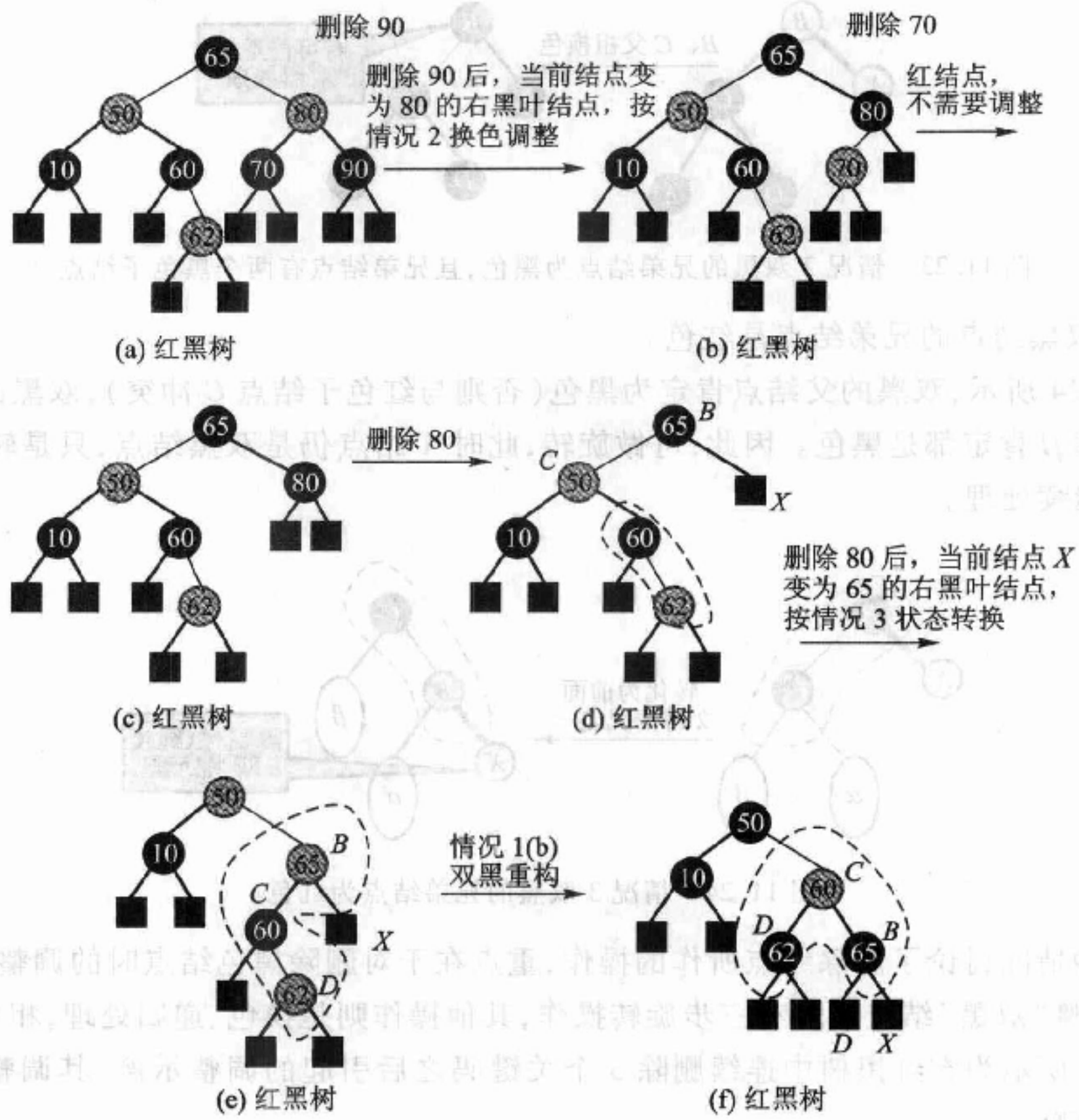


图 11.25 红黑树连续删除 90、70、80，引起的双黑调整示例

结点指针都保存在堆栈中，以方便回溯。此外，还可以采用自顶向下的迭代式插入/删除调整方法，其调整速度更快。

红黑树在很多地方都有应用，例如 C ++ STL 的 set、multiset、map、multimap 等数据结构都应用了红黑树的变体。在 Linux 内核中，用于组织虚存“区间”的数据结构也是红黑树。

本 章 小 结

索引就是把关键码与其主文件中的数据记录位置相关联的过程，索引文件则是用于记录这种联系的文件组织结构。索引技术的目的是在支持高效数据（尤其是外存的大量数据）检索的同时，支持高效的插入和删除操作，它是数据库的核心技术之一。

实际应用中，经常需要根据属性的值来查找记录，因此采用倒排索引技术。每一项都包括一

个属性值和具有该属性值的那些记录地址。由于不是由记录来确定属性值，而是由属性值来确定记录的位置，因此称为倒排索引，带有倒排索引的文件称为倒排文件。

线性索引被组织成简单的（关键码，指针）的序列，按照关键码的顺序进行排序，其常用的检索技术是二分检索。数据规模太大，索引文件无法完全存储到内存时，可以采用二级线性索引。

多分树静态索引结构在文件创建、初始装入记录时生成，并固定不变，插入和删除操作非常不方便。只有当文件再组织时才允许改变索引结构。

B/*B*+树是动态的索引结构，允许动态地插入或删除记录，索引结构本身也可能发生改变。改变索引结构的目的是为保持较好的性能，例如较高的检索效率。*B* 树和 *B*+树都是经典的动态索引，本质上是平衡的多分树。*B* 树中的关键码没有重复，父结点中的关键码是其子结点的分界。*B*+树是 *B* 树的一种变形树，它不同于 *B* 树的特点：

（1）有 k 个子结点的结点必然有 k 个关键码。

（2）*B*+中叶结点是关键码的一个全集，跟实际数据记录有关的信息均存放在叶结点中。

（3）内部分支结点仅具有索引作用，可以看做是从叶往根的方向一层层复写子结点中的最大（或最小）关键码。*B*+树本身也有很多变体。

B 树、*B*+树的插入与删除都要特别注意保持其平衡性质，特别是等高、子结点个数、关键码个数的上下界限制。由于 *B* 树和 *B*+树本身具有相似性，又略微不同，因此其操作过程类似，但是有一些差别。

B 树上的检索从根结点开始，在结点包含的关键码中查找给定的关键码，如果找到则查找成功；否则，确定给定关键码可能在的子树，重复上面的操作，直到检索成功或者指针为空为止。

B 树的插入可以说是“插入—分裂”的过程，首先是在恰当的叶结点中添加关键码。*B* 树插入分裂时，中位数要向父结点中传递。分裂过程可能一直传到根，建立一个新根结点，树增高一层。

B 树的删除可以说是“交换—删除—借关键码—合并”的过程。若 *B* 树中被删码不在叶结点层，则先把此关键码与其后继对换位置，然后再删除。*B* 树删除借关键码时，父结点中的分界码要合并到两个相邻结点的关键码集合中，该集合的中位数尽量把集合平等划分为两个新结点，中位数代替父结点中的原分界码；*B* 树删除，发生合并时，父结点中的分界码要拿下来，合并到两个相邻结点的关键码集合中。合并也可以传递到根。

B+树检索时必须继续沿指针向下一直查到叶结点层的关键码，才能找到跟记录有关的信息（例如，指向该记录主文件中位置的指针）。

B+树的插入也是“插入—分裂”的过程，也是在叶结点上开始。如果上溢出导致叶结点分裂，父结点会增加一个复写分界关键码，也有可能要修改原复写分界码的值，目的是保证上层结点中有这两个结点的复写码。分裂也可能传递到根。

B+树的删除是“删除—借关键码—合并”的过程，不需要交换。*B*+树中的关键码在叶结点层删除后，在不影响检索的前提下，在上层的复本可以保留，作为一个“分界关键码”存在。因为

删除而造成结点中关键码数目下溢出的情况，其处理过程与 B 树的处理一样，也是先考虑借关键码，借不了则合并。合并可以传递到根。

B 树只适合随机检索。$B+$ 树的所有叶结点构成一个有序链表，可以按照关键码大小有序地遍历全部记录，因此 $B+$ 树同时支持随机检索和顺序检索（尤其是范围检索），在实际中应用比较多。

$B/B+$ 树适合于检索并取回少量记录的情况，对于唯一值极少（低基数）的数据域适合采用位图索引技术，用位串来代表数据库列中是否存储该数据域取值。计算机可以迅速地执行位图映射操作，这是一种非常高效的索引。

签名文件被广泛地应用于文本信息检索中，其核心思想是把文本的特征用签名位串来表示。签名文件比原文件小得多，因此可以提供更快速的检索。

红黑树也是一种二叉搜索树，它没有第 12 章将要介绍的 AVL 树那样完全平衡的特性。而是利用对树中结点红黑着色的要求降低了平衡性的条件，达到局部平衡，从而既能提高红黑树增删记录的性能，又能达到高效率的检索要求。它的统计性能要好于 AVL 树，且增删记录算法性能好、易实现，因此红黑树是一种很好的索引结构。C ++ STL 的 set、multiset、map、multimap 都应用了红黑树的变体，Linux 内核中的用户态地址空间管理也使用了红黑树。

习 题

1. 假定一个计算机系统有 4 096 个 B 的磁盘块。存储的每一条记录中 4 B 是关键码，64 B 是数据域。记录已经排序，顺序地存放到磁盘文件中。线性索引的结构为（每块的最小关键码，该块的磁盘地址），地址指针为 4 B。通过线性索引访问磁盘文件中的记录。

（1）如果线性索引的大小是 2 MB，最多可以在磁盘文件中存储多少条记录？

（2）如果线性索引也存储在磁盘中（这样它的大小仅受二级索引限制），而且使用如图 11.2 所示的 4096 个字节（即 1024 个关键码值）的二级索引时，文件中最多可以存储多少条记录？二级索引中的每个单元引用线性索引磁盘块中的最小关键码值。

2. 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域组成，数据域是一个字符串。试给出下面一组记录的线性索引（采用图 11.1 那样的图示）：

397 Hello world!

82 XYZ

1038 This string is rather long

1037 This is shorter

42 ABC

2222 Hello new world!

3. 设有一个职工文件，并设该文件由以下 5 个记录组成，其中职工号为关键码。

（1）若该文件为顺序文件，试写出文件的存储结构。

（2）若该文件为索引顺序文件，试写出索引表。

（3）若该文件为倒排文件，试写出关于性别的倒排索引和关于职业的倒排索引。

磁盘地址	职工号	姓　名	性　别	职　业	年　龄	工　资
A	39	张莉	男	程序员	25	7000
B	50	王明	女	分析员	31	8500
C	10	李想	男	程序员	28	7500
D	75	丁悦	女	终端操作员	18	5000
E	27	赵驰	男	分析员	33	8000

4. 对于图 11.9 所示的 B 树，依次进行如下操作：插入关键码 480，删除关键码 375，分别画出每次操作后 B 树的形状。

5. 画出能由 1、2、3、4、5、6、7、8 构成的所有 5 阶 B 树。

6. 求满足如下条件的序列的最短长度：按照序列顺序插入关键码能生成高度为 3 的 5 阶 B 树。

7. 假设按如下方法修改从 B 树中删除元素的方式：如果一个结点既有最相邻左兄弟也有最相邻右兄弟，那么在合并前对两个兄弟都要做检查。从一棵高度为 h 的 B 树中删除元素时需要的最大磁盘访问次数是多少？

8. 设有一棵阶 $m=3$ 的 B 树，如图 11.26 所示。其中 $a,b,\cdots,g$ 是结点的名称，结点内的整数为关键码。若在图中插入关键码 55，画出插入后 B 树的状况，计算完成该插入所需要的访外次数（假定一次访外存取一个结点），并具体说明每次访外的功能。

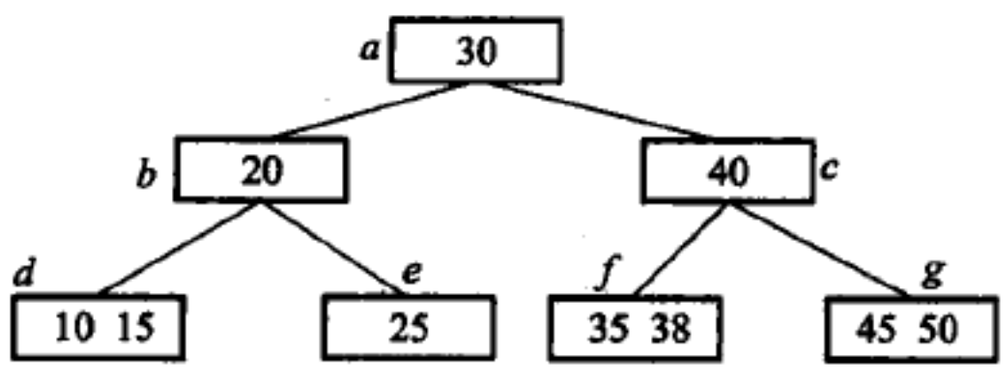


图 11.26　习题 8. 的 3 阶 B 树图示

9. 偶数阶的 B 树插入上溢出时，中位数有两个，需要注意采用统一的策略。例如，取第二个中位数，即分裂后左$\lceil (m-1)/2 \rceil$个关键码，右$\lfloor (m-1)/2 \rfloor$个关键码；或者取第一个中位数，分裂后左$\lfloor (m-1)/2 \rfloor$个关键码，右$\lceil (m-1)/2 \rceil$个关键码。试画出对如图 12.27 所示的 4 阶 B 树进行下列操作后的 B 树。

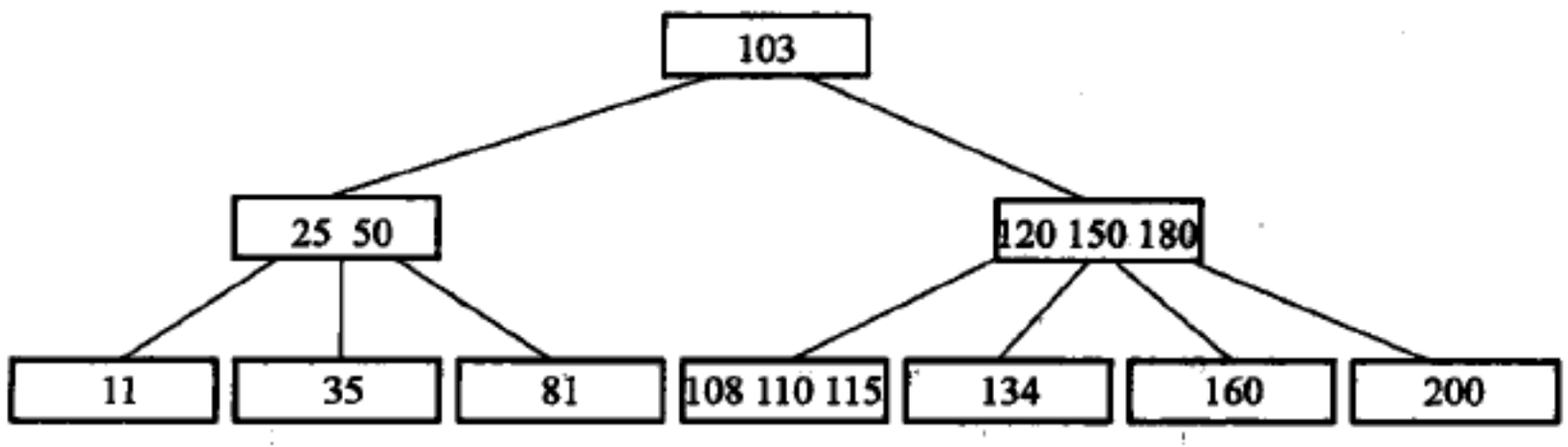


图 11.27　习题 9. 的 4 阶 B 树

(1) 分裂时采用第 2 个中位数为分界码，试画出插入关键码 113 后的 B 树。

(2) 分裂时采用第 1 个中位数为分界码，试画出插入关键码 113 后的 B 树。

(3) 在原树中删除关键码50。

10. 分别画出在如图11.28所示的3阶B树中,进行下列操作后的B树。

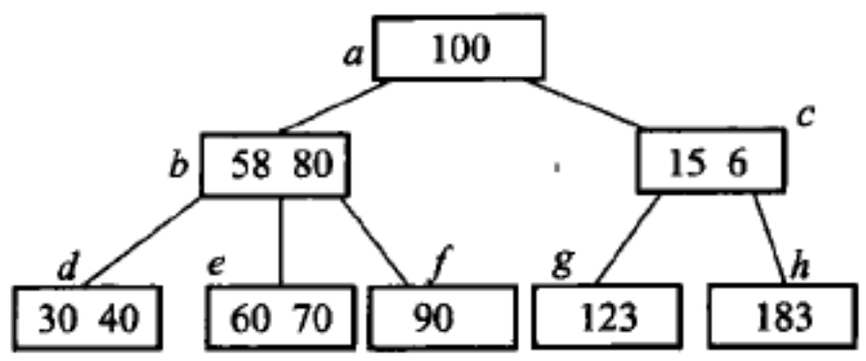


图11.28 习题10.的3阶B树

(1) 插入关键码75。

(2) 删除关键码156。

11. 在B树定义中,特性(2)的意义是什么?如果把$\lceil m/2 \rceil$改成$\lceil m/3 \rceil$或者$\lceil 2m/3 \rceil$是否可以?所得到的树结构和B树有何区别?

12. B树中的某个关键码不在叶结点中,试证明它的前驱和后继(自然顺序)必然在叶结点中。

13. 在含有n个关键码的m阶B树中进行检索时,最多访问多少个结点?

14. 试证明:高度为h的2-3树(3阶B树)中叶结点数目在2^{h-1}与3^{h-1}之间。

15. 假设一个数据文件每个记录对象需要占用128 B(其中关键码占用4 B),且所有记录均已按关键码有序地存储在主磁盘文件中。设磁盘页块大小为2 048 B,若主存中有12 MB空间可以用来存储索引结构,索引项中每一个地址指针占8 B。试简要回答以下问题(写明计算过程)。

(1) 使用B树索引,B树的阶m最多可以为多少?

注:在B树中找到关键码的同时,应该可以得到其在主文件中的地址。

(2) 4层m阶B树,最多可以索引多少字节的数据文件?

注:独根B树算1层,空B树算0层;要求根据题目给出的数据,并给出计算结果和具体的计算过程。

(3) 假设尽量把B树的头几层放入内存(本题规定不能超过12 MB),那么给定关键码,通过B树查找到(2)小题中主数据文件的一个记录,最少需要几次访外?最多需要几次访外?

(4) 上述问题(1)~(3)中的B树换成B^*树,试回答相关的问题。

16. 假定一个文件有一兆(1000000)个记录,每个记录占200 B,其中关键码占50 B。一个磁盘页块(结点)有1 000 B,页块指针为5 B。若用B^+树组织索引,并假定所有页块(结点)都尽可能装满,则需要多少索引页块(即B^+树中有多少个非叶结点)?在上述文件组织中存取一个记录需要多少次外存访问?

17. 对于下面两种B^+树,列表给出它们在1、2、3、4和5层(独根是一层树)的不同情况下,能够存储的最大记录数和最小记录数。

(1) 如图11.9那样的B^+树,其内部结点阶为50,叶结点阶为50。

(2) 如图11.12那样的混合型B^+树,其内部结点阶为100,叶结点阶为15。

18. 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。试给出插入这些记录后的4阶B^+树。假定叶结点最多可放3条记录。

19. 已知一组关键码为(20, 30, 50, 52, 60, 68, 70),试依次插入关键码。

(1) 生成一棵3阶的B树,画出每次插入一个关键码后B树的结构,并画出删除50和68后的B树状态。

(2) 生成一棵3阶的B^+树,画出每次插入一个关键码后B+树的结构,并画出删除50和68后的B^+树

状态。

20. 假定一个文件由 15 个记录组成，每个记录的关键码均为整数，分别为：$1^2, 2^2, 3^2, \cdots, 15^2$。每个数据页块存放 3 个记录。要求：

(1) 用 B 树组织索引，设 $m=3$，依次将上述 15 个关键码插入 B 树，画出插入后的 B 树的结构图。

(2) 用 B^+ 树组织索引，设 $m=3$，依次将上述 15 个记录插入文件中，画出插入后的整个文件的结构图（包括 B^+ 树索引）。

21. 在如图 11.29 所示的 2 阶 B^+ 树中，删除关键码 35。试讨论删除后应该是 2 层 B 树，还是 3 层 B 树。

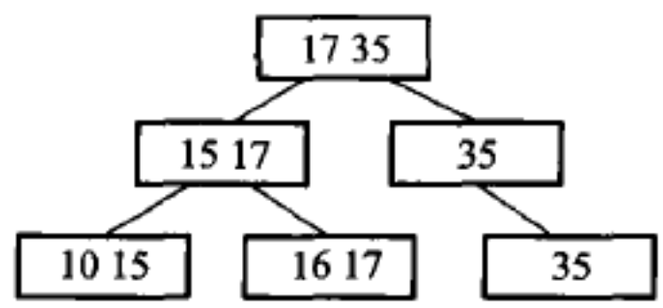


图 11.29　2 阶 B^+ 树图示

22. 图 11.30 所示为一棵混合型 B^+ 树，假定内部结点的阶为 4，叶结点的阶为 3。

(1) 画出插入关键码“C”以后的 B^+ 树。

(2) 画出删除关键码“I”以后的 B^+ 树。

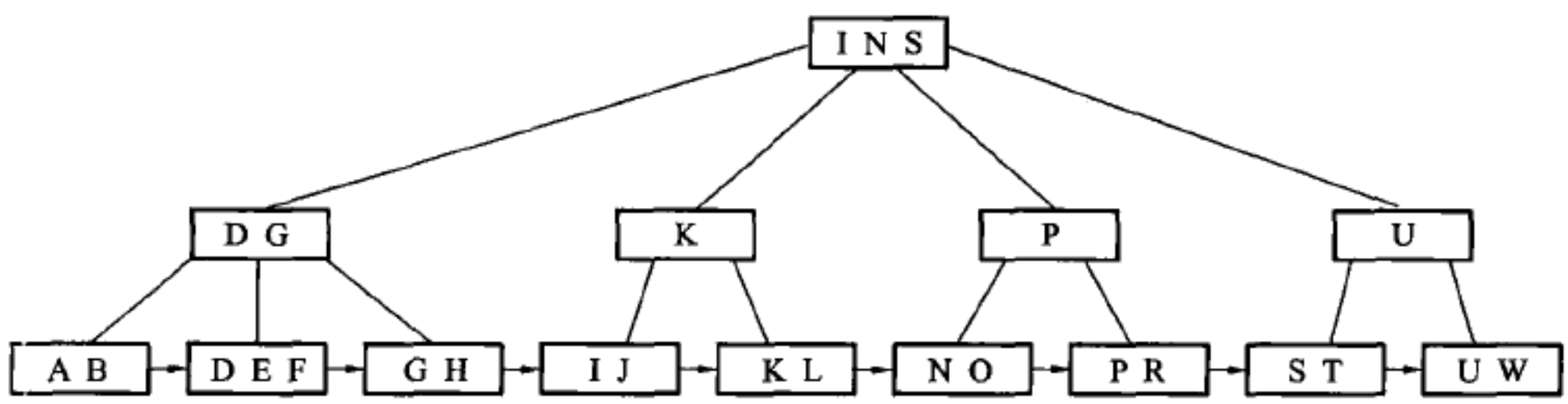


图 11.30　混合型 B^+ 树，内部结点阶为 4、叶结点阶为 3

23. 下面是顾客记录，格式为“顾客号：(年龄，月收入)”。

1：(25，600)　2：(45，600)　3：(50，750)　4：(50，1000)

5：(50，1200)　6：(70，1100)　7：(85，1400)　8：(30，2600)

9：(25，4000)　10：(45，3500)　11：(50，2750)　12：(60，2600)

(1) 试写出年龄位图索引向量集合。

(2) 试写出月收入位图索引向量集合。

(3) 找出年龄范围为 20～40 且月收入为 1 000～2 000 元的顾客。

24. 考虑一个具有 1 000 000 个记录的文件，其数据域 F 有 m 个不同值。

(1) F 的位图索引有多少个字节？

(2) 假定数据域 F 的每个取值每隔 m 个记录出现一次，试设计一种合适的压缩存储方案，并计算该压缩索引需要多少个字节？

25. 对如图 11.31 所示的红黑树，画出下面插入和删除过程的主要变换步骤。

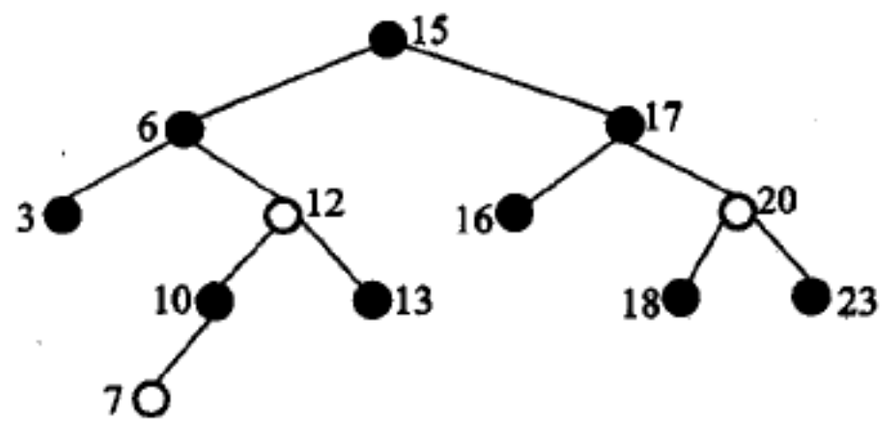


图 11.31 红黑树

(1) 插入结点 19。

(2) 删除结点 10。

26. 讨论采用自顶向下的红黑树插入和删除调整方法,并给出插入和删除的实例。

上 机 题

1. 对如图 11.1 和 11.2 所示的变长记录实现一个二级线性索引。假定磁盘块的长度是1 024 B。数据库文件中记录的长度一般在 20 ~ 200 B 之间,包括 4 B 的关键码值。索引文件中的每条记录存储一个关键码值和一字节偏移量,这个偏移量表示相应记录第一个字节在数据库文件中的字节偏移量。顶层索引(存储在内存中)应当是一个简单的数组,存储索引文件相应块的最小关键码值。

2. 试参照 B 树的插入、删除算法实现 B^+ 树的插入、删除算法。

3. 试设计一个遍历 B 树的算法,使输出的关键码序列递增有序。

4. B^* 树是内部结点包含至少$\lceil 2m/3 \rceil$个子结点的 B 树,试考虑 B^* 树的插入和删除操作造成的分裂与合并应该采取怎样的策略并实现之。

5. 现有红黑树 T1、T2 和关键码 z,T1 中的关键码都比 z 小,T2 中的关键码都比 z 大。试将 T1、T2 和 z 合并成一棵新的红黑树 T3。树中的关键码不重复。

(1) N 是操作结点总数的上限,试分析你的算法的时间代价,如果高于 $O(\log N)$,要调整算法。

(2) 在不给定关键码 z 的情况下,实现两棵红黑树的合并操作。

第 12 章　高级数据结构

有一些应用程序不能使用简单的线性表来有效地实现。人们可以对线性表进行扩展，实现一些功能更强大、具有更多操作的高级线性结构。此外，为了满足一些特定的应用需要，人们也设计了几种特殊结构的树，这些高级树结构在一些领域有着广泛的应用。在本章将陆续介绍几种常用的高级线性结构和高级树结构。

12.1　多维数组

多维数组（multi - array）是一维数组（即向量）的扩充，数组的数组就是多维数组。例如，二维数组可以看做“数据元素是一维数组”的一维数组，三维数组可以看做“数据元素是二维数组”的一维数组，依次类推。数组作为一种数据结构，其特点是结构中的元素本身可以是具有某种结构的数据，同一个数组中的元素具有相同的数据类型。

12.1.1　多维数组的存储

在 Pascal 语言中，允许指明多维数组每一维的下界和上界，对于一般的 n 维数组，可以表示为 ELEM $A[c_1..d_1][c_2..d_2]\cdots[c_n..d_n]$，$c_i$ 和 d_i 是各维下标的下界和上界。因此其元素个数为：

$$\prod_{i=1}^{n}(d_i - c_i + 1)$$

在 C/C ++ 中，多维数组每一维的下界都是 0。多维数组可以看做是线性表的推广。例如，C ++ 中二维数组（或称矩阵）

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-2} & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-2} & a_{1,n-1} \\ \cdots & \cdots & & \cdots & \cdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-2} & a_{m-1,n-1} \end{pmatrix}$$

可以看成是由 m 个行向量所组成的向量，也可以看成是 n 个列向量所组成的向量。

二维数组中的每个元素 $a_{i,j}$ 都属于两个向量：第 i 行的行向量和第 j 列的列向量（C ++ 中，i 和 j 都是从 0 开始）。同样，三维数组 $A_{p,m,n}$ 中的每个元素 $a_{i,j,k}$ 都属于 3 个向量，这时每个元素最多可以有 3 个前驱和 3 个后继。推而广之，k 维数组 $A_{n_0,n_1,\cdots,n_{k-1}}$ 的每个元素 $a_{i_0,i_1,\cdots,i_{k-1}}$ 都属于 k 个

向量，可以有 k 个前驱和 k 个后继。

数组结构中元素个数相对固定，一旦生成，只改变元素的值，并不改变元素的相对位置和个数。因此，数组在计算机中自然采取顺序存储方法，这种顺序存储与数组的周游有关，即按一定规律系统地访问一个数组中的所有元素，对每个元素恰好访问一次。把数组元素按周游的顺序排列起来，便得到所有数组元素的一个线性序列。

以二维数组 $A_{m,n}$ 为例，可以按照“行优先顺序”和“列优先顺序”两种规律周游。在 Pascal 和 C++ 等语言中，数组就是按“行优先顺序”存储的。“行优先顺序”将数组元素按行向量排列，第 $i+1$ 个行向量紧跟在第 i 个行向量后面。可以得到以下线性化的序列：

$$a_{0,0}, a_{0,1}, \cdots, a_{0,n-1}, a_{1,0}, a_{1,1}, \cdots, a_{1,n-1}, \cdots, a_{m-1,0}, a_{m-1,1}, \cdots, a_{m-1,n-1}$$

把上述规则推广到多维的情况，“行优先顺序”可规定为先排最右的下标，从右向左排，最后排最左的下标。

一个 C++ 的 $d_0 \times d_1 \times \cdots \times d_{n-1}$ 多维数组，声明如下：

ELEM $A[d_0][d_1]\cdots[d_{n-1}]$;

其数组元素的下标范围是 ELEM $A[0..d_0-1][0..d_1-1]\cdots[0..d_{n-1}-1]$。假设每个数组元素所占存储单元空间为 d，则行优先存储公式为：

$$\begin{aligned} loc(A[j_0,j_1,\cdots,j_{n-1}]) &= loc(A[0,0,\cdots,0]) \\ &+ d \cdot [j_0 \cdot d_1 \cdot \cdots \cdot d_{n-1} + j_1 \cdot d_2 \cdot \cdots \cdot d_{n-1} + \cdots + j_{n-2} \cdot d_{n-1} + j_{n-1}] \\ &= loc(A[0,0,\cdots,0]) + d \cdot [\sum_{i=0}^{n-2} j_i \prod_{k=i+1}^{n-1} d_k + j_{n-1}] \end{aligned} \tag{12.1}$$

计算任何一个元素的位置所花费的时间是一定的，因此数组中存取任何一个元素值所需要的时间是一样的，这样的存储结构叫做随机存储结构。

在 FORTRAN 语言里，数组就是按“列优先顺序”存储的：先排最左的下标，从左向右排，最后排最右的下标。“列优先顺序”将数组元素按列向量排列第 j+1 个列向量紧接在第 j 个列向量后面。例如，二维数组可得到以下线性化的序列：

$$a_{0,0}, a_{1,0}, \cdots, a_{m-1,0}, a_{0,1}, a_{1,1}, \cdots, a_{m-1,1}, \cdots, a_{0,n-1}, a_{1,n-1}, \cdots, a_{m-1,n-1}$$

12.1.2 特殊矩阵

二维数组经常被用来表示矩阵。下面介绍几种比较特殊的矩阵。

1. 三角矩阵

三角矩阵是指 n 阶矩阵中的下三角（或上三角）的元素都是 0 或者一个常数 c 的矩阵，如图 12.1 所示。在上三角矩阵中，当数组的下标 $i>j$ 时，数组元素 $a[i][j]=c$；而在下三角矩阵中，当下标 $i<j$ 时，$a[i][j]=c$。只需要存储其上三角或下三角的元素以及一个常数，总共有 $\sum_{i=0}^{n-1}(i+1) = n(n+1)/2 = (n^2+n)/2$ 个非零（或非常数）值。

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-2} & a_{0,n-1} \\ c & a_{1,1} & \cdots & a_{1,n-2} & a_{1,n-1} \\ & \cdots & & & \\ c & c & \cdots & c & a_{n-1,n-1} \end{pmatrix} \qquad \begin{pmatrix} a_{0,0} & c & c & \cdots & c & c \\ a_{1,0} & a_{1,1} & c & \cdots & c & c \\ & \cdots & & & & \\ a_{n-1,0} & a_{n-1,1} & \cdots & & a_{n-1,n-1} & \end{pmatrix}$$

图 12.1 上三角矩阵和下三角矩阵,其中 c 为常数

假设以一维数组 list[0..$(n^2+n)/2-1$]作为 n 阶下三角矩阵 $\boldsymbol{A}$ 的存储结构,元素 $a_{i,j}$前面共有 i 行,共$(i^2+i)/2$ 个非零元素;到达第 i 行的第 j 个位置。因此,原来矩阵中的元素 $a_{i,j}$与线性表相应元素的对应位置为 list[$(i^2+i)/2+j$]($i \geqslant j$)。

2. 对称矩阵

对称矩阵指的是一个 n 阶矩阵,其元素满足性质 $a_{i,j}=a_{j,i}, 0 \leqslant (i, j) < n$。为了节省空间,可以只存储相同数据的一个值,例如存储其下三角的值,而对角线之上的值通过对称关系映射过去。假设以一维数组 list[0..$n(n+1)/2-1$]作为 n 阶对称矩阵 $\boldsymbol{A}$ 的存储结构,则 list[k]与矩阵元 $a_{i,j}$之间存在着一一对应的关系:

$$k=\begin{cases} j(j+1)/2+i, \text{当 } i<j \\ i(i+1)/2+j, \text{当 } i \geqslant j \end{cases}$$

12.1.3 稀疏矩阵

稀疏矩阵(sparse matrix)的非零元素非常少,而且分布也不规律,在实际应用中比较常见。稀疏因子用来描述稀疏矩阵的非零元素情况,它定义为在 $m \times n$ 的矩阵中,有 t 个非零元素,则稀疏因子为 $\delta=t/(m \times n)$。通常,当这个值小于 0.05 时,可以认为是稀疏矩阵。

存储稀疏矩阵时,因为其大部分元素都是 0,所以只需要存储那些非零元素。每一个元素除了本身的值,还要存储它们在矩阵中的位置。因此,一般使用三元组(i,j,a_{ij})来顺序存储稀疏矩阵,其中 i 是该元素的行号,j 是该元素的列号,$a_{i,j}$是该元素的值。例如,三元组表{(0,1,3),(1,1,5),(1,2,6),(2,0,2)}表示的矩阵为:

$$\begin{bmatrix} 0 & 3 & 0 \\ 0 & 5 & 6 \\ 2 & 0 & 0 \end{bmatrix}$$

但是,使用三元组存储稀疏矩阵有很多不便,尤其是进行矩阵的查找、添加与删除等运算都十分麻烦。因此,目前经常使用十字链表法来存储稀疏矩阵。

十字链表由两组链表组成,分别代表行和列的指针序列。链表中的每一个结点都包含两个指针,一个指向其同行的下一个元素,一个指向其同列的下一个元素。此外,每个结点还包括行和列的值以及该结点存储的元素值。每个结点既是行链表的一个结点,也是列链表的一个结点。

刚才的三元组矩阵可使用十字链表表示，如图 12.2 所示。

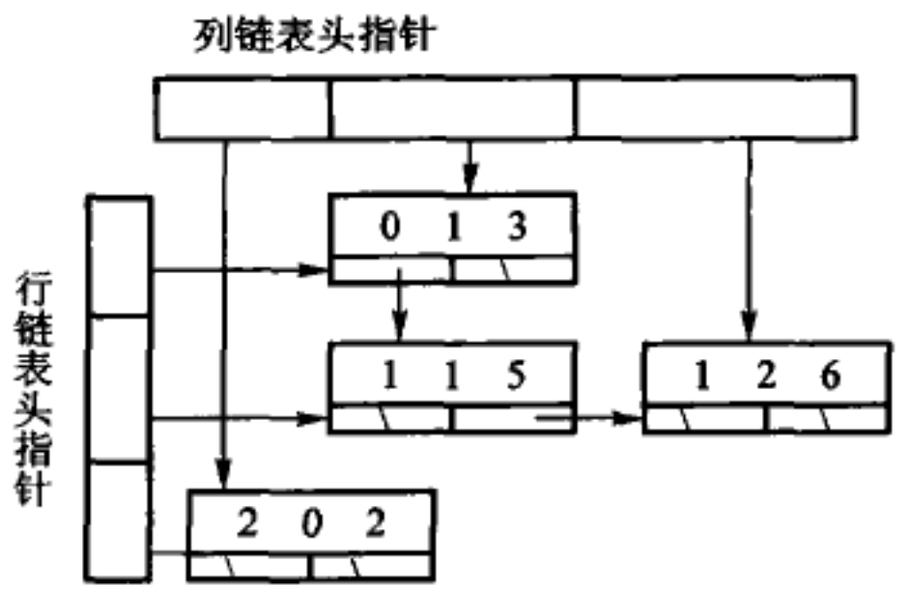


图 12.2 稀疏矩阵的十字链表

下面讨论稀疏矩阵的乘法。根据一般矩阵相乘的规律

$$\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$$

其中，$\boldsymbol{A}[c_1..d_1][c_3..d_3]$，$\boldsymbol{B}[c_3..d_3][c_2..d_2]$，$\boldsymbol{C}[c_1..d_1][c_2..d_2]$。$p = d_1 - c_1 + 1$，$m = d_3 - c_3 + 1$，$n = d_2 - c_2 + 1$；$\boldsymbol{A}$ 为 $p \times m$ 的矩阵，$\boldsymbol{B}$ 为 $m \times n$ 的矩阵，乘得的结果 $\boldsymbol{C}$ 为 $p \times n$ 的矩阵，并有

$$c_{ij} = \sum_{k=c_3}^{d_3} a_{ik} \times b_{kj}$$

经典矩阵乘法的程序如下：

```
int i, j, k;
for (i = c1; i <= d1; i++)
  for (j = c2; j <= d2; j++) {
      sum = 0;
      for (k = c3; k <= d3; k++)
          sum = sum + A[i,k] * B[k,j];
      C[i, j] = sum;
  }
```

容易看出，经典的顺序存储的矩阵相乘所需要的时间代价为 $O(p \times m \times n)$。

稀疏矩阵相乘时，首先判断两个矩阵是否可以相乘，然后使用第一个矩阵的行链表和第二个矩阵的列链表进行相乘，得到的结果先放入三元组，然后等到所有结果都得到，再生成结果矩阵。

若矩阵 $\boldsymbol{A}$ 中行向量的非零元素个数最多为 t_a，矩阵 $\boldsymbol{B}$ 中列向量的非零元素个数最多为 t_b，矩阵 $\boldsymbol{C}$ 中列向量的非零元素个数最多为 t_c，假设 $\boldsymbol{C}$ 矩阵中非 0 元素的个数总和为 N_c。每计算一个 $c_{i,j}$的时间主要用于顺着行 i 和列 j 寻找的过程，其循环次数最多为 $t_a + t_b$，每次循环所花时间是一个常量，记为 c_0，故计算一个 $c_{i,j}$的时间为 $c_0 \times (t_a + t_b)$。计算全部 $c_{i,j}$的时间为 $c_0 \times (t_a + t_b) \times p \times n$。如果保留乘积 $\boldsymbol{C}$ 矩阵的当前列指针向量位置，指向已插入到 $\boldsymbol{C}$ 中各列最新的非零元素，则可使每次插入的时间为一常数，使总执行时间降低为 $O((t_a + t_b) \times p \times n)$。

前面提到过经典顺序存储的矩阵相乘所需要的时间代价为 $O(p \times m \times n)$。当 $t_a + t_b \ll m$

时,采用稀疏矩阵进行乘法运算还是比较快的。

12.2 广义表和存储管理

广义表(generalized list)在文本处理、人工智能和计算机图形学等领域得到广泛应用。例如,在人工智能领域使用的 LISP(LISt Processing)语言中,所有的概念和对象都是用广义表表示的。函数的调用关系、内存空间的引用关系等,都可以用广义表来表示。

在 C++高级语言中有 new 和 delete 等内存空间操作,要求动态地分配存储空间,回收被"释放"的存储空间。在分配和回收过程中,需要解决碎片问题,进行存储压缩。另外,可能由于程序员忘记 delete 已经不再使用的指针等,造成内存泄漏(memory leak)而产生许多无用单元(garbage),因此需要对无用单元进行有效的收集。这就要求操作系统有一套有效的内存管理技术,本质上是利用链表和广义表来实现的。

存储管理的最后一个问题是存储空间溢出的管理。操作系统中详细地讨论了虚拟存储等溢出的处理技术,限于篇幅,本书不再赘述。

12.2.1 广义表的定义和存储结构

线性表是由 $n(n\geqslant 0)$ 个数据元素组成的有限且有序的序列,例如 $(x_0, x_1, \cdots, x_{n-1})$。用"NULL"或者"()"来表示一个空的线性表。线性表的每个元素都具有相同的数据类型,例如数字、符号或者字符串。纯线性表如图 12.3 所示。

如果一个线性表中还包括一个或者多个子表,则称之为广义表(generalized List)也称 multi-list。广义表是 $n(n\geqslant 0)$ 个数据元素 $x_0,x_1,\cdots,x_i,\cdots,x_{n-1}$ 的有序序列,一般记作:

$$L=(x_0,x_1,\cdots,x_i,\cdots,x_{n-1})$$

其中,L 是广义表的名称,n 是它的长度。每个 $x_i(0\leqslant i\leqslant n-1)$ 是 L 的成员,可以是单个元素,也可以是一个广义表,分别称为广义表 L 的原子(atom),也称"单元素"或子表(sublist)。当广义表 L 非空时,称第一个元素 x_0 为 L 的表头(head),称其余元素组成的表 $(x_1,\cdots,x_i,\cdots,x_{n-1})$ 为 L 的表尾(tail)。广义表中包括的元素还可以是广义表,因此这是一个递归定义。通过限制广义表的子表形式,可以得到复杂程度不同的广义表。

纯表(pure list)对应于一棵树,如图 12.4 所示。从根结点到任何叶结点只有一条路径,也就是说任何一个元素(原子、子表)只能在广义表中出现一次。在纯表中,每对括号都对应于一个内部结点,括号中的成员对应于该结点的子结点,表中的原子对应于叶结点。

再入表(reentrantlist)也称"可重入表",允许共享子表或者原子。因此,从根结点开始,某些结点可以通过多个路径访问到。如果再入表没有回路,则图示对应于一个DAG(有向无环图),其元素(包括原子和子表)可能会在表中多次出现。如果允许回路,则再入表图示可以对应于任

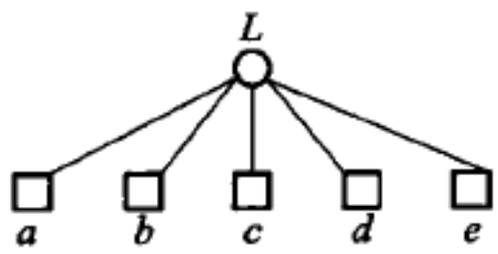


图 12.3 纯线性结构表

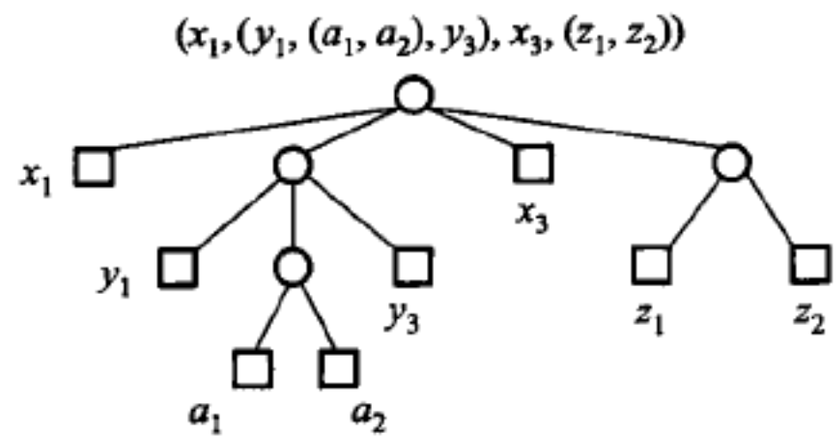


图 12.4 纯表的树表示

意的有向图。

从图 12.5 可以看出，最好是人为地把需要被共享的单原子定义为一个单元素的子表，因为这样更方便标号和引用。为了简便，本章约定允许对子表和原子进行标号，每当重复出现一个标号，就可以代替对应于这个标号的元素。这样，图 12.5 中表的括号表示法可以写成：

$$((L_1:(a,b)),(L_1,c,L_2:(d)),(L_2,e,L_3:(f,g)),L_3)$$

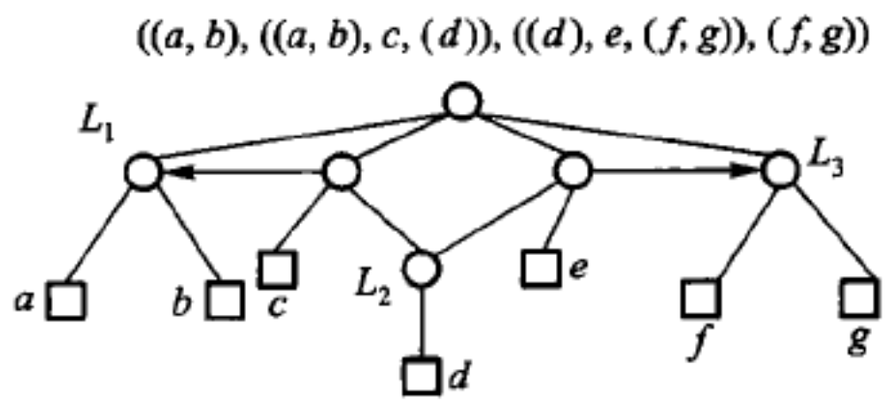


图 12.5 一个再入表的例子，其中没有标志箭头的边都是指向下方的

循环表(cyclic list)也称“递归表”，对应于包含回路的有向图，它允许表中某个子表直接或者间接地把自己作为表或者其子表的一个表目。为了用括号表示法写出这样一个表，需要使用标号。图 12.6 就是循环表的一个图示。

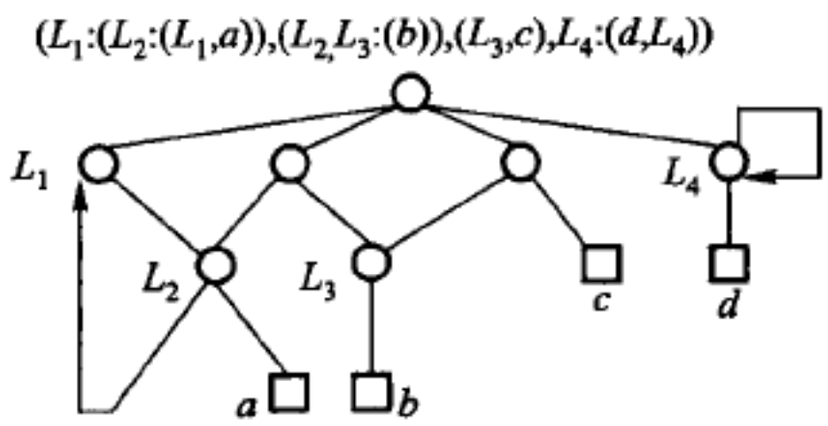


图 12.6 循环表可以看到这是一个有环图

图 12.6 中，因为 L_1 和 L_4 调用自身，导致了回路。其他没有箭头的边，方向都是从上至下的。循环表也可以算作一种存在回路的特殊再入表。

用数学中集合的包含关系可以表示上述几种类型的广义表：

图 ⊇ 再入表(循环表) ⊇ 纯表 ⊇ 线性表

广义表的“深度”是指展开表之后所含括号的层数。循环表的深度为无穷大，线性表的深度

为 1，图 12.4 和图 12.5 的深度都是 3。

根据广义表的不同形式，可以用线性表、树和图数据结构，或者这些数据结构的修改形式来实现。因为纯表等价于树，可以使用树形结构来表示。同理，再入表、循环表可以用图来表示。此外，还可以用类似于树形结构的方法，每个结点记录本层的下一个兄弟 next 指针，而子表结点则另有一个域 sublist 指向整条子表，如图 12.7 所示。其结点定义如下：

TAG = 0 或 1	data/sublist	next

```
typedef enum {ATOM, LIST} TAG;                    // ATOM = 0:单元素;LIST = 1:子表
typedef struct GLNode{
    TAG tag;
    union {
        ElemType data;
        GLNode *sublist;                          // 子表的头指针
    };
    GLNode *next;                                 // 后继结点
};
```

但是，图所示 12.7 的结构在子表头插入或者删除结点时会比较复杂。例如，不能直接删除元素 data 所在的结点，因为它被两个结点所引用。首先，必须周游整个广义表，找出所有指向 data 的指针；然后再逐一加以修改，使得这些指针指向结点 D_2。

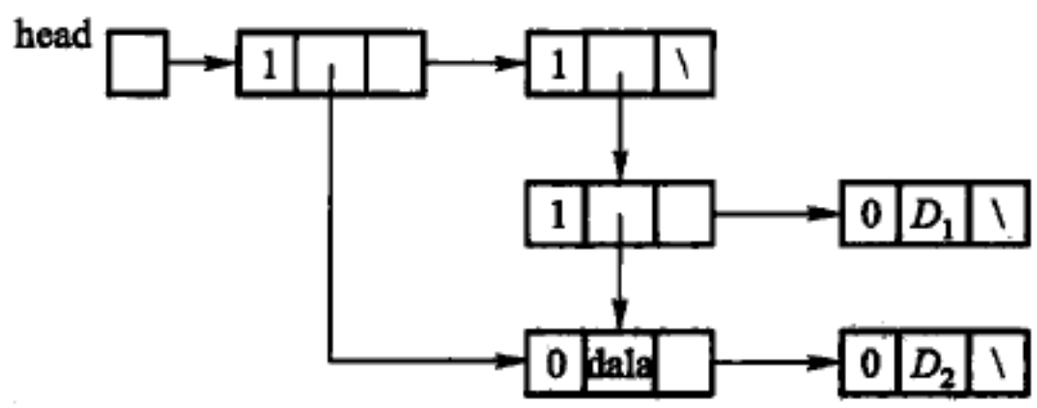


图 12.7 没有头结点的广义表

与带头结点的单链表类似，可以增加一个表头结点，如图 12.8 所示。当 tag 取值为 HEAD（这里取 -1）时，表示头结点，不存储任何表元素；而且可以增加一个 ref 域来记录该子表被引用的次数，以便删除操作使用；此外，还可以增加 name 域，以便给子表命名。

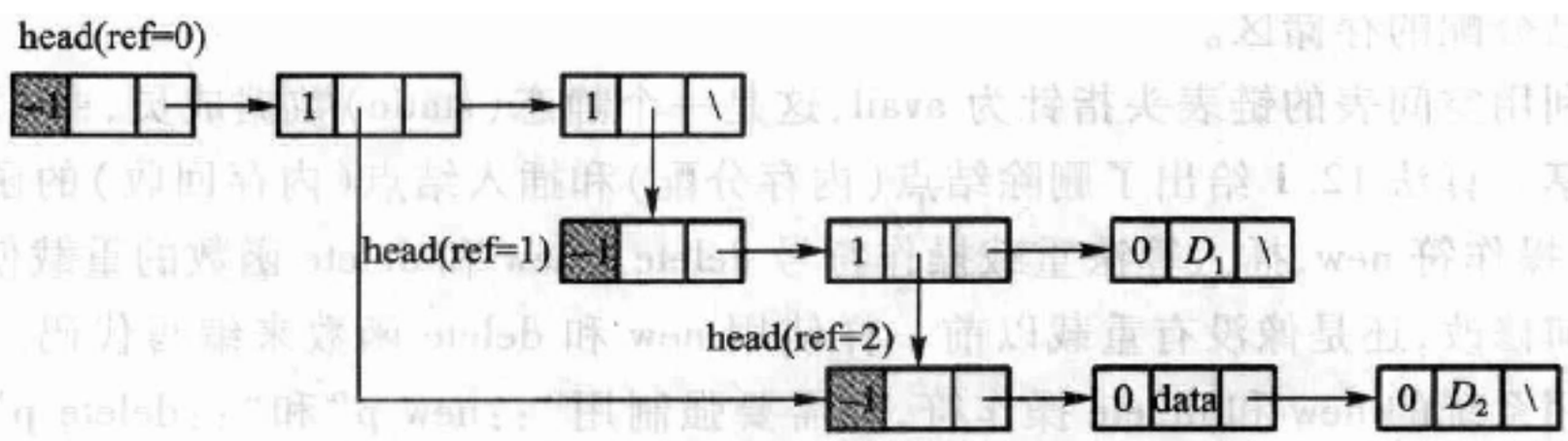


图 12.8 带头结点的广义表

现在删除结点 data，只要简单地将其从子表中摘除即可，不影响引用这个子表的其他指针。用带表头结点的存储形式表示如图 12.6 所示的广义表，可得到图 12.9。

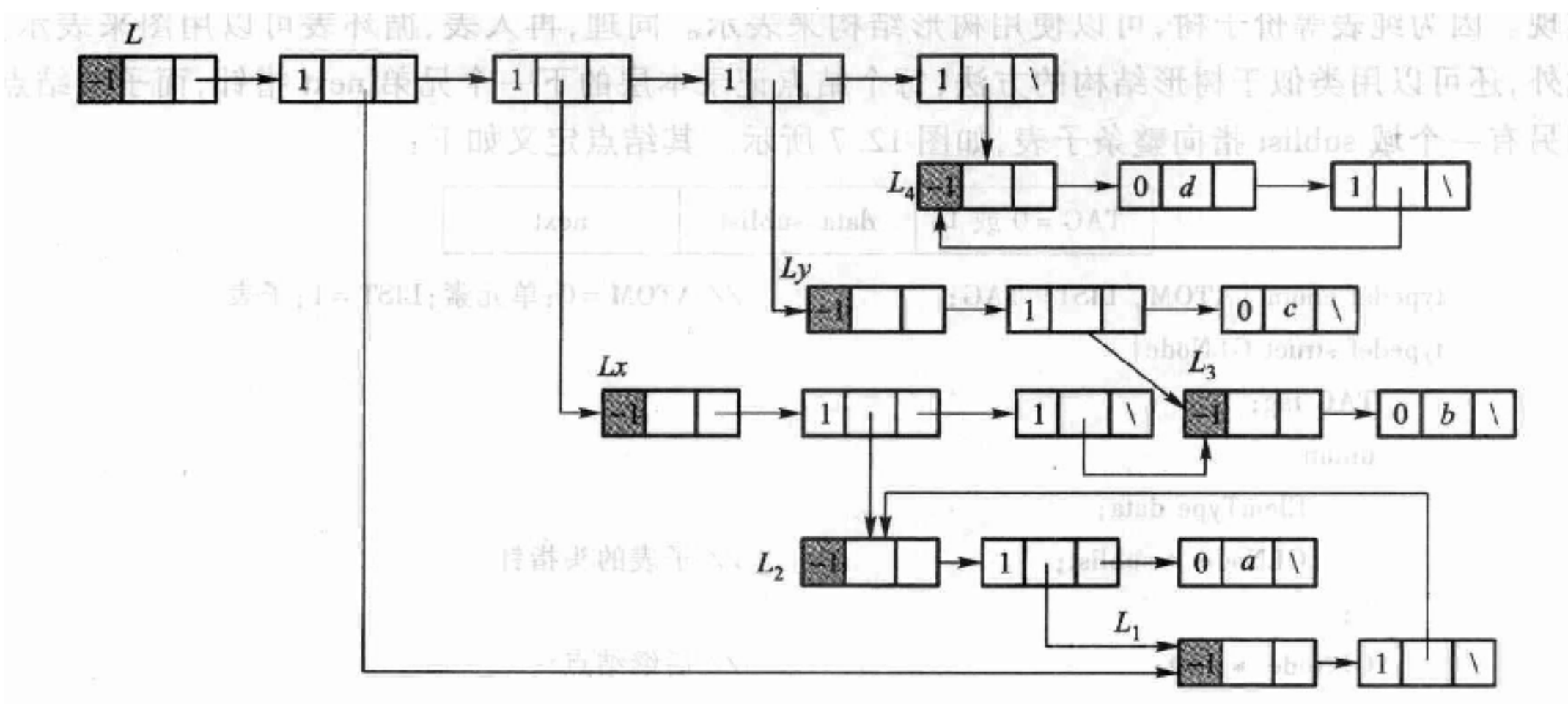


图 12.9 带表头结点的循环广义表

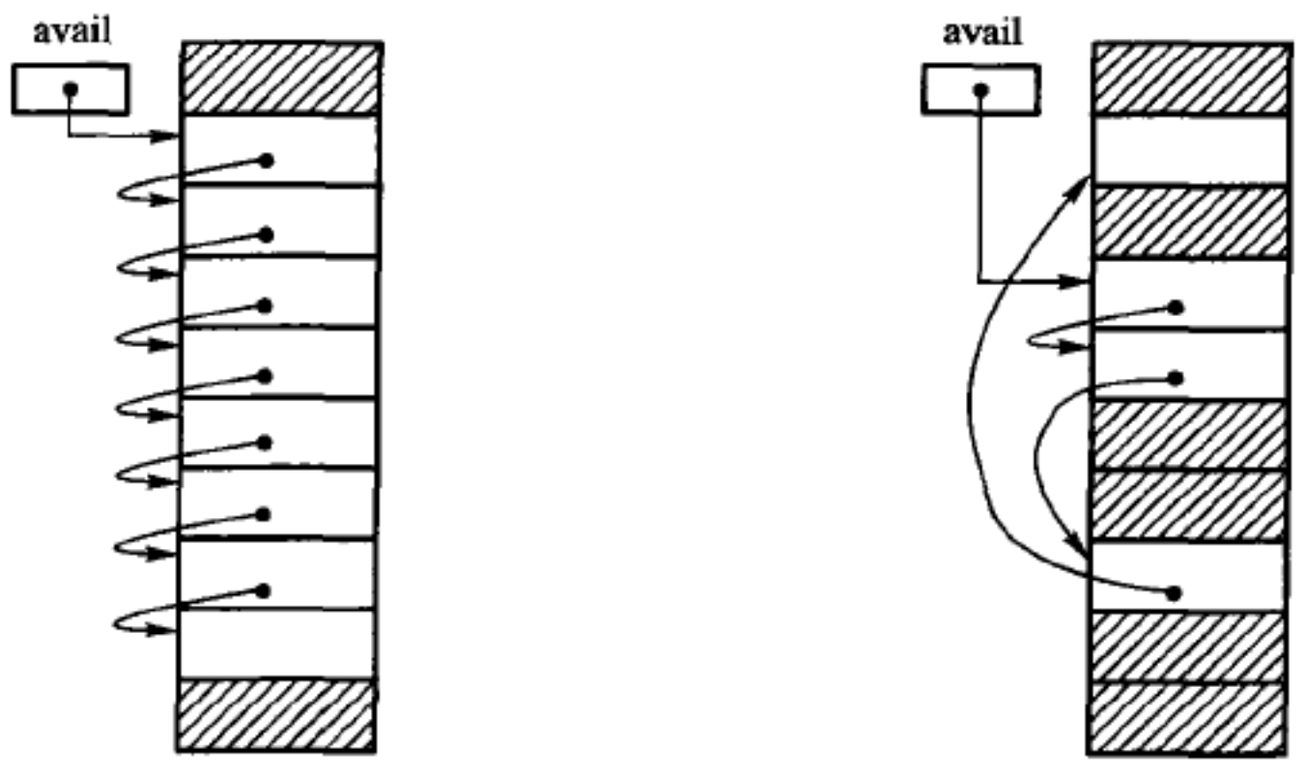
从本质上来看，广义表是线性与树形结构的推广，因此广义表的 ADT 可以借用树形结构的子结点/兄弟结点链，再增加图中的 mark 标志以避免循环访问。

12.2.2 可利用空间表

为了进行动态存储分配，可以把存储器看成一组变长块数组：其中有些块已分配，释放后可以回收；把空闲块链接到一起形成一个可利用空间表(freelist)，用于存储分配。

对于长度固定的存储分配请求，可以把可利用空间表组织成链栈的形式。在系统运行初期将整个可利用空间划分成固定大小的数据块，而且利用指针域把这些数据块链接起来，形成一个单链表。以后每执行一次 new p 操作就从可利用空间中取走一个数据块，并用 p 指向该数据块；每执行一次 delete p 操作就把 p 指向的数据块插入到可利用空间表的链表中。经过一段时间的运行之后，可利用空间表的链接就不再具有连续性，而变得非常杂乱，如图 12.10 所示。图中阴影区域表示已分配的存储区。

假设可利用空间表的链表头指针为 avail，这是一个静态(static)数据成员，由 LinkNode 类的所有对象共享。算法 12.1 给出了删除结点(内存分配)和插入结点(内存回收)的函数。删除算法重载 C++操作符 new，插入算法重载操作符号 delete。new 和 delete 函数的重载使得应用程序可以不做任何修改，还是像没有重载以前一样使用 new 和 delete 函数来编写代码。如果程序员需要直接引用系统的 new 和 delete 操作符，则需要强制用“::new p”和“::delete p”。这种强制在整个程序运行完毕后十分必要，可以把 avail 所占用的空间都交还给系统(真正释放空间)。



(a) 初始状态的可利用空间表　　(b) 系统运行一段时间后的可利用空间表

图 12.10　结点等长的可利用空间表

【算法 12.1】　支持可利用空间表的单链表定义和函数重载。

```
template <class Elem>
class LinkNode{                                          // 可利用空间表结点类
    private:
        static LinkNode *avail;                          // 可利用空间表头指针
    public:
        Elem value;                                      // 结点值
        LinkNode *next;                                  // 指向下一结点的指针
        LinkNode(const Elem & val, LinkNode *p);         // 构造函数
        LinkNode(LinkNode *p = NULL);                    // 构造函数
        void *operator new(size_t size);                 // 重载 new 运算符
        void operator delete(void *p);                   // 重载 delete 运算符
};

// 重载 new 运算符实现
template <class Elem>
void *LinkNode<Elem>::operator new(size_t size){         // size_t 是系统保留类型
    if (avail == NULL)                                   // 可利用空间表为空
        return ::new LinkNode[size];                     // 利用系统的 new 分配空间
    LinkNode<Elem> *temp = avail;                        // 从可利用空间表中取走一个结点
    avail = avail->next;                                 // 可利用空间表栈指针调整
    return temp;                                         // 返回新分配的空间
}

//重载 delete 运算符实现
```

```
template <class Elem>
void LinkNode<Elem>::operator delete(void *p) {
    ((LinkNode<Elem> *) p)->next = avail;        // 结点释放给可利用空间表
    avail = (LinkNode<Elem> *)p;                 // 可利用空间表栈指针调整
}
```

【算法 12.1 结束】

12.2.3 存储的动态分配和回收

算法 12.1 中的简单算法针对只有一种长度的空闲块的情况,但实际应用中要求被分配的空闲块有多种长度。一种常见的情况是动态分配内存时要求为整型、浮点型、双精度浮点型等类型的变量分配存储空间。算法 12.1 也不能很好地解决存储区共享的问题。

本节将介绍把不同长度的结点(空闲块)组织在一个可利用空间表中的技术。分配时并不是所有结点都能满足,而需要按申请的长度进行检索,找到其空间大于等于申请长度的结点,从中截取合适的长度;回收时也不能简单地释放,而必须考虑刚刚被删除的结点空间能否与其他结点合并,组成较大的结点,以便能满足后来较大长度结点的分配请求。

这种处理方法的优点是可以解决存储空间的共享问题,缺点是分配和回收的算法比较复杂,并且在系统长期动态运行的过程中,这种方法有可能使整个空间被分割成许多大小不等的碎块,而某些碎块由于太小而长期得不到使用,这就产生了所谓的碎片问题。

因为空闲块的长度是不定的,所以可利用空间表中每个结点都需要记录本结点的长度。一种常见的方法是,对于一个需要 m 字节空间的请求,存储管理器可能会分配稍多于 m 字节的空间,额外的空间留给存储管理器进行存储管理,例如存放块的标记位、链表指针和块长度。标记位用来区别这个块是空闲块还是已被分配的块。这些信息域使得存储管理器能够根据需要分配和回收空间。

空闲块的分配和回收不是简单地在可利用空间表的一端进行,可利用空间表不再是一个堆栈结构。事实上,为了检索方便,通常用双链表来实现这种可利用空间表。

图 12.11 所示为存储管理器块示例。图 12.11(a)是空闲块的结构,在块的开始处标记位,值“+”表示空闲块的;k 为块长度域;两个指针用于组织可利用空间的双链表结构;块结束处同样也有块长度域和标记位。图 12.11(b)是已分配块的结构,块开始处和结尾处记标记为“-”,表示已分配块;k 为块长度域。

下面将针对图 12.11 的存储块结构,分别讨论可利用空间表的动态分配与回收。

1. 动态分配

动态分配就是要在可利用空间表中找到一个合适的空闲块以满足存储请求。也就是说,为了分配一个 N 字节大小的空间,就要利用某种策略搜索到一个长度大于等于 N 字节的空闲块进行分配;如果可利用空间表中不存在这样的结点,就不能进行分配。

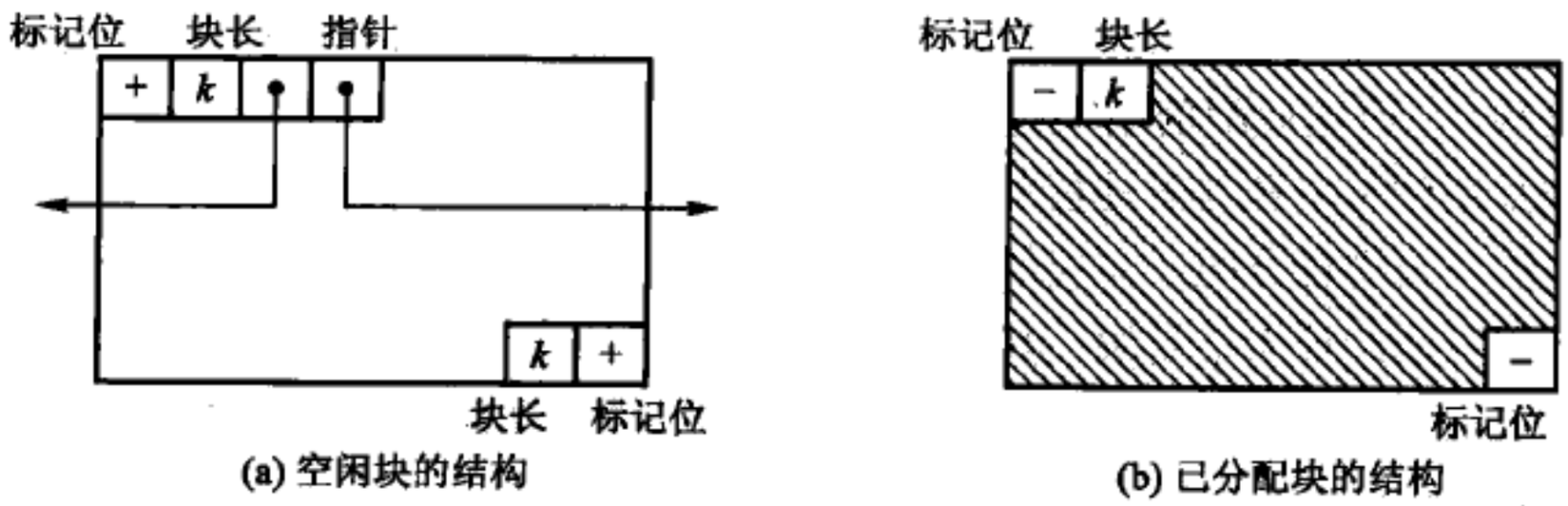


图 12.11 存储器看到的块结构

如果能够找到大于 N 字节的空闲块，一种可能的做法是整个块交给请求者，整个空闲块的首尾标记位都修改为“-”，表示它是已分配块。另一种做法是对于一个长度为 $K(K>N)$ 字节的空闲块，存储管理器把 N 字节分配给请求者，剩余的 $K-N$ 字节形成一个新的空闲块；对已分配块和剩余空闲块设置相应的标记、长度和链指针域。不管采用哪种做法，都会存在所谓碎片(fragmentation)的问题。在动态存储管理中存在两种类型的碎片。像上述的第一种做法，把多于请求字节数的空间分配给请求者，会浪费空闲块的空间，从而产生内部碎片(internal fragmentation)。如果按照前述的第二种做法，把满足请求后的 $K-N$ 个字节作为新的空闲块，则会产生大量的不能满足一般性请求的小空闲块，称为外部碎片(external fragmentation)。这两种碎片的区别如图 12.12 所示。

图 12.12 外部碎片和内部碎片

在实际应用中，有的存储管理器牺牲内部碎片的空间，使得存储管理更加简单。例如，文件管理系统中以簇为基本单位进行文件空间的分配就牺牲了内部碎片从而避免了外部碎片。

下面先介绍一些可能产生外部碎片的方法，这类方法通称为顺序适配(sequential fit)。常见的顺序适配方法有 3 种：首先适配(first fit)、最佳适配(best fit)和最差适配(worst fit)。

首先，适配从可利用空间表的表头开始，顺次在表中进行搜索，一旦找到第一个长度大于等于请求块长度的空闲块，就进行分配。一种简单改进是：记住前一次搜索到达的位置，从该位置开始新的搜索。当搜索到达链表尾部时，重新从链表头处开始搜索。首先，适配的优点是速度快，缺点是可能把较大块拆分成较小的块，导致后来难以满足对大块的申请。

最佳适配在所有大于等于请求长度的块中找出最小的一块进行分配。最佳适配的优点很显然，它可以使得无法满足大请求块的可能性降到最低。但最佳适配可能使得外部碎片问题变得非常严重，因为那些满足请求后剩下的空闲块非常小，对将来的请求已没有什么用处。实现最佳适配有两种方案。其一，检索整个无序的可利用空间表，找到满足分配请求的最小空闲块。其二，把空闲块按照从小到大的顺序排列成优先队列，使得检索时只需要从表头往后查看，直至找

到第一块满足分配请求的空闲块;但回收空闲块时,需要插入到优先队列中合适的位置。这两种方案耗时都比较多,最佳适配的结果是空闲块的长度变化很大。

最差适配采用的是一种与最佳适配完全相反的策略,它在所有长度大于等于请求块长的空闲块中找出最大的一块进行分配。实现最差适配有两种方案,其一,检索整个无序的可利用空间表,找到整个可利用空间的最大空闲块。其二,把空闲块按照从大到小的顺序排列成优先队列,检索时如果表头结点满足分配请求,则把它分配出去,否则无法满足存储分配要求;但回收空闲块时,需要插入到优先队列中合适的位置。最差适配方法试图让剩余部分不太碎小,因此能满足将来的分配需求。最差适配方法将使得空闲块长度趋于一致,适合于存储分配长度请求比较均匀的情况。

例如,假设可利用空间表中包含3个可利用块,其大小分别是1200、1000、3000,现有如下一串存储分配的请求:

600、500、900、2200

(1) 采用最佳适配。分配600以后,空闲块为1200、400、3000;分配500后,空闲块为700、400、3000;分配900后,空闲块为700、400、2100;再分配2200,发生内存溢出。

(2) 采用首先适配。分配600以后,空闲块为600、1000、3000;分配500后,空闲块为100、1000、3000;分配900后,空闲块为100、100、3000;分配2200后,空闲块为100、100、800。

(3) 采用最差适配。分配600以后,空闲块为1200、1000、2400;分配500后,空闲块为1200、1000、1900;分配900后,空闲块为1200、1000、1000;再分配2200,发生溢出。

2. 回收

采用顺序适配方法,当一个块被释放时,需要将其重新链回到可利用空间表中。如果该块与可利用空间表中其他空闲块物理地址相邻,则需要合并这些相邻块,这样存储管理器才能满足将来尽可能大的存储请求。

空闲块与已分配块的结构如图12.13所示。因为空闲块的首尾都有块长度域,所以块合并也变得很简单。首先,检查块 M 左邻的存储单元 L,根据其标记位可以判断块 L 是否为空闲块。如果不是,则简单地把 M 插入可利用空间表。如果如图中所示,L 是空闲块,则把 L 的长度扩展到包含 M。然后,检查 M 的右邻块 N 的标记位,如果块 N 是空闲块,则要把 N 从可利用空间表中删除,同时扩展 M 的长度(在本例则是扩展已包含 M 的 L 的长度)。经过回收操作之后,形成了一个包含原来的 L、M、N 三个块长度的大空闲块。

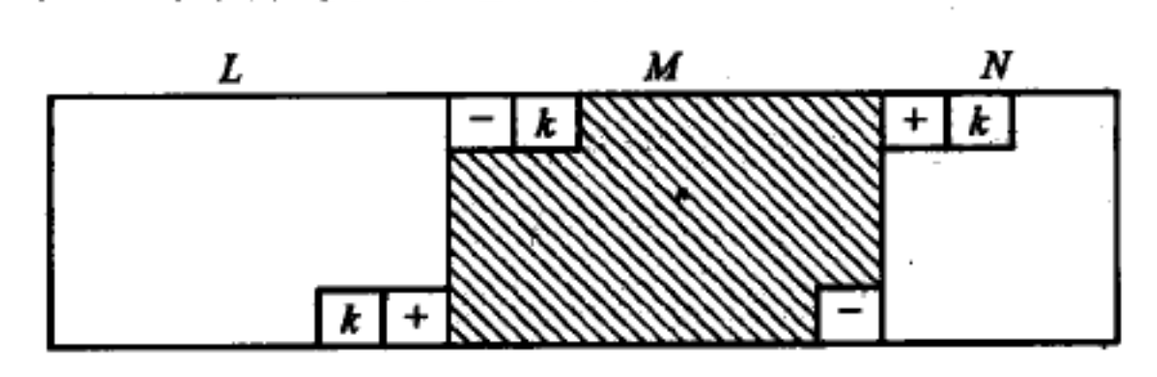


图12.13 把块 M 释放回可利用空间表

很难笼统地讲哪种适配策略最好。需要考虑以下因素来确定采用哪种顺序适配方案:用户的要求、分配或回收效率对系统的重要性、所分配空间的长度变化范围以及分配和回收的频率

等。在实际应用中，由于首先适配其分配和回收的速度比较快，而且支持比较随机的存储请求，因而应用得更广泛。

12.2.4 失败处理策略和无用单元回收

在系统运行过程中，如果遇到因内存不足而无法满足一个存储请求，存储管理器可以有两种行为：一是什么都不做，直接返回一个系统错误信息；二是使用失败处理策略（failure policy）来满足请求。

把外部碎片收集起来可能得到连续的大块存储空间能够满足存储请求，这就是存储压缩（compact）。此过程需要普查整个内存空间，把有用的结点移到内存的一端，把碎片移到另一端，当然同时必须修改所有正在使用的内存结点指针值。存储压缩既费时间又十分危险，通常只有内存碎片很多却分配不出有用空间而即将产生溢出时才使用。

存储压缩之后，应用程序的数据地址将发生变化。如果应用程序以某种方式依赖于数据的绝对位置，就会引起严重的后果。句柄（handle）是对存储位置的二级间接指引，可以使得存储地址相对化。存储管理器在进行内存分配时不是返回一个指向空闲块的指针，而是返回一个指向变量的指针，这个变量指向存储位置，就是通常所说的句柄。移动存储块位置时，只需要修改句柄的值，不需要修改应用程序。

在非实时的系统中也可以采用延迟响应存储请求的策略，直到有足够的存储空间可用为止。例如，一个进程将一直等待，直到存储管理器为它分配了足够的存储空间才开始运行。

最彻底的失败处理策略是无用单元收集（garbage collection）。所谓无用单元是指那些可以回收而没有回收的结点空间。

在高级程序设计语言中，程序员常犯的一种错误就是忘记释放动态申请的变量（或对象），有些情况下程序要做到及时释放无用单元也有一定困难，因为在类似于广义表这样复杂的结构中存在着共享结点和递归成分。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）。如果没有一个程序变量指向一块存储空间，以后就不可能再访问这块空间。

这些无用结点的存在会影响空间的使用效率，因此存储管理系统要有能力把这些无用结点找出来，并送回到可利用的空间表中。程序设计语言采取不同的方法处理无用单元，Java 程序设计语言的无用单元收集器会根据需要回收这些未使用的空间，使得程序员更容易进行程序开发。

无用单元收集的算法较多，首先介绍引用计数（ reference count ）算法。系统为每一个动态分配的存储块加入一个计数域，当一个新指针指向该存储块时，存储块的引用计数就会加 1；当某指针不再指向这个块时，其引用计数就会减 1。一旦存储块的引用计数变为 0，这个存储块就成为无用单元，立即放回到可利用的空间表中。UNIX 的文件系统就采用了这种方法：在 UNIX 中，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；每当删除一个文件时，实际上是把它的链接域简单地减 1；如果文件的链数目减少到 0，就把文件占用的空间释放回可利用的空间表，供重新利用。

引用计数算法的优点是它不需要一个明显的无用单元收集阶段，每当存储单元成为无用单元就立即把它放到可利用的空间表中。但引用计数方法的缺点是必须为每一个存储对象维护一个引用计数。如果有很多较小的对象，系统的额外负担就会太多。当存在循环引用的无用单元时，就会出现另一个严重问题：每个存储对象都被指向一次，但是对象集合仍然是无用单元，因为没有指针指向对象集合。如图 12.14 所示，所有的存储单元都有非零的引用计数，但从整体而言都是无用单元。

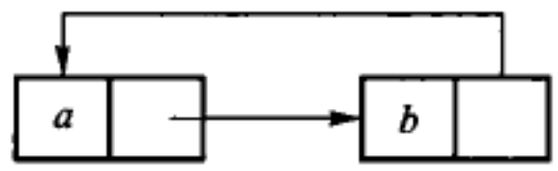


图 12.14 无用单元循环引用

因此，只有当存储对象没有循环地链接在一起时，引用计数才能工作。

少量无用单元的存在并不影响系统的正常运行，关键是当无用单元积累到一定阶段时，怎样从整个存储空间中找出哪些结点是有用的，哪些结点是无用的。另外一种无用单元的收集方法称为标记/清除(mark/sweep)方法。

标记/清除算法的实质就是周游广义表。在这种方法中，每一个存储对象只需要一个简单的标记位。在无用单元收集阶段，首先清除所有的标记位，然后从变量表中的每一个变量开始，沿着指针进行深度优先搜索。每遇到一个存储单元，就设置其标记位为“已访问”。最后，访问所有存储单元，对存储区域进行清理。此时，所有未标记的单元就认为是无用单元，可以释放到可利用的空间表中。

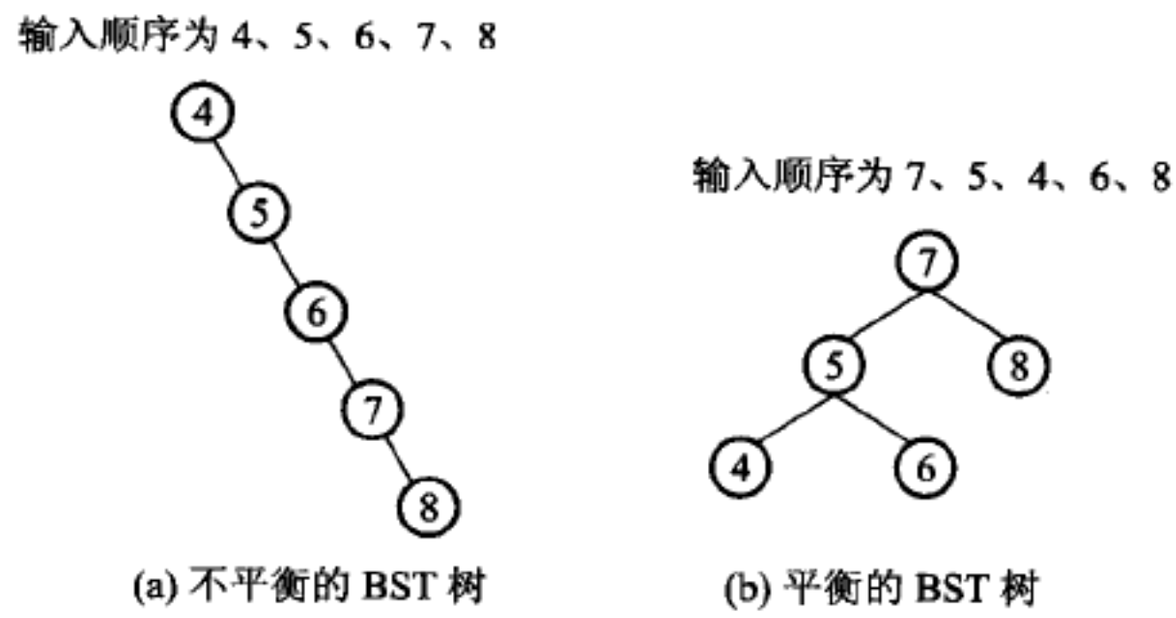
标记/清除方法的优点是它比引用计数方法需要的空间少，而且对于循环情况也能工作。但它隐藏了一个很大的缺陷，这就是进行处理所需要的空间需求。周游广义表的算法实质上是一个递归算法，要么利用递归实现，在这种情况下编译器的运行系统维护一个栈；要么存储管理器维护它自己的栈。而无用单元收集算法往往是在内存空间很紧张、存储分配请求失败时进行的，因此必须尽可能地采用最少的空间代价。

Deutsch - Schorr - Waite 无用单元收集算法是一种比较有效的标记/清除方法。它采用逆转链的广义表周游，而不需要用于栈的额外空间。在周游中每深入一步，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，将其逆转设置为指向前一步刚经过的结点。

12.3 Trie 结构和 Patricia 树

前面 5.6 节所介绍的 BST(二叉搜索树)是一种组织上基于对象空间(object space)分解的数据结构，即关键码范围的分解是由存储在树中的对象(关键码值)决定的。BST 并不是一种平衡的树，其树结构与输入数据的顺序有很大的关系。如果按照升序输入，将导致所生成的 BST 退化成线性表，例如图 12.15(a)，查找一个关键码的时间代价为 $O(n)$。而图 12.15(b)就是比较

平衡的二叉树,其查找时间代价仅为 $O(\log n)$。



(a) 不平衡的 BST 树　　(b) 平衡的 BST 树

图 12.15　BST 的平衡性示意图

若要消除这样的不平衡,最根本的办法就是改变关键码的划分方式。可以在关键码范围内对树结构中每一个结点预定义划分位置。最简单的方法就是对对象(这里是关键码)的范围进行划分,可以平均划分,也可以按照某种方式不平均划分。

假设划分因子为 2,即每次仅把当前范围分为两部分(对应于二叉树)。首先,定义根结点,根结点把整个关键码分为两部分:在进行插入时,不考虑插入的顺序,小于根的关键码都放到左子树中,大于根则放到右子树。按照此规则所产生的树不依赖于关键码的插入顺序,其深度只受关键码精度的影响,因此最坏的情况下就等于存储关键码所需要的位数。例如,如果关键码是 0 ~ 255 之间的整数,那么关键码的精度就是 8 个二进制位。这样,如果有两个关键码:10000010 和 10000011,它们前面的 7 位都是相同的,因此直到第 8 次划分才能将这两个关键码分开,这棵搜索树深度也为 8。

这种对关键码范围进行均分的方法称之为关键码空间(key space)分解。基于关键码分解的数据结构,叫做 Trie 结构。"trie"这个词来源于"retrieval",暗示了它的主要应用:信息检索(information retrieval)。这种数据结构被很多人用来存储英文字符串,尤其是大规模的英文词典,在自然语言理解系统中经常用到。与 $B+$ 树一样,Trie 树内部结点仅作为占位符引导检索过程,数据记录只存储在叶结点中。

这种结构并不能够保证树是平衡的。考虑一种极端的情况,如果所选择的根结点不合适,导致所有的关键码都小于根结点,那么这棵树的右子树将没有任何元素。这是因为关键码分布得很不均衡,导致了树结构失衡。

Trie 树基于两个原则:关键码集合是固定的;能够对结点进行分层标记。如果所有的元素都可以使用数字或者字母等来标记,就可以考虑使用 Trie 结构。例如,如果元素可以用 0 ~ 9 的数字组合来标记,就可以在根结点分出 10 个子结点,分别标记 0 ~ 9,然后每个子结点又可以分出 10 个结点,如此下去直到所有的元素都能够被区分开。

图 12.16 所示为一棵二叉 Trie 树的例子,实际上就是二叉树。每个内部结点都有两个子结点,左结点标记为 0,右结点标记为 1。因为最大元素为 63,所以关键码的范围为[0,64)(2^6 =

64,这是大于 63 的、最小的 2 的幂)。在根结点将关键码范围平分为 2,一半是全部小于 32 的关键码,一半是全部大于等于 32 的关键码,因此 2、5、9、17 被划分到左子树,而 41、45 和 63 被划分到右子树。然后,再划分每个子结点。在 32 ~ 63 的范围内,以 48 为界进行平分。大于 48 的只有 63,所以右子树就是一个叶结点,表示 63。而剩下的元素 41、45 无法在下一层通过判断是否大于 40 来区分,因此要进入更深的一层,通过判断它们是否大于 44 来区分。最后,将所有的元素都放到叶结点上,整个右边子树的划分结束。采用同样方法对左边结点进行划分,最终就得到如图 12.16 所表示的树。

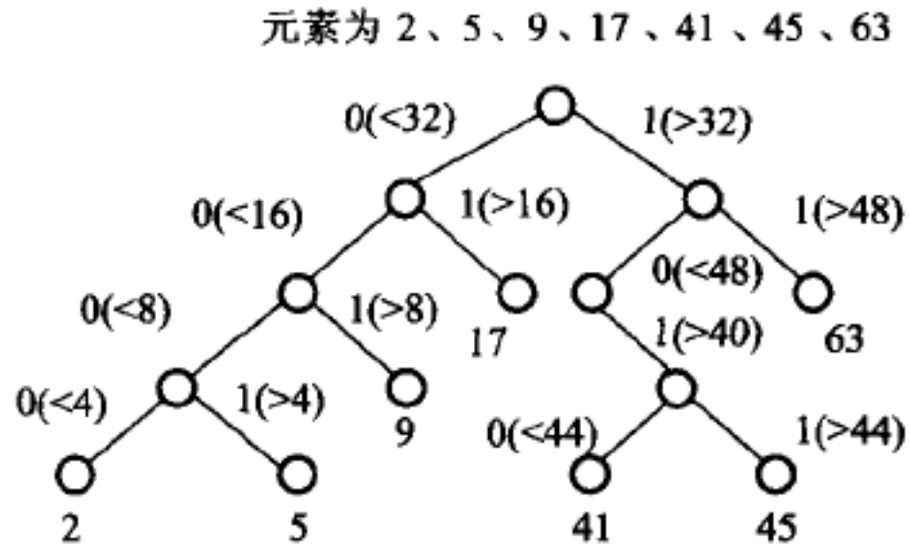


图 12.16 一棵二叉 Trie 树

可以看到,在同一棵子树下面的结点都共享了它们父结点所具有的前缀,例如,2(0000)和 5(0001)。这让人们联想到以前介绍过的 Huffman 编码树(Huffman coding tree),它具有类似的前缀性质,实际上 Huffman 树也是一种二叉 Trie 树。

前面提到了 Trie 结构经常被用于存储字典中的单词,通常称为"字符树",如图 12.17 所示。因为英文的单词是由 26 个字母组成的(为了简单,忽略大小写),所以英文字符树每一个内部结点都有 26 个子结点。如果最长的一个单词长度为 10,那么这棵树在最坏情况下可能有 10 层。这样的树叫做字母表 Trie 结构。

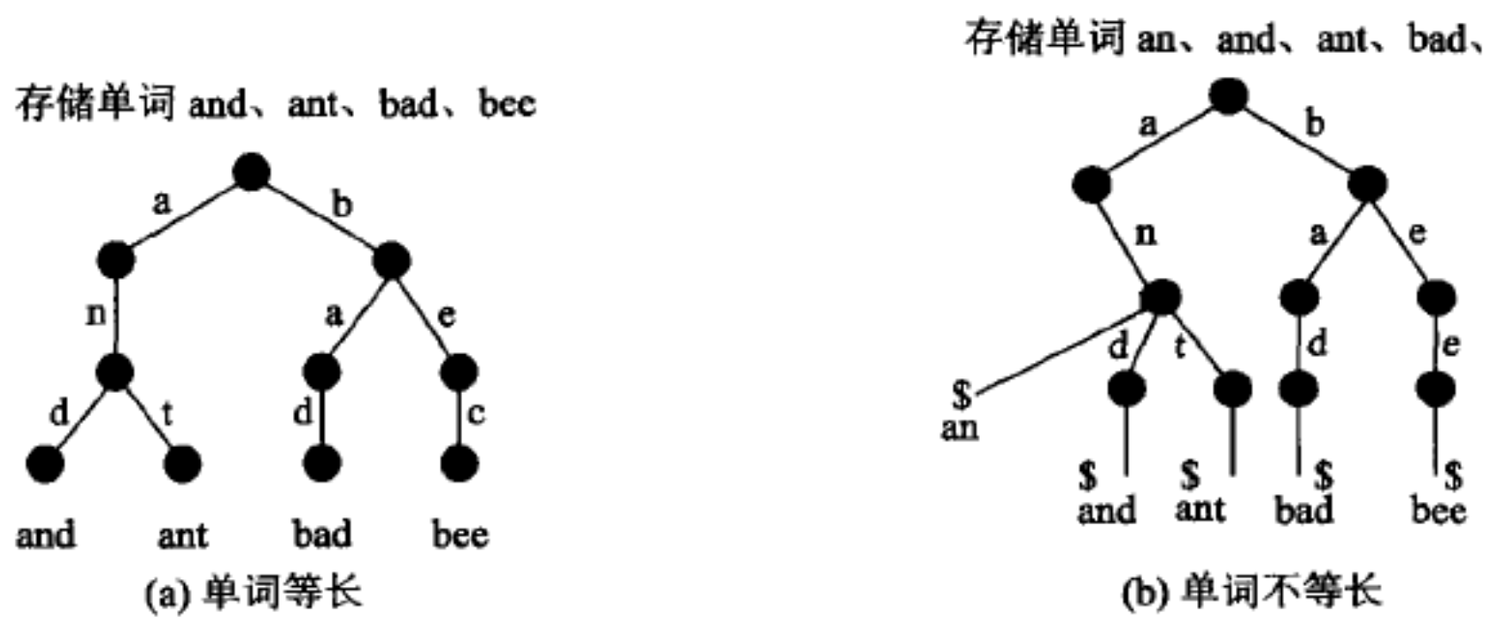


图 12.17 存储单词的 Trie 结构

在字符树中,一棵子树代表具有相同前缀的关键码的集合。例如图 12.17(a),对于单词长度为 3 的单词集{and, ant, bad, bee},"an"子树代表具有相同前缀 an - 的关键码集合{and,

ant},这样的树又叫做前缀树(prefix tree)。在图 12.17(a)中,其内部结点不存储单词信息,只有叶结点才存储单词信息。

在字符树中检索给定的关键码的过程是:首先用待查关键码的第一个字符与各树根所索引的字符相比较,相等则进入到该树;第二步用待查关键码的第二个字符与该树根的各个子结点进行比较,然后再沿着前次比较相等的分支进行进一步的检索,直到进行到某一层,该层所有结点的字符都与待查关键码相应位置的字符不同为止,这说明此关键码在树目录里没有出现;若检索一直进行到树叶,就在树目录里找到了给定的关键码。

在 Trie 树中插入关键码时,首先根据插入记录的关键码找到需要插入的结点位置;如果该结点是叶结点,就将为其分裂出两个子结点,分别存储这个记录和以前的那个记录;如果是内部结点,则在那个分支上应该是空的,所以直接为该分支建立一个新的叶结点即可。删除则与插入相反,如果一个被删除结点的父结点没有其他的子结点,则需要合并,否则只需将此分支设置为空即可。

注意:字符树中的单词可能不等长,例如在 12.17(a)中加入长度为 2 的单词 an,an 是 and 和 ant 的前缀。在这种情况下,需要对单词集进行扩展,增加一个特殊的结束符,用"$"来表示,新的单词集为{an, and, ant, bad, bee}。单词不等长的字符树如图 12.17(b)所示,其内部结点也不存储单词信息,只有叶结点"$"才存储有效的单词信息。

观察 12.17(a)中的 bad,从 a 开始,只要一个分支;不需要继续进行分裂结点的动作,即可以压缩靠近叶结点的单线局部路径。单词 bee 的情况类似。内部单分支路径 an 不需要压缩,避免将来插入新单词时在内部产生过多分裂。图 12.18 是图 12.17(b)的改进。

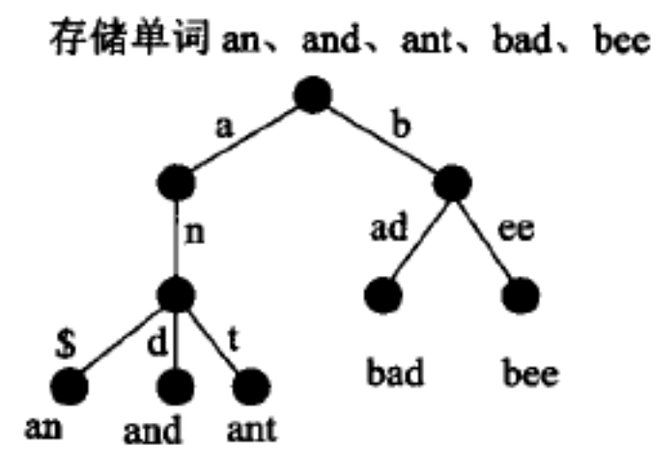


图 12.18 对 Trie 树的改进

Trie 结构显然也不是平衡的。例如,用它来存取英文单词时,显然 t 子树下的分支比 z 子树下的分支多出很多(以 t 开头的单词比以 z 开头的单词多得多)。

而且,每次有 26 个分支因子使得树结构过于庞大,不便于检索。可以使用每个字母的 ASCII 二进制编码来代表这个字母,关键码只有编码 0 和 1,而不是 a ~ z 的 26 个字母,这样就可以把图 12.17 变成一棵二叉 Trie 树。

为了得到更加平衡的结构,D. Morrision 发明了 Trie 结构的一种变体 PATRICIA(Practical Algorithm To Retrieve Information Coded In Alphanumeric)。PATRICIA Trie 不是对关键码大小范围进行划分,而是根据关键码每一个二进制位的编码来划分,所生成的树是二叉树。图 12.19 是对应于图 12.16 的一个 PATRICIA Trie 结构。

结点中的标号表示它是在比较第几位(从 0 开始编位),然后根据那一位是 0 还是 1 来划分左右两个子树,因此标号为 2 的结点其右子树一定是编码形式为 xx1xxx。例如,在图 12.19 中检索关键码 5,5 的编码为 000101(这里取 6 位 2 进制,因为最大的数是 63,用 6 位二进制表示即可)。首先比较第 0 位,进入左子树;然后在第 1 位仍然是 0,还是进入左子树;在第 2 位还是 0,

仍进入左子树;第 3 位变成了 1,进入右子树,就找到了位于叶结点的数字 5。

观察图 12.19,可以发现 41 和 45 在第二位的比较没有差别,可以压缩得到一棵图 12.20 那样更为简洁的树。注意,这里压缩了内部单分支路径,这是因为 PATRICIA 的二进制路径往往比较长,不同于图 12.18 那样的字符树。

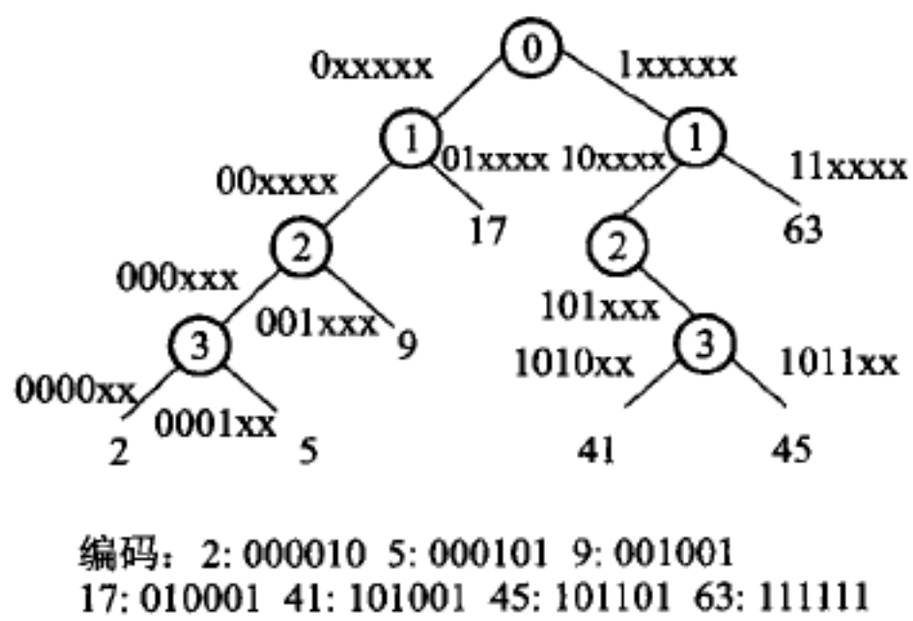


图 12.19 x 表示那个位可以为 1 或者 0

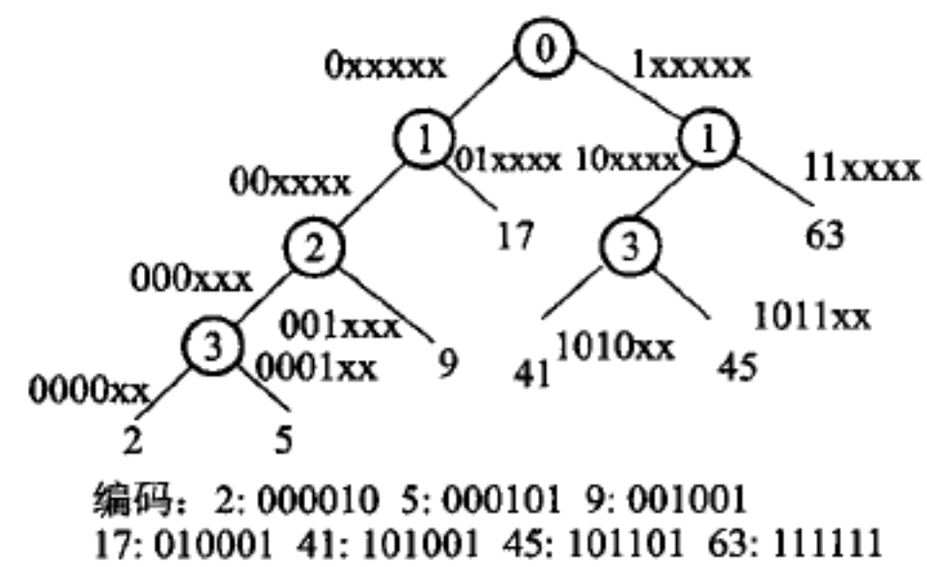


图 12.20 最后得到的 PATRICIA Trie 结构

在 PATRICIA Trie 结构中,每个内部结点都代表一个位的比较,必然产生两个子结点,因此它是一棵满二叉树。如果要进行一次检索,最多只需要比较数次关键码位即可。

对于中文字符串,普通的 Trie 结构不太合适,因为中文词是由汉字组成的。常用的汉字有几千个,而不像英文那样只有 26 个字母。但是,把中文字符串看做二进制代码以后,就可以与英文一样用 PATRICIA Trie 树来处理。实际应用中,PATRICIA Trie 确实是一种比较好的中文字典组织方式。

12.4 改进的二叉搜索树

BST 很有可能变得非常不平衡,例如退化为 n 个结点的线性结构,从而导致检索操作和更新操作的代价为 $O(n)$。

如果希望检索效率非常高,还需要考虑二叉树结点的访问频率。Huffman 树得到总编码长度最小的二叉树,类似的,人们希望得到平均检索长度最短的二叉搜索树,即最佳二叉搜索树。

对于所有结点的概率都相等的情况,最佳二叉搜索树(best BST)类似于完全二叉树形状(但是,并不严格要求最后一层的结点从左到右排列)。要保持完全二叉树形状也需要在插入操作时对 BST 进行很大的调整。平衡的二叉搜索树(balanced BST,或 AVL BST)对 BST 的平衡性要求不太严格,它既照顾到更新操作的代价,又兼顾完成操作后树结构的检索性能,其树高为 $O(\log n)$。

如果要求 BST 总是保持最佳二叉搜索树形状,则需要在更新操作时对树结构进行很大的修改。伸展树(splay tree)不要求 BST 结构总是保持最佳检索状态,而是努力使得每次访问时尽量

使 BST 更加平衡，即检索效率更高。

本节介绍最佳二叉搜索树、平衡的二叉搜索树和伸展树，它们对二叉搜索树的结构或者操作规则进行部分改进，使得树保持类似于平衡的状态，从而达到较好的检索效率。

12.4.1 最佳二叉搜索树

根据 5.6 节 BST 的知识，对应于关键码集合 K：

$K = \{xal, wan, wil, zol, yo, xul, yum, wen, wim, zi, yon, xem, wul, zom\}$

可以得到如图 12.21 所示的二叉搜索树。

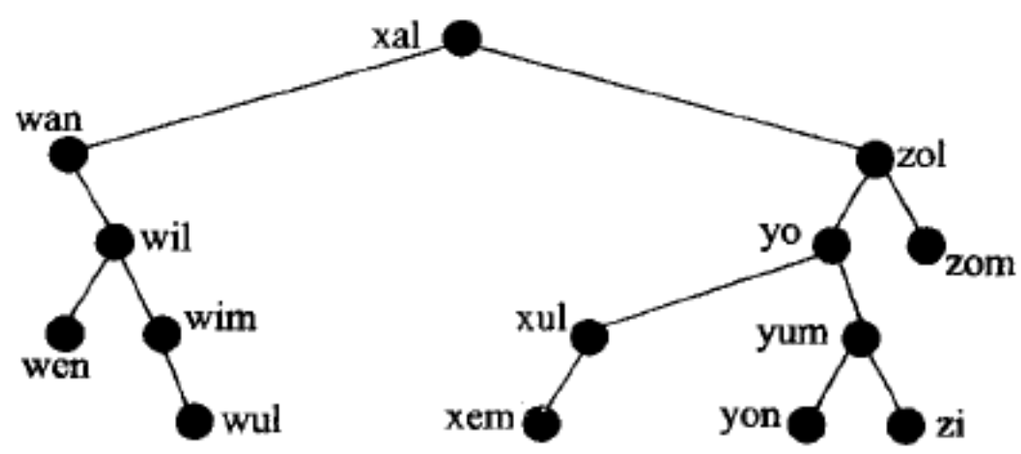


图 12.21 二叉搜索树

同一个关键码集合，其关键码插入二叉搜索树的次序不同，从而构成不同的二叉搜索树。例如，还是上述关键码集合 K，若首先插入最后一个关键码 zom，把它当作根，然后依次插入倒数第二个关键码，倒数第三个关键码，……，就得到如图 12.22 所示的二叉搜索树。

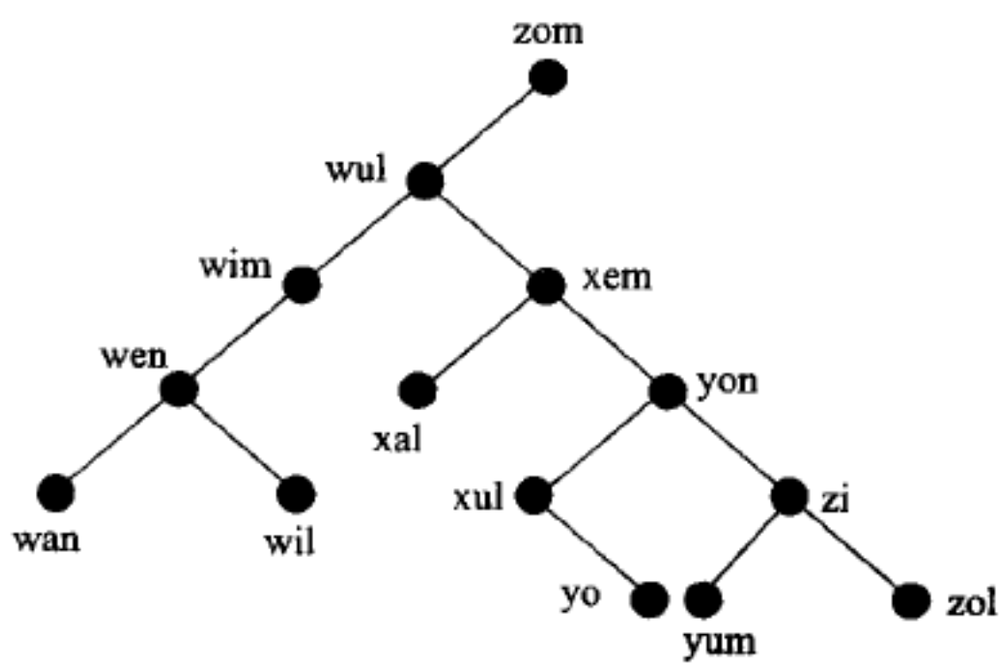


图 12.22 关键码与图 12.21 相同的是一棵 BST

一个拥有 n 个关键码的集合，关键码可以有 $n!$ 种不同的排列法。可能会有读者认为正好可以构成 $n!$ 棵二叉搜索树。其实不然，因为不同排列所构成的二叉搜索树有可能相同。例如，从空的二叉搜索树开始，按照排列 $\{2,1,3\}$ 的顺序插入关键码与按照排列 $\{2,3,1\}$ 的顺序插入所构成的二叉搜索树完全相同。

可以观察到，这种非前序排列的序列总是可以找到与其相对应的一个合法前序排列，从空的

二叉搜索树开始，分别按照这两种排列的顺序插入关键码，最后所得到的二叉搜索树完全相同。事实上，{2,3,1}不是任何二叉搜索树的前序排列。

这个问题等价于：当二叉树的中序排列是{1,2,3,…,n}时，这棵二叉树有多少种可能的前序排列。可以证明，这 $n!$ 种排列中，只有 $c_{2n}^{n}-c_{2n}^{n-1}=\frac{1}{n+1}c_{2n}^{n}$（组合数学中称之为 Catalan 数）种前序排列能够构成二叉搜索树。

通常，可以用检索效率来评价这 $c_{2n}^{n}/(n+1)$ 棵二叉搜索树，什么形状的二叉搜索树比较好。

第 5 章讨论过扩充二叉树的概念，它是满二叉树，新增加的空树叶（以下称为外部结点）的个数等于原来二叉树的结点（以下称为内部结点）个数加 1。对应于图 12.21 的二叉搜索树的扩充二叉树如图 12.23 所示，图中用小方框表示空树叶。

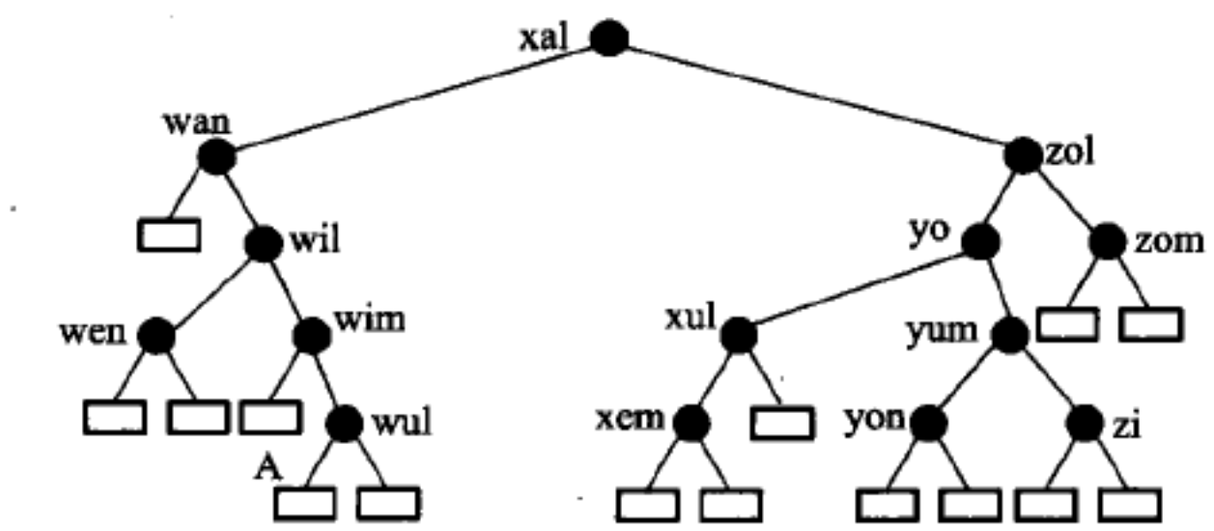


图 12.23 扩充二叉树

在扩充的二叉搜索树里，关键码值最小的内部结点的左子结点（外部结点）代表了值小于该内部结点的可能关键码的集合；关键码值最大的内部结点的右子结点（外部结点）代表了值大于该内部结点的可能关键码的集合；除此之外，每个外部结点代表其值处于原来二叉搜索树的两个相邻结点的关键码值之间的可能关键码的集合。例如，在图 12.23 中，外部结点 A 代表其值处于 wim 和 wul 之间的可能关键码集合。

对于 n 个结点的二叉树，外部路径长度 E 定义为从扩充二叉树的根到每个外部结点的路径长度之和，内部路径长度 I 定义为扩充二叉树里从根到每个内部结点的路径长度之和，则有 $E=I+2n$。

现在回过头来考虑二叉搜索树的检索效率。从根开始比较，若不等就进入下面一层继续比较。因此，对于成功的检索，比较的次数就是关键码所在的层数加 1（根为第 0 层）。对于不成功的检索，比较次数就等于被检索的关键码所属的那个外部结点的层数。

在二叉搜索树里，检索一个关键码的平均比较次数为

$$ASL(n)=\frac{1}{W}\left[\sum_{i=1}^{n}p_i(l_i+1)+\sum_{i=0}^{n}q_i l'_i\right]$$

其中 l_i 是第 i 个内部结点的层数，l'_i 是第 i 个外部结点的层数，p_i 是检索第 i 个内部结点所代表的关键码的频率，q_i 是被检索的关键码属于第 i 个外部结点代表的可能关键码集合（即处于第 i 个和第 $i+1$ 个内部结点之间）的频率。p_i、q_i 也叫做结点的权，令

$$W = \sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i$$

p_i/W 是检索第 i 个内部结点所代表的关键码的概率，q_i/W 是被检索的关键码属于第 i 个外部结点代表的可能关键码集合的概率。

把检索中平均比较次数最小，即 $ASL(n)$ 最小的二叉搜索树称做最佳二叉搜索树。

下面就讨论一下什么样的二叉搜索树是最佳的。

1. 检索所有结点的概率都相等的情况

检索所有结点的概率都相等，即所有结点的权都相等：

$$\frac{p_1}{W} = \frac{p_2}{W} = \cdots = \frac{p_n}{W} = \frac{q_0}{W} = \frac{q_1}{W} = \cdots = \frac{q_n}{W} = \frac{1}{2n+1}$$

此时

$$\begin{aligned} ASL(n) &= \frac{1}{2n+1}\left(\sum_{i=1}^{n}(l_i+1) + \sum_{i=0}^{n} l'_i\right) \\ &= \frac{1}{2n+1}\left(\sum_{i=1}^{n} l_i + n + \sum_{i=0}^{n} l'_i\right) \\ &= \frac{1}{2n+1}(I+n+E) \qquad (注：E = I+2n) \\ &= (2I+3n)/(2n+1) \end{aligned} \tag{12.2}$$

因此，要平均比较次数 $ASL(n)$ 最小，就是要内部路径长度 I 最小。

在一棵二叉树中，路径长度为 0 的结点仅有一个，路径长度为 1 的结点至多有两个，路径长度为 2 的结点至多有 4 个，等等。因此，有 n 个结点的二叉树其内部路径长度 I 至少等于序列

0,1,1,2,2,2,2,3,3,3,3,3,3,3,4,4,…

的前 n 项和，这个和写成 $\sum_{k=1}^{n} \lfloor \log_2 k \rfloor$。

可以证明：

$$\sum_{k=1}^{n} \lfloor \log_2 k \rfloor = (n+1)\lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + 2 \tag{12.3}$$

这个最小的 I 值显然在下面这样二叉树里可以达到：只有最下面的两层结点的度数可以小于 2，其他结点度数必须等于 2。在所有结点的权相等的情况下，这样的二叉搜索树是最佳二叉搜索树，其平均检索长度为：

$$ASL(n) = \frac{2(n+1)\lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n + 2 \rfloor} + 4 + 3n}{2n+1} \tag{12.4}$$

这个 $ASL(n)$ 是 $\Theta(\log_2 n)$ 量级的。

图 12.24 所示的最佳二叉搜索树可以这样构造：首先将集合 K 里的关键码排序

{wan, wen, wil, wim, wul, xal, xem, xul, yo, yon, yum, zi, zol, zom}

然后，用二分法依次检索这些关键码，并把在检索中遇到的在二叉搜索树中还没有的关键码依次

插入二叉搜索树中，这样就可以得到最佳的二叉搜索树。

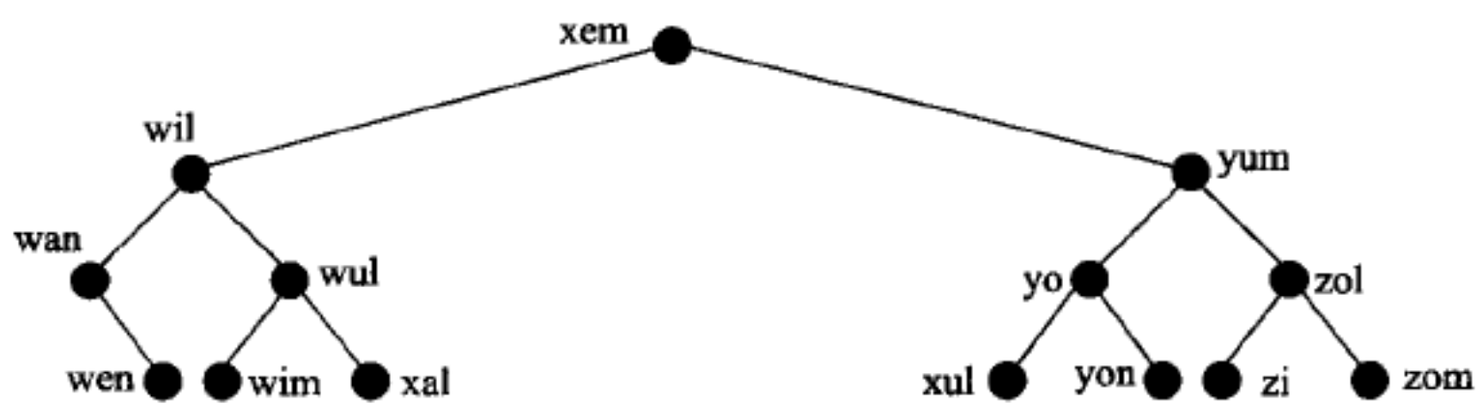


图 12.24 最佳二叉搜索树

反过来，如果关键码按值递增的顺序依次插入到二叉搜索树中，则将得到退化为线性的二叉搜索树，如图 12.25 所示。对这样的二叉搜索树进行检索，实际上就是对线性表的顺序检索，平均比较次数为 $O(n)$。

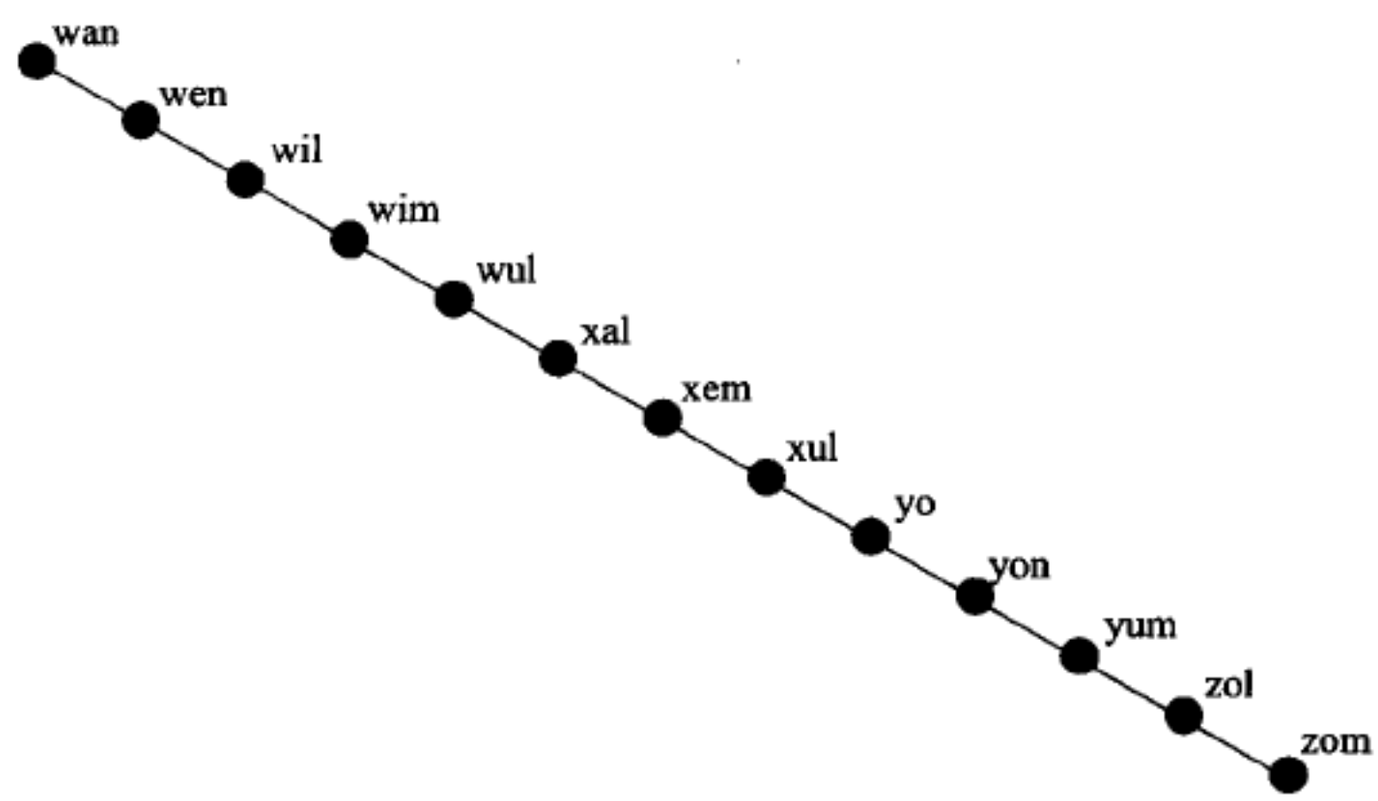


图 12.25 退化为线性的二叉搜索树

按任意顺序把关键码插入到二叉搜索树中，其检索效率如何呢？平均比较次数是接近最坏的情况 $O(n)$，还是接近最好的情况 $O(\log_2 n)$？可以证明，对 $c_{2n}^{n}/(n+1)$ 种二叉搜索树进行平均，得到的平均检索次数仍是 $O(\log_2 n)$。即大多数的二叉搜索树和最佳二叉搜索树检索效率差别不大，只有少数情况下，检索的平均比较次数为 $O(n)$。

2. 检索各结点的概率不相等的情况

检索各结点的概率不相等的情况，即在具有不等权结点的二叉搜索树中进行检索。在这种情况下，一般来说图 12.24 所示的那种二叉搜索树不是最佳的，例如图 12.26 所示的由同一个关键码集合组成的两棵不同的二叉搜索树就是这样（图中结点旁边的数字表示权重）。

现在的问题是，给定一个排好序的关键码集合 $\{key_1, key_2, \cdots, key_n\}$ 以及权的集合 $\{p_1, p_2, \cdots, p_n, q_0, q_1, \cdots, q_n\}$，此时构造使得 $ASL(n)$ 值最小的最佳二叉搜索树，也就是要确定使得二叉搜索树的开销 (cost) $\sum_{i=1}^{n} p_i(l_i+1) + \sum_{i=0}^{n} q_i l'_i$ 为最小的二叉搜索树。

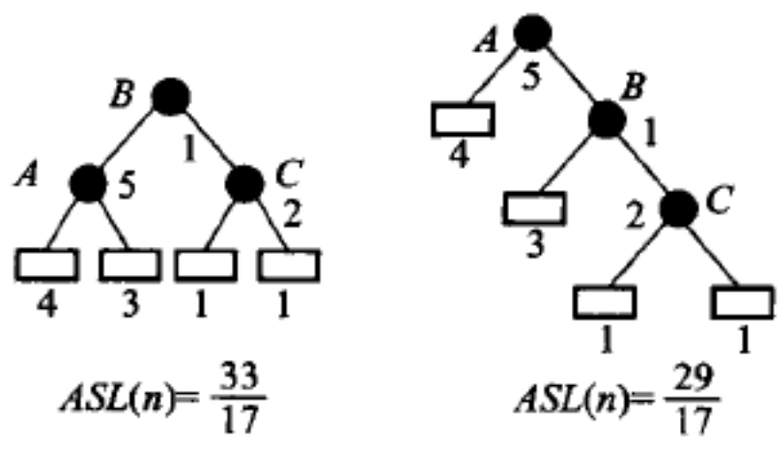


图 12.26 具有不等权结点的二叉搜索树

最佳二叉搜索树有一个特点:其任何子树都是最佳二叉搜索树。利用这一事实,可以从底层逐步构造最佳二叉搜索树:先构造包含一个结点的最佳二叉搜索树,再构造包含两个结点的最佳二叉搜索树,……,直到把所有的结点都包含进去。后来构造的较大的最佳二叉搜索树用前面构造的较小的最佳二叉搜索树作为其子树。

设包含关键码 $key_{i+1}, key_{i+2}, \cdots, key_j$ 为内部结点($0 \leqslant i \leqslant j \leqslant n$),结点的权为($q_i, p_{i+1}, q_{i+1}, \cdots, p_j, q_j$)的最佳二叉搜索树为 $t(i,j)$,其根为 $r(i,j)$,其开销为 $C(i,j)$,其权的总和为 $W(i,j) = p_{i+1} + \cdots + p_j + q_i + q_{i+1} + \cdots + q_j$。

第一步构造包含一个结点的最佳二叉搜索树,即找 $t(0,1), t(1,2), \cdots, t(n-1,n)$。

第二步构造包含两个结点的最佳二叉搜索树,即找 $t(0,2), t(1,3), \cdots, t(n-2,n)$。

再构造包含 3 个、4 个结点的最佳二叉搜索树,……,直到最后构造 $t(0,n)$。

如何构造最佳二叉搜索树 $t(i,j)$? 根据最佳二叉搜索树的子树必是最佳二叉搜索树的原理可知:

$$C(i,j) = W(i,j) + \min_{i<k\leqslant j}(C(i,k-1) + C(k,j)) \tag{12.5}$$

公式 12.4 的意思是,用下列方法来找包含 $key_{i+1}, key_{i+2}, \cdots, key_j$ 为内部结点的最佳二叉搜索树 $t(i,j)$:分别用 $key_{i+1}, key_{i+2}, \cdots, key_j$ 为根,考虑 $j-i$ 棵二叉搜索树。以 key_k 为根的二叉搜索树其左子树包含 $key_{i+1}, \cdots, key_{k-1}$,而包含这些关键码为内部结点的最佳二叉排序树 $t(i,k-1)$ 已在前面的步骤确定,$C(i,k-1)$ 已求出;其右子树包含 $key_{k+1}, key_{k+2}, \cdots, key_j$,以这些关键码为内部结点的最佳二叉搜索树 $t(k,j)$ 也已在前面的步骤确定,$C(k,j)$ 已求出。对于 $i<k\leqslant j$ 找出使 $C(i,k-1)+C(k,j)$ 最小的那个 k_0,以 key_{k_0} 为根,$t(i,k_0-1)$ 为左子树,$t(k_0,j)$ 为右子树的那棵二叉搜索树就是所要求的 $t(i,j)$。其开销 $C(i,j)$ 等于其根的左子树的开销 $C(i,k_0-1)$ 加上右子树开销 $C(k_0,j)$,再加上结点的总的权 $W(i,j)$。

每一步构造出一棵最佳二叉搜索树 $t(i,j)$ 就记下其根 $r(i,j)$、开销 $C(i,j)$,最后构造出 $t(0,n)$,此时就确定了整个最佳二叉搜索树的结构。

算法 12.2 给出了构造具有不等权结点的最佳二叉搜索树算法。

【算法 12.2】 最佳二叉搜索树算法。

```
// 动态规划算法求解最优二叉搜索树问题
// 输入参数:a[ ]内部结点的权,b[ ]外部结点的权,n 结点数
```

```
// 输出参数:w[ ][ ]权的总和,r[ ][ ]结果二叉树,c[ ][ ]构造树的代价
void OptimalBST(int a[ ], int b[ ], int n, int c[N+1][N+1], int r[N+1][N+1], int w[N+1][N
+1]) {
    int i, j;                                          // i、j 为下标变量
    int d,m;                                           // d 为结点数,m 记录权重值
    int k0, k;                                         // 记录根结点
    for (i = 0; i <= n; i++)                           // 数组元素清零
      for (j = 0; j <= n; j++) {
            c[i][j] = 0;            r[i][j] =0;
            w[i][j] = 0;
      }
    // 初始化,确定每一个总权和 w[i,j]
    for (i = 0; i <= n; i++) {
          w[i][i] = b[i];
          //求出权和 w[i.j]
          for (j = i+1; j <= n; j++)
            w[i][j] = w[i][j-1] + a[j] + b[j];
    }

    //确定一个结点的最佳二叉搜索树,一个结点的 c[j-1, j]就是对应的 w[j-1, j]
    for (j = 1; j <= n; j++) {
          c[j-1][j] = w[j-1][j];
          r[j-1][j] = j;
    }

    // 确定 d 个结点的最佳二叉树
    for (d = 2; d <= n; d++) {
          for (j = d; j <= n; j++){
                i = j - d;
                m = c[i+1][j];
                k0 = i + 1;                            // 以 K_{i+1}为根
                for (k = i+2; k <= j; k++) {
                      if (c[i][k-1] + c[k][j] < m) {   // 动态规划的核心步骤
                            m = c[i][k-1] + c[k][j];   // 把当前最小权值记录下来
                            k0 = k;
                      }
                }
                c[i][j] = w[i][j] + m;                 // 记录最小权
```

```
                r[i][j] = k0;                    // 记录树根
            }
        }
    }
```

【算法 12.2 结束】

例如，给出关键码集合 $\begin{pmatrix} B, & D, & F, & H \\ key_1, & key_2, & key_3, & key_4 \end{pmatrix}$ 及权的序列 $\begin{pmatrix} 1, & 5, & 4, & 3, & 5, & 4, & 3, & 2, & 1 \\ p_1, & p_2, & p_3, & p_4, & q_0, & q_1, & q_2, & q_3, & q_4 \end{pmatrix}$。

表 12.1 给出了关键码和内部外部结点的权值列表，构造最佳二叉搜索树的过程如图 12.27 所示。其中，$r(i,j)$、$C(i,j)$、$W(i,j)$分别如表中矩阵所示。对于包含 n 个关键码的集合，构造最佳二叉搜索树需要进行的计算次数为 $\sum_{j=1}^{n}\sum_{i=0}^{j-1}(j-i+1) \approx \frac{n^3}{6}$，这是计算 $C(i,j)(n \leqslant i < j \leqslant n)$所需的计算次数。这个数目还可以缩减到 $O(n^2)$，但是对于大的关键码集合，构造最佳二叉搜索树的代价仍然非常高。

表 12.1 最佳二叉搜索树的关键码和内部外部结点权值列表

标号 i		1		2		3		4	
关键码 key		B		D		F		H	
权值编码	q_0	p_1	q_1	p_2	q_2	p_3	q_3	p_4	q_4
权值	5	1	4	5	3	4	2	3	1

上面所介绍的最佳二叉搜索树的构造方法是一种很典型的动态规划算法(dynamic programming)。动态规划是一种非常经典的算法设计策略，可将问题分解为若干个子问题，先自底向上分别求解最优子结构，然后由这些子问题的解得到原问题的解。

动态规划法的最优子结构递归定义与分治法类似(例如二分检索、快速排序、归并排序这些典型的分治策略算法)。分治策略要求子问题之间相互独立，可以根据最优子结构直接用递归法求解；而动态规划方法所处理的子问题相互交叉。直接用递归法来求解具有重叠子问题性质的后一类问题时，经常要大量重复计算公共的子问题；而动态规划对每个子问题仅求解一次，并将结果填写到若干张表中，下次计算同一子问题时直接利用表中的结果。动态规划的具体形式虽然多种多样，但一般都具有“填表”这一基本模式。

动态规划与分治策略在分析最优解的结构(最优算子)时非常类似。很多人通常忽视子问题相互交叉的情况，而直接从最优解的结构(往往是递归定义结构)写出递归算法。很多情况下，递归算法的时间代价往往是 $O(2^n)$，注意，这一类问题必须用动态规划法来解决。

本书 7.5 节介绍的 Dijkstra 算法、Floyd 算法都是动态规划法，其中 Dijkstra 退化成了贪心法。

i \ j	0	1	2	3	4
0	0	1	2	2	2
1		0	2	2	3
2			0	3	3
3				0	4
4					0

$r(i,j)$

i \ j	0	1	2	3	4
0	0	10	28	43	57
1		0	12	27	40
2			0	9	19
3				0	6
4					0

$C(i,j)$

i \ j	0	1	2	3	4
0	5	10	18	24	28
1		4	12	18	22
2			3	9	3
3				2	6
4					1

$W(i,j)$

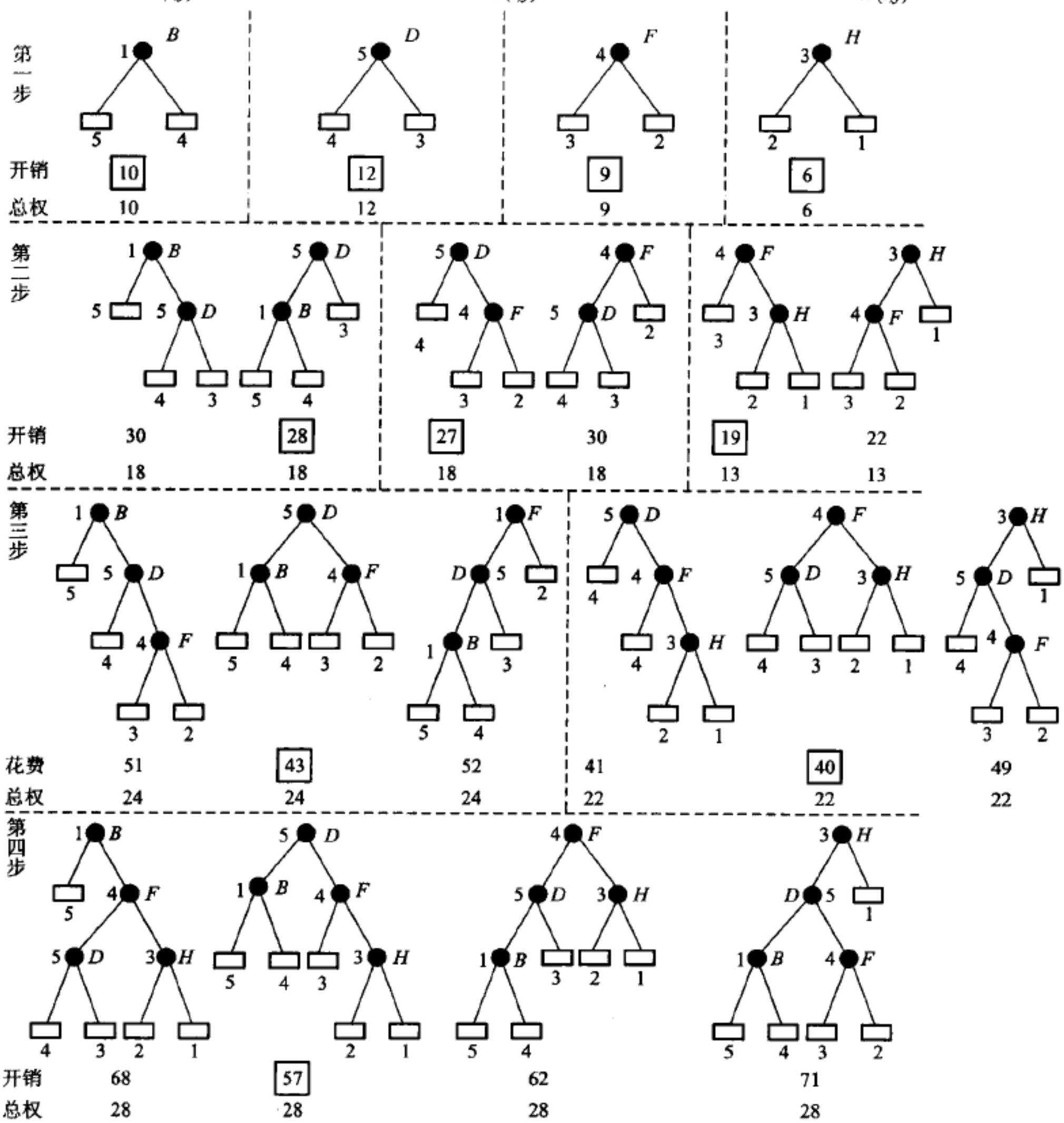


图 12.27 构造最佳二叉搜索树的过程

12.4.2 平衡的二叉搜索树

BST 在最好的情况下搜索只需要 $O(\log n)$ 的时间代价，而在最坏时却需要 $O(n)$ 的时间。如果能够找到一种方法，使得 BST 不受输入的影响，始终保持平衡状态，就能得到很好的检索效率。基于这种想法，Adelson - Velskii 和 Landis 发明了 AVL（发明者姓氏的首字母）树，这是一种平衡的二叉搜索树，其任何结点的左子树和右子树高度最多相差 1。

AVL 树的性质如下：

（1）一棵空二叉树是一个 AVL 树。

（2）如果 T 是一棵具有 n 个结点的 AVL 树，则其高度为 $O(\log n)$。

（3）如果 T 是一棵 AVL 树，那么其左右子树 T_L、T_R 也是 AVL 树，并且 $|h_L - h_R| \leqslant 1$，h_L、h_R 是其左右子树的高度。

可以把 AVL 树看做是 BST 的一个特例，例如图 12.28(a)所示的 BST 就是一个 AVL 树，而图 12.28(b)所示的 BST 就不是一个 AVL 树。

因为 AVL 树的高度为 $O(\log n)$，所以在其上进行的操作，例如搜索、插入和删除等，都只需要 $O(\log n)$ 的时间代价。现在关键的问题就是如何保持一棵 AVL 树的结构，使得不论对其进行何种操作，都不改变其 AVL 特性。并且，为保持这样的树结构所进行的操作都要在 $O(\log n)$ 的时间内完成，以保证总时间代价仍然在 $O(\log n)$ 时间内。

为了方便描述，下面介绍一个新概念——平衡因子（balance factor），用 $bf(x)$ 来表示结点 x 的平衡因子。定义 $bf(x) = x$ 的右子树高度 $- x$ 的左子树高度。

对于一个 AVL 树中的结点，其平衡因子可能取值为 0，1 和 -1，如图 12.29 所示。

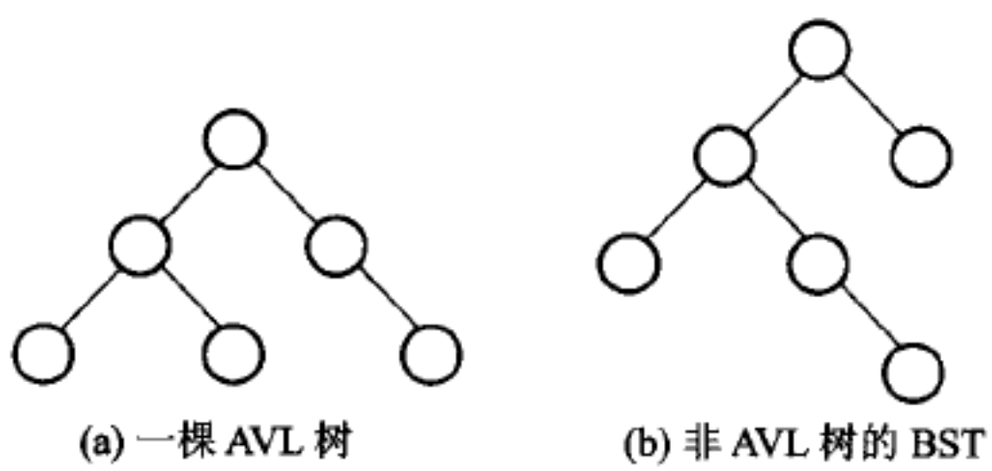


图 12.28 AVL 树与非 AVL 树

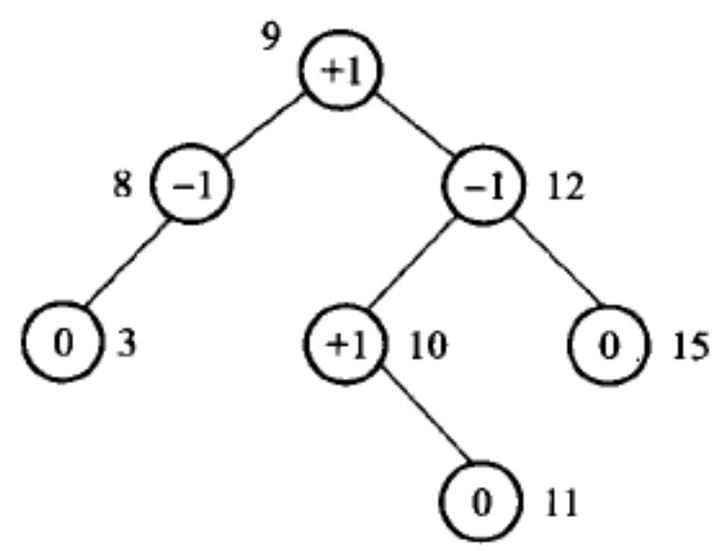


图 12.29 带平衡因子的 AVL 树

1. AVL 树结点的插入

相对于其他操作，插入操作最为重要，可以通过不停地向一棵空二叉树插入结点来建立任意一棵 AVL 树。因此，先讨论 AVL 树的插入操作。

插入一个新结点作为叶结点时，相应子树的根结点变化大致有三类：

（1）结点原来是平衡的，现在成为左重或右重的，此时该结点的前驱结点的状态也要发生变化。

(2) 结点原来是某一边重的，而现在成为平衡的，此时前驱结点不改变状态，因为这棵子树的高度未变。

(3) 结点原来就是左重或右重的，而新结点又加到重的一边，此时这个结点就不再满足 AVL 树条件，这样的结点叫做“危急结点”。

第(3)种情况将导致新树不是 AVL 树。例如，原来平衡的 AVL 树中插入元素 17 将出现图 12.30(a)所示的情况，结点 12 的右子树高度比左子树大 2，破坏了 AVL 树的定义，而正确的结构应该是如图 12.30(b)所示的形式。

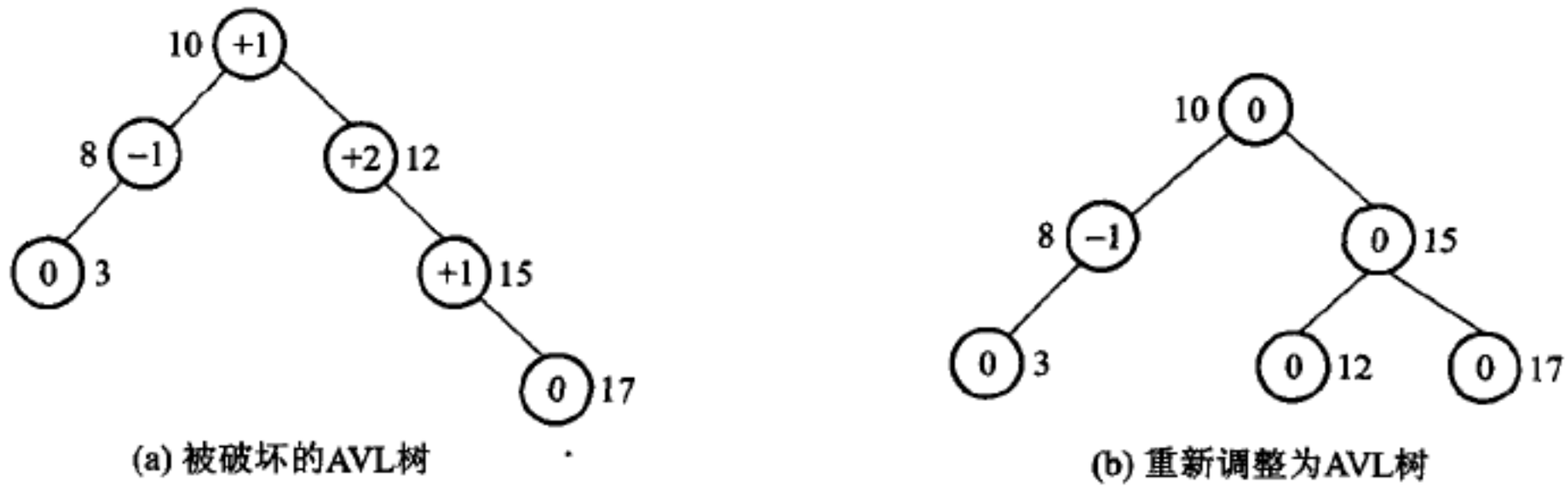


图 12.30 AVL 树的调整示意图

为了恢复平衡，需要移动不平衡子树的结点位置。可以根据每个结点的平衡因子来判断一棵子树是否平衡。正常情况下，一个结点 A 的平衡因子只能是 0、1、−1，而在非正常情况有如图 12.31 所示的 4 种可能：

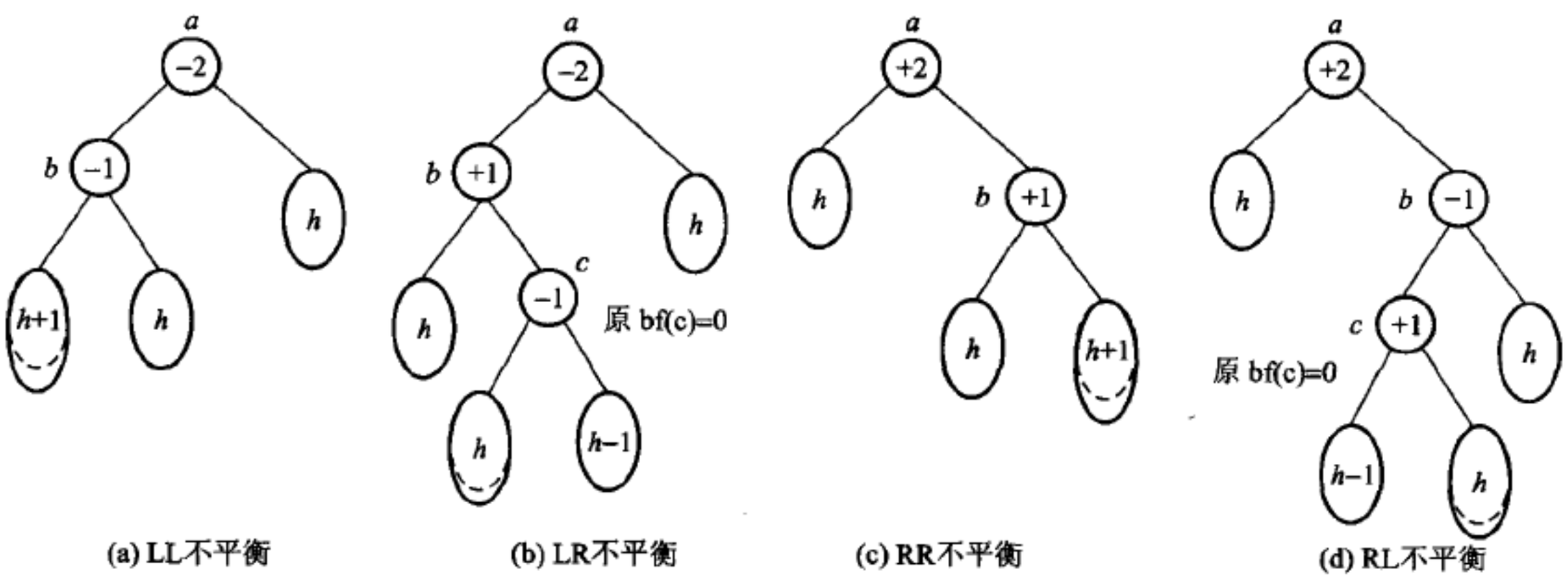


图 12.31 AVL 树的不平衡类型。圆圈表示结点；椭圆表示子树，内部符号表示其高度

(1) LL 型：导致不平衡的结点为 A 的左子树的左子树，此时 A 的平衡因子为 −2。

(2) LR 型：导致不平衡的结点为 A 的左子树的右子树，此时 A 的平衡因子为 −2。

(3) RL 型：导致不平衡的结点为 A 的右子树的左子树，这时 A 的平衡因子为 2。

(4) RR 型：导致不平衡的结点为 A 的右子树的右子树，这时 A 的平衡因子为 2。

其中,LL 型和 RR 型是对称的,LR 型和 RL 型是对称的。不平衡的结点一定在根结点与新加入结点之间的路径上,并且其平衡因子只能是 2 或者 -2,如果是 2 表示它在插入前的平衡因子是 1,如果是 -2 表示它在插入前的平衡因子是 -1。

可以使用一系列称为旋转(rotation)的局部操作来解决这个问题。第 1 种情况和第 4 种情况可以通过单旋转(single rotation)来解决,第 2 种情况和第 3 种情况可以通过双旋转(double rotation)来解决。这样调整的整个过程称为重构(restructuring),过程如下:

(1) 首先,从新加入的结点 u 向根的方向往上搜索 BST(不称其为 AVL,是因为可能出现不平衡的情况),直到找到距离 u 最近的不平衡祖先结点 a(平衡因子为 2 或者 -2)。如果直到根都没有找到这样的祖先,则不做任何动作(已经平衡),返回。

(2) 令 a 结点指向结点 u 的路径上的第一个子结点为 b,第二个子结点为 c,然后根据树的形态进行单旋转或者双旋转。

- 如果是单旋转,调整过程只涉及 a 和 b 这两个结点。对图 12.32 所示的 LL 调整,令 b 代替 a 成为新根,a 作为 b 的右子结点,a 和 b 的子树根据 BST 的性质分别挂接到 b 和 a 结点下,保持中序周游序列为 $T_0\ b\ T_1\ a\ T_2$。RR 不平衡时,进行类似的调整。

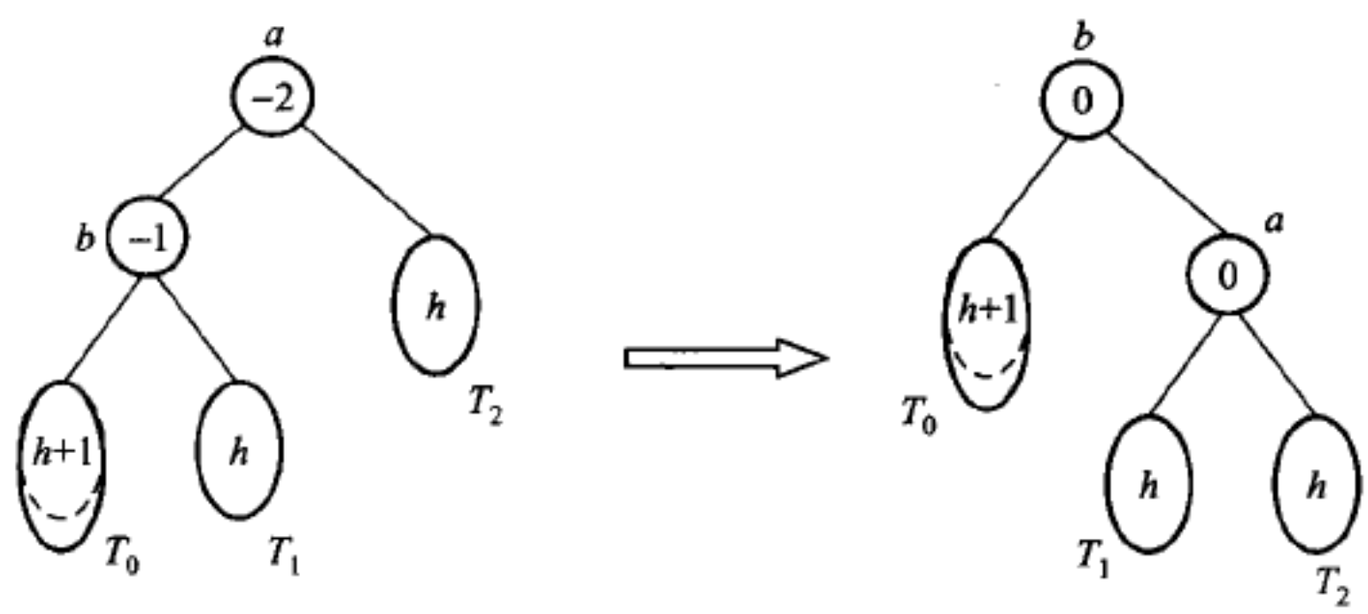


图 12.32 LL 型不平衡经过单旋转后重新变成平衡结构,RR 型单旋转类似

- 如果是双旋转,对图 12.33 所示的 RL 型进行调整。此时,先作(c,b)右转,把 T_2 接到 b 上;然后,再作(a,c)左转,把 T_1 接到 a 上。

下面解释一下旋转过程。以双旋转图 12.33 为例,参加旋转的总共有 7 个部分。3 个结点:a、b、c;4 棵子树 a 的左子树 T_0,b 的右子树 T_3,c 的左子树 T_1 和右子树 T_2。不管这 7 个部分以前是如何组织为一棵不平衡的树,旋转的目的只是使它们重新组成一个新的 AVL 结构。将这 7 部分按照下面的方式放入一个数组:

T_0	a	T_1	c	T_2	b	T_3

数组满足中序周游的性质:$T_0 < a < T_1 < c < T_2 < b < T_3$。

旋转后得到的新树,保持了原来的中序周游顺序:$T_0\ a\ T_1\ c\ T_2\ b\ T_3$。而且,在以上旋转处理中仅需改变少数几个指针,新的子树高度为 $h+2$,保持插入前子树的高度不变,因此原来二叉树在

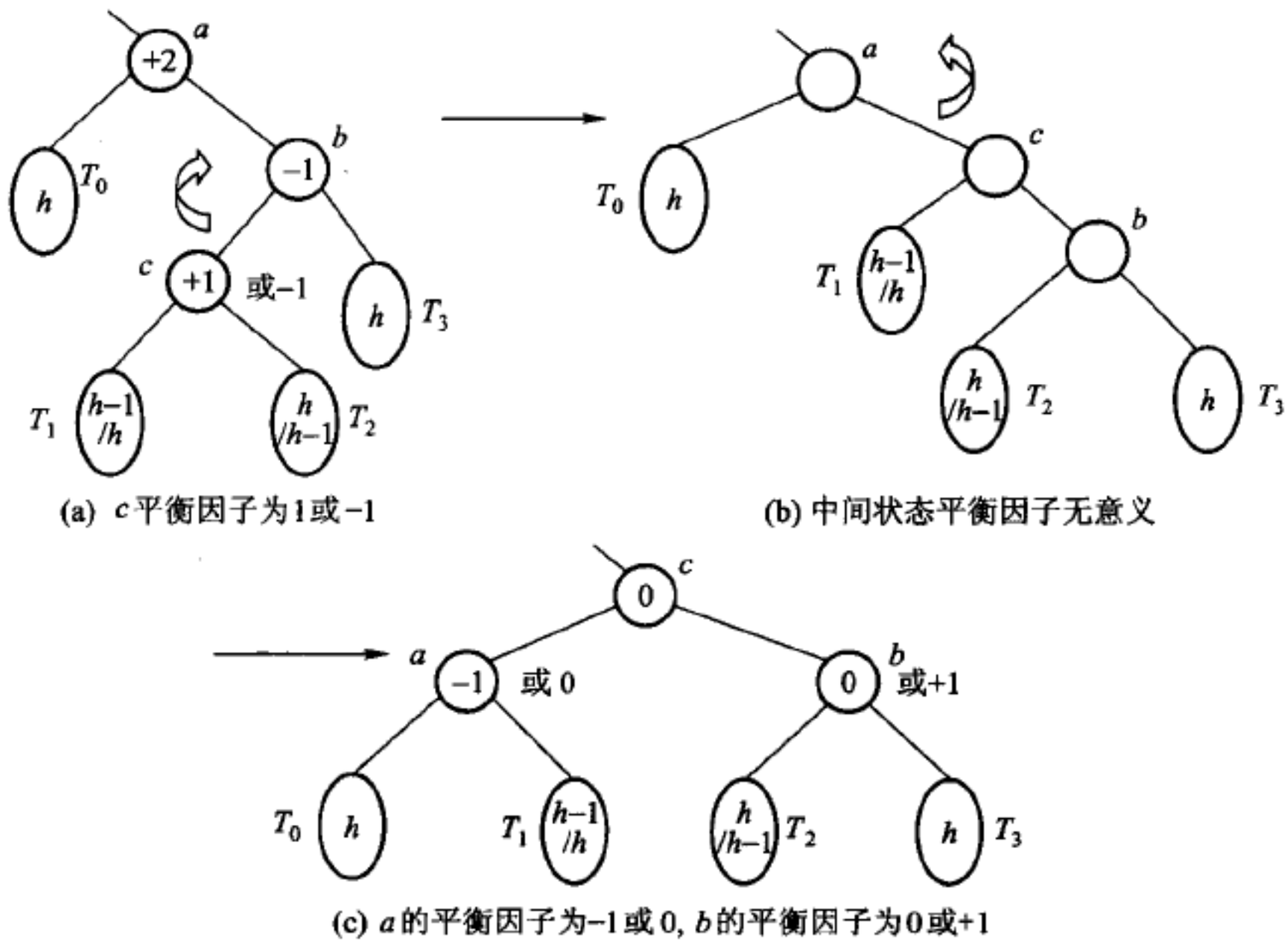


图 12.33 RL 型双旋转示意图，LR 型双旋转类似

a 结点上面的其余部分(若有)总是保持平衡的。

图 12.34 是从空 AVL 树开始，顺序插入单词{cup, cop, copy, hit, hi, his, hia}后得到的 AVL 树，单词之间按照字典顺序比较。

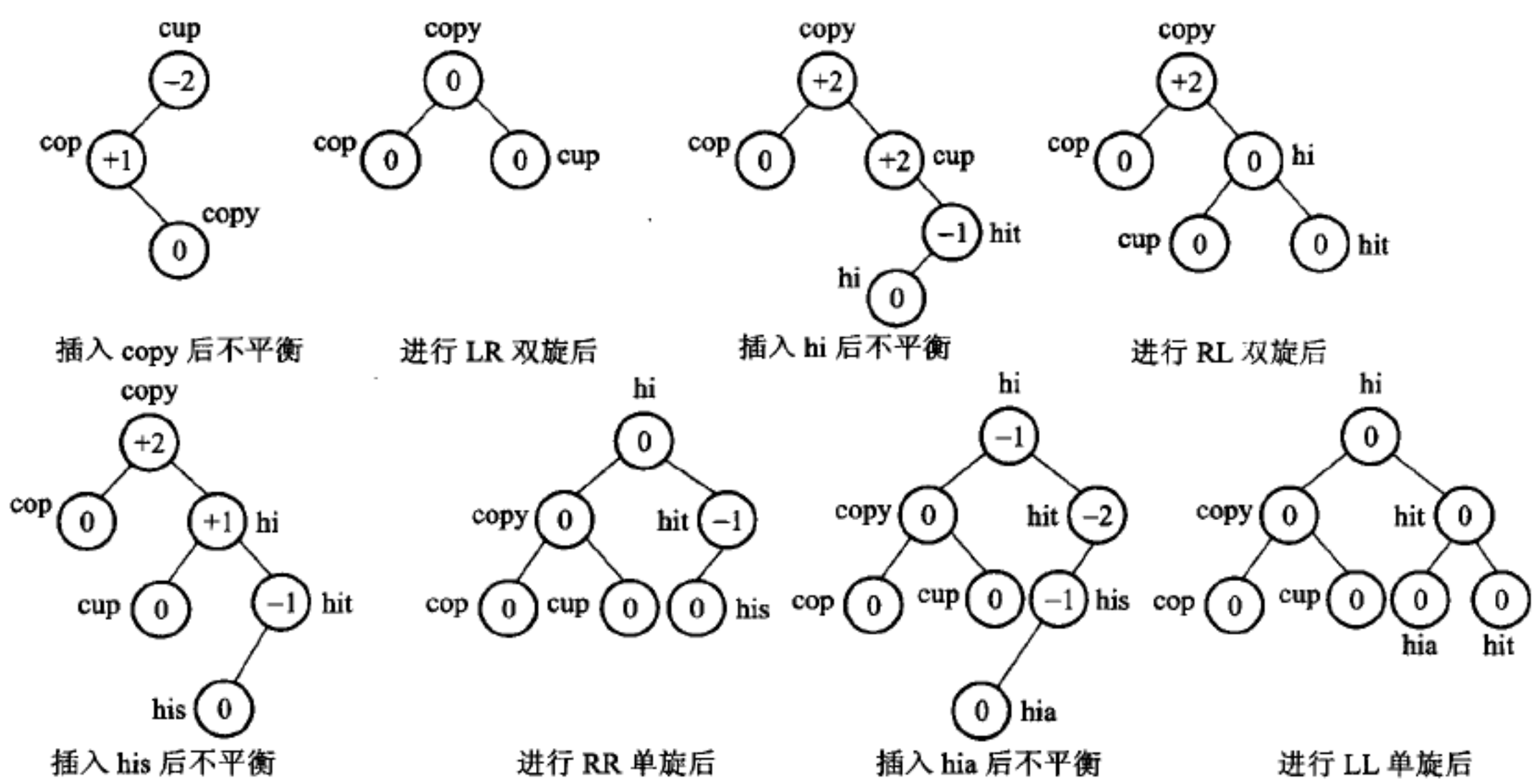


图 12.34 AVL 树的插入过程

2. AVL 树结点的删除

从删除结点的意义上来说，AVL 树的删除操作与 BST 一样。从调整平衡因子的角度来讲，删除是插入的逆操作。

AVL 树的删除是比较复杂的过程，下面具体讨论一下删除的过程，如图 12.35 所示（每种情况只列举了一种特例）。

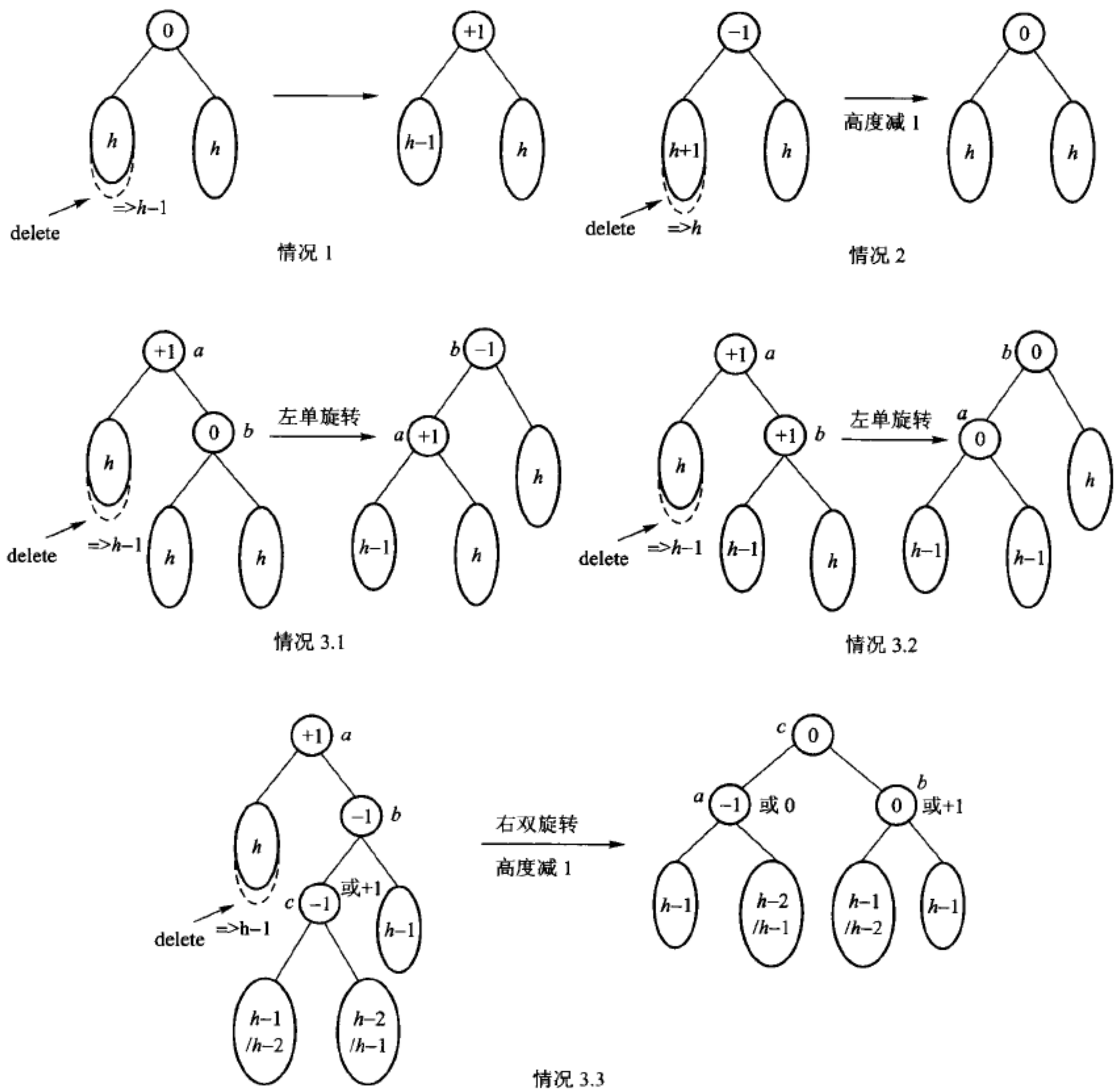


图 12.35 AVL 树删除之后的平衡操作

(1) 首先参考第 5 章中 BST 结点删除方法,删除结点,替代结点 a 为当前结点。

(2) 删除结点后,会导致子树的高度以及平衡因子的变化,需要沿着被删除结点到根结点的路径来调整这种变化。如果发现需要改动平衡因子,则修改之;如果发现不需要修改,则不必继续向上回溯,用一个布尔变量 modified 来记录,其初值为 TRUE,当 modified = FALSE 时,回溯停止。

其中有以下几种情况:

- 第一种情况是当前结点 a 的平衡因子 bf(a) 为 0,如果其左子树或者右子树被缩短,则其平衡因子改为 1 或者 -1,同时 modified = FALSE。因为以 a 为根的子树高度未变,这样的变化不会影响到上面的结点,调整可以结束。
- 第二种情况是当 bf(a) 不为 0,但是其较高的子树被缩短时,则其平衡因子修改为 0,同时 modified = TRUE,需要继续向上修改。
- 第三种情况是 bf(a) 不为 0,且其较低的子树被缩短时,结点 a 必然不再平衡。假设其较高子树的根结点为 b,会出现下面 3 种情况:

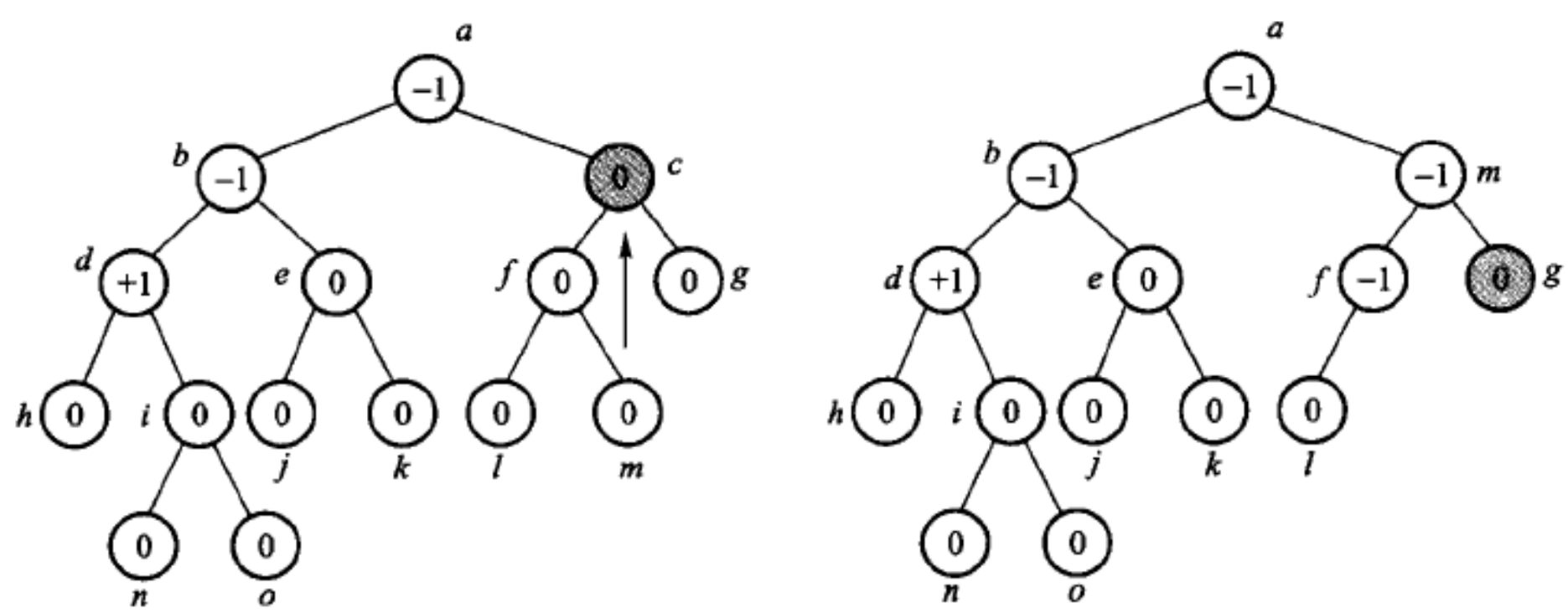
情况 3.1:如果 b 的平衡因子 bf(b) = 0,执行一个单旋转来恢复结点 a 的平衡,并设置 modified = FALSE。

情况 3.2:如果 bf(b) = bf(a),则执行一个单旋转来恢复平衡,结点 a、b 平衡因子都变为 0,modified = TRUE。

情况 3.3:如果 b 和 a 的平衡因子相反,则执行一个双旋转来恢复平衡,先围绕 b 旋转,再围绕 a 旋转。新的根结点平衡因子为 0,其他结点应做相应的处理,并且 modified = TRUE。

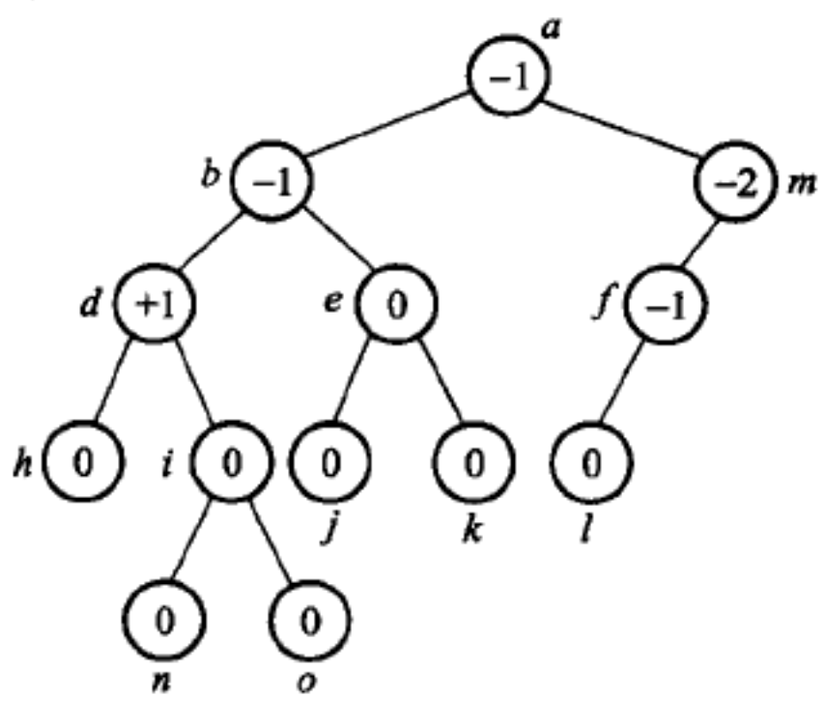
与插入的操作类似,从被删除的结点向上查找到其祖父结点,然后开始单旋转或者双旋转重构操作。插入操作最多只需要旋转一次就调整到位;而删除时一次旋转结束并不意味着树已经平衡,因为这可能会导致其祖先结点发生新的不平衡。因此,这样的重构调整要一直进行下去,直到根结点为止。

图 12.36 所示为对 AVL 树进行删除操作的图示。

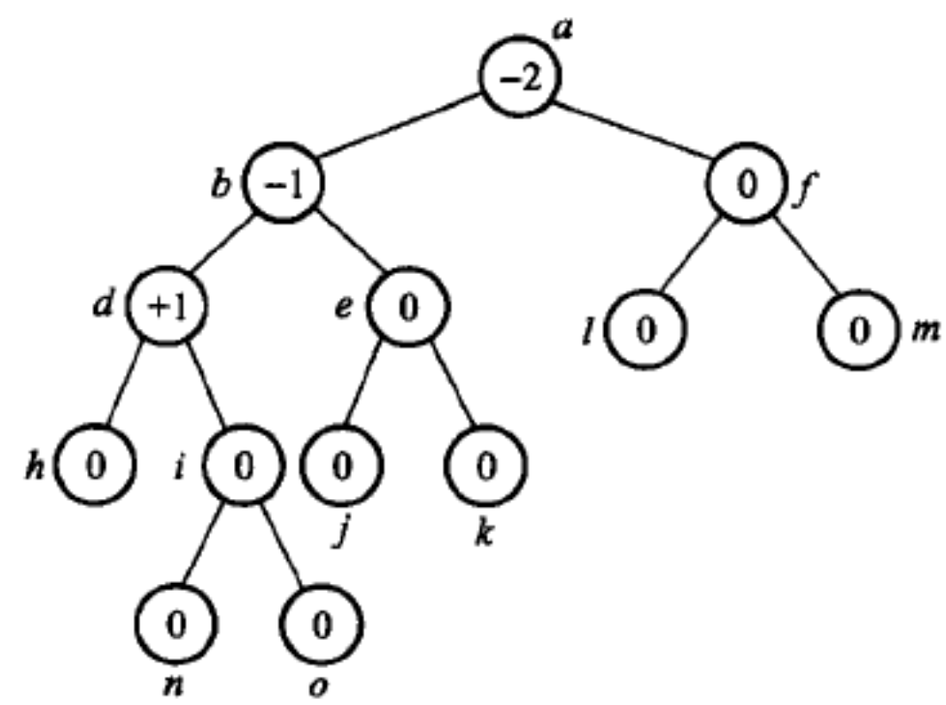


(a) 删除结点c,则需要使用其中序前驱m

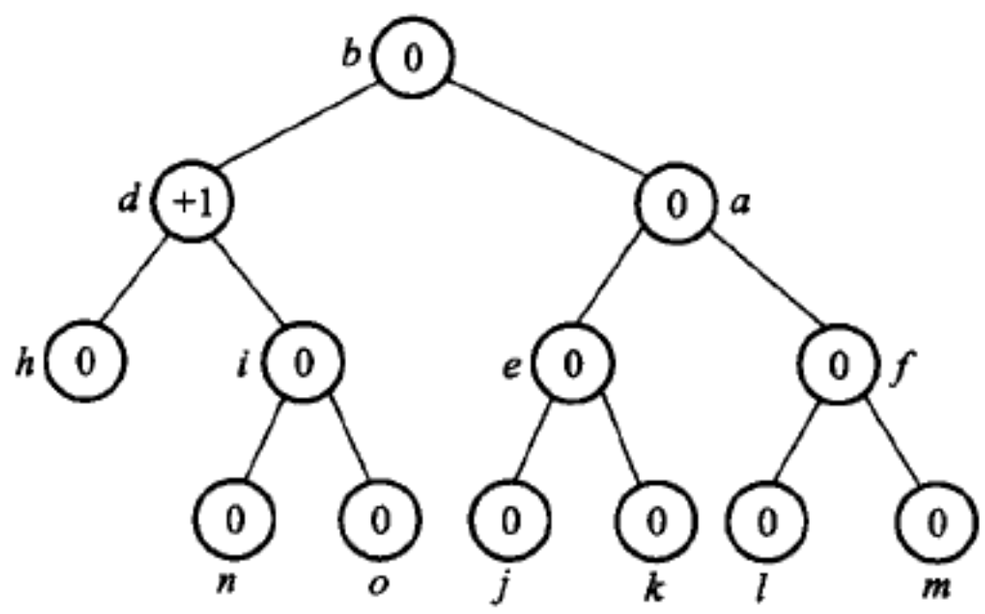
(b) 删除结点g



(c) 需要对m进行LL旋转



(d) LL旋转完毕，回溯调整m父结点a,需要以a为根进行LL旋转



(e) 调整完毕，AVL树重新平衡

图 12.36 AVL 树的删除过程

3. AVL 树的效率

AVL 树的检索、插入和删除效率都是 $O(\log_2 n)$，这是因为具有 n 个结点的 AVL 树的高度一定是 $O(\log n)$。要说明这一点，只需说明有 n 个结点的 AVL 树的最大高度不超过 $K\log_2 n$ 即可，这里 K 是一个小的常数。

为了证明，在此构造一系列临界 AVL 树 $T_1, T_2, T_3, \cdots$。其中，T_i 的高度是 i，使每棵具有高度 i 的其他 AVL 树都比 T_i 的结点个数多。图 12.37 表示了 T_1、T_2、T_3 和 T_4。为了构造 T_i，先分别构造 T_{i-1}和 T_{i-2}，T_i 即是以 T_{i-1}和 T_{i-2}为其左、右子树的 AVL 树（因为既要保证它们不破坏整棵树的 AVL 结构，又要要求它们包含最少的结点，所以必然高度相差 1，而且它们也都是 AVL 树）。对于每一个 i，T_i 都具有 AVL 性质，而且只要从 T_i 中删去一个结点，就不再是 AVL 树或高度不再是 i，因此所构造的是在结点数一定时最接近于不平衡的 AVL 树，这种树在结点一样多的 AVL 树中具有最大高度，可用此树来计算 AVL 树高度的上限值。

先计算上面所说的 AVL 树的结点个数，然后建立高度与结点个数的关系。令 $t(i)$ 为 T_i 的结点数，从图 12.37 可看出有下列关系成立：

$$t(1) = 2$$

$$t(2) = 4$$

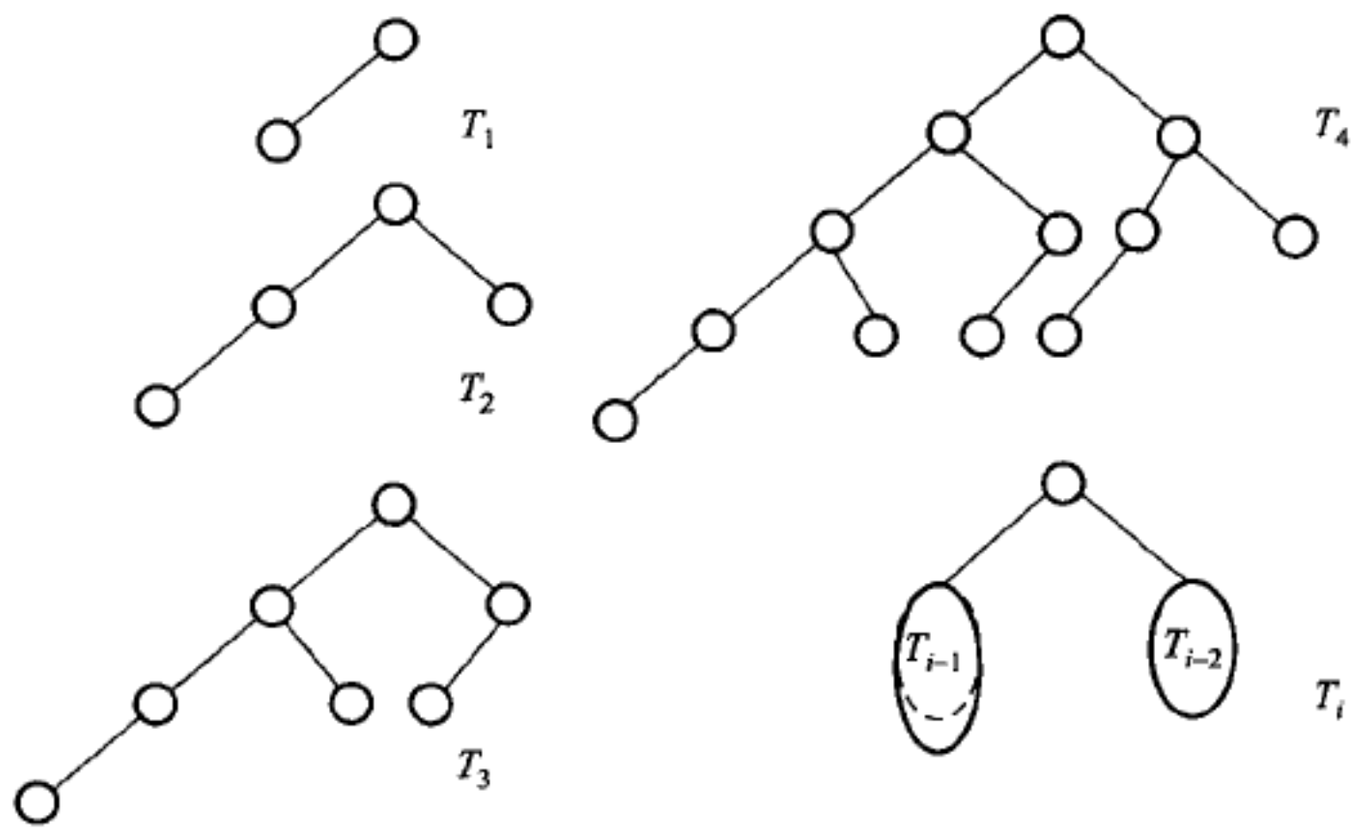


图 12.37 最接近于不平衡的 AVL 树

$$t(i) = t(i-1) + t(i-2) + 1 \tag{12.6}$$

对于 $i>2$,此关系很类似于定义 Fibonacci 数的那些关系:

$$F(0) = 0$$

$$F(1) = 1$$

$$F(i) = F(i-1) + F(i-2)$$

对于 $i>1$,仅检查序列的前几项就可有

$$t(i) = F(i+3) - 1 \tag{12.7}$$

Fibonacci 数满足渐近公式

$$F(i) = \phi^i/\sqrt{5}, \text{这里 } \phi = (1+\sqrt{5})/2$$

由此,可得近似公式

$$t(i) \approx \frac{1}{\sqrt{5}}\phi^{i+3} - 1$$

解出高度 i 与结点个数 $t(i)$ 的关系

$$\phi^{i+3} \approx \sqrt{5}(t(i)+1)$$

$$i+3 \approx \log_\phi \sqrt{5} + \log_\phi(t(i)+1) \tag{12.8}$$

由换底公式 $\log_\phi X = \log_2 X/\log_2 \phi$ 和 $\log_2 \phi \approx 0.694$ 求出近似上限

$$i < \frac{3}{2}\log_2(t(i)+1) - 1 \tag{12.9}$$

因为 $t(i)$ 是结点个数,即对于一个有 n 个结点的 AVL 树,其高度不超过 $1.5\log_2(n+1)$,这就是所要证明的结果。

由于 AVL 树的检索、插入和删除操作都是从根到叶结点单路径进行的局部运算,因此 AVL

树的检索、插入和删除效率都是 $O(\log_2 n)$。Linux 采用 AVL 树描述进程的虚拟内存段。

二叉搜索树(包括红黑树、AVL 树、伸展树等)适用于组织内存中小规模的目录。对于较大的、存放在外存储器上的文件,用二叉搜索树来组织索引就不太合适。若以结点作为内外存交换的单位,则检索中在找到需要的关键码之前平均要对外存进行 $\log_2 n$ 次访问,这是很费时的。

在文件索引中大量使用每个结点包含多个关键码的 B 树,尤其是 B^+ 树。

12.4.3 伸展树

还有一种与 AVL 树类似的改进 BST,叫做伸展树(splaying tree),由 John Edward Hopcroft 和 Robert Endre Tarjan 于 1985 年共同发明,与第 10.1 节所提到的线性表调整技术,同属于自调整数据结构(self-adjusting data structure)。

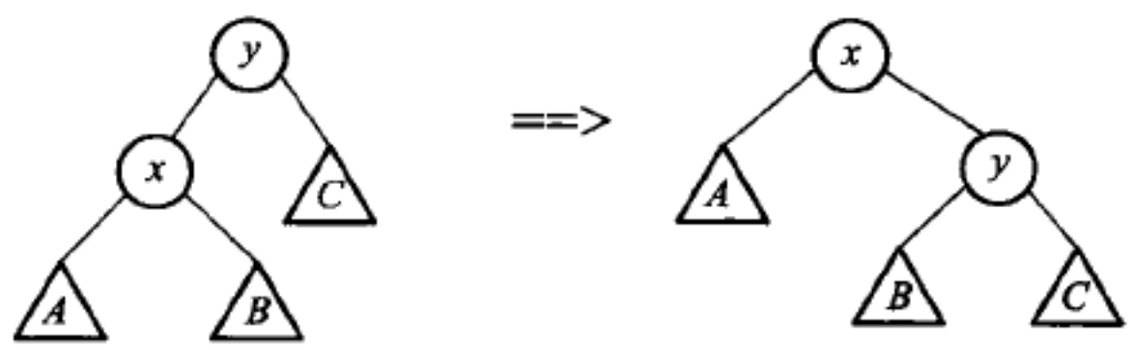
与 AVL 不同的是,伸展树的一次访问不一定会使树结构更加平衡,也不能保证最终树高平衡。但它能保证访问的总代价不高,达到最令人满意的性能。

伸展树与 AVL 树一样,实际上也不是一种新数据结构,而只是改进 BST 性能的一组规则,控制检索、插入、删除等操作对 BST 的修改,从而避免标准 BST 操作在最差情况下的线性时间代价。

每当访问一个结点 x 时(例如,当 x 被插入、删除或者被检索时),伸展树就完成一次称为展开(splaying)的过程。插入和检索 x 时,结点 x 作为当前结点;删除结点 x 时,x 的父结点作为当前结点。沿着当前结点到根结点的方向,进行一系列展开操作,直到把当前结点移到 BST 的根。就像在 AVL 树中一样,结点 x 的一次展开由一组旋转(rotation)组成。一次旋转通过调整结点 x 相对于其父结点和祖父结点的位置,将其移到树结构中的更高层。旋转有 3 种类型:单旋转、一字型双旋转、之字型双旋转。

当被访问结点 x 是根结点的子结点时,完成单旋转(single rotation),在保持 BST 特性的情况下把结点 x 与其父结点交换位置,如图 12.38 所示。实际上伸展树的单旋转与 AVL 树的单旋转是一样的。

图 12.38(b)访问 x 和图 12.38 (a)访问 y 是互为单旋转的结果,只是其访问的结点不同。



(a) 开始树结构以结点 y 作为父结点　　(b) 旋转发生以后的树

图 12.38　当被展开的结点是根结点的子结点时,才会发生伸展树的单旋转

当被访问结点 x 离根结点的层数大于等于 2 时,发生双旋转(double rotation),涉及结点 x、结点 x 的父结点(称为 y)和结点 x 的祖父结点(称为 z)。双旋转的结果是把结点 x 在树结构中

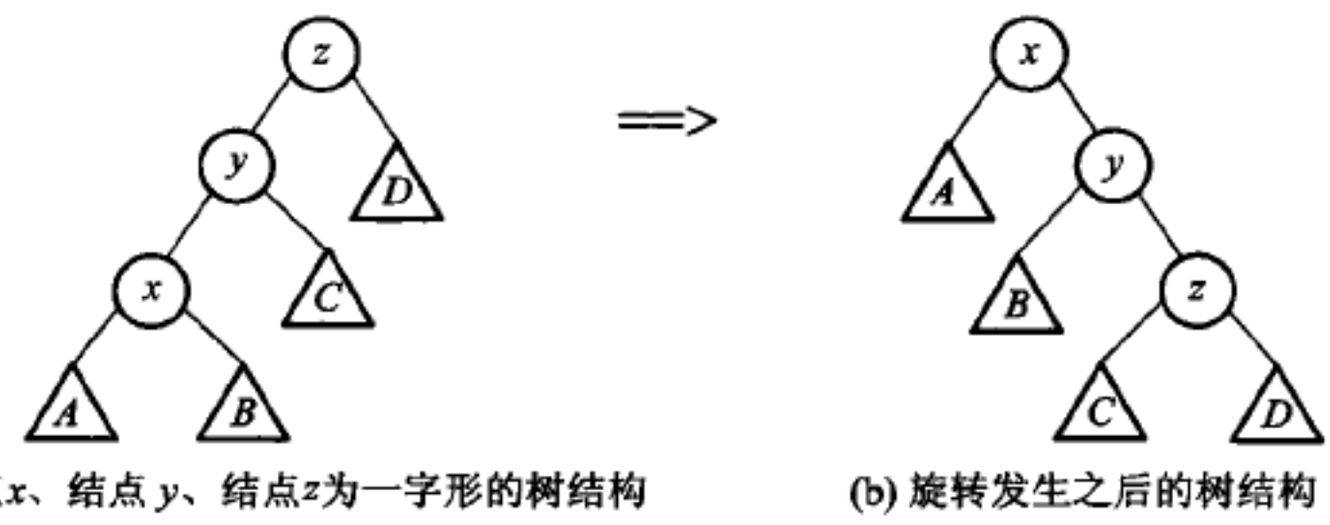
向上移两层。与 AVL 树不同，伸展树需要两种类型的双旋转。

第一种双旋转称为一字形旋转（zigzig rotation），也称为同构调整（homogeneous configuration）。当出现以下两种情况之一时，就会发生一字形旋转：

（1）结点 x 是结点 y 的左子结点，结点 y 是结点 z 的左子结点。

（2）结点 x 是结点 y 的右子结点，结点 y 是结点 z 的右子结点。

如图 12.39 所示，被访问的结点是 x，结点 x、结点 y、结点 z 为一字形的树结构。旋转发生之后，子树 A、B、C 和 D 的位置进行了相应地改变，以保持 BST 的特性。



(a) 结点x、结点 y、结点z为一字形的树结构　　(b) 旋转发生之后的树结构

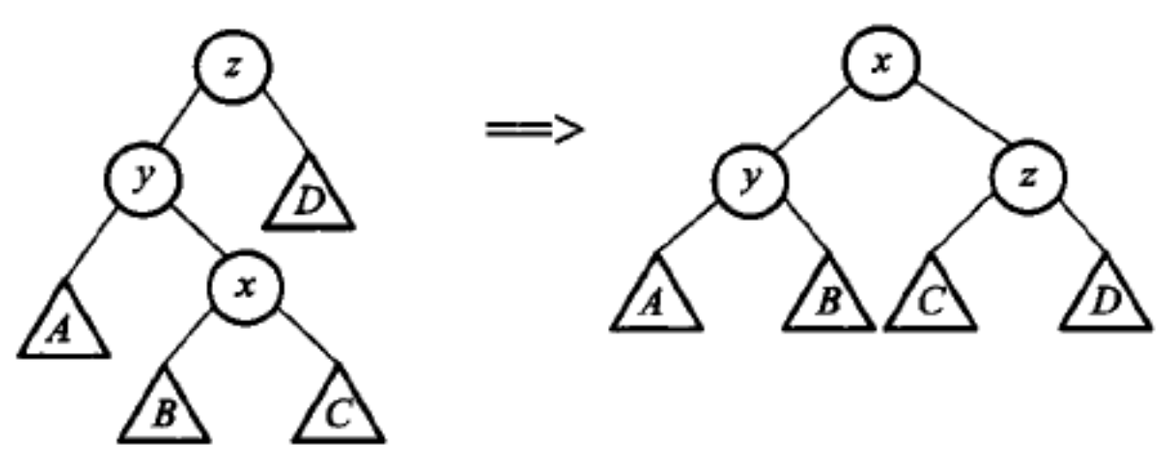
图 12.39　伸展树的一字形旋转

第二种双旋转称为之字形旋转（zigzag rotation），也称为异构调整（heterogeneous configuration）。当出现以下两种情况之一时，就会发生之字形旋转：

（1）结点 x 是结点 y 的左子结点，结点 y 是结点 z 的右子结点。

（2）结点 x 是结点 y 的右子结点，结点 y 是结点 z 的左子结点。

也就是说，当结点 z、结点 y、结点 x 形成一个之字形时就进行之字形旋转，如图 12.40 所示。旋转发生之后，子树 A、B、C 和 D 的位置进行了相应地改变，以保持 BST 的特性。实际上，之字形旋转与 AVL 树的双旋转是一样的。



(a) 结点x、结点y、结点z为之字形的树结构　(b) 旋转发生之后的树结构

图 12.40　伸展树的之字形旋转

之字形旋转趋向于使树结构更加平衡。例如，图 12.40 的旋转使子树 B 和 C 上升一层，而使子树 D 下降一层。一字形旋转一般不会降低树结构的高度，而只是把新访问的记录向根结点提升。

被访问结点 x 的伸展过程包括一系列双旋转，直到结点 x 到达根结点或者根结点的子结点。

如果需要，再进行一次单旋转使结点 x 成为根结点。旋转的结果将使访问最频繁的结点靠近树结构的根层，从而减少访问代价。

图 12.41 说明了伸展树的调整过程，注意该伸展树的内部结点只是作为索引的代号（可以把这些小写字母表示的内部结点当作变量，在此并不需要知道它们的数值，不要误认为内部结点值不符合 BST 中序访问的有序性），外部叶结点保持 BST 的中序访问性质。

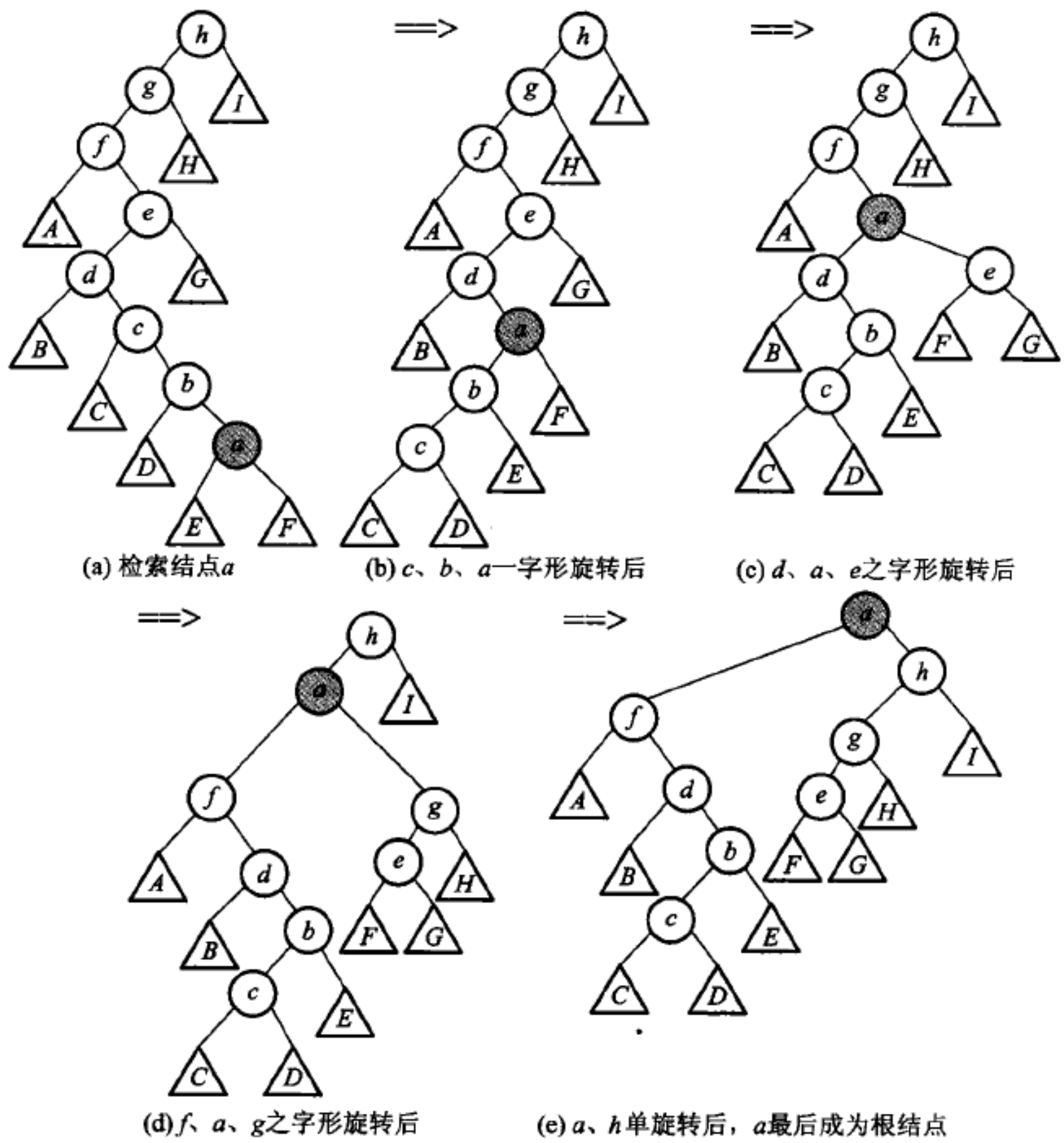


图 12.41 在伸展树中完成一次检索以后再展开的例子

在图 12.41 中检索 a，伸展树检索过程与 BST 完全一样。但是，一旦找到了这个值，就需要通过伸展过程把它移到根结点。在图 12.41 中需要 4 次旋转。第一次是 c、b、a 一字形旋转，其结果如图 12.41(b)所示。第二次是 d、a、e 之字型旋转，其结果如图 12.41(c)所示。第三次是 f、a、g 之字形旋转，其结果如图 12.41(d)所示。最后一步是一个单旋转，其结果如图 12.41(e)所示。注意，展开过程使得树结构的层次更浅了。

伸展树的插入操作与 BST 相同，但是结点插入后还要被展开到根结点。同样，采用 BST 删除结点策略从伸展树中删除一个结点后，被删除结点的父结点需要展开到根结点。

半伸展（semi - splaying）树是伸展树的变体，它对于同构的“一字形”（zig - zig 型）的情况，类似于 AVL 树的 LL、RR 型那样处理。图 12.42 比较了半伸展和全伸展在同构的“一字形”（zig - zig 型）情况处理的区别。下一轮的伸展，当前结点上移到父结点，父结点位置开始旋转，即半伸展不需要把检索结点移到根。图 12.43 是连续两次访问结点 T，半伸展树的变化示例。

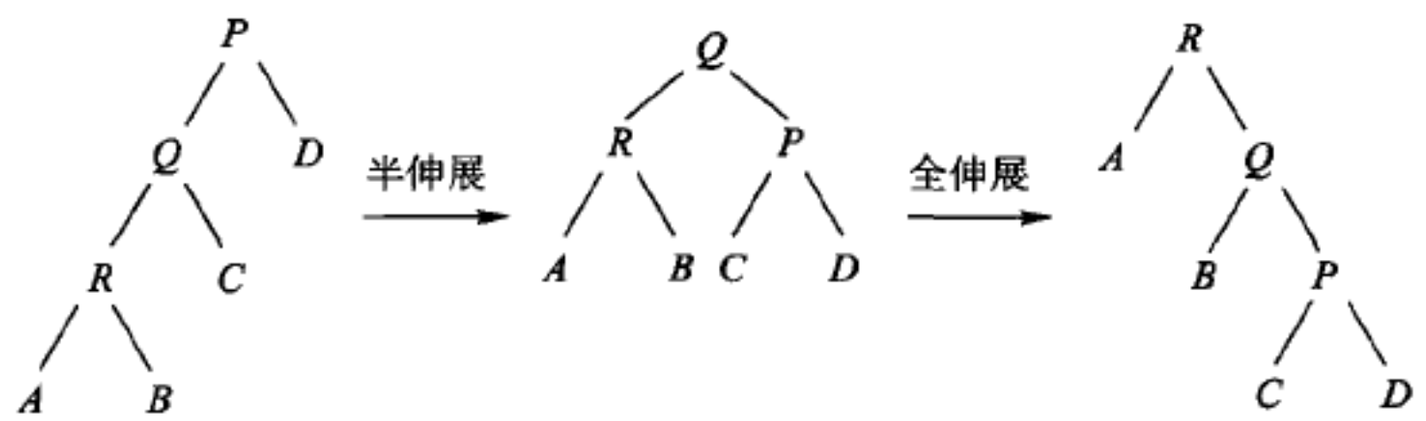


图 12.42 半伸展和全伸展的区别

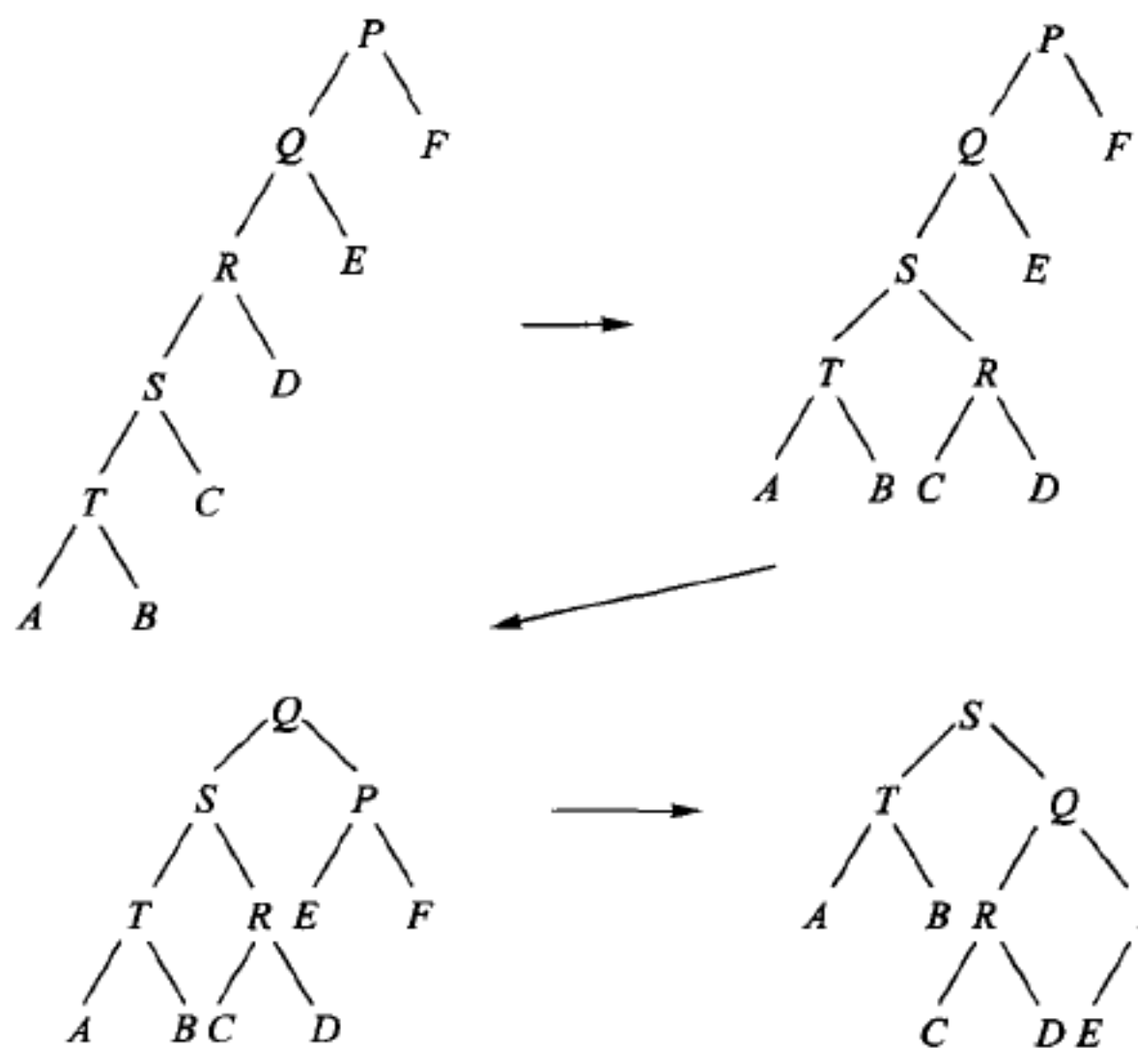


图 12.43 连续两次访问结点 T，半伸展树的变化示例

如果记录的信息都存储外部结点，内部结点只作为索引结构。检索到某个外部结点时，其内部父结点作为当前结点，然后开始调整。伸展的过程中，要注意外部结点不能被旋转到根上。例如，在如图 12.44 所示的半伸展树中，外部结点 $a,b,\cdots,h$ 存储记录信息，连续检索 aabg 后，形成半伸展树的详细过程如图 12.45 所示。

在图 12.44 中读 a 后，当前结点为 A，结点 A、B、D 进行“一字形”旋转，变成图 12.45(a)。继续读 a，当前结点为 A，结点 A、B 进行半旋转，得到图 12.45(b)。继续读 b，当前结点为 B，结点 B、A 半旋转，回到图 12.45(a)。继续读 g，当前结点为 G，结点 G、F、D “一字形” 旋转，F 变为根

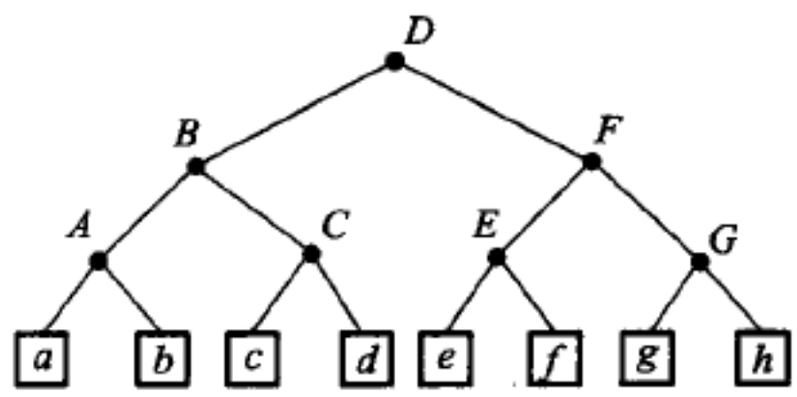
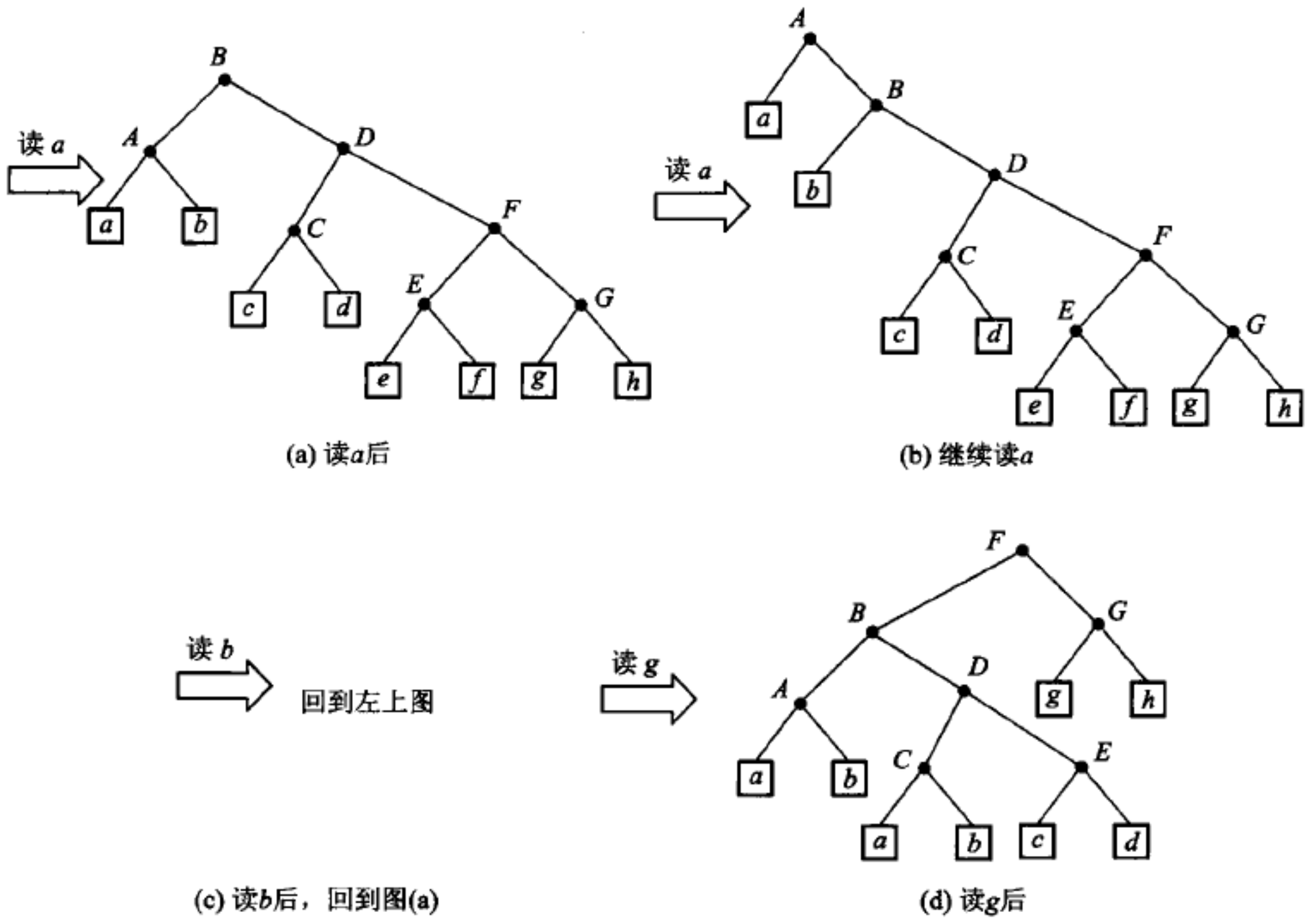


图 12.44 外部结点存储记录信息的半伸展树



(a) 读a后

(b) 继续读a

(c) 读b后，回到图(a)

(d) 读g后

图 12.45 对图 12.44 连续读 *aabg* 后，半伸展树的变化过程

B 的右结点；当前结点转为 *F*，结点 *F*、*B* 进行半旋转，得到图 12.45(d)。

伸展树与 AVL 树在操作上只有略微的差别。伸展树与结点被访问(包括插入、查找等)的频率相关，能够进行更加动态的调整；而 AVL 树的结构只与插入、删除的顺序有关，与插入、删除或者检索的频率无关。

有分析表明，对于一个具有 n 个结点的伸展树，进行一组 m 次操作（插入、删除、查找操作），当 $m \geq n$ 时，总代价为 $O(m \log n)$。

伸展树不能保证每一个单个操作是有效率的，即某次插入或检索操作可能花费 $O(n)$ 时间。但是，能够保证 m 次操作总共需要 $O(m \log n)$ 时间，即每次访问操作的平均代价为 $O(\log n)$，这是检索树结构的必要性能保证。

FreeBSD UNIX 在虚拟内存系统采用了改进的伸展树，用于存储虚拟地址空间中一段区域的起始和终止地址等其他相关信息，以更快地满足内存区域匹配的要求。

本章小结

为了适应一些特定应用，需要功能更强大、具有更多操作的高级数据结构。本章介绍了多维数组、广义表等高级线性表结构以及相关的内存管理技术，还介绍了 Trie 和 PATRICIA 两种字符索引结构以及最佳二叉搜索树、平衡二叉搜索树 AVL、伸展树 3 种改进的 BST。

多维数组可以看做是线性表的推广。数组一般采取顺序存储方法，因为数组结构中元素个数相对固定，一旦生成，只改变元素的值，并不改变元素的相对位置和个数。把数组元素按周游的顺序排列起来，得到一个包含所有数组元素的线性序列。数组的周游分为行优先顺序和列优先顺序两种方式。

二维数组常被用来表示矩阵，数组的第一维和第二维矩阵分别对应矩阵的行和列。三角矩阵、对称矩阵和对角矩阵是常用的几类比较特殊的矩阵。稀疏矩阵是非零元素非常少的矩阵，它常用的存储方式是十字链表法。

广义表示是对线性表的一种扩展，广义表则允许线性表中还包含一个或多个子表。用数学中结合的包含关系可以表述上述几种类型的广义表：

图⊇再入表⊇循环表⊇纯表⊇线性表

根据广义表的不同形式，可以用线性表、树和图等数据结构或其修改形式来存储和实现广义表。广义表的周游通常采用深度优先的策略。

为支持动态数据结构，操作系统需要一套有效的内存管理技术，动态地进行内存的分配和回收。在实际的操作系统中，内存管理本质上是利用链表和广义表来实现的。内存管理最基本的问题是存储的分配和回收，需要解决碎片问题以及存储空间溢出时无用单元的回收管理等。内存分配是通过可利用空间表的组织来进行的，这是把存储区中当前还未使用的空间通过链表方式管理起来的一种方法。

BST 树是一种基于对象空间分解的数据结构，即关键码范围的分解是由存储在树中的对象决定的，这种分解方式受关键码输入的影响。Trie 结构对关键码范围进行均分，它基于两个原则：有一个固定的关键码集合和对于结点的分层标记。其内部结点仅作为占位符引导检索过程，数据记录只存储在叶结点中。英文字符 Trie 树的每个内部结点都有 26 个子结点（如果算上空格，则为 27 个）。PATRICIA 树是 Trie 树的变体，它根据关键码的二进制编码来进行划分，是一种比较好的中文字典组织方式。

最佳二叉搜索树、平衡二叉搜索树 AVL、伸展树是二叉搜索树 BST 的几个变体。

检索中平均比较次数最小的二叉搜索树称为最佳二叉搜索树。最佳二叉搜索树满足最优子问题和子问题重叠性质，可以使用动态规划算法来构造。

AVL 树中任何结点的左子树和右子树高度最多相差 1。插入新结点按照 BST 的插入算法，如果插入后仍然是 AVL 树，则不调整；否则，需要根据 LL、LR、RL、RR 四种类型进行调整。AVL 树的删除结点采用 BST 的基本删除算法。如果删除后需要调整，则从被删除的结点找到第一个失衡的祖先结点，然后开始单旋转或多旋转操作。这样的调整操作向根的方向传递，直到树平衡为止。AVL 树放宽了 BST 的平衡要求，它既照顾到更新操作的代价，又兼顾完成操作后树结构的检索功能，其树高为 $O(\log n)$。

伸展树不要求 BST 结构总是保持平衡或最佳检索状态，而是在每次访问时通过一系列双旋转、单旋转将被访问结点移到根结点，这样尽量使最近访问的结点容易被检索到，其检索效率比较高。

第 11 章介绍的红黑树也是一种改进的 BST，它是一种常用的索引结构，统计性能要好于 AVL 树。其平衡要求比 AVL 树更低，利用对树中结点红黑着色的要求达到局部平衡。

二叉搜索树（包括红黑树、AVL 树、伸展树等）适用于组织内存中小规模的目录。AVL 树的高度不超过 $1.5\log_2(n+1)$，具有 n 个内部结点的红黑树最大树高是 $2\log_2(n+1)+1$，因此最坏情况下的红黑树的高度大于最坏情况下具有相同（内部）结点数目的 AVL 树。AVL 树和伸展树适合于检索较多的应用，插入删除操作比较频繁的应用往往采用红黑树。

习　题

1. 设计一个算法，将按行优先原则顺序存储的二维数组 $A_{n,n}$ 转置（$a_{i,j}$ 与 $a_{j,i}$ 交换），要求转置结果仍占用原来的存储空间。

2. 编写程序，建立一个 10×10 的对角线元素为 0，其余元素为 1 的二维数组，并将该数组打印出来。成功后改写该程序，使其对角线元素仍然为 0，其余元素为空格字符的二维数组。

3. 若在 $m\times n$ 的矩阵中存在一个元素 $A[i,j]$ 满足：$A[i,j]$ 是第 i 行元素中最小值，且是第 j 列元素中最大值，则称矩阵 $\boldsymbol{A}$ 有鞍点。试写一个算法，找出矩阵 $\boldsymbol{A}$ 的一个鞍点，若 $\boldsymbol{A}$ 中不存在鞍点，则给出某种信息（采用行优先原则顺序存储矩阵 $\boldsymbol{A}_{mn}$）。

4. 编写一个广义表逆置算法，在所有级别逆置子表，包括最高级。

（1）例如，纯表 $(x_1,(y_1,y_2,(z_1,z_2),y_4),(w_1,w_2),x_4)$ 的逆置为 $(x_4,(w_2,w_1),(y_4,(z_2,z_1),y_2,y_1),x_1)$。

（2）循环表 $(L1:(L2:(L1,a)),(L2,L3:(b)),(L3,c),L4:(d,L4))$ 的逆置对应于 $(L4:(L4,d),(c,L3:(b)),(L3,L2:(a,L1)),L1:(L2))$。

5. 画出下列广义表的图形表示和它们的存储表示：

（1）$D:(A:(c),B:(e),C:(a,L(b,A,d)))$

（2）$J1:(J2:(J1,a,J3:(J1)),J3)$

6. 利用广义表的 Head 和 Tail 运算，把原子 d 分别从下列广义表中分离出来，$L1=((((((a),b),d),e))$；$L2=(a,(b,((d)),e))$。

7. 设计一个算法，求广义表的深度。

8. 设计一个算法，删除广义表中所有值等于 x 的原子项。

9. 设计一个算法,判断两个广义表是否相等。

10. 3 种顺序适配方法各有其使用范围。现假定存储区中有大小为 900、1 500、700 的三块空存储块。试给出符合以下几种情况的存储请求的例子:

(1) 首先适配法可满足请求,另两种方法不可满足请求。

(2) 最佳适配法可满足请求,另两种方法不可满足请求。

(3) 最差适配法可满足请求,另两种方法不可满足请求。

11. 组织成循环链表的可利用空间表附加什么条件时,首次适配策略转变为最佳适配策略吗?

12. 给出下面一组值{cup,cop,copy,hit,hi,his,hia,he}的二叉 Trie 结构。

13. 给出下面一组值{cup,cop,copy,hit,hi,his,hia,he}的 PAT Trie 结构。

14. 将关键码 1,2,3,…,(2^k-1)依次插入到一棵初始为空的 AVL 树中,试证明结果是一棵高度为 k 的完全满二叉树。

15. 已知元素个数为 12 的字典,其元素集合为:

{Jan, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec}

(1) 试按元素的次序依次插入一棵初始时为空的二叉搜索树,画出插入完成之后的二叉搜索树,并求其在等概率情况下查找成功的平均查找长度。

(2) 按元素顺序构造一棵 AVL 树,并求其在等概率情况下查找成功的平均查找长度。

16. AVL 树是否允许重复关键码,例如图 12.46 中有两个关键码都是 12 的结点,为什么?

17. 高度为 h 的 AVL 树上的最少结点个数与最多结点个数分别是多少?

18. 从空的二叉树开始,根据字典顺序(注意:"he" < "tea" < "teacher"),严格按照 AVL 树(平衡的二叉搜索树)插入算法,依次插入{head、he、tea、teach、twin、hot、toss}。

(1) 试画出插入后的 AVL 树。

(2) 在上题生成的 AVL 树中,依次删除 head、he、tea、teach、twin、hot、toss。试画出每执行一次删除操作,并根据必要进行调整后的 AVL 树。

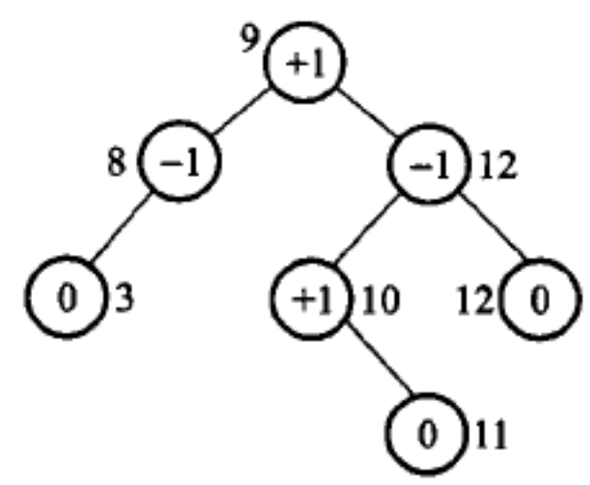


图 12.46 习题 17 的 AVL 树

上 机 题

1. 设二维数组 $a[1..m, 1..n]$含有 $m \times n$ 个整数。写出一个算法来判断 a 中所有元素是否互不相同,输出相关信息(yes/no),并分析你的算法的时间复杂度。

2. 广义表具有可共享性,因此在周游一个广义表时必须为每一个结点增加一个标志域 mark,以记录该结点是否访问过。一旦某一个共享的子表结点被做了访问标志,以后就不再访问它。

(1) 试定义该广义表的类结构。

(2) 实现对一个非递归的广义表进行周游的算法。

3. 试设计一个数据结构和算法,判断两个.cpp 程序(程序都是完整的,即类型定义和函数定义都在一个文件中)是否相似。可以返回 0(完全不同)至 1(完全相同)之间的值。

4. 实现稀疏矩阵表示方法,并支持矩阵的下列操作:

(1) 在给定位置向矩阵中插入一个元素。

(2) 从矩阵中给定位置删除一个元素。

(3) 返回矩阵中给定位置元素的值。

(4) 对一个矩阵进行转置。

(5) 两个矩阵相加。

5. 基于图 12.11 的结构,模拟存储管理系统,支持 3 种顺序适配方法的分配和回收操作:首先适配、最佳适配和最差适配。测试你的系统,分析总结在什么情况下哪种方法完成得最好。

6. 用 Trie 树实现一个字典,要求具有 create、insert、delete、search 功能。

7. 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。

8. 编写一个程序,对一段英文材料,用 Trie 树进行存储。要求:提供算法周游 Trie 树,并打印出所有单词,单词按照长度的升序排序,长度相同的单词按字母顺序。

9. 对于一段英文材料,构造 PATRICIA 树,并统计每个字母出现的次数。

10. 后缀是指从某个位置 i 开始到整个串末尾结束的一个特殊子串。一个长度为 n 的字符串 S,约定 S[n] = '$',也就是说 S 以一个特殊字符 '$'结尾。一个后缀树(suffix tree)是对于一个给定的字符串,包含其所有后缀字符串的压缩 Trie 结构。

例如,字符串"MALAYALAM $"的后缀字符串集合中有 10 个子串,分别是:

子串序号: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

{MALAYALAM $,ALAYALAM $,LAYALAM $,AYALAM $,YALAM $,ALAM $,LAM $,AM $,M $,$}

图 12.47 就是字符串"MALAYALAM $"的后缀树。

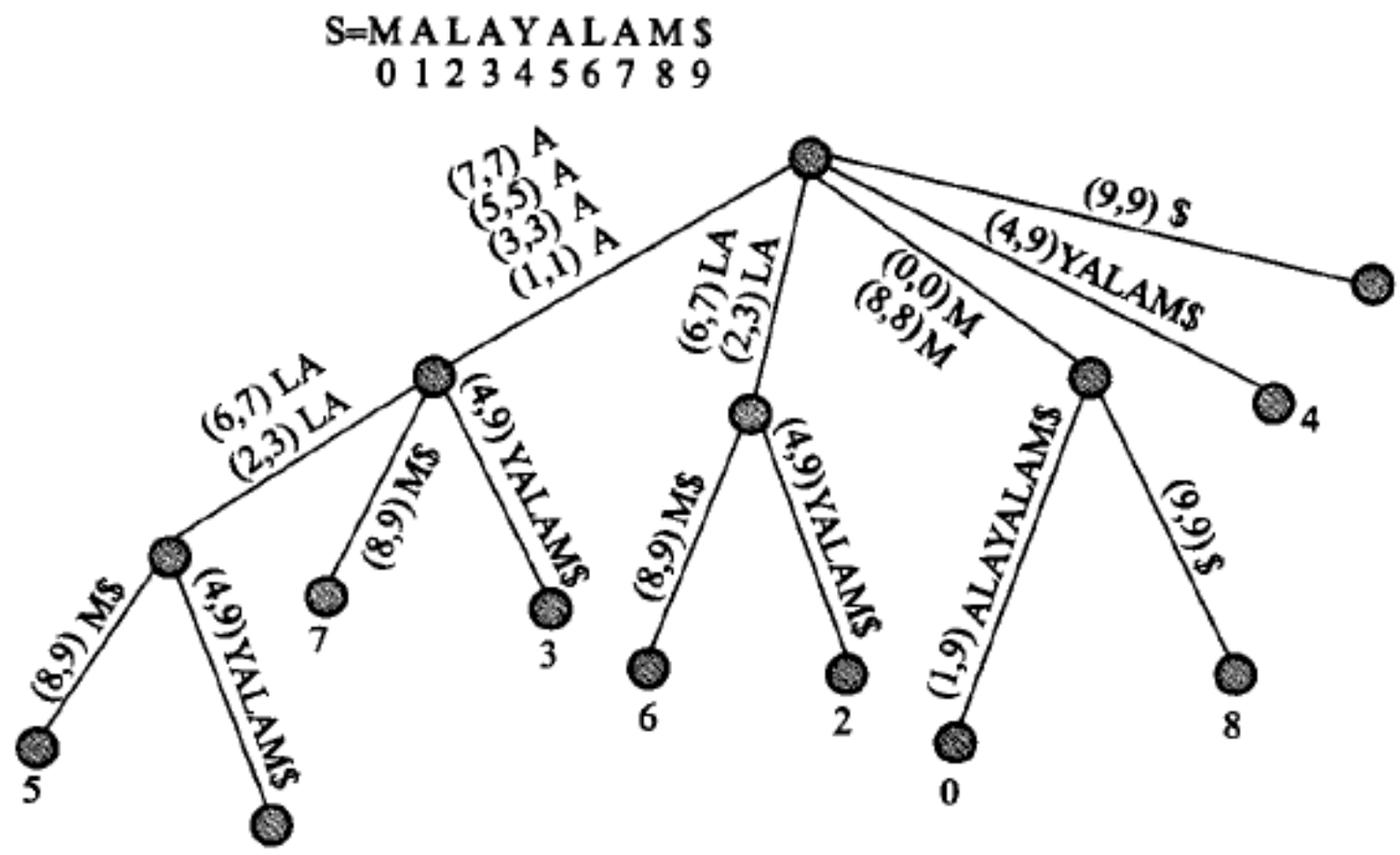


图 12.47 带路径压缩的后缀树

后缀树的好处是它允许使用通配符(wildcard)对字符串进行检索。例如,检索关键码"ALA * "表示找到所有以"ALA"作为前 3 个字符的字符串,这可以很容易地使用一个普通 Trie 结构实现。在一个普通 Trie 结构中检索" * ALA"效率很低,但是在一棵后缀树中却是高效率的。

后缀树提供了很好的全索引结构,适合于各种字符串算法,例如查找字符串中的子串、找出串中最长的重复子串、查找两个字符串的公共子串、判断回文串等。后缀树在中文切词、共现词关联分析、频繁模式挖掘、聚类分

析、基因序列对比分类等研究领域也有很广泛的应用。

后缀树有很多变种和发展，其目标是尽量减少空间消耗。为了减少构造后缀字典树所花费的时间与空间，Edward McGreight 于 1976 年非在线的后缀树算法，逆序处理字符串，采用路径压缩的方法构造后缀树。该后缀树最多有 $2n$ 个结点，构造后缀树的时间代价为 $O(n)$。

1995 年，Ukkonen 提出了在线的后缀树算法，在从左到右输入的同时构造出后缀树，时间代价为 $O(n)$。

试根据自己的兴趣和能力完成后缀树的练习。例如：

(1) 对一个词典中的单词或短语实现后缀树。

(2) 分析若干篇文章中的频繁共现词汇。

(3) 利用后缀树，对搜索引擎的返回结果进行聚类。

…

11. 在字符{A, C, G, T}组成的 DNA 序列中，A ⟷ T 和 C ⟷ G 是互补对。

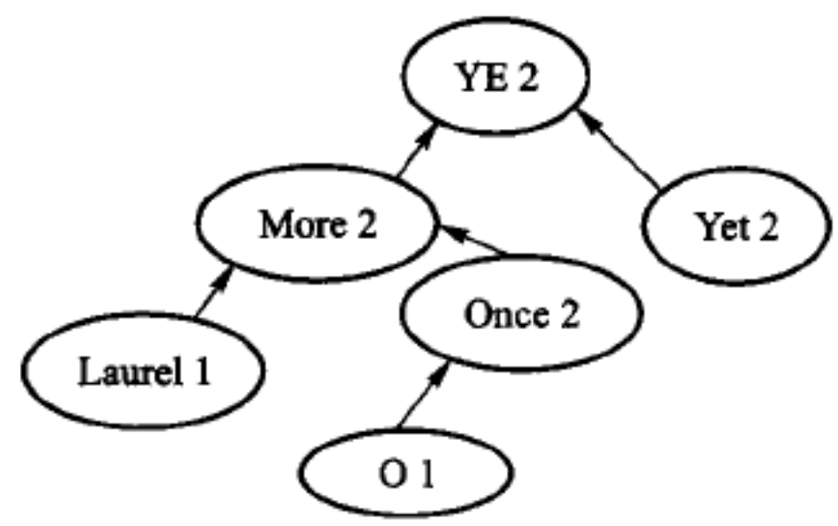
编写算法，判断一个 DNA 序列中所有互补回文串（例如，ATCATGAT 的补串是 TAGTACTA，与原串形成互补回文串）。

12. 利用半伸展树，存储文件中单词词频。

可以利用半伸展树来存储文件中单词词频。图 12.48 显示了一个短文件的结点树。

扫描一个文本文件，并计算出这一文件中每个词的词频。为简化起见，略去标点符号、按照字典排列的顺序对单词排序的，并忽略大小写。例如，man's 将被当成 man 和 s；类似地，分隔符也被忽略，如 pre - existence 被当作 pre 和 existence。

用一棵 BST 作为一个文件中单词的存储结构，并用 semi - splay 技术对这棵树进行辅助维护，以使输入流中的下一个单词出现在树比较接近于根的位置的几率较大。



用于处理生成该树的文本是：yet once more, o ye Laurel，…，显然它也是一棵二分检索树。

图 12.48 存储词频信息的半伸展树

半伸展技术构造一棵自调整词频 BST 的方法如下：

(1) 如果在文件中第一次遇到一个词，把这个单词插入 BST 树。否则，从与该单词对应的结点处开始处理，这一单词的频率增加 1。

(2) 向根结点的移动是通过改变所涉及结点的链接来完成的。如果单词没有在树中找到，将创建一个新的叶结点，然后插入树中。

(3) 通过一个指向父结点的指针来保存结点的前驱。因此，从每一个结点都可以访问该结点的所有前驱，直到树根结点。

注意：旋转与否只取决于是否被访问（包括查找、插入等），跟词本身的词频没有必然关系。但是可以想像，访问频率越高（只要不都是连续访问），它被旋转到根结点的机会就越多，相对而言它会比其他较少访问的结点更接近根结点。

作业要求：创建一个较长的测试文件，使用 semisplaying 技术来组织一棵 BST。在所有的单词都处理完之后，要求对树进行一次中序周游，将所有结点的词频数打印出来。

13. 试设计一个模拟的阅卷管理系统，该系统支持多位教师集体阅卷（试卷为电子文档）。系统基本要求：

(1) 每份试卷应该随机调阅，分别由两位教师批阅，取其平均成绩。

(2) 阅卷教员发出调阅试卷要求，系统随机调阅一份未完成的试卷（还没有人阅过，或者只有一位教师阅

过)给该阅卷者。

(3) 每位教师可以检查自己批阅过的试卷,并录入和修改成绩。考生姓名和考号对教师保密,教师只能看到系统编号。

读者根据自己的能力添加其他功能:

① 支持大数据量,例如 100 万以上。

② 支持并发操作,例如允许多位教员读数据。

③ 两位阅卷教师给出的成绩相差太大(例如 >10%)时,试卷派发给阅卷组长。

④ 自动调整每位教员的成绩权重,不同教师给予不同的权重。例如,有位教师特别爱给高分,则根据系统计算其权重应该为 0.8;有位教师特别爱给低分,则根据系统计算其权重应该为 1.2。

14. 简单的职工管理系统。职工对象包括姓名、部门、开始工作时间、职务、年龄、性别、生日、学历、住址、电话、工资。

要求:

(1) 实现增加一个职工的信息(包括加入批量的数据)。

批量导入的数据格式如下:

"CHRISTINE","A00","1965-01-01","PRESIDENT",18,"女","1933-08-24","undergraduate","Beijing","010-62755440",1000.00

"PHILIP","E11","1972-06-19","OPERATOR",14,"男","1936-10-27","high school","Shanghai","022-12345678",400.00

"黎明","E21","1947-05-05","操作工",16,"男","1926-05-17","高中","北京","010-62761226",500.00

(2) 应该允许某些参数不输入。例如:"黎明","E21","1947-05-05","操作工",16,"男","1926-05-17","高中","北京","010-62761226",500.00

这个数据有 11 项参数,除了名字不能省,可以缺少性别或者地址等其他参数。

(3) 删除满足条件的职工对象。

(4) 查询符合某些条件的职工对象。

(5) 对检索出的某些对象进行修改。

(6) 根据某些需要对显示的结果进行排序。

(7) 实现对某些域建立索引(可以是 B 树、B^+ 树、散列索引或者其他高效率的索引),并比较使用索引和不使用索引的查询速度。

(8) 对于输入的职工信息要有基本的纠错能力,例如像输入出生年月日是否合法。

(9) 对于排序当然要能够按照用户的格式进行,包括字母排序、数字排序、日期排序、升序、降序、多列排序等。

(10) 数据量规模在一万以上。

说明:

(1) 本实习题必须以 2~3 人一组(不能单独行动,也不能超过 3 人)。

(2) 一定要实现基本功能,并能很好的演示。

(3) 本次上机题,要求每个人都参与编程,并参与编写文档,在文档中说明每个人的分工。

提示:数据库文件使用 employee.dat,如果一个文件装不下可以使用 employee1.dat, employee2.dat,...。索引文件使用 name.idx, age.idx,...。

加分功能：

(1) 支持带有 *、? 等通配符的字符串查询。需要正则表达式匹配(Regular Expression)算法，可以上网查到源代码。

(2) 支持简单的 SQL 语句的查询。可以查《数据库概论》、《数据库系统实现》等书籍。注意，只需要支持最简单的 SQL 查询，可以读 mySQL、BerkeleyDB 等开放数据库源代码，但不能完全照搬。只能截取部分功能融合到自己的系统中。

(3) 能支持百万以上的职工数据量。

15. 搜索引擎

搜索引擎是一种收集、组织和处理大量信息资源，并向用户提供高效检索服务的工具。大体上，搜索引擎可以分为本地搜索引擎和网络搜索引擎两种，而这两种搜索引擎的基本原理也是相似的，只是由于数据容量和应用环境的不同，在设计和实现上稍有区别。

由于搜索引擎涉及图、排序、索引、检索等数据结构技术，而且设计搜索引擎这个综合上机题也能充分激发学生综合应用数据结构知识的兴趣，因此多年来北京大学计算机系一直把此题作为计算机专业学生的数据结构学期综合上机实习题。

目前互联网上使用的搜索引擎不计其数，比较典型的有 Google(www.google.com)、百度(www.baidu.com)和天网(e.pku.edu.cn)等。尽管它们在界面和功能上有一些差别，但是作为搜索引擎的基本结构却大体相同。一个搜索引擎可以分为 3 个部分，如图 12.49 所示。

(1) 网络爬虫(spider、web crawler)：也称为“搜索器”，在互联网上沿网页链接访问并保存页面信息，作为检索信息的来源。这一部分由爬虫程序自动完成，一般都日夜不停地运行。爬虫搜集的信息类型可以是文本型文档(例如，HTML 文本、XML 文本、正文文本等)、字处理型文档(例如，Word 文档、ppt 文档、PostScript 文档、PDF 文档等)、多媒体信息(例如，地图、图形、图像、音频和视频等)。Web 页作为结点，网页之间的链接作为边，可以把 Web 页的关系表示为图。网络爬虫的工作，本质上就是图的搜索。

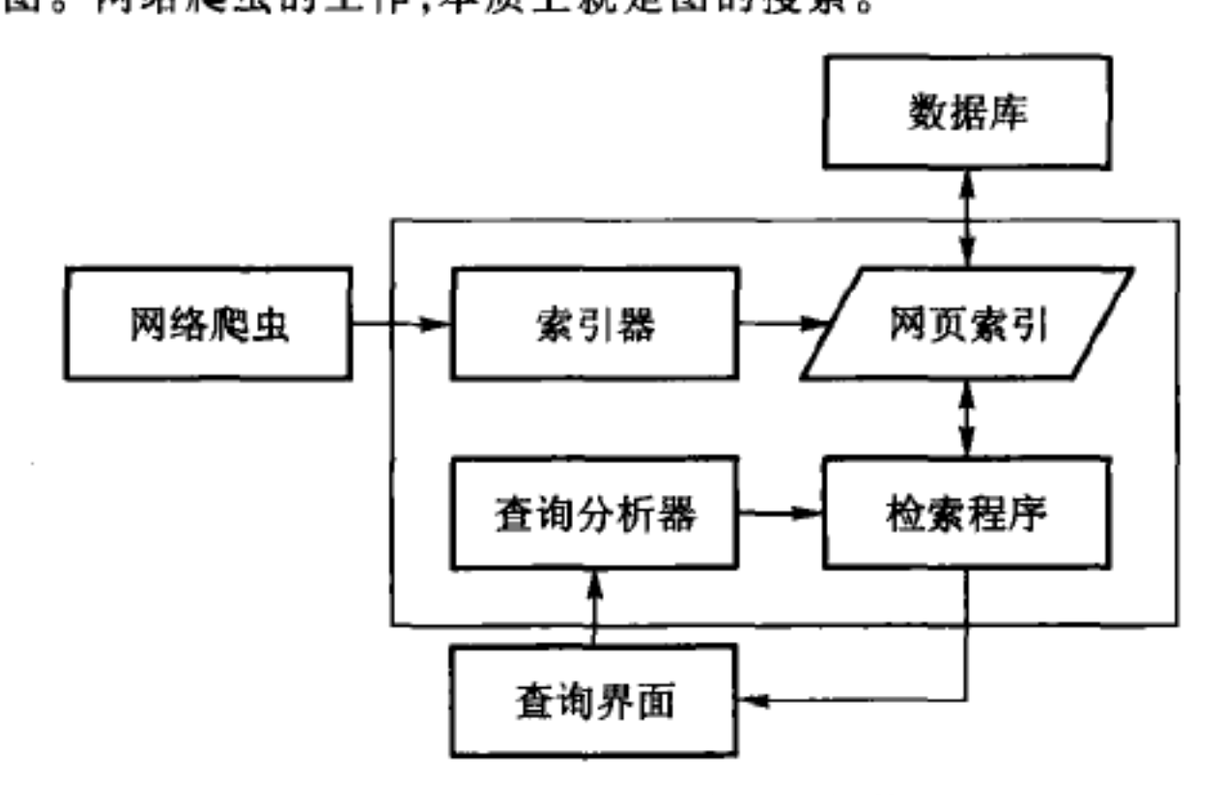


图 12.49 搜索引擎基本架构

(2) 索引器(indexer)：对爬虫收集来的网页信息，构造出便于快速访问和检索的索引数据结构。为了支持高效检索，索引器一般采用倒排索引(inverted index)。第 11 章介绍过，倒排索引的核心数据结构是倒排表(inverted table)，其每一表项记录某一单词所出现的文档编号序列。同时，为了快速定位待查询的单词，索引器还提供了高效的单词索引结构。由于对文档建立索引的过程一般在后台完成，因此对于建立索引这个过程的时间效率要求不太高。但是，检索索引的响应速度要求很高，一个搜索引擎的有效性在很大程度上取决于索引的质量。

(3) 查询处理(retriever):也称为"检索器",分为前台和后台两部分。前台提供用户检索的界面并接收用户查询输入(通常是关键词组合);后台主要包括数据库管理程序和结果检索程序。检索接口有直接用户接口和API接口两种,其中直接面对用户的接口能支持用户查询输入、查询结果显示,有些搜索引擎还提供用户反馈机制。另外,绝大部分实用搜索引擎都加入了结果排级(ranking)的功能模块以改进搜索结果。google的成功就在于它所发明的PageRank算法有效地解决了文档的相关性问题。当然,它还成功地使用了位置信息,也就是说它把与查询词相对位置最吻合的文档排在前面。

搜索引擎实习目标:应用已经掌握的数据结构的基础知识,设计并实现一个搜索引擎原型(可以是网页搜索、FTP搜索,也可以是MP3音乐、电影等专题搜索)。该原型能对一定规模的资源进行索引,能在合理的响应时间内满足用户的检索请求并返回结果。

16. Google Suggest的"提示"文本框等利用了Ajax控件,其后台有一个用户常用查询词词典。试帮助设计一个英文词典,该字典要能很好地支持前缀串查询。例如,输入comp,在提示框中应该能动态地返回如图12.50所示的提示,返回查询结果最多的10个相关词,按照访问频率在输入框中顺序排列这10个词。

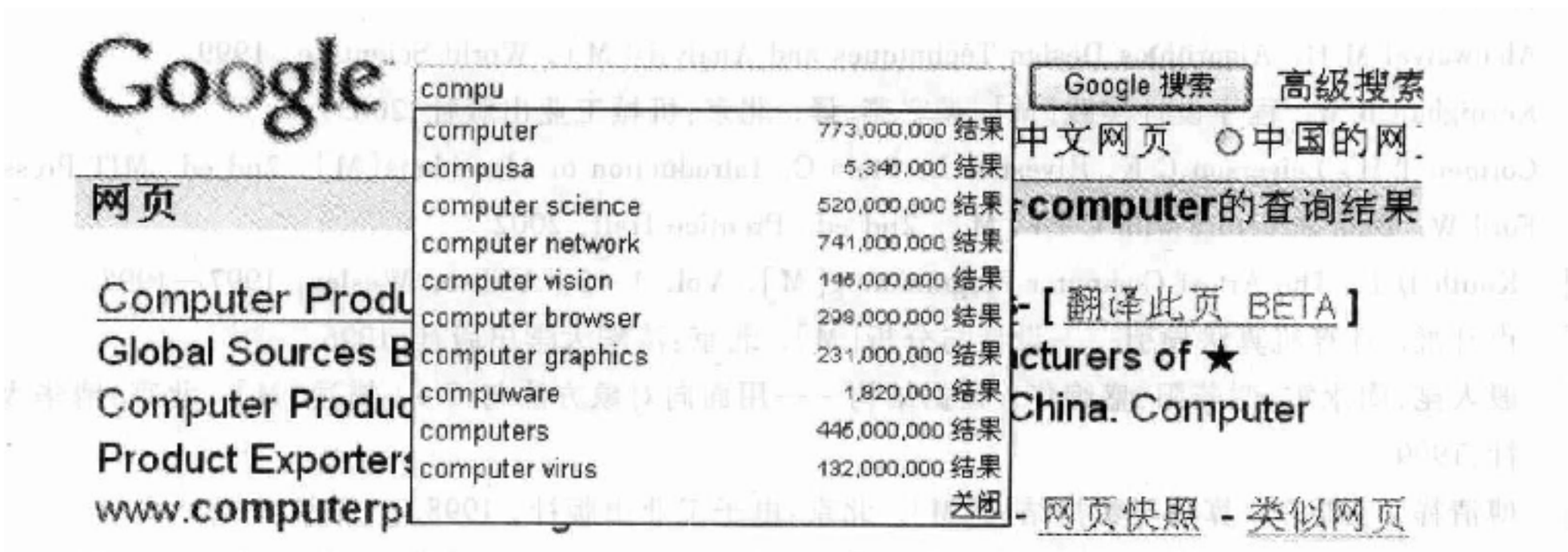


图 12.50 Google的Ajax输入提示框

参考文献

[1] 许卓群,杨冬青,唐世渭,张铭. 数据结构[M]. 北京:高等教育出版社,2004.

[2] 张铭,赵海燕,王腾蛟. 数据结构与算法——学习指导与习题解析[M]. 北京:高等教育出版社,2005.

[3] 许卓群,张乃孝,杨冬青,唐世渭. 数据结构[M]. 北京:高等教育出版社,1985.

[4] Clifford A S. 数据结构与算法分析——C++[M]. 2版. 张铭,刘晓丹. 译. 北京:电子工业出版社,2002.

[5] 张铭,黄维通. 从标准 Pascal 到 Delphi 4.0——计算引论和程序设计基础[M]. 北京:北京大学出版社,1999.

[6] Alsuwaiyel M H. Algorithms Design Techniques and Analysis[M], World Scientific, 1999.

[7] Kernighan B W. 程序设计实践[M]. 裘宗燕. 译. 北京:机械工业出版社,2003.

[8] Cormen T H, Leiserson C E, Rivest R L, Stein C. Introduction to Algorithms[M]. 2nd ed. MIT Press, 2001.

[9] Ford W. Data Structure with C++[M]. 2nd ed. Prentice Hall, 2002

[10] Knuth D E. The Art of Computer Programming[M]. Vol. 1~3. Addison Wesley, 1997—1998.

[11] 卢开澄. 计算机算法导引——设计与分析[M]. 北京:清华大学出版社,1996.

[12] 殷人昆,陶永雷,谢若阳,盛绚华. 数据结构——用面向对象方法与 C++ 描述[M]. 北京:清华大学出版社,1999.

[13] 傅清祥,王晓东. 算法与数据结构[M]. 北京:电子工业出版社,1998.

[14] Aho A V, Sethi R, Ullman J D. Compilers:Principles, Techniques, and Tools[M]. Addison-Wesley, 1986.

[15] Aho V A, Ullman J D, Hopcroft J E. Data Structures and Algorithms[M]. Addison-Wesley, 1983.

[16] Kruse R L, Ryba A J. Data Strutures and Program Design in C++[M]. Pearson Education, 2001.

[17] Shani S. Data Structures, Algorithms, and Applications in C++[M]. McGraw-Hill, 1998.

[18] 严蔚敏,吴伟民. 数据结构题集.(C 语言版)[M]. 北京:清华大学出版社,1999.

[19] 张乃孝,裘宗燕. 数据结构——C++ 与面向对象的途径[M]. 北京:高等教育出版社,2001.

[20] 胡学刚. 数据结构算法设计指导[M]. 北京:清华大学出版社,1999.

[21] 严蔚敏,吴伟民. 数据结构——C 语言版[M]. 北京:清华大学出版社,2001.

[22] Hughes C, Hughes T. Mastering the Standard C++ Classes[M]. John Wiley & Sons, 1999.

[23] Kruse R L, Ryba A J. Data Structures and Program Design in C++[M]. Pearson Addison-Wesley, 2001.

[24] Weiss M A. Data Structures and Algorithm Analysis in C++[M]. 2nd ed. Pearson Addison-Wesley, 1998.

[25] Silberschatz A, Korth H F, Sudarshan S. 数据库系统概念[M]. 3版. 杨冬青,唐世渭,等. 译. 北京:机械工业出版社,2000.

[26] Hector G M, Ullman J D, Widsom J. 数据库系统实现[M]. 杨冬青,唐世渭,徐其钧,等. 译. 北京:机械工业出版社,2001.