

江苏省精品课程主讲教材

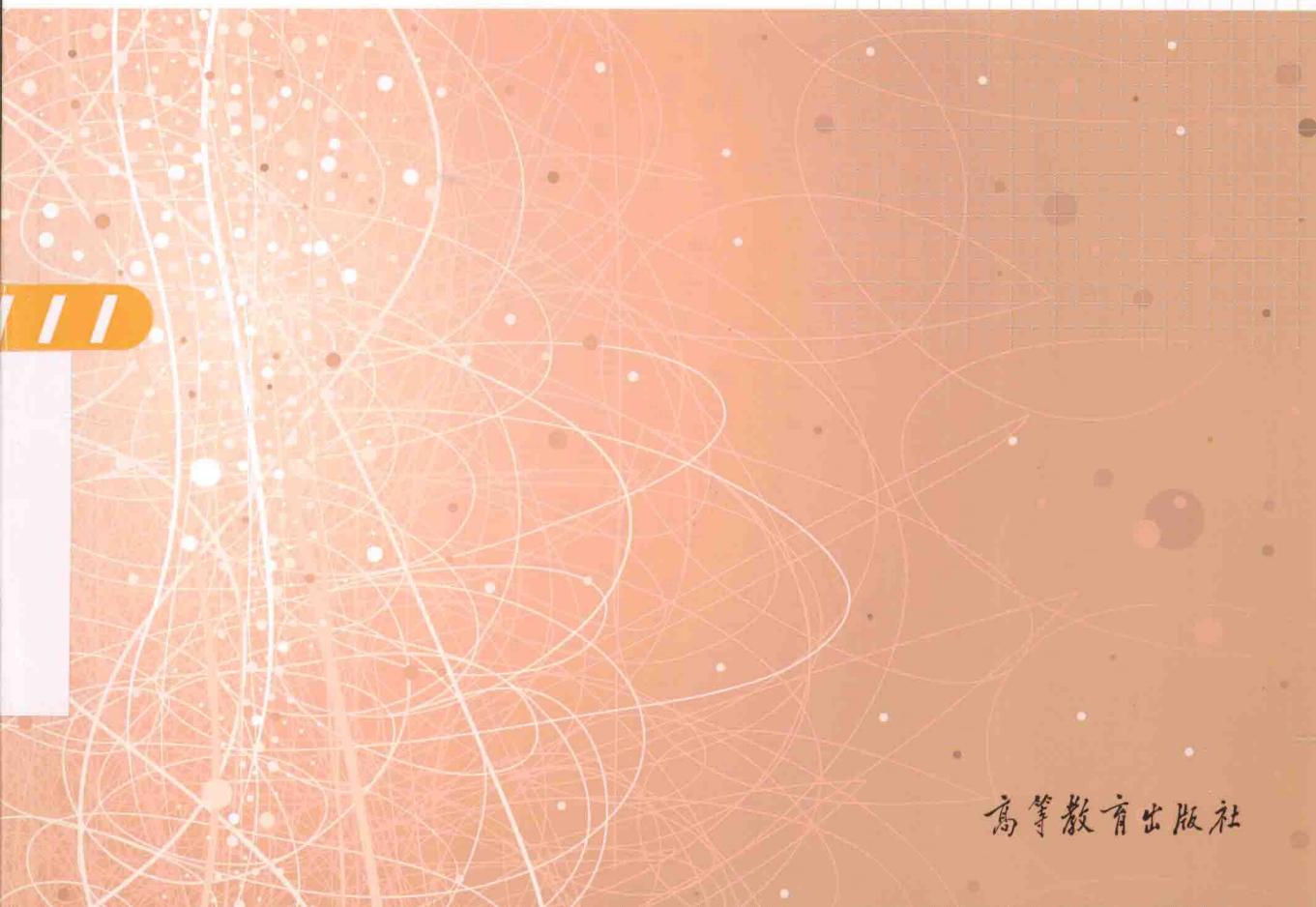
数据结构

(C++语言描述)

吉树林 陈 波 主 编

王 琼 周俊生 于 冷 编 著

*Data Structures
in C++*



高等教育出版社

本书特色

- 阐述常用数据结构的基本概念、逻辑关系、存储结构、操作运算及其实现算法，以及查找算法和排序算法，并对算法的性能进行分析。
- 使用C++类定义数据结构以及C++伪代码描述算法，给出多个经典算法和典型题例。
- 每章均附有小结、习题和上机实验题，附录包含多套考试样卷和课程设计题。
- 配套丰富的教学资源，包括电子教案、教学视频、习题解答、拓展资料等。

ISBN 978-7-04-040560-6



9 787040 405606 >

定价 27.00元

014058951

TP311.12

江苏省精品课程主讲教材

277

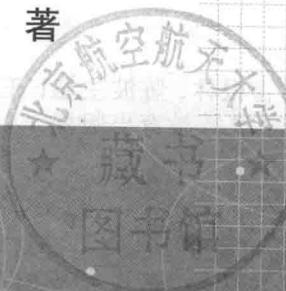
数据结构

(C++语言描述)

吉格林 陈波 主编

王琼 周俊生 于泠 编著

*Data Structures
in C++*



北航 C1746223

TP311.12
277

高等教育出版社·北京

内容提要

本书是江苏省精品课程“数据结构”的建设成果。全书共10章，介绍了各种常用的数据结构（线性表、栈和队列、串、数组和特殊矩阵、广义表、树和二叉树、图）的基本概念、逻辑关系、存储结构、操作运算及其实现算法；阐述了各种常用的查找算法和排序算法，并对各种算法的性能进行分析。书中使用C++类定义各种数据结构，利用C++伪代码描述算法，并给出了许多经典算法和典型题例。每章均附有小结、习题和上机实验题，附录给出了6套课程考试样卷和5道课程设计题。

本书既注重基本原理，又重视算法实现；既体现先进性，又强调实用性；内容丰富，重点突出，条理清晰，由浅入深，语言流畅，具有特色。本书的PPT课件和相关教学资源可从江苏省精品课程和南京师范大学精品课程“数据结构”网站 <http://computer.njnu.edu.cn/datastructure/index.asp> 下载。

本书可作为高等学校计算机类专业及相关专业“数据结构”课程的教材，也可供从事计算机软件开发人员参考。

图书在版编目(CIP)数据

数据结构：C++语言描述 / 吉根林，陈波主编；王琼，周俊生，于泠编著。--北京：高等教育出版社，
2014.8

ISBN 978-7-04-040560-6

I. ①数… II. ①吉… ②陈… ③王… ④周… ⑤于… III. ①数据结构 - 高等学校 - 教材②C 语言 - 程序设计 - 高等学校 - 教材 IV. ①TP311. 12②TP312

中国版本图书馆 CIP 数据核字(2014)第 161846 号

策划编辑 倪文慧	责任编辑 倪文慧	封面设计 王 洋	版式设计 杜微言
插图绘制 杜晓丹	责任校对 胡美萍	责任印制 尤 静	

出版发行	高等教育出版社	咨询电话	400-810-0598
社 址	北京市西城区德外大街 4 号	网 址	http://www.hep.edu.cn
邮 政 编 码	100120		http://www.hep.com.cn
印 刷	三河市华润印刷有限公司	网上订购	http://www.landraco.com
开 本	850mm×1168mm 1/16		http://www.landraco.com.cn
印 张	18.25	版 次	2014 年 8 月第 1 版
字 数	390 千字	印 次	2014 年 8 月第 1 次印刷
购书热线	010-58581118	定 价	27.00 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 40560-00

前　　言

“数据结构”是计算机类专业及相关专业的核心课程，主要研究和分析计算机存储、组织数据的方式和相关操作运算算法。通过本课程学习，学生不仅应当掌握数据结构和算法的基本概念和技术，掌握线性表、栈、队列、串、数组和特殊矩阵、广义表、树和二叉树、图等常用数据结构及相关算法，以及排序、查找等重要技术，而且能够针对应用问题选择合适的数据结构，并设计相应的操作运算算法。

南京师范大学的“数据结构”课程经过多年建设与探索，对教学内容与教学方法进行了改革与实践，有效地提高了教学质量，并被评为江苏省精品课程。课程组在教学中贯彻下列指导思想：

(1) 基础性。数据结构、算法和程序设计是计算机科学的核心知识，本课程应为学生的软件开发能力培养打下扎实的基础。

(2) 系统性。本课程以系统的观点研究数据组织和操作算法，重点在抽象思维、算法设计等方面加强学生的能力培养。

(3) 先进性。由于本课程的新思想和新方法不断产生，因此需要不断更新教学内容以拓宽学生的知识面，适应计算机应用和发展的需要。

(4) 实践性。本课程是一门实践性很强的课程，在课程实验中不仅要进行验证性实验，训练程序设计技能和操作能力，更应包括设计算法的创新性实验能力。

本书集作者多年讲授“数据结构”课程的教学经验，体现了科学性、先进性和实用性原则，既注重基本原理又重视算法实现，力求内容丰富、重点突出、条理清晰、由浅入深、语言流畅、具有特色。全书使用 C++ 类定义各种数据结构，利用 C++ 伪代码描述算法，并给出了许多经典算法和典型题例。每章均附有小结、习题和上机实验题；附录给出课程考试样卷和课程设计题，以供教学参考。

本书共分 10 章。第 1 章简要介绍数据结构和算法的基本概念；第 2~5 章介绍线性结构及其算法，包括线性表、栈、队列、串、数组和特殊矩阵；第 6~8 章介绍非线性结构及其算法，包括广义表、树、二叉树和图；第 9 章介绍各种常用的查找算法；第 10 章介

绍各种常用的排序算法。本书的 PPT 课件和相关教学资源可从江苏省精品课程和南京师范大学精品课程“数据结构”网站 <http://computer.njnu.edu.cn/datastructure/index.asp> 下载。

本书由南京师范大学“数据结构”课程组编写，其中，第 1 章和第 10 章由吉格林编写，第 2 章和第 3 章由陈波编写，第 6 章和第 7 章由王琼编写，第 5 章和第 8 章由周俊生编写，第 4 章和第 9 章由于泠编写。全书由吉格林和陈波担任主编，并负责统稿、修改和定稿。

由于作者水平有限，书中难免存在不妥之处，敬请读者批评指正。

编者 E-mail: glji@njnu.edu.cn

编者

2014 年 6 月

目 录

第1章 绪论	1
1.1 数据结构课程的研究内容	1
1.2 基本概念及术语	2
1.3 算法与算法分析	5
1.3.1 算法	5
1.3.2 算法分析	8
本章小结	10
习题1	11
上机实验题1	12
第2章 线性表	13
2.1 线性表的基本概念	13
2.2 线性表的存储结构	14
2.2.1 顺序存储结构	14
2.2.2 链式存储结构	15
2.3 线性表的操作算法	18
2.3.1 顺序表的操作算法	18
2.3.2 链表的操作算法	23
2.4 线性表的应用	33
2.5 顺序表和链表的综合比较	38
本章小结	39
习题2	39
上机实验题2	40
第3章 栈和队列	41
3.1 栈	41
3.1.1 栈的基本概念	41

3.1.2 栈的存储结构	41
3.1.3 栈的操作算法	43
3.1.4 栈的应用	47
3.2 队列	54
3.2.1 队列的基本概念	54
3.2.2 队列的存储结构	54
3.2.3 队列的操作算法	56
3.2.4 队列的应用	59
本章小结	60
习题3	60
上机实验题3	61
第4章 串	62
4.1 串的基本概念	62
4.2 串的存储结构	62
4.2.1 串的顺序存储结构	62
4.2.2 串的链式存储结构	63
4.3 串的操作算法	64
4.3.1 串的基本操作算法	64
4.3.2 串的模式匹配	65
4.3.3 串的应用	71
本章小结	73
习题4	73
上机实验题4	74
第5章 数组和特殊矩阵	75
5.1 数组	75
5.1.1 数组的基本概念	75

5.1.2 数组的存储结构	76	7.3.1 二叉树的顺序存储结构	105
5.2 特殊矩阵的压缩存储	77	7.3.2 二叉树的链式存储结构	107
5.2.1 对称矩阵的压缩存储	77	7.4 二叉树的遍历	110
5.2.2 三角矩阵的压缩存储	78	7.4.1 二叉树遍历的概念	111
5.2.3 对角矩阵的压缩存储	79	7.4.2 二叉树遍历算法	114
5.2.4 稀疏矩阵的压缩存储	79	7.4.3 二叉树的构造和析构算法 ..	117
本章小结	85	7.5 二叉树的其他操作算法	123
习题 5	86	7.6 线索二叉树	126
上机实验题 5	86	7.6.1 线索二叉树的概念	126
第 6 章 广义表	87	7.6.2 线索二叉树的存储结构	128
6.1 广义表的概念	87	7.6.3 线索二叉树的操作算法	129
6.2 广义表的存储结构	88	7.7 树的存储结构与算法	134
6.2.1 广义表中结点的结构	88	7.7.1 树的存储结构	134
6.2.2 广义表的存储结构	89	7.7.2 树的操作算法	140
6.3 广义表的操作算法	91	7.8 Huffman 树与 Huffman 编码	145
6.3.1 构造算法	91	7.8.1 Huffman 树的定义	145
6.3.2 遍历广义表	92	7.8.2 Huffman 树的构造	147
6.3.3 广义表的其他操作算法	93	7.8.3 Huffman 编码算法	150
本章小结	96	7.8.4 Huffman 译码算法	151
习题 6	97	7.8.5 Huffman 树的其他应用——	
上机实验题 6	97	程序设计流程优化	152
第 7 章 树和二叉树	98	7.9 等价类问题	154
7.1 树的概念和性质	98	7.9.1 等价类问题	154
7.1.1 树的定义	98	7.9.2 等价类的实现	155
7.1.2 树的基本术语	100	7.9.3 性能分析与改进	156
7.1.3 树的基本性质	101	本章小结	158
7.2 二叉树的概念和性质	102	习题 7	158
7.2.1 二叉树的定义	102	上机实验题 7	160
7.2.2 二叉树的基本性质	103		
7.3 二叉树的存储结构	105		
第 8 章 图	161		
8.1 图的基本概念	161		
8.1.1 图的定义	161		

8.1.2 图的基本术语	162	9.2.1 顺序查找	207
8.2 图的存储结构	166	9.2.2 折半查找	209
8.2.1 邻接矩阵表示法	166	9.2.3 分块查找	212
8.2.2 邻接表表示法	170	9.3 树表的查找	213
8.3 图的遍历	173	9.3.1 二叉排序树	214
8.3.1 图遍历的概念	173	9.3.2 平衡二叉树	220
8.3.2 深度优先搜索	173	9.3.3 B 树	224
8.3.3 广度优先搜索	175	9.3.4 B + 树	230
8.3.4 图遍历算法的应用	177	9.4 Hash 查找	231
8.4 最小生成树	180	9.4.1 Hash 查找的基本概念	231
8.4.1 最小生成树的概念及其 性质	180	9.4.2 Hash 表的构造	232
8.4.2 Prim 算法	182	9.4.3 Hash 查找算法及分析	236
8.4.3 Kruskal 算法	185	本章小结	238
8.5 最短路径	188	习题 9	238
8.5.1 最短路径的概念	188	上机实验题 9	239
8.5.2 单源最短路径	188	第 10 章 排序	241
8.5.3 每对顶点之间的最短路径	193	10.1 排序的基本概念	241
8.6 AOV 网与拓扑排序	196	10.2 冒泡排序	242
8.6.1 有向无环图与 AOV 网的 概念	196	10.3 选择排序	244
8.6.2 拓扑排序	197	10.4 插入排序	245
*8.7 AOE 网与关键路径	201	10.4.1 直接插入排序	245
8.7.1 AOE 网的概念	201	10.4.2 折半插入排序	247
8.7.2 关键路径	201	10.5 希尔排序	248
本章小结	203	10.6 快速排序	249
习题 8	204	10.7 堆排序	252
上机实验题 8	205	10.8 归并排序	257
第 9 章 查找	206	10.8.1 二路归并排序的非递归 实现	258
9.1 查找的基本概念	206	10.8.2 二路归并排序的递归 实现	260
9.2 顺序表的查找	207	10.9 基数排序	261

10.9.1 多关键字排序	262	10.9.2 链式基数排序	262	本章小结	264	习题 10	266	上机实验题 10	267	附录	268	A.2 数据结构试题 B	268		
附录 A 数据结构试题			268	A.3 数据结构试题 C			270	A.4 数据结构试题 D			271	A.5 数据结构试题 E			275
A.1 数据结构试题 A			268	A.6 数据结构试题 F			276	附录 B 数据结构课程设计题			278	参考文献			280
第 10 章 多关键字排序与链式基数排序															
第 11 章 数据结构课程设计题															
第 12 章 参考文献															

第1章 绪论

“数据结构”是计算机科学与技术、软件工程、信息安全、信息管理等专业的重要核心课程，主要分析计算机中数据的组织方式、存储结构和处理方法。数据结构课程的学习将为计算机及相关专业的后续课程（如操作系统、编译原理、数据库原理、软件工程等）的学习打下基础。实际上，要编写出“好”的程序，需要选择合理的数据结构和好的算法，而“好”算法的选择在很大程度上取决于描述实际问题所采用的数据结构。因此，要编写出“好”的程序，仅学习程序设计语言是不够的，还必须很好地掌握数据结构的基本知识和基本技能。本章将概要地介绍数据结构课程的研究内容、基本概念和基本思想。

1.1 数据结构课程的研究内容

数据结构起源于程序设计。随着计算机科学技术的发展，计算机应用领域不再局限于科学计算，而是更多地应用于信息处理、智能控制、办公自动化等领域。不仅计算机处理的对象由数值发展到字符串、表格、图形、图像、声音等数据，而且处理的数据量也越来越大。在程序设计中，应如何来组织和处理这样的数据呢？这正是“数据结构”课程需要研究的问题。

在使用计算机解决问题时，一般要经过下面几个步骤。首先，要将实际问题抽象出数学模型；然后，要针对数学模型设计出求解算法；最后，要编写程序并上机调试，直到求出最终结果。数值计算问题的数学模型一般可由数学方程或数学公式来描述。然而，对于非数值计算问题，如图书资料的检索、人-机博弈、课程表编排、最短路径求解等问题，它们的数学模型无法用数学方程或数学公式来描述，而是要使用线性表、树、图等数据结构来描述，并且要对这些模型设计相应算法来求解。数据结构就是研究计算机在解决非数值计算问题中使用的数据对象以及它们之间的关系和操作算法的学科，具体主要包含三个方面的内容：数据的逻辑结构、数据的存储结构和数据的操作算法。

数据结构作为一门独立的课程是1968年首先在美国开设的。在这之前，它的某些内容在其他课程中已有涉及。1968年，Stanford大学的D. E. Knuth教授建立了数据结构的最初体系，他所著的《计算机程序设计艺术：第1卷 基本算法》是一本较系统地阐述数据的逻辑结构、存储结构及其操作算法的著作。后来不同语言描述的数据结构图书相继出版，如有PASCAL语言、C语言、C++语言、Java语言等。我国于20世纪80年代初开始开设“数据结构”课程，该课程不仅是计算机专业核心课程，而且是其他信息类专业的必修课。

“数据结构”课程的内容随着程序设计技术的发展而发展，经历了结构化阶段和面向对象阶段。20世纪60—80年代，计算机开始广泛应用于非数值计算领域，数据组织成为程序设计的重要问题。人们认识到程序设计规范的重要性，提出了结构化程序设计的思想。数据结构概念的引入对程序设计的规范化起到了重要作用。图灵奖获得者瑞士计算机科学家N.Wirth教授曾提出“程序=数据结构+算法”。由此可以看出，数据结构和算法是构成程序的两个重要组成部分。

20世纪90年代以来，面向对象技术成为最流行的程序设计技术。在面向对象技术中，现实世界的实体被看作是一个对象。对象由属性和方法构成，属性描述实体的状态和特征，方法用以改变实体的状态或行为。一组具有相同属性和方法的对象集合称为类，每个具体的对象称为类的一个实例。例如，“学生”是一个类，“张丽”、“李明”等对象都是“学生”类的实例。

数据结构主要强调两个方面的内容：数据之间的关系，即数据之间的逻辑结构和存储结构；针对这些关系的基本操作。这两个方面实际上蕴含着面向对象的思想：类描述实体的属性和行为，而数据结构描述数据之间的关系及其基本操作。类与数据结构之间的对应关系如图1-1所示。

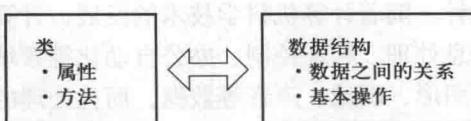


图1-1 类与数据结构之间的对应关系

值得一提的是，数据结构的发展并未终结。一方面，数据结构将继续随着程序设计技术的发展而发展；另一方面，面向专门领域的数据结构得到研究和重视，如研究人员提出了一些空间数据结构。

1.2 基本概念及术语

本节将介绍数据结构相关的基本概念及术语。

1. 数据

数据（Data）是信息的载体，是指所有能输入到计算机中并能被计算机程序识别和处理的符号集合。数据可以分为两大类：一类是整数、实数等数值数据，另一类是图形、图像、声音、文字等非数值数据。

2. 数据元素

数据元素（Data Element）是组成数据的基本单位，在计算机程序中通常作为一个整体进

行考虑和处理。构成数据元素的不可分割的最小单位称为数据项。例如，对于学生档案登记表，每个学生的档案是一个数据元素，而档案中的学号、姓名、出生日期等是数据项。数据元素是讨论数据时涉及的最小数据单位，其中的数据项一般不予考虑。

数据元素具有广泛的含义。一般来说，能独立、完整地描述问题世界的一切实体都是数据元素。例如，对弈中的棋盘格局、教学计划中的某门课程、一年中的4个季节，甚至一次学术报告、一场足球比赛都是数据元素。数据元素又称为元素、结点、顶点或记录。

3. 数据对象

数据对象（Data Object）是具有相同性质的数据元素的集合，是数据的子集。在实际应用中处理的数据元素通常具有相同性质。例如，学生档案登记表中每个数据元素具有相同数目和类型的数据项，所有数据元素（学生的档案）的集合就构成了一个数据对象。又如，字母数据集合 $M = \{'A', 'B', \dots, 'Z'\}$ 也是一个数据对象。

4. 数据结构

数据结构（Data Structure）是指数据元素及其相互关系的集合。这种相互关系即数据的组织形式，可分为数据的逻辑结构和数据的存储结构。

数据的逻辑结构（Logical Structure）是指数据元素之间的逻辑关系，即数据元素之间的关联方式或邻接关系。数据的逻辑结构分为以下4类：

- (1) 集合。数据元素之间的关系是属于同一个集合，除此之外，没有任何关系。
- (2) 线性结构。数据元素之间存在着一对一的线性关系。
- (3) 树结构。数据元素之间存在着一对多的层次关系。
- (4) 图结构。数据元素之间存在着多对多的任意关系。

树结构和图结构也称为非线性结构。

数据的逻辑结构常用逻辑结构图来描述，其描述方法是：将每一个数据元素看做一个结点，用圆圈表示；元素之间的逻辑关系用结点之间的连线表示。如果强调关系的方向性，则用带箭头的连线表示关系。图1-2描述了4种基本的数据逻辑结构。

为了更确切地描述一种数据结构，通常采用二元组形式化定义数据结构：

$$\text{Data_Structure} = (D, R)$$

其中， D 是数据元素的有限集合， R 是 D 上关系的有限集合。

例如，有一种数据结构 $T = (D, R)$ ，其中：

$$D = \{a, b, c, d, e, f, g, h, i, j\}$$

$$R = \{(a, b), (a, c), (a, d), (b, e), (c, f), (c, g), (d, h), (d, i), (d, j)\}$$

显然，数据结构 T 是一棵树形结构。

数据的存储结构（Storage Structure）又称物理结构，是数据及其逻辑结构在计算机中的表示。换言之，存储结构除了存储数据元素之外，还隐式或显示地存储数据元素之间的逻辑关

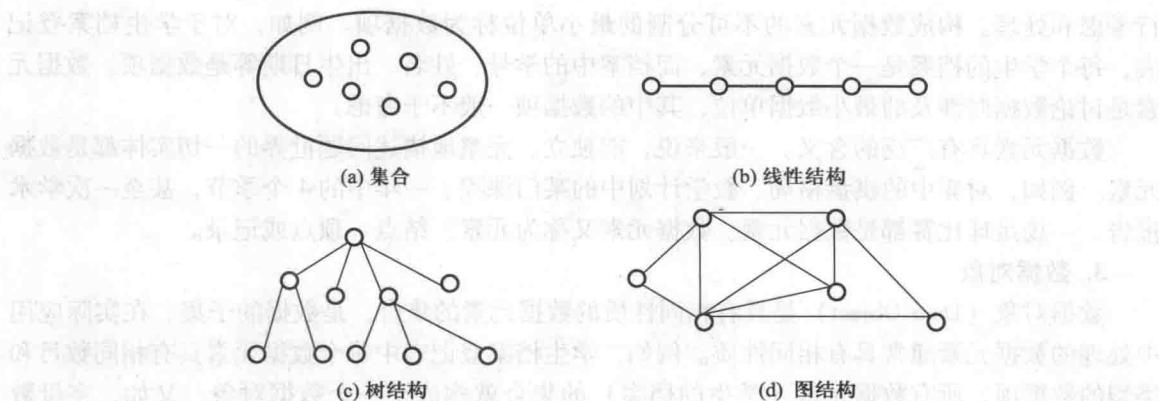


图 1-2 4 种基本的数据逻辑结构

系。通常有两种存储结构：顺序存储结构和链接存储结构。

顺序存储结构的基本思想是用一组连续的存储单元存储数据元素，数据元素之间的逻辑关系是由元素的存储位置来表示的。例如，线性表(a, b, c)的顺序存储示意图如图 1-3 所示。

链接存储结构的基本思想是用一组任意的存储单元存储数据元素，数据元素之间的逻辑关系是用指针来表示的。例如，线性表(a, b, c)的链接存储示意图如图 1-4 所示。

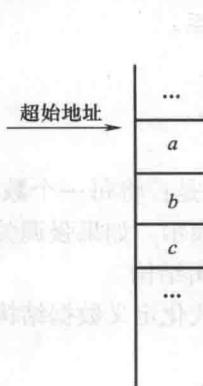


图 1-3 线性表的顺序存储示意图

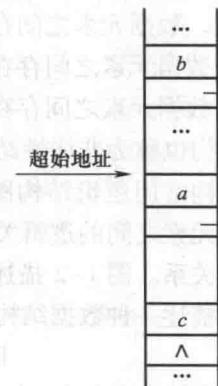


图 1-4 线性表的链接存储示意图

数据的逻辑结构和存储结构是密切相关的两个方面。一般来说，一种数据的逻辑结构可以用多种存储结构来存储；而采用不同的存储结构，其数据处理的效率往往是不同的。

5. 数据类型

数据类型 (Data Type) 是一组值的集合以及定义于这个值集上的一组操作的总称。每种程序设计语言都定义了自己的数据类型，如整型、实型、字符型、指针、数组、结构体、类

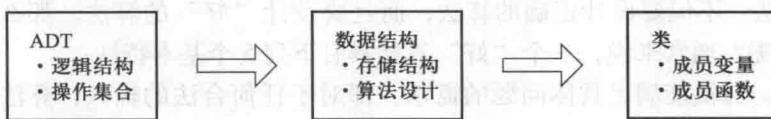
等。数据类型规定了该类型数据的取值范围和对这些数据所能采取的操作。例如，C++ 中的整型变量可以取的值是机器所能表示的最小负整数和最大正整数之间的任何一个整数，允许的操作有 +、-、*、/、%、<、<=、>、>=、==、!= 等。

6. 抽象数据类型

抽象数据类型（Abstract Data Type, ADT）是一种数据结构以及定义在该结构上的一组操作的总称，可被理解为对数据类型的进一步抽象。数据类型和 ADT 的区别在于：数据类型是指高级程序设计语言支持的基本数据类型，而 ADT 是指自定义的数据类型。例如，本课程将要学习的表、栈、队列、树、图等结构就是一些不同的 ADT。

ADT 包括定义和实现两个方面。其中，定义是独立于实现的，仅给出 ADT 的逻辑特征，而不必考虑如何在计算机中实现。ADT 的特征是将使用与实现分离，从而实现封装和信息隐藏。例如，整数的数学概念和施加到整数的运算构成一个 ADT，C++ 的变量类型 int 是对这个 ADT 的物理实现。各种程序设计语言都有整数类型。尽管它们在不同处理器上实现的方法不同，但由于其 ADT 相同，因此在用户看来都是相同的。

在设计 ADT 时，需要把 ADT 的定义和实现分开来，定义部分只包含数据逻辑结构的定义和所允许的操作集合。一方面，使用者依据这些定义来使用 ADT，即通过操作集合对该 ADT 进行操作；另一方面，ADT 的实现者依据这些定义来完成该 ADT 的各种操作的具体实现。图 1-5 所示的是 ADT 的不同视图。



(a) 使用视图：ADT 的定义 (b) 设计视图：ADT 的设计 (c) 实现视图：ADT 的实现

图 1-5 ADT 的不同视图

C++ 中的类体现了抽象数据类型的思想。在 ADT 中定义的每个操作由类中的成员函数来实现，数据以及数据之间的逻辑关系由类中的成员变量来实现。

1.3 算法与算法分析

1.3.1 算法

1. 什么是算法

算法（Algorithm）是计算机求解特定问题的方法和步骤，是指令的有限序列。通常一个问题可以有多种算法，一个给定算法解决一个特定的问题。

算法具有下列 5 个重要特性：

- (1) 输入。一个算法有零个或多个输入（即算法可以无输入），这些输入通常取自于某个特定的对象集合。
- (2) 输出。一个算法有一个或多个输出（即算法必须要有输出），通常输出与输入之间有着某种特定的关系。
- (3) 有穷性。对任何合法的输入，一个算法必须在执行有穷步之后结束，且每一步都在有穷时间内完成。
- (4) 确定性。算法中的每一条指令都必须有确切的含义，即不存在二义性。而且，在任何条件下，给定的算法对于相同的输入只能得到相同的输出。
- (5) 可行性。算法描述的操作可以通过已经实现的基本操作执行有限次来实现。

程序（Program）是对一个算法使用某种程序设计语言的具体实现，原则上任一算法可以用任何一种程序设计语言实现。算法的有穷性意味着不是所有的计算机程序都是算法。例如，操作系统是一个在无限循环中执行的程序，而不是一个算法。但是可以把操作系统的各个任务看成是一个单独的问题，每个问题由操作系统中的一个子程序通过特定的算法来实现，得到输出结果后便终止。

2. 算法的评价

数据结构与算法之间存在着本质联系。本课程学习的目的就是要在某一种数据结构基础上学习算法设计方法。不但要设计正确的算法，而且要设计“好”的算法。那么什么样的算法是“好”的算法呢？通常来说，一个“好”算法具有下列 5 个基本特性：

- (1) 正确性。算法能满足具体问题的需求，即对于任何合法的输入，算法都会得出正确的结果。
- (2) 健壮性（鲁棒性）。算法对非法输入的抵抗能力，即对于错误的输入，算法应能识别并做出处理，而不是产生错误动作或陷入瘫痪。
- (3) 可读性。好的算法应该便于人们理解和相互交流。可读性好的算法有助于人们对算法的理解；反之，难懂的算法易于隐藏错误且难于调试和修改。
- (4) 高效率。算法的效率通常是指算法的执行时间。对于同一个问题，如果有多个算法可以使用，那么执行时间短的算法效率高。
- (5) 低存储空间需求。算法需要的存储空间是指算法在执行过程中所需要的最大存储空间，它与问题规模有关。一个“好”算法应该占用较少的辅助空间。

3. 算法的描述方法

设计好了一个算法之后，必须清楚、准确地将所设计的求解步骤表达出来，即描述算法。算法描述方法通常有自然语言、流程图、伪代码和程序设计语言等。下面以欧几里得算法（用辗转相除法求两个自然数 m 和 n 的最大公约数，并假设 $m \geq n$ ）为例，介绍算法的描述。

方法。

(1) 自然语言

用自然语言描述算法的最大优点是容易理解，缺点是容易出现二义性，且算法描述通常都很冗长。欧几里得算法用自然语言描述如下：

- ① 输入 m 和 n 。
- ② 求 m 除以 n 的余数 r 。
- ③ 若 r 等于 0，则 n 为最大公约数，算法结束；否则执行第④步。
- ④ 将 n 的值放在 m 中，将 r 的值放在 n 中。
- ⑤ 重新执行第②步。

(2) 流程图

用流程图描述算法的优点是直观易懂，缺点是严密性不如程序设计语言，且灵活性不如自然语言。欧几里得算法的流程图如图 1-6 所示。

在计算机应用的早期，使用流程图描述算法占有统治地位。但是实践证明，除了一些非常简单的算法以外，这种描述方法使用起来非常不方便。

(3) 伪代码

伪代码是介于自然语言和程序设计语言之间的算法描述方法。计算机科学家对伪代码的书写形式没有做出严格的规定，它采用某一种程序设计语言的基本语法，操作指令可以结合自然语言来设计。伪代码的算法描述中自然语言的成分有多少，取决于算法的抽象级别。抽象级别高的伪代码中自然语言多一些，抽象级别低的伪代码中程序设计语言的语句多一些。只要具备程序设计语言基础的人都能阅读伪代码。用 C++ 伪代码描述的欧几里得算法如下：

```
int CommonFactor( int m,int n )
{
    r = m% n;
    while( r!=0 )
    {
        m = n;
        n = r;
        r = m% n;
    }
    return n;
}
```

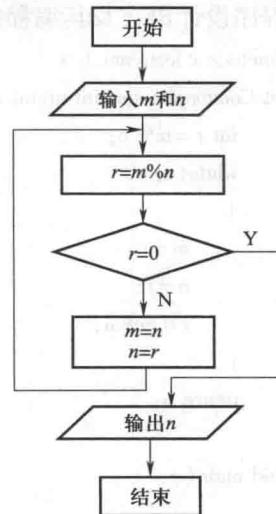


图 1-6 欧几里得算法的流程图

虽然伪代码不是一种实际的编程语言，但是由于它在表达能力上类似于编程语言，同时极小化了描述算法的不必要的技术细节，因此是比较合适的描述算法的方法，被称为算法语言。

本教材采用基于 C++ 语言的伪代码来描述算法，以使得算法的描述简明清晰，既不拘泥于 C++ 语言的实现细节，又容易转换为 C++ 程序。

(4) 程序设计语言

用程序设计语言描述的算法能由计算机直接执行，但缺点是抽象性差，使算法设计者拘泥于描述算法的具体细节，忽略了“好”算法和正确逻辑的重要性；此外，还要求算法设计者掌握程序设计语言及其编程技巧。欧几里得算法用 C++ 语言书写的程序如下：

```
#include <iostream.h>
int CommonFactor( int m, int n )
{
    int r = m % n;
    while( r != 0 )
    {
        m = n;
        n = r;
        r = m % n;
    }
    return n;
}
void main()
{
    cout << CommonFactor( 63, 54 ) << endl;
}
```

1.3.2 算法分析

一种数据结构的优劣是由实现其各种操作的算法决定的，对数据结构的分析实质上就是对实现各种操作的算法进行分析。分析时除了要验证算法是否能够正确解决问题外，还需要对算法的效率进行评价。对于一个实际问题的解决可以提出若干算法，那么如何从这些可行的算法中找出最有效的算法呢？或者有了一个解决实际问题的算法，如何来分析它的性能呢？这些问题都需要通过算法分析来确定。通常来说，算法分析主要分析算法的时间代价和空间代价这两个主要指标。

可能有人会认为，随着计算机功能的日益强大，程序的运行效率变得越来越不那么重要了。然而，计算机功能越强大，人们就越想去尝试解决更复杂的问题，而更复杂的问题就需要更大的计算量。实际上，我们不仅需要算法，而且需要“好”的算法。以破解密码的算法为

例，理论上，通过穷举法列出所有可能的输入字符的组合情况可以破解任何密码；但是，如果密码较长或组合情况太多，那么这个破解算法的执行就需要很长时间，可能是几年、十几年甚至更多，这样的算法显然没有实际意义。因此，在选择和设计算法时要有效率的观念，这一点比提高计算机本身的速度更为重要。

1. 度量算法效率的方法

如何度量一个算法的效率呢？一种方法是事后统计的方法，即先将算法实现，然后输入适当的数据运行，测算其时间和空间开销。事后统计的方法有以下缺点：

- 编写程序实现算法将花费较多的时间和精力。
- 所得实验结果依赖于计算机的软硬件等环境因素，有时容易掩盖算法本身的优劣。

因此，通常采用事前分析估算的方法——渐进复杂度（Asymptotic Complexity），它是对算法所消耗资源的一种估算方法。

2. 算法的时间复杂度

不考虑与计算机软、硬件有关的因素，影响算法时间代价的最主要因素是问题规模。问题规模是指输入量的多少。一般来说，它可以从问题描述中得到。例如，对一个具有 n 个整数的数组进行排序，其问题规模是 n ；对一个 m 行 n 列的矩阵进行转置，其问题规模是 $m \times n$ 。一个显而易见的事实是：几乎所有的算法对于规模更大的输入都需要运行更长的时间。例如，需要更多时间来对更大的数组排序，更大的矩阵转置需要更长的时间。因此，运行算法所需要的时间是问题规模 n 的函数。

要精确地表示算法的运行时间函数常常是很困难的，即使能够给出，也可能是个相当复杂的函数，函数的求解本身也是相当复杂的。算法时间分析度量的标准不是针对实际执行时间精确算出算法执行的具体时间，而是针对算法中语句的执行次数做出估计。假设算法中语句执行的总次数是问题规模 n 的某个函数 $f(n)$ ，可将算法时间量度记作 $T(n) = O(f(n))$ 。它表示随着问题规模 n 的增大，算法执行时间增长率和 $f(n)$ 增长率相同，称为算法的渐近时间复杂度，简称时间复杂度（Time Complexity），通常用大写字母 O 表示。时间复杂度是一个数量级的概念。在计算任何算法的时间复杂度时，可以忽略所有低次幂和最高次幂的系数，即

$$O(a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0) = O(n^m)$$

例 求下列各程序段的时间复杂度。

① `for(i=1;i<100;i++)`

`s++;`

该程序段的语句执行次数是常量，时间复杂度记为 $O(1)$ ，称为常量阶。

② `for(i=0;i<n;i++)`

`s+=i;`

该程序段的语句执行次数 $f(n) = 2n + 1$ ，时间复杂度 $T(n) = O(f(n)) = O(n)$ ，称为线

性阶。

```
③ for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        s++;
```

该程序段的语句执行次数 $f(n) = n + 1 + n(n + 1) + n^2 = 2n^2 + 2n + 1$, 时间复杂度 $T(n) = O(f(n)) = O(n^2)$, 称为平方阶。

```
④ for(i=0;i<n;i++)
    for(j=i;j<n;j++)
        s++;
```

这是一个二重循环, 循环体 $s++$ 的执行次数为: $n + (n - 1) + (n - 2) + \dots + 2 + 1 = n(n + 1)/2$;

该程序段的语句执行次数是 n^2 数量级, 因此时间复杂度 $T(n) = O(n^2)$ 。

```
⑤ for(i=1;i<=n;i=2*i)
    s++;
```

设循环体语句 $s++$ 的执行次数为 $f(n)$, 则有 $2^{f(n)} \leq n$, 即 $f(n) \leq \log_2 n$, 因此该段程序的时间复杂度为 $O(\log_2 n)$, 称为对数阶。

数据结构中常用的时间复杂度有 7 个: $O(1)$ 常量阶, $O(n)$ 线性阶, $O(n^2)$ 平方阶, $O(n^3)$ 立方阶, $O(2^n)$ 指数阶, $O(\log_2 n)$ 对数阶和 $O(n \log_2 n)$ 二维阶。

3. 算法的空间复杂度

算法的存储空间需求采用空间复杂度进行量度, 记做

$$S(n) = O(f(n))$$

其中, n 为问题的规模。一般情况下, 一个程序在机器上执行时, 除了需要寄存本身所用的指令、常数、变量和输入数据以外, 还需要一些对数据进行操作的辅助存储空间。由于对于输入数据所占的具体存储量只取决于问题本身, 与算法无关, 因此只需要分析该算法在实现时所需要的辅助空间单元个数即可。若算法执行时所需要的辅助空间相对于输入数据量而言是个常数, 则称这个算法为原地工作, 辅助空间为 $O(1)$ 。

算法执行时间的耗费和所占存储空间的耗费两者之间是存在矛盾的, 难以兼得。即算法执行时间上的节省一定是以增加空间存储为代价的, 反之亦然。不过, 就一般情况而言, 常常以算法执行时间作为算法优劣的主要衡量指标。

本章小结

“数据结构”是计算机专业的核心课程, 该课程较系统地介绍软件设计中常用的数据结构

及相应的存储结构和操作算法，以及常用的查找和排序技术，内容非常丰富。本课程的教学目标是要求学生学会分析数据对象特征，掌握数据组织方式和存储方法，并设计相应的操作算法，初步掌握算法的时间复杂度和空间复杂度的概念及分析技巧，培养良好的程序设计技能。

通过本章的学习，学生应达到下列学习目标：

- (1) 明确学习“数据结构”课程的意义，了解数据结构课程的研究内容。
- (2) 掌握数据结构的基本概念及术语。数据结构包括数据的逻辑结构、存储结构及操作算法 3 个部分。数据的逻辑结构分为线性结构和非线性结构两种，数据的存储结构分为顺序存储和链接存储两种。与数据结构相关的术语包括数据、数据元素、数据对象、数据类型和抽象数据类型等。
- (3) 理解算法的特性及算法的评价标准，了解算法的时间复杂度和空间复杂度的分析方法。

习题 1

1.1 简述数据结构与数据类型的区别与联系。

1.2 简述程序与算法的区别与联系。

1.3 算法具有什么特性？评价算法优劣的指标有哪些？

1.4 数据的逻辑结构和存储结构分别有哪些种类？

1.5 分析以下各程序段，求解相应的时间复杂度。

(1) $i = 1;$

$s = 0;$

 while($i < n$)

 {

$s = s + 10 * i;$

$i ++;$

 }

(2) $i = 1;$

$j = 0;$

 while($i + j < n$)

 if($i > j$) $j ++;$

 else $i ++$

(3) $y = 1;$

 while($y * y \leq n$) $y = y + 1;$

(4) $i = n;$

 while($i > 0$) $i = i / 2;$

```
(5) for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        for(k=1;k<=n;k++)
            s++;
(6) for(i=1;i<=n;i++)
    for(j=1;j<=i;j++)
        s++;
```

1.6 对下列用二元组表示的数据结构，试分别画出对应的逻辑结构图，并指出属于何种结构。

- (1) $A = (D, R)$, 其中 $D = \{x, y, z, p\}$, $R = \{\}$ 。
- (2) $B = (D, R)$, 其中 $D = \{a, b, c, d, e\}$, $R = \{(a, b), (b, e), (e, d), (d, c)\}$ 。
- (3) $G = (D, R)$, 其中 $D = \{a, b, c, d, e\}$, $R = \{(a, d), (c, e), (b, e), (a, b), (b, c), (d, e)\}$ 。
- (4) $T = (D, R)$, 其中 $D = \{1, 2, 3, 4, 5, 6\}$, $R = \{(6, 5), (6, 2), (2, 3), (2, 4), (2, 1)\}$ 。

1.7 对一个整型数组 $a[n]$ 设计一个排序算法，用 C++ 作为代码描述，并分析其时间复杂度。

1.8 何谓抽象数据类型？请谈谈对它的理解。

上机实验题 1

实验题 编写程序，求一个数组中的最大值和最小值。基本要求如下：

- (1) 利用随机函数生成若干个随机数，存放到一个数组中。
- (2) 要求程序结构模块化，求数组中的最大值和最小值用一个函数实现。
- (3) 分析算法的时间性能。

第2章 线性表

2.1 线性表的基本概念

线性表是一种简单的，也是最基本的线性结构。线性表的特点是数据元素之间仅具有单一前驱和后继关系，在一个线性表中数据元素的类型必须是相同的。

在实际应用中，线性表是一种常见的数据类型。例如，字符串“Data structure”是一个线性表，表中数据元素的类型为字符型。又如，学生情况信息表是一个线性表（如表 2-1 所示），表中数据元素的类型为由学号、姓名、性别、年龄、专业等组成的结构体类型。

表 2-1 学生情况信息表

学 号	姓 名	性 别	年 龄	专 业	...
130101	鲍国强	男	20	计算机	...
130102	陈 平	女	20	计算机	...
130103	李小虎	男	21	计算机	...
130104	朱 蕾	女	20	计算机	...
...

从上面的例子可以看出，线性表是具有相同数据类型的 $n(n \geq 0)$ 个数据元素组成的有限序列，通常记为

$$L = (a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$$

其中， a_i 是序号为 i 的数据元素 ($i = 1, 2, \dots, n$)。 a_1 称为表头元素， a_n 称为表尾元素。

线性表中所含数据元素的个数 n 称为线性表的长度， $n = 0$ 时称该线性表为空表。

线性表中相邻元素之间存在着顺序关系， a_{i-1} 称为 a_i 的直接前驱（简称为前驱）， a_{i+1} 称为 a_i 的直接后继（简称为后继）。

非空线性表的特点如下：

- (1) 有且仅有一个表头结点 a_1 ，它没有前驱，而仅有一个后继 a_2 。

(2) 有且仅有一个表尾结点 a_n , 它没有后继, 而仅有一个前驱 a_{n-1} 。

(3) 其余的结点 $a_i (2 \leq i \leq n-1)$ 都有且仅有一个前驱 a_{i-1} 和一个后继 a_{i+1} 。

线性表的长度可以根据需要加长或缩短, 即对线性表的数据元素不仅可以访问, 还可以进行插入和删除等操作。

2.2 线性表的存储结构

线性表有两种存储结构: 顺序存储和链式存储。下面分别讨论这两种存储结构。

2.2.1 顺序存储结构

线性表的顺序存储是指在内存中用地址连续的一块存储空间顺序存放线性表的各元素, 用这种存储形式存储的线性表又称为顺序表。

因为内存中的地址空间是线性的, 所以用物理上的相邻关系来实现数据元素之间的逻辑相邻关系, 既简单又自然。

如图 2-1 所示, 设顺序表第一个元素的存储地址 (首地址) 为 $Loc(a_1)$, 每个数据元素占 d 个存储单元, 则第 i 个数据元素的地址为

$$Loc(a_i) = Loc(a_1) + (i-1) \times d \quad 1 \leq i \leq n$$

这就是说, 只要知道顺序表首地址和每个数据元素所占存储单元的个数, 就可求出每个数据元素的地址, 即可以在 $O(1)$ 时间内存取数据元素, 这是顺序表可随机存取的特点。

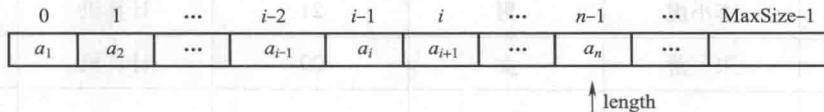


图 2-1 顺序表的存储示意图

在 C++ 程序设计语言中, 一维数组在内存中占用的存储空间是一组连续的存储区域。因此, 用一维数组来表示顺序表是一种简单而合适的方法。

需要注意的是, C++ 语言中数组下标从 0 开始, 因此顺序表中序号为 i 的元素, 存储在数组中的下标为 $i - 1$ 。

由于线性表的数据元素类型可以是任意的, 因此采用 C++ 模板机制。下面给出顺序表的类模板。

```
template < class T, int MaxSize >
```

```
class SeqList
```

```
{
```

```

T data[ MaxSize ] ;           //用于存放数据元素的数组
int length;                  //顺序表中元素的个数
public:
    SeqList( ) ;             //无参构造函数
    SeqList( T a[ ] , int n ) ; //有参构造函数
    int ListLength( ) ;       //求线性表的长度
    T Get( int pos ) ;        //按位查找,取顺序表的第 pos 个元素
    int Locate( T item ) ;    //按值查找,求顺序表中值为 item 的元素序号
    void PrintSeqList( ) ;    //遍历顺序表,按序号依次输出各元素
    void Insert( int i , T item ) ; //在顺序表中第 i 个位置插入值为 item 的元素
    T Delete( int i ) ;       //删除顺序表的第 i 个元素
};

```

其中 MaxSize 表示数组的最大容量。同时由于顺序表要进行插入、删除等操作，因此顺序表的长度是可变的，可用一个变量 length 来记录当前顺序表中元素的个数。

2.2.2 链式存储结构

线性表的链式存储结构又称链表，它用一组物理上不一定相邻的存储单元来存储线性表中的数据元素。

为了建立起数据元素之间的逻辑关系，对线性表中的每个数据元素 a_i ，除了存放数据元素的自身信息外，还需要存储其后继元素所在的地址信息，这个地址信息称为指针。数据元素自身信息和指针这两部分组成了数据元素的存储映像，称为结点。一般来说，一个结点可以包含一个或多个指针。含有一个指针使其指向后继的称为单链表，含有两个指针分别指向前驱和后继的称为双向链表。

1. 单链表

在单链表中，每个数据元素由一个结点表示，该结点包含两部分信息：数据元素自身的信
息和该元素后继的存储地址。数据域存放数据元素信息，指针域存放其后继地址，如图 2-2 所示。

结点定义如下：

```

template < class T >
struct Node
{
    T data;
    Node < T > * next;
};

```

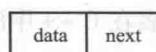


图 2-2 单链表结点结构

图 2-3 是线性表($a_1, a_2, a_3, a_4, a_5, a_6$)对应的单链表可能的存储示意图。

在单链表中,由于每个结点的存储地址存放在其前驱结点的 next 域中,因此第一个结点是没有前驱结点的,它的地址就是整个链表的开始地址,必须将第一个结点的地址放到一个指针变量(如图 2-3 所示的 head)中,该指针变量称为头指针。这样就可以从头指针开始,依次找到每个结点。通常用头指针来标识一个单链表。最后一个结点没有后继,其指针域必须置空(NULL),表明此单链表到此结束。

在单链表中,为了方便操作,有时在链表的第一个结点前加入一个头结点。头结点的类型与其他数据结点相同,标识链表的头指针变量 head 中存放该头结点的地址。这样即使是空表,头指针变量 head 也不为空了。头结点的加入使得单链表无论是否为空,头指针始终指向头结点,从而使空表和非空表的处理成为一致。

头结点的加入完全是为了操作的方便,它的数据域可以不存放任何信息,也可以存放有关链表的整体信息,如表长;指针域中存放的是第一个数据结点的地址,空表时其中为空。

图 2-4 (a) 和图 2-4 (b) 分别是带头结点的单链表空表和非空表的示意图。

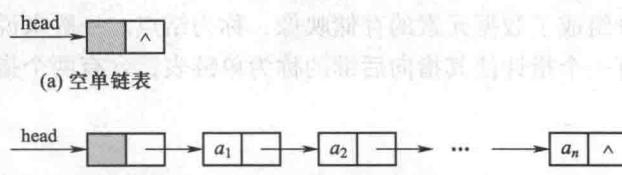


图 2-3 单链表存储示意图

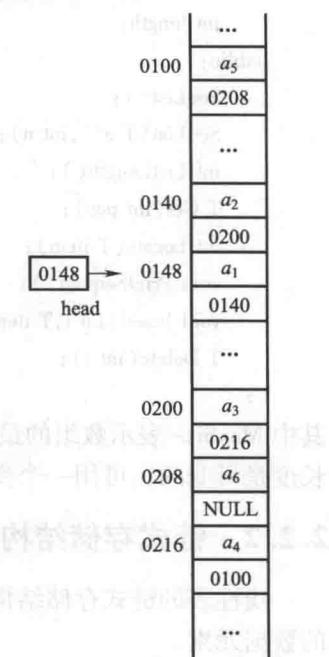


图 2-4 带头结点的单链表

单链表在 C++ 中的类模板实现如下:

```
template < class T >
class LinkList
{
    Node < T > * head;           //单链表的头指针
public:
    LinkList();                  //建立带头结点的空链表
    LinkList(T a[], int n);     //建立有 n 个元素的单链表
```

```

~ LinkList();
    //析构函数

int ListLength();
    //求单链表的长度

T Get( int pos );
    //按位查找,取单链表中第 pos 个结点的元素值

int Locate( T item );
    //按值查找,求单链表中值为 item 的元素序号

void PrintLinkList();
    //遍历单链表,按序号依次输出各元素

void Insert( int i, T item );
    //在单链表中第 i 个位置插入元素值为 item 的结点

T Delete( int i );
    //在单链表中删除第 i 个结点

};


```

2. 循环链表

如果将单链表最后一个结点的指针域指向头结点, 就使得整个链表形成了一个环。这种链表称为单循环链表, 简称循环链表, 如图 2-5 所示。

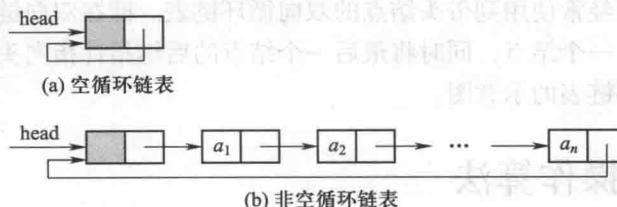


图 2-5 带头结点的单循环链表

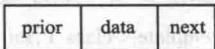
循环链表的操作与单链表类似, 区别仅在于判断单链表结束的条件是指针是否为空, 而判断循环链表结束的条件是指针是否指向头结点。

在单链表中只能从头结点开始遍历(依次访问)整个链表, 但在循环链表中则可以从任意结点开始遍历整个链表。不仅如此, 如果需要对链表常做的操作是在表尾进行, 那么可以改变一下链表的标识方法, 不用头指针而用一个指向表尾结点的指针来标识, 这有助于提高操作效率。

3. 双向链表

上述循环链表尽管可以从任意结点出发访问到其他任何结点, 但由于结点中只有一个指向其后继结点的指针域 `next`, 因此如果已知某结点的指针为 `p`, 那么其后继结点的指针则为 `p->next`, 而要找其前驱则只能顺着各结点的 `next` 域进行。也就是说, 找后继的时间复杂度是 $O(1)$, 找前驱的时间复杂度是 $O(n)$ 。如果希望找前驱的时间复杂度也能达到 $O(1)$, 则只能付出空间的代价, 图 2-6 双向链表结点结构即为每个结点增加一个指向后驱的指针域, 结点的结构为如图 2-6 所示, 用这种结点组成的链表称为双向链表。

与单链表类似, 双向链表通常也是用头指针标识, 增加头结点同样可以简化双向链表的操



作。图 2-7 所示的是带头结点的双向链表的示意图。

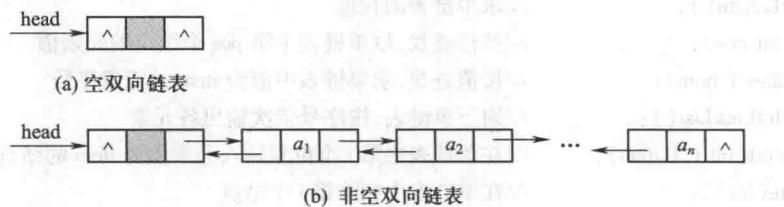


图 2-7 带头结点的双向链表示意图

可以看到，如果已知某结点的指针为 p，则其后继结点的指针是 $p \rightarrow \text{next}$ ，其前驱结点的指针是 $p \rightarrow \text{prior}$ 。

在实际应用中，还经常使用到带头结点的双向循环链表。即在双向链表的基础上，将头结点的前驱指针指向最后一个结点，同时将最后一个结点的后继指针指向头结点。读者可自行给出带头结点的双向循环链表的示意图。

2.3 线性表的操作算法

2.3.1 顺序表的操作算法

1. 初始化操作——构造函数

本小节为顺序表提供了两个构造函数用以初始化。其中，无参构造函数用于创建一个空表，有参构造函数 $\text{SeqList}(T a[], \text{int } n)$ 用于创建长度为 n ，元素为数组 a 中元素的顺序表。下面分别给出这两个构造函数的具体描述。

算法 2-1 顺序表的无参构造函数

```
template < class T, int MaxSize >
```

```
SeqList < T, MaxSize > :: SeqList( )
```

```
{ length = 0; }
```

算法 2-2 顺序表的有参构造函数

```
template < class T, int MaxSize >
```

```
SeqList < T, MaxSize > :: SeqList( T a[], int n )
```

```
{
```

```
if( n > MaxSize )
```

```
{
```

```
cerr << "参数非法";
```

```

    exit(1);
}
for(i=0;i<n;i++)
    data[i] = a[i];
length = n;
}

```

2. 求顺序表的长度

由于在顺序表的类模板定义中，成员变量 length 用于记录当前线性表中元素的个数，因此要求顺序表长度只需直接返回 length 即可，如算法 2-3 所示。

算法 2-3 求顺序表的长度

```

template < class T, int MaxSize >
int SeqList < T, MaxSize > ::ListLength()
{
    return length;
}

```

3. 按位查找

由于顺序表中第 pos 个元素存储在数组 data 中下标为 pos - 1 的位置，因此按位查找实现如算法 2-4 所示。

算法 2-4 顺序表的按位查找

```

template < class T, int MaxSize >
T SeqList < T, MaxSize > ::Get( int pos )
{
    if( pos < 1 || pos > length )
    {
        cerr << "查找位置非法";
        exit(1);
    }
    return data[ pos - 1 ];
}

```

4. 按值查找

按值查找将待查找的值 item 与顺序表中的元素依次进行比较，如果查找到具有 item 值的元素时，则返回该元素的序号；否则返值 0，表明查找失败，如算法 2-5 所示。

算法 2-5 顺序表的按值查找

```

template < class T, int MaxSize >
int SeqList < T, MaxSize > ::Locate( T item )
{
    for(i=0;i<length;i++)

```

```

if( data[ i ] == item )
    return i + 1;
return 0;
}

```

5. 遍历顺序表

遍历顺序表只需依次输出数组 data 中的元素即可，如算法 2-6 所示。

算法 2-6 遍历顺序表

```

template < class T, int MaxSize >
void SeqList < T, MaxSize > ::PrintSeqList()
{
    for( i = 0; i < length; i++ )
        cout << data[ i ] << endl;
}

```

6. 插入

顺序表的插入是指在表的第 i 个位置上插入一个值为 item 的新元素，插入后使原长度为 n 的顺序表变成长度为 $n+1$ 的表，如图 2-8 所示。

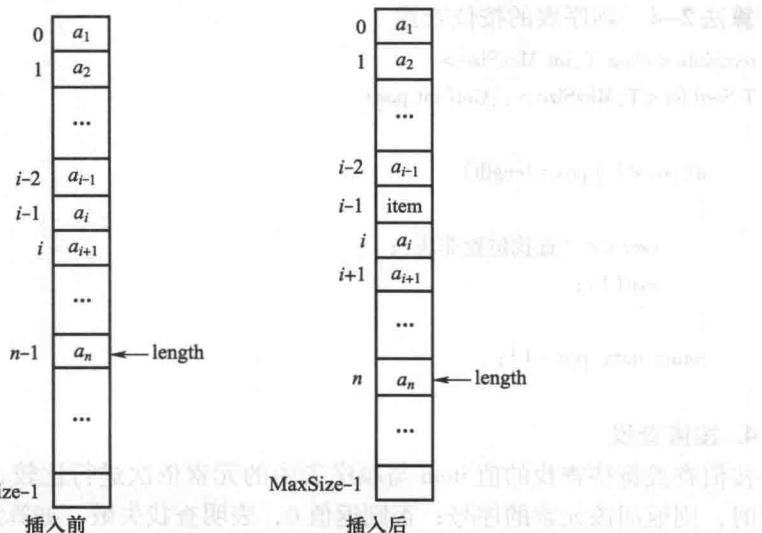


图 2-8 顺序表中的插入

顺序表的插入算法步骤如下：

- ① 检查顺序表的存储空间是否已到最大值（被占满），若是则停止插入，抛出上溢异常；否则执行第②步。

② 检查插入位置 i 是否合法，若不合法，则停止插入，抛出插入位置异常；否则执行第③步。

③ 从最后一个元素向前直至第 i 个元素（下标为 $i-1$ ）为止，将每一个元素均后移一个存储单元，将第 i 个元素的存储位置空出。

④ 将新元素 item 写入到第 i 个元素处，即下标为 $i-1$ 的位置。

⑤ 将顺序表长度加 1。

顺序表插入算法的具体实现如算法 2-7 所示。

算法 2-7 顺序表的插入

```
template < class T, int MaxSize >
void SeqList < T, MaxSize > ::Insert( int i, T item )
{
    if( length >= MaxSize )
    {
        cerr << "上溢" ;
        exit(1) ;
    }
    if( i < 1 || i > length + 1 )
    {
        cerr << "插入位置非法" ;
        exit(1) ;
    }
    for(j = length - 1; j >= i - 1; j -- )
        data[ j + 1 ] = data[ j ];
    data[ i - 1 ] = item;
    length ++ ;
}
```

可以看出，顺序表上的插入运算，时间主要消耗在数据的移动上。对于一个长度为 n 的顺序表 $(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$ ，在第 i 个位置上插入 item，需要从 a_i 到 a_n 都要向后移动一个位置，共需要移动 $n-i+1$ 个元素。而 i 的取值范围为 $1 \leq i \leq n+1$ ，即有 $n+1$ 个位置可以插入。设在第 i 个位置上插入的概率为 p_i ，则平均移动数据元素的次数为

$$E_{in} = \sum_{i=1}^{n+1} p_i (n - i + 1)$$

设 $p_i = 1/(n+1)$ ，即为等概率情况，则

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

这说明在顺序表上进行插入操作，需移动表中一半的数据元素。该算法的时间复杂度为 $O(n)$ 。

7. 删除

顺序表的删除运算是指将表中第 i 个元素从线性表中去掉，删除后使原长度为 n 的线性表变成长度为 $n - 1$ 的线性表 $(a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$ ，如图 2-9 所示。

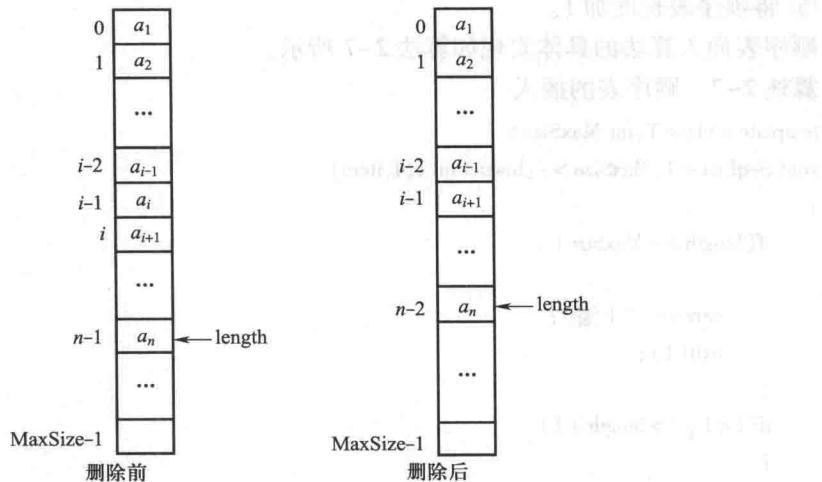


图 2-9 顺序表中的删除

顺序表中的删除算法步骤如下：

- ① 检查顺序表是否为空，若是则抛出下溢异常；否则执行第②步。
 - ② 检查删除位置 i 是否合法，若不合法，则抛出删除位置异常；否则执行第③步。
 - ③ 取出被删除元素。
 - ④ 从第 $i + 1$ 个元素（下标为 i ）向后直至最后一个元素为止，将每一个元素均前移一个存储位置。
 - ⑤ 将顺序表的长度减 1。
 - ⑥ 返回被删除元素的值。
- 顺序表删除算法的具体实现如算法 2-8 所示。

算法 2-8 顺序表的删除

```
template < class T, int MaxSize >
T SeqList < T, MaxSize > ::Delete( int i )
{
    if( length == 0 )
```

```

    {
        cerr << "下溢";
        exit(1);
    }
    if(i < 1 || i > length)
    {
        cerr << "删除位置非法";
        exit(1);
    }
    x = data[i - 1];
    for(j = i; j < length; j++)
        data[j - 1] = data[j];
    length--;
    return x;
}

```

与插入操作相同，顺序表的删除操作时间主要消耗在移动表中元素上。对于一个长度为 n 的顺序表 $(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$ ，删除第 i 个元素时，其后面的元素 $a_{i+1} \sim a_n$ 都要向前移动一个位置，共移动了 $n-i$ 个元素，因此平均移动数据元素的次数为

$$E_{de} = \sum_{i=1}^n p_i (n - i)$$

在等概率情况下， $p_i = 1/n$ ，则

$$E_{de} = \sum_{i=1}^n p_i (n - i) = \frac{1}{n} \sum_{i=1}^n (n - i) = \frac{n-1}{2}$$

这说明在顺序表上进行删除操作大约需要移动表中一半的元素。该算法的时间复杂度为 $O(n)$ 。

2.3.2 链表的操作算法

本小节的讨论均基于带头结点的链表。

1. 单链表的初始化——构造函数

本小节为单链表提供了两个构造函数用以初始化。其中，无参构造函数用于创建一个带头结点的空单链表，有参构造函数 $\text{LinkList}(T a[], \text{int } n)$ 用于创建含有数组 a 中 n 个元素的带头结点的单链表。下面分别给出这两个构造函数的具体描述。

算法 2-9 单链表的无参构造函数

template < class T >

```
LinkList < T > :: LinkList()
{
    head = new Node < T > ;
    head -> next = NULL;
}
```

算法 2-10 单链表的有参构造函数

```
template < class T >
LinkList < T > :: LinkList( T a[], int n)
{
    head = new Node < T > ; //生成头结点
    rear = head; //指针 rear 用于指向当前单链表的最后一个结点
    for( i = 0; i < n; i ++ ) //为每个数组元素生成一个新结点，并插入到链表尾部
    {
        s = new Node < T > ;
        s -> data = a[ i ]; //指针 s 用于指向新结点
        rear -> next = s;
        rear = s;
    }
    rear -> next = NULL; //单链表建立完毕，将最后一个结点的指针域置空
}
```

算法 2-10 采用的是尾部插入法，即每次生成的新结点均插在链表的尾部，为了操作简单，算法中设计了一个指针 rear，让其一直指向当前单链表的最后一个结点。

还可以采用头部插入法来构造单链表，即每次新生成的结点均插在头结点的后面。头部插入法要比尾部插入法更为简单些，请读者自行完成。

2. 求单链表长度

设置一个指针，从单链表第一个数据结点开始向后扫描，直至单链表结束。在扫描过程中，利用计数器同步计数，如算法 2-11 所示。

算法 2-11 求单链表长度

```
template < class T >
int LinkList < T > :: ListLength()
{
    num = 0;
    p = head -> next;
    while( p )
    {
```

```

    p = p -> next;
    num++;
}
return num;
}

```

3. 按位查找

由于单链表不能像顺序表那样进行随机访问，而是必须从头指针开始依次访问，因此按位查找算法步骤如下：

- ① 初始化指针 p 和计数器 j。
- ② 当 p 不为空或 j 不等于 pos 时，p 后移指向下一个结点，同时 j 加 1。
- ③ 若 p 为空，则抛出查找位置非法异常；否则 p 指向需查找的元素，返回 p 所指向结点的数据。

按位查找的具体实现如算法 2-12 所示。

算法 2-12 单链表的按位查找

```

template < class T >
T LinkList < T > ::Get( int pos )
{
    p = head -> next;
    j = 1; // p 初始化为第一个数据结点的地址, j 初始化为 1
    while( p && j < pos )
    {
        p = p -> next;
        j++;
    }
    if( !p || j > pos )
    {
        cerr << " 查找位置非法 ";
        exit( 1 );
    }
    else
        return p -> data;
}

```

请读者思考，在初始化指针 p 和计数器 j 时，算法 2-12 是将 p 指向第一个数据结点，j 初始值为 1。除此之外，p 和 j 的初始值还可以取什么？

4. 按值查找

按值查找将待查找的值 item 与单链表中的每个结点元素依次进行比较，如果查找到具有 item 值的元素时，则返回该元素的序号；否则返回值 0，表明查找失败。

按值查找的具体实现如算法 2-13 所示。

算法 2-13 单链表的按值查找

```
template < class T >
int LinkList < T > :: Locate(T item)
{
    p = head -> next; j = 1;
    while( p&&p -> data != item)
    {
        p = p -> next;
        j++;
    }
    if(p)
        return j;
    else
        return 0;
}
```

5. 遍历单链表

遍历单链表依次输出单链表中除头结点外的每个结点的数据域即可，其具体实现如算法 2-14 所示。

算法 2-14 遍历单链表

```
template < class T >
void LinkList < T > :: PrintLinkList( )
{
    p = head -> next;
    while( p)
    {
        cout << p -> data << endl;
        p = p -> next;
    }
}
```

6. 插入

插入操作是指将值为 item 的新结点插入到单链表第 i 个位置上。根据插入的方法，可以分

为后插和前插两种，下面分别介绍这两种方法。

(1) 后插法

如图 2-10 所示，设 p 指向单链表中的某结点， s 指向待插入的值为 item 的新结点，将结点 s 插入到结点 p 的后面，即为后插法。

后插法的操作步骤如下：

- ① $s \rightarrow \text{next} = p \rightarrow \text{next};$
- ② $p \rightarrow \text{next} = s;$

注意：两个指针的操作顺序不能交换。

(2) 前插法

如图 2-11 所示，设 p 指向链表中某结点， s 指向待插入的值为 item 的新结点，将结点 s 插入到结点 p 的前面，即为前插法。

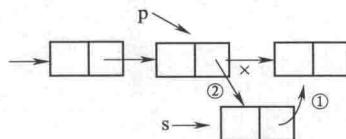


图 2-10 在结点 p 之后插入结点 s

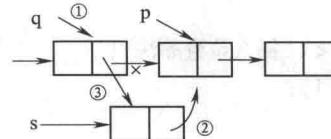


图 2-11 在结点 p 之前插入结点 s

与后插法不同的是，前插法首先要找到结点 p 的前驱结点 q ，然后再完成在结点 q 之后插入结点 s 。设单链表头指针为 head ，前插法的操作步骤如下：

- ```

① $q = \text{head};$
while($q \rightarrow \text{next} \neq p$) //找结点 p 的直接前驱
 $q = q \rightarrow \text{next};$
② $s \rightarrow \text{next} = q \rightarrow \text{next};$
③ $q \rightarrow \text{next} = s;$

```

从上述分析可以看出，后插的效率比前插高，后插操作的时间复杂性为  $O(1)$ ，前插操作的时间复杂度为  $O(n)$ 。

因此，在设计将值为 item 的新结点插入到单链表第  $i$  个位置的算法时，采用后插法，即要插人在第  $i$  个位置就是插人在第  $i - 1$  个位置的后面。算法中首先需要找到第  $i - 1$  个结点的位置。算法步骤如下：

- ① 工作指针  $p$  初始化，累加器  $j$  清零。
- ② 查找第  $i - 1$  个结点，并将指针  $p$  指向该结点。
- ③ 若  $p$  为空，即查找不到，则抛出插入位置非法异常；否则，生成元素值为 item 的新结点  $s$ ，并按后插法将其插入到结点  $p$  的后面。

后插法的具体实现如算法 2-15 所示。

### 算法 2-15 单链表的插入

```

template < class T >
void LinkList < T > ::Insert(int i, T item)
{
 p = head;
 j = 0;
 while(p && j < i - 1) //找到第 i - 1 个结点的位置
 {
 p = p -> next;
 j++;
 }
 if(!p)
 {
 cerr << "插入位置非法";
 exit(1);
 }
 else
 {
 s = new Node < T >; //生成元素值为 item 的新结点 s
 s -> data = item;
 s -> next = p -> next; //用后插法将 s 插入到结点 p 的后面
 p -> next = s;
 }
}

```

### 7. 删除

设 q 指向单链表中某结点，删除结点 q 的操作如图 2-12 所示。

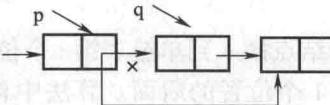


图 2-12 删除结点 q

从图 2-12 可知，要删除结点 q，首先要找到结点 q 的前驱结点 p，然后再完成指针的操作即可。指针的操作步骤如下：

```

p -> next = q -> next;
delete q;

```

下面给出删除第  $i$  个结点的算法。

从上述分析可以看出，要删除第  $i$  个结点，必须先找到第  $i-1$  个结点。因此，删除结点的算法步骤如下：

- ① 工作指针  $p$  初始化，累加器  $j$  清零。
- ② 查找第  $i-1$  个结点，并将指针  $p$  指向该结点。
- ③ 若  $p$  为空或  $p$  不存在后继结点，则抛出删除位置非法异常；否则删除结点  $p$  的后一个结点。

删除结点的具体实现如算法 2-16 所示。

### 算法 2-16 单链表的删除

```
template < class T >
T LinkList < T > ::Delete(int i)
{
 p = head;
 j = 0;
 while(p && j < i - 1) //查找第 i-1 个结点
 {
 p = p -> next;
 j++;
 }
 if(!p || !p -> next)
 {
 cerr << "删除位置非法";
 exit(1);
 }
 else
 {
 q = p -> next;
 x = q -> data;
 p -> next = q -> next;
 delete q;
 return x;
 }
}
```

### 8. 单链表的析构函数

由于单链表类中由 new 运算符生成的结点空间无法自动释放，因此需要利用析构函数将单

链表的存储空间加以释放，其具体实现如算法 2-17 所示。

### 算法 2-17 单链表的析构函数

```
template < class T >
LinkList < T > :: ~ LinkList()
{
 p = head;
 while(p)
 {
 q = p;
 p = p -> next;
 delete q;
 }
 head = NULL;
}
```

## 9. 单链表的其他操作举例

### 例 2-1 逆置单链表

将一个单链表按逆序链接，即若原单链表中存储元素的次序为  $a_1, a_2, \dots, a_{n-1}, a_n$ ，则逆序链接后变为  $a_n, a_{n-1}, \dots, a_2, a_1$ 。

图 2-13 (a) 所示为逆置前的单链表，图 2-13 (b) 所示为逆置后的单链表。

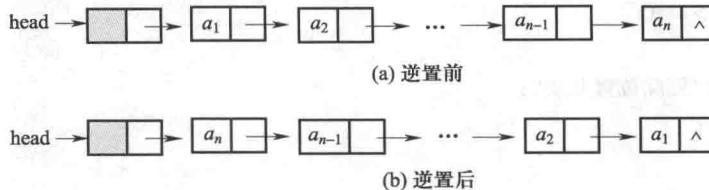


图 2-13 单链表的逆置

假设逆置后的单链表是一个新建的链表，但表中的结点不是新生成的，而是从原单链表（待逆置）中依次删除得到。逆置单链表的算法步骤如下：

- ① 将逆置后的单链表初始化为一空表。
- ② 依次删除原单链表中的结点，并将其插入到逆置后单链表的表头，直至原链表为空。

逆置单链表的具体实现如算法 2-18 所示。

### 算法 2-18 逆置单链表

```
template < class T >
void LinkList < T > :: Invert()
```

```

{
 p = head -> next;
 head -> next = NULL; //将逆置后的单链表初始化为空表
 while(p!=NULL) //遍历原单链表中的结点
 {
 q = p;
 p = p -> next;
 q -> next = head -> next; //将结点插入到逆置后单链表的表头
 head -> next = q;
 }
}

```

### 例 2-2 合并有序单链表

已知递增有序的两个单链表 L1 和 L2，要求将 L2 合并到 L1 中，且结果链表依然保持递增有序。

为了实现合并操作，设置 3 个指针，指针 p1 和 p2 分别指向单链表 L1 和 L2 中等待比较的数据结点，指针 p3 始终指向结果单链表的表尾。

合并有序单链表的算法步骤如下：

① 初始化指针，使指针 p1 指向单链表 L1 的第一个数据结点，p2 指向单链表 L2 的第一个数据结点，指针 p3 指向单链表 L1 的头结点。

② 当 p1 和 p2 都不为空时，如果  $p1 -> data < p2 -> data$ ，则将 p1 所指向的结点链入结果有序单链表的表尾，并将指针 p1 和 p3 后移；否则，将 p2 所指向的结点链入结果有序单链表的表尾，并将指针 p2 和 p3 后移。

③ 如果 p1 不为空，则将 p1 所指向的剩余结点链入结果有序单链表的表尾；如果 p2 不为空，则将 p2 所指向的剩余结点链入结果有序单链表的表尾。

合并有序单链表的具体实现如算法 2-19 所示。在 C++ 中，为了使 Merge 函数中能直接访问 LinkList 的私有成员 head，本书将该函数声明为 LinkList 的友元，定义方法如下所示。

```

template < class T >
class LinkList
{
 ...
public:
 ...
 friend void Merge(LinkList < T > &L1 ,LinkList < T > &L2); //合并有序单链表,声明为友元
 ...
};

```

### 算法 2-19 合并有序单链表

```

void Merge(LinkList &L1 ,LinkList &L2)
{
 p1 = L1. head -> next;
 p2 = L2. head -> next;
 p3 = L1. head;
 while((p1 != NULL) && (p2 != NULL))
 {
 //处理两个表非空时的情况。p1、p2 指向当前需比较的结点,
 //p3 指向结果有序单链表的表尾
 if((p1 -> data) < (p2 -> data))
 {
 p3 -> next = p1;
 p1 = p1 -> next;
 p3 = p3 -> next;
 }
 else
 {
 p3 -> next = p2;
 p2 = p2 -> next;
 p3 = p3 -> next;
 }
 }
 if(p1 != NULL)
 p3 -> next = p1;
 if(p2 != NULL)
 p3 -> next = p2;
 delete L2. head;
 L2. head = NULL;
}

```

### 10. 双向链表的操作

双向链表的求表长度、查找等操作与单链表基本相同，不同的是插入和删除操作，下面分别介绍这两种操作。

#### (1) 双向链表中的插入

如图 2-14 所示，设 p 指向双向链表中某结点，s 指向待插入的值为 x 的新结点，将结点 s 插入到结点 p 的后面。

双向链表中的插入操作步骤如下：

- ①  $s \rightarrow \text{prior} = p;$
- ②  $s \rightarrow \text{next} = p \rightarrow \text{next};$
- ③  $p \rightarrow \text{next} \rightarrow \text{prior} = s;$
- ④  $p \rightarrow \text{next} = s;$

指针操作的顺序不是唯一的，但也不是任意的。操作②和③中都要使用  $p \rightarrow \text{next}$  来找到结点  $p$  的后继，而操作④改变了  $p \rightarrow \text{next}$  的值，因此操作④必须在操作②和③之后进行。

### (2) 双向链表中的删除

设  $p$  指向双向链表中某结点，删除结点的操作示意图如图 2-15 所示。

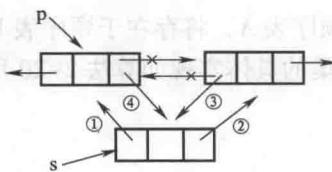


图 2-14 双向链表中的插入操作

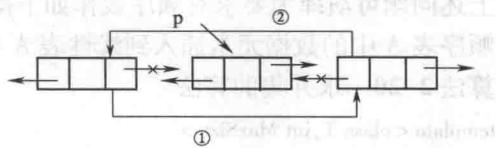


图 2-15 双向链表中的删除操作

双向链表中的删除操作步骤如下：

- ①  $p \rightarrow \text{prior} \rightarrow \text{next} = p \rightarrow \text{next};$
- ②  $p \rightarrow \text{next} \rightarrow \text{prior} = p \rightarrow \text{prior};$
- `delete p;`

请读者思考，上述两个步骤的顺序是否可以颠倒？

## 2.4 线性表的应用

### 1. 求集合的并集

#### (1) 问题描述

假设有两个集合 A 和 B，试编写算法求  $A = A \cup B$ 。

#### (2) 数据结构的设计

可以采用线性表来表示集合，线性表按存储结构分为顺序表和链表两种。本节介绍采用顺序表结构的方法。顺序表结构的定义如下：

```
template < class T, int MaxSize > // 定义类模板 SeqSet
class SeqSet
{
 T data[MaxSize]; // 存放数据元素的数组
```

```

int length; //集合的长度
int Locate(T item); //查找,求线性表中值为 item 的元素序号
void Insert(int i, T item); //在线性表中第 i 个位置插入值为 item 的元素
public:
 SeqSet(); //无参构造函数
 SeqSet(T a[], int n); //有参构造函数
 void Union(SeqSet &set); //求并集
 void Print(); //输出集合各元素
};

```

### (3) 算法的设计

上述问题可演绎为要求对顺序表作如下操作：扩大顺序表 A，将存在于顺序表 B 中而不存在于顺序表 A 中的数据元素插入到线性表 A 中去。求并集的具体实现如算法 2-20 所示。

#### 算法 2-20 求并集的算法

```

template < class T, int MaxSize >
void SeqSet < T, MaxSize > ::Union(SeqSet &set)
{
 int i;
 for(i = 0; i < set.length; i++)
 if(Locate(set.data[i]) == 0)
 Insert(length, set.data[i]);
}

```

在上述算法中利用 Locate() 函数来判断数据元素是否存在于集合中，Locate() 函数的实现见算法 2-5 所示。上述算法还使用了 Insert() 函数将数据元素插入集合中，Insert() 函数的实现见算法 2-7 所示。

从本例看出，可以较方便地使用顺序表的基本操作算法来实现新的功能。

读者可以自行写出求集合的交、差等其他算法。

### 2. 一元多项式相加

#### (1) 问题描述

已知按升幂表示的两个一元多项式  $A(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$  和  $B(x) = b_0 + b_1x + b_2x^2 + \dots + b_mx^m$ ，求  $A(x) + B(x)$ 。

#### (2) 数据结构的设计

对于任意一元多项式  $P(x) = p_0 + p_1x + p_2x^2 + \dots + p_nx^n$ ，可以抽象为一个由“系数 - 指数”对构成的线性表，且线性表中各元素的指数项是递增的，即  $P = ((p_0, 0), (p_1, 1), (p_2, 2), \dots, (p_n, n))$ 。

在多项式相加时,由于所产生的结果多项式的项数和次数都是难以预计的,因此计算机实现时可采用单链表来表示。多项式中的每一项为单链表中的一个结点,每个结点包含3个域:系数域(coef)、指数域(exp)和指针域(next),其形式如图2-16所示。

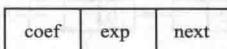


图2-16 一元多项式单链表的结点结构

一元多项式单链表结构的定义如下:

```
struct PolyNode
{
 float coef; //系数域
 int exp; //指数域
 PolyNode * next; //指针域
};
```

只需将2.2.2小节中单链表类模板 template < class T > class LinkList 定义中的 Node换成 PolyNode,就可得到一元多项式单链表类模板的定义。

设有两个一元多项式为  $A(x) = 12 + 3x^2 + 8x^7 + 5x^9$  和  $B(x) = -3x^2 + 10x^7 + 6x^8 + 7x^{12}$ ,它们的链表结构如图2-17所示。



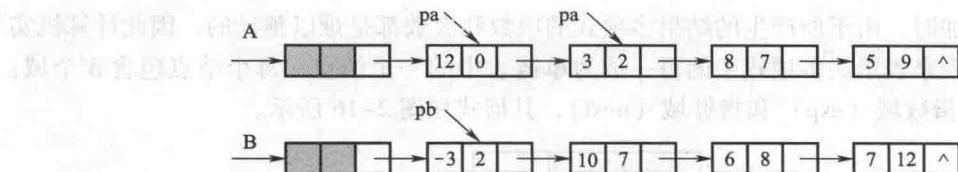
图2-17 一元多项式单链表

### (3) 算法的设计

一元多项式相加的运算规则为:两个多项式中所有指数相同的项的对应系数相加,若和不为零,则构成和多项式中的一项;所有指数不同的项均复制到和多项式中。

利用单链表存放一元多项式后,多项式相加就可以转化为合并两个有序单链表。分别用指针 pa 和 pb 指向两个一元多项式单链表的开始结点。合并过程实际上是对两个单链表中相应结点的指数域进行比较,根据比较结果来决定操作方法,其具体过程如下:

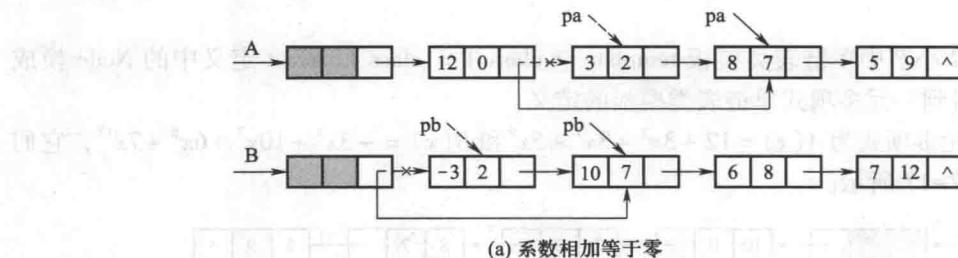
① 如果  $pa \rightarrow exp < pb \rightarrow exp$ ,表明 pa 所指结点应该为结果中的结点,则 pa 指针后移,如图2-18所示。

图 2-18  $pa \rightarrow \exp < pb \rightarrow \exp$  时的示意图

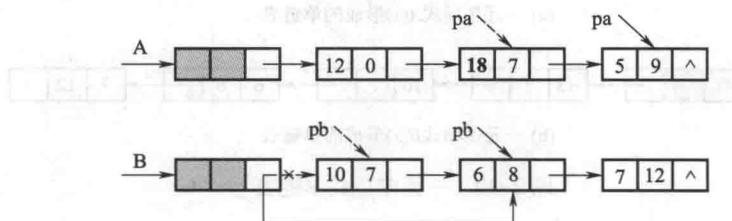
② 如果  $pa \rightarrow \exp = pb \rightarrow \exp$ , 则合并同类项, 即系数相加  $pa \rightarrow \text{coef} = pa \rightarrow \text{coef} + pb \rightarrow \text{coef}$ 。

若相加结果  $pa \rightarrow \text{coef} = 0$ , 结果中不再有系数为  $pa \rightarrow \exp$  的项, 则删除  $pa$  和  $pb$  所指结点, 且  $pa$  和  $pb$  指针均后移, 如图 2-19 (a) 所示。

若相加结果  $pa \rightarrow \text{coef} \neq 0$ , 表明  $pa$  所指结点为结果中的结点, 则删除  $pb$  所指结点, 且  $pa$  和  $pb$  指针均后移, 如图 2-19 (b) 所示。



(a) 系数相加等于零



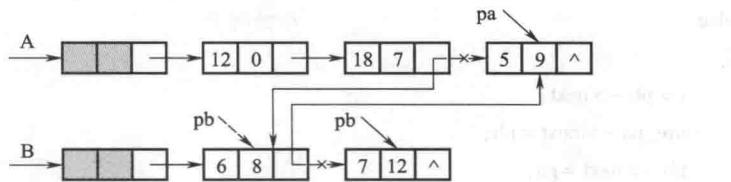
(b) 系数相加不等于零

图 2-19  $pa \rightarrow \exp = pb \rightarrow \exp$  时的示意图

③ 如果  $pa \rightarrow \exp > pb \rightarrow \exp$ , 表明  $pb$  所指结点应该为结果中的结点, 则将  $pb$  所指结点链入单链表 A 中, 且  $pb$  指针后移, 如图 2-20 所示。

此外, 还应当考虑两个一元多项式项数不同的情况。一元多项式相加的具体实现如算法 2-21 所示。

### 算法 2-21 一元多项式相加

图 2-20  $pa \rightarrow \text{exp} > pb \rightarrow \text{exp}$  时的示意图

```

void Add(LinkList < T > &A, LinkList < T > &B)
{
 pa = A. head -> next; pre_pa = A. head; //pa 指向多项式 A 的开始结点
 pb = B. head -> next; pre_pb = B. head; //pb 指向多项式 B 的开始结点
 while(pa&pb)
 {
 if(pa -> exp < pb -> exp) //情况 1
 {
 pre_pa = pa;
 pa = pa -> next;
 }
 else if(pa -> exp == pb -> exp) //情况 2
 {
 pa -> coef = pa -> coef + pb -> coef;
 if(pa -> coef == 0) //系数相加等于 0
 {
 pre_pa -> next = pa -> next; //删除 pa 所指结点
 delete pa;
 pa = pre_pa -> next;
 }
 else //系数相加不等于 0
 {
 pre_pa = pa;
 pa = pa -> next;
 }
 pre_pb -> next = pb -> next; //删除 pb 所指结点
 delete pb;
 pb = pre_pb -> next;
 }
 }
}

```

```

else //情况 3
{
 r = pb->next;
 pre_pa->next = pb;
 pb->next = pa;
 pb = r;
}
if(pb) //处理多项式 B 剩下的项
{
 pre_pa->next = pb;
 delete B.head;
}
}
}

```

## 2.5 顺序表和链表的综合比较

线性表的顺序表和链表两种表示方法各有优劣：顺序表可按序号进行随机访问，但其插入和删除操作由于需要移动表长一半的元素，因此时间复杂度均为 $O(n)$ ；单链表只能从表头开始依次向后扫描，因此无法按序号进行随机访问，但其插入和删除无需移动元素，因此在给出单链表某个结点的指针后，其插入和删除操作的时间复杂度均为 $O(n)$ 。

在实际应用时，选用哪种结构应根据具体问题作具体分析，通常可从以下两个方面考虑。

(1) 能否预先确定线性表的长度 $n$ ，能否在程序执行过程中预先确定 $n$ 的变化范围

由于顺序表需要预分配一定长度的存储空间，因此如果事先不能明确知道线性表的大致长度，则有可能对存储空间预分配得过大，致使在程序执行过程中很大一部分的存储空间得不到充分利用，从而造成浪费。但是，如果估计得太小，那么又将造成频繁地进行存储空间的再分配。链表的显著优点之一就是其存储分配的灵活性。不需要为链表预分配空间，链表中的结点可在程序执行过程中随时应需要动态生成（只要内存尚有可分配的空间）。因此，当线性表的长度变化较大或难以估计最大值时，应当采用链表存储结构；反之，当线性表的长度变化不大，且能事先确定变化的大致范围时，应当采用顺序存储结构。

(2) 对线性表进行哪些主要操作

顺序表是一种随机存储的结构，对顺序表中任一元素进行存取的时间是相同的；而链表是一种顺序存取的结构，对链表中的每个结点都必须从头指针所指结点起顺链扫描。因此，如果线性表需频繁查询却很少进行插入和删除时，宜采用顺序表作存储结构。另外，由于顺序表中以一维数组存储数据元素，数组中第 $i$ 个分量的元素即为线性表中第 $i$ 个数据元素，因此对于那些和位序密切相关的操作采用顺序表则方便多了。

反之，由于在顺序表中进行插入和删除时需要移动近乎表长一半的元素，所以在线性表中元素个数很多，特别是当每个元素占用的空间也较多时，移动元素的时间开销很大。而在链表的任何位置上进行插入或删除时，只需要修改少量指针。因此，若线性表需频繁进行插入或删除操作时，则宜采用链表作存储结构。

## 本章小结

线性表是一种典型的线性结构，它是元素之间约束力最强的一类数据结构。非空线性表的表头元素有唯一后继且无前驱，表尾元素有唯一前驱且无后继，其余各元素均有唯一前驱和后继。

线性表的存储结构分为顺序存储和链式存储，分别称为顺序表和链表。本章分别介绍了线性表在上述两种存储结构基础上的基本操作算法，并比较了这两种存储结构的特点，最后还给出了线性表的应用实例。

本章学习要点如下：

- (1) 线性表的基本概念。
- (2) 顺序表的定义及操作。
- (3) 链表的定义及操作。
- (4) 线性表的应用。

## 习题 2

2.1 线性表可用顺序表或链表存储。试问：

- (1) 两种存储表示各有哪些主要优缺点？
- (2) 如果有  $n$  个表同时并存，并且在处理过程中各表的长度会动态发生变化，表的总数也可能自动改变；那么应选用哪种存储表示？为什么？
- (3) 如果表的总数基本稳定，且很少进行插入和删除，但要求以最快的速度存取表中的元素；那么应采用哪种存储表示？为什么？

2.2 试编写算法，从顺序表中删除具有最小值的元素并由函数返回最小值，空出的位置由最后一个元素填补；若顺序表为空，则显示出错信息并退出运行。

- 2.3 试编写算法，从顺序表中删除具有给定值  $x$  的所有元素。
- 2.4 试编写算法，从有序表中删除其值在给定值  $s$  和  $t$ （要求  $s$  小于  $t$ ）之间的所有元素。
- 2.5 试编写算法，从顺序表中删除所有值重复的元素，使所有元素的值均不同。例如，对于线性表(2, 8, 9, 2, 5, 5, 6, 8, 7, 2)，则执行此算法后变为(2, 8, 9, 5, 6, 7)。注意，表中元素未必是排好序的，且每个值的第一次出现应当保留。

2.6 试编写算法，将元素为整数的顺序表( $a_1, a_2, \dots, a_n$ )重新排列为以 $a_i$ 为界的两部分： $a_i$ 前面的值均比 $a_i$ 小， $a_i$ 后面的值都比 $a_i$ 大，要求算法的时间复杂度为 $O(n)$ 。

2.7 试编写算法，根据一个元素类型为整型的单链表生成两个单链表，使得第一个单链表中包含原单链表中所有元素值为奇数的结点，第二个单链表中包含原单链表中所有元素值为偶数的结点，原有单链表保持不变。

2.8 设表 $L$ 用数组表示，且各元素值递增有序。试写一算法，将元素 $x$ 插入到表 $L$ 的适当位置，使得表中元素仍保持递增有序。

2.9 试编写算法，求循环链表中结点的个数。

2.10 已知一个单链表，试编写复制单链表的算法。

2.11 已知一个无序单链表，表中结点的 data 字段为正整数。试编写算法按递增次序打印表中结点的值。

## 上机实验题 2

### 实验题 2.1 顺序表的编程实现与测试。

(1) 编写 main() 函数对 class SeqList 进行测试，要求使用菜单选择各项功能。

(2) 扩展顺序表 class SeqList 的功能（增加成员函数或友元函数）并进行测试：排序；归并两个有序顺序表。

### 实验题 2.2 用顺序表编程实现一个简易的商品管理系统并完成报告。

商品信息包括商品代码、商品名称、价格、库存量等。对商品库存表的管理包括把它读入到线性表中，对它进行必要的处理，以及把处理后的结果写回到文件中。假设对商品库存表的处理包括以下选项：

(1) 打印（遍历）库存表。

(2) 按商品代码修改记录的当前库存量。若查找到对应的记录，则从键盘上输入其修正量，把它累加到当前库存量域后，再把该记录写回原有位置；若没有查找到对应的记录，则表明是一条新记录，应接着从键盘上输入该记录的商品名称、最低库存量和当前库存量的值，然后把该记录追加到库存表中。

(3) 按商品代码删除指定记录。

(4) 按商品代码对库存表中的记录排序。

(5) main() 函数中使用菜单选择各项功能。

### 实验题 2.3 单链表的编程实现与测试。

(1) 编写 main() 函数对 class LinkList 进行测试，要求使用菜单选择各项功能。

(2) 扩展单链表 class LinkList 的功能（增加成员函数或友元函数）并进行测试：原地逆置；合并两个有序单链表。

### 实验题 2.4 用单链表编程实现一个简易的高校学籍管理系统并完成报告。

(1) 学生信息包括学号、姓名、性别、专业和出生年月等，采用单链表存储方式。

(2) 提供建立、查询、删除、增加和修改等功能。

(3) main() 函数中使用菜单选择各项功能。

# 第3章 栈和队列

从数据结构的角度看，栈和队列也是一种线性表，它们是操作受限的线性表。栈必须按后进先出的规则进行操作，队列必须按先进先出的规则进行操作。

## 3.1 栈

### 3.1.1 栈的基本概念

栈是只能在一端进行插入或删除的线性表。允许插入、删除的一端称为栈顶，另一端称为栈底。当栈中没有元素时称为空栈。

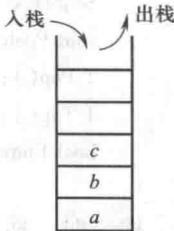
栈有插入和删除两个主要的操作。栈的插入操作常称为入栈（压栈），栈的删除操作常称为出栈（弹栈）。栈的主要特点是后进先出（Last In First Out，LIFO），即出栈元素只能是位于栈顶的元素，而入栈元素也只能放在栈顶位置。如图 3-1 所示，栈中有 3 个元素，进栈的顺序是  $a$ 、 $b$ 、 $c$ ，当需要出栈时其顺序为  $c$ 、 $b$ 、 $a$ 。

在日常生活中有很多后进先出的例子。例如，一叠堆在一起的盘子，要从中取出一只或放入一只，只有从顶部操作比较方便。在程序设计中，图 3-1 栈示意图也常常需要栈这样的数据结构，如编译程序中判断表达式括号匹配、算术表达式求值等问题。

**例 3-1** 假定有 4 个元素 A、B、C、D 按某种次序进栈，试写出所有可能的出栈序列，注意每一个元素进栈后都允许出栈，如 ACDB 就是一种出栈序列。

**解** 可能的出栈序列有：ABCD、ABDC、ACBD、ACDB、ADCB、BACD、BADC、BCAD、BCDA、BDCA、CBAD、CBDA、CDBA、DCBA。

当有  $n$  个元素按照某种顺序压入栈中，且可在任意时刻弹出时，所获得可能的出栈序列个数可用 Catalan 列计算，即  $\frac{1}{n+1} C_{2n}^n$ 。



### 3.1.2 栈的存储结构

由于栈是操作受限的线性表，因此线性表的存储结构对栈也是适用的，只是操作不同。

栈的具体操作的实现取决于栈的存储结构，存储结构不同，其操作算法描述也不同。下面分别介绍栈的顺序存储结构和链接存储结构。

### 1. 栈的顺序存储结构

利用顺序存储方式实现的栈又称为顺序栈。类似于顺序表的定义，栈中的数据元素用一个预设足够长度的一维数组来存储。栈底位置可以设置在数组的任一个端点，本书将栈底位置设在低下标 0 处。由于栈顶位置会随着入栈和出栈而变化，因此可用一个整型变量来表示当前栈顶的位置，通常称该整型变量为栈顶指针。

顺序栈类模板的 C++ 实现如下：

```
template < class T, int MaxSize >
class SeqStack
{
 T data[MaxSize]; //存放栈元素的数组
 int top; //栈顶指针,指示栈顶元素在数组中的下标
public:
 SeqStack(); //构造函数
 void Push(T x); //入栈
 T Pop(); //出栈
 T Top(); //取栈顶元素(元素并不出栈)
 bool Empty(); //判断栈是否为空
};
```

栈空时，栈顶指针  $\text{top} = -1$ ；栈满时，栈顶指针  $\text{top} = \text{MaxSize} - 1$ 。入栈后，栈顶指针加 1；出栈后，栈顶指针减 1。顺序栈的操作如图 3-2 所示。

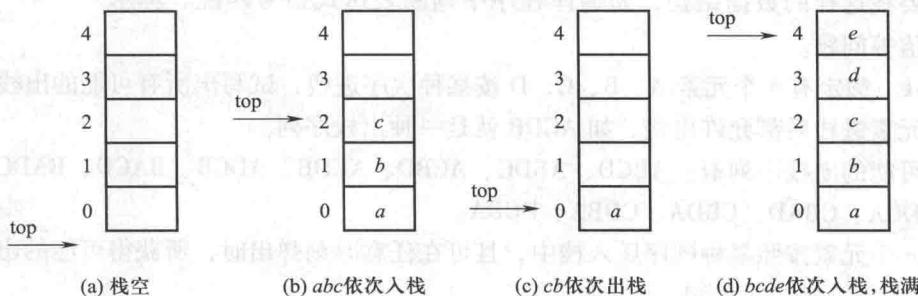


图 3-2 顺序栈的操作

### 2. 栈的链接存储结构

利用链接存储方式实现的栈又称为链栈。链栈中的结点仍可采用在第 2 章中已经定义的

Node 结点类型，用指针 top 指向栈顶元素，链栈通常可表示为如图 3-3 的形式。从图中可以看出，链栈的实质是不带头结点的单链表。

下面给出链栈类模板的 C++ 实现。

```
template < class T >
class LinkStack
{
 Node < T > * top; // 栈顶指针
public:
 LinkStack(); // 构造函数
 ~LinkStack(); // 析构函数
 void Push(T x); // 入栈
 T Pop(); // 出栈
 T Top(); // 取栈顶元素(元素不出栈)
 bool Empty(); // 判断链栈是否为空栈
};
```

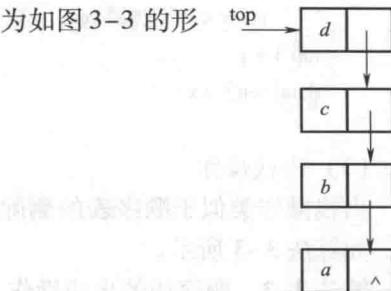


图 3-3 链栈

### 3.1.3 栈的操作算法

#### 1. 顺序栈的操作

由于顺序栈的基本操作是顺序表的简化，因此很容易写出顺序栈的基本操作算法。

##### (1) 初始化

初始化栈就是要构造一个空栈，由构造函数来实现，如算法 3-1 所示。

##### 算法 3-1 顺序栈的初始化

```
template < class T, int MaxSize >
SeqStack < T, MaxSize > :: SeqStack()
{
 top = -1;
}
```

##### (2) 入栈操作

顺序栈的入栈操作类似于顺序表的插入操作。在栈顶插入一个新元素  $x$ ，使  $x$  成为新的栈顶元素，同时栈顶指针需要发生变化，如算法 3-2 所示。

##### 算法 3-2 顺序栈的入栈操作

```
template < class T, int MaxSize >
void SeqStack < T, MaxSize > :: Push(T x)
{
 if(top == MaxSize - 1)
```

```

 { cerr << "上溢"; exit(1); }
 top++;
 data[top] = x;
}

```

### (3) 出栈操作

出栈操作类似于顺序表的删除操作。将栈顶元素从栈中删除，同时栈顶指针需要发生变化，如算法 3-3 所示。

### 算法 3-3 顺序栈的出栈操作

```

template < class T, int MaxSize >
T SeqStack < T, MaxSize > ::Pop()
{
 if(top == -1)
 { cerr << "下溢"; exit(1); }
 x = data[top];
 top--;
 return x;
}

```

### (4) 取栈顶元素

将栈顶元素作为结果返回，栈顶指针不变化，如算法 3-4 所示。

### 算法 3-4 顺序栈的取栈顶元素操作

```

template < class T, int MaxSize >
T SeqStack < T, MaxSize > ::Top()
{
 if(top == -1)
 { cerr << "下溢"; exit(1); }
 return data[top];
}

```

### (5) 判断栈是否为空

判断栈是否为空的具体实现如算法 3-5 所示。

### 算法 3-5 判断顺序栈是否为空

```

template < class T, int MaxSize >
bool SeqStack < T, MaxSize > ::Empty()
{
 return top == -1;
}

```

## 2. 链栈的操作

链栈的基本操作是单链表基本操作的简化。

### (1) 初始化

由构造函数建立一个无头结点的单链表，如算法 3-6 所示。

#### 算法 3-6 链栈初始化

```
template < class T >
LinkStack < T > ::LinkStack()
{
 top = NULL;
}
```

### (2) 入栈操作

入栈类似于在单链表中进行表头插入，如算法 3-7 所示。

#### 算法 3-7 链栈入栈操作

```
template < class T >
void LinkStack < T > ::Push(T x)
{
 s = new Node < T > ;
 s -> data = x; //申请一个数据域为 x 的结点 s
 s -> next = top;
 top = s; //将结点 s 插在栈顶
}
```

### (3) 出栈操作

出栈类似于在单链表中进行表头删除，如算法 3-8 所示。

#### 算法 3-8 链栈出栈操作

```
template < class T >
T LinkStack < T > ::Pop()
{
 if(top == NULL) { cerr << "下溢" ; exit(1); }
 x = top -> data; //暂存栈顶元素
 p = top;
 top = top -> next; //删除栈顶结点
 delete p;
 return x;
}
```

#### (4) 取栈顶元素

返回栈顶指针所指向结点的数据域即可，如算法 3-9 所示。

#### 算法 3-9 链栈中取栈顶元素操作

```
template < class T >
T LinkStack < T > ::Top()
{
 if(top == NULL)
 { cerr << "下溢" ; exit(1); }
 return top -> data;
}
```

#### (5) 判断栈是否为空

判断栈是否为空的具体实现如算法 3-10 所示。

#### 算法 3-10 判断链栈是否为空

```
template < class T >
bool LinkStack < T > ::Empty()
{
 return top == NULL;
}
```

#### (6) 析构函数

链栈的析构函数类似于单链表的析构函数，需要释放链栈所占用的空间，如算法 3-11 所示。

#### 算法 3-11 链栈的析构函数

```
template < class T >
LinkStack < T > ::~LinkStack()
{
 p = top;
 while(p)
 {
 q = p;
 p = p -> next;
 delete q;
 }
 top = NULL;
}
```

### 3.1.4 栈的应用

#### 1. 算术表达式求值

##### (1) 问题描述

输入包含 +、-、\*、/、圆括号和正整数组成的中缀算术表达式，以 '@' 作为表达式结束符，计算该表达式的运算结果。

##### (2) 利用中缀表达式直接求值

在程序设计语言中，操作符位于两个操作数中间的表达式称为中缀表达式。

中缀表达式的计算比较复杂，在计算过程中，既要考虑括号的作用，又要考虑运算符的优先级，还要考虑运算符出现的先后次序。由于各运算符实际的运算次序往往与它们在表达式中出现的先后次序是不一致的，因此在求值时不能简单地进行从左到右运算，必须先算运算级别高的，再算运算级别低的，同一级别的运算才从左到右执行，这种方法称为“算符优先法”。算符间的优先关系如表 3-1 所示。

表 3-1 算符间的优先关系

| 栈顶运算符 pre_op \ 当前运算符 c | + | - | * | / | ( | ) | @ |
|------------------------|---|---|---|---|---|---|---|
| +                      | > | > | < | < | < | > | > |
| -                      | > | > | < | < | < | > | > |
| *                      | > | > | > | > | < | > | > |
| /                      | > | > | > | > | < | > | > |
| (                      | < | < | < | < | < | = |   |
| )                      | > | > | > | > |   | > | > |
| @                      | < | < | < | < | < |   | = |

要实现中缀表达式直接求值，必须设置两个栈：一个栈存放操作符，记做 OPTR；另一个栈存放操作数，记做 OPND。

中缀表达式求值算法步骤如下：

初始时，操作数栈为空，操作符栈中有一个元素 '@'。在进行表达式求值时，从左到右扫描，依次读入表达式中的每个字符，直至表达式结束。

- 若读到的是操作数，则一律进入操作数栈，并读入下一个字符。
- 若读到的是操作符 c，则应用操作符栈的栈顶元素 pre\_op 与之进行比较，会出现以下 3 种情况：

- ① 若  $\text{pre\_op} < c$ , 则将  $c$  入栈, 并读入下一个字符。
- ② 若  $\text{pre\_op} = c$ , 则将  $\text{pre\_op}$  出栈, 并读入下一个字符。
- ③ 若  $\text{pre\_op} > c$ , 则将  $\text{pre\_op}$  出栈, 并在操作数栈中退栈 2 次, 依次得到操作数  $b$  和  $a$ 。然后进行  $a \text{ pre\_op } b$  运算, 并将运算的结果压入操作数栈中。

扫描完毕时, 操作数栈中只有一个元素, 即为运算的结果。

利用算符优先法, 对输入的算术表达式 “ $8/(5-3)@$ ” 求值的操作过程如表 3-2 所示。

表 3-2 算符优先法操作步骤

| 步骤 | 当前读到的字符 | OPTR 栈  | OPND 栈 | 说明                                                                                       |
|----|---------|---------|--------|------------------------------------------------------------------------------------------|
| 1  | 8       | @       |        | 操作数 8 入 OPND 栈                                                                           |
| 2  | /       | @       | 8      | 操作符 '@' < '/' , 操作符 '/' 入 OPTR 栈                                                         |
| 3  | (       | / @     | 8      | 操作符 '/' < '(', 操作符 '(' 入 OPTR 栈                                                          |
| 4  | 5       | ( / @   | 8      | 操作数 5 入 OPND 栈                                                                           |
| 5  | -       | ( / @   | 5 8    | 操作符 '(' < '-' , 操作符 '-' 入 OPTR 栈                                                         |
| 6  | 3       | - ( / @ | 5 8    | 操作数 3 入 OPND 栈                                                                           |
| 7  | )       | - ( / @ | 3 5 8  | 操作符 '-' > ')' , OPTR 栈顶元素 '-' 出栈, 并在操作数栈中退出两个元素 3 和 5。计算 $5 - 3$ 的值, 并将该计算值压入 OPND 栈     |
| 8  | )       | ( / @   | 2 8    | 操作符 '(' = ')' , OPTR 栈顶元素 ')' 出栈                                                         |
| 9  | @       | / @     | 2 8    | 操作符 '/' > '@' , OPTR 栈顶元素 '/' 出栈, 并在操作数栈 OPND 中退出两个元素 2 和 8。计算 $8/2$ 的值, 并将该计算值压入 OPND 栈 |
| 10 | @       | @       | 4      | 操作符 '@' = '@' , OPTR 栈顶元素 '@' 出栈                                                         |

### 算法 3-12 算符优先法求算术表达式的值

```
double Expression_Eval()
```

```

 SeqStack < char, 100 > OPTR; //操作符栈
 SeqStack < double, 100 > OPND; //操作数栈
 OPTR.Push('@');
 ch = getchar();
 while(ch != '@' || OPTR.Top() != '@')
 {
 if(ch >='0' && ch <='9')
 {
 OPND.Push(ch - '0');
 ch = getchar();
 }
 else
 {
 pre_op = OPTR.Top();
 switch(Precede(pre_op, ch))
 {
 case '<': //情况①
 OPTR.Push(ch);
 ch = getchar();
 break;
 case '=': //情况②
 OPTR.Pop();
 ch = getchar();
 break;
 case '>': //情况③
 b = OPND.Pop();
 a = OPND.Pop();
 pre_op = OPTR.Pop();
 OPND.Push(Operate(a, pre_op, b));
 break;
 }
 }
 return OPND.Top();
 }
}

```

注意：算法 3-12 只能对 10 以内的算术表达式求值，其中完成计算的函数 Operate() 和进行算符优先级比较的函数 Precede() 请读者自行完成。

### (3) 利用后缀表达式求值

波兰科学家卢卡谢维奇很早就提出了算术表达式的另一种表示，即后缀表示，又称逆波兰式。后缀是指把操作符放在两个操作数的后面。采用后缀表示的算术表达式被称为后缀算术表达式或后缀表达式。在后缀表达式中不存在括号，也不存在优先级的差别，计算过程完全按照运算符出现的先后次序进行。

#### 例 3-2 将下列各中缀表达式转换为后缀表达式

- ①  $3/5+6$
- ②  $16-9*(4+3)$
- ③  $2*(x+y)/(1-x)$
- ④  $(25+x)*(a*(a+b)+b)$

解 相应的后缀表达式是：

- ①  $3\ 5/6+$
- ②  $16\ 9\ 4\ 3\ +\ *\ -$
- ③  $2\ x\ y\ +\ *\ 1\ x\ -\ /$
- ④  $25\ x\ +\ a\ a\ b\ +\ *\ b\ +\ *$

将中缀表达式变成等价的后缀表达式时，表达式中操作数次序不变，而操作符次序会发生变化，同时需要去掉圆括号。因此，设置一个栈 OPTR 用以存放操作符。

把中缀表达式转换为后缀表达式算法的基本思路是从左到右扫描中缀表达式，依次读入表达式中的每个字符，对于不同类型的字符按不情况进行处理。

- 若读到的是操作数，则输出该操作数，并读入下一个字符。
- 若读到的是左括号，则把它压入到 OPTR 栈中，并读入下一个字符。
- 若读到的是右括号，则表明括号内的中缀表达式已经扫描完毕。将 OPTR 栈从栈顶直到左括号之前的操作符依次出栈并输出，然后将左括号也出栈，并读入下一个字符。
- 若读到的是操作符  $c$ ，则应用操作符栈的栈顶元素 pre\_op 与之进行比较：
  - ① 若  $pre\_op < c$ ，则将  $c$  入栈，并读入下一个字符；
  - ② 若  $pre\_op \geq c$ ，则将  $pre\_op$  出栈并输出。

按照以上过程扫描到中缀表达式结束符@时，把栈中剩余的操作符依次出栈并输出，就得到了转换成的后缀表达式。

例 3-3 用一个操作符栈来模拟将输入的中缀算术表达式 “ $8/(5-3)@$ ” 转换成立后缀表达式的过程。

解 操作过程如表 3-3 所示。

表 3-3 中缀表达式转换成后缀表达式的操作步骤

| 步骤 | 当前读到的字符 | OPTR 栈  | 输出        | 说明                                                  |
|----|---------|---------|-----------|-----------------------------------------------------|
| 1  | 8       | @       | 8         | 操作数 8 输出                                            |
| 2  | /       | @       | 8         | 操作符 '@' < '/'，操作符 '/' 入 OPTR 栈                      |
| 3  | (       | / @     | 8         | 操作符 '(' 入 OPTR 栈                                    |
| 4  | 5       | ( / @   | 8 5       | 操作数 5 输出                                            |
| 5  | -       | ( / @   | 8 5       | 操作符 '(' < '-'，操作符 '-' 入 OPTR 栈                      |
| 6  | 3       | - ( / @ | 8 5 3     | 操作数 3 输出                                            |
| 7  | )       | ( / @   | 8 5 3 -   | 读到的是操作符 ')'，将 OPTR 栈从栈顶直到左括号之前的操作符依次出栈并输出，然后将左括号也出栈 |
| 8  | @       | / @     | 8 5 3 - / | 扫描到中缀表达式结束符 '@'，把栈中剩余的操作符依次出栈并输出                    |

将中缀表达式转换成等价的后缀表达式求值时，不需要再考虑操作符的优先级，只需从左到右扫描一遍后缀表达式即可。可设置一个栈 OPND 用以存放操作数。

后缀表达式求值算法的基本思路是从左到右扫描，依次读入表达式中的每个字符，直至表达式结束。

- 若读到的是操作数，则入栈 OPND。
- 若读到的是操作符，则在 OPND 栈中退出两个元素，先退出的在操作符右，后退出的在操作符左，然后用该操作符进行运算，并将运算的结果压入 OPND 栈中。

后缀表达式扫描完毕时，OPND 栈中仅有一个元素，即为运算的结果。

**例 3-4** 用一个操作数栈来模拟例 3-3 所得的后缀算术表达式“8 5 3 - / @”的求值过程。

解 操作过程如表 3-4 所示。

表 3-4 后缀表达式求值

| 步骤 | 当前读到的字符 | OPND 栈      | 说明                                                  |
|----|---------|-------------|-----------------------------------------------------|
| 1  | 8       |             | 操作数 8 入栈                                            |
| 2  | 5       | 8           | 操作数 5 入栈                                            |
| 3  | 3       | 5<br>8      | 操作数 3 入栈                                            |
| 4  | -       | 3<br>5<br>8 | 在栈 OPND 中退出两个元素 3 和 5，计算 $5 - 3$ 的值，并将该计算值压入 OPND 栈 |
| 5  | /       | 2<br>8      | 在栈 OPND 中退出两个元素 2 和 8，计算 $8 / 2$ 的值，并将该计算值压入 OPND 栈 |
| 6  | @       | 4           | 结果输出                                                |

## 2. 栈与递归

### (1) 递归的含义

递归是指函数直接调用自己或通过一系列调用语句间接调用自己，该函数称为递归函数。递归是程序设计的有效方法之一，可用递归方法求解的问题必须同时具备以下两个条件：

① 一个问题可以转化为若干个性质相同、解法相同的小问题，而小问题还可分解为更小的问题……上述转化具有相同的规律，并使问题逐步简化。

② 存在明确的递归出口，即递归的终止条件。当问题规模降低到一定程度时，可以直接求解。

根据上述条件，适合用递归方法求解的问题有：

① 数学上定义为递归的函数。例如，求整型数  $n$  的阶乘的定义是

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

还有很多递归性质的函数，如 Fibonacci 级数、Ackerman 函数等。

② 数据的结构是递归的。例如，本书第 6 章将要介绍的广义表，其元素也可以是一个子表，而子表也是表；第 7 章将要介绍的树形结构，每个结点可以有 0 至多个子树，而子树又是一棵树。

③ 解题的方式用递归解法比用递推解法更为简单。例如，汉诺塔问题、八皇后问题等。

递归算法的设计步骤主要包含以下 3 步：

- ① 分析原问题  $f(x)$ ，假设出合理的较小问题  $f(x')$ 。
- ② 假定  $f(x')$  是可解的，在此基础上确定  $f(x)$  的解，即找出  $f(x)$  与  $f(x')$  之间的关系。注意，按这种关系分解下去能使问题得到一个简单的基本解，以此作为递归出口。

③ 将分析得到的递归模型转换成用相应的算法语言描述的函数。

### (2) 栈和递归

在用高级语言编写的程序中，对于函数的调用，系统是通过栈来实现的。一个递归函数的执行过程类似于多个函数的嵌套调用，因此在执行递归函数的过程中也需要一个递归工作栈。它的作用如下：

- ① 将递归调用时的实际参数和函数返回地址传递给下一层的递归函数。
- ② 保存本层的参数和局部变量，以便从下一层返回时重新使用它们。

算法 3-13 给出了求整型数  $n$  的阶乘的递归设计。

### 算法 3-13 求阶乘的递归算法

```
int Fact(int n)
{
 if(n == 0) return 1;
 else return n * Fact(n - 1);
}
```

图 3-4 所示的是  $n=3$  时求阶乘递归算法的运行轨迹。

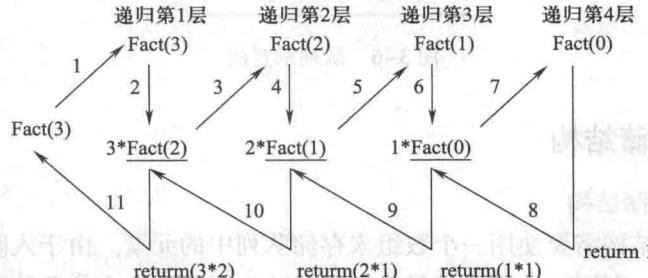


图 3-4 求阶乘递归算法的运行轨迹

图 3-5 所示的是  $n=3$  时求阶乘递归算法运行时的工作栈中的状况。

|         | 参数 | 返回地址         |
|---------|----|--------------|
| Fact(0) | 0  | 3            |
| Fact(1) | 1  | 4            |
| Fact(2) | 2  | 4            |
| Fact(3) | 3  | 调用 Fact(3) 处 |

图 3-5 递归工作栈示意图

## 3.2 队列

### 3.2.1 队列的基本概念

在日常生活中经常会遇到需要排队的情况，在软件设计中也经常出现类似情况，如操作系统中的打印作业的处理问题就需要使用队列这一数据结构。

队列是一种在一端进行插入，而在另一端进行删除的线性表。允许插入的一端称为队尾，允许删除的一端称为队头。当队列中没有元素时称为空队列。

队列有两个主要的操作：插入和删除。队列的插入操作常称为入队，队列的删除操作常称为出队。队列的主要特点是先进先出（First In First Out，FIFO），即出队元素只能是位于队头的元素，而入队元素也只能放在队尾位置。图 3-6 所示为有 5 个元素的队列。入队的顺序依次为  $a_1$ 、 $a_2$ 、 $a_3$ 、 $a_4$ 、 $a_5$ ，出队时的顺序将依然是  $a_1$ 、 $a_2$ 、 $a_3$ 、 $a_4$ 、 $a_5$ 。

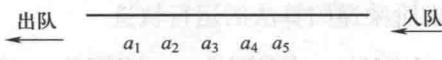


图 3-6 队列示意图

### 3.2.2 队列的存储结构

#### 1. 队列的顺序存储结构

队列的顺序存储结构需要使用一个数组来存储队列中的元素。由于入队和出队分别在两端进行，为了简化操作，利用两个整形变量分别存放队头元素前一个位置的下标和队尾元素的下标，通常称这两个整型变量为队头指针和队尾指针。

```
template < class T, int MaxSize > // 定义类模板 SeqQueue
class SeqQueue
{
 T data[MaxSize]; // 存放队列元素的数组
 int front, rear; // 队头和队尾指针
public:
 SeqQueue(); // 构造函数,置空队
 void EnQueue(T x); // 将元素 x 入队
 T DeQueue(); // 将队头元素出队
 T GetQueue(); // 取队头元素(并不删除)
```

```

 bool Empty();
} //判断队列是否为空

```

本书将数组下标 0 端设为队头，入队操作时可以先使队尾指针后移一个位置， $\text{rear} = \text{rear} + 1$ ，再向该位置写入新元素；出队操作时队头指针后移一个位置， $\text{front} = \text{front} + 1$ 。顺序队列的操作如图 3-7 所示。

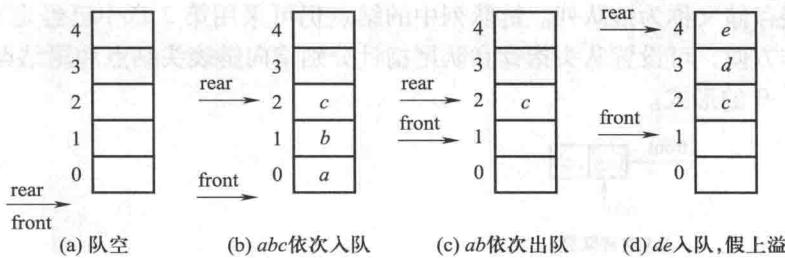


图 3-7 顺序队列的操作

从图 3-7 中可以看出，如果还有元素需要入队，就会出现假上溢现象。解决假上溢的方法是将存储数据元素的一维数组看成是头尾相接的循环结构，即循环队列。在循环队列中，入队和出队操作中队头指针和队尾指针不是直接加 1，而是采用加 1 取模的方式。入队操作时  $\text{rear} = (\text{rear} + 1) \% \text{MaxSize}$ ，出队操作时  $\text{front} = (\text{front} + 1) \% \text{MaxSize}$ 。

在循环队列中如何判断队列满和队列空呢？如图 3-8 所示，由于队列为空和为满的条件均为  $\text{front} == \text{rear}$ ，因此会产生歧义。消除歧义的方法如下：

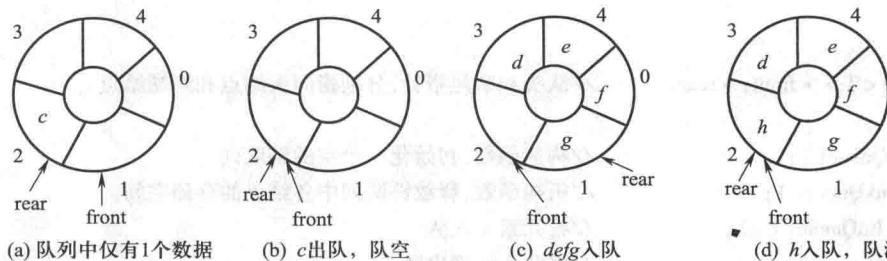


图 3-8 循环队列队空和队满的判定

(1) 浪费一个元素空间。将图 3-8 (c) 所示的情况视为队满，此时的状态是队尾指针加 1 就会从后面赶上队头指针。在这种情况下，队满的条件是  $(\text{rear} + 1) \% \text{MaxSize} == \text{front}$ ，这样就能与空队区别开。

(2) 设置一个辅助标志变量 `flag`。例如，当  $\text{front} == \text{rear}$  且 `flag == false` 时（此时刚有元素出队列）表示队空；当  $\text{front} == \text{rear}$  且 `flag == true` 时（此时刚有元素进队列）表示队满。请读

者思考仅用标志变量 flag 如何判断队列状态。

(3) 使用一个计数器记录队列中元素的个数。附设一个存储队中元素个数的变量如 num。当 num == 0 时表示队空, 当 num == MaxSize 时表示队满。

本书采用第一种方法, 其他方法的实现留给读者完成, 参见习题 3.6 和习题 3.7。

## 2. 队列的链接存储结构

队列的链接存储又称为链队列。链队列中的结点仍可采用第 2 章中已经定义的 Node 结点类型。为了操作方便, 可设置队头指针和队尾指针分别指向链表头结点和尾结点。链队列通常可表示为如图 3-9 的形式。



图 3-9 链队列

链队列类模板的 C++ 实现如下:

```
template < class T >
class LinkQueue
{
private:
 Node < T > * front, * rear; //队头和队尾指针, 分别指向头结点和终端结点
public:
 LinkQueue(); //构造函数, 初始化一个空的链队列
 ~LinkQueue(); //析构函数, 释放链队列中各结点的存储空间
 void EnQueue(T x); //将元素 x 入队
 T DeQueue(); //将队头元素出队
 T GetQueue(); //取链队列的队头元素
 bool Empty(); //判断链队列是否为空
};
```

## 3.2.3 队列的操作算法

### 1. 循环队列的操作

#### (1) 循环队列的初始化

**算法 3-14 循环队列的初始化**

```
template < class T, int MaxSize >
SeqQueue < T, MaxSize > :: SeqQueue()
{
 front = rear = 0;
}
```

**(2) 循环队列的入队****算法 3-15 循环队列的入队**

```
template < class T, int MaxSize >
void SeqQueue < T, MaxSize > :: EnQueue(T x)
{
 if((rear + 1) % MaxSize == front)
 { cerr << "上溢" ; exit(1); }
 rear = (rear + 1) % MaxSize; //队尾指针在循环意义下加1
 data[rear] = x; //在队尾处插入元素
}
```

**(3) 循环队列的出队****算法 3-16 循环队列的出队**

```
template < class T, int MaxSize >
T SeqQueue < T, MaxSize > :: DeQueue()
{
 if(rear == front)
 { cerr << "下溢" ; exit(1); }
 front = (front + 1) % MaxSize; //队头指针在循环意义下加1
 return data[front]; //读取并返回出队前的队头元素
}
```

请读者仿照此算法，给出取队头元素的算法。

**(4) 判断循环队列是否为空****算法 3-17 判断循环队列是否为空**

```
template < class T, int MaxSize >
bool SeqQueue < T, MaxSize > :: Empty()
{
 return rear == front;
}
```

## 2. 链队列的操作

### (1) 初始化链队列

利用构造函数构造一个带头结点的单链表，并将队头和队尾指针均指向头结点，如算法 3-18 所示。

#### 算法 3-18 链队列的初始化

```
template < class T >
LinkQueue < T > :: LinkQueue()
{
 s = new Node < T > ;
 s -> next = NULL;
 front = rear = s;
}
```

### (2) 链队列入队

链队列的入队操作步骤如下：

- ① 产生一个数据域为 x 的新结点 s；
- ② 将结点 s 插入队尾；
- ③ 将队尾指针指向结点 s。

链队列入队操作的具体实现如算法 3-19 所示。

#### 算法 3-19 链队列的入队

```
template < class T >
void LinkQueue < T > :: EnQueue(T x)
{
 s = new Node < T > ; //产生一个数据域为 x 的新结点 s
 s -> data = x;
 s -> next = NULL;
 rear -> next = s; //将结点 s 插入到队尾
 rear = s; //将队尾指针指向结点 s
}
```

### (3) 链队列出队

链队列的出队操作步骤如下：

- ① 如果队空，则抛出下溢异常；
- ② 暂存队头元素；
- ③ 删除队头元素所在结点；
- ④ 如果被删除的队头元素同时也是队尾元素，则修改队尾指针。

链队列出队操作的具体实现如算法 3-20 所示。

### 算法 3-20 链队列的出队

```
template < class T >
T LinkQueue < T > :: DeQueue()
{
 if(rear == front)
 { cerr << "下溢" ; exit(1); }
 p = front -> next;
 x = p -> data;
 front -> next = p -> next;
 if(p -> next == NULL)
 rear = front;
 delete p;
 return x;
}
```

读者可仿照单链表的析构函数给出链队列的析构函数。

## 3.2.4 队列的应用

本节利用报数问题来介绍队列的应用。

### (1) 问题描述

设有  $n$  个人排成一列，从前往后进行“0, 1”报数。凡是报“0”的人出列，凡是报“1”的人立即站到队伍的最后。反复进行报数，直到所有人均出列为止。要求给出这  $n$  个人的出列顺序。

例如， $n=5$  时初始序列为 1、2、3、4、5，出队序列为 1、3、5、4、2。

### (2) 数据结构的设计

可将  $n$  个人排成的队伍用队列进行模拟。这里采用链队列的存储结构。

### (3) 算法的设计

该问题实质是一个反复出队和入队的问题，即报“0”的人出队，报“1”的人入队，直至队列为空。算法的基本思想如下：

反复执行以下步骤，直至队列为空。

① 将队头元素出队，并输出其编号。

② 若队列不空，则再出队一个元素，并将该元素再次入队。

队列求解报数问题的具体实现由算法 3-21 给出。

### 算法 3-21 队列求解报数问题

```

void Number()
{
 LinkQueue < int > linkq;
 for(i = 1; i <= n; i ++) // 初始化队列, 让 n 个人入队
 linkq. EnQueue(i);
 while(!linkq. Empty())
 {
 x = linkq. DeQueue(); // 报"0" 的人出列
 cout << x;
 if(!linkq. Empty()) // 报"1" 的人到队伍最后
 {
 y = linkq. DeQueue();
 linkq. EnQueue(y);
 }
 }
}

```

## 本章小结

从数据结构的角度来看，栈和队列属于线性结构，是操作受限的线性表，其操作是线性表操作的子集。栈的特点是后进先出，队列的特点是先进先出。

本章介绍了栈和队列的定义以及在两种存储结构上的操作，并给出了栈在表达式求值以及递归函数中的应用和队列的应用。

本章学习要点如下：

- (1) 理解栈和队列的特点及它们的差异。
- (2) 掌握顺序栈和链栈的定义及操作。
- (3) 掌握循环队列和链队列的定义及操作。
- (4) 掌握栈和队列的应用。

## 习题 3

3.1 编号为 1、2、3、4、5 的 5 辆列车顺序开进一个栈式结构的站点，问开出车站的顺序有多少种可能？请具体写出所有可能的出栈序列。

3.2 利用栈实现把十进制整数转换为二进制至十六进制之间的任一进制数并输出的功能。

3.3 设有一维数组 stack[StackMaxSize]，将其分配给两个栈 S1 和 S2 使用。试问如何分配数组空间，使

得对任何一个栈，当且仅当数组空间全满时才不能插入？试分别给出两个栈的入栈和出栈算法。

3.4 假设表达式中允许包含 3 种括号：圆括号、方括号和大括号。试编写一个算法，检查表达式中括号是否配对。若能够全部配对则返回 1，否则返回 0。

3.5 现有中缀表达式  $E = ((A - B) / C + D * (E - F)) * G$

(1) 写出与 E 等价的后缀表达式。

(2) 用一个操作符栈来模拟表达式的转换过程，画出在将 E 转换成分缀表达式的过程中栈内容的变化图。

(3) 用一个操作数栈来模拟后缀表达式的求值过程，画出对 (2) 中所得到的后缀表达式求值时栈中内容的变化图。

3.6 假设以一维数组  $data[m]$  存储循环队列的元素。若要使这  $m$  个分量都得到应用，则另设一辅助标志变量 flag 判断队列的状态为“空”还是“满”。试编写入队和出队算法。

3.7 假设以一维数组  $data[m]$  存放循环队列的元素，同时设变量 num 表示当前队列中元素的个数，以判断队列的状态为“空”还是“满”。试给出此循环队列满的条件，并编写入队和出队算法。

3.8 假设以带头结点的循环链表表示队列，并且只设一个表尾指针。试编写相应的置队列空、入队和出队操作。

3.9 如何用两个栈来实现队列？请写出队列基本操作的算法。

## 上机实验题 3

### 实验题 3.1 顺序栈的实现与应用。

(1) 编写 main() 函数对 class SeqStack 进行测试，要求使用菜单选择各项功能。

(2) 利用顺序栈，采用算符优先算法编程实现直接计算中缀表达式的值，要求输入中缀算术表达式，计算表达式的值。

(3) 利用顺序栈编程实现先将中缀表达式转换成分缀表达式，再计算分缀表达式的值。

### 实验题 3.2 队列的实现与应用。

(1) 编写 main() 函数对 class SeqQueue 进行测试，要求使用菜单选择各项功能。

(2) 编写一个程序模拟患者在医院等待就诊的情况，主要模拟下面两种场景：

- 患者到达诊室，将病历交给护士，排到等待队列中候诊；
- 护士从等待队列中取出下一位患者的病历，该患者进入诊室就诊。

程序采用菜单方式，其选项及功能说明如下。

1) 排队：输入排队患者的病历号（随机产生），加入到就诊患者排队队列中；

2) 就诊：患者队列中最前面的病人就诊，并将其从队列中删除；

3) 查看：从队首到队尾列出所有排队患者的病历号；

4) 下班：退出运行。

## 第4章 串

串（即字符串）是一种特殊的线性表，它的数据元素是字符。串是计算机非数值处理的主要对象之一。例如，在汇编和高级语言的编译程序中，源程序和目标程序都是字符串数据；信息检索系统、文字编辑系统都是以字符串数据作为对象的。串具有自身的特性，常常把一个串作为一个整体来处理。因此，本章把串作为独立结构的概念加以研究，介绍串的存储结构及基本操作。

### 4.1 串的基本概念

串是由零个或多个任意字符组成的有限序列，一般记作

$$S = "a_1 \ a_2 \cdots \ a_n"$$

其中， $S$  是串名；在本书中，用双引号作为串的定界符，引号引起的字符序列为串值，引号本身不属于串的内容； $a_i (1 \leq i \leq n)$  是一个任意字符，称为串的元素，是构成串的基本单位， $i$  是它在整个串中的序号； $n$  为串的长度，表示串中所包含的字符个数，当  $n=0$  时称为空串。

下面是几个串的例子：

|                              |           |
|------------------------------|-----------|
| $S_1 = "data \ structure"$ , | 长度为 14 的串 |
| $S_2 = "struct"$ ,           | 长度为 6 的串  |
| $S_3 = ""$ ,                 | 空串，长度为 0  |
| $S_4 = " "$ ,                | 空格串，长度为 1 |

如果两个串的长度相等且对应字符都相等，则称两个串是相等的。

串中任意连续的字符组成的子序列称为该串的子串。包含子串的串相应地称为主串。子串的第一个字符在主串中的序号称为子串的位置。

### 4.2 串的存储结构

#### 4.2.1 串的顺序存储结构

串的顺序存储结构是指用固定长度的数组来存储串中的字符序列。

在串的顺序存储中，一般有 3 种方法表示串的长度。

(1) 用一个变量来表示串的长度, 如图 4-1 所示。

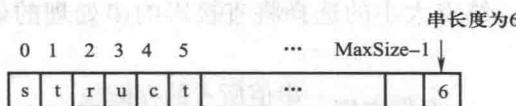


图 4-1 串的顺序存储方式 1

(2) 在串尾存储一个不会在串中出现的特殊字符作为串的终结符, 如图 4-2 所示。

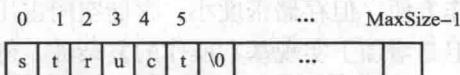


图 4-2 串的顺序存储方式 2

(3) 用数组的 0 号单元存放串的长度, 串值从 1 号单元开始存放, 如图 4-3 所示。

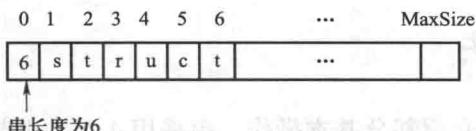


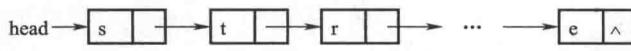
图 4-3 串的顺序存储方式 3

## 4.2.2 串的链式存储结构

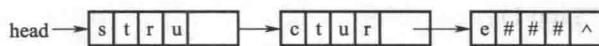
串的链式存储结构有以下两种形式。

### 1. 非压缩形式

如图 4-4 (a) 所示, 在非压缩形式中, 一个结点只存储一个字符, 其优点是操作方便, 但存储利用率低。



(a) 非压缩形式



(b) 压缩形式

图 4-4 串的链接存储示意图

### 2. 压缩形式

为了提高存储空间利用率, 在压缩形式中, 一个结点可存储多个字符, 如图 4-4 (b) 所

示。这实质上是一种顺序与链接相结合的结构。

以链式结构存储串时，结点大小的选择将直接影响串处理的效率。定义串的存储密度如下：

$$\text{存储密度} = \frac{\text{串值所占的存储位}}{\text{实际分配的存储位}}$$

如图 4-4 所示，若设一个字符占 1 个字节，一个指针占 4 个字节，则非压缩形式的存储密度为  $1/5$ ，压缩形式的存储密度为  $1/2$ 。

显然，非压缩形式的操作方便，但存储密度小，存储空间占用量大。一般情况下，以压缩形式存储时，存储密度大，但是增加了实现基本操作的复杂性，例如在串中插入字符时，就可能需要分割结点。在实际应用时，可将串的链式存储和顺序存储结合使用。例如，在文本编辑系统中，整个文本可以看成是一个串，每行是一个子串；可以将一行的串用顺序结构存储，而行与行之间用指针链接。

## 4.3 串的操作算法

串的操作有很多，本节介绍部分基本操作，串采用 4.2.1 小节介绍的第二种顺序存储方式，即在串尾存储一个不会在串中出现的特殊字符作为串的终结符。

### 4.3.1 串的基本操作算法

#### 1. 串连接

在串  $s1$  的后面连接串  $s2$ ， $s1$  改变， $s2$  不改变。串连接操作的具体实现如算法 4-1 所示。

#### 算法 4-1 串连接

```
void StrCat (char * s1 , char * s2)
```

```
{
```

```
 len1 = strlen (s1);
 len2 = strlen (s2);
 if (len1 + len2 > MaxSize - 1)
 { cerr << "超长" ; exit (1);
 i = 0;
 while (s2 [i] != '\0')
 {
 s1 [i + len1] = s2 [i];
 i++;
 }
}
```



无数据图例

无数据图例

s1[i + len1] = '\0'; 完成操作后字符串 s1 的值为 "deedede\0"，即表示子串 "deedede" 在字符串 s1 中定位成功。如果将字符串 s1 的值设为 0，则表示操作失败。

## 2. 串比较

如果  $s1 > s2$ ，则返回任一大于 0 的整数；如果  $s1 < s2$ ，则返回任一小于 0 的整数；如果  $s1 == s2$ ，则返回 0。串比较操作的具体实现如算法 4-2 所示。

### 算法 4-2 串比较

```
int StrCmp(char * s1, char * s2)
{
 int i = 0;
 while (s1[i] == s2[i] && s1[i] != '\0')
 i++;
 return (s1[i] - s2[i]);
}
```

## 3. 串拷贝

将  $s2$  的串值赋值给  $s1$ ， $s1$  原来的值被覆盖掉， $s2$  的值不变。串拷贝操作的具体实现如算法 4-3 所示。

### 算法 4-3 串拷贝

```
void StrCpy(char * s1, char * s2)
{
 int len = strlen(s2);
 if (len > MaxSize - 1)
 cerr << "超长" ; exit(1);
 while (*s1++ = *s2++);
}
```

## 4.3.2 串的模式匹配

串的模式匹配就是子串定位操作，它是一种重要的串运算。给定两个串  $s = "s_0 s_1 \dots s_{n-1}"$  和  $t = "t_0 t_1 \dots t_{m-1}"$ （其中  $n$  和  $m$  分别是串  $s$  和  $t$  的长度），在主串  $s$  中寻找子串  $t$  的过程称为模式匹配， $t$  称为模式。如果在  $s$  中找到等于  $t$  的子串，则称匹配成功，返回  $t$  在  $s$  中的首次出现的下标位置；否则匹配失败，返回 -1。

### 1. 简单的模式匹配算法

这是一种有回溯的模式匹配算法，称为 Brute-Force 算法，简称 BF 算法。算法的基本思想如下：从主串  $s$  中下标为 0 的字符开始，与模式串  $t$  中下标为 0 的字符比较。若相同，则继续逐个比较  $s$  和  $t$  中的后续字符；若不同，从主串  $s$  中下标为 1 的字符开始，与模式串  $t$  中下

标为 0 的字符比较。依此类推，重复上述过程，若  $t$  中字符全部比完，则说明匹配成功；否则说明匹配失败。

**例 4-1** 设主串  $s = "ababcabcacb"$ ，模式  $t = "abcac"$ ，匹配过程如图 4-5 所示。

下面给出算法的具体实现。

#### 算法 4-4 BF 算法

```
int BFmatching(char * s, char * t)
{
 i = 0;
 j = 0;
 n = strlen(s);
 m = strlen(t);
 while(i < n && j < m)
 {
 if(s[i] == t[j])
 {
 i++;
 j++;
 }
 else
 {
 i = i - j + 1;
 j = 0;
 }
 }
 if(j >= m)
 return i - j; // 匹配成功, 返回子串在主串中首次出现的下标位置
 else
 return -1; // 匹配不成功, 返回 -1
}
```



图 4-5 简单的模式匹配

下面分析该算法的时间复杂度。设串  $s$  长度为  $n$ ，串  $t$  长度为  $m$ 。匹配成功的情况下，考虑两种极端情况。

在最好情况下，每趟不成功的匹配都发生在第一对字符比较时。例如， $s = "aaaaabcd"$ ， $t = "bcd"$ 。设匹配成功发生在  $s_i$  处，则在前  $i$  趟比较中，匹配均不成功。因为每趟不成功的匹配都发生在第一对字符比较时，所以前面  $i$  趟匹配中共比较了  $i$  次，第  $i+1$  趟成功的匹配共比较了  $m$  次。因此，总共比较了  $i+m$  次，所有匹配成功的可能共有  $n-m+1$

种。设从  $s_i$  开始与  $t$  串匹配成功的概率为  $p_i$ ，在等概率情况下  $p_i = 1/(n - m + 1)$ ，平均比较的次数为

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

因此，最好情况下的时间复杂度是  $O(n + m)$ 。

在最坏情况下，每趟不成功的匹配都发生在  $t$  的最后一个字符。例如， $s = "aaaaaab"$ ， $t = "aaab"$ 。设匹配成功发生在  $s_i$  处，因为在前面  $i$  趟匹配中共比较了  $i \times m$  次，第  $i + 1$  趟成功的匹配共比较了  $m$  次，所以总共比较了  $(i + 1) \times m$  次。因此，平均比较的次数为

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

一般情况下，由于  $m \ll n$ ，因此最坏情况下的时间复杂度是  $O(n * m)$ 。

## 2. KMP 模式匹配算法

BF 算法尽管简单但效率较低，D. E. Knuth、J. H. Morris 和 V. R. Pratt 共同设计了一种对 BF 算法做了很大改进的模式匹配算法，简称为 KMP 算法。该算法的改进之处在于取消了主串的回溯，从而使算法效率有所提高。

### (1) KMP 算法的思想

分析 BF 算法的执行过程可知，造成 BF 算法速度慢的原因是回溯，即在某趟的匹配过程失败后，对于  $s$  串要回到本趟开始字符的下一个字符， $t$  串要回到首字符。但是，这些回溯并不是必要的。

对于如图 4-5 所示的匹配过程，在第三趟匹配过程中， $s_2 \sim s_5$  和  $t_0 \sim t_3$  是匹配成功的， $s_6 \neq t_4$  匹配失败，因此有了第四趟，其实这一趟是不必要的。因为在第三趟中有  $s_3 = t_1$ ，而  $t_0 \neq t_1$ ，因此肯定有  $t_0 \neq s_3$ 。同样第五趟也是没有必要的，因此，从第三趟之后可以直达到第六趟。进一步分析第六趟中的第一对字符  $s_5$  和  $t_0$  的比较也是多余的，因为第三趟中已经比过了  $s_5$  和  $t_3$ ，并且  $s_5 = t_3$ ，而  $t_0 = t_3$ ，必有  $s_5 = t_0$ ，因此第六趟的比较可以从第二对字符  $s_6$  和  $t_1$  开始进行。也就是说，第三趟匹配失败后，指针  $i$  不动，而是将模式串  $t$  向右滑动，用  $t_1$  “对准”  $s_6$  继续进行匹配，依此类推。这样的处理方法可以使指针  $i$  是无回溯的。

综上所述，希望某趟在  $s_i$  和  $t_j$  匹配失败后，指针  $i$  不回溯，模式  $t$  向右滑动至某个位置  $k$  上，使得  $t_k$  对准  $s_i$  继续向右进行。显然，现在问题的关键是将串  $t$  滑动到哪个位置上。不妨设位置为  $k$ ，即  $s_i$  和  $t_j$  匹配失败后，指针  $i$  不动，模式  $t$  向右滑动，使  $t_k$  和  $s_i$  对准继续向右进行比较，如图 4-6 所示。

要满足这一假设，就要有如下关系成立：

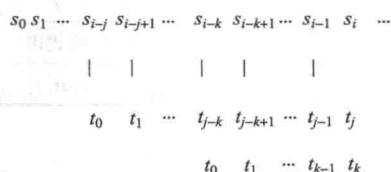


图 4-6 主串和模式串匹配的一般情况

$$t_0 t_1 \cdots t_{k-1} = s_{i-k} s_{i-k+1} \cdots s_{i-1} \quad (4-1)$$

式(4-1)左边是 $t_k$ 前面的 $k$ 个字符,右边是 $s_i$ 前面的 $k$ 个字符。

本趟匹配失败是在 $s_i$ 和 $t_j$ 之处,已经得到的部分匹配结果是

$$t_0 t_1 \cdots t_{j-1} = s_{i-j} s_{i-j+1} \cdots s_{i-1} \quad (4-2)$$

因为 $k < j$ ,所以有

$$t_{j-k} t_{j-k+1} \cdots t_{j-1} = s_{i-k} s_{i-k+1} \cdots s_{i-1} \quad (4-3)$$

式(4-3)左边是 $t_j$ 前面的 $k$ 个字符,右边是 $s_i$ 前面的 $k$ 个字符。通过式(4-1)和式(4-3)得到关系

$$t_0 t_1 \cdots t_{k-1} = t_{j-k} t_{j-k+1} \cdots t_{j-1} \quad (4-4)$$

因此,某趟在 $s_i$ 和 $t_j$ 匹配失败后,如果模式串中有满足式(4-4)的子串存在,即模式中的前 $k$ 个字符与模式中 $t_j$ 字符前面的 $k$ 个字符相等时,那么模式 $t$ 就可以向右“滑动”至使 $t_k$ 和 $s_i$ 对准,继续向右进行比较即可。

## (2) next 数组

模式中的每一个 $t_j$ 都对应一个 $k$ 值。由式(4-4)可知,这个 $k$ 值仅依赖于模式串 $t$ 本身字符序列的构成,而与主串 $s$ 无关。用 $\text{next}[j]$ 表示 $t_j$ 对应的 $k$ 值,next数组定义如下:

$$\text{next}[j] = \begin{cases} -1 & \text{当 } j=1 \\ \max\{k \mid k \leq j, \text{且 } t_0 t_1 \cdots t_{k-1} = t_{j-k} t_{j-k+1} \cdots t_{j-1}\} & \text{其他} \end{cases}$$

## (3) 求模式串 next 数组值的方法

求 next 数组的算法思想是利用递推,即已知 $\text{next}[0] = -1, \dots, \text{next}[j] = k$ ,求 $\text{next}[j+1]$ 。

求 $\text{next}[j+1]$ 的步骤如下:

- ① 判断串的 $t_j$ 是否等于串的 $t_k$ 。
- ② 若两者相等,有 $\text{next}[j+1] = k+1$ ,得解;否则由于 $t_j \neq t_k$ 失配,应将 $t_{\text{next}[k]}$ 与 $t_j$ 再作比较,即取 $k = \text{next}[k]$ ,转①。

整个 next 数组的求法只需赋初值 $\text{next}[0] = -1$ , $j$ 由 0 开始逐次递增求 $\text{next}[j+1]$ 即可。

**例 4-2** 设有模式串 $t = "abcaababc"$ ,则它的 next 数组值如图 4-7 所示。

| $j$              | 0  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|------------------|----|---|---|---|---|---|---|---|---|
| 模式串              | a  | b | c | a | a | b | a | b | c |
| $\text{next}[j]$ | -1 | 0 | 0 | 0 | 1 | 1 | 2 | 1 | 2 |

图 4-7 next 数组值

注意:任何模式串的 $\text{next}[0] = -1$ , $\text{next}[1] = 0$ 。

求 next 数组的具体实现如算法 4-5 所示。

### 算法 4-5 求 next 数组

```

void getnext(char * t)
{
 j = 0; k = -1;
 m = strlen(t);
 next[0] = -1;
 while(j < m - 1) //只需循环 m - 1 次,因为每次的 next 值由前面的决定
 {
 if(k == -1 || t[j] == t[k])
 {
 j++;
 k++;
 next[j] = k;
 }
 else
 k = next[k];
 }
}

```

### (4) KMP 算法描述

在求得模式的 next 数组之后，匹配可按如下步骤进行：

- ① 设以  $i$  和  $j$  分别指示主串和模式中的比较字符的下标，令  $i$  的初值为 0， $j$  的初值为 0。
- ② 在匹配过程中若  $s_i = t_j$ ，则  $i$  和  $j$  分别加 1，继续比较  $s$  和  $t$  的下一个字符；若  $s_i \neq t_j$ ，则  $i$  不变， $j$  退到  $\text{next}[j]$  位置再比较。若是  $j$  退到值为 -1（即模式的第一个字符失配），则此时  $i$  和  $j$  分别加 1，表明从主串的下一个字符起和模式重新开始匹配。

**例 4-3** 仍以前面简单模式匹配中使用的主串  $s = "ababcabcacb"$  和模式  $t = "abcac"$  为例来说明 KMP 匹配算法过程。

模式串  $t = "abcac"$  的 next 数组值如图 4-8 所示。

| $j$              | 0  | 1 | 2 | 3 | 4 |
|------------------|----|---|---|---|---|
| 模式串              | a  | b | c | a | c |
| $\text{next}[j]$ | -1 | 0 | 0 | 0 | 1 |

图 4-8 next 数组值

KMP 匹配过程如图 4-9 所示，整个过程仅需 3 趟。在已知 next 数组的情况下，KMP 模式匹配如算法 4-6 所示。

### 算法 4-6 KMP 算法

```

int KMPmatching(char * s, char * t)
{
 i = 0;
 j = 0;
 n = strlen(s);
 m = strlen(t);
 while(i < n&&j < m)
 {
 if(j == -1 || s[i] == t[j])
 {
 i++;
 j++;
 }
 else
 j = next[j];
 }
 if (j >= m)
 return i - m;
 else
 return -1;
}

```

根据算法 4-5 可知, 求 next 算法的时间复杂度是  $O(m)$ 。因此, KMP 算法的时间复杂度是  $O(n + m)$ 。

#### (5) next 数组的缺陷和改进

利用模式的 next 数组进行模式和主串的匹配时, 可能还会有多余的比较, 从而影响匹配效率。

例如, 已知主串  $s = "aaabaaaab"$  和模式串  $t = "aaaab"$ , 利用 KMP 算法进行模式匹配。

模式串  $t$  的 next 数组值如图 4-10 所示。

根据 KMP 算法, 当  $i = 3$ 、 $j = 3$  时,  $s_3 \neq t_3$ 。由  $\text{next}[j]$  的指示还需要进行  $i = 3$ 、 $j = 2$ ,  $i = 3$ 、 $j = 1$ ,  $i = 3$ 、 $j = 0$  等 3 次比较。实际上, 因为模式中下标为 0、1、2 的字符和下标为 3 的字符都相等, 因此不再需要和主串中下标为 3 的字符作比较, 而可以将模式一气向右滑动 4 个字符的位置直接进行  $i = 4$ 、 $j = 0$  的字符比较。

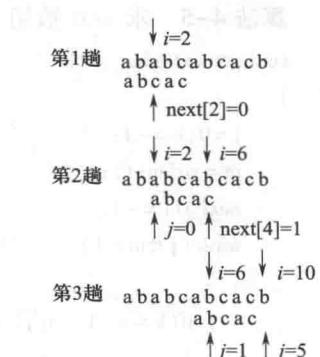


图 4-9 KMP 模式匹配

图 4-10 next 数组的值

| $j$                 | 0  | 1  | 2  | 3  | 4 |
|---------------------|----|----|----|----|---|
| 模式串 $t$             | a  | a  | a  | a  | b |
| $\text{next}[j]$    | -1 | 0  | 1  | 2  | 3 |
| $\text{nextval}[j]$ | -1 | -1 | -1 | -1 | 3 |

图 4-10 next 数组的值

避免不必要的重复比较的方法是对 next 数组加以修正。在算法中求得  $\text{next}[j] = k$  后，要继续判断  $t_k$  和  $t_j$  是否相等，若相等还需使  $\text{next}[j] = \text{next}[k]$ 。修正后的 next 数组称为 nextval 数组。

求 nextval 数组值的算法读者可自行完成。

### 4.3.3 串的应用

文本编辑是面向用户的系统应用软件，广泛用于文本的输入和修改，如 Microsoft Word 软件。虽然各种文本编辑软件的功能强弱不同，但其基本功能均包括插入、删除、修改和查找等串的基本操作。

一个文本可由若干页构成，每一页又由若干行组成，因此，在设计文本编辑软件时，可将一个文本看成一个串，将页看成是文本串的子串，而行则可看做是页的子串。在进行存储时，可以将一行的串用顺序结构存储，而行与行之间用指针链接。

例如，对于下列 C++ 源程序：

```
void main()
{
 int a,b;
 cin >> a >> b;
 cout << a > b? a:b;
}
```

输入内存后的储存形式如图 4-11 所示。图中符号“↙”表示换行符。

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| v | o | i | d |   | m | a | i | n | ( | ) | ↙ | { | i | n | t |   | a | , | b |
| ; | ↙ |   | c | i | n | > | > | a | > | > | b | ; | ↙ |   | c | o | u | t | < |
| < | a | > | b | ? | a | : | b | ; | ↙ | } | ↙ |   |   |   |   |   |   |   |   |

图 4-11 文本串示例

为了管理文本串中的页和行，在进入文本编辑时，编辑软件先为文本串建立相应的页表和行表。页表的每一项列出页号和该页的起始行号，行表的每一项则指示每一行的行号、起始地址和该行子串的长度。行表和页表与串是分开存储的。行表和页表反映了串的存储情况，相当于是串的一种查找索引，因此也称为串的存储映像。通过串的存储映像可以更方便地对串值进行大量的同类操作。

设图 4-11 所示文本串只占一页，起始行号为 1，则该文本串的行表如表 4-1 所示。

表 4-1 图 4-11 所示正文串的行表

| 行 号 | 起始 地 址 | 长 度 |
|-----|--------|-----|
| 1   | 100    | 12  |
| 2   | 110    | 10  |
| 3   | 120    | 12  |
| 4   | 131    | 16  |
| 5   | 146    | 2   |

在文本编辑软件中设立页指针、行指针和字符指针，分别指示当前操作的页、行和字符。如果在某行内插入或删除若干字符，那么要修改行表中该行的长度；如果该行长度因插入而超出了原分配给它的存储空间，那么要为该行重新分配存储空间，并修改该行的起始位置。例如，对上述源程序进行编辑，将其中的第 4 行修改成

```
cout << a > b? a:b << endl;
```

修改后的行表如表 4-2 所示。

表 4-2 修改后的行表

| 行 号 | 起始 地 址 | 长 度 |
|-----|--------|-----|
| 1   | 100    | 12  |
| 2   | 110    | 10  |
| 3   | 120    | 12  |
| 4   | 148    | 23  |
| 5   | 146    | 2   |

当插入或删除一行时，必须同时对行表也进行插入和删除。若被删除的“行”是所在页的起始行，则还要修改页表中相应页的起始行号（应修改成下一行的行号）。为了查找方便，将行表按行号递增的顺序安排，因此对行表进行插入或删除时需移动操作之后的全部表项。页表的维护与行表类似，在此不再赘述。由于对文本的访问是以页表和行表作为索引的，因此在删除一页或一行时，可以只对页表或行表作相应修改，不必删除所涉及的字符，这样可以节省不少时间。

以上仅概述了文本编辑中涉及的基本操作，具体算法请读者作为实验来完成。

## 本章小结

串是数据元素为字符的线性表，同时串又具有自身的特性。串的处理在计算机非数值处理中占有重要的地位。本章介绍了串的基本概念、基本操作以及模式匹配算法。

本章学习要点如下：

- (1) 掌握串的基本概念和基本操作算法。
- (2) 掌握串的简单模式匹配算法和 KMP 算法。

## 习题 4

4.1 不调用 C/C++ 的字符串函数，完成 StrStr() 函数，即把主串中子串及以后的字符全部返回。例如，主串是"12345678"，子串是"234"，那么函数的返回值就是"2345678"。

4.2 计算下列串的 next 数组：

- (1)"ABCDEFG"
- (2)"AAAAAAA"
- (3)"BABBAAB"
- (4)"AAAAAAB"
- (5)"ABCABDAAABC"
- (6)"ABCABDABEABCABDABF"
- (7)"ABBACXY"

4.3 已知主串  $s = "cbaacbcacbcaacbcbc"$ ，模式串  $t = "cbcaacbcbc"$ ，求出  $t$  的 next 数组值和 nextval 数组值，并画出 KMP 算法匹配。

4.4 编写输入两个字符串  $s$  和  $t$ ，统计串  $s$  包含串  $t$  个数的算法。

4.5 编写从串  $s$  中删除所有与串  $t$  相同的子串的算法。

4.6 编写求串  $s$  和串  $t$  的最大公共子串的算法。

4.7 编写一个高效率的算法来颠倒单词在字符串里的出现顺序。例如，把字符串"Do or do not, there is no try." 转换为"try. no is there not, do or Do"。假设所有单词都以空格为分隔符，标点符号也当做字母来对待。试阐述算法的设计思路并对解决方案的执行效率进行评估。

4.8 编写一个高效率的算法来删除字符串里的给定字符。算法的调用函数如下所示：

```
void RemoveChars(char str[],char remove[]);
```

注意：remove 中的所有字符都必须从 str 中删除干净。例如，如果 str 是"Battle of the Vowels: Hawaii VS. Grozny"，remove 是"aeiou"，这个函数将把 str 转换为"Bttl f th Vwls: Hw VS. Grzny"。试阐述算法的设计思路并对解决方案的执行效率进行评估。

## 上机实验题 4

### 实验题 4.1 实现串的模式匹配等算法。

- (1) 采用顺序存储方式存储串，建立两个字符串  $s$ 、 $t$ ，利用 BF 算法求串  $t$  在串  $s$  中首次出现的位置。
- (2) 采用顺序存储方式存储串，建立两个字符串  $s$ 、 $t$ ，利用 KMP 算法求串  $t$  在串  $s$  中首次出现的位置。
- (3) 采用顺序存储方式存储串，建立两个字符串  $s$ 、 $t$ ，利用改进的 KMP 算法求串  $t$  在串  $s$  中首次出现的位置。

### 实验题 4.2 利用恺撒密码对文件进行加解密。

恺撒密码是一种置换密码，它的加密原理是将字母替换为它后面的另一个字母，从而起到加密作用。假如有这样一段明文 "security"，用偏移量为 3 的恺撒密码加密后，密文为 "vhfxulwb"。

这种加密方法可以依据移位的不同产生新的变化。将明文记为  $ch$ ，密文记为  $c$ ，位移量（密钥）记作  $key$ ，更具一般性的恺撒密码加密过程可记为如下的变换：

$$c \equiv (ch + key) \bmod n \quad (\text{其中 } key \text{ 为位移量, } n \text{ 为基本字符个数})$$

同样，解密过程可表示为

$$ch \equiv (c - key + n) \bmod n \quad (\text{其中 } key \text{ 为偏移量, } n \text{ 为基本字符个数})$$

基本要求：

- (1) 输入一段英文（字符串），采用恺撒密码加密成密文，偏移量（密钥）由用户输入。
- (2) 读入密文字符串，解密成明文，偏移量（密钥）由用户输入。
- (3) 用文件实现输入和输出。

# 第5章 数组和特殊矩阵

前面几章讨论的线性表、栈、队列和串都是线性结构，它们共同的逻辑结构特征是每个数据元素至多有一个直接前驱和直接后继。本章介绍的数组是程序设计中常用的数据类型，它是线性表的推广（一维数组除外），即每个数据元素可能有多个直接前驱和直接后继。本章主要介绍多维数组和特殊矩阵的存储实现。

## 5.1 数组

### 5.1.1 数组的基本概念

数组（Array）是由一组类型相同的数据元素构成的有序集合，每个数据元素称为一个数组元素，每个数组元素都和一组唯一的下标值对应。数组可分为一维数组和多维数组。一维数组可看成是一种线性表。二维数组中的每个元素  $a_{ij}$  都是属于两个向量：第  $i$  行的行向量和第  $j$  列的列向量，即每个数组元素受两个线性关系的约束。推而广之， $n$  维数组中的每个元素则受到  $n$  个线性关系的约束。

二维数组以及多维数组可以看成是线性表的推广，其中，二维数组（或称矩阵）可以看成是这样一个线性表：它的每个数据元素也是一个线性表。例如，图 5-1 所示的二维数组  $A$ ，可以看成是一个线性表：

$$A = (a_0, a_1, \dots, a_{m-1})$$

其中，每个数据元素是一个行向量形式的线性表，

$$a_i = (a_{i0}, a_{i1}, \dots, a_{i,n-1}) \quad (0 \leq i \leq m-1)$$

或者

$$A = (a_0, a_1, \dots, a_{n-1})$$

其中，每个数据元素是一个列向量形式的线性表，即

$$a_j = (a_{0j}, a_{1j}, \dots, a_{m-1,j}) \quad (0 \leq j \leq n-1)$$

依此类推，一个  $n$  维数组可以看成是一个线性表，它的每个数据元素是一个  $n-1$  维数组。

对数组的操作主要是存取数组元素，即向某个数组元素存入数据和获取某个数组元素中的数据。

$$A_{mn} = \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

图 5-1 二维数组图例

## 5.1.2 数组的存储结构

数组一旦建立，数组元素个数和元素间的关系就不再发生变化，对数组一般不做插入和删除操作。因此，对数组的存储表示宜采用顺序存储方法。

在采用顺序存储结构表示多维数组时，首先必须按某种次序将数组元素排成一个线性序列，然后再将这个线性序列存放在存储器中。以图 5-1 所示的二维数组  $A$  为例，通常有两种顺序存储方式。

### 1. 行优先顺序

将数组元素按行排列，第  $i+1$  个行向量紧接在第  $i$  个行向量后面。例如，以二维数组  $A_{mn}$  为例，按行优先存储的线性序列为

$$a_{00}, a_{01}, \dots, a_{0,n-1}, a_{10}, a_{11}, \dots, a_{1,n-1}, \dots, a_{m-1,0}, a_{m-1,1}, \dots, a_{m-1,n-1}$$

### 2. 列优先顺序

将数组元素按列排列，第  $j+1$  个列向量紧接在第  $j$  个列向量之后。例如，以二维数组  $A_{mn}$  为例，按列优先存储的线性序列为

$$a_{00}, a_{10}, \dots, a_{m-1,0}, a_{01}, a_{11}, \dots, a_{m-1,1}, \dots, a_{0,n-1}, a_{1,n-1}, \dots, a_{m-1,n-1}$$

将以上规则可以推广到多维数组的情况。对于行优先顺序，可规定为先排最右的下标，从右向左，最后排最左下标；对于列优先顺序，先排最左下标，从左向右，最后排最右下标。

例如，三维数组  $A_{mnp}$  中的元素可按行优先顺序排列成：

$$\left. \begin{array}{ccccccccc} a_{000} & a_{001} & a_{002} & \cdots & a_{00,p-1} \\ a_{010} & a_{011} & a_{012} & \cdots & a_{01,p-1} \\ \vdots & & & & & & & & \\ a_{0,n-1,0} & a_{0,n-1,1} & a_{0,n-1,2} & \cdots & a_{0,n-1,p-1} \\ a_{100} & a_{101} & a_{102} & \cdots & a_{10,p-1} \\ a_{110} & a_{111} & a_{112} & \cdots & a_{11,p-1} \\ \vdots & & & & & & & & \\ a_{1,n-1,0} & a_{1,n-1,1} & a_{1,n-1,2} & \cdots & a_{1,n-1,p-1} \\ \vdots & & & & & & & & \\ a_{m-1,0,0} & a_{m-1,0,1} & a_{m-1,0,2} & \cdots & a_{m-1,0,p-1} \\ a_{m-1,1,0} & a_{m-1,1,1} & a_{m-1,1,2} & \cdots & a_{m-1,1,p-1} \\ \vdots & & & & & & & & \\ a_{m-1,n-1,0} & a_{m-1,n-1,1} & a_{m-1,n-1,2} & \cdots & a_{m-1,n-1,p-1} \end{array} \right\} \begin{array}{l} i=0 \\ i=1 \\ \vdots \\ i=m-1 \end{array}$$

如果要检索某个元素，只要记住第一个元素的地址（也称基地址）、每一维的大小以及每个数组元素在内存中占用的单元数，就可以用元素的下标值通过简单的函数关系，计算出该元

素的存放地址，从而可以实现对数组元素的随机存取。

例如，将二维数组  $A_{mn}$  按行优先顺序存储在内存以后，元素  $a_{ij}$  的地址计算公式为

$$LOC(a_{ij}) = LOC(a_{00}) + (i \times n + j) \times d$$

式中， $LOC(a_{ij})$  表示元素  $a_{ij}$  的存储地址， $d$  表示每个数组元素在内存中占用的单元数。

类似地，三维数组  $A_{mnp}$  按行优先顺序存储，其元素  $a_{ijk}$  的地址计算公式为

$$LOC(a_{ijk}) = LOC(a_{000}) + (i \times n \times p + j \times p + k) \times d$$

读者不难推广到更高维的情况。

## 5.2 特殊矩阵的压缩存储

在科学与工程计算中，矩阵是一种常用的数学模型。在用高级语言编制程序时，常将一个矩阵描述为一个二维数组。

在矩阵中的非零元素呈某种规律分布或者矩阵中出现大量零元素的情况下，会占用许多单元去存储重复的非零元素或零元素，这对高阶矩阵会造成极大的存储空间浪费。为了节省存储空间，可以对这类特殊矩阵进行压缩存储：为多个相同的非零元素只分配一个存储空间，对零元素不分配空间。

特殊矩阵主要包括对称矩阵、三角矩阵、对角矩阵和稀疏矩阵等。下面分别讨论它们的压缩存储。

### 5.2.1 对称矩阵的压缩存储

在一个  $n$  阶矩阵  $A$  中，若元素满足下述性质：

$$a_{ij} = a_{ji} \quad 0 \leq i, j \leq n - 1$$

则称矩阵  $A$  为对称矩阵。图 5-2 所示的是一个对称矩阵的例子。

由于对称矩阵中的元素关于主对角线对称，因此只要存储矩阵中上三角或下三角中的元素即可，这样能节约近一半的存储空间。不失一般性，假设按行优先顺序存储下三角部分的元素，如图 5-3 所示。

|                                                                                                                                           |          |                       |                                    |          |                                                                            |
|-------------------------------------------------------------------------------------------------------------------------------------------|----------|-----------------------|------------------------------------|----------|----------------------------------------------------------------------------|
| $A = \begin{bmatrix} 1 & 5 & 2 & 8 & 7 \\ 5 & 0 & 6 & 0 & 3 \\ 2 & 6 & 4 & 5 & 1 \\ 8 & 0 & 5 & 9 & 7 \\ 7 & 3 & 1 & 7 & 0 \end{bmatrix}$ | $a_{00}$ | $a_{10} \quad a_{11}$ | $a_{20} \quad a_{21} \quad a_{22}$ | $\vdots$ | $a_{n-1,0} \quad a_{n-1,1} \quad a_{n-1,2} \quad \cdots \quad a_{n-1,n-1}$ |
|-------------------------------------------------------------------------------------------------------------------------------------------|----------|-----------------------|------------------------------------|----------|----------------------------------------------------------------------------|

图 5-2 对称矩阵示例

图 5-3  $n$  阶方阵的下三角部分

对于一个  $n$  阶矩阵，其下三角部分共有  $n \times (n + 1)/2$  个元素。按行优先顺序将这些元素存放在一个一维数组  $sa[n(n + 1)/2]$  中，如图 5-4 所示。为了便于随机访问对称矩阵中的元素，需要给出  $a_{ij}$  和  $sa[k]$  之间的对应关系。若  $i \geq j$ ，则下三角矩阵中位于元素  $a_{ij}$  前面的共有  $i$  行非零元素，且本行位于  $a_{ij}$  前面的还有  $j$  个非零元素。若  $i \leq j$ ，表明该元素位于上三角部分，则求元素  $a_{ji}$  在一维数组  $sa$  中的位置。据此可推出  $a_{ij}$  和  $sa[k]$  之间的对应关系为

$$k = \begin{cases} i \times (i + 1)/2 + j, & \text{当 } i \geq j \\ j \times (j + 1)/2 + i, & \text{当 } i < j \end{cases}$$

|          |          |          |          |          |          |          |          |          |             |             |          |               |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|-------------|-------------|----------|---------------|
| $a_{00}$ | $a_{10}$ | $a_{11}$ | $a_{20}$ | $a_{21}$ | $a_{22}$ | $\cdots$ | $a_{ij}$ | $\cdots$ | $a_{n-1,0}$ | $a_{n-1,1}$ | $\cdots$ | $a_{n-1,n-1}$ |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|-------------|-------------|----------|---------------|

图 5-4 对称矩阵的压缩存储

## 5.2.2 三角矩阵的压缩存储

以主对角线划分，三角矩阵有上三角和下三角两种。上三角矩阵如图 5-5 (a) 所示，它的下三角（不包括主对角线）中的元素均为常数  $c$ 。下三角矩阵则正好相反，上三角中的元素均为常数  $c$ ，如图 5-5 (b) 所示。

|          |          |          |          |               |
|----------|----------|----------|----------|---------------|
| $a_{00}$ | $a_{01}$ | $a_{02}$ | $\cdots$ | $a_{0,n-1}$   |
| $c$      | $a_{11}$ | $a_{12}$ | $\cdots$ | $a_{1,n-1}$   |
| $c$      | $c$      | $a_{22}$ | $\cdots$ | $a_{2,n-1}$   |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$      |
| $c$      | $c$      | $c$      | $\cdots$ | $a_{n-1,n-1}$ |

(a) 上三角矩阵

|             |             |             |          |               |
|-------------|-------------|-------------|----------|---------------|
| $a_{00}$    | $c$         | $c$         | $c$      | $c$           |
| $a_{10}$    | $a_{11}$    | $c$         | $c$      | $c$           |
| $a_{20}$    | $a_{21}$    | $a_{22}$    | $c$      | $c$           |
| $\cdots$    | $\cdots$    | $\cdots$    | $\cdots$ | $\cdots$      |
| $a_{n-1,0}$ | $a_{n-1,1}$ | $a_{n-1,2}$ | $\cdots$ | $a_{n-1,n-1}$ |

(b) 下三角矩阵

图 5-5 三角矩阵

在存储三角矩阵时，类似于对称矩阵，只需要存储其上（下）三角中的元素，另外再加一个常数  $c$  的存储空间即可。对于图 5-5 (a) 所示的上三角矩阵，可将其压缩存储到一个一维数组  $sa[n(n + 1)/2 + 1]$  中，其中常数  $c$  存放在数组的最后一个分量中。

对于上三角矩阵，若按行优先顺序存放矩阵中的元素  $a_{ij}$  时，则  $sa[k]$  和  $a_{ij}$  的对应关系为

$$k = \begin{cases} i \times (2n - i + 1)/2 + j - i, & \text{当 } i \leq j \\ n \times (n + 1)/2, & \text{当 } i > j \end{cases}$$

对于下三角矩阵，若按行优先顺序存放矩阵中的元素  $a_{ij}$  时，则  $sa[k]$  和  $a_{ij}$  的对应关系是：

$$k = \begin{cases} i \times (i + 1)/2 + j, & \text{当 } i \geq j \\ n \times (n + 1)/2, & \text{当 } i < j \end{cases}$$

### 5.2.3 对角矩阵的压缩存储

对角矩阵是指矩阵中所有的非零元素集中在以主对角线为中心的带状区域中，即除了主对角线和主对角线相邻两侧的若干条对角线上的元素之外，其余元素皆为零。例如，图 5-6 所示的是一个三对角矩阵。推而广之，一个  $k$  对角矩阵 ( $k$  为奇数)  $A$  是满足下述条件的矩阵：若  $|i-j| > (k-1)/2$ ，则元素  $a_{ij} = 0$ 。

对角矩阵的压缩存储方法是，将位于以主对角线为中心的带状区域中的非零元素按某种顺序（如行优先顺序、列优先顺序或按对角线的顺序）压缩存储到一个一维数组中。

例如，可将三对角矩阵  $A$  中的非零元素按行优先顺序

存放到数组  $sa[3n-2]$  中。在三对角矩阵  $A$  中，除第一行和最后一行只有两个非零元素外，其他每行中均有三个非零元素，可知矩阵中位于  $a_{ij}$  之前的非零元素共有  $i$  行，共包含  $3i-1$  个非零元素，且  $a_{ij}$  所在的第  $i$  行前面还有  $j-(i-1)$  个非零元素。因此可推出， $sa[k]$  与三对角矩阵中的元素  $a_{ij}$  存在的对应关系为

$$k = 3 \times i - 1 + j - (i - 1) = 2 \times i + j$$

从而可以直接实现对三对角矩阵中元素的随机存取。

### 5.2.4 稀疏矩阵的压缩存储

在前面讨论的几种特殊矩阵中，元素的分布都具有一定的规律，因此可以按某种顺序压缩存储到一个一维数组中，并能够实现随机存取。但是，对于那些非零元素在矩阵中的分布没有规律的特殊矩阵，如稀疏矩阵，则需要寻找其他方法来实现压缩存储。

对于稀疏矩阵很难给出一个确切的定义。一般认为，设矩阵  $A_{mn}$  中有  $s$  个非零元素，若  $s$  远远小于矩阵元素的总数（即  $s \ll m \times n$ ），则称  $A$  为稀疏矩阵。

对于稀疏矩阵，若以常规方法存储，即以二维数组表示，则会带来下面两个问题：

- (1) 零值元素占的空间很大；
- (2) 计算中进行了很多和零值的运算，而很多和零值的运算可能是无意义的。

为解决上述问题，需要对矩阵进行压缩存储，即只存储稀疏矩阵中的非零元素。由于稀疏矩阵中的非零元素的分布一般是没有规律的，因此在存储各非零元素值的同时，还必须同时记下它们所在的行和列的位置。这样，稀疏矩阵中的一个非零元素  $a_{ij}$  需由一个三元组  $(i, j, a_{ij})$  唯一确定。

将稀疏矩阵中的所有非零元素对应的三元组所构成的集合按行优先顺序排成一个线性表，

图 5-6 三对角矩阵

称为三元组表。以下的讨论中，均假定三元组表是按行优先顺序排列的。

例如，图 5-7 所示的稀疏矩阵  $A$  所对应的三元组线性表为

$((0, 1, 3), (0, 4, 7), (1, 5, 10), (2, 1, -5), (3, 2, 6), (3, 4, -9), (5, 3, 1))$

对上述三元组线性表的不同存储方法可引出稀疏矩阵不同的压缩存储方法，主要有三元组顺序表和十字链表两种。

$$A = \begin{bmatrix} 0 & 3 & 0 & 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 10 & 0 \\ 0 & -5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 6 & 0 & -9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & -5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 7 & 0 & 0 & -9 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

图 5-7 稀疏矩阵示例

### 1. 三元组顺序表

假设以顺序存储结构来表示三元组表，则可得到稀疏矩阵的一种压缩存储方法——三元组顺序表。

为了完整地表示一个稀疏矩阵的信息，除了存储稀疏矩阵中非零元素对应的三元组表外，还需要保存该矩阵的行数、列数、非零元素个数等信息。因此，基于三元组顺序表结构的稀疏矩阵类可定义如下：

```
template < class T >
struct Triple
{
 int r, c; // 该非零元素的行下标与列下标
 T elem; // 该非零元素的元素值
};

template < class T >
class SparseMatrix
{
public:
 vector < Triple < T > > triList; // 存储矩阵中非零元素的三元组表
 int rows, cols, num; // 矩阵的行数、列数和非零元素个数

 SparseMatrix(); // 无参构造函数
 SparseMatrix(Triple < T > * tlist, int rs, int cs, int n); // 有参构造函数
 void trans(SparseMatrix&B); // 矩阵转置运算
```

```

SparseMatrix& plus(SparseMatrix&B); //矩阵加法运算
SparseMatrix& mult(SparseMatrix&B); //矩阵乘法运算
void print(); //打印矩阵信息
};


```

本章引入 C++ 标准模板库 (Standard Template Library, STL) 中的 vector 容器来定义类 SparseMatrix 中的三元组表，而没有使用常规的一维数组表示。vector 是一个多功能的模板类，用其定义的三元组表是一个动态数组，对该三元组的操作会较一维数组灵活且方便。读者可以查阅相关 C++ 书籍进一步了解和应用。在使用 Visual C++ 6.0 编译程序时，读者需要注意，vector < Triple < T > > 中的 Triple < T > 前后都要加空格。

对稀疏矩阵进行压缩存储后，由于无法直接对矩阵元素实现随机访问，因此会使得一些矩阵运算的实现可能变得复杂。常用的矩阵运算包括矩阵转置、矩阵加、矩阵减、矩阵乘等。这里仅讨论基于三元组顺序表结构的矩阵转置运算的实现。

对于一个  $m \times n$  的矩阵  $A$ ，其转置矩阵  $B$  是一个  $n \times m$  的矩阵，且  $a_{ij} = b_{ji}$ ，其中  $0 \leq i \leq m - 1, 0 \leq j \leq n - 1$ 。例如，图 5-7 所示的矩阵  $A$  和  $B$  互为转置矩阵。下面讨论稀疏矩阵转置运算的两种实现算法：朴素转置算法和快速转置算法。

### (1) 朴素转置算法

在三元组顺序表的存储结构下，将矩阵  $A$  转置为矩阵  $B$  的关键在于如何将存储矩阵  $A$  中非零元素的三元组表转换为矩阵  $B$  对应的三元组表，如图 5-8 所示。

| r | c | elem |
|---|---|------|
| 0 | 1 | 3    |
| 0 | 4 | 7    |
| 1 | 5 | 10   |
| 2 | 1 | -5   |
| 3 | 2 | 6    |
| 3 | 4 | -9   |
| 5 | 3 | 1    |

A.triList

| r | c | elem |
|---|---|------|
| 1 | 0 | 3    |
| 1 | 2 | -5   |
| 2 | 3 | 6    |
| 3 | 5 | 1    |
| 4 | 0 | 7    |
| 4 | 3 | -9   |
| 5 | 1 | 10   |

B.triList

图 5-8 矩阵的转置

由于矩阵  $A$  的列是矩阵  $B$  的行，因此一种朴素的做法是按矩阵  $A$  的列序进行转置，这样所生成的矩阵  $B$  的三元组表必定是按行优先顺序存放的。但是，为了依次找到矩阵  $A$  中每一列的非零元素，需要对矩阵  $A$  的三元组表从头至尾扫描一趟。也就是说，为了处理矩阵  $A$  的  $n$  列的非零元素，需要对矩阵  $A$  的三元组表进行  $n$  趟扫描。朴素矩阵转置的具体操作如算法 5-1 所示。

### 算法 5-1 朴素矩阵转置算法

```

template < class T >
void SparseMatrix < T > :: trans(SparseMatrix < T > & B)
{
 B. rows = cols;
 B. cols = rows;
 B. num = num;
 B. triList. resize(num); /* resize(n) 用于改变容器大小并创建对象。调用此函数后，就可以引用容器
 内的对象了，如通过下标运算符[]来引用元素对象 */
 if(num == 0)
 return; // 若非零元素个数为 0，则转置结束
 q = 0;
 for(col = 0; col < cols; ++ col) // 按矩阵 A 的列序进行转置
 for(p = 0; p < num; ++ p)
 if(triList[p].c == col) {
 B. triList[q].r = triList[p].c;
 B. triList[q].c = triList[p].r;
 B. triList[q].elem = triList[p].elem;
 ++ q;
 }
}

```

分析算法 5-1 的时间复杂度，该算法的主要时间耗费是在 col 和 p 的两重循环上。对于一个  $m$  行  $n$  列且非零元素个数为  $t$  的稀疏矩阵而言，该算法的时间复杂度为  $O(t \times n)$ 。在最坏情况下，稀疏矩阵中的非零元素个数  $t$  与  $m * n$  具有相同的数量级，上述算法的时间复杂度为  $O(m * n^2)$ 。显然这种情况下，朴素矩阵转置算法的效率较低。

#### (2) 快速转置算法

要提高算法 5-1 的效率，必须要减少对矩阵  $A$  的三元组表进行扫描的趟数。一种快速的矩阵转置算法思想是直接按矩阵  $A$  的行序进行转置，即通过一趟扫描矩阵  $A$  的三元组表，同时将每个非零元素转置后直接放入矩阵  $B$  的三元组表中的适当位置。

显然，实现此算法的关键是每次从矩阵  $A$  的三元组表中取出一个非零元素后，如何确定该元素转置后在矩阵  $B$  的三元组表中的相应位置？实际上，如果能预先确定矩阵  $A$  的每一列的第一个非零元素在矩阵  $B$  的三元组表中的位置，每一列的其他非零元素则依次排在该位置的后面；那么在按行序扫描矩阵  $A$  的每一个非零元素时，就可以直接对它们在矩阵  $B$  的三元组表中进行定位了。

为此，需要引入两个辅助数组：cnum[cols]，每个分量表示矩阵  $A$  的某一列的非零元素个

数;  $\text{cpot}[cols]$ , 每个分量的初始值表示矩阵  $A$  的某一列的第一个非零元素在  $B$  中的位置。

显然, 对数组  $\text{cnum}$  的各个元素值的初始化可以通过对矩阵  $A$  的三元组表扫描一趟完成。在数组  $\text{cnum}$  初始化的基础上, 对数组  $\text{cpot}$  的初始化可按如下的递推关系给出:

$$\begin{cases} \text{cpot}[0] = 0; \\ \text{cpot}[col] = \text{cpot}[col - 1] + \text{cnum}[col - 1] & 1 \leq col \leq cols - 1 \end{cases}$$

例如, 对于图 5-7 所示的矩阵  $A$  初始化数组  $\text{cnum}$  和  $\text{cpot}$  的值如表 5-1 所示。

表 5-1 数组  $\text{cnum}$  和  $\text{cpot}$  的初始化值

| col                | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|--------------------|---|---|---|---|---|---|---|
| $\text{cnum}[col]$ | 0 | 2 | 1 | 1 | 2 | 1 | 0 |
| $\text{cpot}[col]$ | 0 | 0 | 2 | 3 | 4 | 6 | 7 |

综上所述, 快速矩阵转置算法的具体描述如算法 5-2 所示。

### 算法 5-2 快速矩阵转置算法

```
template < class T >
void SparseMatrix < T > ::trans(SparseMatrix < T > & B)
{
 B. rows = cols;
 B. cols = rows;
 B. num = num;
 B. trilist. resize(num);
 if(num == 0)
 return; //若非零元素个数为, 则转置结束
 int * cnum = new int[cols];
 int * cpot = new int[cols];
 for(col = 0; col < cols; ++ col)
 cnum[col] = 0; //置初值
 for(t = 0; t < num; ++ t)
 ++ cnum[triList[t].c]; //初始化数组 cnum 的元素值
 cpot[0] = 0;
 for(col = 1; col < cols; ++ col) //初始化数组 cpot 的元素值
 cpot[col] = cpot[col - 1] + cnum[col - 1];
 for(p = 0; p < num; ++ p)
 {
 col = triList[p].c; //取当前非零元素的列号
 q = cpot[col]; //取当前非零元素在 B 中的位置
 }
}
```

```

B. triList[q]. r = triList[p]. c;
B. triList[q]. c = triList[p]. r;
B. triList[q]. elem = triList[p]. elem;
++ cpot[col];
 // 预置本列的下一个非零元素在 B 中的位置
}
delete[] cnum;
delete[] cpot;
}

```

分析算法 5-2 的时间复杂度，此算法中有 4 个平行的 for 循环。对于一个  $m$  行  $n$  列且非零元素个数为  $t$  的稀疏矩阵而言，循环次数分别为  $n$  和  $t$  两种。因此，算法时间复杂度为  $O(n+t)$ ，显然优于朴素转置算法的时间复杂度。

## 2. 十字链表

三元组顺序表是用顺序存储结构来存储稀疏矩阵中的非零元素。当非零元素的位置或个数经常发生变化时，做插入和删除操作不便，而链接存储结构可以克服这一不足。

十字链表是一种常用的稀疏矩阵的链接存储结构。十字链表存储稀疏矩阵的基本做法是：链表中的每个结点由 5 个域组成，即除行号  $r$ 、列号  $c$  和元素值  $elem$  之外，增加了两个指针域：向右指针域  $right$ （用以指向同一行中的下一个非零元素）和向下指针域  $down$ （用以指向同一列中的下一个非零元素），如图 5-9 所示。

|        |     |         |
|--------|-----|---------|
| $r$    | $c$ | $elem$  |
| $down$ |     | $right$ |

稀疏矩阵中同一行的非零元素通过  $right$  指针域按列号顺序链接成一个线性链表，同一列的非零元素则通过  $down$  指针域按行号顺序链接成一个线性链表。这样每个非零元素既是某个行链表中的一个结点，同时又是某个列链表中的一个结点。每个结点好像处在一个十字交叉口上，因此称这种链表为十字链表。一个稀疏矩阵对应的十字链表由多个行链表和多个列链表组合而成，如何表示这样复杂结构的链表呢？一种有效的方法是引入两个一维指针数组分别存储各个行链表的头指针和各个列链表的头指针，然后用这两个指针数组表示十字链表。若给定如下稀疏矩阵：

$$\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 2 & 0 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

则该矩阵对应的十字链表结构如图 5-10 所示。

十字链表的结点类型和基于十字链表结构的稀疏矩阵类型定义如下。

```

template < class T >
struct CrossNode
{

```

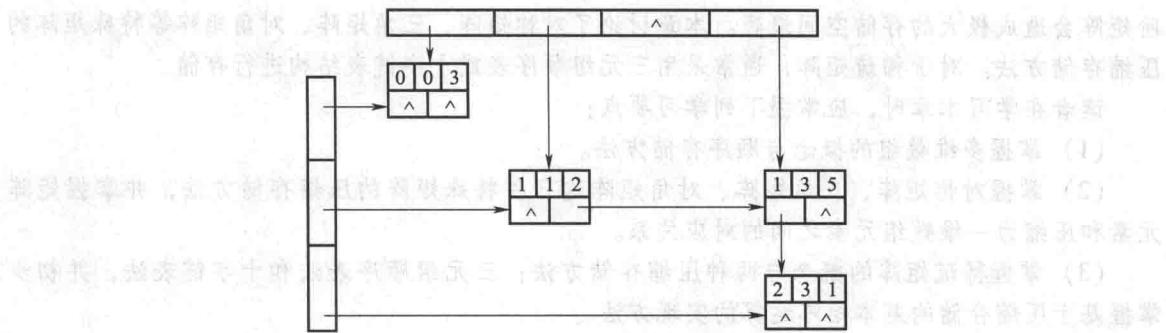


图 5-10 一个稀疏矩阵的十字链表结构

```

int r,c; //该非零元素所在的行号和列号
T elem;
CrossNode * right, * down;
};

template < class T >
class CrossMatrix
{
 vector < CrossNode < T > * > rheads,cheads;
 int rows,cols,num; //矩阵的行数、列数和非零元素个数
public:
 CrossMatrix(); //无参构造函数
 CrossMatrix(int r,int c,int n); //构造函数
 void trans(CrossMatrix& B); //矩阵转置运算
 CrossMatrix&plus(CrossMatrix& B); //矩阵加法运算
 CrossMatrix&mult(CrossMatrix& B); //矩阵乘法运算
 void print(); //打印矩阵信息
};

```

## 本章小结

多维数组是线性表的推广，每个数组元素受到多个关系的约束。对数组的存储表示宜采用顺序存储方法，存放的顺序主要有行优先顺序和列优先顺序两种。

矩阵是在科学与工程计算问题中一种常用的数学模型，存储矩阵的一种自然方法是使用二维数组。但是，当矩阵中的非零元素呈某种规律分布或者矩阵中出现大量的零元素时，这对高

阶矩阵会造成极大的存储空间浪费。本章讨论了对称矩阵、三角矩阵、对角矩阵等特殊矩阵的压缩存储方法。对于稀疏矩阵，通常采用三元组顺序表或十字链表结构进行存储。

读者在学习本章时，应掌握下列学习要点：

- (1) 掌握多维数组的概念与顺序存储方法。
- (2) 掌握对称矩阵、三角矩阵、对角矩阵这三种特殊矩阵的压缩存储方法，并掌握矩阵元素和压缩为一维数组元素之间的对应关系。
- (3) 掌握稀疏矩阵的概念与两种压缩存储方法：三元组顺序表法和十字链表法，并初步掌握基于压缩存储的基本矩阵运算的实现方法。

## 习题 5

5.1 按行优先顺序列出四维数组  $A[3][2][3][2]$  的所有元素在内存中的存储次序。

5.2 设有三对角矩阵  $A_{n \times n}$ ，将其按行优先顺序压缩存储于一维数组  $b[3 \times n - 2]$  中，使得  $a_{ij} = b[k]$ ，请用  $k$  表示  $i, j$  的下标变换公式。

5.3 画出矩阵  $M$  的三元组顺序表和十字链表。

$$M = \begin{bmatrix} 8 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 12 & 0 \\ 0 & -5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 6 & 0 & 5 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 & 0 & 0 \end{bmatrix}$$

5.4 若在矩阵  $A_{m \times n}$  中存在一个元素  $a_{ij}$  ( $0 \leq i \leq m-1, 0 \leq j \leq n-1$ ) 满足  $a_{ij}$  是第  $i$  行元素中最小值，且又是第  $j$  列元素中最大值，则称此元素值为该矩阵的一个马鞍点。假设以二维数组存储矩阵  $A_{m \times n}$ ，试编写求出矩阵中所有马鞍点的算法。

5.5 编写算法计算一个稀疏矩阵的对角线元素之和，要求稀疏矩阵用三元组顺序表表示。

5.6 编写两个稀疏矩阵相加的算法，要求稀疏矩阵用十字链表表示。

## 上机实验题 5

**实验题** 实现一个能进行稀疏矩阵基本运算的运算器程序。

**基本要求：**

- (1) 以三元组顺序表结构表示稀疏矩阵，实现矩阵转置和两个矩阵相加、相减的运算。
- (2) 稀疏矩阵的输入形式采用三元组表示，程序可以对三元组的输入顺序加以限制，如按行优先。
- (3) 矩阵转置和矩阵相加或相减所得结果矩阵另外生成。
- (4) 运算结果矩阵以通常的阵列形式输出。

# 第6章 广义表

在线性表的定义中，要求每个结点都应该是类型相同的基本数据元素。在广义表的定义中放松了这个约定：每个结点可以属于基本数据类型，也可以属于广义表类型。这种递归定义的结构，为我们展开了非线性结构的天地。

## 6.1 广义表的概念

广义表是  $n(n \geq 0)$  个数据元素组成的有限序列，一般记作

$$\text{GList} = (a_1, a_2, \dots, a_n)$$

其中， $a_i$  是广义表中第  $i$  个数据元素， $n$  是广义表的长度。若  $n=0$  时，广义表称为空表。

在线性表的定义中， $a_i(1 \leq i \leq n)$  只限于是单个元素；而在广义表的定义中， $a_i$  可以是单个元素，也可以是广义表，分别称为广义表 GList 的原子和子表。可以说，广义表结构是由若干原子或子表组成的线性序列，其中每个子表又可能包含原子和子表。这种逐渐深入的层次关系，是本章学习的重点和难点。

广义表结构中所有原子的数据类型都是相同的，这是和线性表相同之处；不同之处是广义表的数据元素之间不仅有先后关系，更有元素内部的层次关系。

通常用大写字母表示广义表，用小写字母表示原子。下面是一些广义表的例子：

$$A = ()$$

$$B = (a, b, c)$$

$$C = (a, (b, c, d), e)$$

$$D = ((a, b), c, (d, (e, f), g))$$

$$E = (a, (), (((), ()), b))$$

最简单的广义表结构就是线性表，如  $A$ 、 $B$  表，其中  $A$  表示一个空表， $B$  表示的广义表由 3 个数据元素组成。较复杂的广义表结构一般有更多的括号嵌套，如  $C$ 、 $D$ 、 $E$  表，其中， $C$  包含 3 个元素，第 2 个元素是子表； $D$  包含 3 个元素，第 1、3 个元素是子表，且第 3 个元素中又包含子表； $E$  包含 4 个元素，其中第 2 个元素是空表，第 3 个元素是由 2 个空表组成的广义表。

初学者应特别留意广义表中的顺序关系和层次关系，这两种关系可以通过长度和深度概念来强化。广义表的长度是指广义表中数据元素的个数。需要注意的是，数据元素可能是原子，也可能是子表。长度概念表达的是顺序关系中的元素个数，而非表中所有原子的个数。例如，

表 C、D、E 中数据元素的个数分别是 3、3、4，而原子个数是 5、7、2。

广义表的深度是指表中层次关系的最大深度。如表 A、B、C、D、E 的深度分别是 1、1、2、3、3。但是，这个定义有些含糊不清。严谨的定义是原子结点的深度是 0，广义表的深度是表中所有结点的最大值加 1。作为特例，空表的深度记作 1。由于表 B 所有元素都是原子结点，因此其深度为 1。表 C 中，由于第 1、第 3 结点的深度为 0，第 2 结点的深度是 1，因此深度为 2。依此类推，表 D 和表 E 的深度都是 3。

广义表的另一对概念是表头和表尾，表头是指广义表中的第一个元素，表尾指除表头外其余元素组成的广义表。例如，上述 A 表的表头是 ()，表尾是 ()；B 表的表头是 a，表尾是 (b, c)；C 表的表头是 a，表尾是 ((b, c, d), e)；D 表的表头是 (a b)，表尾是 (c, (d, (e, f), g))；E 表的表头是 a，表尾是 (((), ((), ()), b))。

表头、表尾概念的重要性体现在它们既是广义表的元素存取方法、广义表的分解方法，也暗示了广义表的一种构造方法。

## 6.2 广义表的存储结构

由于广义表的非线性特征，且定义中对各个子表的长度没有任何限制，因此如果试图采用顺序存储结构，必将引入许多人为约束，使广义表的存储、应用复杂化。因此，广义表一般采用链式存储结构。

### 6.2.1 广义表中结点的结构

由于广义表中每个结点可能是原子或子表，因此在广义表中存在两类结点。在高级语言中，指针所指的数据类型一般是固定的，为便于结点的统一操作，一般借助于联合（Union）类型，将两类结点存储结构归一化。为了在同一个结构类型中区别出原子和子表结点，引入了枚举类型 GListNodeType，定义如下：

```
enum GListNodeType { ATOM, LIST };
```

广义表结点的类型定义如下，结点结构如图 6-1 所示。

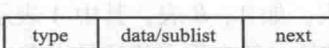


图 6-1 广义表结点的结构

```
template < class T >
struct GListNode
{
 GListNodeType type; // 结点类型
 T data; // 原子结点的数据
 GListNode *next; // 指向下一个结点的指针
};
```

```

union
{
 T data; // 原子值
 GListNode * sublist; // 子表头指针
};

GListNode < T > * next; // 存储后继结点指针
};

```

若 type 域取值为 ATOM，则 data 域存储原子的值；若 type 域取值为 LIST，则 sublist 域存储子表的头指针。在绘制存储结构时，type 域的 ATOM 值以 0 表示，LIST 值以 1 表示。

## 6.2.2 广义表的存储结构

在广义表的存储结构中，每个子表结点相当于子表的头结点。由于广义表及其子表在逻辑结构上完全相同，在存储规则也应一致，因此约定广义表结构有头结点。头结点的 type 域值取 LIST，其 sublist 域值指向广义表中的第一个数据结点。如此约定有利于简化程序结构，提高程序的可读性。图 6-2 所示的是 6.1 节中表 A、B、C、D、E 的存储结构。

如图 6-2 所示，每个广义表的头结点的 next 域都为空值，这是因为结构中只存储了一个广义表。若广义表的头结点的 next 域不为空值，则所存储的就是多个广义表结构。在这种情形下，广义表的头结点和子表结点完全相似，所有针对若干子表结点的算法都可以自然扩展为若干广义表的算法。这种扩展方法留给读者自行思考。

参照上述结点结构的定义，下面给出了广义表类模板 GLIST 的定义，其中核心数据成员 head 是广义表头结点的指针。

```

template < class T >
class GLIST
{
private:
 GListNode < T > * head;
 GListNode < T > * DoCreate(char s[], int &i); // 与广义表构造函数相关
 GListNode < T > * Copy(GListNode < T > * p); // 与拷贝构造函数相关
 void Traverse(GListNode < T > * p); // 与遍历广义表的 Traverse() 函数相关
 void Free(GListNode < T > * p); // 与析构函数相关
 int Depth(GListNode < T > * p); // 与计算广义表的 Depth() 函数相关

public:
 GLIST(); // 无参构造函数
 GLIST(char s[]); // 根据字符串 s 构造广义表对象
 GLIST(GLIST < T > &gl); // 拷贝构造函数

```

```

~ GList();
void Traverse();
int Length();
int Depth();
};

//析构函数
//遍历广义表
//计算广义表的长度
//计算广义表的深度

```

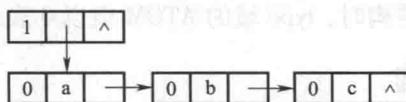
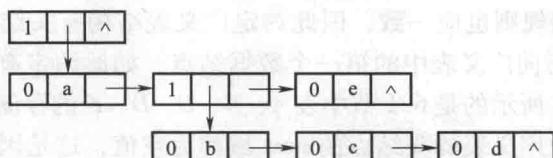
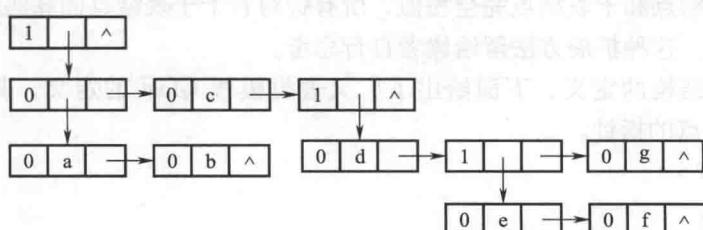
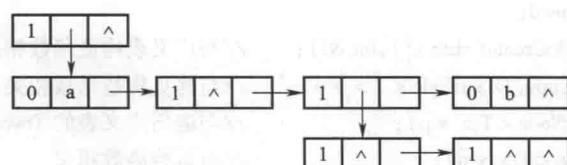
(a) 广义表  $A = ()$  的存储结构(b) 广义表  $B = (a, b, c)$  的存储结构(c) 广义表  $C = (a, (b, c, d), e)$  的存储结构(d) 广义表  $D = ((a, b), c, (d, (e, f), g))$  的存储结构(e) 广义表  $E = (a, ( ), (( ), ( )), b)$  的存储结构

图 6-2 广义表的存储结构示例

广义表的基本算法（如构造、遍历等算法），将在 6.3 节详细讨论。在算法实现中，还涉及一些私有函数，也将在后文中详述。

## 6.3 广义表的操作算法

### 6.3.1 构造算法

在上述广义表类模板中，设计了两种构造函数。

- (1) 利用无参构造函数创建空的广义表。
- (2) 根据字符串表示的元素序列构造广义表。例如，根据元素序列“(a,(b,c,d),e)”构造广义表对象。

以下算法 6-1 是利用无参构造函数创建空的广义表。

#### 算法 6-1 初始化空广义表

```
template < class T >
GList < T > ::GList()
{
 head = new GListNode < T > ;
 head -> type = LIST;
 head -> sublist = head -> next = NULL;
}
```

以下算法 6-2 通过提取字符串序列中的基本元素和括号层次依次创建结点，逐层构建广义表。

#### 算法 6-2 根据字符串 s 中的元素序列构造广义表

```
template < class T >
GListNode < T > * GList < T > ::DoCreate(char s[], int &i)
{
 GListNode < T > * p;
 while(s[i] == ' ' || s[i] == ',')
 i ++; // 滤掉字符串 s 中的空格与逗号
 char e = s[i]; // 从字符串 s 中取出一个有效字符
 i ++;

 if(e == '(') // 构造子表结点
 {
 p = new GListNode < T > ;
 p -> type = LIST;
```

```

 p -> sublist = DoCreate(s, i);
 p -> next = DoCreate(s, i);
 return p;
}

if(e == ')' || e == '\0')
 return NULL;
p = new GListNode < T >; // 构造原子结点
p -> type = ATOM;
p -> data = e;
p -> next = DoCreate(s, i);
return p;
}

template < class T >
GList < T > ::GList(char s[])
{
 int i = 0;
 head = DoCreate(s, i);
}

```

### 6.3.2 遍历广义表

遍历广义表是对表中的每个数据元素访问且只访问一次。为简化问题，本节假设访问是输出操作。为了使输出显得全面、有条理，要求输出广义表的括号表达式，即原子结点的访问操作是输出其值，子表结点的访问操作是输出一对括号。

广义表的遍历过程既有同层元素的线性关系的遍历，也有子表结点的层次关系的遍历，这可以采用递归函数实现。以下算法 6-3 中的函数 Traverse(p) 的参数 p 是子表结点指针，函数依次访问 p 所指的子表中的每个结点。若是原子结点，则直接输出其值；若是子表结点，则输出一对括号，并调用递归函数以遍历子表。

#### 算法 6-3 广义表的遍历算法

```

template < class T >
void GList < T > ::Traverse(GListNode < T > * p)
{
 if(p == NULL)
 return;
 if(p -> type == ATOM)
 cout << p -> data;

```

```

else
{
 cout << '(';
 Traverse(p->sublist);
 cout << ')';
}

if(p->next)
 cout << ",";
 Traverse(p->next);

}

template < class T >
void GLList < T > ::Traverse()
{
 Traverse(head);
}

```

算法 6-3 中的函数 `Traverse(p)` 是 6.2.2 小节广义表类模板中的私有函数，实现遍历指针 `p` 所指的广义表的功能。函数 `Traverse()` 以头指针 `head`（私有成员变量）为参数，实现遍历当前对象中的整个广义表的功能。需要说明的是，在主调函数中调用遍历函数时，需将广义表对象的头指针 `head` 作为实参传入。但是，`head` 是私有数据成员，在主调函数中不能使用，即不能写成 `gla.Traverse(gla.head)`。因此，将该递归函数定义为私有成员函数，另外再定义一个公有无参函数 `Traverse()` 来供 `main()` 函数调用，而在 `Traverse()` 函数中调用 `Traverse(head)` 即可。

遍历算法是广义表的核心算法，它的程序框架可以为许多广义表应用算法所套用。若将访问操作理解成释放空间、申请空间等，则对应算法即变身为析构函数、拷贝构造函数。因此，请读者以图 6-2 中的广义表结构为例，由简单到复杂，逐一调试本程序，以便事半功倍地学习、掌握广义表的算法。

### 6.3.3 广义表的其他操作算法

#### 1. 复制广义表

复制广义表是根据已有广义表对象，创建一个结构、数据完全相同的广义表，最常见形式是拷贝构造函数。

在以下算法 6-4 中，函数 `Copy(p)` 的功能是复制一棵以 `p` 为头指针的广义表，返回新建广义表的头指针。显然，函数 `Copy(p)` 需要对以 `p` 为头指针的广义表进行遍历。在访问每个结点时，需要申请空间创建新结点，并参照已有广义表中结点对新结点赋值。因此，函数 `Copy(p)` 在整体上借用了广义表遍历的程序框架；在将若干新建数据结点链接成链表的步骤

中，使用了创建线性链表的技术。

#### 算法 6-4 广义表的复制算法

```
template < class T >
GListNode < T > * GList < T > ::Copy(GListNode < T > * p)
{
 if(p == NULL)
 return NULL;
 GListNode < T > * newp = new GListNode < T > ; // 创建一个结点
 newp -> type = p -> type; // 复制
 if(p -> type == LIST)
 newp -> sublist = Copy(p -> sublist);
 else
 newp -> data = p -> data;
 newp -> next = Copy(p -> next); // 继续复制下一个结点
 return newp;
}
template < class T >
GList < T > ::GList(GList < T > &gl)
{
 head = Copy(gl. head);
}
```

函数 Copy(p) 是私有函数，拷贝构造函数 GList(GList < T > &gl) 借助于已有对象的头指针 head 作为参数，调用 Copy 函数复制整个广义表结构，并将新建广义表的头指针赋值给当前对象的数据成员 head。

#### 2. 计算广义表的长度

在广义表的算法中，求广义表长度的难度最低。因为计算广义表的长度只需要考虑结点的 next 域指针。可以认为，该算法是求线性链表长度算法在广义表结构上的应用。以下算法 6-5 给出了计算广义表长度的具体实现。

#### 算法 6-5 计算广义表长度的算法

```
template < class T >
int GList < T > ::Length()
{
 GListNode < T > * p;
 n = 0;
 p = head -> sublist;
 while(p)
```

```

 {
 p = p -> next;
 n++;
 }
 return n;
}

```

### 3. 计算广义表的深度

广义表的深度定义为：原子结点的深度是0，广义表的深度是表中所有结点的深度的最大值加1。计算广义表深度必须首先计算每个结点的深度，而子表结点深度的计算方法与整个表深度的计算方法相同，显然，这可以借助广义表遍历程序框架实现。在以下算法 6-6 中，函数 Depth(p) 返回以 p 为头指针的广义表的深度。

#### 算法 6-6 计算广义表深度的算法

```

template < class T >
int GList < T > ::Depth(GListNode < T > * p)
{
 if(p -> type == ATOM)
 return 0;
 maxdepth = 0;
 GListNode < T > * q;
 q = p -> sublist;
 while(q)
 {
 depth = Depth(q);
 if(depth > maxdepth)
 maxdepth = depth;
 q = q -> next;
 }
 return maxdepth + 1;
}
template < class T >
int GList < T > ::Depth()
{
 return Depth(head);
}

```

函数 Depth(p) 是私有函数。函数 Depth() 利用 head 作为参数，计算并返回当前对象中整

个广义表的深度。

#### 4. 释放广义表的存储空间

广义表对象的建立是一个动态申请空间的过程，因此当对象析构时需要释放其中的所有结点空间，否则将造成内存泄露。每个结点必须被释放且仅被释放一次，因此析构算法也可以借助广义表遍历的程序框架。以下算法 6-7 中函数 Free(p) 的参数 p 是广义表头指针，函数功能是释放以 p 为头指针的广义表。

#### 算法 6-7 广义表的析构算法

```
template < class T >
void GLList < T > ::Free(GListNode < T > * p)
{
 if(p == NULL)
 return;
 if(p -> type == LIST)
 Free(p -> sublist);
 Free(p -> next);
 delete p;
}

template < class T >
GLList < T > :: ~ GLList()
{
 Free(head);
}
```

函数 Free(p) 是私有函数，析构函数 ~GLList() 利用 head 作为参数析构整个广义表结构。

## 本章小结

本章内容以广义表结构为中心展开，介绍了广义表的原子、子表、长度、深度、表头、表尾等概念，从不同侧面描述了广义表结构中的顺序关系和层次关系，同时也为广义表的创建和存取操作提供了方法。广义表的重点和难点是其中的层次关系，这也是几乎所有广义表算法的关键。遍历算法是广义表最重要的算法，它的程序框架为其他应用算法提供了样板。

本章的主要学习要点如下：

(1) 掌握广义表的定义。

(2) 重点掌握广义表的存储结构。

(3) 掌握广义表的基本运算，包括创建广义表、释放广义表、遍历广义表、求广义表的长度和深度。

(4) 灵活运用广义表结构解决一些综合应用问题。

## 习题 6

6.1 画出下列广义表的存储结构，写出其长度、深度以及表头和表尾：

$$A = (a, b, c)$$

$$B = (a, (b, (c)), d)$$

$$C = ((a, b), (c, d))$$

$$D = (a, (b, (), c), ((d), e))$$

$$E = (((a, b), (((), d), (e f)))$$

6.2 已知广义表  $A = (x, y, z)$ ,  $B = (a, (b, (c)), d)$ , 请分别画出满足以下条件的广义表结构：

(1) 以  $A$  为表头，以  $B$  为表尾。

(2) 以  $B$  为表头，以  $A$  为表尾。

## 上机实验题 6

广义表类的实现及应用。

基本要求：

(1) 采用链式存储结构实现。

(2) 以形式如"((a, b), (c, d), e, f)"的字符串作为参数，构造对象。

(3) 以菜单选择各项功能，如遍历输出广义表，计算广义表的深度、长度。

(4) 扩展广义表 class CList 的功能并进行测试。

① 替换算法：将广义表中某种原子值全部替换为指定值；

② 删除算法：删除广义表中所有值为指定值的原子结点。

# 第7章 树和二叉树

前面章节讨论的线性表、栈、队列、串等数据结构都属于线性结构，线性结构主要描述具有单一前驱和后继关系的数据对象。树形结构是一种比线性结构更复杂的、适合描述层次关系的数据对象。在客观世界中，许多事物的信息结构属于树形结构，如操作系统中的文件管理、互联网中的域名系统、编译程序中源程序的语法结构等。

树形结构不仅常用于具有层次关系的数据表示，而且表达了大多数问题求解的思路。问题的求解过程常被分解为若干小问题的求解过程，而每个小问题的求解又被分解为若干更小问题的求解。比较线性结构的学习，树形结构及其算法的学习难度有了较大提升。但是，读者掌握了这些内容之后，解决实际问题的能力也会有一个飞跃。

## 7.1 树的概念和性质

### 7.1.1 树的定义

树（Tree）是由  $n(n \geq 0)$  个结点组成的有限集合。若  $n = 0$ ，则称为空树。任意一棵非空树满足以下条件：

- (1) 有且仅有一个特定的称为根（Root）的结点；
- (2) 当  $n > 1$  时，除根结点以外的其余结点可划分为  $m(m > 0)$  个互不相交的有限集  $T_1, T_2, \dots, T_m$ ，其中每个集合又是一棵树，称为这个根的子树（SubTree）。

因为在树的定义中又用到树的定义，所以树的定义是递归的。它刻画了树的固有特性，即一棵树由若干棵子树构成，而子树又由更小的若干棵子树构成。

树是一种非线性数据结构。它的每一个结点可以有零个或多个后继，但有且仅有一个前驱（根结点除外）；这些数据结点按分支关系组织起来，清晰地反映了数据元素之间的层次关系。可以看出，数据元素之间存在的关系是一对多的关系。

图 7-1 所示的是 3 棵由简到繁的树结构。在图 7-1 (c) 中，结点  $a$  是根结点，其余结点被分成 3 个互不相交的子集  $T_1 = \{b, e, f, g\}, T_2 = \{c, h\}, T_3 = \{d, i, j\}$ ， $T_1, T_2, T_3$  称为结点  $a$  的子树。

上述的树结构定义通俗易懂，但其中涉及根、子树的定义还是不够清晰。下面采用集合形式给出树的形式定义：

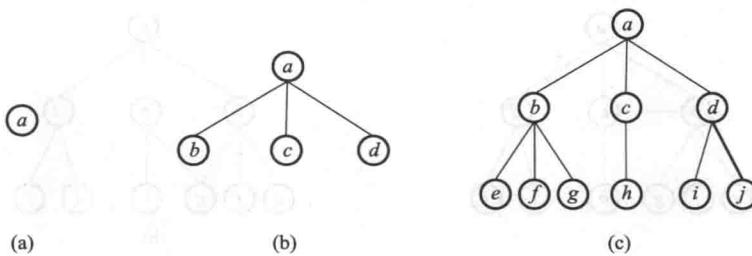


图 7-1 树形结构

$$\text{Tree} = (D, S)$$

$$D = \{a_i \mid a_i \in \text{ElemSet}, 1 \leq i \leq n\},$$

$S = \{\langle a_i, a_j \rangle \mid a_i, a_j \in \text{ElemSet}, 1 \leq i \leq n, 1 \leq j \leq n\}$ , 其中每个元素只有一个前驱, 可以有零个或多个后继, 有且仅有一个元素没有前驱}

ElemSet 是数据元素的集合。

(1) 当  $n=1$  时,  $S=\emptyset$ 。

(2) 当  $n>1$  时, 数据对象中存在唯一无前驱的数据元素, 称为根结点; 除根以外的结点存在如下划分:

- 可将  $D - \{\text{root}\}$  划分成  $m (m > 0)$  个互不相交的子集  $D_1, D_2, \dots, D_m$ , 对任意子集  $D_i$ , 唯一存在  $x_i \in D_i$ , 有  $\langle \text{root}, x_i \rangle \in S$ 。

- 对应  $D - \{\text{root}\}$  的划分, 可将  $S - \{\langle \text{root}, x_1 \rangle, \langle \text{root}, x_2 \rangle, \dots, \langle \text{root}, x_m \rangle\}$  唯一划分成互不相交的子集  $S_1, S_2, \dots, S_m$ ,  $S_i$  是  $D_i$  上的二元关系。 $(D_i, S_i)$  是一棵树, 称为根结点的子树。

数据对象  $D$  的定义表明树中数据元素的类型是相同的。比较复杂的是其中数据关系的定义, 下面以图 7-1 (c) 为例进行说明。

数据对象  $D = \{a, b, c, d, e, f, g, h, i, j\}$ , 数据关系以二元组集合的形式给出:  $S = \{\langle a, b \rangle, \langle a, c \rangle, \langle a, d \rangle, \langle b, e \rangle, \langle b, f \rangle, \langle b, g \rangle, \langle c, h \rangle, \langle d, i \rangle, \langle d, j \rangle\}$ 。显然, 唯一无前驱的数据元素是  $a$ , 因此  $a$  是根结点。除根结点外的结点集可分为  $D_1 = \{b, e, f, g\}$ ,  $D_2 = \{c, h\}$ ,  $D_3 = \{d, i, j\}$ 。与之对应的结构划分为  $S_1 = \{\langle b, e \rangle, \langle b, f \rangle, \langle b, g \rangle\}$ ,  $S_2 = \{\langle c, h \rangle\}$ ,  $S_3 = \{\langle d, i \rangle, \langle d, j \rangle\}$ 。如此逐层深入分解下去, 显然,  $(D_1, S_1)$ 、 $(D_2, S_2)$ 、 $(D_3, S_3)$  符合树的定义, 是  $a$  结点的子树。

需要注意的是, 虽然有些结构看似树结构, 但是其实不是。例如, 图 7-2 (a) 所示的结构较图 7-1 (c) 所示的结构增加了关系序偶  $\langle b, c \rangle$ , 图 7-2 (b) 所示的结构较图 7-1 (c) 所示的结构增加了关系序偶  $\langle c, g \rangle$ 。无论将新添序偶加入  $S_1$  或  $S_2$ , 都将导致  $(D_1, S_1)$  或  $(D_2, S_2)$  无法满足树的定义, 因此它们不属于树结构。

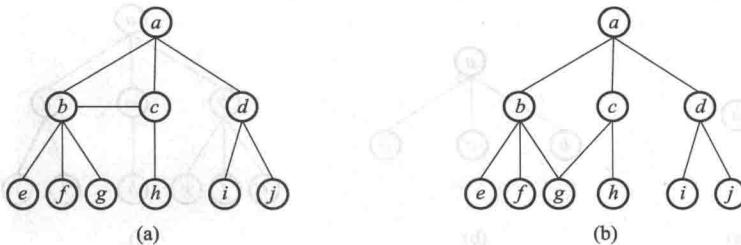


图 7-2 易混淆的非树形结构

## 7.1.2 树的基本术语

### 1. 结点的度和树的度

某结点所拥有的子树个数称为该结点的度 (Degree)。度为 0 的结点称为叶子结点 (Leaf)，度不为 0 的结点称为分支结点 (Branch)。例如，在图 7-1 (c) 中， $e, f, g, h, i, j$  是叶子结点， $a, b, c, d$  是分支结点，它们的度分别是 3、3、1、2。

在衡量一棵树中全部结点的度时，使用树的度这一概念，它是该树中所有结点度的最大值。通常将度为  $m$  的树称为  $m$  叉树。例如，图 7-1 (c) 所示的树的度是 3。

### 2. 孩子、双亲、兄弟结点

某结点的子树的根结点被称为该结点的孩子结点 (Child)，该结点被称为其孩子结点的双亲结点 (Parent)。具有同一个双亲结点的孩子结点互称兄弟结点 (Sibling)。例如，在图 7-1 (c) 中， $a$  是  $b, c, d$  的双亲结点， $b$  是  $e, f, g$  的双亲结点， $e, f, g$  互称兄弟结点。

### 3. 路径和路径长度

对于任意两个结点  $k_i$  和  $k_j$ ，若树中存在一个结点序列  $k_i, k_{i1}, k_{i2}, \dots, k_{in}, k_j$ ，使得序列中除  $k_j$  外的任一结点都是其在序列中的前一个结点的后继，则称该结点序列为从  $k_i$  到  $k_j$  的路径 (Path)。路径上经过的边的个数称为路径长度 (Path Length)。显然，从根结点到每个结点的路径是唯一的。例如，在图 7-1 (c) 中，从根结点到  $f$  结点的路径是  $abf$ ，路径长度是 2。

### 4. 子孙结点和祖先结点

每个结点的所有子树中的结点被称为该结点的子孙结点。从根结点到达某结点的路径上经过的所有结点 (除自身外) 被称为该结点的祖先结点。例如，在图 7-1 (c) 中，除根结点外的所有结点都是根结点的子孙结点，结点  $g$  的祖先结点是  $b$  和  $a$ ，所有结点共同的祖先结点是根结点。

### 5. 结点的层次和树的高度

结点的层次是这样定义的：根结点的层次是 1；若某结点的层次是  $i$ ，则其孩子结点的层次为  $i+1$ 。树的高度是树中所有结点的层次的最大值。树的高度还有一种等价的定义方式：设空

树高度为 0，非空树的高度等于所有子树高度的最大值加 1。例如，在图 7-1 (a) 所示的树的高度是 1，图 7-1 (b) 所示的树的高度是 2，图 7-1 (c) 所示的树的高度是 3。

### 6. 有序树和无序树

在一棵树中，如果结点的各子树从左到右是有序的，即交换了结点各子树的相对位置后构成了不同的树，则称这棵树为有序树（Ordered Tree）；反之，称为无序树（Unordered Tree）。若为有序树，则图 7-3 (a) 所示的和图 7-3 (b) 所示的是两棵不同的树；若为无序树，则图 7-3 (a) 所示的和图 7-3 (b) 所示的为同一棵树。实际中，无序树和有序树的区别主要取决于实际应用的意义。除特殊说明外，在数据结构中讨论的树一般都是有序树。

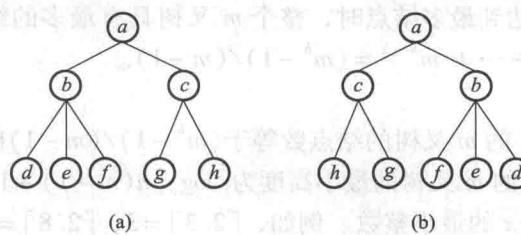


图 7-3 有序树和无序树

### 7. 森林

$m (m \geq 0)$  棵互不相交的树的集合称为森林（Forest）。需要注意的是，森林中的所有树之间没有共同的数据元素。当  $m = 1$  时，森林就退化成了一棵树；当  $m = 0$  时，表示一个空森林。

森林的定义说明，可以将树的算法经过重复迭代扩展为森林的算法。这个问题留待读者在算法学习之余探索思考。

### 7.1.3 树的基本性质

**性质 1** 树中的结点总数等于所有结点的度之和加 1。

**证明** 根据树的定义，在一棵树中，除根结点之外，每个结点有且只有一个双亲结点。也就是说，每个结点和指向它的一个分支是一一对应的。因此，除根结点之外的结点总数等于所有结点的分支数，从而可得树中的结点总数等于所有结点的度之和加 1。

证毕。

性质 1 展现了根结点在树结构中的特殊性。

**性质 2** 在  $m$  叉树中，第  $i$  层上最多有  $m^{i-1}$  个结点 ( $i \geq 1$ )。

**证明** 采用数学归纳法证明：

① 当  $i = 1$  时，第 1 层只有一个根结点， $m^{i-1} = m^0 = 1$ ，命题成立。

② 假设对于树的第  $i - 1$  层，本命题成立，即在  $m$  叉树中第  $i - 1$  层至多有  $2^{i-2}$  个结点。根

据度的定义，在 $m$ 叉树中，每个结点最多有 $m$ 个孩子。因此，第*i*层上的结点数最多为第*i*-1层上结点数的 $m$ 倍，即最多为 $m^{i-2} \times m = m^{i-1}$ 。

**证毕。**

**推广** 当一棵 $m$ 叉树的第*i*层有 $m^{i-1}$ ( $i \geq 1$ )个结点时，称该层是满的。若一棵 $m$ 叉树的每一层都是满的，则称其为满 $m$ 叉树。显然，在所有高度相同的 $m$ 叉树中，满 $m$ 叉树的结点总数最多。

**性质3** 高度为 $h$ 的 $m$ 叉树最多有 $(m^h - 1)/(m - 1)$ 个结点。

**证明** 由树的性质2可知，第*i*层上最多的结点数为 $m^{i-1}$ ( $i = 1, 2, \dots, h$ )。显然，当高度为 $h$ 的 $m$ 叉树上每一层都达到最多结点时，整个 $m$ 叉树具有最多的结点数。 $m$ 叉树各层最多结点数之和 $m^0 + m^1 + m^2 + \dots + m^{h-1} = (m^h - 1)/(m - 1)$ 。

**证毕。**

**推广** 当一棵高度为 $h$ 的 $m$ 叉树的结点数等于 $(m^h - 1)/(m - 1)$ 时，称该树为满 $m$ 叉树。

**性质4** 具有 $n$ 个结点的 $m$ 叉树的最小高度为 $\lceil \log_m [n(m-1) + 1] \rceil$ 。

注： $\lceil x \rceil$ 表示大于等于 $x$ 的最小整数。例如， $\lceil 2.3 \rceil = 3$ ,  $\lceil 2.8 \rceil = 3$ 。

**证明** 设具有 $n$ 个结点的某 $m$ 叉树的高度为 $h$ ，若该树中前 $h-1$ 层都是满的，无论第 $h$ 层满或不满，该树具有最小的高度。由性质3得，如此形态的 $m$ 叉树的结点数最多为

$$(m^h - 1)/(m - 1)$$

即

$$n \leq (m^h - 1)/(m - 1)$$

化简得

$$n(m-1) + 1 \leq m^h$$

取对数得

$$\log_m(n(m-1) + 1) \leq h$$

即树的最小高度值

$$h = \lceil \log_m(n(m-1) + 1) \rceil$$

**证毕。**

例如，对于具有 $n$ 个结点的二叉树，其最小高度 $h = \lceil \log_2(n+1) \rceil$ ，若 $n=20$ ，则最小高度 $h = \lceil \log_2 21 \rceil = 5$ 。对于三叉树，其最小高度 $h = \lceil \log_3(2n+1) \rceil$ ，若 $n=20$ ，则最小高度 $h = \lceil \log_3 41 \rceil = 4$ 。

## 7.2 二叉树的概念和性质

### 7.2.1 二叉树的定义

二叉树结构是一种简化的树形结构。它是由 $n(n \geq 0)$ 个结点组成的有限集合，该集合或者为空，或者是由一个根结点和两棵互不相交的称为左子树和右子树的二叉树组成。

二叉树的形式化定义类似树的形式化定义，即在数据对象中，可将除根结点以外的结点和关系划分成  $m (m \leq 2)$  个互不相交的子集  $(D_i, S_i)$ 。每个  $(D_i, S_i)$  构成一棵二叉树，称为左子树和右子树。图 7-4 所示的是 3 棵由简到繁的二叉树结构。

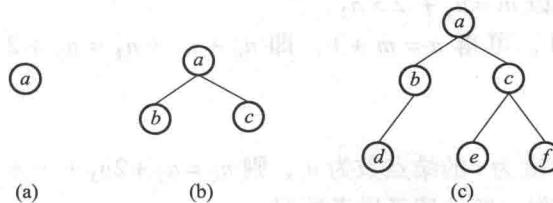


图 7-4 二叉树结构的示意图

二叉树的特点是每个结点的度只可能是 0 或 1 或 2。二叉树是有序树，即使某结点只有一棵子树，也要区分该子树是左子树还是右子树。图 7-5 (a)、(b) 所示是不同的二叉树。

二叉树的结构相对简单，其运算也自然简单，便于初学者入门。由于多叉树可以借助一定的规则转换为二叉树结构，因此二叉树结构在应用中具有非常重要的地位。

二叉树具有 5 种基本形态，图 7-6 (a) 所示的是空二叉树，图 7-6 (b) 所示的二叉树只有一个根结点，图 7-6 (c) 所示的二叉树的根结点只有左子树，图 7-6 (d) 所示的二叉树根结点只有右子树，图 7-6 (e) 所示的二叉树根结点同时有左子树和右子树。

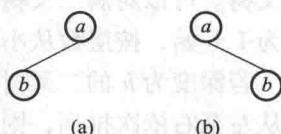


图 7-5 两棵不同的二叉树

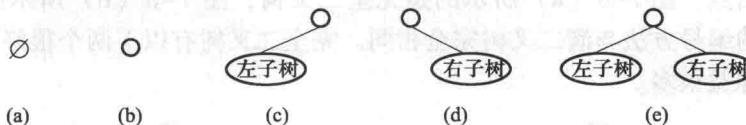


图 7-6 二叉树的 5 种基本形态

## 7.2.2 二叉树的基本性质

二叉树中的结点数和深度性质为二叉树存储结构的选择提供了预备知识和思考空间。

**性质 1** 非空二叉树中，第  $i$  层最多有  $2^{i-1} (i \geq 1)$  个结点。

由树的性质 2 可推出。

**性质 2** 高度为  $h$  的二叉树中结点总数最多为  $2^h - 1$ 。

由树的性质 3 可推出。

**性质 3** 设某二叉树中叶子结点数为  $n_0$ ，度为 2 的结点数为  $n_2$ ，则  $n_0 = n_2 + 1$ 。

**证明** 首先考察二叉树的结点总数。设二叉树中度为1的结点数是 $n_1$ ，结点总数记作 $n$ 。因为二叉树中所有结点的度只能是0、1、2，所以结点总数 $n = n_0 + n_1 + n_2$ 。

其次考察二叉树的分支总数。将二叉树的分支总数记作 $m$ 。因为所有的分支是由度为1和度为2的结点发出的，所以 $m = n_1 + 2 \times n_2$ 。

最后，由树的性质1，可得 $n = m + 1$ ，即 $n_0 + n_1 + n_2 = n_1 + 2 \times n_2 + 1$ 。化简得 $n_0 = n_2 + 1$ 。

证毕。

**推广** 设某 $m$ 叉树，度为 $i$ 的结点数为 $n_i$ ，则 $n_0 = n_2 + 2n_3 + \dots + (m-1)n_m + 1$ 。

证明方法与性质3相似，留给读者思考练习。

由满 $m$ 叉树的定义可知，高度为 $h$ 且有 $2^h - 1$ 个结点的二叉树称为满二叉树。它的特点是，每一层上的结点数都达到最大结点数，所有分支结点都有左、右子树，所有的叶子结点都集中在底层。图7-7所示的是一棵满二叉树。可以对满二叉树的结点进行连续编号，约定编号从根结点为1开始，按层数从小到大、同层中从左到右顺序编号。

若深度为 $h$ 的二叉树，前 $h-1$ 层都是满的，并且第 $h$ 层的结点从左至右依次排列，则称为完全二叉树。可以将满二叉树理解成完全二叉树的特殊情形，即满二叉树是底层结点数满的完全二叉树。完全二叉树有这样的性质：底层的结点一定是从左至右依次排列的；叶子结点只可能在最低两层；只有最低两层的结点的度可能是0或1，其余层的结点一律是双分支结点。图7-8(a)所示的是完全二叉树，图7-8(b)所示的不是完全二叉树。完全二叉树的编号方法与满二叉树完全相同。完全二叉树有以下两个很好的性质，对于其存储结构的选择裨益很多。

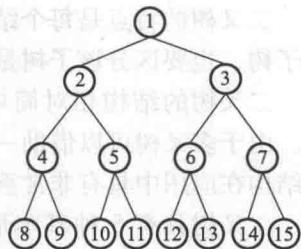


图7-7 满二叉树

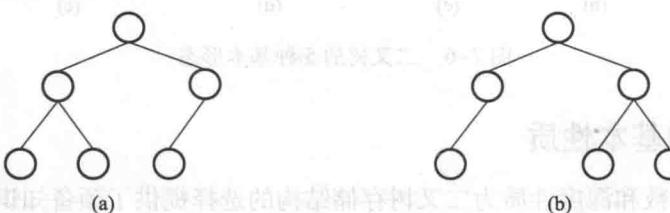


图7-8 完全二叉树和非完全二叉树

**性质4** 具有 $n$ 个结点的完全二叉树的高度为 $\lfloor \log_2 n \rfloor + 1$ 。

注： $\lfloor x \rfloor$ 表示小于等于 $x$ 的最大整数。例如， $\lfloor 5.2 \rfloor = 5$ ， $\lfloor 5.8 \rfloor = 5$ 。

**证明** 在有 $n$ 个结点的各种二叉树结构中，完全二叉树是高度最小的二叉树结构。设高度

为  $h$ , 由二叉树性质 2 和完全二叉树的定义有

$$2^{h-1} - 1 < n \leq 2^h - 1 \text{ 或 } 2^{h-1} - 1 \leq n < 2^h$$

因此有  $h - 1 \leq \log_2 n < h$ 。因为  $h$  是整数, 所以  $h = \lfloor \log_2 n \rfloor + 1$ 。

由 7.1.3 小节中树的性质 4 可知, 具有  $n$  个结点的二叉树的最小高度为  $\lceil \log_2(n+1) \rceil$ , 因此相应的完全二叉树的高度就是  $\lceil \log_2(n+1) \rceil$ 。

证毕。

**性质 5** 对有  $n$  个结点的完全二叉树编号后, 第  $i$  个结点 ( $1 \leq i \leq n$ ) 的编号, 有如下性质:

- ① 若  $i > 1$ ,  $i$  的双亲结点编号是  $\lfloor i/2 \rfloor$ 。
- ② 若  $2i < n$ ,  $i$  的左孩子结点编号是  $2i$ ; 否则, 结点  $i$  无左孩子结点。
- ③ 若  $2i+1 < n$ ,  $i$  的右孩子结点编号是  $2i+1$ ; 否则, 结点  $i$  无右孩子结点。

图 7-9 所示的是性质 5 的示例。

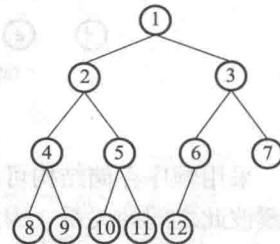


图 7-9 完全二叉树中  
结点的编号关系

## 7.3 二叉树的存储结构

### 7.3.1 二叉树的顺序存储结构

二叉树的顺序存储结构是用一组地址连续的存储单元来存放二叉树的数据元素。在这种存储结构中, 必须确定好二叉树中各数据元素的存放次序, 使得各数据元素的存放位置能反映出数据元素之间的关系。

二叉树的性质 5 为二叉树的顺序存储指明了存储规则, 即依照完全二叉树的结点编号次序依次存放各个结点。需要注意的是, C/C++ 中数组的起始地址为 0, 编号为  $i$  的结点存储在下标为  $i-1$  的单元内。

图 7-10 (a) 所示为完全二叉树, 图 7-10 (b) 所示是其顺序存储结构, 连续的存储空间共有 7 个单元, 存储了 6 个数据元素。

二叉树顺序存储的类模板定义如下, 这里只写出了类中的主要数据成员:

```
template < class T, int MaxSize >
class BiTree_Seq
{
 T Data[MaxSize]; // 连续空间的起始地址
```

```

int length; // 顺序存储的结点个数
...
|;

```

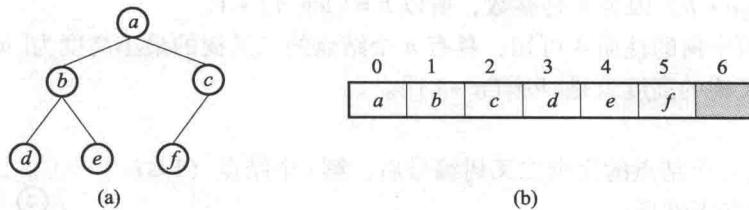


图 7-10 完全二叉树的顺序存储结构

采用顺序存储结构可以直接存取二叉树中的任意数据元素。由于每个数据元素的存储位置暗藏彼此之间的关系，因此可以根据结点的编号直接计算出它的父结点、左右孩子结点的位置。类似地，结点的查找、统计，结点路径的识别都能非常便捷地计算出来。

对于满二叉树、完全二叉树来说，顺序存储结构的存储效率是极高的，所有的空间仅仅用来存储数据元素的值，结点之间关系的存储未占用任何空间。

但是，对于一般二叉树而言，如何利用完全二叉树的编码规则来存储数据呢？解决的方法是补足不存在的结点，用特殊数据标识这些替补结点，使整棵树在形式上满足完全二叉树的定义。图 7-11（a）所示的不是完全二叉树。经过增补一些虚拟结点后，图 7-11（b）所示成为完全二叉树，其顺序存储结构如图 7-11（c）所示。

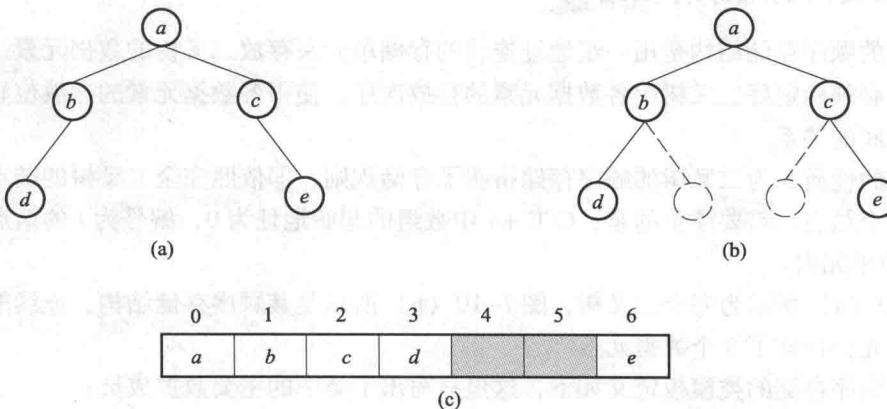


图 7-11 非完全二叉树的顺序存储结构

虽然非完全二叉树采用顺序存储结构存在部分内存空间的浪费，但是如果空间浪费不多，能得到直接存取的优点，那么也是值得的。对于那些单分支结点较多、高度变化较大的二叉树

而言，顺序存储结构是不合适的。如图 7-12 (a) 所示的二叉树每个结点只有右子树，为其增补虚拟结点后如图 7-12 (b) 所示。为了存储 4 个数据元素，需要占用 15 个存储空间，如图 7-12 (c) 所示。若二叉树的高度更大，则空间浪费现象将更加惊人。因此，对于一般二叉树通常采用下一小节介绍的链式存储结构。

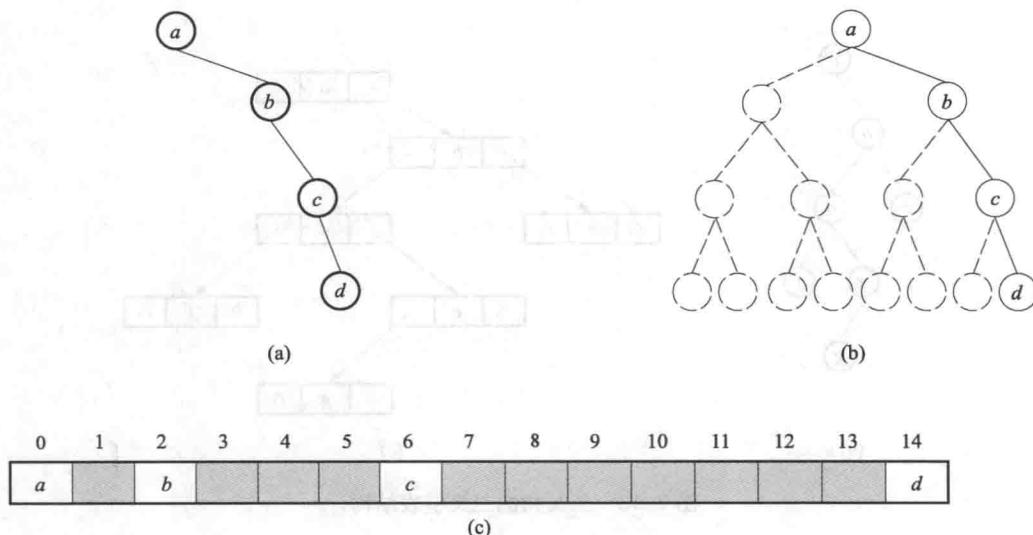


图 7-12 退化的二叉树的顺序存储结构

### 7.3.2 二叉树的链式存储结构

#### 1. 二叉链表结构

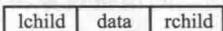
在线性链表中，每个结点只有一个后继结点，因此结点中只包含一个指针域。二叉树中的每个结点有两个孩子（后继）结点，为便于向下检索，结点结构中应该包含两个指针域，分别指向左、右孩子结点，这就是二叉链表结构。结点结构如图 7-13 (a) 所示。图 7-13 (b) 所示的是某二叉树的逻辑结构图，图 7-13 (c) 所示的是其对应的二叉链表结构。

二叉链表结构定义如下：

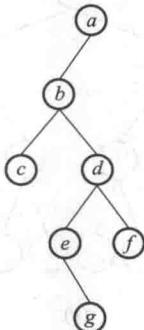
```
template < class T >
struct BiNode
{
 T data; // 结点数据
 BiNode < T > * lchid; // 左孩子的指针
```

```
BiNode < T > * rchild; // 右孩子的指针
```

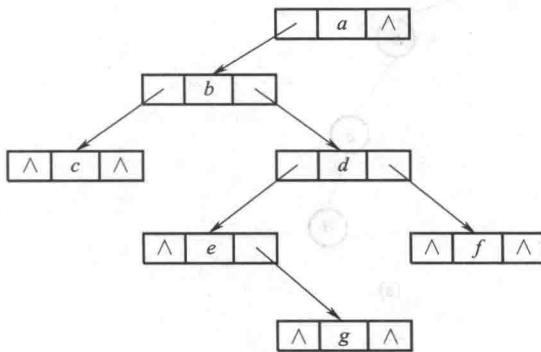
```
};
```



(a)



(b)



(c)

图 7-13 二叉树的二叉链表结构

如图 7-13 (c) 所示, 可以发现其中有许多空指针域。设二叉树有  $n$  个结点, 则有  $2n$  个指针域。由于除根结点外每个结点都有一个指针指向它, 即只有  $n - 1$  个指针域不为空, 因此共有  $n + 1$  个指针域为空。如何充分利用这些指针域将在后面详细讨论。

## 2. 三叉链表结构

在二叉树的二叉链表结构中, 检索结点的孩子结点非常方便, 但检索结点的父结点、祖先结点就困难了许多。为便于向上检索, 需要在结点结构中增加一个指针域, 包含双亲结点的地址。这样的结构被称为三叉链表结构。其结构如图 7-14 (a) 所示。图 7-14 (c) 所示的是图 7-14 (b) 所示二叉树的三叉链表结构图。

三叉链表结构定义如下:

```
template < class T >
struct TriNode
{
 T data; // 结点数据
 TriNode < T > * parent; // 双亲结点的指针
 TriNode < T > * lchild; // 左孩子的指针
```

```
 TriNode < T > * rchild; // 右孩子的指针
};
```

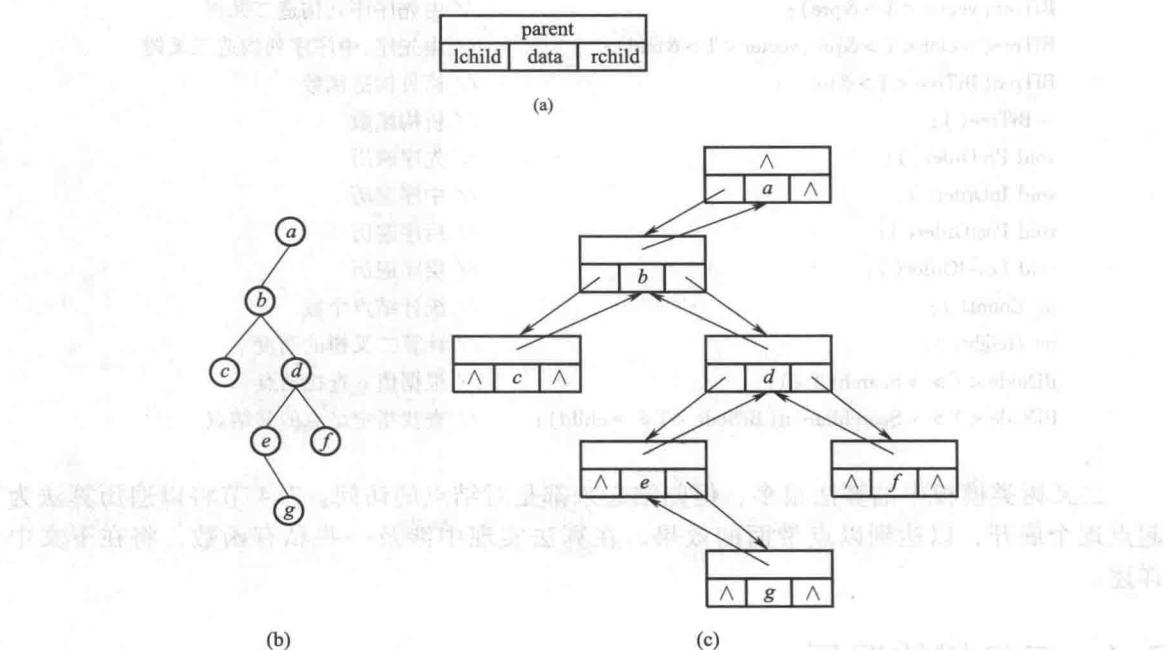


图 7-14 二叉树的三叉链表结构

三叉链表结构和二叉链表结构大同小异，主要区别在于三叉链表结构多占用了一些内存空间。在涉及查找祖先结点的算法中，三叉链表的运算方便了许多，效率也有很大的提高。

在下面的算法讨论中，除特殊说明外，二叉树一律采用二叉链表存储结构。

二叉树的类模板是本章最重要的数据结构。下面以二叉链表结构为基础构造二叉树的类模板 BiTree。该类中核心数据成员 root 是根结点的指针，简称根指针。对二叉树中任意结点的访问都可以通过根指针沿特定的路径实现。

```
template < class T >
class BiTree
{
private:
 BiNode < T > * root; // 根指针
}
```

```

...
// 涉及的私有成员函数见 7.4 节和 7.5 节

public:
 BiTree() { root = NULL; } // 无参构造函数, 构造空树
 BiTree(vector < T > &pre); // 由先序序列构造二叉树
 BiTree(vector < T > &pre, vector < T > &mid); // 由先序、中序序列构造二叉树
 BiTree(BiTree < T > &tree); // 拷贝构造函数
 ~BiTree(); // 析构函数
 void PreOrder(); // 先序遍历
 void InOrder(); // 中序遍历
 void PostOrder(); // 后序遍历
 void LevelOrder(); // 层序遍历
 int Count(); // 统计结点个数
 int Height(); // 计算二叉树的高度
 BiNode < T > * Search(T e); // 根据值 e 查找结点
 BiNode < T > * SearchParent(BiNode < T > * child); // 查找指定结点的父结点
;

```

二叉树类模板中的算法很多，但归结起来都是对结点的访问。7.4节将以遍历算法为起点逐个展开，以达到以点带面的效果。在算法实现中涉及一些私有函数，将在下文中详述。

## 7.4 二叉树的遍历

在二叉树的许多应用中，常常要求在二叉树中查找具有某种特征的结点，或者对二叉树中全部结点逐一进行某种处理。这就提出了一个遍历二叉树的问题，即按一定规则访问二叉树中的每个结点，且每个结点只能访问一次。这里所说的访问的含义很广，可以是对结点的数据和关系进行调整、统计的多种操作。在以下算法讨论和实现中，将访问定义为输出操作，以达到简化数据运算，强调结构操作的目的。

二叉树比线性结构复杂的是，由于它的每一个结点可能有两个后继，因此它的遍历规则不是显而易见的。本节将讨论 4 种遍历规则：先序遍历、中序遍历、后序遍历和层次遍历。遍历算法的结果是访问过程中结点被访问的序列，即遍历序列。遍历序列是二叉树结构线性化的表示，它因遍历规则不同而各异。

遍历算法是二叉树操作算法的基础。下面以图 7-15 所示的二叉树为例，分别说明这四种遍历算法。

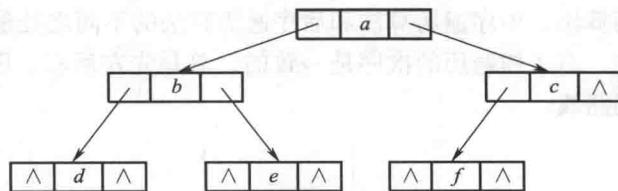


图 7-15 二叉树遍历示例

### 7.4.1 二叉树遍历的概念

#### 1. 先序遍历、中序遍历和后序遍历

先序遍历、中序遍历和后序遍历算法的思路紧扣二叉树的定义，存在许多共性。由于二叉树由根结点、左子树、右子树组成，因此遍历二叉树自然就转换成了访问根结点和对左、右子树的遍历。在具体遍历根结点、左子树、右子树的过程中，存在先后次序的区别，即依据访问根结点的位置在先、在中、在后，分别对应了二叉树的先序遍历算法、中序遍历算法和后序遍历算法。

先序遍历算法的步骤如下：

- ① 若二叉树为空，则遍历结束；
- ② 访问根结点；
- ③ 先序遍历根结点的左子树；
- ④ 先序遍历根结点的右子树。

针对图 7-15 所示的二叉树先序遍历，得到先序遍历序列 *abdefc*。

中序遍历算法的步骤如下：

- ① 若二叉树为空，则遍历结束；
- ② 中序遍历根结点的左子树；
- ③ 访问根结点；
- ④ 中序遍历根结点的右子树。

针对图 7-15 所示的二叉树中序遍历，得到中序遍历序列 *dbeafc*。

后序遍历算法的步骤如下：

- ① 若二叉树为空，则遍历结束；
- ② 后序遍历根结点的左子树；
- ③ 后序遍历根结点的右子树；
- ④ 访问根结点。

针对图 7-15 所示的二叉树后序遍历，得到后序遍历序列 *debfa*。

二叉树的先序遍历算法、中序遍历算法和后序遍历算法的不同之处是访问根结点的时机不同；其共同之处是对左、右子树遍历的次序是一致的，总是先左后右。图 7-16 中的虚线就是这三种遍历算法的共同路线。

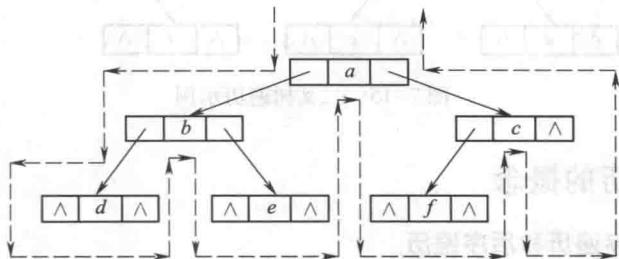


图 7-16 二叉树先、中、后序遍历的流程

二叉树的先序遍历算法、中序遍历算法和后序遍历算法看似简单，但其中的递归深度和执行步骤极易使初学者混淆。初学者可参考图 7-17 所示的结构，由简到繁，逐个练习遍历过程。在对左、右子树进行遍历时，要记录当前步骤和递归深度，以锻炼和提升自己的层次思维能力。熟练掌握遍历算法规则，必能为学习、理解二叉树的应用算法带来事半功倍的效果。

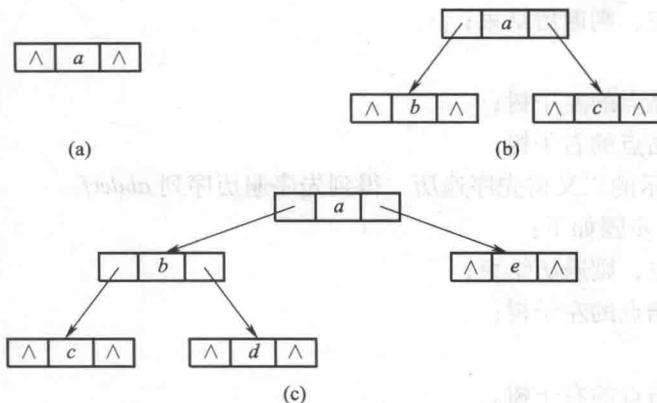


图 7-17 遍历算法的测试案例“由简到繁的二叉树”

二叉树的先序遍历、中序遍历和后序遍历有广泛的应用。图 7-18 所示的是一棵表达式树，其中叶子结点存储了运算数，分支结点存储了运算符，存储的表达式是  $a * b + c/d$ 。该二叉树的先序序列是  $+ * ab/cd$ ，称为前缀表达式；中序序列是  $a * b + c/d$ ，就是常用的中缀表达式；后序序列为  $ab * cd/ +$ ，称为后缀表达式。当表达式采用二叉树结构存储时，利用遍历算法可以实现快速求值运算（无须进行算符优先级比较），在时间、空间效率上比单纯利用栈对

字符串表达式求值都有所提高。

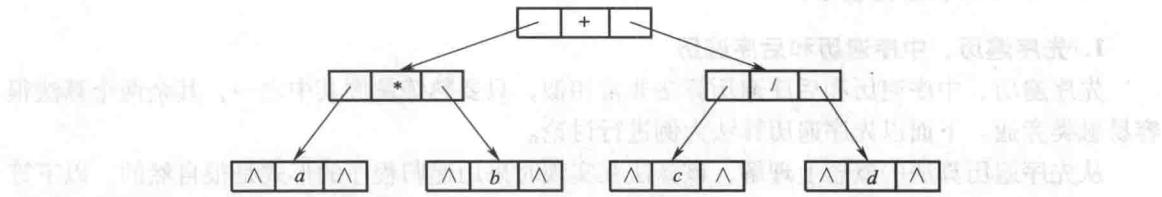


图 7-18 表达式树

## 2. 层次遍历

层次遍历的规则是从根结点开始从上到下逐层遍历，在同一层中则按从左到右的顺序对结点依次访问。对图 7-15 所示的二叉树，层次序列是 *abcdef*。二叉树层次遍历算法的步骤如下：

- ① 若二叉树为空，遍历结束。
- ② 将根指针加入指针队列。
- ③ 若指针队列不空，则执行步骤④；若指针队列为空，则遍历结束。
- ④ 出队列，取队首元素，记队首元素所指的结点为当前结点，访问当前结点。
- ⑤ 若当前结点的左孩子指针不空，则将左孩子地址进指针队列；若当前结点的右孩子指针不空，则将右孩子地址进指针队列。
- ⑥ 转步骤③。

层次遍历的结果很直观，但算法描述却使人感觉有些冗长。这是因为图例中的二叉树都很简单，一眼望去同层关系非常直观。假设一棵有 10 000 个结点的二叉树存储在内存中，每个结点只存储自身孩子结点的地址，那么应当以什么为依据查找同层结点呢？这就不简单了。

层次遍历算法借助一个指针队列作为辅助结构存储曾经访问过的结点的地址序列，以便实现对这些结点的孩子结点的依次访问。因此，算法中频繁地进队列、出队列就成了顺理成章的事。

在指针队列的变化过程中，每个结点的指针都要进一次队列和出一次队列。每一次的进队列操作就像是将指针所指的子树的遍历问题提交给了队列。队列的先进先出特性很自然使问题按先后次序排队，从而实现了从左到右、从上到下的遍历效果。每一次的出队列操作，都访问一个结点，这不仅保证了对每个结点的访问，而且保证了每个结点只被访问一次的遍历要求。

为了较好地掌握层次遍历，请读者以图 7-16 为例逐步练习算法步骤，仔细观察指针队列的变化过程，切不要以会写层次遍历序列为满足。

### 7.4.2 二叉树遍历算法

#### 1. 先序遍历、中序遍历和后序遍历

先序遍历、中序遍历和后序遍历算法非常相似，只要熟练掌握其中之一，其余两个算法很容易触类旁通。下面以先序遍历算法为例进行讨论。

从先序遍历算法的概念上理解，该算法在实现时采用递归程序的形式是很自然的。以下算法 7-1 采用成员函数的形式，给出先序遍历算法的递归程序。函数 PreOrder(p) 先序遍历以 p 为根指针的二叉树。

#### 算法 7-1 二叉树的先序遍历算法

```
template < class T >
void BiTree < T > ::PreOrder(BiNode < T > * p)
{
 if(p == NULL)
 return; // ①若二叉树为空，则遍历结束
 cout << p -> data; // ②访问当前结点
 PreOrder(p -> lchild); // ③先序遍历当前结点的左子树
 PreOrder(p -> rchild); // ④先序遍历当前结点的右子树
}
template < class T >
void BiTree < T > ::PreOrder()
{
 PreOrder(root);
}

函数 PreOrder(BiNode < T > * p) 是 7.3.2 小节中 class BiTree 类模板未提及的私有函数，功能是先序遍历以 p 为根指针的二叉树。需要注意的是，在实现 7.3.2 小节的类模板程序时，还需要补充 PreOrder(BiNode < T > * p) 等私有成员函数的声明。
```

在主调函数中调用先序遍历函数时，需要将二叉树的根结点指针 root 作为实参传入。但是，root 是私有数据成员，在 main( ) 函数中不能使用，即不能写成 bitree.PreOrder( bitree.root )。因此，将该递归函数定义为私有成员函数，另外再定义一个公有无参函数 PreOrder( ) 来供主调函数调用，而在 PreOrder( ) 函数中调用 PreOrder( root ) 函数即可。

函数 PreOrder( p ) 中的 4 条语句分别对应 7.4.1 小节中先序遍历算法的 4 个步骤。语句②实现最简单的访问功能，语句③、④分别实现对左右子树的遍历。若将语句②置于语句③、④之间，则该函数实现中序遍历功能；若将语句②置于语句③、④之后，则该函数实现后序遍历功能。

先序遍历、中序遍历和后序遍历函数的形式看似简单，其实颇有难度。其中难度在于理解遍历中的函数调用的层次关系和指令执行的顺序关系，这是树形结构中的关键思维训练。

为全面展示先序遍历过程，以图 7-17 中的表达式树为例制作了表 7-1。表中序号对应着函数 PreOrder( ) 中的语句序号，从上到下的各行是函数的指令执行序列，向右缩进的表行是上一层递归调用语句的执行细节，向右缩进的深度是递归调用的深度。建议初学者参照表 7-1 格式，针对图 7-17 所示的由简到繁的二叉树结构，进行这样的制表练习，以达到熟练掌握二叉树遍历程序框架的目的。

表 7-1 函数 PreOrder( ) 的执行过程

| 函数的指令执行序列       |                        | 符号说明           |
|-----------------|------------------------|----------------|
| PreOrder( & + ) |                        | & + 表示结点 + 的指针 |
| ①               | if( p == NULL) return; |                |
| ②               | cout << p -> data;     | 访问结点 +         |
| ③               | PreOrder( & * );       | & * 表示结点 * 的指针 |
| ①               | if( p == NULL) return; |                |
| ②               | cout << p -> data;     | 访问结点 *         |
| ③               | PreOrder( &a );        | &a 表示结点 a 的指针  |
| ①               | if( p == NULL) return; |                |
| ②               | cout << p -> data;     | 访问结点 a         |
| ③               | PreOrder( NULL );      | 遍历空的左子树        |
| ④               | PreOrder( NULL );      | 遍历空的右子树        |
| ④               | PreOrder( &b );        | &b 表示结点 b 的指针  |
| ①               | if( p == NULL) return; |                |
| ②               | cout << p -> data;     | 访问结点 b         |
| ③               | PreOrder( NULL );      | 遍历空的左子树        |
| ④               | PreOrder( NULL );      | 遍历空的右子树        |
| ④               | PreOrder( &/ );        | &/ 表示结点 / 的指针  |
| ①               | if( p == NULL) return; |                |
| ②               | cout << p -> data;     | 访问结点 *         |
| ③               | PreOrder( &c );        | &c 表示结点 c 的指针  |

续表

| 函数的指令执行序列 |                        | 符号说明          |
|-----------|------------------------|---------------|
| ①         | if( p == NULL) return; |               |
| ②         | cout << p -> data;     | 访问结点 c        |
| ③         | PreOrder( NULL);       | 遍历空的左子树       |
| ④         | PreOrder( NULL);       | 遍历空的右子树       |
| ④         | PreOrder( &d);         | &d 表示结点 d 的指针 |
| ①         | if( p == NULL) return; |               |
| ②         | cout << p -> data;     | 访问结点 d        |
| ③         | PreOrder( NULL)        | 遍历空的左子树       |
| ④         | PreOrder( NULL)        | 遍历空的右子树       |

仅观察先序遍历、中序遍历和后序遍历算法的 4 行代码很难计算出它们的时间复杂度和空间复杂度。但从程序流程来观察，因为每个结点访问且仅访问一次，所以假设二叉树中有  $n$  个结点，在若干次递归调用中结点的访问操作必然进行了  $n$  次。因此，这三种遍历算法的时间复杂度为  $O(n)$ 。

在遍历程序的代码中几乎看不到使用的辅助空间，这样容易使人误以为遍历算法有极优的空间复杂度。实际上，每次函数调用都会产生系统内部的函数调用栈的进栈操作。函数调用栈必须保存函数返回地址、参数、局部变量等信息。因此，函数调用过程中消耗的内存和递归调用的深度成正比关系。由表 7-1 可以看到，遍历中递归调用的深度等于二叉树的高度。因此，若设二叉树的高度为  $h$ ，则遍历算法的空间复杂度应该为  $O(h)$ 。

先序遍历、中序遍历和后序遍历算法的实现也有非递归程序的形式。这需要在程序中创建一个辅助的栈空间来实现遍历左、右子树和回溯的过程。非递归程序具有和递归程序一样的时间复杂度和空间复杂度，只是实现细节稍显繁琐，学习难度也有些提高。但是相比递归程序，非递归程序有了更多的程序控制权，因此应用更加灵活。

由于先序遍历、中序遍历和后序遍历算法的递归程序的框架逻辑清晰、简明易懂，且具有广泛的应用价值，因此非常适合初学者学习与练习。下面的大多数程序都参照了此程序框架。

## 2. 层次遍历

在层次遍历算法的实现中需要使用队列结构。为简化程序细节，可以使用第 3 章中的队列结构，也可以使用标准模板库中的队列类型。以下算法 7-2 中采用了 3.2.2 小节中介绍的链队列。

### 算法 7-2 二叉树的层次遍历算法

```

template < class T >
void BiTree < T > ::LevelOrder()
{
 if(root == NULL) // ①若二叉树为空,则遍历结束
 return;
 LinkQueue < BiNode < T >*> Q;
 Q. EnQueue(root); // ②将根指针加入指针队列
 while(!Q. Empty()) // ③若指针队列不空,则循环
 {
 BiNode < T >* p = Q. DeQueue(); // ④出队列,得到当前指针 p
 cout << p -> data; // ④访问当前结点
 // ⑤若当前结点有左、右孩子,则左、右孩子地址进指针队列
 if(p -> lchild != NULL)
 Q. EnQueue(p -> lchild);
 if(p -> rchild != NULL)
 Q. EnQueue(p -> rchild);
 }
}

```

函数 LevelOrder() 层次遍历当前二叉树对象。由于作为成员函数可以直接存取类中的数据成员，因此不需要传递参数。注释中的数字标号对应着 7.4.1 小节中层次遍历算法的步骤编号。

假设二叉树中有  $n$  个结点，在层次遍历算法中，由于每个结点的指针都需要进、出一次队列，因此其时间复杂度是  $O(n)$ 。层次遍历算法的空间复杂度取决于指针队列的使用空间，而同时进入队列的指针最多涉及二叉树中两层的结点，因此其空间复杂度与二叉树的具体形态有关。若层次遍历一棵满二叉树，则指针队列最多需要  $n/2$  个空间存储底层结点的指针，算法的空间复杂度是  $O(n)$ 。

### 7.4.3 二叉树的构造和析构算法

#### 1. 由单个遍历序列构造二叉树

在二叉树的遍历中，无论先序序列、中序序列、后序序列，还是层次序列，和二叉树结构都不存在一一对应的关系。因为每个结点的左、右子树都可能缺失，所以无法确定在遍历序列中的后继结点是左孩子还是右孩子，或是有其他关系的结点。例如，图 7-19 所示的 5 种二叉树的先序序列都是 abc。结点 b 和后继结点 c 的关系，在图 7-19 (a) 和图 7-19 (d) 中是右孩子关系，在图 7-19 (b) 和图 7-19 (e) 中是左孩子，在图 7-19 (c) 中是兄弟关系。

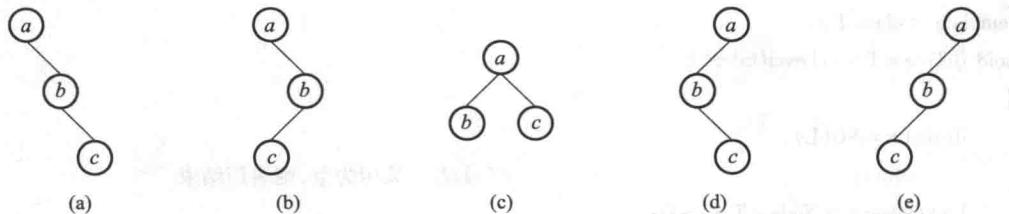


图 7-19 同一个遍历序列对应的多种二叉树

稍微调整一下遍历规则可以建立起遍历序列和二叉树结构的一一对应关系。原遍历算法步骤①规定，若二叉树为空，则遍历结束。调整后的遍历算法步骤①规定，若二叉树为空，输出字符 \* 之后遍历结束。调整后的遍历结果称为带空指针标记的遍历序列。符号 \* 的意义在于标识空指针，它可以是任何一种与结点数据相异的特殊数据。例如，当结点数据都是正数时，可选取负数作为特殊数据来标记空指针。

如图 7-15 所示，其先序序列是 abdecf，带空指针标记的先序序列是 abd \*\*\* e \*\*\* cf \*\*\*。显然，由任意一棵二叉树可以唯一确定一个带空指针标记的先序序列，同样一个合理的带空指针标记的先序序列也能唯一决定一棵二叉树的结构。

正是这种一一对应的关系的存在使得可以将带空指针标记的遍历序列作为构造二叉树对象的参数。以下构造函数的参数 pre 是带空指针标记的先序遍历序列，它是一个 T 型向量。构造完成时，将二叉树的根指针存储在数据成员 root 中。

### 算法 7-3 由带空指针标记的先序序列构造二叉树的算法

```
template < class T >
BinNode < T > * BinTree < T > ::CreateByPre(vector < T > &pre, int &i)
{
 T e = pre[i]; i ++; // 提取当前数据
 if(e == '*')
 return NULL; // 若是特殊数据，返回空指针
 BinNode < T > * p = new BinNode < T > ; // 创建新结点
 p -> data = e;
 p -> lchild = CreateByPre(pre, i); // 创建左子树
 p -> rchild = CreateByPre(pre, i); // 创建右子树
 return p;
}
template < class T >
BinTree < T > ::BinTree(vector < T > &pre)
{
```

```

int i = 0; // 向量 pre 的下标变量
root = CreateByPre(pre, i);
}

```

函数 CreateByPre( vector < T > &pre, int &i ) 是 7.3.2 小节 class BiTree 类模板中未提及的私有函数，其功能是根据带空指针标记的先序序列创建二叉树。由于每次调用 CreateByPre( ) 函数都需要引用向量 pre 中的下一个数据，因此将参数 pre 和 i 设成引用变量，以达到数据的共享、提高参数传递效率的目的。函数 CreateByPre( ) 套用先序遍历的程序框架，将创建二叉树的过程分解为创建根结点和创建左、右子树等 3 部分。

## 2. 由两个遍历序列构造二叉树

假设二叉树中每个结点的值均不相同，在同时知道先序遍历序列和中序遍历序列时是可以唯一确定一棵二叉树结构的。每种遍历序列都保存了二叉树中的部分关系，如先序遍历序列的首元素一定是根元素；在已知根元素位置的前提下，可以很容易在中序遍历序列中区分出左、右子树的中序遍历序列。

设已知某二叉树的先序遍历序列是 ABHFDECKG，中序遍历序列是 HBDFAEKCG，图 7-20 展现了依据先序遍历序列和中序遍历序列构造二叉树的过程。

如图 7-20 (a) 所示，根据先序序列知道 A 是根结点。结合中序序列知，A 的左子树的先序序列是 BHFD，中序序列是 HBDF；A 的右子树的先序序列是 ECKG，中序序列是 EKCG。

如图 7-20 (b) 所示，根据 A 的左子树先序序列知道 B 是 A 的左子树的根结点。结合中序序列知，B 的左子树的先序序列和中序序列都是 H；B 的右子树的先序序列是 FD，中序序列是 DF。

如图 7-20 (c) 所示，根据 A 的右子树先序序列知道 E 是 A 的右子树的根结点。结合中序序列知，E 的左子树的先序序列和中序序列都是空；E 的右子树的先序序列是 CKG，中序序列是 KCG。

如图 7-20 (d) 所示，根据 B 的右子树先序序列知道 F 是 B 的右子树的根结点。结合中序序列知，F 的左孩子是 D，F 的右孩子为空。

如图 7-20 (e) 所示，根据 E 的右子树先序序列知道 C 是 E 的右子树的根结点。结合中序序列知，C 的左孩子是 K，C 的右孩子是 G。

显然，可以将先序序列和中序序列作为构造二叉树类对象的参数。以下算法 7-4 中构造函数的参数 pre 是先序序列，mid 是中序序列，它们都是 T 型向量。构造完成时，将二叉树的根指针存储在数据成员 root 之中。

### 算法 7-4 由先序序列和中序序列构造二叉树的算法

```

template < class T >
BiNode < T > * BiTree < T > ::CreateByPreMid(vector < T > &pre, vector < T > &mid,

```

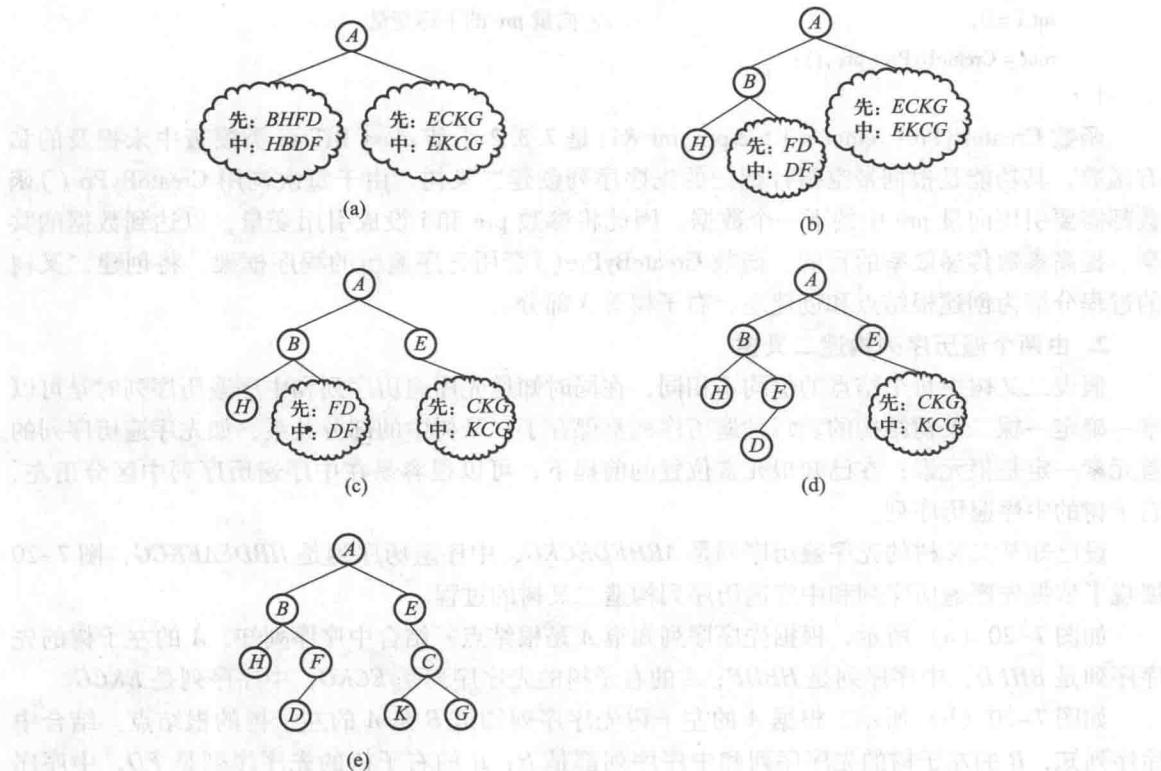


图 7-20 根据先序序列和中序序列构造二叉树的过程

```

int ipre,int imid,int n)
{
 if(n == 0)
 return NULL;
 BinNode < T > * p = new BinNode < T > ; // 创建新结点
 p -> data = pre[ipre];
 for(i = 0 ; i < n ; i ++)
 if(pre[ipre] == mid[imid + i])
 break;
 p -> lchild = CreateByPreMid(pre, mid, ipre + 1, imid, i); // 创建左子树
 p -> rchild = CreateByPreMid(pre, mid, ipre + i + 1, imid + i + 1, n - i - 1); // 创建右子树
 return p;
}

```

```
template < class T >
BiTree < T > :: BiTree(vector < T > &pre, vector < T > &mid)
{
 n = pre.size();
 root = CreateByPreMid(pre, mid, 0, 0, n);
}
```

函数 CreateByPreMid( vector < T > &pre, vector < T > &mid, int ipre, int imid, int n) 是 7.3.2 小节 class BiTree 类模板中未提及的私有函数。该函数参数 pre 和 mid 中分别存储了二叉树的先序序列和中序序列，参数 ipre 表明先序序列在 pre 向量中的起始位置，参数 imid 表明中序序列在 mid 向量中的起始位置，参数 n 表明先序序列和中序序列中元素的个数。

函数 CreateByPreMid 套用先序遍历的程序框架，将创建二叉树的过程分解为创建根结点和左右子树三部分。在创建根结点之后，在中序序列中查找根元素的位置，以此为依据确定左右子树遍历序列的范围，进行递归调用创建左右子树。

实际上，若已知后序序列和中序序列，同样也可以构造唯一的二叉树结构，方法与算法 7-4 类似。这个问题留给读者思考练习。

### 3. 拷贝构造函数

拷贝构造函数是二叉树类的基本方法之一。算法 7-5 中，拷贝构造函数利用被复制对象中的 root 作为参数，复制整个二叉树，并将新创建的二叉树的根指针赋值给当前对象的 root。

#### 算法 7-5 二叉树的拷贝构造算法

```
template < class T >
BiNode < T > * BiTree < T > :: Copy(BiNode < T > * p)
{
 if(p == NULL)
 return NULL;
 BinNode < T > * newp = new BinNode < T >; // 新建结点
 newp -> data = p -> data;
 newp -> lchild = Copy(p -> lchild); // 复制左子树
 newp -> rchild = Copy(p -> rchild); // 复制右子树
 return newp;
}

template < class T >
BiTree < T > :: BiTree(BiTree < T > &tree)
{
 root = Copy(tree.root);
}
```

函数 Copy( BiNode < T > \* p ) 是 7.3.2 小节 class BiTree 类模板中未提及的私有函数。该函数复制一棵以 p 为根指针的二叉树，返回新二叉树的根指针。函数 Copy() 借鉴了先序遍历的程序框架，将复制二叉树的目标分解成复制根结点、复制左子树和复制右子树这三部分。

#### 4. 析构函数

二叉链表对象的建立是一个动态申请空间的过程。因此，当对象析构时需要释放二叉链表中的所有结点。每个结点必须被释放，且只能被释放一次。显然，析构函数的过程与遍历非常相似。若将遍历算法中的访问理解成释放空间，则遍历程序可以很自然地变形为析构程序。利用后序遍历算法的框架实现的析构算法如算法 7-6 所示。

##### 算法 7-6 二叉树的析构算法

```
template < class T >
```

```
void BiTree < T > ::Free(BiNode < T > * p)
```

```
{
```

```
 if(p == NULL)
```

```
 return;
```

```
 Free(p -> lchild); // 释放左子树
```

```
 Free(p -> rchild); // 释放右子树
```

```
 delete p; // 释放根结点
```

```
}
```

```
template < class T >
```

```
BiTree < T > :: ~BiTree()
```

```
{
```

```
 Free(root);
```

```
}
```

函数 Free( BiNode < T > \* p ) 是 7.3.2 小节 class BiTree 类模板中未提及的私有函数。该函数释放以 p 为根指针的二叉树的所有结点空间。它将二叉树析构的过程分解为释放根结点、析构左子树和析构右子树这三部分，套用了后序遍历的程序框架。值得注意的是，函数 Free() 在析构左右子树之后才释放根结点。若套用先序或中序遍历的程序框架，则在根结点释放后才析构左右子树可能导致出现左、右孩子指针值无效的情形。

可以看到，构造函数和析构函数均套用了遍历程序的框架，这体现了该框架的应用意义。该程序框架的本质是将问题的求解过程分解为两个同等性质的小规模问题的求解过程。在程序设计方法中，这是典型的分治法的应用。在许多应用问题的求解中，该框架具有很好的示范性和启发性。

## 7.5 二叉树的其他操作算法

### 1. 计算二叉树的结点数

计算二叉树的结点数是遍历算法的基本应用之一。在算法 7-7 中，函数 Count( BiNode < T > \* p) 是 7.3.2 小节 class BiTree 类模板中未提及的私有函数，它的功能是计算以 p 为根指针的二叉树的结点数。它套用了后序遍历程序的框架，将计数目标分解为对左子树和右子树的计数，通过逐层递归调用完成。

#### 算法 7-7 计算二叉树结点数的算法

```
template < class T >
int BiTree < T > ::Count(BiNode < T > * p)
{
 if(p == NULL)
 return 0;
 left = Count(p ->lchild);
 right = Count(p ->rchild);
 return 1 + left + right;
}

template < class T >
int BiTree < T > ::Count()
{
 return Count(root);
}
```

在函数 Count() 中，以私有成员变量 root 为参数调用重载函数，得到当前对象中二叉树的结点数。

将本程序稍做调整可计算满足某些特定条件的结点数量。例如，计算单、双分支结点和叶子结点的数目，统计等于某关键值的结点数目等。这些留给读者思考。

### 2. 计算二叉树的高度

参照 7.1.2 小节中树的高度的定义，得到二叉树的高度定义为：若二叉树为空，则其高度是 0；否则二叉树的高度是左、右子树高度的最大值加 1。因此，计算二叉树的高度可以被分解为计算左、右子树的高度。

#### 算法 7-8 计算二叉树的高度的算法

```
template < class T >
```

```

int BiTree < T > ::Height(BiNode < T > * p)
{
 if(p == NULL)
 return 0;
 left = Height(p ->lchild);
 right = Height(p ->rchild);
 if(left > right)
 return left + 1;
 else
 return right + 1;
}

template < class T >
int BiTree < T > ::Height()
{
 return Height(root);
}

```

函数 Height( BiNode < T > \* p) 是 7.3.2 小节 class BiTree 类模板中未提及的私有函数，它计算以 p 为根指针的二叉树的高度。函数 Height() 以根指针 root 为参数调用重载函数，得到当前对象中二叉树的高度。不难看出，在计算二叉树高度的过程中，每个结点的高度都被计算了一次。设二叉树中结点数是 n，则算法的时间复杂度是  $O(n)$ 。

在某些应用场合中，若对时间效率要求较高，则可以采用以空间换时间的策略。在结点中增加一个高度域。在第一次求高度的过程中，同时记录各个结点的高度值；再次计算二叉树高度时，时间复杂度就是  $O(1)$  了。在这种策略中，即使发生结点的增加、删除操作，也只需调整局部结点的高度值，其时间代价也是较小的。

### 3. 根据关键值查找结点

查找操作是二叉树的常用操作之一。查找操作依次检查二叉树中每个结点，若发现某结点值与关键值相等，则返回该结点指针。

在算法 7-9 中，函数 Search( BiNode < T > \* p, T e) 是 7.3.2 小节 class BiTree 类模板中未提及的私有函数，其功能是在以 p 为根指针的二叉树中查找与关键值 e 相等的结点。该函数借鉴先序遍历程序的框架，将查找过程分解为根结点比较、左子树查找和右子树查找这三部分。若查找成功，则终止查找过程，返回相应结点的指针；若所有结点都与值 e 不相等，则返回 NULL。函数 Search( p, e) 和先序遍历的程序框架的不同之处是它未必进行了完整的遍历过程。在查找成功的情形下，可能有部分结点没有被比较过。

**算法 7-9** 二叉树的查找算法

```

template < class T >
BiNode < T > * BiTree < T > ::Search(BiNode < T > * p, T e)
{
 if(p == NULL) // 查找失败
 return NULL;
 if(p -> data == e) // 查找成功
 return p;
 BinTreeNode < T > * q = Search(p -> lchild, e);
 if(q != NULL)
 return q; // 若在左子树中查找成功,则返回查找结果
 return Search(p -> rchild, e); // 返回在右子树中的查找结果
}

template < class T >
BiNode < T > * BiTree < T > ::Search(T e)
{
 return Search(root, e);
}

```

函数 Search(T e) 借助根指针 root 作为参数, 在当前对象的整个二叉树内查找值为 e 的结点。

**4. 查找结点的父结点**

在二叉链表结构中, 利用现成的指针关系可以较方便地查找子结点和子孙结点。查找父结点则不然, 需要依赖遍历算法访问每个结点, 检查它们左、右孩子是否符合查找条件。

在算法 7-10 中, 函数 SearchParent( BiNode < T > \* p, BiNode < T > \* child) 是 7.3.2 小节 class BiTree 类模板中未提及的私有函数, 其功能是在以 p 为根指针的二叉树中查找孩子指针为 child 的结点指针。借鉴遍历程序的框架, 将查找过程分解为根结点的比较和左、右子树的查找。若查找成功, 则函数返回相应结点指针; 否则返回 NULL。

**算法 7-10** 二叉树中查找结点的父结点的算法

```

template < class T >
BiNode < T > * BiTree < T > ::SearchParent(BiNode < T > * p, BiNode < T > * child)
{
 if(p == NULL || child == NULL)
 return NULL; // 查找失败
 if(p -> lchild == child || p -> rchild == child)

```

```

 return p; // 查找成功
BinTreeNode < T > * q = SearchParent(p -> lchild, child);
if(q != NULL)
 return q; // 若在左子树中查找成功, 则返回查找结果
return SearchParent(p -> rchild, child); // 返回在右子树中的查找结果
}

template < class T >
BiNode < T > * BiTree < T > ::SearchParent(BiNode < T > * child)
{
 return SearchParent(root, child);
}

```

函数 `SearchParent( BiNode < T > * child )` 借助根指针 `root` 作为参数, 在当前对象的整个二叉链表结构中查找孩子指针为 `child` 的父结点的指针。

本节讲述了二叉树的 4 个常用算法并多次展示了遍历程序框架的应用。建议读者将本节算法进行对比学习, 感受共性, 在编程实践中展开想象的空间, 充分感受遍历算法的魅力。

## 7.6 线索二叉树

### 7.6.1 线索二叉树的概念

前面讨论的二叉树各种遍历算法的本质是将树形结构转换为线性序列, 以便于简化问题。在遍历序列中, 每个结点都有自己的前驱和后继, 求结点的前驱和后继属于基本操作。快速地实现这两个基本操作对二叉树许多算法的性能有重要意义。

求结点的前驱和后继最简单的方法是在遍历过程中寻求答案, 其缺点是时间复杂度等同遍历算法的时间复杂度  $O(n)$ , 这对于基本操作而言显然效率太低。

为了在遍历序列中快速查找结点的前驱和后继, 可以利用二叉链表中空的指针域指向结点在遍历序列中的前驱和后继。这些指向前驱和后继的指针称为线索, 加入了线索的二叉树称为线索二叉树。

常见的线索二叉树结构是结点中非空的指针域保持不变, 结点中空的左孩子指针域 `lchild` 存放该结点的前驱的地址, 结点中空的右孩子指针域 `rchild` 存放该结点后继的地址。由于孩子指针域可能存储两种指针, 因此为相互区别, 在结点结构中增加左标记域 `ltype` 和右标记域 `rtype`。图 7-21 所示的是增加了标记域后的结点结构。

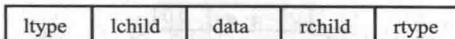


图 7-21 线索二叉树中的结点结构

标记域只能取两种值，取值类型 BiThrNodeType 定义如下：

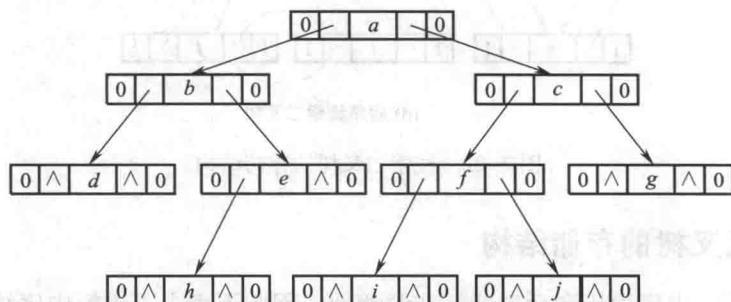
```
enum BiThrNodeType {LINK, THREAD} ;
```

标记域的取值含义为：当 ltype = LINK 时，lchild 存储左孩子指针；当 ltype = THREAD 时，lchild 存储前驱指针。当 rtype = LINK 时，rchild 存储右孩子指针；当 rtype = THREAD 时，rchild 存储后继指针。在绘制结构图时，一般用 0 表示 LINK，用 1 表示 THREAD。

线索二叉树中结点的结构定义如下：

```
template < class T >
struct BiThrNode
{
 BiThrNodeType ltype, rtype;
 T data;
 BiThrNode < T > * lchild, * rchild;
};
```

根据结点中存储的遍历序列的种类，线索二叉树分为先序线索二叉树、中序线索二叉树和后序线索二叉树。图 7-22 (a) 所示的是尚未线索化的二叉树，其中 ltype 和 rtype 域的值均是 LINK，图中以 0 表示；图 7-22 (b) 所示的二叉树是对图 7-22 (a) 所示的二叉树进行先序线索化后得到的先序线索二叉树；图 7-22 (c) 所示的二叉树是对图 7-22 (a) 所示的二叉树进行中序线索化后得到的中序线索二叉树；图 7-22 (d) 所示的二叉树是对图 7-22 (a) 所示的二叉树进行后序线索化后得到的后序线索二叉树。图中带箭头的实线表示的是结点间的父子关系，带箭头的虚线表示的是在遍历序列中结点间的前驱后继关系。



(a) 尚未线索化的二叉树

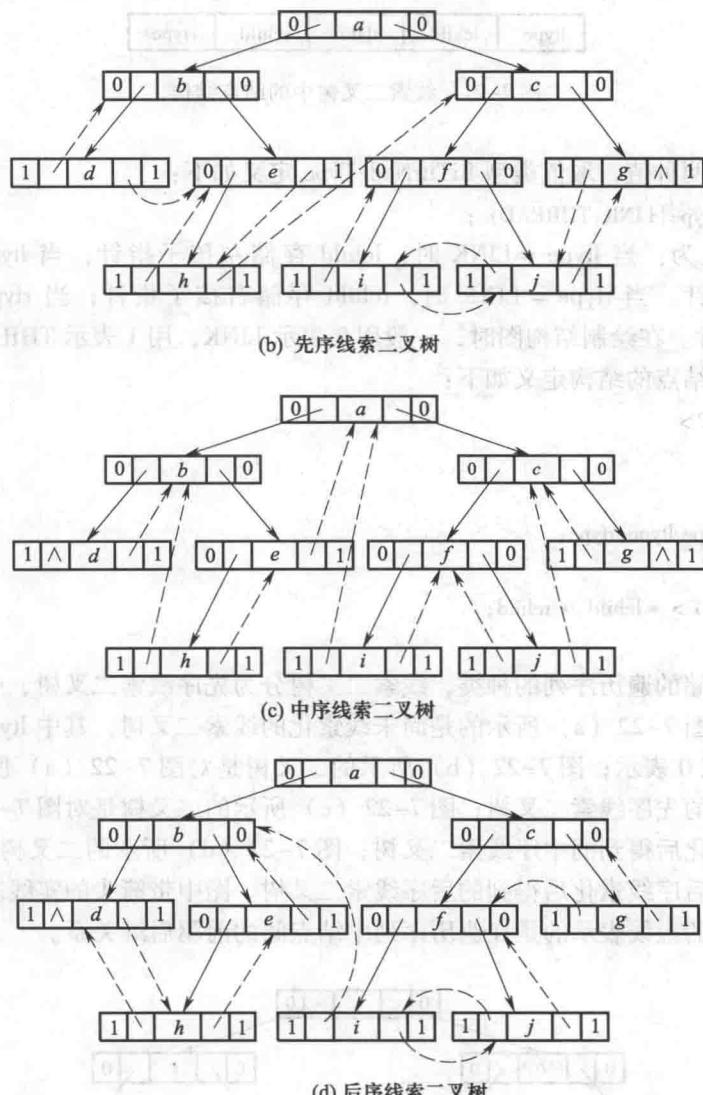


图 7-22 线索二叉树的存储结构

## 7.6.2 线索二叉树的存储结构

由于前序遍历、中序遍历和后序遍历非常相似，因此下面主要讲解中序线索二叉树的类模板 InBinThrTree 和相关算法。与二叉树的类模板相似，在线索二叉树中最核心的数据成员也是

根据指针 root。

```
template < class T >
class InBiThrTree {
public:
 BiThrNode < T > * root;
 ...
private:
 InBiThrTree() { root = NULL; }
 InBiThrTree(vector < T > &pre); // 根据先序序列创建二叉树
 void InThreaded(); // 中序线索化
 ~InBiThrTree();
 BiThrNode < T > * GetNext(BiThrNode < T > * p); // 求中序遍历中的后继结点
 BiThrNode < T > * GetPrev(BiThrNode < T > * p); // 求中序遍历中的前驱结点
 void Traves(); // 利用线索进行中序遍历
 BiThrNode < T > * GetParent(BiThrNode < T > * p); // 求父结点地址
};
```

创建线索二叉树的一般途径是先创建二叉树，再进行线索化。

线索二叉树类与二叉树类拥有许多相同的基本操作，如创建、遍历、查找等。它们的不同之处在于，线索二叉树创建时生成的结点类型 BiThrNode 具有 5 个域。本书没有给出 InBiThrTree (vector < T > &pre) 创建函数，请读者自行参照算法 7-3 完成。此外，在查找结点时线索二叉树类有更多的指针可以利用，因此算法在效率上更胜一筹。

### 7.6.3 线索二叉树的操作算法

#### 1. 线索化算法

在初学者眼里，中序线索二叉树的存储结构图颇有些繁琐，让人望而却步。实际上，创建这样的结构，借助二叉树遍历的程序框架可以极大地降低学习难度。

假设已建立二叉树对象，其中所有结点都是 BiThrNode 类型，且所有结点的 ltype 域和 rtype 域都设置为 LINK。线索化算法的任务是根据遍历次序，逐个检查、线索化每个结点。针对每个结点需要建立前驱线索和后继线索。

对某结点建立前驱线索，若结点的左孩子域不空，则保持左孩子和左标记域保持不变；否则将左孩子域置为前驱结点的指针，左标记域置为 THREAD。

对某结点建立后继线索，若结点的右孩子域不空，则保持右孩子和右标记域保持不变；否则将右孩子域置为后继结点的指针，右标记域置为 THREAD。

显然，建立所有结点的前驱、后继线索应该在遍历过程中完成，每一次的建立线索应该是在一对互为前驱、后继的结点之间进行。

在算法 7-11 中，函数 InThreaded( BiThrNode < T > \* &p ) 实现对 p 所指结点二叉树中序线索化的功能。函数采用二叉树中序遍历的程序框架，将整个二叉树的线索化分解成 3 部分。算法中指针 p 指向当前进行线索化的结点，全局指针变量 prenode 指向刚刚访问过的结点。

- ① 如果二叉链表 p 为空，那么空操作返回。
- ② 对 p 的左子树建立线索。
- ③ 对 p 所指结点建立线索。
  - 若 p 没有左孩子，则为 p 加上前驱线索；
  - 若 p 没有右孩子，则将 p 右标志置为 1；
  - 若结点 prenode 右标志为 1，则为 prenode 加上后继线索；
  - 令 prenode 指向刚刚访问的结点 p。
- ④ 对 p 的右子树建立线索。

### 算法 7-11 二叉树的中序线索化算法

```
template < class T >
void InBiThrTree < T > ::InThreaded(BiThrNode < T > * &p)
{
 if(p == NULL) // ①如果二叉链表 p 为空，则空操作返回
 return;
 InThreaded(p ->lchild); // ②对 p 的左子树建立线索
 // ③对 p 所指结点建立线索
 if(p ->lchild == NULL) // 对 p 的左指针处理
 {
 p ->ltype = THREAD;
 p ->lchild = prenode;
 }
 if(p ->rchild == NULL) // 对 p 的右指针处理
 p ->rtype = THREAD;
 if(prenode != NULL)
 {
 if(prenode ->rtype == THREAD) // 设置 prenode 的后继线索
 prenode ->rchild = p;
 }
 prenode = p;
 InThreaded(p ->rchild); // ④对 p 的右子树建立线索
}
```

```

} // InBiThrTree < T > : : InThreaded()

template < class T >
void InBiThrTree < T > : : InThreaded()
{
 InThreaded(root);
}

```

需要注意的是，算法 7-11 中的 prenode 变量是 BiThrNode 类型的指针，可定义成全局变量。

## 2. 求后继结点和前驱结点的算法

当二叉树被线索化之后，求任意结点的前驱指针和后继指针的问题就很自然地被提了出来。显然，应该充分利用线索树中的线索。但是，仅依赖线索是不够的，孩子结点指针也应被综合考虑。下面以中序线索树为例进行详细说明。

考虑求任意结点的后继指针的算法。在图 7-22 (c) 中，因为结点 e 的 rtype 是 THREAD，所以结点 e 的 rchild 域存储的就是后继指针。但是，这种快捷的方法不能用于那些 rtype 域是 LINK 的结点，如结点 a、结点 b 等。

对于 rtype 域是 LINK 的结点，需要充分利用二叉树遍历的特点。若某结点存在右子树，则该结点在中序序列中的后继就是右子树中最左下方的结点。例如，结点 b 的右子树中最左下方的结点是结点 h，就是结点 b 的后继，结点 a 的右子树最左下方的结点是结点 i，就是结点 a 的后继。因为最左下方的结点没有左孩子，所以其标记是 ltype 域的值等于 THREAD。

以下算法 7-12 中的函数 GetNext( BiThrNode < T > \* p ) 返回 p 所指结点的后继指针。

### 算法 7-12 在中序线索二叉树中求结点的后继指针的算法

```

template < class T >
BiThrNode < T > * InBiThrTree < T > : : GetNext(BiThrNode < T > * p)
{
 if(p -> rtype == THREAD)
 return p -> rchild;
 p = p -> rchild;
 while(p -> ltype == LINK)
 p = p -> lchild;
 return p;
}

```

求任意结点的前驱指针的算法与求后继指针的算法相似。若结点的 ltype 是 THREAD，则其 lchild 域存储的就是前驱指针。例如，图 7-22 (c) 中的结点 d、g、h、i、j 就是这种情形。

若结点的 ltype 是 LINK，则需要在该结点的左子树中寻找。其前驱结点是左子树中最右下

方结点。例如，图 7-22 (c) 中的结点  $a$ 、 $b$ 、 $c$ 、 $e$ 、 $f$  是这种情形。因此最右下方的结点没有右孩子，所以其标记是 rtype 域的值等于 THREAD。

以下算法 7-13 的函数 GetPrev(p) 返回 p 所指结点的前驱指针。

### 算法 7-13 在中序线索二叉树中求结点的前驱指针的算法

```
template < class T >
BiThrNode < T > * InBiThrTree < T > ::GetPrev(BiThrNode < T > * p)
{
 if(p -> ltype == THREAD)
 return p -> lchild;
 p = p -> lchild;
 while(p -> rtype == LINK)
 p = p -> rchild;
 return p;
}
```

与利用遍历算法查找结点的前驱、后继相比，利用线索查找前驱、后继指针的效率有了很大的提高，最好时间复杂度是  $O(1)$ ，最差时间复杂度是  $O(k)$ ， $k$  是二叉树的深度。

### 3. 遍历算法

基于二叉链表结构的二叉树先序遍历算法、中序遍历算法和后序遍历算法，由于需要使用栈空间，因此大量的进栈、出栈操作会造成时间、空间效率的遗憾。在二叉线索树中，可以利用上述的求后继指针的算法，不使用栈空间而实现更快速的遍历算法。对中序线索树进行中序遍历的算法如下：

- ① 找到中序遍历的起点，即二叉树最左下方的结点，将其作为当前结点；
- ② 访问当前结点；
- ③ 找到当前结点的后继结点，将其置为当前结点；
- ④ 若当前结点指针不为空，转②；否则，遍历结束。

以下算法 7-14 中实现对中序线索二叉树的中序遍历。首先从根指针 root 找到中序遍历的起始结点指针 p，然后反复调用函数 GetNext(p)，依次找到中序序列中每个结点的指针。显然，算法的空间复杂度是  $O(1)$ ，时间复杂度是  $O(n)$ 。

### 算法 7-14 中序遍历中序线索二叉树

```
template < class T >
void InBiThrTree < T > ::Traveser()
{
 BiThrNode < T > * p = root;
 while(p -> ltype == LINK) // 找到中序遍历的起点
```

```

p = p ->lchild;
while(p != NULL)
{
 cout << p -> data << " ";
 p = GetNext(p);
}
}

```

在掌握本算法后，可以思考在先序线索二叉树中的先序遍历算法和在后序线索二叉树中的后序遍历算法，以加深对遍历、线索的理解，提高链式结构的操作能力。

#### 4. 求父结点的算法

在二叉树的二叉链表结构中，查找父结点指针需要借助于遍历程序的框架，时间复杂度较大；在线索二叉树结构中可以借助线索改善查找效率。下面以中序线索树为例，详细说明。

设任意结点的指针是  $p$ ，其父结点的指针是  $\text{parent}$ ， $*p$  和  $*\text{parent}$  存在如下两种关系：

(1) 若  $*p$  是  $*\text{parent}$  的左孩子，则  $*p$  的最右下方结点的后继结点一定是  $*p$  的父结点，如图 7-23 (a) 所示。

(2) 若  $*p$  是  $*\text{parent}$  的右孩子，则  $*p$  的最左下方结点的前驱结点一定是  $*p$  的父结点，如图 7-23 (b) 所示。

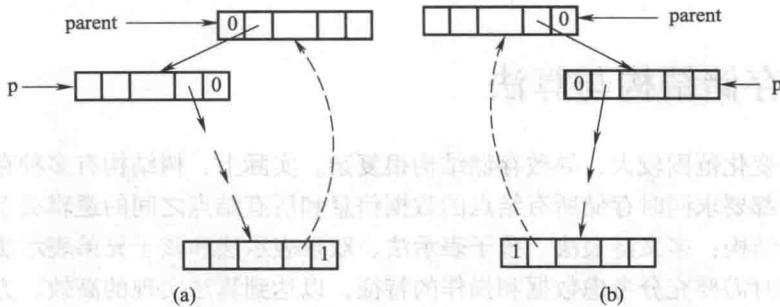


图 7-23 中序线索二叉树的父子结点间的关系

以下算法 7-15 中的函数  $\text{GetParent}(\text{BiThrNode} < \text{T} > *p)$  为求  $p$  所指结点的父结点进行了一次猜测：沿  $p$  所指结点的右孩子链，查找没有右孩子的结点；该结点的后继指针可能是所要求的双亲指针，也可能不是（即后继指针或为  $\text{NULL}$ ，或不为所要求的双亲指针）。若是后者情形，则沿  $p$  所指结点的左孩子链查找没有左孩子的结点，该结点的后继指针一定是所要求的父指针。

#### 算法 7-15 在中序线索化树中，查找结点的父结点

```
template < class T >
```

```

BiThrNode < T > * InBiThrTree < T > ::GetParent(BiThrNode < T > * p)
{
 // 尝试图 7-23(a) 所示的情形
 if(p == NULL)
 return NULL;
 BiThrNode < T > * parent;
 parent = p;
 while(parent -> rtype == LINK)
 parent = parent -> rchild;
 parent = parent -> rchild; // parent 是 * p 的最右下方结点的后继指针
 if(parent && parent -> lchild == p)
 return parent; // 猜测 * p 是否是 * parent 的左孩子
 // 尝试图 7-23(b) 所示的情形
 parent = p;
 while(parent -> ltype == LINK)
 parent = parent -> lchild;
 parent = parent -> lchild; // parent 是 * p 的最左下方结点的前驱指针
 return parent; // parent 一定是 * p 的父指针
}

```

## 7.7 树的存储结构与算法

树结点度的变化范围较大，导致存储结构很复杂。实际上，树结构有多种存储结构。无论哪种存储结构，都要求同时存储所有结点的数据信息和所有结点之间的逻辑关系。本节介绍树的4种常用存储结构：多叉链表法、孩子表示法、双亲表示法和孩子兄弟表示法。它们各有优缺点，具体选用时需要充分考虑数据和操作的特征，以达到算法实现的高效、方便的目的。由于孩子兄弟表示法能将树转换为二叉链表结构，因此本节以它为基础讨论树的常用算法。

### 7.7.1 树的存储结构

#### 1. 多叉链表表示法

二叉树的二叉链表结构采用两个指针域存储结点的孩子指针。树的多叉链表表示法延伸了这种结构设计：若树的度为  $K$ ，则在结点结构中设置  $K$  个孩子指针域，使所有结点同构。结点结构定义如下：

```

template < class T >
struct ChildrenNode

```

```

 {
 T data;
 ChildrenNode < T > * children[K];
 }
}

```

基于 ChildrenNode 结构, 定义树的类模板 ChildrenTree。

```

template < class T >
class ChildrenTree
{
private:
 ChildrenNode < T > * root; // 树的根指针
public:
 ChildrenTree();
 ~ChildrenTree();
 // 取 p 所指结点的第 i 个孩子的指针
 ChildrenNode < T > * GetChild(ChildrenNode < T > * p, int i);
 // 取 p 所指结点的父结点的指针
 ChildrenNode * GetParent(ChildrenNode * p);
 ...
};

```

图 7-24 (b) 是图 7-24 (a) 所示的多叉树的存储结构图。

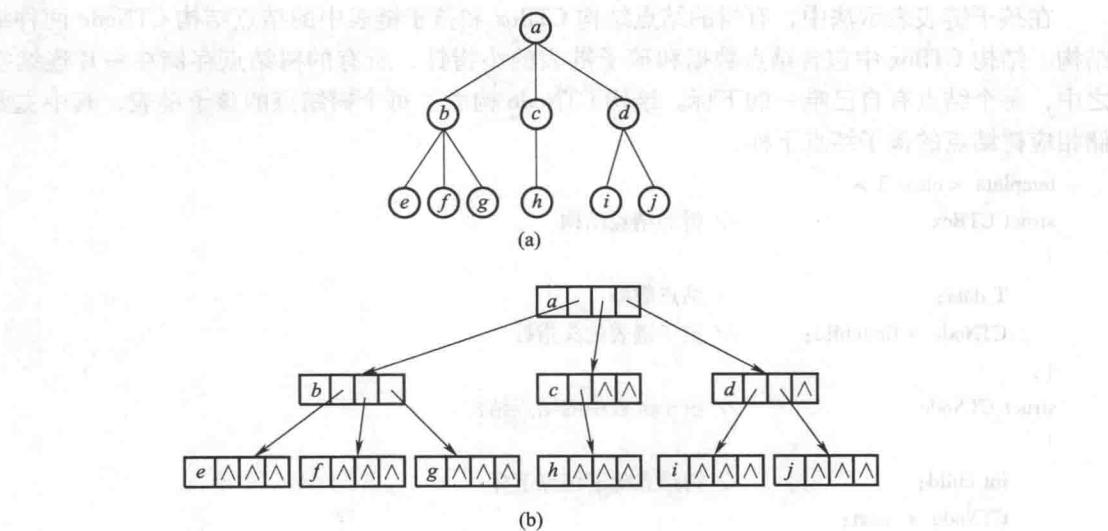


图 7-24 树的多叉链表表示法

采用多叉链表表示法存储树，许多算法设计可以直接参照二叉树的二叉链表结构的算法。其优点是简单易学，缺点是存在许多指针域的浪费。设树中结点数是  $n$ ，树的度是  $k$ ，则共使用了  $n \times k$  个指针域，而这其中只有  $n - 1$  个非空指针域。

## 2. 孩子链表表示法

在树的一些应用中，存在结点之间的关系变化较大，而结点集合变化较小的情形。针对这种操作特征，可以采用孩子链表表示法：结点集合采用顺序存储结构，关系集合采用链式存储结构。其具体结构如图 7-25 所示。

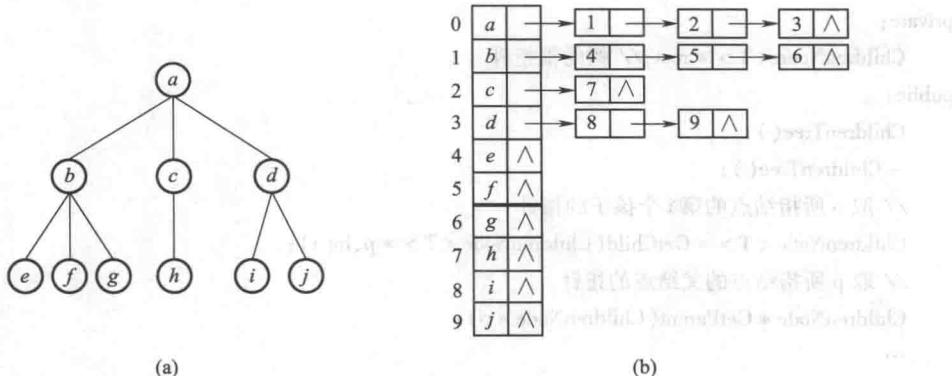


图 7-25 树的孩子链表表示法

在孩子链表表示法中，有树的结点结构 CTBox 和孩子链表中的结点结构 CTNode 两种结点结构。结构 CTBox 中包含结点数据和孩子链表的头指针，所有的树结点存储在一片连续空间之中，每个结点都有自己唯一的下标。结构 CTNode 构成了每个树结点的孩子链表，其中主要存储相应树结点的孩子结点下标。

```
template < class T >
struct CTBox // 树的结点结构
{
 T data; // 结点数据
 CTNode * firstchild; // 孩子链表的头指针
};

struct CTNode // 孩子链表中的结点结构
{
 int child; // 树结点的子结点下标
 CTNode * next;
};
```

以 CTBox 和 CTNode 为基础，定义树的类模板 CTree。其中，tree 是存储所有树结点的连续空间首地址，size 是该空间的大小，n 是该空间中存储的结点数。

```
template < class T >
class CTree
{
 CTBox < T > * tree;
 int size;
 int n;
public:
 CTree();
 ~CTree();
 // 取 p 所指结点的第 i 个孩子的指针
 ChildrenNode < T > * GetChild(ChildrenNode < T > * p, int i);
 // 取 p 所指结点的父结点的指针
 ChildrenNode * GetParent(ChildrenNode * p);
 ...
};
```

当树的度较大时，CTree 类可以减少多叉链表表示法的空间浪费。但是，当插入、删除结点时，却会涉及多个孩子链表的调整，还有可能造成存储空间的再分配，因此时间复杂度较大。在 CTree 类中，利用孩子链表可以方便、快捷地查找指定结点的孩子结点。但是，查找双亲结点则需遍历所有的孩子链表，因此效率就低得多了。

### 3. 双亲表示法

由树的定义可知，树中每个结点（除根结点外）都有且仅有一个双亲结点。根据此特征可对树中每个结点附加一指示双亲的指针，并将所有结点以顺序结构组织在一起。由于采用顺序存储结构，因此双亲指针域可定义为整型，存储双亲结点的下标。树结点的存储结构定义如下：

```
template < class T >
struct PTNode
{
 T data;
 int parent; // 双亲指针域
};
```

基于 PTNode 结构，定义树的类模板 PTree。其中，Tree 是存储所有结点的连续空间，size 是该空间的大小，n 是该空间中存储的结点数。

```
template < class T >
class PTree
```

```

 PTNode * tree;
 int size;
 int n;
public:
 PTree();
 ~PTree();
 int GetChild(int p, int i); // 取第 p 个结点的第 i 个孩子下标
 int GetParent(int p); // 取第 p 个结点的父结点下标
 ...
};

图 7-26 (b) 是图 7-26 (a) 所示的多叉树的存储结构图。在大多数应用中，连续空间的下标从 0 开始计数，因此每个结点（除根结点外）parent 域的取值都大于等于 0。根结点的 parent 的取值有两种方式：一种是赋值为 -1，以表示它的特殊性；另一种是赋值为负数，其绝对值代表整个空间中的结点数字，这有一举两得之用。

```

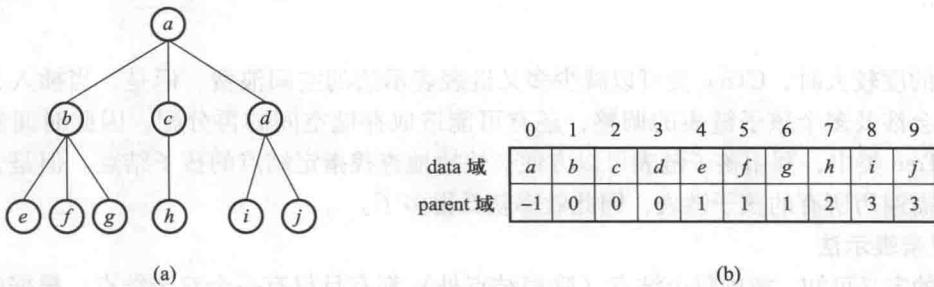


图 7-26 树的双亲表示法

在 PTree 类中，不仅利用结点的双亲指针域很容易找到其双亲结点，而且查找其所有祖先结点也非常便利、高效。若需要查找指定结点的孩子或子孙结点，则需遍历整个树的存储空间，效率就低得多了。

#### 4. 孩子兄弟表示法

树中的关系主要分成父子关系和兄弟关系两类。在树的孩子兄弟表示法中，每个结点存储只存储这两类关系中的最亲密的代表，即第一个孩子结点的指针 firstchild 和下一个兄弟结点指针 nextsibling。结点的结构定义如下：

```

template < class T >
struct CSNode // 树结点类定义
{
 T data;

```

```
CSNode < T > * firstchild, * nextsibling;
```

}；

以树结点结构 CSNode 为基础, 定义树的类模板 CSTree。

```
template < class T >
```

```
class CSTree
```

```
{
```

```
 CSNode < T > * root; // 树的根指针
```

```
public:
```

```
 CSTree();
```

```
 ~CSTree();
```

```
 ...
```

```
};
```

图 7-27 (b) 是图 7-27 (a) 所示的多叉树的存储结构图。由于每个结点只有两个指针域, 因此倾向于将其理解成左子树和右子树。一般将其存储结构绘制成如图 7-27 (c) 所示的形式, 左子树也称为孩子链表, 右子树也称为兄弟链表。

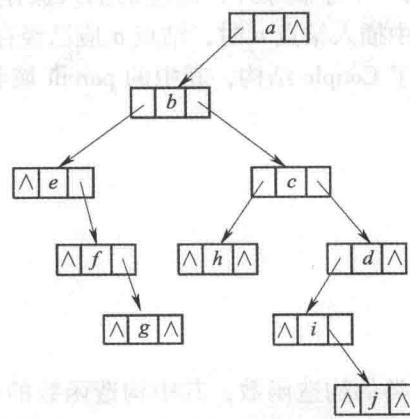
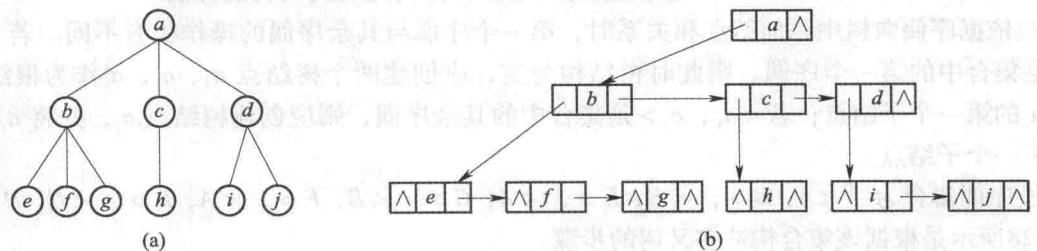


图 7-27 树的孩子兄弟表示法

对于任意一棵多叉树，采用了孩子兄弟表示法存储之后，其内存结构是一个二叉链表形式（特殊之处是根结点无右子树）。对内存中的这种二叉链表结构，按指针域 `firstchild` 和 `nextsibling` 的意义还原必然得到唯一的树结构。

因为孩子兄弟表示法建立起了树和二叉树之间的对应关系，所以常常将其称为树的二叉树表示法。相比树的其他存储结构，孩子兄弟表示法既简化了结构，又可以将许多二叉树的优秀算法移植到树结构的应用中来，因此具有很好的学习、应用价值。

### 7.7.2 树的操作算法

孩子兄弟表示法将树结构存储为二叉链表结构，这使许多二叉树算法稍加调整就可成为解决树应用问题的算法。本节以孩子兄弟表示法为基础讨论树的常用算法。本节讨论的逻辑结构是树，而面对的存储结构是二叉链表，它们之间的联系和区别读者一定要仔细分清。

#### 1. 构造算法

在树的形式定义中，采用序偶集合表示结点之间的父子关系。例如，序偶  $\langle a_i, a_j \rangle$  表示结点  $a_i$  是  $a_j$  的双亲结点。下面以序偶集合作为输入参数讨论树类的构造算法。

在依据序偶向树中添加结点和关系时，第一个序偶与其余序偶的操作稍有不同。若  $\langle a_i, a_j \rangle$  是集合中的第一个序偶，则此时树结构为空，应创建两个树结点  $a_i$ 、 $a_j$ ， $a_i$  作为根结点， $a_j$  是  $a_i$  的第一个子结点；若  $\langle a_i, a_j \rangle$  是集合中的其余序偶，则应创建树结点  $a_j$ ，并将  $a_j$  作为  $a_i$  的下一个子结点。

设序偶集合为  $\{ \langle A, B \rangle, \langle B, E \rangle, \langle A, C \rangle, \langle B, F \rangle, \langle A, D \rangle, \langle C, H \rangle \}$ ，图 7-28 所示是根据该集合构造多叉树的步骤。

为保证任意序偶  $\langle a_i, a_j \rangle$ （第一个序偶除外）对应的插入操作能够正常执行，应对序偶集合中的次序作如下约定：在向树中插入结点  $a_j$  时，结点  $a_i$  应已经存在于树结构之中。

为便于表示序偶的集合，定义了 `Couple` 结构，其中的 `parent` 域和 `child` 域存储的是具有父子关系的结点的值。

```
template < class T >
struct Couple
{
 T parent;
 T child;
};
```



以下算法 7-16 展示了 `CSTree` 类的构造函数，其中构造函数的参数 `ps` 是 `Couple` 的向量，即序偶的集合。构造函数根据第一个序偶创建了根结点，然后循环调用函数 `InsertNode (p)`，完成每个序偶对应的创建子结点、添加父子关系的任务。

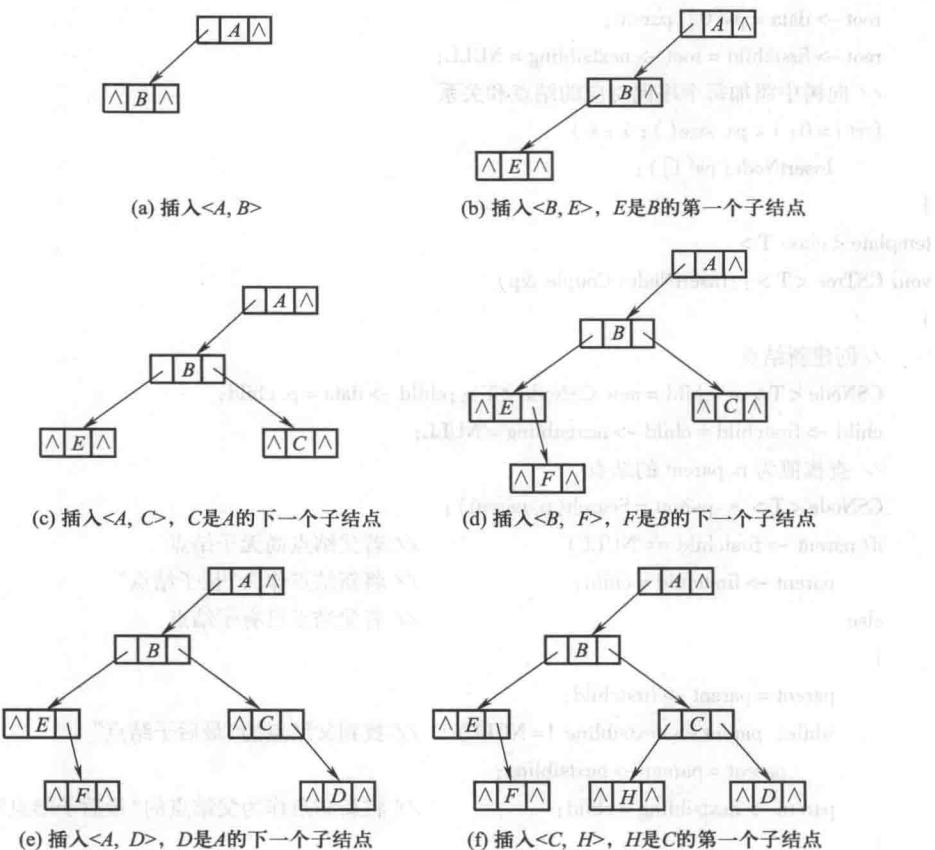


图 7-28 树的构造过程示例

**算法 7-16 树的构造算法**

```
template < class T >
CSTree < T > ::CSTree (vector < Couple > &ps)
```

```
{
 if (ps.size() == 0)
 {
 root = NULL;
 return;
 }
 // 创建根结点
 root = new CSNode < T > (ps[0]);
 for (int i = 1; i < ps.size(); i++)
 {
 Couple p = ps[i];
 CSNode < T > *n = new CSNode < T > (p);
 n->parent = root;
 root->children.push_back(n);
 }
}
```

```

root -> data = ps[0]. parent;
root -> firstchild = root -> nextsibling = NULL;
// 向树中添加每个序偶对应的结点和关系
for(i = 0; i < ps. size(); i ++)
 InsertNode(ps[i]);
}

template < class T >
void CSTree < T > ::InsertNode(Couple &p)
{
 // 创建新结点
 CSNode < T > * child = new CSNode < T > ; child -> data = p. child;
 child -> firstchild = child -> nextsibling = NULL;
 // 查找值为 p. parent 的结点
 CSNode < T > * parent = Search(p. parent);
 if(parent -> firstchild == NULL) // 若父结点尚无子结点
 parent -> firstchild = child; // 将新结点作为“长子结点”
 else // 若父结点已有子结点
 {
 parent = parent -> firstchild;
 while(parent -> nextsibling != NULL) // 找到父结点的“最后子结点”
 parent = parent -> nextsibling;
 parent -> nextsibling = child; // 将新结点作为父结点的“最后子结点”
 }
}

```

在函数 InsertNode() 中，在创建子结点后、添加父子关系前需要查找到父结点的指针，这恰好可利用 7.5 节中介绍的函数 Search(T e)。若父结点尚无子结点，则可以将新建结点作为其第一个孩子；若父结点已有子结点，则需找到它的最后子结点，将新建结点作为该子结点的下一个兄弟结点，即父结点的最后子结点。

## 2. 计算树的高度

计算树的高度与计算二叉树的高度有很大的区别。树采用了二叉树表示法，存储结构是二叉树，逻辑结构是树，要求计算出二叉树所代表的树的深度。例如，图 7-27 所示的二叉树的深度是 6，对应的树的深度是 3。

在 7.1.2 小节中已经定义了树的高度：空树高度为 0，非空树的高度等于所有子树高度的最大值加 1。显然，计算树的高度必须计算根结点的所有子树的高度，而根结点的所有子结点指针都存储在其第一个孩子结点的兄弟链表之中。以下算法 7-17 中函数 Height(p) 计

算以 p 为根指针的树的高度。

### 算法 7-17 计算树高度的算法

```
template < class T >
int CSTree < T > ::Height(CSNode < T > * p)
{
 if(p == NULL)
 return 0;
 int maxheight = 0;
 for(p = p -> firstchild; p; p = p -> nextsibling)
 {
 int height = Height(p); // 计算各个子树的高度
 if(height > maxheight)
 maxheight = height;
 }
 return maxheight + 1;
}
```

```
template < class T >
int CSTree < T > ::Height()
{
 return Height(root);
}
```

函数 Height() 以成员变量 root 为参数调用重载函数，得到当前对象的树的高度。

### 3. 计算树中所有结点的度

在多叉树中，结点的度是结点的一个重要指标。有时在结点结构中增加了一个整数域 degree 以记录结点的度值。计算、存储所有结点的度也是树的基本算法之一。

在采用了二叉树表示法后，树的结点的度等于其第一个孩子结点的兄弟链表中的结点个数。图 7-29 所示是一棵采用了二叉树表示法的树结构，其中每个结点中包含的数字是该结点的度。

在以下算法 7-18 中，函数 Degree(p) 套用了二叉树遍历的程序框架，计算、存储以 p 为根指针的树中的所有结点的度。

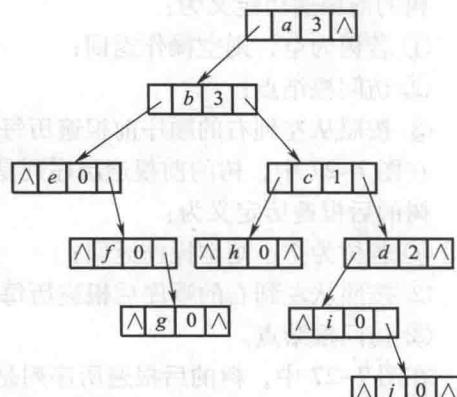


图 7-29 计算树中所有结点的度的示例

### 算法 7-18 计算树中所有结点的度的算法

```

template < class T >
void CSTree < T > ::Degree(CSNode < T > * p)
{
 if(p == NULL)
 return;
 p -> degree = 0; // 初始化 p 所指结点的度
 for(CSNode < T > * child = p -> firstchild; child != NULL; child = child -> nextsibling)
 p -> degree++;
 Degree(p -> firstchild); // 计算左子树中所有结点的度
 Degree(p -> nextsibling); // 计算右子树中所有结点的度
}

template < class T >
void CSTree < T > ::Degree()
{
 Degree(root);
}

```

函数 `Degree()` 以成员变量 `root` 为参数调用重载函数，计算、存储当前对象中的树的所有结点的度。

#### 4. 树的遍历

由树的定义可知，一棵树由根结点和多棵子树构成。因此，树的遍历就是按一定的次序访问根结点和遍历所有子树，通常有前根遍历和后根遍历两种。

树的前根遍历定义为：

- ① 若树为空，则空操作返回；
- ② 访问根结点；
- ③ 按照从左到右的顺序前根遍历每一棵子树。

在图 7-27 中，树的前根遍历序列是 `abefgchdij`。

树的后根遍历定义为：

- ① 若树为空，则空操作返回；
- ② 按照从左到右的顺序后根遍历每一棵子树；
- ③ 访问根结点。

在图 7-27 中，树的后根遍历序列是 `efgbhcidja`。

当多叉树采用孩子兄弟表示法存储为二叉链表结构时，可以借助二叉链表的遍历算法实现多叉树的遍历算法。在多叉树（逻辑结构）与二叉树（存储结构）的对应关系中，根结点的

地位不变，子树集合被转变为了左子树。

在其存储结构（二叉链表）中，树的先根遍历是先访问根结点，再依次先根遍历左子树（即树的所有子树）。显然，在二叉链表的存储结构上，树的先根遍历可以采用先序遍历算法实现。在图 7-27（c）中，树的先根遍历序列和二叉链表结构上的先序序列完全相同，都是 *abefgchdij*。

在其存储结构（二叉链表）中，树的后根遍历是先依次后根遍历左子树（即树的所有子树），再访问根结点。显然，在二叉链表的存储结构上，树的后根遍历可以采用中序遍历算法实现。在图 7-27（c）中，树的后根遍历序列和二叉链表结构上的中序序列完全相同，都是 *efgbhcijda*。

因此，树的先根遍历算法和后根遍历算法可以完全套用二叉树的先序遍历算法和中序遍历算法。

## 7.8 Huffman 树与 Huffman 编码

### 7.8.1 Huffman 树的定义

Huffman 树也称最优树，是一类特殊的二叉树。由它产生的编码称为 Huffman 编码，是一种非常高效的信息压缩技术，在文件存储、信息传递方面有广泛的应用。

#### 1. 信息编码的方法

信息编码是指将信息符号串转换为编码文件，其主要目标之一是提高信息的存储效率。存储效率可以用编码系统的平均码长来衡量。设某编码系统中有  $n$  个符号，每个符号的码长分别是  $L_1, L_2, \dots, L_n$ ，各自出现的频率分别是  $F_1, F_2, \dots, F_n$ ，则

$$\text{平均码长} = \sum_{i=1}^n L_i F_i$$

信息编码的方法可分为定长编码和不定长编码两大类。

定长编码是指在编码系统中，每个符号的代码长度相等。例如，在常用的 ASCII 码中，每个符号的编码都是一个字节。

不定长编码应用于各种符号的使用频率差异较大的场合，其基本思想是利用各种符号出现的统计频率来编码，使经常出现的符号的编码较短，不常出现的符号的编码较长。由于使用不定长编码的目的是使信息经过编码后的编码文件长度尽可能短，因此也称为统计编码。相比定长编码，不定长编码不仅节省磁盘空间，还能提高传递和运算速度。

假设某信息系统中有 4 种符号，分别记作 A、B、C、D，它们各自出现的统计频率是 5%、2%、45%、48%。若采用定长编码方式，则每个符号的编码需要 2 位，如 A 是 00，B 是 01，

C是10, D是11, 平均码长是2。若采用不定长编码方式, 假设A是000, B是001, C是01, D是1, 则平均码长 $=3 \times 5\% + 3 \times 2\% + 2 \times 45\% + 1 \times 48\% = 0.15 + 0.06 + 0.9 + 0.48 = 1.59$ 。显然, 在这种情形下, 不定长编码的效率要优于定长编码。

但是, 由于不定长编码的码长不固定, 因此在识别编码串时, 存在各符号的码串相互混淆的可能。例如, 上述不定长编码中, 若将D的编码改为0, 则编码串“000”的意义既可以是一个符号A, 也可以是3个符号D。因此, 必须为不定长编码增加如下约束条件: 在同一编码系统中, 任何符号的编码不能是另一符号编码的前缀。满足此条件的编码也被称为前缀编码, 有效的统计编码一定是前缀编码。

显然, 如何设计高效率的前缀编码是一个有实际应用意义的问题。

## 2. Huffman 树的定义

Huffman 树的定义涉及以下基本概念。

在二叉树结构中, 设有  $n$  个叶子结点, 每个叶子结点有一个权值, 记作  $w_i$  ( $1 \leq i \leq n$ ), 从根结点到各个叶子结点的路径长度记作  $l_i$ , 则该二叉树的带权路径长度 (Weighted Path Length, WPL)

$$WPL = \sum_{i=1}^n w_i l_i$$

在叶子结点集合确定的前提下, 二叉树可以有许多不同的形态, 每种形态都对应着一棵树的带权路径长度。例如, 已知4个叶子结点的权值分别是2、4、5、7, 图7-30所示的是利用它们构造的3种形态的二叉树。其中, 图7-30(a)所示的二叉树的  $WPL = 46$ , 图7-30(b)所示的二叉树的  $WPL = 36$ , 图7-30(c)所示的二叉树的  $WPL = 35$ 。

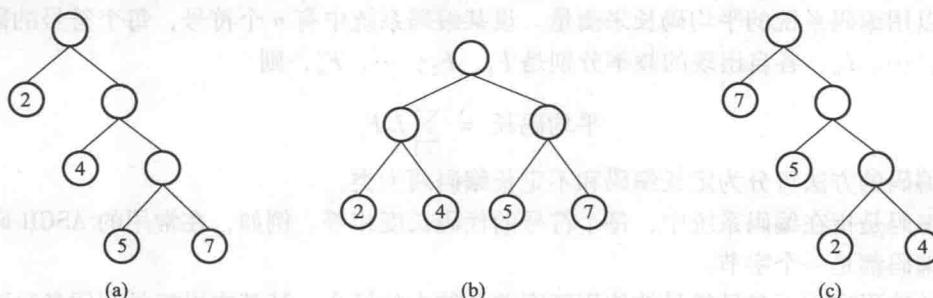


图 7-30 带权路径长度的计算示例

Huffman 树是在叶子结点集合确定的前提下, 所有可能的二叉树形态中, 树的带权路径长度最小的二叉树。

## 3. Huffman 编码

当 Huffman 树应用于信息编码领域时, 叶子结点被用于表示待编码的符号。将每个结点的

左分支记作 0，右分支记作 1，则每个叶子结点的路径都可以用一个 0/1 串的编码来表示，称为 Huffman 编码。因为任何叶子结点的路径不会是另一个叶子结点的路径的前缀，所以，Huffman 编码属于前缀编码。

在这种情形下，叶子结点的路径长度就是相应符号的编码长度，当叶子结点的权值是待编码符号出现的统计频率时，树的带权路径长度就是编码系统的平均码长。因为 Huffman 树的带权路径长度最小，所以 Huffman 编码是编码效率最高的前缀编码。一些压缩/解压缩软件就是基于 Huffman 编码实现的。

## 7.8.2 Huffman 树的构造

### 1. Huffman 树的构造算法

观察图 7-30 所示的 3 棵二叉树的形态和树的带权路径长度，可以得出构造 Huffman 树的大致规则。例如，Huffman 树不应存在度为 1 的结点；权值大的叶子结点离根结点越近，树的带权路径长度越小。但是，这种经验还不足以指导构造 Huffman 树。1952 年，Huffman 提出了如下的 Huffman 树的构造算法。

设已知  $n$  个结点的权值为  $w_1, w_2, \dots, w_n$ 。

① 根据  $(w_1, w_2, \dots, w_n)$  构成森林  $F = (T_1, T_2, \dots, T_n)$ ，每棵二叉树  $T_i$  有且只有一个根结点，权为  $w_i$ ，左右子树为空。

② 在森林  $F$  中选取两棵二叉树根结点的权值最小的树  $T_i, T_j$ ，它们的权值分别为  $w_i, w_j$ 。把这两棵二叉树作为左、右子树合成一棵新树  $T_k$ ， $T_k$  的根结点的权  $w_k = w_i + w_j$ 。

③ 在  $F$  中删除  $T_i, T_j$ ，加入  $T_k$ 。

④ 重复②和③，循环  $n-1$  次，直到  $F$  中只剩一棵树，即为 Huffman 树。

假设某信息系统中有 5 种符号，分别记作 A、B、C、D、E，它们各自出现的统计频率是 6%、14%、53%、15%、12%。Huffman 树的构造过程如图 7-31 所示，图 7-31 (a) 所示的是有 5 棵树的森林，图 7-31 (b)、图 7-31 (c)、图 7-31 (d) 和图 7-31 (e) 所示的是每步合并两棵根结点权值最小的树，图 7-31 (e) 所示的是最终建成的 Huffman 树，(f) 是从 (e) 中读出的 Huffman 编码。

### 2. Huffman 树的存储结构

在设计 Huffman 树的结点结构时，考虑到在寻求 Huffman 编码的过程中，需要多次从叶子结点向根结点上溯，因此采用三叉链表结构：每个结点除了包含左右孩子指针域 lchild 和 rchild，还包含了父指针域 parent。

```
struct HuffmanNode // 哈夫曼树结点的定义
{
 char data; // 待编码的符号
```

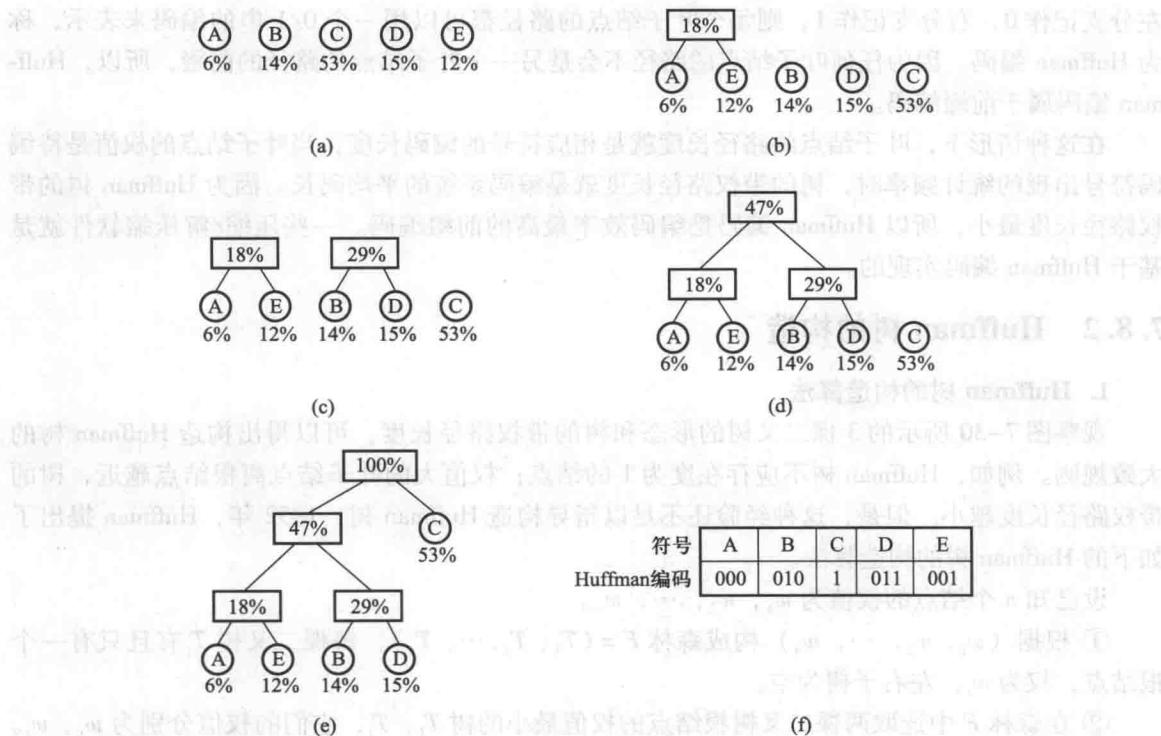


图 7-31 Huffman 树的构造过程

```

double weight; // 符号出现的频率
int parent, lchild, rchild; // 父结点、左孩子结点、右孩子结点的位置
};
```

设叶子结点数为  $n$ , 从 Huffman 算法知, Huffman 树中的分支结点数一定是  $n - 1$ 。由此可见, Huffman 树的存储空间是可以预见的, 因此采用静态空间存储树结构, 称为静态三叉链表结构。在静态链表中, 由于结点的位置由下标表示, 下标的取值范围大于等于 0, 因此使用 -1 来表示空值。Huffman 树的类定义如下:

```

class HuffmanTree
{
private:
 vector < HuffmanNode > hufftree; // 树中所有结点的存储空间
 int n; // 叶子结点数
 ...
public:
```

```
HuffmanTree(vector < HuffmanNode > &leafs);
~HuffmanTree();
vector < int > GetCode(int i); // 取编码
string Decode(vector < int > &source) // 译码
};

构造函数的参数 leafs 是叶子结点的向量，其中存储了所有待编码的字符和相应的权值。构造函数根据叶子结点数初始化向量 hufftree 的空间，用以存储所有的叶子结点和分支结点。
```

以图 7-31 所示的构造过程为例，图 7-32 所示的是图 7-31 (a) 所示的存储结构。在建树之初，向量 hufftree 中存储了 5 个结点，它们的指针域值均为 -1，表示这些结点同时是叶子结点和根结点，它们组成了一个包含 5 棵二叉树的森林。

|          | 0   | 1   | 2   | 3   | 4   | 5 | 6 | 7 | 8 |
|----------|-----|-----|-----|-----|-----|---|---|---|---|
| data 域   | 'A' | 'B' | 'C' | 'D' | 'E' |   |   |   |   |
| weight 域 | 6%  | 14% | 53% | 15% | 12% |   |   |   |   |
| parent 域 | -1  | -1  | -1  | -1  | -1  |   |   |   |   |
| lchild 域 | -1  | -1  | -1  | -1  | -1  |   |   |   |   |
| rchild 域 | -1  | -1  | -1  | -1  | -1  |   |   |   |   |

图 7-32 Huffman 树初始化时的存储结构

图 7-33 所示的是图 7-31 (e) 所示的存储结构。在建树过程中，向量 hufftree 中被陆续添加了 4 个分支结点，最终构成了一棵 Huffman 树，最后的分支结点就是根结点。

|          | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8    |
|----------|-----|-----|-----|-----|-----|-----|-----|-----|------|
| data 域   | 'A' | 'B' | 'C' | 'D' | 'E' |     |     |     |      |
| weight 域 | 6%  | 14% | 53% | 15% | 12% | 18% | 29% | 47% | 100% |
| parent 域 | 5   | 6   | 8   | 6   | 5   | 7   | 7   | 8   | -1   |
| lchild 域 | -1  | -1  | -1  | -1  | -1  | 0   | 1   | 5   | 7    |
| rchild 域 | -1  | -1  | -1  | -1  | -1  | 4   | 3   | 6   | 2    |

图 7-33 Huffman 树建完时的存储结构

### 3. Huffman 树的构造函数

在构造函数的实现中，需要进行  $n - 1$  次的合并树操作，每次需要在森林中选取权值最小的两个根结点。因此，引入函数 SelectSmall( least, less, i )，在 hufftree 的前  $i$  个向量范围内，找到最小的根结点下标 least 和次小的根结点下标 less，请读者自行完成。以下是 Huffman 树的

构造函数，请读者将之与图 7-31、图 7-32 和图 7-33 所示的内容相结合，以便细致地理解相关内容。

### 算法 7-19 Huffman 树的构造函数

```
HuffmanTree::HuffmanTree(vector < HuffmanNode > &leafs)
{
 n = leafs.size(); // 将叶子结点数赋给数据成员 n
 hufftree.resize(2 * n - 1); // 为树中所有结点预留向量空间
 for(i=0; i<n; i++) // 初始化 hufftree, 注意只有一个根结点
 {
 hufftree[i].data = leafs.data;
 hufftree[i].weight = leafs.weight;
 hufftree[i].parent = hufftree[i].lchild = hufftree[i].rchild = -1;
 }
 for(i=n; i<2 * n - 1; i++) // n-1 次合并根结点, i 是新分支结点的下标
 {
 int least, less;
 SelectSmall(least, less, i); // 找到最小、次小的根结点下标
 hufftree[least].parent = hufftree[less].parent = i;
 hufftree[i].parent = -1;
 hufftree[i].lchild = least;
 hufftree[i].rchild = less;
 hufftree[i].weight = hufftree[least].weight + hufftree[less].weight;
 }
}
```

### 7.8.3 Huffman 编码算法

由于每个符号的 Huffman 编码是一个 0/1 串，因此其最高效的存储方法是采用位串的形式。但是，为了减少细节的难度，下面的函数 GetCode(i) 采用整数向量存储第  $i$  个符号的 Huffman 编码，并返回之。第  $i$  个符号存储在 Huffman 树中第  $i$  个叶子结点中。为计算它的编码向量，需要向根结点追溯它的结点路径。在路径中左孩子关系记作编码 0，右孩子关系记作编码 1。由于编码是从叶子结点向根结点逐个读出，次序恰好逆置，因此每个编码数字应该添加在编码向量的首部，这样才能得到编码向量的正确次序。

### 算法 7-20 Huffman 编码算法

```
vector < int > HuffmanTree::GetCode(int i)
```

```
{
```

```

vector<int> code; // 第 i 个符号的编码向量
p = i; // p 是当前结点的下标
parent = hufftree[i].parent; // parent 是当前结点的父结点的下标
while(parent != -1) // 只有根结点的 parent 域为 -1
{
 if(hufftree[parent].lchild == p)
 code.insert(code.begin(), 0); // 在编码向量首部插入 0
 else
 code.insert(code.begin(), 1); // 在编码向量首部插入 1
 p = parent;
 parent = hufftree[parent].parent; // 沿父指针上溯
}
return code;
}

```

在读取所有符号的 Huffman 编码之后，可以对信息源的符号逐个编码，得到编码文件。例如，按图 7-31 所示的 Huffman 编码，设信息源是“BACECDC”，则编码文件是“010000100110111”。此过程属于线性表算法，在此略去。

### 7.8.4 Huffman 译码算法

当已知编码文件，需要还原信息符号串时，需要依据编码时的原 Huffman 树进行译码。下面的函数 Decode( vector<int>&source) 假设当前对象中已建有 Huffman 树结构，参数 source 是编码文件（0/1 向量），返回值是原信息符号串。

每个符号的译码工作都是从 Huffman 树的根向叶子结点的下行过程。逢 0 向左孩子下行，逢 1 向右孩子下行。当下行遇到叶子结点时，该叶子中的 data 域的值就是译码符号。

#### 算法 7-21 Huffman 译码算法

```

string HuffmanTree::Decode(vector<int>&source)
{
 string target = ""; // 译码的目标：原信息符号串
 root = hufftree.size() - 1; // 根结点下标
 p = root; // 将当前结点下标 p 置为根结点下标
 for(i = 0; i < source.size(); i++)
 {
 if(source[i] == 0)
 p = hufftree[p].lchild; // 逢 0 向左孩子下行
 else

```

```

 p = hufftree [p]. rchild; // 逢 1 向右孩子下行
 if(hufftree [p]. lchild == -1 && hufftree [p]. rchild == -1) // hufftree [p] 是叶子结点
 {
 target = target + hufftree [p]. data; // 在目标串末尾添加译码符号
 p = root; // 将当前结点再次置为根结点
 }
}
return target;
}

```

读者可以将 Huffman 树的创建算法、读编码算法和译码算法结合起来，设计一个压缩及解压软件。

### 7.8.5 Huffman 树的其他应用——程序设计流程优化

Huffman 树的应用不仅局限在信息编码中，它对程序设计流程的优化也有指导意义。例如，在计算机语言的入门教材中，大多数读者都阅读过以下程序片段，它将百分制的成绩转换为等级成绩。

```

if(x < 60)
 grade = "不及格";
else
 if(x < 70)
 grade = "及格";
 else
 if(x < 80)
 grade = "中";
 else
 if(x < 90)
 grade = "良";
 else
 grade = "优";

```

这个程序片段的判定过程可以用二叉树表示，如图 7-34 所示。这种表达判断过程的树形结构称为判定树。以上程序片段是正确的，判定树的逻辑也非常清晰。从判定树可以明显看出，不同的结论需要的判断次数不等：“不及格”需要判断 1 次，“及格”需要判断 2 次，“中”需要判断 3 次，“良”和“优”需要判断 4 次。判断次数恰好是每个叶子结点到根的路径长度。若所有成绩段的数据分布是平均的，则该程序段的平均判断次数是  $(1 + 2 + 3 + 4 + 4)/5 = 2.8$ 。这也就是判定树的带权路径长度。

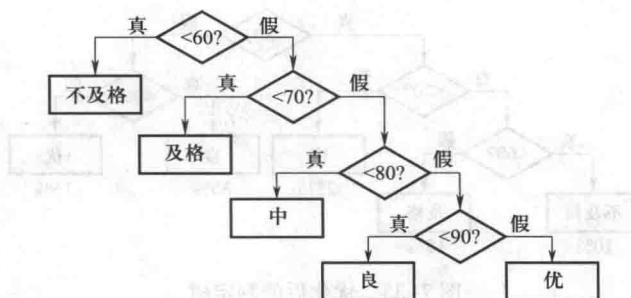


图 7-34 优化前的判定树

若在程序实际执行中，成绩数据呈现如表 7-2 所示的分布特征，则以上程序段的平均判断次数是  $1 \times 0.1 + 2 \times 0.15 + 3 \times 0.25 + 4 \times 0.35 + 4 \times 0.15 = 3.15$ 。

表 7-2 成绩数据的分布特征

| 成绩值域 | [0,59] | [60,69] | [70,79] | [80,89] | [90,100] |
|------|--------|---------|---------|---------|----------|
| 概 率  | 10%    | 15%     | 25%     | 35%     | 15%      |

可以参照 Huffman 树的思想，将概率视作各叶子结点的权值，以树的带权路径长度最小为目标，优化判定树和程序的结构，对上述程序调整如下。相应的判定树如图 7-35 所示。

```

if(x < 80)
 if(x < 70)
 if(x < 60)
 grade = "不及格";
 else
 grade = "及格";
 else
 grade = "中";
else
 if(x < 90)
 grade = "良";
 else
 grade = "优";

```

图 7-35 所示的二叉树的带权路径长度，即程序段的平均判断次数 =  $0.10 \times 3 + 0.15 \times 3 + 0.25 \times 2 + 0.35 \times 2 + 0.15 \times 2 = 2.25$ 。显然，Huffman 树概念的应用提高了程序的效率。

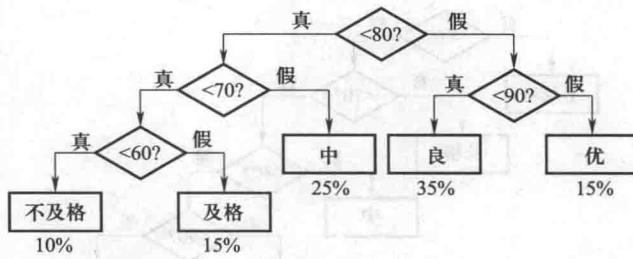


图 7-35 优化后的判定树

## 7.9 等价类问题

### 7.9.1 等价类问题

在离散数学中，等价关系被定义为：如果集合  $S$  中的关系是自反的、对称的和传递的，则称这种关系是一个等价关系。若  $R$  是集合  $S$  中的等价关系， $x$  是  $S$  中的任意元素，所有与  $x$  存在等价关系的元素构成的集合称为由  $x$  生成的一个  $R$  等价类。由等价关系  $R$  可产生集合  $S$  的唯一划分，即可以按  $R$  将  $S$  划分为若干不相交的子集  $S_1, S_2, \dots$ ，它们的并集是  $S$ ，则这些子集  $S_i$  称为  $S$  的  $R$  等价类。

实际上，等价类描述的是一个集合中若干子集及其之间的关系。在现实世界中，等价类有着广泛的应用背景。许多应用问题都可以归纳为按给定的等价关系将某集合划分为若干等价类，通常称此类问题为等价问题。例如，在网络中计算最小生成树的算法就用到了等价类的思想。构造等价类一般利用形如  $(x, y)$  的等价偶对的集合作为已知条件，其中  $x, y \in S$ ，表示  $x$  和  $y$  之间存在等价关系。假设集合  $S$  有  $n$  个元素，构造等价类的算法如下：

- ① 令  $S$  中每个元素各自形成一个只含有单个成员的子集，记作  $S_1, S_2, \dots, S_n$ 。
- ② 考察偶对  $(x, y)$ ，判断  $x$  和  $y$  所属子集，设  $x \in S_i, y \in S_j$ 。若  $i = j$ ，则  $x$  和  $y$  已属同一个等价类，无须操作；若  $i \neq j$ ，则将  $S_i$  并入  $S_j$ 。
- ③ 重复执行步骤②，当所有偶对都处理完时，所有剩余子集即为  $S$  的  $R$  等价类。

设集合  $S = \{a, b, c, d, e\}$ ，等价偶对有  $(a, c), (b, d), (c, e)$ ，则构造等价类的过程如图 7-36 所示。其中图 7-36 (a) 所示的是 5 个元素各自形成一个子集；图 7-36 (b) 所示的是考察了偶对  $(a, c)$  后，子集  $\{a\}$  和子集  $\{c\}$  合并后的情形；图 7-36 (c) 所示的是考察了偶对  $(b, d)$  后，子集  $\{b\}$  和子集  $\{d\}$  合并后的情形；图 7-36 (d) 所示的是考察了偶对  $(c, e)$  后，子集  $\{c, a\}$  和子集  $\{e\}$  合并后的情形。

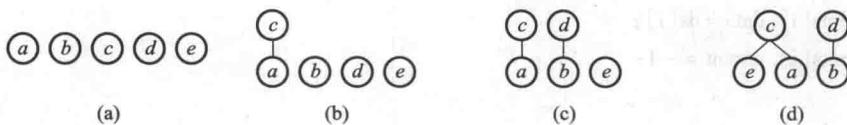


图 7-36 等价类的构造过程

## 7.9.2 等价类的实现

从构造等价类的算法可见，其数据基础是一个数据集合。这个数据集合在等价类的构造过程中展现了如下特点：从图 7-36（a）到图 7-36（d）可以明显地看出一个森林的变化过程，图 7-36（a）所示的是 5 棵树的森林，之后的每一步变化都使得森林中树的数量减一；因为每个子集中元素不能预先确定，所以无法预测森林中树的度数；整个森林的元素个数是预定的。因此，可以确定这个数据集合是一个森林结构，其中每棵树表示一个等价类，其最适合的存储结构应是树的双亲表示法，即每个结点的结构采用 7.7.1 节中定义的 PTNode 类型。

分析构造等价类的算法，可以看到其中包含了 3 个基本操作：将每个数据成员初始化为一个子集（树），判断某个元素所属的子集（树），以及合并两个互不相交的子集（树）。

基于以上分析，定义等价类的类模板 MFSet 如下：

```
template < class T >
class MFSet
{
public:
 vector < PTNode < T >> Sets; // 所有子集的元素的存储空间
 MFSet(vector < T > &ds); // 初始化每个数据成员为一棵树
 ~MFSet();
 int Find(int i); // 判断第 i 元素所属的子集编号
 void Merge(int i, int j); // 合并子集 i 和子集 j
};
```

在构造函数中，将每个结点的双亲指针设置为 -1，以表示根结点，具体实现代码如下：

```
template < class T >
MFSet::MFSet(vector < T > &ds)
{
 Sets. resize(ds. size());
 for(int i = 0; i < ds. size(); i++)
 Sets[i]. parent = -1;
```

```

Sets[i]. data = ds[i];
Sets[i]. parent = -1;
}
}

```

在函数 Find 的实现中，很自然地将根结点的下标作为子集的编号。根结点的标记是其 parent 域的值为 -1。因此，函数 Find 的流程就是沿着双亲指针链查找 parent 域为负数的结点，然后返回该结点的下标。

```

template < class T >
int MFSet < T > ::Find(int i)
{
 while(Sets[i]. parent >= 0)
 i = Sets[i]. parent;
 return i;
}

```

函数 Merge(i, j) 实现根结点下标分别是 i 和 j 的两个子集（树）的合并。在树的双亲表示法中，实现这个操作非常简单，只需要将一棵树根结点的 parent 域值置为另一棵树根结点的下标即可。

```

template < class T >
void MFSet < T > ::Merge(int i, int j)
{
 Sets[i]. parent = j;
}

```

### 7.9.3 性能分析与改进

在构造等价类的过程中，由于需要多次调用函数 Find() 和 Merge()，因此有必要分析它们的时间复杂度。函数 Merge() 的时间复杂度很简单，是  $O(1)$ 。函数 Find() 的时间复杂度取决于树的高度，设树高为  $k$ ，则其时间复杂度是  $O(k)$ 。因此如何降低树的高度成为提高构造等价类算法性能的关键。

设集合  $S = \{a, b, c, d, e\}$ ，等价偶对有  $(a, b), (b, c), (c, d)$ ，则按照类模板 MFSet 的实现方法，构造等价类的过程如图 7-37 所示，图 7-37 (b)、图 7-37 (c) 和图 7-37 (d) 所示的是依次考察等价偶对后的森林结构。

在图 7-35 (d) 中，表示等价类的树已退化成了单链结构，树的高度已近似等于整个集合的元素个数。虽然这种最坏情况不会时时出现，但如何避免最坏情况却是一定要考虑的。

一种常用的改进方法是在合并两个子集时，将元素较少的树的根结点的 parent 域值置为元

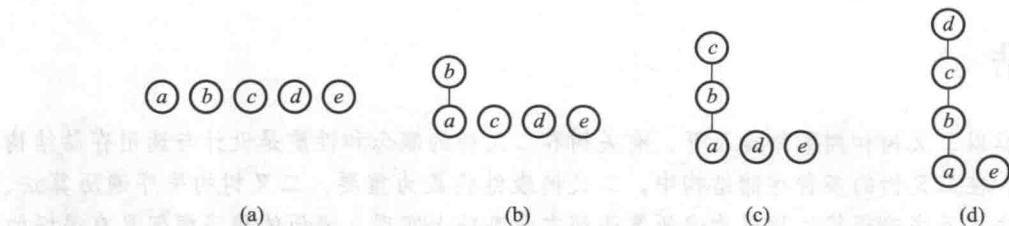


图 7-37 等价类的构造过程

素较多的树的根结点的下标。为能较容易地读取树中的结点个数，将根结点的 parent 域值设为树中结点个数的负数。

根据以上改进，只需修改函数 Merge()，它将比较两棵树的元素个数并调整两个根结点的 parent 域值。

```
template < class T >
void MFSet < T > ::Merge(int i, int j)
{
 if(Sets[i].parent < Sets[j].parent) // 子集 i 的元素个数较多
 {
 Sets[i].parent += Sets[j].parent; // 增加子集 i 的元素个数
 Sets[j].parent = i; // 将子集 j 并入子集 i
 }
 else
 {
 Sets[j].parent += Sets[i].parent; // 增加子集 i 的元素个数
 Sets[i].parent = j; // 将子集 i 并入子集 j
 }
}
```

采用图 7-37 所示的示例数据，按照改进后的 Merge() 方法，构造等价类的过程如图 7-38 所示，图 7-38 (b)、图 7-38 (c) 和图 7-38 (d) 所示的是依次考察等价偶对后的森林结构。

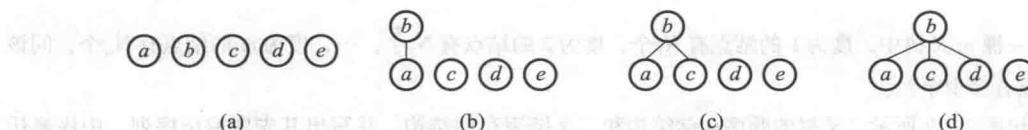


图 7-38 改进后等价类的构造过程

## 本章小结

本章内容以二叉树和树的结构展开，有关树和二叉树的概念和性质是设计与选用存储结构的预备知识。在二叉树的多种存储结构中，二叉链表结构最为重要，二叉树的先序遍历算法、中序遍历算法、后序遍历算法和层次遍历算法都在此基础上实现。遍历的程序框架具有很好的示范性，许多二叉树算法参考了这个程序框架。线索二叉树增加了结构的难度，但线索的利用提高了检索结点的效率。本章还介绍了树的4种存储结构。由于孩子兄弟表示法将树结构存储为二叉树结构，因此以它为重点讲解了树的若干常用算法。最后，本章给出了一个综合性应用实例——Huffman树的应用。

本章的主要学习要点如下：

- (1) 掌握树的相关概念和基本性质，包括树的度、树的高度、结点数等概念及其相关性质。
- (2) 掌握二叉树的相关概念和基本性质，包括满二叉树、完全二叉树的概念及其性质。
- (3) 重点掌握二叉树的存储结构，包括顺序存储结构和链式存储结构。
- (4) 重点掌握二叉树遍历算法和各种常用操作算法，包括构造二叉树算法、统计结点数、计算二叉树高度和查找等算法。
- (5) 掌握线索二叉树的概念、结构，以及线索指针的应用算法，包括中序线索树的线索化和遍历等算法。
- (6) 掌握树的存储结构，包括多叉链表表示法、孩子链表表示法、双亲表示法和孩子兄弟表示法，并掌握基于孩子兄弟链表表示法的常用操作算法。
- (7) 掌握 Huffman 树的概念、构造方法和 Huffman 编码算法。
- (8) 掌握等价类概念以及利用树构造等价类的方法。
- (9) 灵活运用二叉树结构解决一些综合应用问题。

## 习题 7

- 7.1 若一棵  $m$  叉树中，度为 1 的结点有  $N_1$  个，度为 2 的结点有  $N_2$  个， $\dots$ ，度为  $m$  的结点有  $N_m$  个，问该树的叶子结点有多少个？
- 7.2 画出图 7-39 所示二叉树的顺序存储结构和二叉链表存储结构，并写出其先序遍历序列、中序遍历序列和后序遍历序列。
- 7.3 试找出分别满足下列条件的所有二叉树：
  - (1) 先序序列和中序序列相同；

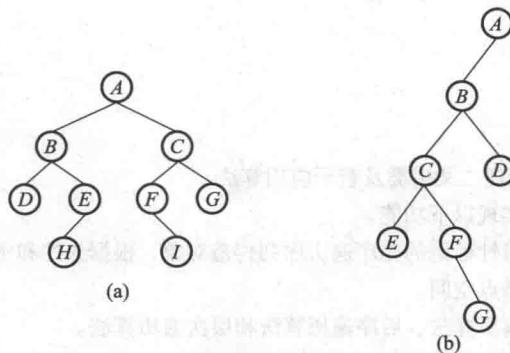


图 7-39 习题 7.2 和 7.9 的图例

(2) 中序序列和后序序列相同；

(3) 先序序列和后序序列相同。

7.4 已知某二叉树的后序序列是 GEF CDBA，中序序列是 AEGCFBD，请画出该二叉树的二叉链表结构图，并写出先序遍历序列。

7.5 设有 168 个结点的完全二叉树，请问叶子结点、单分支结点、双分支结点各有多少个？

7.6 已知二叉树采用二叉链表结构存储， $p$  和  $q$  是二叉树中两个结点的指针，试编写算法，求与它们最近的共同祖先结点的地址。

7.7 已知一个采用顺序存储结构的二叉树对象，试编写算法创建该对象的二叉链表结构。

7.8 试编写算法求二叉树中各结点的平衡因子（左右子树高度之差）。

7.9 画出图 7-39 所示二叉树的先序线索树、中序线索树和后序线索树的存储结构。

7.10 采用孩子兄弟表示法存储图 7-40 所示的树，试绘出其存储结构图。

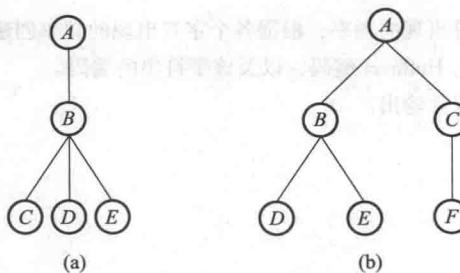


图 7-40 习题 7.10 的图例

7.11 已知在一份电文中只使用了 7 个字符 A、B、C、D、E、F、G，其统计频率分别为 5%、29%、7%、8%、14%、20%、17%，建立一棵 Huffman 树（要求左孩子权小于等于右孩子的权），试写出每个字符所对应的 Huffman 编码。

7.12 已知集合  $S = \{a, b, c, d, e, f, g, h\}$ ，等价偶对有  $(a, c), (b, d), (d, f), (c, f), (g, h)$ ，请绘

出集合  $S$  上的等价类结构。

## 上机实验题 7

**实验题 7.1** 编写程序，实现二叉树类及若干应用算法。

要求采用二叉链表结构，实现以下功能。

- (1) 构造函数：根据带空指针标记的先序遍历序列构造对象，根据先序和中序遍历序列构造对象。
- (2) 析构函数：释放所有结点空间。
- (3) 先序遍历算法、中序遍历算法、后序遍历算法和层次遍历算法。
- (4) 统计叶子结点、单分支结点、双分支结点的个数。
- (5) 计算二叉树的高度。
- (6) 交换每个结点的左右孩子。
- (7) 输出根结点到每个叶子结点的路径。

**实验题 7.2** 编写程序，实现树类及若干应用算法。

要求采用孩子兄弟表示法，实现以下功能。

- (1) 构造函数：根据序偶集合构造对象。
- (2) 析构函数：释放所有结点空间。
- (3) 先根、后根遍历算法。
- (4) 计算每个结点的度。
- (5) 计算树的高度。
- (6) 输出根结点到每个叶子结点的路径。

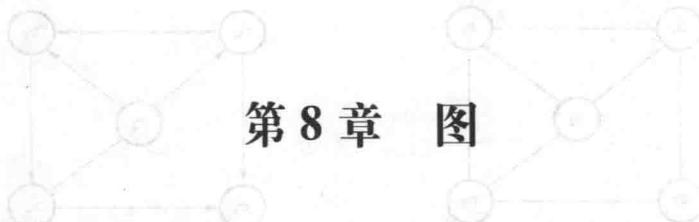
**实验题 7.3** 编写程序，实现 Huffman 编码。

要求实现以下功能：

- (1) 输入字符串，统计各字符出现的频率，根据各个字符出现的频率创建 Huffman 树。
- (2) 输出该字符串中各字符的 Huffman 编码，以及该字符串的编码。
- (3) 输入相关编码串进行译码并输出。



图 7.10 例题 7.3 中的 Huffman 树



## 第8章 图

图是一种非常典型和常用的非线性结构，它相对于树形结构更为复杂。树形结构是以分支关系定义的层次结构，树中的每个结点可以有多个后继结点（即孩子结点），但仅有一个前驱结点（即双亲结点），且结点之间没有回路。而在图中任意两个结点之间均可以有边相连，即任意两个元素之间都有可能相关。实际上，树和线性表都可以看成是一种受限形式的图，而图则可以看成是所有数据结构的一种最泛化表示。数学中的图论分支注重于研究图的纯数学性质，而本章则着重介绍图的存储结构与一些典型的图的操作与应用算法。

### 8.1 图的基本概念

#### 8.1.1 图的定义

一个图可以使用两个集合进行定义。第一个集合是点的集合，这些点在图术语中一般被称为顶点（Vertex）；第二个集合是连接两个顶点的边（Edge）的集合。图的具体定义如下。

图是由顶点集合及顶点间的关系集合组成的一种数据结构：

$$\text{Graph} = (V, E)$$

其中， $V = \{x \mid x \in \text{某个数据对象}\}$ ，它是顶点的有穷非空集合； $E = \{(x, y) \mid x, y \in V\}$  或  $E = \{<x, y> \mid x, y \in V\}$  是顶点之间关系的有穷集合，称为边集。

如果图中代表边的顶点对是无序的，那么称该图为无向图。无向图中代表边的顶点对通常用圆括号括起来，称为无向边，用以表示一种对称的关系。例如， $(v_i, v_j)$  和  $(v_j, v_i)$  所表示的是同一条边。如果希望图中的边表达的是一种不对称的关系，那么必须建立有向图。有向图中表示边的顶点对则是有序的，通常用尖括号括起来，称之为有向边，也称为弧。例如， $<v_i, v_j>$  表示的是从  $v_i$  到  $v_j$  的一条弧，其中顶点  $v_i$  称为弧尾，顶点  $v_j$  弧头。显然， $<v_i, v_j>$  与  $<v_j, v_i>$  是表示两条不同的弧。

如图 8-1 所示， $G_1$  是一个无向图， $G_2$  是一个有向图。 $G_1$  的顶点集  $V = \{v_0, v_1, v_2, v_3, v_4\}$ ，边集  $E = \{(v_0, v_1), (v_0, v_2), (v_0, v_3), (v_1, v_2), (v_1, v_4), (v_2, v_4), (v_3, v_4)\}$ 。 $G_2$  的顶点集  $V = \{v_0, v_1, v_2, v_3, v_4\}$ ，有向边集  $E = \{<v_0, v_1>, <v_0, v_3>, <v_1, v_4>, <v_2, v_0>, <v_1, v_2>, <v_2, v_4>, <v_4, v_3>\}$ 。

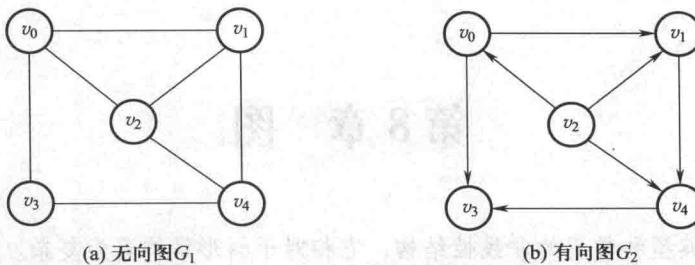


图 8-1 无向图与有向图示例

## 8.1.2 图的基本术语

在阐述图的各种算法与应用之前，需要先给出与图相关的基本概念与术语。

### 1. 邻接点

在一个无向图中，如果  $(u, v)$  是图中的一条边，则称该边的两个端点  $u$  与  $v$  互为邻接点 (Adjacent)。在有向图中，若存在一条有向边  $< u, v >$ ，则称该边是顶点  $u$  的一条出边，同时也是顶点  $v$  的一条入边；也称顶点  $u$  邻接到顶点  $v$ ，或顶点  $v$  邻接自顶点  $u$ 。

### 2. 顶点的度、入度与出度

无向图中，一个顶点  $v$  的度 (Degree) 是指与它相关联的边的条数，即以  $v$  为端点的边的数量，记作  $TD(v)$ 。在有向图中，顶点的度等于该顶点的入度与出度之和。顶点  $v$  的入度是以  $v$  为终点的有向边的条数，记作  $ID(v)$ ；顶点  $v$  的出度是以  $v$  为始点的有向边的条数，记作  $OD(v)$ ；顶点  $v$  的度  $TD(v) = ID(v) + OD(v)$ 。若一个无向图中有  $n$  个顶点和  $e$  条边，则有

$$e = \frac{1}{2} \sum_{i=1}^n TD(v_i)$$

### 3. 权和网

某些图的每条边都具有一个与之相关的具有某种实际意义的数，称为权 (Weight)。这种带权图也称为网 (Network)。

### 4. 完全图

在无向图中，若任意两个顶点之间均存在一条边，则该图称为无向完全图 (Undirected Complete Graph, UCG)。含有  $n$  个顶点的无向完全图有  $n(n - 1)/2$  条边。在有向图中，若任意两个顶点之间都存在方向相反的两条弧，则称该图为有向完全图 (Directed Complete Graph, DCG)。含有  $n$  个顶点的有向完全图有  $n(n - 1)$  条边。例如，图 8-2 (a) 所示的是一个无向完全图，图 8-2 (b) 所示的是一个有向完全图。

### 5. 子图

设有两个图  $G = (V, E)$  和  $G' = (V', E')$ 。若  $V' \subseteq V$  且  $E' \subseteq E$ ，则称图  $G'$  是图  $G$  的子图

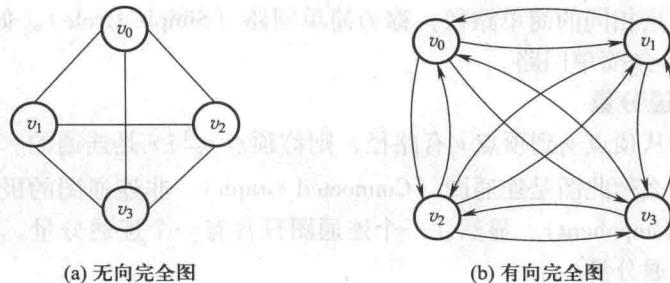


图 8-2 完全图示例

(Subgraph)。

### 6. 路径

在图  $G = (V, E)$  中, 若从顶点  $v_i$  出发, 经过一个系列顶点  $v_{p1}, v_{p2}, \dots, v_{pm}$ , 最后到达顶点  $v_j$ , 则称顶点序列  $(v_i, v_{p1}, v_{p2}, \dots, v_{pm}, v_j)$  为从顶点  $v_i$  到顶点  $v_j$  的路径 (Path)。若该图是无向图, 则边  $(v_i, v_{p1})$ 、 $(v_{p1}, v_{p2})$ 、 $\dots$ 、 $(v_{pm}, v_j)$  应属于  $E$ ; 若该图是有向图, 则有向边  $< v_i, v_{p1} >$ 、 $< v_{p1}, v_{p2} >$ 、 $\dots$ 、 $< v_{pm}, v_j >$  应属于  $E$ 。

### 7. 路径长度

不带权图的路径长度 (Path Length) 是指此路径上经过的边的数量。带权图 (网) 的路径长度是指路径上经过的各边的权之和。

### 8. 简单路径

若路径上经过的各顶点均不互相重复, 则称这样的路径为简单路径 (Simple Path)。如图 8-3 (a) 所示,  $(v_0, v_1, v_3, v_2)$  是一条简单路径, 其长度为 3。如图 8-3 (b) 所示,  $(v_0, v_1, v_3, v_0, v_1, v_2)$  则是一条非简单路径。

### 9. 回路或环

若路径上的第一个顶点与最后一个顶点相同, 则称这样的路径为回路或环 (Cycle)。第一

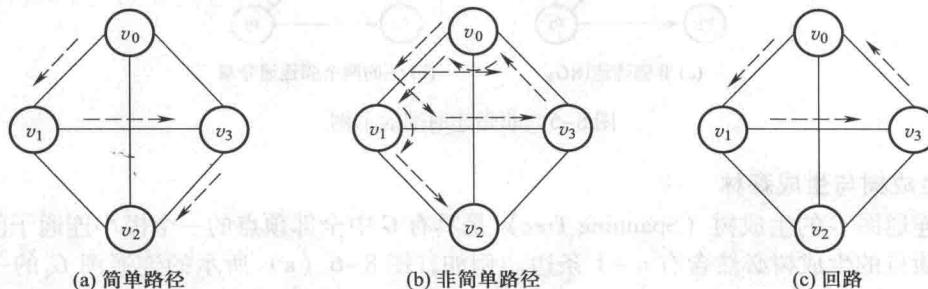


图 8-3 图的路径示例

一个顶点与最后一个顶点相同的简单路径，称为简单回路（Simple Cycle）。如图 8-3 (c) 所示， $(v_0, v_1, v_3, v_0)$  是一条简单回路。

### 10. 连通图与连通分量

在无向图中，若从顶点  $v_i$  到顶点  $v_j$  有路径，则称顶点  $v_i$  与  $v_j$  是连通的。如果图中任意一对顶点都是连通的，那么称此图是连通图（Connected Graph）。非连通图的极大连通子图叫做连通分量（Connected Component）。显然，一个连通图只含有一个连通分量。例如，图 8-4 所示的非连通图有两个连通分量。

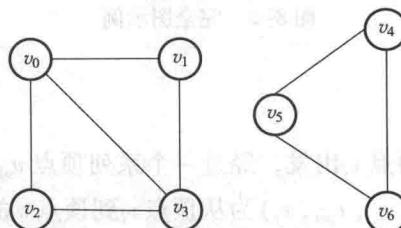


图 8-4 非连通图的示例

### 11. 强连通图与强连通分量

在有向图中，若对于每一对顶点  $v_i$  和  $v_j$ ，都存在一条从  $v_i$  到  $v_j$  和从  $v_j$  到  $v_i$  的路径，则称此图是强连通图（Strongly Connected Graph）。非强连通图的极大强连通子图叫做强连通分量（Strongly Connected Component）。图 8-5 (a) 所示的图  $G_3$  是一个非强连通图，它包含图 8-5 (b) 所示的两个强连通分量。

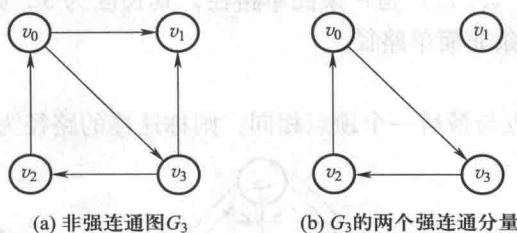


图 8-5 非强连通图的示例

### 12. 生成树与生成森林

一个连通图  $G$  的生成树（Spanning Tree）是具有  $G$  中全部顶点的一个极小连通子图。一棵含有  $n$  个顶点的生成树必然含有  $n - 1$  条边。例如，图 8-6 (a) 所示的连通图  $G_4$  的一棵生成树如图 8-6 (b) 所示。非连通图的每个连通分量分别可以得到一棵生成树，各个连通分量的生成树则组合构成了一个生成森林（Spanning Forest）。

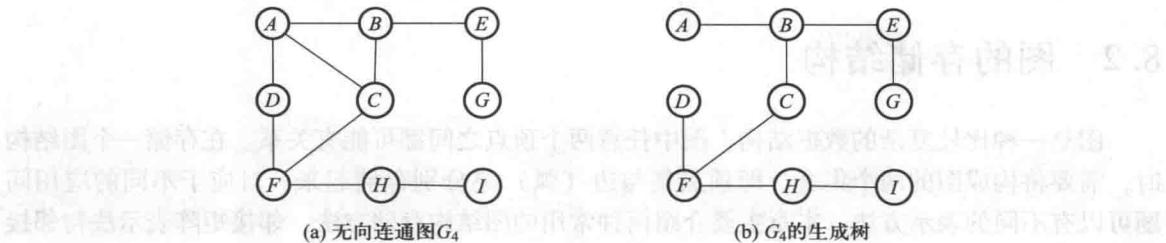


图 8-6 连通图及其生成树示例

在一个有向图中,如果恰有一个顶点的入度为0,其余顶点的入度均为1,则该有向图是一棵有向树。一个有向图的生成森林是由若干棵有向树组成的,它含有图中全部顶点,但只有构成若干棵不相交的有向树的弧。例如,图8-7(a)所示的有向图 $G_5$ 对应的生成森林如图8-7(b)所示。

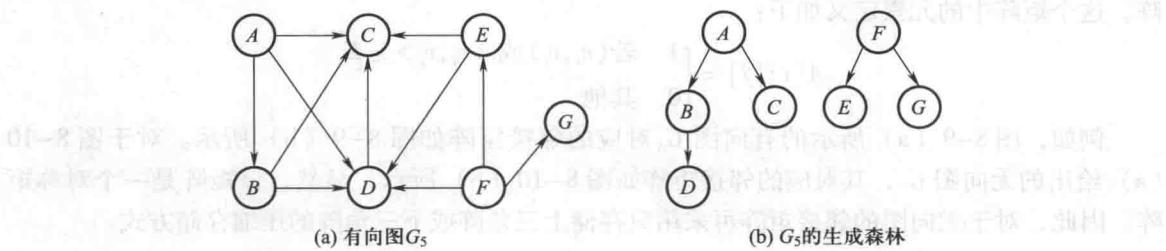


图 8-7 有向图及其生成森林示例

本章中仅考虑简单图的情形,即不考虑在一个顶点带自身环(如图8-8(a)所示)和两个顶点之间存在重复边(如图8-8(b)所示)的非简单图。

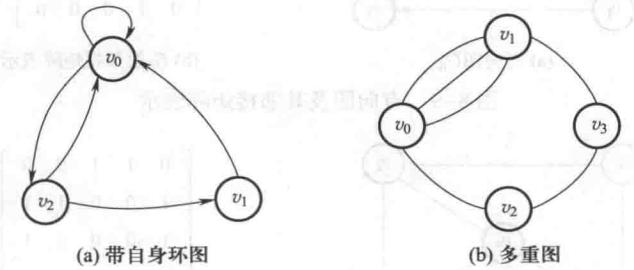


图 8-8 非简单图的示例

## 8.2 图的存储结构

图是一种比较复杂的数据结构，图中任意两个顶点之间都可能有关系。在存储一个图结构时，需要将构成图的两个集合，即顶点集与边（弧）集分别存储起来。对于不同的应用问题可以有不同的表示方法。本章主要介绍两种常用的图结构存储方法：邻接矩阵表示法与邻接表表示法。

### 8.2.1 邻接矩阵表示法

邻接矩阵表示法的基本思想是引入两个数组：一个用于记录图中各个顶点信息的一维数组，称为顶点表；另一个用于表示图中各个顶点之间关系的二维数组，称为邻接矩阵。

设图  $G = (V, E)$  是具有  $n(n > 0)$  个顶点的图，则图  $G$  所对应的邻接矩阵  $A$  是一个  $n$  阶方阵，这个矩阵中的元素定义如下：

$$A[i][j] = \begin{cases} 1 & \text{若 } (v_i, v_j) \text{ 或 } \langle v_i, v_j \rangle \in E \\ 0 & \text{其他} \end{cases}$$

例如，图 8-9 (a) 所示的有向图  $G_6$  对应的邻接矩阵如图 8-9 (b) 所示。对于图 8-10 (a) 给出的无向图  $G_7$ ，其对应的邻接矩阵如图 8-10 (b) 所示，显然，该矩阵是一个对称矩阵。因此，对于无向图的邻接矩阵可采用只存储上三角阵或下三角阵的压缩存储方式。

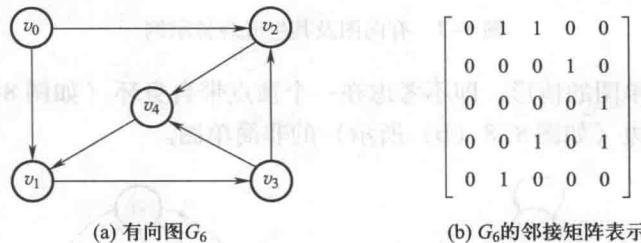


图 8-9 有向图及其邻接矩阵表示

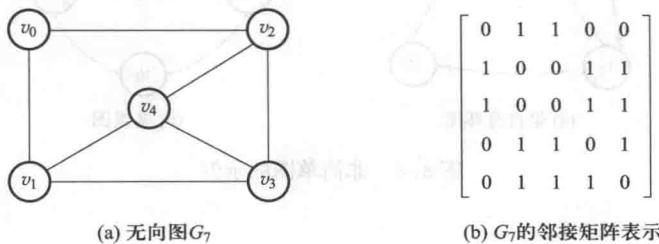


图 8-10 无向图及其邻接矩阵表示

对于带权图(网),需要对邻接矩阵中的元素值定义进行修改,让元素值表示相应顶点之间的边的权值,即有

$$A[i][j] = \begin{cases} w_{ij} & \text{若 } (v_i, v_j) \text{ 或 } \langle v_i, v_j \rangle \in E \\ 0 & i=j \\ \infty & \text{其他} \end{cases}$$

其中,  $\infty$  符号可用计算机中的一个足够大的数代替,以与边的权重相区别。例如,图 8-11 所示的是一个有向网  $N_1$  及其相应的邻接矩阵表示。

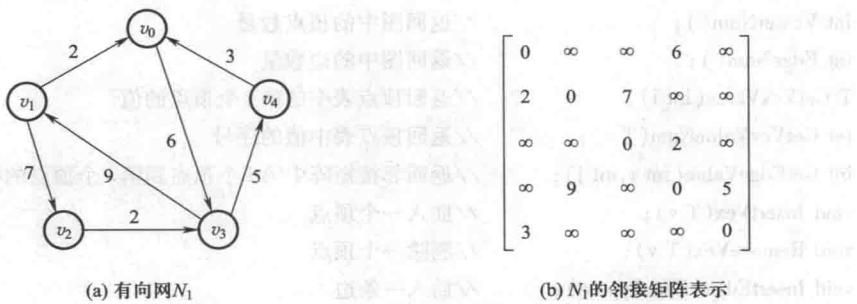


图 8-11 有向网及其邻接矩阵表示

基于邻接矩阵表示法的图类 MGraph 定义如下。

```

//图的类型定义:无向图、无向网、有向图、有向网
enum GraphType { undigraph, digraph, undinetwork, dinetwork } ;

template < class T >
struct EdgeType //本类型定义也适用于后面的邻接表结构
{
 T head, tail;
 int cost;
 EdgeType(T h, T t, int c)
 {
 head = h; tail = t; cost = c;
 }
};

template < class T >
class MGraph
{
private:

```

```

int vexnum,edgenum;
GraphType kind;
vector<vector<int>> edges; //邻接矩阵
vector<T> vexs; //顶点表
void DFS(int v, bool * visited); //连通分量的深度优先遍历

public:
MGraph(GraphType t,T v[], int n,int e);
~MGraph(){};
int VexterNum(); //返回图中的顶点数量
int EdgeNum(); //返回图中的边数量
T GetVexValue(int i); //返回顶点表中的第 i 个顶点的值
int GetVexValueNum(T v); //返回顶点表中值的序号
int GetEdgeValue(int i,int j); //返回邻接矩阵中第 i 个顶点到第 j 个顶点的权值
void InsertVex(T v); //插入一个顶点
void RemoveVex(T v); //删除一个顶点
void InsertEdge(EdgeType e); //插入一条边
void DeleteEdge(EdgeType e); //删除一条边
void DFSTraverse(); //深度优先遍历
void BFSTraverse(); //广度优先遍历
void PrintEdges(); //遍历邻接矩阵
void PrintVexs(); //遍历顶点表
void Prim(int v); //Prim 算法求最小生成树
void Kruskal(vector<EdgeType> &tree); //Kruskal 算法构造最小生成树
void Dijkstra(int v,int path[],int dist[]); /* 求单源最短路径 */
void Floyd(int path[][MAXV],int D[][MAXV]); /* 求各顶点对之间最短路径 */
...
};


```

对类 MGraph 中的邻接矩阵数据成员引入 vector 容器表示，而没有使用常规的二维数组表示，其目的是希望数组空间的大小能随着图中的顶点个数而动态变化。对于非带权图，vector 容器中的元素类型是 int 类型；对于网，vector 容器中的元素类型则与网中的权值类型相对应，如 int 或 float 类型等。

下面介绍类 MGraph 构造函数的实现。构造函数的主要任务就是建立图的邻接矩阵存储，即对图的顶点表和邻接矩阵成员分别进行初始化。下面以建立无向网的邻接矩阵存储结构为例，给出相应的构造函数实现，具体描述如算法 8-1 所示。

### 算法 8-1 无向网的邻接矩阵建立算法

```

template < class T >
MGraph < T > :: MGraph(GraphType t, T v[], int n, int e)
//参数 t 表示图的类型,参数 v 为存储各顶点值的数组,参数 n 和 e 分别为顶点数和边数
{
 vexnum = n;
 edgenum = e;
 kind = t;
 vexs. resize(vexnum);
 edges. resize(vexnum);
 for(i = 0; i < n; i++)
 vexs[i] = v[i];
 for(i = 0; i < n; i++)
 edges[i]. resize(vexnum);
 for(i = 0; i < n; i++)
 {
 for(j = 0; j < n; j++)
 if(i == j)
 edges[i][j] = 0; //网图邻接矩阵对角线元素为0
 else edges[i][j] = INFINITY;
 }
 for(i = 0; i < e; i++)
 {
 cin >> va >> vb; //输入一条边的两个端点的序号和边的权值
 cin >> w;
 edges[va][vb] = edges[vb][va] = w; //无向网图
 }
}

```

虽然算法 8-1 中设置了处理图类型的参数,但是没有对图类型的处理展开,对无向图、有向图和有向网的处理请读者自行完成。

采用邻接矩阵表示法存储图很容易确定图中任意两个顶点之间是否有边相连,也容易求出各个顶点的度。对于无向图,邻接矩阵的第  $i$  行(或第  $i$  列)中非零元素(或非 $\infty$  元素)个数为第  $i$  个顶点的度。对于有向图,邻接矩阵的第  $i$  行非零元素(或非 $\infty$  元素)个数为第  $i$  个顶点的出度,第  $i$  列非零元素(或非 $\infty$  元素)个数就是第  $i$  个顶点的入度。

基于邻接矩阵表示法还容易判断图中两个顶点之间是否有长度为  $m$  的路径相连,只需要考察矩阵  $A^m$  的相应元素值即可。

采用邻接矩阵表示法具有的缺点是邻接矩阵占用的存储单元个数只与图中的顶点个数有关，而与边的数目无关。这样对于顶点数较大的稀疏图；其对应的邻接矩阵中会存在大量的无用单元，从而会导致存储空间的浪费和较低的计算效率。

## 8.2.2 邻接表表示法

邻接表表示法可看成是针对邻接矩阵表示法的缺点的一种改进，其基本思想是对图中的每个顶点  $v_i$  都分别建立一个对应的单链表（称为“边表”）来存储所有邻接于顶点  $v_i$  的边（对于有向图是指以  $v_i$  为尾的弧），边表中的每个结点分别对应于邻接于顶点  $v_i$  的一条边，这样就可以把图中所有边的信息记录下来。边表中的每个结点主要包含两个域，其中邻接点域（Adjvex）指示与顶点  $v_i$  邻接的点在图中的位置，链域（Nextedge）指示下一条边或弧的结点。此外，对于图中所有顶点信息还需要使用一个一维数组（称为“顶点表”）来存放。对于顶点表中的每个顶点单元，需要存放的内容有该顶点信息值（Data）以及一个指向由邻接于该顶点的所有的边组成的边表的头指针（Firstedge）。

例如，图 8-12 和图 8-13 所示的是一个有向图与一个无向图的邻接表表示实例。对于无向图，由于每条边会在边表中出现两次，因此边表中的结点总数是图中顶点个数的两倍。

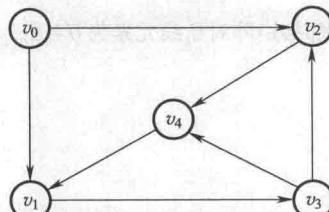
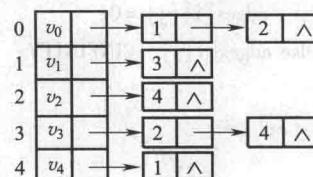
(a) 有向图  $G_6$ (b)  $G_6$  的邻接表表示

图 8-12 有向图及其邻接表表示

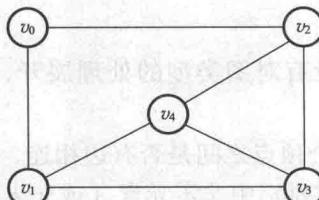
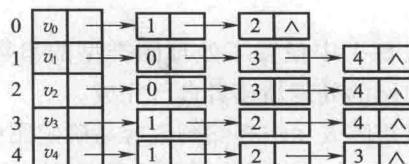
(a) 无向图  $G_7$ (b)  $G_7$  的邻接表表示

图 8-13 无向图及其邻接表表示

基于邻接表存储表示的图类 ALGraph 定义如下。

```

struct EdgeNode //边表的结点结构类型
{
 int adjvex; //该边的终点位置
 EdgeNode * nextedge; //指向第一条边的指针
};

template < class T >
struct VexNode //顶点表的元素结构类型
{
 T data; //顶点信息
 EdgeNode * firstedge; //指向该顶点对应的边表的指针
};

template < class T >
class ALGraph
{
public:
 int vexnum, edgenum; //图中的顶点数、边数
 vector < VexNode < T >> adjlist; //顶点表
 GraphType kind; //图的类型标记
 void DFS(int v, bool * visited); //连通图的深度优先遍历

 ALGraph(GraphType t, T vexs[], int n, int e); //构造函数
 ~ALGraph(); //析构函数
 EdgeNode * FirstEdge(int v); //返回第 v 个顶点对应的边表的头指针
 int VexterNum(); //返回图中的顶点数量
 int EdgeNum(); //返回图中的顶点数量
 T GetVexValue(int i); //返回图中的第 i 个顶点的值
 void InsertVex(T v); //插入一个顶点
 void RemoveVex(T v); //删除一个顶点
 void InsertEdge(EdgeType < T > e); //插入一条边
 void DeleteEdge(EdgeType < T > e); //删除一条边
 void DFSTraverse(); //深度优先遍历图
 void BFSTraverse(); //广度优先遍历图
 void TopoSort(); //拓扑排序算法
};

```

在存储一个网时，可在边表的每个结点结构中增加一个权值域（如 weight 域），以存储该边所关联的权值。

下面以建立无向图的邻接表存储结构为例，给出类 ALGraph 的相应构造函数实现，具体描述见算法 8-2。

### 算法 8-2 无向图的邻接表建立算法

```
template < class T >
ALGraph < T > :: ALGraph(GraphType t, T vexs[], int n, int e)
//参数 t 表示图的类型,参数 vexs 为存储各顶点值的数组,参数 n 和 e 分别为顶点数和边数
{
 EdgeNode * p;
 vexnum = n; //确定图的顶点个数和边数
 edgenum = e;
 kind = t;
 adjlist. resize(vexnum);
 for(i = 0; i < vexnum; ++ i) //初始化顶点表
 {
 adjlist[i]. data = vexs[i];
 adjlist[i]. firstedge = 0;
 }
 for(j = 0; j < edgenum; j++) //依次输入所有的边的信息
 {
 cin >> va >> vb; // 输入一条边邻接的两个顶点的序号
 p = new EdgeNode; //产生第一个表结点
 p -> adjvex = vb;
 p -> nextedge = adjlist[va]. firstedge; // 插在表头
 adjlist[va]. firstedge = p;
 p = new EdgeNode; // 产生第二个表结点
 p -> adjvex = va;
 p -> nextedge = adjlist[vb]. firstedge; // 插在表头
 adjlist[vb]. firstedge = p;
 }
}
```

采用邻接表表示法表示图结构的优点是在边稀疏的情况下可以节省存储空间，且易于确定图中任一顶点的度数和它的所有邻接点。该表示法的缺点是在邻接表中不能直接判定任意两个顶点之间是否有边相连。此外，使用该表示法也不方便找到指向某个顶点的所有边，需要扫描所有边表中的边结点以查看有多少个边结点中的邻接点域为某个指定的顶点。为此，可以设计有向图的逆邻接表。逆邻接表中与顶点  $v_i$  对应的边表中的每个边结点存储的是指向顶点  $v_i$  的入

弧信息。例如，图 8-14 所示的是有向图  $G_6$  的逆邻接表示例。

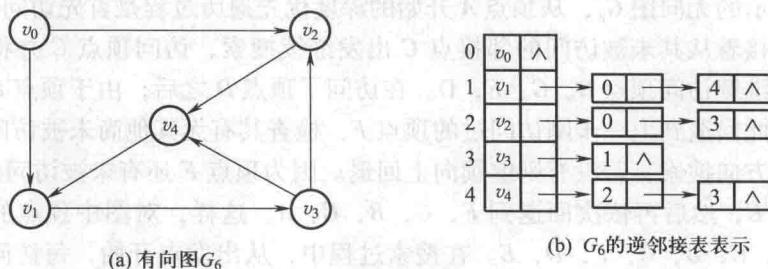


图 8-14 有向图及其逆邻接表表示

## 8.3 图的遍历

### 8.3.1 图遍历的概念

图的遍历操作是指从图中某一顶点出发，沿着某条搜索路径访问图中所有的顶点，使每个顶点仅被访问一次。图的遍历在概念上和树的遍历类似，但比树的遍历复杂。一方面，从图中的某个出发点开始遍历图并不能保证到达图中的所有顶点，因为图中可能存在非连通的情形；另一方面，因为图中的任一对顶点之间可能存在多条路径，即图中可能存在回路，在访问完某个顶点之后可能会沿着某些边又回到了曾经访问过的顶点，所以图的遍历算法必须避免陷入一个无限循环。

为解决以上问题，图遍历算法中需要设置一个辅助数组 `visited[]` 来标记各顶点是否被访问过，它的初始状态为 `false`。在图的遍历过程中，一旦某顶点  $v_i$  被访问，就立即置 `visited[i]` 为 `true`，以防止其被多次访问。

根据遍历过程中搜索规律的不同，图的遍历方法主要分为深度优先搜索与广度优先搜索两种。

### 8.3.2 深度优先搜索

图的深度优先搜索类似于树的先根遍历，其搜索规则可递归定义如下：

- ① 假设初始状态是图中所有顶点都未被访问，则可从图中某个顶点  $v$  出发，访问此顶点，然后依次从  $v$  的未被访问的邻接点出发深度优先遍历图，直至所有与  $v$  有通路的顶点都被访问到；
- ② 若此时图中还有顶点未被访问到，则另选一个未被访问的顶点作起点，重复步骤①，直到图中所有顶点都被访问到。

顾名思义，图的深度优先遍历算法所遵循的搜索策略是尽可能深地搜索一个图。例如，对图 8-15 (a) 所示的无向图  $G_8$ ，从顶点  $A$  开始的深度优先遍历过程是首先访问出发点  $A$ ，将其标记为已访问；接着从其未被访问的邻接点  $C$  出发继续搜索，访问顶点  $C$  并将其标记为已访问；依此类推，继续访问顶点  $B, G, F, D$ 。在访问了顶点  $D$  之后，由于顶点  $D$  的所有邻接点均已访问，因此回退到上一步刚访问过的顶点  $F$ ，检查其有无其他尚未被访问的邻接点。若有则继续向深度方向搜索，若没有则继续向上回退。因为顶点  $F$  还有未被访问的邻接点  $E$ ，所以接着访问顶点  $E$ ，然后再依次回退到  $F, G, B, C, A$ 。这样，对图中顶点的深度优先遍历的访问顺序为  $A, C, B, G, F, D, E$ 。在搜索过程中，从出发点开始，每访问一个新的顶点必将经过一条边。这样为访问图中除出发点以外的  $n-1$  个顶点，将经过  $n-1$  条不同的边。若将所访问的  $n$  个顶点和经过的  $n-1$  条边都记录下来，就得到了一棵生成树，称为深度优先搜索生成树，如图 8-15 (b) 所示。

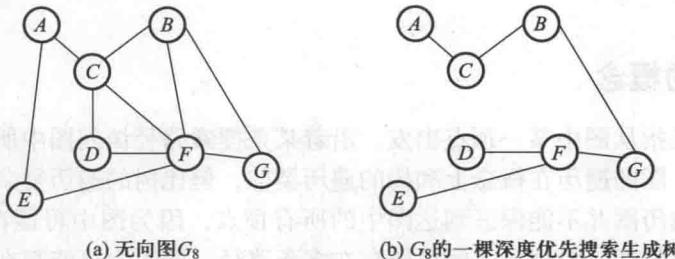


图 8-15 图的深度优先遍历

由于图的深度优先遍历规则是递归定义的，因此可以很容易地给出图的深度优先遍历算法。若以邻接矩阵作为图的存储结构，则一个连通图的深度优先遍历递归算法描述如算法 8-3 所示。

### 算法 8-3 连通图的深度优先遍历递归算法

```
template < class T >
void MGraph < T > ::DFS(int v, bool * visited)
{
 cout << vexs[v]; // 访问第 v 个顶点
 visited[v] = true; // 设置访问标志为 true(已访问)
 for(i = 0; i < vexnum; i ++)
 if(edges[v][i] == 1 && !visited[i])
 DFS(i, visited);
}
```

对于无向图来说，若此图是非连通图，则执行一次上述 DFS 算法只能访问初始出发点  $v$  所

在的连通分量的所有顶点，其他连通分量中的顶点是不可能访问到的。因此，需要从各个连通分量中选择一个出发点，分别调用 DFS 算法进行深度优先遍历。对于有向图来说，若从初始出发点到图中的其余各个顶点都有路径存在，则执行一次上述 DFS 算法，可以访问图中的所有顶点；否则，需要从未被访问的顶点中选择新的顶点作为出发点，调用 DFS 算法继续进行深度优先遍历，直到图中所有顶点均被访问到。对一般图的深度优先遍历算法描述如算法 8-4 所示。

#### 算法 8-4 图的深度优先遍历递归算法

```
template < class T >
void MGraph < T > ::DFSTraverse()
{
 bool * visited = new bool[vexnum]; //建立访问标记数组
 for(v = 0 ; v < vexnum ; v ++)
 visited[v] = false; //初始化访问标志数组（未被访问）
 for(v = 0 ; v < vexnum ; v ++)
 if(!visited[v])
 DFS(v, visited); //对尚未访问的顶点调用 DFS
 delete [] visited;
}
```

基于算法 8-4，可以判断一个图是否连通或求解一个非连通图中的连通分量个数，具体实现请读者自己思考。

下面分析算法 DFSTraverse 的时间复杂度。设图中有  $n$  个顶点、 $e$  条边。算法 DFSTraverse 主要时间耗费是在对 DFS 的调用上。从 DFSTraverse 中调用 DFS 和从 DFS 内部递归调用自己，总的调用次数是  $n$  次，而每次调用 DFS 算法的时间耗费则是对当前出发点查找其邻接点的过程。因此，深度优先遍历图的过程是对每个顶点查找其邻接点的过程。若以邻接矩阵作为图的存储结构，则调用一次 DFS 的时间复杂度为  $O(n)$ ，因此算法 DFSTraverse 的总的时间复杂度是  $O(n^2)$ 。若以邻接表作为图的存储结构，可类似推出算法的时间复杂度为  $O(n + e)$ 。

### 8.3.3 广度优先搜索

图的广度优先搜索类似于树的层次遍历。假设从图中某个顶点  $v$  出发，在访问了  $v$  之后，接着依次访问  $v$  的各个未曾访问过的邻接点  $w_1, w_2, \dots, w_t$ ；然后再依次访问与  $w_1, w_2, \dots, w_t$  邻接的且未曾访问过的顶点；依此类推，直至图中所有与初始出发点  $v$  有路径相通的顶点都已访问。若此时图中还有未被访问的顶点，则任选其中之一作为起点，重新开始上述过程，直至图中所有顶点都被访问到。

例如，对于图 8-15 (a) 所示的无向图  $G_8$  从顶点 A 出发进行广度优先遍历过程是，首先

访问出发点  $A$ ，将其标记为已访问；然后依次访问  $A$  的邻接点  $C$ 、 $E$ ，并置上相应的访问标记；接下来进入下一层，将先访问顶点  $C$  的未被访问的邻接点  $B$ 、 $F$ 、 $D$ ，再访问顶点  $E$  的未被访问的邻接点。因为顶点  $E$  的邻接点均已被访问，所以将继续进入下一层，首先访问顶点  $B$  的未被访问的邻接点  $G$ ；因为所有顶点均已被访问，所以搜索过程结束。这样，对图中顶点的广度优先遍历的访问顺序为： $A$ 、 $C$ 、 $E$ 、 $B$ 、 $F$ 、 $D$ 、 $G$ 。类似于图的深度优先遍历过程，若把广度优先遍历过程中所访问的顶点与所经过的边都记录下来，就得到了一棵广度优先搜索生成树，如图 8-16 所示。

显然，上述搜索过程的特点是尽可能先横向搜索，因此称为广度优先搜索。广度优先遍历是一种分层的搜索过程，每向前走一步可能访问一批顶点，不像深度优先搜索那样有往回退的情况。因此，广度优先搜索不是一个递归过程，其算法也不是递归的，其关键之处在于怎样保证先被访问的顶点的邻接点要先于后被访问的顶点的邻接点被访问。为了实现逐层按序访问，算法需要使用一个队列记忆正在访问的这一层和上一层的顶点，以便按序逐层访问。为避免重复访问，同样需要一个辅助数组  $\text{visited}[\cdot]$  给已访问过的顶点加标记。

若采用邻接矩阵作为图的存储结构，则图的广度优先遍历算法描述如算法 8-5 所示。算法 8-5 的结构类似于 7.4.2 节中的二叉树层次遍历算法 7-2。算法 8-5 中需要使用队列结构，本处使用了 3.2.2 节中定义的顺序队列结构，实际上也可以使用标准模板库中的队列类型，这样对于队列大小的定义和使用更为灵活。

### 算法 8-5 图的广度优先遍历算法

```
template < class T >
void MGraph < T > ::BFSTraverse()
{
 SeqQueue < int, 10 > q; //构造队列 q, 其长度为 10
 bool * visited = new bool [vexnum];
 for (i = 0; i < vexnum; i ++)
 visited[i] = false;
 for (i = 0; i < vexnum; i ++)
 {
 if(!visited[i])
 {
 cout << vexs[i];
 visited[i] = true;
 q.Enqueue(i);
 while (!q.IsEmpty())
 {
 int v = q.Dequeue();
 for (j = 0; j < adjvexnum[v]; j ++)
 if (!visited[adjvex[v][j]])
 cout << adjvex[v][j];
 visited[adjvex[v][j]] = true;
 }
 }
 }
}
```

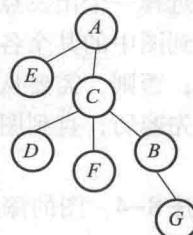


图 8-16  $G_8$  的一棵广度优先搜索生成树

```

 q. EnQueue(i);
 while(!q. Empty())
 {
 u = q. DeQueue();
 for(j = 0; j < vexnum; j++)
 {
 if(edges[u][j] == 1&&!visited[j])
 {
 cout << vexs[j];
 visited[j] = true;
 q. EnQueue(j);
 }
 }
 delete [] visited;
 }
}

```

算法 8-5 中广度优先遍历的时间耗费与深度优先遍历类似，也是对每个顶点分别查找其邻接点的过程。因此，广度优先遍历算法的时间复杂度与深度优先遍历算法相同。

### 8.3.4 图遍历算法的应用

因为很多基于图的问题的求解往往是在遍历过程中完成的，所以图的遍历算法是很多图处理算法的核心与基础。图的遍历算法包括深度优先搜索和广度优先搜索两种，在不同的应用场合可选择不同的遍历算法。下面基于两个算法设计问题分别给出这两种遍历算法的应用示例。

**例 8-1** 设计一个算法，找出连通图  $G$  中从顶点  $u$  到顶点  $v$  长度为  $n$  的所有简单路径。假设图  $G$  采用邻接表存储结构。

**算法设计思路：**本题可以利用深度优先搜索策略，基本方法是从顶点  $u$  开始，以深度优先方式搜索整个图。因为在执行对图的深度优先搜索过程中，可以找出所有从顶点  $u$  到  $v$  的路径，且由于在搜索过程中对每个顶点仅访问一次；所以搜索到的所有从顶点  $u$  到顶点  $v$  的路径一定都是简单路径。注意，为了通过执行一次深度优先遍历过程搜索出所有的从顶点  $u$  到顶点  $v$  的简单路径，在深度优先遍历的每次递归过程的最后，需要将本次递归中所访问的当前顶点的访问标记恢复为未访问状态。为了记录搜索过程中所经过的线路，需引入一个数组  $path$  和一个整型变量  $k$  分别用来保存所经过的路径和当前的路径长度。若当前扫描到的顶点为  $v$ ，且

路径长度为  $n$ ；则表明找到了一条路径，并输出它。具体的算法描述如算法 8-6 所示。

### 算法 8-6 简单路径的搜索算法

```

int path[MAXSIZE] ; //暂存遍历过程中的路径
bool visited[MAXSIZE] ; //标记顶点访问过的辅助数组,其初始元素值为 false
template < Class T >
void Find_All_Path(ALGraph < T > &G, int u, int v, int n, int k) // k 表示当前路径长度
{
 path[k] = u; //加入当前路径中
 visited[u] = true;
 if(u == v && k == n) //找到了一条简单路径
 {
 cout << "Found one path! \n";
 for(i=0; i <= n; i++)
 cout << path[i] << endl; //输出路径
 }
 else
 for(p = G. FirstEdge(u); p; p = p -> nextedge)
 {
 w = p -> adjvex;
 if(!visited[w])
 Find_All_Path(G, w, v, n, k+1); //继续寻找
 }
 visited[u] = false;
 path[k] = 0; //回溯
}
void main()
{
 ...
 Find_All_Path(G, u, v, n, 0); //在主函数中初次调用时 k 值应为 0
 ...
}

```

注意，在函数 `Find_All_Path()` 初次被调用时， $k$  值应为 0。

**例 8-2** 设计一个算法，判断一个给定的连通图  $G$  是否为二部图 (Bipartite)。假设图  $G$  采用邻接表存储结构。

首先给出二部图的定义。一个二部图  $G = (V, E)$  是一个无向图，它的顶点集合可以被划分

成  $V = X \cup Y$ , 并具有下述性质: 每条边  $e \in E$ , 它有一个端点在  $X$  中, 而另一个端点在  $Y$  中。图 8-17 所示的是一个二部图示例。

**算法设计思路:** 为了便于实现二部图的判定, 可以想象将集合  $X$  中的顶点着红色, 集合  $Y$  中的顶点着蓝色。测试一个图是二部图的任务可看成是对给定图的每个顶点着红色或蓝色, 并使得每条边都有一个红端点和一个蓝端点。这样可以设计一个二部图判断算法: 任取一个顶点  $s \in V$  并且将它着红色<sup>①</sup>, 接着找出顶点  $s$  的所有邻接点并将这些顶点都着为蓝色; 接下来再分别将这些顶点的所有邻接点都着为红色, ……如此交替对顶点进行着色, 直到图中所有顶点都被着色。如果这个过程给出了一个有效的红/蓝着色, 即其中每条边的两个端点的颜色均相反, 则该图是一个二部图; 否则, 若发现某条边的两个端点颜色相同, 则该图为非二部图。

很显然, 由于上述对图中顶点的着色过程基本与图的广度优先遍历过程一致, 因此可以在图的广度优先遍历算法的基础上给出上述图的顶点着色过程的实现, 即在广度优先遍历算法的实现中再加上一个记录所有顶点颜色的数组  $\text{color}[]$ , 并在遍历过程中相应地记录每个顶点的颜色, 进而给出二部图的判断算法。具体的算法描述如算法 8-7 所示, 其中使用了 3.2.2 节中的顺序队列。

### 算法 8-7 二部图的判定算法

```
template < class T >
bool Check_Bipartite()
{
 int * visited = new int[vexnum];
 int * color = new int[vexnum];
 SeqQueue < int, 20 > s;
 for(i = 0; i < vexnum; i++)
 visited[i] = false;
 visited[0] = true; //从第 0 个顶点开始处理
 color[0] = 1; //1 表示红色, -1 表示蓝色
 s. EnQueue(0);
 while(!s. Empty())
 {
 u = s. DeQueue();
 ...
```

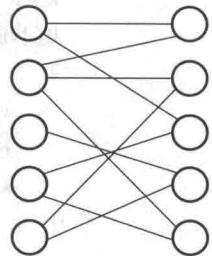


图 8-17 二部图示例

<sup>①</sup> 注意: 这样做没问题, 因为顶点  $s$  必须着某种颜色, 不妨将其着为红色。

```

for(p = FirstEdge(u) ; p ; p = p -> nextedge)
{
 k = p -> adjvex;
 if(!visited[k])
 {
 visited[k] = true;
 color[k] = - color[u];
 s. EnQueue(k);
 }
}
for(i = 0 ; i < vexnum ; i ++)
{
 for(p = FirstEdge(i) ; p ; p = p -> nextedge)
 {
 k = p -> adjvex;
 if(color[i] == color[k])
 return false;
 delete [] visited;
 delete [] color;
 return true;
 }
}
delete [] visited;
delete [] color;
return true;
}

```

算法 8-7 给出了对一个连通图  $G$  进行二部图判断的算法。如果图  $G$  是非连通图，那么该如何进行二部图判断？请读者自己思考。

## 8.4 最小生成树

### 8.4.1 最小生成树的概念及其性质

设  $G = (V, E)$  是一个无向连通网，生成树上各边的权值之和称为该生成树的代价。在  $G$  的所有生成树中，其中代价最小的生成树称为最小生成树（Minimum Spanning Tree, MST）。

假设要在  $n$  个城市之间建立一个通信网络至少需要架设  $n - 1$  条通信线路，而每两个城市之间架设一条通信线路的成本是不一样的，那么如何设计一个施工方案才能使得总造价最小呢？

用一个无向连通网来表示  $n$  个城市以及  $n$  个城市之间可能设置的通信线路，其中网的顶点表示城市，边的权值表示两个城市之间架设一条通信线路的代价。对于  $n$  个顶点的连通网可以建立许多不同的生成树，每棵生成树都可以是一个通信网。这样，设计一个总代价最小的通信网络方案的问题就是构造连通网的最小生成树的问题。

下面介绍一个用于构造最小生成树的基本性质（MST 性质）。

**MST 性质** 假设  $G = (V, E)$  是一个无向连通网， $U$  是顶点集  $V$  的一个非空子集，称顶点集合  $U$  和集合  $V - U$  构成顶点集合  $V$  的一个割（Cut）。对于割的每一条最小权值的交叉边  $(u, v)$ ，其中  $u \in U$ ,  $v \in V - U$ ，必存在一棵包含边  $(u, v)$  的最小生成树。

MST 性质也称为割性质。

**证明** 采用反证法。

假设  $G$  中任何一棵最小生成树都不包含  $(u, v)$ 。设  $T$  是一棵最小生成树，但不包含  $(u, v)$ 。由于  $T$  是最小生成树，所以  $T$  是连通的，因此，有一条从  $u$  到  $v$  的路径，且该路径上必有一条连接两个顶点集  $U$ 、 $V - U$  的边  $(u', v')$ ，其中  $u' \in U$ ,  $v' \in V - U$ ，如图 8-18 所示。当把边  $(u, v)$  加入到  $T$  中后，得到一个含有边  $(u, v)$  的回路。删除边  $(u', v')$ ，上述回路即被消除。由此得到另一棵生成树  $T'$ ， $T$  和  $T'$  的区别仅在于用边  $(u, v)$  替代了  $(u', v')$ 。因为  $(u, v)$  的权  $\leq (u', v')$  的权，所以  $T'$  的权  $\leq T$  的权，则  $T'$  一定是一棵最小生成树。这与假设矛盾，由此性质得证。

证毕。

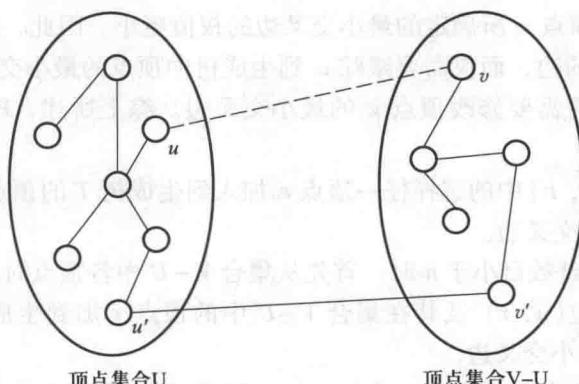


图 8-18 MST 性质证明示例

下面介绍两种最小生成树的构造算法：Prim 算法和 Kruskal 算法。这两种算法都可以看成是应用贪心算法设计策略的例子。尽管这两种算法所做出的贪心选择方式不同，但它们都利用了上述 MST 性质。

### 8.4.2 Prim 算法

设  $G = (V, E)$  是一个含有  $n$  个顶点的无向连通网， $T = (U, TE)$  是  $G$  的一棵最小生成树，其中  $U$  是  $T$  的顶点集， $TE$  是  $G$  的边集。

Prim 算法的基本思想是初始时从  $V$  中任取一个顶点（如  $v_1$ ），将其放入  $U$  中，使  $U = \{v_1\}$ ，这样集合  $U$  与集合  $V - U$  就构成图  $G$  的一个割。只要  $U$  是  $V$  真子集，就不断进行如下的贪心选择：选择一条权值最小的交叉边  $(v_i, v_j)$ ，其中  $v_i \in U$ ,  $v_j \in V - U$ ，并把该边  $(v_i, v_j)$  和其不在最小生成树中的顶点  $v_j$  分别加入  $T$  的顶点集  $U$  和边集  $TE$  中。不断重复这个过程直到图  $G$  中的  $n$  个顶点均被加入到最小生成树  $T$  中。

实现 Prim 算法的关键步骤是每次找出一条最短交叉边，然后将其加入最小生成树中。一种蛮力（Brute-force）做法是通过检查所有从最小生成树  $T$  中的顶点到最小生成树  $T$  外的顶点的边，找出一条最短边。例如，在查找第  $k$  条最短交叉边时，生成树  $T$  中已有  $k$  个顶点和  $k - 1$  条边，此时从  $T$  中顶点到  $T$  外顶点的边数最多可达  $k(n - k)$  条。从这么多的边中查找一条最短边，其时间复杂度可达  $O(k(n - k))$ 。显然，这种蛮力法的效率是很低的。是否存在更有效的改进算法呢？

可以注意到，逐个增加顶点到最小生成树中的过程实际上是一种增量式的变化。为实现 Prim 算法，可利用这种增量式变化的特点。由于关注的是从生成树  $T$  外的每个顶点到生成树  $T$  中顶点的最小交叉边，因此当将顶点  $v$  加入生成树  $T$  中时，对每一个非生成树  $T$  中顶点  $w$  而言，唯一可能引起的变化是增加顶点  $v$  可能使得顶点  $w$  比以前更接近生成树，即从顶点  $w$  到顶点  $v$  的权值可能比以前顶点  $w$  所确定的最小交叉边的权值更小。因此，并不需要检查从顶点  $w$  到生成树  $T$  中所有顶点的边，而仅需要跟踪  $w$  到生成树中顶点的最小交叉边，并检查当顶点  $v$  加入到生成树  $T$  中时是否需要修改顶点  $w$  的最小交叉边。综上所述，Prim 算法的基本过程可描述如下：

- ① 从连通网  $N = \{V, E\}$  中的选择任一顶点  $v_0$  加入到生成树  $T$  的顶点集合  $U$  中，并为集合  $V - U$  中的各顶点置最小交叉边。
- ② 当生成树  $T$  中顶点数目小于  $n$  时，首先从集合  $V - U$  中各顶点对应的最小权值边中选取最短边  $(u, v)$ ，然后将边  $(u, v)$  及其在集合  $V - U$  中的顶点  $v$  加到生成树  $T$  中，并调整集合  $V - U$  中各顶点对应的最小交叉边。

初始状态时，由于集合  $U$  中仅有一个顶点  $v_0$ ，因此对集合  $V - U$  中的各个顶点而言，其对应的最小权值可能是该顶点到  $v_0$  的边的权值或  $\infty$ （当该顶点与  $v_0$  之间没有边时）。

对于图 8-19 (a) 所示的无向连通网  $N_2$ , 使用上述 Prim 算法思想从顶点  $v_1$  出发构造最小生成树的过程如图 8-19 (b) 至图 8-19 (g) 所示。图中带阴影的顶点表示属于集合  $U$ , 即已经表示加入最小生成树中的顶点; 实线边表示已经加入最小生成树的边, 虚线边表示一个顶点属于集合  $U$  而另一个顶点不属于集合  $U$  的交叉边, 每条虚线边旁所标注的权值表示对集合  $V - U$  中的每个顶点当前所确定的最小交叉边的权值。

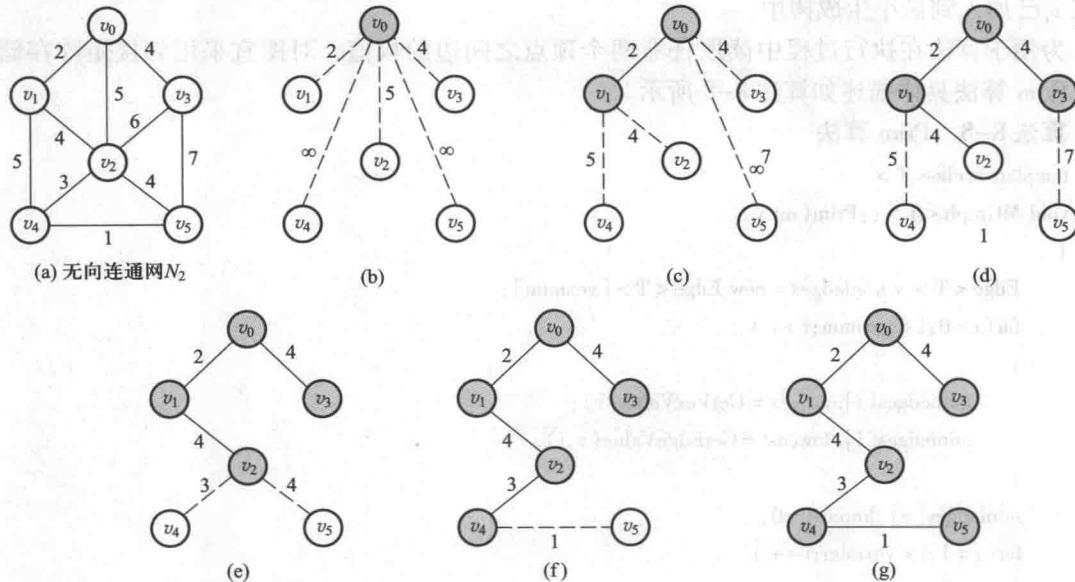


图 8-19 Prim 算法构造最小生成树的过程示例

为实现上述思想, 需要引入数据结构来记录下列信息:

① 从生成树  $T$  外的每个顶点 (即集合  $V - U$  中的每个顶点) 到生成树  $T$  中顶点 (即集合  $U$  中顶点) 的最小交叉边;

② 这些最小交叉边相应的权值。

引入一个辅助数组  $\text{miniedges}[]$ , 用于存放集合  $V - U$  中的各顶点到集合  $U$  中顶点的最小交叉边及其权值。 $\text{miniedges}[]$  数组的元素类型定义如下:

```
template < class T >
struct Edge
{
 T adjvex;
 float lowcost;
};
```

对于每个顶点  $v_i \in V - U$ , 在数组 miniedges[ ] 中对应一个相应元素 miniedges[  $i$  ] (假定顶点  $v_i$  的序号为  $i$ ), 且

$$\text{miniedges}[ i ]. \text{lowcost} = \text{Min} \{ \text{cost}( v_i, u ) \mid u \in U \}$$

其中,  $\text{cost}( v_i, u )$  表示从顶点  $v_i$  到顶点  $u$  的边的权值, adjvex 记录该最短边所关联的另外一个位于集合  $U$  中的顶点。若将某个数组元素 miniedges[  $i$  ] 的 lowcost 成员值设为 0, 则表示相应的顶点  $v_i$  已加入到最小生成树中。

为便于算法在执行过程中读取任意两个顶点之间边的权值, 对图宜采用邻接矩阵存储结构。Prim 算法具体描述如算法 8-8 所示。

### 算法 8-8 Prim 算法

```
template < class T >
void MGraph < T > ::Prim(int v)
{
 Edge < T > * miniedges = new Edge < T > [vexnum];
 for(i = 0 ; i < vexnum ; i ++)
 {
 miniedges[i]. adjvex = GetVexValue(v);
 miniedges[i]. lowcost = GetEdgeValue(v, i);
 }
 miniedges[v]. lowcost = 0;
 for(i = 1 ; i < vexnum ; i ++)
 {
 k = MiniNum(miniedges);
 cout << miniedges[k]. adjvex << " -->" << GetVexValue(k) << endl;
 miniedges[k]. lowcost = 0;
 for(j = 0 ; j < vexnum ; j ++)
 {
 if(GetEdgeValue(k, j) < miniedges[j]. lowcost)
 {
 miniedges[j]. adjvex = GetVexValue(k);
 miniedges[j]. lowcost = GetEdgeValue(k, j);
 }
 }
 }
 delete [] miniedges;
}
```

算法 8-8 中的 `MinNum()` 函数用于在数组 `miniedges` 中查找集合  $V - U$  中的具有最小权值的顶点，可以将它定义为私有成员函数，具体实现请读者自己完成。

分析 Prim 算法的时间复杂度。假设图中的顶点个数为  $n$ ，因为算法中存在两重 for 循环，所以算法的时间复杂度为  $O(n^2)$ ，与图中边的数量无关。

### 8.4.3 Kruskal 算法

Kruskal 算法从另一种途径构造网的最小生成树。

设  $G = (V, E)$  是一个含有  $n$  个顶点的无向连通网， $T = (U, TE)$  是  $G$  的一棵最小生成树。Prim 算法的特点是集合  $TE$  中的边总是形成单棵树，而在 Kruskal 算法中集合  $TE$  中的边是一个不断生长的森林。

Kruskal 算法的基本思想是：初始状态时， $U = V$ ， $TE = \emptyset$ 。初始状态的生成树可看成是一个森林，其中每棵树都由单个顶点构成。接下来不断地向生成树  $T$  中加入边，每加入一条边，则将森林中的某两棵树合并成一棵树，直到将初始森林中的  $n$  棵树最后合并成一棵树。为使生成树上总的权值之和达到最小，应使每一条边上的权值尽可能小，因此将  $G$  中的边按权值从小到大顺序依次选取。若选取的边不使  $T$  形成回路，则将它加入  $TE$  中；若选取的某条边使  $T$  构成回路，则将其舍弃。如此重复，直至选出  $n - 1$  条边。此时的  $T$  即为一棵最小生成树。

图 8-20 所示的是用 Kruskal 算法构造无向连通网  $N_2$  的最小生成树的过程。

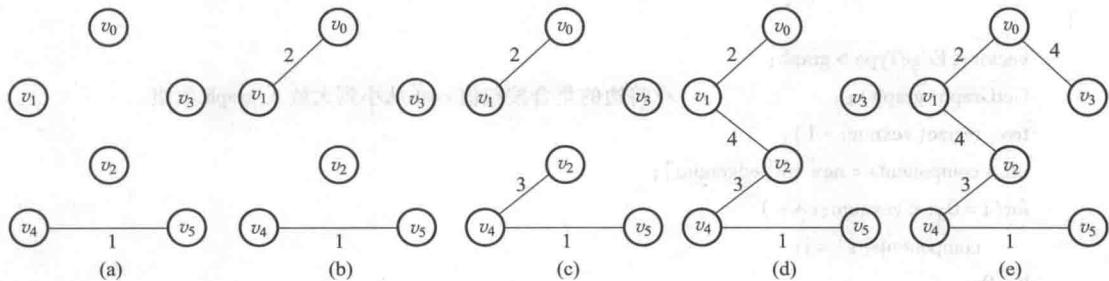


图 8-20 Kruskal 算法构造最小生成树的过程示例

在实现 Kruskal 算法时，可以将网中的所有边存储到一个边集数组中。例如，图 8-19 (a) 所示的无向连通网  $N_2$  对应的边集数组如图 8-21 所示。可以采用 8.2.1 节中的 `EdgeType` 结构体类型定义该边集数组。

为提高算法执行过程中查找最小权值边的速度，可以采用一种排序算法（如堆排序算法）对边集数组中的边按权值进行排序。接下来，Kruskal 算法的关键问题就是如何判断所选取的边加入  $T$  中是否会产生回路，即判断边的两个顶点是否在同一个连通分量（即森林中的一棵

| 下标 | 0     | 1     | 2     | 3     | 4     | 5     | 6     | 7     | 8     | 9     |
|----|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 始点 | $v_4$ | $v_0$ | $v_2$ | $v_0$ | $v_1$ | $v_2$ | $v_1$ | $v_0$ | $v_2$ | $v_3$ |
| 终点 | $v_5$ | $v_1$ | $v_4$ | $v_3$ | $v_2$ | $v_5$ | $v_4$ | $v_2$ | $v_3$ | $v_5$ |
| 权值 | 1     | 2     | 3     | 4     | 4     | 4     | 5     | 5     | 6     | 7     |

图 8-21 无向连通网  $N_2$  对应的边集数组表示

树) 中。这可以通过引入称为并查集 (Union – Find Set) 的数据结构来解决。

并查集是一种支持查找一个元素所属的集合以及两个元素各自所属集合的合并等运算的数据结构。并查集的一种最简单实现就是维护数组 components[ ]，数组元素 components[  $i$  ] 代表序号为  $i$  的顶点所属的集合 (即连通分量) 的编号。初始状态时数组元素 components[  $i$  ] 的值即为  $i$ 。当两个顶点的集合编号不同时，将这两个顶点所构成的边加入到生成树中一定不会形成回路。当这条边被加入生成树后，需要对这两个顶点所属的集合进行合并，也就是将两个顶点集合的编号统一。

在下面描述的 Kruskal 算法实现中，首先利用私有成员 GetGraph( ) 函数将图的边按权值排序后存入边集数组 graph 中，而边集数组 tree 则用于保存和返回算法所构造的最小生成树  $T$ 。GetGraph( ) 函数请读者自行完成。Kruskal 算法的具体描述如算法 8-9 所示。

### 算法 8-9 Kruskal 算法

```
template < class T >
void MGraph < T > ::Kruskal(vector < EdgeType > &tree)
{
 vector < EdgeType > graph;
 GetGraph(graph); // 将边的集合按权值 cost 从小到大放入 graph 数组
 tree. resize(vexnum - 1);
 int * components = new int[edgenum];
 for(i = 0 ; i < vexnum ; i ++)
 components[i] = i;
 k = 0;
 j = 0;
 while(k < vexnum - 1) // 最小生成树的边有 vexnum - 1 条
 {
 h1 = graph[j]. head;
 t1 = graph[j]. tail;
 h2 = components[h1];
 t2 = components[t1];
 if(h2 != t2) // 若两个顶点属于不同的集合，则该边可加入最小生成树中
 tree. push_back(graph[j]);
 k++;
 if(h2 < t2)
 for(i = t2 ; i < edgenum ; i++)
 components[i] = h2;
 else
 for(i = h2 ; i < edgenum ; i++)
 components[i] = t1;
 j++;
 }
}
```

```

 {
 tree[k].head = h1 ;
 tree[k].tail = t1 ;
 tree[k].cost = graph[j].cost ;
 k ++ ;
 for(i = 0 ; i < vexnum ; i ++) //将两个集合统一编号
 if(components[i] == t2)
 components[i] = h2 ;
 }
 j ++ ;
}
delete [] components ;
}

```

显然，Kruskal 算法的效率与所选择的排序算法的效率以及并查集数据结构的实现效率有关。若采用第 10 章介绍的比较高效的堆排序算法排序，并查集采用树结构实现，则 Kruskal 算法的时间复杂度可达到  $O(e \log_2 e)$ 。相比 Prim 算法而言，Kruskal 算法更适用于求解稀疏网（指边数较少的网）的最小生成树。

为了直观地显示 Kruskal 算法结果，可以增加一个非成员函数，根据边集数组 tree 中记录的内容输出最小生成树中边的序列，下面给出一种输出函数：

```

template < class T >
void PrintSubTree(MGraph < T > &g, vector < EdgeType > tree) //输出最小生成树
{
 i = 0 ;
 while(i < tree. size())
 {
 cout << g. GetVexValue(tree[i].head) << " -->" << g. GetVexValue(tree[i].tail) << endl ;
 i ++ ;
 }
}

```

在主函数中的调用函数 PrintSubTree() 的方法如下：

```

void main()
{
 ...
 MGraph < char > MG(t, A, n, e); //利用构造函数生成图的邻接矩阵
 ...
}

```

```

vector < EdgeType > Tree;
MG.Kruskal(Tree); //生成最小生成树
PrintSubTree(MG,Tree); //输出最小生成树
...
}

```

## 8.5 最短路径

### 8.5.1 最短路径的概念

对于一个不带权图，若从图中一个顶点到另外一个顶点存在一条路径，则称路径上所经过的边的数量为路径长度。由于在图中从一个顶点（源点）到另一个顶点（终点）可能存在多条不同的路径，各条路径的长度也可能不同，因此把路径长度最短（即路径经过的边数最少）的那条路径称为最短路径。

对于一个带权图，图中的各条边都关联一个相应的权值，通常将一条路径上经过的各边的权值之和称为路径长度，也称带权路径长度。在带权图中从源点到终点之间存在的多条不同路径中，把带权路径长度最短的那条路径称为最短路径。如果把不带权图中的每条边的权值都看成1，那么不带权图与带权图的最短路径的定义可以统一。

图的最短路径问题有非常广泛的应用。例如，要设计一个交通咨询系统，一般先采用图表示一个实际的交通网络，图中顶点表示各个城市，边的权值可表示两个城市之间的交通距离。基于这个交通网络，如果要查询从一个城市到另一个城市的最短交通路线，那么该问题实际就是一个图的最短路径问题。

求图的最短路径问题通常可分为两类。一类是求图中某顶点到其余各顶点的最短路径问题，也称为单源最短路径问题；另一类是求图中每对顶点之间的最短路径问题。

### 8.5.2 单源最短路径

单源最短路径问题是指给定一个带权有向图  $G$  与源点  $v$ ，求从顶点  $v$  到  $G$  中其他顶点的最短路径。本节所讨论的最短路径算法假定图中边的权值为非负的情形。

例如，对于图 8-22 所示的带权有向图  $G_0$ ，假定顶点  $v_0$  为源点，则从源点到图中其余各顶点的最短路径如表 8-1 所示。表中按最短路径长度递增的顺序依次给出了从源点到各顶点的最短路径。

仔细观察这张表可以发现，若按长度递增的顺序生成从源点到其他顶点的最短路径，则对于目前正在生成的最短路径而言，除终点以外，其余中间顶点的最短路径均已生成。例如，当

生成从源点  $v_0$  到顶点  $v_4$  的最短路径时，该路径上的中间顶点  $v_1, v_3, v_2$  的最短路径在此之前均已生成，这是因为从源点到它们的最短路径都比到顶点  $v_4$  的最短路径长度小。当然，从源点到其自身可看成存在一条长度为 0 的最短路径。

表 8-1  $G_9$  中从  $v_0$  到其余各顶点的最短路径

| 源 点   | 中 间 顶 点    | 终 点   | 最 短 路 径 长 度 |
|-------|------------|-------|-------------|
| $v_0$ |            | $v_1$ | 10          |
| $v_0$ |            | $v_3$ | 30          |
| $v_0$ | $v_3$      | $v_2$ | 50          |
| $v_0$ | $v_3, v_2$ | $v_4$ | 60          |

正是基于以上的观察与分析，Dijkstra 提出了一种按路径长度递增次序，逐步产生从源点到其他各顶点的单源最短路径求解思想：

① 假设从源点  $v_0$  到各终点  $v_1, v_2, \dots, v_k$  间存在最短路径，其路径长度分别为  $l_1, l_2, \dots, l_k$ 。假设  $l_p$  为其中的最小值，即从源点  $v_0$  到终点  $v_p$  的路径最短。显然这条路径上只有一条弧，否则它就不可能是所有最短路径中长度最短者。

② 第二条长度次短的（设从源点  $v_0$  到终点  $v_q$ ）最短路径只可能产生于下列两种情况：

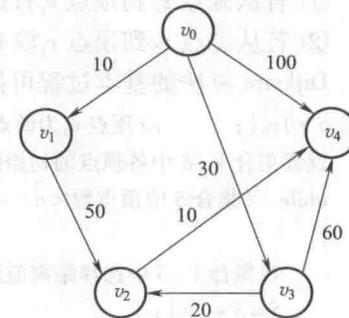
- 一种是从源点到该点有弧  $\langle v_0, v_q \rangle$  存在，且该弧构成  $v_0$  到  $v_q$  的最短路径；
- 另一种是从已求得最短路径的点  $v_p$  到该点有弧  $\langle v_p, v_q \rangle$  存在，且弧  $\langle v_0, v_p \rangle$  和  $\langle v_p, v_q \rangle$  上的权值之和小于弧  $\langle v_0, v_q \rangle$  的权值。

③ 类推到一般情况，假设已求得最短路径的顶点集合  $S = \{v_{p1}, v_{p2}, \dots, v_{pk}\}$ ，则下一条最短路径（设终点为  $x$ ）或者为弧  $\langle v_0, x \rangle$ ，或者为一条只经过  $S$  中的某些顶点而最后到达终点  $x$  的路径。

**证明** 假设此路径上存在一个不在集合  $S$  中出现的顶点，则说明存在终点不在  $S$  而路径长度比此更短的路径。因为是按长度递增的次序来产生各条最短路径的，所以这是不可能的。因此，长度比此路径短的所有路径均已产生，它们的终点必在集合  $S$  中。

**证毕。**

实现上述最短路径求解思想的 Dijkstra 算法与 Prims 算法非常相似，其基本做法是设置一个集合  $S$ ，用于存放已求出的最短路径的顶点，则尚未确定最短路径的顶点属于集合  $V - S$ 。初始状态时  $S$  中只有源点，然后不断地做贪心选择来逐步扩充这个集合，即总是在  $V - S$  中选择当前离源点距离最短或最近的顶点插入到集合  $S$  中，直到集合  $V - S$  中的顶点全部加入到集合  $S$  中。

图 8-22 带权有向图  $G_9$

为便于每次从  $V-S$  中选择当前离源点距离最短的顶点，需要引入一个辅助数组  $\text{dist}[]$ 。它的每一个分量  $\text{dist}[i]$  表示当前所确定的从源点  $v_0$  到终点  $v_i$  的最短路径的长度（最短距离）。数组  $\text{dist}[]$  的初始状态设置如下：

- ① 若从源点  $v_0$  到顶点  $v_i$  有边，则  $\text{dist}[i]$  为该边上的权值；
- ② 若从源点  $v_0$  到顶点  $v_i$  没有边，则  $\text{dist}[i]$  为  $+\infty$ 。

Dijkstra 算法的基本过程可描述如下：

```
 $S = \{v_0\}$; //顶点 v_0 为源点
```

```
设置集合 $V-S$ 中各顶点的初始距离值; //设置数组 dist 的元素值
```

```
while (集合 S 中顶点数 < n)
```

```
{
```

```
 在集合 $V-S$ 中选择距离值最小的顶点 v_j ;
```

```
 $S = S + \{v_j\}$;
```

```
 调整集合 $V-S$ 中剩余顶点的距离值; //修改数组 dist 的元素值
```

```
}
```

显然，算法执行过程中数组  $\text{dist}$  的元素值是一个增量的变化过程，在 while 循环的每一轮迭代中都需要修改数组  $\text{dist}$  的元素值。即当从集合  $V-S$  中选择一个距离最小的顶点  $v_j$  加入集合  $S$  后，需要调整集合  $V-S$  中剩余顶点的最短距离值。因为若加入顶点  $v_j$  作为中间顶点，则可能会使  $V-S$  中的某些终点的最短距离值更小，如图 8-23 所示。因此，对于  $V-S$  中的顶点  $v_i$ ，其最短距离值的调整方法是

$$\text{dist}[i] = \text{Min}\{\text{dist}[i], \text{dist}[j] + \text{cost}(j, i)\}$$

其中， $\text{cost}(j, i)$  表示从顶点  $v_j$  到顶点  $v_i$  的弧  $\langle v_j, v_i \rangle$  上的权值。若  $\langle v_j, v_i \rangle$  不存在，则置  $\text{cost}(j, i)$  为  $\infty$ 。

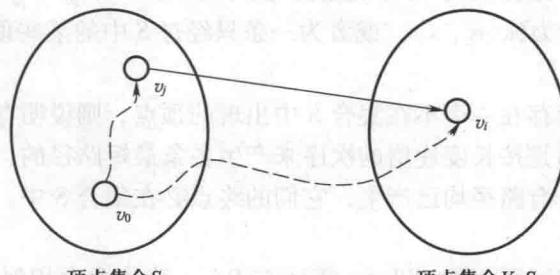


图 8-23 数组  $\text{dist}$  元素值的增量修改图解

图 8-24 所示的是 Dijkstra 算法的执行过程。对带权有向图  $G_9$ ，求从源点  $v_0$  到其他各顶点的最短路径的过程如图 8-24(b) 至图 8-24(f) 所示。图中带阴影的顶点表示已加入集合  $S$  中

的顶点，即已经确定最短路径的顶点；实线边表示已确定的最短路径上的边。集合  $V - S$  中的每个顶点旁所标注的数字表示当前所求出的从源点到该顶点到的最短路径长度（即相应的 dist 元素值），而集合  $S$  的每个顶点旁所标注的数字则表示已经确定的从源点到该顶点的最短路径长度值。

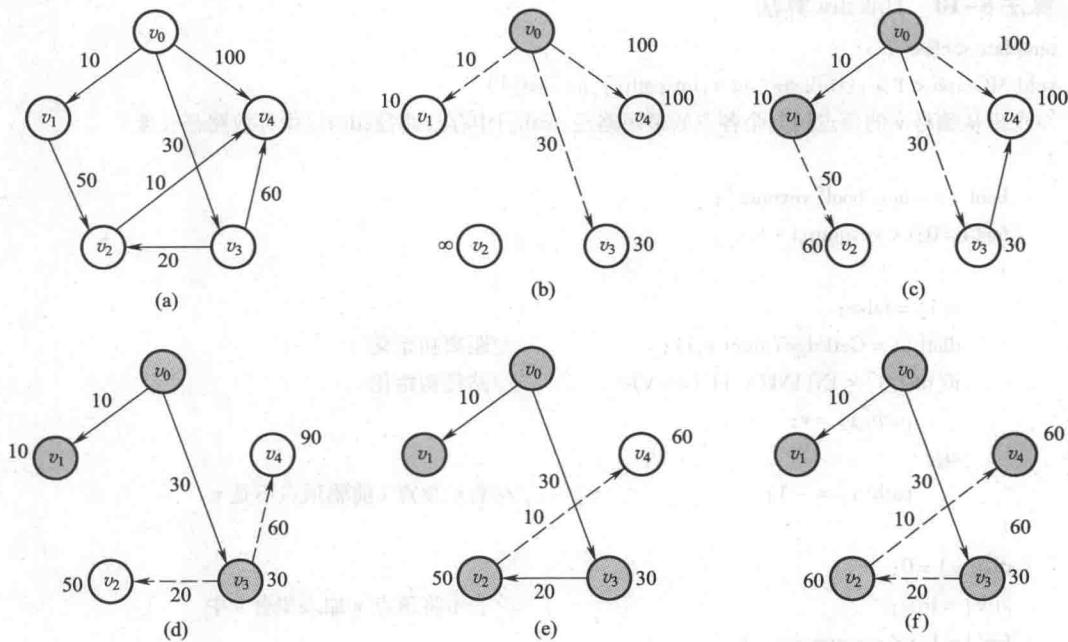


图 8-24 Dijkstra 算法求带权有向图  $G_9$  单源最短路径示例

对比图 8-24(a) 与图 8-24(f) 可以发现，对  $G_9$  求解单源最短路径最后所得到的图 8-24(f) 所示的实际上就是  $G_9$  的一棵有向生成树，因为在图 8-24(f) 中从树根（源点）到其余每个顶点均存在一条有向路径且无回路存在。称这棵有向生成树为  $G_9$  的最短路径树 (Shortest – Paths Tree)，简称 SPT 树。这样，对一个带权有向图求单源最短路径的问题实际上可看成是对一个带权有向图构造 SPT 树的问题。因此也就很容易理解，Dijkstra 算法与求最小生成树的 Prim 算法思想非常相似。

下面介绍 Dijkstra 算法的具体实现。为便于在算法执行过程中快速地求得任意两个顶点之间边的权值，图的存储结构宜采用邻接矩阵方式。为标识图中各顶点在算法执行过程中是否已被加入到集合  $S$  中，即是否已求出最短路径，设置一个一维数组  $s[]$ ，并规定

$$s[i] = \begin{cases} \text{false} & \text{顶点 } v_i \text{ 未加入集合 } S \\ \text{true} & \text{顶点 } v_i \text{ 已加入集合 } S \end{cases}$$

此外,为记录Dijkstra算法所求出的从源点到各顶点的最短路径,引入数组path[], path[i]中保存了从源点到终点 $v_i$ 的最短路径上该顶点的前驱顶点的序号。算法结束时,可根据数组path[]找到源点到 $v_i$ 的最短路径上每个顶点的前驱顶点,并一直回溯至源点,从而推出从源点到 $v_i$ 的最短路径。Dijkstra算法具体描述如算法8-10所示。

### 算法8-10 Dijkstra算法

```
template < class T >
void MGraph < T > :: Dijkstra(int v, int path[], int dist[])
//求出从编号v的顶点到其余各点的最短路径, path[] 中存放路径, dist[] 中存放路径长度
{
 bool * s = new bool[vexnum];
 for(i=0; i < vexnum; i++)
 {
 s[i] = false;
 dist[i] = GetEdgeValue(v, i);
 if(dist[i] < INFINITY || i == v) //距离初始化
 path[i] = v;
 else //路径初始化
 path[i] = -1; //表示顶点i前驱顶点不是v
 }
 dist[v] = 0;
 s[v] = true;
 for(i=1; i < vexnum; i++)
 {
 min = INFINITY; //设置最短路径初值为足够大的数
 for(j=0; j < vexnum; j++)
 if(!s[j] && dist[j] < min)
 k = j;
 min = dist[j];
 s[k] = true; //将离v最近的顶点加入集合s中
 for(int w=0; w < vexnum; w++)
 if(!s[w] && dist[w] > dist[k] + GetEdgeValue(k, w))
 {
 dist[w] = dist[k] + GetEdgeValue(k, w);
 path[w] = k;
 }
 }
}
```

```
delete []s;
```

读者可以增加一个非成员函数 PrintPath( MGraph < T > &g, int path[ ], int dist[ ], int v ) , 用于按数组 path[ ] 和 dist[ ] 中记录的值输出从源点  $v_0$  到各终点的最短路径及其长度。例如，对图  $G_9$  执行算法 8-10 求出源点  $v$  到各终点的最短路径后所计算出的 path 数组如下：

|    |   |   |   |   |
|----|---|---|---|---|
| -1 | 0 | 3 | 0 | 2 |
|----|---|---|---|---|

输出  $v_0$  到终点  $v_4$  的最短路径的过程是  $path[4] = 2$ ，说明该路径的前驱顶点是  $v_2$ ；  $path[2] = 3$ ，说明顶点  $v_2$  的前驱顶点是  $v_3$ ；  $path[3] = 0$ ，回退到源点，则推出源点  $v_0$  到终点  $v_4$  的最短路径是  $v_0 \rightarrow v_3 \rightarrow v_2 \rightarrow v_4$ 。函数 PrintPath( ) 的具体实现请读者自行完成。

容易看出，Dijkstra 算法的时间复杂度为  $O(n^2)$ 。

由于 Dijkstra 算法严格按照路径长度递增的顺序来求最短路径，因此该算法仅适用于图中边的权值为非负的情形。如果图中存在边的权值为负的情形，那么需要采用其他最短路径求解算法，如 Bellman – Ford 算法。

### 8.5.3 每对顶点之间的最短路径

求图中每对顶点之间的最短路径是指已知一个带权有向图，对每一对顶点  $v_i \neq v_j$ ，求出  $v_i$  与  $v_j$  之间的最短路径和最短路径长度。

对于各边权值为非负的带权有向图，求解每对顶点之间的最短路径问题的一种方法是在 Dijkstra 算法的基础上分别以图中每个顶点为源点，调用 Dijkstra 算法  $n$  次，即可求出图中每对顶点之间的最短路径，该方法的时间复杂度为  $O(n^3)$ 。本节将介绍另一种求解每对顶点之间最短路径问题的算法——Floyd 算法。Floyd 算法允许存在权值为负的边，但假设不存在权值为负的回路。

Floyd 算法主要基于以下观察。设图  $G$  的顶点集  $V = \{v_0, v_1, \dots, v_{n-1}\}$ ，对某个序号  $k$  考虑顶点的一个子集  $\{v_0, v_1, \dots, v_k\}$ 。对任意一对顶点  $v_i, v_j \in V$ ，考察从  $v_i$  到  $v_j$  且中间顶点皆属于集合  $\{v_0, v_1, \dots, v_k\}$  的所有路径，即中间顶点序号不大于  $k$  的所有路径，设  $p$  是其中的一条最短的路径。Floyd 算法利用路径  $p$  与各顶点对之间的所有中间顶点都属于  $\{v_0, v_1, \dots, v_{k-1}\}$  时的最短路径之间的联系，这一联系依赖于顶点  $v_k$  是否是路径  $p$  上的一个中间顶点。

(1) 如果  $v_k$  不是路径  $p$  上的中间顶点，那么  $p$  的所有中间顶点均在集合  $\{v_0, v_1, \dots, v_{k-1}\}$  中，因此顶点  $v_i$  与  $v_j$  之间最短路径上的所有中间顶点均属于集合  $\{v_0, v_1, \dots, v_{k-1}\}$ 。

(2) 如果  $v_k$  是路径  $p$  上的中间顶点，则可将路径  $p$  分解为  $v_i \xrightarrow{p_1} v_k \xrightarrow{p_2} v_j$ ，如图 8-25 所示。

由于一条最短路径的子路径必然也是一条最短路径，因此可知  $p_1$  是一条从  $v_i$  到  $v_k$  的最短路径，且其中所有中间顶点皆属于集合  $\{v_0, v_1, \dots, v_{k-1}\}$ 。类似地，可以推出  $p_2$  是一条  $v_k$  到  $v_j$  且所有中间顶点皆属于集合  $\{v_0, v_1, \dots, v_{k-1}\}$  的最短路径。

基于以上观察与分析，可知 Floyd 算法的基本思想是假设求从顶点  $v_i$  到  $v_j$  的最短路径，如果从  $v_i$  到  $v_j$  有弧，则从  $v_i$  到  $v_j$  存在一条长度为其权值的路径，但该路径不一定是最短路径。因为可能存在一条从  $v_i$  到  $v_j$ ，但包含有其他顶点为中间点的路径。

因此，需进行  $n$  次试探，测试从  $v_i$  到  $v_j$  能否有以顶点  $v_0, v_1, v_2, \dots, v_{n-1}$  为中间点的更短路径。首先，考虑路径  $v_i, v_0, v_j$  是否存在，若存在，则比较  $\langle v_i, v_j \rangle$  和  $v_i, v_0, v_j$  两条路径的长度，取其较短者作为从  $v_i$  到  $v_j$  中间顶点的序号不大于 0 的最短路径。接着，在路径上再增加一个中间顶点  $v_1$ 。如果  $v_i, \dots, v_1$  和  $v_1, \dots, v_j$  分别是当前找到的中间顶点的序号不大于 0 的最短路径，那么  $v_i, \dots, v_1, \dots, v_j$  就有可能是  $v_i$  到  $v_j$  的中间顶点序号不大于 1 的最短路径。将它和已经得到的从  $v_i$  到  $v_j$  的中间顶点序号不大于 0 的最短路径进行比较，取长度较短者作为中间顶点序号不大于 1 的最短路径。依此类推，经过  $n$  次试探后，最后得到的必是从  $v_i$  到  $v_j$  的最短路径。

实现 Floyd 算法的关键是保留每一步所求得的所有顶点对之间的当前最短路径长度，为此定义一个  $n$  阶方阵的序列

$$D^{(-1)}, D^{(0)}, \dots, D^{(n-1)}.$$

其中， $D^{(-1)}[i][j] = \text{cost}(i, j)$ ； $D^{(k)}[i][j] = \min \{D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j]\}$ ， $k=0, 1, \dots, n-1$ 。

$D^{(0)}[i][j]$  是从顶点  $v_i$  到  $v_j$  中间顶点的序号不大于 0 的最短路径的长度， $D^{(k)}[i][j]$  是从顶点  $v_i$  到  $v_j$  中间顶点的序号不大于  $k$  的最短路径的长度， $D^{(n-1)}[i][j]$  就是从顶点  $v_i$  到  $v_j$  的最短路径长度。基于以上定义的方阵序列，Floyd 算法的执行过程就是自底向上的按  $k$  值递增顺序计算  $D^{(k)}$  的过程。

Floyd 算法中仅用一个矩阵  $D$  来保存当前所有顶点对的最短路径长度。为保存算法所生成的最短路径本身，需要引入一个二维数组  $\text{path}[\text{MAXV}][\text{MAXV}]$  来保存最短路径（ $\text{MAXV}$  是表示图中最大顶点个数的常量）。在算法的第  $k+1$  轮迭代中，即生成矩阵  $D^{(k)}[i][j]$  时， $\text{path}[i][j]$  中存放的是从顶点  $v_i$  到顶点  $v_j$  中间顶点序号不大于  $k$  的最短路径上的顶点  $v_i$  的后继顶点的序号。在算法结束时，由二维数组  $\text{path}$  的值向后跟踪，可以得到从顶点  $v_i$  到顶点  $v_j$  的路径。若  $\text{path}[i][j] = -1$ ，则表示没有中间顶点。Floyd 算法的具体描述如算法 8-11 所示。

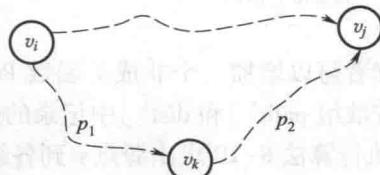


图 8-25 在顶点  $v_i$  和  $v_j$  之间试探加入顶点  $v_k$

### 算法 8-11 Floyd 算法

```

template < class T >
void MGraph < T > ::Floyd(int path[][MAXV] ,int D[][MAXV])
{
 for(i = 0 ;i < vexnum ;i ++)
 for(j = 0 ;j < vexnum ;j ++)
 {
 if(i == j)
 D[i][j] = 0 ;
 else
 D[i][j] = GetEdgeValue(i,j) ;
 if(D[i][j] < INFINITY)
 path[i][j] = j ;
 else
 path[i][j] = - 1 ;
 }
 for(k = 0 ;k < vexnum ;k ++)
 for(i = 0 ;i < vexnum ;i ++)
 for(j = 0 ;j < vexnum ;j ++)
 if(D[i][k] + D[k][j] < D[i][j])
 {
 D[i][j] = D[i][k] + D[k][j] ; //修改最短路径长度
 path[i][j] = path[i][k] ; //修改最短路径
 }
}

```

可以增加非成员函数 OutputPath( MGraph < T > &G, int path[ ][ MAXV ], int D[ ][ MAXV ] ) 用于输出保存于二维数组 path 中的所有路径以及保存于二维数组 D 中的路径长度, 请读者自行实现。

对图 G<sub>9</sub> 执行上述 Floyd 算法, 算法循环迭代过程中最短路径长度矩阵 D 和路径矩阵 path 的变化过程如图 8-26 所示。因为顶点  $v_0$  没有入边, 顶点  $v_4$  没有出边; 所以它们不能作为路径的中间顶点。因此,  $D^{(0)} = D^{(-1)}$ ,  $D^{(4)} = D^{(3)}$ ,  $path^{(0)} = path^{(-1)}$ ,  $path^{(4)} = path^{(3)}$ , 故这些矩阵在图中被省略。算法最后求出的结果由路径长度矩阵  $D^{(4)}$  和路径矩阵  $path^{(4)}$  给出。

容易看出, Floyd 算法的时间复杂度为  $O(n^3)$ 。

$$\begin{array}{l}
 D^{(-1)} = \begin{bmatrix} 0 & 10 & \infty & 30 & 100 \\ \infty & 0 & 50 & \infty & \infty \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 60 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix} \quad \text{path}^{(-1)} = \begin{bmatrix} 0 & 1 & -1 & 3 & 4 \\ -1 & 1 & 2 & -1 & -1 \\ -1 & -1 & 2 & -1 & 4 \\ -1 & -1 & 2 & 3 & 4 \\ -1 & -1 & -1 & -1 & 4 \end{bmatrix} \\
 D^{(0)} = \begin{bmatrix} 0 & 10 & 60 & 30 & 100 \\ \infty & 0 & 50 & \infty & \infty \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 60 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix} \quad \text{path}^{(0)} = \begin{bmatrix} 0 & 1 & 1 & 3 & 4 \\ -1 & 1 & 2 & -1 & -1 \\ -1 & -1 & 2 & -1 & 4 \\ -1 & -1 & 2 & 3 & 4 \\ -1 & -1 & -1 & -1 & 4 \end{bmatrix} \\
 D^{(1)} = \begin{bmatrix} 0 & 10 & 60 & 30 & 70 \\ \infty & 0 & 50 & \infty & 60 \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 30 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix} \quad \text{path}^{(1)} = \begin{bmatrix} 0 & 1 & 1 & 3 & 1 \\ -1 & 1 & 2 & -1 & 2 \\ -1 & -1 & 2 & -1 & 4 \\ -1 & -1 & 2 & 3 & 2 \\ -1 & -1 & -1 & -1 & 4 \end{bmatrix} \\
 D^{(2)} = \begin{bmatrix} 0 & 10 & 50 & 30 & 60 \\ \infty & 0 & 50 & \infty & 60 \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 30 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix} \quad \text{path}^{(2)} = \begin{bmatrix} 0 & 1 & 3 & 3 & 3 \\ -1 & 1 & 2 & -1 & 2 \\ -1 & -1 & 2 & -1 & 4 \\ -1 & -1 & 2 & 3 & 2 \\ -1 & -1 & -1 & -1 & 4 \end{bmatrix} \\
 D^{(3)} = \begin{bmatrix} 0 & 10 & 50 & 30 & 60 \\ \infty & 0 & 50 & \infty & 60 \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 30 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix} \quad \text{path}^{(3)} = \begin{bmatrix} 0 & 1 & 3 & 3 & 3 \\ -1 & 1 & 2 & -1 & 2 \\ -1 & -1 & 2 & -1 & 4 \\ -1 & -1 & 2 & 3 & 2 \\ -1 & -1 & -1 & -1 & 4 \end{bmatrix}
 \end{array}$$

图 8-26 Floyd 算法的迭代过程与最后结果

## 8.6 AOV 网与拓扑排序

### 8.6.1 有向无环图与 AOV 网的概念

不含环的有向图称为有向无环图 (Directed Acyclic Graph, DAG)。如果无向图中没有环, 那么它的每个连通分量都是一棵树。但是, 对于有向图, 可能没有环但仍旧有一个丰富的结构。例如, 某图可能有大量的边, 如果从顶点集合  $\{1, 2, \dots, n\}$  开始, 并且只要  $i < j$ , 就包含一条边  $\langle i, j \rangle$ , 那么得到有向图有  $\binom{n}{2}$  条边, 但是没有环。

由于有向无环图可以用一种自然的方式对优先关系或依赖关系进行描述, 因此在工程计划与管理方面有广泛而重要的应用。一个大的工程往往可以分解为若干相对独立的子工程 (活动), 子工程之间在进行的时间上有一定的相互制约关系。将这些子工程之间的先后关系用有向图表示, 其中顶点表示活动, 有向边表示活动之间的优先制约关系, 称这种有向图为顶点表示活动的网, 简称 AOV 网 (Activity on Vertex Network)。

例如, 一个计算机专业的学生必须学习一系列课程, 如表 8-2 所示。整个学习和训练过

程就是一项工程，每门课程学习就是一项活动，一门课程可能以其他几门课程为先修基础，而它本身又可能是另一些课程的先修基础，各课程之间的先修关系可以用图 8-27 (a) 所示的 AOV 网表示。

表 8-2 课程计划

| 课程代号           | 课程名称    | 先修课程                            |
|----------------|---------|---------------------------------|
| C <sub>1</sub> | 高等数学    | 无                               |
| C <sub>2</sub> | 程序设计语言  | 无                               |
| C <sub>3</sub> | 离散数学    | C <sub>1</sub>                  |
| C <sub>4</sub> | 数据结构    | C <sub>2</sub> , C <sub>3</sub> |
| C <sub>5</sub> | 编译原理    | C <sub>2</sub> , C <sub>4</sub> |
| C <sub>6</sub> | 操作系统    | C <sub>4</sub> , C <sub>7</sub> |
| C <sub>7</sub> | 计算机组成原理 | C <sub>2</sub>                  |

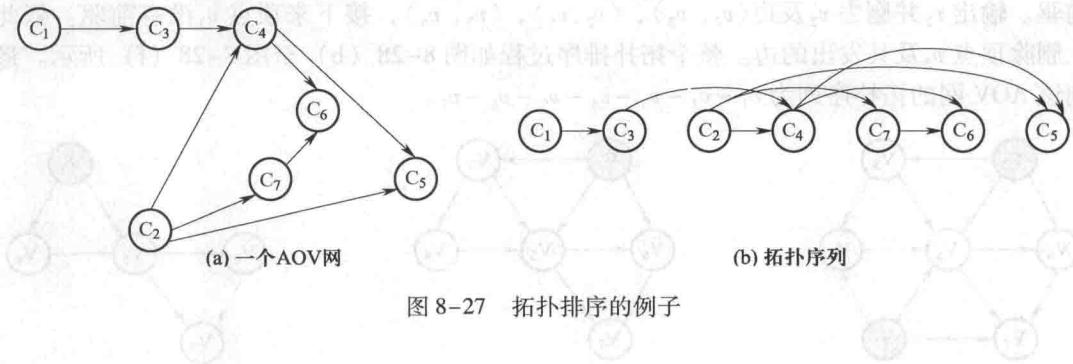


图 8-27 拓扑排序的例子

## 8.6.2 拓扑排序

采用 AOV 网表示一项工程以后，为了判断该工程的可行性，需要检查 AOV 网中是否存在环。在 AOV 网中是不允许存在回路的，因为回路的出现意味着某项活动的开工将以自身工作的完成为先决条件，这种情况称为死锁。检测有向图中是否存在环的一种常用方法是对一个有向图进行拓扑排序。

设图  $G = (V, E)$  是具有  $n$  个顶点的有向图， $V$  中的顶点序列  $v_1, v_2, \dots, v_n$  称为一个拓扑序列，当且仅当该顶点序列满足下列条件：若从  $v_i$  到  $v_j$  有一条路径，则在序列中  $v_i$  必排在  $v_j$  之前。对一个有向图构造其拓扑序列的过程称为拓扑排序。从数学上看，拓扑排序可看成是由顶点间的偏序关系构造全序关系的过程。图 8-27 (b) 说明了拓扑排序的直观含义。对 8-27 (a) 所

示的 AOV 网进行拓扑排序后，将沿水平线方向形成一个顶点序列，使图中所有有向边均从左指向右，如图 8-27 (b) 所示。

如何进行拓扑排序？从上述拓扑排序的定义可知，拓扑序列中的第一个顶点必定是 AOV 网中没有前驱的顶点。拓扑排序算法的基本步骤如下：

- ① 在 AOV 网中选择一个没有直接前驱的顶点（即此顶点入度为 0）并输出它；
- ② 从图中删去该顶点，同时删去所有从它发出的有向边；
- ③ 重复以上两步，直到全部顶点均已输出，拓扑序列就此形成，或者当前图中不存在无前驱的顶点。

显然，拓扑排序算法的结果可能有两种，一种是最终所生成的顶点拓扑序列包含图中的所有顶点，说明该 AOV 网中无环，也说明该 AOV 网对应的工程是可行的；另一种是算法执行完后，AOV 网中还有剩余顶点未被输出，说明该 AOV 网中存在环。从上述拓扑排序的执行过程也可以看出，一个 AOV 网对应的拓扑序列可能不唯一。

以图 8-28 (a) 所示的 AOV 网为例，图中带阴影的顶点表示入度为 0 的顶点。图中  $v_1$  和  $v_2$  没有前驱，可任选一个。假设先输出  $v_1$ ，在删除  $v_1$  及边  $\langle v_1, v_4 \rangle, \langle v_1, v_7 \rangle$  之后，只有顶点  $v_2$  没有前驱。输出  $v_2$  并删去  $v_2$  及边  $\langle v_2, v_3 \rangle, \langle v_2, v_5 \rangle, \langle v_2, v_6 \rangle$ ，接下来顶点  $v_3$  没有前驱。依此类推，删除顶点  $v_3$  及其发出的边。整个拓扑排序过程如图 8-28 (b) 至图 8-28 (f) 所示。最后得到该 AOV 网的拓扑序列为  $v_1 - v_2 - v_3 - v_4 - v_5 - v_6 - v_7$ 。

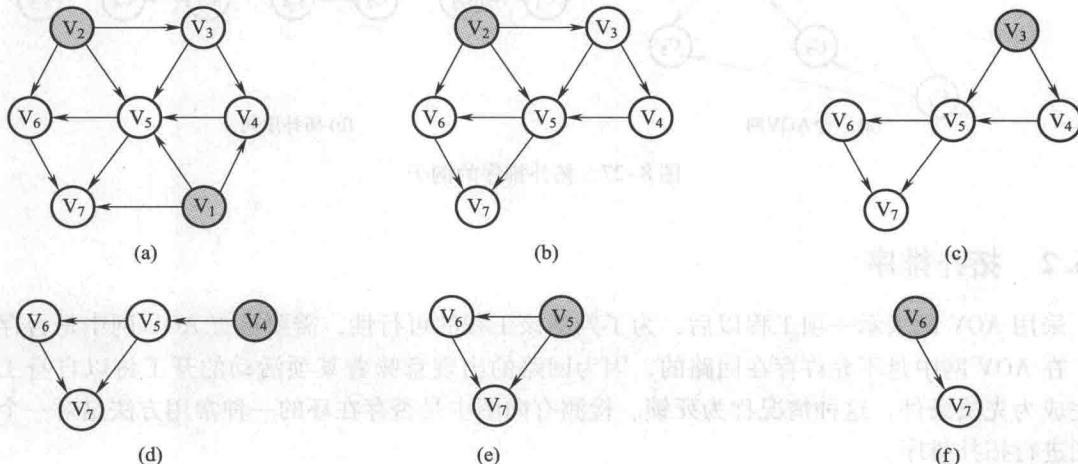


图 8-28 拓扑排序过程示例

为了实现拓扑排序算法，对于 AOV 网宜采用邻接表作为存储结构，因为在拓扑排序过程中需要不断查找各个人度为 0 顶点的所有出边。在实现上述拓扑排序算法过程中，关键需要解

决以下两个问题：

(1) 在每次选出一个入度为 0 的顶点后，如何实现从图中删去该顶点并删去所有从它发出的有向边的操作？

(2) 在拓扑排序过程中，如何方便地依次选出各个入度为 0 的顶点？

对于问题(1)，算法中从图中删去该顶点并删去所有从它发出的有向边的这类操作可以不必真正对图的存储结构来进行处理，而是用弧头顶点的入度减 1 的方法来间接实现。

对于问题(2)，为了便于算法执行过程中考察每个顶点的入度，即查找没有前驱的顶点，可以引入存放各顶点入度的数组 `indegree[ ]`。同时，为了避免每一步选入度为 0 的顶点时重复扫描数组 `indegree[ ]`，可设置一个队列（或栈）来存储所有入度为 0 的顶点。在进行拓扑排序之前，只要对顶点表扫描一遍，将所有入度为 0 的顶点都排入队列中。如果排序过程中出现新的入度为 0 的顶点，也同样将其排入队列中。算法 8-12 中使用了 3.2.2 节中的顺序队列。

综上所述，拓扑排序算法的具体描述如算法 8-12 所示。

### 算法 8-12 拓扑排序算法

```
template < class T >
void ALGraph < T > ::TopoSort()
{
 int * indegree = new int[vexnum];
 SeqQueue < int, 20 > s;
 for(i = 0 ; i < vexnum ; i ++)
 indegree[i] = 0;
 EdgeNode * p;
 for(i = 0 ; i < vexnum ; i ++) //求所有顶点的入度
 {
 p = adjlist[i]. firstedge;
 for(; p ; p = p -> nextedge)
 indegree[p -> advex]++;
 }
 for(i = 0 ; i < vexnum ; i ++) //使入度为 0 的顶点入队
 {
 if(!indegree[i])
 s. EnQueue(i);
 }
 c = 0; //对输出顶点计数
 while(!s. Empty())
 {
 cout << s. DeQueue();
 for(i = 0 ; i < vexnum ; i ++)
 if(!indegree[i])
 s. EnQueue(i);
 c++;
 }
}
```

```

 {
 i = s. DeQueue(); //出队
 cout << adjlist[i]. data; //输出顶点
 ++ c;
 p = adjlist[i]. firstedge;
 for(; p; p = p -> nextedge)
 {
 indegree[p -> adjvex] --;
 if(!indegree[p -> adjvex])
 s. EnQueue(p -> adjvex);
 }
 if(c < vexnum)
 cout << "该 AOV 网存在环!" << endl;
 delete[] indegree;
 }
}

```

拓扑排序算法是对图  $G$  进行遍历，依次访问入度为 0 的顶点所对应的边表。若 AOV 网中没有环，则扫描邻接表中的所有边结点，加上算法开始时为建立入度数组  $indegree[ ]$  而扫描整个邻接表的开销，可知此算法的时间复杂度为  $O(n + e)$ 。

除了上述拓扑排序算法之外，利用图的深度优先搜索过程也可以检查一个有向图中是否存在环。对于无向图来说，若在深度优先搜索中经过一条边后遇到一个已访问过的顶点，则图中必定存在环。但是，该性质对于有向图则未必成立。对于有向图，存在环性质：一个有向图  $G$  是无环的，当且仅当对图  $G$  进行深度优先搜索时没有得到反向边。所谓反向边是指在深度优先生成树中，连接顶点  $u$  到它的某一祖先顶点  $v$  的那些边。

基于环性质，在对有向图进行深度优先搜索中可以检查图中是否存在环，并进行拓扑排序。为检测图中是否存在反向边，可使用类似二叉树后序遍历的思想。即在深度优先搜索中，对当前被访问的任一顶点  $v$  做访问标记，但并不立即输出，而是将输出推迟到它的所有可达顶点（即后继）被输出之后。为了保证输出的顶点序列具有拓扑顺序，输出时应将顶点  $v$  插在它的所有后继顶点之前。这样，在深度优先搜索过程中每个顶点有未被访问、已访问和已输出等 3 种状态。对任一顶点  $v$ ，初始状态为未被访问。当一个顶点被访问后，其状态由未被访问变为已访问，接着检查它的每一个邻接点的状态。若某个邻接点的状态为已访问，则表明图中有环。有兴趣的读者可以自行实现。

## \* 8.7 AOE 网与关键路径

### 8.7.1 AOE 网的概念

DAG 图的另一个典型应用是边表示活动的网，简称 AOE 网（Activity on Edge Network），通常用它表示一个工程的计划或进度。AOE 网是一个带权有向图，图中的有向边表示活动（子工程）；边上的权值表示该活动的持续时间；顶点表示事件，每个事件是活动之间的转接点，即表示所有的人边活动到此结束，所有的出边活动从此开始。在 AOE 网中有两个特殊的顶点（事件）。一个称为源点（Source），表示所有活动的开始；另一个称为汇点（Convergence），表示整个工程的结束。

图 8-29 所示的是一个描述工程计划的 AOE 网。图中每条有向边  $\langle v_1, v_2 \rangle, \langle v_1, v_3 \rangle, \dots, \langle v_6, v_7 \rangle$  都表示一个活动，用  $a_1, a_2, \dots, a_{10}$  分别表示这些活动，边上的数字表示完成该活动（子工程）所需要的天数。图中的 7 个顶点表示 7 个不同的事件，如顶点  $v_1$  表示工程的开工， $v_7$  表示工程的结束， $v_5$  表示活动  $a_6, a_7$  完成和活动  $a_9$  可以开始。

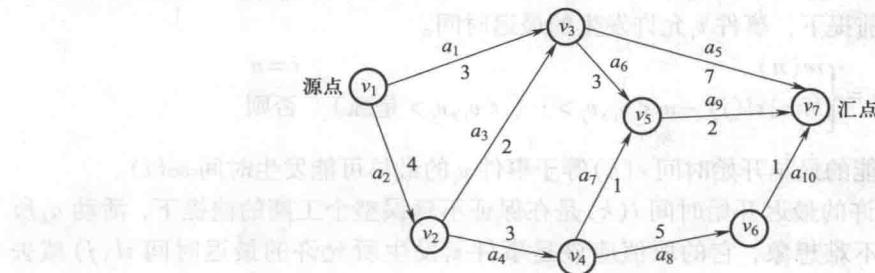


图 8-29 AOE 示例

表示一个实际工程计划的 AOE 网应当是无环且存在唯一的源点和汇点的。如果图中存在多个入度为 0 的顶点，那么可以添加一个虚拟源点，使这个虚拟源点到原来各个人度为 0 的顶点都有一条权值为 0 的边。对多个出度为 0 的顶点的情况可做类似的处理。与 AOV 网不同，对于 AOE 网有待研究的问题是完成整项工程至少需要多少时间和哪些活动是影响工程进度的关键。

### 8.7.2 关键路径

在 AOE 网中有些活动可以并行地进行。例如，在图 8-29 中，活动  $a_1$  和  $a_2$  就可以并行地进行。因此，完成整个工程的最短时间应是从源点到汇点的最长路径的长度（路径长度等于

路径上各边权值之和)。具有最大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 (Critical Activity)。例如，在图 8-29 中， $\{a_2, a_3, a_5\}$  和  $\{a_2, a_4, a_8, a_{10}\}$  是关键路径，其上的活动  $a_2, a_3, a_4, a_5, a_8, a_{10}$  为关键活动，工程完工的最短天数为 13 天。需要注意的是，在一个 AOE 网中可能有不止一条的关键路径。

分析关键路径的目的是找出关键活动，即不按期完成就会影响整个工程的完成时间的活动，从而为决策者提供调度依据。应当投入较多的人力和物力在关键活动上，以保证整个工程按期完成，并可争取提前完成。

如何在一个 AOE 网中找出关键路径呢？由于关键路径由关键活动组成，因此只要找出 AOE 网中的关键活动，就可以找到关键路径。为此需要先定义几个相关的量。假设顶点  $v_1$  为源点， $v_n$  为汇点，事件  $v_i$  的发生时刻为 0。

事件  $v_i$  可能发生的最早发生时间  $ve(i)$  是从源点到顶点  $v_i$  的最长路径长度。

$$ve(i) = \begin{cases} 0, & i = 1 \\ \max_j \{ve(j) + w(v_j, v_i)\} & (\langle v_j, v_i \rangle \text{ 是弧}) \\ \text{否则} & \end{cases}$$

其中， $w(v_j, v_i)$  表示边  $\langle v_j, v_i \rangle$  上的权值。

事件  $v_i$  允许的最迟发生时间  $vl(i)$  是在保证完成汇点  $v_n$  在  $ve(v_n)$  时刻发生的前提下，即在保证不延误整个工期的前提下，事件  $v_i$  允许发生的最迟时间。

$$vl(i) = \begin{cases} ve(n) & i = n \\ \min_j \{vl(j) - w(v_i, v_j)\} & (\langle v_i, v_j \rangle \text{ 是弧}) \\ \text{否则} & \end{cases}$$

活动  $a_k = \langle v_i, v_j \rangle$  可能的最早开始时间  $e(k)$  等于事件  $v_i$  的最早可能发生时间  $ve(i)$ 。

活动  $a_k = \langle v_i, v_j \rangle$  允许的最迟开始时间  $l(k)$  是在保证不延误整个工期的前提下，活动  $a_k$  所允许的最迟开始时间。不难想象，它的取值应该是事件  $v_j$  发生所允许的最迟时间  $vl(j)$  减去  $w(v_i, v_j)$ ，即

$$l(k) = vl(j) - w(v_i, v_j)$$

有了上述几个时间量就可以在 AOE 网中找出各个关键活动。显然，对于活动  $a_k$  而言，若该活动的最早开始时间  $e(k)$  与最迟开始时间  $l(k)$  重合，即  $l(k) = e(k)$ ，则表明该活动没有富余时间，由此可推出活动  $a_k$  一定是关键活动，活动  $a_k$  延期则整个工程延期。依据上述方法，对 AOE 网中的每条有向边可以分别判断其是否为关键活动。关键活动确定以后也就可以推出关键路径了。

这样，在 AOE 网中求关键路径的问题就转化为分别对图中的每个顶点  $v_i$  求出对应的两个量  $ve(i)$  和  $vl(i)$ ，进而推出每个活动  $a_k$  的最早开始时间  $e(k)$  与最迟开始时间  $l(k)$ 。在求顶点  $v_i$  的  $ve(i)$  之前，必须已经求得顶点  $v_i$  的所有前驱顶点的最早发生时间。这样，从源点起，需要按照顶点的拓扑排序，反复应用  $ve$  值的递推公式，方可求出各顶点  $v_i$  的最早开始时间  $ve(i)$ 。

类似地，从汇点  $v_n$  起，按照顶点的拓扑排序的逆序，反复应用  $vl$  值的递推公式，可求出各顶点  $v_i$  的最早开始时间  $vl(i)$ 。

综上所述，求 AOE 网中关键路径的算法过程可描述如下：

- ① 从源点  $v_1$  开始，置  $ve(1) = 0$ 。
- ② 对 AOE 网进行拓扑排序。如发现存在环，则无法求出关键路径，算法终止；否则在拓扑排序过程中依据  $ve$  值的递推公式可求出各个顶点  $v_i$  的最早开始时间  $ve(i)$ 。
- ③ 从汇点  $v_n$  开始，置  $vl(n) = ve(n)$ 。
- ④ 依据  $vl$  值的递推公式，按照拓扑排序的逆序，依次求出各个顶点  $v_j$  的最迟开始时间  $vl(j)$ 。
- ⑤ 根据各顶点的  $ve$  和  $vl$  值，分别求出每个活动的最早开始时间和最迟开始时间。
- ⑥ 对每个活动  $a_k = \langle v_i, v_j \rangle$ ，检测其是否满足  $l(k) = e(k)$ 。若是，则该活动为关键活动。

依据上述算法思想，图 8-29 所示的 AOE 网的计算结果如表 8-3 所示。

表 8-3 图 8-29 所示的 AOE 网的计算结果

| 顶点 $v_i$ | $ve(i)$ | $vl(i)$ | 活动 $a_k$ | $e(k)$ | $l(k)$ | $l(k)-e(k)$ |
|----------|---------|---------|----------|--------|--------|-------------|
| $v_1$    | 0       | 0       | $a_1$    | 0      | 3      | 3           |
| $v_2$    | 4       | 4       | $a_2$    | 0      | 0      | 0           |
| $v_3$    | 6       | 6       | $a_3$    | 4      | 4      | 0           |
| $v_4$    | 7       | 7       | $a_4$    | 4      | 4      | 0           |
| $v_5$    | 9       | 11      | $a_5$    | 6      | 6      | 0           |
| $v_6$    | 12      | 12      | $a_6$    | 6      | 8      | 2           |
| $v_7$    | 13      | 13      | $a_7$    | 7      | 10     | 3           |
|          |         |         | $a_8$    | 7      | 7      | 0           |
|          |         |         | $a_9$    | 9      | 11     | 2           |
|          |         |         | $a_{10}$ | 12     | 12     | 0           |

若 AOE 网中有多条关键路径，那么仅提高一条关键路径上的关键活动的速度并不能导致整个工程缩短工期，而必须提高同时在几条关键路径上的关键活动的速度才能有效。

## 本章小结

图是用于对个体集合上的二元关系进行编码的一种有效方式，是一种非常典型和常用的非

线性结构，具有广泛的应用背景。本章介绍了图的基本概念和两种常用的存储结构，对图的遍历、最小生成树、最短路径、拓扑排序、关键路径等问题进行了详细的讨论，并给出了相应的求解算法。

读者在学习本章时，应掌握下列学习要点：

- (1) 掌握图的概念和基本术语。
- (2) 掌握两种图的存储结构：邻接矩阵表示法和邻接表表示法，并掌握基于这两种存储结构的图的建立算法。
- (3) 掌握图的两种遍历方法：深度优先搜索与广度优先搜索的思想，并掌握这两种遍历方法的算法描述。
- (4) 掌握构造最小生成树的两种构造算法：Prim 算法和 Kruskal 算法的基本思想与算法描述，并学会应用它们解决实际问题。
- (5) 掌握求解单源最短路径问题的 Dijkstra 算法和求解每对顶点之间最短路径的 Floyd 算法的算法思想与算法描述，并学会应用它们解决实际问题。
- (6) 掌握 AOV 网和拓扑排序的概念，掌握拓扑排序算法的算法思想与算法描述。
- (7) 了解 AOE 网和关键路径的概念，了解在 AOE 网中求关键路径的算法的基本过程。

## 习题 8

- 8.1 假设以邻接表表示法作为图的存储结构，设计图的深度优先遍历递归算法。
- 8.2 对图 8-30 所示的连通网，分别用 Prim 算法和 Kruskal 算法构造该网的最小生成树。
- 8.3 试利用 Dijkstra 算法求在图 8-31 所示的有向图中从顶点 A 到其他顶点间的最短路径，写出执行过程中各步的状态。

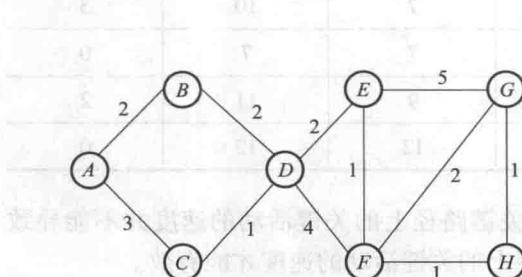


图 8-30 习题 8.2 的图例

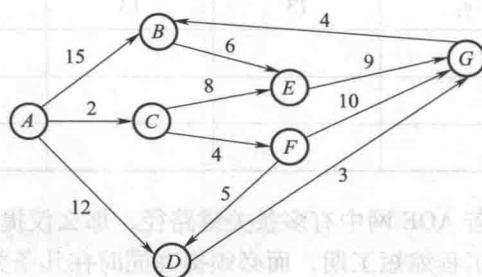


图 8-31 习题 8.3 的图例

- 8.4 试基于图的广度优先搜索策略编写一种算法，判断以邻接表方式存储的有向图中是否存在由顶点  $v_i$  到顶点  $v_j$  的路径 ( $i \neq j$ )。

8.5 试修改 Prim 算法，使之能在邻接表存储结构上实现求有向图的最小生成森林（森林的存储结构为孩子兄弟链表）。

8.6 采用邻接表存储结构，编写一个判断无向图中任意两个给定的顶点之间是否存在一条长度为  $k$  的简单路径的算法。

8.7 给定一个有向图和图中的两个顶点  $u$  和  $v$ ，试编写算法求有向图中从  $u$  到  $v$  的所有简单路径，并以图 8-32 所示的有向图为例手工执行该算法，画出相应的执行过程图。

8.8 假设以邻接矩阵作为图的存储结构，编写算法判断在给定的有向图中是否存在一个简单有向回路。若存在，则以顶点序列的方式输出该回路（找到一条即可）。

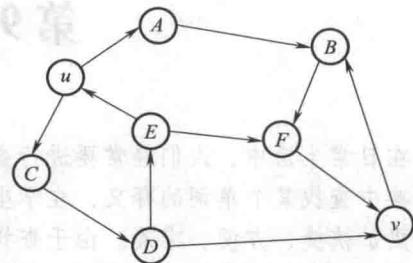


图 8-32 习题 8.7 的图例

## 上机实验题 8

**实验题 8.1** 编写程序，建立图的存储结构，并对图进行遍历操作。

基本要求：

- (1) 建立一个无向图的邻接矩阵。
- (2) 由第(1)步建立的邻接矩阵产生相应的邻接表结构。
- (3) 分别基于邻接矩阵和邻接表结构对图进行深度优先遍历与广度优先遍历。

**实验题 8.2** 编写程序，建立图的存储结构，并求出该图的最小生成树。

基本要求：

- (1) 建立一个无向网的邻接矩阵。
- (2) 分别用 Prim 算法和 Kruskal 算法构造最小生成树。
- (3) 以文本形式输出生成树中各条边以及它们的权值。

**实验题 8.3** 编写程序，建立图的存储结构，求出该图的最短路径。

基本要求：

- (1) 建立一个有向网的邻接矩阵。
- (2) 用 Dijkstra 算法计算从某个结点出发的最短路径并输出。
- (3) 用 Floyd 算法计算该图每对顶点之间的最短路径并输出。

**实验题 8.4** 编写程序，建立图的存储结构，对该图进行拓扑排序。

基本要求：

- (1) 建立一个有向图的邻接表结构。
- (2) 对该图进行拓扑排序并输出排序结果。

# 第9章 查找

在日常生活中，人们经常要进行查找工作，如在电话号码簿中查找某人的电话号码，在英汉字典中查找某个单词的释义，在学生成绩表中查找某个学生的成绩等。计算机的出现使信息查询更加快捷、方便、准确。由于查找是许多程序中最消耗时间的一种操作，因此一个好的查找方法会大大提高程序运行速度。本章将介绍若干种常用的查找算法。

## 9.1 查找的基本概念

### 1. 查找表

由具有同一类型的数据元素（或记录）组成的集合称为查找表。

对查找表经常进行的操作包括查询某个特定的记录是否在查找表中，查询某个特定的记录的各种属性，在查找表中插入记录，以及从查找表中删除记录等。

查找表可分为静态查找表和动态查找表两类。静态查找表是指仅能对查找表进行查询操作，而不改变表本身。动态查找表是指对查找表除进行查找操作外，还可向查找表中插入或删除记录，从而改变表的内容。

### 2. 关键字

关键字是记录中某个项或组合项的值，用它可以标识一个记录。能唯一确定一个记录的关键字称为主关键字，不能唯一确定一个记录的关键字称为次关键字。例如，对于学生成绩表，因为在学校中学生的学号是唯一的，而学生的姓名则可能有同名同姓的，所以“学号”可看成主关键字，“姓名”应视为次关键字。

### 3. 查找

查找是指按给定的某个值  $k$ ，在查找表中查找关键字为给定值  $k$  的记录。若表中存在这样的记录，则称查找成功，此时可给出找到记录的位置或信息；若表中不存在这样的记录，则称查找失败，此时可给出失败标志，如空记录或空指针等。

例如，当用计算机处理学生成绩时，可以使用如表 9-1 所示的结构存储。表中每一行为一条记录，学生的学号为记录的主关键字。

表 9-1 学生成绩表

| 学号       | 姓名  | 英语  | C++程序设计 | 数据结构 |
|----------|-----|-----|---------|------|
| 19130101 | 包蕾  | 88  | 90      | 85   |
| ...      | ... | ... | ...     | ...  |
| 19130124 | 陈晨  | 75  | 90      | 85   |
| 19130125 | 戴华  | 90  | 86      | 73   |
| 19130126 | 顾然  | 86  | 77      | 68   |
| ...      | ... | ... | ...     | ...  |
| 19130143 | 章建明 | 77  | 94      | 83   |

假设给定学号为 19130126，通过查找可得到学生顾然的各科成绩，此时查找成功。若给定值为 19130155，由于表中没有关键字为 19130155 的记录，因此查找失败。

#### 4. 平均查找长度

查找过程中进行的关键字比较次数的数学期望值称为平均查找长度（Average Search Length, ASL），通常将 ASL 作为衡量查找算法优劣的标准。

对具有  $n$  个记录的查找表，查找成功时的平均查找长度

$$ASL = \sum_{i=1}^n P_i C_i$$

其中， $P_i$  为查找第  $i$  个记录的概率， $C_i$  为查找第  $i$  个记录所需的关键字比较次数。

## 9.2 顺序表的查找

本节讨论的查找算法均采用顺序存储结构，并假定关键字为整数数组的长度为常量  $n$ ，查找表的类型可定义为

```
struct Record
{
 int key;
 ...
};

Record r[n];
```

### 9.2.1 顺序查找

顺序查找又称线性查找，是最基本的查找方法。顺序查找的基本思想是从查找表的一端开

始，向另一端逐个按给定值  $k$  与关键字进行比较。若找到，则查找成功，并给出记录在表中的位置；若整个表检测完，仍未找到与  $k$  相同的关键字，则查找失败，给出失败信息（返回 -1）。

### 1. 无监视哨的情况

查找成功时返回该对象的下标序号，失败时返回 -1，具体实现如算法 9-1 所示。

#### 算法 9-1 无监视哨的顺序查找

```
int SeqSearch(Record r[], int n, int k)
{
 int i = 0;
 while(i < n&&r[i].key != k)
 i++;
 if(i < n)
 return i;
 else
 return -1;
}
```

### 2. 有监视哨的情况

在主程序中需要多定义一个单元，用于存放待查找的元素。这样，每次循环只需要进行元素的比较，不需要比较下标是否越界，可以提高算法的执行效率。

查找成功时返回该对象的下标序号，失败时返回 -1，具体实现如算法 9-2 所示。

#### 算法 9-2 有监视哨的顺序查找

```
int SeqSearch2(Record r[], int n, int k)
{
 int i = 0;
 r[n].key = k;
 while(r[i].key != k)
 i++;
 if(i < n)
 return i;
 else
 return -1;
}
```

下面分析顺序查找的平均查找长度。就上述算法而言，对于  $n$  个记录的表，给定值  $k$  与表中第  $i$  个元素关键字相等时，需进行  $i$  次关键字比较，即  $C_i = i$ 。因此，查找成功时顺序查找的平均查找长度

$$\text{ASL} = \sum_{i=1}^n iP_i$$

设每个记录的查找概率相等，即  $P_i = \frac{1}{n}$ ，则等概率情况下有

$$\text{ASL} = \sum_{i=1}^n i \cdot \frac{1}{n} = \frac{n+1}{2}$$

查找不成功时，关键字的比较总次数是  $n+1$  次。

因此查找算法中的基本工作就是关键字的比较，因此查找长度的量级就是查找算法的时间复杂度，顺序查找算法的时间复杂度为  $O(n)$ 。

在许多情况下，查找表中记录的查找概率是不相等的。为了提高查找效率，查找表需依据查找概率越高比较次数越少，查找概率越低比较次数就较多的原则来存储记录。

顺序查找的缺点是当  $n$  很大时，平均查找长度较大、效率低；其优点是对表中记录的存储没有要求。此外，对于链表，只能进行顺序查找。

## 9.2.2 折半查找

当查找表是有序表时，可采用折半查找的方法。折半查找的基本思想是在有序表中，取中间元素作为比较对象。若给定值  $k$  与中间记录的关键字相等，则查找成功；若给定值  $k$  小于中间记录的关键字，则在表的左半区继续查找；若给定值  $k$  大于中间记录的关键字，则在表的右半区继续查找。重复上述查找过程，直到查找成功，或所查找的区域无记录，查找失败。算法步骤如下：

- ①  $\text{low} = 0; \text{high} = n - 1;$  // 设置初始区间
- ② 当  $\text{low} > \text{high}$  时，返回查找失败信息 // 表空，查找失败
- ③ 当  $\text{low} \leq \text{high}$  时， $\text{mid} = (\text{low} + \text{high}) / 2;$  // 取中点
- 若  $k < r[\text{mid}] . \text{key}$ ,  $\text{high} = \text{mid} - 1;$  转 2) // 查找在左半区进行
- 若  $k > r[\text{mid}] . \text{key}$ ,  $\text{low} = \text{mid} + 1;$  转 2) // 查找在右半区进行
- 若  $k = r[\text{mid}] . \text{key}$ , 返回记录在表中位置 // 查找成功

**例 9-1** 有序表按关键字排列为  $\{5, 13, 19, 21, 23, 29, 32, 35, 37, 42, 46, 49, 56\}$ ，在表中查找关键字为 23 和 22 的记录。

解 ① 查找关键字为 23 时的查找过程如图 9-1 所示。

② 查找关键字为 22 时的查找过程如图 9-2 所示。

下面分别给出折半查找的非递归和递归算法。

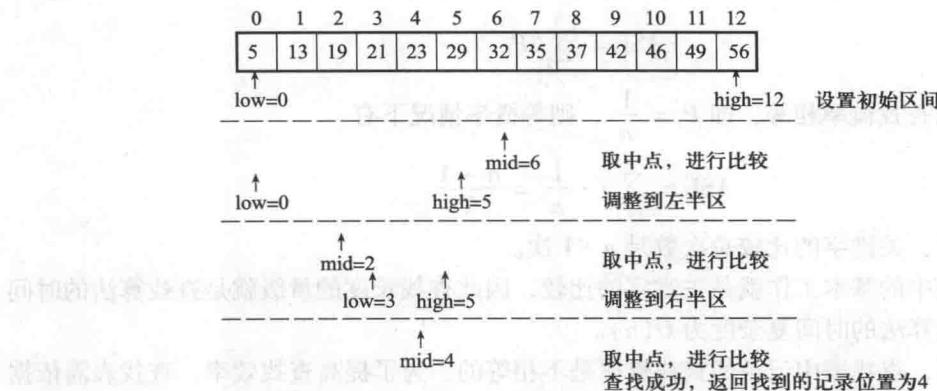


图 9-1 折半查找 23 成功的过程

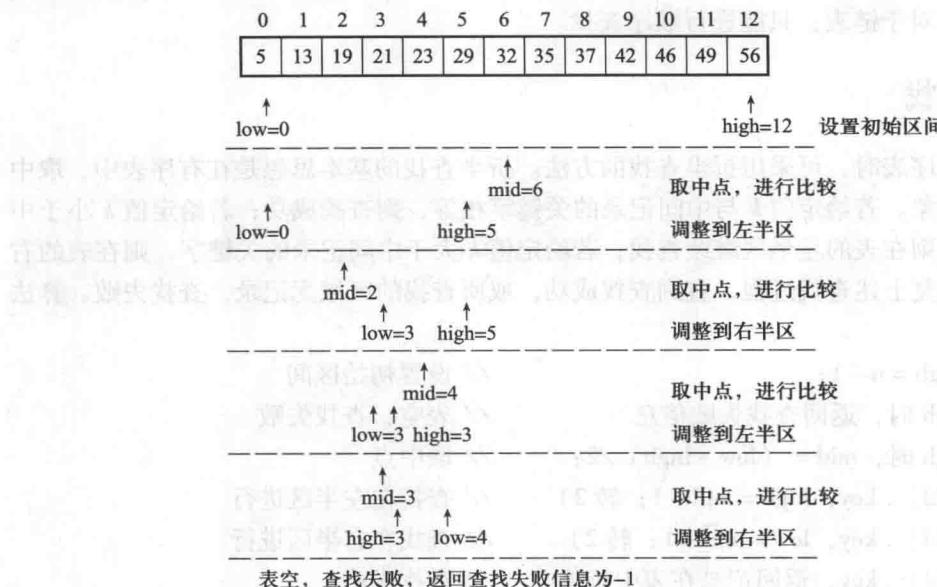


图 9-2 折半查找 22 失败的过程

### 算法 9-3 折半查找非递归算法

```
int BiSearch(Record r[], int n, int k)
//查找成功时返回该对象的下标序号, 失败时返回 -1
{
 low = 0;
 high = n - 1;
```

```

while(low <= high)
{
 mid = (low + high) / 2;
 if(r[mid].key == k)
 return mid;
 else if(r[mid].key < k)
 low = mid + 1;
 else
 high = mid - 1;
}
return -1;
}

```

#### 算法 9-4 折半查找递归算法

```

int BiSearch2(Record r[], int low, int high, int k)
{
 if(low > high)
 return -1;
 else
 {
 mid = (low + high) / 2;
 if(r[mid].key == k)
 return mid;
 else
 if(r[mid].key < k)
 return BiSearch2(r, mid + 1, high, k);
 else
 return BiSearch2(r, low, mid - 1, k);
 }
}

```

可以用一棵二叉树来描述折半查找的过程，称这个二叉树为判定树。图 9-3 所示的是例 9-1 折半查找过程对应的判定树。二叉树中结点内的数值表示有序表中记录的下标。

可以看出，在查找表中任一记录的过程中，从判定树根结点到该记录结点路径上各结点关键字的比较次数就是该记录结点在树中的层数。对于有  $n$  个结点的判定树，树高为  $h$ ，则有  $2^{h-1} < n \leq 2^h - 1$ ，即  $h - 1 < \log_2(n + 1) \leq h$ ，因此  $h = \lceil \log_2(n + 1) \rceil$ 。由此可知，折半查找在



图 9-3 折半查找过程对应的判定树

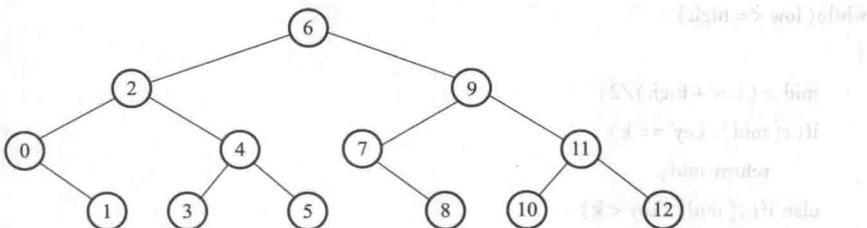


图 9-3 例 9-1 对应的判定树

查找成功时进行的关键字比较次数至多为  $\lceil \log_2(n+1) \rceil$ 。

接下来讨论折半查找的平均查找长度。从图 9-3 可知，长度为 13 的有序表进行折半查找的平均查找长度  $ASL = (1 \times 1 + 2 \times 2 + 3 \times 4 + 4 \times 6) / 13 = 41/13$ 。

一般情况下，以树高为  $h$  的满二叉树（有序表长度  $n=2^h-1$ ）为例。假设表中每个记录的查找是等概率的，即  $P_i = \frac{1}{n}$ ，则树的第  $i$  层有  $2^{i-1}$  个结点。因此，折半查找的平均查找长度

$$\begin{aligned} ASL &= \sum_{i=1}^h P_i C_i = \frac{1}{n} (1 \times 2^0 + 2 \times 2^1 + \dots + h \times 2^{h-1}) \\ &= \frac{n+1}{n} \log_2(n+1) - 1 \\ &\approx \log_2(n+1) - 1 \end{aligned}$$

由此可知，折半查找的时间效率为  $O(\log n)$ 。

### 9.2.3 分块查找

分块查找又称分块索引查找，是对顺序查找的一种改进。分块查找的性能介于顺序查找和折半查找之间，适用于对关键字分块有序的查找表进行查找操作。分块有序是指查找表可按关键字大小分成若干子表（或称块），且前一块中的最大关键字小于后一块中的最小关键字，但是各块内部的关键字不一定有序。分块查找需对子表建立索引表，查找表的每一个子表由索引表中的索引项确定。索引项包括关键字字段（存放对应子表中的最大关键字值）和指针字段（存放子表的起始序号）两个字段。

分块查找过程分两步进行：

- ① 确定要查找的记录所在的子表。用给定值  $k$  在索引表中查找索引项，以确定要查找的记录位于哪个子表中。
- ② 确定要查找的记录的情况。对第①步确定的子表进行顺序查找，以确定要查找的记录的情况。

**例 9-2** 关键字集合为  $\{10, 30, 8, 22, 38, 46, 61, 47, 65, 62, 80, 78\}$ , 共分为 3 块, 建立的查找表及其索引表如图 9-4 所示。

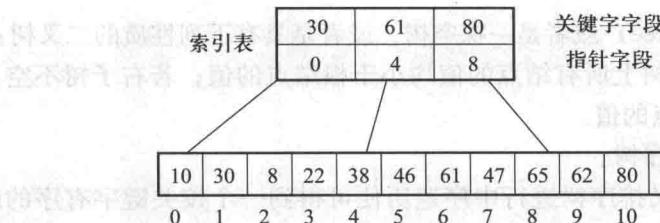


图 9-4 分块查找示例

分块查找由索引表查找和子表查找两步完成。例如, 当查找  $k=38$  时, 从图 9-4 可知, 若关键字等于 38 的记录存在, 因为 38 大于 30 而小于 61, 所以目标记录应当位于第 2 块中, 因此从索引表给出的第 2 块起始序号开始进行顺序查找即可。

设  $n$  个记录的查找表分为  $m$  个子表, 且每个子表均有  $t$  个元素, 则  $t = \frac{n}{m}$ 。这样, 分块查找的平均查找长度

$$ASL = ASL_{\text{索引表}} + ASL_{\text{子表}} = \frac{1}{2}(m+1) + \frac{1}{2}\left(\frac{n}{m}+1\right) = \frac{1}{2}\left(m + \frac{n}{m}\right) + 1$$

由此可见, 平均查找长度不仅和表的总长度  $n$  有关, 而且和子表个数  $m$  有关。

从上述讨论可以看出, 索引表是有序的。因此, 在检索索引表时也可采用折半查找的方法, 这样分块查找的平均查找长度

$$ASL = ASL_{\text{索引表}} + ASL_{\text{子表}} = \log_2(m+1)-1 + \frac{1}{2}\left(\frac{n}{m}+1\right) \approx \log_2(m+1) + \frac{n}{2m}$$

## 9.3 树表的查找

在某些应用软件中, 查找表不是一次性生成的, 而是在使用中逐渐产生的。例如, 在商品管理软件中需要建立一个商品信息表, 对每一批新进的商品都要首先查找是否存在该同类商品。若存在, 则只需增加该商品数量即可; 若不存在, 则需要在表中插入该类新商品。通常称这种在程序运行过程中可进行插入、删除等操作的查找表为动态查找表。

由于动态查找表需要随时进行插入、删除等操作, 因此顺序表已不适合作为它的存储结构, 而二叉排序树则是一个不错的选择。

### 9.3.1 二叉排序树

#### 1. 二叉排序树定义

二叉排序树 (Binary Sort Tree) 或者是一棵空树，或者是具有下列性质的二叉树：

① 若左子树不空，则左子树上所有结点的值均小于根结点的值；若右子树不空，则右子树上所有结点的值均大于根结点的值。

② 左右子树也都是二叉排序树。

由图 9-5 可以看出，对二叉排序树进行中序遍历便可得到一个按关键字有序的序列，因此一个无序序列可通过构造一棵二叉排序树而成为有序序列。

在讨论二叉排序树的操作算法之前，首先给出它的类定义。为了更清晰的描述二叉排序树各类操作算法，假设二叉排序树中结点的数据域只含一个整型数据。

```
struct BiNode
{
 int key;
 BiNode *lchild, *rchild;
};

class BiSortTree
{
public:
 BiNode *root;
 void Insert(BiNode *&ptr, int k); //供插入函数调用
 BiNode *Search(BiNode *ptr, int k); //供查找函数调用
 void Delete (BiNode *&ptr, int k); //供删除函数调用
 void Free(BiNode *ptr); //供析构函数调用

 BiSortTree(int a[], int n); //根据数组 a[n] 建立二叉排序树
 ~BiSortTree(); //析构函数
 void Insert(int k); //插入
 bool Search(int k); //查找
 void Delete (int k); //删除
};
```

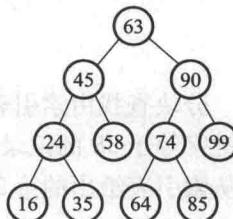


图 9-5 二叉排序树

#### 2. 二叉排序树的插入和建立

在一棵二叉排序树中插入值为  $k$  的结点的步骤如下：

① 若二叉排序树为空，则生成值为  $k$  的新结点  $s$ ，同时将新结点  $s$  作为根结点插入。

② 若  $k$  小于根结点的值，则在根的左子树中插入值为  $k$  的结点。

③ 若  $k$  大于根结点的值，则在根的右子树中插入值为  $k$  的结点。

④ 若  $k$  等于根结点的值，表明二叉排序树中已有此关键字，则无需插入。

从以上描述可知，插入过程是递归的。算法的具体描述如算法 9-5 所示。

### 算法 9-5 二叉排序树插入算法

```
void BiSortTree::Insert(BiNode * &ptr, int k)
//在以 ptr 为根的二叉排序树中插入值为 k 的结点
{
 if(ptr == NULL)
 {
 ptr = new BiNode;
 ptr -> key = k;
 ptr -> lchild = ptr -> rchild = NULL;
 }
 else
 {
 if(k < ptr -> key)
 Insert(ptr -> lchild, k); //插入到左子树
 else if(k > ptr -> key)
 Insert(ptr -> rchild, k); //插入到右子树
 }
}
void BiSortTree::Insert(int k) //插入
{
 Insert(root, k);
}
```

在函数 `Insert (int k)` 中调用私有成员函数 `Insert (BiNode * &ptr, int k)`，利用根指针 `root` 作为参数，完成在当前对象（整个二叉排序树）中插入值为  $k$  的结点。需要注意的是，私有成员函数中的第一个参数 `ptr` 必须是引用。

由于二叉排序树的构造实际上就是从一棵空二叉排序树开始逐个插入结点的过程，因此利用二叉排序树的插入算法可以容易地写出生成一棵具有  $n$  个结点的二叉排序树的算法。设生成二叉排序树的  $n$  个元素由数组提供，其具体实现如算法 9-6 所示。

### 算法 9-6 二叉排序树的建立

```
BiSortTree::BiSortTree(int a[], int n)
{
 root = NULL;
```

```
for (int i = 0; i < n; i++)
 Insert (root, a[i]);
```

在一般情况下，该算法的时间复杂度为  $O(n * \log_2 n)$ 。

**例 9-3** 假设关键字序列为 {61, 86, 70, 53, 67, 52, 88}，建立一棵二叉排序树。

解 建立一棵二叉排序树的过程如图 9-6 所示。

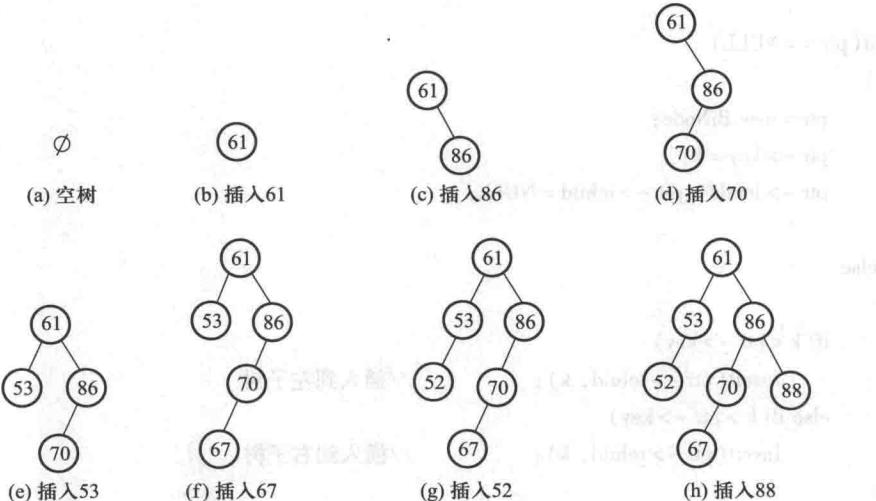


图 9-6 建立二叉排序树的过程

### 3. 二叉排序树的查找过程

根据二叉排序树的定义，在二叉排序树中查找给定值  $k$  的过程如下：

- ① 若二叉排序树为空，则表明查找失败，返回空指针；否则，若给定值  $k$  等于根结点的值，则表明查找成功，返回根结点。
- ② 若给定值  $k$  小于根结点的值，则继续在根的左子树中查找。
- ③ 若给定值  $k$  大于根结点的值，则继续在根的右子树中查找。

这是一个递归查找过程，其具体实现如算法 9-7 所示。

#### 算法 9-7 二叉排序树查找的递归算法

```
BiNode * BiSortTree::Search(BiNode * ptr, int k)
//在以 ptr 为根的二叉排序树中查找关键字为 k 的结点
{
 if(ptr == NULL)
 return NULL;
```

```

 else
 if (ptr -> key == k) //查找成功,返回
 return ptr;
 else if (k < ptr -> key)
 return Search (ptr -> lchild, k); //查找左子树
 else
 return Search (ptr -> rchild, k); //查找右子树
 }
}

bool BiSortTree::Search(int k) //查找
{
 return Search(root,k) != NULL;
}

```

在函数 Search ( int k) 中调用私有成员函数 Search ( BiNode \* ptr, int k)，利用根指针 root 作为参数，完成在当前对象（整个二叉排序树）中查找值为 k 的结点。

下面给出二叉排序树查找的非递归算法。

#### 算法 9-8 二叉排序树查找的非递归算法

```

BiNode * BiSortTree::Search2(BiNode * ptr, int k)
//在以 ptr 为根的二叉排序树中查找值为 K 的结点
{
 while(ptr)
 {
 if(k == ptr -> key)
 return ptr;
 else if(k < ptr -> key)
 ptr = ptr -> lchild;
 else
 ptr = ptr -> rchild;
 }
 return NULL;
}

bool BiSortTree::Search2(int k) //查找
{
 return Search2(root,k) != NULL;
}

```

在二叉排序树上进行查找的过程中，因为给定值 k 同结点比较的次数最少为一次（即树根结点就是待查的结点），最多为树的深度，所以平均查找次数要小于等于树的深度。



若二叉排序树是平衡的（即形态均匀），则进行查找的时间复杂度为  $O(\log_2 n)$ ；若退化为一棵单支树（最极端和最差的情况），则其时间复杂度为  $O(n)$ 。对于一般情况，其时间复杂度可以认为是  $O(\log_2 n)$ 。

#### 4. 二叉排序树的删除操作

二叉排序树的删除比插入要复杂。由于被插入的结点都是被链接到树中的叶子结点上，因此不会破坏树的结构。删除结点则不同，因为可能删除的可能是叶子结点，也可能是分支结点。如果删除的是分支结点，那么就破坏了原有结点之间的链接关系。此时，需要重新修改指针，使得删除结点后的树仍为一棵二叉排序树。

下面分3种情况说明删除结点的操作。

##### (1) 删除叶子结点

删除叶子结点不会影响到其他结点间的关系，只要将被删结点的双亲结点的相应指针置为空并删除该结点即可。

如图9-7所示，当删除关键字为10的叶子结点时，只需要将关键字为24的结点的左指针域置空即可；当删除关键字为78的叶子结点时，只需要将关键字为66的结点的右指针域置空即可。

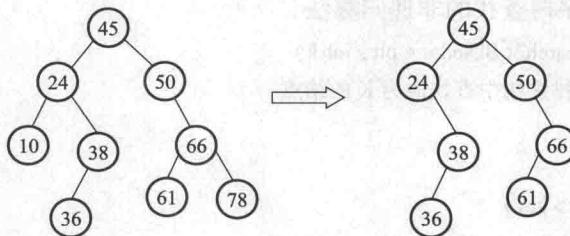


图9-7 删除叶子结点

##### (2) 删除单支结点

因为被删除的结点只有左子树或右子树，即它的孩子只有一个，要么是左孩子，要么是右孩子，所以，这种删除操作也比较简单。当删除该结点时，只要将该结点的孩子变为被删除结点双亲的左孩子或右孩子即可。

如图9-8所示，当删除关键字为38的单支结点时，只要将该结点的左孩子（关键字为36的结点）变成关键字为24的结点的右孩子即可；当删除关键字为50的单支结点时，只要将该结点的右孩子（关键字为66的结点）变成关键字为45的结点的右孩子即可。

##### (3) 删除双支结点

因为待删除的结点有两个后继指针需要妥善处理，所以，这种删除比较复杂。

为了使在执行删除操作时，二叉树的结构不发生巨大的变化，同时保持二叉树的特点，可采用如下方法：首先用被删除结点左子树中值最大的结点替代被删除的结点，然后从左子树中

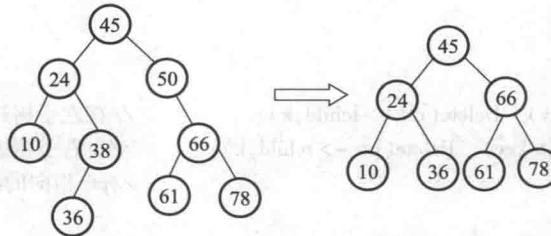


图 9-8 删除单支结点

删除这个值最大的结点；也可先用被删除结点右子树中值最小的结点替代被删除的结点，然后从右子树中删除这个值最小的结点。

如图 9-9 所示，若要删除根结点，则因为它是双支结点，所以首先用它左子树中值最大的结点，即关键字为 38 的结点来替换它，然后再将关键字为 38 的结点删除。此时由于关键字为 38 的结点是单支结点，因此可用方法（2）进行删除。

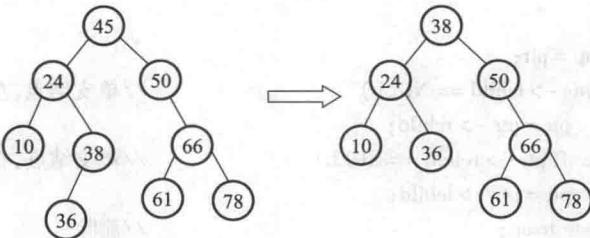


图 9-9 删除双支结点

在二叉排序树中删除值为  $k$  的结点的算法可以是递归的，也可以是非递归的。下面给出递归算法，读者可自行给出非递归算法。

递归算法的步骤如下：

- ① 若二叉排序树为空，则表明不存在删除的结点，不进行删除操作。
- ② 若给定值  $k$  小于根结点的值，则继续在根的左子树中删除。
- ③ 若给定值  $k$  大于根结点的值，则继续在根的右子树中删除。
- ④ 若给定值  $k$  等于根结点的值，则根结点即为要删除的结点，此时需要根据上述分析的叶子结点、单支结点和双支结点 3 种结点情况执行相应的删除操作。

算法的具体实现如算法 9-9 所示。

#### 算法 9-9 二叉排序树的删除操作递归算法

```
void BiSortTree::Delete(BiNode * &ptr, int k)
```

```
// 在以 ptr 为根的二叉排序树中删除值为 k 的结点
```

```

{
 if(ptr!=NULL)
 {
 if(k < ptr -> key) Delete(ptr -> lchild,k); //在左子树进行删除
 else if(k > ptr -> key) Delete(ptr -> rchild,k); //在右子树进行删除
 else //ptr 指向的结点就是要删除的结点
 {
 if(ptr -> lchild != NULL&&ptr -> rchild != NULL) //双支结点
 {
 temp = ptr -> lchild;
 while(temp -> rchild != NULL) //寻找左子树中具有最大值的结点
 temp = temp -> rchild;
 ptr -> key = temp -> key;
 Delete(ptr -> lchild,temp -> key);
 }
 else
 {
 temp = ptr;
 if(ptr -> lchild == NULL) //单支结点,左子树为空
 ptr = ptr -> rchild;
 else if(ptr -> rchild == NULL) //单支结点,右子树为空
 ptr = ptr -> lchild;
 delete temp; //删除
 }
 }
 }
}

void BiSortTree::Delete(int k) //删除
{
 Delete(root,k);
}

```

在函数 `Delete ( int k )` 中调用私有成员函数 `Delete ( BiNode * &ptr, int k )`，利用根指针 `root` 作为参数，完成在当前对象（整个二叉排序树）中删除值为 `k` 的结点。需要注意的是，私有成员函数中的第一个参数 `ptr` 必须是引用。

二叉排序树的析构函数与二叉树的析构函数相同，请读者自行完成。

### 9.3.2 平衡二叉树

由 9.3.1 节的讨论可以看出，二叉排序树的效率取决于二叉排序树的形态。为了获得较好

的查找效率，就要构造一棵形态均匀的二叉排序树，即平衡二叉树。

平衡二叉树或者是一棵空树，或者是具有下列性质的二叉排序树：它的左子树和右子树都是平衡二叉树，且左子树和右子树高度之差的绝对值不超过1，如图9-10所示。

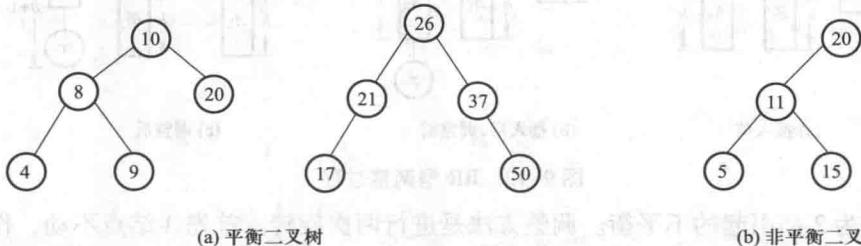


图9-10 平衡二叉树和非平衡二叉树

左子树与右子树高度之差称为结点的平衡因子。由平衡二叉树定义可知，平衡二叉树所有结点的平衡因子只能取-1、0、1这三个值之一。

为了使建立的二叉排序树是平衡的，要求当新结点插入二叉排序树时必须保持所有结点的平衡因子满足平衡二叉树的要求。如果不满足要求，那么就必须进行调整。

设结点A为最小不平衡子树的根结点，对该子树进行调整归纳起来有下面4种情况。

#### (1) LL型

如图9-11所示，这是由于在结点A的左孩子B的左子树上插入结点x，使得A结点的平衡因子由1变为2而引起的不平衡。调整方法是进行一次向右的顺时针旋转。

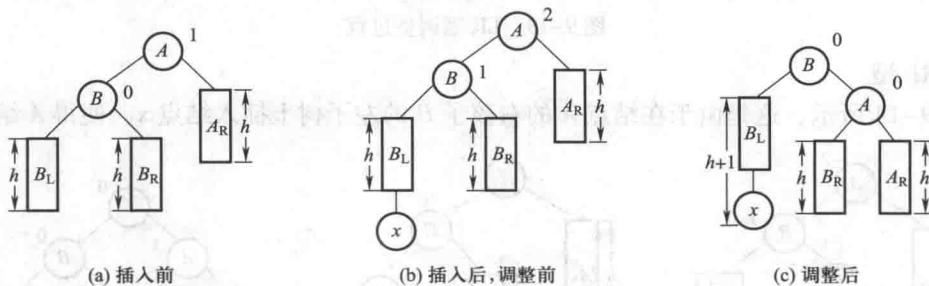


图9-11 LL型调整过程

#### (2) RR型

如图9-12所示，这是由于在结点A的右孩子B的右子树上插入结点x，使得A结点的平衡因子由-1变为-2而引起的不平衡。调整方法是进行一次向左的逆时针旋转。

#### (3) LR型

如图9-13所示，这是由于在结点A的左孩子B的右子树上插入结点x，使得A结点的平

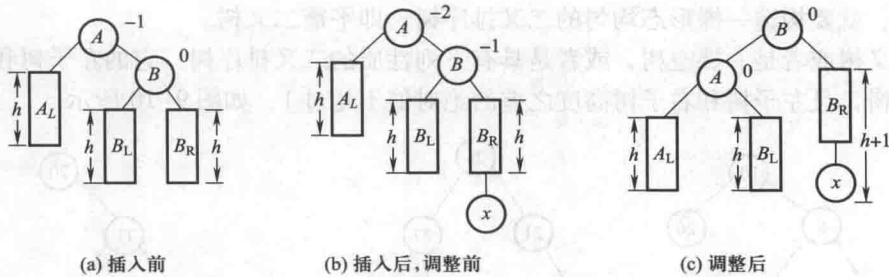


图 9-12 RR 型调整过程

平衡因子由 1 变为 2 而引起的不平衡。调整方法是进行两次旋转。首先 A 结点不动，作一次向左的逆时针旋转，将支撑点由结点 B 调整到结点 C 处；然后再进行一次向右的顺时针旋转，将支撑点由结点 A 调整到结点 C 处。

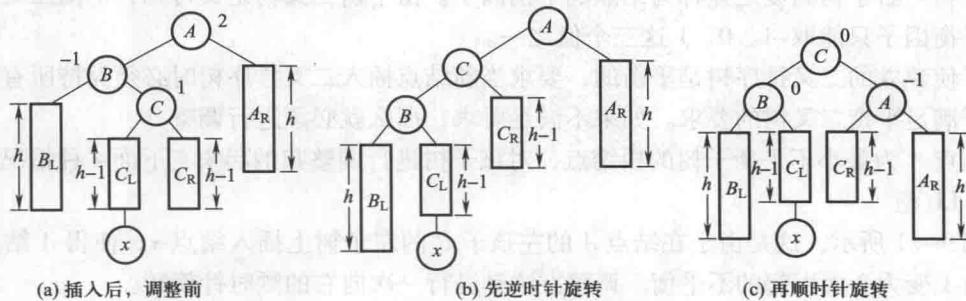


图 9-13 LR 型调整过程

#### (4) RL型

如图 9-14 所示，这是由于在结点 A 的右孩子 B 的左子树上插入结点 x，使得 A 结点的平

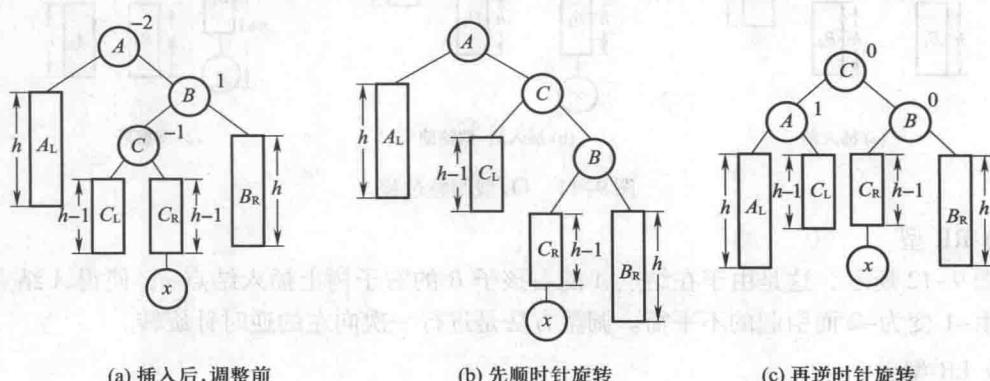


图 9-14 RL 型调整过程

衡因子由-1 变为-2 而引起的不平衡。调整方法是进行两次旋转。首先 A 结点不动，作一次向右的顺时针旋转，将支撑点由结点 B 调整到结点 C 处；然后再进行一次向左的逆时针旋转，将支撑点由结点 A 调整到结点 C 处。

**例 9-4** 设一组关键字序列为{4, 5, 7, 2, 1, 3, 6}，试建立一棵平衡二叉树。

解 平衡二叉树生成步骤如图 9-15 所示。

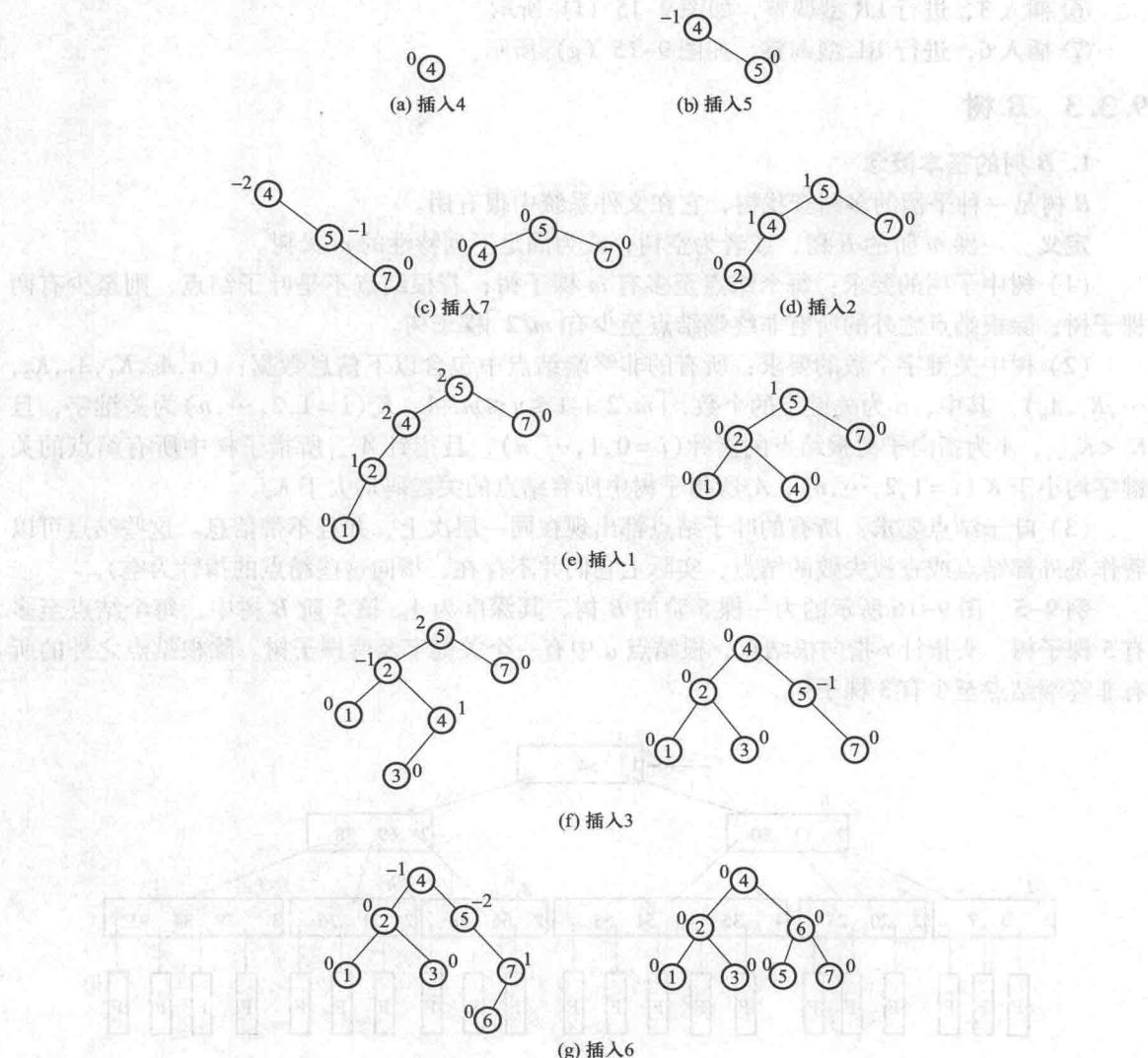


图 9-15 平衡二叉树的生成过程示例

- ① 插入 4，不需调整，如图 9-15 (a) 所示。
- ② 插入 5，不需调整，如图 9-15 (b) 所示。
- ③ 插入 7，进行 RR 型调整，如图 9-15 (c) 所示。
- ④ 插入 2，不需调整，如图 9-15 (d) 所示。
- ⑤ 插入 1，进行 LL 型调整，如图 9-15 (e) 所示。
- ⑥ 插入 3，进行 LR 型调整，如图 9-15 (f) 所示。
- ⑦ 插入 6，进行 RL 型调整，如图 9-15 (g) 所示。

### 9.3.3 B 树

#### 1. B 树的基本概念

B 树是一种平衡的多路查找树，它在文件系统中很有用。

**定义** 一棵  $m$  阶的 B 树，或者为空树，或为满足下列特性的  $m$  叉树。

(1) 树中子树的要求：每个结点至多有  $m$  棵子树；若根结点不是叶子结点，则至少有两棵子树；除根结点之外的所有非终端结点至少有  $\lceil m/2 \rceil$  棵子树。

(2) 树中关键字个数的要求：所有的非终端结点中包含以下信息数据： $(n, A_0, K_1, A_1, K_2, \dots, K_n, A_n)$ 。其中， $n$  为关键字的个数， $\lceil m/2 \rceil - 1 \leq n \leq m - 1$ ， $K_i$  ( $i = 1, 2, \dots, n$ ) 为关键字，且  $K_i < K_{i+1}$ ， $A_i$  为指向子树根结点的指针 ( $i = 0, 1, \dots, n$ )，且指针  $A_{i-1}$  所指子树中所有结点的关键字均小于  $K_i$  ( $i = 1, 2, \dots, n$ )， $A_n$  所指子树中所有结点的关键码均大于  $K_n$ 。

(3) 叶子结点要求：所有的叶子结点都出现在同一层次上，并且不带信息。这些结点可以看作是外部结点或查找失败的结点，实际上它们并不存在，指向这些结点的指针为空。

**例 9-5** 图 9-16 所示的为一棵 5 阶的 B 树，其深度为 4。该 5 阶 B 树中，每个结点至多有 5 棵子树。头指针  $r$  指向根结点，根结点  $a$  中有一个关键字及两棵子树。除根结点之外的所有非终端结点至少有 3 棵子树。

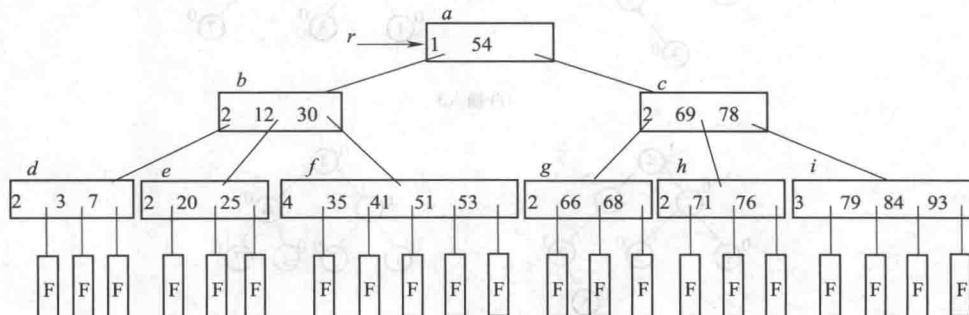


图 9-16 一棵 5 阶的 B 树示例

**例 9-6** 在一棵 7 阶的 B 树中，树根结点的关键字个数最少为 1，最多为  $m - 1 = 6$ ，子树个数最少为 2，最多为  $m = 7$ ；每个非树根结点的关键字个数最少为  $\lceil m/2 \rceil - 1 = \lceil 7/2 \rceil - 1 = 3$ ，最多为  $m - 1 = 6$ ，子树个数最少为  $\lceil m/2 \rceil = \lceil 7/2 \rceil = 4$ ，最多为  $m = 7$ 。

同二叉排序树一样，关键字的插入次序不同将可能生成不同结构的 B 树。

## 2. B 树的查找

B 树的查找类似二叉排序树的查找，不同之处在于 B 树每个结点上是多关键字的有序表。在到达某个结点时，先在有序表中查找。若找到，则查找成功；否则，再到对应的指针信息指向的子树中去查找。当到达叶子结点时，则说明树中没有对应的关键字，查找失败。即 B 树上的查找过程是一个顺指针查找结点和在结点中查找关键字交叉进行的过程。

例如，在图 9-16 所示的 B 树中查找关键字值为 93 的元素。首先，从  $r$  指向的根结点  $a$  开始，结点  $a$  中只有一个关键字且 93 大于它。因此，按  $a$  结点指针域  $A_1$  到结点  $c$  去查找。结点  $c$  有两个关键字，而 93 也都大于它们，应按  $c$  结点指针域  $A_2$  到结点  $i$  去查找。在结点  $i$  中顺序比较关键字，找到关键字  $K_3$ 。

B 树的查找是由两个基本操作交叉进行的过程，即：

- ① 在 B 树上找结点；
- ② 在结点中找关键字。

由于 B 树通常是存储在外存上的，因此操作①就是通过指针在磁盘相对定位，将结点信息读入内存，之后再对结点中的关键字有序表进行顺序查找或折半查找。因为在磁盘上读取结点信息比在内存中进行关键字查找耗时多，所以在磁盘上读取结点信息的次数，即 B 树的层数次是决定 B 树查找效率的首要因素。

那么，对含有  $n$  个关键字的  $m$  阶 B 树，最坏情况下达到多深呢？可按二叉平衡树进行类似分析。首先，讨论  $m$  阶 B 树各层上的最少结点数。

由 B 树定义可知，第一层至少有 1 个结点，第二层至少有 2 个结点。由于除根结点外的每个非终端结点至少有  $\lceil m/2 \rceil$  棵子树，因此第三层至少有  $2(\lceil m/2 \rceil)$  个结点。依此类推，第  $k + 1$  层至少有  $2(\lceil m/2 \rceil)^{k-1}$  个结点，这些结点均为叶子结点。若  $m$  阶 B 树有  $n$  个关键字，则叶子结点即查找不到的结点为  $n + 1$ ，由此有：

$$n + 1 \geq 2 \times (\lceil m/2 \rceil)^{k-1}$$

即

$$k \leq \log_{\lceil m/2 \rceil} \left( \frac{n+1}{2} \right) + 1$$

也就是说，在含有  $n$  个关键字的 B 树上进行查找时，从根结点到关键字所在结点的路径上涉及的结点数不超过

$$\log_{\lceil m/2 \rceil} \left( \frac{n+1}{2} \right) + 1$$

### 3. B树的插入和删除操作

#### (1) 插入操作

在B树上插入关键字与在二叉排序树上插入结点不同，关键字的插入不是在叶结点上进行，而是在最低层的某个非终端结点中添加一个关键字。若该结点上关键字个数不超过 $m-1$ 个，则可直接插入到该结点上；否则，该结点上关键字个数至少达到 $m$ 个，因此使该结点的子树超过了 $m$ 棵。因为这与B树定义不符，所以要调整，即进行结点的分裂。分裂的方法是在关键字加入结点后，将结点中的关键字分成3部分，使得前后两部分关键字个数均大于等于 $(\lceil m/2 \rceil - 1)$ ，而中间部分只有一个结点。前后两部分成为两个结点，中间的一个结点插入到父结点中。若插入父结点而使父结点中关键字个数超过 $m-1$ ，则父结点继续分裂。直到插入某个父结点后，其关键字个数小于 $m$ 。由此可见，B树是从底向上生长的。

**例9-7** 如图9-17所示，假定图9-17(a)是一个3阶B树的简图，图中略去了结点中的关键字个数、查找失败等信息，在此树上依次插入关键字65、24、50和38，其变化过程如图9-17(b)至图9-17(g)所示。

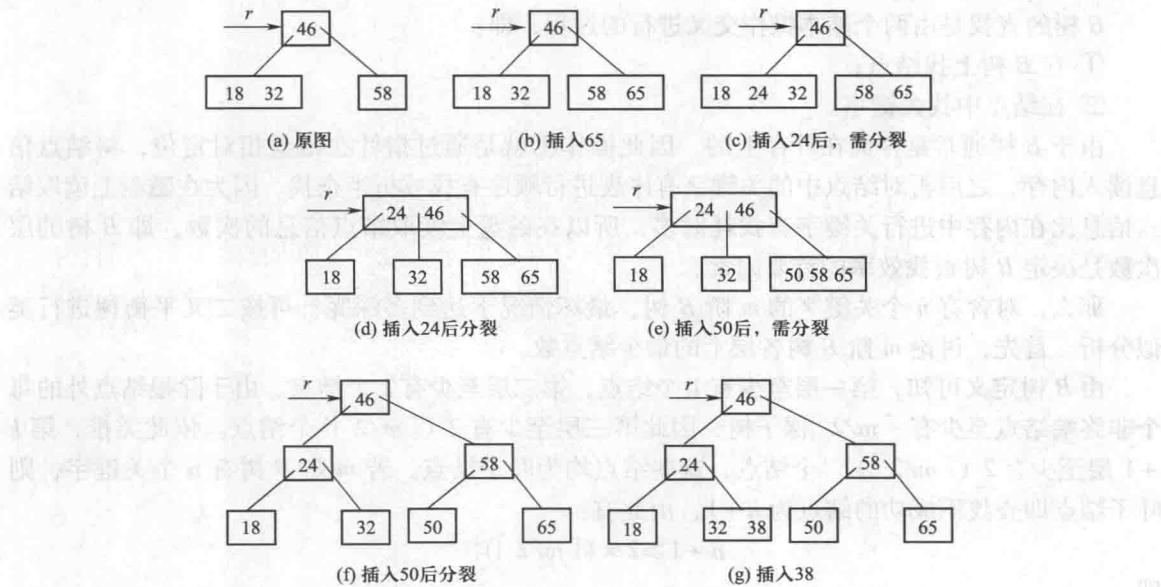


图9-17 B树的插入

在3阶B树中，每个结点的关键字个数最少为1，最多为2。当插入后关键字的个数为3时，结点就必须分裂成两个结点。分裂的方法是让原有结点只保留第1个关键字和它前后的两个指针，让新结点保存原有结点中的最后一个（即第3个）关键字和它前后的两个指针，让原有结点的第2个关键字和指向新结点的指针作为新结点的索引项插入到原有结点的前驱结点中。若没有前驱结点，则就生成一个新的根结点，并将原树根结点和分裂出的结点作为它的两棵子树。

**例 9-8** 有关关键字序列 {20, 54, 69, 84, 71, 30, 78, 25, 93, 41, 7, 76, 51, 66, 68, 53, 3, 79, 35, 12, 15, 65}，建立 5 阶 B 树。

解 建立 B 树的过程如图 9-18 所示。

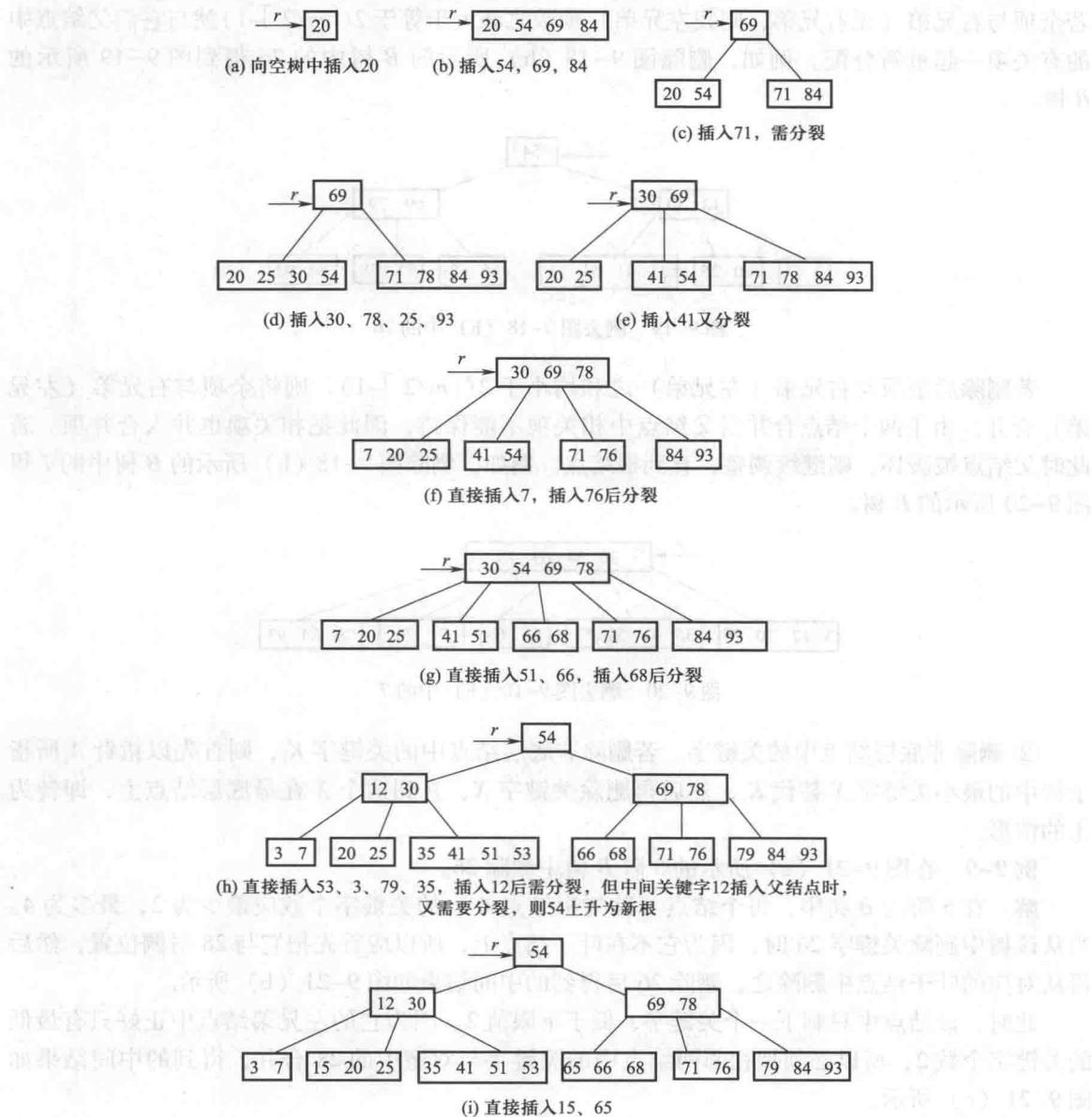


图 9-18 建立 B 树的过程

## (2) 删除操作

分两种情况讨论删除操作。

① 删除最底层结点中的关键字。若结点中关键字个数大于 $\lceil m/2 \rceil - 1$ , 直接删去。否则, 若余项与右兄弟 (无右兄弟, 则找左兄弟) 项数之和大于等于 $2(\lceil m/2 \rceil - 1)$  就与它们父结点中的有关项一起重新分配。例如, 删除图 9-18 (h) 所示的 B 树中的 76 得到图 9-19 所示的 B 树。

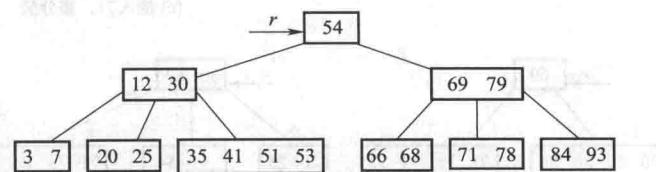


图 9-19 删去图 9-18 (h) 中的 76

若删除后余项与右兄弟 (左兄弟) 之和均小于 $2(\lceil m/2 \rceil - 1)$ , 则将余项与右兄弟 (左兄弟) 合并。由于两个结点合并后父结点中相关项不能保持, 因此把相关项也并入合并项。若此时父结点被破坏, 则继续调整, 直到根结点。例如, 删除图 9-18 (h) 所示的 B 树中的 7 得图 9-20 所示的 B 树。

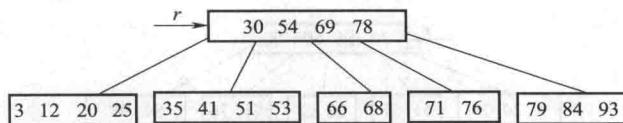


图 9-20 删去图 9-18 (h) 中的 7

② 删除非底层结点中的关键字。若删除非底层结点中的关键字  $K_i$ , 则首先以指针  $A_i$  所指子树中的最小关键字  $X$  替代  $K_i$ , 然后再删除关键字  $X$ , 直到这个  $X$  在最底层结点上, 即转为①的情形。

**例 9-9** 在图 9-21 (a) 所示的 5 阶 B 树中删除 26。

解 在 5 阶的 B 树中, 每个结点 (除树根结点外) 的关键字个数应最少为 2, 最多为 4。当从该树中删除关键字 26 时, 因为它不在叶子结点上, 所以应首先把它与 28 对调位置, 然后再从对应的叶子结点中删除之, 删除 26 后得到的中间结果如图 9-21 (b) 所示。

此时, 该结点中只剩下一个关键字, 低于下限值 2。因为它的左兄弟结点中正好只有最低的关键字个数 2, 所以必须把相邻两结点中的关键字与双亲中的 28 合并, 得到的中间结果如图 9-21 (c) 所示。

因为结点 {12} 只剩下一个关键字, 且它的右兄弟 (没有左兄弟) 结点中只含有两个关键

字；所以必须继续合并，即把结点{12}中的一个关键字和其右兄弟以及双亲合并到一个结点中，使新结点成为新的树根结点，最后得到的结果如图 9-21 (d) 所示，整个 B 树减少了一层。

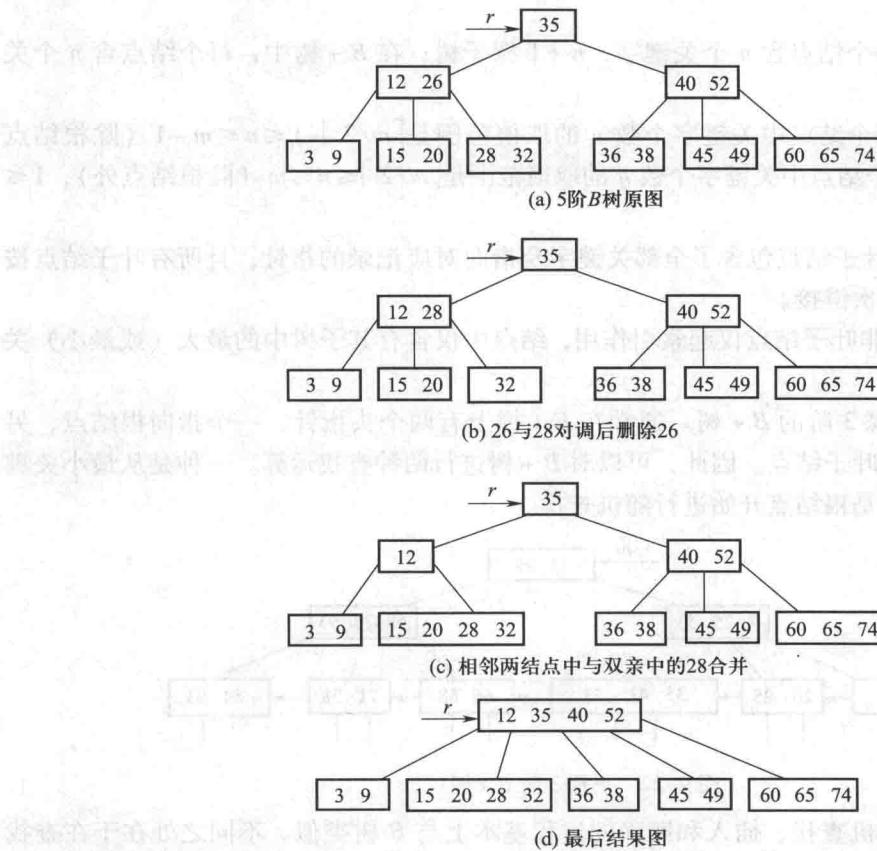


图 9-21 5 阶 B 树中删除 26 的过程

B 树的插入和删除算法比较复杂。在插入时，首先要为待插入的关键字  $K$  查找出插入位置，它必定在某个叶子结点上，假定为  $a$  结点的第  $i$  个位置。然后按照顺序表的插入方法把索引项  $(K, \text{NULL}, \text{recp})$  插入到  $a$  结点的第  $i$  个位置上（ $\text{recp}$  表示对应记录的存储位置）。接下来，再进行插入后的循环处理，直到不需要分裂结点。在删除时，首先要查找出待删除的关键字  $K$  所在的位置。若它不在叶子结点上，则把它同其中序前驱或后继关键字对调位置。然后按照顺序表的删除方法从对应的叶子结点中删除其关键字  $k$  所在的索引项。接下来再进行删除后的循环处理，直到不需要合并结点。

### 9.3.4 B+树

$B+$ 树是应文件系统所需而产生的一种 $B$ 树的变形树。 $m$ 阶的 $B+$ 树和 $m$ 阶的 $B$ 树的差异在于以下4个方面：

(1) 在 $B$ 树中，每个结点含 $n$ 个关键字， $n+1$ 棵子树；在 $B+$ 树中，每个结点含 $n$ 个关键字， $n$ 棵子树。

(2) 在 $B$ 树中，每个结点中关键字个数 $n$ 的取值范围是 $\lceil m/2 \rceil - 1 \leq n \leq m - 1$ （除根结点外）；在 $B+$ 树中，每个结点中关键字个数 $n$ 的取值范围是 $\lceil m/2 \rceil \leq n \leq m$ （除根结点外）， $1 \leq n \leq m$ （根结点）。

(3)  $B+$ 树中所有叶子结点包含了全部关键字及指向对应记录的指针，且所有叶子结点按关键字由小到大顺序依次链接。

(4)  $B+$ 树中所有非叶子结点仅起索引作用，结点中仅含有其子树中的最大（或最小）关键字。

图9-22所示为一棵3阶的 $B+$ 树。通常在 $B+$ 树上有两个头指针，一个指向根结点，另一个指向关键字最小的叶子结点。因此，可以对 $B+$ 树进行两种查找运算。一种是从最小关键字起顺序查找，另一种是根结点开始进行随机查找。

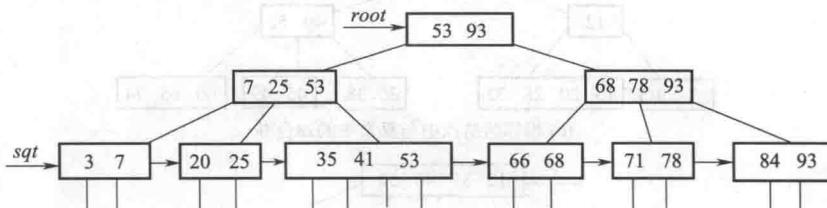


图9-22 一棵3阶 $B+$ 树

在 $B+$ 树上进行随机查找、插入和删除的过程基本上与 $B$ 树类似。不同之处在于在查找时，若非终端结点上的关键字等于给定值，则查找并不终止，而是继续向下直到叶子结点。因此，对于 $B+$ 树，不论查找成功与否，每次查找都是走了一条从根到叶子结点的路径。 $B+$ 树查找的分析类似于 $B$ 树。 $B+$ 树的插入仅在叶子结点上进行。当结点中的关键字个数大于 $m$ 时要分裂成两个结点，它们所含关键字的个数均为 $\lfloor (m+1)/2 \rfloor$ 。而且，它们的双亲结点中应同时包含这两个结点中的最大关键字。 $B+$ 树的删除也仅在叶子结点进行。当叶子结点中的最大关键字被删除时，其在非终端结点中的值可以作为一个分界关键字存在。若因删除而使结点中关键字的个数少于 $\lceil m/2 \rceil$ 时，则其和兄弟结点的合并过程亦和 $B$ 树类似。

**例9-10** 已知一棵3阶 $B+$ 树如图9-23(a)所示，现依次插入关键字16、17、19，插入过程的简图如图9-23(b)和图9-23(c)所示。

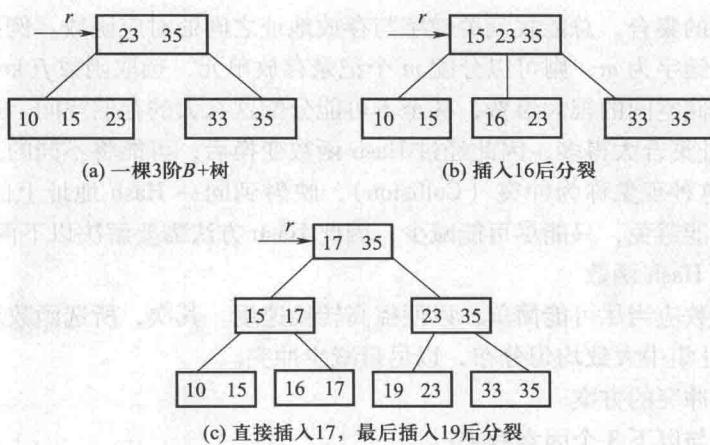


图 9-23 B+树的插入

## 9.4 Hash 查找

### 9.4.1 Hash 查找的基本概念

在上面讨论的查找方法中,由于记录的存储位置与关键字之间不存在确定的关系,因此查找时需要进行一系列对关键字的查找比较。即查找算法是建立在比较的基础上的,查找效率由比较一次缩小的查找范围决定。理想的情况是依据关键字直接得到其对应的记录位置,这要求关键字与记录位置间存在一一对应关系,通过这个关系能很快地由关键字得到对应的记录位置。

**例 9-11** 假设一组记录的关键字分别为{18, 27, 1, 20, 22, 6, 10, 13, 41, 15, 25},选取关键字与记录位置间的函数为 $f(\text{key}) = \text{key \% } 11$ ,选出

① 通过这个函数对 11 个元素可建立如下查找表:

| 位置   | 0  | 1 | 2  | 3  | 4  | 5  | 6 | 7  | 8  | 9  | 10 |
|------|----|---|----|----|----|----|---|----|----|----|----|
| 关键字值 | 22 | 1 | 13 | 25 | 15 | 27 | 6 | 18 | 41 | 20 | 10 |

② 查找时,对给定值  $k$ ,仍然先通过这个函数计算  $k$  在查找表中的位置,再将  $k$  与该位置中元素的关键字比较若相等,则查找成功。

在上述例子中,存储记录的查找表被称为 Hash 表(也称哈希表或散列表),函数  $f(\text{key})$  称为 Hash 函数,根据 Hash 函数计算出的存储位置称为 Hash 地址,基于 Hash 表的查找称为 Hash 查找。

对于  $n$  个记录的集合，总能找到关键字与存放地址之间的对应函数。例如，关键字类型为整型时，若最大关键字为  $m$ ，则可以分配  $m$  个记录存放单元，选取函数  $f(\text{key}) = \text{key}$  即可。但是，这样会造成存储空间的很大浪费，甚至不可能分配这么大的存储空间。通常，由于关键字的集合比 Hash 地址集合大得多，因此经过 Hash 函数变换后，可能将不同的关键字映射到同一个 Hash 地址上。这种现象称为冲突（Collision），映射到同一 Hash 地址上的关键字称为同义词。由于冲突不可能避免，只能尽可能减少；因此 Hash 方法需要解决以下两个问题：

### (1) 构造好的 Hash 函数

首先，所选函数应当尽可能简单，以便提高转换速度。其次，所选函数对关键字计算出的地址应在 Hash 地址集中大致均匀分布，以尽量减少冲突。

### (2) 制定解决冲突的方案

产生冲突主要与以下 3 个因素有关：

① Hash 函数。若 Hash 函数选择得当，则可使 Hash 地址尽可能均匀地分布在 Hash 地址空间上，从而减少冲突的发生。否则，若 Hash 函数选择不当，则可能使 Hash 地址集中于某些区域，从而加大冲突的发生。

② 处理冲突的方法。由于选择适当的 Hash 函数可以减少冲突，但不能避免冲突；因此当冲突发生时，必须有较好的处理冲突的方法。

③ Hash 表的装填因子。Hash 表的装填因子定义为

$$\alpha = \frac{\text{填入表中的记录个数}}{\text{Hash 表的长度}}$$

Hash 表的装填因子是 Hash 表装满程度的标志。因为 Hash 表的表长是定值，所以  $\alpha$  与填入表中的记录个数成正比。 $\alpha$  越大，填入表中的记录较多，产生冲突的可能性就越大；反之， $\alpha$  越小，填入表中的记录较少，产生冲突的可能性就越小。通常，最终的  $\alpha$  的取值应控制在 0.6 ~ 0.9 的范围内。

## 9.4.2 Hash 表的构造

Hash 表是根据设定的 Hash 函数和处理冲突的方法，为一组记录所建立的一种存储结构。下面分别介绍 Hash 函数的构造方法和处理冲突的方法。

### 1. 常用的 Hash 函数

#### (1) 直接定址法

直接定址法是取关键字的某个线性函数值为 Hash 地址。直接定址法的 Hash 函数为

$$\text{Hash}(\text{key}) = a \times \text{key} + b \quad (a, b \text{ 为常数})$$

直接定址法的优点是 Hash 函数计算简单，且不可能有冲突发生。当关键字的集合不是很大且分布基本连续时，可用直接定址法的 Hash 函数。直接定址法的缺点是若关键字分布不连

续，则会造成内存单元的大量浪费。

#### (2) 除留余数法

除留余数法是取关键字除以  $p$  的余数作为 Hash 地址，它的 Hash 函数为

$$\text{Hash}(\text{key}) = \text{key} \% p \quad (p \text{ 是一个整数})$$

除留余数法的优点是计算比较简单，适用范围广，是经常使用的一种 Hash 函数。

这种方法的关键是选择适当的  $p$ 。若  $p$  选得不好则容易产生冲突。例如，若  $p$  取偶数时，则偶数的关键字将映射到 Hash 表的偶数地址，奇数的关键字将映射到 Hash 表的奇数地址，从而增加了冲突的可能；若  $p$  含有质因子即  $p = mn$ ，则所有含有  $m$  或  $n$  因子的关键字的 Hash 地址均为  $m$  或  $n$  的倍数，这也会增加冲突的可能性。

一般情况下，若 Hash 表长度为  $m$ ，则通常  $p$  取小于等于  $m$  的最大素数。

#### (3) 数字分析法

数字分析法是指对关键字中每一位的取值分布情况作出分析，取关键字中某些取值较均匀的数位作为 Hash 地址。该方法适合于所有关键字值已知的情况。

例如，要构造一个记录个数  $n = 80$ ，Hash 表长度  $m = 100$  的 Hash 表。下面给出其中 8 个关键字进行分析：

|                  |                  |                  |                  |
|------------------|------------------|------------------|------------------|
| $K_1 = 61317602$ | $K_2 = 61326875$ | $K_3 = 62739628$ | $K_4 = 61343634$ |
| $K_5 = 62706816$ | $K_6 = 62774638$ | $K_7 = 61381262$ | $K_8 = 61394220$ |

经过分析可知，关键字从左到右的第 1、2、3、6 位取值较集中，不宜作为 Hash 地址；剩余的第 4、5、7、8 位取值较均匀，可选取其中的两位作为 Hash 地址。设选取最后两位作为 Hash 地址，则这 8 位关键字的 Hash 地址集合为 {02, 75, 28, 34, 16, 38, 62, 20}。

#### (4) 平方取中法

平方取中法是指对关键字平方后，按 Hash 表大小取中间的若干位作为 Hash 地址的方法。例如，若 Hash 表长度为 1000，则可取关键字平方值的中间 3 位，如表 9-2 所示。

表 9-2 平方取中法 Hash 函数

| 关 键 字 | 关键字的平方    | Hash 地址 |
|-------|-----------|---------|
| 1234  | 15 22756  | 227     |
| 2143  | 45 92449  | 924     |
| 4132  | 170 73424 | 734     |
| 3214  | 103 29796 | 297     |

平方取中法通常用在不知道关键字分布且关键字的位数不是很大的情况下。

#### (5) 折叠法

折叠法首先将关键字自右到左分成位数相等的几部分，最后一部分位数可以短些；然后将

这几部分叠加求和，并按 Hash 表表长取后几位作为 Hash 地址。

叠加方法有移位法和间界叠加法两种。移位法将各部分的最后一一位对齐相加；间界叠加法从一端向另一端沿各部分分界来回折叠后，最后一位对齐相加。

**例 9-12** 关键字为  $key = 05587463253$ ，设 Hash 表长为 3 位数，则可对关键字每 3 位一部分来分割。

解 关键字分割为如下 4 组：

253 463 587 05

用叠加法计算出 Hash 地址如图 9-24 所示。

对于位数很多的关键字，当每一位上符号分布较均匀时，可采用此方法求得 Hash 地址。

## 2. 处理冲突的方法

由于选择适当的 Hash 函数可以减少冲突，但不能避免冲突；因此当冲突发生时，必须有较好的处理冲突的方法。处理冲突就是在冲突发生时，为关键字为  $key$  的记录安排另一个空的存储位置。常用的处理冲突的方法有开放地址法和链地址法。

### (1) 开放定址法

开放定址法是指当由关键字  $key$  得到的 Hash 地址一旦产生了冲突，就去寻找一个空的 Hash 地址。只要 Hash 表足够大，空的 Hash 地址总能找到并将记录存入。

找空 Hash 地址方法很多，下面介绍其中的 3 种。

① 线性探测法。指当发生冲突时，从冲突发生位置的下一个位置起，依次寻找空的 Hash 地址，即

$$H_i = (\text{Hash}(key) + d_i) \% m \quad (1 \leq i < m)$$

其中， $\text{Hash}(key)$  为 Hash 函数， $m$  为 Hash 表长度， $d_i$  为增量序列  $1, 2, \dots, m-1$ ，且  $d_i = i$ 。

**例 9-13** 为关键字序列为  $\{36, 7, 40, 11, 16, 81, 22, 8, 14\}$  建立 Hash 表，Hash 表表长为 11， $\text{Hash}(key) = key \% 11$ ，用线性探测法处理冲突。

解 建立的 Hash 表如图 9-25 所示。

| 0  | 1  | 2 | 3  | 4  | 5  | 6  | 7 | 8  | 9 | 10 |
|----|----|---|----|----|----|----|---|----|---|----|
| 11 | 22 |   | 36 | 81 | 16 | 14 | 7 | 40 | 8 |    |

图 9-25 线性探测法处理冲突时的 Hash 表

在建立 Hash 表的过程中，36、7、11、16、81 均是由 Hash 函数得到的没有冲突的 Hash

|      |     |      |
|------|-----|------|
| 253  | 364 | 253  |
| 463  |     | 364  |
| 587  |     | 587  |
| + 05 |     | + 50 |
| 1308 |     | 1254 |

Hash(key)=308  
移位法

Hash(key)=254  
间界叠加法

图 9-24 折叠法 Hash 函数示例

地址而直接存入的。

$\text{Hash}(40) = 7$ , 由于 Hash 地址上冲突, 因此需寻找下一个空的 Hash 地址。

由  $H_1 = (\text{Hash}(40) + 1) \% 11 = 8$  得, Hash 地址 8。该地址为空, 将 40 存入。此外, 22、8 同样在 Hash 地址上有冲突, 也是由  $H_1$  找到空的 Hash 地址的。

而  $\text{Hash}(14) = 3$ , Hash 地址冲突。

由  $H_1 = (\text{Hash}(14) + 1) \% 11 = 4$ , 仍然冲突;

由  $H_2 = (\text{Hash}(14) + 2) \% 11 = 5$ , 仍然冲突;

由  $H_3 = (\text{Hash}(14) + 3) \% 11 = 6$ , 找到空的 Hash 地址, 存入。

由于线性探测法可能使第  $i$  个 Hash 地址的同义词存入第  $i+1$  个 Hash 地址, 这样本应存入第  $i+1$  个 Hash 地址的元素变成了第  $i+2$  个 Hash 地址的同义词; 因此可能出现很多元素在相邻的 Hash 地址上堆积起来, 大大降低了查找效率。为此, 可采用二次探测法或双 Hash 函数探测法, 以改善堆积问题。

② 二次探测法。当发生冲突时, 二次探测法寻找下一个散列地址的公式为

$$H_i = (\text{Hash}(\text{key}) \pm d_i) \% m$$

其中,  $d_i$  为增量序列  $\{1^2, -1^2, 2^2, -2^2, \dots, q^2, -q^2\}$ ,  $q \leq m/2$ 。

仍以例 9-13 中的关键字建立 Hash 表。用二次探测法处理冲突, 建立的 Hash 表如图 9-26 所示。

| 0  | 1  | 2  | 3  | 4  | 5  | 6 | 7 | 8  | 9 | 10 |
|----|----|----|----|----|----|---|---|----|---|----|
| 11 | 22 | 14 | 36 | 81 | 16 |   | 7 | 40 | 8 |    |

图 9-26 二次探测法处理冲突时的 Hash 表

对关键字寻找空的 Hash 地址, 只有 14 这个关键字与图 9-25 所示的 Hash 表不同。

$\text{Hash}(14) = 3$ , Hash 地址上冲突。

由  $H_1 = (\text{Hash}(14) + 1^2) \% 11 = 4$ , 仍然冲突;

由  $H_2 = (\text{Hash}(14) - 1^2) \% 11 = 2$ , 找到空的 Hash 地址, 存入。

③ 双 Hash 函数探测法。先用第一个函数  $\text{Hash}(\text{key})$  对关键字计算 Hash 地址。一旦产生地址冲突, 再用第二个函数  $\text{ReHash}(\text{key})$  确定移动的步长因子。最后, 通过步长因子序列由探测函数寻找空的 Hash 地址, 即

$$H_i = (\text{Hash}(\text{key}) + i \times \text{ReHash}(\text{key})) \% m \quad (i = 1, 2, \dots, m-1)$$

其中,  $\text{Hash}(\text{key})$  和  $\text{ReHash}(\text{key})$  是两个 Hash 函数,  $m$  为 Hash 表长度。

例如,  $\text{Hash}(\text{key}) = a$  时产生地址冲突, 就计算  $\text{ReHash}(\text{key}) = b$ ; 则探测的地址序列为  $\{H_1 = (a + b) \% m, H_2 = (a + 2b) \% m, \dots, H_{m-1} = (a + (m-1)b) \% m\}$ 。

## (2) 链地址法(拉链法)

链地址法的基本思想是将所有 Hash 地址相同的记录存储在一个单链表中，在 Hash 表中存储指向各单链表的头指针。

**例 9-14** 关键字序列为{36, 7, 40, 11, 16, 81, 22, 8, 14}建立 Hash 表，Hash 函数为  
 $\text{Hash}(\text{key}) = \text{key \% } 11$

用链地址法处理冲突。

解 建立的 Hash 表如图 9-27 所示。

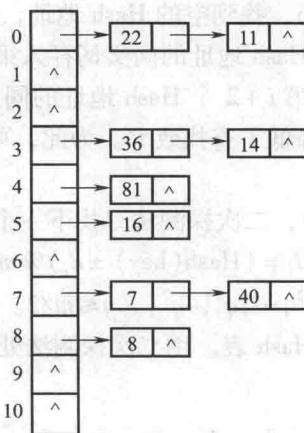


图 9-27 拉链法处理冲突时的 Hash 表示例

### 9.4.3 Hash 查找算法及分析

在 Hash 表上查找给定值  $k$  的过程基本上和建立 Hash 表的过程相同。

首先，根据 Hash 函数求出给定值  $k$  的 Hash 地址。然后，用 Hash 表中位于该地址的记录关键字值与  $k$  值进行比较。如果相等，说明查找成功；否则，按处理冲突的方法，去下一个地址进行查找，直到 Hash 表中某个位置为空（查找不成功）或表中某个位置的记录关键字值与  $k$  值相等（查找成功）。

下面分别给出以线性探测法解决冲突建立的 Hash 表和以链地址法解决冲突建立的 Hash 表的查找算法。为描述清晰，设 Hash 记录为整型数，Hash 表长度为  $m$ ，Hash 函数为  $\text{Hash}(\text{key}) = \text{key \% } m$ 。

#### 算法 9-10 以线性探测法解决冲突建立的 Hash 表的查找算法

```
int HashSearch_1(int hash[], int m, int k)
```

```
{
```

```

pos = k % m; //计算 Hash 地址
t = pos;
while(hash[pos] != EMPTY) //当 Hash 地址中的记录不为空时循环
{
 if(hash[pos] == k)
 return pos; //查找成功,返回下标
 else
 pos = (pos + 1) % m;
 if(pos == t)
 return -1; //查找失败,返回 -1
}
return -1; //查找失败,返回 -1
}

```

### 算法 9-11 以链地址法解决冲突建立的 Hash 表的查找算法

```

Node * HashSearch_2(Node * hash[], int m, int k)
{
 pos = k % m; //计算 Hash 地址
 p = hash[pos]; //p 指向对应单链表的表头
 while (p && p->data != k) //在对应单链表中顺序查找
 p = p->next;
 if (p)
 return p; //查找成功,返回地址
 else
 return NULL; //查找失败,返回空指针
}

```

因为在上述两种算法中,产生冲突后的查找仍然是给定值与关键字进行比较的过程,所以对 Hash 表查找效率的量度依然用平均查找长度来衡量。

在查找过程中,关键字的比较次数取决于产生冲突的多少。冲突少,查找效率就高;冲突多,查找效率就低。因此,影响产生冲突多少的因素也就是影响查找效率的因素。

影响产生冲突多少的因素有 3 个:Hash 函数是否均匀、处理冲突的方法以及 Hash 表的装填因子。

从前面的分析可以看出,尽管 Hash 函数的好坏直接影响冲突产生的频度,但一般情况下都认为所选的 Hash 函数是均匀的。因此,可不考虑 Hash 函数对平均查找长度的影响。

从线性探测法和链地址法处理冲突的例子可以看出,即使对于相同的关键字集合、同样的 Hash 函数,在记录查找等概率情况下,它们的平均查找长度却不同。

在例 9-13 中, 线性探测法的平均查找长度  $ASL = (5 \times 1 + 3 \times 2 + 1 \times 4) / 9 = 5/3$ 。

例 9-14 中, 链地址法的平均查找长度  $ASL = (6 \times 1 + 3 \times 2) / 9 = 4/3$ 。

实际上, Hash 表的平均查找长度是装填因子  $\alpha$  的函数, 只是不同处理冲突的方法有不同的函数。表 9-3 所示的是 4 种处理冲突方法的平均查找长度。

表 9-3 几种处理冲突方法的平均查找长度

| 处理冲突的方法        | 平均查找长度                                                             |                                                                        |
|----------------|--------------------------------------------------------------------|------------------------------------------------------------------------|
|                | 查找成功时                                                              | 查找不到时                                                                  |
| 线性探测法          | $S_{nl} \approx \frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right)$ | $U_{nl} \approx \frac{1}{2} \left( 1 + \frac{1}{(1-\alpha)^2} \right)$ |
| 二次探测法与双 Hash 法 | $S_{nr} \approx -\frac{1}{\alpha} \ln (1-\alpha)$                  | $U_{nr} \approx \frac{1}{1-\alpha}$                                    |
| 拉链法            | $S_{nc} \approx 1 + \frac{\alpha}{2}$                              | $U_{nc} \approx \alpha + e^{-\alpha}$                                  |

Hash 表的优点是存取速度快, 也较节省空间, 静态查找、动态查找均适用。但是, 由于存取是随机的, 因此 Hash 表不便于顺序查找。

## 本章小结

由于查找是数据处理领域中常用的一种基本操作, 同时也是许多程序中最消耗时间的一种操作, 因此一个好的查找方法将会大大提高程序的运行速度。根据查找时查找表是否可进行插入、删除操作, 查找表分为静态和动态两种。本章介绍的顺序表查找属于静态查找表上的查找, 树表查找和 Hash 查找属于动态查找表上的查找。

本章主要学习要点如下:

- (1) 掌握顺序表上的各种查找算法。
- (2) 掌握各种树表查找, 包括二叉排序树、平衡二叉树、B 树和 B+ 树。
- (3) 掌握 Hash 表的构造及查找方法。

## 习题 9

9.1 设有序顺序表中的元素依次为 17, 54, 67, 75, 85, 93, 94, 112, 203, 261。试画出对其进行折半查找时的二叉判定树, 并计算查找成功的平均查找长度和查找不到的平均查找长度。

9.2 若对有  $n$  个元素的有序顺序表和无序顺序表进行顺序搜索, 试就下列 3 种情况分别讨论两者在等查找概率时的平均查找长度是否相同?

(1) 查找失败。

(2) 查找成功, 且表中只有一个关键字等于给定值  $k$  的记录。

(3) 查找成功, 且表中有若干个关键字等于给定值  $k$  的记录, 要求一次查找找出所有记录。

9.3 按照分块查找, 对 144 个元素的表分成多少块最好? 每块的最佳长度是多少? 假定每块的长度为 8, 其平均查找长度是多少?

9.4 设有一个输入数据的序列是 {46, 25, 78, 62, 12, 37, 70, 29}, 试画出从空树起, 逐个输入各个数据而生成的二叉排序树。

9.5 已知下列长度为 12 的表 (Jan, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec)。

(1) 试按表中元素的顺序将每个元素依次插入一棵初始状态为空的二叉排序树, 请画出插入完成之后的二叉排序树, 并求其在等概率情况下查找成功的平均查找长度。

(2) 按表中元素顺序构造一棵平衡二叉树, 并求其在等概率情况下查找成功的平均查找长度。

9.6 图 9-28 所示的是一个 3 阶  $B$  树。试分别画出在插入 65、15、40、30 之后  $B$  树的变化。

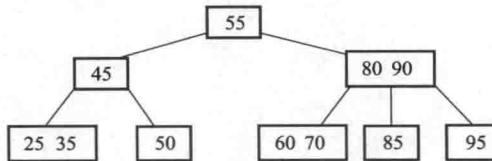


图 9-28 3 阶  $B$  树

9.7 在图 9-29 所示的 4 阶  $B$  树中插入关键字 87, 请画出插入调整后树的形状。

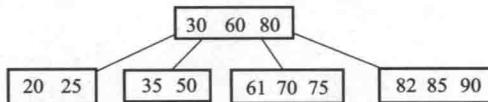


图 9-29 4 阶  $B$  树

9.8 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、4、9、16、25、36、49、64、81、100、121、144、169、196、225, 每个数据页块存放 3 个记录, 要求如下:

(1) 用  $B$  树组织索引, 设阶  $m=3$ , 依次将上述 15 个关键码插入  $B$  树, 画出插入后的  $B$  树。

(2) 用  $B+$  树组织索引, 设阶  $m=3$ , 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)。

## 上机实验题 9

**实验题 9.1 实现顺序表上的查找算法。**

基本要求:

(1) 实现顺序查找的带监督哨算法和不带监督哨算法。

(2) 实现折半查找的递归和非递归算法。

(3) main 函数中使用菜单选择各项功能。

### 实验题 9.2 实现二叉排序树上的查找算法。

基本要求：

- (1) 用二叉链表做存储结构，输入若干整数，建立一棵二叉排序树。
- (2) 判断是否是二叉排序树。
- (3) 在二叉排序树上插入结点。
- (4) 删除二叉排序树上的某一个结点。
- (5) 输入关键字，在二叉排序树上实现查找。



图 9.1 建立二叉排序树



图 9.2 二叉排序树的插入操作

模式对查 (F)

实验题 9.3 实现二叉排序树上的插入操作。

基本要求：

- (1) 用二叉链表做存储结构，输入若干整数，建立一棵二叉排序树。
- (2) 判断是否是二叉排序树。
- (3) 在二叉排序树上插入结点。
- (4) 删除二叉排序树上的某一个结点。
- (5) 输入关键字，在二叉排序树上实现查找。

模式对查 (F)

实验题 9.4 实现二叉排序树上的删除操作。

基本要求：

- (1) 用二叉链表做存储结构，输入若干整数，建立一棵二叉排序树。
- (2) 判断是否是二叉排序树。
- (3) 在二叉排序树上插入结点。
- (4) 删除二叉排序树上的某一个结点。
- (5) 输入关键字，在二叉排序树上实现查找。

模式对查 (F)

实验题 9.5 实现二叉排序树上的查找操作。

基本要求：

- (1) 用二叉链表做存储结构，输入若干整数，建立一棵二叉排序树。
- (2) 判断是否是二叉排序树。
- (3) 在二叉排序树上插入结点。
- (4) 删除二叉排序树上的某一个结点。
- (5) 输入关键字，在二叉排序树上实现查找。

模式对查 (F)

实验题 9.6 实现二叉排序树上的插入操作。

基本要求：

- (1) 用二叉链表做存储结构，输入若干整数，建立一棵二叉排序树。
- (2) 判断是否是二叉排序树。
- (3) 在二叉排序树上插入结点。
- (4) 删除二叉排序树上的某一个结点。
- (5) 输入关键字，在二叉排序树上实现查找。

模式对查 (F)

实验题 9.7 实现二叉排序树上的删除操作。

基本要求：

- (1) 用二叉链表做存储结构，输入若干整数，建立一棵二叉排序树。
- (2) 判断是否是二叉排序树。
- (3) 在二叉排序树上插入结点。
- (4) 删除二叉排序树上的某一个结点。
- (5) 输入关键字，在二叉排序树上实现查找。

模式对查 (F)

# 第 10 章 排 序

排序 (Sorting) 是数据处理中经常使用的操作, 如高考成绩排名、体育比赛排名等。排序的目的是便于查找, 通过排序可以提高查找性能。例如, 字典和电话号码簿中的信息都是按照一定的顺序编排的, 从而方便查找所需信息。可见, 排序是计算机程序设计中一种基础性操作, 研究和掌握各种排序方法非常重要。本章介绍排序的基本概念, 讨论几种比较经典而又重要的排序算法, 并对各种排序算法的性能进行比较。

## 10.1 排序的基本概念

### 1. 排序

设有  $n$  个记录的序列  $\{r_0, r_1, \dots, r_{n-1}\}$ , 其相应的关键字分别是  $\{k_0, k_1, \dots, k_{n-1}\}$ , 排序是将这  $n$  个记录重新排列, 使之按关键字大小递增 (或递减) 有序排列。从操作角度看, 排序是线性结构的一种操作, 待排记录可以用顺序结构或链接结构来存储。本章讨论的排序算法均采用顺序存储结构, 并假定关键字为整数, 数组的长度为  $n$ , 待排数据表的类型定义如下:

```
struct Record
{
 int key;
 ...
};

Record r[n];
```

### 2. 内排序与外排序

根据排序过程中数据所占用存储器的不同, 可将排序分为两类。一类是整个排序过程全部在内存中进行, 称为内排序; 另一类是由于待排记录数据量太大, 内存无法容纳全部数据, 在排序过程中需要借助外存才能完成, 称为外排序。本章将讨论几种重要的内排序算法。

### 3. 排序的稳定性

假设在待排序列中存在多个具有相同关键字的记录, 设  $k_i = k_j$ ,  $0 \leq i \leq n-1$ ,  $0 \leq j \leq n-1$ ,  $i \neq j$ 。若排序前  $r_i$  排在  $r_j$  的前面, 排序后  $r_i$  仍在  $r_j$  的前面, 则称这种排序方法是稳定的; 否则, 称这种排序方法是不稳定的。在某些排序场合, 如选举、比赛等, 对排序方法要求是稳定的。

例如，一组学生记录已按学号排好序，现需要按照考试成绩进行排序，当成绩相同时，要求学号小的排在前面。显然，这时必须选用稳定的排序方法。要证明一种排序方法是稳定的，就必须从算法本身的步骤中加以证明。但是，要证明一种排序方法是不稳定的，则只要给出一个实例说明即可。

#### 4. 排序算法的性能指标

排序算法的性能指标主要包括算法的时间复杂度、空间复杂度和算法的稳定性，其中排序算法的时间开销主要取决于关键字的比较次数和记录的移动次数，空间开销主要是执行算法所需要的辅助存储空间。

## 10.2 冒泡排序

冒泡排序（Bubble Sort）是一种最简单的排序方法，其基本思想是两两比较相邻记录的关键字，若反序，则交换相邻两记录，直到没有反序的记录。冒泡排序的处理过程如下。

首先，将第一个记录的关键字值与第二个记录的关键字值进行比较。若为反序（即  $r[j].key > r[j+1].key$ ），则交换之。然后比较第二个记录和第三个记录的关键字值。依此类推，直到第  $n-1$  个记录关键字值和第  $n$  个记录的关键字值进行比较。上述过程称为冒泡排序的第一趟处理，第一趟处理结果使得关键字值最大的记录被放在最后一个位置上。

接下来进行第二趟处理，即对前  $n-1$  个记录进行类似处理，其结果是使关键字值次大的记录被放到倒数第二的位置上。

一般来说，冒泡排序的第  $i$  趟处理是从  $r[0]$  到  $r[n-i]$ ，依次比较相邻两个记录的关键字，并在反序时交换相邻的记录，其结果是将这  $n-i+1$  个记录中关键字值最大的元素交换到第  $n-i+1$  的位置上。整个排序过程需进行  $n-1$  趟处理，第  $n-1$  趟处理使关键字值为倒数第二的记录放在第二的位置上，这时整个排序过程结束。

下面看一个冒泡排序的例子，如图 10-1 所示。

|         |    |    |    |    |    |    |    |
|---------|----|----|----|----|----|----|----|
| 初始序列：   | 51 | 38 | 49 | 27 | 62 | 05 | 16 |
| 第 1 趟结果 | 38 | 49 | 27 | 51 | 05 | 16 | 62 |
| 第 2 趟结果 | 38 | 27 | 49 | 05 | 16 | 51 |    |
| 第 3 趟结果 | 27 | 38 | 05 | 16 | 49 |    |    |
| 第 4 趟结果 | 27 | 05 | 16 | 38 |    |    |    |
| 第 5 趟结果 | 05 | 16 | 27 |    |    |    |    |
| 第 6 趟结果 | 05 | 16 |    |    |    |    |    |

图 10-1 冒泡排序过程示例

冒泡排序算法描述如算法 10-1 所示。

### 算法 10-1 冒泡排序 BubbleSort

```
void BubbleSort (Record r[], int n) //数组 r 中的数据保存在 r[0] ~ r[n-1] 中
{
 for (i=1; i < n; i++)
 for (j=0; j < n-i; j++)
 if (r[j].key > r[j+1].key)
 {
 temp = r[j];
 r[j] = r[j+1];
 r[j+1] = temp;
 }
}
```

从算法 10-1 的执行过程中不难发现，若在第  $i$  趟循环中相邻记录都是正序，则排序实际上已经完成，此后的处理都是多余的。因此，可对算法 10-1 进行改进。若某一趟循环不发生记录交换，则结束排序过程。改进后的冒泡排序算法如算法 10-2 所示。

### 算法 10-2 改进后的冒泡排序 BubbleSort1

```
void BubbleSort1(Record r[], int n) //数组 r 中的数据保存在 0 ~ n - 1 中
{
 bool exchange = true;
 int i = 1;
 while(exchange)
 {
 exchange = false;
 for(j=0; j < n-i; j++)
 if (r[j].key > r[j+1].key)
 {
 temp = r[j];
 r[j] = r[j+1];
 r[j+1] = temp;
 exchange = true;
 }
 i++;
 }
}
```

算法 10-2 中的变量 exchange 用于标记是否发生了交换。实际上，还可以考虑用其记录发生交换的位置，以进一步提高算法的效率，请读者思考完成。

冒泡排序算法的执行时间取决于排序的趟数。在最好情况下，待排序记录序列为正序，算法只需要执行一趟，进行  $n - 1$  次关键字值的比较，不需要移动记录，时间复杂度为  $O(n)$ 。

在最坏情况下，待排序记录序列为反序，每趟排序中只有一个最大的记录被交换到最终位置；因此算法需要执行  $n - 1$  趟。第  $i$  ( $1 \leq i < n$ ) 趟排序执行  $n - i$  次关键字值的比较和  $n - i$  次记录的交换，这样关键字值的比较次数为

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

记录的移动次数为

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

因此，算法的时间复杂度为  $O(n^2)$ 。

在平均情况下，冒泡排序的时间复杂度与最坏情况同数量级。

冒泡排序只需要一个记录的辅助空间，用来作为记录交换的暂存单元。

冒泡排序是一种稳定的排序方法。

### 10.3 选择排序

这里所说的选择排序是指简单选择排序，其基本思想是每趟排序在当前待排序列中选出关键字值最小的记录并添加到有序序列中。第  $i$  趟的处理过程是在  $n - i + 1$  ( $i = 1, 2, \dots, n - 1$ ) 个记录中，通过  $n - i$  次关键字值的比较选取关键字值最小的记录，并与第  $i$  个记录进行交换。

下面看一个简单选择排序的例子，如图 10-2 所示。

简单选择排序算法描述如算法 10-3 所示。

#### 算法 10-3 简单选择排序 SelectSort

```
void SelectSort (Record r[], int n) //数组 r 中的数据保存在
 //r[0] ~ r[n - 1]
{
 for (i = 0; i < n - 1; i++)
 {
 k = i;
 for (j = i + 1; j < n; j++)
 if (r[j].key < r[k].key)
 k = j;
 }
}
```

|         |                      |
|---------|----------------------|
| 初始序列:   | 49 27 65 97 76 13 38 |
| 第 1 趟结果 | 13 27 65 97 76 49 38 |
| 第 2 趟结果 | 13 27 65 97 76 49 38 |
| 第 3 趟结果 | 13 27 38 97 76 49 65 |
| 第 4 趟结果 | 13 27 38 49 76 97 65 |
| 第 5 趟结果 | 13 27 38 49 65 97 76 |
| 第 6 趟结果 | 13 27 38 49 65 76 97 |

图 10-2 简单选择排序的过程示例

```

if (k!=i)
{
 temp = r[i];
 r[i] = r[k];
 r[k] = temp;
}
}

```

容易看出，在简单选择排序中记录的移动次数较少。在待排记录序列为正序时，记录的移动次数量少，为 0 次；在待排序序列为逆序时，记录的移动次数量多，为  $3(n-1)$  次。

无论记录的初始排列如何，关键字的比较次数都相同。第  $i$  趟排序需进行  $n-i$  次关键字的比较，而简单选择排序需进行  $n-1$  趟排序，则总的比较次数为

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

因此，总的时间复杂度为  $O(n^2)$ 。

在简单选择排序过程中，只需要一个用来作为记录交换的暂存单元。

简单选择排序是一种不稳定的排序方法。

## 10.4 插入排序

插入排序的基本思想是将待排序表看作是左、右两部分，其中左边为有序区，右边为无序区；整个排序过程就是将右边无序区中的记录依次按关键字大小逐个插入到左边有序区中，以构成新的有序区，直到全部记录都排好序。本节介绍两种插入排序方法：直接插入排序和折半插入排序。

### 10.4.1 直接插入排序

直接插入排序具体的排序过程是：

① 将整个待排序的记录序列划分成有序区和无序区，初始时有序区为待排序记录序列中的第一个记录，无序区包括所有剩余其他记录；

② 将无序区的第一个记录插入到有序区的合适位置中，从而使无序区减少一个记录，有序区增加一个记录；

③ 重复执行②，直到无序区中没有记录为止。

下面看一个直接插入排序的例子，如图 10-3 所示。

直接插入排序算法描述如算法 10-4 所示。

|         |                        |
|---------|------------------------|
| 初始序列:   | [12] 15 09 20 06 36 28 |
| 第 1 趟结果 | [12 15] 09 20 06 36 28 |
| 第 2 趟结果 | [09 12 15] 20 06 36 28 |
| 第 3 趟结果 | [09 12 15 20] 06 36 28 |
| 第 4 趟结果 | [06 09 12 15 20] 36 28 |
| 第 5 趟结果 | [06 09 12 15 20 36] 28 |
| 第 6 趟结果 | [06 09 12 15 20 28 36] |

图 10-3 直接插入排序过程示例

### 算法 10-4 直接插入排序 InsertSort

```
void InsertSort(Record r[], int n)
{
 for (i = 1; i < n; i ++)
 {
 temp = r[i];
 for (j = i - 1; j >= 0 && temp. key < r[j]. key; j --)
 r[j + 1] = r[j];
 r[j + 1] = temp;
 }
}
```

直接插入排序算法由两层嵌套循环组成，外层循环控制排序趟数，执行  $n-1$  次；内层循环的执行次数取决于在第  $i$  个记录前有多少个记录的关键字值大于第  $i$  个记录的关键字值。

在最好情况下，待排记录序列为正序，每趟只需与有序序列的最后一个记录的关键字比较一次，移动两次记录。总的比较次数为  $n-1$ ，记录移动的次数为  $2(n-1)$ ，因此时间复杂度为  $O(n)$ 。

在最坏情况下，待排记录序列为逆序，在第  $i$  趟插入时，每  $i$  个记录必须与前面  $i-1$  个记录的关键码做比较，并且每比较一次就要做一次记录的移动，则比较总次数为

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

记录的移动总次数为

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

因此，时间复杂度为  $O(n^2)$ 。

在平均情况下，待排记录序列中各种可能排列的概率相同，在插入第  $i$  个记录时平均需要比较有序区中全部记录的一半，因此总的比较次数为

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

总的移动次数为

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

因此，时间复杂度为  $O(n^2)$ 。

直接插入排序只需要一个记录的辅助空间用来作为待插入记录的暂存单元。

直接插入排序是一种稳定的排序方法。它算法简单、容易实现，当待排记录序列基本有序或待排记录较少时，它是最佳的排序方法。但是，当待排记录个数较多时，大量的比较和移动

操作使它的效率降低。算法 10-4 所示的一个关键步骤是顺序查找待插入位置，读者可以利用 9.2.1 节有监视哨的顺序查找提高算法效率。

## 10.4.2 折半插入排序

直接插入排序利用顺序查找方法确定记录的插入位置。从第 9 章关于查找的讨论中可知，对于有序表采用折半查找方法，其性能优于顺序查找。因此，在插入排序过程中可以利用折半查找方法来确定记录的插入位置，相应的插入排序方法称为折半插入排序。

折半插入排序算法描述如算法 10-5 所示。

### 算法 10-5 折半插入排序 BinInsertSort

```
void BinInsertSort(Record r[], int n)
```

```
{
 for(i = 1; i < n; i++)
 {
 temp = r[i],
 low = 0;
 high = i - 1;
 while(low <= high)
 {
 mid = (low + high) / 2
 if (temp. key < r[mid]. key)
 high = mid - 1;
 else
 low = mid + 1;
 }
 for(j = i - 1; j >= low; j--)
 r[j + 1] = r[j];
 r[low] = temp;
 }
}
```

采用折半插入排序法可减少关键字的比较次数。每插入一个元素，需要比较的次数最多为折半查找判定树的深度。例如，插入第  $i$  个元素时，则需进行  $\log_2 i$  次比较。因此，插入  $n - 1$  个元素的平均比较次数为  $O(n \log_2 n)$ 。

与直接插入排序法相比较，折半插入排序法虽然改善了算法中比较次数的数量级为  $O(n \log_2 n)$ ，但其并未改变移动元素的时间耗费，因此折半插入排序总的时间复杂度仍然是  $O(n^2)$ 。

## 10.5 希尔排序

希尔排序 (Shell Sort) 是对直接插入排序的一种改进，它利用了插入排序的两个性质：

- (1) 若待排记录按关键字值基本有序，则直接插入排序效率很高；
- (2) 若待排记录个数较少，则直接插入排序效率也较高。

因此，希尔排序先将待排序列划分为若干小序列，在这些小序列中进行插入排序；然后逐步扩大小序列的长度，减少小序列的个数，这样使待排序列逐渐处于更有序的状态；最后对全体序列进行一次直接插入排序，从而完成排序。

希尔排序是 D. L. Shell 于 1959 年提出的。在希尔排序中，要解决的关键问题如下：

- (1) 应如何划分待排序列才能保证整个序列逐步向基本有序发展？
- (2) 子序列内如何进行直接插入排序？

下面看一个希尔排序的例子，如图 10-4 所示。

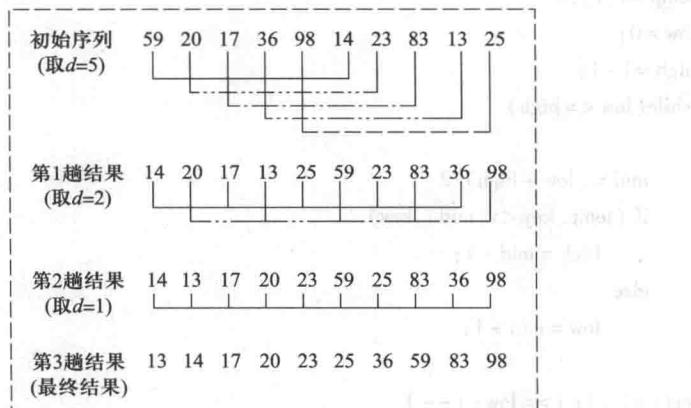


图 10-4 希尔排序过程示例

具体的排序过程是假设待排序的记录为  $n$  个，首先取整数  $d < n$ 。例如，取  $d = \lfloor n/2 \rfloor$ ，将所有相距为  $d$  的记录构成一组，从而将整个待排序记录序列分割为  $d$  个子序列。如图 10-4 所示，首先对每个子序列分别进行直接插入排序；然后再缩小间隔  $d$ 。例如，取  $d = d/2$ ，首先重复上述分割，再对每个子序列分别进行直接插入排序；直到最后取  $d = 1$ ，即将所有记录放在一个进行一次直接插入排序，最终将所有记录重新排列成按关键字有序的序列。

从以上排序过程可得到解决上述关键问题的方法。

问题（1）的解决方法：子序列的构成不能是简单地逐段分割，而是将相距某个“增量”

的记录组成一个子序列，这样才能有效地保证在子序列内分别进行直接插入排序后得到的结果是基本有序而不是局部有序。接下来的问题是增量应如何取？到目前为止尚未有人求得一个最好的增量序列。希尔最早提出的方法是  $d_1 = \lfloor n/2 \rfloor, d_{i+1} = d_i/2$ ，且没有除 1 之外的公因子，最后一个增量必须等于 1。开始时增量的取值较大，每个子序列中的记录个数较少，并且提供了记录跳跃移动的可能，排序效率较高；后来增量逐步缩小，每个子序列中的记录个数增加，但已基本有序，效率也较高。

问题（2）的解决方法：在每个子序列中，将待插入记录和同一子序列中的前一个记录比较。在插入记录  $r[i]$  时，自  $r[i-d]$  起往前跳跃式（跳跃幅度为  $d$ ）查找待插入位置。在查找过程中，记录后移也是跳跃  $d$  个位置。

完整的希尔排序算法如算法 10-6 所示。

#### 算法 10-6 希尔排序算法 ShellSort

```
void ShellSort(Record r[], int n)
{
 for(d = n/2; d >= 1; d = d/2) //以增量为 d 进行直接插入排序
 {
 for(i = d; i < n; i++)
 {
 temp = r[i]; //暂存被插入记录
 for(j = i - d; j >= 0 && temp. key < r[j]. key; j = j - d)
 r[j + d] = r[j]; //记录后移 d 个位置
 r[j + d] = temp;
 }
 }
}
```

对希尔排序算法的时间性能的分析是一个复杂的问题，因为它是所取增量的函数。有人在大量实验的基础上指出，希尔排序的时间性能在  $O(n^2)$  和  $O(n \log_2 n)$  之间。当  $n$  在某个特定范围时，希尔排序的时间性能约为  $O(n^{1.3})$ 。

希尔排序只需要一个记录的辅助空间，用于暂存当前待插入的记录。

希尔排序是一种不稳定的排序方法。

## 10.6 快速排序

1962 年，伦敦 Elliot Brothers Ltd 公司的 Tony Hoare 发明了快速排序（Quick Sort）方法。实际上快速排序名副其实，它几乎是最快的排序算法，被评为 20 世纪十大算法之一。

快速排序算法的基本思想是从待排序记录序列中选取一个记录（通常选取第一个记录）为枢轴，其关键字值设为  $k$ 。将关键字值小于  $k$  的记录移到前面，而将关键字值大于  $k$  的记录移到后面。结果将待排序记录序列分成两个子表，最后将关键字值为  $k$  的记录插到分界线处。将这个过程称为划分。通过一次划分后，以关键字值  $k$  为基准，将待排序序列分成了两个子表。前面子表中所有记录的关键字值均不大于  $k$ ，后面子表中所有记录的关键字值均不小于  $k$ 。对划分后的子表继续按上述原则进行划分，直到所有子表的表长不超过 1，此时待排序记录序列就变成了一个有序序列。

显然，快速排序是一个递归过程，需解决的关键问题是在待排序列中如何进行划分？划分的过程如下：

① 取第一个记录的关键字值作为基准，将第一个记录暂存于  $\text{temp}$  中，设两个变量  $i$ 、 $j$  分别指示将要划分的最左、最右记录的位置。

② 将  $j$  指向的记录关键字值与基准值进行比较。如果  $j$  指向的记录关键字值大，则  $j$  前移一个位置。重复此过程，直到  $j$  指向的记录关键字值小于基准值。若  $i < j$ ，则将  $j$  指向的记录移到  $i$  所指位置。

③ 将  $i$  指向的记录关键字值与基准值进行比较。如果  $i$  指向的记录关键字值小，则  $i$  后移一个位置。重复此过程，直到  $i$  指向的记录关键字值大于基准。若  $i < j$ ，则  $i$  指向的记录移到  $j$  所指位置。

④ 重复②、③步，直到  $i = j$ 。

下面看一个划分的例子。设以第一个记录关键字值为基准，将其存入  $\text{temp}$  中，如图 10-5 所示。

| 初始序列                                   | 33                   | 23 | 59                   | 16           | 41           | 29           | 38  |
|----------------------------------------|----------------------|----|----------------------|--------------|--------------|--------------|-----|
|                                        | $\uparrow i$         |    |                      |              |              | $\uparrow j$ |     |
| 38>33, $j$ 前移一位                        | 33                   | 23 | 59                   | 16           | 41           | 29           | 38  |
|                                        | $\uparrow i$         |    |                      |              | $\uparrow j$ |              |     |
| 29<33, $r[j] \rightarrow r[i]$         | 29                   | 23 | 59                   | 16           | 41           | 29           | 38  |
| $i$ 后移一位                               | $\uparrow i$         |    |                      |              | $\uparrow j$ |              |     |
| 23<33, $i$ 后移一位                        | 29                   | 23 | 59                   | 16           | 41           | 29           | 38  |
|                                        | $\uparrow i$         |    |                      |              | $\uparrow j$ |              |     |
| 59>33, $r[i] \rightarrow r[j]$         | 29                   | 23 | 59                   | 16           | 41           | 59           | 38  |
| $j$ 前移一位                               | $\uparrow i$         |    |                      | $\uparrow j$ |              |              |     |
| 41>33, $j$ 前移一位                        | 29                   | 23 | 59                   | 16           | 41           | 59           | 38  |
|                                        | $\uparrow i$         |    |                      | $\uparrow j$ |              |              |     |
| 16<33, $r[j] \rightarrow r[i]$         | 29                   | 23 | 16                   | 16           | 41           | 59           | 38  |
| $i$ 后移一位                               | $\uparrow i$         |    | $\uparrow j$         |              |              |              |     |
| $i=j$ , $\text{temp} \rightarrow r[i]$ | [29                  | 23 | 16]                  | 33           | [41          | 59           | 38] |
| 一次划分结束                                 | $\uparrow\uparrow i$ |    | $\uparrow\uparrow j$ |              |              |              |     |

图 10-5 一次划分的过程示例

快速排序的划分算法描述如算法 10-7 所示。

### 算法 10-7 快速排序的一次划分算法 Partition

```
int Partition(Record r[], int i, int j)
```

```
{
 temp = r[i];
 while (i < j)
 {
 while (i < j && r[j].key >= temp.key)
 j--;
 if (i < j)
 r[i++] = r[j];
 while (i < j && r[i].key <= temp.key)
 i++;
 if (i < j)
 r[j--] = r[i];
 }
 r[i] = temp;
 return i;
}
```

整个快速排序的过程可递归进行。若待排序列只有一个记录，则递归结束；否则进行一次划分后，再分别对划分得到的两个子序列进行快速排序（递归调用）。

具体的快速排序算法描述如算法 10-8 所示。

### 算法 10-8 快速排序算法 QuickSort

```
void QuickSort (Record r[], int i, int j)
```

```
{
 if (i < j)
 {
 pivot = Partition(r, i, j);
 QuickSort(r, i, pivot - 1);
 QuickSort(r, pivot + 1, j);
 }
}
```

显然，初始调用为 `QuickSort(r, 0, n - 1)`。

快速排序的趟数取决于递归的深度。在最好情况下，每次划分对一个记录定位后，该记录的左侧子序列与右侧子序列的长度相同。在具有  $n$  个记录的序列中，对一个记录定位需要对整

个待划分序列扫描一遍，则所需时间为  $O(n)$ 。

设  $T(n)$  是对  $n$  个记录的序列进行排序的时间。每次划分后，正好把待划分区间划分为长度相等的两个子序列，则有：

$$\begin{aligned} T(n) &\leq 2T(n/2) + n \\ &\leq 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n \\ &\leq 4(2T(n/8) + n/4) + 2n = 8T(n/8) + 3n \\ &\dots \\ &\leq nT(1) + n\log_2 n = O(n\log_2 n) \end{aligned}$$

因此，算法的时间复杂度为  $O(n\log_2 n)$ 。

在最坏情况下，待排序记录序列正序或逆序，每次划分只得到一个比上一次划分少一个记录的子序列（另一个子序列为空）。此时，必须经过  $n-1$  次递归调用才能把所有记录定位，而且第  $i$  趟划分需要经过  $n-i$  次关键字的比较才能找到第  $i$  个记录的基准位置。因此，总的比较次数为

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

记录的移动次数小于等于比较次数。因此，算法的时间复杂度为  $O(n^2)$ 。

在平均情况下，设基准记录的关键字第  $k$  小 ( $1 \leq k \leq n$ )，则有

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

这是快速排序的平均时间性能。可以用归纳法证明，其数量级也为  $O(n \log_2 n)$ 。

由于快速排序是递归的，因此需要一个栈用来存放每一层递归调用的必要信息，其最大容量应与递归调用的深度一致。最好情况下，栈的深度为  $O(\log_2 n)$ ；最坏情况下，因为要进行  $n-1$  次递归调用，所以栈的深度为  $O(n)$ ；平均情况下，栈的深度为  $O(\log_2 n)$ 。

快速排序是一种不稳定的排序方法。

## 10.7 堆排序

堆排序 (Heap Sort) 是 J. W. J. Williams 于 1964 年提出的。它是选择排序的一种改进，改进的着眼点是如何减少关键字的比较次数。由于选择排序没有把前一趟的比较结果保留下来，在后一趟选择时把前一趟已做过的比较又重复了一遍，因此记录的比较次数较多。堆排序利用每趟比较后的结果，也就是在找出关键字值最小记录的同时，也找出关键字值较小的记录，从而减少了在后面选择中的比较次数，从而提高了排序效率。

### 1. 堆的定义

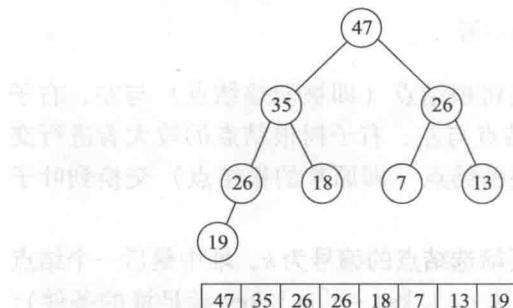
堆 (Heap) 是具有下列性质的完全二叉树：每个结点的值都小于或等于其左、右孩子结

点的值（小顶堆），或者每个结点的值都大于或等于其左、右孩子结点的值（大顶堆）。

如果将堆按层从上到下，每层从左到右对每个结点从 1 开始进行编号，那么结点之间满足如下关系：

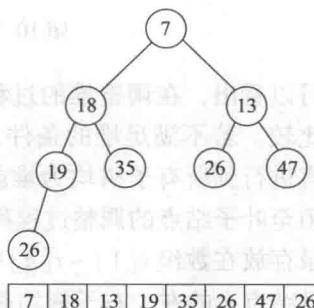
$$\begin{cases} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{cases} \quad \text{或} \quad \begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases} \quad 1 \leq i \leq \lfloor n/2 \rfloor$$

图 10-6 所示的是两个堆的示例。



(a) 大顶堆及其对应的序列

图 10-6 堆的示例



(b) 小顶堆及其对应的序列

从堆的定义可以看出，如果一棵完全二叉树是堆，那么它的根结点（称为堆顶）一定是当前堆中所有结点的最大者（大顶堆）或最小者（小顶堆）。以结点的编号作为下标，将堆用顺序存储结构（即数组）来存储，则堆对应于一组序列，如图 10-6 所示。

## 2. 堆排序

堆排序是利用堆（假设利用大顶堆）的特性进行排序的方法，其基本思想是首先用待排序的记录序列构造成一个堆，此时选出堆中所有记录的最大者，即堆顶记录；然后将它从堆中移走（通常将堆顶记录和堆中最后一个记录交换），并将剩余的记录再调整成堆，这样又找出了次大的记录；依此类推，直到堆中只有一个记录。

在堆排序过程中，需解决的关键问题如下：

- (1) 如何将一个无序序列构造成一个堆（即初始建堆）？
- (2) 当堆顶记录移走后，如何调整剩余记录，使之成为一个新的堆（即重建堆）？

为解决上述问题，首先讨论堆调整问题。即在一棵完全二叉树中，根结点的左、右子树都是堆，如何调整根结点，使整个完全二叉树成为一个堆？

图 10-7 (a) 所示的是一棵完全二叉树，且根结点 28 的左、右子树都是堆。为了将整个二叉树调整为堆，首先将根结点 28 与其左、右子树的根结点比较。根据堆的定义，应将 28 与 35 交换，如图 10-7 (b) 所示。经过这一次交换，破坏了原来左子树的堆结构，需要对左子

树进行调整，调整后的堆如图 10-7（c）所示。

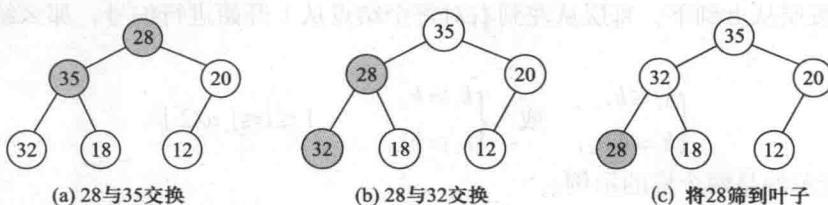


图 10-7 调整堆的示例

由这个例子可以看出，在调整堆的过程中，总是将根结点（即被调整结点）与左、右子树的根结点进行比较。若不满足堆的条件，则将根结点与左、右子树根结点的较大者进行交换。调整过程一直进行到所有子树均为堆或将被调整的结点（即原来的根结点）交换到叶子结点。这个自堆顶至叶子结点的调整过程称为筛选。

假设  $n$  个记录存放在数组  $r[1] \sim r[n]$  中，当前要筛选结点的编号为  $k$ ，堆中最后一个结点的编号为  $m$ ，并且结点  $k$  的左、右子树均是堆（即  $r[k+1].key \sim r[m].key$  满足堆的条件），则筛选算法描述如算法 10-9 所示。

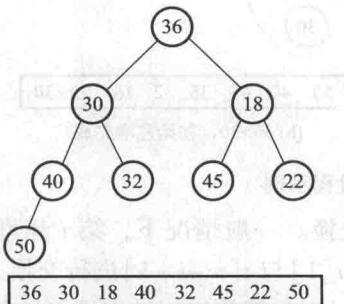
#### 算法 10-9 堆排序中的筛选算法 Sift

```
void Sift(Record r[], int k , int m)
{
 i = k ;
 j = 2 * i ; //置 i 为要筛的结点,j 为 i 的左孩子
 while (j <= m) //筛选还没有进行到叶子
 {
 if (j < m && r[j].key < r[j + 1].key)
 j ++ ; //比较 i 的左右孩子,j 为较大者
 if (r[i].key > r[j].key)
 break; //根结点已经大于左右孩子中的较大者
 else
 r[i]↔r[j]; //将根结点与结点 j 交换
 i=j;
 j=2 * i; //被筛选结点位于原来结点 j 的位置
 }
}
```

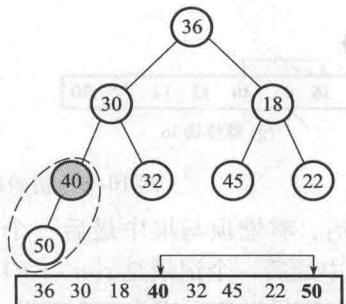
下面来讨论由一个无序序列建堆的过程。

由一个无序序列建堆的过程就是一个反复筛选的过程。因为此序列就是一个完全二叉树的顺序存储，则所有的叶子结点都已经是堆；所以只需从第 $\lfloor n/2 \rfloor$ 个记录（即最后一个分支结点）开始，执行上述筛选过程，直到根结点。图 10-8 所示的是一个初始建堆的例子。初始建堆算法描述为：

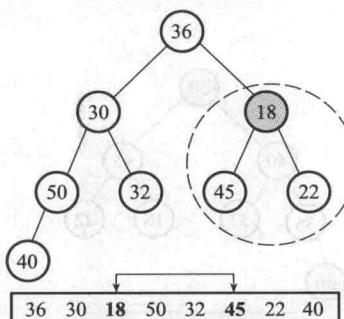
```
for (i = n/2; i >= 1; i--) // 初始建堆, 从最后一个分支结点至根结点进行筛选
 Sift (r, i, n);
```



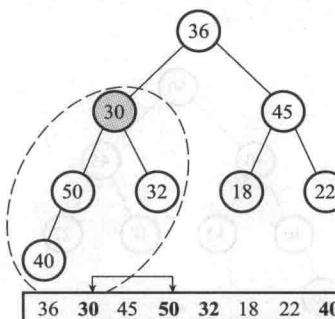
(a) 待排序序列



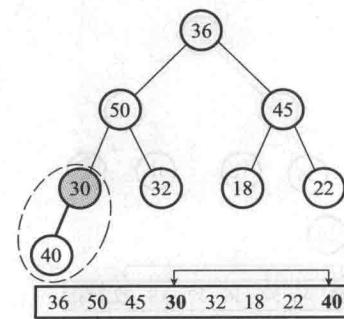
(b) 第1次调整, 篩40



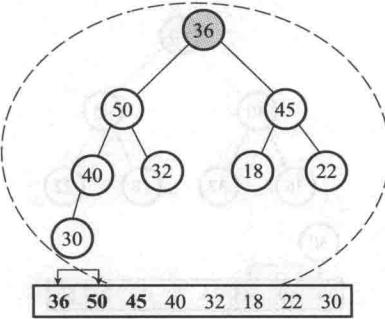
(c) 第2次调整, 篩18



(d) 第2次调整, 篩30



(e) 继续筛30, 直到叶子



(f) 第4次调整, 篩36

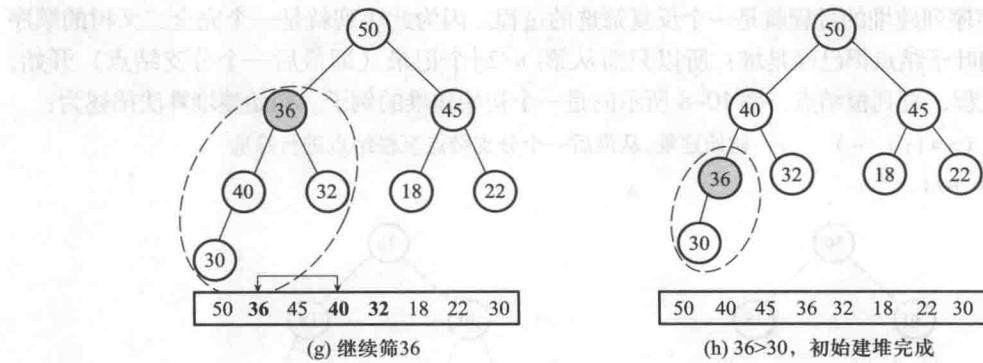
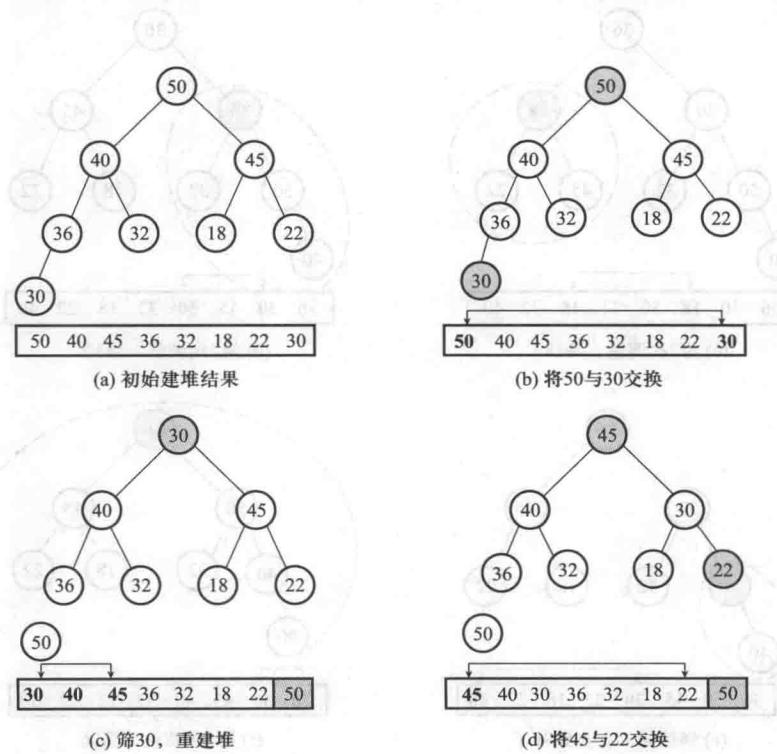


图 10-8 初始建堆的过程示例

初始建堆完成后，将堆顶与堆中最后一个记录交换。一般情况下，第  $i$  趟堆排序中堆有  $n - i + 1$  个记录，堆中最后一个记录是  $r[n - i + 1]$ ，将  $r[1]$  与  $r[n - i + 1]$  进行交换。

接下来调整剩余记录，使之成为一个新堆。这一过程只需筛选根结点即可重新建堆。

图 10-9 所示的是一个堆排序的例子，图中只给出了前两趟的排序结果，其余部分请读者自行给出。



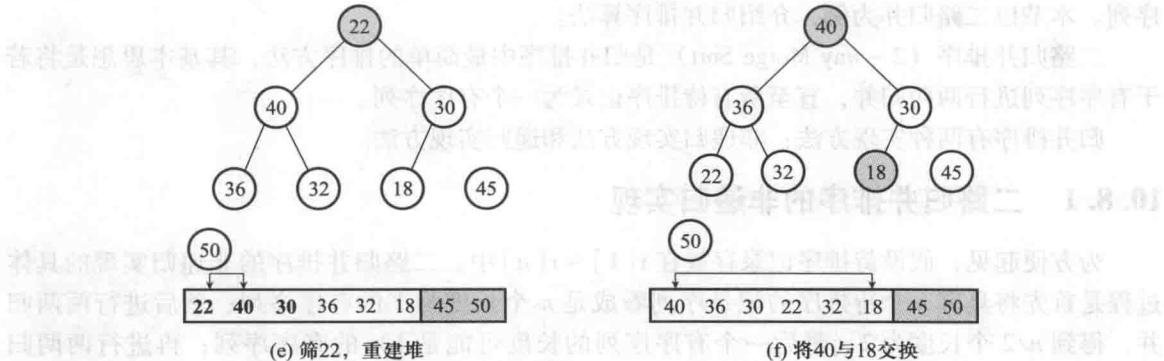


图 10-9 堆排序的过程示例

假设待排序记录存放在  $r[1] \sim r[n]$  中，完整的堆排序算法如算法 10-10 所示。

#### 算法 10-10 堆排序算法 HeapSort

```

void HeapSort(Record r[], int n)
{
 for (i = n/2; i >= 1; i --) // 初始化堆, 从最后一个非终端结点至根结点进行筛选
 Sift(r, i, n);
 for(i = 1; i < n ; i ++) // 重复执行移走堆顶及重建堆的操作
 {
 r[1] ←→ r[n - i + 1]; // 互换堆顶与堆尾
 Sift(r, 1, n - i);
 }
}

```

堆排序的运行时间主要消耗在初始建堆和重建堆时进行的反复筛选上。初始建堆需要  $O(n \log_2 n)$  的时间，第  $i$  趟取堆顶记录重建堆需要用  $O(\log_2 n)$  时间，共需  $n - 1$  趟。因此，总的时间复杂度为  $O(n \log_2 n)$ ，这是堆排序最好、最坏和平均的时间代价。堆排序对原始记录的排序状态并不敏感，这是堆排序相对于快速排序的最大的优点。

在堆排序算法中，只需要一个用来交换的暂存单元。

堆排序是一种不稳定的排序方法。

## 10.8 归并排序

归并排序是通过归并进行排序的一种方法。归并就是将两个或两个以上的有序序列合并成一个有序序列的过程。归并排序的主要思想是将若干有序序列逐步归并，最终归并为一个有序

序列。本节以二路归并为例，介绍归并排序算法。

二路归并排序（2-way Merge Sort）是归并排序中最简单的排序方法，其基本思想是将若干有序序列进行两两归并，直至所有待排序记录为一个有序序列。

归并排序有两种实现方法：非递归实现方法和递归实现方法。

### 10.8.1 二路归并排序的非递归实现

为方便起见，假设待排序记录存放在  $r[1] \sim r[n]$  中。二路归并排序的非递归实现的具体过程是首先将具有  $n$  个待排序的记录序列看成是  $n$  个长度为 1 的有序序列；然后进行两两归并，得到  $n/2$  个长度为 2（最后一个有序序列的长度可能是 1）的有序序列；再进行两两归并，得到  $n/4$  个长度为 4 的有序序列（最后一个有序序列的长度可能小于 4）。如此重复，直至得到一个长度为  $n$  的有序序列。

下面看一个二路归并排序的例子，如图 10-10 所示。

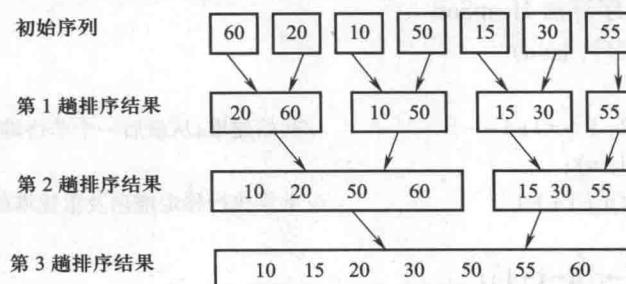


图 10-10 二路归并的排序过程示例

从以上排序过程可以看出，其核心操作是归并操作。设两个相邻的有序序列为  $r[s] \sim r[m]$  和  $r[m+1] \sim r[t]$ ，将这两个有序序列归并成一个有序序列  $r1[s] \sim r1[t]$ 。为此，设三个参数  $i$ 、 $j$  和  $k$  分别指向两个待归并的有序序列和最终有序序列的当前记录。初始时  $i$ 、 $j$  分别指向两个有序序列的第一个记录，即  $i = s$ ， $j = m + 1$ ； $k$  指向存放归并结果的位置，即  $k = s$ 。然后，比较  $i$  和  $j$  所指记录的关键码，取出较小者作为归并结果存入  $k$  所指位置，直至两个有序序列之一的所有记录都取完，再将另一个有序序列的剩余记录顺序送到归并后的有序序列中。

具体的一次归并算法如算法 10-11 所示。

#### 算法 10-11 一次归并算法 Merge

```
void Merge(Record r[], Record r1[], int s, int m, int t)
{
 i = s;
 j = m + 1;
```

```

k = s;
while (i <= m && j <= t)
 if (r[i].key <= r[j].key)
 r1[k++] = r[i++];
 else
 r1[k++] = r[j++];
if (i <= m)
 while (i <= m)
 r1[k++] = r[i++]; //若第一个子序列没处理完,则进行收尾处理
else
 while (j <= t)
 r1[k++] = r[j++]; //若第二个子序列没处理完,则进行收尾处理
}

```

在一趟归并中，除最后一个有序序列外，其他有序序列中记录的个数（称为序列长度）相同，用  $h$  表示。现在的任务是把若干个相邻的长度为  $h$  的有序序列和最后一个长度有可能小于  $h$  的有序序列进行两两归并，把结果存放到  $r1[1] \sim r1[n]$  中。为此，设参数  $i$ ，指向待归并序列的第一个记录。初始时  $i=1$ ，显然归并的步长应是  $2h$ 。在归并过程中，有以下 3 种情况：

- (1) 若  $i \leq n - 2h + 1$ ，则表示待归并的两个相邻有序序列的长度均为  $h$ ，执行一次归并。完成后  $i$  加  $2h$ ，准备进行下一次归并。
- (2) 若  $i < n - h + 1$ ，则表示仍有两个相邻有序序列，一个长度为  $h$ ，另一个长度小于  $h$ 。执行这两个有序序列的归并，完成后退出一趟归并。
- (3) 若  $i \geq n - h + 1$ ，则表明只剩下一个有序序列，直接将该有序序列传送给  $r1$  的相应位置。归并，完成后退出一趟归并。

综上所述，一趟归并排序算法描述如算法 10-12 所示。

### 算法 10-12 一趟归并排序算法 MergePass

```

void MergePass(Record r[], Record r1[], int n, int h)
{
 i = 1;
 while(i <= n - 2h + 1) //待归并记录至少有两个长度为 h 的子序列
 {
 Merge(r, r1, i, i + h - 1, i + 2 * h - 1);
 i += 2 * h;
 }
 if (i < n - h + 1)
 Merge(r, r1, i, i + h - 1, n); //待归并序列中有一个长度小于 h
}

```

```

else
 for (k = i; k <= n; k++)
 r1[k] = r[k]; //待归并序列中只剩一个子序列
}
}

归并排序的非递归算法描述如算法 10-13 所示。
算法 10-13 归并排序的非递归算法 MergeSort1
void MergeSort1(Record r[], int n)
{
 h = 1;
 while(h < n)
 {
 MergePass(r, r1, n, h);
 h = 2 * h;
 MergePass(r1, r, n, h);
 h = 2 * h;
 }
}

```

对二路归并排序算法的时间分析非常直观。将  $r[1] \sim r[n]$  中相邻的长度为  $h$  的有序序列进行两两归并，并把结果存放到  $r1[1] \sim r1[n]$  中，算法的时间复杂性为  $O(n)$ 。由于整个归并排序需要进行  $\log_2 n$  趟，因此算法总的时间复杂性是  $O(n \log_2 n)$ ，这是归并排序算法最好、最坏、平均的时间性能。

由于二路归并排序在归并过程中需要与原始记录序列同样数量的存储空间，以便存放归并结果，因此其空间复杂度为  $O(n)$ 。

归并排序是一种稳定的排序方法。

## 10.8.2 二路归并排序的递归实现

二路归并排序方法也可以用递归的形式描述。即首先将待排序的记录序列分为两个相等的子序列，并分别将这两个子序列用归并方法进行排序；然后调用一次归并算法 Merge；再将这两个有序子序列合并成一个含有全部记录的有序序列。

图 10-11 所示的是一个用递归方法进行归并排序的例子。

归并排序的递归算法描述如算法 10-14 所示。

### 算法 10-14 归并排序的递归算法 MergeSort2

```

void MergeSort2(Record r[], Record r1[], int s, int t)
{
}
```

```

if (s == t)
 r1[s] = r[s];
else
{
 m = (s + t) / 2;
 MergeSort2(r, r2, s, m); //归并排序前半部分
 MergeSort2(r, r2, m + 1, t); //归并排序后半部分
 Merge(r2, r1, s, m, t); //将两个已排序的子序列归并
}
}

```

从第一阶段到最后一阶段，类同于图 10-10 所示的归并排序过程。

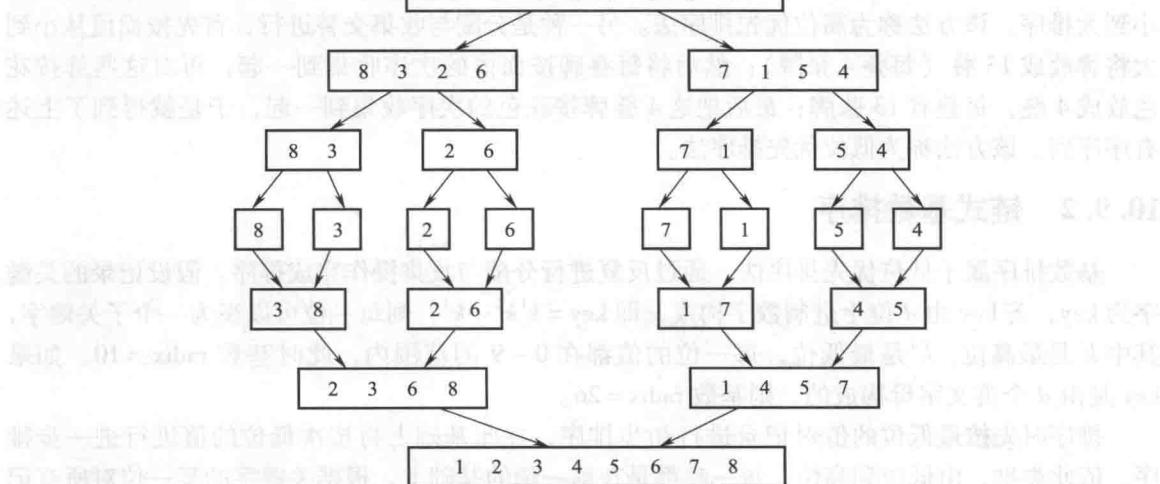


图 10-11 归并排序的递归执行过程

## 10.9 基数排序

基数排序 (Radix Sort) 是和前面介绍的各种排序方法完全不同的一种排序方法。前面介绍的排序方法都需要进行关键字之间的比较和移动记录两种操作，而基数排序不需要进行关键字间的比较。基数排序是一种借助多关键字排序的思想对单逻辑关键字进行排序的方法。

### 10.9.1 多关键字排序

多关键字排序的问题可以通过一个例子来说明。例如，可以将一副扑克牌的排序过程看成由花色和面值两个关键字进行排序的问题。若规定花色和面值的顺序如下：

花色：梅花 < 方块 < 红桃 < 黑桃

面值：A < 2 < 3 < ⋯ < 10 < J < Q < K

并进一步规定花色的优先级高于面值，则一副扑克牌从小到大的顺序为梅花 A，梅花 2，⋯，梅花 K；方块 A，方块 2，⋯，方块 K；红桃 A，红桃 2，⋯，红桃 K；黑桃 A，黑桃 2，⋯，黑桃 K。

具体进行排序时有两种做法。一种是首先按花色分成有序的 4 类，然后按面值对每一类从小到大排序。该方法称为高位优先排序法。另一种是分配与收集交替进行，首先按面值从小到大将牌分成 13 叠（每叠 4 张牌）；然后将每叠牌按面值的次序收集到一起；再对这些牌按花色分成 4 叠，每叠有 13 张牌；最后把这 4 叠牌按花色的次序收集到一起，于是就得到了上述有序序列。该方法称为低位优先排序法。

### 10.9.2 链式基数排序

基数排序属于低位优先排序法，通过反复进行分配与收集操作完成排序。假设记录的关键字为 key，若 key 由  $d$  位十进制数字构成，即  $\text{key} = k^1 k^2 \cdots k^d$ ，则每一位可以视为一个子关键字，其中  $k^1$  是最高位， $k^d$  是最低位，每一位的值都在 0 ~ 9 的范围内，此时基数  $\text{radix} = 10$ 。如果 key 是由  $d$  个英文字母构成的，则基数  $\text{radix} = 26$ 。

排序时先按最低位的值对记录进行初步排序，在此基础上再按次低位的值进行进一步排序。依此类推，由低位到高位，每一趟都是在前一趟的基础上，根据关键字的某一位对所有记录进行排序，直至最高位，这样就完成了基数排序的全过程。

在具体实现时，一般采用链式基数排序。下面通过一个具体的例子来说明链式基数排序的基本过程。

假设对 10 个记录进行排序，每个记录的关键字是 1000 以下的正整数。在此，每个关键字由三位子关键字构成  $k^1 k^2 k^3$ ， $k^1$  代表关键字的百位数， $k^2$  代表关键字的十位数， $k^3$  代表关键字的个位数，基数  $\text{radix} = 10$ 。在进行分配与收集操作时，需要使用链队列。因为队列的数目与基数  $\text{radix}$  相等，所以共设 10 个链队列， $\text{front}[i]$  和  $\text{end}[i]$  分别为队列  $i$  的头指针和尾指针。

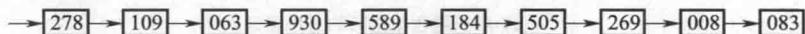
首先将待排序记录存储在一个链表中，如图 10-12 (a) 所示；然后进行如下三趟分配、收集操作。

第一趟分配用最低位子关键字  $k^3$  进行，将所有最低位子关键字相等的记录分配到同一个

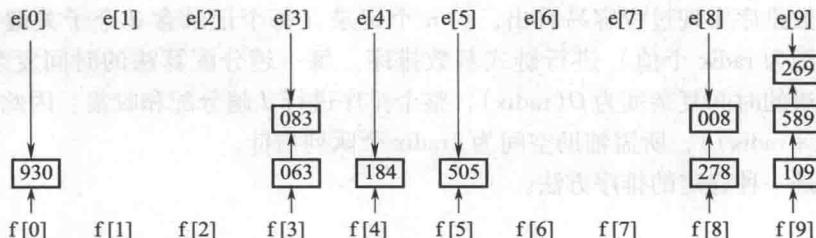
队列中，如图 10-12 (b) 所示，然后进行收集操作。收集时，改变所有非空队列的队尾结点的 next 指针，令其指向下一个非空队列的队头记录，从而将分配到不同队列中的记录重新链成一个链表。第一趟收集完成后，结果如图 10-12 (c) 所示。

第二趟分配用次低位子关键字  $k^2$  进行，将所有次低位子关键字相等的记录分配到同一个队列中，如图 10-12 (d) 所示。第二趟收集完成后，结果如图 10-12 (e) 所示。

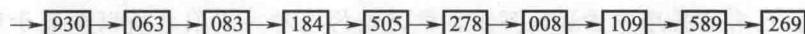
第三趟分配用最高位子关键字  $k^1$  进行，将所有最高位子关键字相等的记录分配到同一个队列中，如图 10-12 (f) 所示。第三趟收集完成后，结果如图 10-12 (g) 所示。至此，整个排序过程结束。



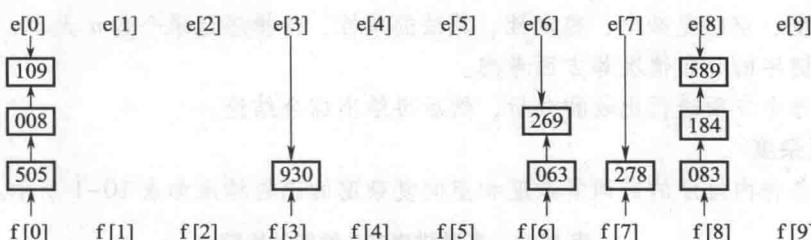
(a) 初始状态



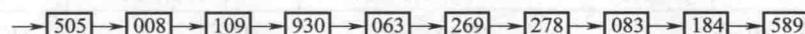
(b) 第一趟分配之后



(c) 第一趟收集之后



(d) 第二胎分配之后



(e) 第二胎收集之后

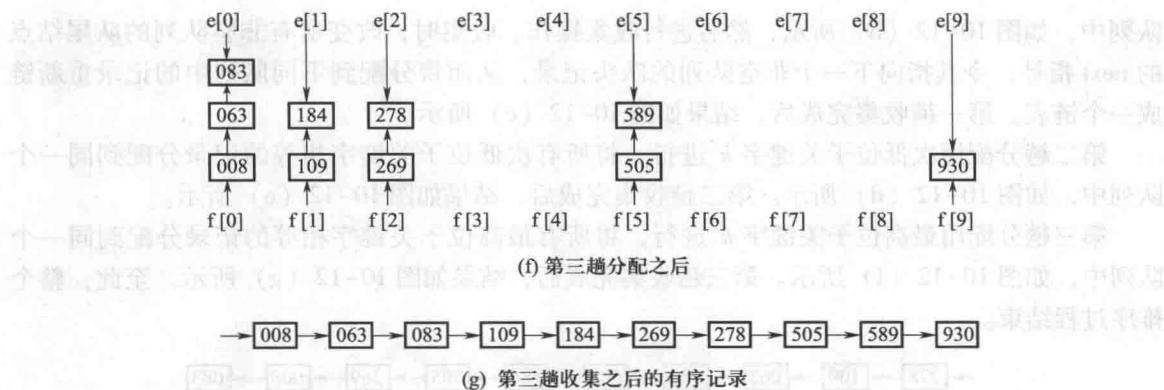


图 10-12 链式基数排序示例

由上述基数排序实现过程容易看出，对  $n$  个记录（每个记录含  $d$  个子关键字，每个子关键字的取值范围为 radix 个值）进行链式基数排序。每一趟分配算法的时间复杂度为  $O(n)$ ，每一趟收集算法的时间复杂度为  $O(\text{radix})$ ，整个排序进行  $d$  趟分配和收集。因此，总的时间复杂度为  $O(d(n + \text{radix}))$ ，所需辅助空间为  $2\text{radix}$  个队列指针。

基数排序是一种稳定的排序方法。

## 本章小结

迄今为止，已有的排序方法远远不止本章讨论的几种。人们之所以热衷于研究排序方法，一方面是由于排序在计算机操作中所处的重要地位；另一方面是由于这些方法各有优缺点，难以得出哪个最好和哪个最坏的结论。因此，排序方法的选用应该根据具体情况而定，一般应该综合时间复杂度、空间复杂度、稳定性、算法简单性、待排序记录个数  $n$  大小、记录本身信息量的大小和关键字的分布情况等方面考虑。

下面先从各个方面进行比较和分析，然后再给出综合结论。

### 1. 时间复杂度

本章所述各种内排序的时间复杂度和空间复杂度的比较结果如表 10-1 所示。

表 10-1 各种排序方法性能的比较

| 排序方法 | 平均情况的时间复杂度                  | 最好情况的时间复杂度   | 最坏情况的时间复杂度 | 空间复杂度  |
|------|-----------------------------|--------------|------------|--------|
| 插入排序 | $O(n^2)$                    | $O(n)$       | $O(n^2)$   | $O(1)$ |
| 希尔排序 | $O(n \log_2 n) \sim O(n^2)$ | $O(n^{1.3})$ | $O(n^2)$   | $O(1)$ |

续表

| 排序方法 | 平均情况的<br>时间复杂度           | 最好情况的<br>时间复杂度           | 最坏情况的<br>时间复杂度           | 空间复杂度                   |
|------|--------------------------|--------------------------|--------------------------|-------------------------|
| 冒泡排序 | $O(n^2)$                 | $O(n)$                   | $O(n^2)$                 | $O(1)$                  |
| 快速排序 | $O(n \log_2 n)$          | $O(n \log_2 n)$          | $O(n^2)$                 | $O(\log_2 n) \sim O(n)$ |
| 选择排序 | $O(n^2)$                 | $O(n^2)$                 | $O(n^2)$                 | $O(1)$                  |
| 堆排序  | $O(n \log_2 n)$          | $O(n \log_2 n)$          | $O(n \log_2 n)$          | $O(1)$                  |
| 归并排序 | $O(n \log_2 n)$          | $O(n \log_2 n)$          | $O(n \log_2 n)$          | $O(n)$                  |
| 基数排序 | $O(d(n + \text{radix}))$ | $O(d(n + \text{radix}))$ | $O(d(n + \text{radix}))$ | $O(\text{radix})$       |

从时间复杂度的平均情况看，有3类排序方法：

(1) 冒泡排序、选择排序和插入排序属于第一类，其时间复杂性为  $O(n^2)$ 。其中，插入排序方法最常用，特别是对于已按关键字基本有序的记录序列。

(2) 堆排序、快速排序和归并排序属于第二类，时间复杂性为  $O(n \log_2 n)$ 。其中，快速排序目前被认为最快的一种排序方法，在待排序记录个数较多的情况下，归并排序比堆排序更快。

(3) 希尔排序介于  $O(n^2)$  和  $O(n \log_2 n)$  之间。

从最好情况看，冒泡排序和插入排序的时间复杂性最好，为  $O(n)$ ；其他排序算法的最好情况与平均情况相同。

从最坏情况看，快速排序的时间复杂性为  $O(n^2)$ 。插入排序和冒泡排序虽然与平均情况相同，但是因为系数大约增加一倍，所以运行速度将降低一半。最坏情况对直接选择排序、堆排序和归并排序影响不大。

由此可知，在最好情况下插入排序和冒泡排序最快；在平均情况下快速排序最快；在最坏情况下堆排序和归并排序最快。

## 2. 空间复杂度

从空间复杂度看，所有排序方法分为3类。归并排序单独属于第一类，其空间复杂度为  $O(n)$ ；快速排序单独属于第二类，其空间复杂度为  $O(\log_2 n) \sim O(n)$ ；其他排序方法归为第二类，其空间复杂度为  $O(1)$ 。

## 3. 稳定性

从稳定性来看，排序方法可分为两类，一类是稳定的，包括冒泡排序、插入排序、归并排序和基数排序；另一类是不稳定的，包括选择排序、希尔排序、快速排序和堆排序。

## 4. 算法简单性

从简单性来看，排序算法可分为两类，一类是简单算法，包括冒泡排序、选择排序和插入

排序；另一类是改进算法，包括希尔排序、堆排序、快速排序和归并排序，这些算法都比较复杂。

### 5. 待排序的记录个数 $n$ 的大小

从待排序的记录个数  $n$  的大小看，因为  $n$  越小， $O(n^2)$  同  $O(n \log_2 n)$  的差距越小，所以  $n$  越小，采用简单排序方法越合适； $n$  越大，采用改进的排序方法越合适。

### 6. 记录本身信息量的大小

从记录本身信息量的大小看，记录本身信息量越大，表明占用的存储空间就越多，移动记录所花费的时间就越多，这对记录的移动次数较多的算法不利。

### 7. 关键字的分布情况

当待排序记录序列为正序时，插入排序和冒泡排序能达到  $O(n)$  的时间复杂度。对于快速排序而言，这是最坏的情况，此时的时间性能蜕化为  $O(n^2)$ 。选择排序、堆排序和归并排序的时间性能不随记录序列中关键字的分布而改变。

综合考虑以上 7 个方面，可得出下面的大致结论，供读者参考。

- (1) 当待排序记录个数  $n$  较大，关键字分布较随机，且对稳定性不作要求时，采用快速排序为宜。
- (2) 当待排序记录个数  $n$  较大，内存空间允许，且要求排序稳定时，采用归并排序为宜。
- (3) 当待排序记录个数  $n$  较大，而只要找出最小的前几个记录，采用堆排序或选择排序为宜。
- (4) 当待排序记录个数  $n$  较小（如小于 100）时，记录已基本有序，且要求稳定时，采用插入排序。例如，在一个已排序序列上略作修改，若改动不大，最好用插入排序把它恢复为有序序列。
- (5) 当待排序记录个数  $n$  较小，记录所含数据项较多，所占存储空间较大时，采用选择排序为宜。
- (6) 由于快速排序和归并排序在待排序记录个数  $n$  值较小时的性能不如插入排序，因此在实际应用时，可将它们和插入排序混合使用。例如，在快速排序中划分的子序列的长度小于某个值时，转而调用插入排序；或者对待排序记录序列先逐段进行插入排序，然后再利用归并操作进行两两归并，直至整个序列有序。

## 习题 10

10.1 什么是内排序？什么是外排序？评价排序方法的主要性能指标有哪些？

10.2 排序方法主要有哪些？试比较他们各自的性能。

10.3 平均时间复杂性为  $O(n \log_2 n)$  的排序算法有哪些？试分析他们的稳定性？

- 10.4 选择排序稳定吗？若不稳定，请举例说明？
- 10.5 堆排序稳定吗？若不稳定，请举例说明？
- 10.6 快堆排序稳定吗？若不稳定，请举例说明？
- 10.7 给定数据序列{12, 5, 9, 20, 6, 31, 26}，对该序列进行排序，分别写出冒泡排序、选择排序、插入排序、希尔排序、快速排序、堆排序、归并排序和基数排序每趟的排序结果。
- 10.8 已知 $(k_1, k_2, \dots, k_n)$ 是堆，试编写算法将 $(k_1, k_2, \dots, k_n, k_{n+1})$ 调整为堆。
- 10.9 设有整型数组 $x$ ，试编写算法将负数集中在数组 $x$ 的一端，正数集中在数组 $x$ 的另一端。要求算法时间复杂性为 $O(n)$ 。
- 10.10 给定有序序列 $A[m]$ 和有序序列 $B[n]$ ，试编写算法将它们归并为一个有序序列 $C[m+n]$ 。

## 上机实验题 10

**实验题** 编写程序，实现插入排序、快速排序、堆排序、归并排序。

**基本要求：**

- (1) 利用随机函数产生 1 000 个随机整数，作为待排序序列。
- (2) 要求程序模块化结构，一个函数实现一种排序算法。
- (3) 记录各种算法的排序过程中数据的移动次数，并进行比较。

## 附录

### 附录 A 数据结构试题

#### A.1 数据结构试题 A

##### 一、简要回答下列问题（共 64 分）

1. 算法与程序有何区别和联系？(6 分)
2. 树的存储方法主要有哪些？任画一个树，举例说明具体存储结构。(6 分)
3. 设有序表的长度为 10，用二分查找方法进行查找，试计算查找成功情况下的平均查找长度 (6 分)
4. 图的遍历方法主要有哪些？任画一个图举例具体说明。(6 分)
5. 画出广义表  $D=((\ ),x,(a,(b,c)))$  的存储结构，并写出广义表类型定义。(6 分)
6. 分别画出一个 B 树和 B+ 树的例子，并指出它们之间的区别。(6 分)
7. 你知道有哪些排序算法？试比较各种排序算法的性能。(8 分)
8. 设一组关键字为  $\{7, 15, 20, 31, 48, 53, 64, 76, 82, 99\}$ ，Hash 函数  $H(key) = key \% 11$ ，Hash 表表长  $m = 11$ ，用线性探测法解决冲突。试构造 Hash 表，并分别计算查找成功和查找失败情况下的平均查找长度。(8 分)
9. 简述利用 Dijkstra 算法求解从某顶点到其余各顶点最短路径的步骤。(12 分)

##### 二、算法设计（共 36 分）

1. 试编写归并排序算法。(12 分)
2. 试编写一个算法将线性表  $L$  中的数据建立一棵二叉排序树。(12 分)
3. 设单链表  $L$  中的结点按 data 域数值递减排列，试设计一个算法将  $L$  中的结点按 data 域数值递增排列，要求算法的时间复杂性为  $O(n)$ 。(12 分)

#### A.2 数据结构试题 B

##### 一、填空题（每空 2 分，共 20 分）

1. 当 \_\_\_\_\_ 时，快速排序算法的时间复杂性最差。
2. 对于具有 300 个记录的文件，采用分块索引查找法查找，其中用二分查找法查找索引

表, 用顺序查找法查找块内元素。假定每块长度均为 30 个元素, 则平均查找长度为 \_\_\_\_\_。

3. 设某二叉树的前序和中序序列均为 ABCDE, 则它的后序序列是 \_\_\_\_\_。

4. 假设以一维数组  $s[n(n+1)/2]$  作为  $n$  阶对称矩阵  $A$  的存储空间, 以行序为主序存储  $A$  的下三角, 则元素  $A[5][6]$  的值存储在  $s[_____]$  中。

5. 若一棵树中度为 1 的结点有  $N_1$  个, 度为 2 结点有  $N_2$  个, …, 度为  $m$  的结点有  $N_m$  个, 则该树的叶结点有 \_\_\_\_\_ 个。

6. 设循环队列的元素存放在一维数组  $Q[0..30]$  中, front 指向队头元素的前一个位置, rear 指向队尾元素。若  $front = 25$ ,  $rear = 5$ , 则该队列中的元素个数为 \_\_\_\_\_。

7. 图的遍历方法主要是 \_\_\_\_\_。

8. 设源串  $S = "bcdecdcb"$ , 模式串  $P = "cdcb"$ , 按 KMP 算法进行模式匹配, 当 " $S_2S_3S_4 = P_1P_2P_3$ ", 而  $S_5 \neq P_4$  时,  $S_5$  应与 \_\_\_\_\_ 比较。

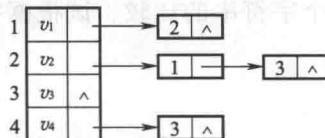
9. 设有序列 {12, 34, 19, 23, 8, 56}, 试建立表长为 7 的 Hash 表。Hash 函数为  $H(key) = key \% 7$ , 用线性探测法解决冲突, 则 56 冲突 \_\_\_\_\_ 次。

10. 求解图的最小生成树的算法有两个, 分别是 \_\_\_\_\_。

## 二、根据要求解答下列问题 (共 44 分)

1. 画出广义表  $D = (x, (), (x, (y, z)))$  的存储结构。(5 分)

2. 根据下图所示的邻接表画出邻接矩阵。(5 分)



3. 请给出堆排序不稳定的例子。(6 分)

4. 设输入下列关键字序列 {12, 34, 56, 8, 5, 18, 15}, 试建立一棵平衡的二叉排序树并写出步骤。(6 分)

5. 分别画出一个  $B$  树和  $B +$  树的例子, 并指出它们之间的区别。(6 分)

6. 举例说明栈和队列的应用。(6 分)

7. 在“数据结构”课程中, 你学习了哪些算法? 请至少列举 20 个算法名称。(10 分)

## 三、算法设计题 (共 36 分)

1. 试编写对顺序表  $L$  进行归并排序的算法。(12 分)

2. 已知二叉树  $T$  的结点结构为

|      |      |       |     |
|------|------|-------|-----|
| left | data | right | bal |
|------|------|-------|-----|

其中,  $\text{bal}$  存储结点的平衡因子 ( $\text{bal} = \text{左子树高度} - \text{右子树高度}$ ), 试编写算法求树  $T$  中各结点的平衡因子。(12 分)

3. 设采用邻接表作为有向图的存储结构, 试编写算法计算有向图中每个顶点的出度和入度。(12 分)

### A.3 数据结构试题 C

#### 一、填空题 (每空 2 分, 共 20 分)

1. 提高程序可读性的措施是: \_\_\_\_\_。

2. 设  $n > 0$ , 且有如下程序段:

```
int i;
i = n;
while (i > 0)
 i = i/10;
```

则该程序段的时间复杂性为 \_\_\_\_\_。

3. 稀疏矩阵的压缩存储方法是 \_\_\_\_\_。

4. 以数组  $Q[0..m]$  存放循环队列中的元素, 变量  $\text{rear}$  和  $\text{qulen}$  分别指示循环队列中队尾元素的实际位置和当前队列中元素的个数, 则队列第一个元素的实际位置是 \_\_\_\_\_。

5. 将序列 {50, 38, 66, 98, 77, 13, 28, 50} 建立一个堆, 该堆是 \_\_\_\_\_。

6. 下列函数的功能是实现两个字符串的比较, 试根据字符串比较运算的定义, 完善该函数。

```
int strcmp(char s[], char t[])
{
 int i;
 for (i = 0; s[i] && t[i]; i++)
 if (s[i] != t[i])
 _____;
 _____;
}
```

7. 下列函数的功能是求带头结点的单链表的表长, 试完善该函数。

```
int count(LinkList head)
{
 _____;
 length = 0;
 while (p != NULL)
```

```

{
 length++;
 _____;
}
_____;
}

```

## 二、简要回答下列问题（共 44 分）

- 分别说明 Huffman 算法、Dijkstra 算法、Prim 算法和 Kruskal 算法的功能。(8 分)
- 举例说明选择排序是不稳定的。(6 分)
- 设有一个广义表  $L = (a, (), (x, (y, z)), (a, b))$ , 试画出它的存储结构。(6 分)
- 图的存储方法主要有哪些? 试举例说明它们的具体存储结构。(8 分)
- 设一组关键字为  $\{7, 15, 20, 31, 48, 53, 64, 76, 82, 99\}$ , Hash 函数  $H(key) = key \% 11$ , Hash 表表长  $m = 11$ , 用线性探测法解决冲突。试构造 Hash 表, 并计算查找成功情况下的平均查找长度。(8 分)
- 时间复杂性为  $O(n \log_2 n)$  的排序方法有哪些? 任选其中一种方法举例说明其排序过程。(8 分)

## 三、算法设计题（共 36 分）

- 编写算法从键盘读入一组整数, 以 9999 作为结束标志, 将这些数据建立一棵二叉排序树。(12 分)
- 任意选择一个算法进行改进, 试给出改进前、后的算法描述, 并说明算法改进前、后的性能区别。(12 分)
- 试编写图的广度优先搜索(遍历)算法。(12 分)

## A.4 数据结构试题 D

### 一、填空题（每题 2 分, 共 20 分）

- 算法的健壮性是指\_\_\_\_\_。
- 对一个线性表分别进行遍历和逆置运算, 其最好的时间复杂度分别为\_\_\_\_\_和\_\_\_\_\_。
- $n$  个数入栈, 所有可能的出栈序列共有\_\_\_\_\_种。
- 下面程序段的时间复杂度为\_\_\_\_\_ ( $n > 1$ )。

```

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

```
- 若某二叉树有 20 个叶子结点, 有 30 个结点仅有一个孩子, 则该二叉树的总的结点数

是

6. 若以{4, 5, 6, 7, 8}作为叶子结点的权值构造 Huffman 树，则该 Huffman 树的根结点权值为\_\_\_\_\_。

7. 线索二叉树的左线索指向其\_\_\_\_\_，右线索指向其\_\_\_\_\_。

8. 若含  $n$  个顶点的图形成一个环，则它的生成树可能有\_\_\_\_\_种。

9. 对于具有 300 个记录的文件，采用分块索引查找法查找，其中用二分查找法查找索引表，用顺序查找法查找块内元素。假定每块长度均为 30 个元素，则平均查找长度为\_\_\_\_\_。

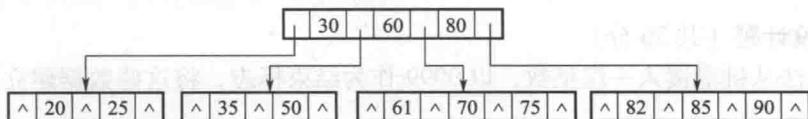
10. 有一个有序表为{1, 3, 9, 12, 32, 41, 45, 62, 75, 77, 82, 95, 100}，当二分查找值为 82 的结点时，经\_\_\_\_\_次比较后查找成功。

二、设一组记录的关键字为{4, 5, 7, 2, 1, 3, 6}，请回答相关问题：

- (1) 按表中元素的顺序依次插入一棵初始为空的二叉排序树，画出插入完成后的二叉排序树，并求出在等概率情况下查找成功的平均查找长度。(3分)

(2) 按表中元素的顺序进行插入，生成一棵 AVL 树，画出该树。并求出在等概率情况下查找成功的平均查找长度。(4分)

三、设有一个 4 阶 B 树, 请回答相关问题:



- (1)  $B +$  树和  $B$  树的主要区别是什么? (4 分)

- (2) 插入关键字 87, 画出插入调整后树的形状。(4 分)

四、已知记录关键字集合为{53, 17, 19, 61, 98, 75, 79, 63, 46, 49}，要求散列到地址区间[0,10]中，散列函数选用除留余数法 ( $\text{mod } 11$ )。请回答相关问题：

- (1) 画出形成的散列表 (若产生冲突, 用开放定址的线性探测再散列法解决)。(4 分)  
(2) 计算在等概率情况下查找成功时的平均查找长度。(2 分)

五、有7个顶点  $(v_1, v_2, v_3, v_4, v_5, v_6, v_7)$  的有向图的邻接矩阵如下所示, 请回答下列问题。

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| ∞ | 2 | 5 | 3 | ∞ | ∞ | ∞ |
| ∞ | ∞ | 2 | ∞ | ∞ | 8 | ∞ |
| ∞ | ∞ | ∞ | 1 | 3 | 5 | ∞ |
| ∞ | ∞ | ∞ | ∞ | 5 | ∞ | ∞ |
| ∞ | ∞ | ∞ | ∞ | ∞ | 3 | 9 |
| ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | 5 |
| ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |

- (1) 画出该有向图。(2 分)
- (2) 画出邻接表。(2 分)
- (3) 写出从  $v_1$  出发的深度优先遍历和广度优先遍历序列。(4 分)
- (4) 将图看成 AOE 网, 画出关键活动及相应的有向边, 写出关键路径的长度。(4 分)

## 六、设计算法。(24 分)

1. 已知一棵树  $T$  用二叉链表表示, 其结点形式如下所示, 试编写一算法求树  $T$  中各结点的度数。(12 分)

| left | data | right | degree |
|------|------|-------|--------|
|------|------|-------|--------|

(1) 写出相应的结构定义。

(2) 编写算法。

2. 判断有向图是否存在回路。若存在返回 1, 否则返回 0。(12 分)

(1) 写出相应的结构定义。

(2) 编写算法。

## 七、阅读下列函数, 回答相关问题:

```
int arrange (int a[], int l, int h, int x)
{ // l 和 h 分别为数据区的下界和上界
 int i, j, t;
 i = l;
 j = h;
 while (i < j)
 {
 while (i < j && a[j] >= x)
 j -- ;
 while (i < j && a[i] < x)
 i ++ ;
 if (i < j)
 {
 t = a[j];
 a[j] = a[i];
 a[i] = t;
 }
 }
 if (a[i] < x)
```

```

 return i;
 else
 return i - 1;
}

```

(1) 写出该函数的功能。(4分)

(2) 写一个调用上述函数实现下列功能的算法：对一整型数组  $b[n]$  中的元素重新排列，将所有负数均调整到数组的低下标端；将所有正数均调整到数组的高下标端；若有零值，则置于两者之间，并返回数组中零元素的个数。(6分)

#### 八、数组 a 存储了 n 个整数，请回答相关问题：

(1) 请完善对数组 a 进行堆排序的程序。(6分)

```
void HeapAdjust (int a[], int h, int s)
```

```

{
 rc = a[h];
 for (j = _____; j <= s; j *= 2)
 {
 if ((j < s) && (a[j] < a[j + 1])) ++j;
 if (! (rc <= a[j]))
 break;
 _____;
 h = j;
 }
 _____;
}
```

```
void HeapSort (int a[], int n) //对 a[1], a[2], ..., a[n]进行堆排序
```

```

{
 for (i = _____; i > 0; --i)
 HeapAdjust (a, i, n);
 for (i = _____; i > 1; --i)
 {
 t = a[1];
 a[1] = a[i];
 a[i] = t;
 _____;
 }
}
```

(2) 上面程序建成的堆是大顶堆还是小顶堆? (2 分)

(3) 对  $n$  个元素进行初始建堆的过程中, 最多进行\_\_\_\_\_次数据比较。(2 分)

(4) 堆排序稳定吗? 请举例说明。(3 分)

## A.5 数据结构试题 E

### 一、判断题 (每题 1 分, 共 10 分)

1. 线性表的逻辑顺序与存储顺序总是一致的。 ( )
2. 线索二叉树中, 任一结点均有指向前驱和后继的线索。 ( )
3. 栈、队列、数组和串都是线性结构。 ( )
4. 建立双向链表的目的是为了方便插入和删除。 ( )
5. 图的生成树是该图的极小连通子图。 ( )
6. 树的后序遍历序列与其对应二叉树的后序遍历序列相同。 ( )
7. 二叉排序树的充要条件是任一结点的值均大于其左孩子的值, 小于其右孩子的值。 ( )
8. 如果某排序算法是不稳定的, 则该排序算法没有实用价值。 ( )
9. 稀疏矩阵压缩存储后, 就会失去随机存取功能。 ( )
10. 归并排序可以用递归和非递归两种方法实现。 ( )

### 二、填空题 (共 20 分, 每空 2 分)

1. 设源串  $s = "bababaaba"$ , 模式串  $p = "babaa"$ , 按照 KMP 算法进行模式匹配, 当 " $s_1s_2s_3s_4$ " = " $p_1p_2p_3p_4$ ", 而  $s_5 \neq p_5$  时,  $s_5$  应与\_\_\_\_\_比较。

2. 下列算法的时间复杂性是\_\_\_\_\_。

```
int fun(int n)
{
 int i = 1, s = 1;
 while (s < n)
 s += ++i;
 return i;
```

3. 表达式  $3/(x+2)-8$  所对应的后缀表达式是\_\_\_\_\_。

4. 假设以一维数组  $s[n(n+1)/2]$  作为  $n$  阶对称矩阵  $A$  的存储空间, 以行序为主序存储  $A$  的下三角, 则元素  $A[5][8]$  的值存储在  $s[_____]$  中。

5. 下列函数的功能是实现两个字符串的比较, 试根据字符串比较运算的定义, 完善该函数。

```

int strcmp(char s[], char t[])
{
 int i;
 for (i=0; s[i]==t[i]; i++)
 if (s[i] != t[i])
 _____;
 _____;
}

```

6. 最坏情况下，堆排序的时间复杂性为\_\_\_\_\_。
7. 具有 100 个结点的完全二叉树，其叶子结点数为\_\_\_\_\_。
8. 利用\_\_\_\_\_算法可以判断一个有向图是否存在回路。
9. 对于具有 100 个记录的文件，利用分块索引方法查找，用顺序查找法查找索引表和块内元素，假定每块长度均为 10 个元素，则平均查找长度为\_\_\_\_\_。

### 三、简要回答下列问题（共 30 分）

1. 评价一个排序算法好坏的标准是什么？你知道有哪些排序算法？试比较它们各自的性能。（10 分）
2. 图的存储方法主要有哪些？试举例具体说明图的存储结构；并说明 Prim, Kruskal, Dijkstra, Floyd 算法的功能。（10 分）
3. 已知一组关键字为 {26, 36, 41, 38, 44, 15, 68, 12, 06, 51, 25}，Hash 函数定义为  $H(key) = key \% 13$ 。用拉链法解决冲突，建立 Hash 表，分别计算查找成功和查找失败情况下的平均查找长度。（10 分）

### 四、算法设计题（共 40 分）

1. 试编写折半查找算法。（10 分）
2. 设有整型数组  $x$ ，试编写算法将负数集中在数组  $x$  的一端，正数集中在数组  $x$  的另一端。（10 分）
3. 设采用邻接矩阵存储有向图，试编写算法计算有向图中每个结点的入度和出度，入度和出度分别存入数组  $in$  和  $out$  中。（10 分）
4. 设采用二叉链表存储二叉排序树，试编写算法在二叉排序树中求任意两个不同结点的共同祖先。（10 分）

## A.6 数据结构试题 F

### 一、判断题（每题 1 分，共 10 分）

1. 中序遍历二叉排序树得到的是有序序列。

( )

2. 算法的健壮性就是指算法具有容错能力。 (1)
3. 顺序表可以随机存取数据。 ( )
4. 二分查找方法适用于线性表。 ( )
5. 在建立平衡二叉树过程中, 当插入新结点时该树失去平衡, 则要进行旋转操作。 ( )
6. 用冒泡排序法对  $n$  个数据排序必须进行  $(n-1)$  趟。 ( )
7. 对  $n$  个已排好的数据进行插入排序, 需比较  $(n-1)$  次。 ( )
8. 求解关键路径是对 AOV 网进行的操作。 ( )
9. 稀疏矩阵的压缩存储思想是只存储非零元素。 ( )
10.  $B +$  树中所有非叶子结点仅起索引作用。 ( )

## 二、填空题 (共 20 分, 每题 2 分)

1. 对于具有 300 个记录的文件, 利用分块索引方法查找, 用二分查找法查找索引表, 假定每块长度均为 30 个元素, 则平均查找长度为 \_\_\_\_\_。
2. 下列算法的时间复杂性是 \_\_\_\_\_。  

```
int fun(int n)
{
 int i = 0;
 while(n > 0)
 {
 n = n/2;
 i++;
 }
 return i;
}
```
3. 图的遍历方法主要有 \_\_\_\_\_。
4. 求解图的最小生成树的算法有两个, 分别是 \_\_\_\_\_。
5. 假设以一维数组  $s[n(n+1)/2]$  作为  $n$  阶对称矩阵  $A$  的存储空间, 以行序为主序存储  $A$  的下三角, 则元素  $A[4][6]$  的值存储在  $S[_____]$  中。
6. 用 A、B、C、D、E、F 构造 Huffman 树, 其权值分别为 10、30、8、12、15、25, 则该 Huffman 树根结点的权为 \_\_\_\_\_。
7. 设栈  $S$  和队列  $Q$  的初始状态均为空, 元素 abcdefg 依次进入栈  $S$ 。若每个元素出栈后立即进入队列  $Q$ , 且 7 个元素出队的顺序是 bdfeag, 则栈  $S$  的容量至少是 \_\_\_\_\_。
8. Dijkstra 算法的功能是 \_\_\_\_\_。
9. 归并排序的时间复杂性为 \_\_\_\_\_。

10. 设模式串  $p = "babac"$ , 按照 KMP 算法进行串的模式匹配, 其中 next 数组的值分别为\_\_\_\_\_。

### 三、简要回答下列问题 (共 30 分)

1. 树的存储方法有哪些? 试举例说明 (8 分)
2. 输入序列 {36, 30, 18, 40, 32, 45, 23, 50}。试建立一个大顶堆。(6 分)
3. 关键字序列为 {36, 7, 40, 11, 16, 81, 22, 8, 14}, Hash 函数为  $H(key) = key \% 11$ , 用链地址法处理冲突。试建立 Hash 表, 并计算查找成功情况下的平均查找长度。(8 分)
4. 已知图  $G = (V, E)$ ,  $V = \{A, B, C, D, F\}$ ,  $E = (\langle A, B, 2 \rangle, \langle B, C, 7 \rangle, \langle C, D, 2 \rangle, \langle D, F, 5 \rangle, \langle F, A, 3 \rangle, \langle A, D, 6 \rangle, \langle D, B, 9 \rangle)$ 。(8 分)

(1) 画出图  $G$ ;

(2) 画出它的邻接矩阵;

(3) 画出它的邻接表。

### 四、算法设计题 (40 分)

1. 编写一个算法, 由键盘输入  $n$  个整数, 将其建立一个单链表。(10 分)
2. 试编写快速排序算法。(10 分)
3. 设采用二叉链表存储二叉树, 试编写算法在二叉树中查找 data 域为  $x$  的结点。(10 分)
4. 设用邻接表存储有向图, 试编写对有向图进行拓扑排序的算法。(10 分)

## 附录 B 数据结构课程设计题

要求:

(1) 每生做两题, 必做题 + 选做题。选做题安排如下: 学号尾数为 1、6 的学生做第 1 题; 学号尾数为 2、7 的学生做第 2 题; 学号尾数为 3、8 的学生做第 3 题; 学号尾数为 4、9 的学生做第 4 题; 学号尾数为 5、0 的学生做第 5 题。

(2) 学生需提交课程设计报告和软件。报告内容包括题目, 算法思想描述, 程序结构, 测试结果、收获与体会等。

### 一、必做题

编程实现希尔、快速、堆排序、归并排序算法。要求首先随机产生 10 000 个数据存入磁盘文件, 然后读入数据文件, 分别采用不同的排序方法进行排序并将结果存入文件中。

### 二、选做题

#### 1. 压缩软件

建立一个文本文件 A (可以是 C/C++ 源程序), 统计该文件中各字符频率。首先对各字符进行 Huffman 编码, 将该文件翻译成 Huffman 编码文件 B; 然后将 Huffman 编码文件译码成

文件 C，并对文件 A 与 C 进行比较。

## 2. 链表的维护与文件形式的保存

用链表结构的有序表表示某商场家电部的库存模型。当有提货或进货时，需要对该链表及时进行维护。每个工作日结束之后，将该链表中的数据以文件形式保存。每日开始营业之前，需将以文件形式保存的数据恢复成链表结构的有序表。

链表结点的数据域包括家电名称、品牌、单价和数量，以单价的升序体现链表的有序性。程序功能包括创建表、营业开始（读入文件恢复链表数据）、进货（插入）、提货（更新或删除）、查询信息、更新信息和营业结束（链表数据存入文件）等。

## 3. 利用 Hash 技术和二分查找技术统计某个 C 源程序中的关键字出现的频度

扫描一个 C 源程序，利用两种方法统计该源程序中的关键字出现的频度，并比较各自查找的比较次数。

(1) 用 Hash 表存储源程序中出现的关键字，利用 Hash 查找技术统计该程序中的关键字出现的频度。用线性探测法解决 Hash 冲突。设 Hash 函数为

$$\text{Hash}(\text{key}) = [(\text{key 的第一个字母序号}) \times 100 + (\text{key 的最后一个字母序号})] \% 41$$

(2) 用顺序表存储源程序中出现的关键字，利用二分查找技术统计该程序中的关键字出现的频度。

## 4. 管道铺设施工的最佳方案选择

$n(n > 10)$  个居民区之间需要铺设煤气管道。假设任意两个居民区之间都可以铺设煤气管道，但代价不同。要求事先将任意两个居民区之间铺设煤气管道的代价存入磁盘文件中。设计一个最佳方案使得这  $N$  个居民区之间铺设煤气管道所需代价最小，并将结果以图形方式在屏幕上输出。

## 5. 求解最短路径

设有  $n(n > 10)$  个城市之间的交通图。假设任意两个城市之间不一定有直接交通线路，权表示乘车时间。要求事先将交通图信息将存入磁盘文件中，求从某城市出发到其他城市的最少乘车时间和乘车路线。要求将结果以图形方式在屏幕上输出。