

北京科海培训中心

李春葆 编著

数据结构 (C语言篇)

习题与解析



清华大学出版社

12-44

北京科海培训中心

数据结构习题与解析

(C 语言篇)

李春葆 编著

清华大学出版社

(京)新登字 158 号

内 容 提 要

本书根据数据结构课程的教学大纲的要求,提供了作者多年教学中积累、收集与验证的有关数据结构的基本内容及相关题解。全书共分 13 章,每章先给出内容概述,然后给出该章的题解,题解分为基本题和习题解析两部分,前者由选择题和填空题两种题型组成,直接给出答案;后者对每个习题的解答给出了完整的过程。

本书概念清晰,习题覆盖面广,既收集了较容易的题目,也收集了难度适中和较高难度的题目,如一些高校计算机专业招收硕士研究生的《数据结构》试题。

本书可作为计算机专业本、专科学生的学习参考书,也是报考计算机专业硕士研究生的考生必读参考书,还适用于自学考试的读者和计算机等级(三级或四级)考试者研习。

版权所有,盗版必究。

本书封面贴有清华大学出版社激光防伪标签,无标签者不得进入各书店。

书 名: 数据结构习题与解析(C 语言篇)

作 者: 李春葆

出版者: 清华大学出版社(北京清华大学校内,邮编 100084)

<http://www.tup.tsinghua.edu.cn>

印刷者: 北京门头沟胶印厂

发 行: 新华书店总店北京科技发行所

开 本: 787×1092 1/16 印张: 22 字数: 535 千字

版 次: 2000 年 1 月第 1 版 2000 年 1 月第 1 次印刷

印 数: 00001~5000

书 号: ISBN 7-302-03786-8/TP·2200

定 价: 28.00 元

前 言

计算机编程中加工处理的对象是数据,而数据具有一定的组织结构,所以学习编写计算机程序仅仅了解计算机语言是不够的,还必须掌握数据组织、存储和运算的一般方法,这便是数据结构课程中所学习和研究的内容,也是我们编写计算机程序的重要基础,由于它对计算机学科起到承前启后的作用,因此本课程被列为计算机等相关专业最重要的专业基础课程。

由于数据结构的原理和算法较抽象,而该课程一般在本科低年级开设,对于具有一些计算机程序设计知识的初学者,理解和掌握其中的原理就困难了。在解答数据结构习题时,往往感到无从下手,作者在多年的教学中感受颇深,本人通过长期的实践,收集与整理编写了这本《数据结构习题与解析》一书,其目的是:通过对习题的解答,使学生充分掌握数据结构的原理以及求解数据结构问题的思路与方法,深化对基本概念的理解,提高分析与解决问题的能力。

本书遵循数据结构课程的教学大纲的要求,从内容上分为13章:第1章是概述,讨论数据结构的基本概念及相关题解;第2章是顺序表,讨论基本顺序表即向量、栈和队列的基本内容及相关题解;第3章是链表,讨论各种链表的基本内容及相关题解;第4章是串,讨论串的基本内容及相关题解;第5章是数组和稀疏矩阵,讨论数组和稀疏矩阵的基本内容及相关题解;第6章是递归,讨论基本递归设计方法及相关题解;第7章是广义表,讨论广义表的基本内容及相关题解;第8章是树形结构,讨论树和二叉树的基本内容及相关题解;第9章是图,讨论图的基本内容及相关题解;第10章是查找,讨论基本查找方法及相关题解;第11章是内排序,讨论基本内排序方法及相关题解;第12章是文件,讨论基本文件组织结构及相关题解;第13章是外排序,讨论基本外排序方法及相关题解。

每章的内容介绍与习题相关,精选了该章所讨论的数据结构的概念、存储方式和基本运算。每章的题解分为基本题和习题解析两部分,前者由选择题和填空题两种题型组成,由于这部分习题是一些基本概念方面的题目,书中只给出答案;习题解析是对每个习题的解答并给出求解思路和解答的完整的过程,这部分内容中包含一些难度较大的习题,也包含一些高校计算机专业招收硕士研究生的数据结构试题,这部分习题前面加有“*”号。书中介绍的程序在Turbo C系统中调试通过。

本书习题覆盖面广,既收集了较容易的题目,也收集了难度适中和较高难度的题目。因此,本书不仅可以作为计算机专业本、专科生数据结构课程的学习参考书,也是报考计算机专业硕士研究生的考生必读复习书,同时适合于数据结构课程自学者和计算机等级(三级或四级)考试者研习。

在编写本书时,作者力求从方法上提高解题的能力,例如,递归问题是学生较难理解的知识点,但在计算机专业知识中又经常遇到的问题,为此,作者专门编写了递归一章,较深入地分析了递归的执行过程,提出了从递归模型到递归设计的步骤。在其他几章中,也采用了类似的解题方法。

由于习题较多,解答上可能存在不够完整和疏漏之处,内容编排上也可能存在不够合理的地方,敬请广大读者批评指正。

作 者

1999.8

目 录

第 1 章 概述	(1)
1.1 基本概念	(1)
1.1.1 数据结构	(1)
1.1.2 存储方式	(2)
1.1.3 算法及其评价	(3)
1.2 基本题	(5)
1.2.1 单项选择题	(5)
1.2.2 填空题(将正确的答案填在相应的空中)	(7)
1.3 习题解析	(8)
第 2 章 顺序表	(14)
2.1 基本概念和运算	(14)
2.1.1 向量	(14)
2.1.2 栈	(16)
2.1.3 队列	(18)
2.2 基本题	(20)
2.2.1 单项选择题	(20)
2.2.2 填空题(将正确的答案填在相应的空中)	(22)
2.3 习题解析	(23)
2.3.1 向量	(23)
2.3.2 栈	(28)
2.3.3 队列	(35)
第 3 章 链表	(47)
3.1 基本概念和运算	(47)
3.1.1 单链表	(47)
3.1.2 双链表	(52)
3.1.3 链栈和链队	(56)
3.2 基本题	(59)
3.2.1 单项选择题	(59)
3.2.2 填空题(将正确的答案填在相应的空中)	(62)
3.3 习题解析	(64)
3.3.1 单链表	(64)
3.3.2 双链表	(87)
第 4 章 串	(94)
4.1 串的存储及其运算	(94)

4.1.1 顺序存储及其基本运算	(94)
4.1.2 链接存储及其基本运算	(97)
4.2 基本题	(101)
4.2.1 单项选择题	(101)
4.2.2 填空题(将正确的答案填在相应的空中)	(101)
4.3 习题解析	(102)
第5章 数组和稀疏矩阵.....	(118)
5.1 基本概念和运算	(118)
5.1.1 多维数组	(118)
5.1.2 稀疏矩阵	(120)
5.2 基本题	(126)
5.2.1 单项选择题(其中 $A[i..j]$ 表示下标从 i 到 j)	(126)
5.2.2 填空题(将正确的答案填在相应的空中)	(128)
5.3 习题解析	(129)
第6章 递归.....	(148)
6.1 递归设计方法	(148)
6.1.1 递归模型	(148)
6.1.2 递归的执行过程	(148)
6.1.3 递归设计	(149)
6.1.4 递归到非递归的转换	(149)
6.2 基本题	(151)
6.2.1 单项选择题	(151)
6.2.2 填空题(将正确的答案填在相应的空中)	(152)
6.3 习题解析	(154)
第7章 广义表.....	(179)
7.1 广义表的表示及其运算	(179)
7.1.1 广义表的表示	(179)
7.1.2 广义表的基本运算	(180)
7.2 基本题	(183)
7.2.1 单项选择题	(183)
7.2.2 填空题(将正确的答案填在相应的空中)	(184)
7.3 习题解析	(185)
第8章 树形结构.....	(196)
8.1 基本概念和运算	(196)
8.1.1 树	(196)
8.1.2 二叉树	(198)
8.1.3 二叉排序树	(203)
8.1.4 树和森林	(206)

8.1.5 Huffman 树	(207)
8.2 基本题	(208)
8.2.1 单项选择题	(208)
8.2.2 填空题(将正确的答案填在相应的空中)	(213)
8.3 习题解析	(219)
第 9 章 图	(264)
9.1 图的存储及其运算	(264)
9.1.1 图的基本术语	(264)
9.1.2 图的存储方式	(265)
9.1.3 图的基本运算	(269)
9.2 基本题	(277)
9.2.1 单项选择题	(277)
9.2.2 填空题(将正确的答案填在相应的空中)	(279)
9.3 习题解析	(279)
第 10 章 查找	(291)
10.1 基本查找方法	(291)
10.1.1 顺序查找	(291)
10.1.2 二分查找	(292)
10.1.3 分块查找	(293)
10.1.4 哈希表查找	(294)
10.1.5 背包问题及其求解函数	(296)
10.2 基本题	(299)
10.2.1 单项选择题	(299)
10.2.2 填空题(将正确的答案填在相应的空中)	(300)
10.3 习题解析	(301)
第 11 章 内排序	(313)
11.1 基本排序方法	(313)
11.1.1 插入排序	(313)
11.1.2 希尔(Shell)排序	(314)
11.1.3 起泡排序	(315)
11.1.4 快速排序	(315)
11.1.5 选择排序	(316)
11.1.6 堆排序	(317)
11.1.7 归并排序	(318)
11.1.8 基数排序	(319)
11.2 基本题	(320)
11.2.1 单项选择题	(320)
11.2.2 填空题(将正确的答案填在相应的空中)	(322)
11.3 习题解析	(323)

第 12 章 文件	(335)
12.1 基本文件组织方式	(335)
12.1.1 顺序文件	(335)
12.1.2 索引文件	(335)
12.1.3 直接存取文件	(337)
12.1.4 多关键字文件	(337)
12.2 基本题	(337)
12.2.1 单项选择题	(337)
12.2.2 填空题(将正确的答案填在相应的空中)	(338)
12.3 习题解析	(338)
第 13 章 外排序	(344)
13.1 基本归并排序法	(344)
13.1.1 磁盘文件归并排序	(344)
13.1.2 磁带文件归并排序	(346)
13.2 基本题	(348)
13.2.1 单项选择题	(348)
13.2.2 填空题(将正确的答案填在相应的空中)	(348)
13.3 习题解析	(348)
参考文献	(353)

第1章 概 述

自从1946年第一台计算机问世以来,计算机技术的发展日新月异。其应用已不再局限于科学计算,而是更多地用于控制、管理及数据处理等非数值计算的处理工作。与此相应,计算机加工处理的对象由纯粹的数值发展到字符、表格和图像等各种具有一定结构的数据,数据结构就是研究数据组织、存储和运算的一般方法的学科。本章讨论数据结构的基本概念及相关题解。

1.1 基本概念

数据是信息的载体,在计算机科学中是指所有能输入到计算机中并由计算机程序处理的符号的总称。

1.1.1 数据结构

数据结构是指同一数据元素类中各数据元素之间存在的关系。数据结构又可以分为下述三个组成部分,它们分别是数据的逻辑结构、数据的存储结构和数据的运算。

数据的逻辑结构是对数据之间关系的描述,所以有时就把数据的逻辑结构简称为数据结构。逻辑结构形式上用一个二元组

$$B=(K,R)$$

来表示,其中 K 是结点即数据元素的有限集合,即 K 是由有限个结点所构成的集合, R 是 K 上的关系的有限集合,即 R 是由有限个关系所构成的集合,而每个关系都是从 K 到 K 的关系。设 r 是一个 K 到 K 的关系, $r \in R$,若 $k, k' \in K$,且 $\langle k, k' \rangle \in r$,则称 k' 是 k 的后继, k 是 k' 的前驱,这时 k 和 k' 是相邻的结点(相对 r 而言);如果不存在一个 k' 使 $\langle k, k' \rangle \in r$,则称 k 为 r 的终端结点;如果不存在一个 k' 使 $\langle k', k \rangle \in r$,则称 k 为 r 的开始结点;如果 k 既不是终端结点也不是开始结点,则称 k 是内部结点。

数据的存储结构是数据的逻辑结构在计算机存储器中的实现,逻辑结构是从逻辑关系上观察数据,它与数据的存储无关,即独立于计算机,而存储结构是依赖于计算机的。计算机存储器是由有限多个存储单元组成的,每个存储单元有唯一的地址,各存储单元的地址是连续编码的,每个存储单元 Z 都有唯一的后续单元 $Z' = \text{succ}(Z)$, Z 和 Z' 称为相邻单元。一片相邻的存储单元的整体叫做存储区域,记做 M 。把 B 存储在计算机中,首先必须建立一个从 K 的结点到 M 的单元的映象 $S: K \rightarrow M$,即对于每一个 $k \in K$,都有唯一的 $Z \in M$ 使得 $S(k) = Z$, Z 为 K 中结点所占存储空间中的起始单元。通常有四种基本的存储映象方法,即顺序方法、链接方法、索引方法和散列方法。

数据的运算是在数据的逻辑结构上定义的操作算法,如检索、插入、删除、更新和排序等。

从逻辑上可以把数据结构分为线性结构和非线性结构。在线性结构中有且仅有一个终端结点和一个开始结点,并且所有结点都最多只有一个前驱和后续,顺序表就是典型的线性结构。非线性结构中可能有多个终端结点和多个开始结点,每个结点可能有多个前驱和多个后续。非线性结构中最重要的是树形结构和图形结构。

1.1.2 存储方式

1. 线性结构的存储方式

线性结构的数据有顺序、链式、索引和散列等四种存储方式。

顺序存储方式是把逻辑上相邻的结点存储在物理上相邻的存储单元里,结点之间的关系由存储单元的邻接关系来体现。其优点是占用最少的存储空间,其缺点是由于每个结点只能使用一整块存储区域,因此可能产生较多的碎片现象。另外,在作插入或删除操作时需移动大量元素。

链式存储方式是将结点所占的存储单元分为两部分,一部分存放结点本身的信息,即为数据项;另一部分存放该结点的后续结点所对应的存储单元的地址,即为指针项。其优点是充分利用所有的存储单元,不会出现碎片现象,其缺点是每个结点占用较多的存储空间。

索引方式是用结点的索引号来确定结点存储地址,其优点是检索速度快,其缺点是增加了额外的索引表,会占用较多的存储空间。另外,在增加和删除数据时还要修改索引表因而会花费较多时间。

散列方式是根据结点的值确定它的存储地址,其优点是检索、增加和删除结点的操作都很快,其缺点是采用不好的散列函数时可能出现结点存储单元的冲突,为解决冲突需要额外的时间和空间开销。

2. 树形结构的存储方式

在树形结构的数据中,每个结点可能有多个后续结点,因此一般只能采用链接的方式进行存储,链接的方式正好能表达树形结构中的父子和兄弟两种层次关系。由于链接的方式不能表达任意多个儿子结点,所以常常使用较规则的树如二叉树,来限制后续结点的最多个数。而其他几种存储方式难以达到这种要求。

也可以在特定规则的情况下,采用顺序结构(如一维数组)来存储树形结构。如图 1.1 的一颗二叉树,用以下数组来存储:

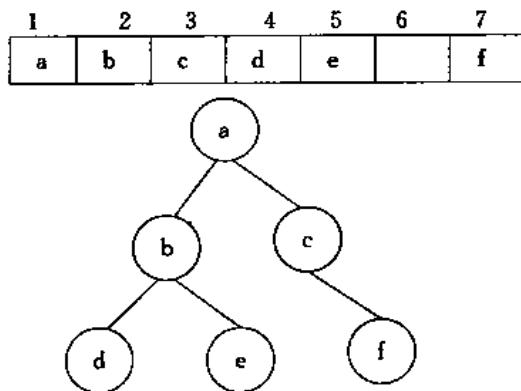


图 1.1 一颗二叉树

3. 图形结构的存储方式

在图形结构的数据中,每个结点可能有多个前驱结点和多个后续结点,因此一般只能采用链接的方式进行存储,同样,由于链接的方式不能表达任意多个后续结点,因此,将链接方式改进成邻接表,即图形结构中的每个结点对应有一个链表,该链表存储这个结点的所有相邻结点。

另处还可以采用矩阵存储图形结构,即用矩阵表示图形结构中任意两个结点之间的关系,这种矩阵称为邻接矩阵。

1.1.3 算法及其评价

算法是解决某一特定类型问题的有限运算序列。描述一个算法可以采用某一种计算机语言,也可以采用流程图等。本书的算法是采用C语言描述的。

算法具有5个基本特性:有穷性、确定性、可行性、输入和输出。

评价一个算法一般从4个方面进行:正确性、运行时间、占用空间和简单性。

正确性是指算法是否正确。运行时间是指一个算法在计算机上运行所花费的时间,采用时间复杂度来量度,所谓时间复杂度是指算法中包含简单操作的次数,一般不必精确计算出算法的时间复杂度,只要大致计算出相应的数量级,如 $O(1)$ 、 $O(\log_2 n)$ 、 $O(n)$ 或 $O(n^2)$ 等。 O 的形式定义为:若 $f(n)$ 是正整数 n 的函数,则 $x_n = O(f(n))$ 表示存在一个正的常数 M ,使得当 $n \geq n_0$ 时满足 $|x_n| \leq M|f(n)|$ 。一般地,常用的时间复杂度有如下关系:

$$O(1) \leq O(\log_2 n) \leq O(n) \leq O(n \log_2 n) \leq O(n^2) \leq O(n^3) \leq \dots \leq O(n^k) \leq O(2^n)$$

占用空间是指在计算机存储器上所占用的存储空间,主要考虑在算法运行过程中临时占用的存储空间的大小,称之为空间复杂度,一般以数量级形式给出。简单性是指算法的易读性等。

例1. 分析以下程序段的时间复杂度。

```
for (i=1; i<n; i++)
{
    y=y+1;           ①
    for (j=0; j<=(2*n); j++)
        x++;         ②
}
```

语句的频度指的是该语句重复执行的次数。一个算法中所有语句的频度之和构成了该算法的运行时间。在本例算法中,其中语句①的频度是 $n-1$,语句②的频度是 $(n-1)(2n+1) = 2n^2 - n - 1$ 。则该程序段的时间复杂度 $T(n) = 2n^2 - n - 1 = O(n^2)$ 。

例2. 分析以下程序段的时间复杂度。

```
i=1;           ①
while (i<=n)
{
    i=i*2;      ②
}
```

其中语句①的频度是1,设语句②的频度 $f(n)$,则有:

$$2^{f(n)} \leq n$$

即 $f(n) \leq \log_2 n$, 取最大值 $f(n) = \log_2 n$

则该程序段的时间复杂度 $T(n) = 1 + f(n) = 1 + \log_2 n = O(\log_2 n)$ 。

例 3. 分析以下程序段的时间复杂度。

```

a=0; b=1;           ①
for (i=2; i<=n; i++) ②
{
    s=a+b;           ③
    b=a;              ④
    a=s;              ⑤
}

```

其中语句①的频度是 2; 语句②的频度是 n ; 语句③的频度是 $n-1$; 语句④的频度是 $n-1$; 语句⑤的频度是 $n-1$ 。则该程序段的时间复杂度 $T(n) = 2 + n + 3 * (n-1) = 4n - 1 = O(n)$ 。

例 4. 有以下程序, 分析其中 `order()` 函数的时间复杂度。

```

int a[] = {2, 5, 1, 7, 9, 3, 6, 8};
order(int j, int m)
{
    int i, temp;
    if (j < m)
    {
        for (i=j; i<=m; i++)
            if (a[i] < a[j])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        i++;
        order(i, m);
    }
}

main()
{
    int i;
    order(0, 7);
    for (i=0; i<=7; i++)
        printf("%d ", a[i]);
}

```

`order()` 函数是一个递归排序过程, 设 $T(n)$ ($n=m+1$) 是排序 n 个元素所需要的时间。

在排序 n 个元素时,算法的计算时间主要花费在递归调用 $\text{order}()$ 上。第一次调用时,处理的元素个数为 $n-1$,也就是对余下的 $n-1$ 个元素进行排序,这部分所需要的计算时间应为 $T(n-1)$ 。

又因为在其中的循环中,需要 $n-1$ 次比较,所以排序 n 个元素所需要的时间为:

$$T(n) = T(n-1) + n - 1 \quad n > 1$$

这样得到如下方程:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(n-1) + n - 1 & n > 1 \end{cases}$$

求解过程为:

$$\begin{aligned} T(n) &= [T(n-2) + (n-2)] + (n-1) \\ &= [T(n-3) + (n-3)] + (n-2) + (n-1) \\ &= \dots \\ &= (T(1) + 1) + 2 + \dots + n - 1 \\ &= 0 + 1 + 2 + \dots + n - 1 \\ &= n(n-1)/2 \\ &= O(n^2) \end{aligned}$$

故 $\text{order}()$ 函数的时间复杂度为 $O(n^2)$ 。

1.2 基本题

1.2.1 单项选择题

1. 数据结构是一门研究非数值计算的程序设计问题中计算机的①以及它们之间的②和运算等的学科。

- ① A. 操作对象 B. 计算方法 C. 逻辑存储 D. 数据映象
② A. 结构 B. 关系 C. 运算 D. 算法

答:①A ②B

2. 数据结构被形式地定义为 (K, R) , 其中 K 是①的有限集合, R 是 K 上的②有限集合。

- ① A. 算法 B. 数据元素 C. 数据操作 D. 逻辑结构
② A. 操作 B. 映象 C. 存储 D. 关系

答:①B ②D

3. 在数据结构中,从逻辑上可以把数据结构分成 ①。

- A. 动态结构和静态结构 B. 紧凑结构和非紧凑结构
C. 线性结构和非线性结构 D. 内部结构和外部结构

答:①C

4. 线性表的顺序存储结构是一种 ① 的存储结构,线性表的链式存储结构是一种 ② 的

存储结构。

- A. 随机存取 B. 顺序存取 C. 索引存取 D. 散列存取

答:①A ②B

5. 算法分析的目的是 ①, 算法分析的两个主要方面是 ②。

- ① A. 找出数据结构的合理性
B. 研究算法中的输入和输出的关系
C. 分析算法的效率以求改进
D. 分析算法的易懂性和文档性

- ② A. 空间复杂性和时间复杂性
B. 正确性和简明性
C. 可读性和文档性
D. 数据复杂性和程序复杂性

答:①C ②A

6. 计算机算法指的是 ①, 它必须具备输入、输出和 ② 等五个特性。

- ① A. 计算方法 B. 排序方法
C. 解决问题的有限运算序列 D. 调度方法

- ② A. 可行性、可移植性和可扩充性
B. 可行性、确定性和有穷性
C. 确定性、有穷性和稳定性
D. 易读性、稳定性和安全性

答:①C ②B

7. 线性表的逻辑顺序与存储顺序总是一致的, 这种说法 ①。

- A. 正确 B. 不正确

答:①B

8. 线性表若采用链式存储结构时, 要求内存中可用存储单元的地址 ①。

- A. 必须是连续的 B. 部分地址必须是连续的
C. 一定是不连续的 D. 连续或不连续都可以

答:①D

9. 在以下的叙述中, 正确的是 ①。

- A. 线性表的线性存储结构优于链表存储结构
B. 二维数组是其数据元素为线性表的线性表
C. 栈的操作方式是先进先出
D. 队列的操作方式是先进后出

答:①B

10. 每种数据结构都具备三个基本运算: 插入、删除和查找, 这种说法 ①。

- A. 正确 B. 不正确

答:①B

1.2.2 填空题(将正确的答案填在相应的空中)

1. 数据逻辑结构包括 ①、②和③三种类型,树形结构和图形结构合称为 ④。

答:①线性结构 ②树形结构 ③图形结构 ④非线性结构

2. 在线性结构中,第一个结点 ① 前驱结点,其余每个结点有且只有 ② 个前驱结点;最后一个结点 ③ 后续结点,其余每个结点有且只有 ④ 个后续结点。

答:①没有 ②1 ③没有 ④1

3. 在树形结构中,树根结点没有 ① 结点,其余每个结点有且只有 ② 个前驱结点;叶子结点没有 ③ 结点,其余每个结点的后续结点可以 ④ 。

答:①前驱 ②1 ③后续 ④任意多个

4. 在图形结构中,每个结点的前驱结点数和后续结点数可以 ①。

答:①任意多个

5. 线性结构中元素之间存在 ① 关系,树形结构中元素之间存在 ② 关系,图形结构中元素之间存在 ③ 关系。

答:①一对一 ②一对多 ③多对多

6. 算法的五个重要特性是 ____、____、____、____、____。

答:有穷性 确定性 可行性 输入 输出

7. 下面程序段的时间复杂度是 ①。

```
for (i=0; i<n; i++)
    for (j=0; j<m; j++)
        A[i][j]=0;
```

答:① $O(m * n)$

8. 下面程序段的时间复杂度是 ①。

```
i=s=0;
while (s<n)
{
    i++; /* i=i+1 */
    s+=i; /* s=s+i */
}
```

答:① $O(n)$

9. 下面程序段的时间复杂度是 ①。

```
s=0;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        s+=B[i][j];
sum=s;
```


答:① $O(n^2)$

10. 下面程序段的时间复杂度是 ①。

```
i=1;
while (i<=n)
    i=i*3;
```

答:① $\log_3 n$

1.3 习题解析

1. 设有数据逻辑结构为:

$B = (K, R)$

$K = \{k_1, k_2, \dots, k_9\}$

$R = \{ \langle k_1, k_3 \rangle, \langle k_1, k_8 \rangle, \langle k_2, k_3 \rangle, \langle k_2, k_4 \rangle, \langle k_2, k_5 \rangle, \langle k_3, k_9 \rangle, \langle k_5, k_6 \rangle, \langle k_8, k_9 \rangle, \langle k_9, k_7 \rangle, \langle k_4, k_7 \rangle, \langle k_4, k_6 \rangle \}$

画出这个逻辑结构的图示,并确定相对于关系 R ,哪些结点是开始结点,哪些结点是终端结点?

解:该题的逻辑结构图示如图 1.2 所示。

开始结点是指无前驱的结点,这里满足该定义的开始结点为 k_1, k_2 。

终端结点是指无后续的结点,这里满足该定义的终端结点为 k_6, k_7 。

该逻辑结构是非线性结构中的图形结构。

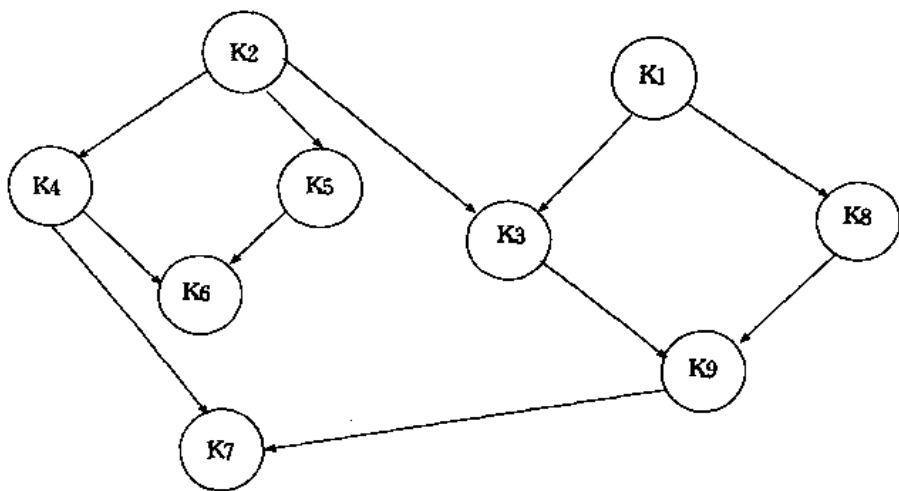


图 1.2 逻辑结构图示

2. 设有如图 1.3 所示的逻辑结构图示,给出它的逻辑结构。

解:本题的逻辑结构如下:

$B = (K, R)$

$K = \{k_1, k_2, \dots, k_9\}$

$R = \{ \langle k_1, k_2 \rangle, \langle k_1, k_3 \rangle, \langle k_3, k_4 \rangle, \langle k_3, k_6 \rangle, \langle k_6, k_8 \rangle, \langle k_4, k_5 \rangle, \langle k_6, k_7 \rangle \}$

$k7>, <k8, k9>\}$

该逻辑结构是一个树形结构,其树根为 $k1$,叶子结点为 $k2$ 、 $k5$ 、 $k7$ 和 $k9$ 。

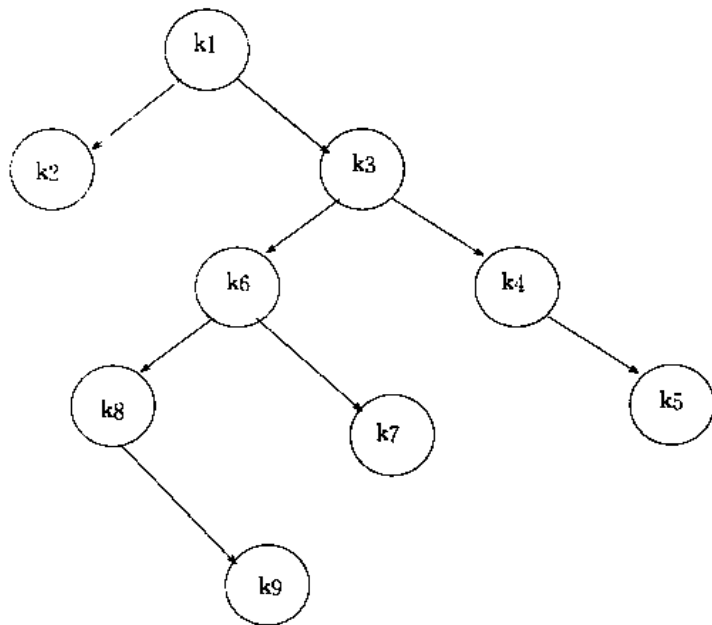


图 1.3 逻辑结构图示

3. 下列是几种用二元组表示的数据结构,画出它们分别对应的逻辑图形表示,并指出它们分别属于何种结构。

(1) $A=(K,R)$, 其中:

$K=\{a,b,c,d,e,f,g,h\}$

$R=\{r\}$

$r=\{<a,b>, <b,c>, <c,d>, <d,e>, <e,f>, <f,g>, <g,h>\}$

(2) $B=(K,R)$, 其中:

$K=\{a,b,c,d,e,f,g,h\}$

$R=\{r\}$

$r=\{<d,b>, <d,g>, <d,a>, <b,c>, <g,e>, <g,h>, <e,f>\}$

(3) $C=(K,R)$, 其中:

$K=\{1,2,3,4,5,6\}$

$R=\{r\}$

$r=\{(1,2), (2,3), (2,4), (3,4), (3,5), (3,6), (4,5), (4,6)\}$

这里的圆括号对表示两结点是双向的。

(4) $D=(K,R)$, 其中:

$K=\{48,25,64,57,82,36,75\}$

$R=\{r1, r2\}$

$r1=\{<25,36>, <36,48>, <48,57>, <57,64>, <64,75>, <75,82>\}$

$r2=\{<48,25>, <48,64>, <64,57>, <64,82>, <25,36>, <82,75>\}$

解:

(1)A 对应逻辑图形如图 1.4 所示,它是一种线性结构。

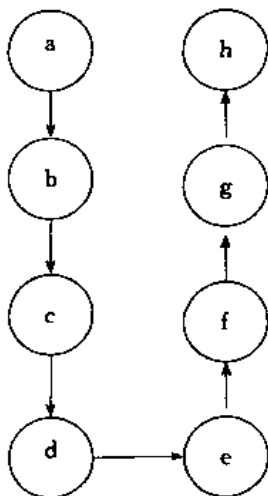


图 1.4 对应 A 的逻辑结构图示

(2)B 对应逻辑图形如图 1.5 所示,它是一种树形结构。

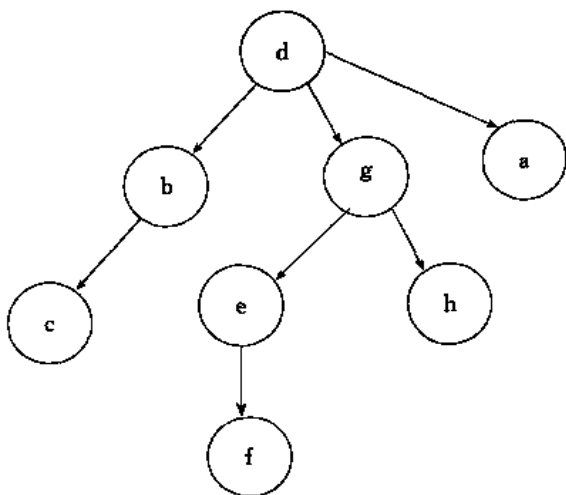


图 1.5 对应 B 的逻辑结构图示

(3)C 对应逻辑图形如图 1.6 所示,它是一种图形结构。

(4)D 对应逻辑图形如图 1.7 所示,它是一种图形结构, r_1 (对应图中虚线部分) 为线性结构, r_2 (对应图中实线部分) 则为树形结构。

4. 有如下递归函数 $\text{fact}(n)$, 分析其时间复杂度。

```

fact(int n)
{
    if (n <= 1) return(1);           ①
    else return(n * fact(n-1));      ②
}
  
```

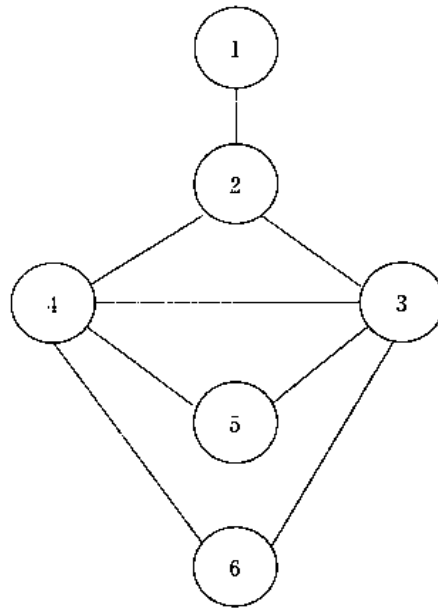


图 1.6 对应 C 的逻辑结构图示

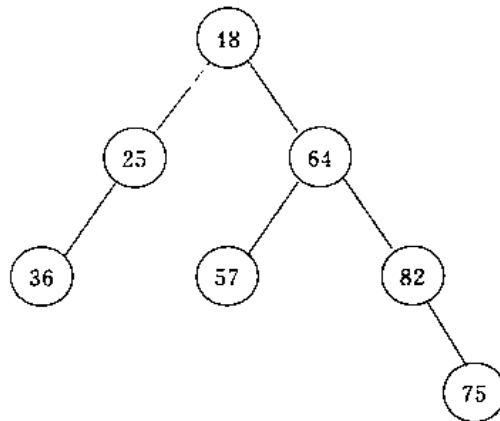


图 1.7 对应 D 的逻辑结构图示

解: 设 $\text{fact}(n)$ 的运行时间函数是 $T(n)$ 。该函数中语句①的运行时间是 $O(1)$, 语句②的运行时间是 $T(n-1) + O(1)$, 其中 $O(1)$ 为运算的时间。

$$\text{因此: } T(n) = \begin{cases} O(1) & n \leq 1 \\ T(n-1) + O(1) & n > 1 \end{cases}$$

$$\begin{aligned} \text{则: } T(n) &= O(1) + T(n-1) \\ &= 2 * O(1) + T(n-2) \end{aligned}$$

...

$$= (n-1) * O(1) + T(1)$$

$$= n * O(1)$$

$$= O(n)$$

即 $\text{fact}(n)$ 的时间复杂度为 $O(n)$ 。

5. 指出下列各算法的时间复杂度。

(1) prime(int n) /* n 为一个正整数 */

```
{
    int i=2;
    while ((n % i) != 0 && i * 1.0 < sqrt(n)) i++;
    if (i * 1.0 > sqrt(n))
        printf("%d 是一素数\n", n);
    else
        printf("%d 不是一素数\n", n);
}
```

(2) sum1(int n) /* n 为一个正整数 */

```
{
    int p=1, sum=0, i;
    for (i=1; i<=n; i++)
    {
        p *= i;
        sum += p;
    }
    return(sum);
}
```

(3) sum2(int n) /* n 为一个正整数 */

```
{
    int sum=0, i, j;
    for (i=1; i<=n; i++)
    {
        p=1;
        for (j=1; j<=i; j++) p *= j;
        sum += p;
    }
    return(sum);
}
```

解: 算法的时间复杂度是由嵌套最深层语句的频度决定的。

(1) prime 的嵌套最深层语句:

i++;

它的频度由条件 $((n \% i) != 0 \ \&\& \ i * 1.0 < \text{sqrt}(n))$ 决定, 显然 $i * 1.0 < \text{sqrt}(n)$, 即执行频度小于 $\text{sqrt}(n)$, 所以其时间复杂度是 $O(\sqrt{n})$ 。

(2) sum1 的嵌套最深层语句:

p *= i; sum += p;

它的频度为 n 次, 所以其时间复杂度是 $O(n)$ 。

(3) sum2 的嵌套最深层语句:

```
p *= j;
```

它的频度为 $1+2+3+\dots+n=n(n+1)/2$ 次, 所以其时间复杂度是 $O(n^2)$ 。

6. 求两个 n 阶矩阵的乘法 $C=A \times B$, 其算法如下:

```
#define MAX 100
void maxtrixmult(int n, float a[MAX][MAX], b[MAX][MAX], float c[MAX][MAX])

{
    int i, j, k;
    float x;
    for (i=1; i<=n; i++)           ①
    {
        for (j=1; j<=n; j++)       ②
        {
            x=0;                    ③
            for (k=1; k<=n; k++)    ④
                x+=a[i][k]*b[k][j]; ⑤
            c[i][j]=x;              ⑥
        }
    }
}
```

分析该算法的时间复杂度。

解: 该算法中主要语句的频度分别是:

- ① $n+1$
- ② $n(n+1)$
- ③ n^2
- ④ $n^2(n+1)$
- ⑤ n^3
- ⑥ n^2

则时间复杂度为所有语句的频度之和 $T(n)=2n^3+3n^2+2n+1=O(n^3)$ 。

第2章 顺序表

线性结构中的所有结点按它们之间的关系可以排成一个线性序列:

$$k_1, k_2, \dots, k_n$$

其中 k_1 是开始点, k_n 是终端结点, k_i 是 k_{i+1} 的前驱结点, 而 k_{i+1} 是 k_i 的后续结点 ($i=1, 2, \dots, n-1$)。

通常把上述线性序列称为“线性表”, 把线性结构中的结点称为元素或“表目”。将一个线性表存放到计算机中, 可以采用不同的方法, 其中最简单而自然的就顺序的方法, 即把表目按其索引值从小到大一个接一个地存放在相邻的单元里。顺序方法存储的线性表简称“顺序表”, 顺序表是一种紧凑结构。

2.1 基本概念和运算

最简单也是最常用的顺序表有向量、栈和队列, 下面分别讨论这些基本顺序表的概念和运算。

2.1.1 向量

1. 定义

向量指的是所有元素都是同一类型结点的线性表。

向量的定义如下:

```
typedef ElemType vector[n0]
```

这里的 ElemType 可以是任何相应的数据类型如 int, float 或 char 等, 在算法中, 我们规定 ElemType 缺省是 int 类型。向量中的元素个数 n 小于或等于某一整数 n_0 。

说明在 C 语言中, 数组的下标是从 0 开始的, 但为了描述算法简洁, 本书中的向量与文献[2]中一致, 规定从下标 1 开始, 这样, 读者可不必考虑下标 0 的数组值。

2. 向量的建立

输入 n 个整数, 产生一个存储这些整数的向量 A 的函数如下:

```
void create(A, n)
vector A;
int n;
{
    int i;
    for (i=1; i<=n; i++)
        scanf("%d", A[i]);
}
```

3. 向量的存储方法

向量通常的存储方法是顺序存储,每个元素在存储器中占用的空间大小相同,若第一个元素存放的位置是 $LOC(k_1)$,每个元素占用的空间大小为 s ,则元素 k_i 的存放位置为:

$$LOC(k_i) = LOC(k_1) + s * (i - 1)$$

任给一个 i ,便可以很快计算出 $LOC(k_i)$,因此,对顺序存储的向量要查找任何一个元素都很方便。

4. 向量的插入

在一个有 n 个元素的向量 A 中的第 i 个元素之前插入一个元素 x 的函数如下:

```
void insert(A,n,x)
vector A;
int n,x;
{
    int j;
    if (i<1 || i>n) printf("i 值错误! \n");
    else
    {
        for (j=n;j>=i;j--) A[j+1]=A[j]; /* 将第 i 个元素及其后的元素后移 * */
        A[i]=x;
        n++; /* 向量长度增 1 * */
    }
}
```

5. 向量的删除

在一个有 n 个元素的向量 A 中删除第 i 个元素的函数如下:

```
void delete(A,n)
vector A;
int n;
{
    int j;
    if (i<1 || i>n) printf("i 值错误! \n");
    else
    {
        for (j=i;j<=n;j++) A[j]=A[j+1]; /* 将第 i 个元素之后的元素前移 * */
        n--; /* 向量长度减 1 * */
    }
}
```

6. 向量的查找

在一个有 n 个元素的向量 A 中查找元素值为 x 的元素的函数如下:


```

void find(A,n,x)
vector A;
int n,x;
{
    int j;
    i=1;
    while (i<=n && A[i]<>x) i++;
    if (i<=n)
        printf("找到了! \n");
    else
        printf("未找到\n");
}

```

2.1.2 栈

栈是限定仅在栈顶一端进行压入(push)或弹出(pop)操作的线性数据结构。栈的主要特点是“后进先出”,即后进栈的元素先处理。通常栈可以用顺序方式存储,分配一块连续的存储区域存放栈中的表,并用一个变量指向当前的栈顶。

1. 栈的定义

假设栈的元素个数最大不超过整数 m_0 ,所有的元素都具有同一数据类型 $ElemType$,则可用下列方式来定义栈类型 $stack$:

```

typedef struct
{
    ElemType s[m0];
    int top;
} stack;

```

这里的 $ElemType$ 可以是任何相应的数据类型如 int , $float$ 或 $char$ 等,在算法中,我们规定 $ElemType$ 缺省是 int 类型。其中变量 top 指向栈的栈顶,称为栈指针。 m_0 是一个正整数,表示栈中可容纳的最多元素数,例如用以下宏定义设置其值为 100:

```
#define m0 100
```

一般栈的形式如图 2.1 所示。

2. 栈的压入 push(ST,x)

将整数 x 插入到 ST 栈中的函数如下:

```

void push(ST,x)
stack *ST;
int x;
{
    if (ST->top==m0) printf("栈上溢出! \n"); /* 若栈满则显示相应信息 */
    else /* 否则栈指针 top 增 1,将 x 赋给栈顶的元素 */
    {

```

```
ST->top = ST->top + 1;
ST->s[ST->top] = x;
```

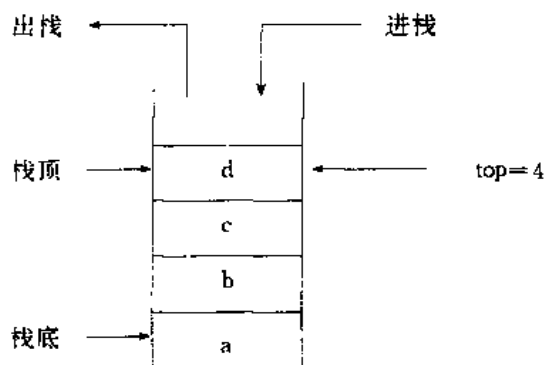


图 2.1 栈中元素和栈顶指针之间的关系

3. 栈的弹出 pop(ST)

从栈中弹出栈顶元素的函数如下：

```
void pop(ST)
stack * ST;

if (ST->top == 0) printf("栈下溢出! \n");    /* 若栈空则显示相应信息 */
else
    ST->top--;    /* 否则栈指针减 1, 即栈顶为下一个元素 */
```

4. 读栈顶元素 top(ST, x)

读取栈顶元素而保持栈不变的函数如下：

```
void top(ST, x)
stack * ST;
int x;

if (ST->top == 0) printf("无栈顶元素! \n"); /* 若栈为空则显示相应信息 */
else
    x = ST->s[ST->top];    /* 否则把栈顶元素赋给 x, 但保持栈不变 */
```

5. 判定栈是否为空 empty(ST)

判定栈 ST 是否为空栈的函数如下：

```
int empty(ST)
stack * ST;
```

```

    if (ST->top == 0) return(1);    /* 若栈为空则返回 true */
    else return(0);                /* 否则返回 false */
}

```

6. 取栈顶元素 ptop(ST)

从栈中取出栈顶元素并从栈中删除该栈顶元素的函数如下:

```

int ptop(ST)
stack * ST;
{
    top(ST, x);    /* 将栈顶元素赋给 x */
    pop(ST);        /* 将栈顶元素弹出 */
    return(x);      /* 返回 x 值 */
}

```

2.1.3 队列

队列是一种线性表,所有的插入都只允许在表的一端进行插入,在表的另一端进行删除。进行删除的一端叫队列的头,进行插入的一端叫队列的尾。

1. 队列的存储方式

假设队列的元素个数最大不超过整数 m_0 ,所有的元素都具有同一数据类型 datatype,则可用下列方式来定义队列类型 queue:

```

typedef struct
{
    ElemType q[m0];
    int front, rear;
} queue;

```

这里的 ElemType 可以是任何相应的数据类型如 int, float 或 char 等,在算法中,我们规定 ElemType 缺省是 int 类型。变量 front 指向队列的头部,变量 rear 指向队列的尾部。

一般队列的形式如图 2.2 所示。

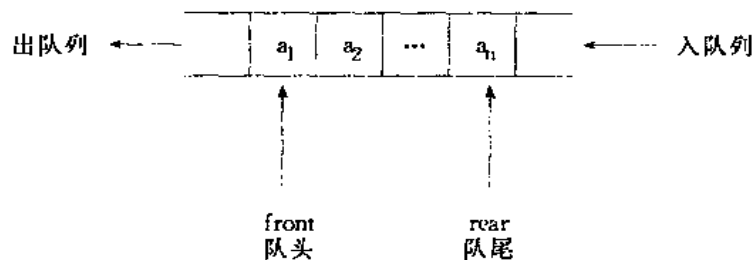


图 2.2 队列

2. 队列的插入 enq(QU, x)

将整数 x 插入到 QU 队列中的函数如下:

```
void enq(QU.x)
queue * QU;
int x;

if (QU->rear == m0) printf("队列上溢出! \n");
else

    QU->rear++;          /* 队尾指针后移 */
    QU->q[QU->rear] = x;  /* 新元素赋给队尾单元 */
    if (QU->front == 0) QU->front = 1;
    /* 若原为空队,则进行插入后,同时把队首指针置为1 */
```

3. 队列元素的删除

删除一个 QU 队列尾部元素的函数如下:

```
void deleteq(QU)
queue * QU;

if (QU->front == 0) printf("队列下溢出! \n");
else

    if (QU->front == QU->rear) /* 队列空的情况 */

        QU->front = 0;
        QU->rear = 0;

    else QU->front--;          /* 队列不为空的情况 */
```

4. 读队列的头结点 fnode(QU.x)

读取队列 QU 的队列头元素的函数如下:

```
void fnode(QU.x)
queue * QU;
int x;

if (QU->front == 0) printf("队列下溢出! \n");
else x = QU->q[QU->front];
```

5. 判定队列是否为空 qempty(QU)

判定队列 QU 是否为空的函数如下:

```

int qempty(QU)
queue * QU;
{
    if (QU->front == 0) return(1);    /* 为空,则返回 true */
    else return(0);                  /* 不为空,则返回 false */
}

```

2.2 基本题

2.2.1 单项选择题

1. 一个向量第一个元素的存储地址是 100, 每个元素的长度为 2, 则第 5 个元素的地址是 ①。

- A. 110 B. 108 C. 100 D. 120

答: ①B

[第 5 个元素的地址 = $100 + 2 * (5 - 1) = 108$]

2. 一个栈的入栈序列是 a, b, c, d, e, 则栈的不可能的输出序列是 ①。

- A. edcba B. decba
C. dceab D. abcde

答: ①C

[栈的特点是先进后出, 所以在 C 中 eab 是不可能产生的。]

3. 若已知一个栈的入栈序列是 1, 2, 3, ..., n, 其输出序列为 $p_1, p_2, p_3, \dots, p_n$, 若 $p_1 = n$, 则 p_i 为 ①。

- A. i B. $n - i$ C. $n - i + 1$ D. 不确定

答: ①C

[当 $p_1 = n$, 即 n 是最先出栈的, 根据栈的原理, n 必定是最后入栈的, 那么输入顺序必定是 1, 2, 3, ..., n, 则出栈的序列是 n, ..., 3, 2, 1, 所以答案是 C。]

4. 栈结构通常采用的两种存储结构是 ①。

- A. 顺序存储结构和链表存储结构
B. 散列方式和索引方式
C. 链表存储结构和数组
D. 线性存储结构和非线性存储结构

答: ①A

5. 判定一个栈 ST(最多元素为 m_0) 为空的条件是 ①。

- A. $ST \rightarrow top < 0$ B. $ST \rightarrow top = 0$
C. $ST \rightarrow top < m_0$ D. $ST \rightarrow top = m_0$

答: ①B

6. 判定一个栈 ST(最多元素为 m_0) 为栈满的条件是 ①。

- A. (rear-front + m) % m B. read-front + 1
C. read-front - 1 D. read-front

答:①A

14. 栈和队列的共同点是 ①。

- A. 都是先进后出
- B. 都是先进先出
- C. 只允许在端点处插入和删除元素
- D. 没有共同点

答:①C

15. 表达式 $a * (b + c) - d$ 的中缀表达式是 ①。

- A. $abcd + -$
- B. $abc + * d -$
- C. $abc * + d -$
- D. $- + * abcd$

答:①

2.2.2 填空题(将正确的答案填在相应的空中)

1. 向量、栈和队列都是 ① 结构,可以在向量的 ② 位置插入和删除元素;对于栈只能在 ③ 插入和删除元素;对于队列只能在 ④ 插入元素和 ⑤ 删除元素。

答:①线性 ②任何 ③栈顶 ④队尾 ⑤队首

2. 向一个长度为 n 的向量的第 i 个元素 ($1 \leq i \leq n+1$) 之前插入一个元素时,需向后移动 ① 个元素。

答:① $n-i+1$

3. 向一个长度为 n 的向量中删除第 i 个元素 ($1 \leq i \leq n$) 时,需向前移动 ① 个元素。

答:① $n-i$

4. 向栈中压入元素的操作是 ①。

答:①先移动栈顶指针,后存入元素

5. 对栈进行退栈时的操作是 ①。

答:①先取出元素,后移动栈顶指针

6. 在一个循环队列中,队首指针指向队首元素的 ①。

答:①前一个位置

7. 从循环队列中删除一个元素时,其操作是 ①。

答:①先移动队首元素,后取出元素

8. 在具有 n 个单元的循环队列中,队满时共有 ① 个元素。

答:① $n-1$

9. 一个栈的输入序列是 12345,则栈的输出序列 43512 是 ①。

答:①不可能的

10. 一个栈的输入序列是 12345,则栈的输出序列 12345 是 ①。

答:①可能的

2.3 习题解析

2.3.1 向量

1. 已知一个向量 A, 其中的元素按值非递减有序排列, 编写一个函数插入一个元素 x 后保持该向量仍按递减有序排列。

解: 本题的算法思想是: 先找到适当的位置, 然后后移元素空出一个位置, 再将 x 插入。实现本题功能的函数如下:

```
void insert(vector A, int n, x)                /* 向量 A 的长度为 n */
{
    int i, j;
    if (x >= A[n]) A[n+1] = x; /* 若 x 大于最后的元素, 则将其插入到最后 */
    else
    {
        i = 1;
        while (x >= A[i]) i++; /* 查找插入位置 i */
        for (j = n; j >= i; j--) A[j+1] = A[j]; /* 移出插入 x 的位置 */
        A[i] = x;
        n++; /* 将 x 插入、向量长度增 1 */
    }
}
```

2. 已知一个向量中的元素按元素值非递减有序排列, 编写一个函数删除向量中多余的值相同的元素。

解: 本题的算法思想是: 由于向量中的元素按元素值非递减有序排列, 值相同的元素必为相邻的元素, 因此依次比较相邻两个元素, 若值相等, 则删除其中一个, 否则继续向后查找。实现本题功能的函数如下:

```
void del(vector A, int n)                      /* 向量 A 的长度为 n */
{
    int i = 1, j;
    while (i <= n-1)
    {
        if (A[i] != A[i+1]) i++; /* 元素值不相等, 继续向下找 */
        else
        {
            for (j = (i+2); j <= n; j++) A[j-1] = A[j]; /* 删除第 i+1 个元素 */
            n--; /* 向量长度减 1 */
        }
    }
}
```

3. 编写一个函数将一个向量 A (有 n 个元素, 且任何元素均不为 0) 分拆成两个向量, 使 A 中大于 0 的元素存放在 B 中, 小于 0 的元素存放在 C 中。

解:本题的算法思想是:依次遍历 A 的元素,比较当前的元素值,大于 0 者赋给 B(假设有 p 个元素),小于 0 者赋给 C(假设有 q 个元素)。实现本题功能的函数如下:

```
void ret(vector A,int n,vector B,int p,vector C,int q)
{
    int i;
    p=0;q=0;
    for (i=1;i<=n;i++)
    {
        if (A[i]>0)
        {
            p++;
            B[p]=A[i];
        }
        if (A[i]<0)
        {
            q++;
            C[q]=A[i];
        }
    }
}
```

4. 已知在一维数组 $A[1, m+n]$ 中依次存放着两个向量 (a_1, a_2, \dots, a_m) 和 (b_1, b_2, \dots, b_n) , 编写一个函数将两个向量的位置互换, 即把 (b_1, b_2, \dots, b_n) 放到 (a_1, a_2, \dots, a_m) 的前面。

解:本题的算法思想是:由于向量的插入与删除操作需要移动大量的元素,所用时间多,这里采用先将:

$A: (a_1, a_2, \dots, a_m, b_1, b_2, \dots, b_n)$

的所有元素逆置,即使之变成:

$A: (b_n, \dots, b_2, b_1, a_m, \dots, a_2, a_1)$

然后将 (b_n, \dots, b_2, b_1) 逆置为 (b_1, b_2, \dots, b_n) , 将 (a_m, \dots, a_2, a_1) 逆置为 (a_1, a_2, \dots, a_m) , 这样便得到最终结果:

$A: (b_1, b_2, \dots, b_n, a_1, a_2, \dots, a_m)$

先编写一个逆置的函数如下,其功能是逆置 A 中 $A[l]$ 到 $A[h]$ 的部分:

```
void invert(A,l,h)
vector A;
int l,h;
{
    int i,x;
    for (i=l;i<=(l+h)/2;i++)
    {
```

```
x=A[i];A[i]=A[i-h-i];A[i-h-i]=x; /* 将 A[i] 与 A[i-h-i] 元素互换 */
```

那么,实现本题功能的函数如下:

```
void exchange(A.m.n)
vector A;
int m,n;

invert(A.1.m-n);
invert(A.1.n);
invert(A.n+1.m-n);
```

* 5. 编写一个函数从一给定的向量 A 中删除元素值在 x 到 y ($x \leq y$) 之间的所有元素,要求以较高的效率来实现。

解:本题的算法思想是:先将 A 向量中所有元素值在 x 到 y 之间的元素置成一个特殊的值(如 0),并不立即删除它们,然后从最后向前依次扫描,对于该特殊值的元素便移动其后面的元素将其删除,这种算法比每删除一个元素后立即移动其后元素效率要高一些。实现本题功能的过程如下:

```
void del(A.n.x.y)
vector A;
int n,x,y;

int i,k;
for (i=1;i<=n;i++)
    if (A[i]>=x && A[i]<=y) A[i]=0;
for (i=n;i>=1;i--)
    if (A[i]==0)
    {
        for (k=i;k<=(n-1);k++) A[k]=A[k+1];
        n--;
    }
;
```

* 6. 编写一个函数用不多于 $3n/2$ 的平均比较次数,在一个向量 A 中找出最大和最小值的元素。

解:本题的算法思想是:如果在查找出最大和最小值的元素时各扫描一遍所有元素,则至少要比 较 $2n$ 次,为此,使用一趟扫描找出最大和最小值的元素。实现本题功能的函数如下:

```
void maxmin(A.n)
vector A;
```

```

int n;
{
    int max,min,i;
    max=A[1];min=A[1];
    for (i=2;i<=n;i++)
        if (A[i]>max) max=A[i];
        else if (A[i]<min) min=A[i];
    printf("max=%d,min=%d\n",max,min);
}

```

在这个函数中,最坏情况是向量 A 的元素以递减顺序排列,这时 $(A[i]>\max)$ 条件均不成立,这时比较的次数为 $n-1$,另外每次都要比较 $A[i]<\min$,同样所花比较次数为 $n-1$,因此,总的比较次数为:

$$2(n-1)$$

最好的情况是向量 A 的元素递增次序排列,这时 $(A[i]>\max)$ 条件均成立,不会再执行 else 的比较,所以总的比较次数为:

$$n-1$$

平均比较次数为:

$$(2(n-1)+n-1)/2=3n/2-3/2$$

所以该函数的平均比较次数不多于 $3n/2$ 。

7. 设 $A=(a_1,a_2,\dots,a_m)$ 和 $B=(b_1,b_2,\dots,b_n)$ 均为向量, A' 和 B' 分别为 A 和 B 中除去最大共同前缀后的子表(例如, $A=(x,y,y,z,x,z)$, $B=(x,y,y,z,y,x,x,z)$, 则两者的最大共同前缀为 x,y,y,z , 在两向量中除去最大共同前缀后的子表分别为 $a'=(x,z)$, $b'=(y,x,x,z)$)。若 $A'=B'=\text{空表}$, 则 $A=B$; 若 $A'=\text{空表}$, $B'\neq\text{空表}$, 或两者均不空且 A' 首元小于 B' 首元, 则 $A<B$; 否则 $A>B$ 。编写一个函数根据上述方法比较 A 和 B 的大小。

解: 本题已经给出了详细的算法, 这里不再讨论。实现本题功能的函数如下:

```

void compare(A,B,m,n)
vector A,B;
int m,n;
{
    vector AS,BS;
    int i=1,j,ms=0,ns=0;
    while (A[i]==B[i]) i++; /* i 是最大共同前缀之后的第一个元素的下标 */
    for (j=i;j<=m;j++)
    {
        AS[j-i+1]=A[j];ms++; /* AS 为 A 的子表 */
    }
    for (j=i;j<=n;j++)
    {

```

```

    BS[j-1+1]=B[j];ns++; /* BS 为 B 的子表 */
}
if (ms==ns && ms==0) printf("A=B\n");
else

    if (ms==0 && ns>0) || (ms>0 && ns>0 && AS[1]<BS[1])
        printf("A<B\n");
    else
        printf("A>B\n");
}
}

```

8. 有两个向量 A (有 m 个元素) 和 B (有 n 个元素), 其元素均以从小到大的升序排列, 编写一个函数将它们合并成一个向量 C, 要求 C 的元素也是以从小到大的升序排列。

解: 本题的算法思想是: 依次扫描通过 A 和 B 的元素, 比较当前的元素的值, 将较小值的元素赋给 C, 如此直到一个向量扫描完毕, 然后将未完的一个向量的余下所有元素赋给 C 即可。实现本题功能的函数如下:

```

void link(A,B:vector;m,n:integer;VAR C:vector)
vector A,B,C;
int m,n;
{
    int i=1,j=1,k=1,l;
    while (i<=m && j<=n) /* 扫描通过 A 和 B, 将当前扫描的较小元素赋给 C */
        if (A[i]<B[j])
        {
            C[k]=A[i];i++;k++;
        }
        else
        {
            C[k]=B[j];j++;k++;
        }
    if (j==n) /* 当 A 未完时, 当 A 的余下元素赋给 C */
        for (l=(i+1);l<=m;l++)
            C[k]=A[l];k++;
    if (i==m) /* 当 B 未完时, 当 B 的余下元素赋给 C */
        for (l=(j+1);l<=n;l++)
            C[k]=B[l];k++;
}

```

2.3.2 栈

1. 对于一个栈,给出输入项 A,B,C。如果输入项序列由 A,B,C 所组成,试给出全部可能的输出序列。

解:本题利用栈的“后进先出”的特点,有如下几种情况:

A 进 A 出 B 进 B 出 C 进 C 出 产生输出序列 ABC
 A 进 A 出 B 进 C 进 C 出 B 出 产生输出序列 ACB
 A 进 B 进 B 出 A 出 C 进 C 出 产生输出序列 BAC
 A 进 B 进 B 出 C 进 C 出 A 出 产生输出序列 BCA
 A 进 B 进 C 进 C 出 B 出 A 出 产生输出序列 CBA
 不可能产生输出序列 CAB。

2. 有字符串次序为 $3 * - y - a / y \uparrow 2$, 试利用栈排出将次序改变为 $3y - * ay \uparrow / -$ 的操作步骤。(可用 X 代表扫描该字符串函数中顺序取一字符进栈的操作,用 S 代表从栈中取出一字符加到新字符串尾的出栈的操作)。例如:ABC 变为 BCA, 则操作步骤为 XXSXSS。

解:实现上述转换的进出栈操作如下:

3 进 3 出 * 进 - 进 y 进 y 出 - 出 * 出 - 进 a 进
 a 出 / 进 y 进 y 出 \uparrow 进 2 进 2 出 \uparrow 出 / 出 - 出

所以操作步骤为:

XSXXXSSXSXSXSXSXS

* 3. 有两个栈 s1 和 s2 共享存储空间 $c[1, m]$, 其中一个栈底设在 $c[1]$ 处, 另一个栈底设在 $c[m_0]$ 处, 分别编写 s1 和 s2 的进栈 push(i, x)、退栈 pop(i) 和设置栈空 setnull(i) 的函数, 其中 $i=1, 2$ 。注意: 仅当整个空间 $c[1, m_0]$ 占满时才产生上溢。

解: 该共享栈的结构如图 2.3 所示, 两栈的最多元素个数为 m_0 , $top1$ 是栈 1 的栈指针, $top2$ 是栈 2 的栈指针, 当 $top2 = top1 + 1$ 时出现上溢出, 当 $top1 = 0$ 时栈 1 出现下溢出, 当 $top2 = m_0 + 1$ 时栈 2 出现下溢出。根据上述原理得到如下函数:

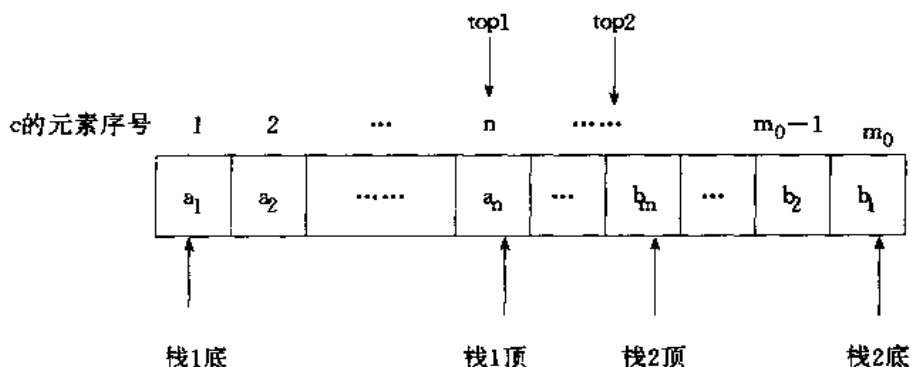


图 2.3 共享栈

/* top1, top2 和 m0 均为已赋初值的 int 型全局变量 */

void push(x, i)

int x, i;

if (top1 == top2 - 1) printf("上溢出! \n");

else

if (i == 1) /* 对第一个栈进行入栈操作 */

{

top1++; c[top1] = x;

}

else /* 对第二个栈进行入栈操作 */

{

top2--; c[top2] = x;

}

/* 函数 pop */

void pop(i)

int i;

if (i == 1) /* 对第一个栈进行出栈操作 */

if (top1 == 0) printf("栈 1 下溢出! \n");

else

{

pop = c[top1]; top1--;

}

else /* 对第二个栈进行出栈操作 */

if (top2 == m0 + 1) printf("栈 2 下溢出! \n");

else

{

pop = c[top2]; top2++;

}

/* 函数 setnull */

setnull(i)

int i;

if (i == 1) top1 = 0;

else top2 = m0 + 1;

* 4. 证明: 有可能从初始输入序列 $1, 2, \dots, n$, 利用一个栈得到输出序列 p_1, p_2, \dots, p_n (p_1, p_2, \dots, p_n 是 $1, 2, \dots, n$ 的一种排列) 的充分必要条件是: 不存在这样的 i, j, k 满足 $i < j < k$ 同时 $p_i < p_k < p_j$.

证明:【充分条件】如果不存在这样的 i, j, k 满足 $i < j < k$ 同时 $p_j < p_k < p_i$, 即对于输入序列:

$\dots, p_i, \dots, p_k, \dots, p_i, \dots$ ($p_j < p_k < p_i$)

不存在这样的输出序列:

$\dots, p_i, \dots, p_j, \dots, p_k, \dots$

(或简单地对于输入序列 1, 2, 3 不存在输出序列 3, 1, 2)

从中看到, p_i 后进先出, 是满足栈的特点, 因为 p_i 最大也就是在 p_j 和 p_k 之后进入, 却在输出序列中排在 p_j 和 p_k 之前, 同时也说明, 在 p_k 之前先进入的 p_j 不可能在 p_k 之后出来, 反过来说明满足先进后出的特点, 所以构成一个栈。

【必要条件】如果初始输入序列是 $1, 2, \dots, n$, 假设是进栈, 又同时存在这样的 i, j, k 满足 $i < j < k$ 同时 $p_j < p_k < p_i$, 即对于输入序列:

$\dots, p_i, \dots, p_k, \dots, p_i, \dots$ ($p_j < p_k < p_i$)

存在这样的输出序列:

$\dots, p_i, \dots, p_j, \dots, p_k, \dots$

从中看到, p_i 先进后出, 是满足栈的特点, 因为 p_i 最大也就是在 p_j 和 p_k 之后进入, 同时看到在 p_k 之前先进入的 p_j 却在 p_k 之前出来, 反过来说明不满足先进后出的特点, 与前面的假设是栈不一致, 本题即证。

* 5. 假定一个有 n 个“栈”的铁路转轨网络, 如图 2.4 所示 $n=4$ 时的情况。初始时有 2^n 列列车位于网络右边。试证明: 通过实施一系列适当的操作(要求操作的方向与图中箭头的方向相一致), 最终在左边可形成 $2^n!$ 种不同的列车排列顺序。

注: 假设每个栈足以容纳所有的列车。

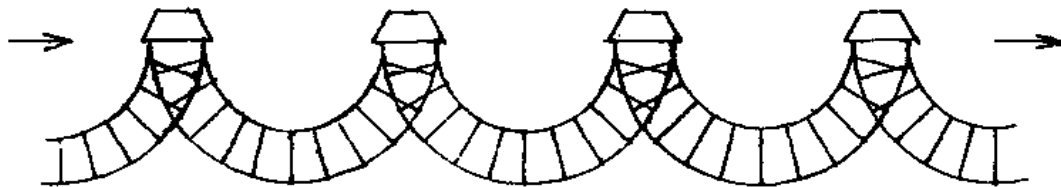


图 2.4 4 个铁路栈

证明: 本题若能证明在上述条件下, 对于输入序列:

a_1, a_2, \dots, a_m (a 表示一列列车, $m = 2^n$)

产生的输出序列可以是任意的 a_1, a_2, \dots, a_m 的排列组合, 那么根据排列组合的原理, 输出的排列顺序是 $2^n!$ 种, 下面我们利用数学归纳法来证明满足这种情况。

当 $n=1$ 时, 输入序列为 a_1, a_2 , 显然通过一个栈后的输出序列可以为 a_1, a_2 和 a_2, a_1 , 即输出序列为 $2^1! = 2$ 种。

假设 $m=n$ 时成立, 即 2^{n-1} 列列车通过 n 个“栈”后可以产生这 2^n 列列车的任意排列序

列。

当 $m = n + 1$ 时,对于要求产生的任意输出序列:

$a_{i1}, a_{i2}, \dots, a_{ik}, a_{j1}, a_{j2}, \dots, a_{jk}$ ($k = 2^n$, 共有 $k + k = 2k = 2^{n+1}$ 列列车)

先通过第一个“栈”,将初始序列:

a_1, a_2, \dots, a_m ($m = 2^{n+1}$)

中的 $a_{j1}, a_{j2}, \dots, a_{jk}$ 这 k 列列车不考虑顺序进入第一个“栈”,余下的 $a_{i1}, a_{i2}, \dots, a_{ik}$ (不考虑顺序)不入第一个“栈”,这样将输入序列分成两部分,现在还余下 n 个“栈”,依前面的假设,先可以将未入第一个“栈”的 $a_{i1}, a_{i2}, \dots, a_{ik}$ (不考虑顺序)通过后面的 $n-1$ “栈”产生有序的输出序列:

$a_{i1}, a_{i2}, \dots, a_{ik}$

然后对于入第一个“栈”的序列 $a_{j1}, a_{j2}, \dots, a_{jk}$ (不考虑顺序)通过后面的 $n-1$ “栈”产生有序的输出序列:

$a_{j1}, a_{j2}, \dots, a_{jk}$

这样便产生最终输出序列:

$a_{i1}, a_{i2}, \dots, a_{ik}, a_{j1}, a_{j2}, \dots, a_{jk}$

为此我们证明本题。

* 6. 假设一个算术表达式中包含圆括弧、方括弧和花括弧三种类型的括弧,编写一个判别表达式中括弧是否正确配对的函数 $\text{correct}(\text{exp}, \text{tag})$;其中:exp 为字符串类型的变量,表示被判别的表达式,tag 为布尔型变量。

解:本题使用一个栈 st 进行判定,将 '('、'[' 或 '{' 入栈,当遇到 ')', ']' 或 '}' 时,检查当前栈顶元素是否是对应的 ')', ']' 或 '}', 若是则退栈,否则返回表示不配对。当整个算术表达式检查完毕时栈为空,表示括号正确配对;否则不配对。实现本题功能的函数如下:

```
#define m0 100          /* m0 为算术表达式中最多字符个数 */
correct(exp, tag)
char exp[m0];
int tag;

char st[m0];
int top=0, i=1;
tag=1;
while (i<=m0 && tag)

    if (exp[i]=='(' || exp[i]=='[' || exp[i]=='{')
        /* 遇到 '(', '[' 或 '{', 则将其入栈 */
        ;

        top++;
        st[top]=exp[i];
```



```

    }
    if (exp[i] == ')') /* 遇到')',若栈顶是'(',则继续处理,否则以不配对返回 */
        if (st[top] == '(') top--;
        else tag = 0;
    if (exp[i] == ']') /* 遇到']',若栈顶是'[',则继续处理,否则以不配对返回 */
        if (st[top] == '[') top--;
        else tag = 0;
    if (exp[i] == '}') /* 遇到'}',若栈顶是'{',则继续处理,否则以不配对返回 */
        if (st[top] == '{') top--;
        else tag = 0;
    i++;
}
if (top > 0) tag = 0; /* 若栈不空,则不配对 */
}

```

7. 编写一个函数将一般算术表达式转化为波兰表达式。

解:假设表达式中的符号以字符形式由键盘输入(为简单起见,设算术表达式中参加运算的数都只有一位数字),该算术表达式存放在字符型数组 str 中,其波兰表示式依次存放在字符型数组 exp 中,在处理函数中用一个字符型数组 stack 作为栈。设字符“#”为表达式的终止符,将算术表达式转换成波兰表示的方法如下。

依次从键盘输入表达式中的字符 c,对于每一个 c:

- ① 若 c 为数字,则将 c 依次存入数组 exp 中;
- ② 若 c 为左括弧“(”,则将此括弧压入栈 stack;
- ③ 若 c 为右括弧“)”,则将栈 stack 中左括弧“(”以前的字符依次弹出存入数组 exp 中,然后将左括弧“(”弹出;
- ④ 若 c 为“+”或“-”,则将当前栈 stack 中“(”以前的所有字符(运算符)依次弹出存入数组 exp 中,然后将 c 压入栈 stack 中;
- ⑤ 若 c 为“*”或“/”,则将当前栈 stack 中的栈顶端连续的“*”或“/”弹出并依次存入数组 exp 中,然后将 c 压入栈 stack 中;
- ⑥ 若 c 为“#”,则将栈 stack 中的所有运算符依次弹出并存入数组 exp 中,然后再将 c 存入数组 exp 中,最后可得到表达式的波兰表示在数组 exp 中。

根据上述转换原理得到的函数如下:

```

#define m0 100 /* m0 为算术表达式中最多字符个数 */
void trans()
{
    char str[m0]; /* 存储原算术表达式 */
    char exp[m0]; /* 存储转换成的波兰表达式 */
    char stack[m0]; /* 作为栈使用 */
    char ch;
    int i, j, t, top = 0; /* t 作为 exp 的下标, top 作为 stack 的下标, i 作为 str 的下标 */
    i = 0; /* 获取用户输入的表达式 */

```

```

do
{
    i++;
    scanf("%c",&str[i]);
    ! while (str[i] != '#' && i != m0);
    t=1,i=1;
    ch=str[i];i++;
    while (ch != '#')
    {
        if (ch>='0' && ch<='9')          /* 判定为数字 */
        {
            exp[t]=ch;t++;
        }
        else
        {
            if (ch=='(')                  /* 判定为左括号 */
            {
                top++;stack[top]=ch;
            }
            else
            {
                if (ch==')')              /* 判定为右括号 */
                {
                    while (stack[top] != '(')
                    {
                        exp[t]=stack[top];top--;t++;
                    }
                    top--;
                }
                else
                {
                    if (ch=='+' || ch=='-') /* 判定为加减号 */
                    {
                        while (top != 0 && stack[top] != '(')
                        {
                            exp[t]=stack[top];top--;t++;
                        }
                        top++;stack[top]=ch;
                    }
                    else
                    {
                        if (ch=='*' || ch=='/') /* 判定为 '*' 或 '/' 号 */
                        {
                            while (stack[top]== '*' || stack[top]== '/')
                            {
                                exp[t]=stack[top];top--;t++;
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```

        top++;stack[top]=ch;
    }
    ch=str[i];i++;
}
while (top!=0)
{
    exp[t]=stack[top];t++;top--;
}
exp[t]='#';
for (j=1;j<=t;j++) printf("%c",exp[j]);
printf("\n");
}

```

8. 编写一个函数求波兰表达式的值,其中波兰表达式是在该函数中输入的。

解:对波兰表示式求值函数中要用到一个数栈 stack,其实现函数如下:用户先以字符形式由键盘输入一个波兰表达式(为简单起见,设波兰表达式中的参加运算的数都只有一位数字),该波兰表达式存放在字符型数组 exp 中,从波兰表示式的开始依次扫描这个波兰表示式,当遇到运算对象时,就把它压入数栈 stack;当遇到运算符时,就执行两次弹出数栈 stack 中的数的操作,对弹出的数进行该运算符所指定的运算,再把结果压入数栈 stack,重复上述函数,直至扫描到表达式的终止符"#",在数栈顶得到表达式的值。

根据上述计算原理得到的函数如下:

```

#define m0 100          /* m0 为算术表达式中最多字符个数 */
void compvalue()
{
    char exp[m0];        /* 存储用户输入的波兰表达式 */
    float stack[m0],d;    /* 作为栈使用 */
    char c;
    int i=0,t=1,top=0;    /* t 作为 exp 的下标,top 作为 stack 的下标 */
    do                    /* 获取用户输入的波兰表达式 */
    {
        i++;
        scanf("%c",&exp[i]);
    } while (exp[i]!='#' && i!=m0);
    exp[i+1]='\0';
    c=exp[t];t++;
    while (c!='#')
    {
        if (c>='0' && c<='9') /* 判定为数字字符 */
        {
            d=c-'0';          /* 将数字字符转换成对应的数值 */
            top++;
            stack[top]=d;

```

```

    }
    else                /* 判定是运算符 */
    {
        switch (c)
        {
            case '+': stack[top-1] = stack[top-1] + stack[top];
                break;
            case '-': stack[top-1] = stack[top-1] - stack[top];
                break;
            case '*': stack[top-1] = stack[top-1] * stack[top];
                break;
            case '/': if (stack[top] != 0)
                stack[top-1] = stack[top-1] / stack[top];
                else
                    printf("除零错误! \n");
        }
        top--;
    }
    c = exp[t]; t++;
}

printf("计算结果是: %g", stack[top]);

```

2.3.3 队列

1. 假设 $Q[1,10]$ 是一个顺序队列, 初始状态为 $front=rear=0$, 画出做完下列操作后队列的头尾指针的状态变化情况, 若不能入队, 请指出其元素, 并说明理由。

d, e, b, g, h 入队
 d, e 出队
 i, j, k, l, m 入队
 b 出队
 n, o, p, q, r 入队

解: 本题入队和出队的变化函数如图 2.5 所示, 当元素 d, e, b, g, h 入队后, $rear=5$, $front=1$; 元素 d, e 出队, $rear=5$, $front=2$; 元素 i, j, k, l, m 入队, $rear=10$, $front=2$; 元素 b 出队, $rear=10$, $front=3$, 此时若再让 n, o, p, q, r 入队, 由于 $rear=10=m0$, 故栈上溢出。

2. 假设 $CQ[0,10]$ 是一个循环队列, 初始状态为 $front=rear=1$, 画出做完下列操作后队列的头尾指针的状态变化情况, 若不能入队, 请指出其元素, 并说明理由。

d, e, b, g, h 入队
 d, e 出队
 i, j, k, l, m 入队
 b 出队

n, o, p, q, r 入队

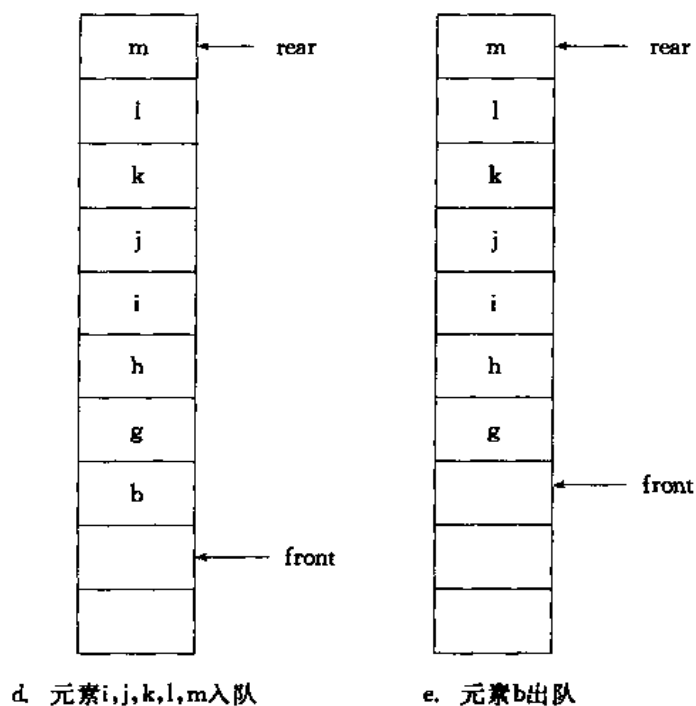
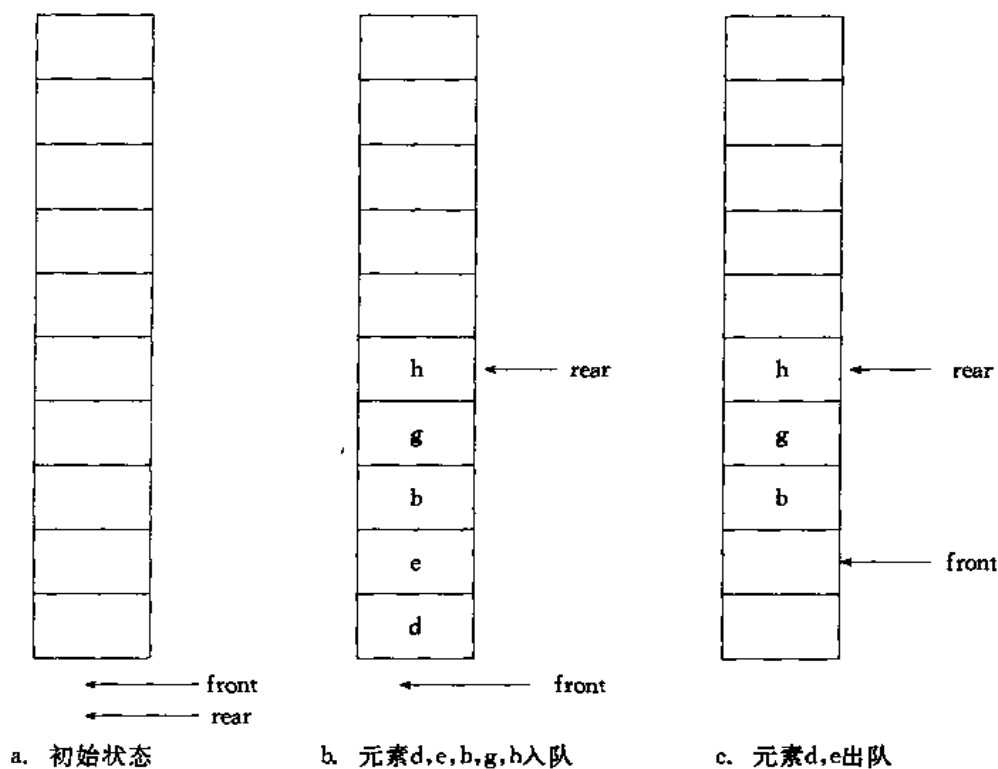


图 2.5 顺序队列入和出队列的变化情况

解: 本题入队和出队的变化函数如图 2.6 所示, 当元素 d, e, b, g, h 入队后, $rear = 6$.

front=1;元素d,e出队, rear=6, front=3;元素i,j,k,l,m入队, rear=0, front=3;元素b出队后, rear=0, front=4,此时让n,o,p入队,当q入列时,由于 rear=3, front=4,有 rear+1=front,故栈上溢出。

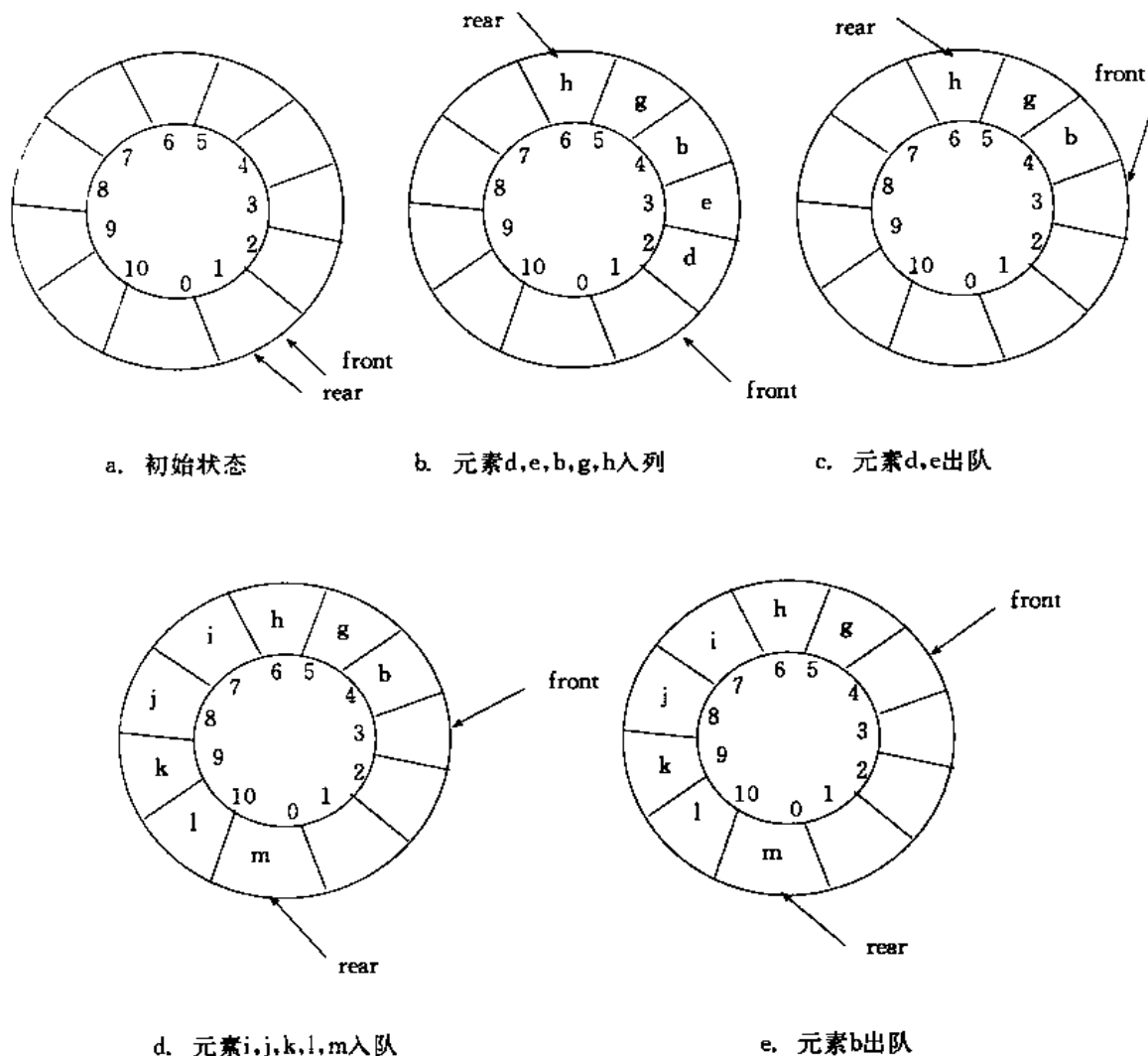


图 2.6 循环队列入和出队列的变化情况

* 3. 编写向顺序分配的环形队列 $QU[0, m0-1]$ 中插入一个结点的函数 enqueue 和从该队列中取出一个结点的 dequeue 函数。

解: 队列的特点是在尾部入队, 顶部出队, 在循环队列中, 最初队列为空时 front 和 rear 都指向同一位置, 当元素入队时, 由于是循环的, 所以 rear 位置前移, 即:

$$QU \rightarrow rear = (QU \rightarrow rear + 1) \% m0$$

把插入的元素放到 rear 的新位置; 当出队时, 先将 front 前移一个位置, 即:

$$QU \rightarrow front = (QU \rightarrow front + 1) \% m0$$

将 front 新位置的元素取出即可。根据上述原理得到如下函数:

```

#define m0 100
enqueue(QU,x)
queue QU;
int x;
{
    if ((QU->rear+1) % m0 == QU->front) printf("队列上溢出! \n");
    else
    {
        QU->rear=(QU->rear+1) % m0;
        QU->q[QU->rear]=x;
    }
}

dequeue(QU,x)
queue QU;
int x;
{
    if (QU->rear == QU->front) printf("队列下溢出! \n");
    else
    {
        QU->front=(QU->front+1) % m0;
        x=QU->q[front];
    }
}

```

4. 用单链表实现队列如图 2.7 所示,并令 $\text{front}=\text{rear}=\text{NIL}$ 表示队列为空,编写实现队列的如下五种运算的函数:

makenull:将队列置成空队列;
 front:返回队列的第一个元素;
 enqueue:把元素 x 插入到队列的后端;
 dequeue:删除队列的第一个元素;
 empty:判定队列是否为空。

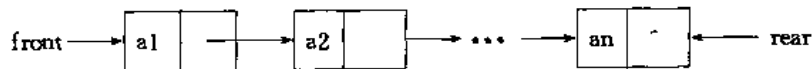


图 2.7 用单链表实现队列

解:依题意,定义单链表结点类型如下:

```

typedef struct linknode
{
    int data;
    struct linknode * next;
} node;

```

根据单链表的特点,实现队列的五种运算的函数或函数如下:

```
void makenull(front, rear)
node * front, * rear;
{
    front=NULL;rear=NULL;
}

void front(front, rear, x)
node * front, * rear;
int x;
{
    if (front==NULL) printf("队列下溢出! \n");
    else x=front->data;
}

void enqueue(front, rear, x)
node * front, * rear;
int x;
{
    node * p;
    if (rear==NULL) /* 若队列为空,则直接建立一个链表 */
    {
        rear=(node *)malloc(sizeof(node)); /* 建立一个新结点,由 rear 所指向 */
        rear->data=x;
        front=rear;
    }
    else /* 若不为空,则建立一个结点将其链接到末尾 */
    {
        p=(node *)malloc(sizeof(node)); /* 建立一个新结点,由 p 所指向 */
        p->data=x;
        rear->next=p;
        rear=p;
    }
}

void dequeue(front, rear)
node * front, * rear;
{
    node * p;
    if (front==NULL) printf("队列下溢出! \n");
    else
    {
        p=front;
        front=front->next;
        free(p);
    }
}
```



```

    }

    int empty(front)
    node * front;
    {
        if (front == NULL) return(1);
        else return(0);
    }

```

5. 如果用一个循环数组 $qu[0, m_0-1]$ 表示队列时, 该队列只有一个头指针 $front$, 不设队尾指针 $rear$, 而改置计数器 $count$ 用以记录队列中结点的个数。

(1) 编写实现队列的 5 个基本运算;

(2) 队列中能容纳元素的最多个数还是 m_0-1 吗?

解: (1) 依题意, 有如下条件:

队列为空: $count == 0, front == 1$

队列为满: $count == m_0$

队列尾的第一个元素位置 $== (front + count) \% m_0$

队列首的第一个元素位置 $== (front + 1) \% m_0$

队列类型定义如下:

```
typedef int qu[m0];
```

由此得到如下对应上述基本运算的 5 个函数如下:

```

/* m0 定义为全局变量 */
void makenull(front, count)    /* 使队列 q 为空 */
int front, count;
{
    front = 1;
    count = 0;
}

int empty(count)              /* 判定队列 q 是否为空 */
int count;
{
    if (count == 0) return(1);
    else return(0);
}

void pop(q, front, count, x)   /* 取队列头元素给 x */
qu q;
int front, count, x;
{
    if (count == 0) printf("队列下溢出\n");
    else
    {

```

```

        front = (front - 1) % m0;
        x = q[front];
    }

void enqueue(q.x, front, count) /* 将元素 x 入队列 */
qu q;
int x, front, count;
{
    int place;
    if (count == m0) printf("队列上溢出\n");
    else
    {
        count++;
        place = (front + count) % m0;
        q[place] = x;
    }
}

void dequeue(q, front, count) /* 删除队列头元素 */
qu q;
int front, count;
{
    if (count == 0) printf("队列下溢出\n");
    else
    {
        front = (front - 1) % m0;
        count--;
    }
}

```

(2) 队列中可容纳的最多元素个数为 $m0$ 个。

6. 假定用一个循环单链表表示队列(称为循环队列), 该队列只设一个队尾指针 $rear$, 不设队首指针, 编写如下函数:

- (1) 向循环链队中插入一个元素为 x 的结点;
- (2) 从循环链队中删除一个结点。

解: (1) 依题意, 定义本题队列的结点类型如下:

```

typedef struct linknode
{
    int data;
    struct linknode * next;
} qu;

```

队列插入一个结点的操作是在队尾进行的, 所以应在该循环链队的尾部插入一个结点,

其函数如下:

```

inqueue(rear, x)          /* 向队列中插入 x 结点 */
qu * rear;
int x;
{
    qu * head, * s;
    s = (qu *) malloc(sizeof(qu)); /* 建立一个新结点 */
    s->data = x;
    if (rear == NULL) /* 循环队列为空,则建立一个结点的循环队列 */
    {
        rear = s;
        rear->next = s;
    }
    else /* 循环队列不为空,则将 s 插到后面 */
    {
        head = rear->next;
        rear->next = s;
        rear = s; /* rear 始终指向最后一个结点 */
        rear->next = head;
    }
}

```

(2) 队列的删除结点是在队首进行的,所以应删除该循环链表的第一个结点,其函数如下:

```

void delqueue(rear)
qu * rear;
{
    if (rear == NULL) printf("队列下溢出! \n");
    else
    {
        head = rear->next; /* head 指向队首结点 */
        if (head == rear) rear = NULL /* 只有一个结点,则直接删除该结点 */
        else rear->next = head->next; /* 否则删除第一个结点 */
        free(head); /* 释放队首结点 */
    }
}

```

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

enqueue: 插入一个元素;

dequeue: 删除一个元素;

queue_empty: 判定队列为空。

解: 由于栈的特点是先进后出,为了模拟先进先出的队列,必须用两个栈,一个栈(s1)用

于插入元素,另一个栈(s2)用于删除元素,每次删除元素时应将前一个栈的所有元素读出然后进入第二个栈中,这样才能达到模拟队列的效果,这里使用栈的一些基本操作如下:

push(ST,x):栈的压入

ptop(ST,x):退出栈顶元素赋给 x

sempty(ST):判定栈是否为空

本题的函数如下:

```
void enqueue(s1,x)
stack s1;
int x;
{
    if (s1->top==m0) printf("队列上溢出! \n");
    else push(s1,x);
}

void dequeue(s1,s2,x)
stack s1,s2;
int x;
{
    s2->top=0; /* 将 s2 清空,将 s1 的元素退栈后推入 s2,此时 s1 为空了 */
    while (! sempty(s1)) push(s2,ptop(s1));
    ptop(s2,x); /* 将 s1 的栈顶元素退栈并赋给 x */
    while (! sempty(s2)) push(s1,ptop(s2));
    /* 将 s2 的所有元素退栈并推入 s1 中 */
}

int queue-empty(s1)
stack s1;
{
    if sempty(s1) return(1);
    else return(0);
}
```

* 8. 编写一个程序求解迷宫问题。迷宫是一个如图 2.8 所示的 m 行 n 列的 0—1 矩阵,其中 0 表示无障碍,1 表示有障碍。设入口为(1,1),出口为(m,n),每次移动只能从一个无障碍的单元移到其周围 8 个方向上任一无障碍的单元,编制程序给出一条通过迷宫的路径或报告一个“无法通过”的信息。

```

入口→ 0 0 0 1 0 0 0 1 0 0 0 1 0 0 1
        0 1 0 0 0 1 0 1 0 0 0 1 1 1 1
        0 1 1 1 1 1 0 1 0 0 1 1 1 0 1
        1 1 0 0 0 1 1 0 1 1 0 0 1 0 1
        1 0 0 1 0 1 1 1 1 0 1 0 1 0 1
        1 0 1 0 0 1 0 1 0 1 0 1 0 1 0
        1 0 1 1 1 1 1 0 0 1 1 1 1 0 0
        1 1 1 0 1 1 1 1 0 1 0 1 0 1 0
        1 0 1 0 1 0 1 1 1 0 1 0 0 0 1
        0 1 0 1 0 1 0 0 0 1 1 0 0 1 0→出口

```

图 2.8 迷宫的示意图

解：要寻找一条通过迷宫的路径，就必须进行试探性搜索，只要有路可走就前进一步，无路可走时，退回一步，重新选择未走过的可走的路，如此继续，直至到达出口或返回入口（无法通过迷宫）。我们可使用如下的数据结构： $mg[1..m, 1..n]$ 表示迷宫，为了算法方便，在四周加上一圈“哨兵”即变为数组 $mg[0..m+1, 0..n+1]$ 表示迷宫，用数组 zx, zy 分别表示 X, Y 方向的移动增量，其值如表 2.1 所示。

表 2.1 zx, zy 数组的方向取值

方向	北	东北	东	东南	南	西南	西	西北
下标	1	2	3	4	5	6	7	8
zx	-1	-1	0	1	1	1	0	-1
zy	0	1	1	1	0	-1	-1	-1

在探索前进路径时，需要将搜索的踪迹记下来，记录的踪迹应包含当前位置以及前驱位置。在搜索函数中，将所有需要搜索的位置形成一个队列，将队列中的每一个元素可能到达的位置加入到队列之中，当队列中某元素所有可能到达的位置全部加入到队列之后，即从队列中将该元素去掉。用变量 $front$ 及 $rear$ 分别表示队列的首与尾，当 $rear$ 指示的元素已到达出口 (m, n) 时，根据 $rear$ 所指示的前驱序号可回溯得到走迷宫的最短路径。

根据上述搜索函数得到的程序如下：

```

#include <stdio.h>
#define m 10                /* 行数 */
#define n 15                /* 列数 */
struct stype
{
    int x, y, pre;
} sq[400];
int mg[m+1][n+1];
int zx[8], zy[8];
void printlj(rear)
int rear;
{
    int i;

```

```

i=rear;
do
{
    printf("( %d, %d) ",sq[i].x,sq[i].y);
    i=sq[i].pre;
    while (!i==0);
}

void mglj()
{
    int i,j,x,y,v,front,rear,find;
    sq[1].x=1;sq[1].y=1;sq[1].pre=0;          /* 从(1,1)开始搜索 */
    find=0;
    front=1;rear=1;mg[1][1]=-1;
    while (front<=rear && ! find)
    {
        x=sq[front].x; y=sq[front].y;
        for (v=1;v<=8;v++)                    /* 循环扫描每个方向 */
        {
            i=x-zx[v];j=y-zy[v];              /* 选择一个前进方向(i,j) */
            if (mg[i][j]==0)                   /* 如果该方向可走 */
            {
                rear++;                          /* 进入队列 */
                sq[rear].x=i;
                sq[rear].y=j;
                sq[rear].pre=front;
                mg[i][j]=-1; /* 将其赋值-1,以避免回过来重复搜索 */
            }
            if (i==m && j==n)                   /* 找到了出口 */
            {
                printf("rear");
                find=1;
            }
        }
        front++;
    }

    if (! find) printf("不存在路径! \n");
}

main()
{
    int i,j;
    for (i=1;i<=m;i++)
        for (j=1;j<=n;j++) scanf("%ld",&mg[i][j]);
    for (i=0;i<=m+1;i++)
    {
        mg[i][0]=1;mg[i][n+1]=1;
    }
    for (j=0;j<=n+1;j++)
    {
        mg[0][j]=1;mg[m+1][j]=1;
    }
}

```

```
zx[1]=-1;zx[2]=-1;zx[3]=0;zx[4]=1;  
zx[5]=1;zx[6]=1;zx[7]=0;zx[8]=-1;  
zy[1]=0;zy[2]=1;zy[3]=1;zy[4]=1;  
zy[5]=0;zy[6]=-1;zy[7]=-1;zy[8]=-1;  
mgj();  
}
```

本程序求解图 2.8 的迷宫所得结果如下:

```
(10,15) (9,14) (8,15) (7,14) (6,13) (5,12) (4,11) (3,10) (3,9) (4,8) (3,7)  
(2,7) (1,6) (1,5) (2,4) (1,3) (1,2) (1,1)
```

第3章 链 表

链表是一种可以实现动态分配的存储结构,它不需要一组地址连续的存储单元,而是用一组任意的,甚至是在存储空间中零散分布的存储单元存放线性表的数据;同时在进行元素插入和删除时,链表不必像向量那样移动大量元素,从而克服了向量存储结构的缺点。但同时也失去了顺序表可随机存取的优点。

本章讨论链表的存储结构、基本运算及相关的题解。

3.1 基本概念和运算

常用的链表有单链表和双链表,单链表中又包含循环单链表,双链表中又包含循环双链表。在设计链表时,有带头结点的链表和不带头结点的链表两种形式。所谓带头结点的链表是使用一个专门的不存储任何数据的结点,其指针作为该链表的表头指针,该结点的指针域指向链表的第一个结点,这在程序设计时很方便;不带头结点的链表是该链表的表头指针即为该链表的第一个结点的指针。除非特别指定,本章中的链表都是指不带头结点的链表。

3.1.1 单链表

1. 单链表的存储结构

在单链表中分配给每个结点的存储单元可分为两个部分:一部分存放结点的数据,称为 data 域,另一部分存放指向结点后续结点的指针,称为 next 域,终端结点没有后续结点,其 next 域为 NULL,在计算机中可以表示成零或负数,另外还需要一个表头变量 head 指向链表的第一个结点。

单链表的结点类型 node 定义如下:

```
typedef struct linknode
{
    ElemType data;
    struct linknode * next;
} node;
```

这里的 ElemType 可以是任何相应的数据类型如 int, float 或 char 等,在算法中,我们规定 ElemType 缺省是 int 类型。

2. 建立一个单链表

输入一系列整数,以 0 标志结束,将这些整数作为 data 域建立一个单链表的函数如下:

```
void creat()
{
    node * head, * p, * s
```



```

int x,cycle=1;                /* cycle 是循环控制变量 */
head=(node *)malloc(sizeof(node)); /* 建立头结点,由 head 所指向 */
p=head;
while (cycle)
{
    scanf("%d",&x);
    if (x!=0)
    {
        s=(node *)malloc(sizeof(node)); /* 建立下一个结点,由 s 所指向 */
        s->data=x;
        p->next=s;                /* 把 s 结点链接到前面建立的单链表中 */
        p=s;
    }
    else cycle=0;
}
head=head->next;                /* 删除头结点 */
p->next=NULL;
}

```

如果输入的整数序列是:

6 10 3 6 7 5 0

则建立的单链表如图 3.1 所示。



图 3.1 单链表

循环单链表的结构与普通单链表一样,只是普通单链表的最后一个结点的 next 域取值为 NULL,而循环单链表的最后一个结点的 next 域指向第一个结点,如果要生成循环单链表,只需把上述函数 creat()中最后的一个语句 p->next=NULL 改为:

```
p->next=head;
```

即生成一个循环单链表,例如采用上述输入生成的循环单链表如图 3.2 所示。

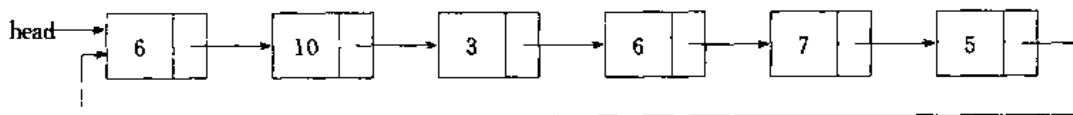


图 3.2 循环单链表

3. 查找某个结点

在已建立好的单链表(表头指针为 head)中查找元素值为 x 的函数如下:

```
void find(head,x)
```

```

node * head;
int x;
:
node * p;
p = head;
while (p->data != x && p != NULL) p = p->next;
if (p != NULL)
    printf("结点找到了! \n");
else
    printf("结点未找到! \n");
:

```

在循环单链表中查找某个结点的函数如下:

```

void find1(head, x)
node * head;
int x;

:

node * p;
if (head->data == x) printf("结点找到了! \n");
/* 若头结点为要找的结点,则显示相应信息并返回 */
else

    p = head->next;          /* 否则查找该结点 */
    while (p->data != x && p != head) p = p->next;
    if (p == head)
        printf("结点找到了! \n");
    else
        printf("结点未找到! \n");
:

```

4. 求单链表的长度

计算一个已建立好的单链表(表头指针为 head)的结点个数的函数如下:

```

int length(head)
node * head;
:
int n = 0;
node * p;
p = head;
while (p != NULL)
:
    p = p->next;
    n++;
:

return(n);

```

```
}
```

求一个循环单链表的长度的函数如下:

```
int length(head)
node * head;
{
    node * p;
    int n=0;
    if (head==NULL) n=0           /* 如果为空链表,则长度赋值 0 并返回 */
    else
    {
        p=head->next;           /* 如果不为空链表,则长度从 1 算起 */
        n=1;
        while (p!=head)
        {
            p=p->next;
            n++;
        }
    }
    return(n);
}
```

5. 在单链表中插入一个结点

在单链表中第 i 个结点 ($i \geq 0$) 之后插入一个元素为 x 的结点的函数如下:

```
void insert(head,i,x)
node * head;
int i,x;
{
    node * s,* p;
    int j;
    s=(node *)malloc(sizeof(node));    /* 建立一个待插入的结点 s */
    s->data=x;
    if (i==0)                          /* 如果 i=0,则将 s 所指结点插入到表头后返回 */
    {
        s->next=head;
        head=s;
    }
    else
    {
        p=head;j=1;                   /* 在单链表中查找第 i 个结点,由 p 所指向 */
        while (p!=NULL && j<i)
        {
            j++;
        }
    }
}
```

```

        p=p->next;
    }
    if (p!=NULL)                /* 若查找成功,则把 s 插入到其后 */
    {
        s->next=p->next;
        p->next=s;
    }
    else
        printf("未找到! \n");
}
}

```

6. 从单链表中删除一个结点

从单链表中删除一个其值等于给定值 x 的结点的函数如下:

```

void delete(head,x)
node * head;
int x;
{
    node * p, * q;
    if (head==NULL) printf("链表下溢! \n");    /* 如果单链表为空,则下溢处理 */
    if (head->data==x)                /* 如果表头结点值等于 x 值,则删除之 */
    {
        p=head;
        head=head->next;
        free(p);
    }
    else
    {
        q=head;p=head->next;    /* 从第二个结点开始查找其值为 x 的结点 */
        while (p!=NULL && p->data!=x)
            if (p->data!=x)        /* 在查找时,p 指向该结点,q 指向其前一结点 */
            {
                q=p;p=p->next;
            }
        if (p!=NULL)                /* 若找到了该结点,则进行删除处理 */
        {
            q->next=p->next;
            free(p);
        }
        else                        /* 未找到时,显示相应信息 */
            printf("未找到! \n");
    }
}

```

3.1.2 双链表

双链表中的结点除了包含一个指向后续结点的指针外,还有一个指向前驱结点的指针,这样可提供方便的双向查找功能。

1. 双链表的定义

双链表中结点的类型定义为:

```
typedef struct linknode
{
    ElemType data;
    struct linknode *left, *right;
} dnode;
```

这里的 ElemType 可以是任何相应的数据类型如 int, float 或 char 等,在算法中,我们规定 ElemType 缺省是 int 类型。

2. 建立一个双链表

输入一系列整数,以 0 标志结束,将这些整数作为 data 域建立一个双链表的函数如下:

```
void creat()
{
    dnode *head, *p, *s;
    int cycle=0; /* cycle 是循环控制变量 */
    head=(dnode *)malloc(sizeof(dnode)); /* 建立头结点,由 head 所指向 */
    p=head;
    while (cycle)
    {
        scanf("%d",&x);
        if (x!=0)
        {
            s=(dnode *)malloc(sizeof(dnode)); /* 建立下一个结点,由 s 所指向 */
            s->data=x;
            p->right=s; /* 把 s 所指结点链接到前面建立的双链表中 */
            s->left=p;
            p=s;
        }
        else cycle=1;
    }
    head=head->next; /* 删除头结点 */
    head->left=NULL; /* 将最后一个结点的 left 域置为 NULL */
    p->right=NULL; /* 将第一个结点的 right 域置为 NULL */
}
```

如果输入的整数序列是:

6 10 3 6 7 5 0

则建立的双链表如图 3.3 所示。

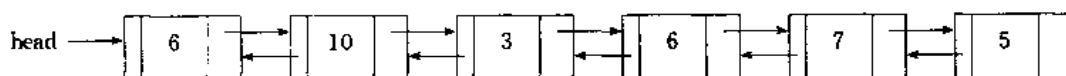


图 3.3 双链表

循环双链表的结构与普通双链表一样,只是普通双链表的第一个结点的 left 域置为 NULL,最后一个结点的 right 域置为 NULL,而循环双链表的最后一个结点的 right 域指向第一个结点,第一个结点的 left 域指向最后一个结点。如果要生成循环双链表,只需把上述函数 create()中最后的两个语句

```
head->left=NULL
```

```
p->right=NULL
```

改为:

```
head->left=p
```

```
/* 这里 p 指向最后一个结点 */
```

```
p->right=head
```

即生成一个循环双链表,例如采用上述输入生成的循环双链表如图 3.4 所示。

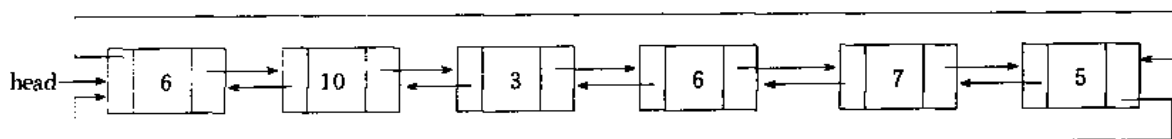


图 3.4 循环双链表

3. 查找某个结点

在已建立好的双链表(表头指针为 head)中查找元素值为 x 的函数如下:

```

void find(head,x)
dnode * head;
int x;
{
    dnode * p;
    p=head;
    while (p->data!=x && p!=NULL) p=p->right;
    if (p!=NULL)
        printf("结点找到了! \n");
    else
        printf("结点未找到! \n");
}
  
```

在循环双链表中查找某个结点的函数如下:

```

void find1(VAR head; dnode; VAR x; integer);
dnode * head;
int x;
{
    dnode * p;
    if (head->data == x) printf("结点找到了! \n");
        /* 若头结点为要找的结点,则显示相应信息并返回 */
    else /* 否则查找该结点 */
    {
        p = head->right;
        while (p->data != x && p != head) p = p->right;
            /* p = p->right 亦可改为 p = p->left */
        if (p == head)
            printf("结点找到了! \n");
        else
            printf("未找到! \n");
    }
}

```

4. 在双链表中插入一个结点

在双链表中第 i 个结点 ($i \geq 0$) 之后插入一个元素为 x 的结点的函数如下:

```

void insert(head, i, x)
dnode * head;
int i, x;
{
    dnode * s, * p;
    int j;
    s = (dnode *) malloc(sizeof(dnode)); /* 建立一个待插入的结点,由 s 指向 */
    s->data = x;
    if (i == 0) /* 如果 i=0,则将 s 所指结点插入到表头后返回 */
    {
        s->right = head;
        head->left = s;
        head = s;
    }
    else
    {
        p = head; j = 1; /* 在双链表中查找第 i 个结点,由 p 所指向 */
        while (p != NULL && j < i)
        {
            j++;
            p = p->right;
        }
    }
}

```

```

if (p != NULL) /* 若查找成功,则把 s 插入到 p 之后 */
{
    if (p->right == NULL) /* 若 p 是最后一个结点,则直接把 s 链接起来 */
    {
        p->right = s;
        s->right = NULL;
        s->left = p;
    }
    else
    {
        s->right = p->right;
        p->right->left = s;
        p->right = s;
        s->left = p;
    }
}
else
    printf("未找到! \n");
}
}

```

5. 从双链表中删除一个结点

从双链表中删除一个其值等于给定值 x 的结点的函数如下:

```

void delete(head, x)
dnode * head;
int x;
{
    dnode * p, * q;
    if (head == NULL) printf("链表下溢! \n"); /* 如果双链表为空,则下溢处理 */
    if (head->data == x) /* 如果表头结点值等于 x,则删除之 */
    {
        p = head;
        head = head->right;
        head->left = NULL;
        free(p);
    }
    else
    {
        p = head->right; /* 从第二个结点开始查找其值为 x 的结点 */
        while (p != NULL && p->data != x)
            if (p->data == x) /* 在查找时, p 指向该结点 */
                p = p->right;
        if (p != NULL) /* 若找到了该结点,则进行删除处理 */
            if (p->right == NULL) /* 若 p 所指结点是最后一个结点,则直接删除之 */
                {

```



```

    p->left->right=NULL;
    free(p);
}
else /* 若 p 所指结点不是最后结点,则把 p 的左右两个结点链接起来 */
{
    p->left->right=p->right;
    p->right->left=p->left;
    free(p);
}
else /* 未找到时,显示相应信息 */
    printf("未找到! \n");
}
}

```

3.1.3 链栈和链队

1. 链栈

链栈是栈的链接存储表示,或者说它是只允许在表头进行插入和删除运算的单链表。链栈的单链表的表头指针叫做栈顶指针。

图 3.5 中,图(a)是一个链栈示意图,其中 HS 表示栈顶指针;图(b)是图(a)入栈一个新结点的结果;图(c)是图(a)出栈一个结点后的结果。

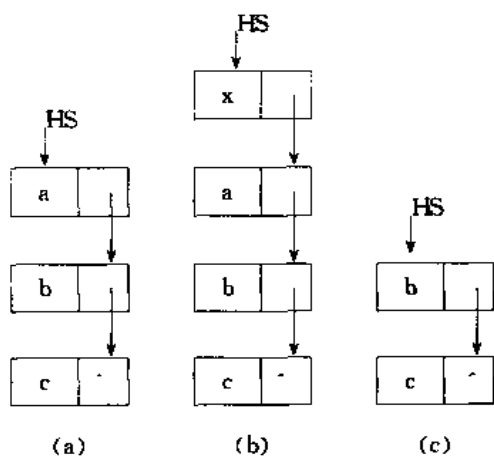


图 3.5 链栈的入栈和出栈操作示意图

链栈的几个基本操作函数如下。

(1) 链栈的入栈 $\text{push}(\text{HS}, x)$

在栈顶指针是 HS 的链栈中插入一个值为 x 的结点的函数如下:

```

void push(HS, x)
node * HS;
int x;
{

```

```

node * s;
s=(node *)malloc(sizeof(node)); /* 建立一个结点指针 */
s->data=x;
s->next=HS;
HS=s;
;

```

(2) 链栈的出栈 pop(HS)

在栈顶指针是 HS 的链栈中出栈一个结点的函数如下:

```

int pop(HS)
node * HS;
;
int x;
node * p;
if (HS == NULL) printf("向下溢出\n");
else
;
x=HS->data;
p=HS;
HS=HS->next;
free(p);
return(x);
;

```

(3) 判定栈是否为空 empty(HS)

判定栈顶指针是 HS 的链栈是否为空的函数如下:

```

int empty(HS)
node * HS;
;
if (HS == NULL) return(1);
else return(0);
;

```

2. 链队

链队是队列的链接存储表示,或者说它是只允许在表尾进行插入,在表头进行删除运算的单链表。一个链队需要队首和队尾两个指针,其中队首指针 first 指向单链表的表头,队尾指针 rear 指向单链表的表尾。

first 和 rear 均为 node 的指针类型,定义以下结构类型:

```

struct linkqueue
;
node * first, * rear;
;

```

则每个链队对应有一个 linkqueue 的指针类型的变量,给出其队首和队尾两个指针值。

图 3.6 中,图(a)是一个链队示意图,其中 HQ 是一个链队变量;图(b)是图(a)入队一个新结点的结果;图(c)是图(a)出队一个结点后的结果。

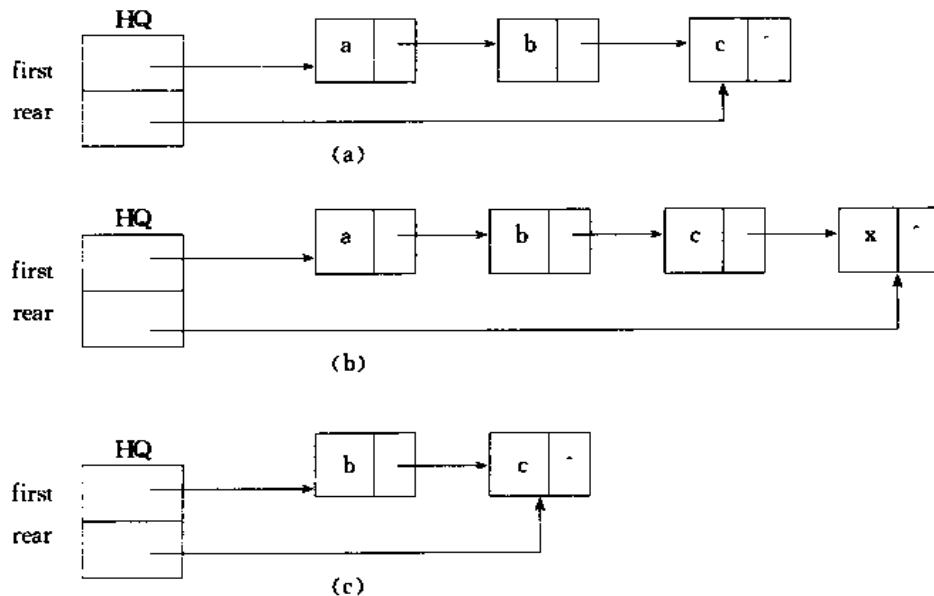


图 3.6 链队的入队和出队操作示意图

链队的几个基本操作函数如下:

(1) 链队的插入 insert(HQ,x)

在 HQ 的链队中插入一个值为 x 的结点的函数如下:

```
void insert(HQ,x)
struct linkqueue * HQ;
int x;
{
    struct linkqueue * s;
    s=(struct linkqueue *)malloc(sizeof(struct linkqueue));
    s->data=x;
    s->next=NULL;
    if (HQ->rear==NULL)
    {
        HQ->first=s;
        HQ->rear=s;
    }
    else
    {
        HQ->rear->next=s;
        HQ->rear=s;
    }
}
```

(2) 链栈的删除 delete(HQ)

在 HQ 的链队中删除一个结点的函数如下:

```
int delete(HQ);
struct linkqueue * HQ;

struct linkqueue * p;
int x;
if (HQ->first == NULL) printf("向下溢出\n");
else
{
    x = HQ->first->data;
    p = HQ->first;
    if (HQ->first == HQ->rear) /* 只有一个结点的情况 */
    {
        HQ->first = NULL;
        HQ->rear = NULL;
    }
    else
        HQ->first = HQ->first->next;
    free(p);
    return(x);
}
```

(3) 判定链队是否为空 qempty(HQ)

判定 HQ 的链队是否为空的函数如下:

```
int qempty(HQ)
struct linkqueue * HQ;
{
    if (HQ->first == NULL) return(1);
    else return(0);
}
```

3.2 基本题

3.2.1 单项选择题

1. 不带头结点的单链表 head 为空的判定条件是 ①。

- | | |
|----------------------|----------------------|
| A. head = NULL | B. head->next = NULL |
| C. head->next = head | D. head != NULL |

答: ①A

2. 带头结点的单链表 head 为空的判定条件是 ①。

- A. head=NULL B. head->next=NULL
C. head->next=head D. head!=NULL

答: ①B

3. 非空的循环单链表 head 的尾结点(由 p 所指向)满足 ①。

- A. p->next=NULL B. p=NULL
C. p->next=head D. p=head

答: ①C

4. 在循环双链表的 p 所指结点之后插入 s 所指结点的操作是 ①。

- A. p->right=s; s->left=p; p->right->left=s; s->right=p->right;
B. p->right=s; p->right->left=s; s->left=p; s->right=p->right;
C. s->left=p; s->right=p->right; p->right=s; p->right->left=s;
D. s->left=p; s->right=p->right; p->right->left=s; p->right=s;

答: ①D

5. 在一个单链表中, 已知 q 所指结点是 p 所指结点的前驱结点, 若在 q 和 p 之间插入 s 结点, 则执行 ①。

- A. s->next=p->next; p->next=s;
B. p->next=s->next; s->next=p;
C. q->next=s; s->next=p;
D. p->next=s; s->next=q;

答: ①C

6. 在一个单链表中, 若 p 所指结点不是最后结点, 在 p 之后插入 s 所指结点, 则执行 ①。

- A. s->next=p; p->next=s;
B. s->next=p->next; p->next=s;
C. s->next=p->next; p=s;
D. p->next=s; s->next=p;

答: ①B

7. 在一个单链表中, 若删除 p 所指结点的后续结点, 则执行 ①。

- A. p->next=p->next->next;
B. p=p->next; p->next=p->next->next;
C. p->next=p->next;
D. p=p->next->next

答: ①A

8. 假设双链表结点的类型如下:

```
typedef struct linknode
{
    int data;           /* 数据域 */
```

```

struct linknode *llink;    /* llink 是指向前驱结点的指针域 */
struct linknode *rlink;    /* rlink 是指向后续结点的指针域 */
} bnode

```

下面给出的算法段是要把一个 q 所指新结点作为非空双向链表中的 p 所指结点的前驱结点插入到该双链表中,能正确完成要求的算法段是 ①。

- A. $q \rightarrow rlink = p;$
 $q \rightarrow llink = p \rightarrow llink;$
 $p \rightarrow llink = q;$
 $p \rightarrow llink \rightarrow rlink = q;$
- B. $p \rightarrow llink = q;$
 $q \rightarrow rlink = p;$
 $p \rightarrow llink \rightarrow rlink = q;$
 $q \rightarrow llink = p \rightarrow llink;$
- C. $q \rightarrow llink = p \rightarrow llink;$
 $q \rightarrow rlink = p;$
 $p \rightarrow llink \rightarrow rlink = q;$
 $p \rightarrow llink = q;$
- D. 以上都不对

答:①C

9. 从一个具有 n 个结点的单链表中查找其值等于 x 结点时,在查找成功的情况下,需平均比较 ① 个结点。

- A. n B. $n/2$ C. $(n-1)/2$ D. $(n+1)/2$

答:①D

10. 在一个具有 n 个结点的有序单链表中插入一个新结点并仍然有序的时间复杂度是 ①。

- A. $O(1)$ B. $O(n)$ C. $O(n^2)$ D. $O(n \log_2 n)$

答:①B

11. 给定有 n 个元素的向量,建立一个有序单链表的时间复杂度是 ①。

- A. $O(1)$ B. $O(n)$ C. $O(n^2)$ D. $O(n \log_2 n)$

答:①C

12. 向一个栈顶指针为 HS 的链栈中插入一个 s 所指结点时,则执行 ①。

- A. $HS \rightarrow next = s;$
B. $s \rightarrow next = HS \rightarrow next; HS \rightarrow next = s;$
C. $s \rightarrow next = HS; HS = s;$
D. $s \rightarrow next = HS; HS = HS \rightarrow next;$

答:①C

13. 从一个栈顶指针为 HS 的链栈中删除一个结点时,用 x 保存被删结点的值,则执行

①。

- A. $x=HS; HS=HS \rightarrow next;$
 B. $x=HS \rightarrow data;$
 C. $HS=HS \rightarrow next; x=HS \rightarrow data;$
 D. $x=HS \rightarrow data; HS=HS \rightarrow next$

答:①D

14. 在一个链队中,假设 f 和 r 分别为队首和队尾指针,则插入 s 所指结点的运算时 ①。

- A. $f \rightarrow next=s; f=s;$
 B. $r \rightarrow next=s; r=s;$
 C. $s \rightarrow next=r; r=s;$
 D. $s \rightarrow next=f; f=s;$

答:①B

15. 在一个链队中,假设 f 和 r 分别为队首和队尾指针,则删除一个结点的运算时 ①。

- A. $r=f \rightarrow next;$
 B. $r=r \rightarrow next;$
 C. $f=f \rightarrow next;$
 D. $f=r \rightarrow next$

答:①C

3.2.2 填空题(将正确的答案填在相应的空中)

1. 单链表是 ① 的链接存储表示。

答:①线性表

2. 可以使用 ① 表示树形结构。

答:①双链表

3. 在双链表中,每个结点有两个指针域,一个指向 ①,另一个指向 ②。

答:①前驱结点 ②后续结点

4. 在一个单链表中的 p 所指结点之前插入一个 s 所指结点时,可执行如下操作:

- (1) $s \rightarrow next = \text{①};$
 (2) $p \rightarrow next = s;$
 (3) $t = p \rightarrow data;$
 (4) $p \rightarrow data = \text{②};$
 (5) $s \rightarrow data = \text{③};$

答:① $p \rightarrow next$ ② $s \rightarrow data$ ③ t 5. 在一个单链表中删除 p 所指结点时,应执行以下操作:

$q = p \rightarrow next;$
 $p \rightarrow data = p \rightarrow next \rightarrow data;$
 $p \rightarrow next = \text{①};$

free(q);

答: ① p->next->next

6. 带有一个头结点的单链表 head 为空的条件是 ①。

答: ① head->next=NULL

7. 在一个单链表中 p 所指结点之后插入一个 s 所指结点时, 应执行 s->next= ① 和 p->next= ② 的操作。

答: ① p->next ② s

8. 非空的循环单链表 head 的尾结点(由 p 所指向), 满足条件 ①。

答: ① head->next=p

9. 在栈顶指针为 HS 的链栈中, 判定栈空的条件是 ①。

答: ① HS==NULL

10. 在栈顶指针为 HS 的链栈中, 计算该链栈中结点个数的函数是 ①。

答: ①

```
int count(HS)
node * HS;
{
    node * p;
    int n=0;
    p=HS;
    while (p!=NULL)
    {
        n++;
        p=p->next;
    }
    return(n);
}
```

11. 在 HQ 的链队中, 判定只有一个结点的条件是 ①。

答: ① HQ->front==HQ->rear

12. 在 HQ 的链队中, 计算该链队中结点个数的函数是 ①。

答: ①

```
int count(HQ)
struct linkqueue * HQ;
{
    struct linkqueue * p;
    int n;
    p=HQ->first;
    if (p==NULL) return(0);
    n=1;
```



```

while (p != HQ->rear)
{
    n++;
    p = p->next;
}
return(n);
}

```

13. 对于一个具有 n 个结点的单链表, 在已知 p 所指结点后插入一个新结点的时间复杂度是 ①; 在给定值为 x 的结点后插入一个新结点的时间复杂度是 ②。

答: ① $O(1)$ ② $O(n)$

3.3 习题解析

3.3.1 单链表

1. 有一个单链表(不同结点的数据域值可能相同), 其头指针为 $head$, 编写一个函数计算数据域为 x 的结点个数。

解: 本题是遍历通过该链表的每个结点, 每遇到一个结点, 结点个数加 1, 结点个数存储在变量 n 中。实现本题功能的函数如下:

```

int count(head)
node * head;
{
    node * p;
    int n = 0;
    p = head;
    while (p != NULL)
    {
        if (p->data == x) n++;
        p = p->next;
    }
    return(n);
}

```

2. 已知一个单链表如图 3.7 所示, 编写一个函数将该单链表复制一个拷贝。

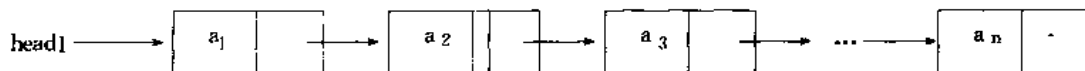


图 3.7 一个单链表

解: 本题的算法思想是依次查找该单链表(其头指针为 $head1$)中的每个结点, 对每个结点复制一个新结点并链接到新的单链表(其头指针为 $head2$)中。实现本题功能的函数如下:

```

void copy(head1, head2)
node * head1, * head2;
{
    node * p, * q;
    head2 = (node *) malloc(sizeof(node)); /* 建立一个头结点 */
    q = head2; p = head1;
    while (p != NULL)
    {
        s = (node *) malloc(sizeof(node)); /* 复制一个新结点 s */
        s->data = p->data;
        q->next = s; /* 把 s 链接到 q 之后 */
        q = s;
        p = p->next;
    }
    q->next = NULL; /* 将最后一个结点的 next 域置为 NULL */
    p = head2; /* 删除头结点 */
    head2 = head2->next;
    free(p);
}

```

* 3. 有一个单链表 L (至少有 1 个结点), 其头结点指针为 head, 编写一个函数将 L 逆置, 即最后一个结点变成第一个结点, 原来倒数第二个结点变成第二个结点, 如此等等。

解: 本题采用的算法是: 从头到尾扫描单链表 L, 将第一个结点的 next 域置为 NULL, 将第二个结点的 next 域指向第一个结点, 将第三个结点的 next 域指向第二个结点, 如此等等, 直到最后一个结点, 便用 head 指向它, 这样达到了本题的要求。实现本题功能的函数如下:

```

void invert(head)
node * head;
{
    node * p, * q, * r;
    p = head;
    q = p->next;
    while (q != NULL) /* 当 L 没有后续结点时终止 */
    {
        r = q->next;
        q->next = p;
        p = q;
        q = r;
    }
    head->next = NULL;
    head = p; /* p 指向 L 的最后一个结点, 现改为头结点 */
}

```

* 4. 已知一个循环单链表如图 3.8 所示,编写一个函数将所有箭头方向取反。

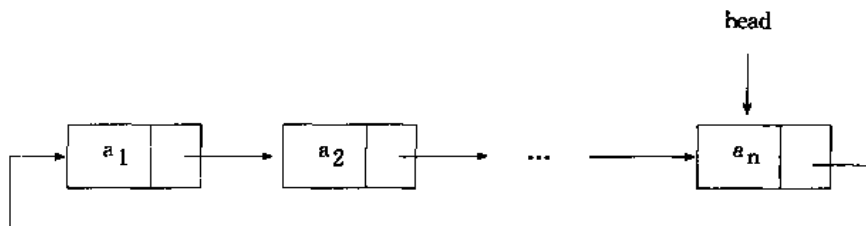


图 3.8 一个循环单链表

解:本题采用的算法是:从头到尾扫描循环单链表 L,将第一个结点的 next 域置为 NULL,将第二个结点的 next 域指向第一个结点,将第三个结点的 next 域指向第二个结点,如此等等,直到最后一个结点,使用 head 指向它,这样达到了本题的要求,因为不是普通单链表,所以判定最后一个结点时不能用 $p \rightarrow \text{next} = \text{NULL}$ 作为条件,而是用 q 指向第一个结点,以 $p! = q$ 作为条件。

实现本题功能的函数如下:

```
void invert()
{
    node *p, *q, *r;
    p=head;
    q=p->next;    /* 指向数据为 a1 的结点 */
    while (p!=q)  /* *p 不是第一个结点时循环 */
    {
        r=head;
        while (r->next!=p) r=r->next;
        p->next=r;    /* 结点的 next 逆向 */
        p=p->next;
    }
    q->next=head;    /* 数据域为 a1 的结点 next 指向 head 所指的结点 */
}
```

5. 如果以单链表表示集合,设计算法建立先后输入的两个集合的差。

解:假设两个已知集合 A 和 B,根据集合运算的规则可知,集合 $A-B$ 中包含所有属于集合 A 而不属于集合 B 的元素。因此,为了求 $A-B$,需先建立表示集合 A 的单链表,然后对集合 B 中的每个元素 x,在集合 A 的链表中进行查找,若存在和 x 相同的元素,则从该链表中删除。其中函数 creatlist()用于建立一个单链表;函数 subs()用于计算两个集合之差。实现本题功能的程序如下:

```
#include <stdio.h>
typedef struct linknode
{
    int data;
    struct linknode *next;
```

```

    } node;

node * creatlist() /* 建立单链表 */
{
    node * head, * r, * s;
    int x;
    head = (node *) malloc(sizeof(node)); /* 建立单链表的头结点,由 head 指针所指向 */
    r = head;
    printf("输入系列整数,以 0 标志结束\n");
    scanf("%d", &x);
    while (x != 0) /* x=0 则退出 while 循环 */
    {
        s = (node *) malloc(sizeof(node));
        s->data = x;
        r->next = s;
        s->next = NULL;
        r = s;
        scanf("%d", &x);
    }
    r->next = NULL;
    s = head; /* 删除头结点 */
    head = head->next;
    free(s);
    return(head);
}

void subs()
{
    node * p, * p1, * p2, * q, * heada, * headb;
    heada = creatlist();
    headb = creatlist();
    p = heada;
    p1 = p; /* p1 指向 p 所指结点的前一个结点,开始时均指向头结点 */
    while (p1 != NULL)
    {
        q = headb;
        while (q->data != p->data && q != NULL) q = q->next;
        /* 判定 p 所指结点是否在 B 中 */
        if (q != NULL) /* p 所指结点的元素在 B 中,则要删除之 */
        {
            if (p == heada)
            {
                heada = heada->next; /* 该结点为头结点,删除它 */
                p1 = heada;
            }
        }
    }
}

```

```

    }
    else if (p->next==NULL) p1->next=NULL; /* 该结点为最后一结点,删除它 */
        else p1->next=p->next;

    p2=p->next; /* p2 作为临时变量,保存 p 所指结点的下一个结点 */
    p->next=NULL; /* 清除 p 所指结点 */
    free(p);
    p=p2;
}
else /* p 所指结点的元素不在 B 中,则直接下移 p */
{
    p1=p;
    p=p->next;
}
}
p=heada; /* 显示删除后的结果 */
if (p==NULL)
    printf("两集合相减的结果为空\n");
else
    printf("两集合相减的结果:\n");
while (p!=NULL)
{
    printf(" %d ",p->data);
    p=p->next;
}
}

main()
{
    subs();
}

```

6. 已知两个整数集合 A 和 B,它们的元素分别依元素值递增有序存放在两个单链表 HA 和 HB 中,编写一个函数求出这两个集合的并集 C,并要求表示集合 C 的链表的结点仍依元素值递增有序存放。

解:假设 HA 和 HB 的头指针分别为 ha 和 hb,先设置一个空的循环单链表 HC,头指针为 hc,之后依次比较 HA 和 HB 的当前结点的元素值,将较小元素值的结点复制一个并链接到 HC 的末尾,若相等,则将其中之一复制一个并链接到 HC 的末尾。当 HA 或 HB 为空后,把另一个链表的余下结点都复制并链接到 HC 的末尾。实现本题功能的函数如下:

```

void union(ha,hb,hc)
node * ha, * hb, * hc;
{
    node * p, * q, * r, * s;
    hc=(node *)malloc(sizeof(node));

```

```

    /* 建立一个头结点,r 总是指向 HC 链表的最后一个结点 */
    r=hc;p=ha;q=hb;
    while (p!=NULL && q!=NULL)
    {
        if (p->data<q->data)
        {
            s=(node *)malloc(sizeof(node));
            s->data=p->data;
            r->next=s;
            r=s;
            p=p->next;
        }
        else if (p->data>q->data)
        {
            s=(node *)malloc(sizeof(node));
            s->data=q->data;
            r->next=s;
            r=s;
            q=q->next;
        }
        else /* p->data==q->data 的情况 */
        {
            s=(node *)malloc(sizeof(node));
            s->data=q->data;
            r->next=s;
            r=s;
            p=p->next;
            q=q->next;
        }
    }
    if (p==NULL) /* 把 q 及之后的结点复制到 HC 中 */
    while (q!=NULL)
    {
        s=(node *)malloc(sizeof(node));
        s->data=q->data;
        r->next=s;
        r=s;
        q=q->next;
    }
    if (q==NULL) /* 把 p 及之后的结点复制到 HC 中 */
    while (p!=NULL)
    {
        s=(node *)malloc(sizeof(node));

```

```

    s->data=p->data;
    r->next=s;
    r=s;
    p=p->next;
}
r->next=NULL;
s=hc;
hc=hc->next;
free(s); /* 删除头结点 */
}

```

7. 已知有两个单链表 A 和 B, 其头指针分别为 heada 和 headb, 编写一个函数从单链表 A 中删除自第 i 个元素起的共 len 个元素, 然后将它们插入到单链表 B 的第 j 个元素之前。

解: 首先编写一个从单链表 A 中删除自第 i 个元素起的共 len 个元素的函数如下:

```

node *del(heada,i,len)
node *heada;
int i,len;
{
    node *p,*q;
    int k;
    if (i==1) /* 删除该单链表的前 len 个元素 */
        for (k=1;k<=len;k++)
        {
            q=heada; /* q 指向要删除的结点 */
            heada=heada->next;
            free(q);
        }
    else
    {
        p=heada;
        for (k=1;k<=i-2;k++) p=p->next; /* p 指向要删除的一组结点的前一个结点 */
        for (k=1;k<=len;k++) /* 删除 len 个结点 */
        {
            q=p->next; /* q 指向要删除的结点 */
            p->next=q->next;
            free(q);
        }
    }
    return(heada);
}

```

把一个头指针为 heada 的单链表插入到单链表 B 的第 j 个元素之前的函数如下:

```

node *insert(heada,headb,j)

```

```

node * heada, * headb;
int j;
{
    node * p, * q;
    p = heada;
    while (p != NULL) p = p->next; /* p 指向最后一个结点 */
    if (j == 1) /* 若 j 为 1, 则把 heada 的链表插到 headb 链表之前 */
    {
        p->next = headb;
        headb = heada;
    }
    else
    {
        q = headb;
        for (k = 1; k <= j - 2; k++) q = q->next; /* q 指向其后插入 heada 链表的结点 */
        p->next = q->next; /* 把 heada 链表插入到 q 结点之后 */
        q->next = heada;
    } /* 最后生成的链表的头指针为 headb */
    return(headb);
}

```

最后完成本题功能的函数如下:

```

node * fun(heada, headb, i, j, len)
node * heada, * headb;
int i, j, len;
{
    node * p, * q;
    p = del(heada, i, len);
    q = insert(p, headb, j);
    return(q);
}

```

* 8. 有一个有序单链表(从小到大排列), 表头指针为 head, 编写一个函数向该单链表中插入一个元素为 x 的结点, 使插入后该链表仍然有序。

解: 本题算法的思想是先建立一个待插入的结点, 然后依次与链表中的各结点的数据域比较大小, 找到插入该结点的位置, 最后插入该结点。实现本题功能的函数如下:

```

node * Insertorder(head, x)
node * head;
int x;
{
    node * s, * p, * q;
    s = (node *) malloc(sizeof(node)); /* 建立一个待插入的结点 */
    s->data = x;
}

```



```

s->next=NULL;
if (head==NULL || x<head->data) /* 若单链表为空或 x 小于第一个结点的 data 域 */
{
    s->next=head;          /* 则把 s 结点插入到表头后面 */
    head=s;
}
else
{
    q=head;    /* 为 s 结点寻找插入位置, p 指向待比较的结点, q 指向 p 的前驱结点 */
    p=q->next;
    while (p!=NULL && x>p->data) /* 若 x 小于 p 所指结点的 data 域值 */
        /* 则退出 while 循环 */

        if (x>p->data)
        {
            q=p;
            p=p->next;
        }
    s->next=p; /* 将 s 结点插入到 q 和 p 之间 */
    q->next=s;
}
return(head);
}

```

* 9. 编写一个函数交换单链表中 p 所指向的位置和其后续位置上的两个结点, head 指向该链表的表头, p 指向该链表中的某一结点。

解: 本题的算法思想是: 如果 p 存在后续结点, 看它是否是头结点, 如果是, 则交换后要改变该链表的 head, 若不是头结点, 则直接交换。实现本题功能的函数如下:

```

node * swap(head, p)
node * head, * p;
{
    node * q, * r, * s;
    q=p->next;
    if (q!=NULL) /* 若 p 存在后续结点, 则进行相应处理 */
    {
        if (p==head) /* 若 p 指向头结点, 则将该链表的前两个结点交换位置 */
        {
            head=head->next;
            s=head->next;
            head->next=p;
            p->next=s;
        }
        else /* 若 p 指向第二个之后的结点 */
        {

```

```

    r=head;          /* 查找 p 的前驱结点 */
    while (r->next!=p) r=r->next;
    r->next=q;        /* 交换 p 和 q 的位置 */
    p->next=q->next;
    q->next=p;
}
return(head);
}
else
    return(NULL);    /* 若 p 不存在后续结点,则返回 NULL */
}

```

* 10. 有两个循环单链表,链表头指针分别为 head1 和 head2,如图 3.9 所示,编写一个函数将链表 head1 链接到链表 head2 之后,链接后的链表仍保持是循环链表形式。

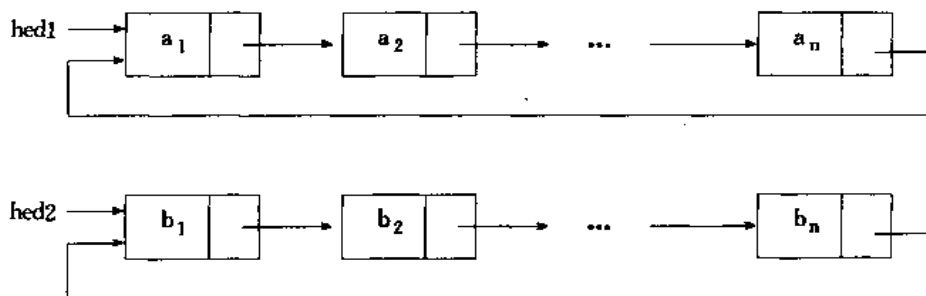


图 3.9 两个循环单链表

解:本题的算法思想是:先找到两链表的尾指针,将第一个链表的尾指针与第二个链表的头结点链接起来,再使之成为循环的。实现本题功能的函数如下:

```

node *link(head1,head2)
node *head1, *head2;
{
    node *p, *q;
    p=head1;          /* 找到 head1 的表尾,用 p 指向它 */
    while (p->next!=head1) p=p->next;
    q=head2;          /* 找到 head2 的表尾,用 q 指向它 */
    while (q->next!=head2) q=q->next;
    p->next=head2;     /* 将 head2 链表链接到 head1 链表之后 */
    q->next=head1;     /* 仍保持是循环链表 */
    return(head1);
}

```

* 11. 编写一个函数将一个头结点指针为 a 的单链表 A 分解成两个单链表 A 和 B,其头结点指针分别为 a 和 b,使得 A 链表中含有原链表 A 中序号为奇数的元素,而 B 链表中含有原链表 A 中序号为偶数的元素,且保持原来的相对顺序。

解:其函数是将单链表 A 中的所有偶数字号的结点删除,并在删除时把这些结点链接起来构成单链表 B。实现本题功能的函数如下:

```
void disa(a,b)
node *a,&b;
{
    node *r,*p,*q;
    p=a;
    b=a->next;
    r=b;
    while (p!=NULL && p->next!=NULL)
    {
        q=p->next;          /* q 指向偶数字号的结点 */
        p->next=q->next;     /* 将 q 从原 A 中删除掉 */
        r->next=q;           /* 将 q 结点链接到 B 链表的末尾 */
        r=q;                /* r 总是指向 B 链表的最后一个结点 */
        p=p->next;           /* p 指向原链表 A 中的奇数字号的结点 */
    }
    r->next=NULL;          /* 将生成 B 链表中的最后一个结点的 next 域置空 */
}
```

* 12. 已知两个单链表 A 与 B 分别表示两个集合,其元素递增排列,编写一个函数求出 A 和 B 的交集 C,要求 C 同样以元素值递增的单链表形式存储。

解:交集指的是两个单链表的元素值相同的结点的集合,为了操作方便,先让单链表 C 带有一个头结点 c,最后将其删除掉。实现本题功能的函数如下:

```
node * Inter(a,b)
node *a,*b;
{
    node *p,*q,*r,*s,*c;
    c=(node *)malloc(sizeof(node)); /* 建立单链表 C 的头指针 c */
    r=c;p=a;q=b;
    while (p!=NULL && q!=NULL)
    {
        if (p->data<q->data) p=p->next;
        else if (p->data>q->data) q=q->next;
        else { /* 此时找到了一个元素值相同的结点,在 C 中生成一个结点 */
            s=(node *)malloc(sizeof(node));
            s->data=p->data;
            r->next=s; /* 把 s 结点链接到 C 的末尾 */
            r=s;      /* r 始终指向 C 链表的最后一个结点 */
            p=p->next;
            q=q->next;
        }
    }
    r->next=NULL;
```

```

c=c->next;          /* 删除 C 链表的头结点 */
return(c);
}

```

13. 有两个具有相同结点个数的单链表 A 和 B, $A=\{a_1, a_2, \dots, a_n\}$, $B=\{b_1, b_2, \dots, b_n\}$, 编写一个函数将其合并成一个链表 C, $C=\{a_1, b_1, a_2, b_2, \dots, a_n, b_n\}$ 。

解: 假设 a, b, c 分别为指向 A, B, C 单链表头结点的指针, 本题的算法思想是: 循环遍历两个链表, 每循环一次, 复制一个 A 的结点到 C 中, 再复制一个 B 的结点到 C 中, 如此直到链表遍历完为止。实现本题功能的函数如下:

```

node * sum(a, b)
node * a, * b;
{
    node * r, * s, * p, * q, * c;
    c=(node *)malloc(sizeof(node)); /* 生成一个头结点 c */
    r=c; p=a; q=b;
    while (p!=NULL)
    {
        s=(node *)malloc(sizeof(node));
        /* 复制一个与 A 链表中结点相同的结点, 把它链接到 C 中 */
        s->data=p->data;
        r->next=s;
        p=p->next;
        r=s; /* r 始终指向 C 链表的最后一个结点 */
        new(s); /* 复制一个与 B 链表中结点相同的结点, 把它链接到 C 中 */
        s->data=q->data;
        r->next=s;
        q=q->next;
        r=s; /* r 始终指向 C 链表的最后一个结点 */
    }
    r->next=NULL;
    c=c->next; /* 删除 C 链表的头结点 */
    return(c);
}

```

14. 有一个单链表, 其结点的元素值以非递减有序排列, 编写一个函数删除该单链表中多余的元素值相同的结点。

解: 本题采用的算法是: 从头到尾扫描该单链表, 并作这样的操作: 若当前结点的元素值与后续结点的元素值不相等, 则指针后移, 否则删除该后续结点, 直到扫描所有的结点。实现本题功能的函数如下:

```

node * del(head)
node * head;
{

```

```

node *q;
if (head! = NULL)
{
    /* 当前结点的元素值与后续结点的元素值不相等,则指针后移,
       否则删除该后续结点 */
    while (p->next! = NULL)
        if (p->data! = p->next->data) p=p->next;
        else
        {
            q=p->next;
            p->next=q->next;
            free(q);
        }
    return(head);
}

```

15. 假设在长度大于1的循环单链表中,既无头结点也无头指针, p 为指向该链表中某个结点的指针,编写一个函数删除该结点的前驱结点。

解:本题利用循环单链表的特点,通过 p 指针可循环找到其前驱结点 q 及 q 的前驱结点 r ,然后将其删除。实现本题功能的函数如下:

```

node *del(p)
node *p;
{
    node *q, *r;
    q=p;          /* 查找p结点的前驱结点,用q指针指向 */
    while (q->next! = p) q=q->next;
    r=q;          /* 查找q结点的前驱结点,用r指针指向 */
    while (r->next! = q) r=r->next;
    r->next=p;     /* 删除q所指的结点 */
    free(q);
    return(p);
}

```

16. 假设 $X=(x_1, x_2, \dots, x_n)$ 和 $Y=(y_1, y_2, \dots, y_m)$ 是两个单链表,编写一个函数将这两个单链表合并成一个单链表,结点的物理位置不变,这个新的单链表为:

$$Z = \begin{cases} (x_1, y_1, x_2, y_2, \dots, x_m, y_m, x_{m+1}, \dots, x_n) & \text{当 } n \geq m \text{ 时} \\ (x_1, y_1, x_2, y_2, \dots, x_n, y_n, y_{n+1}, \dots, y_m) & \text{当 } n < m \text{ 时} \end{cases}$$

解:假设 X, Y 和 Z 链表分别具有头结点的指针 x, y 和 z 。实现本题功能的函数如下:

```

node *link(x, y)
node *x, *y;
{

```

```

node *r, *s, *p, *q, *z;
z=(node *)malloc(sizeof(node));    /* 建立一个头结点 */
r=z;p=x;q=y;
while (p!=NULL || q!=NULL)
{
    if (p!=NULL) /* 如果 X 链表还存在可取的结点,则复制一个同样的结点链接到 Z 中 */
    {
        s=(node *)malloc(sizeof(node));
        s->data=p->data;
        r->next=s;
        r=s;
        p=p->next;
    }
    if (q!=NULL) /* 如果 Y 链表还存在可取的结点,则复制一个同样的结点链接到 Z 中 */
    {
        s=(node *)malloc(sizeof(node));
        s->data=q->data;
        r->next=s;
        r=s;
        q=q->next;
    }
}
r->next=NULL;
z=z->next;    /* 删除头结点 */
return(z);
}

```

* 17. 假设有两个已排序的单链表 A 和 B,编写一个函数将它们合并成一个链表 C 而不改变其排序性

解:这里采用链表合并的方法,设原两链表的头指针分别为 p 和 q,每次比较 p->data 和 q->data 的值,把值较小的结点作为新链表的结点,这一过程直到一个链表为空为止。当一个链表为空而另一个链表不为空时,只要将不空的链表指针赋给新链表中最后一个结点的指针域即可。实现本题功能的函数如下:

```

node * mergelink(p,q)
node * p, * q;
{
    node * h, * r;
    h=(node *)malloc(sizeof(node));    /* 建立头结点 */
    h->next=NULL;
    r=h;
    while (p!=NULL && q!=NULL)
        if (p->data<=q->data)

```

```

    {
        r->next=p;
        r=p;
        p=p->next;
    }
    else
    {
        r->next=q;
        r=q;
        q=q->next;
    }
    if (p==NULL) r->next=q;
        /* 若 A 链表的结点已取完,则把 B 的所有余下的结点链接到 C 中 */
    if (q==NULL) r->next=p;
        /* 若 B 链表的结点已取完,则把 A 的所有余下的结点链接到 C 中 */
    p=h->next;
    free(h);
}

```

18. 某百货公司仓库中有一批电视机,按其价格从低到高的次序构成一个循环单链表,每个结点有价格、数量和链指针三个域,现新到 m 台价格为 h 的电视机,编写一个函数修改原循环链表。

解:依题意建立如下链表结构:

```

struct list
{
    float price;      /* 价格域 */
    int number;       /* 数量域 */
    struct list * next; /* 链指针域 */
}

```

其算法是先建立一个待插入的结点,然后在该循环单链表(假设其头指针为 $head$)中找到插入的位置,再把该结点插入。实现本题功能的函数如下:

```

struct list * insert(head,m,h)
struct list * head;
int m;
float h;
{
    struct list * q, * s;
    s = (struct list *) malloc(sizeof(struct list));
    s->price=h;
    s->number=m;
    /* 该链表为空时建立一个循环链表 */
    if (head==NULL)

```

```

{
    head=s;
    head->next=head; /* 构成一个循环链表 */
}
else
{
    /* 第一个结点的 price 域小于 h, 则把 s 所指结点作为第一个结点插入 */
    if (head->price>h)
    {
        q=head->next; /* 查找该循环链表的最后一个结点, 由 q 指向 */
        while (q!=head) q=q->next;
        s->next=head;
        head=s;
        q->next=head;
    }
    else
    {
        /* 查找相应的结点(q 所指向), 在之后插入 s 所指结点 */
        q=head->next;
        while (q->data<h && q!=head) q=q->next;
        s->next=q->next; 在 q 之后插入 s 结点 */
        q->next=s;
    }
}
return(head);
}

```

* 19. 输入一个名单, 有 n 个名字, 每个名字不超过 10 个字符, 按字典顺序将名单的序号排出单链表(即每个名字对应的链表值, 是其后继名字的序号, 最后一个链表值为 0)。

(1) 将原名单按每行为: 原序号、原名字、链表值的次序打印出来。

(2) 按链表顺序打印名单。

调试数据:

HILL
SLATER
WHITE
KULP
STEELE
HAAKE
LONKVICH
HATCHEB
FUERLE
ZIMMERMAN
CHALLSTROM

JACOBWITZ

READER

解:依题意建立如下单链表结构:

```
struct list
{
    int no;           /* 存放原序号 */
    char data[10];    /* 存放名字 */
    int val;          /* 存放链表值 */
    struct list * next;
};
```

先根据用户输入的名字建立一个单链表 head,其函数为 createlist(),它所建立的单链表的结点元素值是无序的,现在要对其排序,所需函数为 sort();给新生成的有序单链表的所有结点的 val 域赋值的函数为 setval()。根据上述函数建立的实现(1)功能和(2)功能的完整程序如下:

```
#include <stdio.h>
#include <string.h>
struct list
{
    int no;           /* 存放原序号 */
    char data[10];    /* 存放名字 */
    int val;          /* 存放链表值 */
    struct list * next;
};

struct list * createlist(int n)
{
    struct list * head, * p, * q, * s;
    int i;
    head = (struct list *) malloc(sizeof(struct list)); /* 建立一个头结点 */
    q = head;
    for (i = 1; i <= n; i++)
    {
        s = (struct list *) malloc(sizeof(struct list));
        printf("第%d个名字:", i);
        scanf("%s", s->data); /* 输入名字 */
        s->no = i;
        q->next = s;          /* 把 s 结点链接到单链表的末尾 */
        q = s;
    }
    q->next = NULL;
    p = head;
    head = head->next; /* 删除单链表表头结点 */
}
```

```

    free(p);
    return(head);
}

struct list * sort(head)
struct list * head;
{
    struct list * head1, * p, * s, * q, * r;
    head1=NULL; /* 建立一个生成的有序单链表的头结点的指针,开始时为 NULL */
    p=head;
    while (p!=NULL)
    {
        s=(struct list *)malloc(sizeof(struct list)); /* 建立一个待插入的结点 */
        strcpy(s->data,p->data);
        s->no=p->no;
        s->next=NULL;
        /* 若生成的有序单链表为空或 s 的元素值小于第一个结点的值,则把 s 结点插入到表头 */
        if (head1==NULL || strcmp(s->data,head1->data)<0)
        {
            s->next=head1;
            head1=s;
        }
        else
        {
            /* 为 s 结点寻找插入位置,r 指向待比较的结点,q 指向 r 的前驱结点 */
            q=head1;r=q;
            while (r!=NULL && strcmp(s->data,r->data)>0)
                if (strcmp(s->data,r->data)>0)
                {
                    q=r;
                    r=r->next;
                }
            if (r==NULL) /* s 作为最后一个结点 */
            {
                q->next=s;
                s->next=NULL;
            }
            else
            {
                s->next=r; /* 将 s 结点插入到 q 和 r 之间 */
                q->next=s; /* 此时 s 结点插入完毕,待插入下一个结点 */
            }
        }
    }
}

```

```

        p=p->next;                /* 移到下一个结点 */
    }
    return(head1);
}

struct list * setval(head)
struct list * head;
{
    struct list * p;
    p=head;
    while (p->next!=NULL)
    {
        p->val=p->next->no;
        p=p->next;
    }
    p->val=0;
    return(head);
}

main()
{
    struct list * head, * p;
    int n,i;
    printf("名字个数:");
    scanf("%d",&n);
    head=createlist(n);
    head=sort(head);
    head=setval(head);
    printf("实现功能(1)的结果:\n");
    for (i=1;i<=n;i++)
    {
        p=head;
        while (p->no!=i) p=p->next; /* 查找 no 值为 i 的结点 */
        printf("%5d%12s%5d\n",p->no,p->data,p->val);
    }
    printf("\n 实现功能(2)的结果:\n");
    p=head;
    while (p!=NULL)
    {
        printf("%10s\n",p->data);
        p=p->next;
    }
}

```

输入本题的调试数据,产生的结果如下:

实现功能(1)的结果:

1	HILL	12
2	SLATER	5
3	WHITE	10
4	KULP	7
5	STEELE	3
6	HAAKE	8
7	LONKVICH	13
8	HATCHEB	1
9	FUERLE	6
10	ZIMMERMAN	0
11	CHALLSTROM	9
12	JACOBWITZ	4
13	READER	2

实现功能(2)的结果:

CHALLSTROM
FUERLE
HAAKE
HATCHEB
HILL
JACOBWITZ
KULP
LONKVICH
READER
SLATER
STEELE
WHITE
ZIMMERMAN

20. 给出用单链表存储多项式的结构,并编写一个产生多项式链表的函数。

解:用单链表存储多项式的结构如下:

```
struct pnode
{
    int coef;          /* 系数 */
    int exp;           /* 指数 */
    struct pnode * link;
}
```

假设输入一组多项式的系数和指数,以输入系数为0标志结束,注意在建立多项式链表时,总是按照指数从大到小顺序排列的。产生该多项式链表(其头指针为 head)的函数如下:

```
struct pnode * poly()
{
    ...
```

```

struct pnode * head, * r, * s;
int m, n;
head = (struct pnode *) malloc(sizeof(struct pnode)); /* 建立一个头结点 */
r = head;
printf("输入系数 n 和指数 m:");
scanf("%d %d", &n, &m);
while (n != 0) /* 若 n=0 则退出 while 循环 */
{
    s = (struct pnode *) malloc(sizeof(struct pnode)); /* 建立一个新结点 s */
    s->coef = n;
    s->exp = m;
    r->link = s; /* 把 s 链接到 r 的后面 */
    r = s;
    printf("输入系数 n 和指数 m:");
    scanf("%d %d", &n, &m);
}
r->next = NULL;
head = head->next; /* 删除头结点 */
return(head);
}

```

21. 根据上题的多项式链表结构,编写一个函数实现两个多项式相加的运算。

解:假设两个多项式链表 A 和 B,其头指针分别是 heada 和 headb,这两个多项式相加后的多项式链表为 C,其头指针为 headc。实现本题功能的函数如下:

```

struct pnode * padd(heada, headb)
struct pnode * heada, * headb;
{
    struct pnode * headc, * p, * q, * s;
    int x;
    p = heada; q = headb;
    headc = (struct pnode *) malloc(sizeof(struct pnode));
    r = headc;
    while (p != NULL && q != NULL)
    {
        if (p->exp == q->exp) /* 两结点指数相等时,将两系数相加生成新结点插入 C 中 */
        {
            x = p->coef + q->coef;
            if (x != 0)
            {
                s = (struct pnode *) malloc(sizeof(struct pnode));
                s->coef = x;
                s->exp = p->exp;
                r->link = s;
            }
        }
    }
}

```

```

        r=s;
    }
    p=p->link;
    q=q->link;
}
else /* 两结点的指数不相等时,将其中较小系数的结点复制成一新结点插入 C 中 */
    if (p->exp<q->exp)
    {
        s=(struct pnode *)malloc(sizeof(struct pnode));
        s->coef=q->coef;
        s->exp=q->exp;
        r->link=s;
        r=s;
        q=q->link;
    }
    else
    {
        s=(struct pnode *)malloc(sizeof(struct pnode));
        s->coef=p->coef;
        s->exp=p->exp;
        r->link=s;
        r=s;
        p=p->link;
    }
}

while (p!=NULL) /* 复制 A 的余下部分 */
{
    s=(struct pnode *)malloc(sizeof(struct pnode)); /* 复制一个结点 s */
    s->coef=p->coef;
    s->exp=p->exp;
    r->link=s; /* 把 s 链接到 C 中 */
    r=s;
    p=p->link;
}

while (q!=NULL) /* 复制 B 的余下部分 */
{
    s=(struct pnode *)malloc(sizeof(struct pnode)); /* 复制一个结点 s */
    s->coef=q->coef;
    s->exp=q->exp;
    r->link=s; /* 把 s 链接到 C 中 */
    r=s;
    q=q->link;
}

```

```

    r->link=NULL;                                /* 最后结点的 link 域置空 */
    s=headc;                                       /* 删除 C 的头结点 */
    headc=headc->link;
    free(s);
    return(headc);
}

```

22. 编写一个程序,根据 test_data[]数组中给定的整数序列建立一个单链表,然后对该单链表进行排序(元素值从小到大排列),最后显示排序后的结果。

解:依题意,本程序由两个函数组成:create()函数用于根据数组的元素建立一个单链表;sort()函数对该单链表排序,其原理是,用 p 指针遍历所有结点,其前是已排好序的结点,其后是待排序的结点,从该链表中找出一个 data 值最小的结点并删除之,将删除结点插入到 p 所指结点的后面。采用两重循环完成排序工作,外层 while 判定是否还有待排序的结点,没有时退出循环,排序工作完成;有待排序结点时,则执行循环体一次,进行排序。实现本题功能的程序如下:

```

#include <stdio.h>
#include <stdlib.h>
typedef struct linknode
{
    int data;
    struct linknode * next;
} node;

node * create(int a[],int n)                                /* 建立一个单链表 */
{
    node * h,* q;
    for (h=NULL;n;n--)
    {
        q=(node *)malloc(sizeof(node));
        q->data=a[n-1];
        q->next=h;
        h=q;
    }
    return(h);
}

void sort(node * * h)
{
    node * p,* q,* r,* s,* h1;
    h1=p=(node *)malloc(sizeof(node));                  /* 建立一个头结点 */
    p->next=*h;
    while (p->next!=NULL)
    {
        q=p->next;

```

```

    r=p;
    while (q->next!=NULL)
    {
        if (q->next->data<r->next->data) r=q;
        q=q->next;
    }
    if (r!=p)
    {
        s=r->next;
        r->next=s->next;
        s->next=p->next;
        p->next=s;
    }
    p=p->next;
}

*h=h1->next;
free(h1);
}

int test_data[]={5,9,3,1,2,7,8,6,4};
main()
{
    node *h,*p;
    h=create(test_data,sizeof(test_data)/sizeof(int));
    printf("排序前:\n");
    for (p=h;p;p=p->next) printf("%2d ",p->data);
    printf("\n");
    sort(&h);
    printf("排序后:\n");
    for (p=h;p;p=p->next) printf("%2d ",p->data);
    printf("\n");
}

```

本程序的执行结果如下:

排序前:

5 9 3 1 2 7 8 6 4

排序后:

1 2 3 4 5 6 7 8 9

3.3.2 双链表

* 1. 有一个循环双链表,每个结点由两个指针(right 和 left)以及关键字(key)构成,p 指向其中某一结点,编写一个函数从该循环双链表中删除 p 所指向的结点。

解:本题的关键是找出 p 所指结点的前后结点,这可以通过循环指针找到。实现本题功

能的函数如下:

```

struct dlist
{
    int key;
    struct dlist * left, * right;
}

void delnode(p)
struct dlist * p;
{
    struct dlist * q, * r;
    q=p;                                /* 查找 p 的左边结点,由 q 所指向 */
    while (q->right != p) q=q->right;
    r=p;                                /* 查找 p 的右边结点,由 r 所指向 */
    while (r->left != p) r=r->left;
    q->right=r;                          /* 删除 q 与 r 之间的结点 p */
    r->left=q;
    free(p);
}

```

2. 设有如图 3.10 所示的循环双链表 $L=(a,b,c,d)$, 编写一个函数将该链表转换为 $L=(b,a,c,d)$ 。

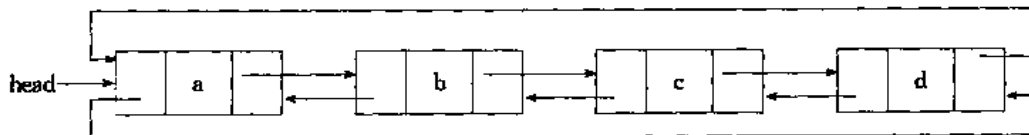


图 3.10 一个循环双链表

解: 本题实际上是交换前面两个结点。实现本题功能的函数如下:

```

dnode * swap(head)
dnode * HEad;
{
    dnode * p, * q;
    p=head->right;                      /* p 指向 b 结点 */
    q=head->left;                        /* 保存 a 结点的左指针 */
    head->right=p->right;                 /* 把 a 结点的右指针指向 c 结点 */
    head->right->left=head;               /* 把 c 结点的左指针指向 a 结点 */
    p->right=head;                       /* 把 p 结点插入作为第一个结点 */
    p->left=q;
}

```

* 3. 设有一个循环双链表, 其中有一结点的指针为 p , 编写一个函数将 p 与其右边的一个结点进行交换。

解:本题利用循环双链表的特点先找到 p 结点的右边结点 q ,然后将 p 与 q 进行交换。实现本题功能的函数如下:

```
void swap()
{
    dnode *p, *q;
    q=p->right;          /* q 指向 p 右边的一个结点 */
    if (q==NULL) printf("不能进行交换! \n");
                        /* 若该链表只有一个结点,则无法交换 */
    else
    {
        p->right=q->right;    /* 删除 p 右边的结点 */
        p->right->left=p;
        p->left->right=q;      /* 把 q 插入到 p 的左边 */
        q->left=p->left;
        p->left=q;
        q->right=p;
    }
}
```

4. 假设有一个循环双链表,其结点包含三个域:pre、data 和 next,其中 data 为数据域, next 为指针域,其值为后续结点的地址,pre 也为指针域,其初值为空(NULL),编写一个函数将此循环单链表改为循环双链表。

解:依题意,双链表的结构定义如下:

```
struct dlist
{
    float data;
    struct dlist *pre, *next;
}
```

假设该循环双链表的头指针为 head。实现本题功能的函数如下:

```
struct dlist *trans(head)
struct dlist *head;
{
    struct dlist *p, *q;
    p=head->next;
    q=head;
    while (p!=head) /* 依次从左向右通过每个结点,对每个结点置 pre 值 */
    {
        p->pre=q;
        p=p->next;
        q=q->next;
    }
}
```

```

head->pre=p; /* 此时 p 指向最后一个结点,将第一个结点的 pre 域指向最后一个结点 */
return(head);
}

```

5. 假设在算法描述语言中引入指针的二元运算“异或”(用“ \odot ”表示),若 a 和 b 为指针,则 $a \odot b$ 的运算结果仍为原指针类型,且:

$$a \odot (a \odot b) = (a \odot a) \odot b = b$$

$$(a \odot b) \odot b = a \odot (b \odot b) = a$$

则可利用一个指针域来实现双向链表。每个结点有两个域: data 域和 link 域, link 域存放该结点前驱与后续结点指针(不存在时为 NULL)的异或。若设指针 h 指向链表中的第一个结点, e 指向链表中最后一个结点, 则可实现从前向后或从后向前遍历此双向链表(这种链表亦称为对称表)。编写一个函数从前向后输出该链表中各元素的值。

解:依题意,该链表的结构如下:

```

struct dlist
{
    int data;
    struct dlist * link;
}

```

现在这样的链表已建立好了,且头指针为 h ,尾指针为 e 。对其中的一个 p 指向的结点,假设其前驱结点指针为 pre ,其后续结点指针为 $next$,那么有:

$$p \rightarrow link = pre \odot next$$

也就是说,当 p 和其前驱结点指针 pre 已知时,可以计算出 p 的后续结点指针:

$$pre \odot p \rightarrow link = pre \odot (pre \odot next) = (pre \odot pre) \odot next = next$$

即:

$$next = pre \odot p \rightarrow link$$

这说明,第三个结点的指针是通过第一个结点的指针与第二个结点的 link 域“异或”得到的。

同样,当 p 和其后续结点 $next$ 已知时,可以计算出 p 的前驱结点指针 pre :

$$p \rightarrow link \odot next = (pre \odot next) \odot next = pre \odot (next \odot next) = pre$$

即:

$$pre = p \rightarrow link \odot next$$

同样说明第一个结点指针是通过第二个结点的 link 域与第三个结点的指针“异或”得到的。

根据这个原理得到实现本题功能的函数如下:

```

void find(h,e)

```

```

struct dlist *h, *e;
{
    struct dlist *pre;
    pre=NULL;p=h;
    while (p!=e)                                /* 遍历最后一结点前的所有结点 */
    {
        printf(" %d ",p->data);
        p=pre<>p->link;
        pre=p;
    }
    printf(" %d ",e->data);                      /* 遍历最后一结点 */
}

```

6. 采用上题中所述的存储结构,编写一个函数在第*i*个结点之前插入一个其 data 域为*x* 的结点的函数。

解:依上题的分析,插入一个结点的函数应修改其前驱结点和后续结点的 link 域并计算该结点的 link 域,实现本题功能的函数如下:

```

void insert(h,e,i,x)
struct dlist *h,*e;
int i,x;
{
    struct dlist *s,*q,*pre,*next;
    int j;
    s=(struct dlist *)malloc(struct dlist);    /* 建立一个待插入的结点 */
    s->data=x;
    if (i==1) /* 在第一个结点前插入 s,这涉及到修改 s 和 h 所指结点的 link 域 */
    {
        s->link=NULL<>h;
        next=NULL<>h->link;    /* next 指向原链表 h 的后续结点 */
        h->link=s<>next;
        h=s;                    /* h 仍作为头指针 */
    }
    else
    {
        pre=NULL;p=h;j=1;
        while (j<=i && p!=e) /* 查找第 i 个结点,找到后 pre 是其前驱结点 */
        {
            j++;
            p=pre<>p->link;
            pre=p;
        }
        if (p==e && j<i) printf(" 无法找到相应的结点! \n");
        /* i 值大于所有结点个数,显示相应信息并返回 */
    }
}

```

```

else
{
    s->link=pre->p->link;    /* 找到了第 i 个结点,在之前插入 s */
    next=pre->p->link;      /* next 指向 p 的后续结点 */
    p->link=s->next;        /* 修改 p 的 link 域 */
    q=pre->link->p;         /* q 指向 pre 的前驱结点 */
    pre->link=q->s;         /* 修改 pre 的 link 域 */
}
}
}

```

7. 采用上题中所述的存储结构,编写一个函数删除第 i 个结点的函数。

解:依上题的分析,插入一个结点的函数应修改其前驱结点和后续结点的 link 域以及计算该结点的 link 域,实现本题功能的函数如下:

```

struct dlist * del(h,e,i)
struct dlist * h, * e;
int i;
{
    struct dlist * p, * h, * pre, * pre1, * next, * next1;
    int j;
    if (i==1) /* 删除第一个结点,这涉及到修改 h 所指结点的 link 域 */
    {
        next=NULL->h->link;    /* next 指向原链表 h 的后续结点 */
        if (next==NULL)        /* 只有一个结点的情况 */
        {
            free(h);
            h=NULL;
            e=NULL;
        }
        else
        {
            p=h->next->link;    /* p 指向第三个结点,可能为 NULL */
            next->link=NULL->p; /* 修改第二个结点的 link 域,使其成为头结点 */
            free(h);
            h=next;            /* h 仍作为头指针 */
        }
    }
    else
    {
        pre=NULL; p=h; j=1;
        while (j<=i && p!=e) /* 查找第 i 个结点 p,找到后 pre 是其前驱结点 */
        {
            j++;

```

```

    p = pre->link;
    pre = p;
}
if (p == e and j < 1) printf("无法找到相应的结点! \n");
    /* i 值大于所有结点个数, 显示相应信息并返回 */
else
{
    next = pre->link;      /* next 指向 p 的后续结点 */
    if (next == NULL)      /* p 是最后一个结点 */
    {
        pre1 = pre->link;  /* pre1 指向 pre 的前驱结点 */
        pre->link = pre1;  /* 修改 pre 的 link 域 */
        e = pre;
        free(p);
    }
    else                  /* p 不是最后一个结点的情况 */
    {
        next1 = p->link;   /* next1 指向 next 的后续结点 */
        next->link = pre->next1;
        pre1 = pre->link;  /* pre1 指向 pre 的前驱结点 */
        pre->link = pre1->next; /* 修改 pre 的 link 域 */
        free(p);
    }
}
}
}

```

第4章 串

串(或字符串)是由零个或多个字符组成的有限序列,一般记为:

$$s = "a_0 a_1 \dots a_{n-1}" \quad (n \geq 0)$$

其中 s 是串名,用双引号括起来的字符序列是串的值; $a_i (0 \leq i \leq n-1)$ 可以是字母、数字或其它字符,串中的字符个数 n 称为串的长度。零个字符的串称为空串,其长度为 0。

串中任意个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,通常称字符在序列中的序号为该字符在串中的位置。子串在主串中的位置则以子串的第一个字符在主串中的位置来表示。

4.1 串的存储及其运算

串是一种特殊的线性表,它的每个结点仅由一个字符组成,因此存储串的方法也就是存储线性表的一般方法,存储串有顺序存储(即把串的字符顺序地存入连续的存储单元中)和链接存储(用单链表形式存储串)。

4.1.1 顺序存储及其基本运算

顺序存储采用一般线性表的存储结构来定义串的类型:

```
typedef struct
{
    char vec[m0];
    int len;
} orderstring;
```

其中 vec 域用来存储字符串, len 域用来存储字符串的当前长度, $m0$ 常量表示允许所存储字符串的最大长度。例如,用以下语句定义 $m0$ 为 100:

```
#define m0 100
```

以顺序方式存储串的基本运算如下:

1. 连接两个串

把两个串 $r1$ 和 $r2$ 首尾相连成一个串 r , 其中 $r1$ 在前, $r2$ 在后。

```
orderstring * concat(r1, r2)
orderstring * r1, * r2;
{
    int i;
    orderstring * r;
```

```

printf("r1=%s,r2=%s\n",r1->vec,r2->vec);
if (r1->len+r2->len>m0) printf("上溢出\n");
/* 若两串长度之和大于 m0,则进行溢出处理 */
else
{
    for (i=0;i<r1->len;i++) /* 将 r1 串传给 r */
        r->vec[i]=r1->vec[i];
    for (i=0;i<r2->len;i++) /* 将 r2 串传给 r */
        r->vec[r1->len+i]=r2->vec[i];
    r->vec[r1->len+1]='\0'; /* 最后一个位置赋给 '\0' */
    r->len=r1->len+r2->len; /* r 串长度等于两串长度之和 */
}
return(r);
}

```

2. 取子串

从串 r_1 中的第 i 个字符开始,把连续 j 个字符组成的子串赋给 r 。

```

orderstring * substring(r1,i,j)
orderstring * r1;
int i,j;
{
    int k;
    orderstring * r;
    if (i+j-1>r1->len) printf("超界\n");
    /* 若 i,j 的值超出允许的范围,则进行'超界'处理 */
    else
    {
        for (k=0;k<j;k++) r->vec[k]=r1->vec[i+k-1];
        /* 将 r1 中指定的子串传送给 r */
        r->len=j; /* 把子串长度赋给 r 的长度域 */
        r->vec[r->len]='\0';
    }
    return(r);
}

```

3. 删除子串

从串 r 中删除从第 i 个字符开始的,长度为 j 的一个子串。

```

orderstring * delsubstring(r,i,j)
orderstring * r;
int i,j;
{
    int k;
    if (i+j-1>r->len) printf("超界\n");

```



```

        /* 若 i,j 的值超出允许的范围,则进行'超界'处理 */
    else
    {
        for (k=i+j-1;k<r->len;k++) r->vec[k-j]=r->vec[k];
        /* 将被删除子串后面的所有字符依次前移 j 个位置 */
        r->len=r->len-j; /* r 的长度减少 j */
        r->vec[r->len]='\0';
    }
    return(r);
}

```

4. 插入子串

把串 r1 插入到串 r 中第 i 个字符开始的位置上。

```

orderstring * insert(r,r1,i)
orderstring * r,* r1;
int i;
{
    int k;
    if (i>=r->len || r->len+r1->len>m0) printf("不能插入! \n");
    else
    {
        for (k=r->len-1;k>=i;k--) r->vec[r1->len+k]=r->vec[k];
        /* 从第 i 个位置起空出连续 r1->len 个位置 */
        for (k=0;k<r1->len;k++) r->vec[i+k-1]=r1->vec[k];
        /* 把 r1 写入 r 中空出的位置上 */
        r->len=r->len+r1->len;
        /* 把 r 的长度修改为 r 和 r1 的长度之和 */
        r->vec[r->len]='\0';
    }
    return(r);
}

```

5. 求子串位置

求子串位置的运算又叫做串的模式匹配运算。从串 r1 中求出首次与 r2 相同的子串的起始位置,若 r1 中不存在 r2,则返回 0。

```

int position(r1,r)
orderstring * r1,* r;
{
    int i,j,k;
    for (i=0;r->vec[i];i++)
        for (j=i,k=0;r->vec[j]==r1->vec[k];j++,k++)
            if (! r1->vec[k+1]) /* 子串结尾,表示查找子串成功 */
                return(i);
}

```

```

return(-1);                /* 未找到子串 */
}

```

4.1.2 链接存储及其基本运算

下面讨论链接存储方法,用单链表形式存储串,链表中每个结点的类型如下:

```

typedef struct node
{
    char info;
    struct node * link;
} linkstring;

```

如图 4.1 所示是串 $s = 'abcd'$ 的链接存储图示。

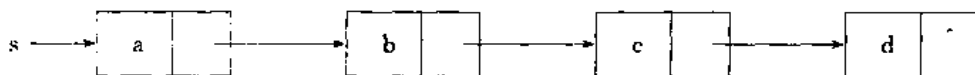


图 4.1 串的链接存储表示

以链接方式存储串的基本运算如下:

1. 连接两个串

把两个串 $r1$ 和 $r2$ 首尾相连成一个串 r , 其中 $r1$ 在前, $r2$ 在后。

```

linkstring * concat(r1, r2)
linkstring * r1, * r2;
{
    node * p, * q, * s;
    r = (linkstring *) malloc(sizeof(linkstring));
    q = r;                /* 建立一个头结点 */
    p = r1;
    while (p != NULL)      /* 将 r1 串复制给 r */
    {
        s = (linkstring *) malloc(sizeof(linkstring));
                                /* 复制一个与 r1 当前结点相同的结点 */
        s->info = p->info;
        q->link = s;
        q = s;                /* q 总是指向 r 的最后一个结点 */
        p = p->link;
    }
    p = r2;
    while (p != NULL)      /* 将 r2 串复制给 r */
    {
        s = (linkstring *) malloc(sizeof(linkstring));
        s->info = p->info;

```

```

    q->link=s;
    q=s;                                /* q 总是指向 r 的最后一个结点 */
    p=p->link;
}
q->link=NULL;
r=r->link;                              /* 删除头结点 */
}

```

2. 取子串

从串 r_1 中的第 i 个字符开始,把连续 j 个字符组成的子串赋给 r 。

```

linkstring * substring(r1,i,j)
linkstring * r1;
int i,j;
{
    int k;
    linkstring * p, * q, * s, * r;
    p=r1;k=1;
    while (k<i && p!=NULL)
    {
        p=p->link;k++;
    }
    if (p==NULL) printf("i 出错\n");
    else
    {
        r=(linkstring *)malloc(sizeof(linkstring));
        q=r;k=1;                                /* 建立一个头结点 */
        while (k<=j && p!=NULL)                /* 复制 j 个结点 */
        {
            s=(linkstring *)malloc(sizeof(linkstring));
            /* 复制一个与 r1 当前结点相同的结点 */
            s->info=p->info;
            q->link=s;
            q=s;                                /* q 总是指向 r 的最后一个结点 */
            p=p->link;
            k++;
        }
        q->link=NULL;
        r=r->link;                              /* 删除头结点 */
    }
    return(r);
}

```

3. 删除子串

从串 r 中删除从第 i 个字符开始的,长度为 j 的一个子串。

```

linkstring *delsubstring(r,i,j)
linkstring *r;
int i,j;
{
    int k;
    linkstring *p,*q,*s;
    p=r;q=p;k=1;
    while (k<i and p!=NULL) do
    {
        q=p;                                /* q 指向 p 的前一个结点 */
        p=p->link;k++;
    }
    if (p==NULL) printf("i 出错\n");
    else
    {
        k=1;
        while (k<j && p!=NULL)                /* 查找 j 个结点之后的结点 */
        {
            p=p->link;k++;
        }
        if (p==NULL) printf("j 出错\n");
        else                                /* 这时 p 指向最后一个要删除的结点 */
        {
            s=q->link;                        /* s 指向要删除结点的头结点 */
            q->link=p->link;                    /* 从 r 中删除了所有要删除的结点 */
            p->link=NULL;
            p=s;
            while (s!=NULL)                    /* 释放所有删除的结点 */
            {
                p=s->link;
                free(s);
                s=p;
            }
        }
    }
    return(r);
}

```

4. 插入子串

把串 r1 插入到串 r 中第 i 个字符开始的位置上。

```

linkstring *insert(r,r1,i)
linkstring *r,*r1;
int i;

```

```

{
    int k;
    linkstring *p,*q;
    p=r;k=1;
    while (k<i && p!=NULL)          /* 查找第 i 个结点,找到后由 p 所指向 */
    {
        p=p->link;k++;
    }
    if (p==NULL) printf("i 出错\n");
    else
    {
        q=r1;                        /* 查找 r1 的最后一个结点,由 q 所指向 */
        while (q!=NULL) q=q->link;
        q->link=p->link;              /* 把 r1 插入进去 */
        p->link=r1;                   /* 把 r1 链接到 p 之后 */
    }
    return(r);
}

```

5. 求子串位置

从串 r1 中求出首次与 r2 相同的子串的起始位置,若 r1 中不存在 r2,则返回 0。

```

int position(r1,r2)
linkstring *r1,*r2;
{
    linkstring *p,*p1,*q,*q1;
    int i=0;
    p=r1;
    while (p!=NULL)                  /* 循环扫描 r1 的每个结点 */
    {
        q=r2;
        while (q!=NULL)              /* 依次扫描 r2 的每个结点 */
        {
            if (p->info==q->info)      /* 判定是否与 r1 的当前结点相等 */
            {
                /* 若相等,则判定 r1 其后的元素是否与 r2 之后所有元素依次相同 */
                p1=p->link;q1=q->link;
                while (p1->info==q1->info)
                {
                    p1=p1->link;
                    q1=q1->link;
                }
                if (q1==NULL) position=i; /* 若相同则返回相同的子串的起始位置 */
            }
            else q=q->link;
        }
    }
}

```

```

    }
    p = p->link;
    l++;
}
}

```

4.2 基本题

4.2.1 单项选择题

1. 空串与空格串是相同的,这种说法 ①。

- A. 正确 B. 不正确

答:①B

2. 串是一种特殊的线性表,其特殊性体现在 ①。

- A. 可以顺序存储 B. 数据元素是一个字符
C. 可以链接存储 D. 数据元素可以是多个字符

答:①B

3. 设有两个串 p 和 q,求 q 在 p 中首次出现的位置的运算称作 ①。

- A. 连接 B. 模式匹配
C. 求子串 D. 求串长

答:①B

4. 设串 $s_1 = 'ABCDEFGH'$, $s_2 = 'PQRST'$, 函数 $con(x, y)$ 返回 x 和 y 串的连接串, $subs(s, i, j)$ 返回串 s 的从序号 i 的字符开始的 j 个字符组成的子串, $len(s)$ 返回串 s 的长度, 则 $con(subs(s_1, 2, len(s_2)), subs(s_1, len(s_2), 2))$ 的结果串是 ①。

- A. BCDEF B. BCDEFG
C. BCPQRST D. BCDEFEF

答:①D

4.2.2 填空题(将正确的答案填在相应的空中)

1. 串的两种最基本的存储方式是 ①。

答:①顺序存储方式和链接存储方式

2. 两个串相等的充分必要条件是 ①。

答:两个串的长度相等且对应位置的字符相同

3. 空串是①,其长度等于②。

答:①零个字符的串;②零

4. 空格串是①,其长度等于②

答:①由一个或多个空格字符组成的串,②其包含的空格个数

5. 设 $s = 'I_AM_A_TEACHER'$, 其长度是 ①。

答: ①14

6. 设 $s1 = 'GOOD'$, $s2 = '_'$, $s3 = 'BYE!'$, 则 $s1$, $s2$ 和 $s3$ 连接后的结果是 ①。

答: ①'GOODBYE!'

4.3 习题解析

1. 编写下列算法(假定下面所用的串均采用顺序存储方式, 参数 ch , $ch1$ 和 $ch2$ 均为字符型):

(1) 将串 r 中所有其值为 $ch1$ 的字符换成 $ch2$ 的字符。

(2) 将串 r 中所有字符按照相反的次序仍存放在 r 中。

(3) 从串 r 中删除其值等于 ch 的所有字符。

(4) 从串 $r1$ 中第 $index$ 个字符起求出首次与字符 $r2$ 相同的子串的起始位置。

(5) 从串 r 中删除所有与串 $r3$ 相同的子串(允许调用第(4)小题的函数和第(3)小题的删除子串的函数)。

解:

(1) 本小题的算法思想是: 从头到尾扫描 r 串, 对于值为 $ch1$ 的元素直接替换成 $ch2$ 即可。其函数如下:

```
orderstring * trans(r, ch1, ch2)
orderstring * r;
char ch1, ch2;
{
    int i;
    for (i=0; i<(r->len); i++)
        if (r->vec[i]==ch1) r->vec[i]=ch2;
    return(r);
}
```

(2) 本小题的算法思想是: 将第一个元素与最后一个元素交换, 第二个元素与倒数第二个元素交换, 如此下去, 便将该串的所有字符反序了。其函数如下:

```
orderstring * invert(r)
orderstring * r;
{
    int i;
    char x;
    for (i=0; i<((r->len)/2); i++)
    {
        x=r->vec[i];
        r->vec[i]=r->vec[r->len-i-1];
        r->vec[r->len-i-1]=x;
    }
}
```

```

    }
    return(r);
}

```

(3)本小题的算法思想是:从头到尾扫描 r 串,对于其值为 ch 的元素采用移动的方式进行删除。其函数如下:

```

orderstring * delall(r,ch)
orderstring * r;
char ch;
{
    int i,j;
    for (i=0;i<r->len;i++)
        if (r->vec[i]==ch)
        {
            for (j=i;j<r->len;j++)
                r->vec[j]=r->vec[j+1];
            r->len=r->len-1;
        }
    return(r);
}

```

(4)本小题的算法思想是:从第 $index$ 个元素开始扫描 $r1$,当其元素值与 $r2$ 的第一个元素的值相同时,判定它们之后的元素值是否依次相同,直到 $r2$ 结束为止,若都相同则返回,否则继续上述过程直到 $r1$ 扫描完为止。其函数如下:

```

int partposition(r2,r1,index)
orderstring * r2,* r1;
int index;
{
    int i,j,k;
    for (i=index;r2->vec[i];i++)
        for (j=i,k=0;r2->vec[j]==r1->vec[k];j++,k++)
            if (! r1->vec[k+1])
                return(i);
    return(-1);
}

```

(5)本小题的算法思想是:从位置 1 开始调用第(4)小题的函数 $partposition()$,若找到了一个相同子串,则调用 $delsubstring()$ 将其删除,再查找后面位置的相同子串,方法与以上相同。其函数如下:

```

orderstring * delstringall(r,r3)
orderstring * r,* r3;
{
    int i=0;

```



```

while (i < r->len) /* 当调用 delsubstring() 进行删除操作时, r->len 也减小了 */
{
    if (partposition(r, r3, i) != -1) delsubstring(r, i, r3->len);
    i++;
}
}

```

2. 若 x 和 y 是两个采用顺序结构存储的串, 编写一个比较两个串是否相等的函数。

解: 两个串相等表示对应的字符必须都相同, 所以扫描两串, 逐一比较相应位置的字符, 若相同继续比较直到全部比较完毕, 如果都相同则表示两串相等, 否则表示两串不相等, 由此得到如下函数:

```

int same(x, y)
orderstring *x, *y;
{
    int i=0, tag=1;
    if (x->len != y->len) return(0);
    else
    {
        while (i < x->len && tag)
        {
            if (x->vec[i] != y->vec[i]) tag=0;
            i++;
        }
        return(tag);
    }
}

```

* 3. 对于采用顺序结构存储的串 x , 编写一个函数删除其值等于 ch 的所有字符, 要求具有较高的效率。

解: 本题的算法思想是: 先扫描 x 的所有元素, 对于其值等于 ch 的元素, 用一个特殊的 '*' (假设原字符串中没有包含 '*' 字符) 来替换, 最后从后向前物理删除其值为 '*' 的所有结点, 这样减少结点移动的次数, 提高了算法的效率, 其函数如下:

```

orderstring *delall(x, ch)
orderstring *x;
char ch;
{
    int i=0, j, k, n;
    while (i < x->len) /* 扫描 x 的所有元素 */
    {
        if (x->vec[i] == ch) x->vec[i] = '*';
        i++;
    }
}

```

```

for (j=x->len-1;j>=0;j--)
    if (x->vec[j]=='*')
    {
        for (k=j;k<x->len;k++)
            x->vec[k]=x->vec[k+1]; /* 移动字符进行覆盖 */
        x->len--; /* 串长度减1 */
    }
return(r);
}

```

4. 采用顺序结构存储串 s , 编写一个函数删除 s 中第 i 个字符开始的 j 个字符。

解: 本题的算法思想是: 先判定 s 串中要删除的内容是否存在, 若存在则将第 $i+j-1$ 之后的字符前移 j 个位置。其函数如下:

```

orderstring * delij(s,i,j)
orderstring * s;
int i,j;
{
    int h;
    if (i+j<s->len)
    {
        for (h=i;h<(i+j);h++) /* 第 i+j-1 之后的字符都前移 j 个位置 */
            s->vec[h]=s->vec[h+j];
        s->len-=j;
        return(s);
    }
    else printf("无法进行删除操作\n");
}

```

5. 采用顺序存储方式存储串, 编写一个函数将串 s_1 中的第 i 个字符到第 j 个字符之间的字符(不包括第 i 个和第 j 个字符)用 s_2 串替换, 函数名为 $stuff(s_1,i,j,s_2)$ 。例如: $stuff('abcd',1,3,'xyz')$ 返回 'axyzcd'。

解: 本题算法思想是: 先提取 s_1 的前 i 个字符 str_1 , 再取第 j 个字符及之后的所有字符 str_2 , 将 str_1, s_2, str_2 连接起来便构成了结果串, 其函数如下:

```

orderstring * stuff(s1,i,j,s2)
orderstring * s1, * s2;
int i,j;
{
    orderstring * s;
    int top,h,k;
    s=(orderstring *)malloc(sizeof(orderstring));
    if (1<=j && i<s1->len && j<s1->len)
    {
        for (h=0;h<i;h++) s->vec[h]=s1->vec[h];

```

```

/* 把 s1 的前 i 个字符赋给 s */
s->len=i;
h=0;
while (h<s2->len) /* 连接 s2 串 */
{
    s->vec[s->len+h]=s2->vec[h];
    h++;
}
s->len=s->len+s2->len;
s->vec[s->len]='\0';
top=s->len;
for (top=s->len,h=j-1,h<s1->len,h++,top++)
    /* 连接 s1 的第 i 个字符及之后的字符 */
    s->vec[top]=s1->vec[h];
s->len=top;
s->vec[s->len]='\0';
}
return(s);
}

```

* 6. 采用顺序结构存储串,编写一个函数 `substring(s1,s2)`,用于判定 `s2` 是否是 `s1` 的子串。

解:本题的算法思想是:设 $s1 = 'a_0a_1 \dots a_m'$

$s2 = 'b_0b_1 \dots b_n'$

从 `s1` 中找与 `b0` 匹配的字符 `ai`,若 $a_i = b_0$,则判定 $a_{i+1} = b_1, \dots, a_{i+n} = b_n$,若都相等,则结果是子串,否则继续比较 `ai` 之后的字符,本函数如下:

```

int substring (s1,s2)
orderstring * s1, * s2;
{
    int i,j,k,yes=0;
    i=0;
    while (i<s1->len && ! yes)
    {
        j=0;
        if (s1->vec[i]==s2->vec[j])
        {
            k=i+1;
            j++;
            while (s1->vec[k]==s2->vec[j])
            {
                k++;
                j++;
            }
        }
    }
}

```

```

    }
    if (j == s2->len) yes = 1;
    else i++;
}
else i++;
}
return(yes);
}

```

7. 采用顺序结构存储串,编写一个函数求串 s 中出现的第一个最长重复子串的下标和长度。

解: 本题的算法思想是: 先给最长重复子串的下标 $index$ 和长度 $length$ 均赋值为 0。

设 $s = 'a_1a_2 \dots a_n'$, 扫描通过串 s , 对于当前字符 a_i , 判定其后是否有相同的字符, 若有记为 a_j , 再判定 a_{i+1} 是否等于 a_{j+1} , a_{i+2} 是否等于 a_{j+2}, \dots , 如此找到一个不同的字符为止, 即找到了一个重复出现的子串, 将其下标 $index1$ (实际上为 i) 与长度 $length1$ 记下来, 将 $length1$ 与 $length$ 相比较, 保留较长的子串 $index$ 和 $length$ 。再从 $a_{j+length1}$ 之后找重复子串。然后对于 a_{i+1} 之后的字符采用上述函数, 最后的 $index$ 与 $length$ 即记录下最长重复子串的下标与长度。例如 $s = 'aabcdababce'$

首先 $index = 0, length = 0$

$i = 0$:

$s = 'aabcdababce'$

↑↑
i j

$index = 0, length = 1$

$s = 'aabcdababce'$

↑ ↑
i j

$index = 0, length = 1$

$s = 'aabcdababce'$

↑ ↑
i j

$index = 0, length = 1$

$i = 1$:

$s = 'aabcdababce'$

↑ ↑
i j

$index = 1, length = 2$

$s = 'aabcdababce'$

↑ ↑
i j

$index = 1, length = 3$

i=2;

s='aabcdababce'

↑ ↑
i j

index=1, length=3(不变)

s='aabcdababce'

↑ ↑
i j

index=1, length=3(不变)

i=3;

s='aabcdababce'

↑ ↑
i j

index=1, length=3(不变)

i=4;

s='aabcdababce'

↑
i

index=i, length=3(不变, j 未找到)

i=5;

s='aabcdababce'

↑ ↑
i j

index=1, length=3(不变)

i=6;

s='aabcdababce'

↑ ↑
i j

index=1, length=3(不变)

i=7;

s='aabcdababce'

↑
i

index=1, length=3(不变, j 未找到)

i=8;

s='aabcdababce'

↑
i

index=1, length=3(不变, j 未找到)

i=9;

s='aabcdababce'

↑
i

index=1, length=3(不变, j 未找到)

```
i=10;
```

```
s='aabcdababce'
```

```
  ^
  i
```

```
index=1, length=3(不变,j 未找到)
```

结果:index=1, length=3,即'abc'是最大重复子串。

其程序如下,其中 maxsubstr()函数用于实现本题功能:

```
#include <stdio.h>
#define m0 30
typedef struct
{
    char vec[m0];
    int len;
} orderstring;

void maxsubstr(s)
orderstring *s;
{
    int index=0,length=0,length1,i=0,j,k;
    while (i<s->len)
    {
        j=i+1;
        while (j<s->len)
        {
            if (s->vec[i]==s->vec[j]) /* 找一个子串,其序号为 i,长度为 length1 */
            {
                length1=1;
                for(k=1;s->vec[i+k]==s->vec[j+k];k++)
                    length1++;
                if (length1>length) /* 将较大长度者赋给 index 与 length */
                {
                    index=i;
                    length=length1;
                }
                j+=length1;
            }
            else j++;
        }
        i++; /* 继续扫描第 i 字符之后的字符 */
    }
    printf("最长重复子串:");
    for (i=0;i<length;i++)
        printf("%c",s->vec[index+i]);
}
```

```

    }
    main()
    {
        orderstring *r;
        strcpy(r->vec,"aabcdababce");
        r->len=11;
        maxsubstr(r);
    }

```

本程序的执行结果如下:

最长重复子串:abc

8. 采用顺序结构存储串,编写一个函数,求串s和串t的一个最长公共子串。

解:其原理与前面的一题相同,只是上题是扫描一个串,这里改为扫描两个串。其中 index 指出最长公共子串在s中的序号,length 指出最长公共子串的长度。其程序如下,其中 maxcomstr()函数用于实现本题功能:

```

#include <stdio.h>
#define m0 30
typedef struct
{
    char vec[m0];
    int len;
} orderstring;
void maxcomstr(s,t)
orderstring *s,*t;
{
    int index=0,length=0,i,j,k,length1;
    i=0; /* i 作为扫描 s 的指针 */
    while (i<s->len)
    {
        j=0; /* j 作为扫描 t 的指针 */
        while (j<t->len)
        {
            if (s->vec[i]==t->vec[j]) /* 找一个子串,其在 s 中的序号为 i,长度为 length1 */
            {
                length1=1;
                for (k=1;s->vec[i+k]==t->vec[j+k];k++)
                    length1=length1+1;
                if (length1>length) /* 将较大长度者赋给 index 与 length */
                {
                    index=i;
                    length=length1;
                }
            }
        }
    }
}

```

```

        j += length1; /* 继续扫描 t 中第 j+length1 字符之后的字符 */
    }
    else j++;
}
i++; /* 继续扫描 s 中第 i 字符之后的字符 */
}
printf("最长公共子串:");
for (i=0; i<length; i++)
    printf("%c", s->vec[index+i]);
}
main()
{
    orderstring *r, *r1;
    strcpy(r->vec, "aabcdababce");
    r->len=11;
    strcpy(r1->vec, "12abcabcbace");
    r1->len=12;
    maxcomstr(r, r1);
}

```

本程序的执行结果如下:

最长公共子串:abcda

9. 采用顺序结构存储串,编写一个实现串通配符匹配的函数 `pattern_index()`,其中的通配符只有'?',它可以和任一字符匹配成功,例如,`pattern_index("?re", "there are")`返回的结果是6。

解:本题和4.1.1节所述的求子串位置算法相似,只是增加了'?'的处理功能。实现本题功能的函数如下:

```

int pattern_index(substr, str)
orderstring * substr, * str;
{
    int i, j, k;
    for (i=0; str[i]; i++)
        for (j=i, k=0; (str[j]==substr[k]) || (substr[k]=='?'); j++, k++)
            if (!substr[k+1])
                return(i);
    return(-1);
}

```

10. 采用顺序结构存储串,编写一个函数计算一个子串在一个字符串中出现的次数,如果该子串不出现则为0。

解:本题是4.1.1节所述的求子串位置算法的扩展,在找到子串后不是退出,而是继续查找,直到整个字符串查找完毕。实现本题功能的函数如下:


```

int str_count(substr, str)
orderstring * substr, * str;
{
    int i=0, j, k, count=0;
    for (i=0; str->vec[i]; i++)
        for (j=i, k=0; (str->vec[j]==substr->vec[k]); j++, k++)
            if (k==substr->len-1) /* 可用!substr->vec[k+1]作为条件 */
                count++;
    return(count);
}

```

11. 采用顺序结构存储串,编写一个实现串比较运算的函数 `strcmp(s,t)`,串比较以词典方式进行,当 s 大于 t 时返回1, s 与 t 相等时返回0, s 小于 t 时返回-1。

解:本题的算法思想是:先比较 s 和 t 公共长度的部分的相应字符,若前者字符大于后者字符,则返回1;若前者字符小于后者字符,则返回-1;否则相等时继续比较;当所有公共长度的部分的相应字符均相同时,再比较两者的长度,当两者的长度相等时返回0,前者长度大于后者长度,则返回1;若前者长度小于后者长度,则返回-1。实现上述算法的函数如下:

```

int strcmp(a,b)
orderstring * a, * b;
{
    int i,minlen;
    if (s->len<t->len) minlen=s->len; /* 计算:minlen<=min(m,n) */
    else minlen=t->len;
    i=0;
    while (i<=minlen)
    {
        if (a->vec[i]<b->vec[i]) return(-1); /* s<t */
        else if (a->vec[i]>b->vec[i]) return(1); /* s>t */
        else i++;
    }
    /* 这里是公共长度部分均相同的情况 */
    if (s->len==t->len) return(0); /* s=t */
    else if (s->len<t->len) return(-1); /* s<t */
    else return(1); /* s>t */
}

```

12. 若 x 是采用单链表存储的串,编写一个函数将其中的所有 c 字符替换成 s 字符。

解:本题采用的算法是:逐一扫描 x 的每个结点,对于每个数据域为 c 的结点修改其元素值为 s 。对应的函数如下:

```

linkstring * trans(x,c,s)
linkstring * x;
char c,s;

```

```

{
    linkstring * p;
    p=x;
    while (p!=NULL)
    {
        if (p->data=='c') p->data='s';
        p=p->link;
    }
    return(x);
}

```

13. 若 x 和 y 是两个单链表存储的串,编写一个函数找出 x 中第一个不在 y 中出现的字符。

解:本题的算法思想是:从头到尾扫描 x ,对于 x 的每个结点 c ,判定是否在 y 中,若在,则继续扫描 x ,若不在,则给出该 c 并返回。对应的函数如下:

```

char findfirst(x,y)
linkstring * x, * y;
{
    linkstring * p;
    char c;
    p=x;
    if (x==NULL) printf("x 为空\n");
    else
    {
        while (found(p->data,y)) p=p->link;
        c=p->data;
    }
    return(c);
}
/* 函数为 found,其功能是:若 head 的链表中包含有 data 域为 x 的 */
/* 结点则返回1,否则返回0 */
int found(d,head)
linkstring head;
char d;
{
    while (head!=NULL && head->data!=d)
        head=head->link;
    if (head==NULL) return(0);
    else return(1);
}

```

* 14. 一个仅由字母组成的字符串 l ,长度为 n ,其结构如图4.2所示,每个结点的 $data$ 字段只存放一个字母。有人设计了一个函数 $find$,旨在去掉字符串中所有 X 字母(结点返回存

储池),并将串中的一个最大字母排列到串尾($A < B, \dots < Y < Z$)。试分析以下函数能否正确工作,改正其所有逻辑错误,使之能实现上述要求。

```

find(l)
{
    r ← 0                      /* r, g, p 为指针变量, w 为工作状态标志 */
    q ← 0
    p ← 1
    w ← 1
    while p ≠ 0 do
        if (w = 1) then        /* 寻找第一个不为 x 的结点 */
            if (data(p) = X) then ret(p) /* 结点 p 归还存储池 */
            else [ r ← p
                    q ← p
                    p ← LINK(p)
                    w ← 0
                  ]
        else                    /* 删除 X, 同时将大字母向后移 */
            if DATA(p) = X then ret(p)
            else
                if DATA(p) > DATA(q) then
                    [ r ← p
                      LINK(p) ← q
                      LINK(q) ← LINK(p)
                    ]
                else
                    [ r ← q
                      q ← p
                      p ← LINK(p)
                    ]
    }
ret(x)
/* x 指向一个要归还到存储池的结点, av 为存储池指针 */
{
    link(x) ← av
    av ← x
}

```

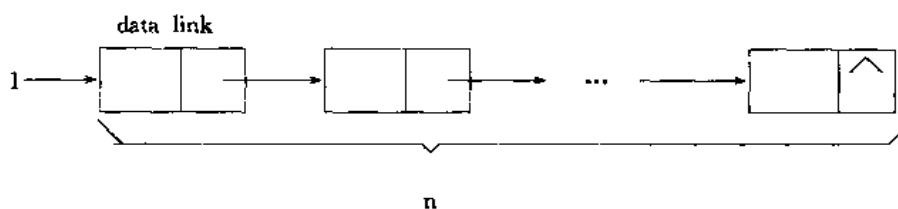


图4.2 一个链表

解:本题的函数存在多处错误,其函数修改如下:

```
void find(l,x)
linkstring *l;
char x;
{
    linkstring *p,*q,*r;
    int w;
    p=l;
    w=1;
    while (p!=NULL)
        if (w==1)                                /* 处理第一个结点 */
            if (p->data==x)                        /* 删除第一个结点 */
            {
                l=p->link;
                ret(p);
            }
            else
            {
                q=p;
                p=p->link;
                w=0;
            }
        else                                       /* 处理其他结点 */
            if (p->data==x)                        /* 删除其值为 x 的结点 */
            {
                q->link=p->link;
                p=p->link;
                ret(p);
            }
            else
                if (p->data>q->data)                /* 交换 p,q 两结点 */
                {
                    r=p->link;
                    p->link=q;
                }
            }
}
```

```

        q->link=r;
    }
    else
    {
        q=p;
        p=p->link;
    }
}
/* 函数 ret */
void ret(x);
/* x 指向一个要归还到存储池的结点,av 为存储池指针 */
{
    av->link=x;
    av=x;
}

```

15. 若 s 和 t 是用单链表存储的两个串,设计一个函数将 s 串中首次与串 t 匹配的子串逆置。

解:设 s 和 t 是用带头结点的单链表表示的,首先在 s 串中查找首次与串 t 匹配的子串,若未找到,显示相应信息并返回;否则将该子串逆置,其函数如图4.3所示,先将子串的第一个结点链接到 p 的前面,再将该子串的第二个结点链接到前面移动的第二个结点的前面,如此下去,便逆置了该子串。实现本题功能的函数如下:

```

linkstring *invert_substring(s,t)
linkstring *s,*t;
{
    linkstring *prior,*p,*t1,*r,*q,*u;
    prior=s;
    p=s;
    t1=t;
    if (p=NULL || t1=NULL) printf("出错!\n");
    else
    {
        while (p!=NULL && t1!=NULL) /* 在 s 串中首次与串 t 匹配的子串 */
        {
            if (p->data==t1->data)
            {
                p=p->link;
                t1=t1->link;
            }
            else
            {
                prior=prior->link;
            }
        }
    }
}

```

```

        p=prior->link;
    }
    t1=t->link;
}
if (t1!=NULL) printf("s 中未有与 t 相匹配的子串!"); /* 未找到子串 */
else /* 找到了与 t 相匹配的子串, prior 指向该子串的第一个结点的前一个结点, */
    /* p 指向该子串的最后一个结点的下一个结点 */
{
    q=prior->link; /* 对 p 中的该子串进行逆置 */
    r=q->link;
    q->link=p;
    while (r!=p)
    {
        u=r->link;
        r->link=q;
        q=r;
        r=u;
    }
    prior->link=q;
}
}
}

```

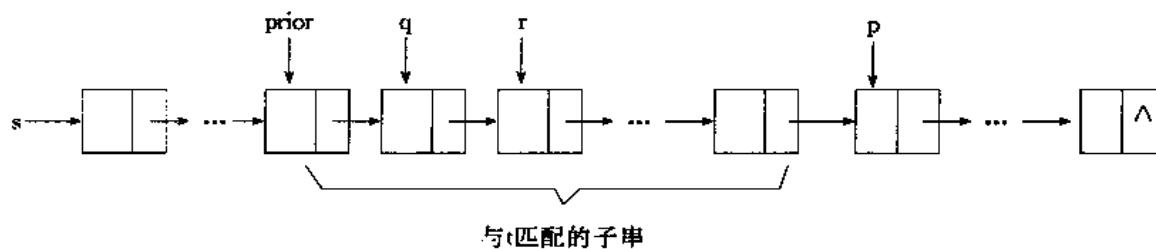


图4.3 查找并逆置子串

第5章 数组和稀疏矩阵

数组是一种常用的数据类型,前面讨论过的向量就是一维数组,本章主要讨论多维数组。多维数组是一种复杂的数据结构,数组元素之间的关系,既不是线性的,也不是树形的,但所有元素必须具有相同的数据类型。

矩阵通常是指二维数组,稀疏矩阵则是指大多数元素值为0、只有少数元素值不为0的矩阵,因此为了节省存储空间,稀疏矩阵要采用不同的存储方法。

5.1 基本概念和运算

本节讨论多维数组特别是矩阵的基本运算和稀疏矩阵的存储方式以及在各种存储方式下稀疏矩阵基本运算的实现等。

5.1.1 多维数组

多维数组是向量的推广,例如,二维数组(或称矩阵)

$$A_{mn} = \begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,n-1} \\ \dots & \dots & \dots & \dots \\ a_{m-1,0} & a_{m-1,1} & \dots & a_{m-1,n-1} \end{bmatrix}$$

可以看成是由 m 个行向量组成的向量,也可以看成是 n 个列向量组成的向量,其中每个元素 $a_{i,j}$ 都有两个前驱结点 $a_{i-1,j}$ 和 $a_{i,j-1}$ 以及两个后续结点 $a_{i+1,j}$ 和 $a_{i,j+1}$ 。这种规则可以推广到三维以上的数组。

多维数组的定义也很简单,以二维数组为例,其类型定义如下:

```
typedef ElemType maxix[m][n];
```

这里的 ElemType 可以是任何相应的数据类型如 int, float 或 char 等,在算法中,我们规定 ElemType 缺省是 int 类型。其中, m 是行数, n 是列数,在 C 语言中, m 和 n 必须是常量或先定义的常量宏,假如定义它们为:

```
#define m 4
#define n 5
```

数组存储在一片相邻的存储单元里,以二维数组为例,当 $A[m][n]$ 按“行优先顺序”存储时,元素 $a_{i,j}$ 的地址计算为:

$$LOC(a_{i,j}) = LOC(a_{0,0}) + i * n + j$$

按“列优先顺序”存储时,地址计算为:

$$\text{LOC}(a_{ij}) = \text{LOC}(a_{0,0}) + j * m + i$$

下面以二维数组即矩阵为例来讨论多维数组的基本运算。

1. 转置矩阵

转置是一种最简单的矩阵运算, 对于一个 $m \times n$ 的矩阵 A_{mn} , 其转置矩阵是一个 $n \times m$ 的矩阵 B_{nm} , 且 $B[i][j] = A[j][i]$, $0 \leq i < n, 0 \leq j < m$, 其函数如下:

```
void trsmat(A,B)
maxix A,B;
{
    int i,j;
    for (i=0;i<m;i++)
        for (j=0;j<n;j++)
            B[j][i]=A[i][j];
}
```

2. 矩阵相加

两个大小均为 $m \times n$ 的矩阵的元素相加后得到一个 $m \times n$ 的矩阵 C , 其运算是把 A 与 B 的相同位置的元素相加得到 C , 即 $c_{i,j} = a_{i,j} + b_{i,j}$, $0 \leq i < m, 0 \leq j < n$, 其函数如下:

```
void addmat(C,A,B)
maxix A,B,C;
{
    int i,j;
    for (i=0;i<m;i++)
        for (j=0;j<=n;j++)
            C[i][j]=A[i][j]+B[i][j];
}
```

3. 矩阵相乘

矩阵相乘是另一种常用的矩阵运算, 其方法是我们所熟悉的, 设:

$$C=A \times B$$

其中 A 是 $m \times n$ 矩阵, B 是 $n \times k$ 矩阵, 则 C 为 $m \times k$ 矩阵, 其函数如下:

```
void mutmatC(A,B)
maxix A,B,C;
{
    int i,j,k;
    for (i=0;i<m;i++)
        for (j=0;j<=k;j++)
        {
            C[i][j]=0;
            for (k=0;k<n;k++)
                C[i][j]=C[i][j]+A[i][k]*B[k][j];
        }
}
```



```

    }
}

```

5.1.2 稀疏矩阵

当二维数组 A_{mn} 中有 k 个非零元素,若 $k \ll m * n$,则称 A 为稀疏矩阵。

在存储稀疏矩阵时,为了节省存储单元,最好只存放其中的非零元素,常用的存储稀疏矩阵的方法有顺序存储和链接存储,下面分别讨论这两种方法。

1. 顺序存储

稀疏矩阵的顺序存储方法也有几种,如三元组表示法和伪地址表示法等。

(1) 三元组表示法

采用这种方法时,线性表中的每个结点对应稀疏矩阵的一个非零元素,其中包括 3 个字段,分别为该元素的行下标、列下标和值,结点间的次序按矩阵的行优先顺序排列(跳过 0 元素)。另外,用第 0 行的第 1 个元素存储矩阵的行数目,第 0 行的第 2 个元素存储矩阵的列数目,第 0 行的第 3 个元素存储矩阵中非零元素的数目,这个线性表用顺序的方法存储在连续的存储区里,例如,如图 5.1 所示的稀疏矩阵用三元组 a 表示如图 5.2 所示。

稀疏矩阵三元组表示的数据类型定义如下:

```
typedef ElemType smatrik[maxterms][3];
```

这里的 ElemType 可以是任何相应的数据类型如 int, float 或 char 等,在算法中,我们规定 ElemType 缺省是 int 类型。其中 maxterms 是最多的非零元素个数,必须或先定义的常量宏。

$$A = \begin{bmatrix} 5 & 0 & 0 & 7 \\ 0 & -3 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{bmatrix}$$

图 5.1 一个矩阵 A

a	1	2	3
0	3	4	4
1	1	1	5
2	1	4	7
3	2	2	-3
4	3	1	4

图 5.2 矩阵 A 对应的三元组表示

建立三元组存储方法

输入一个稀疏矩阵 A 建立三元组存储方法的函数如下:

```
void crt_matrix(A,B)
smatrik A,B: /* A 是一个稀疏矩阵,B 是产生的相对应的三元组 */
```

```

{
    int i,j,k=1;
    for (i=0;i<n;i++) /* 按行优先顺序扫描 A 的元素,不为 0 者存入 B 中 */
        for (j=0;j<m;j++)
            if (A[i][j] != 0)
            {
                B[k][0]=i;
                B[k][1]=j;
                B[k][2]=A[i][j];
                k++;
            }
    B[0][0]=m;
    B[0][1]=n;
    B[0][2]=k-1; /* 存入非 0 元素个数 */
}

```

元素查找

在 A 的三元组存储矩阵中查找数据值为 x 的结点的函数如下：

```

int findval(A,x)
smatrix A;
int x;
{
    int i,t;
    t=A[0][2]; /* 非 0 元素个数 */
    i=1;
    while (i<=t && A[i][2] != x) i++; /* 查找等于 x 的元素值 */
    if (i<=t) return(1);
    else return(0);
}

```

(2) 伪地址表示法

所谓伪地址就是元素在矩阵中按行优先(或列优先)顺序的相对位置(包括 0 元素一起算)。用伪地址表示法存储稀疏矩阵和三元组法相似,只是线性表的每个结点包含两个字段,一个是非 0 元素的伪地址,另一个是元素的值。这种方法共需 $2N$ (N 为非 0 元素个数)个存储单元,比三元组法节省,但要付出的代价是要计算伪地址。 $m \times n$ 的稀疏矩阵 A 的元素 $A[i][j]$ 的伪地址可用 $(i-1) * n + j$ 来计算地址。

图 5.1 的矩阵用伪地址表示法如图 5.3 所示。有关伪地址表示法的建立和查找基本运算与三元组表示法相似,这里不再讨论。

伪地址	值
1	5
4	7
6	-3
9	4

图 5.3 矩阵 A 对应的伪地址表示法

2. 链接方法

链接方法也有几种,如带行指针向量的单链表表示法和十字链表方法等。

(1) 带行指针向量的单链表表示法

该方法在每个结点中增加一个指针字段,指向本行的下一个非 0 元素,使每个行向量链接成一个单链表,再把每个行向量的头指针组成一个表头指针向量,便构成了带行指针向量的单链表表示。

图 5.1 的矩阵的带行指针向量的单链表表示法如图 5.4 所示。有关带行指针向量的单链表表示法的建立和查找基本运算与单链表的运算相似,这里不再讨论。

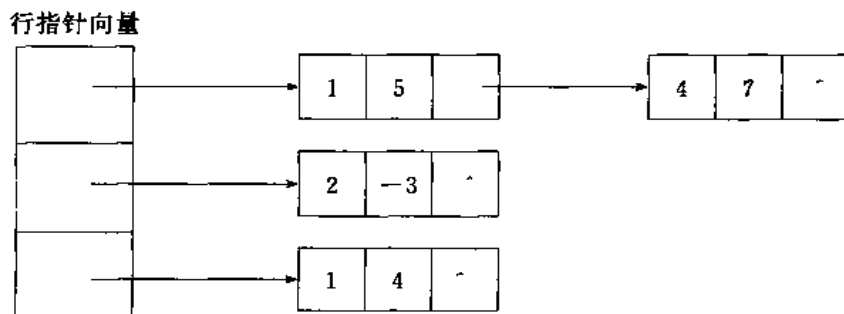


图 5.4 矩阵 A 的带行指针向量的单链表表示法

(2) 十字链表方法

在十字链表中,稀疏矩阵的每一行用一个带表头结点的循环链表表示,每一列也用一个带表头的循环链表表示,在这个结构中,除表头结点外,每个结点都代表矩阵中的一个非零元素,它由 5 个域组成:行域(row)、列域(col)、数据域(val)、向下域(down)和向右域(right),结点结构和存储表示如图 5.5 所示。

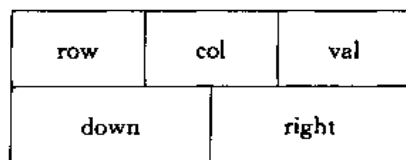


图 5.5 十字链表结点结构

图 5.1 的稀疏矩阵,用十字链表表示如图 5.6 所示。

为了使所有结点的存储结构一致,规定表头结点与非零元素结点的结构完全一致,只是将其行域和列域置为零,由于每一行链表的表头(行头)与每一列链表的表头(列)的行域、列域的值均为零,故这两组表头结点可以共用,即同行号、同列号的行头和列头共存储在一个结点之中,只是将其逻辑分开,起到共享资源的效果。由此可知稀疏矩阵的十字链表表示的结点的总数等于非零元素的个数加上行头和列头,再加上总表头 hm。

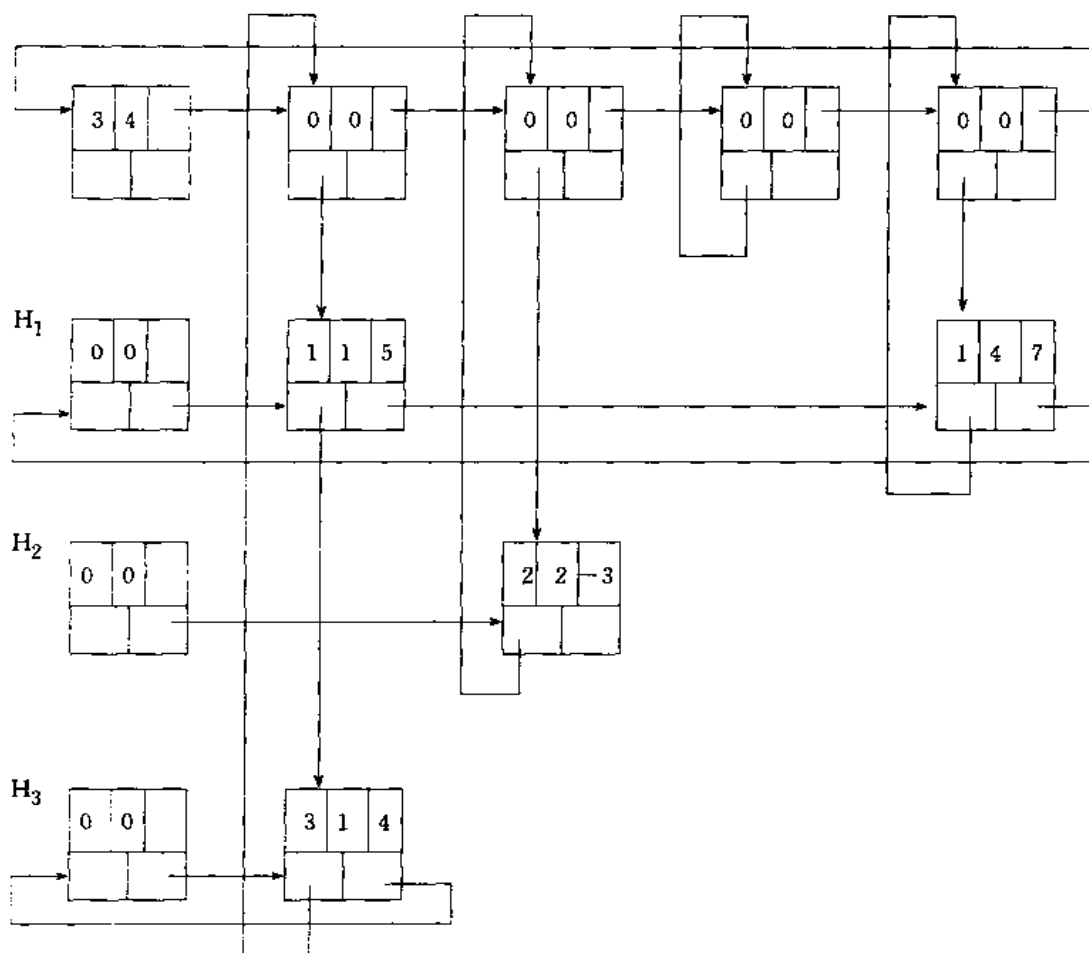


图 5.6 矩阵 A 对应的十字链表表示

十字链表的结点的定义描述如下:

```
struct matnode
{
    int row, col;
    struct matnode * right, * down;
    union { int val;
            struct matnode * next;
        } tag;
};
```

建立十字链表

输入矩阵的非零元素建立十字链表并按行方式打印该十字链表的完整程序如下:

```
#include <stdio.h>

struct matnode
{
    int row,col;
    struct matnode *right,*down;
    union { int val;
           struct matnode *next;
        } tag;
};

struct matnode * createmat()
{
    int m,n,t,s,i,r,c,v;
    struct matnode *h[100],*p,*q; /*h[]是十字链表每行的表头指针数组*/
    printf("行数 m,列数 n,非零元个数 t:");
    scanf("%d,%d,%d",&m,&n,&t);
    p=(struct matnode *)malloc(sizeof(struct matnode));
    h[0]=p;
    p->row=m;
    p->col=n;
    s=m>n?m:n;
    for (i=1;i<=s;i++)
    {
        p=(struct matnode *)malloc(sizeof(struct matnode));
        h[i]=p;
        h[i-1]->tag.next=p;
        p->row=p->col=0;
        p->down=p->right=p;
    }
    h[s]->tag.next=h[0];
    for (i=1;i<=t;i++)
    {
        printf("\t 第%d 元素(行号 r,列号 c,值 v):",i);
        scanf("%d,%d,%d",&r,&c,&v);
        p=(struct matnode *)malloc(sizeof(struct matnode));
        p->row=r;
        p->col=c;
        p->tag.val=v;
        q=h[r];
        while (q->right!=h[r] && q->right->col<c)
            q=q->right;
```

```

    p->right=q->right;
    q->right=p;
    q=h[c];
    while(q->down!=h[c] && q->down->row<r)
        q=q->down;
    p->down=q->down;
    q->down=p;
}
return(h[0]);
}
void prmat(struct matnode *hm)
{
    struct matnode *p,*q;
    printf("\n 按行表输出矩阵元素:\n");
    printf("row=%d col=%d\n",hm->row,hm->col);
    p=hm->tag.next;
    while (p!=hm)
    {
        q=p->right;
        while (p!=q)
        {
            printf("\t%d. %d, %d\n",q->row,q->col,q->tag.val);
            q=q->right;
        }
        p=p->tag.next;
    }
}
main()
{
    struct matnode *hm;
    hm=createmat();
    prmat(hm);
}

```

元素查找

在以 hm 为头的十字链表中查找数据值为 x 的结点的完整程序如下:

```

#include <stdio.h>
struct matnode
{
    int row,col;
    struct matnode *right,*down;
    union { int val;
        struct matnode *next;
    };
}

```

```

        } tag;
    };

int findmat(struct matnode * hm, int x, int * rown, int * coln)
{
    struct matnode * p, * q;
    p = hm->tag.next;
    while (p != hm)
    {
        q = p->right;
        while (q != hm)
        {
            if (q->tag.val == x)
            {
                * rown = q->row;
                * coln = q->col;
                return(1);
            }
            q = q->right;
        }
        p = p->tag.next;
    }
    return(0);
}

main()
{
    struct matnode * hm;
    int i, j, x;
    hm = createmat(); /* 建立十字链表 */
    printf("查找值:");
    scanf("%d", &x);
    if (findmat(hm, x, &i, &j))
        printf("%d 在第%d 行第%d 列\n", x, i, j);
    else
        printf("未找到\n");
}

```

5.2 基 本 题

5.2.1 单项选择题(其中 $A[i..j]$ 表示下标从 i 到 j)

1. 常对数组进行的两种基本操作是①。
- A. 建立与删除 B. 索引和修改

C. 查找和修改

D. 查找与索引

答:① C

2. 二维数组 M 的成员是 6 个字符(每个字符占一个存储单元)组成的串,行下标 i 的范围从 0 到 8,列下标 j 的范围从 1 到 10,则存放 M 至少需要 ① 个字节; M 的第 8 列和第 5 行共占 ② 个字节;若 M 按行优先方式存储,元素 $M[8][5]$ 的起始地址与当 M 按列优先方式存储时的 ③ 元素的起始地址一致。

① A. 90 B. 180 C. 240 D. 540

② A. 108 B. 114 C. 54 D. 60

③ A. $M[8][5]$ B. $M[3][10]$ C. $M[5][8]$ D. $M[0][9]$

答:① D ② A ③ B

3. 二维数组 M 的元素是 4 个字符(每个字符占一个存储单元)组成的串,行下标 i 的范围从 0 到 4,列下标 j 的范围从 0 到 5, M 按行存储时元素 $M[3][5]$ 的起始地址与 M 按列存储时元素 ① 的起始地址相同。

A. $M[2][4]$ B. $M[3][4]$ C. $M[3][5]$ D. $M[4][4]$

答:① B

4. 数组 A 中,每个元素 A 的长度为 3 个字节,行下标 i 从 1 到 8,列下标 j 从 1 到 10,从首地址 SA 开始连续存放在存储器内,存放该数组至少需要的单元数是 ①。

A. 80 B. 100 C. 240 D. 270

答:① C

5. 数组 A 中,每个元素 A 的长度为 3 个字节,行下标 i 从 1 到 8,列下标 j 从 1 到 10,从首地址 SA 开始连续存放在存储器内,该数组按行存放时,元素 $A[8][5]$ 的起始地址为 ①。

A. $SA+141$ B. $SA+144$ C. $SA+222$ D. $SA+225$

答:① C

6. 数组 A 中,每个元素 A 的长度为 3 个字节,行下标 i 从 1 到 8,列下标 j 从 1 到 10,从首地址 SA 开始连续存放在存储器内,该数组按列存放时,元素 $A[5][8]$ 的起始地址为 ①。

A. $SA+141$ B. $SA+180$ C. $SA+222$ D. $SA+225$

答:① B

7. 稀疏矩阵一般的压缩存储方法有两种,即 ①。

A. 二维数组和三维数组

B. 三元组和散列

C. 三元组和十字链表

D. 散列和十字链表

答:① C

8. 若采用三元组压缩技术存储稀疏矩阵,只要把每个元素的行下标和列下标互换,就完成了对该矩阵的转置运算,这种观点 ①。

A. 正确

B. 错误

答:① B

9. 设矩阵 A 是一个对称矩阵,为了节省存储,将其下三角部分(如图 5.7 所示)按行序

存放在一维数组 $B[1, n(n-1)/2]$ 中, 对下三角部分中任一元素 $a_{i,j} (i \geq j)$, 在一组数组 B 中下标 k 的值是 ①。

- A. $i(i-1)/2+j-1$ B. $i(i-1)/2+j$
C. $i(i+1)/2+j-1$ D. $i(i+1)/2+j$

答: ① B

$$A = \begin{bmatrix} a_{1,1} & & & \\ a_{2,1} & a_{2,2} & & \\ \dots & & & \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{bmatrix}$$

图 5.7 矩阵 A 的下三角部分

5.2.2 填空题(将正确的答案填在相应的空中, 其中 $A[i,j]$ 表示下标从 i 到 j)

1. 已知二维数组 $A[m][n]$ 采用行序为主方式存储, 每个元素占 k 个存储单元, 并且第一个元素的存储地址是 $LOC(A[0][0])$, 则 $A[i][j]$ 的地址是 ①。

答: ① $LOC(A[0][0]) + (n * i + j) * k$

2. 二维数组 $A[10][20]$ 采用列序为主方式存储, 每个元素占一个存储单元, 并且 $A[0][0]$ 的存储地址是 200, 则 $A[6][12]$ 的地址是 ①。

答: ① 332

3. 二维数组 $A[10..20][5..10]$ 采用行序为主方式存储, 每个元素占 4 个存储单元, 并且 $A[10][5]$ 的存储地址是 1000, 则 $A[18][9]$ 的地址是 ①。

答: ① 1208

4. 有一个 10 阶对称矩阵 A , 采用压缩存储方式(以行序为主存储, 且 $A[0][0]=1$), 则 $A[8][5]$ 的地址是 ①。

答: ① 42

5. 设 n 行 n 列的下三角矩阵 A 已压缩到一维数组 $S[1..n * (n+1)/2]$ 中, 若按行序为主存储, 则 $A[i][j]$ 对应的 S 中的存储位置是 ①。

答: ① $i * (i+1)/2 + j + 1$

6. 一个稀疏矩阵如图 5.8 所示, 则对应的三元组表示为 ①。

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

图 5.8 一个稀疏矩阵

答: ① 如图 5.9 所示。

	1	2	3
0	4	4	4
1	1	3	2
2	2	1	3
3	3	3	-1
4	3	4	5

图 5.9 三元组表示

5.3 习题解析

1. 按行优先顺序和按列优先顺序列出四维数组 $A[2][2][2][2]$ 所有元素在内存中的存储次序。

解: 该四维数组 A 的按行优先顺序在内存中的存储次序:

```

A[0][0][0][0]
A[0][0][0][1]
A[0][0][1][0]
A[0][0][1][1]
A[0][1][0][0]
A[0][1][0][1]
A[0][1][1][0]
A[0][1][1][1]
A[1][0][0][0]
A[1][0][0][1]
A[1][0][1][0]
A[1][0][1][1]
A[1][1][0][0]
A[1][1][0][1]
A[1][1][1][0]
A[1][1][1][1]

```

该四维数组 A 的按列优先顺序在内存中的存储次序:

```

A[0][0][0][0]
A[1][0][0][0]
A[0][1][0][0]
A[1][1][0][0]
A[0][0][1][0]
A[1][0][1][0]
A[0][1][1][0]
A[1][1][1][0]
A[0][0][0][1]

```

```

A[1][0][0][1]
A[0][1][0][1]
A[1][1][0][1]
A[0][0][1][1]
A[1][0][1][1]
A[0][1][1][1]
A[1][1][1][1]

```

2. 设给定 n 维数组 $A[l_1, u_1][l_2, u_2] \dots [l_n, u_n]$, 如果 $A[l_1][l_2] \dots [l_n]$ 的存储地址是 a , 每个元素占用 1 个存储单元。求出 $A[i_1][i_2] \dots [i_n]$ 的存储地址。

解: 若整个数组采用按行优先存储, 则 $A[i_1][i_2] \dots [i_n]$ 的存储地址如下:

$$\begin{aligned} \text{LOC}(A[i_1][i_2] \dots [i_n]) = & a + (i_1 - l_1) * (u_2 - l_2 + 1) * \dots * (u_n - l_n + 1) + \\ & (i_2 - l_2) * (u_3 - l_3 + 1) * \dots * (u_n - l_n + 1) + \\ & \vdots \\ & (i_n - l_n) \end{aligned}$$

若整个数组采用按列优先存储, 则 $A[i_1][i_2] \dots [i_n]$ 的存储地址如下:

$$\begin{aligned} \text{LOC}(A[i_1][i_2] \dots [i_n]) = & a + (i_n - l_n) * (u_n - l_n + 1) * \dots * (u_2 - l_2 + 1) + \\ & (i_n - l_n) * (u_n - l_n + 1) * \dots * (u_3 - l_3 + 1) + \\ & \vdots \\ & (i_1 - l_1) \end{aligned}$$

* 3. 对于二维数组 $A[m][n]$, 其中 $m \leq 80, n \leq 80$, 先读入 m 和 n , 然后读该数组的全部元素, 对如下三种情况分别编写相应函数:

- (1) 求数组 A 靠边元素之和;
- (2) 求从 $A[0][0]$ 开始的互不相邻的各元素之和;
- (3) 当 $m=n$ 时, 分别求两条对角线上的元素之和, 否则打印出 $m \neq n$ 的信息。

解:

(1) 本小题是计算数组 A 的最外围的 4 条边的所有元素之和, 先分别求出各边的元素之和, 累加后减除 4 个角的重复相加的元素即为所求。

(2) 本小题的互不相邻是指上、下、左、右、对角线均不相邻, 即求第 0, 2, 4, ... 的各行中第 0, 2, 4, ... 列的所有元素之和, 函数中用 i 和 j 变量控制即可。

(3) 本小题中一条对角线是 $A[i][i]$, 其中 $(0 \leq i \leq m-1)$, 另一条对角线是 $A[m-i-1, i]$, 其中 $(0 \leq i \leq m-1)$, 因此用循环实现即可。实现本题功能的程序如下:

```

#include <stdio.h>
/* 实现(1)小题功能的函数 */
void proc1(maxix A)
{
    int s=0, i, j;
    for (i=0; i<m; i++)    /* 第一列 */
        s=s+A[i][1];
}

```

```

    for (i=0;i<m;i++)    /* 最后一列 */
        s=s+A[i][n];
    for (j=0;j<n;j++)    /* 第一行 */
        s=s+A[1][j];
    for (j=0;j<m;j++)    /* 最后一行 */
        s=s+A[m][j];
    s=s-A[0][0]-A[0][n-1]-A[m-1][0]-A[m-1][n-1];
        /* 减去 4 个角的重复元素值 */
    printf("s=%d\n",s);
}
/* 实现(2)小题功能的函数 */
void proc2(maxix A)
{
    int s=0,i,j;
    i=0;
    do
    {
        j=0;
        do
        {
            s=s+A[i][j];
            j=j+2;    /* 跳过一列 */
        } while (j<n);
        i=i+2;    /* 跳过一行 */
    } while (i<m);
    printf("s=%d\n",s);
}
/* 实现(3)小题功能的函数 */
void proc3(maxix A)
{
    int i,s;
    if (m!=n) printf("m≠n");
    else
    {
        s=0;
        for (i=0;i<m;i++)
            s=s+A[i][i];    /* 求第一条对角线之和 */
        for (i=0;i<n;i++)
            s=s+A[n-i-1][i];    /* 累加第二条对角线之和 */
        printf("s=%d\n",s);
    }
}
main()
{

```

```

int m,n,i,j;
maxix A;
printf("m,n:");
scanf("%d,%d",&m,&n);
printf("元素值:\n");
for (i=0;i<m;i++)
    for (j=0;j<n;j++)
        scanf("%d",&A[i][j]);
proc1(A);
proc2(A);
proc3(A);
}

```

* 4. 有数组 $A[4][4]$, 把 1 到 16 个整数分别按顺序放入 $A[0][0], \dots, A[0][3], A[1][0], \dots, A[1][3], A[2][0], \dots, A[2][3], A[3][0], \dots, A[3][3]$ 中, 编写一个函数获取数据并求出两条对角线元素的乘积。

解: 数组 $A[3][3]$ 中一条对角线是 $A[i][i]$ 其中 $(0 \leq i \leq 3)$, 另一条对角线是 $A[3-i][i]$ 其中 $(0 \leq i \leq 3)$, 因此用循环扫描两条对角线中的每个元素, 依次计算其乘积。因此, 实现本题功能的函数如下:

```

void mmult()
{
    maxix A;
    int i,s;
    for (i=0;i<4;i++)
        for (j=0;j<4;j++)
            scanf("%d",&A[i][j]);
    s=1;
    for (i=0;i<4;i++)
        s=s*A[i][i]; /* 求第一条对角线之积 */
    for (i=0;i<4;i++)
        s=s*A[3-i][i]; /* 累加第二条对角线之积 */
    printf("两条对角线元素之积:%d\n",s);
}

```

5. n 只猴子要选大王, 选举办法如下: 所有猴子按 $1, 2, \dots, n$ 编号围坐一圈, 从第 1 号开始按 $1, 2, \dots, m$ 报数, 凡报 m 号的退出到圈外, 如此循环报数, 直到圈内剩下一只猴子时, 这只猴子就是大王。 n 和 m 由键盘输入, 打印出最后剩下的猴子号。编写一个程序实现上述函数。

解: 本题用一个含有 n 个元素的数组 a , 初始时 $a[i]$ 中存放猴子的编号 i , 计数器 $count$ 的值为 0。从 $a[i]$ 开始循环报数, 每报一次, 计数器的值加 1, 凡报到 m 时便打印出 $a[i]$ 的值 (退出圈外的猴子的编号), 同时将 $a[i]$ 的值改为 0 (以后它不再参加报数), 计数器的值重新置为 0。

该函数一直进行到 n 只猴子全部退出圈外为止,最后退出的猴子就是大王。因此,实现本题功能的程序如下:

```
#include <stdio.h>
typedef int maxix[100];
main()
{
    maxix a;
    int count,d,i,m,n;
    do
    {
        printf("输入 n 和 m:");
        scanf("%d,%d",&n,&m);
    } while (n<=m);
    for (i=0;i<n;i++)
        a[i]=i+1;
    count=0;d=0; /* d 用来记录退出圈外的猴子数目 */
    while (d<n)
        for (i=0;i<n;i++)
            if (a[i] != 0)
            {
                count++; /* 报一次数 */
                if (count==m) /* 第 i 个猴子退出 */
                {
                    printf("%d ",a[i]);
                    a[i]=0;
                    count=0;
                    d++;
                }
            }
    }
```

执行本程序:

```
输入 n 和 m:10,2 \
2 4 6 8 10 3 7 1 9 5
```

* 6. 如果矩阵 A 中存在这样的一个元素 $A[i][j]$ 满足下列条件: $A[i][j]$ 是第 i 行中值最小的元素,且又是第 j 列中值最大的元素,则称之为该矩阵的一个马鞍点。编写一个函数计算出 $m \times n$ 的矩阵 A 的所有马鞍点。

解:依题意,先求出每行的最小值元素,放入 $\min[m]$ 之中,再求出每列的最大值元素,放入 $\max[n]$ 之中,若某元素既在 $\min[i]$ 中,又在 $\max[j]$ 中,则该元素 $A[i][j]$ 便是马鞍点,找出所有这样的元素,即找到了所有马鞍点。因此,实现本题功能的程序如下:

```
#include <stdio.h>
```

```

#define m 3
#define n 4
void minmax(int A[m][n])
{
    int i, j, have = 0;
    int min[m], max[n];
    for (i = 0; i < m; i++) /* 计算出每行的最小值元素, 放入 min[m] 之中 */
    {
        min[i] = A[i][0];
        for (j = 1; j < n; j++)
            if (A[i][j] < min[i]) min[i] = A[i][j];
    }
    for (j = 0; j < n; j++) /* 计算出每列的最大值元素, 放入 max[n] 之中 */
    {
        max[j] = A[0][j];
        for (i = 1; i < m; i++)
            if (A[i][j] > max[j]) max[j] = A[i][j];
    }
    for (i = 0; i < m; i++) /* 判定是否为马鞍点 */
    for (j = 0; j < n; j++)
        if (min[i] == max[j])
        {
            printf("(%d, %d): %d\n", i, j, A[i][j]); /* 显示马鞍点 */
            have = 1;
        }
    if (!have)
        printf("没有鞍点\n");
}

main()
{
    int a[m][n];
    int i, j;
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &a[i][j]);
    minmax(a);
}

```

7. 现有如下的稀疏矩阵 A 如图 5.10 所示, 要求画出以下各种表示法。

15	0	0	22	0	-15
0	13	3	0	0	0
0	0	0	-6	0	0
0	0	0	0	0	0
91	0	0	0	0	0
0	0	28	0	0	0

图 5.10 稀疏矩阵 A

- (1)三元组表示法；
 (2)伪地址表示法；
 (3)带行指针向量的单链表表示法；
 (4)十字链表示法。

解：

- (1)三元组表示法如图 5.11 所示。

	1	2	3
0	6	6	8
1	1	1	15
2	1	4	22
3	1	6	-15
4	2	2	13
5	2	3	3
6	3	4	-6
7	5	1	91
8	6	3	28

图 5.11 三元组表示

- (2)伪地址表示法如图 5.12 所示。

伪地址	值
1	15
4	22
6	-15
8	13
9	3
16	-6
25	91
33	28

图 5.12 伪地址表示

- (3)带行指针向量的单链表表示法如图 5.13 所示。

行指针向量

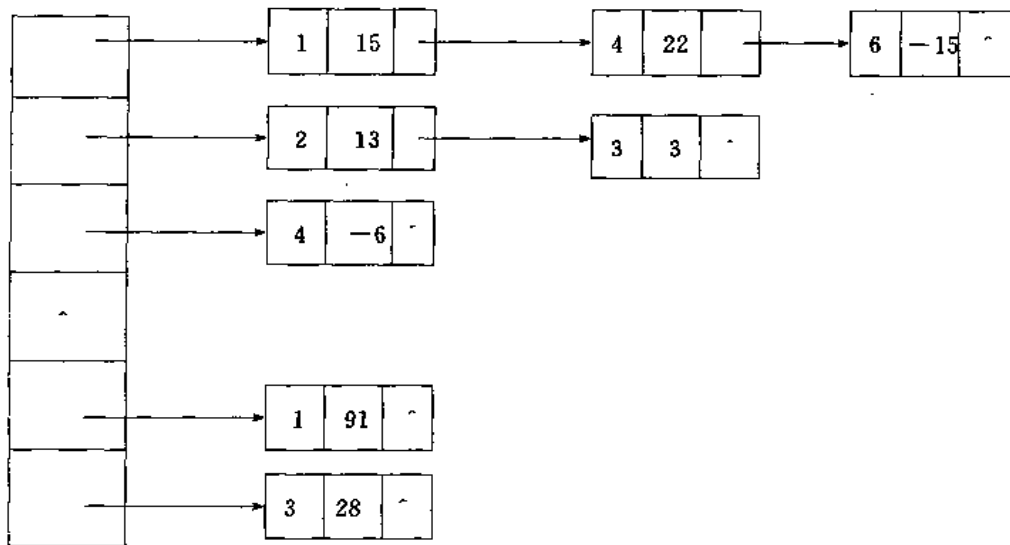


图 5.13 单链表表示

(4) 十字链表示法如图 5.14 所示。

8. 假设稀疏矩阵 A 采用三元组表示, 编写一个函数计算其转置矩阵 B, 要求 B 也采用三元组表示。

解: 三元组表示中要求按行的顺序存放, 所有转置过程不能直接将行下标和列下标转换, 还必须使得列按顺序存放。因此在 A 中首先找出第一列中的所有元素, 它们是转置矩阵第一行的非 0 元素, 并把它们依次放在转置矩阵三元组数组 B 中; 然后依次找出第二列中的所有元素, 把它们依次放在数组 B 中; 按照同样的方法逐列进行, 直到找出第 n 列的所有元素, 并把它们依次放在数组 B 中。实现本题功能的函数如下:

```

void transpose(A,B)
smatrix A,B; /* A 是稀疏矩阵的三元组形式,B 是存放 A 的转置矩阵的三元组数组 */
{
    int m,n,p,q,t,col;
    /* m 为 A 中的行数,n 为 A 中的列数;t 为 A 中非 0 元素个数 */
    /* q 为 B 的下一项位置,p 为 A 的当前项 */
    m=A[0][0]; n=A[0][1]; t=A[0][2];
    B[0][0]=n; B[0][1]=m; B[0][2]=t; /* 产生第 0 行的结果 */
    if (t>0) /* 非 0 矩阵才做转置 */
    {
        q=1;
        for (col=1; col<=n; col++) /* 按列转置 */
            for (p=1; p<=t; p++)
                if (A[p][1]==col)
                {
                    B[q][0]=A[p][1];

```

$$B[q][1]=A[p][0];$$

$$B[q][2]=A[p][2];$$

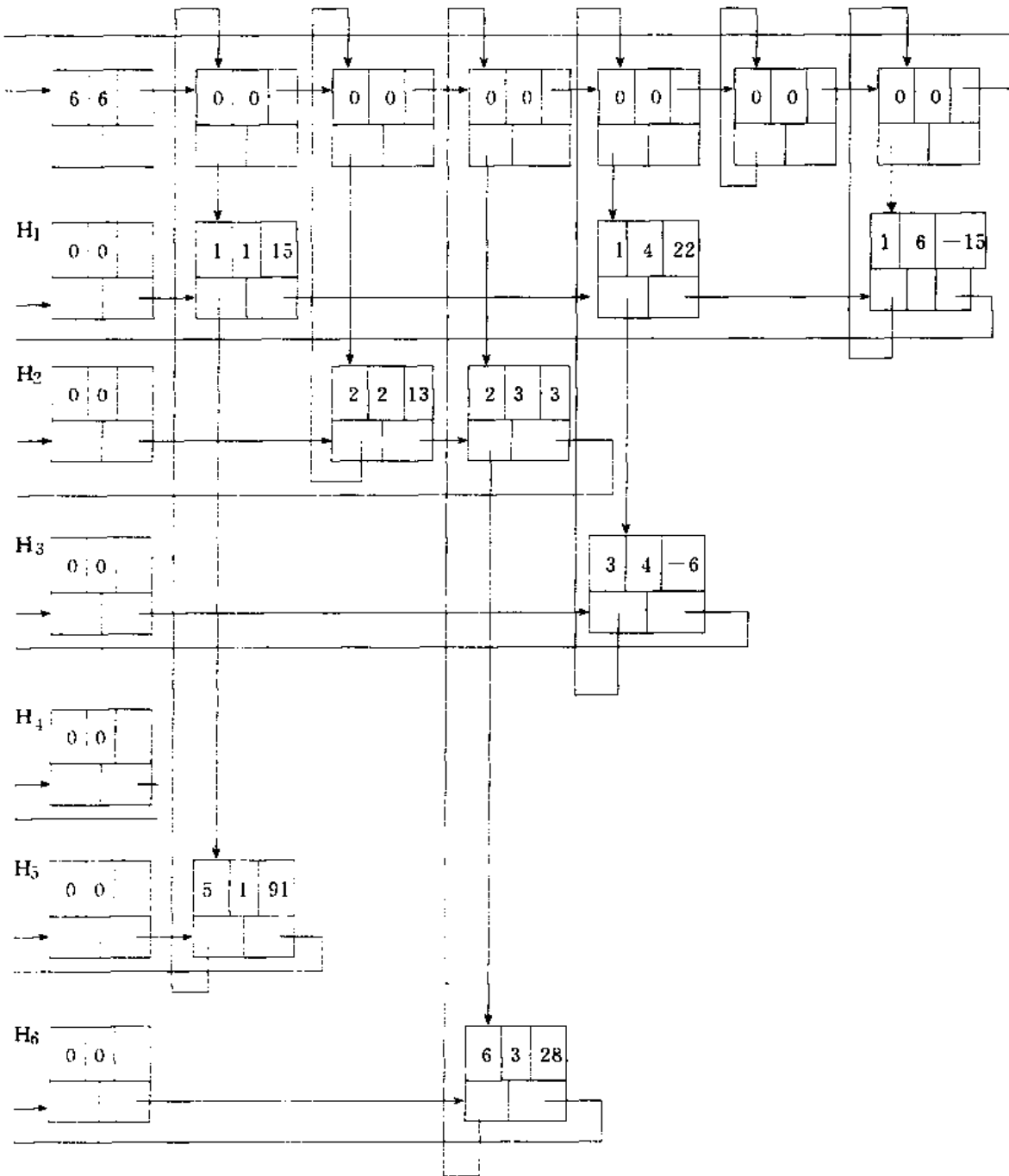
$$q++;$$


图 5.14 十字链表示

9. 假设稀疏矩阵 A 和 B(具有相同的大小 $m \times n$) 都采用三元组表示,编写一个函数计算 $C=A+B$,要求 C 也采用三元组表示。

解:本题采用的算法思想是:依次扫描 A 和 B 的行号和列号,若 A 的当前项的行号等于 B 的当前项的行号,则比较其列号,将较小列的项存入 C 中,如果列号也相等,则将对应的元素值相加后存入 C 中;若 A 的当前项的行号小于 B 的当前项的行号,则将 A 的项存入 C 中;若 A 的当前项的行号大于 B 的当前项的行号,则将 B 的项存入 C 中,这样产生了 C。因此,实现本题功能的函数如下:

```
void matadd(A,B,C)
smatrik A,B,C;
{
    int i=1,j=1,k=1;
    while (i<=A[0][2] && j<=B[0][2])
        /* 若 A 的当前项的行号等于 B 的当前项的行号,则比较其列号,将较小列的项 */
        /* 存入 C 中,如果列号也相等,则将对应的元素值相加后存入 C 中 */
        if (A[i][0]==B[j][0])
            if (A[i][1]<B[j][1])
            {
                C[k][0]=A[i][0];
                C[k][1]=A[i][1];
                C[k][2]=A[i][2];
                k++;
                i++;
            }
            else if (A[i][1]>B[j][1])
            {
                C[k][0]=B[j][0];
                C[k][1]=B[j][1];
                C[k][2]=B[j][2];
                k++;
                j++;
            }
            else
            {
                C[k][0]=B[j][0];
                C[k][1]=B[j][1];
                C[k][2]=A[i][2]+B[j][2];
                k++;
                i++;
                j++;
            }
        else if (A[i][0]<B[j][0])
            /* 若 A 的当前项的行号小于 B 的当前项的行号,则将 A 的项存入 C 中 */
```

```

    {
        C[k][0]=A[i][0];
        C[k][1]=A[i][1];
        C[k][2]=A[i][2];
        k++;
        i++;
    }
else
/* 若 A 的当前项的行号大于 B 的当前项的行号,则将 B 的项存入 C 中 */
{
    C[k][0]=B[j][0];
    C[k][1]=B[j][1];
    C[k][2]=B[j][2];
    k++;
    j++;
}
C[0][0]=A[0][0]; /* 产生第 0 行的结果 */
C[0][1]=A[0][1];
C[0][2]=k-1;
}

```

10. 假设稀疏矩阵 A 和 B(分别为 $m \times n$ 和 $n \times l$ 矩阵)采用三元组表示,编写一个函数计算 $C=A * B$,要求 C 也是采用稀疏矩阵的三元组表示。

解:本题采用矩阵相乘的基本方法,关键是通过给定的行号 i 和列号 j 找出原矩阵的对应元素值,这里设计了一个函数 value,当在三元组表示中找到时返回其元素值,找不到时说明原该位置处的元素值为 0,因此返回 0。然后利用该函数计算出 C 的行号 i 和列号 j 处的元素值,若该值不为 0,则存入其三元组表示的矩阵中,否则不存入。因此,实现本题功能的函数如下:

```

int value(C,i,j,integer)
smatrik C;
int i,j;
{
    int k=1;
    while (k<=C[0][2] && C[k][0]!=i && C[k][1]!=j) k++;
    if (k<=C[0][2]) return(C[k][2]);
    else return(0);
}

matmul(m,n,k,A,B,C)
int m,n,k;
smatrik A,B,C;
{
    int i,j,p,s;

```

```

p=1;
for (i=0;i<m;i++)
    for (j=0;j<k;j++)
    {
        s=0;
        for (s=0;s<n;s++)
            s=s+value(A,i,s)*value(B,s,j);
        if (s!=0) /*产生一个三元组元素*/
        {
            C[k][0]=i;
            C[k][1]=j;
            C[k][2]=s;
            k++;
        }
    }
C[0][0]=m; /*产生第0行的结果*/
C[0][1]=k;
C[0][2]=p-1;
}

```

* 11. 假设稀疏矩阵只存放其非 0 元素的行号、列号和数值,以一维数组顺次存放,行号为 -1 作结束标志。例如,如图 5.15 所示的稀疏矩阵 M,则存放在一维数组 D 中:

$D[0]=1, D[1]=1, D[2]=1, D[3]=1, D[4]=5,$
 $D[5]=10, D[6]=3, D[7]=9, D[8]=5, D[9]=-1$

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 10 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

图 5.15 稀疏矩阵 M

现有两个如上方法存储的稀疏矩阵 A 和 B,它们均为 m 行 n 列,分别存放在数组 A 和 B 中,编写求矩阵加法 $C=A+B$ 的算法,C 亦放在数组 C 中。

解:依题意,本题采用了一个稀疏矩阵到一维数组之间的映射关系,在进行加法运算时,依次扫描 A 和 B 的行列值,且以行为主,当行列相同时,将第三个元素值相加产生的三个元素放入结果数组中,不相同,将 A 或 B 的三个元素直接放入结果数组中。因此,实现本题功能的过程如下:

```

matrixadd(A,B,C)
int A[MAX],B[MAX],C[MAX]; /*MAX是一个预定义的常量*/
{
    int i=1,j=1,k=1; /*i是A的下标,j是B的下标,k是C的下标,这些下标都从0开始*/

```

```

while (A[i] != -1 && B[j] != -1)          /* 循环直到 A 和 B 均结束 */
{
    if (A[i] == B[j])                    /* 行相等 */
    {
        if (A[i+1] == B[j+1]) C[k+2] = A[i+2] + B[j+2]; /* 且列相等 */
        {
            if (C[k+2] != 0)                /* 结果值大于 0 时放入 C 中 */
            {
                C[k] = A[i];
                C[k+1] = A[i+1];
                k = k+3;
                i = i+3;
            }
        }
        else if (A[i+1] < B[j+1]) /* A 的列小于 B 的列, 将 A 的 3 个元素直接放入 C 中 */
        {
            C[k] = A[i];
            C[k+1] = A[i+1];
            C[k+2] = A[i+2];
            k = k+3;
            i = i+3;
        }
        else /* B 的列小于 A 的列, 将 B 的 3 个元素直接放入 C 中 */
        {
            C[k] = B[j];
            C[k+1] = B[j+1];
            C[k+2] = B[j+2];
            k = k+3;
            j = j+3;
        }
    }
    else if (A[i] < B[j]) /* A 的行小于 B 的行, 将 A 的 3 个元素直接放入 C 中 */
    {
        C[k] = A[i];
        C[k+1] = A[i+1];
        C[k+2] = A[i+2];
        k = k+3;
        i = i+3;
    }
    else /* B 的行小于 A 的行, 将 B 的 3 个元素直接放入 C 中 */
    {
        C[k] = B[j];
        C[k+1] = B[j+1];
        C[k+2] = B[j+2];
    }
}

```

```

        k=k+3;
        j=j+3;
    }
    /* 循环结束 */
    if (A[i] == -1) /* A 结束, B 还有元素, 则将 B 的所剩元素直接放入 C 中 */
        while (B[j] != -1)
        {
            C[k]=B[j];
            C[k+1]=B[j+1];
            C[k+2]=B[j+2];
            k=k+3;
            j=j+3;
        }
    else /* B 结束, A 还有元素, 则将 A 的所剩元素直接放入 C 中 */
        while (A[i] != -1)
        {
            C[k]=A[i];
            C[k+1]=A[i+1];
            C[k+2]=A[i+2];
            k=k+3;
            i=i+3;
        }
}

```

* 12. 已知 A 和 B 为两个 $n \times n$ 阶的对称矩阵, 输入时, 对称矩阵只输入下三角形元素, 存入一维数组, 如图 5.16 所示(图中 x 可以是任何整数), 编写一个计算对称矩阵 A 和 B 的乘积的函数。

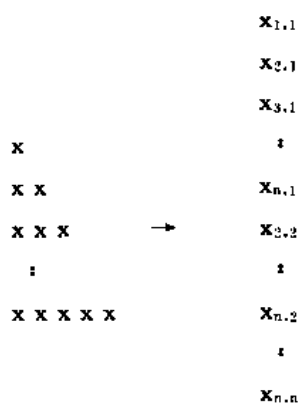


图 5.16 矩阵的存储转换形式

解: 依题意, 对称矩阵第 i 行和第 j 列的元素的数据在一维数组中的位置是:

$$\frac{i * (i-1)}{2} + j \quad (\text{当 } i \geq j \text{ 时})$$

$$\frac{j * (j-1)}{2} + i \quad (\text{当 } i < j \text{ 时})$$

因此,实现本题功能的函数如下:

```
void mult(a,b,c,n)
int a[MAX],b[MAX]; /* MAX 是一个预定义的常量 */
matix c;
int n;
{
    int i,j,k,t1,t2,s;
    for (i=0;i<n;i++)
        for (j=0;j<n;j++)
        {
            s=0;
            for (k=0;k<n;k++)
            {
                if (i>=k) t1=i * (i-1) / 2 + k;
                else t1=k * (k-1) / 2 + i;
                if (k>=j) t2=k * (k-1) / 2 + j;
                else t2=j * (j-1) / 2 + k;
                s=s+a[t1] * b[t2];
            }
            c[i][j]=s;
        }
}
```

13. 已知两个稀疏矩阵 A 和 B 采用十字链表方式存储,计算 $C=A+B$, C 也采用十字链表方式存储。

解:依题意, $C=A+B$,则 C 中的非零元素 c_{ij} 只可能有 3 种情况:或者是 $a_{ij}+b_{ij}$,或者是 $a_{ij}(b_{ij}=0)$ 或者是 $b_{ij}(a_{ij}=0)$ 。因此,当 B 加到 A 上时,对 A 矩阵的十字链表来说,或者是改变结点的 val 域值($a_{ij}+b_{ij} \neq 0$),或者不变($b_{ij}=0$),或者插入一个新结点($a_{ij}=0$),还可能是删除一个结点($a_{ij}+b_{ij}=0$)。整个运算可从矩阵的第一行起逐行进行。对每一行都从行表头出发分别找到 A 和 B 在该行中的第一个非零元素结点后开始比较,然后按 4 种不同情况分别处理(假设 pa 和 pb 分别指向 A 和 B 的十字链表中行值相同的两个结点):

(1) 若 $pa \rightarrow col = pb \rightarrow col$ 且 $pa \rightarrow val = pb \rightarrow val \neq 0$,则只要将 $a_{ij}+b_{ij}$ 的值送到 pa 所指结点的值域中即可。

(2) 若 $pa \rightarrow col = pb \rightarrow col$ 且 $pa \rightarrow val + pb \rightarrow val = 0$,则需要在 A 矩阵的十字链表中删除 pa 所指结点,此时需改变同一行中前一结点的 right 域值,以及同一列中前一结点的 down 域值。

(3) 若 $pa \rightarrow col < pb \rightarrow col$ 且 $pa \rightarrow col \neq 0$ (即不是表头结点),则只需要将 pa 指针往右推进一步,并重新加以比较。

(4) 若 $pa \rightarrow col > pb \rightarrow col$ 或 $pa \rightarrow col = 0$,则需要在 A 矩阵的十字链表中插入一个

值为 b_{ij} 的结点。

实现本题功能的程序如下：

```
#include <stdio.h>
#define MAX 100
struct matnode
{
    int row,col;
    struct matnode *right,*down;
    union { int val;
            struct matnode *next;
        } tag;
};

struct matnode * createmat(struct matnode * h[])
/* h 是建立的十字链表各行首指针的数组 */
{
    int m,n,t,s,i,r,c,v;
    struct matnode * p,*q;
    printf("行数 m,列数 n,非零元个数 t:");
    scanf("%d,%d,%d",&m,&n,&t);
    p=(struct matnode *)malloc(sizeof(struct matnode));
    h[0]=p;
    p->row=m;
    p->col=n;
    s=m>n?m:n;
    for (i=1;i<=s;i++)
    {
        p=(struct matnode *)malloc(sizeof(struct matnode));
        h[i]=p;
        h[i-1]->tag.next=p;
        p->row=p->col=0;
        p->down=p->right=p;
    }
    h[s]->tag.next=h[0];
    for (i=1;i<=t;i++)
    {
        printf("\t第 %d 元素(行号 r,列号 c,值 v):",i);
        scanf("%d,%d,%d",&r,&c,&v);
        p=(struct matnode *)malloc(sizeof(struct matnode));
        p->row=r;
        p->col=c;
        p->tag.val=v;
        q=h[r];
```

```

    while (q->right != h[r] && q->right->col < c)
        q = q->right;
    p->right = q->right;
    q->right = p;
    q = h[c];
    while (q->down != h[c] && q->down->row < r)
        q = q->down;
    p->down = q->down;
    q->down = p;
}
return(h[0]);
}

void prmat(struct matnode *hm)
{
    struct matnode *p, *q;
    printf("\n 按行表输出矩阵元素:\n");
    printf("row = %d col = %d\n", hm->row, hm->col);
    p = hm->tag.next;
    while (p != hm)
    {
        q = p->right;
        while (p != q)
        {
            printf("\t%d, %d, %d\n", q->row, q->col, q->tag.val);
            q = q->right;
        }
        p = p->tag.next;
    }
}

struct matnode *colpred(i, j, h)
/* 根据 i(行号)和 j(列号)找出矩阵第 i 行第 j 列的非零元素在十字链表中的前驱结点 */
int i, j;
struct matnode *h[];
{
    struct matnode *d;
    d = h[j];
    while (d->down->col != 0 && d->down->row < i)
        d = d->down;
    return(d);
}

struct matnode *addmat(ha, hb, h)
struct matnode *ha, *hb, *h[];

```

```

{
    struct matnode *p, *q, *ca, *cb, *pa, *pb, *qa;
    if (ha->row != hb->row || ha->col != hb->col)
    {
        printf("两个矩阵不是同类型的,不能相加\n");
        exit(0);
    }
    else
    {
        ca=ha->tag.next;
        cb=hb->tag.next;
        do
        {
            pa=ca->right;
            pb=cb->right;
            qa=ca;
            while (pb->col != 0)
                if (pa->col < pb->col && pa->col != 0)
                {
                    qa=pa;
                    pa=pa->right;
                }
            else
                if (pa->col > pb->col || pa->col == 0)
                {
                    p=(struct matnode *)malloc(sizeof(struct matnode));
                    *p=*pb;
                    p->right=pa;
                    qa->right=p;
                    qa=p;
                    q=colpred(p->row, p->col, h);
                    p->down=q->down;
                    q->down=p;
                    pb=pb->right;
                }
            else
            {
                pa->tag.val += pb->tag.val;
                if (pa->tag.val == 0)
                {
                    qa->right=pa->right;
                    q=colpred(pa->row, pa->col, h);
                    q->down=pa->down;
                }
            }
        } while (ca != 0);
    }
}

```

```
        free(pa);
    }
    else qa=pa;
    pa=pa->right;
    pb=pb->right;
}
ca=ca->tag.next;
cb=cb->tag.next;
} while (ca->row==0);
}
return(h[0]);
}

main()
{
    struct matnode *hm, *hm1, *hm2;
    struct matnode *h[MAX], *h1[MAX];
    printf("第一个矩阵:\n");
    hm1=createmat(h);
    printf("第二个矩阵:\n");
    hm2=createmat(h1);
    hm=addmat(hm1,hm2,h);
    prmat(hm);
}
```

第6章 递 归

在定义一个过程或函数时又出现了调用本过程或者函数的成分,即调用它自己本身,这称之为直接递归,若过程或函数 p 调用过程或函数 q ,而 q 又调用 p ,这称之为间接递归。递归是计算机学科中经常遇到的问题。本章讨论递归模型及递归程序设计的相关问题。

6.1 递归设计方法

递归设计先要确定求解问题的递归模型,了解递归的执行过程,在此基础上进行递归程序设计,同时还要掌握从递归过程到非递归过程的转换过程。

6.1.1 递归模型

递归模型反映一个递归问题的递归结构,例如:

$$\begin{aligned} f(1) &= 1 \\ f(n) &= n * f(n-1) \quad n > 1 \end{aligned}$$

第一个式子给出了递归的终止条件,第二个式子给出了 $f(n)$ 的值与 $f(n-1)$ 的值之间的关系,我们把第一个式子称为递归出口,把第二个式子称为递归体。

一般地,一个递归模型是由递归出口和递归体两部分组成,前者确定递归到何时为止,后者确定递归的方式。

递归出口的一般格式为:

$$f(s_0) = m_0$$

这里的 s_0 与 m_0 均为常量,有的递归问题可能有几个递归出口。

递归体的一般格式为:

$$f(s) = g(f(s_1), f(s_2), \dots, f(s_n), c_1, c_2, \dots, c_m)$$

这里的 s 是一个递归“大问题”, s_1, s_2, \dots, s_n 为递归“小问题”, c_1, c_2, \dots, c_m 是若干个可以直接(用非递归方法)解决的问题, g 是一个非递归函数,反映了递归问题的结构。

6.1.2 递归的执行过程

实际上,递归是把一个不能或不好直接求解的“大问题”转化成一个或几个“小问题”来解决,再把这些“小问题”进一步分解成更小的“小问题”来解决,如此分解,直至每个“小问题”都可以直接解决(此时分解到递归出口)。

注意:递归分解不是随意的分解,递归分解要保证“大问题”与“小问题”相似,即求解过程与环境都相似。

为了讨论方便,简化上述递归模型:

$$\begin{aligned} f(s_0) &= m_0 \\ f(s) &= g(f(s'), c) \end{aligned}$$

求 $f(s_n)$ 的分解过程如下:

$$\begin{aligned} &f(s_n) \\ &\downarrow \\ &f(s_{n-1}) \\ &\downarrow \\ &\dots \\ &\downarrow \\ &f(s_1) \\ &\downarrow \\ &f(s_0) \end{aligned}$$

一旦遇到递归出口,分解过程结束,开始求值过程,所以分解过程是“量变”过程,即原来的“大问题”在慢慢变小,但尚未解决,遇到递归出口后,便发生了“质变”,即原递归问题便转化成直接问题。上面的求值过程如下:

$$\begin{aligned} &f(s_0) = m_0 \\ &\downarrow \\ &f(s_1) = g(f(s_0), c_0) \\ &\downarrow \\ &f(s_2) = g(f(s_1), c_1) \\ &\downarrow \\ &\dots \\ &\downarrow \\ &f(s_n) = g(f(s_{n-1}), c_{n-1}) \end{aligned}$$

这样 $f(s_n)$ 便计算出来了,因此,递归的执行过程由分解和求值两部分构成。

6.1.3 递归设计

递归设计先要给出递归模型,再转换成对应的 C 语言函数。

从递归的执行过程看,要解决 $f(s)$,不是直接求其解,而是转化为计算 $f(s')$ 和一个常量 c' ,求解 $f(s')$ 的方法与环境 and 求解 $f(s)$ 的方法与环境是相似的,但 $f(s)$ 是一个“大问题”,而 $f(s')$ 是一个“较小问题”,尽管 $f(s')$ 还未解决,但向解决目标靠近了一步,这就是一个“量变”,如此到达递归出口时,便发生了“质变”,递归问题解决了。因此,递归设计就是要给出合理的“较小问题”,然后确定“大问题”的解与“较小问题”之间的关系,即确定递归体;最后朝此方向分解,必然有一个简单基本问题解,以此作为递归出口。由此得出递归设计的步骤如下:

- (1) 对原问题 $f(s)$ 进行分析,假设出合理的“较小问题” $f(s')$;
- (2) 假设 $f(s')$ 是可解的,在此基础上确定 $f(s)$ 的解,即给出 $f(s)$ 与 $f(s')$ 之间的关系;
- (3) 确定一个特定情况(如 $f(1)$ 或 $f(0)$)的解,由此作为递归出口。

6.1.4 递归到非递归的转换

求解递归问题有两种方式,一种是直接求值,不需要回溯的;另一种是不能直接求值,需

要回溯的。这两种方式在转换成非递归问题时采用的方法也不相同。前者使用一些中间变量保存中间结果,称为直接转换法;后者需要回溯,所以要用栈保存中间结果,称为间接转换法,下面分别讨论这两种方法。

1. 直接转换法

该方法使用一些中间变量保存中间结果。

例 1. 编写一个过程采用非递归方法计算如下递归函数的值:

$$\begin{aligned} f(1) &= 1 \\ f(2) &= 1 \\ f(n) &= f(n-1) + f(n-2) \quad n > 2 \end{aligned}$$

解:

```
int f(int n)
{
    int i, s;
    s1 = 1;    /* s1 用于保存 f(n-1) 的值 */
    s2 = 1;    /* s2 用于保存 f(n-2) 的值 */
    s = 1;
    for (i = 3; i <= n; i++)
    {
        s = s1 + s2;
        s2 = s1;
        s1 = s;
    }
    return(s);
}
```

2. 间接转换法

该方法使用栈保存中间结果。其一般的过程如下:

```
将初始状态 s0 进栈
while (栈不为空)
{
    退栈, 将栈顶元素赋给 s
    if (s 是要找的结果) 返回
    else
    {
        寻找到 s 的相关状态 s1
        将 s1 进栈
    }
}
```

例 2. 编写一个过程采用非递归方法计算一棵二叉树的所有结点个数。

解: 先定义如下常量和变量类型:

```
#define n 100
typedef struct node
{
    char data;
    struct node * left, * right;
} bitree;
```

计算一棵二叉树的所有结点个数的 counter 函数如下:

```
int counter(bitree * t)
{
    bitree * st[n], * p;
    int top, count=0;
    if (t!= NULL)
    {
        top=1;
        stack[top]=t;          /* 将根结点入栈 */
        while (top>0)
        {
            count++;          /* 结点计数器增1 */
            node=stack[top];   /* 将栈顶结点退栈并赋给 node */
            top--;
            if (node->left!= NULL)
            {
                top++;
                stack[top]=node->left;
            }
            if (node->right!= NULL)
            {
                top++;
                stack[top]=node->right;
            }
        }
    }
    return(count);
}
```

6.2 基本题

6.2.1 单项选择题

1. 递归函数 $f(n)=f(n-1)+n(n>1)$ 的递归出口是 ①。
 A. $f(1)=0$ B. $f(1)=1$ C. $f(0)=1$ D. $f(n)=n$
 答: ①B

2. 递归函数 $f(n)=f(n-1)+n(n>1)$ 的递归体是 ①。

A. $f(1)=0$ B. $f(0)=1$ C. $f(n)=f(n-1)+n$ D. $f(n)=n$

答:①C

3. 将递归算法转换成对应的非递归算法时,通常需要使用 ①。

A. 栈 B. 队列 C. 链表 D. 树

答:①A

6.2.2 填空题(将正确的答案填在相应的空中)

1. 将 $f=1+1/2+1/3+\dots+1/n$ 转化成递归函数,其递归出口是①,递归体是②

答:① $f(1)=1$ ② $f(n)=f(n-1)+1/n$

2. 有如下函数说明:

```
FUNCTION f(x,y:integer):integer;
{
    f=x MOD y + 1
}
```

已知 $a=10, b=4, c=5$ 则执行 $k=f(a,b)+f(a,c)$ 后, k 的值是①, 执行 $k=f(a+b,c)-f(a,b+c)$ 后, k 的值是②, 执行 $k=f(f(a+c,b),f(b,c))$ 后, k 的值是③。

答:①4 ②3 ③5

3. 下列程序利用递归方法判别由链表表示的两个非递归列表是否相等,其中的非递归列表定义为:一个列表或者是没有元素的空列表,或者是由元素序列组成的一个列表,其中的元素可以是一个字符或者满足本定义的另一列表。这种列表的例子如图 6.1 所示。列表 s 由两个元素组成,第一个元素是字符 a (标志是 f),第二个元素是另一个列表(标志是 t)。它由两个元素即(标志都为 f)字符 b 和字符 c 组成。两个非递归列表相等是指它们的元素个数相等,且表中元素依次相同。程序如下:

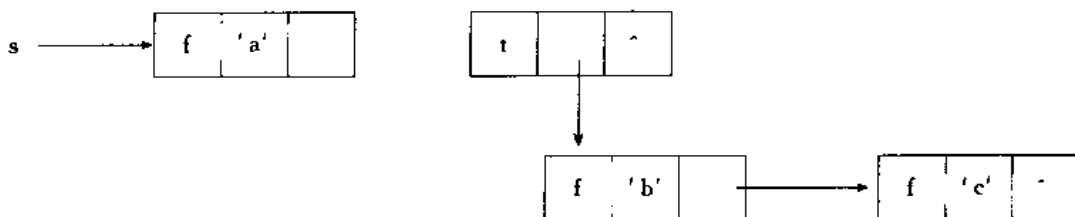


图 6.1 一个列表

```
struct listpointer
{
    struct link * listpointer;
    int tag;
    union {
        char data;
        struct listpointer * dlink;
    }
}
```

```

        } val;
    };
    int equal(s,t)
    struct listpointer * s,* t;
    {
        int same;
        if (s==NULL && t==NULL) ①;
        else if (②)
        {
            if (③)
            {
                if (s->tag==0) same= (④);
                else same= ⑤;
                if (same) ⑥;
            }
        }
        return(same);
    }

```

答: ①return(1)

②(s!=NULL && t!=NULL)

③s->tag==t->tag

④s->val.data==t->val.data

⑤equal(s->val.dlink,t->val.dlink)

⑥same==equal(s->link,t->link)

4. 有如下递归过程:

```

void print(int w)
{
    int i;
    if (w!=0)
    {
        print(w-1);
        for (i=1;i<=w;i++) printf("%3d",w);
        printf("\n");
    }
}

```

调用语句 print(4)的结果是 ①。

答: ① 1

2 2

3 3 3

4 4 4 4

5. 有如下递归过程:

```
void reverse(int m)
{
    printf("%d", m%10);
    if (m/10 != 0) reverse(m/10);
}
```

调用语句 reverse(582)的结果是 ①。

答:①285

6. 有如下递归函数:

```
int dunno(int m)
{
    int value;
    if (m==0) value=3;
    else value=dunno(m-1)+5;
    return(value);
}
```

执行语句 printf("%d\n", dunno(3));的结果是 ①。

答:①18

* 7. 利用整型变量的后继函数 succ 写出 $a+b$ (a 和 b 均为非负整数)的递归定义是 ①。

答:① $f(a, 0) = a$;

$f(0, b) = b$;

$f(a, b) = f(\text{succ}(a), \text{succ}(b)) + 2$

* 8. 设 a 是含有 n 个分量的整数数组, 写出一个求 n 个整数的平均值的递归定义 ①。

答:① $f(n) = ((n-1)/n) * f(n-1) + a[n]/n$, 其中 $f(n) = (a[1] + a[2] + \dots + a[n])/n$

9. 设 a 是含有 n 个分量的整数数组, 写出求该数组中最大整数的递归定义 ①, 写出求该数组中最小整数的递归定义 ②。

答:① $f(n) = \max(f(n-1), a[n])$ 其中 $f(n)$ 为 $a[1]$ 到 $a[n]$ 中的最大整数, $\max(a, b)$ 为 a, b 的最大值

② $f(n) = \min(f(n-1), a[n])$ 其中 $f(n)$ 为 $a[1]$ 到 $a[n]$ 中的最小整数, $\min(a, b)$ 为 a, b 的最小值

10. 设 a 是含有 n 个分量的整数数组, 写出求 n 个整数之和的递归定义 ①, 写出求 n 个整数之积的递归定义 ②。

答:① $f(n) = f(n-1) + a[n]$ 其中 $f(n) = a[1] + a[2] + \dots + a[n]$

② $f(n) = f(n-1) * a[n]$ 其中 $f(n) = a[1] * a[2] * \dots * a[n]$

6.3 习题解析

* 1. 编写一个递归过程, 它读入一串任意长的字符串, 该串字符以 "." 作为结束, 要求打

印出它们的倒序字符串。

解：依题意，首先获取用户按键，如果不是‘.’字符，则递归调用该过程，否则显示该字符。实现本题功能的过程如下：

```
void reverse()
{
    scanf("%c",&ch);
    if (ch != '.')
    {
        reverse();
        printf("%c",ch);
    }
}
```

2. McCarthy 函数定义如下：

$$M(x) = \begin{cases} x-10 & x > 100 \\ M(M(x+11)) & x \leq 100 \end{cases}$$

- (1) 编写一个递归函数计算给定 x 的 $M(x)$ 值；
- (2) 编写一个非递归函数计算给定 x 的 $M(x)$ 值。

解：

- (1) 本函数是一个递归函数，其递归出口是：

$$M(x) = x - 10 \quad x > 100$$

递归体是：

$$M(x) = M(M(x+11)) \quad x \leq 100$$

实现本题功能的递归函数如下：

```
int m(int x)
{
    int y;
    if (x > 100) return(x-10);
    else
    {
        y = m(x+11);
        return(m(y));
    }
}
```

(2) 非递归函数采用直接方法，用 level 记录调用的层数，当 level 为 0 时，结果便计算出来了，实现本题功能的非递归函数如下：

```
int m1(int x)
{
    int level = 1, y;
```

```

    if (x>100) return(x-10);
    else
    {
        level++;
        y=x+11;
    }
    while (level>0)
    {
        if (y>100)          /* 参数 y 小于 100 时, 层数减 1, y=y-12 */
        {
            level--; y=y-10;
        }
        else                /* 否则层数 level 增 1, 参数 y=y+11 */
        {
            level++; y=y+11;
        }
    }
    return(y);              /* 层数 level 为 0 时, y 即为最终计算的函数值 */
}

```

* 3. 已知有一个多项式 $F_n(X)$, 可递归定义如下:

$$F_n(X) = \begin{cases} 1 & \text{当 } n=0 \text{ 时} \\ 2X & \text{当 } n=1 \text{ 时} \\ 2XF_{n-1}(X) - 2(n-1)F_{n-2}(X) & \text{当 } n>1 \text{ 时} \end{cases}$$

试写出计算 $F_n(X)$ 值的递归函数的过程。

解: 依题中的递归函数 $f(n, x)$ 的定义, 直接得到如下计算其值的过程:

```

float f(int n, float x)
{
    float s;
    if (n==0) return(1);
    else
    {
        if (n==1) return(2 * x);
        else
        {
            s = 2 * x * f(n-1, x) - 2 * (n-1) * f(n-2, x);
            return(s);
        }
    }
}

```

4. 编写一个程序计算如下函数 $P_n(x)$ 的值:

$$P_n(x) = \begin{cases} 1 & n=0 \\ x & n=1 \\ ((2n-1) * x * P_{n-1}(x) - (n-1) * P_{n-2}(x)) / n & n>1 \end{cases}$$

解:本函数是一个递归函数,其递归出口是:

$$\begin{cases} P_n(x)=1 & n=0 \\ P_n(x)=x & n=1 \end{cases}$$

递归体是

$$P_n(x) = ((2n-1) * x * P_{n-1}(x) - (n-1) * P_{n-2}(x)) / n \quad n>1$$

实现本题功能的程序如下:

```
float f(int n, float x)
{
    float s;
    if (n==0) return(1);
    else if (n==1) return(x);
    else
    {
        s=((2*n-1)*x*f(n-1,x)-(n-1)*f(n-2,x))/n;
        return(s);
    }
}

main()
{
    int n;
    float x;
    printf("n,x:");
    scanf("%d,%f",&n,&x);
    printf("p=%g\n",f(n,x));
}
```

* 5. 编写一个过程判定两棵二叉树是否相似,所谓两棵二叉树 s 和 t 相似,要么它们都为空或都只有一个根结点,要么它们的左右子树均相似。

解:依题意,得到如下判定两棵二叉树 s 与 t 是否相似的递归函数 like:

$$\text{like}(s,t) = \begin{cases} 1 & \text{若 } s=t=\text{NULL} \\ 0 & \text{若 } s,t \text{ 有一个为 NULL, 另一不为 NULL} \\ \text{like}(s->\text{left}, t->\text{left}) \text{ and } \text{like}(s->\text{right}, t->\text{right}) & \text{其他情况} \end{cases}$$

由此,实现本题功能的过程如下:

```
int like(bintree *s, bintree *t)
{
    int same;
    if (s==NULL && t==NULL) return(1); /* 都为空的情况 */
}
```

```

else if ((s == NULL && t != NULL) || (s != NULL && t == NULL)) return(0);
/* s, t 有一个为 NULL, 另一个不为 NULL */

else
{
    same = like(s->left, t->left);
    if (!same) same = like(s->right, t->right);
    return(same);
}
}

```

6. 编写一个计算一棵二叉树 t 的高度的过程。

解: 所谓一棵二叉树 t 的高度, 指的是根结点的高, 即该树中所有结点的最大的层号, 其递归定义为: 若一棵二叉树为空, 则其高度为 0, 否则其高度等于左(或右)子树的最大高度加 1, 则有:

$$\text{height}(t) = \begin{cases} 0 & \text{若 } t = \text{NULL} \\ \max(\text{height}(t \rightarrow \text{left}), \text{height}(t \rightarrow \text{right})) + 1 & \text{若 } t \neq \text{NULL} \end{cases}$$

由此, 实现本题功能的过程如下:

```

int height(bintree *t);
{
    int he, he1, he2;
    if (s == NULL) return(0);
    else
    {
        he1 = height(t->left);
        he2 = height(t->right);
        if (he1 > he2) he = he1 + 1;
        else he = he2 + 1;
        return(he);
    }
}

```

* 7. 编写一个递归算法, 求出二叉树中所有叶结点的最大和最小枝长。二叉树用标准表示法。

解: 最大枝长的定义为: 对于只有一个结点的二叉树, 其最大枝长为 0; 若其左子树为空, 则最大枝长为右子树的最大枝长加 1; 若其右子树为空, 则最大枝长为左子树的最大枝长加 1; 否则为其左子树最大枝长和右子树最大枝长的最大者加 1。由此得到如下递归定义:

$$\text{maxlength}(t) = \begin{cases} 0 & \text{当 } t \rightarrow \text{left} = \text{NULL} \text{ 且 } t \rightarrow \text{right} = \text{NULL} \text{ 时} \\ \text{maxlength}(t \rightarrow \text{left}) + 1 & \text{当 } t \rightarrow \text{right} = \text{NULL} \text{ 时} \\ \text{maxlength}(t \rightarrow \text{right}) + 1 & \text{当 } t \rightarrow \text{left} = \text{NULL} \text{ 时} \\ \max(\text{maxlength}(t \rightarrow \text{left}), \text{maxlength}(t \rightarrow \text{right})) + 1 & \text{当 } t \rightarrow \text{right} \text{ 和 } t \rightarrow \text{left} \text{ 均不为空时} \end{cases}$$

实现本题功能的函数如下:

```

int maxlength(bitree *t)
{
    int max,max1,max2;
    if (t->left==NULL && t->right==NULL) return(0);
    else if (t->left==NULL) return(maxlength(t->right)+1);
        else if (t->right==NULL) return(maxlength(t->left)+1);
        else
        {
            max1=maxlength(t->left);
            max2=maxlength(t->right);
            if (max1>max2) then max=max1+1;
            else max=max2+1;
            return(max);
        }
}

```

最小枝长的定义为:对于只有一个结点的二叉树,其最小枝长为0;若其左子树为空,则最小枝长为右子树的最小枝长加1;若其右子树为空,则最小枝长为左子树的最小枝长加1;否则为其左子树最小枝长和右子树最小枝长的最小者加1。由此得到如下递归定义:

$$\text{minlength}(t) = \begin{cases} 0 & \text{当 } t \rightarrow \text{left} = \text{NULL} \text{ 且 } t \rightarrow \text{right} = \text{NULL} \text{ 时} \\ \text{minlength}(t \rightarrow \text{left}) + 1 & \text{当 } t \rightarrow \text{right} = \text{NULL} \text{ 时} \\ \text{minlength}(t \rightarrow \text{right}) + 1 & \text{当 } t \rightarrow \text{left} = \text{NULL} \text{ 时} \\ \min(\text{minlength}(t \rightarrow \text{left}), \text{minlength}(t \rightarrow \text{right})) + 1 & \text{当 } t \rightarrow \text{right} \text{ 和 } t \rightarrow \text{left} \text{ 均不为空时} \end{cases}$$

实现本题功能的函数如下:

```

int minlength(bitree *t)
{
    int min,min1,min2;
    if (t->left==NULL && t->right==NULL) return(0);
    else if (t->left==NULL) return(minlength(t->right)+1);
        else if (t->right==NULL) return(minlength(t->left)+1);
        else
        {
            min1=minlength(t->left);
            min2=minlength(t->right);
            if (min1>min2) then min=min2+1;
            else min=min1+1;
            return(min);
        }
}

```

* 8. 编写对二叉树实施前序遍历的非递归算法,并对算法执行于如图 6.2 所示的二叉树时的情况进行跟踪(即给出各阶段栈的内容及输出的结点序列)。

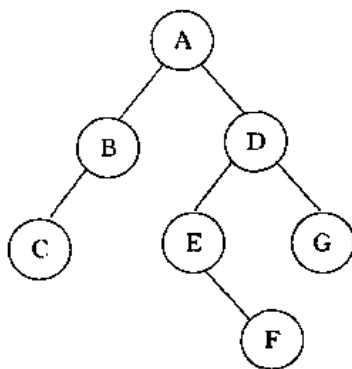


图6.2 一棵二叉树

解：二叉树前序遍历的递归算法如下：

```

void preorder(bitree *t)
{
    if (t!=NULL)
    {
        printf("%c",t->data);
        preorder(t->left);
        preorder(t->right)
    }
}
  
```

将其转换成非递归算法如下：

```

void preorder(bitree *t)
{
    bitree stack[100],node;
    int top=1;
    stack[top]=t; /* 将根结点 t 入栈 */
    while (top>0)
    {
        node=stack[top]; /* 出栈 */
        top--;
        printf("%c",node->data);
        if (node->right!=NULL) /* 若右子树不为空,则将其根结点入栈 */
        {
            top++;
            stack[top]=node->right;
        }
        if (node->left!=NULL) /* 若左子树不为空,则将其根结点入栈 */
        {
            top++;
            stack[top]=node->left;
        }
    }
}
  
```

跟踪如图 6.2 所示的二叉树时的情况如图 6.3 所示。

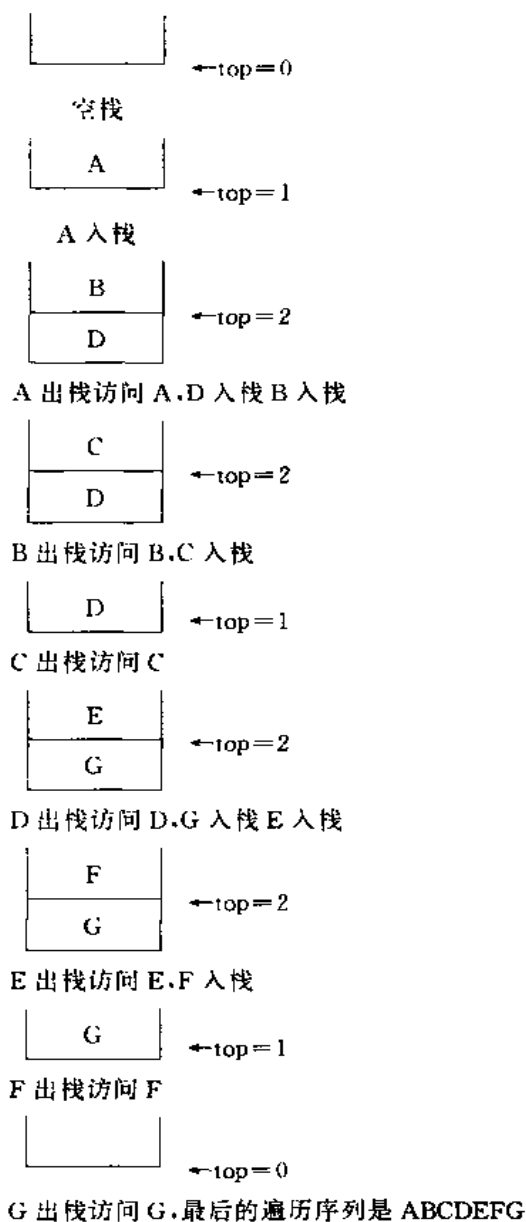


图 6.3 跟踪图 6.2 的二叉树时栈的变化情况

9. 求两个正整数 m 和 n 的最大公约数可以用如下 $\text{gcd}(m, n)$ 公式表示:

$$\text{gcd}(m, n) = \begin{cases} \text{gcd}(n, m) & \text{当 } m < n \text{ 时} \\ m & \text{当 } n = 0 \text{ 时} \\ \text{gcd}(n, m \% n) & \text{其他} \end{cases}$$

(1) 编写一个计算 $\text{gcd}(m, n)$ 的递归过程;

(2) 将上述过程转换成非递归过程;

(3)画出计算 $\text{gcd}(20,6)$ 的过程及栈的状态变化,给出计算结果。

解

(1)依题意,得到实现本题功能的递归函数如下:

```
int gcd(int m,int n)
{
    int k;
    if (n>m) return(gcd(n,m));
    else if (n==0) return(m);
    else
    {
        k=m % n;
        return(gcd(n,k));
    }
}
```

(2)将上述递归函数转换成非递归函数时,使用了一个栈,用于存储调用参数和函数值,由于本题很特殊,所有情况的函数值都相等,故不需要保存函数值,且当条件满足即 $n=0$ 时便退出,这里采用一个二维数组作为栈,其内容如下:

stack[top][1]存储 m 之值

stack[top][2]存储 n 之值

由此,实现本题功能的非递归函数如下:

```
int gcd(int m,int n)
{
    int k,top=1;
    stack[top][1]=m;
    stack[top][2]=n;
    while (stack[top][2] != 0)          /* 循环直到 n=0 为止 */
    {
                                                /* 若 m<n,则(n,m)入栈 */
        if (stack[top][1]<stack[top][2])
        {
            k=stack[top][1];
            top++;
            stack[top][1]=stack[top-1][2];
            stack[top][2]=k;
        }
        else                                /* 否则,(n,m % n)入栈 */
        {
            k=stack[top][1] % stack[top][2];
            top++;
            stack[top][1]=stack[top-1][1];
```

```

        stack[top][2]=k;
    }
    }
    return(stack[top][1]);
}

```

本题亦可采用直接转换法得到如下非递归函数 gcd2():

```

int gcd2(int m,int n)
{
    int r;
    do
    {
        r=m % n;
        m=n;
        n=r;
    } while (r != 0);
    return(m);
}

```

(3) 依照(2)的非递归函数 gcd() 的执行过程, 计算 gcd(20,6) 之值栈变化过程, 如图 6.4 所示。

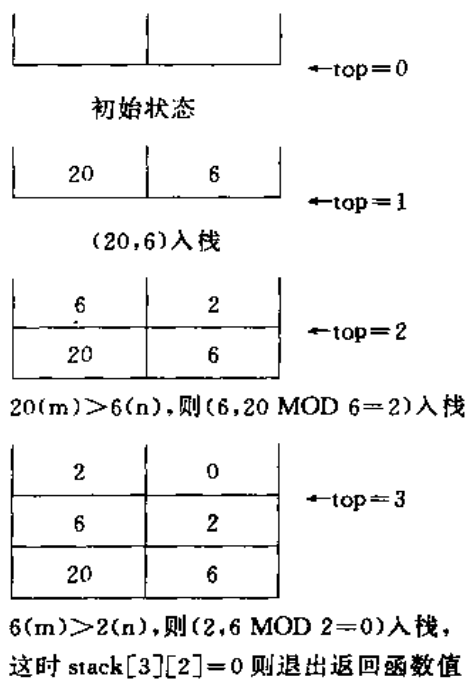


图 6.4 计算 gcd(20,6) 的过程及栈的状态变化

* 10. 已知递归函数(其中 DIV 为整除):

$$F(n) = \begin{cases} 1 & \text{当 } n=0 \text{ 时} \\ n * F(n \text{ DIV } 2) & \text{当 } n>0 \text{ 时} \end{cases}$$

- (1) 写出求 $F(n)$ 递归算法;
- (2) 写出求 $F(n)$ 的非递归算法;
- (3) 画出计算 $F(17)$ 的过程及栈的状态变化, 给出计算结果。

解:

- (1) 依题意, 得到实现本题功能的递归函数如下:

```
int findval1(int n)
{
    int f;
    if (n == 0) return(1);
    else
    {
        f = findval1(n/2);
        return(n * f);
    }
}
```

(2) 将上述递归函数转换成非递归函数时, 使用了一个栈, 用于存储调用参数和函数值, 这里采用一个二维数组作为栈, 其内容如下:

stack[top][0] 存储 $F(n)$ 之值
 stack[top][1] 存储 n 之值
 stack[top][2] 存储 $F(n/2)$ 之值

因此有如下关系:

$$\text{stack}[\text{top}][0] = \text{stack}[\text{top}][1] * \text{stack}[\text{top}][2]$$

stack[top][*] 与 stack[top-1][*] 有如下关系:

$$\text{stack}[\text{top}][2] = \text{stack}[\text{top}-1][0]$$

由此, 实现本题功能的非递归函数如下:

```
#define maxlen 200
int findval(int n)
{
    int stack[maxlen][2];
    int top = 1, fval;
    stack[top][1] = n;    /* 初值进栈 */
    while (n != 0)        /* n 不为 0 进栈 */
    {
        top++;
        n = n/2;          /* DIV 为整除 */
        stack[top][1] = n;
    }
    stack[top][0] = 1;    /* 给栈顶的 stack[top][0] 赋初值 */
```

```

while (top>1)    /* 开始退栈求值 */
{
    fval=stack[top][0]; /* 保存栈顶的 stack[top][0]之值 */
    top--;           /* 退栈 */
    stack[top][2]=fval;
    stack[top][0]=stack[top][1]*stack[top][2];
}
return(stack[top][0]);
}

```

(3) 依照(2)的非递归函数的执行过程, 计算 $F(17)$ 之值栈变化过程如图 6.5 所示。

			←top=0
--	--	--	--------

初始状态

	17		←top=1
--	----	--	--------

17 进栈

	8		←top=2
	17		

8 进栈

	4		←top=3
	8		
	17		

4 进栈

	2		←top=4
	4		
	8		
	17		

2 进栈

	1		←top=5
	2		
	4		
	8		
	17		

1 进栈

	0		$\leftarrow \text{top} = 6$
	1		
	2		
	4		
	8		
	17		

0 进栈

1	0		$\leftarrow \text{top} = 6$
	1		
	2		
	4		
	8		
	17		

 $f(0) = 1$, 因此 $\text{stack}[\text{top}][0] = 1$

1	1	1	$\leftarrow \text{top} = 5$
	2		
	4		
	8		
	17		

退栈, 计算 $\text{stack}[\text{top}][0] = 1 * 1 = 1$

2	2	1	$\leftarrow \text{top} = 4$
	4		
	8		
	17		

退栈, 计算 $\text{stack}[\text{top}][0] = 2 * 1 = 2$

8	4	2	$\leftarrow \text{top} = 3$
	8		
	17		

退栈, 计算 $\text{stack}[\text{top}][0] = 4 * 2 = 8$

64	8	8	$\leftarrow \text{top} = 2$
	17		

退栈, 计算 $\text{stack}[\text{top}][0] = 8 * 8 = 64$

1088	17	64
------	----	----

$\leftarrow \text{top} = 1$

退栈, 计算 $\text{stack}[\text{top}][1] = 17 * 64 = 1088$

返回 1088 的函数值

图 6.5 计算 $F(17)$ 之值栈变化过程

11. 有如下递归计算公式:

$$\begin{cases} C(n, 0) = 1 & n \geq 0 \\ C(n, n) = 1 & n \geq 0 \\ C(n, m) = C(n-1, m) + C(n-1, m-1) & n > m \geq 0 \end{cases}$$

(1) 编写一个计算 $C(n, m)$ 的递归过程;

(2) 将上述过程转换成非递归过程;

(3) 画出计算 $C(5, 3)$ 的过程及栈的状态变化, 给出计算结果。

解: (1) 依题意, 得到实现本题功能的递归函数如下:

```
int comb1(int n, int m)
{
    if ((n >= 0 && m == 0) || (n >= 0 && m == n)) return(1);
    else
    {
        if (n > m && n >= 0 && m >= 0)
            return(comb1(n-1, m) + comb1(n-1, m-1));
        else
        {
            printf("n, m 值不正确\n");
            return(-1);
        }
    }
}
```

(2) 将上述递归函数转换成非递归函数时, 使用了一个栈, 用于存储调用参数和函数值, 这里采用一个二维数组作为栈, 其内容如下:

$\text{stack}(\text{top}, 1)$ 存储 $C(n, m)$ 之值

$\text{stack}(\text{top}, 2)$ 存储 n 之值

$\text{stack}(\text{top}, 3)$ 存储 m 之值

由此, 实现本题功能的非递归函数如下:

```
#include <stdio.h>
#define maxlen 200
int comb2(int n, int m)
{
    int stack[maxlen][3], top = 1, s1, s2;
```



```

if (n<m || n<0 || m<0)
{
printf("n,m 值不正确\n");
return(-1);
}
stack[top][1]=0;          /* 初值 0 进栈 */
stack[top][2]=n;          /* 初值 n 进栈 */
stack[top][3]=m;          /* 初值 m 进栈 */
do                          /* 循环 */
{
printf("top= %d  %d  %d  %d\n",top,stack[top][1],stack[top][2],stack[top][3]);
/* 如果栈顶为(0,n,m),表示要对其分解,将(0,n-1,m)入栈 */
if (stack[top][1]==0)
{
top++;
stack[top][1]=0;
stack[top][2]=stack[top-1][2]-1;
stack[top][3]=stack[top-1][3];
/* 如果栈顶位置 2 的值等于 0 或位置 2 之值等于位置 3 之值,则给位置 1 赋初值 1 */
if (stack[top][3]==0 || stack[top][2]==stack[top][3])
stack[top][1]=1;
}
/* 如果栈顶位置 1 的值大于 0,表示该值已计算好,若栈 top-1 处位置 1 的值为 */
/* (0,n,m),表示要求第二部分,因此,(0,n-1,m-1)入栈 */
if (top>=2 && stack[top][1]>0 && stack[top-1][1]==0)
{
top++;
stack[top][1]=0;
stack[top][2]=stack[top-2][2]-1;
stack[top][3]=stack[top-2][3]-1;
/* 如果栈顶位置 2 的值等于 0 或位置 2 之值等于位置 3 之值,则给位置 1 赋初值 1 */
if (stack[top][3]==0 || stack[top][2]==stack[top][3])
stack[top][1]=1;
}
/* 如最栈顶和次栈顶位置 1 的值都大于 0,则退栈两次,计算新栈顶位置 1 之值 */
if (top>2 && stack[top][1]>0 && stack[top-1][1]>0)
{
s1=stack[top][1];
s2=stack[top-1][1];
top=top-2;
stack[top][1]=s1+s2;
}
/* 当栈中再次只有一个元素时,则计算完成,退出循环 */
} while (top>1);
return(stack[1][1]);
}

```

执行如下主程序:

```

main()
{

```

```
int n=5,m=3;
printf("comb: %d\n",comb2(n,m));
```

其结果为:

```
top=1  0  5  3
top=2  0  4  3
top=4  0  3  2
top=6  0  2  1
top=6  2  2  1
top=4  3  3  2
top=2  4  4  3
top=3  0  4  2
top=4  0  3  2
top=6  0  2  1
top=6  2  2  1
top=4  3  3  2
top=5  0  3  1
top=6  0  2  1
top=6  2  2  1
top=5  3  3  1
top=3  6  4  2
comb:10
```

(3) 依照(2)的非递归函数的执行过程, 计算 $C(5,3)$ 之值栈变化过程, 如图 6.6 所示。

--	--	--

初始状态

0	5	3	←top=1
---	---	---	--------

(0,5,3)入栈

0	4	3	←top=2
0	5	3	

循环第1次: $stack(1,1)=0$, 则 $(0,5-1=4,3)$ 入栈

1	3	3	←top=3
0	4	3	
0	5	3	

循环第2次: $stack[2][1]=0$ 则 $(0,4-1=3,3)$ 入栈.

因 $stack[3][2]=stack[3][3]=3$, 则 $stack[3][1]=1$

0	3	2	←top=4
1	3	3	
0	4	3	
0	5	3	

循环第2次: $stack[3][1] > 0$ 且 $stack[3-1=2][1] = 0$,
 则 $(0, 4-1=3, 3-1=2)$ 入栈

1	2	2	←top=5
0	3	2	
1	3	3	
0	4	3	
0	5	3	

循环第3次: $stack[4][1] = 0$, 则 $(0, 3-1=2, 2)$ 入栈,
 因 $stack[5][2] = stack[5][3] = 2$, 则 $stack[5][1] = 1$

0	2	1	←top=6
1	2	2	
0	3	2	
1	3	3	
0	4	3	
0	5	3	

循环第3次: $stack[5][1] > 0$ 且 $stack[4][1] = 0$,
 则 $(0, 3-1=2, 2-1=1)$ 入栈

1	1	1	←top=7
0	2	1	
1	2	2	
0	3	2	
1	3	3	
0	4	3	
0	5	3	

循环第4次: $stack[6][1] = 0$, 则 $(0, 2-1=1, 1)$ 入栈,
 因 $stack[7][2] = stack[7][3] = 1$, 则 $stack[7][1] = 1$

1	1	0	$\leftarrow \text{top}=8$
1	1	1	
0	2	1	
1	2	2	
0	3	2	
1	3	3	
0	4	3	
0	5	3	

循环第4次: $\text{stack}[7][1] > 0$ 且 $\text{stack}[6][1] = 0$, 则 $(0, 2-1=1, 1-1=0)$ 入栈, 因 $\text{stack}[6][3] = 0$, 则 $\text{stack}[6][1] = 1$

2	2	1	$\leftarrow \text{top}=6$
1	2	2	
0	3	2	
1	3	3	
0	4	3	
0	5	3	

循环第4次: $\text{stack}[8][1] > 0$ 且 $\text{stack}[7][1] > 0$, 则退两次栈, $\text{stack}[6][1] = \text{stack}[8][1] + \text{stack}[7][1] = 2$

3	3	2	$\leftarrow \text{top}=4$
1	3	3	
0	4	3	
0	5	3	

循环第5次: $\text{stack}[6][1] > 0$ 且 $\text{stack}[5][1] > 0$, 则退两次栈, $\text{stack}[4][1] = \text{stack}[6][1] + \text{stack}[5][1] = 3$

4	4	3	$\leftarrow \text{top}=2$
0	5	3	

循环第6次: $\text{stack}[4][1] > 0$ 且 $\text{stack}[3][1] > 0$, 则退两次栈, $\text{stack}[2][1] = \text{stack}[4][1] + \text{stack}[3][1] = 4$

0	4	2	$\leftarrow \text{top}=3$
4	4	3	
0	5	3	

循环第7次: $\text{stack}[2][1] > 0$ 且 $\text{stack}[1][1] = 0$, 则 $(0, 5-1=4, 3-1=2)$ 入栈

0	3	2	$\leftarrow \text{top}=4$
0	4	2	
4	4	3	
0	5	3	

循环第 8 次: $\text{stack}[3][1]=0$, 则 $(0, 4-1=3, 2)$ 入栈

1	2	2	$\leftarrow \text{top}=5$
0	3	2	
0	4	2	
4	4	3	
0	5	3	

循环第 9 次: $\text{stack}[4][1]=0$, 则 $(0, 3-1=2, 2)$ 入栈,
因 $\text{stack}[5][2]=\text{stack}[5][3]=2$, 则 $\text{stack}[5][1]=1$

0	2	1	$\leftarrow \text{top}=6$
1	2	2	
0	3	2	
0	4	2	
4	4	3	
0	5	3	

循环第 9 次: $\text{stack}[5][1]>0$ 且 $\text{stack}[4][1]=0$,
则 $(0, 3-1=2, 2-1=1)$ 入栈

1	1	1	$\leftarrow \text{top}=7$
0	2	1	
1	2	2	
0	3	2	
0	4	2	
4	4	3	
0	5	3	

循环第 10 次: $\text{stack}[6][1]=0$, 则 $(0, 2-1=1, 1)$ 入栈,
因 $\text{stack}[7][2]=\text{stack}[7][3]=1$, 则 $\text{stack}[7][1]=1$

1	1	0	$\leftarrow \text{top} = 8$
1	1	1	
0	2	1	
1	2	2	
0	3	2	
0	1	2	
1	1	3	
0	5	3	

循环第 10 次: $\text{stack}[7][1] > 0$ 且 $\text{stack}[6][1] = 0$, 则 $(0, 2-1=1, 1-1=0)$ 入栈, 因 $\text{stack}[8][3] = 0$, 则 $\text{stack}[8][1] = 1$

2	2	1	$\leftarrow \text{top} = 6$
1	2	2	
0	3	2	
0	1	2	
1	1	3	
0	5	3	

循环第 10 次: $\text{stack}[8][1] > 0$ 且 $\text{stack}[7][1] > 0$, 则退两次栈,
 $\text{stack}[6][1] = \text{stack}[8][1] + \text{stack}[7][1] = 2$

3	3	2	$\leftarrow \text{top} = 4$
0	1	2	
1	1	3	
0	5	3	

循环第 11 次: $\text{stack}[6][1] > 0$ 且 $\text{stack}[5][1] > 0$, 则退两次栈
 $\text{stack}[4][1] = \text{stack}[6][1] + \text{stack}[5][1] = 3$

0	3	1	$\leftarrow \text{top} = 5$
3	3	2	
0	1	2	
1	1	3	
0	5	3	

循环第 12 次: $\text{stack}[4][1] > 0$ 且 $\text{stack}[3][1] = 0$, 则 $(0, 4-1=3, 2-1=1)$ 入栈

0	2	1	←top=6
0	3	1	
3	3	2	
0	4	2	
4	4	3	
0	5	3	

循环第 13 次: $stack[5][1]=0$, 则 $(0, 3-1, 1)$ 入栈

1	1	1	←top=7
0	2	1	
0	3	1	
3	3	2	
0	4	2	
4	4	3	
0	5	3	

循环第 14 次: $stack[6][1]=0$, 则 $(0, 2-1=1, 1)$ 入栈,

因 $stack[7][2]=stack[7][3]=1$, 则 $stack[7][1]=1$

1	1	0	←top=8
1	1	1	
0	2	1	
0	3	1	
3	3	2	
0	4	2	
4	4	3	
0	5	3	

循环第 14 次: $stack[7][1]>0$ 且 $stack[6][1]=0$, 则 $(0, 2-1=1, 1-1=0)$ 入栈,

因 $stack[7][3]=0$, 则 $stack[7][1]=1$

2	2	1	←top=6
0	3	1	
3	3	2	
0	4	2	
4	4	3	
0	5	3	

循环第 14 次: $stack[8][1]>0$ 且 $stack[8][1]>0$, 则退两次栈,

$$\text{stack}[6][1] = \text{stack}[8][1] + \text{stack}[7][1] = 2$$

1	2	0	$\leftarrow \text{top} = 7$
2	2	1	
0	3	1	
3	3	2	
0	4	2	
4	4	3	
0	5	3	

循环第 15 次: $\text{stack}[6][1] > 0$ 且 $\text{stack}[5][1] = 0$, 则 $(0, 3-1=2, 1-1=0)$ 入栈,
因 $\text{stack}[7][3] = 0$ 则 $\text{stack}[7][1] = 1$

3	3	1	$\leftarrow \text{top} = 5$
3	3	2	
0	4	2	
4	4	3	
0	5	3	

循环第 15 次: $\text{stack}[7][1] > 0$ 且 $\text{stack}[6][1] > 0$, 则退两次栈,
 $\text{stack}[5][1] = \text{stack}[7][1] + \text{stack}[6][1] = 3$

6	4	2	$\leftarrow \text{top} = 3$
4	4	3	
0	5	3	

循环第 16 次: $\text{stack}[5][1] > 0$ 且 $\text{stack}[4][1] > 0$, 则退两次栈,
 $\text{stack}[3][1] = \text{stack}[5][1] + \text{stack}[4][1] = 6$

10	5	3	$\leftarrow \text{top} = 1$
----	---	---	-----------------------------

循环第 17 次: $\text{stack}[3][1] > 0$ 且 $\text{stack}[2][1] > 0$, 则退两次栈,
 $\text{stack}[1][1] = \text{stack}[3][1] + \text{stack}[2][1] = 10$
stack 栈中再次只有一个元素, 则计算完成, 退出循环

图 6.6 计算 $C(5, 3)$ 之值栈变化过程

12. 设有 3 个分别命名为 X, Y 和 Z 的塔座, 在塔座 X 上插有 n 个直径各不相同, 从小到大依次编号为 $1, 2, \dots, n$ 的圆盘, 现要求将 X 塔座上的 n 个圆盘移到塔座 Z 上并仍按同样顺序叠放, 圆盘移动时必须遵守以下规则: 每次只能移动一个圆盘; 圆盘可以插在 X, Y 和 Z 中任一塔座; 任何时候都不能将一个较大的圆盘放在较小的圆盘上。要求用递归和非递归两种方法求解, 并在 $n=4$ 时比较两者的结果。

解: 递归求解的原理是, 设 $\text{hanoi}(n, x, y, z)$ 表示将 n 个圆盘从 x 通过 y 移动到 z 上, 则


```

{
    n1=stack[top].ns;
    a1=stack[top].x;
    b1=stack[top].y;
    c1=stack[top].z;
    stack[top].no=1;
    stack[top].ns=n1-1;
    stack[top].x=b1;
    stack[top].y=a1;
    stack[top].z=c1;
    top++;
    stack[top].no=0;
    stack[top].ns=n1;
    stack[top].x=a1;
    stack[top].y=c1;
    top++;
    stack[top].no=1;
    stack[top].ns=n1-1;
    stack[top].x=a1;
    stack[top].y=c1;
    stack[top].z=b1;
}
while (top>0 && (stack[top].no==0 || stack[top].ns==1))
{
    if (top>0 && stack[top].no==0) /* 将第 n 个圆盘从 x 移到 z 退栈 */
    {
        printf("\t 将第%d 个盘片从%c 移动到%c\n", stack[top].ns, stack[top].x, stack[
        [top].y);
        top--;
    }
    if (top>0 && stack[top].ns==1) /* hanoi(1,x,y,z)退栈 */
    {
        printf("\t 将第%d 个盘片从%c 移动到%c\n", stack[top].ns, stack[top].x, stack[top].
        z);
        top--;
    }
}
}
}
main()
{
    int n=4;
    printf("递归求解结果:\n");

```

```
hanol1(n, 'X', 'Y', 'Z');  
printf("非递归求解结果:\n");  
hanol2(n, 'X', 'Y', 'Z');  
}
```

本程序的执行结果如下:

递归求解结果:

将第 1 个盘片从 X 移动到 Y
将第 2 个盘片从 X 移动到 Z
将第 1 个盘片从 Y 移动到 Z
将第 3 个盘片从 X 移动到 Y
将第 1 个盘片从 Z 移动到 X
将第 2 个盘片从 Z 移动到 Y
将第 1 个盘片从 X 移动到 Y
将第 4 个盘片从 X 移动到 Z
将第 1 个盘片从 Y 移动到 Z
将第 2 个盘片从 Y 移动到 X
将第 1 个盘片从 Z 移动到 X
将第 3 个盘片从 Y 移动到 Z
将第 1 个盘片从 X 移动到 Y
将第 2 个盘片从 X 移动到 Z
将第 1 个盘片从 Y 移动到 Z

非递归求解结果:

将第 1 个盘片从 X 移动到 Y
将第 2 个盘片从 X 移动到 Z
将第 1 个盘片从 Y 移动到 Z
将第 3 个盘片从 X 移动到 Y
将第 1 个盘片从 Z 移动到 X
将第 2 个盘片从 Z 移动到 Y
将第 1 个盘片从 X 移动到 Y
将第 4 个盘片从 X 移动到 Z
将第 1 个盘片从 Y 移动到 Z
将第 2 个盘片从 Y 移动到 X
将第 1 个盘片从 Z 移动到 X
将第 3 个盘片从 Y 移动到 Z
将第 1 个盘片从 X 移动到 Y
将第 2 个盘片从 X 移动到 Z
将第 1 个盘片从 Y 移动到 Z

第7章 广 义 表

广义表是 $n(n \geq 0)$ 个数据元素 a_1, a_2, \dots, a_n 的有限序列。其中 $a_i (1 \leq i \leq n)$ 或是单个数据元素(或称为原子),或仍然是一个广义表。广义表通常记作 $GL = (a_1, a_2, \dots, a_n)$, 其中 GL 是广义表的名字, n 是其长度。如果 a_i 也是一个广义表, 则称它为 GL 的子表。 a_1 是表头, 其余元素组成的表称为表尾。广义表通常简称为表。

7.1 广义表的表示及其运算

7.1.1 广义表的表示

广义表通常用圆括号括起来, 用逗号分隔广义表中的元素, 本书中我们约定用小写字母表示原子, 用大写字母表示广义表, 例如:

```
A = ( )
B = (a, b)
C = (c)
D = (B, c) = ((a, b), c)
E = (B, C) = ((a, b), (c))
```

其中 A 是一个空广义表, B 和 C 是两个仅包含原子的广义表, 而 D 和 E 是两个包含子表的广义表。

广义表的长度指该广义表中所包含的元素(包括原子和子表)的个数。

广义表的深度指该广义表中所包含括号的层数。

广义表中的结点具有不同的结构, 即原子结点和子表元素结点, 为了将两者统一, 用了—个标志 tag, 当其为 0 时表示是原子结点, 其 data 域存储结点值, link 域指向下一个结点; 当其 tag 为 1 时表示是子表结点, 其 sublist 为指向子表的指针。因此, 广义表采用如下结构存储:

```
typedef struct linknode
{
    int tag; /* 在建立广义表时只能是 0 或 1 */
    struct linknode * link;
    union {
        char data;
        struct linknode * sublist;
    } val;
} gnode;
```

同链表的存储一样,广义表存储结构也有带表头结点和不带表头结点两种存储形式,为了便于运算,我们在这里均采用带表头结点的存储形式。

例如,上述广义表 D 的存储结构如图 7.1 所示。

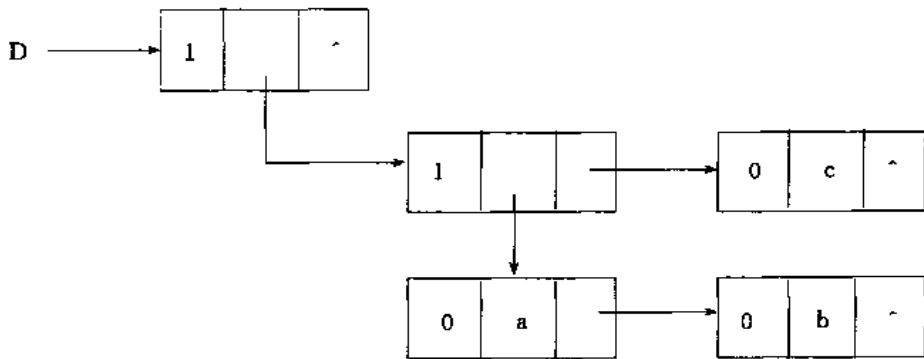


图 7.1 广义表 D 的存储结构

7.1.2 广义表的基本运算

1. 建立广义表算法

根据字符串 s 建立相应广义表。其递归定义如下：

基本项： $\begin{cases} \text{置空广义表} & \text{当 } s \text{ 为空表时} \\ \text{建原子结点的子表} & \text{当 } s \text{ 为单字符串时} \end{cases}$

归纳项：假设 subs 为脱去 s 中最外层括号对的子串，记为“ s_1, s_2, \dots, s_n ”，其中 $s_i (i=1, 2, \dots, n)$ 为非空字符串。对每个 s_i 建立一个表结点，并令其 val. sublist 域的指针为 s_i 建立的子表的头指针，除最后建立的尾指针为 NULL 外，其余表结点的尾指针均这样指向。

函数 disastr(char *s, char *hstr) 的功能是从字符串 s 中取出第 1 个 ‘,’ 之前的子串赋给 hstr，并使 s 成为删除子串 hstr 和 ‘,’ 之后的剩余串。若串 s 中没有字符 ‘,’，则操作后的 hstr 即为操作前的 s，而操作后的 s 为空串。

建立广义表算法的函数如下：

```
void disastr(char s[], char hstr[])
{
    int i=0, j=0, k=0, r=0;
    char rstr[100];
    while (s[i] && (s[i] != ',' || k))
    {
        if (s[i] == '(') k++;
        else if (s[i] == ')') k--;
        if (s[i] != ',' || s[i] == ',' && k)
        {
            hstr[j] = s[i];
            j++;
        }
    }
}
```

```

    j++;
}
}
hstr[j]='\0';
if (s[i]=='.' ) i++;
while (s[i])
{
    rstr[r]=s[i];
    r++;
    i++;
}
rstr[r]='\0';
strcpy(s,rstr);
}

gnode * gcreat(char s[])
{
    gnode * p, * q, * r, * gh;
    char subs[100],hstr[100],rstr[100];
    int len;
    len=strlen(s);
    if (! strcmp(s,"()")) gh=NULL;
    else
    if (len==1) /* 原子的情况 */
    {
        gh=(gnode *)malloc(sizeof(gnode)); /* 建立一个新结点 */
        gh->tag=0; /* 构造原子结点 */
        gh->val.data= * s;
        gh->link=NULL;
    }
    else /* 子表的情况 */
    {
        gh=(gnode *)malloc(sizeof(gnode)); /* 建立一个新结点 */
        gh->tag=1;
        p=gh;
        s++; /* 除掉前面的一个 '(' */
        strncpy(subs,s,len-2);
        subs[len-2]='\0';
        do
        {
            disastr(subs,hstr); /* 将 subs 分为表头和表尾 */
            r=gcreat(hstr);
            p->val.sublist=r;

```

```

    q=p;
    len=strlen(subs);
    if (len>0)
    {
        p=(gnode *)malloc(sizeof(gnode));
        p->tag=1;
        q->link=p;
    }
    } while (len>0);
    q->link=NULL;
}
return(gh);
}

```

2. 打印广义表算法

对给定存储结构的广义表,打印出其表示格式。

采用的算法思想是:先打印出一个'('号,若遇到 tag=0 的结点 p,是一个原子结点,如果有后续结点,则打印该结点值之后加一个','号,若没有后续结点,则只打印该结点值。如果 tag=1,为子表的情况,则递归调用本函数打印其子表,子表打印完后,最后打印一个')'号。

实现上述算法的函数如下:

```

void prtlist(gnode *h)
{
    gnode *p, *q;
    printf("(");
    if (h)
    do
    {
        p=h->val.sublist;
        q=h->link;
        while (q && p && ! p->tag) /* 为原子结点且有后续结点的情况 */
        {
            printf("%c",p->val.data);
            p=q->val.sublist;
            q=q->link;
        }
        if (p && ! p->tag) /* 为原子结点且无后续结点的情况 */
        {
            printf("%c",p->val.data);
            break;
        }
        else /* 为子表的情况 */
    }
}

```

```

    if (!p) printf("( )");
    else prtlist(p); /* 为子表的情况 */
    if (q) printf(",");
    h=q;
}
} while (h);
printf("\n");
}

```

3. 查找算法

在给定的广义表中查找数据域为 x 的结点。

算法思想是：若遇到 $\text{tag}=0$ 的原子结点 p ，如果是要找的结点 ($p \rightarrow \text{val.data}=x$)，则查找成功；若遇到 $\text{tag}=1$ 的结点 p ，则递归调用本函数在该子表中查找；若未找到数据域为 x 的结点且还有后续元素，则递归调用本函数查找后续每个元素，直至遇到 link 域为 NULL 的元素。

设 $f(p, x)$ 为查找函数，当成功时为 true ，否则为 false ，则有如下递归模型：

$$\begin{cases} f(p, x) = \text{true} & \text{若 } p \rightarrow \text{tag} = 0 \text{ 且 } p \rightarrow \text{val.data} = x \\ f(p, x) = f(p \rightarrow \text{link}, x) & \text{若 } p \rightarrow \text{tag} = 0 \text{ 且 } p \rightarrow \text{val.data} \neq x \\ f(p, x) = f(p \rightarrow \text{val.sublist}, x) \text{ or } f(p \rightarrow \text{link}, x) & \text{若 } p \rightarrow \text{tag} = 1 \end{cases}$$

由此，实现上述算法的函数如下：

```

int locate(gnode *p, char x)
{
    int find=0;
    if (p != NULL)
    {
        if (!p->tag && p->val.data == x) /* 当前结点为原子结点且其 data 为 x */
            return(1);
        else if (p->tag) find=locate(p->val.sublist, x); /* 查找子表 */
        if (find) return(1);
        else return(locate(p->link, x)); /* 查找后续结点 */
    }
    else
        return(0);
}

```

7.2 基本题

7.2.1 单项选择题

1. 广义表 $((a), a)$ 的表头是①，表尾是②。
A. a B. b C. (a) D. $((a))$

答①C ②C

2. 广义表((a))的表头是①,表尾是②。

A. a B. (a) C. () D. ((a))

答①B ②C

3. 广义表((a,b),c,d)的表头是①,表尾是②。

A. a B. b C. (a,b) D. (c,d)

答①C ②D

4. 广义表(a,b,c,d)的表头是①,表尾是②。

A. a B. b C. (a,b) D. (b,c,d)

答①A ②D

5. 广义表((a,b,c,d))的表头是①,表尾是②。

A. a B. () C. (a,b,c,d) D. ((a,b,c,d))

答①C ②B

6. 一个广义表的表头总是一个广义表,这个断言是 ①。

A. 正确的 B. 错误的

答①B

7. 一个广义表的表尾总是一个广义表,这个断言是 ①。

A. 正确的 B. 错误的

答①A

7.2.2 填空题(将正确的答案填在相应的空中)

1. 广义表(((a)))的表头是 ①,表尾是 ②。

答:①((a)) ②()

2. 广义表((a),((b),c),(((d))))的表头是 ①,表尾是 ②。

答:①(a) ②(((b),c),(((d))))

3. 广义表((a),((b),c),(((d))))的长度是 ①,深度是 ②。

答:①3 ②4

4. 广义表(a,(a,b),d,e,((i,j),k))的长度是 ①,深度是 ②。

答:①5 ②3

5. 设 HEAD[p]为求广义表 p 的表头函数, TAIL[p]为求广义表 p 的表尾函数,其中[]是函数的符号,给出下列广义表的运算结果:

HEAD[(a,b,c)]的结果是 ①。

TAIL[(a,b,c)]的结果是 ②。

HEAD[((a),(b))的结果是 ③。

TAIL[((a),(b))的结果是 ④。

HEAD[TAIL[(a,b,c)]]的结果是 ⑤。

TAIL[HEAD((a,b),(c,d))]]的结果是 ⑥。

HEAD[HEAD[(a,b),(c,d)]]的结果是 ⑦。

TAIL[TAIL[(a,(c,d))]]的结果是 ⑧。

答: ①a ②(b,c) ③(a) ④((b)) ⑤b ⑥(b) ⑦a ⑧(-)

7.3 习题解析

* 1. 假设用下述两种结点的链表作广义表的存储结构:

表结点:

tag=1	dlink	rlink
-------	-------	-------

元素结点:

tag=0	data
-------	------

(1) 画出下列广义表的存储结构: (((a,b)), ((-), (d)), (e,f));

(2) 给出如图 7.2 所示的存储结构对应的广义表。

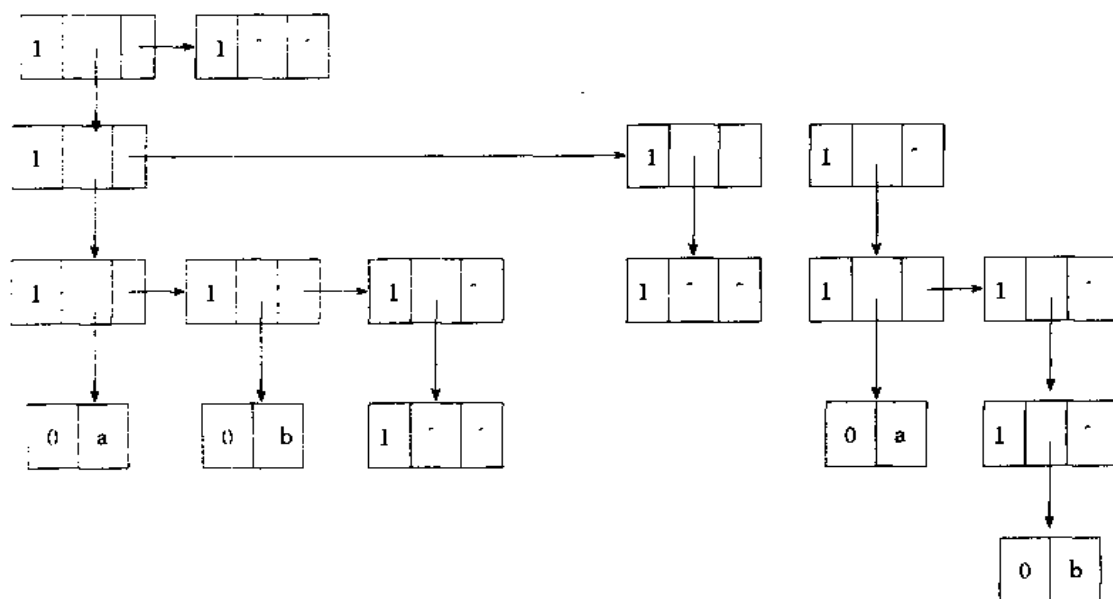


图 7.2 广义表对应的存储结构

解: (1) 依题意, 本小题的广义表对应的存储结构如图 7.3 所示。

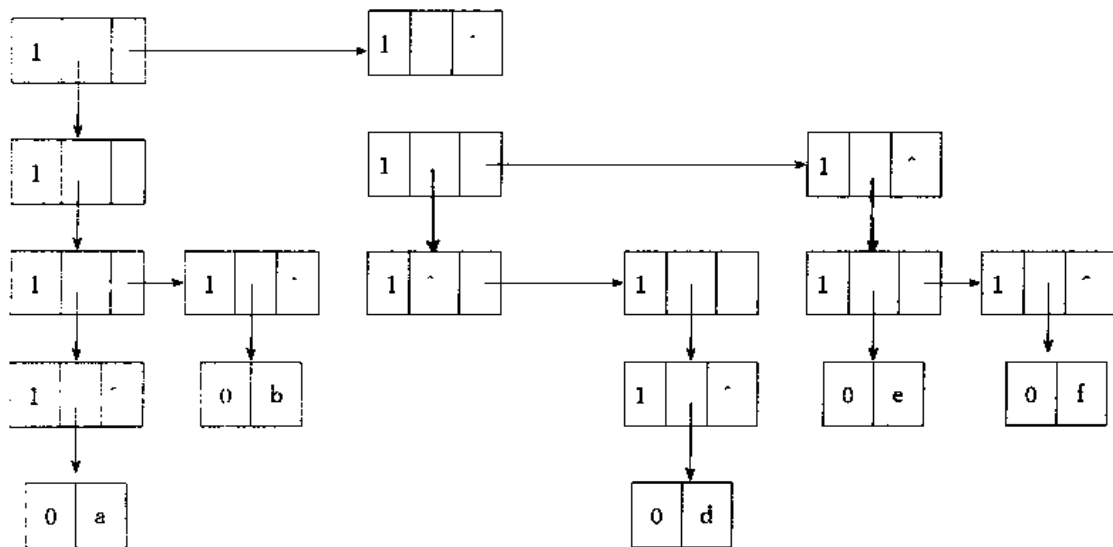


图 7.3 广义表对应的存储结构

(2)图对应的广义表为:(((a,b,(c,d)),(e)),(f))。

2. 根据表头和表尾的定义,设计分别实现求表头和表尾操作的 head()和 tail()函数,并编写一个求广义表表头和表尾的程序。

解:一个广义表的表头指的是该广义表的第一个元素。由此,实现上述算法的函数如下:

```
gnode * head(gnode * p)
{
    if (! p) return(NULL); /* 空表无表头 */
    else if (! p->tag) return(NULL); /* 原子无表头 */
    else return(p->val.sublist);
}
```

一个广义表的表尾指的是除去该广义表的第一个元素后的所有剩余部分。由此,实现上述算法的函数如下:

```
gnode * tail(gnode * p)
{
    if (p == NULL) return(NULL); /* 空表无表尾 */
    else if (p->link == NULL) return(NULL); /* 无表尾 */
    else return(p->link);
}
```

由此得到一个求广义表表头和表尾的程序如下:

```
main()
{
    char * s;
    gnode * glist, * h, * t;
    strcpy(s, "(a.(b.c.d))");
    glist=gcreat(s);
```

```

printf("\n 广义表:");
prtlist(glist);
h=head(glist);
printf("\n 表头:"); /* 为原子的情况,要进行特殊处理 */
if (h->tag==0) printf("%c",h->val.data);
else prtlist(h);
t=tail(glist);
printf("\n 表尾:");
prtlist(t);
}

```

本程序的执行结果如下:

```

广义表:(a.(b.c.d))
表头:a
表尾:((b.c.d))

```

3. 编写一个函数计算一个广义表的长度。例如一个广义表为(a,(b,c),((e))),其长度为3。

解:依题意,本题可用直接求解和递归求解两种方法。直接求解的算法思想:扫描通过广义表的第一层的每个结点,直至遇到link域为NULL的元素,每扫描一个结点,长度累加器增1。采用递归求解的递归模型如下:

$$\begin{cases} f(p)=0 & \text{若 } p=\text{NULL 或空表} \\ f(p)=f(\text{tail}(p))+1 & \text{若 } p \neq \text{NULL} \end{cases}$$

例如,计算广义表(a,(b,c),((e)))的长度的递归函数如下:

```

f[(a,(b,c),((e)))]→求值为 3
      ↓
f[((b,c),((e)))]→求值为 2
      ↓
f(((e)))→求值为 1
      ↓
f[(.)]
      ↓
      返回 0

```

因此,实现本题功能的直接求解函数如下:

```

int length(gnode *p)
{
    int n=0;
    if (p!=NULL && p->tag==1)
    {
        while (p!=NULL)
        {

```

```

        p=p->link;
        n++;
    }
    return(n);
}

```

实现本题功能的递归求解函数如下:

```

int length1(gnode *p)
{
    if (p==NULL) return(0);
    else return(length1(tail(p))+1); /* tail()为第2题编写的函数 */
}

```

4. 编写一个函数计算一个广义表的原子结点个数。例如一个广义表为(a,(b,c),(e)),其原子结点个数为4。

解:依题意,得到计算一个广义表的原子结点个数的递归模型如下:

$$\begin{cases} f(p)=0 & \text{若 } p=\text{NULL} \\ f(p)=1+f(p->\text{link}) & \text{若 } p->\text{tag}=0 \\ f(p)=f(p->\text{val. sublist})+f(p->\text{link}) & \text{若 } p->\text{tag}=1 \end{cases}$$

因此,实现本题功能的函数如下:

```

int counter(gnode *p)
{
    int m,n;
    if (p==NULL) return(0);
    else
    {
        if (p->tag==0) n=1;
        else n=counter(p->val. sublist);
        if (p->link!=NULL)
            m=counter(p->link);
        else m=0;
        return(m+n);
    }
}

```

5. 编写一个函数计算一个广义表的所有原子结点数据域(数据域为整数型)之和。例如一个广义表为((3,4),5,((6,3))),其所有原子结点数据域之和为21。

解:依题意,得到计算一个广义表的所有原子结点数据域之和的递归模型如下:

$$\begin{cases} f(p)=0 & \text{若 } p=\text{NULL} \\ f(p)=p->\text{val. data}+f(p->\text{link}) & \text{若 } p->\text{tag}=0 \\ f(p)=f(p->\text{val. sublist})+f(p->\text{link}) & \text{若 } p->\text{tag}=1 \end{cases}$$

因此,实现本题功能的函数如下:

```
int sum(gnode *p)
{
    int m,n;
    if (p==NULL) return(0);
    else
    {
        if (p->tag==0) n=p->val.data;
        else n=sum(p->val.sublist);
        if (p->link!=NULL)
            m=sum(p->link);
        else m=0;
        return(m+n);
    }
}
```

* 6. 编写一个函数接受任一无共享子表的非递归表 L, 求出此表的深度, 设表以链表形式存放, 每个结点有三个域:

tag	data/sublist	link
-----	--------------	------

$$\text{tag} = \begin{cases} 0 & \text{表示该结点为原子} \\ 1 & \text{表示该结点为表} \end{cases}$$

例如 $L=((A,B),C,((D,E),F))$ 的深度为 3。

解: 依题意, 本题采用的算法思想: 扫描通过广义表的第一层的每个结点 (使用 $p=p->\text{link}$ 语句), 对每个结点递归调用计算出其子表的深度, 取最大的子表深度, 然后加 1 即为该广义表的最大深度, 即递归模型如下:

$$\begin{cases} \text{maxdh}(p)=0 & p \text{ 为原子即 } p->\text{tag}=0 \\ \text{maxdh}(p)=1 & p \text{ 为空表即 } p->\text{tag}=1 \text{ 且 } p->\text{val.sublist}=\text{NULL} \\ \text{maxdh}(p)=\max(\text{maxdh}(p_1), \dots, \text{maxdh}(p_n))+1 & \text{否则, } p=(p_1, p_2, \dots, p_n) \end{cases}$$

因此,实现本题功能的函数如下:

```
int depth(gnode *p)
{
    int h,maxdh;
    gnode *q;
    if (p->tag==0) return(0); /* 若表头结点 tag 为 0, 则表示该表为原子 */
    else if (p->tag==1 && p->val.sublist==NULL) maxdh=1; /* 空表的情况 */
    else /* 否则, 要进行递归求解 */
    {
        maxdh=0; /* 赋初值 */
        while (p!=NULL)
```

```

    /* 循环扫描广义表的第一层的每个结点,对每个结点求其子表深度 */
    q=p->val.sublist;
    h=depth(q);
    if (h>maxdh) maxdh=h; /* 取最大的子表深度 */
    p=p->link;
};
return(maxdh+1); /* 最大子表深度加1即为该广义表的深度 */
}

```

7. 编写一个函数将两个广义表合并成一个广义表。合并是指元素的合并,例如两个广义表 $((a,b),(c))$ 与 $(a,(e,f))$ 合并后的结果是 $((a,b),(c),a,(e,f))$ 。

解:依题意,本题采用的算法思想:先找到第一个广义表的最后一个结点,将其链接到第二个广义表的首元素上即可。因此,实现本题功能的函数如下:

```

gnode * append(gnode * p, gnode * q)
{
    gnode * r;
    if (p != NULL && p->tag == 1)
    {
        r=p;
        while (r != NULL) r=r->link; /* r 指向最后一个结点 */
        if (q != NULL && q->tag == 1) r->link=q;
        else write('q 为非法广义表');
    }
    else write('p 为非法广义表');
    return(p);
}

```

8. 编写一个函数复制一个广义表,包括该广义表的所有原子结点和非原子结点。

解:将广义表 p 复制到广义表 q 的递归模型如下:

$$\left\{ \begin{array}{ll} \text{copy}(p,q) \rightarrow q = \text{NULL} & \text{当 } p = \text{NULL} \text{ 时} \\ \text{copy}(p,q) \rightarrow q \rightarrow \text{tag} = p \rightarrow \text{tag}, q \rightarrow \text{val.data} = p \rightarrow \text{val.data} & \text{当 } p \text{ 只有一个原子结点时,直接复制} \\ \text{copy}(p,q) \rightarrow \text{copy}(p \rightarrow \text{val.sublist}, q \rightarrow \text{val.sublist}), \text{copy}(p \rightarrow \text{link}, q \rightarrow \text{link}) & \text{否则,先复制第一个元素的子表(若存在),然后复制其后续元素} \end{array} \right.$$

因此,实现本题功能的函数如下:

```

void copy(gnode * p, gnode * q)
{
    if (p == NULL) q = NULL;
    else
    {
        q = (gnode *) malloc(sizeof(gnode));
        q->tag = p->tag;
    }
}

```

```

    if (p->tag==0) q->val.data=p->val.data; /* 原子结点直接复制 */
    else copy(p->val.sublist,q->val.sublist); /* 子表结点要递归调用复制子表 */
    copy(p->link,q->link); /* 复制该结点的后续表 */
}
}

```

9. 编写一个与二叉树前序遍历的递归算法相似的遍历广义表的算法,按照广义表的逻辑结构顺序,打印广义表所有元素结点上的数据域。

解:依题意,得到前序遍历广义表的递归模型如下:

```

{ preorder(p)→write(p->val.data),preorder(p->link)
  若 p 为原子结点,显示该结点数据域值,然后递归调用本函数遍历所有后续元素
{ preorder(p)→preorder(p->val.sublist),preorder(p->link)
  若 p 是子表结点,递归调用本函数遍历该子表,然后递归调用本函数遍历所有后续元素

```

因此,实现本题功能的函数如下:

```

preorder(gnode *p)
{
    if (p!=NULL)
    {
        if (p->tag==0) printf("%c",p->val.data);
        else preorder(p->val.sublist);
        if (p->link!=NULL)
            preorder(p->link);
    }
}

```

例如,广义表(a,(b,c,d),e,((f)))的输出结果是:

a b c d e f

10. 编写一个函数判定两个广义表是否相等。相等的含义是指两个广义表具有相同的存储结构,对应的原子结点的数据域值也相同。

解:依题意,得到判定两个广义表是否相等的递归模型如下:

```

{ same(p,q)=same(p->link,q->link)
  若 p->tag=0 且 q->tag=0 且 p->val.data=q->val.data
{ same(p,q)=same(p->val.sublist,q->val.sublist) and
  若 p->tag=1 且 q->tag=1 same(p->link,q->link)
{ same(p,q)=false
  若 p->tag=0 且 q->tag=0 且 p->val.data≠q->val.data
  或 p->tag=0 且 q->tag=1 或 p->tag=1 且 q->tag=0
{ same(p,q)=false
  若 p=NULL 且 q≠NULL 或 p≠NULL 且 q=NULL

```

因此,实现本题功能的函数如下:

```

int same(gnode *p,gnode *q)
{
    int faig=1;

```



```

if (p! =NULL &&. q! =NULL)
{
    if (p->tag==0 &&. p->tag+=0)
        if (p->val. data! =q->val. data) flag=0;
    else if (p->tag==1 &&. q->tag==1)
        flag=same(p->val. sublist,q->val. sublist);
    else flag=0;
    if (flag) flag=same(p->link,q->link);
}
else
{
    if (p==NULL &&. q! =NULL) flag=0;
    if (p! =NULL &&. q==NULL) flag=1;
}
return(flag);
}

```

* 11. 编写一个函数删除广义表中所有值为 x 的元素。例如删除广义表 $((a,b),a,(d,a))$ 中所有 a 的结果是广义表 $((b),(d))$ 。

解:依题意,得到删除广义表中所有值为 x 的元素的递归模型如下:

$$\begin{cases}
 f(p,x)=\text{NULL} & \text{若 } p\text{-}\rightarrow\text{tag}=0 \text{ 且 } p\text{-}\rightarrow\text{val. data}=x \text{ 且 } p\text{-}\rightarrow\text{link}=\text{NULL} \\
 f(p,x)=f(\text{tail}(p),x) & \text{若 } p\text{-}\rightarrow\text{tag}=0 \text{ 且 } p\text{-}\rightarrow\text{val. data}=x \text{ 且 } p\text{-}\rightarrow\text{link!}=\text{NULL} \\
 f(p,x)=\text{head}(p) & \text{若 } p\text{-}\rightarrow\text{tag}=0 \text{ 且 } p\text{-}\rightarrow\text{val. data}\neq x \text{ 且 } p\text{-}\rightarrow\text{link}=\text{NULL} \\
 f(p,x)=\text{append}(\text{head}(p),f(\text{tail}(p),x)) & \text{若 } p\text{-}\rightarrow\text{tag}=0 \text{ 且 } p\text{-}\rightarrow\text{val. data}\neq x \text{ 且 } p\text{-}\rightarrow\text{link!}=\text{NULL} \\
 f(p,x)=f(\text{head}(p),x) & \text{若 } p\text{-}\rightarrow\text{tag}=1 \text{ 且 } p\text{-}\rightarrow\text{link}=\text{NULL} \\
 f(p,x)=\text{append}(f(\text{head}(p),x),f(\text{tail}(p),x)) & \text{若 } p\text{-}\rightarrow\text{tag}=1 \text{ 且 } p\text{-}\rightarrow\text{link}\neq\text{NULL}
 \end{cases}$$

这里的 $\text{append}(a,b)$ 功能是将广义表 a 与 b 作为元素的广义表连接起来。

例如,求解 $f(((a,b),a,(d,a)))$ 的递归调用和求值过程如图 7.4 所示。

因此,实现本题功能的函数如下:

```

gnode * delall(gnode * p,char x)
{
    gnode * q,*r,*s,*t;
    if (p==NULL) q=NULL;
    else
    {
        /* 先删除第一个元素中所有的 x,其结果由 q 所指向 */
        if (p->tag==0) /* 若第一个元素为原子结点 */
        {
            if (p->val. data! =x)
            {
                q=(gnode *)malloc(sizeof(sizeof(gnode))); /* 复制该原子结点 */
                q->tag=0;
            }
        }
    }
}

```

```

    q->val.sublist=NULL;
    q->val.data=p->val.data;
}
else s=NULL;
}
else delall(p->val.sublist,x,s); /* 若第一个元素为子表结点 */
/* 再将 p 的后续部分进行递归删除,其结果由 s 所指向 */
if (p->link!=NULL)
{
    t=delall(p->link,x); /* 产生的 t 需作为广义表 q 的元素 */
    s=(gnode *)malloc(sizeof(gnode));
    s->tag=1;
    s->link=NULL;
    s->val.sublist=t;
    /* 将 s 连接到 q 的后面,这样便删除了所有的 x */
    if (s!=NULL)
    {
        r=q->val.sublist; /* 进入 q 的元素层 */
        while (r!=NULL) r=r->link; /* 找到最后一个元素 */
        r->link=s; /* 将 s 连接到最后 */
    }
}
return(q);
}

```

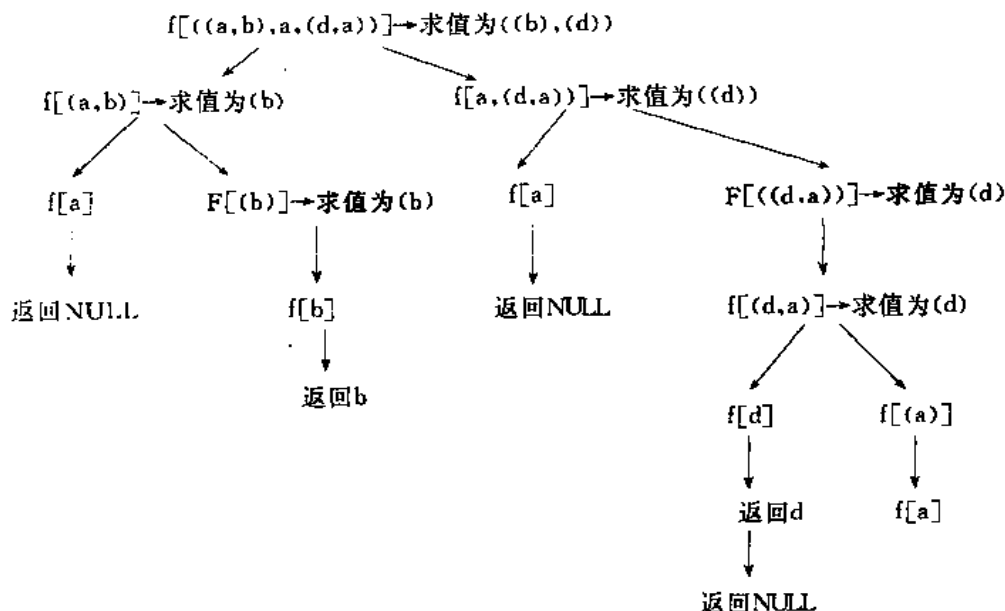


图 7.4 递归调用和求值过程

12. 编写一个函数将一个广义表逆置,例如原来广义表为 $((a,b),c,(d,e))$,经逆置后为 $((e,d),c,(b,a))$ 。

解:依题意,得到逆置广义表的递归模型如下:

$$\begin{cases} f(p)=\text{NULL} & \text{若 } p=\text{NULL} \\ f(p)=\text{head}(p) & \text{若 } p\rightarrow\text{tag}=0 \text{ 且 } p\rightarrow\text{link}=\text{NULL} \\ f(p)=\text{append}(f(\text{tail}(p)),\text{head}(p)) & \text{若 } p\rightarrow\text{tag}=0 \text{ 且 } p\rightarrow\text{link}\neq\text{NULL} \\ f(p)=\text{append}(f(\text{tail}(p)),f(\text{head}(p))) & \text{若 } p\rightarrow\text{tag}=1 \end{cases}$$

这里的 $\text{append}(a,b)$ 功能是将广义表 a 与 b 作为元素的广义表连接起来。

例如:求解 $f(((a,b),c,(d,e)))$ 的递归调用和求值过程如图 7.5 所示。

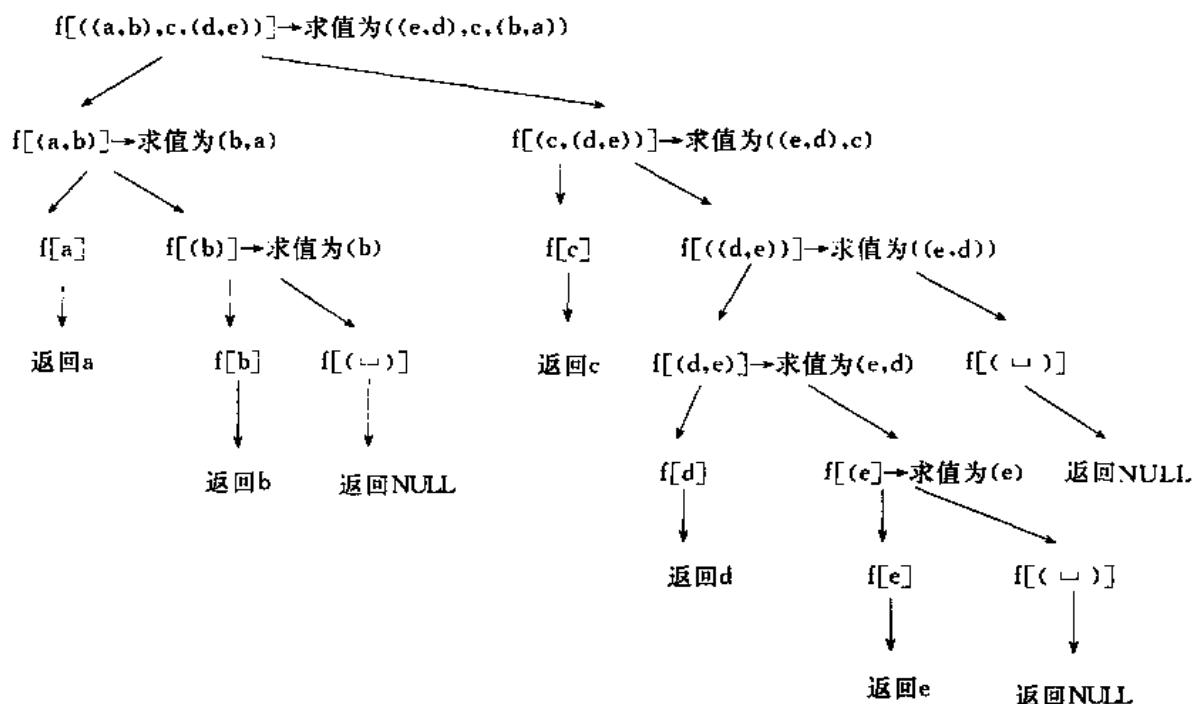


图 7.5 递归调用和求值过程

因此,实现本题功能的函数如下:

```

void reverse(gnode *p,gnode *q)
{
    gnode *r,*s,*t;
    if (p==NULL) q=NULL;
    else
    {
        /* 先将第一个元素进行逆置,其结果由 s 所指向 */
        if (p->tag==0) /* 若第一个元素为原子结点 */
        {
            s=(gnode *)malloc(sizeof(gnode)); /* 复制该原子结点 */
            s->tag=0;
        }
    }
}
  
```

```
s->val.sublist=NULL;
s->val.data=p->val.data;
}
else reverse(p->val.sublist,s); /* 若第一个元素为子表结点 */
/* 再将p的后续部分进行递归逆置,其结果由q所指向 */
if (p->link!=NULL)
{
    reverse(p->link,t); /* 产生的t需作为广义表q的元素 */
    q=(gnode *)malloc(sizeof(gnode));
    q->tag=1;
    q->link=NULL;
    q->val.sublist=t;
    /* 将s连接到q的后面,这样便进行了整个表p的逆置 */
    if (q!=NULL)
    {
        r=q->val.sublist; /* 进入q的元素层 */
        while (r!=NULL) r=r->link; /* 找到最后一个元素 */
        r->link=s; /* 将s连接到最后 */
    }
}
else q=s; /* 若无后续部分 */
}
```

第8章 树形结构

树形结构的简称为树,它是一种重要的非线性数据结构。它为计算机应用中出现的具有层次关系或分支关系的数据提供了一种自然的表示方法。

8.1 基本概念和运算

下面讨论树、一般二叉树、二叉排序树、线索二叉树和 Huffman 树的各种存储方式和相应的运算。

8.1.1 树

1. 树的几个基本术语

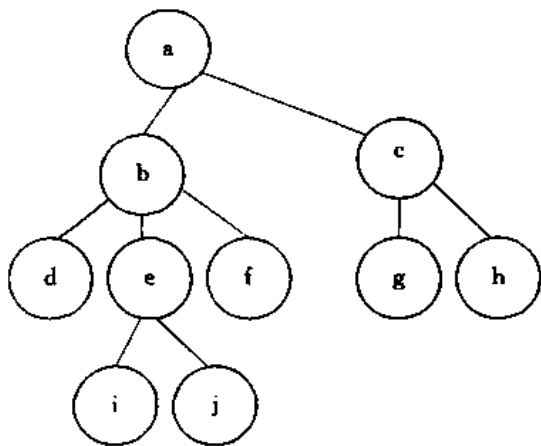
- 树:树是 $n(n>0)$ 个结点的有限集合 T 。在一棵树中满足如下两个条件:
 - 有且仅有一个称作根的结点;
 - 其余的结点可分为 $m(m\geq 0)$ 棵互不相交的有限集合 T_1, T_2, \dots, T_n , 其中每个集合又都是一棵树, 并称其为根的子树。
- 因此,树的定义是递归的,树是一种递归数据结构。树的这种定义为树的递归处理带来了很大的方便。
- 结点的度:树中每个结点具有的子树数或者后继结点数称为该结点的度。
 - 树的度:树中所有结点的度的最大值称之为树的度。
 - 分支结点:度大于 0 的结点称为分支结点或非终端结点。
 - 叶子结点:度为 0 的结点称为叶子结点或终端结点。
 - 儿子结点:一个结点的后继称之为该结点的儿子结点。
 - 父亲结点:一个结点称为其后继结点的父亲结点。
 - 子孙结点:一个结点的所有子树中的结点称之为该结点的子孙结点。
 - 祖先结点:从树根结点到达到一个结点的路径上通过的所有结点称为该结点的祖先结点。
 - 兄弟结点:具有同一父亲的结点互相称之为兄弟结点。
 - 结点的层数:树具有一种层次结构,根结点为第一层,其儿子结点为第二层,如此类推得到每个结点的层数。
 - 树的深度:树中结点的最大层数称为树的深度或高度。
 - 森林:0 个或多个不相交的树的集合称为森林。

2. 树的表示

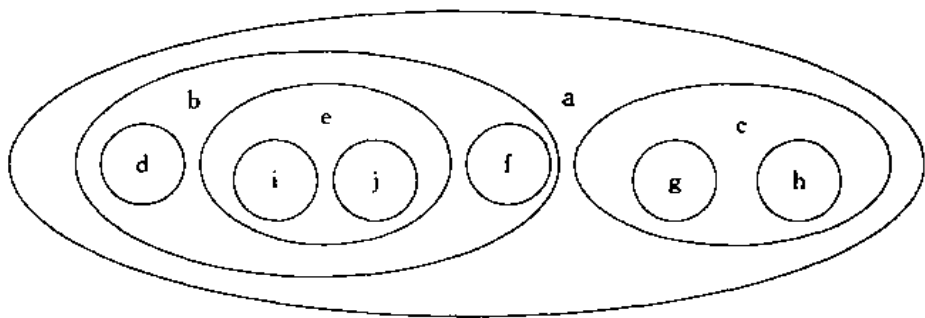
树有树形表示法、文氏图表示法、凹入表表示法和嵌套括号表示法等。一个如下逻辑结构的树:

$T = (K, N)$
 $K = \{a, b, c, d, e, f, g, h, i, j\}$
 $N = \{\langle a, b \rangle, \langle a, c \rangle, \langle b, d \rangle, \langle b, e \rangle, \langle b, f \rangle, \langle c, g \rangle, \langle c, h \rangle, \langle e, i \rangle, \langle e, j \rangle\}$

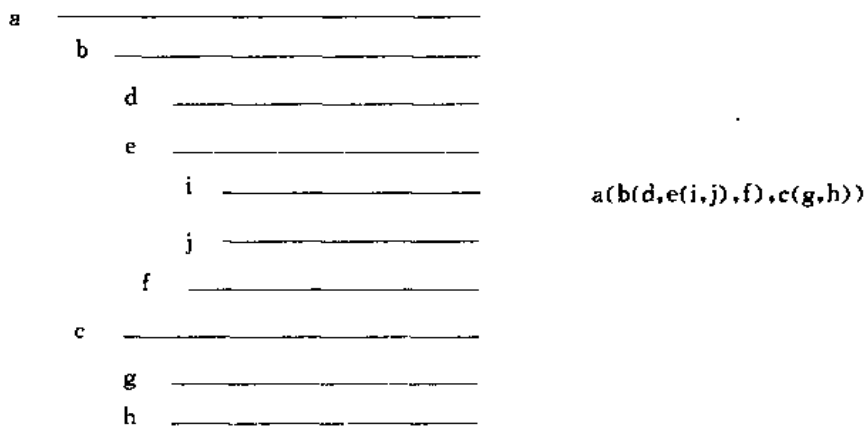
采用各种表示方法如图 8.1 所示。



(a) 树形表示法



(b) 文氏图表示法



(c) 凹入表表示法

(d) 嵌套括号表示法

 $a(b(d, e(i, j), f), c(g, h))$

图 8.1 树的各种表示法

8.1.2 二叉树

1. 二叉树的几个基本术语

- 二叉树: 二叉树由结点的有限集合构成, 这个有限集合或者为空集, 或者是由一个根结点及两棵不相交的分别称之为这个根的左子树和右子树的二叉树组成。二叉树是一种特殊的树。
- 满二叉树: 在一棵二叉树中, 若第 i 层的结点数为 $2^i - 1$ 时, 则称此层的结点数是满的, 当树中的每一层都是满的, 则此二叉树称之为满二叉树。
- 完全二叉树: 在一棵二叉树中, 除最后一层外, 若其余层都是满的, 并且最后一层或者是满的, 或者是在右边缺少连续若干个结点, 则此二叉树称之为完全二叉树。

2. 二叉树的存储

二叉树有顺序存储、链接存储和穿线树存储等几种存储方法。

(1) 顺序存储

顺序存储一棵二叉树时, 首先要对该树中每个结点进行编号, 然后以各结点的编号为下标, 把各结点的值对应存储到一维数组中。树中各结点的编号与等深度的完全二叉树中对应位置上结点的编号相同。

其编号过程为: 首先把树根结点的编号定为 1, 然后按照层次从上到下、每层从左到右的顺序, 对每一结点进行编号, 当它的双亲结点的编号为 i 时, 若它为左孩子, 则编号为 $2i$, 若为右孩子, 则编号为 $2i+1$ 。

如图 8.2 所示, 其中(a)的二叉树用顺序存储方法表示(采用一维数组 data 来进行顺序存储)为(b)。

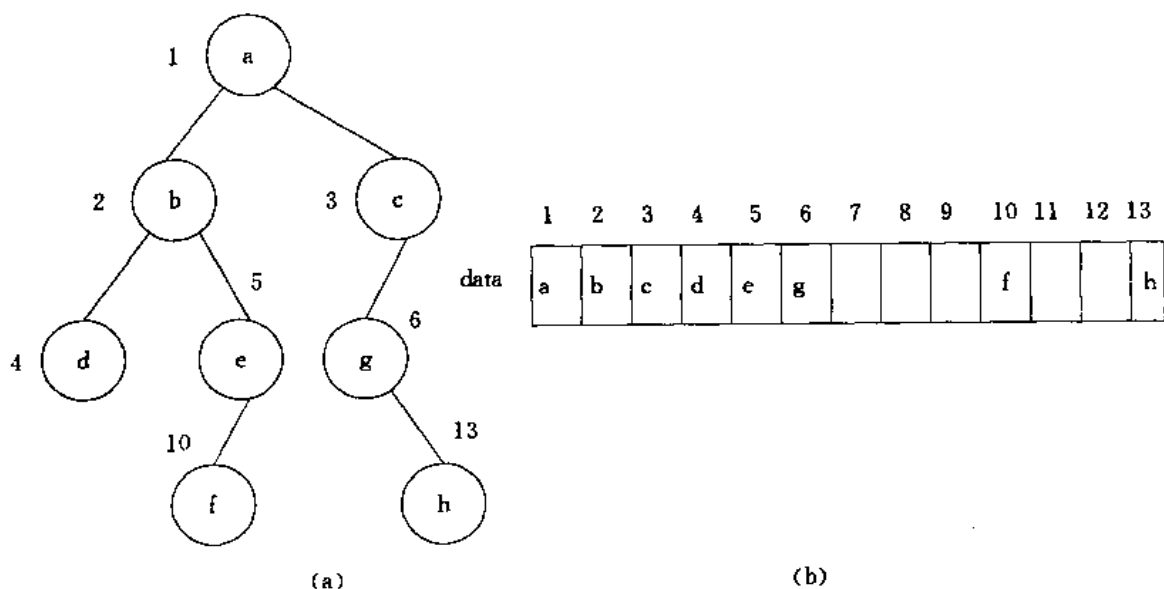


图 8.2 二叉树的顺序存储方法

(2) 链接存储

在二叉树的链接存储中,通常采用的方法是:每个结点中设置三个域,即值域、左指针域和右指针域,其结点结构如下:

left	data	right
------	------	-------

其中 data 表示值域,用于存储放入结点的数据元素, left 和 right 分别表示左指针域和右指针域,用以分别存储左孩子和右孩子结点(即左右子树的根结点)的存储位置(即指针)。

链接存储的指针类型和结点定义如下:

```
typedef struct bnode
{
    ElemType data;
    struct bnode * left, * right;
} btree;
```

这里的 ElemType 可以是任何相应的数据类型如 int, float 或 char 等,在算法中,我们规定 ElemType 缺省是 int 类型。

如图 8.3 所示,其中(a)的二叉树用链接存储方法表示为(b)。链接存储方法存在大量的空指针,浪费存储空间,但二叉树的运算算法实现比较简单直观。

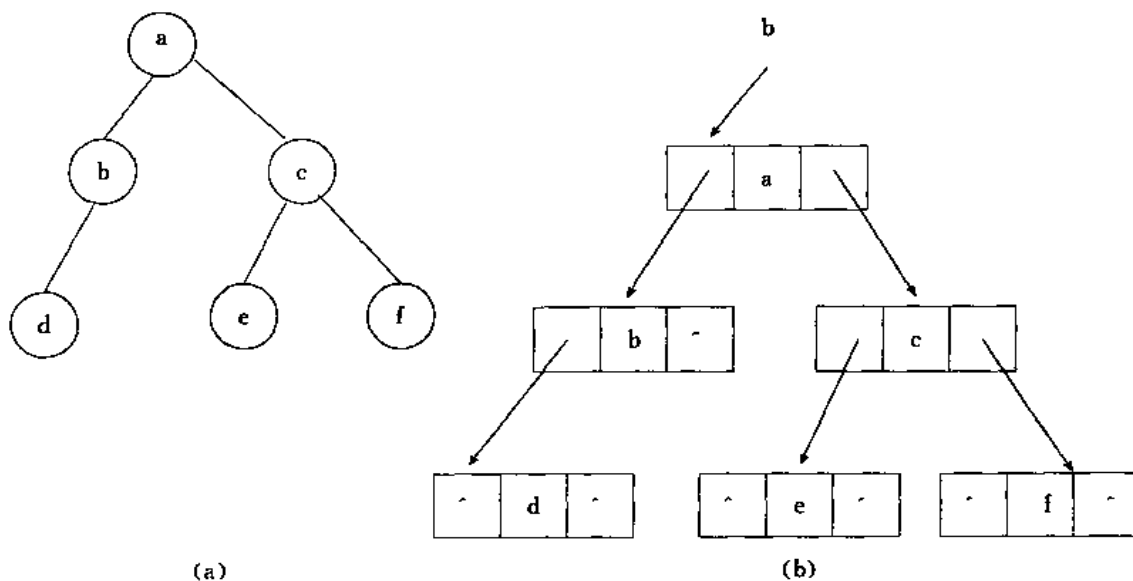


图 8.3 二叉树的链接存储表示

(3) 穿线树存储

链接存储中无左子树或右子树的结点的 left 域或 right 域为空,这样浪费了存储空间,(当一棵二叉树有 n 个结点,便有 $n+1$ 个指针域为 NULL。因为 n 个结点一共有 $2n$ 个指针域,除根结点外,每个结点有且仅有一个指向它的指针,于是共有 $n-1$ 个指针,即只有 $n-1$ 个指针域被有效使用)因此,可以把每个空着的 left 域或 right 域用于分别指向某种遍历次

序的前驱结点和后续结点,这样在遍历这种二叉树时,可由此信息直接找到遍历次序下的前驱结点或后续结点,从而加速了遍历速度,这显然比递归遍历要快得多。这种在结点的空指针域中存放的该结点在某次遍历次序下的前驱结点或后续结点的指针叫做线索,其中在空的左指针域中存放的指向其前驱结点的指针叫做左线索,在空的右指针域中存放的指向其后续结点的指针叫做右线索。对一棵二叉树中的所有结点的空指针域按照某种遍历次序加线索的过程叫做线索化,被线索化了的二叉树称作线索二叉树。

图 8.4 中(b)就是(a)的二叉树加中序线索而得到的中序线索二叉树。

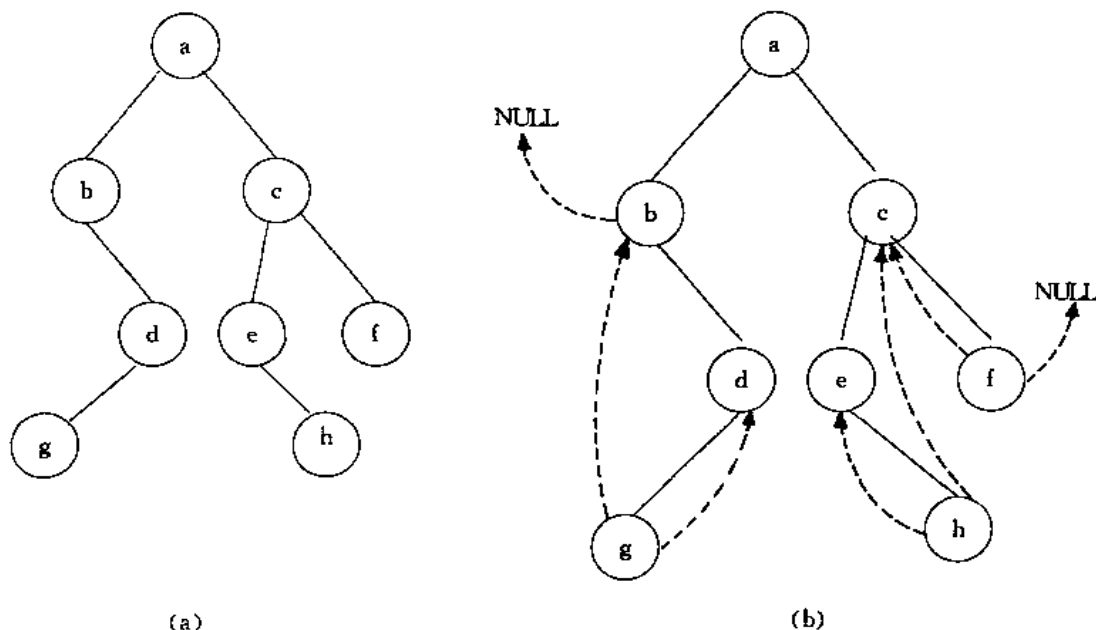


图 8.4 中序线索二叉树

在一个线索二叉树中,为了区别每个结点的左、右指针域所存放的是孩子指针,还是线索,必须在结点结构中增加两个线索标志域,即左线索标志域,用 ltag 表示,另一个是右线索标志域,用 rtag 表示。ltag 和 rtag 只需取两种值,以区别其对应的指针域保存的是孩子指针还是线索,假定取 0 和 1 两种值,并定义如下:

$$\begin{aligned} \text{ltag} &= \begin{cases} 0 & \text{left 域指向结点的左孩子} \\ 1 & \text{left 域指向结点的某种遍历次序下的前驱(左线索)} \end{cases} \\ \text{rtag} &= \begin{cases} 0 & \text{right 域指向结点的右孩子} \\ 1 & \text{right 域指向结点的某种遍历次序下的后续(右线索)} \end{cases} \end{aligned}$$

增加线索标志域后的结点结构为:

left	ltag	data	rtag	right
------	------	------	------	-------

这种结点类型和相应结点的指针类型定义如下:

```
typedef struct tnode
```

```

{
    ElemType data;
    int ltag, rtag; /* 这里的 ltag 和 rtag 只能取值 0 或 1 */
    struct tnode * left, * right;
} tbtreenode;

```

对一种二叉树进行中序线索化的算法思想是：一边中序遍历一边建立线索。若访问结点的左孩子为空，则建立前驱线索；若右孩子为空，则建立后续线索。其函数如下：

```

void inthread(p, pre)
tbtreenode * p, * pre;
/* p 为当前结点, pre 为 p 的前驱结点, 开始调用时 p 为根结点指针, pre 为 NULL */
{
    if (p != NULL)
    {
        inthread(p->left, pre); /* 左子树线索化 */
        /* 若当前结点的左子树为空, 则建立指向其前驱结点的前驱线索 */
        if (p->left == NULL)
        {
            p->ltag = 1;
            p->left = pre;
        }
        else p->ltag = 0;
        /* 若前驱结点不为空, 且其右孩子为空, 则建立该前驱结点指向当前结点的后续线索 */
        if (pre != NULL && pre->right == NULL)
        {
            pre->rtag = 1;
            pre->right = p;
        }
        else p->rtag = 0;
        pre = p; /* 中序向前遍历一个结点 */
        inthread(p->right, pre);
    }
}

```

3. 二叉树的运算

为了简单直观, 这里的运算均假设二叉树是采用链接存储方式存储的。

(1) 二叉树的遍历

- 前序遍历: 按照先访问根结点, 再访问左子树, 最后访问右子树的次序访问二叉树中所有结点, 且每个结点仅访问一次。其递归算法如下:

```

void preorder(btree * p)
{
    if (p != NULL)
    {

```

```

        printf("%d ", p->data);
        preorder(p->left);
        preorder(p->right);
    }
}

```

- 中序遍历:按照先访问左子树,再访问根结点,最后访问右子树的次序访问二叉树中所有结点,且每个结点仅访问一次。其递归算法如下:

```

void inorder(btree *p)
{
    if (p != NULL)
    {
        inorder(p->left);
        printf("%d ", p->data);
        inorder(p->right);
    }
}

```

- 后序遍历:按照先访问左子树,再访问右子树,最后访问根结点的次序访问二叉树中所有结点,且每个结点仅访问一次。其递归算法如下:

```

void postorder(btree *p)
{
    if (p != NULL)
    {
        postorder(p->left);
        postorder(p->right);
        printf("%d ", p->data);
    }
}

```

(2) 输出二叉树

给定一个二叉树,输出其嵌套括号表示。

采用的算法是:首先输出根结点,然后再依次输出它的左子树和右子树,不过在输出左子树之前要打印出左括号,在输出右子树之后要打印出右括号;另外,依次输出的左、右子树要至少有一个不为空,若都为空就不必输出了。

因此,求输出二叉树的函数如下:

```

void print(btree *b)
{
    if (b != NULL)
    {
        printf("%d", b->data);
        if (b->left != NULL || b->right != NULL)
        {

```

```

        printf("(");
        print(b->left);
        if (b->right != NULL) printf(",");
        print(b->right);
        printf(")");
    }
}

```

(3) 求二叉树的深度

若一棵二叉树为空,则其深度为 0,否则其深度等于左子树或右子树的最大深度加 1,即有如下递归模型:

$$\begin{cases} \text{depth}(b) = 0 & \text{若 } b = \text{NULL} \\ \text{depth}(b) = \max(\text{depth}(b \rightarrow \text{left}), b \rightarrow \text{right}) + 1 & \text{其他} \end{cases}$$

因此,求二叉树的深度的递归函数如下:

```

int depth(btree * b)
{
    int dep1, dep2;
    if (b == NULL) return(0);
    else
    {
        dep1 = depth(b->left);
        dep2 = depth(b->right);
        if (dep1 > dep2) return(dep1 + 1);
        else return(dep2 + 1);
    }
}

```

8.1.3 二叉排序树

1. 二叉排序树的定义

所谓二叉排序树,指的是一棵为空的二叉树,或者是一棵具有如下特性的非空二叉树:

- 若它的左子树非空,则左子树上所有结点的值均小于根结点的值;
- 若它的右子树非空,则右子树上所有结点的值均大于根结点的值;
- 左、右子树本身又各是一棵二叉排序树。

2. 二叉排序树的查找

在二叉排序树 b 中查找 x 的过程为:

- (1) 若 b 是空树,则搜索失败,否则
- (2) 若 x 等于 b 的根结点的数据域之值,则查找成功,否则
- (3) 若 x 小于 b 的根结点的数据域之值,则搜索左子树;否则
- (4) 查找右子树。

因此,查找二叉排序树返回成功找到的结点指针的函数如下:

```
btree * search(btree * b,int k)
{
    if (b==NULL) return(NULL);
    else
    {
        if (b->data==x) return(b);
        if (x<b->data) return(search(b->left));
        else return(search(b->right));
    }
}
```

3. 二叉排序树的生成

先讨论向一个二叉排序树 b 中插入一个结点 s 的算法,其函数为:

- (1) 若 b 是空树,则将 s 所指结点作为根结点插入;否则
- (2) 若 s->data 等于 b 的根结点的数据域之值,则返回;否则
- (3) 若 s->data 小于 b 的根结点的数据域之值,则把 s 所指结点插入到左子树中;否则
- (4) 把 s 所指结点插入到右子树中。

因此,向一个二叉排序树 b 中插入一个结点 s 的函数如下:

```
void insert(b,s)
btree * b,* s;
{
    if (b==NULL) b=s
    else if (s->data==b->data)
        return(); /* 不作任何插入操作 */
    else if (s->data<b->data)
        insert(b->left,s); /* 将 s 插入到左子树中 */
    else if (s->data>b->data)
        insert(b->right,s); /* 将 s 插入到右子树中 */
}
```

生成二叉排序树的过程是先有一个空树 b,然后向该空树插入一个个结点实现的,因此,根据用户输入的一系列正整数(以-1 标志结束)生成一棵二叉排序树的函数如下:

```
void creat(btree * b)
{
    int x;
    btree * s;
    b=NULL;
    do
    {
```

```

scanf("%d",&x);                /* 读入一个整数 */
s=(bnode *)malloc(sizeof(bnode)); /* 产生一个树结点 */
s->data=x;
s->left=NULL;
s->right=NULL;
insert(b,s);                    /* 插入该结点 */
} while (x!= -1);
}

```

4. 二叉排序树的删除

删除二叉排序树 b 中一个数据域为 x 的结点的过程为:

- (1) 首先查找到数据域为 x 的结点 p 。
- (2) 若 p 所指没有左子树,则用右子树的根代替被删的结点。
- (3) 若 p 所指结点有左子树,则在其左子树中找到最右结点 r 来代替被删的结点(即将 r 所指结点的右指针置成 p 所指结点的右子树的根,然后用 p 所指结点的左子树的根结点代替被删的 p 所指结点)。

因此,在二叉排序树 b 中删除一个数据域为 x 的结点的函数如下:

```

void delnode(btree *b,int x)
{
    btree *p,*q,*r,*t;
    p=b;                /* p 指向待比较的结点 */
    q=NULL;             /* q 指向 p 的前驱结点 */
    while (p!=NULL && p->data!=x)
    {
        if (x<p->data)
        {
            q=p;
            p=p->left;
        }
        else
        {
            q=p;
            p=p->right;
        }
    }
    if (p==NULL) printf("未发现数据域为%d的结点\n",x);
    else if (p->left==NULL) /* 被删结点无左子树 */
    {
        if (q==NULL) t=p->right;
        else if (q->left==p) q->left=p->right;
        else q->right=p->right;
    }
    else /* 被删结点有左子树 */
    {

```

```

/* 查找被删结点的左子树中的最右结点,即刚好小于x的结点 */
r=p->left;
while (r->right != NULL)
    r=r->right;
/* 被删结点的右子树作为r的右子树 */
r->right=p->right;
/* 被删结点的左子树根代替被删结点 */
if (q==NULL) t=p->left;      /* 被删结点是根结点 */
else if (q->left==p) q->left=p->left;
    else q->right=p->left;
}
}

```

8.1.4 树和森林

1. 树的存储

树也有多种存储方法,这里仅讨论最常用的孩子兄弟表示法。即以二叉树链表作为树的存储结构。链表中结点的两个指针域分别指向该结点的第一个孩子结点和下一个兄弟结点。如图 8.5 所示(a)中树采用孩子兄弟表示成(b)的形式。

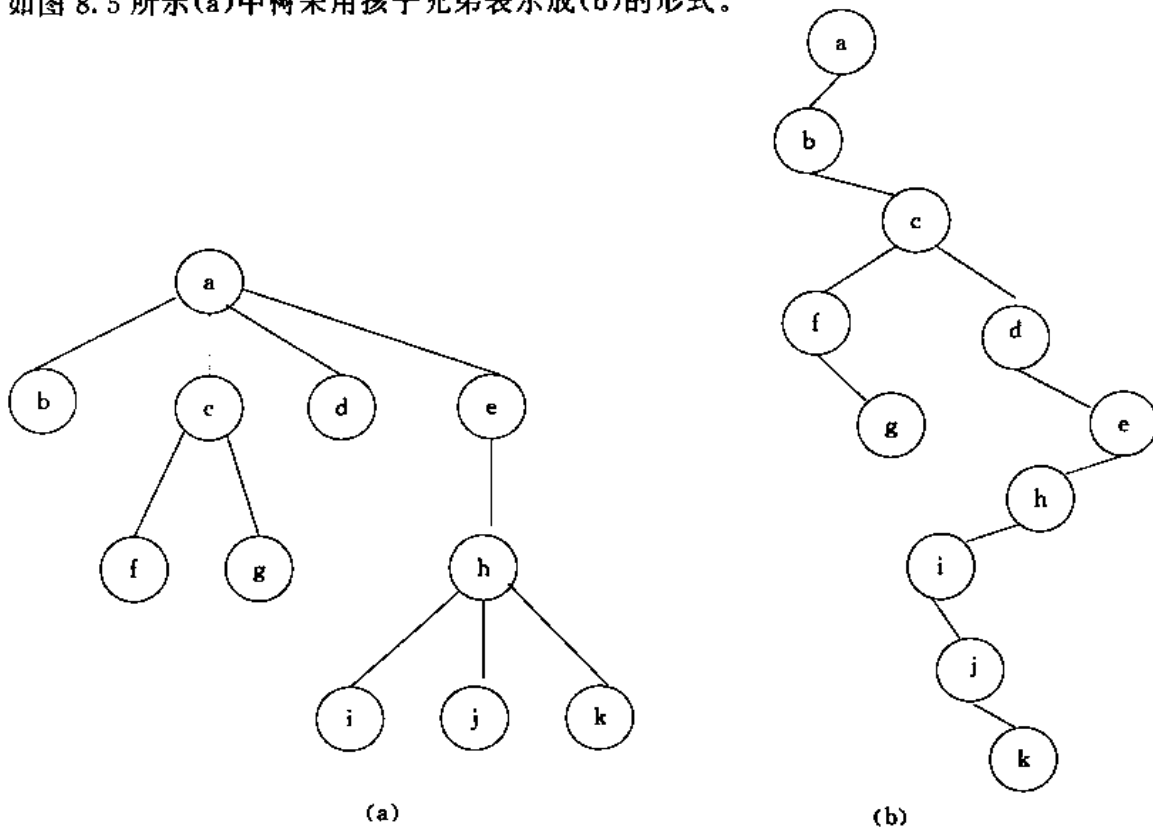


图 8.5 一棵树及其孩子兄弟表示

2. 森林和二叉树的转换

(1) 森林转换成二叉树

若 $F = \{T_1, T_2, \dots, T_m\}$ 是森林, 则可按如下规则转换成二叉树 $B = (\text{root}, \text{LB}, \text{RB})$:

- 若 F 为空, 即 $m=0$, 则 B 为空树。
- 若 F 不为空, 即 $m \neq 0$, 则 B 的根 root 即为森林中第一棵树的根; B 的左子树 LB 是从 T_1 中根结点的子树森林 $F_1 = \{T_{11}, T_{12}, \dots, T_{1m}\}$ 转换而成的二叉树; 其右子树 RB 是从森林 $F' = \{T_2, T_3, \dots, T_m\}$ 转换而成的二叉树。

(2) 二叉树转换成森林

如果 $B = (\text{root}, \text{LB}, \text{RB})$ 是一棵二叉树, 则可按如下规则转换成森林 $F = \{T_1, T_2, \dots, T_m\}$:

- 若 B 为空, 则 F 为空。
- 若 B 不为空, 则 F 中第一棵树 T_1 的根即为二叉树 B 的根 root ; T_1 中根结点的子树森林 F_1 是由 B 的左子树 LB 转换而成森林; F 中除 T_1 之外其余树组成的森林 $F' = \{T_2, T_3, \dots, T_m\}$ 是由 B 的右子树 RB 转换而成的森林。

8.1.5 Huffman 树

1. 基本术语

- 路径和路径长度: 若在一棵树中存在着一个结点序列 k_1, k_2, \dots, k_j , 使得 k_i 是 k_{i+1} 的父亲 ($1 \leq i < j$), 则称该结点序列是从 k_1 到 k_j 的路径。从 k_1 到 k_j 所经过的分支数称为这两点之间的路径长度。
- 结点的权和带权路径长度: 树中的结点上赋予的一定意义的实数称之为该结点的权。从树根结点到该结点之间的路径长度与该结点上权的乘积称为该结点的带权路径长度。
- 树的带权路径长度: 树的带权路径长度定义为树中所有叶子结点的带权路径长度之和, 记为:

$$\text{WPL} = \sum_{k=1}^n w_k l_k$$

其中 n 表示叶子结点个数, w_i 和 l_i 分别表示叶子结点 k_i 的权值和根到 k_i 之间的路径长度。

- Huffman 树: Huffman 树又称最优二叉树, 它是 n 个带权叶子结点构成的所有二叉树中, 带权路径长度 WPL 最小的二叉树。

2. 构造 Huffman 树

构造 Huffman 树的算法如下:

- (1) 根据与 n 个权值 $\{w_1, w_2, \dots, w_n\}$ 对应的 n 个结点构成 n 棵二叉树的森林 $F = \{T_1, T_2, \dots, T_n\}$, 其中每棵二叉树 $T_i (1 \leq i \leq n)$ 都只有一个权值为 w_i 的根结点, 其左、右子树均为空;

(2) 在森林 F 中选出两棵根结点的权值最小的树作为一棵新树的左、右子树,且置新树的附加根结点的权值为其左、右子树上根结点的权值之和;

(3) 从 F 中删除这两棵树,同时把新树加入 F 中;

(4) 重复(2)和(3),直到 F 中只含有一棵树为止,此树便是哈夫曼树。

3. Huffman(哈夫曼)编码

Huffman 编码的应用很广泛,利用 Huffman 树构造的用于通信的二进制编码称为 Huffman 编码。例如,有一段电文“CAST-TAT-A-SA”。统计电文中字母的频度 $f('C')=1, f('S')=2, f('T')=3, f('A')=4$ 。用频度 $\{1, 2, 3, 3, 4\}$ 为权值生成 Huffman 树,并在每个叶子上注明对应的字符。树中从根到每个叶子都有一条路径,对路径上的各分支约定指向左子树根的分支表示“0”码,指向右子树的分支表示“1”码,取每条路径上的“0”或“1”的序列作为和各个叶子对应的字符的编码,这就是 Huffman 编码。对应图 8.6 的 Huffman 树,上述字符编码为:

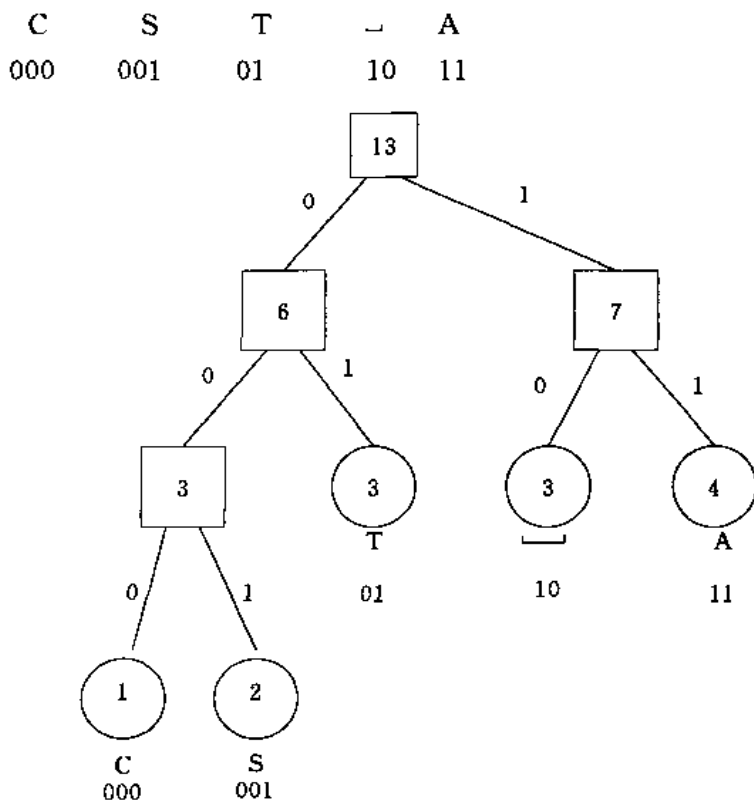


图 8.6 Huffman 编码树

8.2 基本题

8.2.1 单项选择题

1. 如图 8.7 所示的 4 棵二叉树中,① 不是完全二叉树。

答:①C

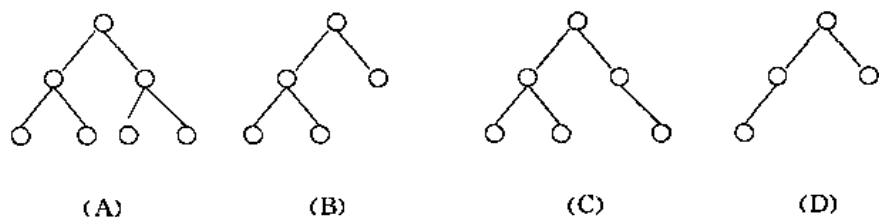


图 8.7 4 棵二叉树

2. 如图 8.8 所示的 4 棵二叉树,① 是平衡二叉树。

答:①B

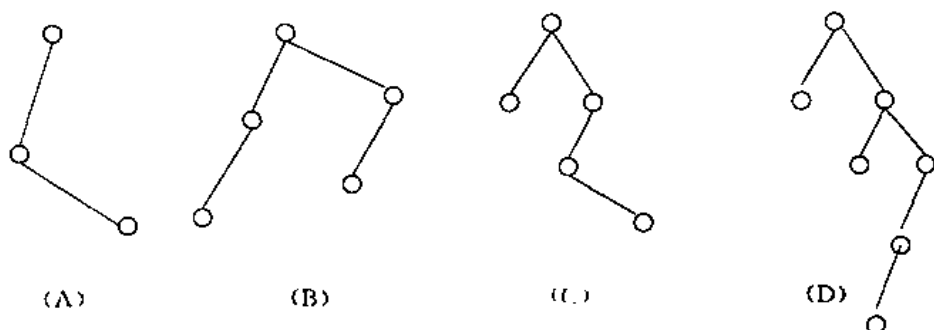


图 8.8 4 棵二叉树

3. 在线索化二叉树中, t 所指结点没有左子树的充要条件是 ①。

A. $t \rightarrow \text{left} = \text{NULL}$

B. $t \rightarrow \text{ltag} = 1$

C. $t \rightarrow \text{ltag} = 1$ 且 $t \rightarrow \text{left} = \text{NULL}$

D. 以上都不对

答:①B

4. 二叉树按某种顺序线索化后,任一结点均有指向其前驱和后继的线索,这种说法

①。

A. 正确

B. 错误

答:①B

5. 二叉树的前序遍历序列中,任意一个结点均处在其子女结点的前面,这种说法 ①。

A. 正确

B. 错误

答:①A

6. 由于二叉树中每个结点的度最大为 2,所以二叉树是一种特殊的树,这种说法 ①。

A. 正确

B. 错误

答:①B

7. 设高度为 h 的二叉树上只有度为 0 和度为 2 的结点,则此类二叉树中所包含的结点数至少为 ①。

- A. $2h$ B. $2h-1$ C. $2h+1$ D. $h+1$

答:①B

8. 如图 8.9 所示二叉树的中序遍历序列是 ①。

- A. abcdgef B. dfefbagc
C. dbaefcg D. defbagc

答:①C

9. 已知某二叉树的后序遍历序列是 dabec, 中序遍历序列是 debac, 它的前序遍历序列是 ①。

- A. acbed B. decab
C. deabc D. cedba

答:①D

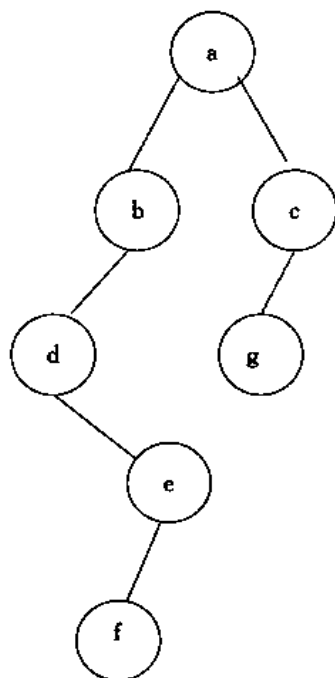


图 8.9 一棵二叉树

10. 如果 T2 是由有序树 T 转换而来的二叉树, 那么 T 中结点的前序就是 T2 中结点的 ①。

- A. 前序 B. 中序 C. 后序 D. 层次序

答:①A

11. 如果 T2 是由有序树 T 转换而来的二叉树, 那么 T 中结点的后序就是 T2 中结点的 ①。

- A. 前序 B. 中序 C. 后序 D. 层次序

答:①B

12. 某二叉树的前序遍历结点访问顺序是 $abdgcfeh$, 中序遍历的结点访问顺序是 $dgbaechf$, 则其后序遍历的结点访问顺序是 ①。

- A. $bdgcefha$ B. $gdbecfha$
C. $bdgaechf$ D. $gdbehfca$

答: ①D

13. 二叉树为二叉排序树的充分必要条件是任结点的值均大于其左孩子的值、小于其右孩子的值。这种说法 ①。

- A. 正确 B. 错误

答: ①B

14. 按照二叉树的定义, 具有 3 个结点的二叉树有 ① 种。

- A. 3 B. 4 C. 5 D. 6

答: ①C

15. 一棵二叉树如图 8.10 所示, 其中序遍历的序列为 ①。

- A. $abdgcfeh$ B. $dgbaechf$
C. $gdbehfca$ D. $abcdefgh$

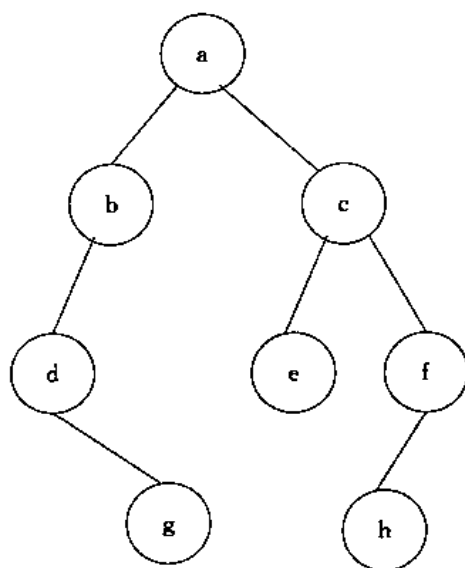


图 8.10 一棵二叉树

答: ①B

16. 树的基本遍历策略可分为先根遍历和后根遍历; 二叉树的基本遍历策略可分为先序遍历、中序遍历和后序遍历。这里, 我们把由树转化得到的二叉树叫做这棵树对应的二叉树。结论 ① 是正确的。

- A. 树的先根遍历序列与其对应的二叉树的先序遍历序列相同
B. 树的后根遍历序列与其对应的二叉树的后序遍历序列相同
C. 树的先根遍历序列与其对应的二叉树的中序遍历序列相同

D. 以上都不对

答:①A

17. 深度为5的二叉树至多有①个结点。

A. 16 B. 32 C. 31 D. 10

答:①C

18. 在一非空二叉树的中序遍历序列中,根结点的右边①。

A. 只有右子树上的所有结点 B. 只有右子树上的部分结点
C. 只有左子树上的部分结点 D. 只有左子树上的所有结点

答:①A

19. 树最适合用来表示①。

A. 有序数据元素
B. 无序数据元素
C. 元素之间具有分支层次关系的数据
D. 元素之间无联系的数据

答:①C

20. 任何一棵二叉树的叶结点在先序、中序和后序遍历序列中的相对次序①。

A. 不发生改变 B. 发生改变
C. 不能确定 D. 以上都不对

答:①A

21. 实现任意二叉树的后序遍历的非递归算法而不使用栈结构,最佳方案是二叉树采用①存储结构。

A. 二叉链表 B. 广义表存储结构
C. 三叉链表 D. 顺序存储结构

答:①C

22. 对一个满二叉树,m个树叶,n个结点,深度为h,则①。

A. $n=h+m$ B. $h+m=2n$ C. $m=h-1$ D. $n=2^h-1$

答:①D

23. 如果某二叉树的前序为stuwv,中序为uwtvs,那么该二叉树的后序为①。

A. uwvts B. vwuts C. wuvts D. wutsv

答:①C

24. 具有五层结点的二叉平衡树至少有①个结点。

A. 10 B. 12 C. 15 D. 17

答:①C

25. 如图8.11所示的t2是由有序树t1转换而来的二叉树,那么树t1有①个叶结点。

A. 4 B. 5 C. 6 D. 7

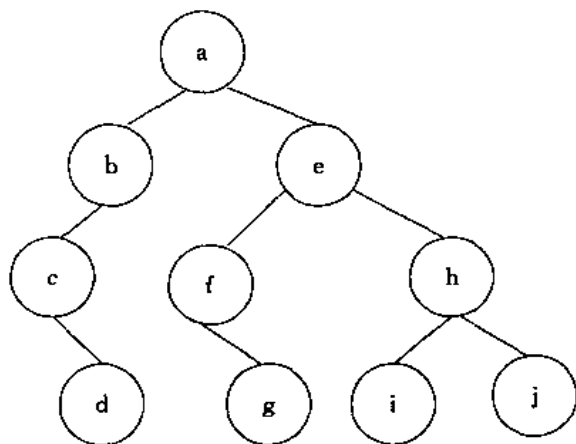


图 8.11 一棵二叉树

答:①D

26. 设 n, m 为一棵二叉树上的两个结点,在中序遍历时, n 在 m 前的条件是 ①。

- A. n 在 m 右方 B. n 是 m 祖先
C. n 在 m 左方 D. n 是 m 子孙

答:①C

27. 线索二叉树是一种 ① 结构。

- A. 逻辑 B. 逻辑和存储
C. 物理 D. 线性

答:①C

8.2.2 填空题(将正确的答案填在相应的空中)

1. 有一棵树如图 8.12 所示,回答下面的问题:

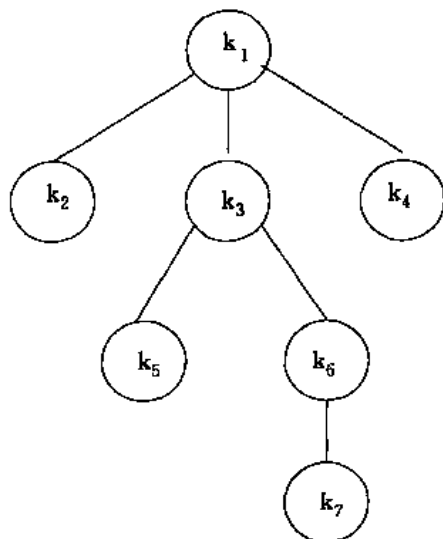


图 8.12 一棵树

- (1) 这棵树的根结点是 ①;
- (2) 这棵树的叶子结点是 ②;
- (3) 结点 k3 的度是 ③;
- (4) 这棵树的度为 ④;
- (5) 这棵树的深度是 ⑤;
- (6) 结点 k3 的子女是 ⑥;
- (7) 结点 k3 的父结点是 ⑦。

答:①k1 ②k2,k5,k7,k4 ③2 ④3 ⑤4 ⑥k5,k6 ⑦k1

2. 指出树和二叉树的三个主要差别①,②,③。

答:①树的结点个数至少为 1,而二叉树的结点个数可以为 0

②树中结点的最大度数没有限制,而二叉树结点的最大度数为 2

③树的结点无左、右之分,而二叉树的结点有左、右之分

3. 从概念上讲,树与二叉树是两种不同的数据结构,将树转化为二叉树的基本目的是 ①。

答:①树可采用二叉树的存储结构并利用二叉树的已有算法解决树的有关问题

4. 一棵二叉树的结点数据采用顺序存储结构,存储于数组 t 中,如图 8.13 所示,则该二叉树的链接表示形式为 ①。

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
e	a	f		d		g			c	j			l	h	i					b

图 8.13 一棵二叉树的顺序存储数组 t

答:①如图 8.14 所示。

5. 深度为 k 的完全二叉树至少有 ① 个结点。至多有 ② 个结点,若按自上而下,从左到右次序给结点编号(从 1 开始),则编号最小的叶子结点的编号是 ③。

答:① 2^{k-1} ② 2^k-1 ③ $2^{k-2}+1$

6. 在一棵二叉树中,度为零的结点的个数为 n_0 ,度为 2 的结点的个数为 n_2 ,则有 $n_0 =$ ①。

答:① n_2+1

7. 一棵二叉树的第 i($i \geq 1$)层最多有 ① 个结点;一棵有 n($n > 0$)个结点的满二叉树共有 ② 个叶子和 ③ 个非终端结点。

答:① 2^{i-1} ② $2^{\lceil \log_2 n \rceil}$ ③ $2^{\lceil \log_2 n \rceil} - 1$

8. 结点最少的树为 ①,结点最少的二叉树为 ②。

答:①只有一个结点的树 ②空的二叉树

9. 现有按中序遍历二叉树的结果为 abc,问有 ① 种不同形态的二叉树可以得到这一遍历结果,这些二叉树分别是 ②。

答:①5 ②如图 8.15 所示

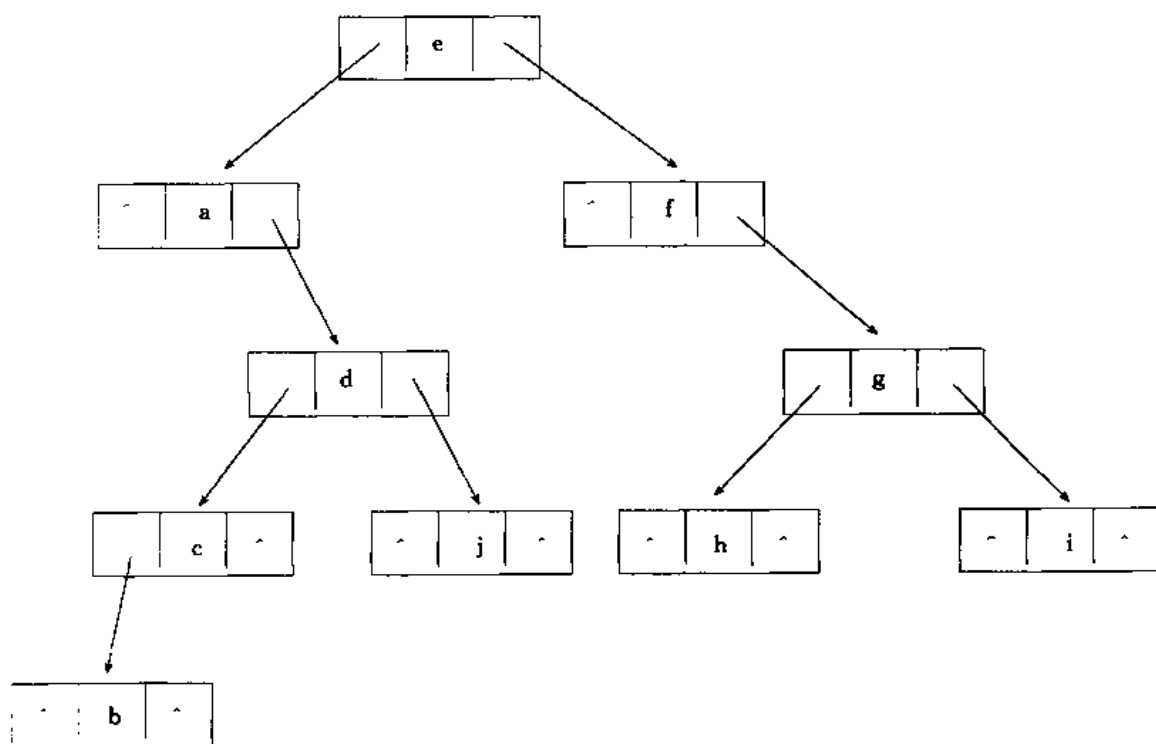


图 8.14 二叉树的链接表示

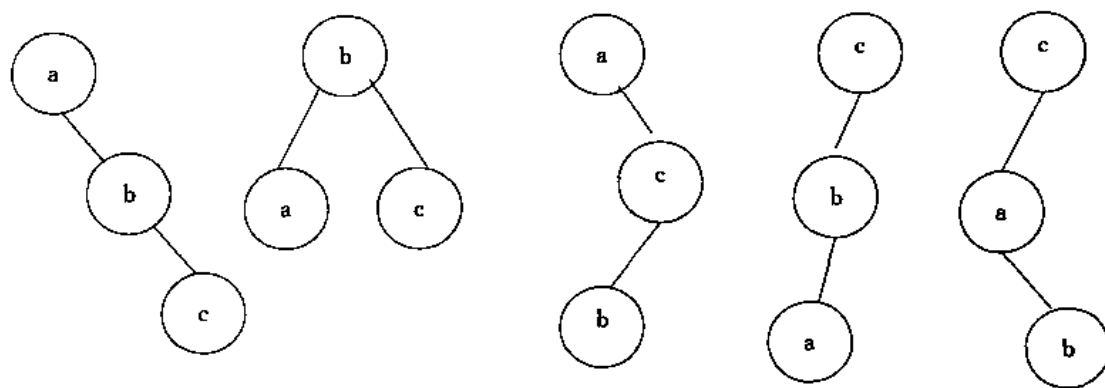


图 8.15 不同形态的二叉树

10. 根据二叉树的定义,具有三个结点的二叉树有 ① 种不同的形态,它们分别是 ②。

答:①5 ②如图 8.16 所示

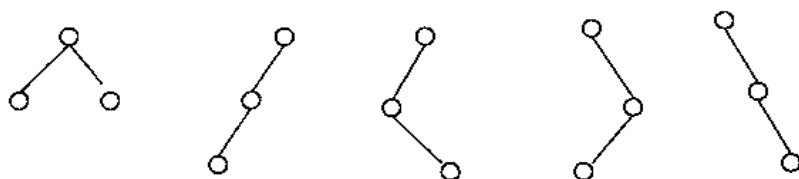


图 8.16 具有三个结点的二叉树

11. 由如图 8.17 所示的二叉树, 回答以下问题:

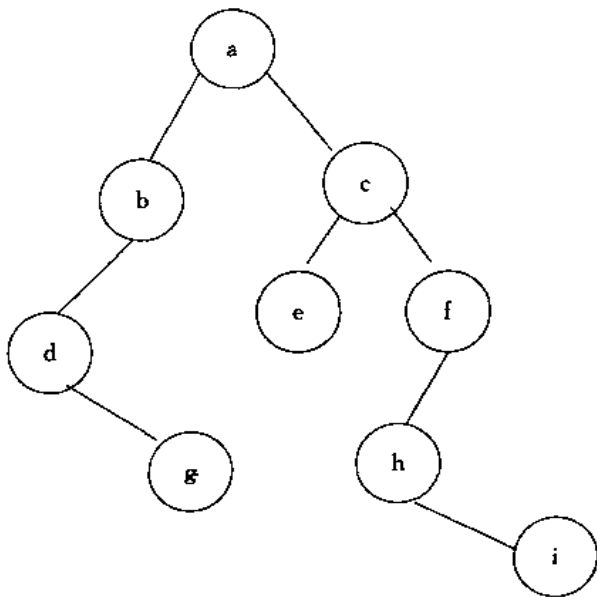


图 8.17 一棵二叉树

- (1) 其中序遍历序列为 ①;
- (2) 其前序遍历序列为 ②;
- (3) 其后序遍历序列为 ③;
- (4) 该二叉树的中序线索二叉树为 ④;
- (5) 该二叉树的后序线索二叉树为 ⑤;
- (6) 该二叉树对应的森林是 ⑥。

答: ① djbaechif

② abdjcefhi

③ jdbeihfca

④ 如图 8.18(a)所示

⑤ 如图 8.18(b)所示

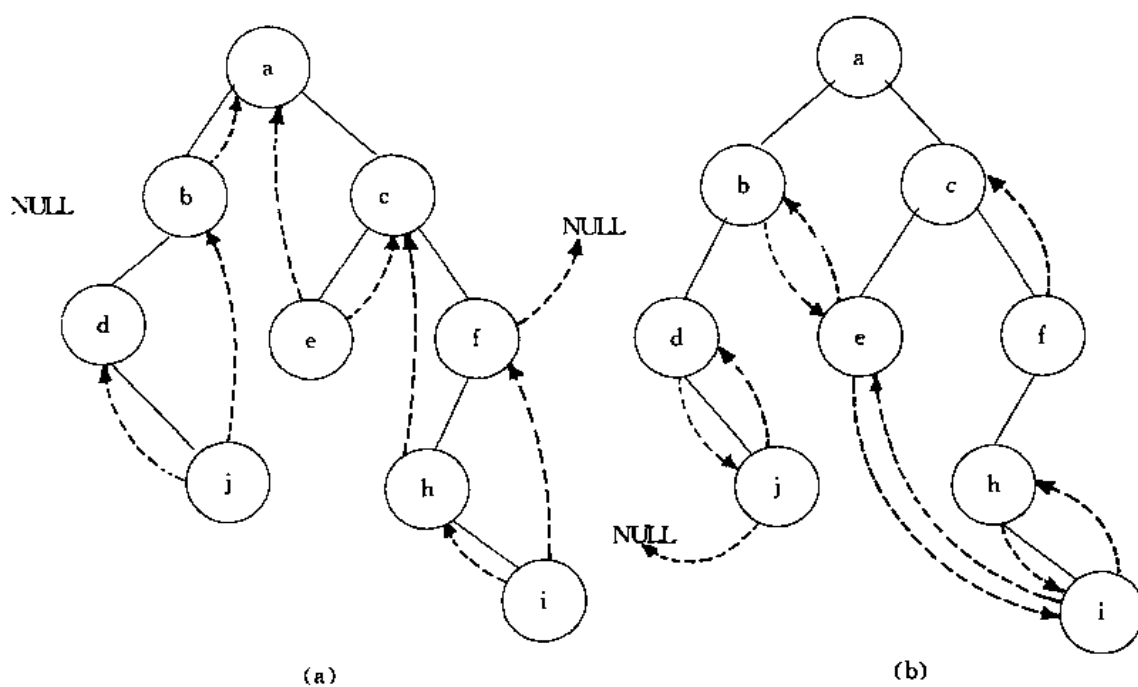


图 8.18 中序和后序线索树

⑥ 如图 8.19 所示

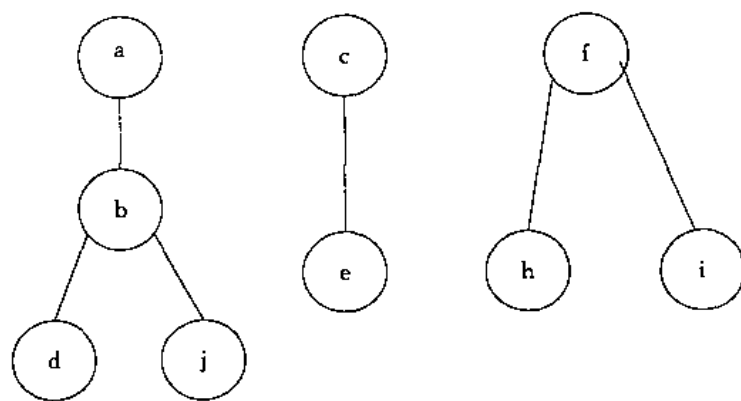


图 8.19 对应的森林

12. 已知一棵树如图 8.20 所示,其孩子兄弟表示为 ①。

答:①如图 8.21 所示

13. 以数据集{4,5,6,7,10,12,18}为结点权值所构造的 Huffman 树为 ①,其带权路径长度为 ②。

答:①如图 8.22 所示 ②165

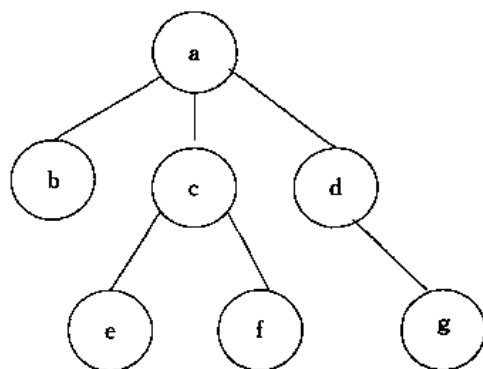


图 8.20 一棵树

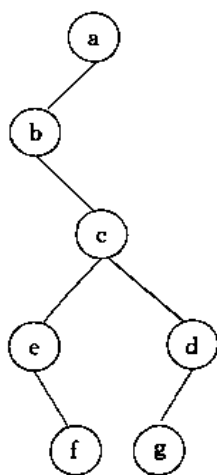


图 8.21 一棵树的孩子兄弟表示

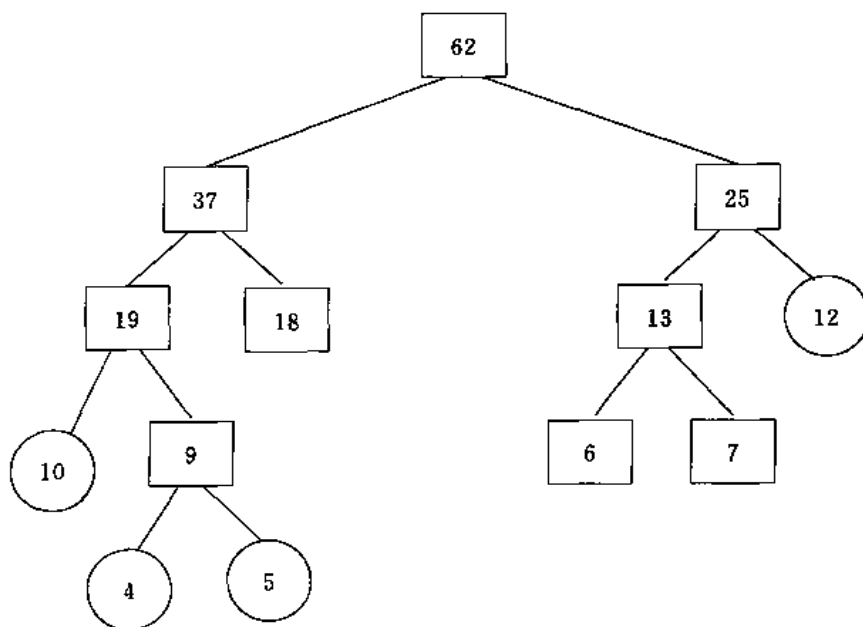


图 8.22 Huffman 树

8.3 习题解析

1. 已知一棵树边的集合为 $\{ \langle i, m \rangle, \langle i, n \rangle, \langle e, i \rangle, \langle b, e \rangle, \langle b, d \rangle, \langle a, b \rangle, \langle g, j \rangle, \langle g, k \rangle, \langle c, g \rangle, \langle c, f \rangle, \langle h, l \rangle, \langle c, h \rangle, \langle a, c \rangle \}$, 画出这棵树, 并回答下列问题:

- (1) 哪个是根结点?
- (2) 哪个是叶子结点?
- (3) 哪个是结点 g 的双亲?
- (4) 哪些是结点 g 的祖先?
- (5) 哪些是结点 g 的孩子?
- (6) 哪些是结点 e 的子孙?
- (7) 哪些是结点 e 的兄弟? 哪些是结点 f 的兄弟?
- (8) 结点 b 和 n 的层次号分别是什么?
- (9) 树的深度是多少?
- (10) 以结点 c 为根的子树的深度是多少?
- (11) 树的度数是多少?

解: 依题意, 树的表示如图 8.23 所示。

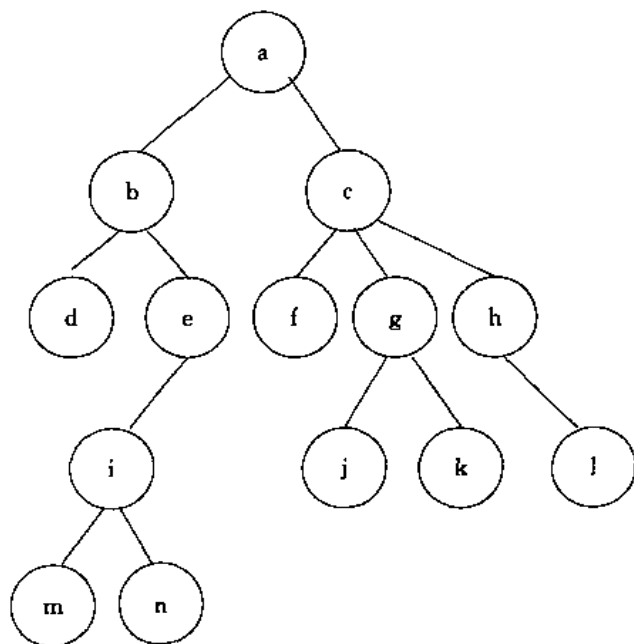


图 8.23 一棵树

- (1) 根结点是: a
- (2) 叶子结点是: d, m, n, f, j, k, l
- (3) g 的双亲是: c
- (4) g 的祖先是: a, c

- (5) g 的孩子是:j,k
 (6) e 的子孙是:i,m,n
 (7) e 的兄弟是 d,f 的兄弟是 g,h
 (8) b 的层次是 2,n 的层次是 5
 (9) 树的深度是 5
 (10) 以结点 c 为根的子树的深度是 3
 (11) 树的度数是 3
 2. 设二叉树 b_t 的存储结构如下:

	1	2	3	4	5	6	7	8	9	10
left	0	0	2	3	7	5	8	0	10	1
data	j	h	f	d	b	a	c	e	g	i
right	0	0	0	9	4	0	0	0	0	0

其中 b_t 为树根结点指针, left、right 分别为结点的左、右孩子指针域, data 为结点的数据域。请完成下列各题:

- (1) 画出二叉树 b_t 的逻辑结构;
 (2) 写出按先序、中序和后序遍历二叉树 b_t 所得到的结点序列;
 (3) 画出二叉树 b_t 的后线索化树。

解: (1) 二叉树 b_t 的逻辑结构如图 8.24 所示。

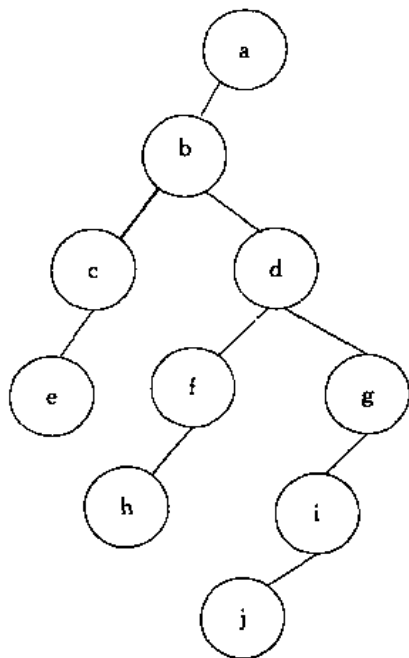


图 8.24 二叉树 b_t 的逻辑结构

- (2) 先序遍历: abcdefhgij;

中序遍历:ecbhfdjiga;

后序遍历:echfjigdba。

(3)二叉树 bt 的后线索化树如图 8.25 所示。

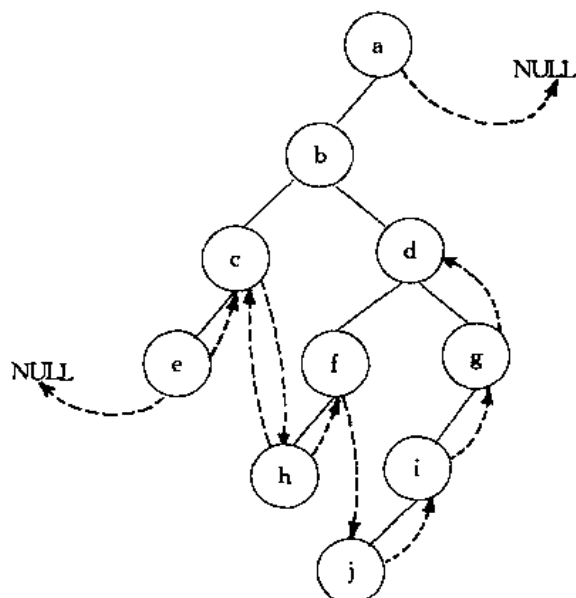


图 8.25 二叉树 bt 的后线索化树

3. (1)对如图 8.26 所示的树,画出采用链接存储结构示意图。这里假设每个结点的形式为:

tag	data	link
-----	------	------

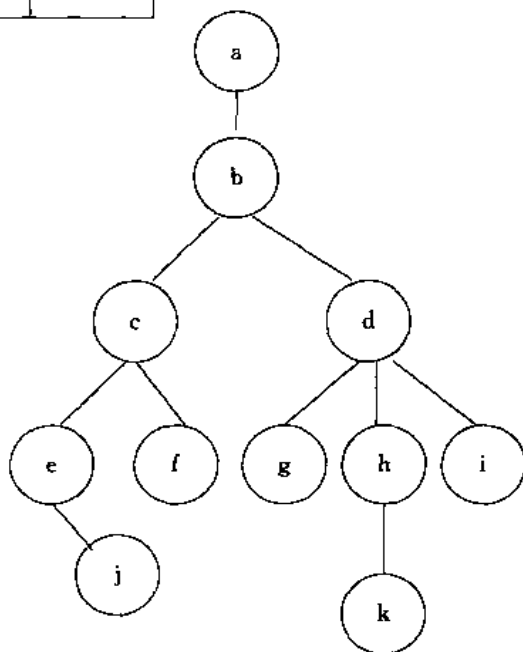


图 8.26 一棵树

其中:tag 为标志域,0 表示是原子结点,1 表示是子树结点;data 为数据域(tag=0)或左子树指针域(tag=1);link 为右子树域。

(2)画出该树的二叉树形式,并说明转换规则。

解:(1)链接存储结构如图 8.27 所示。

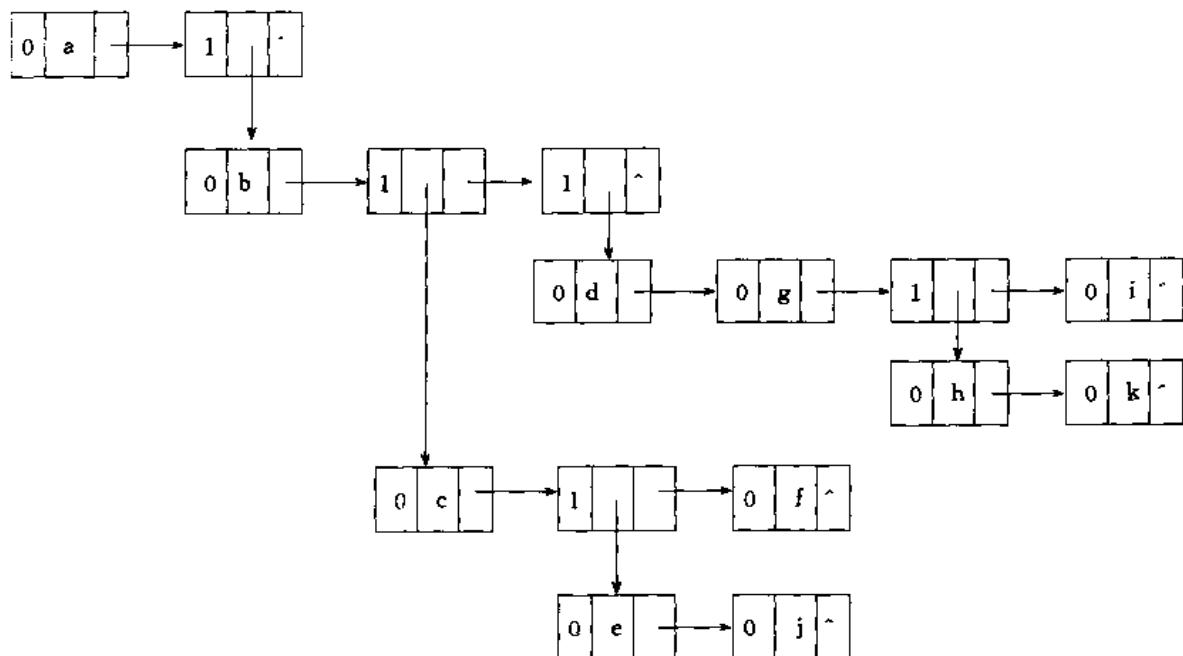


图 8.27 树的二叉树链接存储结构

(2)该树的二叉树形式如图 8.28 所示。树到二叉树的转换规则如下:

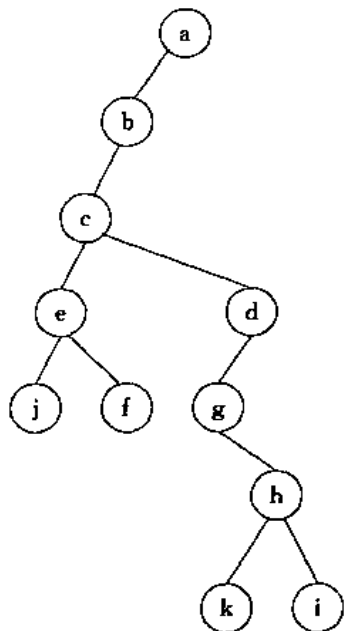


图 8.28 树的二叉树形式

- 一般树的根结点为二叉树的根结点；
- 每个结点的第一个子结点(最左的子树)作为该结点的左孩子；
- 所有具有兄弟关系的结点用指针链接起来。

4. 二叉树结点数值采用顺序存储结构,如图 8.29 所示。

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
a	e	a	f		d		g			c	j			h	i					b

图 8.29 顺序存储结构的二叉树

- (1)画出二叉树表示；
- (2)写出前序遍历,中序遍历和后序遍历的结果；
- (3)写出结点值 c 的父结点,其左、右孩子；
- (4)画出把此二叉树还原成森林的图。

解:(1)该二叉树如图 8.30 所示。

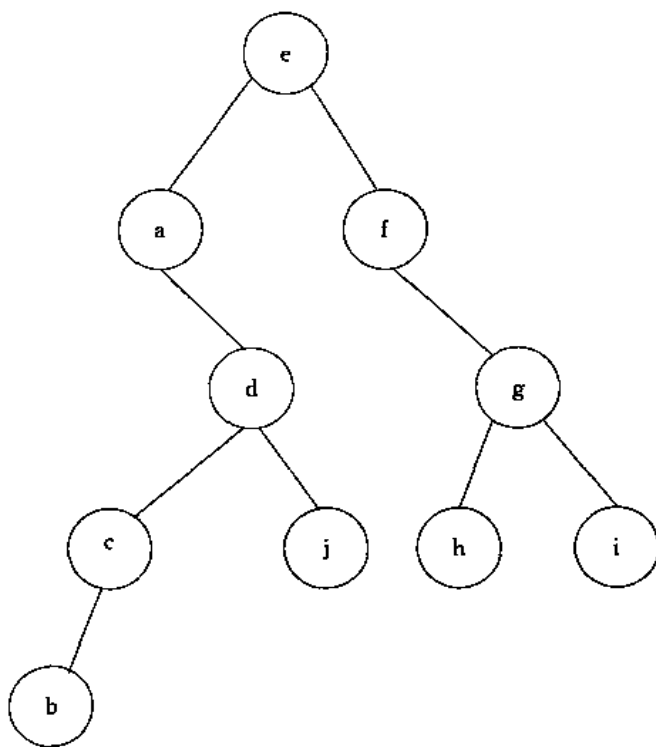


图 8.30 一棵二叉树

(2)本题二叉树的各种遍历结果如下：

前序遍历:eadcbjfhgi

中序遍历:abcdjefhgi

后序遍历:bcjdahigfa

(3)c 的父结点为 d,左孩子为 j,没有右孩子。

(4)还原成的森林如图 8.31 所示。

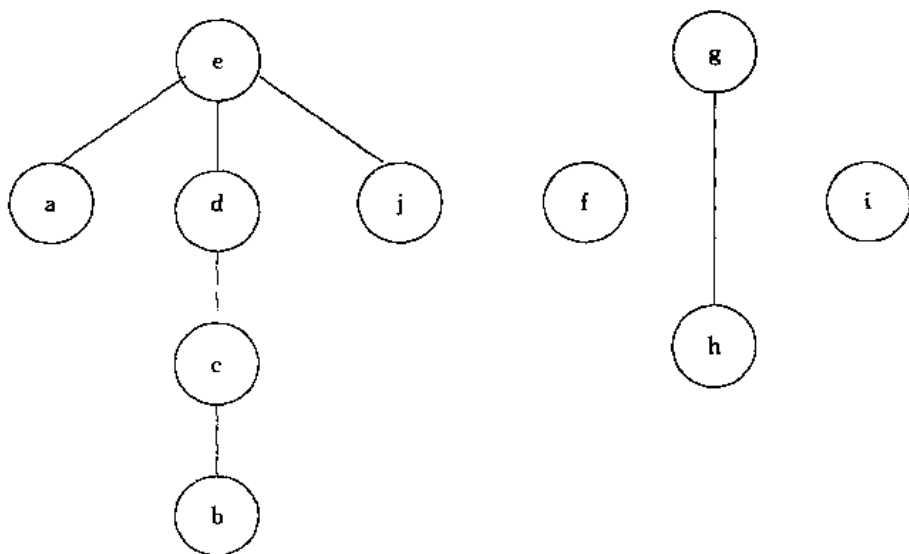


图 8.31 还原成的森林

5. 已知一棵二叉树的中序序列为 $cbedahgijf$, 后序序列为 $cedbhjigfa$, 画出该二叉树的先序线索二叉树。

解: 由后序序列的最后一个结点 a 可推出该二叉树的树根为 a , 由中序序列可推出 a 的左子树由 $cbcd$ 组成, 右子树由 $hgijf$ 组成, 又由 $cbcd$ 在后序序列中的顺序可推出该子树的根结点为 b , 其左子树只有一个结点 c , 右子树由 ed 组成, 显然这里的 e 是根结点, 其右子树为结点 d , 这样可得到根结点 a 的左子树的先序序列为 $bcde$; 再依次推出右子树的先序序列为 $fghij$ 。因此该二叉树如图 8.32 所示。

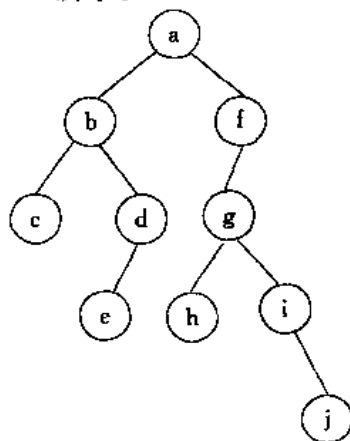


图 8.32 一棵二叉树

设二叉树的先序线索链表如图 8.33 所示。

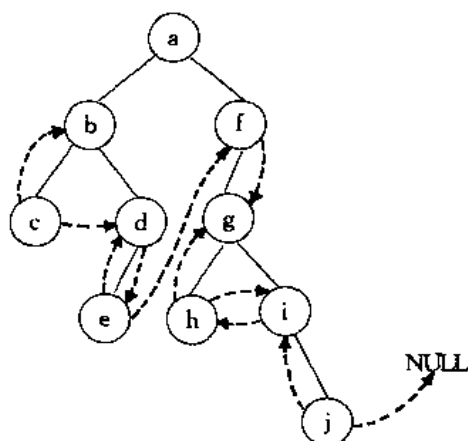


图 8.33 先序线索链表

6. 一棵有 11 个结点的二叉树的存储情况如图 8.34 所示, $\text{left}[i]$ 和 $\text{right}[i]$ 分别为 i 结点左右孩子, 根结点为序号 3 的结点。画出该二叉树并给出前序、中序和后序遍历该树的结点序列。

1	2	3	4	5	6	7	8	9	10	11	
6	-	7	-	8	-	5	-	2	-	-	$\text{left}[i]$
m	f	a	k	b	l	c	r	d	s	e	$\text{data}[i]$
-	-	9	-	10	4	11	-	j	-	-	$\text{right}[i]$

图 8.34 二叉树的存储情况

解: 该二叉树的表示如图 8.35 所示。其各种遍历结果如下:

前序遍历: acbrsedfmlk

中序遍历: rbsceafdlkm

后序遍历: rsbecfklmda

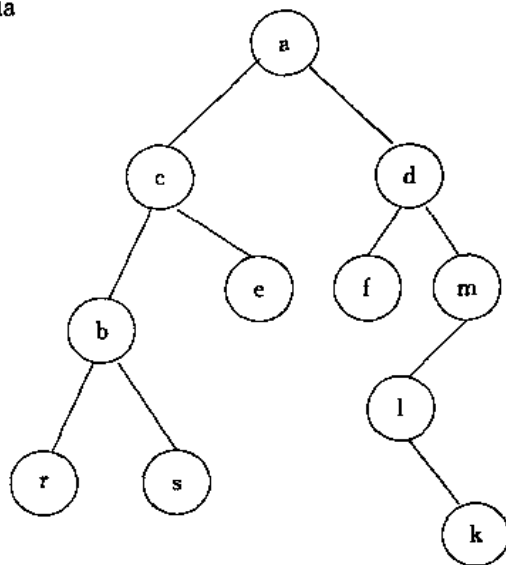


图 8.35 二叉树

7. 设数据集合 $d = \{1, 12, 5, 8, 3, 10, 7, 13, 9\}$, 试完成下列各题:

- (1) 依次取 d 中各数据, 构造一棵二叉排序树 b_t ;
- (2) 如何依据此二叉树 b_t 得到 d 的一个有序序列?
- (3) 画出在二叉树 b_t 中删除“12”后的树结构。

解: (1) 本题产生的二叉排序树如图 8.36 所示。

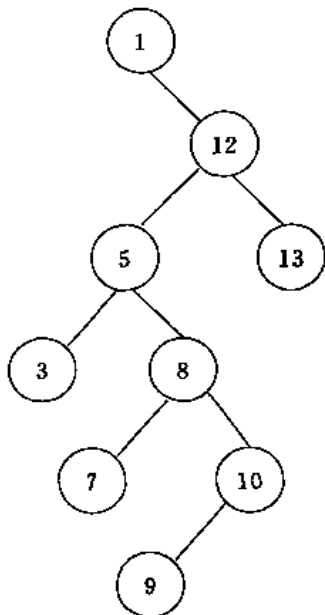


图 8.36 二叉排序树

(2) d 的有序序列为 b_t 的中序遍历次序, 即: 1, 3, 5, 7, 8, 9, 10, 12, 13

(3) 删除“12”后的树结构如图 8.37 所示。

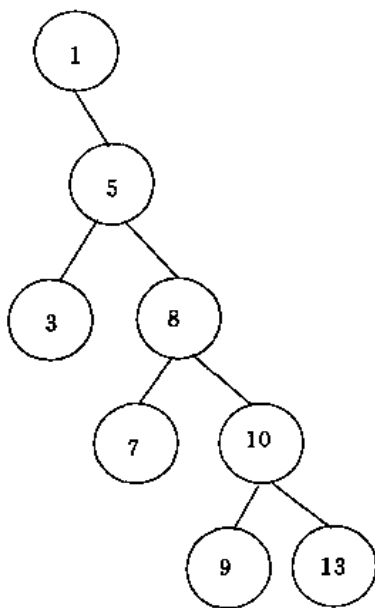


图 8.37 删除“12”后的树结构

8. 输入一个正整数序列{40,28,6,72,100,3,54,1,80,91,38},建立一棵二叉排序树,然后删除结点 72,分别画出该二叉树及删除结点 72 后的二叉树。

解:本题的二叉排序树如图 8.38 所示。删除 72 之后的二叉排序树如图 8.39 所示。

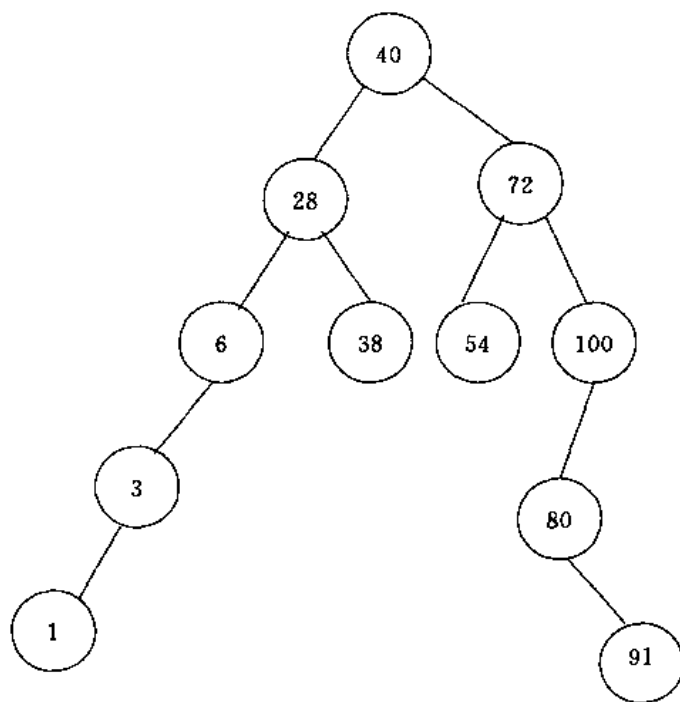


图 8.38 二叉排序树

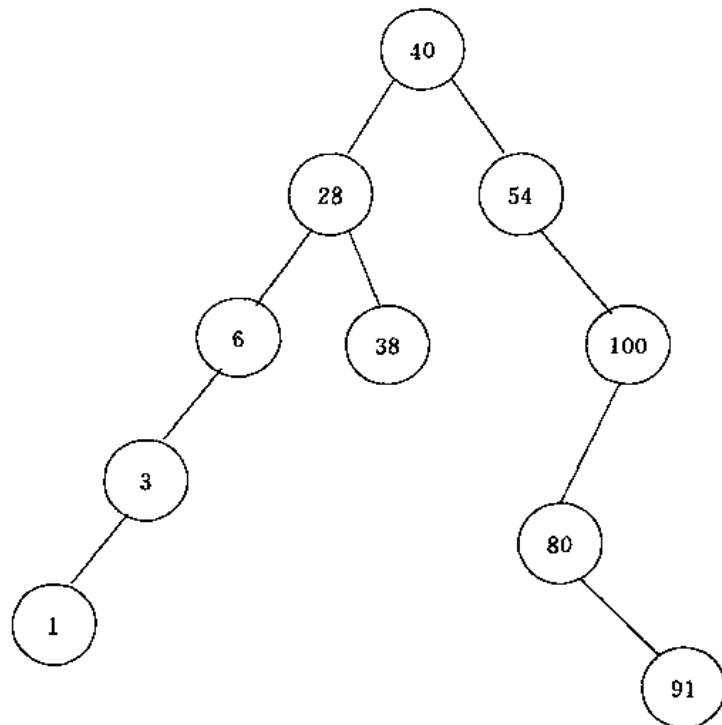


图 8.39 删除“72”之后的二叉排序树

* 9. 已知一棵二叉排序树,其结构如图 8.40(a)所示。

画出依次删除关键字为 $a_1=13, a_2=12, a_3=4, a_4=8$ 的各个结点后,该二叉排序树的结构。

解:删除关键字为 $a_1=13, a_2=12, a_3=4, a_4=8$ 的各个结点时,该二叉排序树的变化情况如图 8.40 所示。

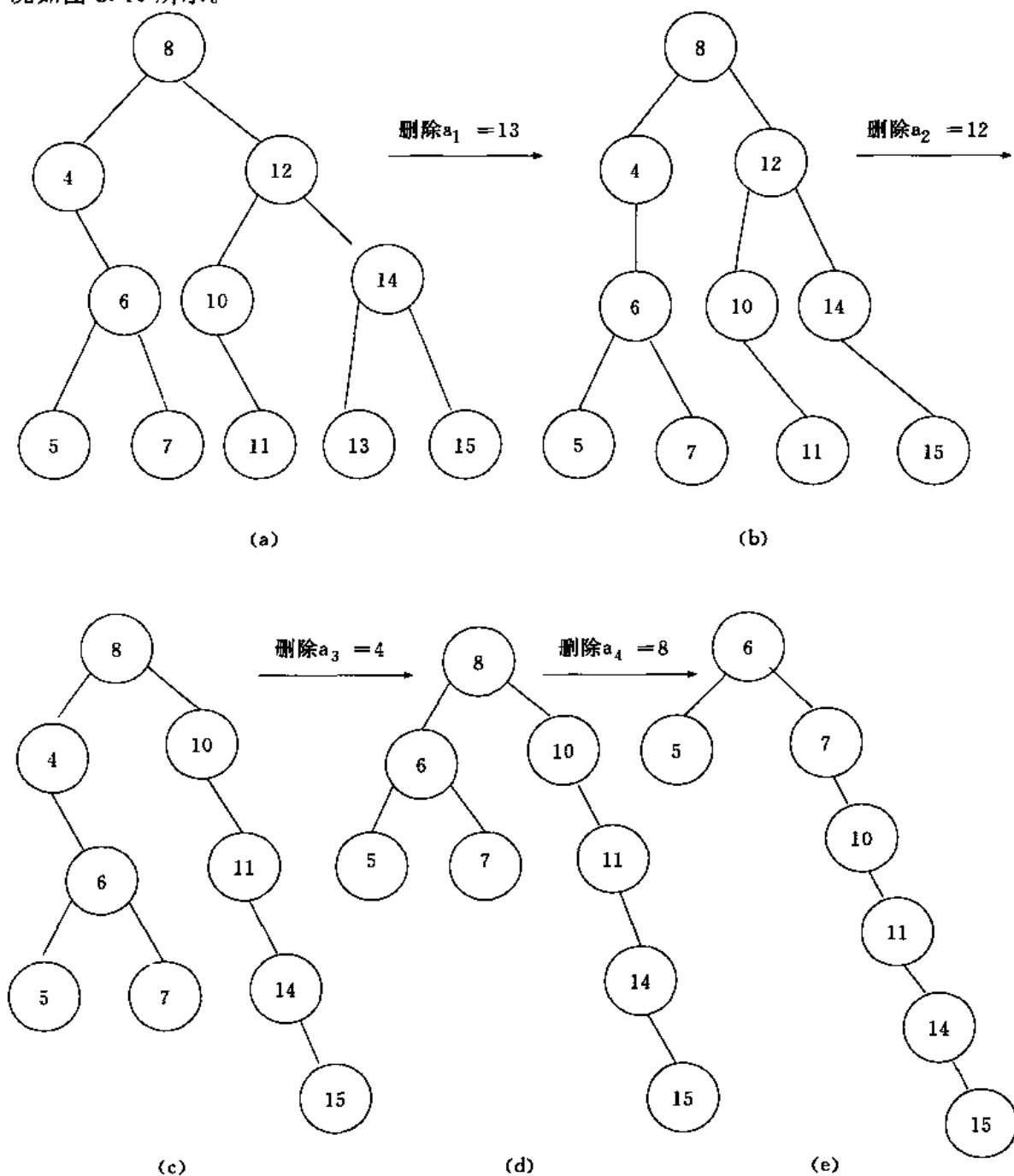


图 8.40 删除各关键字的结点时的变化情况

* 10. 对给定的数列 $R=\{7,16,4,8,20,9,6,18,5\}$,构造一棵二叉排序树,并且:
按中序遍历得出一个新数列 R_1 ;

按后序遍历得到一个新数列 R2:

解:本题产生的二叉排序树如图 8.41 所示。

按中序遍历得出一个新数列 R1 为:4,5,6,7,8,9,16,18,20。

按后序遍历得到一个新数列 R2 为:5,6,4,9,8,18,20,16,7。

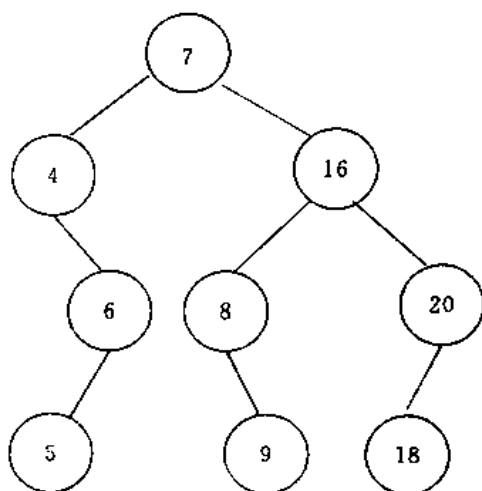


图 8.41 一棵二叉排序树

11. 有一份电文中共使用 5 个字符:a、b、c、d、e,它们的出现频率依次为 4、7、5、2、9,试画出对应的 Huffman 树(请按左子树根结点的权小于等于右子树根结点的权的次序构造),并求出每个字符的 Huffman 编码。

解:依题意,本题对应的 Huffman 树如图 8.42 所示。各字符对应的 Huffman 编码如下:

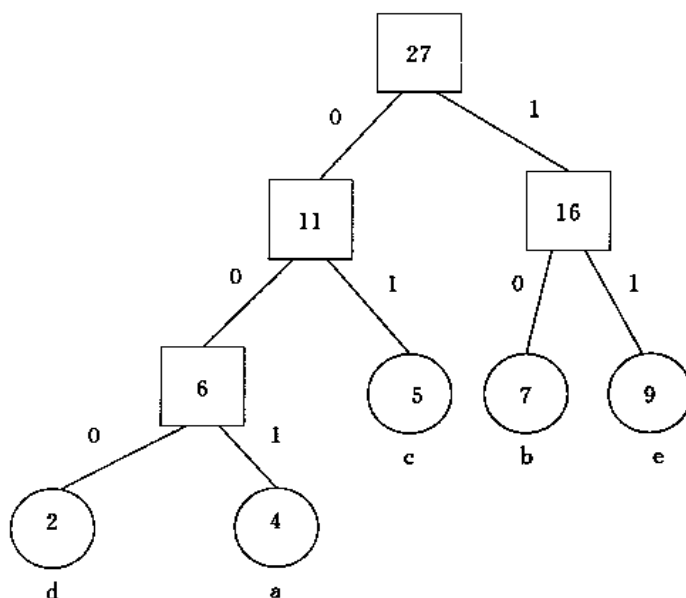


图 8.42 Huffman 树

a:001

b:10

c:01

d:000

e:11

12. 设给定权集 $w = \{2, 3, 4, 7, 8, 9\}$, 试构造关于 w 的一棵哈夫曼树, 并求其加权路径长度 WPL。

解: 本题的哈夫曼树如图 8.43 所示。

其加权路径长度 $WPL = 7 \times 2 + 8 \times 2 + 4 \times 3 + 2 \times 4 + 3 \times 4 + 9 \times 2 = 80$

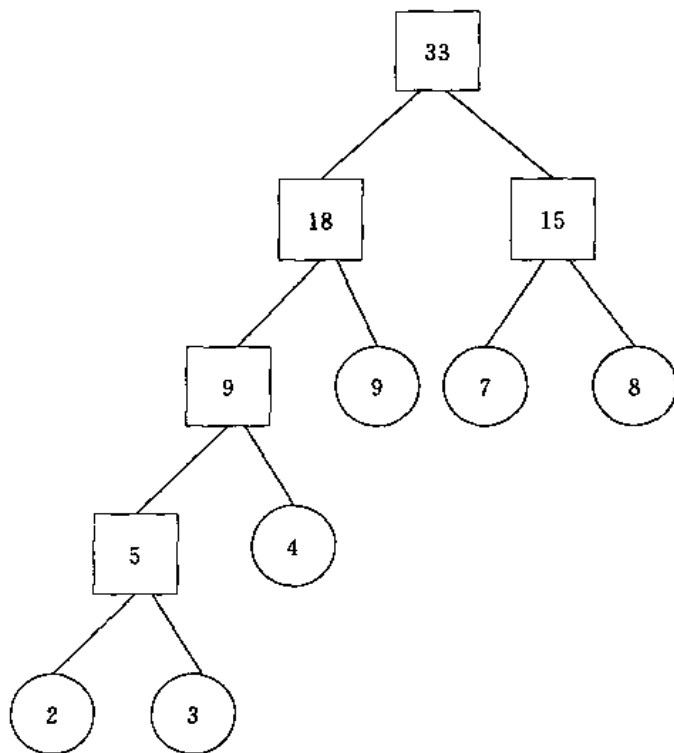


图 8.43 一棵哈夫曼树

13. 试找出分别满足下面条件的所有二叉树:

- (1) 前序序列和中序序列相同;
- (2) 中序序列和后序序列相同;
- (3) 前序序列和后序序列相同;

解: (1) 前序序列和中序序列相同的二叉树为空树或任一结点均无左孩子的非空二叉树;

(2) 中序序列和后序序列相同的二叉树为空树或任一结点均无右孩子的非空二叉树;

(3) 前序序列和后序序列相同的二叉树为空树或仅有一个结点的二叉树;

* 14. 一个 h 层满 k 叉树具有如下性质:

第 h 层上的结点都是叶子结点, 其余各层上的每个结点都有 h 个非空子结点。

若对树中的结点逐层编号(层号由小到大,同一层中从左到右),编码次序是1,2,3,...,试回答如下问题:

- (1)各层上结点的数目是多少?
- (2)编号为 n 的结点的父亲结点的编号是什么?
- (3)编号为 n 的结点的第 i 个儿子的编号是什么?
- (4)编号为 n 的结点的右兄弟的编号是什么?

解:(1)第 i 层的结点数为 k^{i-1} 。

(2)编号为 n 的结点的双亲点为: $\lceil (n-2)/k \rceil + 1$ 。

(3)编号为 n 的结点的第 i 个孩子结点为: $(n-1) * k + i + 1$ 。

(4)编号为 n 的结点有右兄弟的条件是 $(n-1) \% k \neq 0$,其右兄弟的编号是 $n+1$ 。

* 15. 表 8.1 中 m, n 分别是一棵二叉树中两个结点,行号 $i=1,2,3,4$ 分别表示四种 m, n 的相对关系,列号 $j=1,2,3$ 分别表示在前序,中序,后序遍历中 m, n 之间的先后次序关系,要求在 i, j 所表示的关系能够同时发生的方格内打上“√”。

表 8.1 m, n 结点关系表

	前序遍历 n 先被访问	中序遍历 n 先被访问	后序遍历 n 先被访问
n 在 m 的左边			
n 在 m 的右边			
n 是 m 的祖先			
n 在 m 的儿子			

解:依题意, m, n 结点之间打上“√”的行列坐标有(1,1)、(1,2)、(1,3)、(3,1)、(3,2)、(4,2)、(4,3)。

16. 已知一棵度为 m 的树中有 n_1 个度为 1 的结点, n_2 个度为 2 的结点, ... n_m 个度为 m 的结点,问该树中有多少个叶子结点?

解:依题意:设 n 为总的结点个数, n_0 为叶子结点(即度为 0 的结点)的个数,则有:

$$n = n_0 + n_1 + n_2 + \dots + n_m \quad ①$$

又有: $n-1 =$ 度的总数,即:

$$n-1 = n_1 * 1 + n_2 * 2 + \dots + n_m * m \quad ②$$

①-②式得:

$$1 = n_0 - n_2 - 2n_3 - \dots - (m-1)n_m$$

则有: $n_0 = 1 + n_2 + 2n_3 + \dots + (m-1)n_m$

$$= 1 + (i-1)n_i$$

17. 任意一个有 n 个结点的二叉树, 已知它有 m 个叶子结点, 试证明非叶子结点有 $(m-1)$ 个度数为 2, 其余度数为 1。

证明: 依题意设 a 为二叉树中度为 1 的结点数, b 为度 2 的结点数, 则总的结点数为:

$$n = a + b + m \quad ①$$

再看二叉树中分支数, 除根结点外, 其余结点都有一个分支进入, 设 B 为分支数, 则有:

$$n = B + 1$$

由于这些分支是由度为 1 和 2 的结点射出的, 所以又有:

$$B = a + 2b$$

代入上式得:

$$n = a + 2b + 1 \quad ②$$

由①②两式得到:

$$a + b + m = a + 2b + 1$$

所以 $b = m - 1$, 证毕。

18. 假设二叉树中所有非叶子结点都有左、右子树, 则对这种二叉树:

(1) 试问: 有 n 个叶子结点的树中共有多少个结点?

(2) 试证明 $\sum_{i=1}^n 2^{-(l_i-1)} = 1$ 其中: n 为叶子结点的个数, l_i 表示第 i 个叶子结点所在的层次数 (设根结点所在的层次数为 1)。

解: (1) 依题意, 这种二叉树中没有度为 1 的结点, 度为 2 的结点数 n_2 和度为 0 的结点数 n_0 之间满足关系:

$$n_2 = n_0 - 1$$

所以, 总的结点数 $= n_2 + n_0 = n_0 - 1 + n_0 = 2n_0 - 1 = 2n - 1$

(2) 证明: 采用归纳法

当 $n=1$ 时, $l_1=1$, 则 $\sum_{i=1}^1 2^{-(l_i-1)} = 2^0 = 1$, 只有一个根结点, 等式成立。

假设 $n \leq m-1$ 时有 $\sum_{i=1}^{m-1} 2^{-(l_i-1)} = 1$, 只需证明 $n=m$ 时该等式成立即可。

当 $n=m$ 时, 假设在有 $m-1$ 个叶子结点的二叉树的 h_i 层上的叶子结点上加上两个儿子结点, 则总的叶子结点个数增加 1 个, 这样构成一个有 m 个叶子结点的二叉树。

由于在该 m 个叶子结点的二叉树中所有叶子结点的 $2^{-(l_i-1)}$ 之和等于原来 $m-1$ 个叶子结点的二叉树中所有叶子结点的 $2^{-(l_i-1)}$ 之和减去添加两个儿子的叶子结点的 $2^{-(h_i-1)}$ 加上该两个儿子的 $2 + 2^{-(h_i+1-1)}$ 。

$$\text{所以: } \sum_{i=1}^m 2^{-(l_i-1)} = \sum_{i=1}^{m-1} 2^{-(l_i-1)} - 2^{-(h_i-1)} + 2 * 2^{-(h_i+1-1)} = \sum_{i=1}^{m-1} 2^{-(l_i-1)} = 1$$

所以结论成立, 证毕。

* 19. 假设二叉树采用链接存储方式存储, 编写对二叉树进行前序遍历的非递归算法,

并对该算法执行如图 8.44 所示的二叉树时的情况进行跟踪(即给出各阶段栈的内容及输出结点序列)。

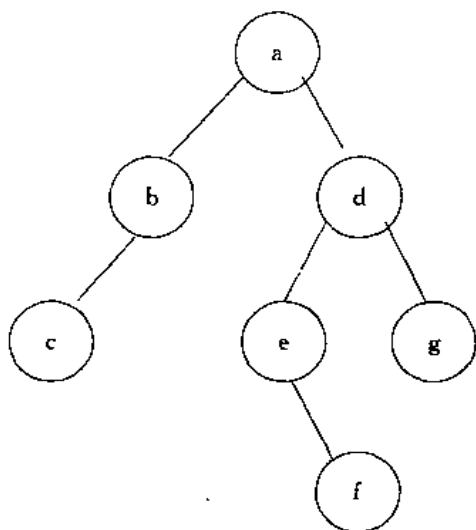


图 8.44 一棵二叉树

解:依题意,使用一个栈 stack 实现非递归的前序遍历。

```

void preorder(b)
btree * b;
{
    btree * stack[m0];
    int top;
    if (b! =NULL)
    {
        top=1; /* 根结点入栈 */
        stack[top]=b;
        while (top>0) /* 栈不为空时循环 */
        {
            p=stack[top]; /* 退栈并访问该结点 */
            top--;
            printf("%d ",p->data);
            if (p->right! =NULL) /* 右孩子入栈 */
            {
                top++;
                stack[top]=p->right;
            }
            if (p->left! =NULL) /* 左孩子入栈 */
            {
                top++;
                stack[top]=p->left;
            }
        }
    }
}
  
```

```

}
}

```

跟踪图 8.44 的二叉树时的情况如图 8.45 所示。

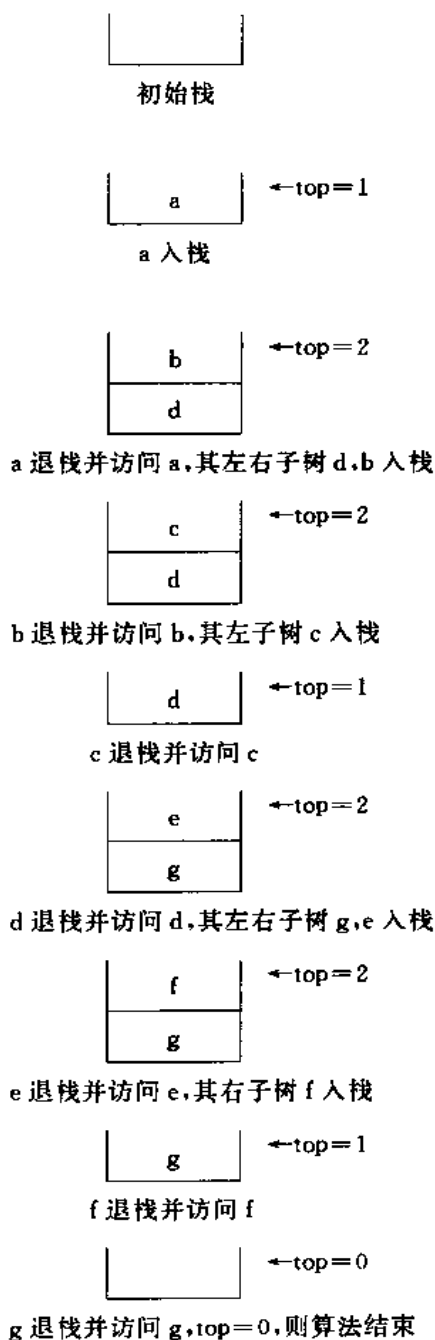


图 8.45 跟踪二叉树时的情况

20. 假设二叉树采用链接存储方式存储,编写一个中序遍历二叉树的非递归函数。

解:根据中序遍历二叉树的递归定义,转换成非递归函数时用一个栈保存返回的结点,先扫描根结点的所有左结点并入栈,出栈一个结点,访问之,然后扫描该结点的右结点并入栈再扫描该右结点的所有左结点并入栈,如此这样,直到栈空为止。

因此,实现本题功能的函数如下:

```
void inorder(btree *b)
{
    btree *stack[m0], *p;
    int top=0;
    p=b;
    do
    {
        while (p!=NULL)                /* 扫描左结点 */
        {
            top++;
            stack[top]=p;
            p=p->left;
        }
        if (top>0)
        {
            p=stack[top];                /* p 所指结点为无左子树的结点或其左子树已遍历过 */
            top--;
            printf("%d ",p->data);        /* 访问结点 */
            p=p->right;                  /* 扫描右子树 */
        }
    } while (p!=NULL && top!=0)
}
```

21. 假设二叉树采用链接存储方式存储,编写一个后序遍历二叉树的非递归函数。

解:根据后序遍历二叉树的递归定义,转换成非递归函数时采用一个栈保存返回的结点,先扫描根结点的所有左结点并入栈,出栈一个结点,然后扫描该结点的右结点并入栈,再扫描该右结点的所有左结点并入栈,当一个结点的左右子树均访问后再访问该结点,如此这样,直到栈空为止。在访问根结点的右子树后,当指针 p 指向右子树树根时,必须记下根结点的位置,以便在遍历右子树之后正确返回,这就产生了一个问题,在退栈回到根结点时如何区别是从左子树返回还是从右子树返回。这里采用两个栈 stack 和 tag,并用一个共同的栈顶指针,一个存放指针值,一个存放左右子树标志(0 为左子树,1 为右子树)。退栈时在退出结点指针的同时区分是遍历左子树返回的还是遍历右子树返回的,以决定下一步是继续遍历右子树还是访问根结点。

因此,实现本题功能的函数如下:

```
void postorder(btree *b)
{
    btree *stack[m0], *p;
    int tag[m0],top=0;
    p=b;
    do
```

```

{
    while (p!=NULL)                /* 扫描左结点 */
    {
        top++;
        stack[top]=p;
        tag[top]=0;
        p=p->left;
    }
    if (top>0)
    {
        p=stack[top]; /* p所指结点为无左子树的结点或其左子树已遍历过 */
        if (tag[top]==1)
        {
            /* p的左右子树都访问过 */
            top--;
            printf("%d ",p->data);    /* 访问结点 */
        }
        if (top>0)
        {
            p=p->right;              /* 扫描右子树 */
            tag[top]=1;              /* 表示当前结点的右子树已访问过 */
        }
    }
} while (p!=NULL && top!=0)
}

```

22. 假设二叉树采用链接存储结构进行存储,root 指向根结点,p 所指结点为任一给定的结点,编写一个求出从根结点到 p 所指结点之间路径的函数。

解:本题采用非递归后序遍历树 root,当后序遍历访问到 p 所指结点时,此时 stack 中所有结点均为 p 所指结点的祖先,由这些祖先便构成了一条从根结点到 p 所指结点之间的路径。

因此,实现本题功能的函数如下:

```

void path(root,p)
btree *root,*p;
{
    btree *stack[m0]; *s;
    int tag[m0],top=0,i,find=0;
    s=root;
    do
    {
        while (s!=NULL)            /* 扫描左结点 */
        {
            top++;

```

```

    stack[top]=s;
    tag[top]=0;
    s=s->left;
}
if (top>0)
{
    s=stack[top]; /* p 所指结点为无左子树的结点或其左子树已遍历过 */
    if (tag[top]==1)
    {
        /* p 的左右子树都访问过 */
        if (s==p) /* 找到 p 所指结点,则显示从根结点到 p 所指结点之间的路径 */
        {
            for (i=1;i<=top;i++) printf("%d ",stack[i]->data);
            find=1;
        }
        else top--; /* 访问结点,不必显示 */
    }
    if (top>0 && ! find)
    {
        p=p->right; /* 扫描右子树 */
        tag[top]=1; /* 表示当前结点的右子树已访问过 */
    }
}
while (find || (s!=NULL && top!=0));
}

```

23. 假设二叉树采用链接存储结构进行存储,root 指向根结点,p 所指结点和 q 所指结点为二叉树中的两个结点,编写一个计算它们最近共同祖先 r 所指结点的函数。

解:本题采用非递归后序遍历树 root,不失一般性,假设 p 所指结点在 q 所指结点的左边,当后序遍历访问到 p 所指结点时,此时 stack 中所有结点均为 p 所指结点的祖先,此时将其复制到 anor 中,然后继续后序遍历访问到 q 所指结点,同样此时 stack 中所有结点均为 q 所指结点的祖先,再将其与 anor 中的结点依次(从 top 到 1)比较,找出最近共同祖先。

因此,实现本题功能的函数如下:

```

btree * ancestor(p,q)
btree * p, * q;
{
    btree * stack[m0], * s, * anor[m0], * r;
    int tag[m0],top=0,find=0;
    s=root;
    do
    {
        while (s!=NULL) /* 扫描左结点 */
        {

```

```

    top++;
    stack[top]=s;
    tag[top]=0;
    s=s->left;
}
if (top>0)
{
    s=stack[top]; /* p 所指结点为无左子树的结点或其左子树已遍历过 */
    if (tag[top]==1)
    {
        /* p 的左右子树都访问过 */
        top--; /* 访问结点,不必显示 */
        if (s==p) /* 找到 p 所指结点,则将其祖先复制到 anor 中 */
            for (top1=1; top1<=top; top1++) anor[top1]=stack[top1];
        if (s==q) /* 找到 q 所指结点,则比较找出最近共同祖先 */
        {
            i=top;
            while (! find)
            {
                j=top1;
                while (j>0 && stack[j]!=anor[j]) j--;
                if (j>0)
                {
                    find=1;
                    r=anor[j];
                }
                else i--;
            }
        }
    }
    if (top>0 && ! find)
    {
        p=p->right; /* 扫描右子树 */
        tag[top]=1; /* 表示当前结点的右子树已访问过 */
    }
}
} while (find || (s!=NULL && top!=0));
return(r);
}

```

24. 假设二叉树采用链表存储结构,设计一个算法求二叉树中指定结点的层数,并对如图 8.46 所示的二叉树说明计算其中 p 结点层数的函数。

解:依题意:采用递归函数,设 h 返回 p 所指结点的高度,初值为 -1,树为空时返回 0, lh 指示树 b 的高度,其初值为 1,得算法如下:

```

void level(b.p,h,lh)
btree *b,*p;
int h,lh;
/* b:二叉树的指针,p:待找的结点,h:p结点的层数,lh:当前的层数 */
{
    if (b==NULL) h=0;          /* 空树时返回 0 */
    else if (p==b) h=lh;       /* 找到结点 p 时 */
    else
    {
        level(p,b->left,h,lh+1); /* 在左子树中查找 */
        if (h==-1) level(p,b->right,h,lh+1); /* 在右子树中查找 */
    }
}

```

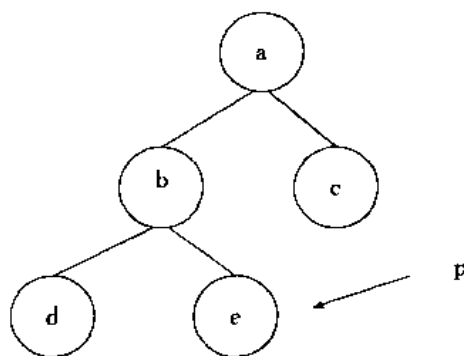


图 8.16 一棵二叉树

在该二叉树中计算 p 结点层数的过程如下：

```

① level('a','e',-1,1)
   ↓
② level('b','e',-1,2)
   ↓
③ level('d','e',-1,3)
   ↓
'd'结点的左、右子树均为空,则回溯到②查找其右子树
   ↓
④ level('e','e',-1,3)
   ↓
h=3 返回到①,因 h 不为-1,故不查找其右子树,本函数执行完毕。

```

* 25. 试设计判断两棵二叉树是否相似的算法,所谓二叉树 t1 和 t2 相似,指的是 t1 和 t2 都是空的二叉树;或者 t1 和 t2 的根结点是相似的,t1 的左子树和 t2 的左子树是相似的且 t1 的右子树与 t2 的右子树是相似的。

解:依题意:本题的递归函数如下:

$$\begin{cases} f(t_1, t_2) = \text{true} & \text{若 } t_1 = t_2 = \text{NULL} \\ f(t_1, t_2) = \text{false} & \text{若 } t_1, t_2 \text{ 之一为 NULL, 另一不为 NULL} \\ f(t_1, t_2) = f(t_1 \rightarrow \text{left}, t_2 \rightarrow \text{left}) \\ \quad \& \& f(t_1 \rightarrow \text{right}, t_2 \rightarrow \text{right}) & \text{若 } t_1, t_2 \text{ 均不为 NULL} \end{cases}$$

因此,实现本题功能的函数如下:

```
int like(b1, b2)
btree * b1, * b2;
{
    int like1, like2;
    if (b1 == NULL && b2 == NULL) return(1);
    else if (b1 == NULL || b2 == NULL) return(0);
    else
    {
        like1 = like(t1->left, t2->left);
        like2 = like(t1->right, t2->right);
        return(like1 && like2);
    }
}
```

26. 假设二叉树 T 中至多有一个结点的数据域值为 x , 试设计一个算法拆去以该结点为根的子树, 使原二叉树分成两棵二叉树, 例如, $x=e$, 二叉树的变化情况如图 8.47 所示。

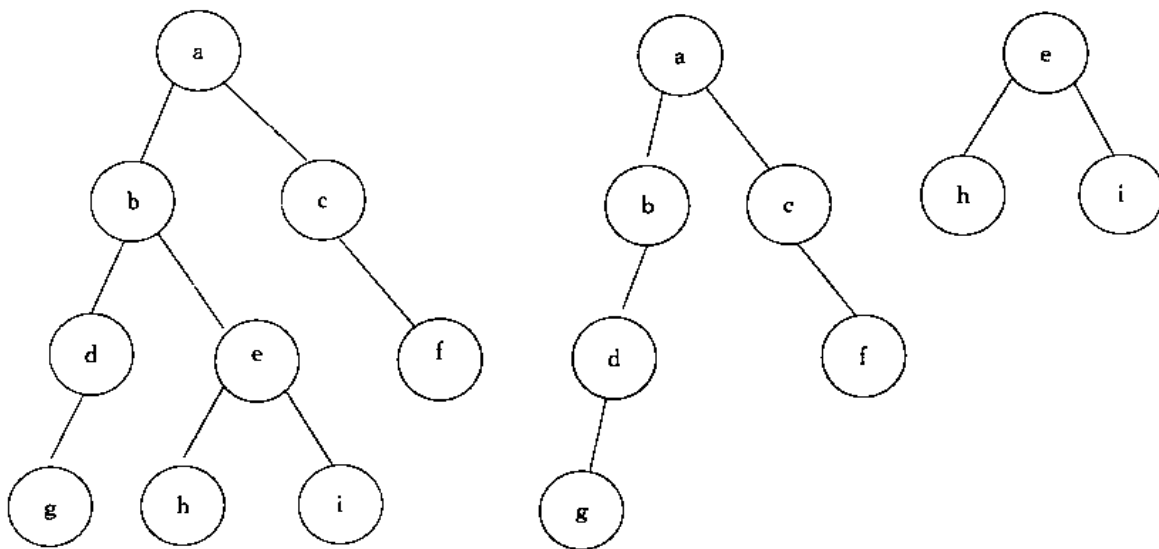


图 8.47 二叉树的分拆过程

解:依题意:本题的算法思想是:先判定 T 的根结点是否是要找的结点,若是,则直接拆开;否则判定在其左子树中进行拆开,若在左子树中未找到,则在右子树中查找。因此,实现本题功能的函数如下:

```
btree * dislink(t, x)
```

```

btree * *t;
int x;
/* t:原二叉树,拆开后的第一棵二叉树;分拆成功后返回第二棵二叉树 */
{
    btree * p, * find;
    if ((*t) != NULL)
    {
        if ((*t)->data == x)                /* 根结点数据域为 x */
        {
            p = *t;
            *t = NULL;
            return(p);
        }
        else
        {
            find = dislink(&(*t)->left, x); /* 在左子树中查找并分解 */
            /* 若左子树中未找到,则在右子树中查找并分解 */
            if (! find) find = dislink(&(*t)->right, x);
            return(find);
        }
    }
    else return(NULL);
}

```

27. 在二叉树中查找值为 x 的结点,试设计打印值为 x 的结点的所有祖先的算法,假设值为 x 的结点不多于 1 个。

解:依题意,本题采用后序遍历的非递归算法,因退栈时需区分其左右子树是否已遍历,因此在结点入栈的同时附带一个标志:0 表示是左子树,1 表示是右子树。用栈 *stack* 保存结点指针及其标志。

因此,实现本题功能的函数如下:

```

void ancestor(btree * t, int x)
{
    struct node
    {
        btree * p;
        int tag;                /* 只取 0 或 1 */
    } s[100];
    int top = 0, i;
    while (1)
    {
        while (t != NULL && t->data != x) /* 将 t 所指结点的所有左结点入栈 */
        {

```

```

    top++;
    s[top].p=t;
    s[top].tag=0;
    t=t->left;
}
if (t!=NULL && t->data==x)    /* 找到了该结点,则打印其祖先 */
{
    printf("结点%d的祖先是:",x);
    for (i=1;i<=top;i++) printf("%d ",s[i].p->data);
    break;
}
else
{
    while (top>0 && s[top].tag==1)
        top--;    /* 当前结点的左右结点都访问过,则退栈 */
    if (top>0) s[top].tag=1;    /* 表示其左结点已访问过 */
    t=s[top].p->right;    /* 开始访问其右结点 */
}
}
}

```

对图 8.47(a)的二叉树 t,调用 ancestor(t,9)的显示结果是:

结点 9 的祖先是:1 2 5

28. 二叉树采用链接存储结构,试设计一个按层次顺序(同一层次自左至右)遍历二叉树的算法。

解:依题意,本算法要采用一个队列 q,先将二叉树根结点入队列,然后退队列,输出该结点,若它有左子树,便将左子树根结点入队列,若它有右子树,便将右子树根结点入队列,如此直到队列空为止。因为队列的特点是先进先出,从而达到按层次顺序遍历二叉树的目的。

因此,实现本题功能的函数如下:

```

#define MAXLEN 100
void translevel(btree *b)
{
    struct node
    {
        btree *vec[MAXLEN];
        int f,r;
    } q;
    q.f=0;    /* 置队列为空队列 */
    q.r=0;
    if (b!=NULL) printf("%d ",b->data);
    q.vec[q.r]=b;    /* 结点指针进入队列 */
}

```

```

q.r=q.r+1;
while (q.f<q.r)                /* 队列不为空 */
{
    b=q.vec[q.f];              /* 队头出队列 */
    q.f=q.f+1;
    if (b->left!=NULL)          /* 输出左孩子,并入队列 */
    {
        printf("%d ",b->left->data);
        q.vec[q.r]=b->left;
        q.r=q.r+1;
    }
    if (b->right!=NULL)         /* 输出右孩子,并入队列 */
    {
        printf("%d ",b->right->data);
        q.vec[q.r]=b->right;
        q.r=q.r+1;
    }
}
;

```

29. 二叉树采用链接存储结构,试设计一个算法计算一棵给定二叉树的所有结点数。

解:依题意,计算一棵二叉树的叶子结点数的递归模型如下:

$$\begin{cases} f(b)=0 & \text{若 } b=NULL \\ f(b)=1 & \text{若 } b->left=NULL \text{ 且 } b->right=NULL \\ f(b)=f(b->left)+f(b->right)+1 & \text{其他} \end{cases}$$

因此,实现本题功能的函数如下:

```

int nodes(btree *b)
{
    int num1,num2;
    if (b==NULL) return(0);
    else if (b->left==NULL && b->right==NULL) return(0);
    else
    {
        num1=nodes(b->left);
        num2=nodes(b->right);
        return(num1+num2+1);
    }
}

```

* 30. 二叉树采用链接存储结构,试设计一个算法计算一棵给定二叉树的叶子结点数。

解:依题意,计算一棵二叉树的叶子结点数的递归模型如下:

$$\begin{cases} f(b)=0 & \text{若 } b=NULL \\ f(b)=1 & \text{若 } b->left=NULL \text{ 且 } b->right=NULL \\ f(b)=f(b->left)+f(b->right) & \text{其他} \end{cases}$$

实现本题功能的函数如下:

```
int leafs(btree *b)
{
    int num1,num2;
    if (b==NULL) return(0);
    else if (b->left==NULL && b->right==NULL) return(1);
    else
    {
        num1=leafs(b->left);
        num2=leafs(b->right);
        return(num1+num2);
    }
}
```

31. 二叉树采用链接存储结构,试设计一个算法计算一棵给定二叉树的单孩子结点数。

解:依题意:计算一棵二叉树的单孩子结点数的递归模型如下:

$$\begin{cases} f(b)=0 & \text{若 } b=NULL \\ f(b)=1 & \text{若 } b->left=NULL \text{ 且 } b->right \neq NULL \\ & \text{或 } b->left \neq NULL \text{ 且 } b->right=NULL \\ f(b)=f(b->left)+f(b->right) & \text{其他} \end{cases}$$

因此,实现本题功能的函数如下:

```
int onechild(btree *b)
{
    int num1,num2;
    if (b==NULL) return(0);
    else if (b->left==NULL && b->right!=NULL)
        || (b->left!=NULL && b->right==NULL) return(1);
    else
    {
        num1=onechild(b->left);
        num2=onechild(b->right);
        return(num1+num2);
    }
}
```

32. 二叉树采用链接存储结构,试设计一个算法计算一棵给定二叉树的双孩子结点数。

解:依题意:计算一棵二叉树的双孩子结点数的递归模型如下:

$$\begin{cases} f(b) = 0 & \text{若 } b = \text{NULL} \\ f(b) = 1 & \text{若 } b \rightarrow \text{left} \neq \text{NULL} \text{ 且 } b \rightarrow \text{right} \neq \text{NULL} \\ f(b) = f(b \rightarrow \text{left}) + f(b \rightarrow \text{right}) & \text{其他} \end{cases}$$

因此,实现本题功能的函数如下:

```
int twochild(b, btree)
{
    int num1, num2;
    if (b == NULL) return(0);
    else if (b->left != NULL && b->right != NULL) return(1);
    else
    {
        num1 = twochild(b->left);
        num2 = twochild(b->right);
        return(num1 + num2);
    }
}
```

* 33. 试证明:已知一棵二叉树的前序序列和中序序列,则可唯一地确定一棵二叉树,并设计由此构造二叉树的递归算法。

证明:采用递归法证明。

当 $n=1$ 时,前序序列和中序序列均只有一个元素,且相同,即为树的根,由此唯一确定了一棵二叉树。

假设 $n \leq m-1$ 时命题均成立,则证明 $n=m$ 时亦成立。

假设前序序列为 a_1, a_2, \dots, a_m , 中序序列为 b_1, b_2, \dots, b_m 。

因为前序序列由前序遍历二叉树所得,则 a_1 即为根结点的元素,又中序序列由中序遍历二叉树所得,则在中序序列中必能找到和 a_1 值相同的元素,设为 b_j ,由此可以得到 $\{b_1, \dots, b_{j-1}\}$ 为左子树的中序序列, $\{b_{j+1}, \dots, b_m\}$ 为右子树的中序序列。

若 $j=1$,即 b_1 为根,此时二叉树的左子树为空, $\{a_2, \dots, a_m\}$ 为右子树的前序序列, $\{b_2, \dots, b_m\}$ 为右子树的中序序列。右子树上的结点数为 $m-1$,由此,这两个序列唯一确定了右子树,也就唯一确定了二叉树。

若 $j=m$,即 b_m 为根,此时二叉树的右子树为空,则子序列 $\{a_2, \dots, a_m\}$ 和 $\{b_1, \dots, b_{m-1}\}$ 唯一确定左子树。

若 $2 \leq j \leq m-1$,则子序列 $\{a_2, \dots, a_j\}$ 和 $\{b_1, \dots, b_{j-1}\}$ 唯一确定了左子树,子序列 $\{a_{j+1}, \dots, a_m\}$ 和 $\{b_{j+1}, \dots, b_m\}$ 唯一确定了右子树。

由此,证明了唯一的根及其左、右子树只能构成一棵确定的二叉树。

本题的算法如下:

设前序序列和中序序列分别存放在两个一维数组 $\text{pre}[1, n]$ 和 $\text{ind}[1, n]$ 中,按前序序列 $\text{pre}[i, j]$ 和中序序列 $\text{ind}[u, v]$ 构造二叉树,其根结点指针为 s 。

```
btree * void bintree(i, j, u, v)
int i, j, u, v;
```

```

{
    int k,l;
    btree * head,* s;
    head=NULL; /* 根指针初始化,head 为空树 */
    if (j>=i)
    {
        head=(btree *)malloc(sizeof(btree)); /* 建立根结点 */
        head->data=pre[i];
        k=u;
        while (ind[k]!=pre[i]) k++; /* 在中序序列中查找根结点 */
        l=i+k-u; /* l 为左子树中最右下结点在前序序列中的位置 */
        if (k==u) head->left=NULL;
        else
        {
            s=bintree(i+1,l,u,k-1); /* 构造左子树 */
            head->left=s;
        }
        if (k==v) head->right=NULL;
        else
        {
            s=bintree(l+1,j,k+1,v); /* 构造右子树 */
            head->right=s;
        }
    }
    return(head);
}

```

* 34. 编写出中序线索二叉树中求结点后继的算法,并以此写出中序遍历二叉树的非递归算法。

解:在中序线索二叉树中求结点后继的算法:由于是中序线索二叉树,后继有时可直接利用线索得到,rtag 为 0 时需要查找,即右子树中最左下的子孙便为后继结点。本函数如下:

```

btree * succ(btree * p)
{
    btree * q;
    if (p->rtag==1) return(p->right); /* 由后继线索直接得到 */
    else
    {
        q=p->right;
        while (q->ltag==0) q=q->left;
        return(q);
    }
}

```

以此给出中序遍历二叉树的非递归算法:只要从头结点出发,反复找到结点的后继,直至结束。本函数如下:

```
void thnorder(btrees)
btrees *t, *h; /* t,原二叉树的根结点指针,h,中序线索二叉树头结点 */
{
    if (t! =NULL)
    {
        p=h;
        do
        {
            printf("%d ",p->data);
            p=succ(p);
        } while (p! =NULL);
    }
}
```

35. 假设二叉排序树 t 的各元素值均不相同,设计一个算法按递增次序打印各元素值。

解:依题意:按中序序列遍历二叉排序树即按递增次序遍历,所以递增打印二叉排序各元素值的函数如下:

```
void incorder(btree *t)
{
    if (t! =NULL)
    {
        incorder(t->left);
        printf("%d ",t->data);
        incorder(t->right);
    }
}
```

36. 假设二叉排序树 t 的各元素值均不相同,设计一个算法按递减次序打印各元素值。

解:依题意:按递增次序遍历二叉排序树的递归步骤为:

- (1)遍历该二叉排序树的右子树
- (2)打印根结点之值
- (3)再遍历该二叉排序树的左子树

因此,实现本题功能的函数如下:

```
void destorder(btree *t)
{
    if (t! =NULL)
    {
        destorder(t->right);
        printf("%d ",t->data);
        destorder(t->left);
    }
}
```


* 37. 设树 b 是一棵采用链接结构存储的二叉树,编写一个把树 b 的左、右子树进行交换的函数。

解:依题意:交换二叉树的左、右子树的递归模型如下:

$$\begin{cases} f(b, t) \rightarrow t = \text{NULL} & \text{若 } b = \text{NULL} \\ f(b, t) \rightarrow \text{复制根结点 } b \text{ 产生 } t, \\ \quad f(b \rightarrow \text{left}, t1), f(b \rightarrow \text{right}, t2), \\ \quad t \rightarrow \text{left} = t2, t \rightarrow \text{right} = t1 & \text{若 } b \neq \text{NULL} \end{cases}$$

因此,实现本题功能的函数如下:

```
btree * swap(btree * b)
{
    btree * t, * t1, * t2;
    if (b == NULL) t = NULL;
    else
    {
        t = (btree *) malloc(sizeof(btree)); /* 复制一个根结点 */
        t->data = b->data;
        t1 = swap(b->left);
        t2 = swap(b->right);
        t->left = t2;
        t->right = t1;
    }
    return(t);
}
```

* 38. 假设以二叉树链表作为存储结构,请编写一个在二叉树中删除所有以结点 x(x 为结点元素值)为根的子树并以某种形式(自定)输出被删除子树的结构算法。

解:依题意:设 print 是以嵌套括号表示输出一个二叉树,本题的算法是:先判定根结点数据域是否为 x,若是则直接输出该二叉树;否则调用的函数对应的递归模型如下:

$$\begin{cases} f(p, x) \rightarrow p \rightarrow \text{left} = \text{NULL}, \text{print}(p \rightarrow \text{left}), f(p \rightarrow \text{right}) \\ \quad \text{若 } p \rightarrow \text{left} \neq \text{NULL} \text{ 且 } p \rightarrow \text{left} \rightarrow \text{data} = x \\ f(p, x) \rightarrow p \rightarrow \text{right} = \text{NULL}, \text{print}(p \rightarrow \text{right}), f(p \rightarrow \text{left}) \\ \quad \text{若 } p \rightarrow \text{right} \neq \text{NULL} \text{ 且 } p \rightarrow \text{right} \rightarrow \text{data} = x \\ f(p, x) \rightarrow f(p \rightarrow \text{left}), f(p \rightarrow \text{right}) & \text{其他} \end{cases}$$

因此,实现本题功能的函数如下:

```
btree * delsubtree(btree * b, int x)
{
    btree * s;
    if (b != NULL)
        if (b->data == x) /* 根结点值等于 x 的情况,直接删除 */
```

```

    {
        print(b);
        s=NULL;
    }
    else s= finddel(b,x);
return(s);
}

btree * finddel(btree * p,int x)
{
    btree * s;
    if (p!=NULL)
    {
        if (p->left!=NULL && p->left->data==x)
        {
            print(p->left);
            p->left=NULL;
        }
        if (p->right!=NULL && p->right->data==x)
        {
            print(p->right);
            p->right=NULL;
        }
        s=finddel(p->left,x);
        p->left=s;
        s=finddel(p->right,x);
        p->right=s;
    }
    return(p);
}

void print(btree * b)
/* 本函数的原理见本章第1节 */
{
    if (b!=NULL)
    {
        printf("%d",b->data);
        if (b->left!=NULL || b->right!=NULL)
        {
            printf("(");
            print(b->left);
            if (b->right!=NULL) printf(",");
            print(b->right);
            printf(")");
        }
    }
}

```

```

    }
  }
}

```

* 39. 试编写一个判断任意给定的二叉树是否为类满二叉树(其任何结点或者为叶子, 或者有左、右两棵非空子树)的递归函数, 并把它转化成非递归函数。

解: 依题意: 类满二叉树的任何结点或者为叶子, 或者恰有两棵非空子树, 由此得到判定一棵二叉树是否为满二叉树的递归模型如下:

$$\begin{cases}
 f(b) = \text{true} & \text{若 } b \rightarrow \text{left} = \text{NULL} \text{ 且 } b \rightarrow \text{right} = \text{NULL} \\
 f(b) = \text{false} & \text{若 } b \rightarrow \text{left}, b \rightarrow \text{right} \text{ 之一为 NULL, 另一不为 NULL} \\
 f(b) = f(b \rightarrow \text{left}) \ \&\& \ f(b \rightarrow \text{right}) & \text{若 } b \rightarrow \text{left} \neq \text{NULL} \text{ 且 } b \rightarrow \text{right} \neq \text{NULL}
 \end{cases}$$

因此, 实现本题功能的递归函数如下:

```

int full(btree *b)
{
    if (b != NULL)
        if (b->left == NULL && b->right == NULL) return(1);
        else if (b->left == NULL || b->right == NULL) return(0);
        else return(full(b->left) && full(b->right));
}

```

将上述递归函数转换成非递归函数时, 使用一个栈 stack, 扫描所有的结点, 一旦遇到有一子树为空者, 即返回 false。因此, 非递归函数如下:

```

int full1(btree *b)
{
    btree stack[100];
    int top=1, ftree=1; /* ftree 先置为 1 */
    if (b != NULL)
    {
        stack[top]=b;
        while (top>0)
        {
            p=stack[top]; /* 退栈 */
            top--;
            if (p->left == NULL && p->right != NULL) ||
                (p->left != NULL && p->right == NULL) ftree=0;
            else if (p->left != NULL && p->right != NULL)
            {
                top++; /* 左右子树根结点入栈 */
                stack[top]=p->right;
                top++;
            }
        }
    }
}

```

```

        stack[top]=p->left;
    }
}
return(ftree);
}

```

* 40. 假设二叉树采用链接方法存储,编写一个递归算法,求出二叉树中所有叶结点的最大和最小枝长。

解:依题意:求二叉树中所有叶结点的最大枝长的递归模型如下:

$$\begin{cases} f(b)=0 & \text{若 } b=\text{NULL} \\ f(b)=\max(f(b-\>\text{left}), f(b-\>\text{right})+1) & b \neq \text{NULL} \end{cases}$$

求二叉树中所有叶结点的最小枝长的递归模型如下:

$$\begin{cases} f(b)=0 & \text{若 } b=\text{NULL} \\ f(b)=\min(f(b-\>\text{left}), f(b-\>\text{right})+1) & b \neq \text{NULL} \end{cases}$$

因此,实现本题功能的递归函数如下:

```

void maxminleaf(btree *b, int *m, *n)
/* *m: 二叉树 b 的最大枝长; *n: 二叉树 b 的最小枝长 */
{
    if (b == NULL)
    {
        *m = 0;
        *n = 0;
    }
    else
    {
        maxminleaf(b->left, &m1, &n1);
        maxminleaf(b->right, &m2, &n2);
        m = max(*m1, *m2) + 1;
        n = min(*m1, *m2) + 1;
    }
}

```

41. 假设二叉树采用链接方法存储,编写一个函数复制一棵给定的二叉树。

解:依题意:复制一棵二叉树的递归模型如下:

$$\begin{cases} f(b)=\text{NULL} & \text{若 } b=\text{NULL} \\ f(b)=p \text{ (} p \text{ 是产生的一个新结点 且;} & \text{其他} \\ \quad p-\>\text{data}=b-\>\text{data} \\ \quad p-\>\text{left}=f(b-\>\text{left}) \\ \quad p-\>\text{right}=f(b-\>\text{right}); \end{cases}$$

因此,实现本题功能的递归函数如下:

```
btree * copy(btree * b)
{
    btree * p;
    if (b != NULL)
    {
        p = (btree *) malloc(sizeof(btree));
        p->data = b->data;
        p->left = copy(b->left);
        p->right = copy(b->right);
        return(p);
    }
    else return(NULL);
}
```

42. 假设二叉树采用链接方法存储,编写一个函数按凹入表表示法打印出该二叉树。

解:本题采用前序遍历的非递归函数,除了使用一个栈外,还增加一个场宽数组 level,它与栈 stack 使用同一下标 top,根结点的场宽设为 6,其左右子树根结点的场宽增 3,这样在遍历显示时按照相应的场宽显示。

因此,实现本题功能的函数如下:

```
void disp(btree * b)
{
    btree * stack[100], * p;
    int level[100], top, n, i;
    if (b != NULL)
    {
        printf("该二叉树表示法:\n");
        top = 1;
        stack[top] = b;                /* 根结点入栈 */
        level[top] = 3;
        while (top > 0)
        {
            p = stack[top];           /* 退栈并凹入显示该结点值 */
            n = level[top];
            for (i = 1; i <= n; i++) /* 其中 n 为显示场宽,字符以右对齐显示 */
                printf(" ");
            printf("%d\n", p->data);
            top--;
            if (p->right != NULL)
            {                          /* 将右子树根结点入栈 */
                top++;
                stack[top] = p->right;
            }
        }
    }
}
```

```

        level[top]=n+3;           /* 显示场宽增 3 */
    }
    if (p->left! =NULL)
    {                               /* 将左子树根结点入栈 */
        top++;
        stack[top]=p->left;
        level[top]=n+3;           /* 显示场宽增 3 */
    }
}
}
}

```

对于图 8.47(a)所示的二叉树 t,调用 disp(t)的结果如下:

该二叉树表示法:

```

1
  2
    4
      7
        5
          8
            9
              3
                6

```

* 43. 用扩充标准形式给定一棵五叉树,data 表示数据域,link1,link2,...,link5 表示 5 个指针域,试编写一个递归函数按后序遍历该五叉树。

解:依题意:递归后序遍历一棵五叉树的步骤为:

后序遍历第 1 棵子五叉树

后序遍历第 2 棵子五叉树

后序遍历第 3 棵子五叉树

后序遍历第 4 棵子五叉树

后序遍历第 5 棵子五叉树

访问根结点

因此,实现本题功能的递归函数如下:

```

typedef struct node
{
    int data;
    struct node * link1, * link2, * link3, * link4, * link5;
} free;

void preorder5(free * b)
{
    if (b! =NULL)

```

```

{
    preorder5(b->link1);
    preorder5(b->link2);
    preorder5(b->link3);
    preorder5(b->link4);
    preorder5(b->link5);
    printf("%d ", b->data);          /* 访问根结点 */
}
}

```

* 44. 试设计一个建立一棵有 m 个结点其数据域值为 $r[i]$ 的二叉树排序树的算法。

解: 依题意: 先设计一个向二叉排序树 b 中插入一个结点 s 的算法, 其过程为:

- (1) 若 b 是空树, 则将 s 所指作为根结点插入; 否则
- (2) 若 $s \rightarrow \text{data}$ 等于 b 的根结点的数据域之值, 则返回; 否则
- (3) 若 $s \rightarrow \text{data}$ 小于 b 的根结点的数据域之值, 则把 s 所指结点插入到左子树中; 否

则

- (4) 把 s 所指结点插入到右子树中。

因此, 向一个二叉排序树 b 中插入一个结点 s 的函数如下:

```

void insert(btree * * b, btree * s)
{
    if (* b == NULL) * b = s;
    else if (s->data < (* b)->data) insert(&((* b)->left), s);
    else if (s->data > (* b)->data) insert(&((* b)->right), s);
}

```

生成二叉排序树的过程是先有一个空树 b , 然后向该空树一个接一个地插入结点实现的, 因此, 生成本题二叉排序树的函数如下:

```

btree * creat(b, r, m)
btree * b;
int r[], m;
{
    int i;
    btree * s;
    b = NULL;
    for (i = 0; i < m; i++)
    {
        s = (btree *) malloc(sizeof(btree));    /* 产生一个树结点 */
        s->data = r[i];
        s->left = NULL;
        s->right = NULL;
        insert(&b, s);                          /* 插入该结点 */
    }
}

```

```

    return(b);
}

main()
{
    btree *p;
    int n,r[10]={3,2,4,8,1,5};
    p=creat(p,r,6);
    disp(p);
}
/* 调用第 42 题的打印二叉树函数 */

```

本程序的执行结果如下：

该二叉树表示法：

```

    3
   /
  2
 /
1
/
4
/
8
/
5

```

* 45. 设计这样的二叉树,用它可以表示父子、夫妻和兄弟三种关系,并编写一个查找任一父亲结点的所有儿子的函数。

解:依题意,这样的二叉树为:一个父亲左子树根结点为其妻子,该妻子结点的右结点为儿子,如图 8.48 所示。

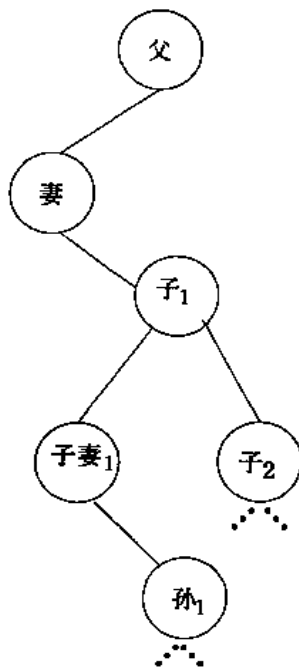


图 8.48 表示三种关系的二叉树

查找任一父亲结点 p 的所有儿子的函数如下：


```

void find(btree *b, int p)
{
    btree *q;
    q=findnode(b,p); /* 调用 findnode 函数在二叉树 b 中找到 p 的结点 */
    if (q!=NULL)
    {
        q=q->left;          /* 找到其妻子结点 */
        q=q->right;         /* 找到其第一个儿子结点 */
        while (q!=NULL)
        {
            printf("%d ", q->data); /* 显示儿子结点的值 */
            q=q->right;          /* 查找其它的儿子结点 */
        }
    }
}

/* 在二叉树 b 中找到 p 的结点的函数 findnode */
btree *findnode(btree *b, int p)
{
    btree *q;
    if (b==NULL) return(NULL);
    else if (b->data==p) return(b);
    else
    {
        q=findnode(b->left,p); /* 在左子树中查找 */
        if (q!=NULL)           /* 找到了则返回该结点指针 */
            return(q);
        else                   /* 未找到则在右子树中查找 */
            return(findnode(b->right,p));
    }
}

```

46. 设计一种根据用户输入建立一棵二叉树并采用凹入表表示法打印出该二叉树。给出一个实例。

解:一般二叉树不同于二叉排序树,后者可以根据用户输入的数据序列自动建立,而二叉树不能这样,用户必须输入更多的信息,为此,这里设计用户输入以下三元组:

父结点值 pvalue,左或右结点 type,当前结点值 cvalue

本程序先在二叉树中找到值为 pvalue 的结点 p,建立一个值为 cvalue 的结点 s,根据 type 将 s 链接到 p 之后,如此这样,直到 pvalue 为 -1,于是便构造了一棵二叉树。实现本题功能的程序如下:

```

#include <stdio.h>
typedef struct bnode
{

```

```

    int data;
    struct bnode * left, * right;
} btree;

btree * find(btree * p, int x)
{
    btree * q;
    if (p == NULL) return(NULL);
    else if (p->data == x) return(p);
    else
    {
        q = find(p->left, x);
        if (q == NULL)
            return(find(p->right, x));
        else return(q);
    }
}

btree * create()
{
    int pvalue, cvalue, type, i = 1;
    btree * p, * q, * s, * head;
    printf("输入建立二叉树序列(以-1表示输入结束)\n");
    printf("第%d个结点=>\n      根结点值:", i++);
    scanf("%d", &cvalue);
    if (cvalue != -1)
    {
        head = (btree *) malloc(sizeof(btree));
        head->data = cvalue;
        head->left = NULL;
        head->right = NULL;
    }
    else return(NULL);
    do
    {
        printf("第%d个结点=>\n      父结点值:", i++);
        scanf("%d", &pvalue);
        if (pvalue != -1)
        {
            do
            {
                printf("  左(0)或右(1)结点:");
                scanf("%d", &type);
            } while (type != 0 && type != 1);
        }
    }
}

```

```

    printf("    当前结点值:");
    scanf("%d",&cvalue);
}
if (pvalue!==-1)
{
    p=head;
    q=find(p,pvalue);
    if (q==NULL)
    {
        s=(btree *)malloc(sizeof(btree));
        s->data=cvalue;
        s->left=NULL;
        s->right=NULL;
        if (type==0) q->left=s;
        if (type==1) q->right=s;
    }
    else
    {
        printf("已建的二叉树中没有指定值的结点\n");
        i--;
    }
}
} while (pvalue!==-1);
return(head);
}

main()
{
    btree *p;
    p=create();
    printf(p);
}

```

本程序的输入如下:

输入建立二叉树序列(以-1表示输入结束)

第1个结点=>

根结点值:1↵

第2个结点=>

父结点值:1↵

左(0)或右(1)结点:0↵

当前结点值:2↵

第3个结点=>

父结点值:1↵

左(0)或右(1)结点:1 ↓
 当前结点值:3 ↓
 第4个结点=>
 父结点值:2 ↓
 左(0)或右(1)结点:0 ↓
 当前结点值:4 ↓
 第5个结点=>
 父结点值:3 ↓
 左(0)或右(1)结点:0 ↓
 当前结点值:5 ↓
 第6个结点=>
 父结点值:3 ↓
 左(0)或右(1)结点:1 ↓
 当前结点值:6 ↓
 第7个结点=>
 父结点值:5 ↓
 左(0)或右(1)结点:0 ↓
 当前结点值:7 ↓
 第8个结点=>
 父结点值:6 ↓
 左(0)或右(1)结点:1 ↓
 当前结点值:9 ↓
 第9个结点=>
 父结点值:7 ↓
 左(0)或右(1)结点:1 ↓
 当前结点值:8 ↓
 第10个结点=>
 父结点值:-1 ↓
 该二叉树表示法:

```

    1
   /
  2
 /  \
4    3
 /    \
5      6
 /      \
7        9
 /
8

```

建立的一棵二叉树如图 8.49 所示。

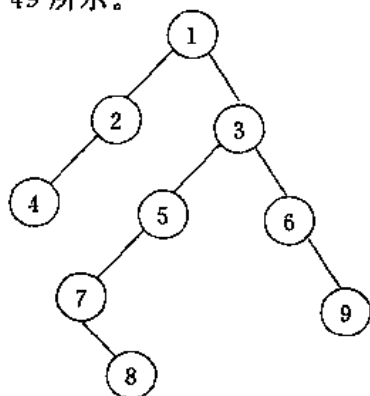


图 8.49 一棵二叉树

47. 根据 Huffman 编码的原理,编写一个程序在用户输入结点权重的基础上建立它的 Huffman 编码。

解:依题意,构造一棵 Huffman 树,由此得到的二进制前缀便为 Huffman 编码。由于 Huffman 树没有度为 1 的结点,则一棵有 n 个叶结点的 Huffman 树共有 $2n-1$ 个结点,设计一个结构数组,存储 $2n-1$ 个结点的值,包括权重、父结点、左结点和右结点等。其程序如下:

```

#include <stdio.h>
#define MAX      21
typedef struct
{
    char data;           /* 结点值 */
    int weight;          /* 权重 */
    int parent;          /* 父结点 */
    int left;            /* 左结点 */
    int right;           /* 右结点 */
} huffnode;
typedef struct
{
    char cd[MAX];
    int start;
} huffcode;

main()
{
    huffnode ht[2 * MAX];
    huffcode hcd[MAX], d;
    int i, k, f, l, r, n, c, m1, m2;
    printf("元素个数:");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
    {

```

```

    getchar();
    printf("第%d个元素=>\n\t 结点值:", i);
    scanf("%c", &ht[i].data);
    printf("\t 权 重:");
    scanf("%d", &ht[i].weight);
}
for (i=1; i<=2*n-1; i++)
    ht[i].parent=ht[i].left=ht[i].right=0;
for (i=n+1; i<=2*n-1; i++)          /* 构造 Huffman 树 */
{
    m1=m2=32767;                      /* l 和 r 为最小权重的两个结点位置 */
    l=r=0;
    for (k=1; k<=i-1; k++)
        if (ht[k].parent==0)
            if (ht[k].weight<m1)
            {
                m2=m1;
                r=l;
                m1=ht[k].weight;
                l=k;
            }
            else if (ht[k].weight<m2)
            {
                m2=ht[k].weight;
                r=k;
            }
    ht[l].parent=i;
    ht[r].parent=i;
    ht[i].weight=ht[l].weight+ht[r].weight;
    ht[i].left=l;
    ht[i].right=r;
}
for (i=1; i<=n; i++)                  /* 根据 Huffman 树求 Huffman 编码 */
{
    d.start=n+1;
    c=l;
    f=ht[i].parent;
    while (f!=0)
    {
        if (ht[f].left==c)
            d.cd[--d.start]='0';
        else

```

```

        d.cd[--d.start]='1';
    c=f;
    f=ht[f].parent;
}
hcd[i]=d;
}
printf("输出 Huffman 编码:\n");          /* 输出 Huffman 编码 */
for (i=1;i<=n;i++)
{
    printf("%c:",ht[i].data);
    for (k=hcd[i].start;k<=n;k++)
        printf("%c",hcd[i].cd[k]);
    printf("\n");
}
}

```

执行本程序及其产生的 Huffman 编码如下:

```

元素个数:5 ↓
第 1 个元素=>
    结点值:a ↓
    权 重:11 ↓
第 2 个元素=>
    结点值:b ↓
    权 重:4 ↓
第 3 个元素=>
    结点值:c ↓
    权 重:2 ↓
第 4 个元素=>
    结点值:d ↓
    权 重:5 ↓
第 5 个元素=>
    结点值:e ↓
    权 重:7 ↓

```

输出 Huffman 编码:

```

a:11
b:011
c:010
d:00
e:10

```

其中产生的 Huffman 树如图 8.50 所示。

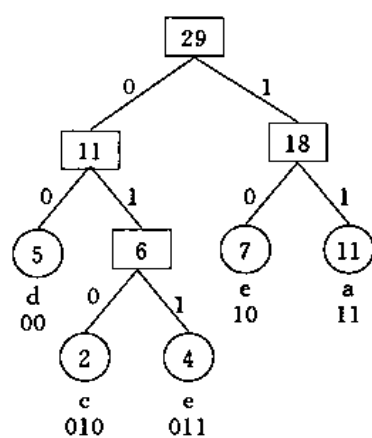


图 8.50 一棵 Huffman 树

第9章 图

图是一种非线性结构,它比树形结构更复杂,甚至可以把前面的线性表和树都看成是简单的图。图的应用十分广泛,很多问题都可以用图表示。

本章讨论图的概念、存储方式和题解。

9.1 图的存储及其运算

9.1.1 图的基本术语

有关图的一些基本术语定义如下:

- 图:图 G 由两个集合 V 和 E 组成,记为 $G=(V,E)$,其中 V 是顶点的有穷非空集合, E 是 V 中顶点偶对有穷集,这些顶点偶对称为边。通常, $V(G)$ 和 $E(G)$ 分别表示图 G 的顶点集合和边集合。 $E(G)$ 也可以为空集。若 $E(G)$ 为空,则图 G 只有顶点而没有边。
- 有向图:对于一个图 G ,若边集合 $E(G)$ 为有向边的集合,则称该图为有向图。
- 无向图:对于一个图 G ,若边集合 $E(G)$ 为无向边的集合,则称该图为无向图。
- 端点和邻接点:在一个无向图中,若存在一条边 $\langle v_i, v_j \rangle$,则称 v_i, v_j 为该边的两个端点,并称它们互为邻接点。
- 起点和终点:在一个有向图中,若存在一条边 $\langle v_i, v_j \rangle$,则称该边是顶点 v_i 的一条出边,顶点 v_j 的一条入边;称 v_i 为起始端点(或起点), v_j 为终止端点(或终点);称 v_i 和 v_j 互为邻接点,并称 v_j 是 v_i 的出边邻接点, v_i 是 v_j 的入边邻接点。
- 度、入度和出度:图中每个顶点的度定义为以该顶点为一个端点的边的数目,记为 $D(v)$ 。对于有向图,顶点 v 的度分为入度和出度,入度是以该顶点为终点的入边数目;出度是以该顶点为起点的出边数目,该顶点的度等于其入度和出度之和。
- 子图:设有两个图 $G=(V,E)$ 和 $G'=(V',E')$,若 V' 是 V 的子集,即 $V' \subseteq V$,且 E' 是 E 的子集,即 $E' \subseteq E$,则称 G' 是 G 的子图。
- 无向完全图:具有 $n(n-1)/2$ 条边的无向图称为无向完全图。
- 有向完全图:具有 $n(n-1)$ 条边的有向图称为有向完全图。
- 稀疏图:边很少(如 $e < n \log_2 n$)的图称为稀疏图。
- 稠密图:边很多的图称为稠密图。
- 路径和路径长度:在一个无向图 G 中,从顶点 v 到顶点 v' 的路径是一个顶点序列 $v_{i0}, v_{i1}, \dots, v_{im}$,其中 $v=v_{i0}, v'=v_{im}$,若该图是无向图,则顶点序列应满足 $(v_{ij-1}, v_{ij}) \in E(G), (1 \leq j \leq m)$;若该图是有向图,则顶点序列应满足 $\langle v_{ij-1}, v_{ij} \rangle \in E(G), (1 \leq j \leq m)$ 。路径长度是指一条路径上经过的边的数目。
- 简单路径:若一条路径上除了开始顶点和结束顶点为同一个顶点外,其余顶点均不

重复出现的路径称为简单路径。

- 回路或环:若一条路径上的开始顶点和结束顶点为同一个顶点,则称该路径为回路或环。
- 连通、连通图和连通分量:在无向图 G 中,若从顶点 v_i 到顶点 v_j 有路径,则称 v_i 和 v_j 是连通的。若图 G 中任意两个顶点都连通,则称 G 为连通图,否则为非连通图。无向图 G 中的极大连通子图称为 G 的连通分量。
- 强连通图和强连通分量:在有向图 G 中,若任意两个顶点 v_i 和 v_j 都连通,即从 v_i 到 v_j 和从 v_j 到 v_i 都存在路径,则称该图是强连通图。有向图 G 中的极大强连通子图称为 G 的强连通分量。
- 权和网:在一个图中,每条边可以标上具有某种含义的数值,该数值称为该边的权。边上带权的图称为带权图,也称为网。

9.1.2 图的存储方式

图有两种基本的存储方式即邻接矩阵和邻接表。

1. 邻接矩阵

邻接矩阵是表示顶点之间相邻关系的矩阵。设 $G=(V,E)$ 是具有 n 个顶点的图,顶点序号依次为 $1,2,\dots,n$,则 G 的邻接矩阵是具有如下定义的 n 阶方阵 A :

$$A[i,j]=\begin{cases} 1 & \text{对于无向图,}(v_i,v_j)\text{或}(v_j,v_i)\in E(G); \text{对于有向图,}\langle v_i,v_j\rangle\in E(G) \\ 0 & \text{其他} \end{cases}$$

此时,邻接矩阵定义如下:

```
int adjmatrix=ARRAY[n][n];
```

若 G 是网,则邻接矩阵可定义为:

$$A[i,j]=\begin{cases} w_{i,j} & \text{若}(v_i,v_j)\text{或}\langle v_i,v_j\rangle\in E(G) \\ 0\text{ 或}\infty & \text{其他} \end{cases}$$

此时,邻接矩阵定义如下:

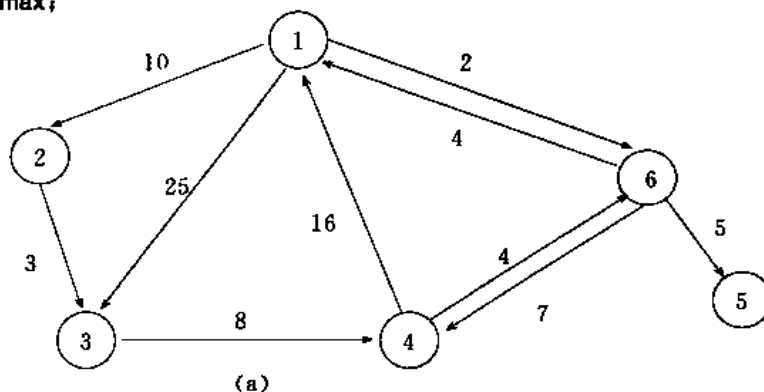
```
int adjnetwork[n][n]; /* n 为顶点个数 */
```

例如 图 9.1 所示为图 G 的带权图及对应的邻接矩阵表示。

为了反映一个图的全面信息,可以采用以下结构:

```
#define MAXVEX 100
struct vertex
{
    int num; /* 顶点编号 */
    char data; /* 顶点的信息 */
};
typedef struct graph
{
    struct vertex vexs[MAXVEX]; /* 顶点集合 */
```

```
int edges[MAXVEX][MAXVEX];    /* 边的集合 */
} adjmax;
```



(b)

	1	2	3	4	5	6	
A =	∞	10	15	∞	∞	2	1
	∞	∞	3	∞	∞	∞	2
	∞	∞	∞	8	∞	∞	3
	16	∞	∞	∞	∞	4	4
	∞	∞	∞	∞	∞	∞	5
	9	∞	∞	7	5	∞	6

图 9.1 图 G 的带权图及对应的矩阵表示

其中 MAXVEX 定义一个图的最多顶点个数, vertex 结构定义一个顶的基本数据, adjlist 定义图的类型, 包含该图的所有顶点和边。

以下函数通过用户交互产生一个图的邻接矩阵表示。

```
admax creagraph(int *n, int *e)
{
    int i, j, k, w;
    char b, t;
    admax adj;
    printf("顶点数(n)和边数(e):");
    scanf("%d, %d", n, e);
    for (i = 1; i <= *n; i++)
    {
        getchar();
        printf("\t 第%d 个顶点的信息:", i);
        scanf("%c", &adj.vexs[i].data);
        adj.vexs[i].num = i;
    }
    for (i = 1; i <= *n; i++)
        for (j = 1; j <= *n; j++)
```

```

    adj.edges[i][j]=0;
    for (k=1;k<=*e;k++)
    {
        getchar();
        printf("第%d条边=>",k);
        printf("\n\t起点:");
        b=getche();
        printf("  终点:");
        t=getche();
        printf("  权值:");
        scanf("%d",&w);
        i=1;
        while (i<=*n && adj.vexs[i].data!=b) i++;
        if (i>*n) {
            printf("输入的起点不存在! \n");
            exit(0);
        }
        j=1;
        while (j<=*n && adj.vexs[j].data!=t) j++;
        if (j>*n) {
            printf("输入的终点不存在! \n");
            exit(1);
        }
        adj.edges[i][j]=w;
    }
    return(adj);
}

```

2. 邻接表

邻接表是图的一种链接存储结构。在邻接表中,对图中每个顶点建立一个单链表,第*i*个单链表中的结点表示依附于顶点 v_i 的边。每个结点由两个域组成:邻接点域(adjvex),用以指示与顶点 v_i 邻接的点在图中的位置;链域(next arc),用以指向依附于顶点 v_i 的下一条边所对应的结点。如果用邻接表存放网中的信息,则还需要在结点中增加一个存放权值的域。无论是存储图或网,需要在每个链表设一表头结点,这些表头结点本身以向量的形式存储。

值得注意的是,一个图的邻接矩阵表示是唯一的,但其邻接表表示不唯一。这是因为邻接表表示中,各边表结点的链接次序取决于建立邻接表的算法以及边的输入次序。也就是说,在邻接表的每个线性链表中,各结点的顺序是任意的。

一个图的邻接表存储结构定义如下:

```

#include <stdio.h>
#define MAXVEX 30
struct edgenode

```

```

    {
        int adjvex;           /* 邻接点序号 */
        char info;           /* 邻接点信息 */
        struct edgenode * next;
    };
struct vexnode
{
    char data;               /* 结点信息 */
    struct edgenode * link;
};
typedef struct vexnode adjlist[MAXVEX];

```

以下函数通过用户交互产生一个图的邻接表表示。

```

void creagraph(adjlist g,int *n)
{
    int e,i,s,d;
    struct edgenode * p,*q;
    printf("结点数(n)和边数(e):");
    scanf("%d,%d",&n,&e);
    for (i=1;i<= *n;i++)
    {
        getchar();
        printf("\t第%d个结点信息:",i);
        scanf("%c",&g[i].data);
        g[i].link=NULL;
    }
    for (i=1;i<=e;i++)
    {
        printf("\n 第%d条边=>\n\t起点序号,终点序号:",i);
        scanf("%d,%d",&s,&d);
        p=(struct edgenode *)malloc(sizeof(struct edgenode));
        q=(struct edgenode *)malloc(sizeof(struct edgenode));
        p->adjvex=d;
        p->info=g[d].data;
        q->adjvex=s;
        q->info=g[s].data;
        p->next=g[s].link;      /* p 插入顶点 s 的邻接表中 */
        g[s].link=p;
        q->next=g[d].link;     /* q 插入顶点 d 的邻接表中 */
        g[d].link=q;
    }
}

```

以下函数用于遍历邻接表表示的一个图。

```

void travgraph(adjlist g,int n)
{
    int i;
    struct vexnode *p;
    printf("遍历图的结果如下:\n");
    for (i=1;i<=n;i++)
    {
        printf("[%d,%c]==>",i,g[i].data);
        p=g[i].link;
        while (p!=NULL)
        {
            printf("(%d,%c)-->",p->adjvex,p->info);
            p=p->next;
        }
        printf("\n");
    }
}

```

例如,图9.1中的图G用邻接表表示如图9.2所示。

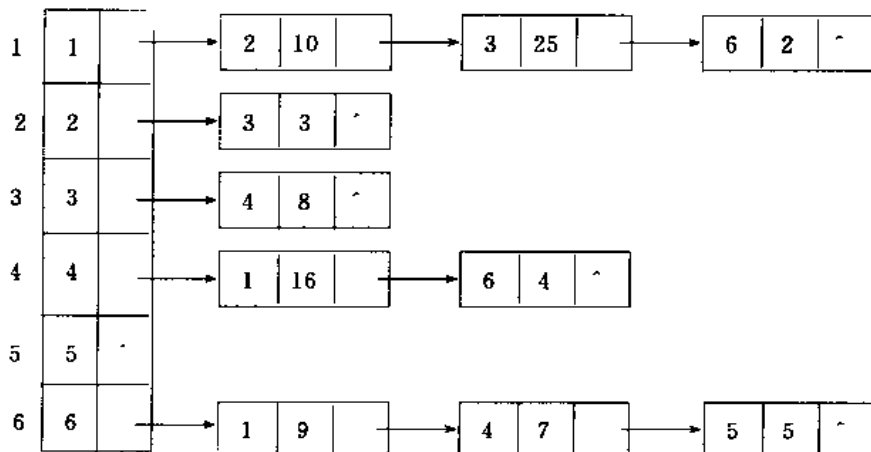


图 9.2 图 G 的邻接表表示

9.1.3 图的基本运算

图的基本运算有深度优先搜索法、宽度优先搜索法、生成最小生成树法和产生最短路径法等。

1. 深度优先搜索法

深度优先搜索法的基本思想是:从图G中某个顶点 v_0 出发,访问 v_0 ,然后选择一个与 v_0 相邻且未被访问过的顶点 v_i 进行访问,再从 v_i 出发选择一个与 v_i 相邻且未被访问的顶点 v_j 进行访问,依次继续。如果当前被访问过的顶点的所有邻接顶点都被访问,则退回到已被访问的顶点序列中最后一个拥有未被访问的相邻顶点的顶点 w ,从 w 出发按同样方法向前

遍历。直到图中所有顶点都被访问为止。

实现深度优先搜索的递归函数如下:

```
int visited[MAXVEX],
void dfs(adjlist adj,int v)
{
    int i;
    struct edgenode *p;
    for (i=1;i<=n;i++) visited[i]=0;      /* 给 visited 数组赋初值 0 */
    visited[v]=1;
    printf("%d ",v);                       /* 取 v 的边的表头指针 */
    p=adj[v]->link;
    while (p!=NULL)
    {
        /* 从 v 的未访问过的邻接点出发进行深度优先搜索 */
        if (visited[p->adjvex]==0) dfs(adjlist,p->adjvex);
        p=p->next;                          /* 找 v 的下一个邻接点 */
    }
}
```

2. 宽度优先搜索法

宽度优先搜索法的基本思想是:首先访问初始点 v_1 ,并将其标记为已访问过,接着访问 v_1 的所有未被访问过的邻接点 $v_{11}, v_{12}, \dots, v_{1n}$,并均标记为已访问过,然后再按照 $v_{11}, v_{12}, \dots, v_{1n}$ 的次序,访问每一个顶点的所有未被访问过的邻接点,并均标记为已访问过,依次类推,直到图中所有和初始点 v_1 有路径相通的顶点都被访问过为止。

实现宽度优先搜索的递归函数如下:

```
int visited[MAXVEX],
int queue[MAXVEX],
void bfs(adjlist adj,int vi)
{
    int front=0,rear=1,v;
    struct edgenode *p;
    visited[vi]=1;                          /* 访问初始顶点 */
    printf("%d ",v);
    queue[rear]=vi;                          /* 初始顶点入队列 */
    while (front!=rear)                      /* 队列不为空 */
    {
        front=(front+1)%m;
        v=queue[front];                      /* 按访问次序依次出队列 */
        p=adj[v]->link;                     /* 找 v 的邻接点 */
        while (p!=NULL)
        {
            if (visited[p->adjvex]==0)
```

```

    {
        visited[p->adjvex]=1;
        printf("%d ",p->adjvex);          /* 访问该点并使之入队列 */
        rear=(rear+1)%m;
        queue[rear]=p->adjvex;
    }
    p=p->next;                            /* 找 v 的下一个邻接点 */
}
}
}

```

3. 生成最小生成树法

在一个连通图 G 中,如果取它的全部顶点和一部分边构成一个子图 G' ,即:

$$V(G')=V(G) \text{ 和 } E(G')\subseteq E(G)$$

若边集 $E(G')$ 中的边既将图 G 中的所有顶点连通又不形成回路,则称子图 G' 是原图 G 的一棵生成树。具有权最小的生成树称为图的最小生成树。

生成最小生成树的算法有两个,即普里姆算法和克鲁斯卡尔算法。

(1) 普里姆算法:假设 $G=(V,E)$ 是一个具有 n 个顶点的连通网, $T=(U,TE)$ 是 G 的最小生成树,其中 U 是 T 的顶点集, TE 是 T 的边集, U 和 TE 的初值均为空。算法开始时,首先从 V 中任取一个顶点(假定取 v_1),将它并入 U 中,此时 $U=\{v_1\}$,然后只要 U 是 V 的真子集(即 $U\subset V$),就从那些其一个端点已在 T 中,另一个端点仍在 T 外的所有边中,找一条最短(即权值最小)边,假定为 (v_i,v_j) ,其中 $v_i\in U, v_j\in V-U$,并把该边 (v_i,v_j) 和顶点 v_j 分别并入 T 的边集 TE 和顶点集 U ,如此进行下去,每次往生成树里并入一个顶点和一条边,直到 $(n-1)$ 次后把所有 n 个顶点都并入到生成树 T 的顶点集中,此时 $U=V$, TE 中包含有 $(n-1)$ 条边, T 就是最后得到的最小生成树。

为了便于在集合 U 和 $V-U$ 之间选择权最小的边,建立两个数组 `closest` 和 `lowcost`,`closest[i]` 表示 U 中的一个顶点,该顶点和 $V-U$ 中的一个顶点构成的边 $(i,closest[i])$ 具有最小的权;`lowcost[i]` 表示边 $(i,closest[i])$ 的权。开始,由于 U 的初值为 $\{1\}$,所以,`closest[i]` 的值为 1, $i=2,\dots,n$,而 `lowcost[i]` 为边 $(1,i)$ 的权, $i=2,\dots,n$ 。

本算法每一步扫描数组 `lowcost`,在 $V-U$ 中找出离 U 最近的顶点,令其为 k ,并打印边 $(k,closest[k])$ 。然后修改数组 `lowcost` 和 `closest`,标记 k 已经加入 U 。这里用 c 表示图的邻接矩阵,`c[i][j]` 和 `c[j][i]` 是边 (i,j) 的权。如果不存在边 (i,j) ,则 `c[i][j]` 的值为一个大于任何权而小于无限大的常数(这里用 32767 表示)。

利用普里姆算法构造最小生成树的函数如下:

```

void prim(c,n)
int c[MAXVEX][n];
/* 已知图的顶点为 1,2,...,n,c[i][j] 和 c[j][i] 为边 (i,j) 的权,打印最小生成树的每条边 */
{
    int lowcost[n],closest[n];
    int i,j,k,min;

```



```

for (i=2;i<=n;i++)          /* 从顶点 v1 开始 */
{
    lowcost[i]=c[1][i];
    closest[i]=1;
}
for (i=2;i<=n;i++)          /* 从 U 之外求离 U 中某一顶点最近的顶点 */
{
    min=lowcost[i];
    k=i;
    for (j=2;j<=n;j++)
        if (lowcost[j]<min)
        {
            min=lowcost[j];
            k=j;
        }
    printf("( %d, %d)", k, closest[i]);    /* 打印边 */
    lowcost[k]=32767;                    /* k 加入到 U 中 */
    for (j=2;j<=n;j++)
        if (c[k][j]<lowcost[j] && lowcost[j]<32767)
        {
            lowcost[j]=c[k][j];
            closest[j]=k;
        }
    }
}

```

(2) 克鲁斯卡尔算法:假设 $G=(V,E)$ 是一个具有 n 个顶点的连通网, $T=(U,TE)$ 是 G 的最小生成树, U 的初值等于 V , 即包含有 G 中的全部顶点, TE 的初值为空集。该算法的基本思想是:将图 G 中的边按权值从小到大的顺序依次选取,若选取的边使生成树 T 不形成回路,则把它并入 TE 中,保留作为 T 的一条边,若选取的边使生成树 T 形成回路,则将其舍弃,如此进行下去,直到 TE 中包含 $n-1$ 条边为止,此时的 T 即为最小生成树。

利用克鲁斯卡尔算法构造最小生成树的函数如下:

```

#define MAXE 100              /* MAXE 为最大的边数 */
struct edges /* 边集类型,存储一条边的起始顶点 bv、终止顶点 tv 和权 w */
{
    int bv, tv, w;
};
typedef struct edges edgeset[MAXE];
int seeks(int set[], int v)
{
    int i=v;
    while (set[i]>0) i=set[i];
}

```

```

    return(i);
}

kruskal(ge,n,e)
int n,e;
edgeset ge; /* ge 表示的图是按权值从小到大排列的 */
{
    int set[MAXE],v1,v2,l,j;
    for (i=1;i<=n;i++) s[i]=0; /* 给 s 中的每个元素赋初值 */
    i=1; /* i 表示待获取的生成树中的边数,初值为 1 */
    j=1; /* j 表示 ge 中的下标,初值为 1 */
    while (j<n && i<=e) /* 按边权递增顺序,逐边检查该边是否应加入到生成树中 */
    {
        v1=seeks(set,ge[i].bv); /* 确定顶点 v 所在的连通集 */
        v2=seeks(set,ge[i].tv);
        if (v1!=v2) /* 当 v1,v2 不在同一顶点集合,确定该边应当选入生成树 */
        {
            printf("( %d,%d) ",ge[i].bv,ge[i].tv);
            set[v1]=v2;
            j++;
        }
        i++;
    }
}

```

4. 最短路径法

求最短路径的 Dijkstra 算法:设有向图 $G=(V,E)$,其中, $V=\{1,2,\dots,n\}$,cost 是表示 G 的邻接矩阵, $\text{cost}[i][j]$ 表示有向边 $\langle i,j \rangle$ 的权。若不存在有向边 $\langle i,j \rangle$,则 $\text{cost}[i][j]$ 的权为无穷大(这里取值为 32767)。设 s 是一个集合,其中的每个元素表示一个顶点,从源点到这些顶点的最短距离已经求出。设顶点 v_0 为源点,集合 s 的初态只包含顶点 v_0 。数组 dist 记录从源点到其他各顶点当前的最短距离,其初值为 $\text{dist}[i]=\text{cost}[v_0][i], i=2,\dots,n$ 。从 s 之外的顶点集合 $V-S$ 中选出一个顶点 w ,使 $\text{dist}[w]$ 的值最小。于是从源点到达 w 只通过 s 中的顶点,把 w 加入集合 s 中调整 dist 中记录的从源点到 $V-S$ 中每个顶点 v 的距离:从原来的 $\text{dist}[v]$ 和 $\text{dist}[w]+\text{cost}[w][v]$ 中选择较小的值作为新的 $\text{dist}[v]$ 。重复上述过程,直到 s 中包含 v 中其余各顶点的最短路径。

最终结果是: s 记录了从源点到该顶点存在路径的顶点集合,数组 dist 记录了从源点到 V 中其余各顶点之间的最短路径, path 是最短路径的路径数组,其中 $\text{path}[i]$ 表示从源点到顶点 i 之间的最短路径的前驱顶点。

采用 Dijkstra 算法求最短路径的函数如下:

```

#define MAXVEX 100 /* 定义最多顶点数 */
void shortpath(cost,dist,path,n,v0)
int cost[MAXVEX][MAXVEX],dist[MAXVEX],path[MAXVEX],n,v0;
{

```

```

int s[MAXVEX], u, vnum, w, wm;
for (w = 1; w <= n; w++) /* 最短路径初始化值 */
{
    dist[w] = cost[v0][w];
    if (cost[v0][w] < 32767) path[w] = v0; /* path 记录当前最短路径 */
}
for (w = 1; w <= n; w++) s[w] = 0;
s[v0] = 1;
vnum = 1; /* s 中顶点个数的初值 */
while (vnum < n - 1) /* 最后一点已无选择余地 */
{
    wm = max;
    u = v0;
    for (w = 1; w <= n; w++)
        if (s[w] == 0 && dist[w] < wm)
        {
            u = w;
            wm = dist[w]; /* 找最小 dist[w] */
        }
    s[u] = 1;
    vnum++; /* u 为找到最短路径的终点 */
    for (w = 1; w <= n; w++)
        if (s[w] == 0 && dist[u] + cost[u][w] < dist[w])
        {
            dist[w] = dist[u] + cost[u][w]; /* 调整非 s 集合点最短路径值 */
            path[w] = u; /* 调整非 s 集合点最短路径 */
        }
    vnum++;
}
}

```

显示最短路径的函数如下:

```

void printpath(dist, path, s, n)
int dist[MAXVEX], path[MAXVEX], s[MAXVEX], n, v0;
{
    int i, k;
    for (i = 1; i <= n; i++)
        if (s[i] == 1)
        {
            k = i;
            while (k != v0)
            {
                printf("%d←", k);
            }
        }
    printf("%d", v0);
}

```

```

    k=path[k];
    /* 通过找到前驱顶点,反向输出最短路径 */
}
printf("%d ",k);
printf("%d ",dist[i]);
}
else
{
    printf("%d←%d",i,v0);
    printf("∞");
    /* 不存在路径时,路径长度为∞ */
}
}

```

如图 9.3 所示有向图,计算从顶点 2 到其他各顶点的最短路径的动态执行情况如图 9.4 所示。最后的输出结果是:

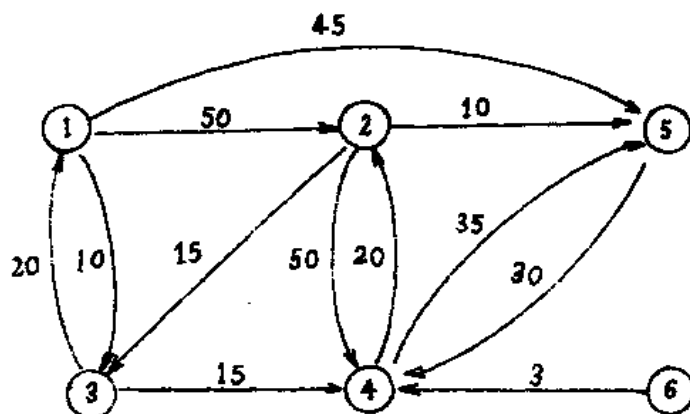


图 9.3 一个有向图

s	dist						path					
	1	2	3	4	5	6	1	2	3	4	5	6
{2}	∞	0	15	50	10	∞		2	2	2	2	
{2,5}	∞	0	15	40	10	∞		2	2	5	2	
{2,5,3}	35	0	15	30	10	∞	3	2	2	3	2	
{2,5,3,4}	35	0	15	30	10	∞	3	2	2	3	2	
{2,5,3,4,1}	35	0	15	30	10	∞	3	2	2	3	2	

图 9.4 最短路径算法的动态执行情况

1←3←2 35

2 0

3←2 15

4←3←2 30

5 ← 2 10

6 ← 2 ∞

5. 拓扑排序法

AOV 网: 顶点表示活动, 边表示活动间的先后关系的有向图称为顶点活动网, 简称 AOV 网。在网中, 若从顶点 i 到顶点 j 有一条有向路径, 则 i 是 j 的前驱, j 是 i 的后继。若 $\langle i, j \rangle$ 是网中一条弧, 则 i 是 j 的直接前驱, j 是 i 的直接后继。

拓扑排序: 是对 AOV 网构造一个线性序列: $(\dots, v_i, \dots, v_k, \dots, v_j, \dots)$, 使所有优先关系 $\langle v_i, v_j \rangle$, 在序列中得以体现, 即 v_i 排在 v_j 之前。客观上线性序列也补充了一些优先关系, 即 v_i, v_k 和 v_k, v_j 之间原没有弧, 但经拓扑排序之后, v_i 与 v_k, v_k 与 v_j 之间形成了一个有序的路径。 v_i 在 v_k 之前, v_k 在 v_j 之前。

拓扑排序步骤:

- (1) 在有向图中选一个没有前驱的顶点, 且输出之。
- (2) 从有向图中删除该顶点, 且删除以该顶点为尾的所有有向边。

重复执行上述步骤, 直至全部顶点都已输出了或图中剩余的顶点中没有前驱顶点为止。

采用邻接表存储 AOV 网, 进行拓扑排序的函数如下 (这里的 data 域改为 int 类型, 存储该顶点的入度):

```
void topsort(adjlist adj, int n) /* adj 为邻接表, 拓扑排序结果用 printf() 输出 */
{
    int m, i, j, ghead;
    struct vexnode *q;
    ghead = 0; /* ghead 是入度为 0 的顶点链表的头指针 */
    m = 0; /* m 指示输出的顶点个数 */
    for (i = 1; i <= n; i++) /* 建立入度为 0 的顶点链表 */
        if (adj[i] -> data == 0)
        {
            adj[i] -> data = ghead;
            ghead = i;
            while (ghead != 0)
            {
                j = ghead;
                ghead = adj[j] -> data; /* 在链表中删除入度为 0 的顶点, 顶点序号为 j */
                q = adj[j] -> link;
                printf("%d ", j); /* 输出顶点 vj 并计数 */
                m++;
                while (q != NULL)
                {
                    k = q -> adjvex;
                    adj[k] -> data = adj[k] -> data - 1; /* 以顶点 vk 为尾的弧头的入度减 1 */
                    if (adj[k] -> data == 0)
```

```

    {
        /* 将新的入度为 0 的顶点插入链表 */
        adj[k] -> data = ghead;
        ghead = k;
    }
    p = p -> next;          /* 找下一条弧 */
}
}
if (m < n) printf("网中有环! \n");    /* 输出顶点数不足 n */
}

```

9.2 基本题

9.2.1 单项选择题

1. 在一个图中,所有顶点的度数之和等于所有边数的 ① 倍。

A. 1/2 B. 1 C. 2 D. 4

答:①C

2. 在一个有向图中,所有顶点的入度之和等于所有顶点的出度之和的 ① 倍。

A. 1/2 B. 1 C. 2 D. 4

答:①B

3. 一个有 n 个顶点的无向图最多有 ① 条边。

A. n B. $n(n-1)$ C. $n(n-1)/2$ D. $2n$

答:①C

4. 具有 4 个顶点的无向完全图有 ① 条边。

A. 6 B. 12 C. 16 D. 20

答:①A

5. 具有 6 个顶点的无向图至少应有 ① 条边才能确保是一个连通图。

A. 5 B. 6 C. 7 D. 8

答:①A

6. 在一个具有 n 个顶点的无向图中,要连通全部顶点至少需要 ① 条边。

A. n B. $n+1$ C. $n-1$ D. $n/2$

答:①C

7. 对于一个具有 n 个顶点的无向图,若采用邻接矩阵表示,则该矩阵的大小是 ①。

A. n B. $(n-1)^2$ C. $n-1$ D. n^2

答:①D

8. 对于一个具有 n 个顶点和 e 条边的无向图,若采用邻接表表示,则表头向量的大小为 ①;所有邻接表中的结点总数是 ②。

① A. n B. $n+1$ C. $n-1$ D. $n+e$

② A. $e/2$ B. e C. $2e$ D. $n+e$

答: ① A ② C

9. 已知一个图如图 9.5 所示, 若从顶点 a 出发按深度搜索法进行遍历, 则可能得到的一种顶点序列为 ①; 按宽度搜索法进行遍历, 则可能得到的一种顶点序列为 ②。

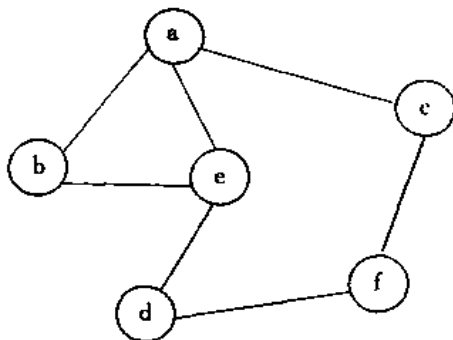


图 9.5 一个无向图

① A. a, b, e, c, d, f

B. a, c, f, e, b, d

C. a, e, b, c, f, d

D. a, e, d, f, c, b

② A. a, b, c, e, d, f

B. a, b, c, e, f, d

C. a, e, b, c, f, d

D. a, c, f, d, e, b

答: ① D ② B

10. 已知一有向图的邻接表存储结构如图 9.6 所示。

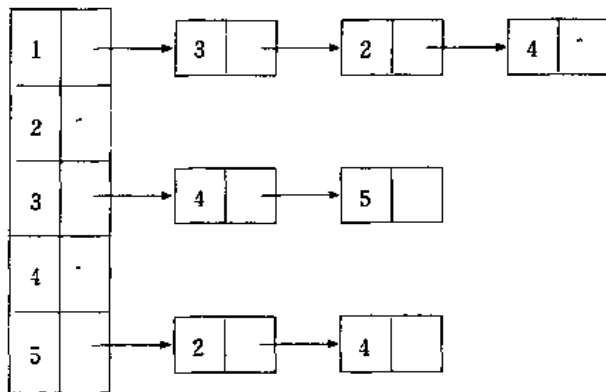


图 9.6 一个有向图的邻接表存储结构

(1) 根据有向图的深度优先遍历算法, 从顶点 v_1 出发, 所得到的顶点序列是 ①。

A. v_1, v_2, v_3, v_5, v_4 B. v_1, v_2, v_3, v_4, v_5

C. v_1, v_3, v_4, v_5, v_2 D. v_1, v_4, v_3, v_5, v_2

(2) 根据有向图的宽度优先遍历算法, 从顶点 v_1 出发, 所得到的顶点序列是 ②。

A. v_1, v_2, v_3, v_4, v_5 B. v_1, v_3, v_2, v_4, v_5

C. v_1, v_2, v_3, v_5, v_4 D. v_1, v_4, v_3, v_5, v_2

答:①C ②B

11. 采用邻接表存储的图的深度优先遍历算法类似于二叉树的 ①。

- A. 先序遍历 B. 中序遍历
C. 后序遍历 D. 按层遍历

答:①A

12. 采用邻接表存储的图的宽度优先遍历算法类似于二叉树的 ①。

- A. 先序遍历 B. 中序遍历
C. 后序遍历 D. 按层遍历

答:①D

13. 判定一个有向图是否存在回路除了可以利用拓扑排序方法外,还可以利用①。

- A. 求关键路径的方法 B. 求最短路径的 Dijkstra 方法
C. 宽度优先遍历算法 D. 深度优先遍历算法

答:①D

9.2.2 填空题(将正确的答案填在相应的空中)

1. n 个顶点的连通图至少 ① 条边。

答:① $n-1$

2. 在无权图 G 的邻接矩阵 A 中,若 (v_i, v_j) 或 $\langle v_i, v_j \rangle$ 属于图 G 的边集合,则对应元素 $A[i][j]$ 等于 ①,否则等于 ②。

答:①1;②0

3. 在无向图 G 的邻接矩阵 A 中,若 $A[i][j]$ 等于 1,则 $A[j][i]$ 等于 ①。

答:①1

4. 已知图 G 的邻接表如图 9.7 所示,其从顶点 v_1 出发的深度优先搜索序列为 ①,其从顶点 v_1 出发的宽度优先搜索序列为 ②。

答:① $v_1, v_2, v_3, v_6, v_5, v_4$; ② $v_1, v_2, v_5, v_4, v_3, v_6$

5. 已知一个图的邻接矩阵表示,计算第 i 个结点的入度的方法是 ①。

答:①求矩阵第 i 列非零元素之和

6. 已知一个图的邻接矩阵表示,删除所有从第 i 个结点出发的边的方法是 ①。

答:①将矩阵第 i 行全部置为零

9.3 习题解析

1. 给出如图 9.8 所示的无向图 G 的邻接矩阵和邻接表两种存储结构。

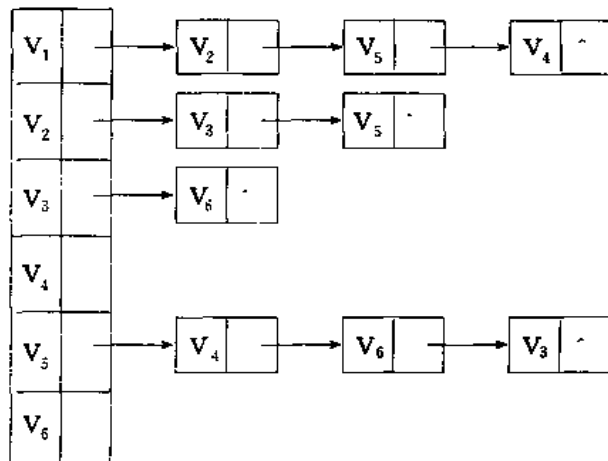


图 9.7 图 G 的邻接表

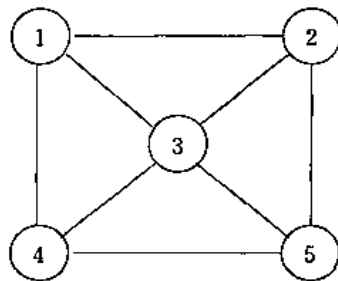


图 9.8 一个无向图 G

解:图 G 对应的邻接矩阵和邻接表两种存储结构分别如图 9.9 和 9.10 所示。

$$A = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix} \end{matrix} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix}$$

图 9.9 图 9.8 的邻接矩阵

* 2. 用宽度优先搜索和深度优先搜索对如图 9.11 所示的图 G 进行遍历(从顶点 1 出发),给出遍历序列。

解:搜索本题图的宽度优先搜索的序列为:1,2,3,6,4,5,8,7,深度优先搜索的序列为:1,2,6,4,5,7,8,3。

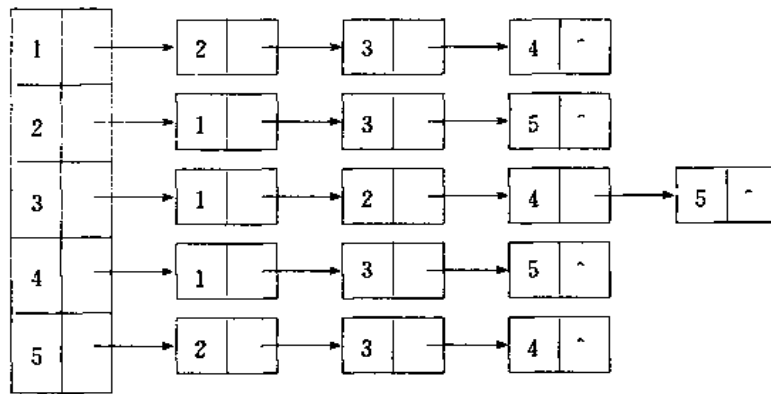


图 9.10 图 9.8 的邻接表

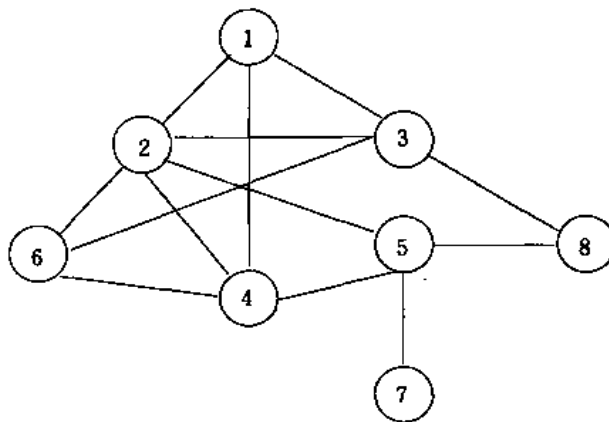


图 9.11 一个无向图 G

3. 使用普里姆算法构造出如图 9.12 所示的图 G 的一棵最小生成树。

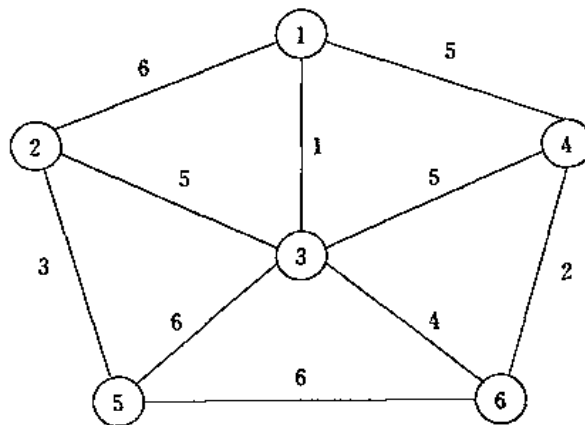


图 9.12 一个无向图 G

解：构造最小生成树的过程如图 9.13 所示。

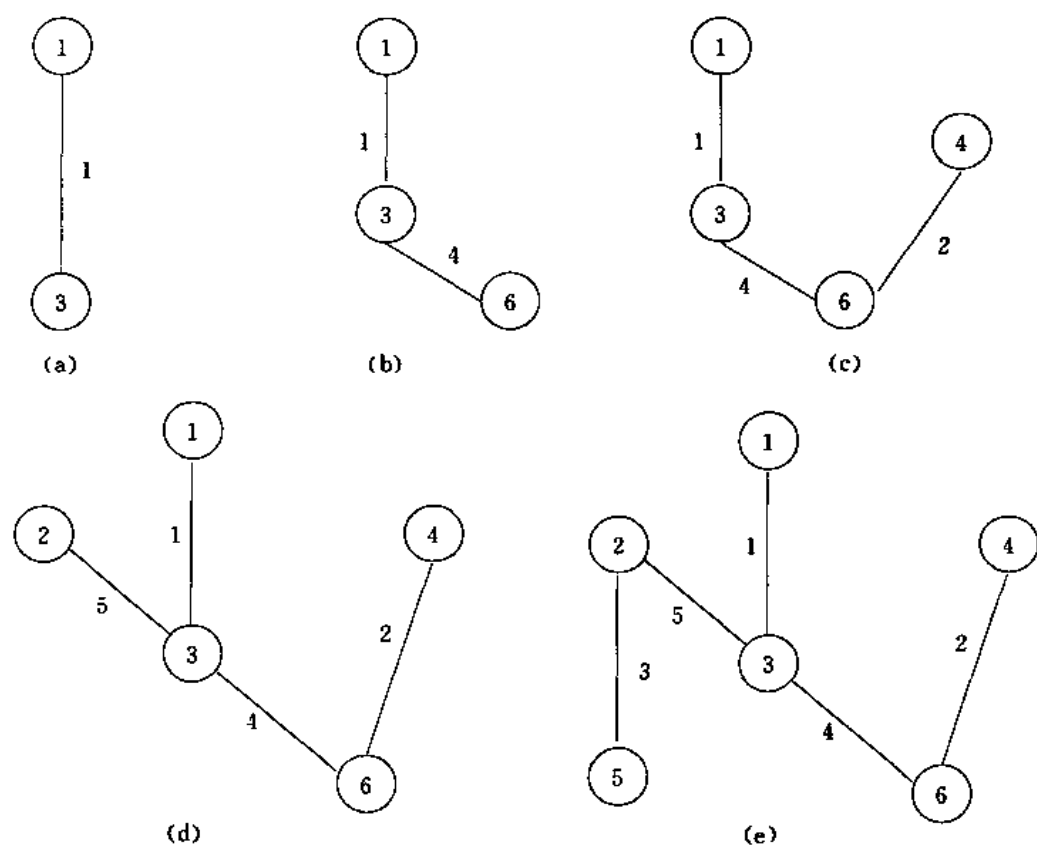


图 9.13 构造最小生成树的过程

4. 使用克鲁斯卡尔算法构造出如图 9.14 所示的图 G 的一棵最小生成树。

解: 使用克鲁斯卡尔算法构造最小生成树的过程如图 9.15 所示。

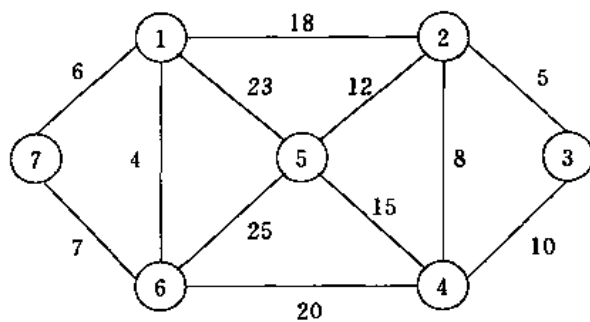


图 9.14 一个无向图 G

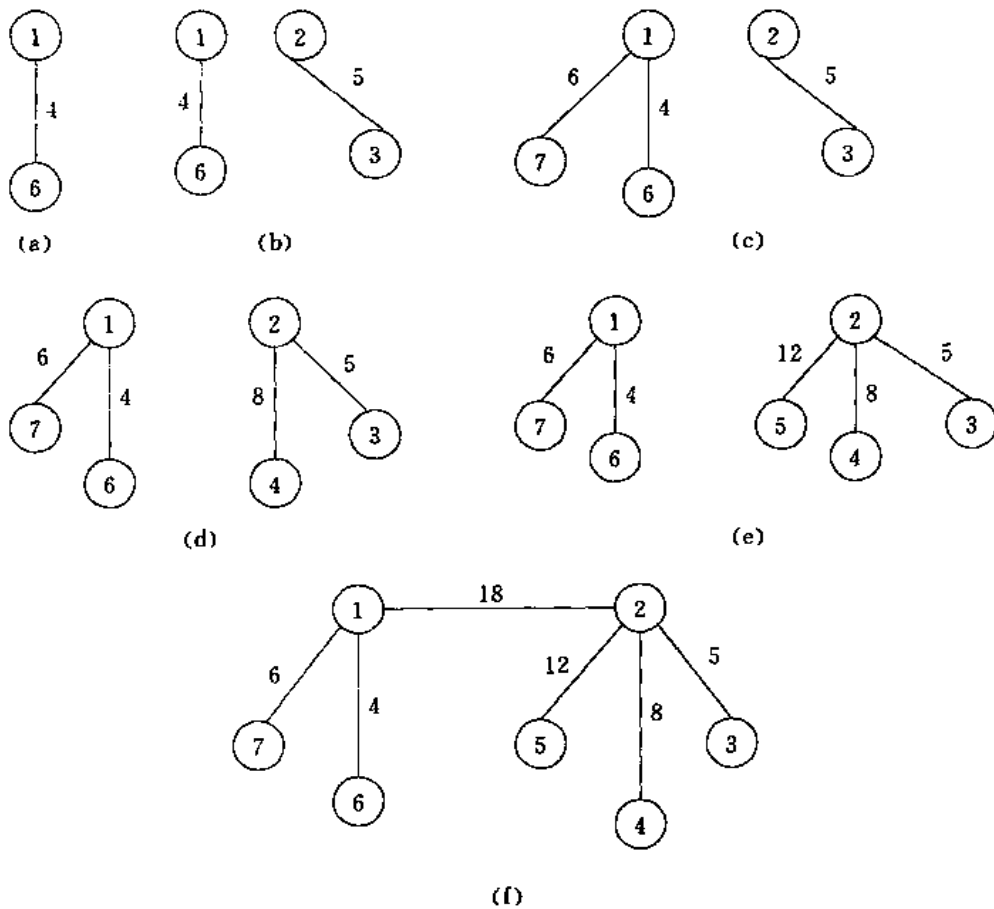


图 9.15 构造最小生成树的过程

* 5. 如图 9.16 所示给出了图 G 及对应的邻接表, 根据给定的深度优先搜索 (dfs) 算法:

```
#define MAXVEX 100      /* 定义最多顶点数 */
int gl[MAXVEX][MAXVEX];
void dfs(int v)
{
    struct vexnode *p;
    printf("%d", v);
    visited[v]=1;
    p=gl[v]->link;    {gl 是该图的邻接表的表头指针数组}
    while (p!=NULL)
    {
        if (visited[p->adjvex]==0) dfs(p->adjvex);
        p=p->next;
    }
}
```

(1) 从顶点 8 出发, 求出其搜索序列。

(2) 指出 p 的整个变化过程。

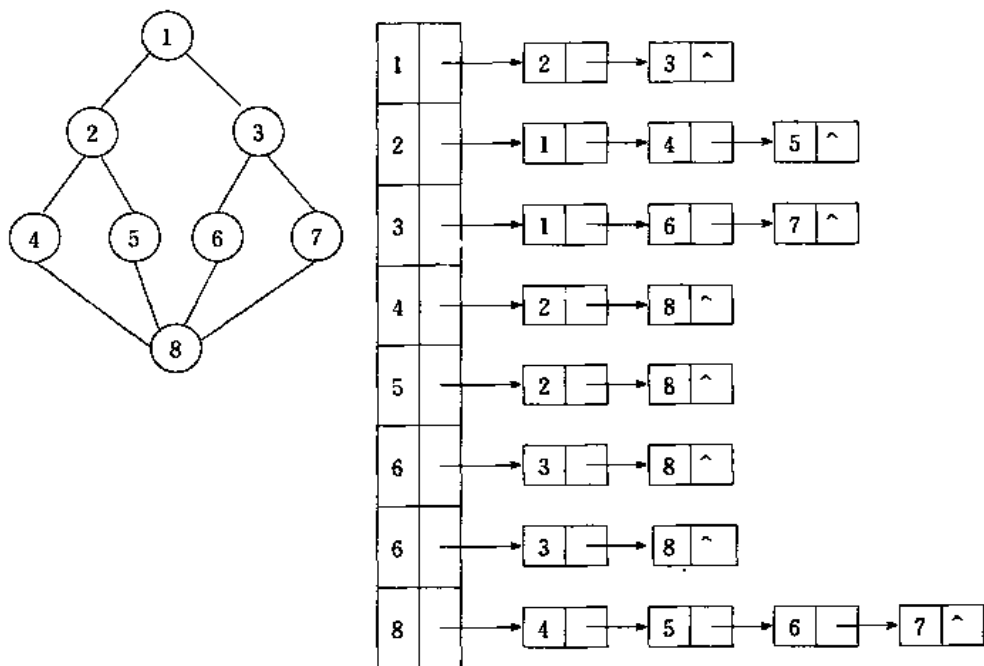


图 9.16 无向图 G 及对应的邻接表

解:(1)从顶点 8 出发的搜索序列为:8,4,2,1,3,6,7,5

(2)p 的整个变化过程为:

p=8(输出) p=4(输出) p=2(输出)

p=1(输出) p=2 p=3(输出) p=1

p=6(输出) p=3 p=8 p=7(输出)

p=3 p=8 p=5(输出)

6. 编写一个函数根据用户输入的偶对(以输入 0 表示结束)建立其有向图的邻接表。

解:本题的算法思想是:先产生邻接表的 n 个头结点(其结点数值域从 1 到 n),然后接受用户输入的 $\langle v_i, v_j \rangle$ (以其中之一为 0 标志结束),对于每条这样的边,申请一个邻接结点,并插入到 v_i 的单链表中,如此反复,直到将图中所有边处理完毕,则建立了该有向图的邻接表。

因此,实现本题功能的函数如下:

```
void creatadjlist(adjlist g)
{
    int i, j, k;
    struct vexnode *s;
    for (k=1; k<=n; k++)          /* 给头结点赋初值 */
    {
        g[k].data=k;
        g[k].link=NULL;
    }
}
```

```

printf("输入一个偶对:");
scanf("%d,%d",&i,&j);
while (i!=0 && j!=0)
{
    s=(struct vexnode *)malloc(sizeof(vexnode));    /* 产生一个单链表结点 s */
    s->adjvex=j;    /* 将 s 插到 i 为表头的单链表的最前面 */
    s->next=g[i].link;    /* 将 s 插入 */
    g[i].link=s;
    printf("输入一个偶对:");
    scanf("%d,%d",&i,&j);
}
}

```

7. 编写一个实现连通图 G 的深度优先遍历(从顶点 v 出发)的非递归函数。

解:本题的算法思想是:第一步,首先访问图 G 的指定起始顶点 v;第二步,从 v 出发,访问一个与 v 邻接的 p 所指顶点后,再从 p 所指顶点出发,访问与 p 所指顶点邻接且未被访问的顶点 q,然后从 q 所指顶点出发,重复上述过程,直到找不到存在未访问过的邻接顶点为止;第三步,退回到尚有未被访问过的邻接点的顶点,从该顶点出发,重复第二、三步,直到所有被访问过的顶点的邻接点都已被访问为止。为此,用一个栈(stack)保存被访问过的结点,以便回溯查找已被访问结点的未被访问过的邻接点。

因此,实现本题功能的函数如下:

```

#define MAXVEX 100    /* 定义最多顶点数 */
void dfs(adjlist g,int v,int n)
{
    struct vexnode *stack[MAXVEX],*p;
    int visited[MAXVEX],top=0;
    for (i=1;i<=n;i++) visited[i]=0;    /* 结点访问标志均置成 0 */
    printf("%d ",v);
    p=g[v].link;
    while (top>0 || p!=NULL)
    {
        while (p!=NULL)
        {
            if (visited[p->adjvex]==0) p=p->next;
            else {
                printf("%d ",p->adjvex);
                visited[p->adjvex]=1;
                top++;    /* 将访问过的结点入栈 */
                stack[top]=p;
                p=g[p->adjvex].link;
            }
        }
        if (top>0)
        {

```

```

        top--;          /* 退栈,回溯查找已被访问结点的未被访问过的邻接点 */
        p=stack[top];
        p=p->next;
    }
}
}

```

* 8. 设计一个函数利用遍历图的方法输出一个无向图 G 中从顶点 v_i 到 v_j 的长度为 l 的简单路径,假设无向图采用邻接表存储结构。

解:本题的算法思想是:利用深度优先搜索遍历,设栈(stack)保存遍历路径上的顶点,并以 d 记下当前的路径长度。从 $v=v_i$ 出发,找 v 的邻接顶点 w ,若 w 已访问过,则找下一个邻接顶点;若 $w=v_i$,且 $d=l-1$,则路径即为所求;若 $w \neq v_i$,且 $d < l-1$,则从 w 出发继续遍历,否则找下一个邻接顶点,若找不到下一个邻接顶点(设 $w=0$),则退栈,找前一顶点的下一个邻接顶点,若栈空,则说明没有这样的路径。

因此,实现本题功能的函数如下:

```

#define MAXVEX 100          /* 定义最多顶点数 */
void path(adj, i, j, l)
adjlist adj;
int i, j, l;
{
    int v, w, top=0, d=0, head;
    int stack[MAXVEX], visited[MAXVEX];
    struct vexnode * p;
    v=vi;
    visited[vi]=1;
    head=1;          /* head 在邻接表头第一次取邻接顶点时为 1, 否则为 0 */
    do               /* w 为 v 在图中的邻接点, 若邻接点已查遍则 w=0 */
    {
        if (head) p=adj[v]->link;
        else
        {
            head=0;
            p=p->next;
        }
        if (p==NULL) w=0;
        else w=p->adjvex;
        if (w!=0)          /* v 还存在未查找的邻接点 */
            if (visited[w]==0) /* w 未查找过 */
                if (w==vj && d==l-1)
                {
                    d++;
                    top++;
                }
            }
        else
        {
            head=1;
            w=0;
        }
    }
}

```

```

        stack[top]=v;
    }
    if (w!=j && d<1-1)
    {
        d++;
        top++;
        stack[top]=v;
        visited[w]=1;
        v=w;
        head=1;
    }
    else if (top>0)
    {
        visited[v]=0;
        v=stack[top];
        head=1;
        top--;
        d--;
    }
} while ((top!=0 || w==0) && (d!=1));
if (top>0)
    for (i=1;i<=top;i++) /* 输出从栈底到栈顶的顶点序列 */
        printf("%d ",stack[i]);
    else printf("没有这样的路径! \n");
}

```

9. 假设有 n 个城市组成一个公路网(有向的),并用代价邻接矩阵表示该网络,编写一个从指定城市 v 到其他各城市的最短路径的函数。

解:本题的算法思想是:假设源点为 v :

- (1)置集合 s 的初态为空
- (2)把顶点 v 放入集合 s 中
- (3)确定从 v 开始的 $n-1$ 条路径:
 - 选取最短距离的顶点 u
 - 把顶点 u 加入集合 s 中
 - 更改距离

因此,实现本题功能的函数如下:

```

#define MAXVEX 100 /* 定义最多顶点数 */
void shortestpath(v, cost, dist)
int v, cost[MAXVEX][MAXVEX];
int dist[MAXV]; /* 最终的 dist[j] (1≤j≤n) 为从顶点 v 到顶点 j 之间的最短路径长度 */
{
    int s[MAXVEX], i, u, num, w;

```



```

for i=1 to n do {置集合 s 的初态为空}
{
    s[i]=0;
    dist[i]=cost[v][i];
}
s[v]=1; /* 把顶点 v 放入集合 s */
dist[v]=0;
num=2;
while (num<n) /* 确定从顶点 v 开始的 n-1 条路径 */
{
    u=choose(s,dist,n);
    s[u]=1;
    num++; /* 把顶点 u 放入 s */
    for (w=1;w<=n;w++)
        if (!s[w])
            if (dist[u]+cost[u][w]<dist[w])
                dist[w]=dist[u]+cost[u][w];
}
}

```

其中 choose() 返回一个满足条件: $\text{dist}[u] = \text{minimum}(\text{dist}[w])$, 且 $s[w]=0$ 的顶点 u , 该函数如下:

```

int choose(s,dist,n)
int s[],dist[],n;
{
    int min,i=1,u;
    while (s[i]==1) i++;
    min=dist[i];
    u=i;
    while (i<=n)
    {
        if (s[i]==1) i++;
        else if (min>dist[i])
        {
            u=i;
            min=dist[i];
        }
        i++;
    }
    choose=u;
}

```

10. 假设图采用邻接表存储,编写一个函数利用深度优先搜索法求出无向图中通过给定

点 v 的简单回路。

解: 本题的算法思想是: 从给定顶点 v_0 出发进行深度优先搜索, 在搜索过程中判别当前访问的顶点是否为 v_0 , 若是, 则找到一条回路; 否则继续搜索。为此, 设一个顺序栈 $cycle$ 记录构成回路的顶点序列, 把访问顶点的操作改为将当前访问的顶点入栈; 相应地, 若从某一顶点出发搜索完再回溯, 则做退栈操作, 同时要求找到的回路的路径应大于 2。另外还设置一个 $found$, 其初值为 0, 当找到回路后为 1。

因此, 实现本题功能的函数如下:

```
#define MAXVEX 100          /* 定义最多顶点数 */
void dfscycle(adj, v0)
adjlist adj;
int v0;
{
    int i, j, top = 0, v = v0, found = 0;
    int visited[MAXVEX], cycle[MAXVEX];
    struct edgenode * p;
    i = 1;
    cycle[i] = v;          /* 从 v 点开始搜索 */
    visited[v] = 1;
    p = adj[v] -> link;
    while ((p != NULL || top > 0) && ! found)
    {
        while (p != NULL && ! found)
        {
            if (p -> adjvex == v0 && i > 2) found = 1;      /* 找到路径长度大于 2 的回路 */
            else if (visited[p -> adjvex] == 0) p = p -> next /* 找下一个邻接点 */
            else
            {
                w = p -> adjvex;    /* 记下路径, 继续搜索 */
                visited[w] = 1;
                i++;
                cycle[i] = w;
                top++;
                stack[top] = p;
                p = adj[w] -> link;
            }
        }
        if (! found && top > 0)      /* 沿原路径退回另选路径进行搜索 */
        {
            p = stack[top];
            top--;
            p = p -> next;
            i--;
        }
    }
}
```

```

if (found)
{
    for (j=1;j<=l;i++) printf("%d,",cycle[j]);    /* 打印回路的顶点序列 */
    printf("%d\n",v);
}
else printf("没有通过给定点 v 的回路! \n");
}

```

11. 编写一个函数计算给定的有向图的邻接矩阵的每对顶点之间的最短路径。

解: 本题采用弗洛伊德算法, 其思想用以下数学表达式描述:

$$A^{(k)}[i,j] = \min(A^{(k-1)}[i,j], A^{(k-1)}[i,k] + A^{(k-1)}[k,j]) \quad (1 \leq i \leq n, 1 \leq j \leq n)$$

其中 k 表示第 k 次迭代运算, $A^{(0)}[i,j] = A[i,j]$ 。

该式是一个迭代表达式, 每迭代一次, 在从顶点 v_i 到顶点 v_j 的最短路径就多考虑一个顶点, 经过 n 次迭代后所得的 $A[i][j]$ 值就是顶点 v_i 到顶点 v_j 的最短路径。实现本题功能的函数如下:

```

#define MAXVEX 100          /* 定义最多顶点数 */
void floyd(a,c,n) /* 已知有向图的邻接矩阵为 c, 求最短路径矩阵 a, n 为顶点数 */
int a[MAXVEX][MAXVEX], c[MAXVEX][MAXVEX], n;
{
    int i,j,k;
    for (i=1;i<=n;i++)      /* 给 a 赋初值 */
        for (j=1;j<=n;j++)
            a[i][j]=c[i][j];
    for (l=1;l<=n;l++)
        a[l][l]=0;
    for (k=1;k<=n;k++)
        for (i=1;i<=n;i++)
            for (j=1;j<=n;j++)
                if (a[i][k]+a[k][j]<a[i][j])
                    a[i][j]=a[i][k]+a[k][j];
}

```

第10章 查 找

查找是指在某种数据结构中找出满足给定条件的结点,若找到满足给定条件的结点,则查找成功,否则查找失败。查找是数据结构中很常用的基本运算。

10.1 基本查找方法

10.1.1 顺序查找

顺序查找又称为线性查找,是一种最简单的查找方法。它是从线性表的一端开始,顺序扫描线性表,依次将扫描到的结点关键字和给定值 k 相比较,若当前扫描到的结点关键字与 k 相等,则查找成功;若扫描结束后,仍未找到关键字等于 k 的结点,则查找失败。

顺序查找的线性表定义如下:

```
#define MAXITEM 100          /* 最多项数 */
struct element
{
    KeyType key;              /* 关键字 */
    ElemType data;            /* 其他数据 */
};
typedef struct sqlist[ITEM];
```

这里的 KeyType 和 ElemType 可以是任何相应的数据类型如 int, float 或 char 等,在算法中,我们规定 KeyType 和 ElemType 缺省是 int 类型。

顺序查找的函数如下,其功能是在线性表 r 中顺序查找关键字为 k 的结点,若找到了,返回其位置 i ;若找不到,返回 0:

```
int seqsearch(r, k, n)
sqlist r;
int k, n;                      /* n 为线性表 r 中元素个数 */
{
    int i = 1;
    while (r[i].key != k) i++;
    if (i > n) i = 0;
    return(i);
}
```

算法分析:对于含有 n 个结点的线性表,结点的查找在等概率即 $P_i = 1/n$ 的前提下,成功的查找,平均查找长度为:

$$\begin{aligned}
 ASL &= \sum_{i=1}^n P_i * C_i \\
 &= \sum_{i=1}^n (1/n) * i \\
 &= (1/n) * (1+2+\dots+i) \\
 &= (n+1)/2
 \end{aligned}$$

10.1.2 二分查找

二分查找要求线性表中的结点必须按关键字值的递增或递减顺序排列。它首先用要查找的关键字 k 与中间位置的结点的关键字相比较,这个中间结点把线性表分成了两个子表,若比较结果相等则查找完成,若不相等,再根据 k 与该中间结点关键字的比较大小确定下一步查找哪个子表,这样递归进行下去,直到找到满足条件的结点或者该线性表中没有这样的结点。

二分查找的函数如下,其功能是在线性表 r 中二分查找关键字为 k 的结点,若找到了,返回其位置 i ;若找不到,返回 0:

```

int binsearch(r, kn)
sqlist r;
int k, n;                                /* n 为线性表 r 中元素个数 */
{
    int i, low=1, high=n, mid, find=0;
    while (low<=high && ! find)
    {
        mid=(low+high) / 2;
        if (k<r[mid].key) high=mid-1
        else if (k>r[mid].key) low=mid+1
        else {
            i=mid;
            find=1;
        }
    }
    if (! find) i=0;
    return(i);
}

```

算法分析:二分查找函数恰好是一条从判定树的根到被查结点的一条路径,而比较的次数恰好是树深。借助于二叉判定树很容易求得二分查找的平均查找长度。假设表长为 n , 树深 $h=\log_2(n+1)$, 所以平均查找长度为:

$$\begin{aligned}
 ASL &= \sum_{i=1}^n P_i C_i \\
 &= \frac{1}{n} \sum_{i=1}^n j * 2^{i-1} = \frac{n+1}{n} \log_2(n+1) - 1 \approx \log_2(n+1) - 1
 \end{aligned}$$

10.1.3 分块查找

分块查找又称为索引顺序查找,其性能介于顺序查找和二分查找之间。分块查找把线性表分成若干块,每一块中的元素存储顺序是任意的,但块与块之间必须按关键字大小有序排列,即前一块中的最大关键字小于后一块中的最小关键字值。另外还需要建立一个索引表,索引表中的一项对应线性表中的一块,索引项由键域和链域组成,键域存放相应块的最大关键字,链域存放指向本块第一个结点的指针。索引表按关键字值递增顺序排列。

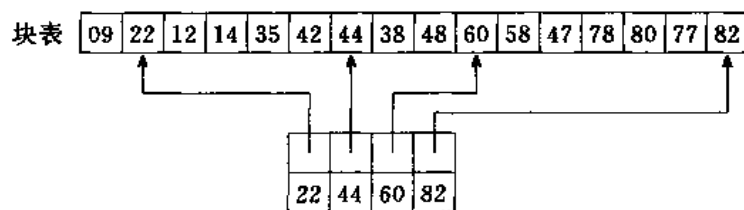
分块查找的查找函数分为两步进行,首先确定待查找的结点属于哪一块,即查找其所在的块;然后在块内查找要查的结点。由于索引表是递增有序的,采用二分查找时,块内元素个数较少,采用顺序法在块内查找,不会对执行速度有太大的影响。

索引表的定义如下:

```
struct indexterm
{
    KeyType key;
    int low,high;
};
typedef struct indexterm index[MAXITEM];
```

这里的 KeyType 可以是任何相应的数据类型如 int, float 或 char 等,在算法中,我们规定 KeyType 缺省是 int 类型。

例如,对于线性表中关键字为:9,22,12,14,35,42,44,38,48,60,58,47,78,80,77,82,其分块索引和索引表如图 10.1 所示。



(a) 分块索引

key	22	44	60	82
low	1	5	9	13
high	4	8	12	16

(b) 索引表

图 10.1 分块索引和索引表

分块查找的函数如下,其功能是在线性表 r 中分块查找关键字为 k 的结点,若找到了,返回其位置 i ;若找不到,返回 0:

```
int blksearch(r,idx,k,bn)
sqlist r;
```

```

index idx;
int k, bn;                /* bn 为块个数 */
{
    int l, low1=1, high1=bn, mid1, hb, find=0;
    while (low1<=high1 && ! find) /* 二分查找索引表 */
    {
        mid1=(low1+high1)/2;
        if (k<idx[mid1].key) high1=mid1-1;
        else if (k>idx[mid1].key) low1=mid1+1;
        else {
            high1=mid1-1;
            find=1;
        }
    }
    if (low1<bn)           /* k 小于索引表内最大值 */
    {
        i=idx[low1].low;   /* 在索引表中确定块起始地址 */
        hb=idx[low1].high; /* 在索引表中确定块终止地址 */
    }
    /* 在指定的块内采用顺序方法进行查找 */
    while (i<hb && r[i].key!=k) i++;
    if (r[i].key!=k) i=0;
    return(i);
}

```

算法分析:分块查找实际上进行两次查找,则整个算法的平均查找长度是两次查找的平均查找长度之和。

假设有 n 个结点,分成 bn 块,每块有 s 个结点,即 $bn=n/s$,每块的查找概率为 $1/bn$,块内每个结点的查找概率为 $1/s$ 。

$$\begin{aligned}
 ASL &\approx \log_2(bn+1)-1+(s+1)/2 \\
 &\approx \log_2(n/s+1)+s/2
 \end{aligned}$$

10.1.4 哈希表查找

哈希表查找不同于前面的几种查找方法,它是通过对记录的关键字值进行某种运算,直接求出记录文件的地址,是关键字到地址的直接转换方法,而不需要反复比较。

假设 f 包含 n 个结点, R_i 为其中的某个结点 ($1 \leq i \leq n$), key_i 是其关键字值,若在关键字值 key_i 与结点 R_i 的地址之间建立某种函数关系,则便可通过这个函数把关键字值转换成相应结点的地址,即有:

$$addr(R_i) = H(key_i)$$

其中, $addr(R_i)$ 称为哈希函数。

1. 常用的的哈希函数

哈希函数的种类很多,这里介绍几种常用的。

(1) 直接地址法

直接地址法的哈希函数 H 对于关键字是数字类型的文件,直接利用关键字求得哈希地址。

$$H(\text{key}) = \text{key} + c$$

在使用时,为了使哈希地址与存储空间吻合,可以调整 c 。

(2) 数字分析法

数字分析法是假设有一组关键字,每个关键字由 n 位数字组成,如 $k_1 k_2 \dots k_n$ 。数字选择法是从中提取数字分布比较均匀的若干位作为哈希地址。

(3) 除留余数法

除留余数法是用关键字 k 除以散列表长度 m 所得余数作为散列地址的方法。对应的散列函数 $H(k)$ 为:

$$H(k) = k \% m$$

(4) 平方取中法

平方取中法是取关键字平方的中间几位作为散列地址的方法,具体取多少位视实际情况而定。

(5) 折叠法

折叠法是首先把关键字分割成位数相同的几段(最后一段的位数可少一些),段的位数取决于散列地址的位数,由实际情况而定,然后将它们的叠加和(舍去最高进位)作为散列地址的方法。

2. 冲突解决方法

解决冲突的方法有两大类:开放地址法和链地址法。

(1) 开放地址法

开放地址法又分为线性探测再散列、二次探测再散列和随机探测再散列。

假设哈希表空间为 $T[0, m-1]$, 哈希函数为 $H(\text{key})$ 。

线性探测再散列解决冲突求“下一个”地址公式是:

$$d_1 = H(\text{key})$$

$$d_{j+1} = (d_j + 1) \% m; j = 1, 2, \dots$$

二次探测再散列解决冲突求“下一个”地址公式是:

$$d_1 = H(\text{key})$$

$$d_{2j} = (d_1 + j^2) \% m$$

$$d_{2j+1} = (d_1 - j^2) \% m \quad j = 1, 2, \dots$$

随机探测再散列解决冲突求“下一个”地址公式是:

$$d_i = H(\text{key})$$

$$d_{j+1} = (d_i + R) \% m$$

$$\text{其中 } R = \begin{cases} \text{伪随机序列} & R_1, R_2, \dots \\ \text{伪随机公式} & R_{i+1} = (aR_i + P) \% Q \end{cases}$$

(2) 链地址法

当存储结构是链表时,多采用链地址法,用链地址法处理冲突的方法是:把具有相同散列地址的关键字值放在同一个链表中,称为同义词链表。通常把具有相同哈希地址的关键字都存放在一个同义词链表中,有 m 个散列地址就有 m 个链表,同时用数组 $t[m]$ 存放各个链表的头指针,凡是散列地址为 i 的记录都以结点方式插入到以 $t[i]$ 为指针的单链表中。

10.1.5 背包问题及其求解函数

背包问题:假设有 n 件物品,记作 d_1, d_2, \dots, d_n , 对应于每个 $d_i (1 \leq i \leq n)$ 都有一个整数重量 $w_i > 0$ 和一个整数价值 $v_i > 0$, 如何从这 n 件物品中挑选某几件物品装进背包,使总重量不超过给定的重量 P , 同时使背包中的物品的价值尽可能高。

背包的装填工作可以这样进行:依次考虑物品 d_1, d_2, \dots, d_n , 或者把它装进背包, 或者不装进背包。如果这样做的话,背包的装填情况就可以用三元组(重量, 价值, m)表示, 其中“重量”是背包中物品的当前重量,“价值”是背包中物品的当前价值, m 表示已经考虑过 d_1, d_2, \dots, d_m 等物品。

为简单起见,以下假定相对价值:

$$v_1/w_1, v_2/w_2, \dots, v_n/w_n$$

是按递减次序排列的,即:

$$v_1/w_1 \geq v_2/w_2 \geq \dots \geq v_n/w_n$$

并假定不会出现退化情况,即:

$$\sum_{i=1}^n w_i \leq P \quad P \text{ 为背包所能承受的重量}$$

和

$$\min\{w_i | 1 \leq i \leq n\} \geq P$$

现在,三元组就是上面所提到的“位置”。三元组 $(0, 0, 0)$ 是初始位置 A , 任何一个三元组(重量, 价值, n)都是终止位置。求给定的位置 $s = (\text{重量}, \text{价值}, m)$ 的后件的规则是:如果 $m = n$, 则 s 无后件,它是一个终止位置;否则, s 有一个后件 $s_2 = (\text{重量}, \text{价值}, m+1)$, 并且只要“重量”+ $w_{m+1} \leq P$ (即只要 d_{m+1} 还可以装进背包时)就还有一个后件 $s_1 = (\text{重量} + w_{m+1}, \text{价值} + v_{m+1}, m+1)$ 。

如果想从一切可以由 A 推出的终止位置(重量, 价值, n)中找出价值最高的终止位置(即以最高价值装填背包),可以用回溯法从对应的解答树中找出要求的解。

在背包问题的解答树中,如果物品 d_{m+1} 放进背包,则结点(重量+ w_{m+1} , 价值+ v_{m+1} , $m+1$)和(重量, 价值, $m+1$)分别是结点(重量, 价值, m)的左、右后件;否则(重量, 价值, m)只有右后件(重量, 价值, $m+1$)。

当重量限制为 $t(1 \leq t \leq P)$ 时,可以由 $n-m$ 种物品 $d_{m+1}, d_{m+2}, \dots, d_n(1 \leq m \leq n)$ 得到价值的上界 $f(t, m)$:

$$f(t, m) = \begin{cases} 0 & \text{当 } m=n \text{ 或 } t=0 \\ f(t-w_{m+1}, m+1) + v_{m+1} & \text{当 } t \geq w_{m+1} \\ (t * v_{m+1}) / w_{m+1} & \text{当 } t < w_{m+1} \end{cases}$$

从上式看到 $f(t, m)$ 是优先考虑 d_m 后面相对价值最高的物品(因为物品的编号是按相对价值从大到小的次序排列的),必要时最后还得“分割”一种物品,才能正好装入 t 单位的重量。因此,从结点 $k=(\text{重量}, \text{价值}, m)$ 出发无法使装填的背包的价值大于 $u(k)=\text{价值}+f(P-\text{重量}, m)$ 。

所以,如果 y 是以某种已知的方法装填背包所得的价值,那就不必再考虑使 $u(k) < y$ 的各个结点 k 的后件,这样可以大大加快检查解答树的结点的速度。

下面给出求解背包问题的程序,该程序从左到右检查解答树的树枝,并使用上述的 $f(t, m)$ 计算估计值 $u(k)$,根据 $u(k)$ 的大小除去不必要的结点。背包问题的输入数据是 $n, w_1, v_1, w_2, v_2, \dots, w_n, v_n, p$ 。用数组 $w_v[1..n]$ 存放输入数据 $w_i, v_i(1 \leq i \leq n)$,并使用一个栈 $s[1..2 * n]$,它的栈顶指针是 top ,栈中的单元用来存放解答树中的结点,每个结点都是一个三元组(重量,价值, m)。栈 s 保留着通向当前正在检查的结点的整条树枝,该树枝上所有结点的第三个分量都标上负号,以便同查找函数中所需的其它结点区别开来。当检查到终止结点时,如果当前背包中的物品的价值比以前的装填方案所得到的价值大,只要打印当前被检查的树枝上的结点,就能知道当前最佳装填方案是什么。

```
#include <stdio.h>
#define MAXN 100
#define MAXN2 200
struct wvtype
{
    int w, v;
};
struct stackitem
{
    int w, v, m;
};
struct stackitem s[MAXN2];
struct wvtype wv[MAXN];
int top, n, i, aux, p = 0, y;
int f(int t, int m)
{
    int s;
    if (m == 0 || t == 0) s = 0;
    else if (t >= wv[m+1].w) s = f(t - wv[m+1].w, m+1) + wv[m+1].v;
    else s = (t * wv[m+1].v) / wv[m+1].w;
    return(s);
}
```

```

}

main()
{
    printf("输入物品种类:");
    scanf("%d",&n);
    for (l=1;i<=n;i++)
    {
        printf("\t第%d种物品(重量,价值):",i);
        scanf("%d,%d",&wv[i].w,&wv[i].v);
    }
    printf("输入背包所能承受的总重量:");
    scanf("%d",&p);
    top=1; /* 根结点的右后件进栈 */
    s[top].w=0;
    s[top].v=0;
    s[top].m=1;
    if (wv[top].w<=p)
    { /* 根结点的左后件进栈 */
        top++;
        s[top].w=wv[1].w;
        s[top].v=wv[1].v;
        s[top].m=1;
    }
    while (top!=0)
        if (s[top].m==n)
        {
            if (s[top].v>y) /* 得到最佳装填方案 */
            {
                y=s[top].v;
                s[top].m=-s[top].m;
                printf("最佳装填方案是:\n");
                for (i=1;i<=top;i++)
                    if (s[i].m<0) printf("%d,%d,%d\n",s[i].w,s[i].v,-s[i].m);
            }
            top--;
        }
        else if (s[top].m>=0)
            if (s[top].v+f(p-s[top].w,s[top].m)<=y) top--;
            else { /* 产生当前结点的右后件且进栈 */
                s[top+1].w=s[top].w;
                s[top+1].v=s[top].v;
                s[top+1].m++;
            }
    }
}

```

```

s[top].m/*给前件作标记 */
top++;
if (s[top].w+vv[s[top].m].w<=p)
{ /*产生当前结点的左后件且进栈 */
    aux=s[top].m;
    s[top+1].w=s[top].w+vv[aux].w;
    s[top+1].v=s[top].v+vv[aux].v;
    s[top+1].m=aux;
    top++;
}
}
else top--;
}

```

10.2 基本题

10.2.1 单项选择题

1. 顺序查找法适合于存储结构为 ① 的线性表。

- A. 散列存储 B. 顺序存储或链接存储
C. 压缩存储 D. 索引存储

答:①B

2. 对线性表进行二分查找时,要求线性表必须 ①。

- A. 以顺序方式存储
B. 以链接方式存储
C. 以顺序方式存储,且结点按关键字有序排序
D. 以链接方式存储,且结点按关键字有序排序

答:①C

3. 采用顺序查找方法查找长度为 n 的线性表时,每个元素的平均查找长度为 ①。

- A. n B. $n/2$ C. $(n+1)/2$ D. $(n-1)/2$

答:①C

4. 采用二分查找方法查找长度为 n 的线性表时,每个元素的平均查找长度为 ①。

- A. $O(n^2)$ B. $O(n\log_2 n)$ C. $O(n)$ D. $O(\log_2 n)$

答:①D

5. 二分查找和二叉排序树的时间性能 ①。

- A. 相同 B. 不相同

答:①B

6. 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82为的

结点时,①次比较后查找成功。

- A. 1 B. 2 C. 4 D. 8

答:①C

7. 设哈希表长 $m=14$, 哈希函数 $H(key)=key \% 11$ 。表中已有 4 个结点:

$addr(15)=4$

$addr(38)=5$

$addr(61)=6$

$addr(84)=7$

其余地址为空

如用二次探测再散列处理冲突,关键字为 49 的结点的地址是 ①。

- A. 8 B. 3 C. 5 D. 9

答:①D

8. 有一个长度为 12 的有序表,按二分查找法对该表进行查找,在表内各元素等概率情况下查找成功所需的平均比较次数为 ①。

- A. 35/12 B. 37/12 C. 39/12 D. 43/12

答:①B

(依题意,构造一棵有序二叉树,共 12 个结点,第一层 1 个结点,第二层 2 个结点,第三层 4 个结点,第四层 5 个结点,则: $ASL=(1*1+2*2+3*4+4*5)/12=37/12$)

9. 采用分块查找时,若线性表中共有 625 个元素,查找每个元素的概率相同,假设采用顺序查找来确定结点所在的块时,每块应分 ① 个结点最佳。

- A. 10 B. 25 C. 6 D. 625

答:①B

10. 如果要求一个线性表既能较快地查找,又能适应动态变化的要求,可以采用 ① 查找方法。

- A. 分块 B. 顺序 C. 二分 D. 散列

答:①A

10.2.2 填空题(将正确的答案填在相应的空中)

1. 顺序查找法的平均查找长度为 ①;二分查找法的平均查找长度为 ②;分块查找法(以顺序查找确定块)的平均查找长度为 ③;分块查找法(以二分查找确定块)的平均查找长度为 ④;哈希表查找法采用链接法处理冲突时的平均查找长度为 ⑤。

答:① $(n+1)/2$ ② $((n+1) * \log_2(n+1))/n-1$ ③ $(s^2+2s+n)/2s$

④ $\log_2(n/s+1)+s/2$ ⑤ $1+\alpha$ (α 为装填因子)

2. 在各种查找方法中,平均查找长度与结点个数 n 无关的查法方法是 ①。

答:① 哈希表查找法

3. 二分查找的存储结构仅限于 ①,且是 ②。

答:①顺序存储结构 ②有序的

4. 在分块查找方法中,首先查找 ①,然后再查找相应的 ②。

答:①索引 ②块

5. 长度为 255 的表,采用分块查找法,每块的最佳长度是 ①。

答:①15

6. 在散列函数 $H(\text{key}) = \text{key} \% p$ 中, p 应取 ①。

答:①素数

7. 假设在有序线性表 $A[1..20]$ 上进行二分查找,则比较一次查找成功的结点数为 ①,则比较二次查找成功的结点数为 ②,则比较三次查找成功的结点数为 ③,则比较四次查找成功的结点数为 ④,则比较五次查找成功的结点数为 ⑤,平均查找长度为 ⑥。

答:①1 ②2 ③4 ④8 ⑤5 ⑥3.7

(依题意,构造一棵有序二叉树,共 20 个结点,第一层 1 个结点,第二层 2 个结点,第三层 4 个结点,第四层 8 个结点,第五层 5 个结点则: $ASL = (1 * 1 + 2 * 2 + 3 * 4 + 4 * 8 + 5 * 5) / 20 = 74 / 20 = 3.7$)

8. 对于长度为 n 的线性表,若进行顺序查找,则时间复杂度为 ①;若采用二分法查找,则时间复杂度为 ②;若采用分块查找(假定总块数和每块长度均接近 \sqrt{n}),则时间复杂度为 ③。

答:① $O(n)$ ② $O(\log_2 n)$ ③ $O(\sqrt{n})$

9. 在散列存储中,装填因子 α 的值越大,则 ①; α 的值越小,则 ②。

答:①存取元素时发生冲突的可能性就越大 ②存取元素时发生冲突的可能性就越小

10.3 习题解析

1. 设有一组关键字 {19, 01, 23, 14, 55, 20, 84, 27, 68, 11, 10, 77}, 采用哈希函数:

$$H(\text{key}) = \text{key} \% 13$$

采用开放地址法的线性探测再散列方法解决冲突,试在 0~18 的散列地址空间中对关键字序列构造哈希表。

解:依题意, $m=19$,线性探测再散列的下一地址计算公式为:

$$d_1 = H(\text{key})$$

$$d_{j+1} = (d_j + 1) \% m; j = 1, 2, \dots$$

其计算函数如下:

$$H(19) = 19 \% 13 = 6$$

$$H(01) = 01 \% 13 = 1$$

$$H(23) = 23 \% 13 = 10$$

$$H(14) = 14 \% 13 = 1 \text{ (冲突)}$$

$H(14) = (1+1) \% 19 = 2$
 $H(55) = 55 \% 13 = 3$
 $H(20) = 20 \% 13 = 7$
 $H(84) = 84 \% 13 = 6$ (冲突)
 $H(84) = (6+1) \% 19 = 7$ (仍冲突)
 $H(84) = (7+1) \% 19 = 8$
 $H(27) = 27 \% 13 = 1$ (冲突)
 $H(27) = (1+1) \% 19 = 2$ (冲突)
 $H(27) = (2+1) \% 19 = 3$ (仍冲突)
 $H(27) = (3+1) \% 19 = 4$
 $H(68) = 68 \% 13 = 3$ (冲突)
 $H(68) = (3+1) \% 19 = 4$ (仍冲突)
 $H(68) = (4+1) \% 19 = 5$
 $H(11) = 11 \% 13 = 11$
 $H(10) = 10 \% 13 = 10$ (冲突)
 $H(10) = (10+1) \% 19 = 11$ (仍冲突)
 $H(10) = (11+1) \% 19 = 12$
 $H(77) = 77 \% 13 = 12$ (冲突)
 $H(77) = (12+1) \% 19 = 13$

因此,各关键字的记录对应的地址分配如下:

$addr(01) = 1$
 $addr(14) = 2$
 $addr(55) = 3$
 $addr(27) = 4$
 $addr(68) = 5$
 $addr(19) = 6$
 $addr(20) = 7$
 $addr(84) = 8$
 $addr(23) = 10$
 $addr(11) = 11$
 $addr(10) = 12$
 $addr(77) = 13$

其他地址为空。

2. 设有一组关键字{19,01,23,14,55,20,84,27,68,11,10,77},采用哈希函数:

$H(key) = key \% 13$

采用开放地址法的二次探测再散列方法解决冲突,试在 0~18 的散列地址空间中对关键字序列构造哈希表。

解:依题意, $m=19$,二次探测再散列的下一地址计算公式为:

$$\begin{aligned}d_1 &= H(\text{key}) \\d_{2j} &= (d_1 + j * j) \% m \\d_{2j+1} &= (d_1 - j * j) \% m; j=1,2,\dots\end{aligned}$$

其计算函数如下:

$$\begin{aligned}H(19) &= 19 \% 13 = 6 \\H(01) &= 01 \% 13 = 1 \\H(23) &= 23 \% 13 = 10 \\H(14) &= 14 \% 13 = 1 \text{ (冲突)} \\H(14) &= (1 + 1 * 1) \% 19 = 2 \\H(55) &= 55 \% 13 = 3 \\H(20) &= 20 \% 13 = 7 \\H(84) &= 84 \% 13 = 6 \text{ (冲突)} \\H(84) &= (6 + 1 * 1) \% 19 = 7 \text{ (仍冲突)} \\H(84) &= (6 - 1 * 1) \% 19 = 5 \\H(27) &= 27 \% 13 = 1 \text{ (冲突)} \\H(27) &= (1 + 1 * 1) \% 19 = 2 \text{ (冲突)} \\H(27) &= (1 - 1) \% 19 = 0 \\H(68) &= 68 \% 13 = 3 \text{ (冲突)} \\H(68) &= (3 + 1 * 1) \% 19 = 4 \\H(11) &= 11 \% 13 = 11 \\H(10) &= 10 \% 13 = 10 \text{ (冲突)} \\H(10) &= (10 + 1 * 1) \% 19 = 11 \text{ (仍冲突)} \\H(10) &= (10 - 1 * 1) \% 19 = 9 \\H(77) &= 77 \% 13 = 12\end{aligned}$$

因此,各关键字的记录对应的地址分配如下:

$$\begin{aligned}\text{addr}(27) &= 0 \\ \text{addr}(01) &= 1 \\ \text{addr}(14) &= 2 \\ \text{addr}(55) &= 3 \\ \text{addr}(68) &= 4 \\ \text{addr}(84) &= 5 \\ \text{addr}(19) &= 6 \\ \text{addr}(20) &= 7 \\ \text{addr}(10) &= 9 \\ \text{addr}(23) &= 10 \\ \text{addr}(11) &= 11\end{aligned}$$

$\text{addr}(77)=12$

其他地址为空。

3. 设有一组关键字{19,01,23,14,55,20,84,27,68,11,10,77},采用哈希函数:

$$H(\text{key})=\text{key} \% 13$$

采用开放地址法的随机探测再散列方法解决冲突,试在 0~18 的散列地址空间中对关键字序列构造哈希表。

解:依题意, $m=19$,随机探测再散列的下一地址计算公式为:

$$d_1=H(\text{key})$$

$$d_i=(d_1+R) \% m$$

取伪随机序列为 3,54,11,36,19,...

其计算函数如下:

$$H(19)=19 \% 13=6$$

$$H(01)=01 \% 13=1$$

$$H(23)=23 \% 13=10$$

$$H(14)=14 \% 13=1 \text{ (冲突)}$$

$$H(14)=(1+3) \% 19=4$$

$$H(55)=55 \% 13=3$$

$$H(20)=20 \% 13=7$$

$$H(84)=84 \% 13=6 \text{ (冲突)}$$

$$H(84)=(6+3) \% 19=9$$

$$H(27)=27 \% 13=1 \text{ (冲突)}$$

$$H(27)=(1+3) \% 19=4 \text{ (仍冲突)}$$

$$H(27)=(1+54) \% 19=17$$

$$H(68)=68 \% 13=3 \text{ (冲突)}$$

$$H(68)=(3+3) \% 19=6 \text{ (仍冲突)}$$

$$H(68)=(3+54) \% 19=0$$

$$H(11)=11 \% 13=11$$

$$H(10)=10 \% 13=10 \text{ (冲突)}$$

$$H(10)=(10+3) \% 19=13 \text{ (仍冲突)}$$

$$H(77)=77 \% 13=12$$

因此,各关键字的记录对应的地址分配如下:

$$\text{addr}(68)=0$$

$$\text{addr}(01)=1$$

$$\text{addr}(55)=3$$

$$\text{addr}(14)=4$$

$$\text{addr}(19)=6$$

$\text{addr}(20)=7$
 $\text{addr}(84)=9$
 $\text{addr}(23)=10$
 $\text{addr}(11)=11$
 $\text{addr}(77)=12$
 $\text{addr}(10)=13$
 $\text{addr}(27)=17$

其他地址为空。

4. 线性表的关键字集合 $\{87, 25, 310, 08, 27, 132, 68, 95, 187, 123, 70, 63, 47\}$, 共有 13 个元素, 已知散列函数为:

$$H(k) = k \% 13$$

采用拉链法处理冲突。设计出这种链表结构, 并计算该表的成功查找的平均查找长度。

解: 依题意, 得到:

$H(87)=87 \% 13=9$
 $H(25)=25 \% 13=12$
 $H(310)=310 \% 13=11$
 $H(08)=08 \% 13=8$
 $H(27)=27 \% 13=1$
 $H(132)=132 \% 13=2$
 $H(68)=68 \% 13=3$
 $H(95)=95 \% 13=4$
 $H(187)=187 \% 13=5$
 $H(123)=123 \% 13=6$
 $H(70)=70 \% 13=5$
 $H(63)=63 \% 13=11$
 $H(47)=47 \% 13=8$

采用拉链法处理冲突的链接表如图 10.2 所示。

成功查找的平均查找长度:

$$\text{ASL} = (1 \times 10 + 2 \times 3) / 13 = 16 / 13 = 1 \frac{3}{13}$$

5. 设线性表的关键字集合 $\text{key} = \{32, 13, 49, 55, 22, 39, 20\}$, 选取哈希函数的方法为“除留余数法”, 解决冲突方法“线性探测再散列”, 请按上述条件求出 key 中各值的地址, 并求对该表的平均查找长度 ASL。

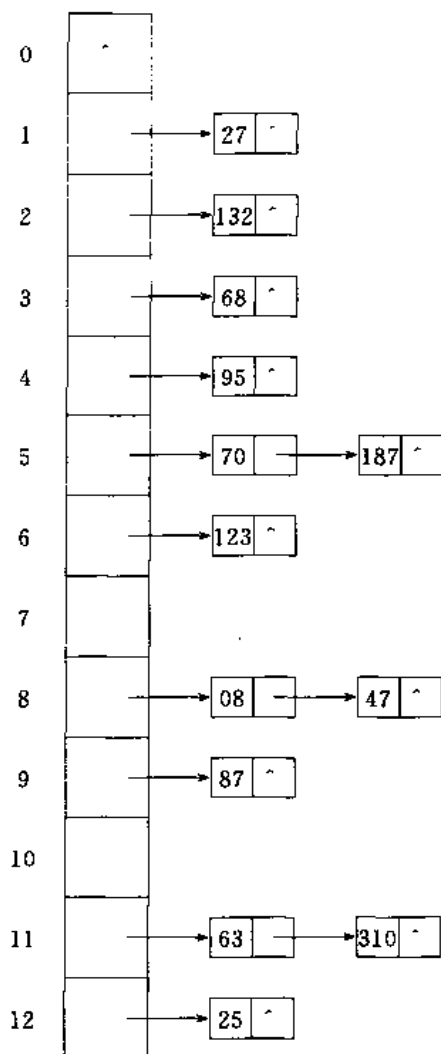


图 10.2 处理冲突的链接表

解:依题意,采用的哈希函数为:

$$H(\text{key}) = \text{key} \% 7$$

所以有:

$$H(32) = 32 \% 7 = 4$$

$$H(13) = 13 \% 7 = 6$$

$$H(49) = 49 \% 7 = 0$$

$$H(55) = 55 \% 7 = 6 \text{ (冲突)}$$

$$H(55) = (6 + 1) \% 8 = 7$$

$$H(22) = 22 \% 7 = 1$$

$$H(39) = 39 \% 7 = 4 \text{ (冲突)}$$

$$H(39) = (4 + 1) \% 8 = 5$$

$$H(20) = 20 \% 7 = 6 \text{ (冲突)}$$

$$H(20) = (6+1) \% 8 = 7 \text{ (仍冲突)}$$

$$H(20) = (6-1) \% 8 = 5 \text{ (仍冲突)}$$

$$H(20) = (6+4) \% 8 = 2$$

$$\text{平均查找长度 ASL} = (1 \times 4 + 2 \times 2 + 4) / 7 = \frac{12}{7} = 1 \frac{5}{7}$$

* 6. 使用的哈希函数:

$$H(\text{key}) = 3\text{key} \% 11$$

并采用开放地址法处理冲突, 随机探测再散列的下一地址计算公式为:

$$d_1 = H(\text{key})$$

$$d_i = (d_{i-1} + (7\text{key}) \% 11) \% 11 \quad (i=2, 3, \dots)$$

试在 0~10 的散列地址空间中对关键字序列(22, 41, 53, 46, 30, 13, 01, 67)构造哈希表, 并求等概率情况下查找成功的平均查找长度, 并设计构造哈希表的完整的函数。

解: 本题的哈希表如下:

0	1	2	3	4	5	6	7	8	9	10
22	7	41	30		53	46			13	01
1	7	1	2		1	1			2	2

因为 $H(\text{key}) = (3\text{key}) \% 11$

所以 $H(22) = 3 * 22 \% 11 = 0$

$$H(41) = 3 * 41 \% 11 = 2$$

$$H(53) = 3 * 53 \% 11 = 5$$

$$H(46) = 3 * 46 \% 11 = 6$$

$$H(30) = 3 * 30 \% 11 = 2 \text{ (冲突)}$$

$$H(30) = (2 + 7 \times 30) \% 11 = 3$$

$$H(13) = 3 * 13 \% 11 = 6 \text{ (冲突)}$$

$$H(13) = (6 + 7 \times 13) \% 11 = 9$$

$$H(01) = 3 * 01 \% 11 = 3 \text{ (冲突)}$$

$$H(01) = (3 + 7 \times 01) \% 11 = 10$$

$$H(67) = 3 * 67 \% 11 = 3 \text{ (冲突)}$$

$$H(67) = (3 + 7 \times 67) \% 11 = 10 \text{ (仍冲突)}$$

$$H(67) = (10 + 7 \times 67) \% 11 = 6 \text{ (仍冲突)}$$

$$H(67) = (6 + 7 \times 67) \% 11 = 2 \text{ (仍冲突)}$$

$$H(67) = (2 + 7 \times 67) \% 11 = 9 \text{ (仍冲突)}$$

$$H(67) = (9 + 7 \times 67) \% 11 = 5 \text{ (仍冲突)}$$

$$H(67) = (5 + 7 \times 67) \% 11 = 1$$

查找成功的平均查找长度为:

$$ASL = (4 \times 1 + 3 \times 2 + 1 \times 7) / 8 = 2.125$$

构造本哈希表的程序如下:

```
#include <stdio.h>
#define m 11
#define n 8
struct hterm
{
    int key;
    int si;
};
struct hterm hashlist[m];
int i,address,sum,d,x[n];
float average;
main()
{
    for (i=1;i<=(m-1);i++)
    {
        hashlist[i].key=0;
        hashlist[i].si=0;
    }
    x[1]=22; x[2]=41; x[3]=53;
    x[4]=46; x[5]=30; x[6]=13;
    x[7]=01; x[8]=67;
    for (i=1;i<=n;i++)
    {
        sum=0;
        address=(3*x[i])%m;
        d=address;
        if (hashlist[address].key==0)
        {
            hashlist[address].key=x[i];
            hashlist[address].si=1;
        }
        else
        {
            do
            {
                d=(d+(x[i]*7))%11;
                sum=sum+1;
                address=d;
            }
            while (hashlist[address].key!=0);
            hashlist[address].key=x[i];
            hashlist[address].si=1;
        }
    }
    average=sum/n;
    printf("average=%f\n",average);
}
```

```

    } while (hashlist[address].key != 0);
    hashlist[address].key = x[i];
    hashlist[address].si = sum + i;
}
}

printf("HashList Addr:");
for (i=0;i<=(m-1);i++) printf(" %3d",i);
printf("\n");
printf("HashList Key:");
for (i=0;i<=(m-1);i++) printf(" %3d",hashlist[i].key);
printf("\n");
printf("Search Length:");
for (i=0;i<=(m-1);i++) printf(" %3d",hashlist[i].si);
printf("\n");
average=0;
for (i=0;i<=(m-1);i++) average=average+hashlist[i].si;
average=average/n;
printf("Average Search Length:ASL(%d)=%.2f",n,average);
}

```

运算结果如下:

```

HashList Addr:  0  1  2  3  4  5  6  7  8  9 10
HashList Key:   22 67 41 30 0 53 46 0 0 13 1
Search Length:  1  7  1  2  0  1  1  0  0  2  2
Average Search Length: ASL(8)=2.12

```

* 7. 画出对长度为 10 的有序表进行二分查找的一棵判定树,并求其等概率时查找成功的平均查找长度。

解:依题意,假设长度为 10 的有序表为 a,进行二分查找的判定树如图 10.3 所示。

查找成功的平均查找长度:

$$ASL = (1 \times 1 + 2 \times 2 + 3 \times 4 + 4 \times 3) / 10 = 2.9$$

* 8. 编写一个函数,利用二分查找算法在一个有序表中插入一个元素 x,并保持表的有序性。

解:依题意,先在有序表 r 中利用二分查找算法查找关键字值等于或小于 x 的结点,mid 指向正好等于 x 的结点或 low 指向关键字正好大于 x 的结点,然后采用移动法插入 x 结点即可。

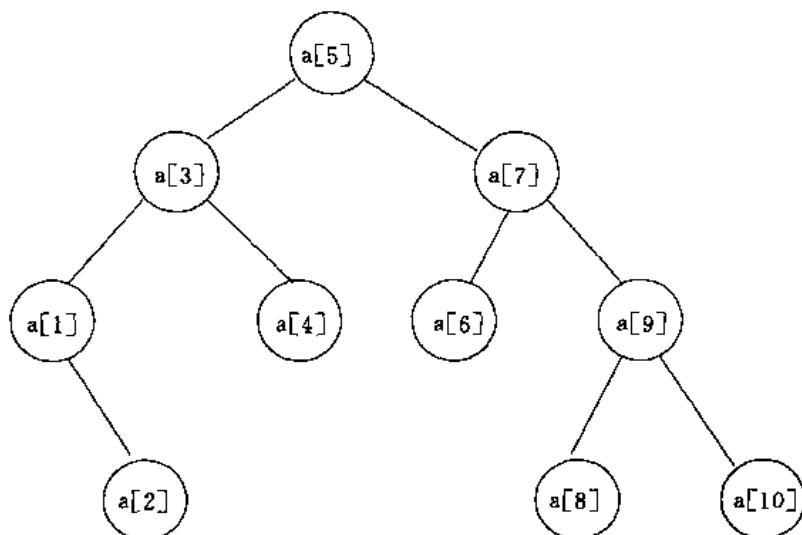


图 10.3 一棵判定树

因此,实现本题功能的函数如下:

```

bininsert(sqlist r, int x, int n)
{
    int low = 1, high = n, mid, inplace, i, find = 0;
    while (low <= high && ! find)
    {
        mid = (low + high) / 2;
        if (x < r[mid].key) high = mid - 1;
        else if (x > r[mid].key) low = mid + 1;
        else {
            i = mid;
            find = 1;
        }
    }
    if (find) inplace = mid;          /* 在 mid 所指的结点之前插入 x 结点 */
    else inplace = low; /* 此时 low 所指的关键字正好大于 x, 即在该结点之前插入 x 结点 */
    for (i = n; i >= inplace; i--)    /* 采用移动法插入 x 结点 */
        r[i + 1].key = r[i].key;
    r[inplace].key = x;
}

```

9. 设给定的散列表存储空间为 $H(1 \sim m)$, 每个 $H(i)$ 单元可存放一个记录, $H[i] (1 \leq i \leq m)$ 的初始值为 NULL, 选取的散列函数为 $H(R.key)$, 其中 $R.key$ 为 R 记录的关键字, 解决冲突方法为“线性探测法”, 编写一个函数将某记录 R 填入到散列表 H 中。

解: 为了简单, 只考虑记录仅包含一个 key 域的情况, 先计算地址 $H(R.key)$, 若无冲突, 则直接填入; 否则利用线性探测法求出下一地址:

$$d_1 = H(key)$$

$$d_{j+1} = (d_j + 1) \% (m + 1); j = 1, 2, \dots$$

直到找到一地址为 NULL, 然后再填入。

因此, 实现本题功能的函数如下:

```
hash()
{
    int i, j;
    j = H[R.key];
    if (H[j] == NULL) H[j] = R.key;
    else
    {
        do
        {
            j = (j + 1) \% (m + 1);
        } while (H[j] != NULL);
        H[j] = R.key;
    }
}
```

* 10. 假设按如下所述在有序的线性表中查找 x : 先将 x 与表中的第 $4j$ ($j=1, 2, \dots$) 项进行比较, 若相等, 则查找成功; 否则由该次比较求得比 x 大的一项 $4k$ 之后续而和 $4k-2$, 然后和 $4k-3$ 或 $4k-1$ 项进行比较, 直到查找成功。

(1) 给出实现上述算法的函数。

(2) 试画出当表长 $n=16$ 时的判定树, 并推导此查找方法的平均查找长度 (考虑查找元素等概率和 $n \% 4 = 0$ 的情况)。

解: 依题意, 实现本题功能的函数如下:

```
find(int a[], int x, int n)
{
    int i = 1, k = n / 4, found = 0;
    while (i <= k && ! found) /* x 与 a[1]~a[4k] 比较 */
    {
        if (a[4 * i] == x) found = 1;
        else if (x < a[4 * i])
        {
            if (x == a[4 * i - 2]) found = 1;
            else if (x < a[4 * i - 2])
            {
                if (x == a[4 * i - 1]) found = 1;
            }
            else if (x == a[4 * i - 3]) found = 1;
        }
        else i++;
    }
    /* x 与 a[4k+1], ..., a[4k+j] 比较 */
}
```



```
j=k%4;  
for (i=1;i<=j;i++)  
    if (x==a[4*k+j]) found=1;  
if (found) printf("查找成功\n");  
else printf("未查找\n");  
}
```

(3)表长 $n=16$ 时的判定树如图 10.4 所示。

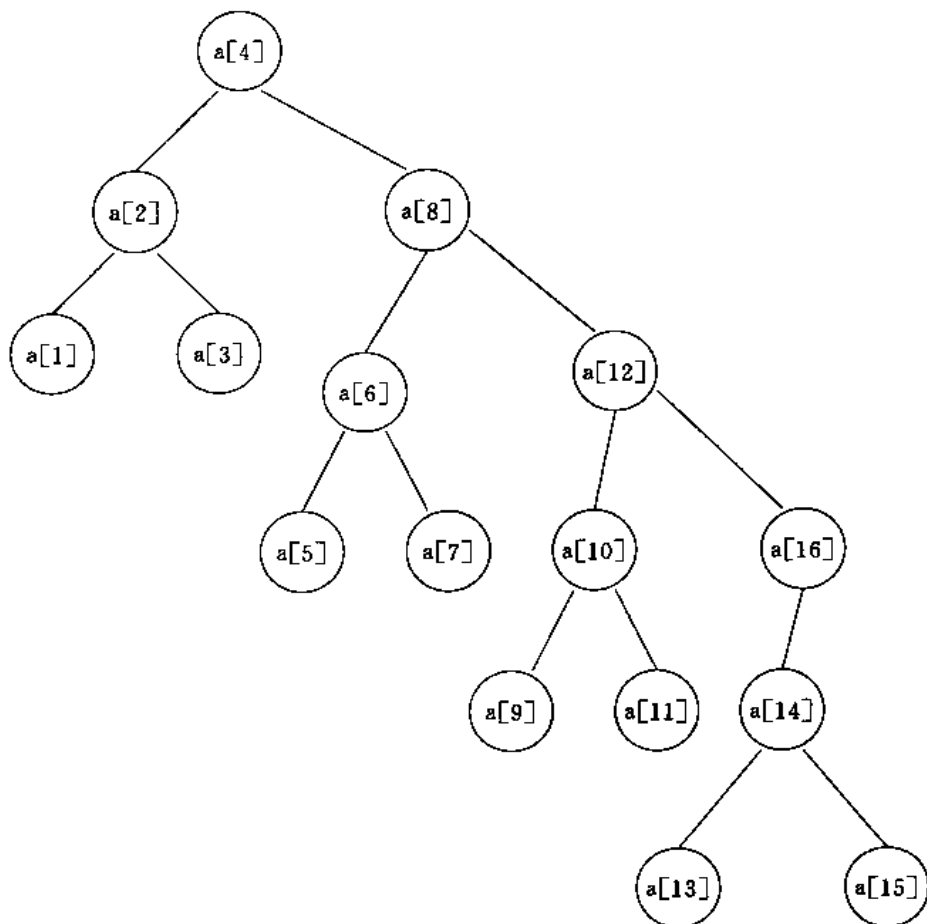


图 10.4 一棵判定树

第 11 章 内 排 序

排序就是把一组无序的记录按其关键字的某种次序排列起来,使其具有一定的顺序,便于进行数据查找。排序是数据处理中经常使用的一种重要运算,其方法很多,应用也很广泛。

排序全部放在内存中进行,不涉及外存的排序方法称为内排序。

11.1 基本排序方法

本小节讨论常用的几种排序方法及其时间复杂度。为了简便,本章讨论的排序均为升序排序,并假定要排序的记录均存储在线性表中,该线性表由排序关键字和其他域组成,其定义如下:

```
#define MAXITEM 100
struct rec
{
    KeyType key;           /* 关键字域 */
    ElemType data;         /* 其他数据域 */
};
struct rec sqliist[MAXITEM];
```

这里的 KeyType 和 ElemType 可以是任何相应的数据类型如 int, float 或 char 等,在算法中,我们规定 KeyType 和 ElemType 缺省是 int 类型。

如果在要排序的序列中,存在多个关键字相同的记录,例如:在 $\{r_1, r_2, \dots, r_n\}$ 序列中,有 $k_i = k_j$, 排序前为 $\{\dots, r_i, \dots, r_j, \dots\}$, 排序后这些记录的相对次序仍保持不变,即排序后为 $\{\dots, r_i, \dots, r_j, \dots\}$, 则称这种排序方法是稳定的, 否则排序后为 $(\dots, r_j, \dots, r_i, \dots)$, 称这种排序方法是不稳定的。

11.1.1 插入排序

插入排序的基本思想是:每一趟将一个待排序的记录,按其关键字值的大小插入到已经排序的部分文件中适当位置上,直到全部插入完成。

假设记录存放在 $r[1, n]$ 之中, $r[1, i-1]$ 是已排好序的记录, $r[i, n]$ 是没排序的记录。插入排序把未排序记录中的 $r[i]$ 插入到 $r[1, i-1]$ 之中,使 $r[1, i]$ 成为有序, $r[i]$ 的插入过程就是完成排序中的一趟。随着有序区的不断扩大,使 $r[1, n]$ 全部有序。实现插入排序的函数如下:

```
void insertsort(sqliist r, int n)
{
    int i, j;
```

```

for (i=2; i<=n; i++)
{
    r[0]=r[i];           /* r[0]是监视哨 */
    j=i-1;
    while (r[0].key<r[j].key) /* 进行元素移动,以便腾出一个位置插入 r[i] */
    {
        r[j+1]=r[j];
        j--;
    }
    r[j+1]=r[i];         /* 在 j+1 位置处插入 r[i] */
}
}

```

插入排序算法的时间复杂度是 $O(n^2)$ 。

11.1.2 希尔(Shell)排序

希尔排序的基本思想是:把记录按下标的一定增量分组,对每组记录使用插入排序,随着增量逐渐减小,所分成的组包含的记录越来越多,到增量的值减小到1时,整个数据合成为一组,构成一组有序记录,则完成排序。实现希尔排序的函数如下:

```

void shellsort(sqlist r, int n)
{
    int i, j, gap, x;
    gap=n/2;           /* 增量置初值 */
    while (gap>0)
    {
        for (i=gap+1; i<=n; i++)
        {
            j=i-gap;
            while (j>0)
            {
                if (r[j].key>r[j+gap].key)
                {
                    x=r[j];           /* 将 r[j]与 r[j+gap]进行交换 */
                    r[j]=r[j+gap];
                    r[j+gap]=x;
                    j=j-gap;
                }
                else j=0;
            }
        }
        gap=gap/2;       /* 减小增量 */
    }
}

```

希尔排序算法的时间复杂度是 $O(n\log_2 n)$ 。

11.1.3 起泡排序

起泡排序的算法思想是:通过无序区中相邻记录关键字间的比较和位置的交换,使关键字最小的记录如气泡一般逐渐往上“漂浮”直至“水面”。整个算法是从最下面的记录开始,对每两个相邻的关键字进行比较,且使关键字较小的记录换至关键字较大的记录之上,使得经过一趟起泡排序后,关键字最小的记录到达最上端,接着,再在剩下的记录中找关键字最小的记录,并把它换在第二个位置上。依此类推,一直到所有记录都有序为止。实现起泡排序的函数如下:

```
void bubblesort(sqlist r,int n)
{
    int i,j,w;
    for (i=1;i<=n-1;i++)
        for (j=n;j>=i+1;j--)
            if (r[j].key<r[j-1].key)          /* 比较 */
            {                                  /* r[j]与r[j-1]进行交换 */
                w=r[j];
                r[j]=r[j-1];
                r[j-1]=w;
            }
}
```

起泡排序算法的时间复杂度是 $O(n^2)$ 。

11.1.4 快速排序

快速排序是由起泡排序改进而得的,它的基本思想是:在待排序的 n 个记录中任取一个记录(通常取第一个记录),将该记录放入最终位置后,数据序列被此记录分割成两部分。所有关键字比该记录关键字小的放置在前一部分,所有比它大的放置在后一部分,并把该记录排在这两部分的中间,这个过程称作一趟快速排序。之后对所有的两部分分别重复上述过程,直至每部分内只有一个记录为止。简而言之,每趟使表的第一个元素入终位,将表一分为二,对子表按递归方式继续这种划分,直至划分的子表长为 1。

一趟快速排序采用从两头向中间扫描的办法,同时交换与基准记录逆序的记录。具体作法是:设两个指示器 i 和 j ,它们的初值分别为指向无序区中第一个和最后一个记录。假设无序区中记录为 $r[s], r[s+1], \dots, r[t]$,则 i 的初值为 s , j 的初值为 t ,首先将 $r[s]$ 移至变量 $r[0]$ 中作为基准,令 j 自 t 起向左扫描直至 $r[j].key < r[0].key$ 时,将 $r[j]$ 移至 i 所指的位置上,然后令 i 自 $i+1$ 起向右扫描直至 $r[i].key > r[0].key$ 时,将 $r[i]$ 移至 j 所指的位置上,依次重复直至 $i=j$,此时所有 $r[k] (k=s, s+1, \dots, j-1)$ 的关键字都小于 $r[0].key$ 而所有 $r[k] (k=j+1, j+2, \dots, t)$ 的关键字必大于 $r[0].key$,则可将 $r[0]$ 中的记录移至 i 所指位置 $r[i]$,它将无序区中记录分割成 $r[s, j-1]$ 和 $r[j+1, t]$,以便分别进行排序。

快速排序的函数如下:

```
void quicksort(sqlist r,int s,int t)          /* 把 r[s]至 r[t]的元素进行快速排序 */
```

```

{
    int i=s, j=t;
    if (s<t)
        do
        {
            r[0]=r[s];
            while (j>i && r[j].key>=r[0].key) j--;
            if (i<j)
            {
                r[i]=r[j]; i++;
            }
            while (i<j && r[i].key<=r[0].key) i++;
            if (i<j)
            {
                r[j]=r[i]; j--;
            }
        } while(i<j);
    r[i]=r[0];
    quicksort(r, s, j-1);
    quicksort(r, j+1, t);
}

```

快速排序算法的时间复杂度是 $O(n\log_2 n)$ 。

11.1.5 选择排序

选择排序的基本思想是,每一趟排序在 $n-i+1$ ($i=1, 2, \dots, n-1$) 个记录中选取关键字最小的记录,并和第 i 个记录交换之。实现选择排序的函数如下:

```

void selectsort(sqlist r, int n)
{
    int i, j, k, temp;
    for (i=1; i<=n-1; i++)
    {
        k=i;
        for (j=i+1; j<=n; j++)
            if (r[j].key<r[k].key) k=j;          /* 用 k 指出每趟在无顺序段的最小元素 */
        temp=r[i];                               /* 将 r[k] 与 r[i] 交换 */
        r[i]=r[k];
        r[k]=temp;
    }
}

```

选择排序算法的时间复杂度是 $O(n^2)$ 。

11.1.6 堆排序

堆排序是在排序过程中,将向量中存储的数据看成是一棵完全二叉树,利用完全二叉树中双亲结点和小孩结点之间的内在关系来选择关键字最小记录。具体做法是:把待排序的文件的關鍵字存放在数组 $r[1,n]$ 之中,将 r 看作一棵二叉树,每个结点表示一个记录,源文件的第一个记录 $r[1]$ 作为二叉树的根,以下各记录 $r[2,n]$ 依次逐层从左到右顺序排列,构成一棵完全二叉树,任意结点 $r[i]$ 的左孩子是 $r[2i]$,右孩子是 $r[2i+1]$,双亲是 $r[i/2]$ 。

堆排序的关键是构造堆,这里采用筛选算法建堆:假若完全二叉树的某一个结点 i 对于它的左子树、右子树已是堆。需要将 $r[2i].key$ 与 $r[2i+1].key$ 之中的最大者与 $r[i].key$ 比较,若 $r[i].key$ 较小则交换,这有可能破坏下一级的堆。于是继续采用上述方法构造下一级的堆。直到完全二叉树中结点 i 构成堆为止。对于任意一棵完全二叉树,从 $i=[n/2]$ 到 1,反复利用上述思想建堆。大者“上浮”,小者被“筛选”下去。其建堆的函数 `sift` 如下:

```
void sift(sqllist r, int l, int m)
{
    int i, j, x;
    i = l;
    j = 2 * i;           /* r[j] 是 r[i] 的左孩子 */
    x = r[i];
    while (j <= m)
    {
        /* 若右孩子较大,则把 j 修改为右孩子的下标 */
        if (j < m && r[j].key < r[j+1].key) j++;
        if (x.key < r[j].key)
        {
            r[i] = r[j];    /* 将 r[j] 调到父亲的位置上 */
            i = j;          /* 修改 i 和 j 的值,以便继续向下筛 */
            j = 2 * i;
        }
        else j = m + 1;    /* 筛运算完成,令 j = m + 1,以便中止循环 */
    }
    r[i] = x;             /* 被筛结点的值放入最终位置 */
}
```

有了初始建堆的函数,利用该函数,将已有堆中的根与最后一个叶子交换,进一步恢复堆,直到一棵树只剩一个根为止。实现堆分类的函数如下:

```
void heapsort(sqllist r, int n)
{
    int i;
    struct rec m;
    for (i = n/2; i >= 1; i--) sift(r, i, n);    /* 建立初始堆 */
    for (i = n; i >= 2; i--)                     /* 进行 n-1 次循环,完成堆排序 */
    {
```

```

        w=r[i];           /* 将第一个元素同当前区间内最后一个元素对换 */
        r[i]=r[1];
        r[1]=w;
        sift(r,i-1,1);     /* 筛 r[1] 结点,得到(i-1)个结点的堆 */
    }
}

```

堆排序算法的时间复杂度是 $O(n\log_2 n)$ 。

11.1.7 归并排序

所谓归并排序就是将两个或两个以上的有序数据序列合并成一个有序数据序列的过程。

设有两个有序关键字表 $s_1=(18,25,37,42)$, $s_2=(20,33,40)$ 。同时将 s_1 和 s_2 存储在数组 $r[1,7]$ 中, s_1 放在 $r[1,4]$ 中, s_2 放在 $r[5,7]$ 中, 现在要归并到一维数组 $r_2[1,7]$ 之中。具体做法是, 只要依次比较这两个有序表中相应记录关键字, 按照“取小”原则复制到 r_2 之中即可。实现归并排序的函数如下:

```

void merge(r,l,m,h;integer,r2)    /* r[l,m]及 r[m+1,h]分别有序,归并后置于 r2 中 */
sqlist r,r2;
int l,m,h;
{
    int i,j,k;
    k=l;           /* k 是 r2 的指示器,i,j 分别为 s1,s2 的指示器 */
    i=l;
    j=m+1;
    while (i<=m && j<=h)
    {
        if (r[i].key<=r[j].key)
        {
            r2[k]=r[i];
            i++;
        }
        else
        {
            r2[k]=r[j];
            j++;
        }
        k++;
    }
    if (i>m)           /* s1 结束 */
        while (j<=h)
        {

```

```

        r2[k]=r[j];          /* 将 s2 复制到 r2 */
        j++; k++;
    }
    else
        while (i<=m)
        {
            r2[k]=r[i];      /* 将 s1 复制到 r2 */
            i++; k++;
        }
    }
}

```

归并排序算法的时间复杂度是 $O(n\log_2 n)$ 。

11.1.8 基数排序

基数排序的思想类似于扑克牌排队的方法。一般地,记录 $r[i]$ 的关键字为 $r[i].key$, $r[i].key$ 是由 d 位数字组成,即 $k^1k^2\dots k^d$, 每一个数字表示关键字的一位,其中 k^1 为最高位, k^d 是最低位,每一位的值都在 $0\leq k^i < rd$ 范围内, rd 称为基数。排序时,先按最低位的值对记录进行排序。在此基础上,再按次低位进行排序。依此类推,由低位向高位,每趟都是根据关键字的一位并在前一趟的基础上对所有记录进行排序,直至最高位,则完成了基数排序的整个过程。

设基数排序中记录的数据类型如下:

```

struct element
{
    int key[d];          /* d 为关键字的位数 */
    int next;
};
element rsqlist[n];

```

实现基数排序的函数如下:

```

void radixsort(rsqlist r, int p, int d, int n)
{
    int f[rd], e[rd];    /* 队的头、尾指示器, rd 是基数, 二进制数 rd 为 2, 十进制数 rd 为 10 */
    for (k=1; k<=n-1; k++) r[k].next=k+1;
    r[n].next=0;
    p=1;                  /* 原始数据串成静态链表, 头指针为 p */
    for (l=d; l>0; l--)   /* 从最后一位关键字开始 */
    {
        for (j=0; j<rd; j++) f[j]=0;    /* 队指示器初值 */
        while (p!=0)
        {
            k=r[p].key[l];    /* 找到队号为 k */
            if (f[k]==0) f[k]=p;

```



```

        else r[e[k]].next=p;
        e[k]=p;
        p=r[p].next;          /* 进行分配 */
    }
    j=0;
    while (f[j]==0) j++;      /* 寻找第一个非空队 */
    p=f[j]; t=e[j];
    while (j<rd-1)
    {
        j++;
        if (f[j]!=0)
        {
            r[t].next=f[j];
            t=e[j];
        }
        /* 进行收集 */
        r[t].next=0;          /* 收尾 */
    }
}

```

基数排序算法的时间复杂度是 $O(d * (rd+n))$, 其中 rd 是基数, d 关键字的位数, n 是元素个数。

11.2 基 本 题

11.2.1 单项选择题

1. 在所有排序方法中,关键字比较的次数与记录的初始排列次序无关的是 ①。

A. 希尔排序 B. 起泡排序 C. 插入排序 D. 选择排序

答:①D

2. 设有 1000 个无序的元素,希望用最快的速度挑选出其中前 10 个最大的元素,最好选用 ① 排序法。

A. 起泡排序 B. 快速排序 C. 堆排序 D. 基数排序

答:①C

3. 在待排序的元素序列基本有序的前提下,效率最高的排序方法是 ①。

A. 插入排序 B. 选择排序 C. 快速排序 D. 归并排序

答:①A

4. 一组记录的排序码为(46,79,56,38,40,84),则利用堆排序的方法建立的初始堆为 ①。

A. 79,46,56,38,40,80

B. 84,79,56,38,40,46

C. 84, 79, 56, 46, 40, 38

D. 84, 56, 79, 40, 46, 38

答:①B

5. 一组记录的关键码为(46, 79, 56, 38, 40, 84), 则利用快速排序的方法, 以第一个记录为基准得到的一次划分结果为 ①。

A. 38, 40, 46, 56, 79, 84

B. 40, 38, 46, 79, 56, 84

C. 40, 38, 46, 56, 79, 84

D. 40, 38, 46, 84, 56, 79

答:①C

6. 一组记录的排序码为(25, 48, 16, 35, 79, 82, 23, 40, 36, 72), 其中含有 5 个长度为 2 的有序表, 按归并排序的方法对该序列进行一趟归并后的结果为 ①。

A. 16 25 35 48 23 40 79 82 36 72

B. 16 25 35 48 79 82 23 36 40 72

C. 16 25 48 35 79 82 23 36 40 72

D. 16 25 35 48 79 23 36 40 72 82

答:①A

7. 排序方法中, 从未排序序列中依次取出元素与已排序序列(初始时空)中的元素进行比较, 将其放入已排序序列的正确位置上的方法, 称为 ①。

A. 希尔排序

B. 起泡排序

C. 插入排序

D. 选择排序

答:①C

8. 排序方法中, 从未排序序列中挑选元素, 并将其依次放入已排序序列(初始时空)的一端的方法, 称为 ①。

A. 希尔排序

B. 归并排序

C. 插入排序

D. 选择排序

答:①D

9. 用某种排序方法对线性表(25, 84, 21, 47, 15, 27, 68, 35, 20)进行排序时, 元素序列的变化情况如下:

(1) 25, 84, 21, 47, 15, 27, 68, 35, 20

(2) 20, 15, 21, 25, 47, 27, 68, 35, 84

(3) 15, 20, 21, 25, 35, 27, 47, 68, 84

(4) 15, 20, 21, 25, 27, 35, 47, 68, 84

则所采用的排序方法是 ①。

A. 选择排序

B. 希尔排序

C. 归并排序

D. 快速排序

答:①D

10. 下述几种排序方法中, 平均查找长度最小的是 ①。

A. 插入排序

B. 选择排序

C. 快速排序

D. 归并排序

答:①C

11. 下述几种排序方法中,要求内存量最大的是 ①。

A. 插入排序 B. 选择排序 C. 快速排序 D. 归并排序

答:①D

12. 快速排序方法在 ① 情况下最不利于发挥其长处。

A. 要排序的数据量太大

B. 要排序的数据中含有多个相同值

C. 要排序的数据已基本有序

D. 要排序的数据个数为奇数

答:①C

11.2.2 填空题(将正确的答案填在相应的空中)

1. 在对一组记录(54,38,96,23,15,72,60,45,83)进行直接插入排序时,当把第7个记录60插入到有序表时,为寻找插入位置需比较 ① 次。

答:①3

2. 在利用快速排序方法对一组记录(54,38,96,23,15,72,60,45,83)进行快速排序时,递归调用而使用的栈所能达到的最大深度为 ①,共需递归调用的次数为 ②,其中第二次递归调用是对 ③ 一组记录进行快速排序。

答:①2 ②4 ③(23,38,15)

3. 在堆排序、快速排序和归并排序中,若只从存储空间考虑,则应首先选取 ① 方法,其次选取 ② 方法,最后选取 ③ 方法;若只从排序结果的稳定性考虑,则应选取 ④ 方法;若只从平均情况下排序最快考虑,则应选取 ⑤ 方法;若只从最坏情况下排序最快并且要节省内存考虑,则应选取 ⑥ 方法。

答:①堆排序 ②快速排序 ③归并排序

④归并排序 ⑤快速排序 ⑥堆排序

4. 在插入排序、希尔排序、选择排序、快速排序、堆排序、归并排序和基数排序中,排序是不稳定的有 ①。

答:①希尔排序、选择排序、快速排序和堆排序

5. 在插入排序、希尔排序、选择排序、快速排序、堆排序、归并排序和基数排序中,平均比较次数最少的排序是 ①,需要内存容量最多的是 ②。

答:①快速排序 ②基数排序

6. 在堆排序和快速排序中,若原始记录接近正序或反序,则选用 ①,若原始记录无序,则最好选用 ②。

答:①堆排序 ②快速排序

7. 在插入和选择排序中,若初始数据基本正序,则选用 ①;若初始数据基本反序,则选用 ②。

答:①插入排序 ②选择排序

8. 对 n 个元素的序列进行起泡排序时,最少的比较次数是 ①。

答:① $n-1$

11.3 习题解析

1. 已知序列{17,18,60,40,7,32,73,65,85},请给出采用起泡排序法对该序列作升序排序时的每一趟的结果。

解:依题意,采用起泡排序法排序的各趟的结果如下:

初始: 17,18,60,40,7,32,73,65,85

1 趟: 17,18,40,7,32,60,65,73,85

2 趟: 17,18,7,32,40,60,65,73,85

3 趟: 17,7,18,32,40,60,65,73,85

4 趟: 7,17,18,32,40,60,65,73,85

5 趟: 7,17,18,32,40,60,65,73,85

第5趟无元素交换,则排序结束。

2. 已知序列{503,87,512,61,908,170,897,275,653,462},请给出采用快速排序法对该序列作升序排序时的每一趟的结果。

解:依题意,采用快速排序法排序的各趟的结果如下:

初始: 503,87,512,61,908,170,897,275,653,462

1 趟: [462,87,275,61,170] 503 [897,908,653,512]

2 趟: [170,87,275,61] 462,503 [897,908,653,512]

3 趟: [61,87] 170 [275] 462,503 [897,908,653,512]

4 趟: 61 [87] 170 [275] 462,503 [897,908,653,512]

5 趟: 61,87,170 [275] 462,503 [897,908,653,512]

6 趟: 61,87,170,275,462,503 [897,908,653,512]

7 趟: 61,87,170,275,462,503 [512,653] 897 [908]

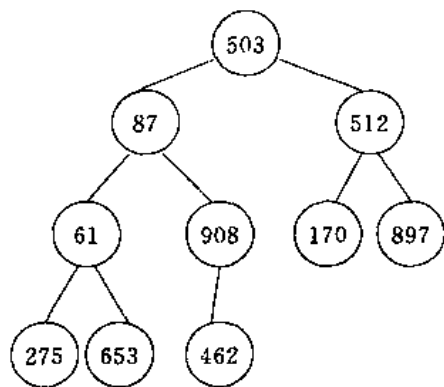
8 趟: 61,87,170,275,462,503,512 [653] 897 [908]

9 趟: 61,87,170,275,462,503,512,653,897 [908]

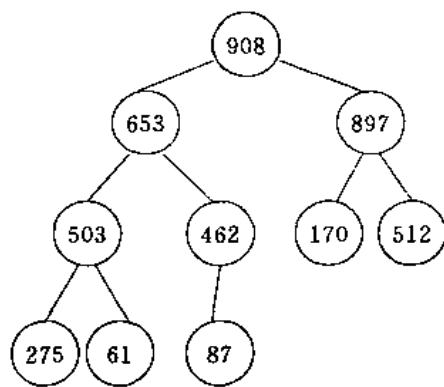
10 趟: 61,87,170,275,462,503,512,653,897,908

3. 已知序列{503,87,512,61,908,170,897,275,653,462},请给出采用堆排序法对该序列作升序排序时的每一趟的结果。

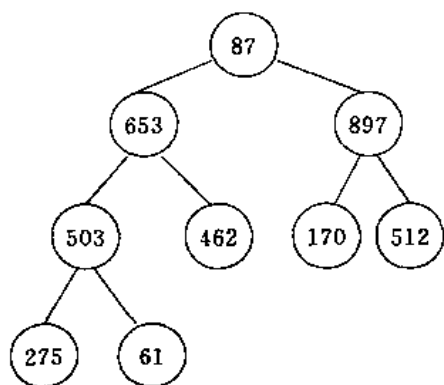
解:依题意,采用堆排序法排序的各趟的结果如图 11.1 所示。



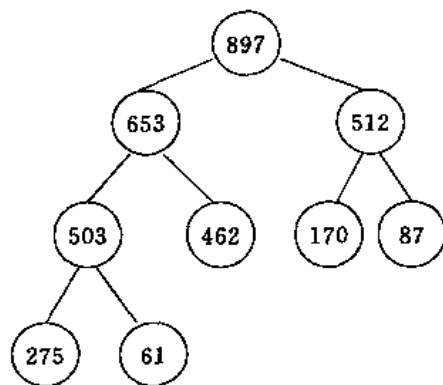
(a) 初始堆



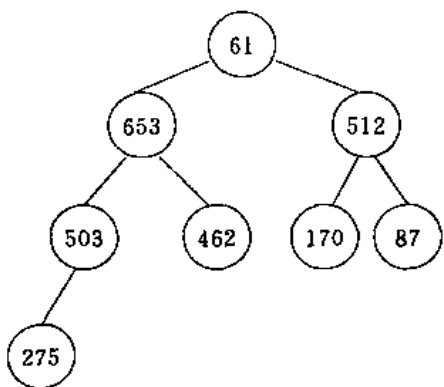
(b) 建堆



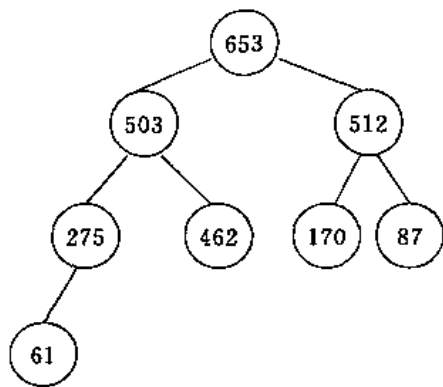
(c) 交换908和87, 输出908



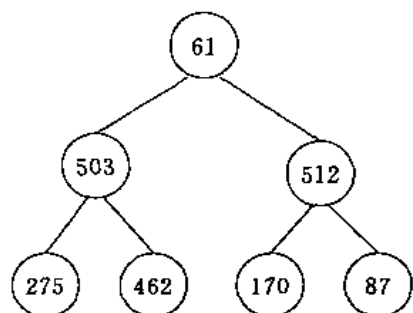
(d) 筛选调整



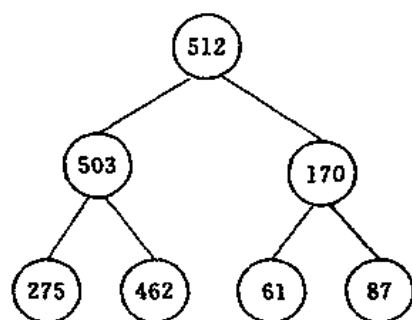
(e) 交换897和61, 输出897



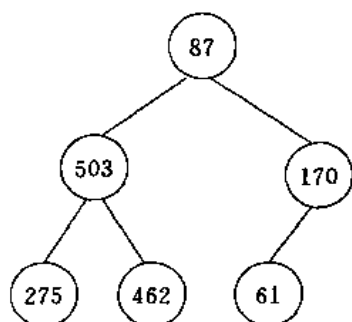
(f) 筛选调整



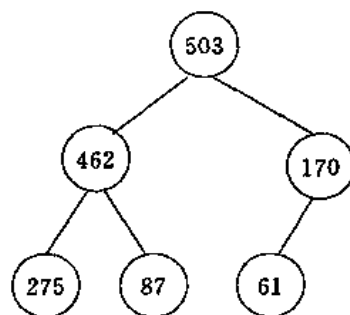
(g) 交换653和61, 输出653



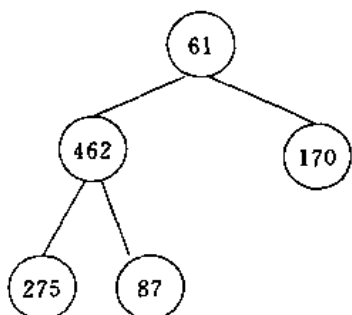
(h) 筛选调整



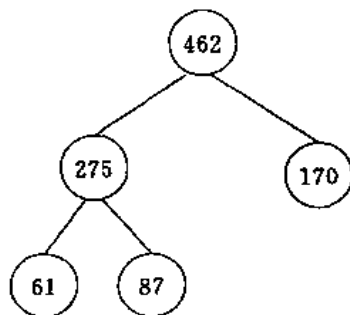
(i) 交换512和87, 输出512



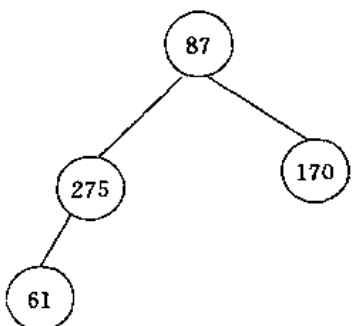
(j) 筛选调整



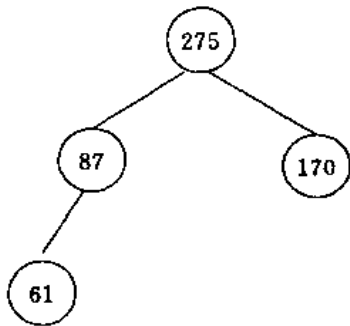
(k) 交换503和61, 输出503



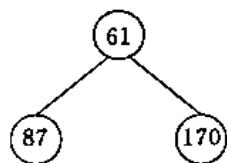
(l) 筛选调整



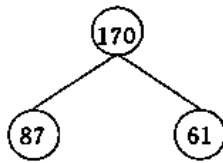
(m) 交换462和87, 输出462



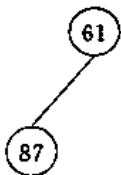
(n) 筛选调整



(o) 交换275和61, 输出275



(p) 筛选调整



(q) 交换170和61, 输出170



(r) 筛选调整



(s) 交换87和61, 输出87

(t) 输出61

图 11.1 数据堆排序过程

4. 已知序列{503,87,512,61,908,170,897,275,653,462},请给出采用基数排序法对该序列作升序排序时的每一趟的结果。

解:依题意,采用基数排序法排序的各趟的结果如下:

初始:503,87,512,61,908,170,897,275,653,462

1 趟(按个位排序):170,61,462,512,503,653,275,87,897,908

2 趟(按十位排序):503,908,512,653,61,462,170,275,87,897

3 趟(按百位排序):61,87,170,275,462,503,512,653,897,908

5. 已知序列{503,17,512,908,170,897,275,653,426,154,509,612,677,765,703,94},请给出采用希尔排序法($d_1=8$)对该序列作升序排序时的每一趟的结果。

解:依题意,采用希尔排序法排序的各趟的结果如下:

初始:503,17,512,908,170,897,275,653,426,154,509,612,677,765,703,94

1 趟($d_1=8$):426,17,509,612,170,765,275,94,503,154,512,908,677,897,703,653

2 趟($d_2=4$):170,17,275,94,426,154,509,612,503,765,512,653,677,897,703,908

3 趟($d_3=2$):170,17,275,94,426,154,503,612,509,653,512,765,677,897,703,908

4 趟($d_4=1$):17,94,154,170,275,426,503,509,512,612,653,677,703,765,897,908

6. 已知序列{70,83,100,65,10,32,7,9},请给出采用插入排序法对该序列作升序排序时的每一趟的结果。

解:依题意,采用插入排序法排序的各趟的结果如下:

初始: (70), 83, 100, 65, 10, 32, 7, 9

1 趟: (70, 83), 100, 65, 10, 32, 7, 9

2 趟: (70, 83, 100), 65, 10, 32, 7, 9

3 趟: (65, 70, 83, 100), 10, 32, 7, 9

4 趟: (10, 65, 70, 83, 100), 32, 7, 9

5 趟: (10, 32, 65, 70, 83, 100), 7, 9

6 趟: (7, 10, 32, 65, 70, 83, 100), 9

7 趟: (7, 9, 10, 32, 65, 70, 83, 100)

7. 已知序列{10, 18, 4, 3, 6, 12, 1, 9, 18, 8}, 请给出采用 Shell 排序法对该序列作升序排序时的每一趟的结果。

解: 依题意, 采用 Shell 排序法排序的各趟的结果如下:

初始: 10, 18, 4, 3, 6, 12, 1, 9, 18, 8

1 趟: 10, 1, 4, 3, 6, 12, 18, 9, 15, 8

2 趟: 4, 1, 6, 3, 10, 8, 15, 9, 18, 12

3 趟: 1, 3, 4, 6, 8, 9, 10, 12, 15, 18

第 3 趟无元素交换, 则排序结束。

8. 已知序列{10, 18, 4, 3, 6, 12, 1, 9, 18, 8}, 请给出采用归并排序法对该序列作升序排序时的每一趟的结果。

解: 依题意, 采用归并排序法排序的各趟的结果如下:

初始: 10, 18, 4, 3, 6, 12, 1, 9, 18, 8

1 趟: [10, 18] [3, 4] [6, 12] [1, 9] [8, 15]

2 趟: [3, 4, 10, 18] [1, 6, 9, 12] [8, 15]

3 趟: [3, 4, 10, 18] [1, 6, 8, 9, 12, 15]

4 趟: [1, 3, 4, 6, 8, 9, 10, 12, 15, 18]

第 4 趟归并完毕, 则排序结束。

* 9. 有 n 个不同的英文单词, 它们的长度相等, 均为 m , 若 $n \gg 50, m < 5$, 试问采用什么排序方法时间复杂性最佳? 为什么?

答: 采用基数排序方法时间复杂性最佳。

因为这里英文单词的长度相等, 且英文单词是由 26 个字母组成的, 满足进行基数排序的条件, 另外, 依题意, $m \ll n$, 基数排序的时间复杂性由 $O(m(n+rm))$ 变成 $O(n)$, 因此时间复杂性最佳。

* 10. 如果只想得到一个序列中第 k 个最小元素之前的部分排序序列, 最好采用什么排序方法? 为什么? 如由这样的一个序列: {57, 40, 38, 11, 13, 34, 48, 75, 25, 6, 19, 9, 7} 得到其第 4 个最小元素之前的部分序列 {6, 7, 9, 11}, 使用所选择的算法实现时, 要执行多少次比较?

解: 采用堆排序最合适, 依题意可知只需取得第 k 个最小元素之前的排序序列时, 堆排序的时间复杂度是 $O(n+k\log_2 n)$, 若 $k \leq n/\log_2 n$, 则可得到的时间复杂度是 $O(n)$ 。

对于序列: {57, 40, 38, 11, 13, 34, 48, 75, 25, 6, 19, 9, 7} 得到其第 4 个最小元素之前的部

分序列{6,7,9,11},使用所选择的算法实现时,其执行比较次数如下:

建堆	20次	得到6
调整	5次	得到7
调整	4次	得到9
调整	5次	得到11
共计34次比较。		

* 11. 对于快速排序的非递归算法,可用队列(而不用栈)实现吗?若能,说明理由;若不能,也要说明理由。

解:可以用队列实现,因为所划分出来的子表可以在任何次序上被继续排序。

* 12. 已知下列各种初始状态(长度为 n)的元素,试问当利用直接插入法进行排序时,至少需要进行多少次比较(要求排序后的文件按关键字从小到大顺序排列)?

(1)关键字自小至大有序($key_1 < key_2 < \dots < key_n$)

(2)关键字自大至小逆序($key_1 > key_2 > \dots > key_n$)

(3)奇数关键字顺序有序,偶数关键字顺序有序($key_1 < key_3 < \dots < key_2 < key_4 < \dots$)

(4)前半部分元素按关键字顺序有序,后半部分元素按关键字顺序逆序($key_1 < key_2, \dots < key_m, key_{m+1} < key_{m+2} < \dots < key_n$,这里的 m 是中间位置)

解:依题意,取各种情况下的最好的比较次数即为最少比较次数。

(1)这种情况下,插入第 i ($2 \leq i \leq n$)个元素的比较次数为1,因此总的比较次数为:

$$1+1+\dots+1=n-1$$

(2)这种情况下,插入第 i ($2 \leq i \leq n$)个元素的比较次数为 i ,因此总的比较次数为:

$$2+3+\dots+n-1=(n-1)(n+2)/2$$

(3)这种情况下,比较次数最少的情况是所有记录关键字均按升序排列,这时,总的比较次数为:

$$n-1$$

(4)在后半部分元素的关键字均大于前半部分元素的关键字时需要的比较次数最少,此时前半部分的比较次数= $m-1$,后半部分的比较次数= $(n-m-1)(n-m+2)/2$,因此,比较次数为:

$$\begin{aligned} & m-1+(n-m-1)(n-m+2)/2 \\ & = (n-2)(n+8)/8 \quad (\text{假设 } n \text{ 为偶数, } m=n/2) \end{aligned}$$

13. 设计一个函数修改起泡排序过程以实现双向起泡排序。

解:起泡排序是从最下面的记录开始,对每两个相邻的关键字进行比较,且使关键字较小的记录换至关键字较大的记录之上,使得经过一趟起泡排序后,关键字最小的记录到达最上端,接着,再在剩下的记录中找关键字最小的记录,并把它换在第二个位置上。依此类推,一直到所有记录都有序为止。双向起泡排序则是每一趟通过每两个相邻的关键字进行比较,产生最小和最大的元素。实现双向起泡排序的函数如下:

```

void dbubble(sqlist r)
{
    int i=1,j,t,b=1;
    struct rec t;
    while (b)
    {
        b=0;
        for (j=n-i+1;j>=i+1;j--)          /* 找出较小的元素放在 r[j]处 */
            if (r[j].key<r[j-1].key)
            {
                b=1;
                t=r[j];
                r[j]=r[j-1];
                r[j-1]=t;
            }
        for (j=i+1;j<=n-i-1;j++)          /* 找出较大的元素放在 r[n-i+1]处 */
            if (r[j].key>r[j+1].key)
            {
                b=1;
                t=r[j];
                r[j]=r[j+1];
                r[j+1]=t;
            }
        i++;
    }
}

```

14. 编写一个递归函数实现归并排序。

解：依题意，归并排序的递归模型为：

$$\begin{cases} f(r,l,h)=r[l] & \text{若 } l=h, \text{即只有一个元素} \\ f(r,l,h)=\text{将有序序列 } f(r,l,m) \text{ 和 } f(r,m+1,h) \text{ 合并起来, } l \neq h \text{ (这里 } m=(l+h)/2) \end{cases}$$

因此，归并排序的递归函数如下，其中 $r[l,h]$ 经排序后放在 $r1[l,h]$ 之中， $r2[l,h]$ 为辅助空间：

```

void mergesort(r,r1,l,h)
sqlist r,r1;
int l,h;
{
    sqlist r2;
    int m;
    if (l==h) r1[l]=r[l];
    else
    {

```

```

        m=(l+h)/2;
        mergesort(r,r2,l,m);
        mergesort(r,r2,m+1,h);
        merge(r2,l,m,h,r1);
    }
}

```

合并函数如下,其中 $r[l,m](s1)$ 及 $r[m+1,h](s2)$ 分别有序,归并后置入 $r2$ 中:

```

void merge(r,l,m,h,r2)
sqlist r,r2;
int l,m,h;
{
    int i,j,k;
    k=l;                                /* k 是 r2 的指示器,i,j 分别为 s1,s2 的指示器 */
    i=l;
    j=m+1;
    while (i<=m && j<=h)
    {
        if (r[i].key<=r[j].key)
        {
            r2[k]=r[i];
            i++;
        }
        else
        {
            r2[k]=r[j];
            j++;
        }
        k++;
    }
    if (i>m)                                /* s1 结束 */
        while (j<=h)
        {
            r2[k]=r[j];                    /* 将 s2 复制到 r2 */
            j++; k++;
        }
    else
        while (i<=m)
        {
            r2[k]=r[i];                    /* 将 s1 复制到 r2 */
            i++; k++;
        }
}

```

}

* 15. 已知奇偶转换排序如下所述:第一趟对所有奇数的 i , 将 $a[i]$ 和 $a[i+1]$ 进行比较, 第二趟对所有偶数的 i , 将 $a[i]$ 和 $a[i+1]$ 进行比较, 每次比较时若 $a[i] > a[i+1]$, 则将二者交换, 以后重复上述二趟过程交换进行, 直至整个数组有序。

(1) 试问排序结束的条件是什么?

(2) 编写一个实现上述排序过程的算法。

解: (1) 排序结束条件为没有交换元素为止。

(2) 实现本题奇偶转换排序的函数如下:

```
void oesort(int a[n])
{
    int i, flag;
    do
    {
        flag=0;
        for (i=1; i<=n; i++)          /* 奇数扫描 */
        {
            if (a[i]>a[i+1])
            {
                flag=1;
                t=a[i+1];
                a[i+1]=a[i];
                a[i]=t;
            }
            i++;
        }
        for (i=2; i<=n; i++)          /* 偶数扫描 */
        {
            if (a[i]>a[i+1])
            {
                flag=1;
                t=a[i+1];
                a[i+1]=a[i];
                a[i]=t;
            }
            i++;
        }
    } while (flag!=0);
}
```

16. 采用单链表作存储结构, 编写一个采用选择排序方法进行升序排序的函数。

解: 依题意, 单链表定义如下:

```

struct node
{
    int key;
    struct node *link;
};

```

因此,实现本题功能的函数如下:

```

struct *selectsort(struct node *h)
{
    struct node *p, *q, *r, *s, *t;
    t=NULL;
    while (h!=NULL)
    {
        p=h;
        q=NULL;
        s=h;
        r=NULL;
        while (p!=NULL)
        {
            if (p->key<s->key)
            {
                s=p;
                p=q;
            }
            q=p;
            p=p->link;
        }
        if (s==h) h=h->link;
        else h=s;
        s->link=t;
        t=s;
    }
    h=t;
    return(h);
}

```

* 17. 编写实现快速排序的非递归函数。

解:依题意,使用一个栈 stack,它是一个二维数组:

stack[i][0]存储子表第一个元素的下标

stack[i][1]存储子表最后一个元素的下标

首先将(1,n)入栈,然后进行如下循环直到栈空;退栈得到 t1,t2,调用数据分割函数 partition(),该函数自动调整好 t1~t2 子表的第一个元素的位置并分解成两个子表 t1~i-1

和 $i+1 \sim t2$, 若这些子表不只一个元素, 则入栈。每次调用 `partition()` 都修改 `r` 的次序, 最后 `r` 便有序了。因此, 实现快速排序的非递归函数如下:

```
void quicksort(sqlist r, int t1, int t2)
{
    int stack[m0][2], i, top = 1;
    stack[top][0] = t1;
    stack[top][1] = t2;
    while (top > 0)
    {
        t1 = stack[top][0];
        t2 = stack[top][1];
        top--;
        partition(r, t1, t2, 1);
        if (t1 < i - 1)
        {
            top++;
            stack[top][0] = t1;
            stack[top][1] = i - 1;
        }
        if (i + 1 < t2)
        {
            top++;
            stack[top][0] = i + 1;
            stack[top][1] = t2;
        }
    }
}
```

实现数据分割的函数如下:

```
void partition(r, l, h, i)
sqlist r;
int l, h, i;
{
    int i = l, j = h;
    struct rec x;
    x = r[i];
    do /* 从右向左扫描, 查找第一个关键字小于 x.key 的记录 */
    {
        while (x.key <= r[j].key && j > i) j--;
        if (j > i) /* 相当于交换 r[i] 和 r[j] */
        {
            r[i] = r[j];

```

```

        l++;
    }
    /* 从左向右扫描,查找第一个关键字大于 x.key 的记录 */
    while (x.key >= r[l].key && l < j) l++;
    if (l < j) /* 已找到 r[l].key > x.key */
    { /* 相当于交换 r[l] 和 r[j] */
        r[l] = r[j];
        j--;
    }
} while (l != j); /* 基准 x 已最终定位 */
r[i] = x;
}

```

* 18. 利用一维数组 A 可以对 n 个整数进行排序。其中一种排序的算法的处理思想是：将 n 个整数分别作为数组 A 的 n 个元素的值，每次（即第 i 次）从元素 A[i]~A[n] 中挑出最小的一个元素 A[k] ($i \leq k \leq n$)，然后将 A[k] 与 A[i] 换位。这样反复 n 次完成排序。编写实现上述算法的函数。

解：依题意，实现本题排序的函数如下：

```

void sort(int A[n], int n)
{
    int i, j, t, minval, minidx;
    for (i = 1; i <= n - 1; i++)
    {
        minval = A[i + 1]; /* 存储 A[i+1] 至 A[n] 之间的最小数 */
        minidx = i + 1; /* 存储 A[i+1] 至 A[n] 之间的最小数的下标 */
        for (j = i + 2; j <= n; j++)
            if (A[j] < minval)
            {
                minval = A[j];
                minidx = j;
            }
        if (minidx != i + 1)
        {
            t = A[i + 1]; /* 将 A[i+1] 与 A[minidx] 进行交换 */
            A[i + 1] = A[minidx];
            A[minidx] = t;
        }
    }
}

```

第12章 文 件

文件是大量性质相同的记录组成的集合,文件存储在外存储器中,如磁盘和磁带等。记录是文件中可存取的基本数据单位,它是由若干数据项组成,而数据项是文件中最小的数据单位,通常由一个或多个数字位或字符组成,用来表示记录的某种属性。

在数据结构中对于文件的运算主要为检索和修改两大类。检索是按记录的逻辑号或关键字值或属性查找某个记录。修改包括对记录的插入、删除和对记录某些数据项的更新。

12.1 基本文件组织方式

文件在存储介质上的组织方式有顺序组织、随机组织和链组织。常用的文件类型有顺序文件、索引文件、直接存取文件和多关键字文件。

12.1.1 顺序文件

顺序文件是最简单的文件,文件的各个记录按逻辑顺序存放在外存的连续区中,即顺序文件中物理记录的顺序和逻辑记录的顺序是一致的。如果文件按关键字有序输入,则形成的顺序文件称为顺序有序文件;否则称为顺序无序文件。

顺序文件是根据记录的序号或记录的相对位置来进行存取的文件组织方式。其特点是存取第 i 个记录,必须先搜索在它之前的 $i-1$ 个记录;插入新的记录时只能加在文件的末尾;若要更新文件中的某个记录,则必须将整个文件进行复制。

12.1.2 索引文件

索引文件是在主文件之外再建立一个指示关键字与其物理记录之间的对应关系的表,这种表称为索引表。索引表与主文件共同构成索引文件。索引文件的检索分成两步完成,首先将索引表读入内存,再根据索引表所指示的物理地址将记录所在的数据块读入内存进行查找。索引表通常是按关键字值升序或降序排列的。若主文件也按关键字值升序或降序排列,则这样构成的索引文件称为索引顺序文件;若主文件是无序的,则构成的索引文件称为索引无序文件。

1. 索引无序文件

索引无序文件是将每个记录的关键字及记录的物理块头地址构造索引表,索引表中索引项按关键字值升序排序。

对索引的关键字查找时,先在索引表中快速(如二分查找)找到该关键字值的索引项,这样可找到该记录块头地址,从而在主文件中直接找到对应的记录。在修改数据时,先在主文件中进行,然后修改索引表。

2. 索引顺序文件

索引顺序文件是有序文件并带有索引,主要用于磁盘的柱面磁道索引。在磁盘上,一个

记录的物理地址是由盘组、柱面和磁道三级定位的。

索引顺序文件是由盘组索引、柱面索引、磁道索引和主文件组成的。主文件是按关键字有序排列的,由此可对每个磁道建立一个磁道索引项,然后对每个柱面建立一个柱面索引项,最后对柱面索引建立一个盘组索引项。

索引顺序文件适应快速的检索。

3. 树索引文件

树索引是一种适合分支查找的动态索引,有B树索引和B+树索引等。

(1) B树索引

B树是一种平衡的多路查找树。一棵 m 阶的B树或为空树,或为满足如下条件的树:

- 每个结点至多有 m 棵子树;
- 根本身若不是叶子,至少有两棵子树;
- 除根外,非终端结点至少有 $m/2$ 棵子树;
- 叶子结点有同层,相当于失败点(查找不到);
- 非叶子结点包含如下信息:

$(n, p_0, k_1, p_1, k_2, \dots, k_n, p_n)$

其中 $n(n \leq m-1)$ 为结点含关键字值的个数, $k_i(i=1, 2, \dots, n)$ 为关键字值,且 $k_i < k_{i+1}$ ($i=1, 2, \dots, n-1$); $p_i(i=0, 1, \dots, n)$ 为指向子树根结点的指针,且指针 p_{i-1} 所指子树中所有结点的关键字值均小于 $k_i(i=1, 2, \dots, n)$, p_n 所指子树中所有结点的关键字值均大于 k_n 。

B树的查找:树中查找关键字值等于 k 的记录索引,只要从根结点开始,依次查询结点中的关键字值,若 $k=k_i(1 \leq i \leq n)$,则可由同一索引项中的物理地址找到相应记录,反之,由 $k_i < k < k_{i+1}$ 在 p_i 指针所指结点中继续进行查找,直至遇到叶子结点仍未找到,则查找失败。

B树的插入:对于叶子结点处于第 $L+1$ 层的B树,插入的关键字总是进入第 L 层的结点,现插入关键字值为 k 的结点,若结点的关键字值个数不超过 $m-1$,则插入完成;否则需把结点分裂成两个。分裂的作法是,取一新结点,把原结点上的关键字值和 k 按升序排序后,分成三部分,中间只有一个,左、右数量基本相等,左部、右部所含关键字值分别放在旧、新结点中,中间的一个关键字值连同新结点的存储位置插到父亲结点中。如果父亲结点也是满的,则要再分裂,再往上插,直至这个过程传到根结点。

B树的删除:对于一棵 L 层的B树,如果删除的关键字在第 L 层,则把它从所在的结点里除掉,这样可能导致此结点所包含的关键字的个数小于 $\lceil m/2 \rceil - 1$ 。这种情况下,考虑该结点的左或右兄弟结点,从兄弟结点中移若干个关键字到该结点中来,同时可能涉及到它们的父亲结点中的一个关键字要做相应的改变,使两个结点所含关键字个数基本相同。只有在兄弟结点的关键字个数也很少,刚好等于 $\lceil m/2 \rceil - 1$ 时,这种移动不能进行。此时,要把删除了关键字的结点,它的兄弟结点和它们的父亲结点中一个关键字合并为一个结点。如果删除的关键字不在第 L 层,则先把此关键字与它在B树中的后续结点对换位置,然后再删除该关键字。

(2) B+树索引

B+树是B树的变种,在组织索引文件时比B树更常用,B+树是对B树作了以下改进:

- 结点关键字个数与度数一致;
- 叶子结点包括所有关键字,叶子结点没有指向记录区的指针;结点间形成顺序链表;
- 非终端结点是索引,放置子树中最大(最小)关键字。

B+树索引文件的查找与删除和B树类似。

12.1.3 直接存取文件

直接存取文件又称为哈希文件或散列文件,即利用哈希函数和处理冲突的方法,把文件记录散列到外存上,通常是磁盘上。

磁盘上的直接存取文件的记录是成组存放的,组也称为桶。桶由一块或若干块组成。

哈希地址是桶的物理地址加上块地址。同一地址的桶中包含若干记录。

哈希文件由桶目录、基桶和溢出桶组成,桶目录存放哈希地址即块号,基桶存放块地址,溢出桶用于存放冲突的块的块地址。

对直接存取文件查找时,先求出桶的哈希地址,读入基桶数据块放入内存,然后在内存中进行顺序查找,找不到再读溢出桶。直接存取文件不能进行顺序查找,但数据插入方便,存取速度快。

12.1.4 多关键字文件

文件中除了对主关键字查询外,还常常包含对次关键字的查询,为此,需要对被查询的次关键字建立相应的索引,这种包含多个关键字索引的文件称为多关键字文件。其组织方法有多重表文件和倒排文件两种。

1. 多重表文件

多重表文件是主关键字顺序有序,对主关键字建立非稠密索引,同时对每个需查询的次关键字记录串成一个链表,对次关键字链表建索引。主文件中包含次关键字索引的链指针(块地址)。

2. 倒排文件

倒排文件与多重表文件的区别是,倒排文件中链指针信息不保存在主文件中,而都保存在索引即倒排表中。具有相同次关键字的记录之间不进行链接,在倒排表中列出具有该次关键字记录的物理地址。因此,在倒排表中是由次关键字来决定和选择记录的物理地址。主文件和倒排表共同组成了倒排文件。

12.2 基 本 题

12.2.1 单项选择题

1. 索引无序文件是指 ①。
A. 主文件无序,索引表有序
B. 主文件有序,索引表无序
C. 主文件有序,索引表有序

D. 主文件无序,索引表无序

答:①A

2. 直接存取文件的特点是 ①。

A. 记录按关键字排序

B. 记录可以进行顺序存取

C. 存取速度快,但占用较多的存储空间

D. 记录不需要排序,存取效率高

答:①D

3. 倒排文件的主要优点是 ①。

A. 便于进行插入和删除运算

B. 便于进行文件的合并

C. 能大大提高关键字的查找速度

D. 能大大节省存储空间

答:①C

12.2.2 填空题(将正确的答案填在相应的空中)

1. 索引文件的检索分两步完成,第一步是 ①,第二步是 ②。

答:①将索引表读入内存查找到相应的物理地址 ②根据索引表所指示的物理地址将记录所在的数据块读入内存进行查找

2. 直接存取文件是用 ① 方法组织的。

答:①哈希

3. 树索引文件的特点是 ①。

答:①一种适合分支查找的动态索引

12.3 习题解析

1. 有如图 12.1 所示的一棵 3 阶 B 树,给出分别插入关键字为 2、12、16、17 和 18 的结点之后的结果。

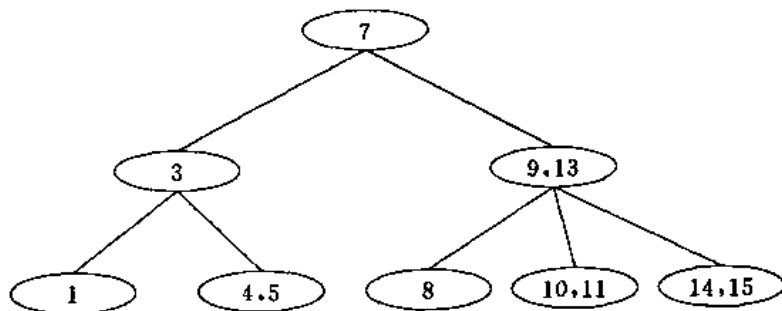
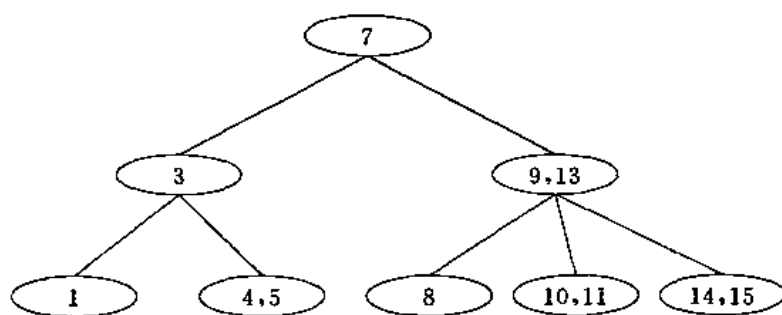


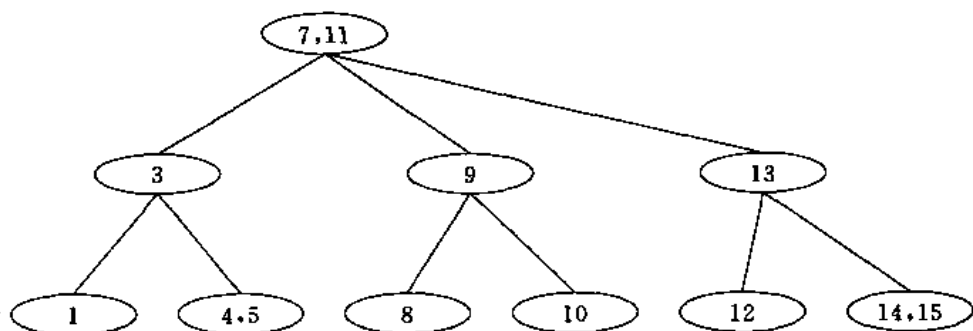
图 12.1 一棵 3 阶 B 树

解:插入关键字为 2、12、16、17 和 18 的结点之后的结果分别如图 12.2(a)、(b)、(c)、(d)

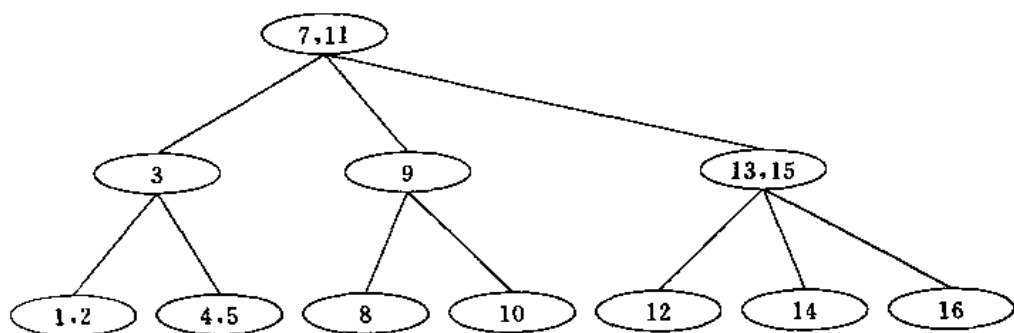
和(e)所示。



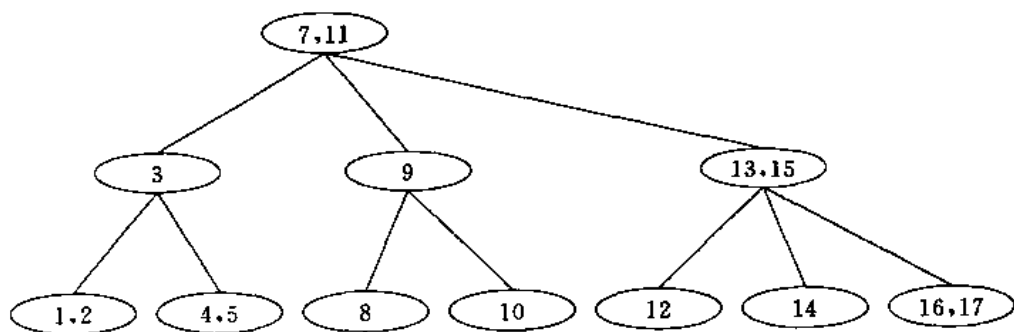
(a) 插入2后



(b) 插入12后



(c) 插入16后



(d) 插入17后

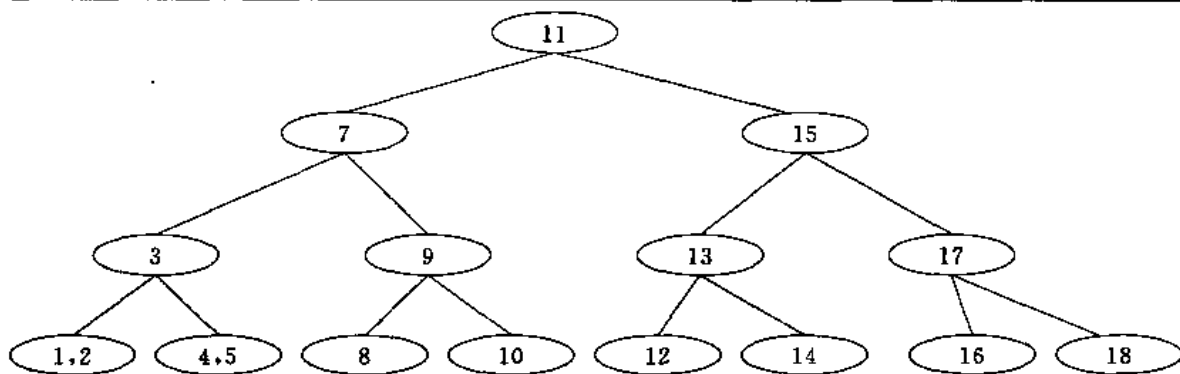


图 12.2 插入结点之后的 B 树

2. 有如图 12.3 所示的 3 阶 B 树, 给出分别删除关键字为 50 和 53 的结点之后的结果。
解: 删除关键字为 50 和 53 的结点之后的结果分别如图 12.4(a) 和 (b) 所示。

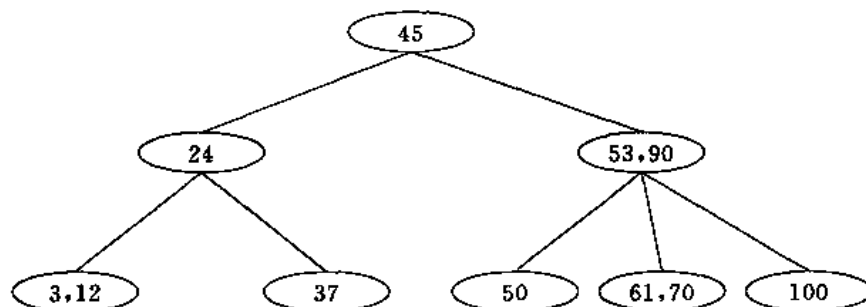
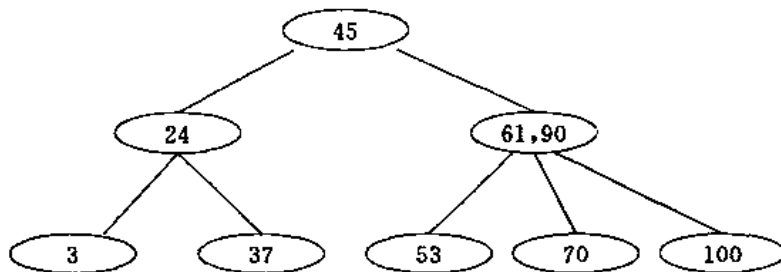
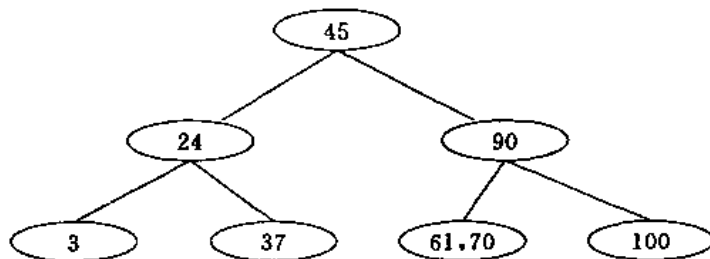


图 12.3 一棵 3 阶 B 树



(a) 删除 50 后



(b) 删除 53 后

图 12.4 删除结点之后的 B 树

3. 设有一个职工文件,每个记录有如下格式:

职工号、姓名、职称、性别、工资

其中“职工号”为主关键字,其他为次关键字,如表 12.1 所列。试用下列结构组织这个文件:

- (1)索引无序文件
- (2)多重表文件
- (3)倒排文件

表 12.1 职工文件

记录	职工号	姓名	职称	性别	工资
1	29	陈军	教授	男	985
2	05	王强	副教授	男	806
3	02	李梅	副教授	女	781
4	38	张兵	讲师	男	580
5	31	付强	助教	男	482
6	43	董威	讲师	男	600
7	17	赵红	教授	女	962
8	46	李芳	助教	女	456

解:(1)索引无序文件如图 12.5 所示。

(2)多重表文件如图 12.6 所示。

(3)倒排文件如图 12.7 所示。

关键字 记录		记录	职工号	姓名	职称	性别	工资
02	3	1	29	陈军	教授	男	985
05	2	2	05	王强	副教授	男	806
17	7	3	02	李梅	副教授	女	781
29	1	4	38	张兵	讲师	男	580
31	5	5	31	付强	助教	男	482
38	4	6	43	董威	讲师	男	600
43	6	7	17	赵红	教授	女	962
46	8	8	46	李芳	助教	女	456

索引表

主文件

图 12.5 索引无序文件结构

记录	职工号	姓名	职称	指针	性别	指针	工资	次关键字	长度	头指针
1	29	陈军	教授	^	男	^	985	教授	2	1
2	05	王强	副教授	^	男	1	806	副教授	2	2
3	02	李梅	副教授	2	女	^	781	讲师	2	4
4	38	张兵	讲师	^	男	2	580	助教	2	5
5	31	付强	助教	^	男	4	482	职称索引		
6	43	董威	讲师	4	男	5	600	次关键字	长度	头指针
7	17	赵红	教授	1	女	3	962	男	5	1
8	46	李芳	助教	5	女	7	456	女	3	3

主文件

性别索引

图 12.6 多重表文件结构

记录	职工号	姓名	职称	性别	工资	次关键字	头指针
1	29	陈军	教授	男	985	教授	1,7
2	05	王强	副教授	男	806	副教授	2,3
3	02	李梅	副教授	女	781	讲师	4,6
4	38	张兵	讲师	男	580	助教	5,8
5	31	付强	助教	男	482	职称索引	
6	43	董威	讲师	男	600	次关键字	指针
7	17	赵红	教授	女	962	男	1,2,4,5,6
8	46	李芳	助教	女	456	女	3,7,8

主文件

性别索引

图 12.7 倒排文件结构

* 4. 凡在图书馆办了借书卡的读者均可借阅一本书,期限为一个月,需要用计算机来管理借、还书的工作。这个系统除了能正确完成日常的借、还书的工作外,还需要帮助管理员进行一些查询工作,例如:有些读者急需借阅某个作者的一本书,但此书被另一读者所借走,需要查一下是谁借走的。又如:有的读者丢失了借书卡,还书时需要查询他所借书的记录。再如:为了使图书流通,管理员每天需给所有到期而未还书的读者寄催还书的通知单。请为该系统设计一个数据文件(包括记录的格式及其在磁盘上的组织方式),并说明该系统功能如何实现(不写算法)。

解:依题意,该系统的数据文件的格式如下:

借书证号、姓名、书名、作者、借书日期

其中主关键字是“借书证号”,由于需要提供多种查询功能,除了按主关键字查询外,还按次关键字(如“作者”、“借书日期”)查询,因此,文件的组织方式可采用多重表文件方式,如图 12.8 所示。

实现该系统功能如下:

借书:在主文件中末尾添加一条借书记录,并修改相应的索引;

还书:这里需要将对应的借书记录删除,但文件没有直接的记录的删除功能,一般是重新生成一个同名的文件进行覆盖,这样很花费时间。为此在主索引中找到该“借书证号”的记录地址,对该地址的记录加上一个特殊的删除标志,在带有删除标志的记录较多时再进行覆盖,这样会节省记录删除的时间。对相应的索引也要修改;

按作者查询借书人:先在作者索引中找到该作者的记录,再到主文件中查找;

按借书证号查询所借书:先在主索引中找到该借书证号的记录,再到主文件中查找;

按借书日期查询过期者:先在借书日期索引中找到日期过期的记录,对每个记录,再到主文件中查找。



图 12.8 图书文件的组织方式

第13章 外排序

外排序指的是对文件中的数据(存储在外存中)进行排序,其数据量很大,不可能也不应该将其全部调到内存中再进行排序,而是把内存作为工作空间,利用它来调整外存储器中数据的位置。

外排序最常用的方法是归并排序法,它分为两个阶段,第一阶段是将文件中的数据分段输入内存,在内存中采用内排序的方法对其分类,这样排序完的文件段,称为归并段,再将其写回外存中,这样在外存中形成许多初始归并段。第二阶段是对这些初始归并段采用某种归并方法,进行多遍归并,最后在外存上形成整个文件的单一归并段,也就完成了这个文件的外排序。

13.1 基本归并排序法

归并排序是与文件的存储介质相关的,常用的文件的存储介质有磁盘和磁带,因此,归并排序有磁盘文件归并排序和磁带文件归并排序。下面分别讨论这两种方法。

13.1.1 磁盘文件归并排序

磁盘文件归并排序法分为如下两个阶段:

1. 初始归并段的生成

初始归并段的生成与第二阶段的排序方法相同,当归并路数 k 较大时,采用一种称为选择树的方法进行内部排序。选择树是一棵二叉树,其中每个结点的关键字取它的两个子结点的关键字中较小者,因此,根结点的关键字是这棵树中所有结点的关键字中最小的。

其过程是由文件输入记录,建立选择树后,每当从选择树中输出一个记录时,树中相应的叶子结点就由下一个输入记录来取代,输出的记录成为当前初始归并段的一部分。如果新输入的记录不能成为当前生成的归并段的一部分,它将等待生成下一个归并段时提供选择。反复进行上述操作,直到所有新输入的记录关键字都小于最后输出记录的关键字时,就生成了一个初始归并段。接着继续生成下一个初始归并段,直到全部记录都处理完为止。

例如,设输入文件的各个记录的关键字为:

15,19,04,83,12,27,11,25,16,34,26,07,10,90,06,...

内存缓冲区可容纳4个记录,采用4路归并的选择树方法生成初始归并段,表13.1给出了生成初始归并段过程中各步的缓冲区内容和输出结果。

表 13.1 初始归并段生成过程

步	1	2	3	4	5	6	7	8	9	10	11	12	13	...
缓冲区内容	15	15	15	(11)	(11)	(11)	(11)	(11)	(11)	11	11	06
	19	19	19	19	25	(16)	(16)	(16)	(16)	16	16	16
	04	12	27	27	27	27	34	(26)	(26)	26	26	26
	83	83	83	83	83	83	83	83	(07)	10	90	90
输出结果	04 12 15 19 25 27 34 83 07 10 11 06 ...													
	生成的第一个初始归并段						生成的第二个初始归并段							

2. k 路归并

有了 m 个初始归并段(都是有序段),便可以进行 k 路归并了,即将 k 个初始归并段采用前面介绍的选择树法进行归并产生一个段,这样, m 个初始归并段产生多个这样的段,然后对这些段再采用选择树法进行归并,如此下去,直到只生成一个段为止,这个段就是最后生成的归并段。

m 个初始归并段进行 k 路归并,归并的遍数为 $\lceil \log_k m \rceil$ 。为了减少归并的遍数, k 越大越好,但归并的路数增多时,CPU 处理内部归并的时间也随之增多。

下面分析上述方法的时间复杂度,第一次建立选择树的比较所花时间为 $O(k-1)=O(k)$,此后每次重建选择树所需时间为 $O(\log_2 k)$,所以处理 n 个记录所需时间为初始建立选择树的时间加上 $n-1$ 次重建选择树的时间,即 $O((n-1) * \log_2 k) + O(k) = O(n * \log_2 k)$ 。这是 k 路归并一遍所需 CPU 处理时间,而归并遍数为 $\lceil \log_k m \rceil$,所以这个阶段总共需要的时间为:

$$O(n * \log_2 k * \log_k m) = O(n * \log_2 m)$$

从中可以看到 k 路归并 CPU 处理时间与 k 无关,这就是采用选择法实现 k 路归并的优点。

例如,有一个磁盘文件共有 4500 个记录,采用前面的方法生成的 6 个初始归并段如图 13.1 所示。将内存工作区三等分,每块能容纳 250 个记录,其中两块作为输入缓冲区,另一块作为输出缓冲区,其归并过程如图 13.2 所示。

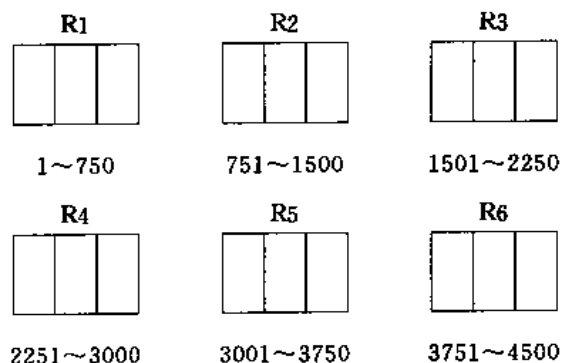


图 13.1 6 个初始归并段

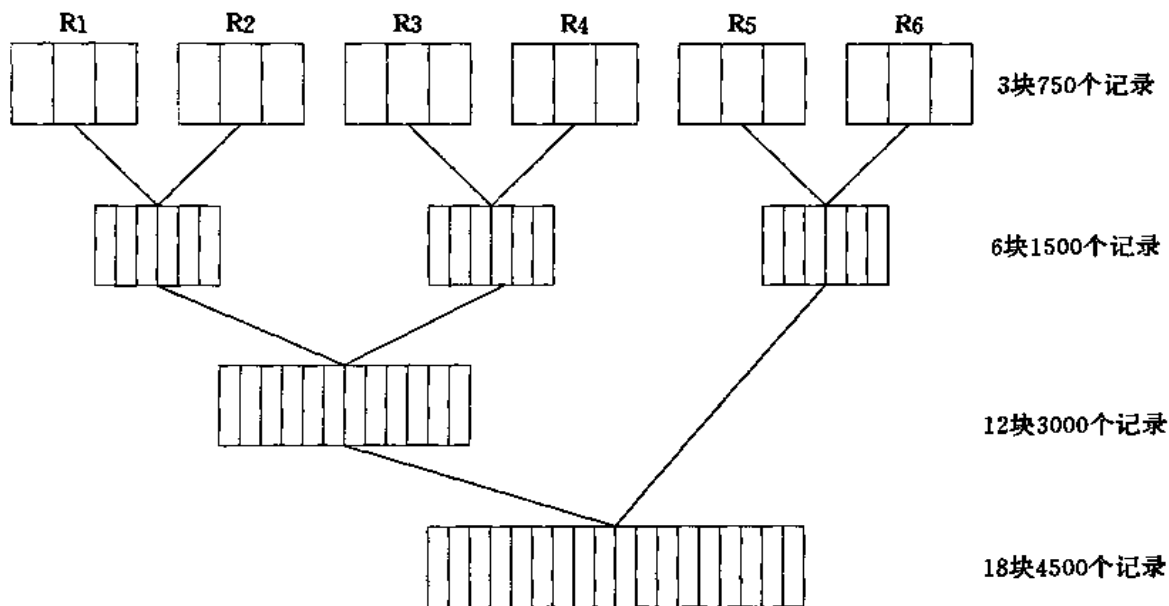


图 13.2 4路归并过程

13.1.2 磁带文件归并排序

磁带文件归并排序的基本步骤类似于磁盘文件归并排序,首先将被排序的文件分段输入,进行内部排序,生成初始归并段再写回磁带上,然后再把这些初始归并段进行多遍归并,最后形成一个单一归并段,就完成了对整个磁带文件的归并工作。磁带文件和磁盘文件排序的主要差别是初始归并段在外存储介质中的分布方式。磁盘是直接存取设备,读/写一个信息块的时间与当前读/写头所处的位置关系不大。而磁带则不然,磁带是顺序存取设备,读写一个信息块的时间与所读位信息块距当前读/写头的位置关系甚大。因此,对磁带文件的排序,研究归并段信息块的分布很重要。下面介绍两种较好的磁带文件归并排序法。

1. 平衡归并排序

k 路平衡归并排序需要使用 $2k$ 台磁带机: $T_1, T_2, \dots, T_k, T_{k+1}, \dots, T_{2k}$ 。开始时将初始归并段均匀地分布在 k 台磁带机 T_1, T_2, \dots, T_k 上,另外 k 台磁带机用作输出,即存放这遍归并的结果,对它们也采用同样形式的分布。当下一遍归并时,原来的输入带 T_1, T_2, \dots, T_k 变为输出带,而原来的输出带变为输入带,如此交替地进行,直到获得单一归并段为止,就完成了磁带文件的排序。

下面以 2 路平衡归并排序为例说明其过程。假设文件中记录总数为 6000,初始归并段长为 1000,4 台磁带机为 T_1, T_2, T_3 和 T_4 。开始时初始归并段的分布如下:

$T_1: R_1(1000), R_3(1000), R_5(1000)$

$T_2: R_2(1000), R_4(1000), R_6(1000)$

$T_3:$

$T_4:$

其中 $R_i(1000) (1 \leq i \leq 6)$ 表示归并段 R_i 的长度为 1000。

经过第一遍归并后,各磁带上的归并段的分布如下:

T_1 :
 T_2 :
 $T_3: R_1(2000), R_3(1000)$
 $T_4: R_2(2000)$

第二遍归并后,各磁带上的归并段的分布如下:

$T_1: R_1(1000)$
 $T_2: R_1(2000)$
 T_3 :
 T_4 :

第三遍归并后,各磁带上的归并段的分布如下:

T_1 :
 T_2 :
 $T_3: R_1(6000)$
 T_4 :

归并的结果在 T_3 上。

2. 多阶段归并排序

多阶段归并排序是使用 $k+1$ 台磁带机,每遍处理都是把 k 台磁带机中的归并段归并为更大的归并段,并将其放到剩下的那台磁带机中。

以 2 路归并为例,使用 3 台磁带机 T_1 、 T_2 和 T_3 ,假设初始归并段的长度为 L 。开始时,初始归并段放在 T_1 和 T_2 上,并且 T_1 和 T_2 中归并段数目不等,不妨假设 T_1 中归并段数目少些。将 T_1 和 T_2 中长为 L 的归并段归并成 $2L$ 的归并段,并将其放入 T_3 中,直到 T_1 空为止。此时 T_2 中还有一些长为 L 的归并段,然后将 T_2 中长为 L 的归并段与 T_3 中长为 $2L$ 的归并段进行归并,使之成为长 $3L$ 的归并段,并把它们放入 T_1 中,直到 T_2 空。再将 T_3 中长为 $2L$ 的归并段与 T_1 中长为 $3L$ 的归并段归并成长为 $5L$ 的归并段,将其放入 T_2 中,如此等等。

例如,初始归并段的数目为 34 段,每段长 L 。开始时初始归并段的分布是 T_1 中 13 段, T_2 中 21 段,其归并过程如下:

i 遍后	T_1	T_2	T_3	
开始	13(1L)	21(1L)		
1		8(1L)	13(2L)	(从 T_1 和 T_2 归并成 13 个 2L 长的段放到 T_3)
2	8(3L)		5(2L)	(从 T_2 和 T_3 归并成 8 个 3L 长的段放到 T_1)
3	3(3L)	5(5L)		(从 T_1 和 T_3 归并成 5 个 5L 长的段放到 T_2)
4		2(5L)	3(8L)	(从 T_1 和 T_2 归并成 3 个 8L 长的段放到 T_3)
5	2(13L)		1(8L)	(从 T_2 和 T_3 归并成 2 个 13L 长的段放到 T_1)
6	1(13L)	1(21L)		(从 T_1 和 T_3 归并成 1 个 21L 长的段放到 T_2)
7			1(34L)	(从 T_1 和 T_2 归并成 1 个 34L 长的段放到 T_3)

13.2 基本题

13.2.1 单项选择题

1. 外排序是指 ①。

- A. 在外存上进行的排序方法
- B. 不需要使用内存的排序方法
- C. 数据量很大,需要人工干预的排序方法
- D. 排序前后数据在外存,排序时数据调入内存的排序方法

答:①D

2. 磁盘文件采用选择法实现 k 路归并时,占用 CPU 的时间与 k ①。

- A. 有关
- B. 无关

答:①B

3. 磁盘文件有 m 个初始归并段,采用 k 路归并时,所需的归并遍数是 ①。

- A. $\log_2 k$
- B. $\log_2 m$
- C. $\log_k m$
- D. $\lceil \log_k m \rceil$

答:①

13.2.2 填空题(将正确的答案填在相应的空中)

1. 在选择树中,“败者”是指 ①。

答:①在一次比较中,没有上升进入其父结点的结点

2. 归并排序有两个基本阶段,第一阶段是 ①,第二阶段是 ②。

答:①生成初始归并段 ②对这些初始归并段采用某种归并方法,进行多遍归并

3. 磁带和磁盘的主要差别是 ①。

答:①磁盘是直接存取设备,读/写一个信息块的时间与当前读/写头所处的位置关系不大,而磁带是顺序存取设备,读写一个信息块的时间与所读位信息块距当前读/写头的位置关系很大。

4. 磁带文件和磁盘文件排序的主要差别是 ①。

答:①初始归并段在外存储介质中的分布方式

13.3 习题解析

1. 归并排序中使用的选择树和堆排序中的堆有什么差别?

答:选择树是由参加比较的 n 个元素作为叶子结点而得到的完全二叉树;而堆是 n 个元素 $R_i (i=1, 2, \dots, n)$ 的序列,它满足性质: $R_i \leq R_{2i}$ 且 $R_i \leq R_{2i+1} (1 \leq i \leq n/2)$,堆是一个含有 n 个结点的完全二叉树。

2. 设有磁盘文件中记录的关键字分别为:

10,20,15,25,12,13,21,30,8,16,10

用选择树法产生初始归并段,问可产生几个初始归并段?每个初始归并段包含哪些记录(工作区能容纳4个记录)。

解:内存缓冲区可容纳4个记录,采用4路归并的选择树方法生成初始归并段,表13.2给出了生成初始归并段过程中各步的缓冲区和输出结果。

表 13.2 初始归并段生成过程

步	1	2	3	4	5	6	7	8	9	10	11
缓冲区内容	10	12	13	21	21	21	(16)	(16)	16	16	
	20	20	20	20	20	(8)	(8)	(8)			
	15	15	15	15	30	30	30	30			
	25	25	25	25	25	25	25	(10)	10		
输出结果	10	12	13	15	20	21	25	30	8	10	16
	生成的第一个初始归并段						生成的第二个初始归并段				

3. 假设4个初始归并段如下:

$R_1: 15, 16, 25, 32$

$R_2: 3, 22, 28, 45$

$R_3: 1, 12, 30, 42$

$R_4: 33, 60$

给出进行4路归并的过程。

解:本题的4路归并过程如下;

(1)输出1

$R_1: 15, 16, 25, 32$

$R_2: 3, 22, 28, 45$

$R_3: 1, 12, 30, 42$

$R_4: 33, 60$

(2)输出1,3

$R_1: 15, 16, 25, 32$

$R_2: 22, 28, 45$

$R_3: 12, 30, 42$

$R_4: 33, 60$

(3)输出1,3,12

$R_1: 15, 16, 25, 32$

$R_2: 22, 28, 45$

$R_3: 30, 42$

$R_4: 33, 60$

(4)输出 1, 3, 12, 15

$R_1: 16, 25, 32$

$R_2: 22, 28, 45$

$R_3: 30, 42$

$R_4: 33, 60$

(5)输出 1, 3, 12, 15, 16

$R_1: 25, 32$

$R_2: 22, 28, 45$

$R_3: 30, 42$

$R_4: 33, 60$

(6)输出 1, 3, 12, 15, 16, 22

$R_1: 25, 32$

$R_2: 28, 45$

$R_3: 30, 42$

$R_4: 33, 60$

(7)输出 1, 3, 12, 15, 16, 22, 25

$R_1: 32$

$R_2: 28, 45$

$R_3: 30, 42$

$R_4: 33, 60$

(8)输出 1, 3, 12, 15, 16, 22, 25, 28

$R_1: 32$

$R_2: 45$

$R_3: 30, 42$

$R_4: 33, 60$

(9)输出 1, 3, 12, 15, 16, 22, 25, 28, 30

$R_1: 32$

$R_2: 45$

$R_3: 42$

$R_4: 33, 60$

(10)输出 1, 3, 12, 15, 16, 22, 25, 28, 30, 32

$R_1:$

$R_2: 45$

$R_3: 42$

$R_4: 33, 60$

(11) 输出 1, 3, 12, 15, 16, 22, 25, 28, 30, 32, 33

$R_1:$

$R_2: 45$

$R_3: 42$

$R_4: 60$

(12) 输出 1, 3, 12, 15, 16, 22, 25, 28, 30, 32, 33, 42

$R_1:$

$R_2: 45$

$R_3:$

$R_4: 60$

(13) 输出 1, 3, 12, 15, 16, 22, 25, 28, 30, 32, 33, 42, 45

$R_1:$

$R_2:$

$R_3:$

$R_4: 60$

(14) 输出 1, 3, 12, 15, 16, 22, 25, 28, 30, 32, 33, 42, 45, 60

$R_1:$

$R_2:$

$R_3:$

$R_4:$

4. 以 10 个长度为 L 的归并段为例, 用 2 路平衡归并法进行排序, 写出归并过程中各磁带内容的变化情况。

解: 2 路平衡归并使用 4 台磁带机: T_1, T_2, T_3 和 T_4 , 开始时初始归并段的分布如下:

$T_1: R_1(1L), R_3(1L), R_5(1L), R_7(1L), R_9(1L)$

$T_2: R_2(1L), R_4(1L), R_6(1L), R_8(1L), R_{10}(1L)$

$T_3:$

$T_4:$

其中 $R_i(1L) (1 \leq i \leq 10)$ 表示归并段 R_i 的长度为 $1L$ 。

经过第一遍归并后, 各磁带上的归并段的分布如下:

$T_1:$

$T_2:$

$T_3: R_1(2L), R_3(2L), R_5(2L)$

$T_4: R_2(2L), R_4(2L)$

经过第二遍归并后,各磁带上的归并段的分布如下:

$T_1: R_1(4L), R_3(2L)$

$T_2: R_2(4L)$

$T_3:$

$T_4:$

经过第三遍归并后,各磁带上的归并段的分布如下:

$T_1:$

$T_2:$

$T_3: R_1(8L)$

$T_4: R_2(2L)$

经过第四遍归并后,各磁带上的归并段的分布如下:

$T_1: R_1(10L)$

$T_2:$

$T_3:$

$T_4:$

5. 以 55 个长度为 L 的归并段为例,用 2 路多阶段归并法进行排序,写出归并过程中各磁带内容的变化情况。

解:2 路多阶段归并使用 3 台磁带机: T_1 、 T_2 和 T_3 ,假设开始时初始归并段的分布是 T_1 中 20 段, T_2 中 35 段,其归并过程如下:

i 遍后	T_1	T_2	T_3	
开始	20(1L)	35(1L)		
1		15(1L)	20(2L)	(从 T_1 和 T_2 归并成 20 个 2L 长的段放到 T_3)
2	15(3L)		5(2L)	(从 T_2 和 T_3 归并成 15 个 3L 长的段放到 T_1)
3	10(3L)	5(5L)		(从 T_1 和 T_3 归并成 5 个 5L 长的段放到 T_2)
4	5(3L)		5(8L)	(从 T_1 和 T_2 归并成 5 个 8L 长的段放到 T_3)
5		5(11L)		(从 T_1 和 T_3 归并成 5 个 11L 长的段放到 T_2)

参考文献

- [1]李春葆编著. 数据结构习题与解析(PASCAL 篇) 清华大学出版社 1999
- [2]严蔚敏, 吴伟民. 数据结构(C 语言版) 清华大学出版社 1997
- [3]许卓群, 张乃孝, 杨冬青, 唐世渭. 数据结构 高等教育出版社 1988
- [4]赵文静编. 数据结构—C++ 语言描述 西安交通大学出版社 1999
- [5]陈文博, 朱青. 数据结构与算法 机械工业出版社 1996
- [6]朱忠才. 数据结构习题集和试题集 学苑出版社 1993
- [7]徐孝凯. 数据结构简明教程 清华大学出版社 1995
- [8]徐孝凯, 魏荣. 数据结构 机械工业出版社 1996
- [9]严蔚敏, 陈文博. 数据结构 机械工业出版社 1990
- [10]刘光富等. 全国计算机硕士研究生入学试题集 湖北科学技术出版社 1985
- [11]李莲治, 姜文清, 郭福顺. 数据结构 大连理工大学出版社 1989
- [12]本书编译组. 数据结构原理 上海科学技术文献出版社 1988
- [13]冯博琴, 刘跃虎, 张浩, 陆诗岩. 新编计算机等级考试练习题及模拟试题(1 级、2 级、3 级) 西安交通大学出版社 1998
- [14]邹海明, 余祥宣. 计算机算法基础 华中理工大学出版社 1985
- [15]陈岐, 陈云霞. 计算机奥林匹克训练教程 中国科学技术大学出版社 1997
- [16]朱敏. PASCAL 程序设计教程 东南大学出版社 1995

清华大学出版社最新计算机图书推荐

图形图像处理技术系列丛书:

3D Studio MAX R3培训教程	40.00元
3D Studio MAX R2.5实用速查手册	22.00元
3D Studio MAX V2实用速查手册	22.00元
3D Studio MAX R2.5学习捷径(材质、动画和粒子系统篇)	52.00元
3D Studio MAX R2.5学习捷径(基本操作与建模篇)	38.00元
3D Studio MAX R2技术精粹(第1卷)(带光盘)	98.00元
3D Studio MAX R2技术精粹(第2卷)(带光盘)	52.00元
3D Studio MAX R2技术精粹(第3卷)(带光盘)	58.00元
Photoshop 5艺术(摄影、美术、产品广告、包装设计实战教程)	72.00元
Photoshop 5影像设计宝典(全彩印刷)	158.00元
急救平面设计实用手册(全彩印刷)	98.00元

计算机应用与培训系列教材:

电脑自维修手册	24.00元
电脑工程师维修教程	62.00元
中文Word 2000培训教程	28.00元
中文Excel 2000培训教程	28.00元
Windows 98学习捷径(中文版)	38.00元
Windows 98学习大全	60.00元
Windows 98学习捷径(中文版)	38.00元
中文Windows 98培训教程	28.00元
中文Office 97培训教程	48.00元
中文Excel 97培训教程	28.00元
中文PowerPoint 97培训教程	24.00元
中文Access 97培训教程	24.00元
Visual Basic 6.0(中文版)学习捷径(带光盘)	43.00元
新编微机培训教程	24.00元
最新微机操作技术	28.00元
Internet操作技术	30.00元
数据结构习题与解析(Pascal篇)	28.00元
数据结构习题与解析(C语言篇)	28.00元
C语言与习题解答	28.00元

ISBN 7-302-03745-0



9 787302 037453 >

定价: 28.00元

读者联系电话: (010)62562448 82589258

出版社网址: WWW.tup.tsinghua.edu.cn

设计制作: 黄庭繁