



第1章 绪论	1
1.1 基础知识篇	1
1.1.1 基本概念和术语	1
1.1.2 数据的逻辑结构	2
1.1.3 数据的存储结构	2
1.1.4 算法的描述	2
1.2 实训篇	3
1.2.1 实训步骤	3
1.2.2 C语言环境中的动态调试法	4
1.2.3 常用的调试途径	5
1.2.4 实训报告的内容与书写格式	6
1.3 习题解析	6
第2章 线性表	10
2.1 基础知识篇	10
2.1.1 基本概念	10
2.1.2 线性表的顺序存储及运算实现	11
2.1.3 线性表的链式存储及运算实现	14
2.1.4 顺序表和链表的比较	17
2.2 实训篇	18
2.2.1 超市密码存储箱系统的设计与实现	18
2.2.2 员工通讯录管理系统	25
2.2.3 约瑟夫环问题	32
2.2.4 模拟蜗牛爬树	35
2.3 实训项目	40
2.4 基本题	41
2.4.1 选择题	41
2.4.2 填空题	41
2.5 习题解析	44
第3章 栈和队列	54
3.1 基础知识篇	54
3.1.1 栈	54
3.1.2 队列	59

3.2 实训篇	65
3.2.1 栈的应用——表达式求值问题	66
3.2.2 队列的应用——迷宫问题	71
3.3 实训项目	75
3.4 基本题	76
3.4.1 选择题	76
3.4.2 填空题	76
3.5 习题解析	78
第4章 串	86
4.1 基础知识篇	86
4.1.1 串及其存储结构	86
4.1.2 串的基本运算	89
4.2 实训篇	92
4.3 实训项目	94
4.4 基本题	95
4.4.1 选择题	95
4.4.2 填空题	95
4.5 习题解析	96
第5章 树	102
5.1 基础知识篇	102
5.1.1 树的基本概念	102
5.1.2 二叉树	102
5.1.3 树的存储结构	106
5.1.4 哈夫曼树	107
5.2 实训篇	108
5.2.1 二叉树的建立及三种遍历的实现	108
5.2.2 哈夫曼编码/译码系统	116
5.2.3 六皇后问题求解及演示	122
5.3 实训项目	125
5.4 基本题	125
5.4.1 选择题	125
5.4.2 填空题	126
5.5 习题解析	127
第6章 图	140
6.1 基础知识篇	140
6.1.1 图的基本概念	140
6.1.2 图的存储结构	143
6.1.3 图的基本操作	145

6.2 实训篇	153
6.2.1 图的遍历问题	154
6.2.2 所有顶点对最短路径问题	156
6.3 实训项目	160
6.4 基本题	161
6.4.1 选择题	161
6.4.2 填空题	163
6.5 习题解析	164
第7章 查找	170
7.1 基础知识篇	170
7.1.1 基本概念	170
7.1.2 静态查找	170
7.1.3 动态查找	172
7.2 实训篇	175
7.2.1 手机短信内容中电话号码和手机号码的识别与提取	176
7.2.2 数据结构教材中各功能函数的存放与查找	180
7.2.3 二叉排序树的使用	184
7.3 实训项目	188
7.4 基本题	188
7.4.1 选择题	188
7.4.2 填空题	189
7.5 习题解析	190
第8章 排序	199
8.1 基础知识篇	199
8.1.1 排序的基本概念	199
8.1.2 插入排序	200
8.1.3 交换排序	201
8.1.4 选择排序	202
8.1.5 归并排序	204
8.1.6 基数排序	204
8.1.7 内部排序法的比较和选择	205
8.2 实训篇	205
8.2.1 药店的药品销售统计系统	205
8.2.2 学生成绩排名	214
8.2.3 电视大赛观众投票及排名系统	218
8.3 实训项目	221
8.4 基本题	222
8.4.1 选择题	222

10 / 数据结构实训与习题解析 □

8.4.2 填空题	222
8.5 习题解析	223
附录 基本题参考答案	232
参考文献	235

第 1 章

绪 论

1.1 基础知识篇

学习要点:

- 数据、数据结构、算法的基本概念
- 算法评价:时间复杂度、空间复杂度的概念
- 程序设计的步骤
- 程序的调试方法
- 程序的测试方法
- 实验报告的内容及格式

1.1.1 基本概念和术语

1. 数据(Data)

数据是对客观事物的符号表示。在计算机科学中其含义是指所有能够输入到计算机中并被计算机程序处理的符号集合。

2. 数据元素(Data Element)

数据元素是数据集合中的一个实体,是计算机程序加工处理的基本单位。数据元素按其组成可分为简单型数据元素和复杂型数据元素。简单型数据元素由一个数据项组成,所谓数据项就是数据中不可再分割的最小单位;复杂型数据元素由多个数据项组成。

3. 数据类型(Data Type)

数据类型是一组性质相同的值的集合以及定义于这个集合上的一组操作的总称。

4. 数据对象(Data Object)

数据对象是性质相同的数据元素组成的集合,是数据的一个子集。例如,整数数据对象的集合可表示为 $N = \{0, \pm 1, \pm 2, \dots\}$, 字母字符数据对象的集合可表示为 $C = \{'A', 'B', \dots, 'Z'\}$ 。

5. 数据结构(Data Structure)

数据结构的基本含义是指数据元素之间的关系,换句话说,数据元素之间的相互关系称为结构,带有结构的数据对象称为数据结构。

1.1.2 数据的逻辑结构

数据结构中所说的“关系”实际上是指数据元素之间的逻辑关系,又称为逻辑结构。常见的数据逻辑结构有:集合、线性结构、树形结构和图形结构。

1.1.3 数据的存储结构

数据的存储结构(或称为物理结构)是指数据结构在计算机存储器中的具体实现。

常见的存储结构:顺序存储结构——借助于数据元素的相对存储位置来表示数据元素之间的逻辑结构;链式存储结构——借助于指示数据元素地址的指针来表示数据元素之间的逻辑结构。

1.1.4 算法的描述

1. 算法的概念

算法是解决某个特定问题的一种方法或一个有限过程。

计算机对数据的操作可以分为数值型和非数值型两种类型。在数值型操作中主要进行的是算术运算;而在非数值型操作中主要进行的是检索、排序、插入、删除等等。

2. 设计算法的基本过程

通过对问题进行详细地分析,抽象出相应的数学模型;

确定使用的数据结构,并在此基础上设计各种操作的算法;

选用某种语言将算法转换成程序;

调试并运行这些程序。

3. 算法应该具有下列五个特性

(1)有穷性:一个算法必须在执行有穷步之后结束。

(2)确定性:算法中的每一步,必须有确切的含义,不会产生二义性。

(3)可行性:算法中描述的每一步操作都可以通过已有的基本操作执行有限次实现。

(4)输入:一个算法应该有零个或多个输入。

(5)输出:一个算法应该有一个或多个输出。这里所说的输出是指与输入有某种特定关系的量。

4. 算法的设计要求

(1)正确性:要求算法能够正确地执行预先规定的功能,并达到所期望的性能要求。

(2)可读性:为了便于理解、测试和修改,算法应该具有良好的可读性。

(3)健壮性:算法中拥有对输入数据、打开文件、读取文件记录、分配内存空间等操作的结果检测,并通过与用户对话的形式做出相应的处理选择。

(4)高效性:算法的效率是指算法执行的时间和占用的存储空间。如果对于一个问题有多个算法可以实现,应尽可能选择执行时间短、占用空间少的算法。

5. 算法的评价

评价一个算法优劣的主要指标是时间效率和空间效率。

(1) 算法的时间效率

算法的时间效率是指算法运行时所需时间的多少。算法的时间效率主要由两个因素决定:所需处理问题的数据量大小;在解决问题的过程中,基本操作的执行次数。

时间效率的分析:如果我们将一个算法所花费的时间设计成一个以数据量 n 为自变量的函数 $T(n)$,这个函数在正整数定义域范围内一定是单调递增的。好的算法应该能够在数据量 n 增长时,使函数 $T(n)$ 的值以比较缓慢的速度增长。

(2) 算法的空间效率

算法的空间效率是指在算法的执行过程中,所占据辅助空间的大小。辅助空间就是除算法代码本身和输入输出数据所占据的空间外,算法临时开辟的存储空间。在有些算法中,占据辅助空间的大小与所处理的数据量有关,而有些却与其无关。后一种是较理想的情况。在设计算法时,应该注意空间效率。

1.2 实训篇

1.2.1 实训步骤

程序设计应该遵循软件开发的一般规律和步骤。其主要步骤如下:

1. 问题分析

正确地理解问题,分析究竟要“做什么”,分析问题已知的信息及其作用,分析在解决问题中对信息处理的规则、要求及限制条件,分析解决问题后应该输出什么样的结果(输出的形式、格式等),并分析得出判定结果是否正确的标准。

2. 设计分析

得出解决问题的思路、主要流程、采用的数据结构类型的说明、主要算法的思想。

3. 设计方案

采用的数据结构类型的定义、主要算法的描述及说明。

4. 编写程序

根据问题的要求和已得到的算法编写程序。

5. 调试程序

目的是发现程序中的语法错误及一些逻辑错误(还有一部分逻辑错误在测试中才能被发现),并修改,使程序能够运行。可以采用 C 语言环境提供的动态调试法。

6. 测试程序

选择若干具有代表性的输入数据,其中包括合理的数据(包括边界值)和不合理的数据,进行测试,尽量使程序中的各语句和分支都被检查到,以便发现程序中的错误和漏洞,然后针对出现的问题,对算法和程序进行修改。

7. 文档编制

开发者编制程序文档的主要目的是为了让别人了解自己编写的算法和程序。文档包括源程序代码、算法(程序)的流程图、开发过程中各阶段的有关记录、算法(程序)的正确

性证明、程序的测试结果、对输入/输出的要求及格式的详细描述等等。

1.2.2 C 语言环境中的动态调试法

1. 单步执行和跟踪进入

使用“Run”菜单中的“Step over”命令或按 F8 功能键可进入“单步执行”方式,单步执行方法的特点是:程序一次执行一行,在执行完一行后就停下来,用户可以检查此时各有关变量和表达式的值,以便发现问题所在。但这种单步执行不能进入函数的内部,如果要进入到函数内部进行跟踪,则应利用“Run”菜单中的“Trace into”(跟踪进入)命令或按 F7 功能键。表 1-1 列出了 Turbo C 环境下 Run 菜单中各选项的功能。

表 1-1 Run 菜单中各选项的功能

选项	功 能
Run	编译,链接生成目标文件和可执行文件并运行。如果选择开关 Debug/Source debug 为 on,则程序执行到每一个断点都会暂停下来;再次选择该选项,程序将继续执行到下一个断点处或执行到结束处。
Program reset	用于动态调试。其作用是终止当前的调试操作,释放分配给程序的内存空间,关闭已打开的文件,但不改变断点设置。
Go to cursor	用于动态调试。使程序从执行行开始运行到编辑窗口中的光标所在行上。若光标所在行不含可执行代码语句,则显示一个 Esc 框,给出警告。
Trace into	用于动态调试。单步跟踪程序执行,并跟踪进入被调用的子函数,包括系统函数。
Step over	用于动态调试。单步跟踪程序执行,但不跟踪进入被调用的子函数。
User screen	显示程序的输出结果(把编辑窗口切换到程序输出窗口)。

2. 设置断点和监视表达式

在程序可能会出现问题或需要重点分析的位置,设置断点和监视表达式,程序执行到这里便暂停运行,这样即可通过观察变量值的变化对程序进行分析。

断点的设置方法:将光标移到某一行上,按 Ctrl + F8 组合键或使用“Break/watch”菜单中的“Toggle breakpoint”命令,此行就被颜色条覆盖,作为断点行。在一个程序中可以设置多个断点,运行时遇到断点行就会暂停,此时可以观察变量值的变化。如果想继续运行,再按一次 Ctrl + F9 组合键即可,程序会从暂停处继续向后运行。若要取消断点行,则将光标移到断点行上,再一次按 Ctrl + F8 组合键,此断点行即被取消;或使用“Break/watch”菜单中的“Clear all breakpoints”命令,这时程序中的所有断点行将全部被取消。

监视表达式的设置方法:按 Ctrl + F7 组合键或使用“Break/watch”菜单中的“Add watch”命令,会弹出一个添加表达式输入框。在输入框中输入表达式并确认,被监视的表达式即设置完成,当程序执行到这个表达式时,在屏幕下方的“Watch”窗口将显示表达式的值。若要删除一个或全部监视表达式,可以使用“Break/watch”菜单中的“Delete watch”命令或“Remove all watches”命令。

表 1-2 列出了 Turbo C 环境下 Break/watch 菜单各选项的功能。

表 1-2 Break/watch 菜单各选项的功能

选项	功能
Add watch	弹出一个 Add watch 窗口,在该窗口中输入监视表达式后,watch 窗口中将显示表达式的值。
Delete watch	删除监视窗口中最后加入的监视表达式。也可把光标移到 watch 窗口中的某一表达式处,用 Delete 键或 Ctrl + Y 键删除。
Edit watch	弹出一个编辑窗口,在该窗口中编辑最后加入的监视表达式。
Remove all watches	删除观察窗口(watch 窗口)中的所有表达式。
Toggle breakpoint	在光标所在行设置断点,再次执行该命令时删除该断点。
Clear all breakpoints	删除所有断点。
View next breakpoint	把光标移到下一个断点处。

1.2.3 常用的调试途径

1. 试探法

分析错误的症状,推测问题所在的位置,利用在程序中设置输出语句,分析内存中的内容等手段来获得错误的线索,一步步地试探和分析错误所在。这种方法效率很低,适合于结构比较简单的程序。

2. 回溯法

从发生错误症状的地方开始,人工沿程序的控制流往回追溯分析源程序代码,直到找出错误原因为止。这种方法对调试小程序较为有效,对于大规模的程序,由于其需要回溯的路径太多而变得不可操作,但可利用它单独调试大程序中的各个模块。

3. 原因排除法

包括对分查找法、归纳法和演绎法。

对分查找法:如果已经知道每个变量在程序中若干个关键点的正确值,则可以用赋值语句或输入语句在程序中关键点附近“注入”这些变量的正确值,然后运行程序并检查所得到的输出。如果输出结果是正确的,则错误原因在程序的前半部分;反之,错误原因在程序的后半部分。对错误原因所在的那部分再反复使用这种方法,直到把出错范围缩小到容易诊断的程度为止。

归纳法:首先把与错误有关的数据组织起来进行分析,以便发现可能的错误原因。然后导出对错误原因的一个或多个假设,并利用已有的数据来证明或排除这些假设;若已有的数据不能证明或排除这些假设,则需设计并执行一些新的测试用例,以获得更多的数据。

演绎法:首先设想所有可能的出错原因,然后试图用测试来排除每一个假设的原因。

如果测试表明某个假设的原因可能是真的原因,则对数据进行细化以准确定位错误。

1.2.4 实训报告的内容与书写格式

表 1-3						实训报告	
姓名		学号		成绩			
同组人姓名		实训日期					
实 训 项 目							
实 训 目 的							
设 计 分 析	(解决问题的思路、主要流程,采用的数据结构类型的说明,主要算法的思想。)						
设 计 方 案	(采用的数据结构类型的定义,主要算法的描述及说明。)						
详 细 设 计	(项目实现的源程序清单,其中要有各自定义函数的详细注释,关键语句也要有注释说明。)						
用 户 手 册	(使用本程序的说明书,说明如何使用此程序,详细列出每一步的操作步骤。)						
调试及结果 分 析	(列举调试过程中所用的不同的输入数据,以及得到的不同的测试结果,分析其原因;在调试过程中遇到的主要问题及其解决方法;算法的时间复杂度及空间复杂度的分析。)						
学 习 心 得	(利用数据结构知识来解决本项目实践的体会,对设计的经验总结及改进思想。)						
教师评审意见							

1.3 习题解析

1. 简述下列术语:数据、数据元素、数据项、数据类型、数据对象、数据结构。

解 数据:指能够输入到计算机中,并能被计算机处理的一切对象。

数据元素:数据元素是数据的基本单位,它可以分割成若干个具有不同属性的项(字段)。

数据项:数据项是具有独立意义的最小数据单位,是对数据元素属性的描述。

数据类型:一组性质相同的值的集合以及定义于这个集合上的一组操作的总称。每个数据项属于某一确定的基本数据类型。

数据对象:性质相同的数据元素的集合,是数据的一个子集。

数据结构:指数据元素之间的关系,它是按照某种关系组织起来的一批数据,以一定的存储方式存储到计算机存储器中。在任何问题中,数据元素都不是孤立存在的,它们之间存在着某种关系,数据元素之间的这种相互关系就称为结构,带有结构的数据对象称为

数据结构。

2. 举出几个线性结构、树形结构和网状结构的数据结构的例子。

解 (1) 线性结构—学校电话(分机)号码表

表 1-4 学校电话(分机)号码表

序号	部门	电话号码
1	校长办公室	0001
2	党委办公室	0002
3	物理系	0003
4	数学系	0004
5	外语系	0005
	⋮	⋮

注:表中每一行为一个数据元素。

(2) 树形结构—市、区、街道隶属关系图

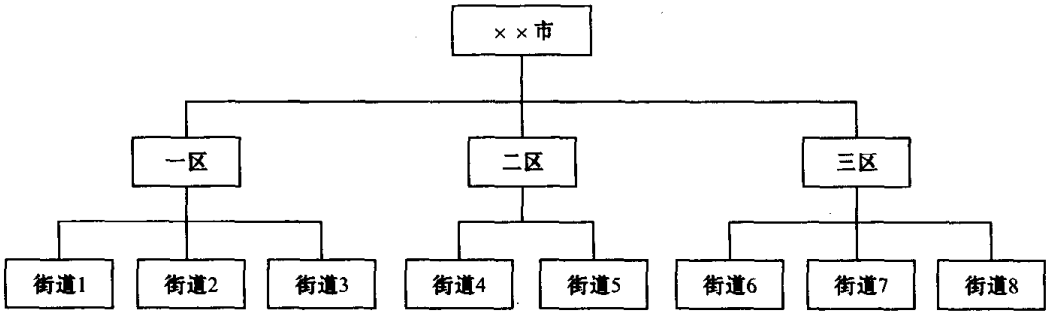


图 1.1 市、区、街道隶属关系图

(3) 网状结构—城市间航线图(此图为虚拟情况)

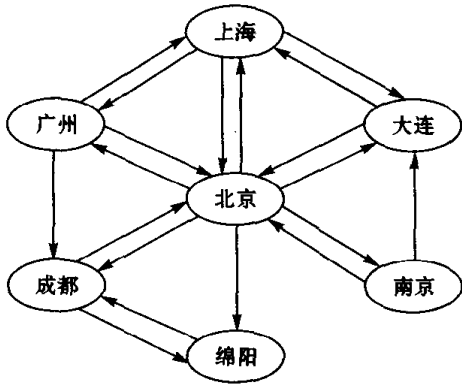


图 1.2 城市间航线图

3. 什么叫算法? 算法设计的目标是什么?

解 算法是对某一特定问题求解步骤的一种描述。算法是由若干条指令组成的有穷序列,其中每一条指令表示计算机的一个或多个操作。算法必须具备 5 个特征:输入、输出、有穷性、确定性、可行性。

算法设计的要求(或目标)包含以下几点:

(1)正确性:所设计的算法要满足具体问题的要求。在给算法输入合理数据的情况下,能在有限的时间内得出正确的结果。

(2)可读性:算法必须可以供他人使用。为了阅读与交流,所设计的算法要让他人能看懂,在算法或程序中可以增加一些注释来提高可读性。

(3)健壮性:当输入的数据不符合要求时,算法应能判断出数据的非法性,并能进行适当的处理。

(4)高效性:对于同一个问题有多个算法可供选择时,应尽可能选择执行时间短、占用空间少的算法。

4. 怎样对算法进行评价?

解 算法首先要具备正确性,然后是可读性和健壮性。在具备了这三个条件后,主要应考虑算法的效率问题,包括算法的时间效率和空间效率两个方面。这是评价算法优劣的主要指标。

时间效率就是考虑一个算法运行时所需时间的多少。但一个算法在计算机上运行所需要的实际时间是较难计算出来的,即使能计算出来,同一算法在不同的计算机上运行所需要的实际时间也是不一样的。由于一个算法运行所需要的时间与算法中语句的执行次数成正比,因此,一般通过统计算法中语句的执行次数(语句频度)来确定算法的时间效率(或称算法的时间复杂度),把算法的时间复杂度记为: $T(n) = O(f(n))$ 。

空间效率是指一个算法在执行过程中所占用的额外存储空间大小,又称为空间频度或空间复杂度。

5. 对下列用二元组表示的数据结构,画出它们的逻辑结构图,并指出它们各属于哪类数据结构。

(1) $S = (D, R)$, 其中

$D = \{a, b, c, d, e, f\}$

$R = \{ \langle a, b \rangle, \langle b, c \rangle, \langle c, d \rangle, \langle d, e \rangle, \langle e, f \rangle \}$

(2) $S = (D, R)$, 其中

$D = \{1, 2, 3, 4, 5, 6, 7\}$

$R = \{ \langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 2, 4 \rangle, \langle 2, 5 \rangle, \langle 3, 6 \rangle, \langle 3, 7 \rangle \}$

(3) $S = (D, R)$, 其中

$D = \{a_1, a_2, a_3, a_4, a_5, a_6\}$

$R = \{ \langle a_1, a_2 \rangle, \langle a_1, a_5 \rangle, \langle a_1, a_6 \rangle, \langle a_2, a_3 \rangle, \langle a_2, a_4 \rangle, \langle a_4, a_5 \rangle, \langle a_5, a_6 \rangle \}$

解 (1)线性结构,逻辑结构图如图 1.3 所示。

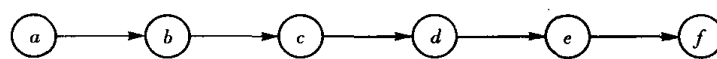


图 1.3 线性结构逻辑结构图

(2) 树形结构, 逻辑结构图如图 1.4 所示。

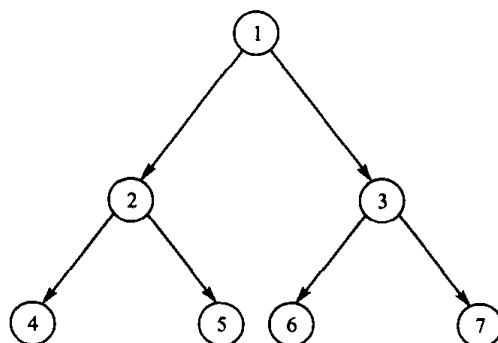


图 1.4 树形结构逻辑结构图

(3) 网状结构, 逻辑结构图如图 1.5 所示。

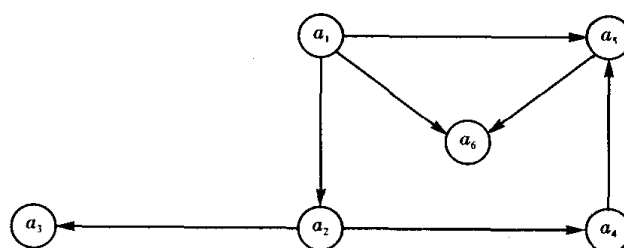


图 1.5 网状结构逻辑结构图

第 2 章

线性表

2.1 基础知识篇

学习要点:

- 线性表的概念
- 线性表的各种存储结构:顺序表、单链表、循环链表、双向链表的定义及基本操作的算法实现
- 线性表各种存储结构的优缺点比较及适用范围

2.1.1 基本概念

1. 线性表的定义

线性表是由 $n(n \geq 0)$ 个类型相同的数据元素组成的有限序列。通常表示成下列形式:

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

式中: L 为线性表名称; a_i 为线性表中第 i 个数据元素 ($i = 1, 2, \dots, n$), 通常我们将它的数据类型抽象为 datatype, datatype 根据具体问题而定, 如在学生情况信息表中, 它是用户自定义的学生类型; 在字符串中, 它是字符型等等。

线性表中数据元素的个数称为线性表的长度, 当 $n = 0$ 时, 线性表为空, 称为空线性表。

2. 线性表的基本操作

(1) 初始化线性表 L : InitList(L)

初始条件: 表 L 不存在

操作结果: 构造一个空的线性表

(2) 销毁线性表 L : DestroyList(L)

初始条件: 表 L 存在

操作结果: 回收表 L 所占的存储空间

(3) 清空线性表 L : ClearList(L)

初始条件: 表 L 存在

操作结果: 表 L 变为空表

(4) 求线性表 L 的长度: ListLength(L)

- 初始条件:表 L 存在
操作结果:返回线性表中所含元素的个数
- (5)判断线性表 L 是否为空:IsEmpty(L)
初始条件:表 L 存在
操作结果:L 为空返回真,否则为假。
- (6)获取线性表 L 中的某个数据元素内容 GetElem(L,i,e)
初始条件:表 L 存在且 $1 \leq i \leq \text{ListLength}(L)$
操作结果:返回线性表 L 中第 i 个元素的值或地址
- (7)检索值为 e 的数据元素 LocateElem(L,e)
初始条件:线性表 L 存在
操作结果:在线性表 L 中查找值为 e 的数据元素,若找到,则返回在线性表 L 中出现的第一个值为 e 的元素的下标或地址,查找成功;否则,返回一特殊值(如空下标值 -1 或空指针),查找失败。
- (8)在线性表 L 中插入一个数据元素 ListInsert(L,i,e)
初始条件:线性表 L 存在,插入位置合理 ($1 \leq i \leq n+1, n$ 为插入前的表长)。
操作结果:在线性表 L 的第 i 个位置上插入一个值为 e 的新元素,这样使原序号为 $i, i+1, \dots, n$, 它的数据元素的序号变为 $i+1, i+2, \dots, n+1$, 插入后表长为 $n+1$ 。
- (9)删除线性表 L 中第 i 个数据元素 ListDelete(L,i)
初始条件:线性表 L 存在,删除位置合理 ($1 \leq i \leq n, n$ 为删除前的表长)。
操作结果:在线性表 L 中删除序号为 i 的数据元素并将它的值存入 e 中,删除后使序号为 $i+1, i+2, \dots, n$, 它的元素的序号变为 $i, i+1, \dots, n-1$, 新表长为 $n-1$ 。

2.1.2 线性表的顺序存储及运算实现

1. 线性表的顺序存储结构

线性表的顺序存储结构是指用一组连续的存储单元依次存储线性表中的每个数据元素。如图 2.1 所示。

存储地址	内存单元
d	a_1
d + L	a_2
$\vdots$	$\vdots$
d + (i - 1)L	a_i
$\vdots$	$\vdots$
d + (n - 1)L	a_n

图 2.1 线性表的顺序存储结构

其中,L 为每个数据元素所占据的存储单元数目。

相邻两个数据元素存储位置的计算公式为:

$$LOC(a_{i+1}) = LOC(a_i) + L$$

线性表中任意一个数据元素的存储位置的计算公式为:

$$LOC(a_i) = LOC(a_1) + (i-1) * L$$

顺序存储结构的定义:

```
typedef struct
{ datatype data[MAXSIZE];
  int length;
} SeqList;
```

顺序存储结构的特点:

(1) 利用数据元素的存储位置来表示线性表中相邻数据元素之间的前后关系,即线性表的逻辑结构与存储结构(物理结构)一致;

(2) 在访问线性表时,可以利用上述给出的数学公式快速地计算出任何一个数据元素的存储地址。因此,我们可以粗略地认为,访问每个数据元素所花费的时间相等。这种存取元素的方法被称为随机存取法,使用这种存取方法的存储结构被称为随机存储结构。

2. 顺序表上基本运算的实现

顺序表的类型定义:

```
typedef struct
{
  ElemType *elem;
  int length;
} SSTable;
```

要注意理解 SSTable ST 定义的 ST 并不是一个真正的顺序表空间,真正的顺序表空间是一个数组,如 ElemType A[100],ST 只是用于记录这个顺序表的信息,例如:基址 ST.elem = A,长度 ST.length = 12。

下面介绍线性表中典型操作的算法实现。

(1) 初始化线性表 L

```
int InitList(SeqList *L)
{
  L->elem = (ElemType *)malloc(LIST_MAX_LENGTH * sizeof(ElemType));
  /* 分配空间 */
  if (L->elem == NULL) return ERROR; /* 若分配空间不成功,返回 ERROR */
  L->length = 0; /* 将当前线性表长度置 0 */
  return OK; /* 成功返回 OK */
}
```

(2) 销毁线性表 L

```
void DestroyList(SeqList *L)
{
  if (L->elem != NULL)
    free(L->elem); /* 释放线性表占据的所有存储空间 */
}
```

(3) 清空线性表 L

```
void ClearList(SeqList *L)
```

```
{
    L->length=0;
```

```
/* 将线性表的长度置为 0 */
```

(4) 求线性表 L 的长度

```
int ListLength (SeqList *L)
```

```
{
    return (L->length);
}
```

(5) 判断线性表 L 是否为空

```
int IsEmpty(SeqList L)
```

```
{
    if (L.length == 0) return TRUE;
    else return FALSE;
}
```

(6) 获取线性表 L 中的某个数据元素的内容

```
int GetElem(SeqList L,int i,ElemType *e)
```

```
{
    if (i < 1 || i > L.length) return ERROR; /* 判断 i 值是否合理,若不合理,返回 ERROR */
    *e = L.elem[i-1]; /* 数组中 i-1 的单元存储着线性表中第 i 个数据元素的内容 */
    return OK;
}
```

(7) 在线性表 L 中检索值为 e 的数据元素

```
int LocateElem(SeqList L,ElemType e)
```

```
{
    for (i=0;i<L.length;i++)
        if (L.elem[i] == e) return i+1;
    return 0;
}
```

(8) 在线性表 L 中第 i 个数据元素之前插入数据元素 e

```
int ListInsert(SeqList *L,int i,ElemType e)
```

```
{
    if (L->length == LIST_MAX_LENGTH) return ERROR; /* 检查是否有剩余空间 */
    if (i < 1 || i > L->length+1) return ERROR; /* 检查 i 值是否合理 */
    for (j=L->length-1;j>=i-1;j--)
        /* 将线性表第 i 个元素之后的所有元素向后移动 */
        L->elem[j+1] = L->elem[j];
    L->elem[i-1] = e; /* 将新元素的内容放入线性表的第 i 个位置 */
    L->length++;
    return OK;
}
```

(9) 将线性表 L 中第 i 个数据元素删除

```

int ListDelete(SeqList *L, int i, ElemType *e)
{
    if (IsEmpty(L)) return ERROR;          /* 检测线性表是否为空 */
    if (i < 1 || i > L->length) return ERROR; /* 检查 i 值是否合理 */
    *e = L->elem[i-1];                      /* 将欲删除的数据元素内容保留在 e 所指示的存储单元中 */
    for (j = i; j <= L->length-1; j++)
        /* 将线性表第 i+1 个元素之后的所有元素向前移动 */
        L->elem[j-1] = L->elem[j];
    L->length--;
    return OK;
}

```

2.1.3 线性表的链式存储及运算实现

线性表链式存储结构的特点如下。

上节中介绍的顺序存储是一种简单、方便的存储方式。它要求线性表的数据元素依次存放在连续的存储单元中,从而利用数据元素的存储顺序表示相应的逻辑顺序,可以随机存取,效率高,但顺序存储也有其缺点:比如在做插入或删除元素的操作时,会产生大量的数据元素移动;对于长度变化较大的线性表,需要一次性分配足够的存储空间,但这些空间常常又得不到充分的利用;若线性表的空间不足了,又难以扩充。

线性表的链式存储结构不要求逻辑上相邻的元素在物理存储单元上也相邻,其基本思想是利用一个指针来反映数据元素之间的逻辑关系,即给每个数据元素附加一个域,用来存放它的直接后继元素在内存中的地址。因此,其结点结构包括两个部分:



其中,结点的 data 部分称为数据域,用于存放结点数据信息;结点的 next 部分称为指针域,用于存放其后继结点的存储地址。

其结点的定义如下:

```

typedef struct node
{
    datatype data;          /* 结点的数据域 */
    struct node *next;      /* 结点的指针域 */
} Node, *LinkList;

```

1. 单链表

所有的结点可以通过指针链接成一个线性表,在这里称之为单链表。

线性表的链式存储结构是指用一组任意的存储单元(可以连续,也可以不连续)存储线性表中的数据元素。为了反映数据元素之间的逻辑关系,对于每个数据元素不仅要表示它的具体内容,还要附加一个表示它的直接后继元素存储位置的信息。假设有一个线性表(a,b,c,d),可用如图 2.2 所示的形式来存储。

存储地址	数据元素值	直接后继存储地址
100	b	120
...	...	...
120	c	160
...	...	...
144	a	100
...	...	...
160	d	NULL
...	...	...

图 2.2 单链表的存储形式

其比较直观的示意图如下所示：

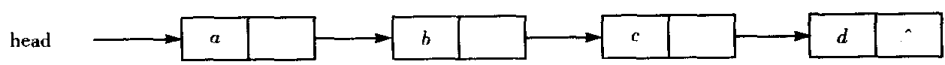


图 2.3 单链表

为了便于实现各种运算,通常在链表的第一个结点之前设置一个附加结点,称为头结点,下图即为带头结点的单链表:

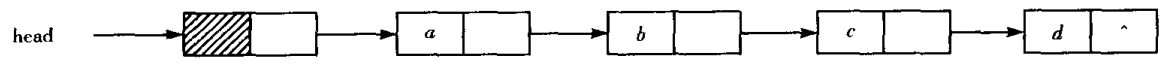


图 2.4 带头结点的单链表

在单链表中,已知某个结点的地址,可以方便地查询其后继结点,但若查询其前驱结点则比较困难。

2. 循环链表

若将链表中的最后一个结点的 next 域指向头结点,则构成循环链表。如图 2.5 所示。

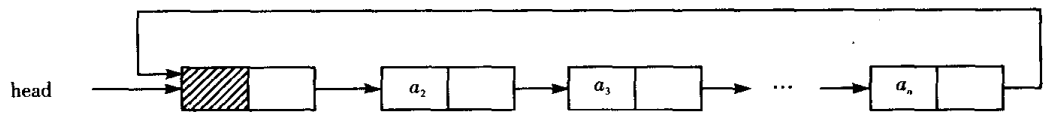


图 2.5 循环链表

实现循环链表的类型定义与单链表完全相同,它的所有操作也都与单链表类似。只是判断链表结束的条件有所不同。下面我们就列举两个循环链表操作的算法实现。

(1) 初始化循环链表 CL

```
int InitList(LinkList *CL)
{
    CL->head = (*LinkList)malloc(sizeof(LinkList));
    if (CL->head)
    {
        CL->head->next = CL->head;
        return OK;
    }
    /* 让 next 域指向它自身 */
    else return ERROR;
}
```

(2) 在循环链表 CL 中检索值为 e 的数据元素

```
LinkList *LocateElem(LinkList CL, ElemType e)
{
    ...
}
```

```

LinkedList *p;
for (p = CL.head->next; (p != CL.head) && (p->elem != e); p = p->next);
if (p != CL.head) return p;
else return NULL;
}

```

3. 双向链表

在单循环链表中,虽然从任一结点出发都可以找到其前驱结点,但是要循环一周,其时间复杂度为 $O(n)$ 。为了能使寻找前驱结点的时间复杂度减小到 $O(1)$,则只能付出空间的代价:每个结点再加一个指向其前驱结点的指针域,结点的结构如图 2.6 所示,由这种结点组成的链表称为双向链表。

双向链表结点的定义如下:

```

typedef struct node
{
    datatype data;
    struct node *prior, *next;
} DNode, *DLinkedList;

```

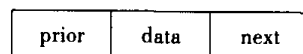
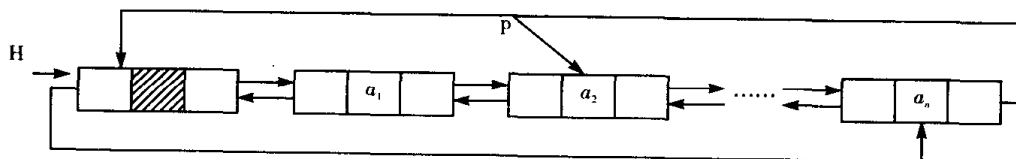


图 2.6 双向链表的结点结构

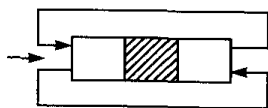
和单链表类似,双向链表通常也是用头指针标识,也可以带头结点和做成循环结构,图 2.7 是带头结点的双向循环链表示意图。显然通过某结点的指针 p 即可以直接得到它的后继结点的指针 $p \rightarrow next$ 和前驱结点的指针 $p \rightarrow prior$ 。这样如果操作中需要找前驱结点时,则不需再用循环。从下面的插入删除运算中可以看到这一点。

设 p 指向双向循环链表中的某一结点,则 $p \rightarrow prior \rightarrow next$ 表示的是 $*p$ 结点之前驱结点的后继结点的指针,即与 p 相等;类似地, $p \rightarrow next \rightarrow prior$ 表示的是 $*p$ 结点之后继结点的前驱结点的指针,也与 p 相等,所以有以下等式:

$$p \rightarrow prior \rightarrow next = p = p \rightarrow next \rightarrow prior$$



(a)非空表



(b)空表

图 2.7 带头结点的双循环链表

双向链表中结点的插入:设 p 指向双向链表中某结点, s 指向待插入的值为 x 的新结点,将 $*s$ 插入到 $*p$ 的前面,插入示意图如图 2.8 所示。

操作如下:

- ① $s \rightarrow prior = p \rightarrow prior;$
- ② $p \rightarrow prior \rightarrow next = s;$

③ $s \rightarrow next = p;$

④ $p \rightarrow prior = s;$

指针操作的顺序不是惟一的,但也不是任意的,操作①必须要放到操作④的前面完成,否则 $*p$ 的前驱结点的指针就丢掉了。

双向链表中结点的删除:

设 p 指向双向链表中某结点,删除 $*p$ 。

操作示意图如图 2.9 所示。

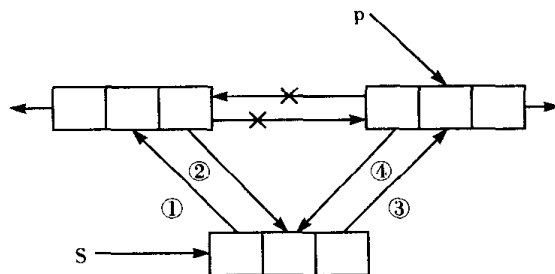


图 2.8 双向链表中结点的插入

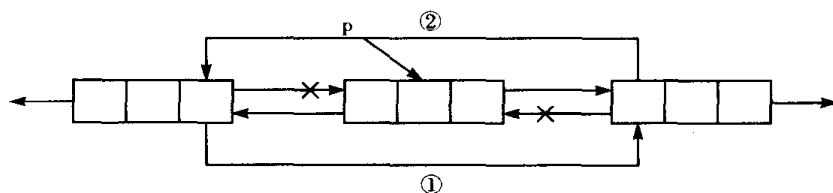


图 2.9 双向链表中结点的删除

操作如下:

① $p \rightarrow prior \rightarrow next = p \rightarrow next;$

② $p \rightarrow next \rightarrow prior = p \rightarrow prior;$

$free(p);$

2.1.4 顺序表和链表的比较

前面介绍了线性表的逻辑结构及它的两种存储结构:顺序表和链表。通过对它们的讨论可知它们各有优缺点,顺序存储有三个优点:

- (1) 方法简单,各种高级语言中都有数组,容易实现。
- (2) 不需要为表示结点间的逻辑关系而增加额外的存储空间。
- (3) 顺序表具有按元素序号随机访问的特点。

但它也有两个缺点:

(1) 在顺序表中做插入删除操作时,平均移动大约表中一半的元素,因此对长度较大的顺序表效率低。

(2) 需要预先分配足够大的存储空间,估计过大,可能会导致顺序表后部大量闲置;预先分配过小,又会造成溢出。

链表的优缺点恰好与顺序表相反。在实际问题中选取存储结构时通常有以下几点考虑:

(1) 基于存储空间规模的考虑

顺序表的存储空间是静态分配的,在程序执行之前必须明确规定它的存储规模,也就是说事先对“MAXSIZE”要有合适的设定,过大造成浪费,过小造成溢出。可见对线性表的长度或存储规模难以估计时,不宜采用顺序表;链表不用事先估计存储规模,但链表的存储密度较低,存储密度是指一个结点中数据元素所占的存储单元和整个结点所占的存储单元之比。例如:一个结点的数据域为 int 类型,需占 $2B$,系统中的地址若为 32 位的,

则结点的指针域需占 4B, 可得出此结点的存储密度 $= 2B / (2B + 4B) = 1/3$, 显然链式存储结构的存储密度是小于 1 的, 因此会增加存储空间的开销。

(2) 基于时间复杂度的考虑

在顺序表中按序号访问一个元素的时间复杂度为 $O(1)$, 而链表中按序号访问一个结点的时间复杂度为 $O(n)$, 所以如果经常做的运算是按序号访问数据元素, 显然顺序表优于链表; 而在顺序表中做插入、删除时平均移动表中一半的元素, 当数据元素的信息量较大且表较长时, 这一点是不应忽视的; 在链表中作插入、删除操作, 虽然也要找插入位置, 但主要是比较操作, 从这个角度考虑显然后者优于前者。

(3) 基于环境的考虑

顺序表容易实现, 任何高级语言中都有数组类型, 而链表的操作是基于指针的, 相对来讲前者简单些, 也是用户考虑的一个因素。

总之, 两种存储结构各有优缺点, 选择哪一种应由实际问题中的主要因素决定。通常大小基本固定、只对其做查询和存取操作的线性表宜选择顺序存储, 而频繁做插入删除操作即动态性较强的线性表宜选择链式存储。

2.2 实训篇

实训目的:

- 理解线性表结构的实质
- 熟练掌握对单链表、循环链表的创建、查询、插入、删除等基本操作

2.2.1 超市密码存储箱系统的设计与实现

案例描述

在大中型超市门口一般都放置有很多存放包的箱子, 顾客可以将不能带入超市的物品暂存在里面, 购物结束后再取回。顾客使用箱子的整个流程为“投一元硬币”——“找到一个空箱子, 同时产生密码”(系统完成)——“打印密码, 打开箱子”(系统完成)——“取密码纸存包, 并关闭箱子, 入超市购物”——“购物结束”——“输入密码”——“找到对应箱子并打开”(系统完成)——“取包”。

现要求设计程序模拟以上系统完成的功能。

- 界面: 在我们的模拟系统中, 箱子在屏幕上被画出来, 并编号, 空箱为蓝色, 被使用时变为红色, 再变为空后则恢复蓝色;
- 通过按“1”键模拟顾客投币;
- 当空箱子被顾客申请得到的同时, 系统自动生成 6 位密码, 此密码不能与正在被使用的任何一个箱子的密码相同。

设计分析

在设计时, 可利用链表来组织所有的箱子, 所有的箱子以结点的形式表示, 结点中存放箱号、密码(满箱有, 空箱无)以及指向下一个结点的指针。空箱结点放在一个链表 1 中, 满箱结点放在另一个链表 2 中。

若有顾客投币(这里用按下“1”键模拟),查看链表1是否为空,若为空,则显示“箱满,请稍候!”;若非空,则取出一个结点,随机产生一个六位的数字密码,并将此密码和链表2中所有结点的密码相比较,若有重复,则再随机产生一个新密码,直到无重复;将密码信息写入此结点,并将其插入链表2;将此箱的颜色改为红色。

若有用户输入6位数字密码,若此时链表2为空,则不予处理;链表2不空,则将此密码和链表2中结点的密码依次比较,直到找到一个相同的结点,将此结点取出插入链表1,并将其对应的箱子改为蓝色。

整个流程如图 2.10 所示:

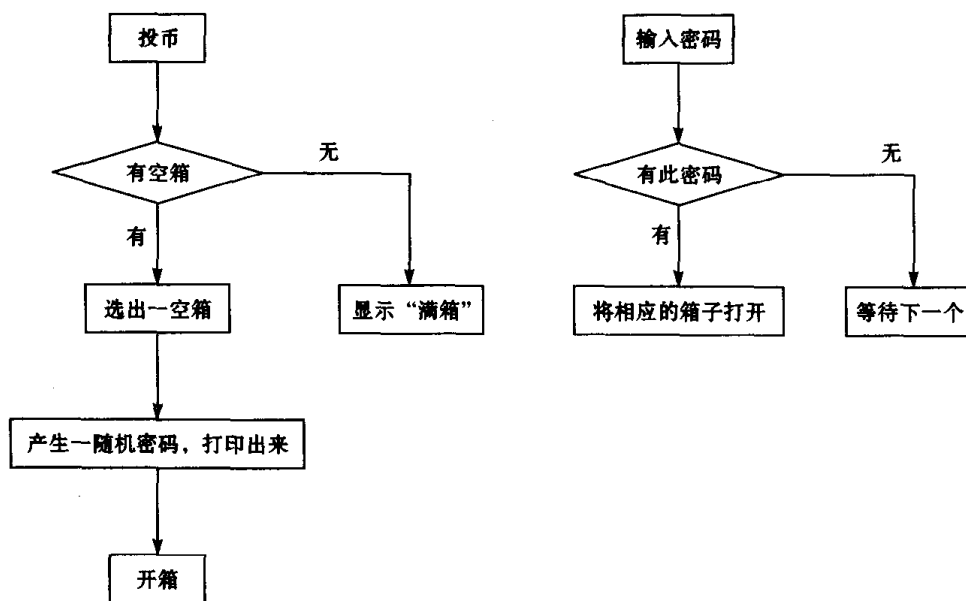


图 2.10 超市密码存储箱系统的流程

设计方案

(1) 存储结构类型定义

/* 密码箱的存储结构类型定义 */

```
typedef struct node {
    int num; /* 箱子的号码 */
    long int password; /* 箱子的密码(满箱有,空箱无) */
    struct node * next; /* 指向下个结点的指针 */
} Node, * LinkList;
```

(2) 主要功能函数设计

①建立链表:最初所有的箱子为空,按照序号依次插入链表 L1 中,链表 L2 为空。

LinkList CreateList1() /* 建立空箱子链表 L1,初始时共 12 个结点 */

```
{
    int i;
    LinkList head, p, q;
    head = (Node *) malloc(sizeof(Node)); /* 创建头结点 */
    head->next = NULL;
    p = head;
```

20 / 数据结构实训与习题解析 □

```

for(i=1;i<=12;i++)                                /* 依次建立 12 个结点,并插入链表中 */
{
    q=(Node *)malloc(sizeof(Node));
    q->num=i;                                         /* 每个结点的编号 */
    q->next=p->next;                                 /* 将结点插入链表 */
    p->next=q;
    p=q;
}
return head;                                         /* 返回链表的头结点 */
}

LinkList CreateList2()                               /* 建立满箱子链表 L2,初始时空 */
{
    LinkList head;
    head=(Node *)malloc(sizeof(Node));             /* 创建头结点 */
    head->next=NULL;
    return head;                                     /* 返回链表的头结点 */
}

```

②寻找空箱子:判断链表 L1 是否为空,若不空,则取出第一个结点(即将其从链表 L1 中删除),然后随机产生一个 6 位密码,并将此密码依次与链表 L2 中所有结点的密码相比较,若发现有重复,则重新生成一个密码,直到没有重复为止。将密码写入此箱子结点的 password 域,然后将此结点插入 L2 链表。

```

long int MiMa()                                       /* 密码产生函数 */
{
    long n,m;
    randomize();
    m=random(9)+1;
    n=(m*100000+random(1000000))%1000000;          /* 产生 6 位数的密码 */
    return(n);
}

int compare(LinkList head,long int x)
/* 比较是否有重复的密码,若有返回 0,无则返回 1 */
{
    Node *q;
    for(q=head;q!=NULL;q=q->next)                  /* 在链表中搜索 */
    {
        if(q->password==x)
            return 0;
    }
    return 1;
}

```

```

}

LinkedList DelNode(LinkedList head)          /* 在链表中删除第一个结点 */
{
    Node *p;
    p = head->next;
    head->next = p->next;
    return p;
}

void InsertNode(LinkedList head, Node *p)     /* 在链表头结点后插入一个结点 */
{
    p->next = head->next;
    head->next = p;
}

```

③取包:输入密码,在链表 L2 中依次查找是否有与顾客输入密码相符的结点,找到后,取出该结点,将其从链表 L2 中删除,然后将其插入链表 L1 中。

```

LinkedList pcompare(LinkedList head, long int x)
/* 开箱前查看是否有此箱,若有将它从 L2 中删除,返回指向它的指针 */
{
    Node *p, *q;
    for(p = head, q = head->next; q != NULL; q = q->next)
    {
        if(q->password == x)
        {
            p->next = q->next;
            return q;
        }
        else p = q;
    }
    return NULL;
}
/* 若无此箱则返回空指针 */

```

(3) 实现程序

```

#include "graphics.h"
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "time.h"
typedef struct node
{
    int num;

```

22 / 数据结构实训与习题解析 □

```

    long int password;
    struct node *next;
}Node, *LinkList;

```

```

void DrawBox()                                /* 在屏幕上画出箱子,共 12 个,3 行 4 列,箱体为蓝色 */

```

```

{
    int x,y,i=1;
    char st[3];
    setcolor(3);
    for(x=300;x<=500;x+=50)
        line(x,50,x,200);
    for(y=50;y<=200;y+=50)
        line(300,y,500,y);
    setfillstyle(SOLID_FILL,1);
    for(y=50;y<=150;y+=50)
        for(x=300;x<=450;x+=50)
        {
            itoa(i,st,10);
            floodfill(x+30,y+30,3);
            outtextxy(x+20,y+25,st);
            i=atoi(st);
            i++;
        }
}

```

/* 在每个箱子上标上箱子的编号:1~12 */

```

int dm(int x)                                /* 辅助计算填充坐标的参数 */

```

```

{
    int m;
    m=x%4;
    if(m==0) m=4;
    return m;
}

```

```

int dn(int x)                                /* 辅助计算填充坐标的参数 */

```

```

{
    int n;
    if(x<=4) n=1;
    else if(x<=8) n=2;
    else if(x<=12) n=3;
    return n;
}

```

```

void fillcolor(int x,int y,int color)      /* 填充颜色函数 */
{
    setfillstyle(SOLID_FILL,color);
    floodfill(x,y,3);
}

main()
{
    int i;
    char k;
    int m,n;
    long int t;
    char string[8];
    LinkList head1,head2,p;
    int graphdriver = DETECT,graphmode;
    initgraph(&graphdriver,&graphmode,"d:\\tc");
    DrawBox();
    head1 = CreateList1();
    head2 = CreateList2();
    printf("1—存包  2—取包  0—退出\n");
    while(1)
    {
        k=getch();
        if(k == '0') exit(0);
        if(k == '1')
        {
            if(head1->next == NULL)
                printf("满箱!\n");
            else
            {
                p = DelNode(head1);          /* 将其从链表 L1 中删除 */
                p->password = MiMa();          /* 生成密码 */
                while(! compare(head2,p->password)) /* 若密码重复,则重新生成 */
                    p->password = MiMa();
                printf("%ld\n",p->password);
                InsertNode(head2,p);          /* 将其插入链表 L2 中 */
                m = dm(p->num);
                n = dn(p->num);
                fillcolor(300 + m * 50 - 10,50 + n * 50 - 10,4); /* 将此箱变为红色 */
            }
        }
    }
}

```

```

if(k == '2') * 将此箱变为蓝色 *
{
    printf("请输入密码:");
    scanf("%ld",&t);
    if(head2 -> next != NULL)
    {
        p = pcompare(head2,t);
        if(p == NULL) continue;
        InsertNode(head1,p);
        m = dm(p -> num);
        n = dn(p -> num);
        fillcolor(300 + m * 50 - 10, 50 + n * 50 - 10, 1); /* 将此箱变为蓝色 */
    }
}
}

```

(4) 运行测试

编译运行后显示:

1—存包 2—取包 0—退出

输入:1

201541

1 号箱子由蓝色变为红色

1	2	3	4
5	6	7	8
9	10	11	12

输入:1

729996

2 号箱子由蓝色变为红色

1	2	3	4
5	6	7	8
9	10	11	12

输入:1

824604

3 号箱子由蓝色变为红色

1	2	3	4
5	6	7	8
9	10	11	12

1	2	3	4
5	6	7	8
9	10	11	12

输入:1

921369

4 号箱子由蓝色变为红色

1	2	3	4
5	6	7	8
9	10	11	12

输入:1

319428

5 号箱子由蓝色变为红色

1	2	3	4
5	6	7	8
9	10	11	12

输入:2

请输入密码:729996(回车)

2 号箱子由红色变为蓝色

1	2	3	4
5	6	7	8
9	10	11	12

输入:2

请输入密码:921369(回车)

4 号箱子由红色变为蓝色

1	2	3	4
5	6	7	8
9	10	11	12

输入:0

退出程序

1	2	3	4
5	6	7	8
9	10	11	12

2.2.2 员工通讯录管理系统

案例描述

为某个单位建立一个员工通讯录管理系统,可以方便地查询每一个员工的办公室电话号码、手机号码及电子邮箱。其功能包括通讯录链表的建立、员工通讯信息的查询、修改、插入与删除以及整个通讯录表的输出。

设计分析

在本设计中,整个通讯录可以采用顺序表或链表方式存储。采用前者,可以提高查询速度;采用后者,可以提高插入与删除记录的效率。在本系统中对查询速度要求不高,所以采用链表存储结构。

设计方案

(1) 存储结构类型定义

/* 员工通讯信息的结构类型定义 */

```

typedef struct {
    char num[5];           /* 员工编号 */
    char name[8];          /* 员工姓名 */
    char phone[9];         /* 办公室电话号码 */
    char call[12];         /* 手机号码 */
} DataType;
/* 通讯录链表的结点类型 */
typedef struct node

```

```

{
    DataType data;         /* 结点的数据域 */
    struct node *next;     /* 结点的指针域 */
} ListNode, *LinkList;

```

(2) 主要功能函数设计

① 建立通讯录链表

可以采用手工输入的方法,逐个输入每个员工的信息,每输入一个自动创建一个结点,并插入链表,直到所有的员工信息输入完毕,通讯录链表也就建立好了。

```

LinkList CreatList()      /* 建立一个通讯录链表 */
{
    LinkList head,s,p,q;
    char ch;
    head=(ListNode *)malloc(sizeof(ListNode)); /* 创建头结点 */
    head->next = NULL;
    p = head;
    do
    {
        s=(ListNode *)malloc(sizeof(ListNode)); /* 创建一个记录结点 */
        printf("请输入姓名:");
        scanf("%s",s->data.name);
        printf("请输入编号:");
        scanf("%s",s->data.num);
        printf("请输入办公室电话号码:");
        scanf("%s",s->data.phone);
        printf("请输入手机号码:");
        scanf("%s",s->data.call);
        s->next = p->next;
        p->next = s;
        printf("继续输入下一记录吗(y/n)?");
        scanf("\n%c",&ch);
    } while(ch == 'y');
    return head;          /* 返回头结点指针 */
}

```

②查询功能:可以按照编号或姓名在链表中查找出对应的记录。

```

ListNode * SearchList(LinkList head)      /* 查询函数 */
{
    ListNode * p;
    int x;
    char num[5];
    char name[8];
    printf("1.按编号查询 2.按姓名查询\n");
    printf("请选择:");
    scanf("%d",&x);
    if(x == 1)
    {
        printf("请输入待查人的编号:");
        scanf("%s",num);
        for(p = head;p != NULL&&strcmp(p->data.num,num) != 0;p = p->next);
    }
    else if(x == 2)
    {
        printf("请输入待查人的姓名:");
        scanf("%s",name);
        for(p = head;p != NULL&&strcmp(p->data.name,name) != 0;p = p->next);
    }
    return p;                          /* 返回指向查找到的记录的指针,若没查找到,返回的是空指针 */
}

```

③插入新的记录:将新记录结点插入到头结点之后,成为链表中的第一个记录结点。也可将新记录插入在记录链表的尾部,成为最后一条记录(读者可自己编写实现算法)。

```

void InsertList(LinkList head,ListNode * p)
{
    p->next = head->next;
    head->next = p;
}

```

④删除记录:可以按照编号或姓名在链表中顺序查找出对应的记录,然后将其删除。本函数调用查询函数。

```

void DelNode(LinkList head)
{
    LinkList p,q;
    char ch;
    p = SearchList(head);
    if(!p)
    {
        printf("没有找到此人的记录\n");
    }
}

```

```

    return;
}
printf("确定要删除吗? (y/n)");
scanf("\n% c",&ch);
if(ch == 'n') return;
for(q = head; q != NULL && q->next != p; q = q->next);
q->next = p->next;
free(p);
printf("删除成功\n");
}

```

⑤ 查看全部记录: 从第一个结点到最后一个结点依次输出所有记录。

```

void PrintList(LinkList head)
{
    LinkList p;
    p = head->next;
    while(p != NULL)
    {
        printf("编号      姓名      办公室电话号码      手机号码\n");
        printf("%s      %s      %s      %s\n",
            p->data.num, p->data.name, p->data.phone, p->data.call);
        p = p->next;
    }
}

```

(3) 实现程序

```

#include "stdio.h"
#include "string.h"
#include "stdlib.h"
/* 员工通讯信息的结构类型定义 */
typedef struct
{
    char num[5];                /* 员工编号 */
    char name[8];               /* 员工姓名 */
    char phone[9];              /* 办公室电话号码 */
    char call[12];              /* 手机号码 */
} DataType;
/* 通讯录单链表的结点类型 */
typedef struct node
{
    DataType data;              /* 结点的数据域 */
    struct node *next;         /* 结点的指针域 */
} ListNode, *LinkList;

```

```

void Menu()                                /* 显示菜单函数 */ /* 插入以上各功能函数 */
{
    printf("      * * * * * 某软件公司员工通讯录 * * * * * \n");
    printf("1 - 建立通讯录 2 - 查看全部记录 3 - 查询 4 - 插入新员工记录 5 - 删除离开员  
      工记录 0 - 退出\n");
    printf("请选择(输入0-5)\n");
}
main()
{
    int t;
    LinkList head,p;
    while(1)
    {
        Menu();
        scanf("%d",&t);                    /* 选择系统子功能 */
        switch(t)
        {
            case 1:
                printf("建立通讯录\n");
                head = CreatList();
                break;
            case 2:
                printf("查看全部记录\n");
                PrintList(head);
                break;
            case 3:
                printf("查询\n");
                p = SearchList(head);
                if(p)
                {
                    printf("编号      姓名      办公室电话号码      手机号码\n");
                    printf("%s      %s      %s      %s\n",
                        p->data.num,p->data.name,p->data.phone,p->data.call);
                }
                else printf("无此人\n");
                break;
            case 4:
                printf("插入新员工记录\n");
                p = (ListNode *)malloc(sizeof(ListNode));
                printf("请输入姓名:");
                scanf("%s",p->data.name);
                printf("请输入编号:");

```

```

scanf("%s",p->data.num);
printf("请输入办公室电话号码:");
scanf("%s",p->data.phone);
printf("请输入手机号码:");
scanf("%s",p->data.call);
InsertList(head,p);
break;
case 5:
printf("删除离开员工记录\n");
DelNode(head);
break;
case 0:
printf("退出\n");
return;
}
}
}

```

(4) 运行测试

编译运行后显示结果为:

*****某软件公司员工通讯录*****

1 - 建立通讯录 2 - 查看全部记录 3 - 查询 4 - 插入新员工记录 5 - 删除离开员工记录 0 - 退出

请选择(输入0-5)

输入:1(回车)

建立通讯录

请输入姓名:刘丽

请输入编号:001

请输入办公室电话号码:2367589

请输入手机号码:13305558668

继续输入下一记录吗(y/n)? y

请输入姓名:王平安

请输入编号:002

请输入办公室电话号码:2365327

请输入手机号码:13305559078

继续输入下一记录吗(y/n)? y

请输入姓名:苏可可

请输入编号:003

请输入办公室电话号码:2364266

请输入手机号码:13305551025

继续输入下一记录吗(y/n)? n

*****某软件公司员工通讯录*****

1 - 建立通讯录 2 - 查看全部记录 3 - 查询 4 - 插入新员工记录 5 - 删除离开员工记录 0 - 退出

请选择(输入0-5)

输入:2(回车)

查看全部记录

编号	姓名	办公室电话号码	手机号码
003	苏可可	2364266	13305551025
002	王平安	2365327	13305559078
001	刘丽	2367589	13305558668

*****某软件公司员工通讯录*****

1 - 建立通讯录 2 - 查看全部记录 3 - 查询 4 - 插入新员工记录 5 - 删除离开员工记录 0 - 退出

请选择(输入0-5)

输入:3(回车)

查询

1.按编号查询 2.按姓名查询

请选择:1(回车)

请输入待查人的编号:002(回车)

编号	姓名	办公室电话号码	手机号码
002	王平安	2365327	13305559078

*****某软件公司员工通讯录*****

1 - 建立通讯录 2 - 查看全部记录 3 - 查询 4 - 插入新员工记录 5 - 删除离开员工记录 0 - 退出

请选择(输入0-5)

输入:3(回车)

查询

1.按编号查询 2.按姓名查询

请选择:2(回车)

请输入待查人的姓名:刘丽(回车)

编号	姓名	办公室电话号码	手机号码
001	刘丽	2367589	13305558668

*****某软件公司员工通讯录*****

1 - 建立通讯录 2 - 查看全部记录 3 - 查询 4 - 插入新员工记录 5 - 删除离开员工记录 0 - 退出

请选择(输入0-5)

输入:4(回车)

插入新员工记录

请输入姓名:李华

请输入编号:004

请输入电话号码:2364063

请输入手机号码:13085552226

***** 某软件公司员工通讯录 *****

1 - 建立通讯录 2 - 查看全部记录 3 - 查询 4 - 插入新员工记录 5 - 删除离开员工记录 0 - 退出

请选择(输入0-5)

输入:5(回车)

删除离开员工记录

1.按编号查询 2.按姓名查询

请选择:1(回车)

请输入待查人的编号:001(回车)

确定要删除吗?(y/n)y

删除成功

***** 某软件公司员工通讯录 *****

1 - 建立通讯录 2 - 查看全部记录 3 - 查询 4 - 插入新员工记录 5 - 删除离开员工记录 0 - 退出

请选择(输入0-5)

输入:0(回车)

退出

2.2.3 约瑟夫环问题

案例描述

约瑟夫环问题:设编号为 $1, 2, 3, \dots, n$ 的 $n(n > 0)$ 个人按顺时针方向围坐一圈,每个人持有一个正整数密码。开始时任选一个正整数做为报数上限 m ,从第一个人开始顺时针方向自0起顺序报数,报到 m 时停止报数,报 m 的人出列,将他的密码作为新的 m 值,从他的下一个人开始重新从0报数。如此下去,直到所有人全部出列为止。令 n 最大值取30。要求设计一个程序模拟此过程,求出出列编号序列。

设计分析

在这里使用单循环链表解决这一问题,然后从第1个结点开始数,数到本次的报数上限 m ,取出它的密码,做为下次的报数上限。然后将此结点从链表中删除,并释放其空间。接下来以下一个结点做为起始点开始数,如此循环,直到链表只剩下一个结点,将其删除,程序结束。

设计方案

(1) 存储结构类型定义

链表结点的存储类型定义:

```
typedef struct node
{
    int number;                /* 人的序号 */
    int password;              /* 密码 */
    struct node *next;         /* 指向下一个结点的指针 */
} Node;
```

(2) 主要功能函数设计

```

struct node *CreatList(int num)          /* 建立循环链表 */
{
    int i;
    struct node *p, *head;
    if((p=(struct node *)malloc(sizeof(struct node))) == NULL)
        exit(0);                          /* 若创建结点失败,则退出 */
    head=p;
    p->next=head;
    for(i=1;i<num;i++)
    {
        if((p->next=(struct node *)malloc(sizeof(struct node))) == NULL)
            exit(0);                      /* 若创建结点失败,则退出 */
        p=p->next;
        p->next=head;
    }
    return head;                          /* 返回头指针 */
}

```

(3) 程序实现

```

#include <stdlib.h>
#include <alloc.h>
typedef struct node
{
    int number;                          /* 人的序号 */
    int password;                        /* 密码 */
    struct node *next;                  /* 指向下一个结点的指针 */
}Node;

main()
{
    int count,i,n=30,m,k=1;              /* 人数 n 为 30 个 */
    struct node *head,*p,*q;
    randomize();                          /* 初始化随机发生器 */
    head=CreatList(n);                   /* 建立有 n 个结点的循环链表 */
    q=head;
    for(i=1;i<=30;i++)                  /* 给每个结点的各个域赋值 */
    {
        head->number=i;                  /* 按顺序给每个人编号 */
        head->password=rand()%10+1;
        /* 每个人的密码随机产生,并将密码值控制在 1-10 */
        head=head->next;
    }
}

```

```

m = rand() % 10 + 1;          /* 第一次的报数上限取随机数,并将其值控制在 1 - 10 */
printf("第一个报数上限为 %2d", m);
count = 0;                    /* 从 0 开始报数 */
while(head -> next != head)    /* 当剩下最后一个人时,退出循环 */
{
    if(count == m)             /* 当前人报的数等于此次报数上限,让其离开 */
    {
        p = head -> next;      /* p 指向此次将离开人的下一个人的位置 */

        printf("%2d 号人离开,报数 %2d——>", head -> number, head -> password);
        if(k % 2 == 0) printf("\n"); /* 控制每行输出两个人 */
        m = head -> password;      /* 下次的报数上限等于此次离开人的密码 */
        q -> next = p;             /* 让 head 从链表中脱节,将前后两个结点连接起来 */
        free(head);                /* 释放 head 指向的结点空间 */
        head = p;                  /* head 指向下个结点,即下次报数的第一个人 */
        count = 0;                 /* 将 count 重新置为 0,下次再从 0 开始数 */
        k++;
    }
    else
    {
        q = head;                  /* q 用于记录将要离开的人的前一个人的位置,用于删除操作 */
        head = head -> next;
        count++;
    }
}
printf("%2d 号人离开\n", p -> number);
free(head);                      /* 让最后一个人也出列 */
}

```

(4) 运行测试

编译运行后显示结果为:

第一个报数上限为 6

```

7 号人离开,报数 6——> 14 号人离开,报数 6——>
21 号人离开,报数 5——> 27 号人离开,报数 3——>
1 号人离开,报数 9——> 12 号人离开,报数 3——>
17 号人离开,报数 4——> 23 号人离开,报数 5——>
30 号人离开,报数 8——> 11 号人离开,报数 9——>
26 号人离开,报数 8——> 9 号人离开,报数 9——>
25 号人离开,报数 7——> 8 号人离开,报数 4——>
18 号人离开,报数 3——> 24 号人离开,报数 1——>
29 号人离开,报数 2——> 4 号人离开,报数 8——>
22 号人离开,报数 10——> 20 号人离开,报数 2——>

```

3号人离开,报数2——>10号人离开,报数4——>
 28号人离开,报数8——>5号人离开,报数6——>
 6号人离开,报数9——>2号人离开,报数2——>
 16号人离开,报数8——>15号人离开,报数3——>
 13号人离开,报数2——>19号人离开

注意:因为每个报数上限都是随机产生的,所以每次运行的结果都不一样。

2.2.4 模拟蜗牛爬树

案例描述

一棵大树高为 h 米,有一个蜗牛要从树底爬到树顶,它每爬 n 米后又向下滑 m 米(当爬到树顶时不再下滑),在树上每隔一米都有一个标志,现设计一程序模拟整个过程,并统计在整个爬树过程中,蜗牛经过每个标志各多少次。

设计分析

在本设计中,用一个双向链表来模拟这棵大树,大树的每个标志作为双向链表中的一个结点,将整个爬行过程模拟成指针在这个双向链表中的移动,当移动到最后一个结点时,即爬到树顶,操作完成。因为蜗牛在爬行过程中,既要向上爬,又会向下滑,而双向链表的特征是既可正序访问,也可逆序访问,正好符合设计要求。每个结点需要设置一个记数位,每当结点被访问一次,其记数位的值自动加1,用于记录经过每个结点的次数。

为了能形象地表示整个爬行过程,我们采用图形模拟,每个标尺由一个绿色方格表示(如树高为12米,则画出12个绿色方格),用一个黄色的圆代表蜗牛。

设计方案

(1) 存储结构类型定义

/* 双向链表结点的类型定义 */

```
typedef struct dnode
{
    int data;                /* 标尺数据 */
    int count;               /* 经过次数记数位 */
    struct dnode *prior;     /* 指向前驱结点的指针 */
    struct dnode *next;      /* 指向后继结点的指针 */
} DNode, *DLinkedList;
```

(2) 主要功能函数设计

① 建立双向链表

```
DLinkedList CreatList()      /* 初始化双向链表 */
{
    DLinkedList L;
    L = (DNode *)malloc(sizeof(DNode)); /* 创建头结点 */
    L->prior = L->next = NULL;
    return L;                  /* 返回头指针 */
}
```

②在双向链表中插入新的结点

```

void ListInsert(DLinkList L,int e)    /* 在双向链表尾部插入新的结点 */
{
    DLinkList p,s;
    for(p=L;p->next!=NULL;p=p->next);
    s=(DNode *)malloc(sizeof(DNode));/* 创建新的结点 */
    s->data=e;                        /* 为新结点的各个域赋值 */
    s->count=0;
    s->next=p->next;                  /* 修改各指针,完成插入操作 */
    s->prior=p;
    p->next=s;
}

```

③在双向链表中从当前位置开始向后访问 n 个结点(模拟蜗牛向上爬 n 米)

```

DLinkList goup(DLinkList p,int n)
{
    int i;
    for(i=1;i<=n;i++)
    {
        p=p->next;
        p->count++;                  /* 前进时,每经过一个结点,其记数位加1 */
    }
    return p;                        /* 返回当前结点地址 */
}

```

④在双向链表中从当前位置开始向前访问 m 个结点(模拟蜗牛向下滑 m 米)

```

DLinkList godown(DLinkList p,int m)
{
    int i;
    for(i=1;i<=m;i++)
    {
        p=p->prior;
        p->count++;                  /* 后退时,每经过一个结点,其记数位加1 */
    }
    return p;                        /* 返回当前结点地址 */
}

```

(3)程序实现

```

#include "stdio.h"
#include "malloc.h"
#include "stdlib.h"
#include "conio.h"
#include "dos.h"
#include "graphics.h"

```

```
/* 双向链表结点的类型定义 */
```

```
typedef struct dnode
```

```
{
    int data;                /* 标尺数据 */
    int count;               /* 经过次数 */
    struct dnode *prior;     /* 指向前驱结点的指针 */
    struct dnode *next;      /* 指向后继结点的指针 */
} DNode, *DLinkedList;
```

```
void DispList(DLinkedList L)    /* 依次输出链表中各结点的值 */
```

```
{
    DLinkedList p = L->next;
    while(p != NULL)
    {
        printf("经过 %d 米标尺 %d 次——>", p->data, p->count);
        p = p->next;
    }
    printf("结束");
}
```

```
void Draw(int n)    /* 根据 n 的值画大树标尺, 每个绿色方块代表一个标尺, 共画 n 个 */
```

```
{
    int k;
    for(k = 1; k <= n; k++)
    {
        setfillstyle(SOLID_FILL, GREEN);
        setcolor(RED);
        rectangle(k * 20, 100, (k + 1) * 20, 120);
        floodfill(k * 20 + 10, 100 + 10, RED);
    }
}
```

```
void Cir1(int x, int y, int color)    /* 画圆, 用黄色的圆代表蜗牛 */
```

```
{
    setfillstyle(SOLID_FILL, color);
    setcolor(color);
    circle(x, y, 5);
    floodfill(x, y, color);
}
```

```
int Cir1N(int x, int y, int n)
```

```
/* 黄色的圆在标尺上向后移动 n 格, 用来模拟蜗牛向上爬 n 米 */
```

```
{
```

```

    int i;
    for(i=0;i<=n;i++)
    {
        Cirl(x,y,YELLOW);
        if(i==n) break;
        delay(20000);
        Cirl(x,y,GREEN);
        x+=20;
    }
    printf("%d",x);
    return x;
}
/* 返回当前圆的位置 */

int CirlM(int x,int y,int m)
/* 黄色的圆在标尺上向前移动 m 格,用来模拟蜗牛向下滑 m 米 */
{
    int i;
    for(i=1;i<=m;i++)
    {
        Cirl(x,y,GREEN);
        x-=20;
        Cirl(x,y,YELLOW);
        delay(20000);
    }
    return x;
}
/* 返回当前圆的位置 */

main()
{
    int i,h,n,m;
    DLinkedList head,q;
    int x=10,y=110;
    int gdriver=DETECT,gmode;
    initgraph(&gdriver,&gmode,"d:/tc");
    printf("请输入大树的高度: h=");
    scanf("%d",&h);
    printf("*** 请输入爬行速度 *** \n");
    printf("每次爬行的高度: n=");
    scanf("%d",&n);
    printf("每次下滑的高度: m=");
    scanf("%d",&m);
    if(n<=m)

```

/* 小圆的初始位置坐标 */

/* 初始化图形系统 */

/* 若 $n \leq m$, 则退出程序 */

```

printf("爬行速度不合理,永远爬不到树顶\n");
exit(0);
}
Draw(h); /* 画出树图 */
head = CreatList();
/* 调用插入函数,依次插入结点,直到所有结点都插入完成,链表建立完毕 */
for(i=1;i<=h;i++)
    ListInsert(head,i);
q = head;
while(q->next!=NULL)
{
    q = goup(q,n); /* 在双向链表中从当前位置开始向后访问 n 个结点 */
    x = CirlN(x,y,n); /* 黄色小圆向后移动 n 格 */
    getch();
    if(q->next == NULL) /* 若已访问到最后一个结点,则结束 */
        break;
    q = godown(q,m); /* 在双向链表中从当前位置开始向前访问 m 个结点 */
    x = CirlM(x,y,m); /* 黄色小圆向前移动 m 格 */
    getch();
}
for(i=0;i<=5;i++) printf("\n");
DispList(head);
}

```

(4) 运行测试

输入数据测试 1:

请输入大树的高度:h=12(12 为用户输入的数据)回车

*** 请输入爬行速度 ***

每次爬行的高度:n=5(5 为用户输入的数据)回车

每次下滑的高度:m=3(3 为用户输入的数据)回车

显示的图形:



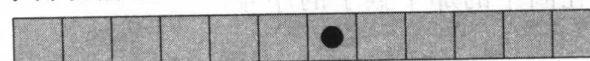
小球自动前进 5 格,如下图:



按下任意键后,小球自动后退 3 格,如下图:



按下任意键后,小球自动前进 5 格,如下图:



按下任意键后,小球自动后退 3 格,如下图:



按下任意键后,小球自动前进 5 格,如下图:



按下任意键后,小球自动后退 3 格,如下图:



按下任意键后,小球自动前进 5 格,如下图:



按下任意键后,小球自动后退 3 格,如下图:



按下任意键后,小球自动前进 5 格,如下图:



(此时移动结束)

显示:

经过 1 米标尺 1 次——> 经过 2 米标尺 2 次——> 经过 3 米标尺 3 次——> 经过 4 米标尺 4 次——> 经过 5 米标尺 4 次——> 经过 6 米标尺 4 次——> 经过 7 米标尺 4 次——> 经过 8 米标尺 4 次——> 经过 9 米标尺 4 次——> 经过 10 米标尺 3 次——> 经过 11 米标尺 2 次——> 经过 12 米标尺 1 次——> 结束

输入数据测试 2:

请输入大树的高度: $h = 16$ (16 为用户输入的数据) 回车;

*** 请输入爬行的速度 ***

每次爬行的高度: $n = 3$ (3 为用户输入的数据) 回车;

每次下滑的高度: $m = 5$ (5 为用户输入的数据) 回车;

爬行速度不合理,永远爬不到树顶

2.3 实训项目

项目内容

猴子选大王问题

问题为: 设编号为 $1, 2, \dots, n$ 的 n 个猴子围坐一圈, 约定从编号为 k ($1 \leq k \leq n$) 的猴子开始报数, 数到 m 的那个猴子出列, 它的下一位又从 1 开始报数, 数到 m 的那个猴子又出列, 依次类推, 直到只剩下一个猴子为止, 它就被选出的大王。

设计要求

由用户输入开始时的猴子数 n , 第一个报数的猴子的编号 k 及报数值 m , 程序输出猴子出圈的顺序及最后留在圈中的那个猴子的序号。

项目提示

用一个不带头结点的循环链表来处理这个问题: 先构成一个有 n 个结点的单循环链

2.4.1 选择题

1. 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_____。
- 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
2. 对顺序存储的线性表,设其长度为 n,且在任何位置上插入或删除操作都是等概率的,则插入一个元素时平均要移动表中的()个元素。
- A. $n/2$ B. $(n+1)/2$ C. $(n-1)/2$ D. n
3. 设单链表指针 p 指向结点 A,结点 A 存在后继结点,若要删除结点 A 的后继结点,则需要修改指针的操作为_____。
- A. $p \rightarrow next = p \rightarrow next \rightarrow next$ B. $p = p \rightarrow next$
- C. $p = p \rightarrow next \rightarrow next$ D. $p \rightarrow next = p$
4. 在非空循环双向链表中,在 q 所指的结点前插入一个由 p 所指结点的过程依次为:
- $p \rightarrow next = q; p \rightarrow prior = q \rightarrow prior;$ _____ ; $q \rightarrow prior = p$ 。
- A. $q \rightarrow next = p$ B. $q \rightarrow prior \rightarrow next = p$
- C. $p \rightarrow prior \rightarrow next = p$ D. $p \rightarrow next \rightarrow prior = p$
5. 某链表中最常用的操作是在最后一个元素之后插入一个元素和删除最后一个元素,则采用_____存储方式最节省运算时间。
- A. 单链表 B. 双链表 C. 单循环链表 D. 带头结点的双循环链表

2.4.2 填空题

1. 带头结点的单链表 H 为空的条件是(指针 head 指向头结点)_____。
2. 非空单循环链表 head 中 * p 是尾结点的条件是(指针 head 指向头结点)_____。
3. 在一个单链表中,在 p 所指结点之后插入一个由指针 s 所指结点,应执行 s -> next = _____ 和 p -> next = _____ 的操作。
4. 在一个单链表中删除 p 所指结点时,依照下面的操作可实现,请填入所缺语句:
q = p -> next;
p -> data = p -> next -> data;

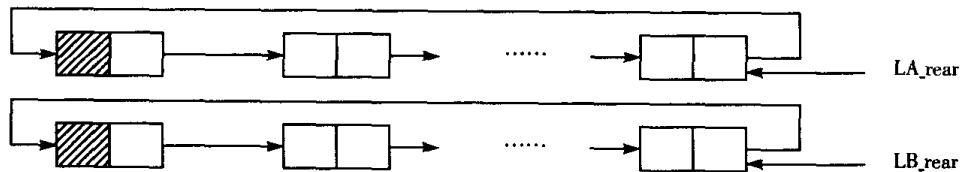
free(q);
5. 在一个单链表中,在 p 所指结点之前插入一个由指针 s 所指结点,可执行以下操

作:

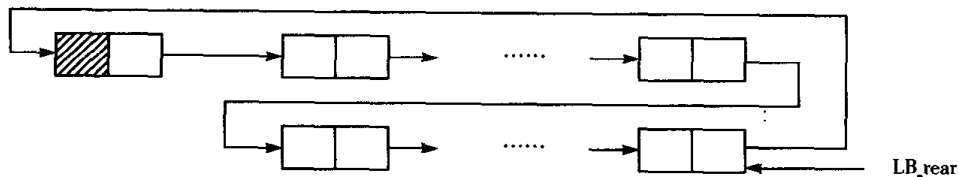
```
s -> next = _____;
p -> next = s;
t = p -> data;
p -> data = _____;
s -> data = _____;
```

6. 有两个仅设尾指针的循环链表 A 与 B 如图 (a) 所示, 尾指针分别为 LA_rear 与 LB_rear, 现在要将两个链表合并成一个链表如图 (b) 所示, 新链表的尾指针为 LB_rear, 请写出所缺的执行语句:

(1) p = LA_rear -> next; (2) _____;
(3) free(LB_rear -> next); (4) _____;



(a)



(b)

7. 函数 merge(int a[], int n, int b[], int m, int *c) 是将两个从小到大有序的整数序列 a 和 b 复制合并成一个有序整数序列 c, 其中形参 n 和 m 分别是数组 a 和 b 的元素个数。

```
void merge(int a[], int n, int b[], int m, int *c)
{
    int i, j;
    for(i = j = 0; i < n && j < m;)
        *c++ = a[i] < b[j] ? a[i++] : b[j++];
    while(①) *c++ = a[i++];
    while(②) *c++ = b[j++];
}
```

8. 在单链表(表头指针为 head)的元素中找出最后一个值为 e 的元素, 返回其指针; 如果找不到, 返回 NULL。完成以下程序。

```
typedef struct LinkNode
{

```

```

    int data;
    struct LinkNode *next;
}Node;
Node *search_link(Node *head,int e)
{
    Node *p,*q;
    q = ①;
    for(p=head; ②;p=p->next)
        if(p->data==e) ③;
    return q;
}

```

9. 设链表结点的类型为

```
typedef struct elem
```

```

{
    int val;
    struct elem *next;
}intNode;

```

函数 `merge(int *a,int *b)` 是将两个升序链表 `a` 和 `b` 合并成一个升序链表。

```

intNode *merge(intNode *a,intNode *b)
{
    intNode *h=a,*p,*q;
    while(b){
        for(p=h;p&&p->val<b->val;q=p,p=p->next);
        if(p==h) ①; else ②;
        q=b; b=b->next; ③;
    }
    return h;
}

```

10. 以下程序中的函数 `first_insert()` 的功能是在已知链表的首结点之前插入一个指定值的结点;函数 `reverse_copy()` 的功能是按已知链表复制出一个新链表,但新链表的结点链接顺序与已知链表的结点链接顺序相反;函数 `print_link()` 用来输出链表中各结点的值;函数 `free_link()` 用来释放链表全部结点空间。

```

#include <stdio.h>
#include <malloc.h>
typedef struct node{
    int val;
    struct node *next;
}Node;
void first_insert(Node *p,int v)
{
    Node *q=(Node *)malloc(sizeof(Node));

```

```

    q->val = v; ①; p = ②;
}

Node * reverse_copy(Node * p)
{
    Node * u;
    for(u = NULL; p; p = p->next) first_insert(③);
    return u;
}

void print_link(Node * p)
{
    for(; ④) printf("%d\t", p->val);
    printf("\n");
}

void free_link(Node * p)
{
    Node * u;
    while (p != NULL) { u = p->next; free(p); ⑤; }
}

main()
{
    Node * link1, * link2;
    int i; link1 = NULL;
    for(i = 1; i <= 10; i++)
        first_insert(&link1, i);
    link2 = reverse_copy(link1);
    print_link(link1); free_link(link1);
    print_link(link2); free_link(link2);
}

```

2.5 习题解析

1. 叙述线性表两种存储结构各自的优缺点。

解 线性表有两种存储结构,它们是顺序存储结构(简称顺序表)和链式存储结构(简称链表)。

顺序表有如下特点:因为顺序表中各元素物理位置隐含其逻辑关系,所以不需要额外存储空间来维系线性表的逻辑结构;又因为顺序表是用数组来存放线性表中各元素的(即连续存储),所以对线性表的存取很容易,但对线性表进行插入或删除操作时需移动

大量元素,消耗时间较多。另外,顺序表需事先分配存储空间,若线性表最大长度较难确定,则必须按线性表最大可能长度分配空间。若线性表长度变化较大,则使存储空间不能得到充分利用;如果存储空间分配过小,又可能导致溢出。

链表有如下特点:它不要求逻辑上相邻的元素在物理位置上也相邻,为了表示出每个元素与其直接后继元素之间的关系,除了存储元素本身的信息外,还需存储一个指示其直接后继的存储位置信息,因此,链表占有较多的存储空间。但线性链表的插入或删除操作较为容易,不需要大量移动元素。另外,链表结点的存取总是从表头开始的。

2. 设计一个算法,将 x 插入到一个有序(从小到大排序)的线性表(顺序存储结构)的适当位置上,以保持线性表的有序性。

解 算法由主函数和插入、输出三个函数组成。

程序如下:

```
#include <stdio.h>
#define MAXLEN 20
void insertsqlist(int x,int s[],int *np) /* 插入结点 x 到有序表中 */
{
    int n;
    n = *np;                               /* 有序表长度 */
    if(n == 0)
        s[n] = x;
    else
    {
        while(s[n-1] > x)                  /* 从后向前移动结点,确定 x 的插入位置 */
        {
            s[n] = s[n-1];
            n = n-1;
        }
        s[n] = x;                          /* 插入 x */
    }
    ++(*np);                               /* 有序表长度加 1 */
}

void print(int s[],int *np)                /* 输出有序表各结点 */
{
    int i,n;
    n = *np;
    for(i=0;i<n;i++)
    {
        printf("%d ",s[i]);
    }
    printf("\n");
}
```

46 / 数据结构实训与习题解析 □

```
main()                                /* 主函数 */
{
    int a[MAXLEN] = {11,33,55,66,77,88,99};
    int c,*e;
    c=30;                             /* 插入 30 */
    *e=7;
    print(a,e);                       /* 插入前输出有序表 */
    insertsqlist(c,a,e);              /* 插入一个结点到有序表 */
    print(a,e);                       /* 插入后输出有序表 */
}
```

输出结果为:

11 33 55 66 77 88 99

11 30 33 55 66 77 88 99

3. 设计一个算法,从顺序表中删除自第 i 个结点开始的 k 个结点。

解 算法由主函数和删除、输出三个函数组成。

程序如下:

```
#include <stdio.h>
#define MAXLEN 20
int deletesqlist(int i,int k,int s[],int *np)
/* 删除自第 i 个结点开始的 k 个结点 */
{
    int j,n;
    n = *np;
    if((n==0) || (i<1) || (i>n))
        return(0);                    /* 删除失败 */
    else
    {
        if((n-i+1)<k)                  /* 从第 i 个结点开始到末尾的个数不足 k 个,直接修改 *np 的值 */
        {
            *np = i-1;
            return(1);
        }
        else                          /* 从数组下标(i+k-1)元素到(n-1)元素向前移动 k 个位置 */
        {
            for(j=i+k-1;j<n;j++)
                s[j-k]=s[j];
            *np = n-k;
            return(1);
        }
    }
}
```

```

void print(int s[],int *np)                /* 输出顺序表各结点 */
{
    int i,n;
    n = *np;
    for(i=0;i<n;i++)
    {
        printf("%d ",s[i]);
    }
    printf("\n");
}

main()
{
    int a[MAXLEN] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
    int i,k,*e;
    *e=20;
    print(a,e);
    i=5;                                     /* 顺序表第5个结点 */
    k=5;                                     /* 删除自第5个结点开始的5个结点 */
    deletesqlist(i,k,a,e);
    print(a,e);
}

```

输出结果为:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

1 2 3 4 10 11 12 13 14 15 16 17 18 19 20

4. 设计一个算法,将顺序表中所有数据域为 x 的结点的数据域替换成 y 。

解 算法由主函数和结点替换、输出三个函数组成。

程序如下:

```

#define MAXLEN 10

void replqlist(int x,int y,int s[],int *np) /* 结点替换 */
{
    int i,n;
    n = *np;
    for(i=0;i<n;i++)
    {
        if(s[i]==x)
            s[i]=y;                       /* 用 y 替换 x */
    }
}

void print(int s[],int *np)
{
    int i,n;

```

```

    n = *np;
    for(i=0; i<n; i++)
    {
        printf("%d ", s[i]);
    }
    printf("\n");
}

main()
{
    int a[MAXLEN] = {11,33,55,66,77,88,99,33,44,77};
    int x,y,*e;
    x=33;
    y=30;
    *e=10;
    print(a,e);                                /* 替换前输出 */
    replqlist(x,y,a,e);                        /* y 替换 x */
    print(a,e);                                /* 替换后输出 */
}

```

输出结果为:

11 33 55 66 77 88 99 33 44 77

11 30 55 66 77 88 99 30 44 77

5. 设计一个算法,产生一个有4个结点的单链表,这些结点的数据域分别是 a,b,c,d,且表头指针是 head。

解 算法由主函数和创建单链表,输出三个函数组成。

程序如下:

```

#include <stdio.h>
typedef struct node
{
    char data;
    struct node *next;
} Node;

Node *create()                                /* 创建单链表,返回一个指向链表表头的指针 head */
{
    Node *head,*q,*p;                        /* 定义指针变量 */
    char i;
    head=(Node *)malloc(sizeof(Node));        /* 申请新的存储空间 */
    q=head;
    for(i='a'; i<='e'; i++)
    {
        p=(Node *)malloc(sizeof(Node));
        p->data=i;
    }
}

```

```

    q->next = p;
    q = p;
}
q->next = NULL;
return(head);                                /* 返回表头指针 head */
}
void print(Node *head)
{
    Node *p;
    p = head->next;
    while(p != NULL)
    {
        printf("%c ", p->data);                /* 输出链表元素 */
        p = p->next;
    }
    printf("\n");
}
main()
{
    Node *a;
    a = create();
    print(a);
}

```

输出结果为:

a b c d

6. 设计一个算法,将结点数据域依次是 $a_1, a_2, \dots, a_n$ ($n \geq 0$) 的一个单链表的所有结点逆置,即第一个结点的数据域变为 a_n ,最后一个结点的数据域变为 a_1 。

解 算法由主函数和创建单链表、就地逆转、输出四个函数组成。

程序如下:

```

#include <stdio.h>
typedef struct node
{
    int data;
    struct node *next;
} Node;
Node *create()                                /* 创建单链表,返回一个指向链表表头结点的指针 head */
{
    Node *head, *q, *p;                        /* 定义指针变量 */
    int a, n;                                  /* 链表大小为 n */
    scanf("%d", &n);
    head = (Node *)malloc(sizeof(Node));        /* 申请新的存储空间,建立表头结点 */
}

```

```

    head->data = n;
    q = head;
    while(n > 0)
    {
        scanf("%d", &a);                                /* 输入新元素 */
        p = (Node *) malloc(sizeof(Node));
        p->data = a;
        q->next = p;
        q = p;
        n = n - 1;
    }
    q->next = NULL;
    return(head);                                        /* 返回表头指针 head */
}

void print(Node * head)                                /* 输出单链表各元素 */
{
    Node * p;
    p = head->next;
    while(p != NULL)
    {
        printf("%d ", p->data);
        p = p->next;
    }
    printf("\n");
}

int jdnz(Node * head)                                  /* 单链表各元素就地逆转 */
{
    Node * p, * q, * r;
    if(((head->next) == NULL) || (((head->next)->next) == NULL))
        return(1);                                     /* 如果单链表为空或只有一个结点,不做就地逆转,返回1 */
    else
        /* 由 p,q,r 三个指针边访问单链表结点,边完成逆转 */
        {
            r = head->next;
            q = r;
            r = r->next;
            do
            {
                p = q;
                q = r;
                r = r->next;
                q->next = p;
            } while(r != NULL);
        }
}

```

```

    (head->next)->next=NULL;
    head->next=q;
    return(0);
}
main()
{
    Node *a;
    a=create();
    print(a);
    jdnz(a);
    print(a);
}

```

输出结果为:

```

5(按回车)                                /* 输入单链表的大小为 5 * /
1 2 3 4 5(按回车)                        /* 输入单链表的 5 个结点值 * /
1 2 3 4 5                                /* 输出单链表各结点值 * /
5 4 3 2 1                                /* 就地逆转后再输出单链表各结点值 * /

```

7. 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, 编写相应函数求数组 A 靠边元素之和。

解 为操作方便, 实际算法将二维数组 $A[m][n]$ 的 m, n 改为 $1 \leq m \leq 10, 1 \leq n \leq 10$ 。

算法由主函数和求数组靠边元素之和、输出数组全部元素、输出数组全部靠边元素四个函数组成。

程序如下:

```

#include <stdio.h>
#define M 10
#define N 10
int sumjzzbys(int a[M][N], int m, int n)    /* 求数组 A 靠边元素之和 * /
{
    int i, j, sum=0;
    if((m==0)&&(n==0))
        return(a[0][0]);
    else
    {
        for(i=0; i<m; i++)
        {
            for(j=0; j<n; j++)
            {
                /* 根据数组中靠边元素下标特征, 筛选数组全部靠边元素, 并累加 * /
                if((i==0) || (i==m-1) || (j==0) || (j==n-1))
                    sum = sum + a[i][j];          /* 累加靠边元素 * /
            }
        }
    }
}

```

```

        return(sum);
    }
}

void print1(int a[M][N],int m,int n)          /* 输出数组 a[M][N]全部元素 */
{
    int i,j;
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
            printf("%d ",a[i][j]);          /* "%d "中有2个空格 */
        printf("\n");
    }
    printf("\n");
}

void print2(int a[M][N],int m,int n)          /* 输出数组 a[M][N]全部靠边元素 */
{
    int i,j;
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
            if((i==0) || (i==m-1) || (j==0) || (j==n-1))
                printf("%d ",a[i][j]);      /* "%d "中有2个空格 */
            else
            {
                printf(" ");                /* " "中有3个空格 */
            }
        printf("\n");
    }
    printf("\n");
}

main()
{
    int t[M][N] = { {1,1,1,1,1,1,1,1,1,1},
                     {2,2,2,2,2,2,2,2,2,2},
                     {3,3,3,3,3,3,3,3,3,3},
                     {4,4,4,4,4,4,4,4,4,4},
                     {5,5,5,5,5,5,5,5,5,5},
                     {6,6,6,6,6,6,6,6,6,6},
                     {7,7,7,7,7,7,7,7,7,7},
                     {8,8,8,8,8,8,8,8,8,8},
    };
    /* 预置数组元素 */

```

```

        {9,9,9,9,9,9,9,9,9},
        {0,0,0,0,0,0,0,0,0}};

int a,b;
scanf("%d",&a);
scanf("%d",&b);
print1(t,a,b);
print2(t,a,b);
printf("%d\n",sumjzzbys(t,a,b));
}

```

输出结果为:

5 5(按回车)

```

1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5
1 1 1 1 1
2           2
3           3
4           4
5 5 5 5 5

```

48

第 3 章

栈和队列

3.1 基础知识篇

学习要点：

- 栈的基本概念和基本运算
- 队列的基本概念和基本运算
- 栈和队列两种存储类型的实现

栈和队列也是线性表,它们是两种特殊的线性表。其特殊性在于其上的操作是线性表操作的子集,即栈和队列的插入、删除操作是受限制的。栈和队列广泛应用于操作系统、编译原理、各种应用软件系统以及程序设计中,因而掌握关于栈和队列的基本操作及其具体应用是解决实际问题的基础。

3.1.1 栈

栈(Stack)又称堆栈,是限定仅在表尾进行插入和删除操作的线性表,即栈的修改是按先进后出的原则进行的,因此,栈又称为先进后出表(First In Last Out,简称 FILO),也可把栈称为后进先出表(Last In First Out,简称 LIFO)。如图 3.1 所示。

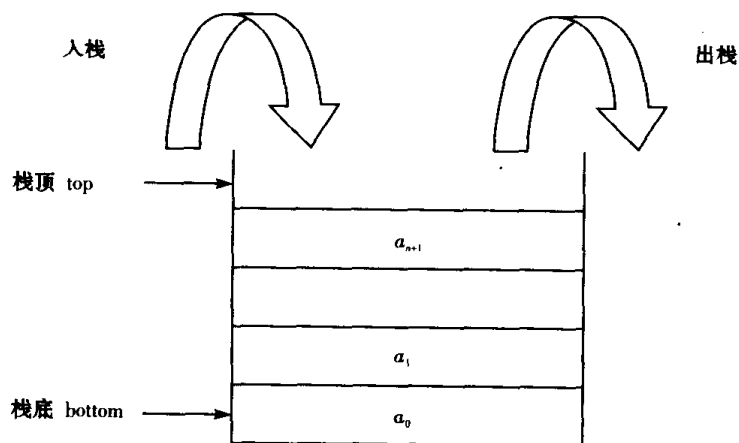


图 3.1 出入栈操作示意图

表尾端称为栈顶(top),栈顶的第一个元素被称为栈顶元素,相对地,把另一端称为栈底(bottom),即表头端称为栈底。不含数据元素的栈称为空栈。

向一个栈插入新元素称为进栈或入栈,它是把该元素放到栈顶元素的上面,使之成为

新的栈顶元素;从一个栈删除元素称为出栈或退栈,它是把栈顶元素删除掉,使其下面的相邻元素成为新的栈顶元素。

例如,假定一个栈 S 为 (a,b,c) ,其中字符 c 的一端为栈顶,字符 c 为栈顶元素。若向 S 压入一个元素 d ,则 S 变为 (a,b,c,d) ,此时字符 d 为栈顶元素;接着若从栈 S 中依次删除两个元素,则首先删除的是字符 d ,接着删除的是字符 c ,栈 S 变为 (a,b) ,栈顶元素为 b 。

在日常生活中,有许多栈的应用例子,如刷洗盘子时,依次把每个洗净的盘子放到洗好的盘子上,相当于进栈;取用盘子时,从一摞盘子上一个接一个地向下拿,相当于出栈。又如向枪支弹夹里装子弹时,子弹被一个接一个地压入,相当于进栈;射击时子弹总是从顶部一个接一个地被射出,相当于子弹出栈。

由于栈是线性表,因而栈的存储结构也可采用顺序和链式两种形式,顺序存储的栈称为顺序栈,链式存储的栈称为链栈。

1. 顺序栈

(1) 基本知识

利用一组地址连续的存储单元依次存储自栈底到栈顶的数据元素,同时附设 top 指针指示栈顶元素的位置。顺序栈的数据类型定义如下:

```
#define MAXSIZE 100          /* 定义足够的长度 */
typedef struct
{
    datatype data[MAXSIZE];    /* datatype 表示元素类型 */
    int top;
} seqstack;
```

通常的做法是以 $top = 0$ 表示空栈,但由于 C 语言中数组的下标约定从 0 开始,则当以 C 语言作描述语言时,设 $top = -1$ 表示栈为空。所以,在顺序存储的栈中, top 的值为 -1 表示栈空,每次向栈中压入一个元素时,首先使 top 增 1,用以指示新的栈顶位置,然后再把元素赋值到这个位置上。每次从栈中弹出一个元素时,首先取出栈顶元素,然后使 top 减 1,此位置上的元素成为新的栈顶元素。由此可知,对顺序栈的插入和删除运算相当于是顺序表(即顺序存储的线性表)的表尾进行的,其时间复杂度为 $O(1)$ 。

下面我们通过一个具体的例子来说明顺序栈的存储结构及其操作。

设一个栈 S 为 (a,b,c,d,e) ,对应的顺序存储结构如图 3.2(a)所示。若向 S 中插入一个元素 f ,则对应如图 3.2(b)所示。若接着执行两次出栈操作后,则栈 S 对应如图 3.2(c)所示。若依次使栈 S 中的所有元素出栈,则栈 S 变为空,如图 3.2(d)所示。在图 3.2 中栈是垂直画出的,并且使下标编号向上递增,这样可以形象地表示出栈顶在上,栈底在下。

在一个顺序栈中,若 top 指向了 $MAXSIZE - 1$ 的位置,则表示栈满;若 top 的值等于 -1 ,则表示栈空。向一个满栈插入元素和从一个空栈删除元素都是不允许的(前者称为上溢,后者称为下溢),应该停止程序运行或进行特别处理。

利用栈底位置相对不变的特性,两个顺序栈可以共享一个一维数据空间,以互补余

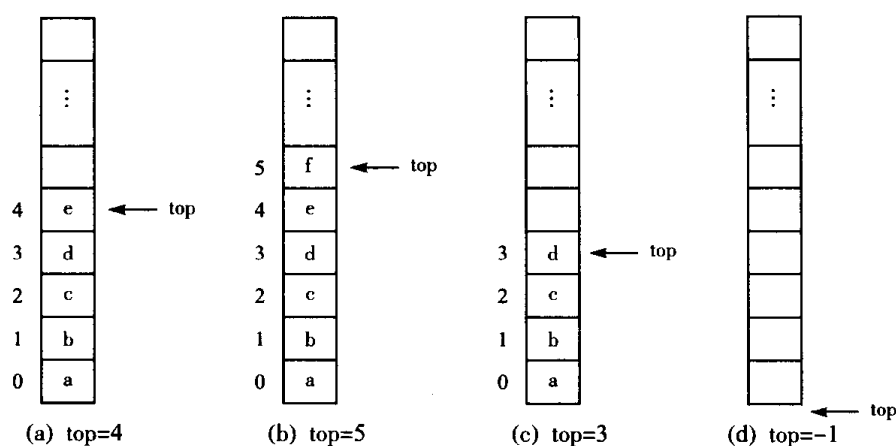


图 3.2 顺序栈实例

缺。具体实现时可以将两个栈的栈底分别设在空间的两端,让它们的栈顶各自向中间延伸,这样,两个栈可以互相调节,它们的逻辑可使用空间是整个存储空间,只有在整个存储空间被两个栈占满时才发生上溢,因此产生上溢的可能性相对较小。具体情况如图 3.3 所示。

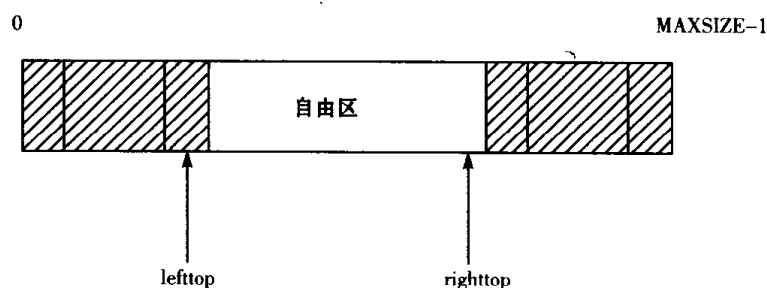


图 3.3 两个栈共享同一存储空间

(2) 基本操作

① 栈的初始化

```
void init_seqstack(seqstack *s)
```

```
{
```

```
    s->top = -1;
```

```
}
```

② 判断栈空

```
int empty_seqstack(seqstack *s)
```

```
{
```

```
    if(s->top == -1) return 1;
```

```
    else return 0;
```

```
}
```

③ 取栈顶元素

```
datatype get_seqstack(seqstack *s)
```

```
{
```

```
    if(s->top == -1) printf("栈空,取元素失败! \n");
```

```

    else return(s->data[s->top]);
}

④入栈
seqstack *push_seqstack(seqstack *s, datatype x)
{
    if (s->top == MAXSIZE-1)
    {
        printf("上溢\n");
        return 0;                      /* 入栈操作失败 */
    }
    else
    {
        s->top++;
        s->data[s->top] = x;
        return s;                      /* 入栈操作成功 */
    }
}

```

```

⑤出栈
int pop_seqstack(seqstack *s, datatype y)
{
    if (s->top < 0)
    {
        printf("下溢\n");
        return 0;                      /* 出栈操作失败 */
    }
    else
    {
        y = s->data[s->top];
        s->top--;
        return 1;                      /* 出栈操作成功 */
    }
}

```

2. 链栈

(1) 基本知识

为了克服栈的顺序存储结构所带来的在操作中需要移动大量数据元素的不足,栈可以采用链式存储结构,称为链栈。栈的链式存储结构与线性表的链式存储结构相同,是通过由结点构成的单链表实现的,此时表头指针被称为栈顶指针,由栈顶指针指向的表头结点被称为栈顶结点。当向一个链栈插入元素时,是把该元素插入到栈顶,即使该元素结点的指针域指向原来的栈顶结点,而栈顶指针则修改为指向该元素结点,使该结点成为新的栈顶结点。当从一个链栈中删除元素时,是把栈顶元素结点删除掉,即取出栈顶元素后,使栈顶指针指向原栈顶结点的后继结点。由此可知,对链栈的插入和删除操作是在单链

表的表头进行的,其时间复杂度为 $O(1)$ 。

下面我们来看一个具体的例子。

设一个栈为(a,b,c),当采用链式存储时,对应的存储结构如图 3.4(a)所示,其中 top 表示栈顶指针,其值为存储元素 c 结点的地址。当向这个栈插入一个元素 d 后,对应如图 3.4(b)所示。当从这个栈依次删除两个元素后,对应如图 3.4(c)所示。当链栈中的所有元素全部出栈后,栈顶指针 top 的值为空,即常量 NULL 所表示的数值 0。

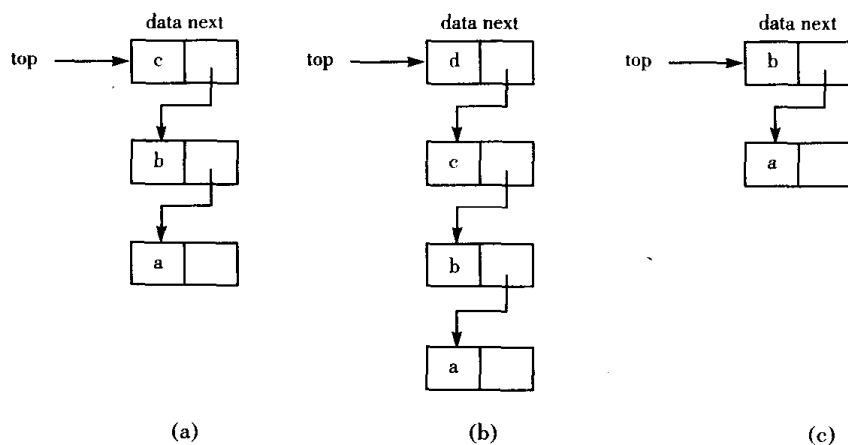


图 3.4 链式存储结构示意图

链栈的数据类型定义如下:

```
typedef struct node
{
    datatype data;
    struct node *next;
}linkstack;
```

(2) 基本操作

① 栈的初始化

```
void init_linkstack(linkstack *s)
{
    s = NULL;
}
```

② 判断栈空

```
int empty_linkstack(linkstack *s)
{
    if(s == NULL) return 1;
    else return 0;
}
```

③ 取栈顶元素

```
datatype get_linkstack(linkstack *s)
{

```

```

    if(s == NULL) printf("栈空,取元素失败! \n");
    else return(s->data);
}

```

④入栈

```

linkstack *push_linkstack(linkstack *s,datatype x)
{
    linkstack *p;
    p=(linkstack*) malloc(sizeof(linkstack));
    p->data=x;
    p->next=s;
    s=p;
    return s;
}

```

⑤出栈

```

int pop_linkstack(linkstack *s,datatype y)
{
    linkstack *p;
    if(s == NULL)
    {
        printf("下溢\n");
        return 0;
    }
    else
    {
        p=s;
        y=s->data;
        s=p->next;
        free(p);
        return 1;
    }
}

```

鉴于栈的操作受限,其元素的插入和删除仅在表的一端进行,因此不必设置头结点。算法中头指针就是栈顶指针 top,top 唯一地确定一个链栈。

3.1.2 队列

队列(Queue)简称队,它也是一种运算受限的线性表,其限制是仅允许在表的一端进行插入,而在表的另一端进行删除。我们把允许插入的一端称作队尾(rear),允许删除的一端称作队首(front);向队列中插入新元素称为进队或入队,新元素进队后就成为新的队尾元素;从队列中删除元素称为离队或出队,元素离队后,其后继元素就成为新的队首元素。由于队列的插入和删除操作分别是在各自的一端进行的,每个元素必然按照进入的次序离队,所以又把队列称为先进先出表(First In First Out,简称 FIFO)。队列也可以通

过顺序和链式存储结构实现。队列示意图如图 3.5 所示。

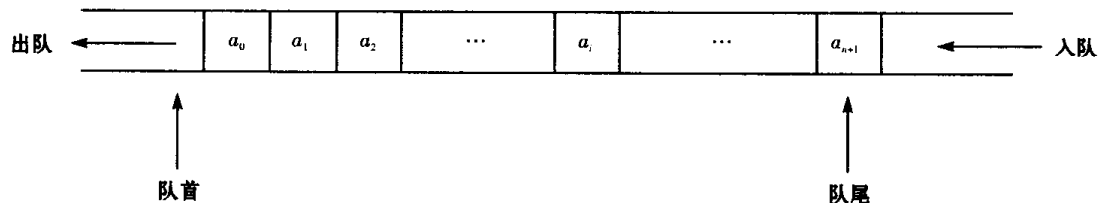


图 3.5 队列示意图

例如,假定有 a, b, c, d 四个元素依次入队,则得到的队列为 (a, b, c, d) ,其中字符 a 为队首元素,字符 d 为队尾元素。若从此队中删除一个元素,则字符 a 出队,字符 b 成为新的队首元素,此队列变为 (b, c, d) ;若接着向该队列插入一个字符 e ,则 e 成为新的队尾元素,此队列变为 (b, c, d, e) ;若接着做三次删除操作,则队列变为 (e) ,此时只有一个元素 e ,它既是队首元素又是队尾元素,当它被删除后队列变为空。

在日常生活中,人们购物或等车时所排的队就是一个队列,新来购物或等车的人接到队尾(即进队),站在队首的人购到物品或上车后离开(即出队),当最后一个人离队后,队列为空。

队列在计算机系统中的应用也非常广泛。例如操作系统中的作业排队。在多道程序运行的计算机系统中,可以同时有多个作业运行,它们的运算结果都需要通过通道输出,若通道尚未完成输出,则后来的作业应排队等待。当通道完成输出时,则从队列的队首退出作业作输出操作,而凡是申请该通道输出的作业都从队尾进入该队列。

计算机系统中输入输出缓冲区的结构也是队列的应用。在计算机系统中经常会遇到两个设备之间的数据传输,通常不同的设备处理数据的速度是不同的,当需要在它们之间连续处理一批数据时,高速设备总是要等待低速设备,这就造成计算机处理效率的大大降低。为了解决这一速度不匹配的矛盾,通常需要在这两个设备之间设置一个缓冲区。这样,高速设备就不必每次都等待低速设备处理完一个数据,而是把要处理的数据依次从一端加入缓冲区,而低速设备从另一端取走要处理的数据。

1. 顺序队列

(1) 基本知识

顺序队列即队列的顺序存储结构,它是用一组地址连续的存储单元存放队列中的元素。

与对栈的插入操作一样,每次在进行队列插入前,也要判断队列是否已满(即数组空间是否已被用完),若是则停止插入,终止程序运行,否则可向队列中插入新元素。在顺序队列中进行插入和删除时,不需要比较和移动任何元素,只需要修改队尾和队首指针,并向队尾写入元素或从队首取出元素,所以其时间复杂度为 $O(1)$ 。数据类型定义如下:

```
typedef struct
{
    datatype data[MAXSIZE];
    int front, rear;
} sequeue;
```

队空的条件: $\text{front} = \text{rear} = -1$; 队满的条件: $\text{rear} = \text{MAXSIZE} - 1$ 。

(2) 基本操作

① 队列的初始化

```
void init_sequeue(sequeue *sq)
{
    sq->front = sq->rear = -1;
}
```

② 判断队列空

```
int empty_sequeue(sequeue *sq)
{
    if(sq->front == sq->rear) return 1;
    else return 0;
}
```

③ 取队首元素

```
datatype get_sequeue(sequeue *sq)
{
    if(sq->front == sq->rear)
    {
        printf("队列为空,取元素失败! \n");
        return 0;
    }
    else return(sq->data[sq->front + 1]);
}
```

④ 入队

```
int insert_sequeue(sequeue *sq, datatype x)
{
    if(sq->rear == MAXSIZE - 1)
    {
        printf("上溢");
        return 0;
    }
    else
    {
        sq->rear = sq->rear + 1;
        sq->data[sq->rear] = x;
    }
    return 1;
}
```

⑤ 出队

```
int delete_sequeue(sequeue *sq, datatype y)
{
}
```

```

if (sq->rear == sq->front)
{
    printf("下溢");
    return 0;
}
else
{
    sq->front = sq->front + 1;
    y = sq->data[sq->front];
}
return 1;
}

```

2. 链队列

(1) 基本知识

链队列是由结点构成的单链表实现的,此时只允许在单链表的表头进行删除和在单链表的表尾进行插入。一个链队列由一个头指针和一个尾指针惟一确定。因此它需要使用两个指针:队首指针 `front` 和队尾指针 `rear`。用 `front` 指向队首(即表头)结点的存储位置,用 `rear` 指向队尾(即表尾)结点的存储位置。

对链队列的插入和删除操作同样不需要比较和移动元素,只需要修改个别相关指针和进行结点的动态分配或回收操作,所以其时间复杂度为 $O(1)$ 。另外,使用链队列不存在队满的问题,因为它使用的结点是动态分配的,只要内存中动态存储区仍有可用空间,就可以得到一个新结点,使之插入到链队列中;链队列也可能为空,此时 `front` 和 `rear` 指针均为空。链队列的初态如图 3.6 所示。

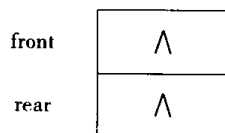


图 3.6 链队列的初态

下面我们一起来看一个具体的例子。

设一个队列为(a,b,c),则对应的链式存储结构如图 3.7(a)所示,当向该链队列插入一个元素 d 后,对应如图 3.7(b)所示,当接着从中删除队首元素 a 后,对应如图 3.7(c)所示。

数据类型定义如下:

```

typedef struct node
{
    datatype data;
    struct node *next;
} qnode;                                /* 链队列结点的类型 */
typedef struct
{
    qnode *front, *rear;
}

```

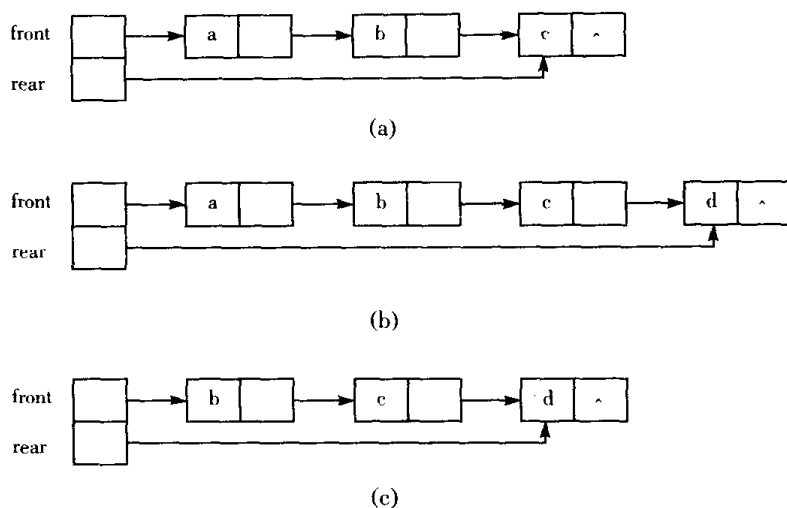


图 3.7 链队列的插入和删除操作示意图

```
linkqueue; /* 包含头尾指针的链队列 */
```

为操作方便,在队首结点前附加一个头指针,且头指针指向头结点。

队空的判定条件: $q \rightarrow \text{front} = q \rightarrow \text{rear}$ 。

(2) 基本操作

① 链队列的初始化

```
void init_linkqueue(linkqueue *q)
{
    q = (struct qnode *)malloc(sizeof(struct qnode));
    q->front = q->rear = NULL;
}
```

② 判断链队列空

```
int empty_linkqueue(linkqueue *q)
{
    if(q->front == q->rear) return 1;
    else return 0;
}
```

③ 取链队列首元素

```
datatype get_linkqueue(linkqueue *q)
{
    if(q->front == q->rear) printf("队列为空,取元素失败! \n");
    else return(q->front->next->data);
}
```

④ 入链队列

```
void insert_linkqueue(linkqueue *q,datatype x)
{
    q->rear->next = (qnode *)malloc(sizeof(qnode));
    q->rear = q->rear->next;
```

```

q->rear->data = x;
q->rear->next = NULL;
}

```

⑤ 出链队列

```

datatype del_linkqueue(linkqueue *q,datatype y)
{
    qnode *p;
    if (q->front == q->rear)           /* 判别是否下溢 */
    {
        printf("下溢\n");
        return 0;
    }
    else                               /* 队首结点的数据域值赋给 y,并删除队首结点 */
    {
        p = q->front->next;
        q->front->next = p->next;
        y = p->data;
        if (p->next == NULL)           /* 若删除后队列为空,则修改 rear */
            q->rear = q->front;
        return y;
    }
}

```

3. 循环队列

(1) 基本知识

对于顺序队列,在操作过程中有可能出现溢出,解决此问题的方法是将顺序存储结构的队列假想成一个首尾相接的圆环,这种设想下的队列称为循环队列,如图 3.8 所示。其数据类型说明与顺序队列相同。循环队列队空的判定条件: $q \rightarrow rear == q \rightarrow front$ 。队满判定条件: $(q \rightarrow rear + 1) \% MAXSIZE == q \rightarrow front$ 。

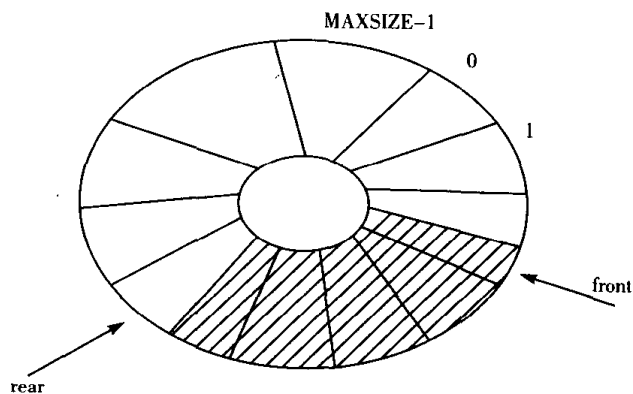


图 3.8 循环队列示意图

(2) 基本操作(以顺序存储结构为例)

①入循环队列

```

int insert_cyqueue(sequeue *cq,datatype x)
{
    if ((cq->rear+1)% MAXSIZE == cq->front)
    {
        printf("溢出\n");
        return 0;
    }
    else
    {
        cq->rear=(cq->rear+1)% MAXSIZE;
        cq->data[cq->rear]=x;
        return 1;
    }
}

```

②出循环队列

```

datatype delete_cyqueue(sequeue *cq, datatype y)
{
    if (cq->front == cq->rear)
    {
        printf("下溢\n");
        return NULL;
    }
    else
    {
        cq->front=(cq->front+1)% MAXSIZE;
        y=cq->data[cq->front];
        return y;
    }
}

```

3.2 实训篇

实训目的:

- 掌握栈和队列的概念和特点
- 掌握栈和队列在顺序和链式存储结构下的插入、删除算法
- 认真分析项目实例中的内容,将相关程序在计算机上运行实现

3.2.1 栈的应用——表达式求值问题

案例描述

求一个数学表达式的值:用户输入一个包含正整数、括号和四则运算符(“+”、“-”、“*”、“/”)的算术表达式,计算其结果。

设计分析

用户在键盘上输入表达式时,通常是运算符位于两个操作数之间,即为中缀表达式。例如: $2 * (4 + 6)$ 。计算时遵循“自左向右计算,先乘除后加减,先括号内后括号外”的规则。因此中缀表达式的计算需要考虑括号、运算符优先级等诸多因素,比较麻烦。而后缀表达式(即波兰表达式)在格式上已经考虑了运算符优先级,消除了括号,运算时只需处理操作数和运算符,比较简洁。

因此,我们把输入的中缀表达式转换成后缀表达式,再编写算法处理计算过程,就能得到最后结果。

设计方案

我们把问题分成两个部分:一个是中缀表达式转换为后缀表达式算法的实现;另一个是对后缀表达式求值。

1. 实现第一个算法,可以设置两个工作栈:操作数栈和运算符栈,分别用来存放表达式中的操作数和运算符。设“\0”为表达式的终止符。算法的基本思想是:

- (1) 两个栈初始为空栈;
- (2) 从左向右扫描表达式,遇到操作数一律进操作数栈。以“\0”为表达式的终止标志;
- (3) 遇到运算符,则判断其优先级是否大于运算符栈栈顶元素的优先级。若大于则进栈,否则栈顶元素出栈并输出到操作数栈;
- (4) 遇到左括号则左括号进运算符栈。若遇到右括号则出栈并输出到操作数栈,一直到遇到左括号为止;
- (5) 重复步骤(2)、(3)、(4),直至整个表达式扫描结束,并且运算符栈为空(即整个表达式转换成后缀表达式)。

例如,要对算术表达式 $2 * (4 + 6) \backslash 0$ 求值,整个过程中操作数栈和运算符栈的状态如表 3-1 所示。

表 3-1 中缀表达式转换为后缀表达式

步骤	运算符栈 栈底→栈顶	操作数栈 栈底→栈顶	表达式扫描
1			2 * (4 + 6) \0
2		2	2 * (4 + 6) \0 ↑
3	*	2	2 * (4 + 6) \0 ↑

续表

步骤	运算符栈 栈底→栈顶	操作数栈 栈底→栈顶	表达式扫描
4	* (	2	2 * (4 + 6) \0 ↑
5	* (	2 4	2 * (4 + 6) \0 ↑
6	* (+	2 4	2 * (4 + 6) \0 ↑
7	* (+	2 4 6	2 * (4 + 6) \0 ↑
8	*	2 4 6 +	2 * (4 + 6) \0 ↑
9		2 4 6 + *	2 * (4 + 6) \0 ↑

注：在以上的讨论中，我们假定所输入的表达式没有语法错误。

2. 第二个算法的基本思想是：

(1) 设一个空栈 `stack`；

(2) 从左向右扫描后缀表达式；

(3) 遇到操作数压入栈 `stack` 中；

(4) 遇到运算符，就执行两次出栈操作，对出栈的两个数进行该运算符指定的运算，把结果压入栈 `stack` 中；

(5) 重复步骤(2)、(3)、(4)，直到扫描到表达式终止符“`\0`”。此时的栈顶元素就是最后结果。

例如，要对后缀表达式 `2 4 6 + * \0` 求值，整个过程中栈 `stack` 的状态和表达式扫描情况如表 3-2 所示。

表 3-2 后缀表达式计算

步骤	栈 <code>stack</code>	表达式扫描
1		2 4 6 + * \0
2	2	2 4 6 + * \0 ↑
3	2 4	2 4 6 + * \0 ↑
4	2 4 6	2 4 6 + * \0 ↑
5	2 10	2 4 6 + * \0 ↑
6	20	2 4 6 + * \0 ↑
7	20	2 4 6 + * \0 ↑

结构定义与算法实现：

```
#define MAXSIZE 100
```

```
/* 定义变量 */
```

```

void trans(char str[],char exp[])    /* 将算术表达式 str 转换成后缀表达式 exp */
{
    struct
    {
        char data[MAXSIZE];          /* 定义用于存放加、减、乘、除等符号的栈 */
        int top;                      /* 定义栈指针 */
    } op;
    char ch;
    int i=0,t=0;
    op.top = -1;
    ch = str[i]; i++;
    while (ch != '\0')                /* 扫描算术表达式数组是否结束 */
    {
        switch(ch)
        {
            case '(':                  /* 判断是否为 '(' */
                op.top++; op.data[op.top] = ch;
                break;
            case ')':                  /* 判断是否为 ')' */
                while (op.data[op.top] != '(')
                {
                    exp[t] = op.data[op.top];
                    op.top--;
                    t++;
                }
                op.top--;
                break;
            case '+':
            case '-':                  /* 判断是否为 '+' 或 '-' */
                while (op.top != -1 && op.data[op.top] != '(')
                {
                    exp[t] = op.data[op.top];
                    op.top--;
                    t++;
                }
                op.top++;
                op.data[op.top] = ch;
                break;
            case '*':
            case '/':                  /* 判断是否为 '*' 或 '/' */
                while (op.data[op.top] == '*' || op.data[op.top] == '/')
                {

```

```

        exp[t] = op.data[op.top];
        op.top --;
        t ++;
    }
    op.top ++; op.data[op.top] = ch;
    break;
case ' ': break;          /* 跳过空格 */
default:
    while (ch >= '0' && ch <= '9')
    {
        exp[t] = ch; t ++;
        ch = str[i]; i ++;
    }
    i --;
    exp[t] = '#'; t ++;    /* 用#表示数值是否结束 */
    }
    ch = str[i]; i ++;
    }
    while (op.top != -1)    /* 扫描结束,栈是否为空 */
    {
        exp[t] = op.data[op.top];
        t ++;
        op.top --;
    }
    exp[t] = '\0';        /* 给后缀表达式添加结束符 */
}

float compvalue(char exp[])    /* 计算后缀表达式的值 */
{
    struct
    {
        float data[MAXSIZE];    /* 存放数值 */
        int top;                /* 定义栈指针 */
    } st;
    float d;
    char ch;
    int t = 0;
    st.top = -1;
    ch = exp[t]; t ++;
    while(ch != '\0')        /* 扫描字符串是否结束 */
    {
        switch(ch)

```

```

    case '+': st.data[st.top-1] = st.data[st.top-1] + st.data[st.top];
        st.top--; break;
    case '-': st.data[st.top-1] = st.data[st.top-1] - st.data[st.top];
        st.top--; break;
    case '*': st.data[st.top-1] = st.data[st.top-1] * st.data[st.top];
        st.top--; break;
    case '/':
        if (st.data[st.top] != 0)
            st.data[st.top-1] = st.data[st.top-1] / st.data[st.top];
        else
        {
            printf("\n 除零错误! \n");    /* 判断分母是否为0, 如为0退出 */
            exit(0);
        }
        st.top--; break;
    default:
        d = 0;
        while(ch >= '0' && ch <= '9')    /* 判断是否为数字字符 */
        {
            d = 10 * d + ch - '0';
            ch = exp[t]; t++;
        }
        st.top++;
        st.data[st.top] = d;
    }
    ch = exp[t]; t++;
}
return st.data[st.top];
}

main()
{
    char str[MAXSIZE], exps[MAXSIZE];

    /* 定义存储算术表达式的字符数组和用于存放后缀表达式的数组 */
    printf("输入一个求值表达式, 只能包含 +, -, *, / 和正整数\n");
    printf("表达式:");
    gets(str);    /* 输入求值表达式 */
    printf("原表达式: %s\n", str);
    trans(str, exps);    /* 调用中缀表达式变后缀表达式的函数 */
    printf("后缀表达式: %s\n", exps);
    printf("计算结果: %g\n", compvalue(exps));
}

```

运行测试:

输入一个求值表达式,只能包含 +, -, *, / 和正整数

表达式: $2 * (4 + 6)$

原表达式: $2 * (4 + 6)$

后缀表达式: $2 \# 4 \# 6 \# + \# *$

计算结果: 20

3.2.2 队列的应用——迷宫问题

案例描述

求迷宫的最短路径:要求设计一个算法找一条从迷宫入口到出口的最短路径。

迷宫是一个如图 3.9 所示的 m 行 n 列的 0-1 矩阵(这里以 5 行 5 列为例),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动到周围 4 个方向上任一无障碍的单元。设计程序输出一条通过迷宫的路径或一个“不存在路径”的信息。

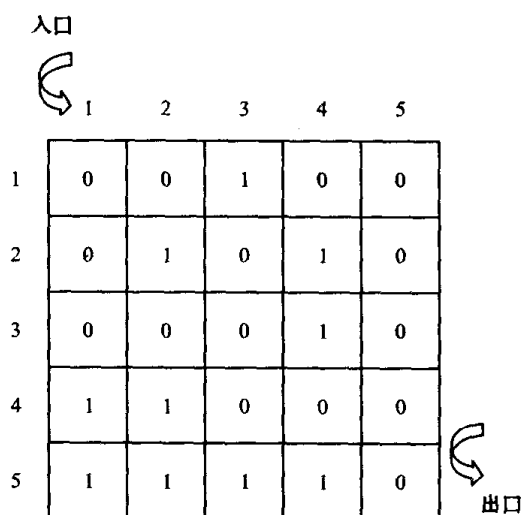


图 3.9 迷宫图

设计分析

我们从迷宫入口点 $(1,1)$ 出发,向四周试探方向,记下所有一步能到达的坐标点;然后依次再从这些点出发,再记下所有一步能到达的坐标点,依此类推,直到到达迷宫的出口点 (m,n) 为止,然后从出口点沿搜索路径回溯直至入口。这样就找到了一条迷宫的最短路径,否则迷宫无路径。由于先到达的点先向下搜索,所以利用了“先进先出”的数据结构——队列来保存已到达的坐标点。

设计方案

本算法的思想是从 $(1,1)$ 开始,利用队列的特点,一层一层向外扩展可走的点,直到找到出口为止,这个方法也就是在后续章节中介绍的关于图的广度优先搜索方法。

算法的变量定义:(1)我们用数组 $mg[1..m, 1..n]$ 表示迷宫,为了算法方便,在四周

加上一圈边框值,即变为数组 $mg[0..m+1,0..n+1]$ 表示迷宫;(2)用数组 zx 、 zy 分别表示 X、Y 方向的移动增量,其值如表 3-3 所示;(3)用一个结构数组 $queue[MAXSIZE]$ 作为队列的存储空间,因为迷宫中每个点至多被访问一次,所以 $MAXSIZE$ 至多等于 $m * n$ 。 $queue$ 的每一个结构有三个域: x 、 y 和 pre ,其中 x 、 y 分别为所到达的点的坐标, pre 为前驱点在 $queue$ 中的坐标;(4)另外还用队首指针 $front$ 和队尾指针 $rear$ 来指向队首和队尾元素。

表 3-3 zx 和 zy 数组的方向值定义

方向	北	东	南	西
下标	1	2	3	4
zx	-1	0	1	0
zy	0	1	0	-1

算法的求解过程:初始状态,队列中只有一个元素 $queue[1]$ 记录入口点的坐标 $(1,1)$,因为该点是出发点,因此没有前驱点, pre 域设为 0,队首指针 $front$ 和队尾指针 $rear$ 均指向它,此后搜索时都是以 $front$ 所指点为搜索的出发点,当搜索到一个可到达点时,将该点的坐标及 $front$ 所指点的位置入队,这样不但记下了到达点的坐标,还记下了它的前驱点。 $front$ 所指点的 4 个方向搜索完毕后,则出队,继续对下一点搜索。搜索过程中遇到出口点则成功,输出迷宫最短路径,算法结束;或者当前队空,即没有搜索点了,表明没有路径,算法也结束。

举例说明,如图 3.9 所示的迷宫,搜索路径的过程如表 3-4 所示。第 12 步找到了出口,该点的 pre 为 10,表示上一步为第 10 步,第 10 步的 pre 为 9,表示上一步为第 9 步,第 9 步的 pre 为 8,表示上一步为第 8 步,第 8 步的 pre 为 6,表示上一步为第 6 步,依此类推,最后到达第 1 步,也就是入口,这样的路径应是: $(1,1) \rightarrow (2,1) \rightarrow (3,1) \rightarrow (3,2) \rightarrow (3,3) \rightarrow (4,3) \rightarrow (4,4) \rightarrow (4,5) \rightarrow (5,5)$ 。

表 3-4 图 3.9 迷宫的路径搜索过程

步号	x	y	pre
1	1	1	0
2	1	2	1
3	2	1	1
4	3	1	3
5	3	2	4
6	3	3	5
7	2	3	6
8	4	3	6

续表

步号	x	y	pre
9	4	4	8
10	4	5	9
11	3	5	10
12	5	5	10

结构定义:

```
struct                                /* 定义结构体 */
{
```

```
    int x,y,pre;
}queue[MAXSIZE];
```

算法实现与分析:

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

```
#define m 5                          /* 定义常量 */
```

```
#define n 5
```

```
#define MAXSIZE 100
```

```
struct                                /* 定义队列结构体 */
```

```
{
    int x,y,pre;
}queue[MAXSIZE];
int mg[m+1][n+1];                    /* 定义全局变量 */
int zx[4],zy[4];
```

```
void printresult(int rear)            /* 输出路径 */
```

```
{
    int i=rear,j;
    do                                /* 反向搜索迷宫路径,将迷宫路径上所有点的 pre 值全部设为 -1 */
    {
        j=i;
        i=queue[i].pre;
        queue[j].pre=-1;
    } while (i!=0);
    printf("查找到的迷宫路径:\n入口 ->");
    i=0;
    while(i<MAXSIZE)                  /* 搜索 pre 为 -1 的点,正向输出迷宫路径 */
    {
        if (queue[i].pre== -1)
            printf("(%d,%d) ->",queue[i].x,queue[i].y);
        i++;
    }
    printf("出口\n");
}
```

```

void maze()                                /* 查找路径 */
{
    int i,j,x,y,v,front,rear,find=0;
    queue[1].x=1;queue[1].y=1;queue[1].pre=0;

    front=1;rear=1;                        /* 从(1,1)开始搜索,且该点入队列 */
    mg[1][1]=-1;                          /* 队列初始化 */
    while (front<=rear && ! find)          /* 标记搜索过的位置 */
    {                                       /* 队列不为空且未找到迷宫出口 */
        x=queue[front].x;y=queue[front].y;
        for (v=1;v<=4;v++)              /* 循环搜索四个方向,可行的方向入队列 */
        {
            i=x+zx[v];j=y+zy[v];        /* 选择一个前进方向 */
            if (mg[i][j]==0)             /* 该方向可行 */
            {
                rear++;
                queue[rear].x=i;
                queue[rear].y=j;
                queue[rear].pre=front;
                mg[i][j]=-1;
            }
            if (i==m && j==n)             /* 找到迷宫出口 */
            {
                printresult(rear);
                find=1;                  /* 退出循环 */
            }
        }
        front++;                          /* 从队列中删除一个元素 */
    }
    if (! find) printf("不存在路径! \n");
}

main()
{
    int i,j;
    for(i=1;i<=m;i++)                    /* 输入迷宫图 */
        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;        /* 定义迷宫边框值 */
    }
    for(j=0;j<=n+1;j++)

```

```

    }
    mg[0][j] = 1; mg[m+1][j] = 1;
    }
    zx[1] = -1; zx[2] = 0; zx[3] = 1; zx[4] = 0; /* 定义方向值 */
    zy[1] = 0; zy[2] = 1; zy[3] = 0; zy[4] = -1;
    maze(); /* 求解迷宫路径 */
}

```

运行测试:

< 输入 >

输入迷宫图(0-1 矩阵形式):

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

< 输出 >

查找到的迷宫路径为:

入口 → (1,1) → (2,1) → (3,1) → (3,2) → (3,3) → (4,3) → (4,4) → (4,5)
→ (5,5) → 出口

3.3 实训项目

项目内容:背包问题。

设计要求

假设有一个能装入总体积为 T 的背包和 n 件体积分别为 $w_1, w_2, \dots, w_n$ 的物品,能否从 n 件物品中挑选若干件恰好装满背包,即使 $w_i + w_j + \dots + w_k = T$,要求找出所有满足上述条件的解。例如:当 $T = 10$,各件物品的体积为 $\{1, 8, 4, 3, 5, 2\}$ 时,可找到下列 4 组解:

$$(1, 4, 3, 2), (1, 4, 5), (8, 2), (3, 5, 2)$$

项目提示

可利用回溯法的设计思想来解决背包问题。首先将物品排成一列,然后顺序选取物品装入背包,假设已选取了前 i 件物品之后背包还没有装满,则继续选取第 $i+1$ 件物品,若该件物品“太大”不能装入,则弃之而继续选取下一件,直至背包装满为止。但如果在剩余的物品中找不到合适的物品以填满背包,则说明“刚刚”装入背包的那件物品“不合适”,应将它取出“弃之一边”,继续再从它“之后”的物品中选取,如此重复,直至求得满足条件的解,或者无解。

由于回溯求解的规则是“后进先出”,显然要用到栈的相关算法来进行实训设计。

3.4 基本题

3.4.1 选择题

- 栈的特点是(),队列的特点是()。
A. 先进先出 B. 先进后出
- 一个栈的进栈序列是 a, b, c, d, e , 则栈的不可能的输出序列是()。
A. $edcba$ B. $decba$ C. $dceab$ D. $abcde$
- 若已知一个栈的进栈序列是 $1, 2, 3, \dots, n$, 其输出序列为 $P_1, P_2, P_3, \dots, P_n$, 若 $P_1 = n$, 则 $P_i (1 \leq i < n)$ 为()
A. i B. $n = i$ C. $n - i + 1$ D. 不确定
- 判定一个队列 QU (最多元素为 $MAXSIZE$) 为空的条件是()。
A. $QU \rightarrow rear - QU \rightarrow front == MAXSIZE$ B. $QU \rightarrow rear - QU \rightarrow front - 1 == MAXSIZE$
C. $QU \rightarrow front == QU \rightarrow rear$ D. $QU \rightarrow front == QU \rightarrow rear + 1$
- 循环顺序队列中是否可以插入下一个元素,()。
A. 与队首指针和队尾指针的值有关
B. 只与队尾指针的值有关, 与队首指针的值无关
C. 只与数组大小有关, 与队首指针和队尾指针的值无关
D. 与曾经进行过多少次插入操作有关

3.4.2 填空题

- 设有编号为 $1, 2, 3, 4$ 的四辆车, 顺序进入一个栈式结构的站台, 试写出这四辆车开出车站的所有可能的顺序 (每辆车入站后出站的时间未知): _____。
- 将下列各中缀表达式转换成后缀表达式。
 $(A * (B + C) + D) * E - F * G$ _____;
 $A * (B - D) + H / (D + E) - S / N * T$ _____;
 $(A - C) * (B + D) + (E - F) / (G + H)$ _____。
- 写出运算式 $3 + 4 / 25 * 8 - 6$ 转换成后缀表达式过程中操作数栈和运算符栈的变化情况:
_____。

4. 顺序栈的入栈操作

```
seqstack * push_seqstack(seqstack * s, datatype x)
{
    if (_____)
    {
        printf("上溢\n");
        return 0;
    }
}
```

```

    }
    else
    {
        s->top++;
        _____ = x;
        return s;          /* 入栈操作成功 */
    }
}

```

5. 链栈的出栈操作

```

int *pop_linkstack(linkstack *s, datatype y)
{
    linkstack *p;
    if(_____)
    {
        printf("下溢\n");
        return 0;
    }
    else
    {
        p = s;
        y = s->data;
        s = _____;
        free(p);
        return 1;
    }
}

```

6. 循环队列的出队操作

```

datatype delete_cyqueue(sequeue *cq, datatype y)
{
    if(_____)
    {
        printf("下溢\n");
        return NULL;
    }
    else
    {
        cq->front = _____;
        y = cq->data[cq->front];
        return y;
    }
}

```

3.5 习题解析

1. 简述栈的基本性质。

解 栈(Stack)又称堆栈,是一种特殊的线性表,它限制线性表中元素的插入和删除,使其操作只能在线性表的一端进行。允许插入和删除的一端为变化的一端,称为栈顶(top),栈顶的第一个元素称为栈顶元素,栈的另一端称为栈底(bottom)。

向一个栈插入新元素又称为进栈或入栈,它是把该元素放到栈顶元素的上面,使之成为新的栈顶元素。从一个栈删除一个元素又称为出栈或退栈,它是把栈顶元素删除掉,使其下面的相邻元素成为新的栈顶元素。也就是说,最先放入栈的元素在栈底,最后放入的元素在栈顶,而删除元素则相反,最后放入的元素最先被删除,最先放入的元素最后被删除。所以又把栈称为后进先出表(Last In First Out),简称为LIFO表。

2. 简述栈和队列的相同点和不同点。

解 栈和队列都是一种特殊的线性表,但栈限制线性表中元素的插入和删除只能在线性表的一端进行,称为栈顶;而队列只能在线性表的一端进行元素的插入,在线性表的另一端进行元素的删除,允许插入的一端称为队尾(rear),允许删除的一端称为队首(front)。另外,栈最后放入的元素最先被删除,最先放入的元素最后被删除。所以又把栈称为后进先出表(Last In First Out),简称为LIFO表。而队列最先入队的元素将最先出队,最后入队的元素将最后出队。因此,又把队列称为先进先出表(First In First Out,简称FIFO)。栈和队列的存储结构都有顺序存储结构和链式存储结构。

3. 什么是队列的上溢? 怎样解决这个问题?

解 队列若采用顺序结构,一般易发生上溢。设队首指针为front,队尾指针为rear,队列的容量为MAXSIZE。当一个顺序结构的队列,在经过频繁的入队和出队操作以后,若 $rear = MAXSIZE$,即队尾指针已到达顺序表的顶部,此刻再要入队已不可能,称为上溢;但并不等于顺序表已无空间,如果队首指针前面有空闲空间时,这种情况又称为假溢出。

解决上溢或假溢出问题的方法有多种:其一,可建立一个足够大的存储空间以避免上溢,但会造成空间利用效率下降;其二,将队头指针固定在队列低端,入队时队尾指针向高端移动,但只要有出队,就将整个队列向低端移动一个元素位置,但这样会造成算法较为复杂;其三,将顺序表首尾相衔,构成循环结构,称为循环队列。在循环队列中,当 $rear = front$ 时,规定是循环队列空的标志; $(rear + 1) \% MAXSIZE = front$ 是循环队列满的标志,入队时 $rear = (rear + 1) \% MAXSIZE$,出队时 $front = (front + 1) \% MAXSIZE$,这种方法既节省存储空间,算法亦较简单。

4. 对于下面的每一步画出栈中元素及栈顶指针示意图:

- (1) 空栈。
- (2) 元素 a, b, c 进栈。
- (3) 元素 d, e, f 进栈。
- (4) 删除元素 f 和 e。
- (5) 元素 g 进栈。

解 见示意图 3.10(a)、(b)、(c)、(d)、(e)。

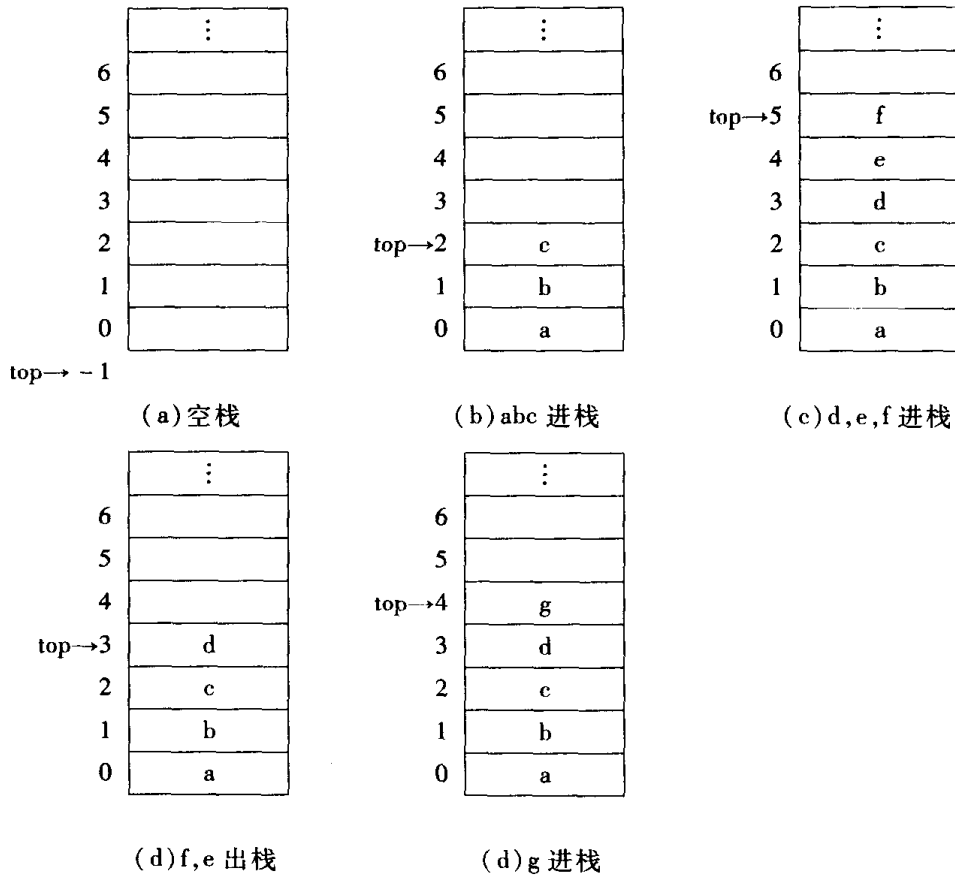


图 3.10 栈中元素及栈顶指针示意图

5. 将下列中缀表达式改写为后缀表达式:

- (1) $8 * 5 - 4$ (2) $2 + 4 / (6 - 4)$ (3) $(2 + a) * (b * (b + c) + c)$

解 解这类题目,首先按四则运算规则分析中缀表达式中的各运算单元(由操作数和运算符组成)的运算次序,并在其下划线,再标上序号;然后,按序号逐个移动各运算单元中的运算符到该运算单元的后面。例如,(3)题 $(2 + a) * (b * (b + c) + c)$ 图示如下:

$$\begin{array}{c}
 \underline{(2 + a) * (b * (b + c) + c)} \\
 \text{①} \qquad \qquad \qquad \text{②} \\
 \qquad \qquad \qquad \underline{\qquad \qquad \text{③} \qquad \qquad} \\
 \qquad \qquad \qquad \underline{\qquad \qquad \text{④} \qquad \qquad} \\
 \qquad \qquad \qquad \underline{\qquad \qquad \text{⑤} \qquad \qquad}
 \end{array}$$

由此,有以下结果:

- (1) $8 \ 5 \ * \ 4 \ -$
 (2) $2 \ 4 \ 6 \ 4 \ - \ / \ +$
 (3) $2 \ a \ + \ b \ b \ c \ + \ * \ c \ + \ *$

6. 已知一个中缀表达式为 $5 - 2 * (20 - (3 + 4)) / 2$,试写出它的后缀算术表达式,并用图示表达转换过程中栈的变化情况。

解 后缀算术表达式为: $5 \ 2 \ 20 \ 3 \ 4 \ + \ - \ * \ 2 \ / \ -$ 。

转换过程中栈的变化情况如图 3.11 所示。

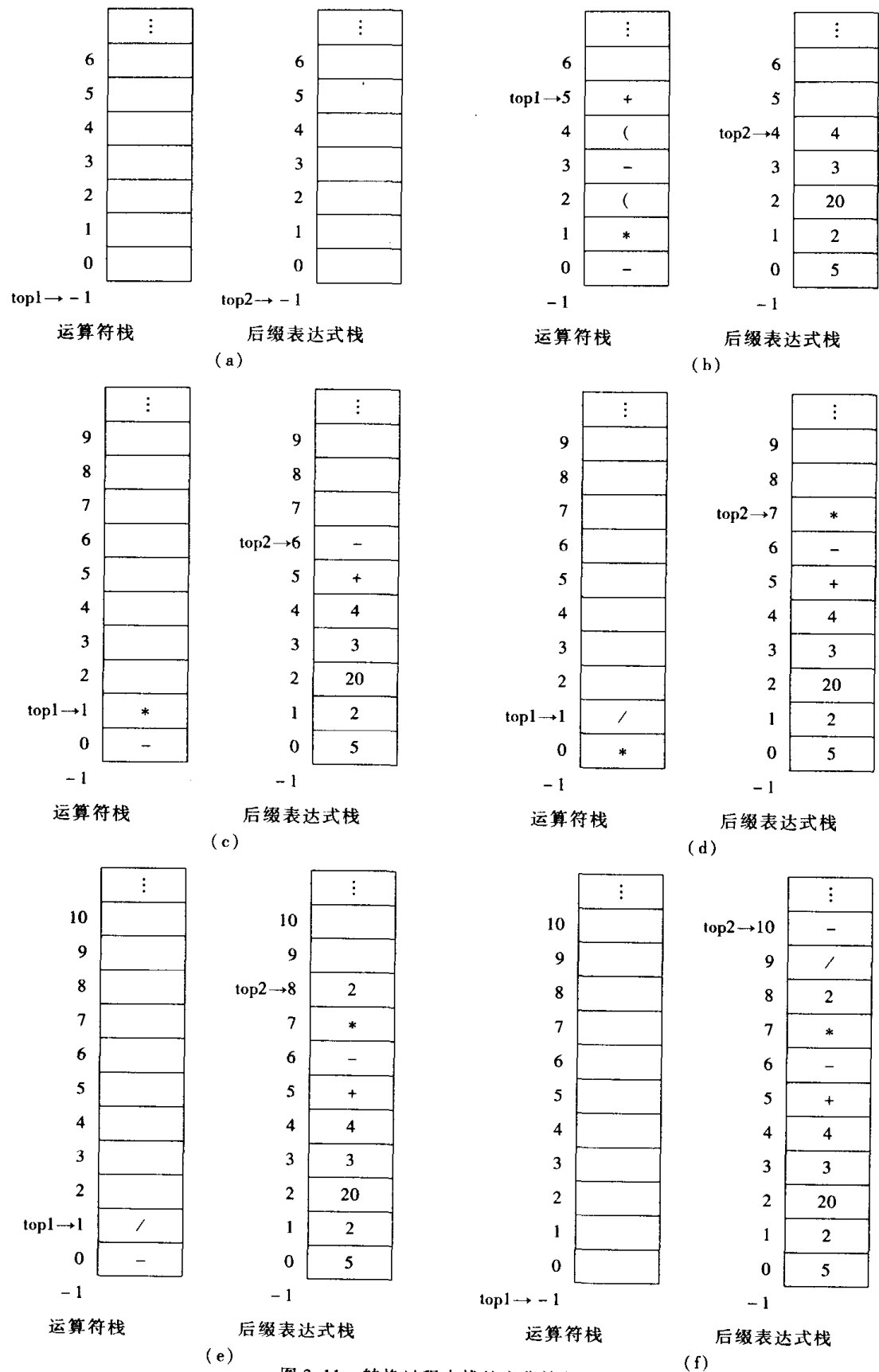


图 3.11 转换过程中栈的变化情况示意图

对图 3.11 的几点说明:

(1) 设置两个栈, 运算符栈和后缀表达式栈。运算符栈暂存转换过程中的运算符, 后缀表达式栈存放转换过程中的后缀表达式。两个栈的数据结构仅为示意(因为后缀表达式栈存放了两种不同的数据类型)。

(2) 为了减少作图, 在转换过程中, 当运算符或操作数连续进栈时, 将几步合并到一个图上; 当运算符连续出栈时, 也将几步合并到一个图上。

7. 设循环队列的容量为 20, 序号从 0 到 19, 经过一系列入队与出队运算后, 有下列情形:

(1) $\text{front} = 5, \text{rear} = 10$

(2) $\text{front} = 10, \text{rear} = 5$

问在这两种情况下, 循环队列中各有多少个元素?

解 设循环队列头指针为 front , 尾指针为 rear , 容量为 MAXLEN , 则循环队列元素个数可按公式: $(\text{rear} - \text{front} + \text{MAXLEN}) \% \text{MAXLEN}$ 求得。

由此, (1) 的答案为 5; (2) 的答案为 15。

8. 设有一个具有 n 个单元的循环队列, 头指针为 front , 尾指针为 rear , 试写出一个算法, 求队列中元素的个数。

解 本题可用 7 题求循环队列元素个数的公式 $(\text{rear} - \text{front} + \text{MAXLEN}) \% \text{MAXLEN}$, 在已知循环队列头指针 front , 尾指针 rear 和容量 MAXLEN 值后可方便求得。但为了对循环队列工作机制有更深入的了解, 我们给出以下算法。

算法由主函数和统计循环队列元素个数的两个函数组成。此程序可以用来检验 7 题的答案。

程序如下:

```
#include <stdio.h>
#define MAXLEN 20
int countcq(int s[], int fp, int rp)      /* 统计循环队列元素的个数 */
{
    int count = 0, front, rear;
    front = fp;
    rear = rp;
    while(front != rear)                  /* 循环队列不空, 继续统计队列元素 */
    {
        front = (front + 1) % MAXLEN;
        count = count + 1;                /* 统计计数器加 1 */
    }
    return(count);
}
main()
{
    int a[MAXLEN] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20};
```

```
int count,front,rear;  
front = 10; /* 可以在 0 到 19 之间修改 front 的值 */  
rear = 5; /* 可以在 0 到 19 之间修改 rear 的值 */  
count = countcq(a,front,rear);  
printf("%d\n",count);
```

输出结果为:

15

9. 试设计用带头结点的线性双向循环链表存储结构表示队列的入队和出队的算法。

解 算法由主函数和创建空双向循环链队列、入队、出队、输出队列元素五个函数组成。

本算法中双向循环链队列只有一个 point 指针,它始终指向附加表头结点。以下为表结点结构(如图 3.12 所示),只有一个表头结点的空双向循环链队列(如图 3.13 所示)和双向循环链队列(如图 3.14 所示)示意图。

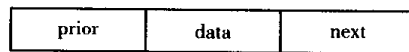


图 3.12 表结点结构

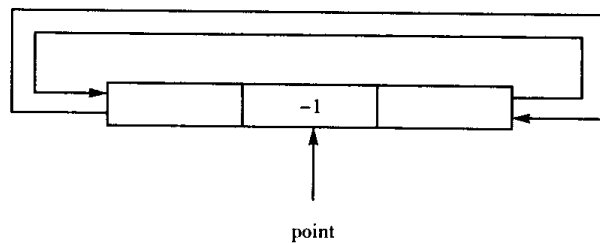


图 3.13 只有一个表头结点的空双向循环链队列

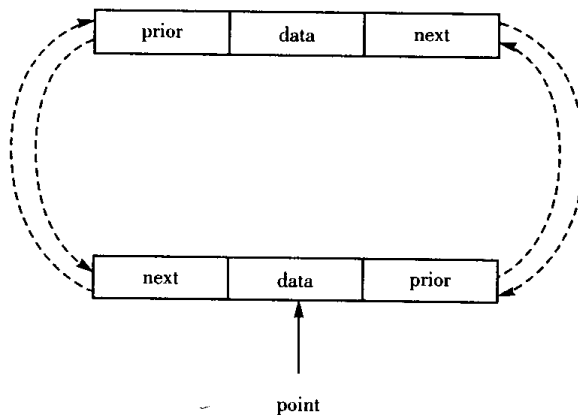


图 3.14 双向循环链队列示意图

程序如下:

```
#include <stdio.h>
typedef struct dupnode
{
    int data;
    struct dupnode *next, *prior;      /* 分别为后继、前驱结点指针 */
} DUPNODE;
DUPNODE *create()                    /* 此函数创建只有一个表头结点的空双向循环链队列 */
{
    DUPNODE *point;
    point = (DUPNODE *)malloc(sizeof(DUPNODE));    /* 申请新的存储空间 */
    point->data = -1;
    point->next = point;
    point->prior = point;
    return(point);
}
void inqueue(int x, DUPNODE *point)    /* 入队, 总是在表头结点的右面插入 */
{
    DUPNODE *p;
    p = (DUPNODE *)malloc(sizeof(DUPNODE));
    p->data = x;
    p->next = point;
    p->prior = point->prior;
    point->prior = p;
    (p->prior)->next = p;
}
void delqueue(int *y, DUPNODE *point) /* 出队, 总是在表头结点的左面删除 */
{
    DUPNODE *q;
    if((point->next) != point)
    {
        q = point->next;
        point->next = q->next;
        (q->next)->prior = point;
        *y = q->data;
        free(q);
    }
}
int print(DUPNODE *point)              /* 输出队列元素 */
{
    DUPNODE *p;
    if((point->next) == point)
```

```

    {
        printf("empty! \n");
        return(0);
    }
    else
    {
        p=point->next;
        while(p!=point)
        {
            printf("%d ",p->data);          /* 输出双向循环链队列元素 */
            p=p->next;
        }
        printf("\n");
        return(1);
    }
}

main()
{
    DUPNODE *point;
    int i,j,k,a,b;
    point=create();                      /* 创建空双向循环链队列 */
    printf("\ninput number of inqueue,please:"); /* 输入准备入队的元素数目到j */
    scanf("%d",&j);
    for(i=1;i<=j;i++)                    /* 将j个元素依次入队 */
    {
        printf("%s%d s","input data ",i," ");
        scanf("%d",&a);
        inqueue(a,point);
    }
    printf("output a queue:");
    print(point);                        /* 输出队列元素 */
    printf("input number of delqueue,please:"); /* 输入准备出队的元素数目到k */
    scanf("%d",&k);
    for(i=1;i<=k;i++)                    /* 将k个元素依次出队 */
        delqueue(b,point);
    printf("output a queue:");
    print(point);                        /* 输出队列元素 */
}

```

输出结果为:

input number of inqueue,please: 5 (按回车)

input data 1:2 (按回车)

input data 2:4 (按回车)

input data 3:6 (按回车)

input data 4:8 (按回车)

input data 5:10(按回车)

output a queue: 2 4 6 8 10

input number of delqueue,please: 3 (按回车)

output a queue: 8 10

第4章

串

4.1 基础知识篇

学习要点:

- 串的基本概念和串的基本运算
- 串的顺序存储结构、链式存储结构
- 串的顺序存储中的连接、相等判断、取子串、插入、删除和子串定位等算法的实现

在计算机的各种应用中,非数值处理问题的应用越来越多。如在汇编程序和编译程序中,源程序和目标程序都是作为字符串数据进行处理的;在事务处理系统中,用户的姓名和地址及货物的名称和规格等也是字符串数据。

计算机在处理数据时,针对字符类数据,往往以字符串形式处理。而在实际应用中,字符串往往具有不同的特点,要实现字符串的处理,就必须根据具体情况使用合适的存储结构和操作。因此我们要研究字符串的存储方式和相关算法,用于解决实际问题。

4.1.1 串及其存储结构

1. 串的概念

串:是由零个或多个字符组成的有限序列。一般记为: $s = "a_1 a_2 \cdots a_n"$ ($n \geq 0$),其中, n 为串中字符的数目,即字符串的长度。

当 $n = 0$ 时,称为空串。为了明确表示,通常以两个相邻的双引号来表示空串 (Null String),如 $s = ""$ 。

仅由空格组成的串称为空格串,如 $s = "\Phi"$;若串中含有空格,在计算串长时,空格应计入串的长度中,如 $s = "I\Phi am\Phi a\Phi student."$ 的长度为 15。

请读者注意,在 C 语言中,用单引号引起来的单个字符与单个字符的串是不同的,如 $s1 = 'a'$ 与 $s2 = "a"$ 是不同的, $s1$ 表示字符,而 $s2$ 表示字符串。

子串:一个串的任意个连续字符组成的子序列称为该串的子串。

主串:包含子串的串称为主串。

位置:字符在序列中的序号称作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。当一个字符在串中多次出现时,以该字符第一

次在主串中出现的位置作为该字符在串中的位置。

两串相等:当且仅当两串的值相等,即两个字符串的长度相等,并且各个对应位置上的字符都相等时,才能称这两个串是相等的。

例1:A、B、C、D为4个字符串,A="china",B="chi",C="na",D="chi na"。

其中,A的长度为5,B的长度为3,C的长度为2,D的长度为6;且B和C都是A和D的子串,B在A和D中的位置都是1,而C在A中的位置是4,在D中的位置是5。

例2:s1、s2、s3为如下的三个串:s1="I'm a student.",s2="student",s3="teacher"。

它们的长度分别为14、7、7;s2是s1的子串,子串s2在s1中的位置为7,也可以说s1是s2的主串;s3不是s1的子串,串s2和s3不相等。

串的逻辑结构与线性表极为相似,它是一种特殊的线性表,其特殊性是指串中的每一个数据元素仅由一个字符组成。

2. 串的存储结构

尽管串的逻辑结构与线性表极为相似,但是串的基本操作和线性表还是有很大差别的。在线性表的基本操作中大多以单个元素作为操作对象,而在串的基本操作中,通常以串的整体作为操作对象。

当然,存储串的方法也就是存储线性表的一般方法,因而串也采用顺序和链式两种存储结构。但由于组成串的结点是单个字符,所以存储时有一些特殊的技巧。

(1) 顺序存储

串的顺序存储结构称为顺序串,类似于线性表的顺序存储结构,即用一组地址连续的存储单元存储串的字符序列。如图4.1所示。

d	a	t	a	Φ	s	t	r	u	c	t	u	r	e
---	---	---	---	---	---	---	---	---	---	---	---	---	---

图4.1 串的顺序存储结构

由于一个字符只占1个字节,而现在大多数计算机的存储器地址都采用字编址,一个字(即一个存储单元)占多个字节,因此顺序存储结构方式有两种。

紧缩格式:一个字节存储一个字符。这种存储方式可以在一个存储单元中存放多个字符,充分地利用了存储空间。但在串的操作运算中,若要分离某一部分字符,则变得非常麻烦。如图4.2所示为以4个字节为一个存储单元的存储结构,每个存储单元可以存放4个字符。对于给定的串s =

d	a	t	a
Φ	s	t	r
u	c	t	u
r	e	\0	

图4.2 紧缩格式的存储结构

"dataΦstructure",在C语言中采用字符'\0'作为串值的结束符,串s的串值连同结束符的长度共15,只需4个存储单元。

非紧缩格式:这种方式是以一个存储单元为单位,每个存储单元仅存放一个字符。这种存储方式的空间利用率较低,例如一个存储单元有4个字节,则空间利用率仅为25%。但这种存储方式中不需要分离字符,因而程序处理字符的速度较快。如图4.3所示。

用字符数组存放字符串时,数据类型说明如下:

```
typedef struct
{
    char ch[MAXSIZE];
    int curlen;          /* 字符串的长度 */
} seqstring;
```

由上述讨论可知,串的顺序存储结构有两大不足之处:一是需要预定义串的最大长度,这在程序运行前是很难估计的。二是由于定义了串的最大长度,使得串的某些操作受限,如串的连接运算等。因此我们需要另一种存储方式——链式存储。

(2) 链式存储

串的各种运算与串的存储结构有着很大的关系,在随机取子串时,顺序存储方式操作起来比较方便。而在对串进行插入、删除等操作时,采用串的链式存储方式则更简单。

串的链式存储结构称为链串,一个链串通常由头指针惟一确定,串的链式存储结构中每个结点包含字符域和指针域,字符域用于存放字符,指针域用于存放指向下一个结点的指针。因此,串可用单链表表示。

用单链表存放字符串时,其数据类型说明如下:

```
typedef struct node
{
    char data;          /* data 域用来存储组成字符串的字符 */
    struct node * next; /* next 域用来指向下一个结点 */
} linkstring;
```

为了提高空间的利用率,我们可以使每个结点存放多个字符,称为块链结构。如图 4.4 所示。

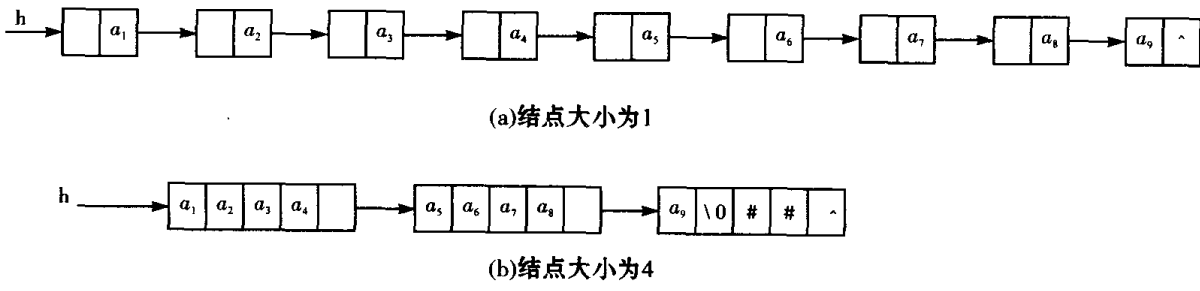


图 4.4 串的块链存储结构

总的来说,串的链式存储结构的优点是插入、删除运算很方便;缺点是数据域只占一个字节,而指针域可能要占多个字节,所以空间利用率很低。

d			
a			
t			
a			
Φ			
s			
t			
r			
u			
c			
t			
u			
r			
e			
\0			

图 4.3 非紧缩格式的存储结构

4.1.2 串的基本运算

串的基本运算有赋值、连接、求串长、求子串、求子串在主串中出现的位置、判断两个串是否相等、删除子串等。我们以 C 语言的库函数表示其中的一些运算,也运用自定义函数来进行说明。

1. 串长度

`strlen(str)`:求字符串的长度。统计字符串 `str` 中字符的个数(不包括'\0'),若 `str` 非空,则返回字符的个数,若 `str` 为空串,则返回 0。

库函数和形参说明如下:

`unsigned int strlen(char *str)`

自定义函数为:

`LenString(s)`:返回串 `s` 的元素个数。针对一已知串,求其元素个数即串长度。

```
int LenString(seqstring s)
{
    return(s.curlen);
}
```

2. 串比较

`strcmp(str1, str2)`:字符串的比较。比较两个字符串 `str1`、`str2`。若 `str1 < str2`,则返回负数;若 `str1 > str2`,则返回正数;若 `str1 = str2`,则返回 0。

库函数和形参说明如下:

`int strcmp(char *str1, char *str2)`

自定义函数为:

`CompareString(s, t)`:串 `s` 和串 `t` 若相等,返回 1,否则返回 0。

```
int CompareString(seqstring s, seqstring t)
{
    int i;
    if(s.curlen != t.curlen)
        return(0);
    else
    {
        for(i=0; i<s.curlen; i++)
            if(s.ch[i] != t.ch[i])
                return(0);
        return(1);
    }
}
```

3. 串插入

自定义函数 `InsertString(s, pos, t)`:将串 `t` 插入到串 `s` 的 `pos` 位置处,并返回串 `s`。

注:字符串 `s` 的数组下标从 0 开始计数;位置序号 `pos` 从 1 开始计数。本章其他算法

设定相同,不再说明。

```
seqstring * InsertString (seqstring * s, int pos, seqstring t)
{
    int i;
    for (i = s->curlen - 1; i >= pos - 1; i--)
        s->ch[i + t	curlen] = s->ch[i];
    for (i = 0; i < t	curlen; i++)
        s->ch[i + pos - 1] = t.ch[i];
    s->curlen += t	curlen;
    return s;
}
```

4. 串删除

自定义函数 DeleteString(s, pos, len): 从串 s 的 pos 位置开始删除 len 长的子串, 并返回串 s。

```
seqstring * DeleteString(seqstring * s, int pos, int len)
{
    int i;
    for (i = pos + len - 1; i < s->curlen; i++)
        s->ch[i - len] = s->ch[i];
    s->curlen = s->curlen - len;
    return s;
}
```

5. 串连接

strcat(str1, str2): 字符串的连接。把字符串 str2 接到 str1 后面, 并删除 str1 的结尾符 '\0', 最后返回串 str1。

库函数和形参说明如下:

```
char * strcat(char * str1, char * str2)
```

自定义函数为:

ConcatString(s, t): 将串 t 连接到串 s 后面形成新串存放到串 s 中, 并返回串 s。

```
seqstring * ConcatString(seqstring * s, seqstring * t)
{
    int i;
    for (i = 0; i < t->curlen; i++)
        s->ch[i + s->curlen] = t->ch[i];
    s->curlen = s->curlen + t->curlen;
    return s;
}
```

6. 串拷贝

strcpy(str1, str2): 字符串的拷贝(赋值)。把 str2 指向的字符串拷贝到 str1 中, 返回 str1。

库函数和形参说明如下:

```
char * strcpy(char * str1,char * str2)
```

自定义函数如下:

```
seqstring * CopyString(seqstring * t,seqstring * s)
```

/* 将 s 指向的字符串拷贝到 t 指向的字符串中,返回 t */

```
{
    int i;
    for (i=0;i<s->curlen;i++)
        t->ch[i]=s->ch[i];
    t->curlen=s->curlen;
    return t;
}
```

7. 串替换

自定义函数 Replace(s,t,v):将串 s 中的子串 t 全部替换成串 v。

```
int Replace(seqstring * s, seqstring t,seqstring v)
```

```
{
    int i,j,k;
    i = SearchSub(*s,t);
    if(i<0)
    {
        printf("error");
        return (0);
    }
    while (i>=0)
    {
        s = DeleteString(s,i+1,t	curlen);
        s = InsertString(s,i+1,v);
        i = SearchSub(*s,t)
    }
    return (1);
}
```

8. 求子串

自定义函数 SubString(s,pos,len):求串 s 中第 pos 个字符起,长度为 len 的子串,并返回该子串。

```
seqstring * SubString (seqstring s,int pos,int len)
```

```
{
    seqstring * t;
    int i;
    for (i=0;i<len;i++)
        t->ch[i]=s.ch[pos+i-1];
    t->curlen=len;
}
```

```
return t;
```

9. 子串查找

`strstr(str1, str2)`: 子串的查找。找出子串 `str2` 在主串 `str1` 中第一次出现的位置(不包括子串 `str2` 的结尾符), 返回该位置的指针, 若找不到, 返回空指针 `NULL`。

库函数和形参说明如下:

```
char * strstr(char * str1, char * str2)
```

自定义函数如下:

```
int SearchSub(seqstring s, seqstring t)
{
    int i=0, j=0;
    while(i<s	curlen && j<t	curlen)
    {
        if (s.ch[i] == t.ch[j])
        { i++; j++; }
        else
        { i=i-j+1; j=0; }
    }
    if(j == t	curlen);
    return(i-t	curlen);
    else return(-1);
}
```

4.2 实训篇

实训目的:

- 掌握串的常用存储结构
- 验证串的基本操作
- 理解项目实例中字符串的相关算法

案例描述

设计程序, 实现文本编辑中内容的插入和删除操作。

设计分析

文本编辑工具是一个面向用户的系统服务程序, 广泛用于各种文稿的编辑和加工中。我们把一篇文稿看成是一个有限字符序列, 称之为文本。文本编辑软件是把用户的所有文本内容看作一个字符串, 进而进行一系列的操作。

设计方案

把输入的文本看成一个字符串, 并采用顺序存储结构。参考 4.1.2 节中串的基本操作, 设计关于文本插入和删除的一系列程序。

结构定义

```
typedef struct string          /* 定义结构体 */
{
    char ch[LEN];
    int len;
} STRING;
```

算法实现与分析

(1) 文本插入

```
#define LEN 10000            /* 定义常量 */
typedef struct string        /* 定义结构体 */
{
    char ch[LEN];
    int len;
} STRING;

void insert (STRING s,STRING s1,int i)
{
    int j;
    for(j=s.len-1;j>=i-1;j--) /* 空出插入位置 */
        s.ch[j+s1.len]=s.ch[j];
    for(j=0;j<s1.len;j++)
        s.ch[j+i-1]=s1.ch[j]; /* 插入文本 */
    s.len=s.len+s1.len;
    printf("\n 插入后文本为% s\n",s.ch); /* 输出文本及其长度 */
    printf("插入后文本长度为% d\n",s.len);
}

main()
{
    STRING a={"A mouse is a pointing device that lets you select and move items
              on your screen.",80},a1={"hand-held",11}; /* 实例说明 */
    int i=13; /* 插入位置 */
    insert(a,a1,i); /* 调用插入函数 */
}
```

(2) 文本删除

```
#define LEN 10000            /* 定义常量 */
typedef struct string        /* 定义结构体 */
{
    char ch[LEN];
    int len;
} STRING;

void delete(STRING s,int i,int j)
{

```

```

    int k;
    if(s.len - i < j)
/* 若从文本 s 的删除位置到文本 s 的最后一个字符的长度小于 j,则直接修改文本的长度即可 */
    {s.len = i; s.ch[s.len] = '\0';}
    else
    {
        for(k = i + j - 1; k < s.len; k++)    /* 删除字符 */
            s.ch[k - j] = s.ch[k];
        s.len = s.len - j;
        s.ch[s.len] = '\0';
        printf("\n 删除后文本为% s\n 文本长度为% d", s.ch, s.len);    /* 输出结果 */
    }
}

main()
{
    STRING s = {"A mouse is a hand-held pointing device that lets you select and
                move items on your screen.", 90};    /* 实例说明 */
    int i = 13, j = 11;    /* 赋值要删除的起始点和删除个数 */
    delete(s, i, j);    /* 调用函数 */
}

```

运行测试

(1) 文本插入

< 输出 >

插入后文本为 A mouse is a hand-held pointing device that lets you select and move items on your screen.

文本长度为 91

(2) 文本删除

< 输出 >

删除后文本为 A mouse is a pointing device that lets you select and move items on your screen.

文本长度为 80

4.3 实训项目

项目内容

试设计一算法,测试一个串 T 的值是否为回文(即从左向右读出的内容与从右向左读出的内容一样)。

设计要求

写出的 C 语言程序应包含规范的串输入以及输出结果。

项目提示

对于一个给定的串 $T = "a_1, a_2 \dots"$, 要判断 T 串是否为回文只需判断当 $i = 0, 1, 2 \dots n/2$ 时, 等差序列 $a_i = a_n - i + 1$ 是否成立即可, 若成立则为回文。

4.4 基本题

4.4.1 选择题

- 串是一种特殊的线性表, 其特殊性体现在_____。
A. 可以顺序存储 B. 数据元素是一个字符
C. 可以链式存储 D. 数据元素可以是多个字符
- 有串 $S1 = "ABCDEFGH"$, $S2 = "PQRST"$, 假设函数 $con(x, y)$ 返回 x 和 y 串的连接串, $subs(s, i, j)$ 返回串 s 的从数组序号为 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 $con(subs(s1, 2, len(s2)), subs(s1, len(s2), 2))$ 的结果串是_____。
A. BCDEF B. BCDEFG C. BCPQRST D. CDEFGFG
- 若串 $s = "software"$, 其子串的个数是_____。
A. 8 B. 37 C. 36 D. 9

4.4.2 填空题

- 串的两种最基本的存储方式是_____、_____。
- 空串是_____, 其长度等于_____。
- 已知两个串: $s1 = "fg\Phi cdb\Phi cabcad"$, $s2 = "abc"$, 试求两个串的长度: $length(s1) =$ _____, $length(s2) =$ _____; 判断串 $s2$ 是否是串 $s1$ 的子串, 是或否? _____; 并指出串 $s2$ 在串 $s1$ 中的位置: _____。
- 已知: $s1 = "I'm a student."$, $s2 = "student"$, $s3 = "teacher"$, 试求下列各运算的结果:
 $strstr(s1, s2):$ _____;
 $strlen(s1):$ _____;
 $strcat(s2, s3):$ _____;
 $DeleteString(s1, 4, 10):$ _____。

5. 串连接

```
seqstring * ConcatString(seqstring * s, seqstring * t)
```

```
{
    int i;
    for(i=0; i<t->curlen; i++)
        s->ch[i+s->curlen] = t->ch[i];
    s->curlen = _____;
    return s;
}
```

6. 串拷贝

```
seqstring * CopyString(seqstring * t, seqstring * s)
{
    int i;
    for (i = 0; i < s->curlen; i++)
        t->ch[i] = _____;
    t->curlen = s->curlen;
    return t;
}
```

7. 求子串

```
seqstring * SubString(seqstring s, int pos, int len)
{
    seqstring * t;
    int i;
    for (i = 0; i < len; i++)
        t->ch[i] = _____;
    t->curlen = len;
    return t;
}
```

4.5 习题解析

1. 简述下列串的概念:

串、串的长度、空串、空格串、子串和两串相等。

解 串:由有限个字符组成的线性序列,记作: $s = "a_1 a_2 a_3 \dots a_n"$ ($n \geq 0$),其中 s 是串的名字,用双引号括起来的字符序列是串的值。 a_i ($1 \leq i \leq n$) 可以是字母、数字或其他字符。

串的长度:串中字符的数目称为串的长度。

空串:指不含任何字符的串,它的长度为 0。

空格串:仅由空格组成的串称为空格串,它的长度为空格符号的个数。一般用“ Φ ”表示,如 $s = "\Phi\Phi\Phi"$,则 s 串长度为 $n = 3$ 。

子串:串中任意个连续字符构成的序列称为该串的子串,而包含该子串的串称为主串。

两串相等:只有当两个串的长度相等,并且各对应位置上的字符都相同时,两个串才相等。

2. 简述串有哪几种存储结构。

解 串一般有两种存储结构:它们是串的静态存储结构和串的动态存储结构。串的静态存储结构,即串的顺序存储结构,是将串定义成字符型数组,由串名可以直接访问到串值。串的静态存储结构又根据计算机的一个机器字(一般由多个字节组成)存放一个

字符或多个字符,细分为串的非紧缩存储结构和串的紧缩存储结构。串的动态存储结构,即串的链式存储结构也称为链串,结构与链表类似,链串中每个结点有两种域,一种是数据域(data)(一个或多个值域),用于存放字符串中的字符(一个或多个字符),另一种是指针域(next),用于存放后继结点的地址。链串的特点是链表中的结点数据只能为字符型。

此外,还有串的索引存储结构,其构造方法是:首先开辟一块地址连续的存储空间(又称为堆),用于存放各串本身的值。另外再建立一个索引表,在索引表的项目中存放串的名字、长度和在存储空间中的起始地址。

3. 已知顺序串 $s = \text{"abcd"}$,写出它的所有子串,并设计算法。

解 $s = \text{"abcd"}$ 的所有子串为: "", "a", "b", "c", "d", "ab", "bc", "cd", "abc", "bcd", "abcd"。

算法由主函数和输出所有子串两个函数组成。本算法不输出空串。

程序如下:

```
#define MAXLEN 25
typedef struct string
{ char ch[MAXLEN];
  int len;
} STRING;

void outsubstr(s)                                /* 输出所有子串 */
STRING s;
{
  int i,j,k;
  for(i=1;i<=s.len;i++) /* 确定 s.len 次循环,代表本次循环可以输出的子串位数 */
  { printf("output %d char:",i);
    for(j=1;j<=s.len-i+1;j++) /* 每次循环能输出的子串个数 */
    {
      for(k=1;k<=i;k++) /* 每次循环输出一个子串 */
      {
        printf("%c",s.ch[j+k-2]);
      }
      printf(" ");
    }
    printf("\n");
  }
  printf("\n");
}

main()
{
  STRING a = {"abcd",4}; /* 可以在这里修改串和长度,以得到不同的结果 */
  outsubstr(a);
}
```

输出结果为:

output 1 char: a b c d

output 2 char: ab bc cd

output 3 char: abc bcd

output 4 char: abcd

4. 已知顺序串 s, 编写一算法, 将 s 串中的所有 x 字符都删除。

解 算法由主函数和搜索字符并删除字符、删除串中第 i 个字符三个函数组成。

程序如下:

```
#define MAXLEN 25

typedef struct string
{
    char ch[MAXLEN];
    int len;
} STRING;

STRING delete(s,i)                                /* 删除 s 串中第 i 个字符 */
STRING s;
int i;
{
    int k;
    if((i<1) || (i>s.len))
    {
        printf("error\n");                        /* 位置值非法,删除无效 */
        return;
    }
    else
    {
        /* 将下标 i(即第 i+1 个)字符到下标 s.len(第 s.len+1 个)字符(为'\0')依次前移一个字符 */
        for(k=i;k<=s.len;k++)
            s.ch[k-1]=s.ch[k];
        s.len=s.len-1;                               /* 串长度减 1 */
        return(s);
    }
}

STRING searchanddelete(s,c)                        /* 搜索字符 c,并删除字符 c */
STRING s;
char c;
{
    int i=0;
    while(i<s.len)                                  /* 从串的下标 0 到下标 s.len-1 搜索字母 c */
    {
        if(s.ch[i]==c)
        {
```

```

        s = delete(s, i + 1);
        i--;

    }
    i++;

    /* 继续搜索下一个字符 */

    return(s);
}

main()
{
    STRING s = {"xxxxdefxhijxxsty", 16};
    printf("\n% s % d \n", s.ch, s.len);
    s = searchanddelete(s, 'x');    /* 将 s 串中的所有 x 字符都删除 */
    printf("\n% s % d \n", s.ch, s.len);
}

```

输出结果为:

xxxxdefxhijxxsty 16

defhijsty 9

5. 已知顺序串 s, 编写一算法, 将 s 串中的所有 x 字符都用 y 字符替换。

解 算法由主函数和替换字符两个函数组成。

程序如下:

```

#define MAXLEN 25
typedef struct string
{ char ch[MAXLEN];
  int len;
} STRING;

STRING replacechar(s, c1, c2)    /* 替换字符 */
STRING s;
char c1, c2;
{
    int i;
    for(i = 0; i < s.len; i++)    /* 搜索 s 串, 找到字符 c1, 就用字符 c2 替换 */
    {
        if(s.ch[i] == c1)
            s.ch[i] = c2;
    }
    return(s);
}

main()
{
}

```

```

    STRING s = {"abcxdefxhijxxsty",16};
    printf("\n% s\n",s.ch);
    s = replacechar(s,'x','y');          /* 将 s 中的所有 x 字符都用 y 字符替换 */
    printf("\n% s\n",s.ch);
}

```

输出结果为:

```
abcxdefxhijxxsty
```

```
abcydefyhijyysty
```

6. 已知顺序串 s, 编写一算法, 统计 s 串中字符 a 出现的次数。

解 算法由主函数和统计 s 串中某字符出现的次数两个函数组成。

程序如下:

```

#define MAXLEN 25
typedef struct string
{ char ch[MAXLEN];
  int len;
} STRING;
int countchar(s,c)          /* 统计 s 串中字符 c 出现的次数 */
STRING s;
char c;
{
    int n=0,i;
    for(i=0;i<s.len;i++)
    {
        if(s.ch[i]==c)
            n++;
    }
    return(n);
}
main()
{
    STRING s = {"abaxdafxhajxxsta",16};
    int n;
    n = countchar(s,'a');
    printf("num = % d\n",n);
}

```

输出结果为:

```
num = 5
```

7. 设有一篇英文短文, 每个单词之间是用空格分开的, 编写一算法, 按照空格数统计短文中单词的个数。

解 算法由主函数和统计文中单词个数两个函数组成。

统计文中单词个数的算法分析如下: 要统计单词的个数先要解决如何判别一个单词,

应该从输入行的开头一个字符一个字符地去辨别。假定把一个文本行放在数组 `s` 中,那么就相当于从 `s[0]` 开始逐个检查数组元素,经过若干个空格符之后找到的第一个字母就是一个单词的开头,此时利用一个统计计数器 `num` 进行累加 1 运算,在此之后若连续读到的是非空格字符,则这些字符属于刚统计到的那个单词,因此不应将计数器 `num` 累加 1,下一次计数应该是在读到一个或几个空格后再遇到非空格字符之时进行。因此,统计一个单词时不仅要满足当前所检查的这个字符是非空格,而且要满足所检查的前一个字符是空格。

程序如下:

```
#define MAXLEN 50
typedef struct string
{
    char ch[MAXLEN];
    int len;
} STRING;

int countwords(s)
STRING s;
{
    char prec = ' ';          /* 前一个字符 */
    char nowc;                 /* 当前字符 */
    int num = 0;
    int i;
    for(i = 0; i < s.len; i++)
    {
        nowc = s.ch[i];
        if((nowc != ' ') && (prec == ' '))    /* ' ' 中间有一个空格 */
            num++;
        prec = nowc;
    }
    return num;
}

main()
{
    STRING s = {"ab c x def x h i j xxs ty ", 32};
    int num;
    num = countwords(s);
    printf("\nwords = %d\n", num);
}
```

输出结果为:

words = 9

第 5 章

树

5.1 基础知识篇

学习要点:

- 树的基本概念和树的基本操作
- 二叉树的定义和性质
- 二叉树的顺序存储结构和链式存储结构
- 二叉树的建立和三种遍历算法
- 二叉树的应用:哈夫曼树、哈夫曼编码

5.1.1 树的基本概念

树是一种常用的非线性结构,它是 $n(n \geq 0)$ 个结点的有限集合。若 $n = 0$,则称为空树;否则,有且仅有一个特定的结点被称为根。当 $n > 1$ 时,其余结点被分成 $m(m \geq 0)$ 个互不相交的子集 $T_1, T_2, \dots, T_m$,每个子集又是一棵树。由此可以看出,树是递归定义的。

5.1.2 二叉树

1. 二叉树的定义

定义:二叉树是一种特殊的树形结构。它与树形结构的区别是:

- (1) 每个结点最多有两棵子树;
- (2) 子树有左右之分,其次序不能任意颠倒。

二叉树也可以用递归的形式定义。即:二叉树是 $n(n \geq 0)$ 个结点的有限集合。当 $n = 0$ 时,称为空二叉树;当 $n > 0$ 时,有且仅有一个结点为二叉树的根,其余结点被分成两个互不相交的子集,一个作为左子集,另一个作为右子集,每个子集又是一个二叉树。

2. 二叉树的性质

二叉树具有下列 5 个重要的性质:

【性质 1】 在二叉树的第 i 层上最多有 2^{i-1} 个结点($i \geq 1$)。

【性质 2】 深度为 K 的二叉树最多有 $2^K - 1$ 个结点($K \geq 1$)。

满二叉树:如果一棵深度为 K 的二叉树拥有 $2^K - 1$ 个结点,则称它为满二叉树。如图 5.1 所示的是一棵深度为 4 的满二叉树。

完全二叉树:有一棵深度为 h ,具有 n 个结点的二叉树,若将它与一棵同深度的满二

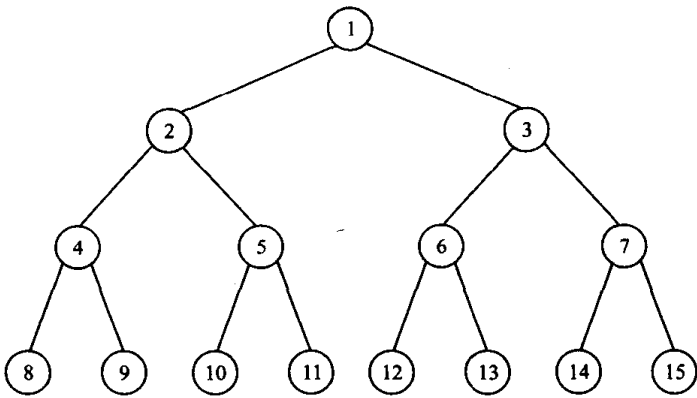


图 5.1 满二叉树

叉树中的所有结点按从上到下,从左到右的顺序分别进行编号,如果该二叉树中的每个结点分别与满二叉树中编号相同的结点的位置一一对应,则称这棵二叉树为完全二叉树。

【性质 3】 对于任意一棵二叉树,如果度为 0 的结点个数为 n_0 ,度为 2 的结点个数为 n_2 ,则 $n_0 = n_2 + 1$ 。

【性质 4】 具有 n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$ 。其中, $\lfloor \log_2 n \rfloor$ 的结果是不大于 $\log_2 n$ 的最大整数。

【性质 5】 对有 n 个结点的完全二叉树中的所有结点按从上到下、从左到右的顺序进行编号,则对任意一个结点 $i(1 \leq i \leq n)$,都有:

- (1) 如果 $i = 1$,则结点 i 是这棵完全二叉树的根,没有双亲;否则其双亲结点的编号为 $\lfloor i/2 \rfloor$;
- (2) 如果 $2i > n$,则结点 i 没有左孩子;否则其左孩子结点的编号为 $2i$;
- (3) 如果 $2i + 1 > n$,则结点 i 没有右孩子;否则其右孩子结点的编号为 $2i + 1$ 。

3. 二叉树的存储结构

二叉树也可以采用两种存储方式:顺序存储结构和链式存储结构。

(1) 顺序存储结构

这种存储结构适用于完全二叉树。其存储形式为:用一组连续的存储单元按照完全二叉树的结点编号的顺序来存放结点内容。图 5.2 是一棵二叉树及其相应的顺序存储结构。

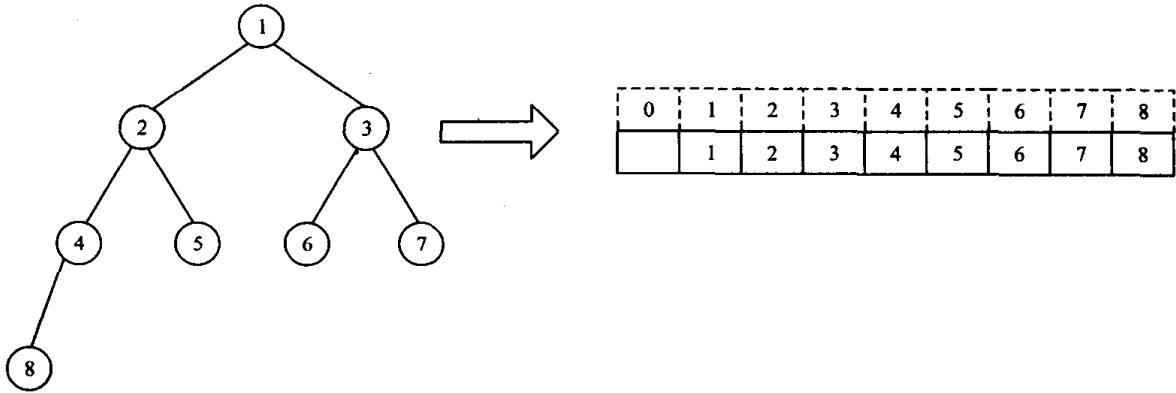


图 5.2 二叉树的顺序存储结构

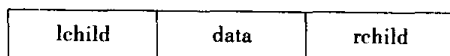
在 C 语言中,这种存储结构的类型定义如下所示:

```
#define MAX_TREE_SIZE 100
typedef struct {
    Elemtyp e elem[MAX_TREE_SIZE];      /* 0 号单元存储根结点 */
    int n;                               /* 完全二叉树的结点个数 */
} SqBiTree;
```

这种存储结构的特点是空间利用率高,寻找孩子和双亲比较容易。

(2) 链式存储结构

在顺序存储结构中,利用编号表示元素的位置及元素之间孩子或双亲的关系,因此对于非完全二叉树,需要将空缺的位置用特定的符号填补,若空缺结点较多,势必造成空间利用率的下降。在这种情况下,就应该考虑使用链式存储结构。常见的二叉树结点结构如下所示:



其中,lchild 和 rchild 是分别指向该结点的左孩子和右孩子的指针,data 存放数据元素的内容。在 C 语言中的类型定义为:

```
typedef struct BiTNode {
    Elemtyp data;
    struct BiTNode * lchild, * rchild;
} BiTNode, * BiTree;
```

图 5.3 是一棵二叉树及相应的链式存储结构。

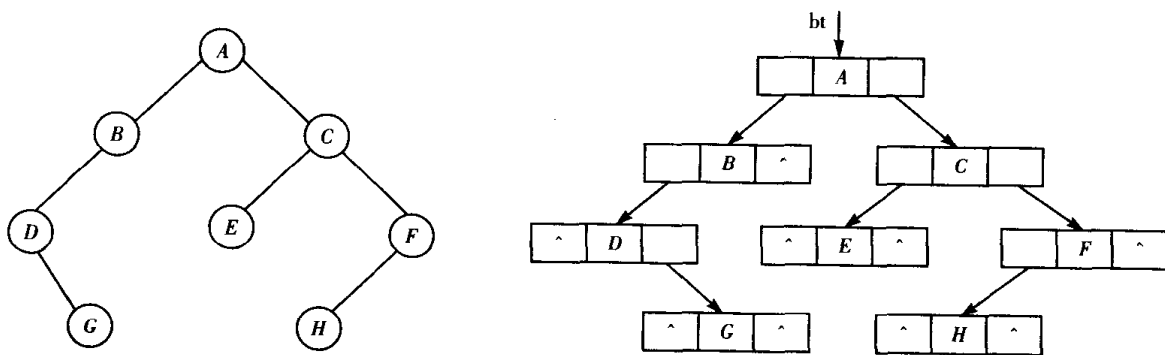


图 5.3 二叉树的链式存储结构

这种存储结构的特点是寻找孩子结点容易,而寻找双亲结点比较困难。因此,若需要频繁地寻找双亲结点,可以给每个结点添加一个指向双亲结点的指针域,其结点结构如下所示:



4. 二叉树的遍历

二叉树是一种非线性的数据结构,在对它进行操作时,需要逐一对每个数据元素实施操作,这样就存在一个操作顺序问题,由此提出了二叉树的遍历操作。所谓遍历二叉树就是按某种顺序访问二叉树中的每个结点一次且仅一次的过程。这里的访问可以是输出、比较、更新、查看元素内容等等各种操作。

根据对根结点访问的顺序不同,二叉树的遍历方式有三种:先序遍历(TLR)、中序遍历(LTR)和后序遍历(LRT)。

(1) 先序遍历

若二叉树为空,则结束遍历操作;否则:

访问根结点;

先序遍历左子树;

先序遍历右子树。

其递归算法如下:

```
void PreOrder(BiTree bt)
{
    BiTNode *p;
    p = bt;
    if(p != NULL)
    {
        visit(p->data);
        PreOrder(p->lchild);
        PreOrder(p->rchild);
    }
}
```

(2) 中序遍历

若二叉树为空,则结束遍历操作;否则:

中序遍历左子树;

访问根结点;

中序遍历右子树。

其递归算法如下:

```
void InOrder(BiTree bt)
{
    BiTNode *p;
    p = bt;
    if(p != NULL)
    {
        InOrder(p->lchild);
        visit(p->data);
        InOrder(p->rchild);
    }
}
```

(3) 后序遍历

若二叉树为空,则结束遍历操作;否则:

后序遍历左子树;

后序遍历右子树;

访问根结点。

其递归算法如下：

```
void PostOrder(BiTree bt)
{
    BiTNode *p;
    p = bt;
    if(p != NULL)
    {
        PostOrder(p->lchild);
        PostOrder(p->rchild);
        visit(p->data);
    }
}
```

5.1.3 树的存储结构

1. 双亲表示法

以一组连续空间存储树的结点,同时在每个结点中附设一个指示器指示其双亲结点在顺序表中的位置。

类型定义：

```
#define MAX_TREE_SIZE 100
/* 结点类型定义 */
typedef struct PTNode{
    Elemtype data;
    int parent;                /* 双亲结点下标 */
} PTNode;
/* 树类型定义 */
typedef struct {
    PTNode nodes[MAX_TREE_SIZE];
    int r,n;                  /* 树根的位置及树的结点数 */
} PTree;
```

图 5.4 展示了一棵树及其双亲表示的存储结构。

这种存储方法的特点是寻找结点的双亲很容易,但寻找结点的孩子比较困难。

2. 孩子表示法

主要描述的是结点的孩子关系。可以采用顺序存储结构或链式存储结构,由于每个结点的孩子数目不定,所以采用链式存储结构更加合适。

图 5.5 展示了一棵树及其孩子表示的链式存储结构。

这种存储结构的特点是寻找结点的孩子比较容易,但寻找双亲比较麻烦,所以,在必要的时候,可以将双亲表示法和孩子表示法结合起来,即将一维数组元素增加一个表示双亲结点的域 parent,用来指示结点的双亲在一维数组中的位置。

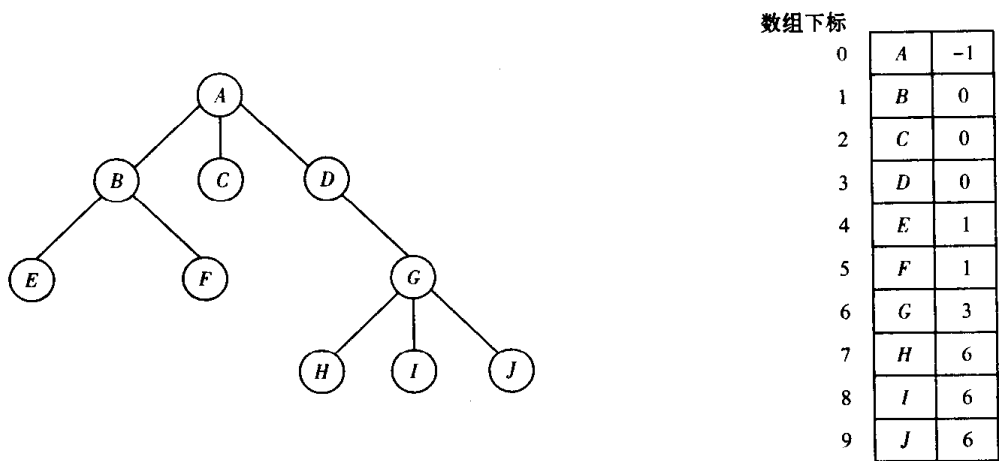


图 5.4 树的双亲表示法

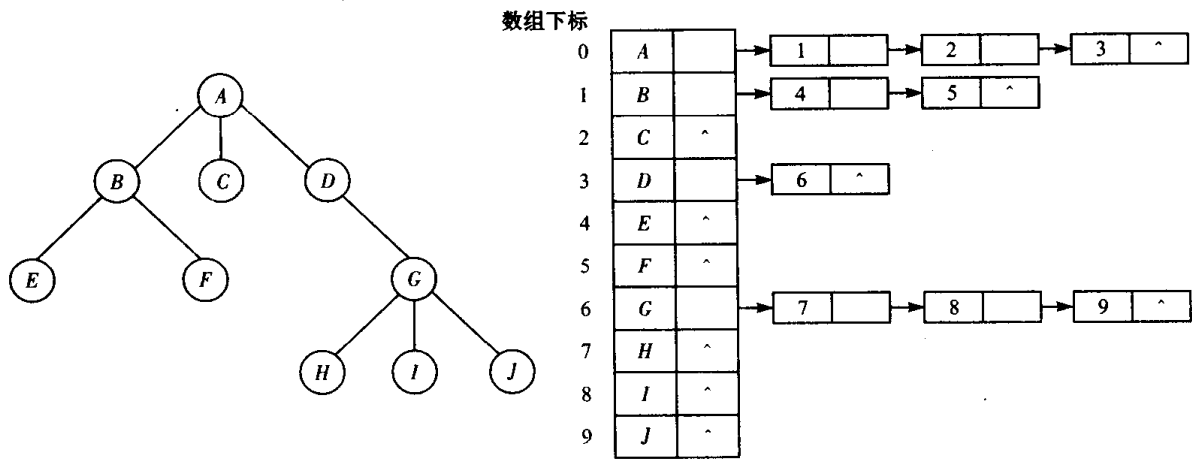


图 5.5 树的孩子表示法

3. 孩子兄弟表示法

孩子兄弟表示法是一种链式存储结构。它通过描述每个结点的一个孩子和兄弟信息来反映结点之间的层次关系,其结点结构为:

firstchild	data	nextsibling
------------	------	-------------

其中,firstchild 为指向该结点第一个孩子的指针,nextsibling 为指向该结点的下一个兄弟的指针。

图 5.6 展示了一棵树及其孩子兄弟表示的链式存储结构。

5.1.4 哈夫曼树

1. 哈夫曼树的定义

哈夫曼树(Huffman)又称为最优二叉树,是一类带权路径最短的树。

路径长度:在二叉树中,一个结点到另一个结点之间的分支数目。

树的路径长度:树的根结点到树中每一个结点的路径长度之和。

带权路径长度:从根结点到某结点的路径长度与该结点的权的乘积。

树的带权路径长度:树中所有叶子结点的带权路径长度之和。

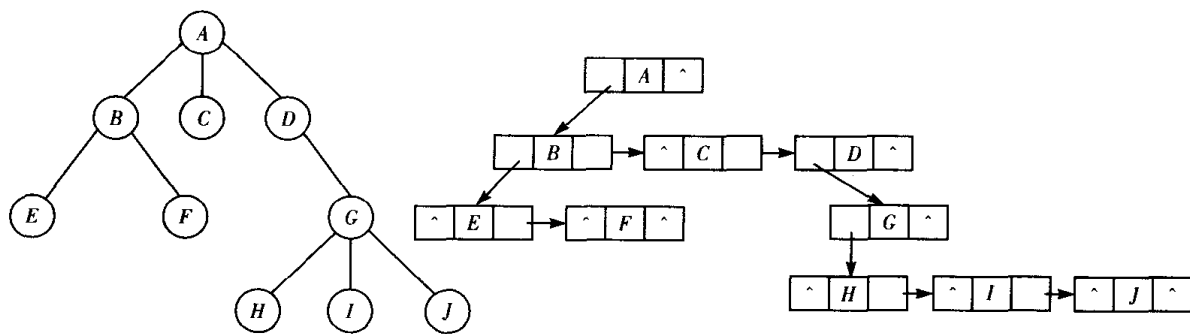


图 5.6 树的孩子兄弟表示法

2. 哈夫曼树的构造

构造哈夫曼树的过程:

- (1) 将给定的 n 个权值 $\{W_1, W_2, \dots, W_n\}$ 作为 n 个根结点的权值, 构造一个具有 n 棵二叉树的森林 $\{T_1, T_2, \dots, T_n\}$, 其中每棵二叉树只有一个根结点;
- (2) 在森林中选取两棵根结点权值最小的二叉树作为左右子树构造一棵新二叉树, 新二叉树的根结点权值为这两棵树根的权值之和;
- (3) 将上面选择的两棵根权值最小的二叉树从森林中删除, 并将新构造的二叉树加入到森林中去;
- (4) 重复步骤(2)和(3), 直到森林中只有一棵二叉树为止。这棵二叉树就是哈夫曼树。

5.2 实训篇

实训目的:

- 理解二叉树的顺序和链式存储结构
- 熟练掌握二叉树的建立, 以及对二叉树进行先序、中序和后序遍历的非递归算法
- 掌握构造哈夫曼树的过程, 利用哈夫曼树求出哈夫曼编码, 以及利用哈夫曼树进行译码
- 理解回溯算法与树的遍历之间的内在联系, 学会运用回溯算法解决实际问题

5.2.1 二叉树的建立及三种遍历的实现

案例描述

编程实现建立一棵二叉树, 然后对其进行先序、中序和后序遍历。

设计分析

在本设计中对于二叉树的存储采用链式存储结构。

二叉树的建立: 利用队列先进先出的特点, 按完全二叉树的层次顺序, 依次输入树的所有结点, 建立一棵二叉树。

具体过程: ①将要输入的二叉树按照其对应的完全二叉树的顺序输入, 若当前位置不存在结点则输入“@”, 以“#”作为结点输入完毕的标志。例如, 若要建立如图 5.7 所示的

一棵二叉树,按层次顺序依次输入的结点序列为:ABDC@ EG@ @ @ @ @ F#。

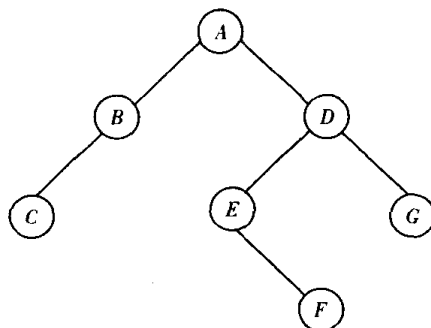


图 5.7 二叉树示意图

②按照输入的结点信息构造树结点,并设置一个用于存放树结点指针的队列,将树结点的地址依次放入队列中;

③若当前树结点是最先进入队列的,则其为树的根结点,否则将其链接到其双亲结点上。因为是按层自左向右输入结点的,所以先输入的结点的孩子结点先进入队列,现在利用队列的队尾下标 rear 指向当前结点,用队列的队首指针 front 指向当前结点的双亲结点。若 rear 为偶数,当前结点作为左孩子插入其双亲结点,若 rear 为奇数,当前结点作为右孩子插入其双亲结点。然后,其双亲结点出队列。

④如此反复,直到输入结束符“#”时为止。

利用栈来实现对二叉树的先序、中序和后序遍历。

先序遍历的非递归算法思想:将二叉树根结点指针入栈,然后执行出栈,获取栈顶元素。若其不为空指针,则访问该指针指向的结点,再判断其是否有左右子树,若有则再将其左右子树的根结点指针分别入栈,再重复以上的操作,直到栈为空,则整个遍历结束。

例如,对上图中的二叉树进行先序遍历过程中,结点指针入栈出栈的情况如图 5.8 所示,得到的遍历序列为:ABCDEFGF。

中序遍历的非递归算法思想:将二叉树根结点指针入栈,若栈顶元素的指针为非空,则应该遍历其左子树,将指向其左子树根结点的指针置入栈中;否则,应该退出一层,此时若是从左子树返回的,则应访问当前栈顶指针所指向的结点,若是从右子树返回的,则说明当前层的结点已经全部被访问过,应该继续退栈。

后序遍历的非递归算法思想: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,在这里必须记下根结点的位置,以便在遍历右子树之后能正确返回。另外,为了能在退栈回到根结点时区分是从左子树返回的还是从右子树返回的,这里采用两个栈,分别用来存放树的结点地址和左右子树标志(0 标志左子树,1 标志右子树),它们共用同一个栈顶指针。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。

设计方案

(1) 存储类型的定义

二叉树存储类型的定义:

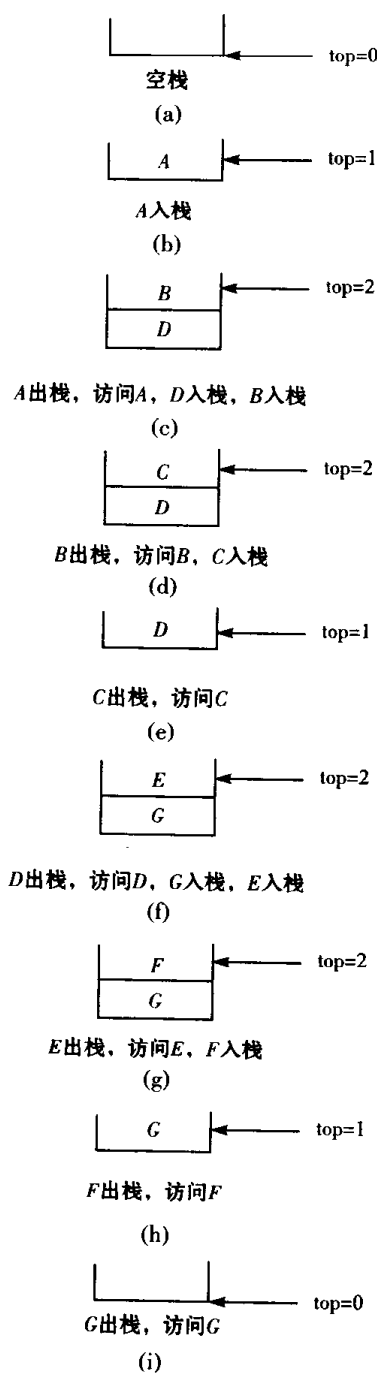


图 5.8 先序遍历二叉树入栈出栈示意图

```

typedef struct node
{
    char data; /* 结点的数据域 */
    struct node *lchild, *rchild; /* 指向左右子树的指针域 */
} BinTNode, * BinTree;

typedef struct
{
    BinTree elem[MAXSIZE];
}

```

```

    int top;                                /* 栈顶下标 */
} SqStack;

(2)按层次建立二叉树的算法
/* 按层次顺序输入创建一棵二叉树 */
BinTree CreateBinTree()
{
    BinTNode * Q[MAXSIZE];                /* 存放树结点指针的队列 */
    BinTNode * s, * bt;
    int front, rear;                       /* 定义队首、队尾指针 */
    char ch;
    bt = NULL;
    front = 1;                             /* 初始化队首、队尾指针 */
    rear = 0;
    printf("请按层次顺序输入树的结点信息,以'#'结束\n");
    while((ch = getchar()) != '#')         /* 循环获得结点的值 */
    {
        s = NULL;
        if(ch != '@')                     /* 若输入的不是空结点,则创建新结点 */
        {
            s = (BinTNode *)malloc(sizeof(BinTNode));
            s->data = ch;                   /* 给新结点的数据域赋值 */
            s->lchild = s->rchild = NULL;    /* 新结点的左右孩子指针置空 */
        }
        rear++;                           /* 队尾后移一位 */
        Q[rear] = s;                       /* 新结点或空结点地址入队列 */
        if(rear == 1)                     /* 入队列的第一个结点为树的根结点 */
            bt = s;
        else
        {
            if(s != NULL && Q[front] != NULL) /* 当前结点和其双亲结点都不是空结点 */
            {
                if(rear % 2 == 0)           /* 若 rear 为偶数,当前结点作为左孩子插入 */
                    Q[front] ->lchild = s;
                else                         /* 若 rear 为奇数,新结点作为右孩子插入 */
                    Q[front] ->rchild = s;
                front++;                     /* 左右孩子都处理完毕,队首指针后移一位,该结点出队列 */
            }
        }
    }
}

```

```
    return bt;                /* 返回根结点地址 */
}
```

(3) 先序遍历二叉树的非递归算法

/* 先序遍历非递归算法 */

```
void PreOrder(BinTree t)
```

```
{
    SqStack S;
    BinTree p;
    S.top=0;
    p=t;
    while (p!=NULL || S.top>0)
    {
        while (p!=NULL)        /* 遍历左子树 */
        {
            printf("%c ",p->data);
            S.elem[S.top]=p;
            S.top++;
            p=p->lchild;
        }
        if (S.top>0)            /* 通过下一次循环中的内嵌 while 实现右子树遍历 */
        {
            S.top--;
            p=S.elem[S.top];
            p=p->rchild;
        }
    }
}
```

(4) 中序遍历二叉树的非递归算法

/* 中序遍历非递归算法 */

```
void InOrder(BinTree t)
```

```
{
    SqStack S;
    BinTree p;
    S.top=0;
    p=t;
    while (p!=NULL || S.top>0)
    {
        while (p!=NULL)        /* 遍历左子树 */
        {
            S.elem[S.top]=p;
            S.top++;
        }
    }
}
```

```

        p=p->lchild;
    }
    if (S.top>0)
    {
        S.top--;
        p=S.elem[S.top];
        printf("% c ",p->data); /* 访问根结点 */
        p=p->rchild;             /* 通过下一次循环实现右子树遍历 */
    }
}
}

```

(5) 后序遍历二叉树的非递归算法

```
void PostOrder(BinTree t)
```

```

{
    SqStack S;
    int tag[MAXSIZE];
    BinTree p;
    S.top=0;
    p=t;
    do
    {
        while (p!=NULL)                /* 遍历左子树 */
        {
            S.top++;
            tag[S.top] = 0;             /* 标记为左子树 */
            S.elem[S.top] = p;
            p=p->lchild;
        }
        while (S.top>0 && tag[S.top] == 1)
        {
            p=S.elem[S.top];
            S.top--;
            printf("% c ",p->data); /* tag 为 1,表示右子树访问完毕,故访问根结点 */
        }
        if (S.top>0)
        {
            p=S.elem[S.top]->rchild; /* 遍历右子树 */
            tag[S.top] = 1;          /* 将当前结点的标志置为 1,表示其右子树已经访问过 */
        }
    }while (S.top>0);
}

```

```
main()
{
    int k;
    BinTree bt;
    bt = NULL;
    while(1)
    {
        printf("请选择(0-5)\n");
        printf("1—建立二叉树 2—先序遍历 3—中序遍历 4—后序遍历 0—退出\n");
        scanf("%d",&k);
        switch(k)
        {
            case 1:
                bt = CreateBinTree();
                break;
            case 2:
                if(bt == NULL) printf("请先建立一棵二叉树! \n");
                else
                {
                    printf("先序遍历序列为:\n");
                    PreOrder(bt); printf("\n");
                }
                break;
            case 3:
                if(bt == NULL) printf("请先建立一棵二叉树! \n");
                else
                {
                    printf("中序遍历序列为:\n");
                    InOrder(bt);printf("\n");
                }
                break;
            case 4:
                if(bt == NULL) printf("请先建立一棵二叉树! \n");
                else
                {
                    printf("后序遍历序列为:\n");
                    PostOrder(bt); printf("\n");
                }
                break;
            case 0:
                printf("退出程序\n");
```

```
exit(0);
```

(6) 运行测试

编译运行后显示结果为:

请选择(0-5)

1—建立二叉树 2—先序遍历 3—中序遍历 4—后序遍历 0—退出

输入:2(回车)

请先建立一棵二叉树!

请选择(0-5)

1—建立二叉树 2—先序遍历 3—中序遍历 4—后序遍历 0—退出

输入:1(回车)

请按层次顺序输入树的结点信息,以'#'结束

输入:ABDC@EG@@@@@F#(回车)

请选择(0-5)

1—建立二叉树 2—先序遍历 3—中序遍历 4—后序遍历 0—退出

输入:2(回车)

先序遍历序列为:

A B C D E F G

请选择(0-5)

1—建立二叉树 2—先序遍历 3—中序遍历 4—后序遍历 0—退出

输入:3(回车)

中序遍历序列为:

C B A E F D G

请选择(0-5)

1—建立二叉树 2—先序遍历 3—中序遍历 4—后序遍历 0—退出

输入:4(回车)

后序遍历序列为:

C B F E G D A

请选择(0-5)

1—建立二叉树 2—先序遍历 3—中序遍历 4—后序遍历 0—退出

输入:0(回车)

退出程序

5.2.2 哈夫曼编码/译码系统

案例描述

利用哈夫曼编码进行通信,可以减少通信的数据量,提高传输效率,缩短信息的传输时间,还有一定的保密性。现在要求编写一程序模拟传输过程,实现在发送前将要发送的字符信息进行编码,然后进行发送,接收后将传来的数据进行译码,即将信息还原成发送前的字符信息。

设计分析

在本例中设置发送者和接收者两个功能。

发送者的功能包括:

- ①输入待传送的字符信息;
- ②统计字符信息中出现的字符种类数和各字符出现的次数(频率);
- ③根据字符的种类数和各字符出现的次数建立哈夫曼树;
- ④利用以上哈夫曼树求出各字符的哈夫曼编码;
- ⑤将字符信息转换成对应的编码信息进行传送。

接收者的功能包括:

- ①接收发送者传送来的编码信息;
- ②利用上述哈夫曼树对编码信息进行翻译,即将编码信息还原成发送前的字符信息。

从以上分析可发现,在本例中的主要算法有三个:

- (1)哈夫曼树的建立;
- (2)哈夫曼编码的生成;
- (3)对编码信息的翻译。

设计方案

(1)哈夫曼树的建立

根据统计函数得到字符的种类数 N 和各自出现的次数,建立 N 个结点;由此构造一棵哈夫曼树。

哈夫曼树的存储结构类型定义为:

```
#define N 30                                /* 叶子结点数,即在信息中最多可出现 30 种字符 */
typedef struct
{
    char data;                               /* 编码对应的字符 */
    int weight;                              /* 结点的权值 */
    int lchild,rchild,parent;               /* 左右孩子及双亲的下标 */
} HTNode;

void CreatHufmTree(HTNode ht[],int n)      /* 建立哈夫曼树 */
{
    int i,k,m1,m2,l,r;
    for(i=1;i<=2*n;i++)                    /* 对所有结点的左右孩子及双亲指针域赋空值 */
```

```

    ht[i].lchild=ht[i].rchild=ht[i].parent=0;
for(i=n+1;i<=2*n-1;i++)
{
    m1=m2=10000;          /* m1 为最小值,m2 为次小值 */
    l=r=0;
    for(k=1;k<=i-1;k++)
        if(ht[k].parent==0&&ht[k].weight<m1)
            /* 在前面 i-1 个结点中选择没有双亲且权值最小的两个结点 */
            {
                m2=m1;
                r=l;
                m1=ht[k].weight;
                l=k;
            }
        else if(ht[k].parent==0&&ht[k].weight<m2)
            {
                m2=ht[k].weight;
                r=k;
            }
    ht[l].parent=i;          /* 下标为 i 的新结点成为权值最小的两个结点的双亲 */
    ht[r].parent=i;
    ht[i].weight=ht[l].weight+ht[r].weight;
                                /* 新结点的权值为两个结点权值之和 */
    ht[i].lchild=l;          /* 权值最小的结点是新结点的左孩子 */
    ht[i].rchild=r;         /* 权值次小的结点是新结点的右孩子 */
}
}

```

(2) 哈夫曼编码的生成

哈夫曼编码的存储结构类型定义为:

```

typedef struct
{
    char bits[N];          /* 存放哈夫曼编码的字符数组 */
    int start;             /* 记录编码的起始位置,因为每种字符的编码长度不同 */
} HCode;

void HufmCode(HTNode ht[], HCode hcd[], int n)
    /* 利用哈夫曼树求出各字符的哈夫曼编码 */
{
    int i, f, c, k;
    HCode cd;              /* 用于临时存放编码串 */
    for(i=1; i<=n; i++)

```

```

}
cd.start = n + 1;
c = i; /* 从叶子结点 ht[i] 开始向上回溯 */
f = ht[i].parent; /* 找到结点 ht[i] 的双亲结点 ht[f] */
while(f != 0) /* 回溯到树根结点为止 */
{
    if(ht[f].lchild == c) /* 若 ht[c] 是 ht[f] 的左孩子, 生成代码为 0 */
        cd.bits[ --cd.start] = '0';
    else /* 若 ht[c] 是 ht[f] 的右孩子, 生成代码为 1 */
        cd.bits[ --cd.start] = '1';
    c = f;
    f = ht[f].parent;
}
hcd[i] = cd; /* 将得到的第 i 种字符的哈夫曼编码存入 hcd[i] 中 */
}
printf("输出哈夫曼编码:\n");
for(i = 1; i <= n; i++)
{
    printf("% c:", ht[i].data);
    for(k = hcd[i].start; k <= n; k++)
        printf("% c", hcd[i].bits[k]);
    printf("\n");
}

```

(3) 对编码信息的翻译

```
void TsCode(char *bit, HTNode ht[], int n) /* 哈夫曼树的译码 */
{
    int i;
    i = 2 * n - 1; /* 将树根结点的下标赋 i, 从根结点出发向下搜索 */
    while( *bit != '\0') /* 若编码没有结束 */
    {
        if( *bit == '0')
            i = ht[i].lchild; /* 走向左孩子结点 */
        else
            i = ht[i].rchild; /* 走向右孩子结点 */
        if(ht[i].lchild == 0) /* 判断是否已经走到叶子结点 */
        {
            putchar(ht[i].data); /* 输出此编码对应的字符 */
            i = 2 * n - 1; /* 重新回到根结点, 准备下一次搜索 */
        }
        bit++; /* 取编码中的下一个代码 */
    }
}
```

(4) 程序实现

```

#include "stdio.h"
#include "string.h"
#include "conio.h"
#include "stdlib.h"
#define N 30                                /* 最大叶子结点数 */
typedef struct
{
    char data;                               /* 编码对应的字符 */
    int weight;                             /* 结点的权值 */
    int lchild, rchild, parent;             /* 左右孩子及双亲的下标 */
} HTNode;

typedef struct
{
    char bits[N];                           /* 存放哈夫曼编码的字符数组 */
    int start;                              /* 编码的起始位置 */
} HCode;

int Stat(char *st, int cnt[], char str[])
/* 统计字符信息中出现的字符种类数和各字符出现的次数 */
{
    char *p;
    int i, j, k, num[27];
    for(i = 1; i <= 26; i++)
        num[i] = 0;
    for(p = st; *p != '\0'; p++)
    {
        *p = tolower(*p);                  /* 若字符信息中有大写字母,则将其转换成小写字母 */
        if(*p >= 'a' && *p <= 'z')
        {
            k = *p - 96;
            num[k]++;
        }
    }
    j = 0;
    for(i = 1; i <= 26; i++)
        if(num[i] != 0)
        {
            j++;
            str[j] = i + 96;
        }
    }

```

```

        cnt[j] = num[i];
    }
    return j;
}

main()
{
    int i, j, k, n, t, x, cnt[27];
    char st[50], sr[27], bm[200];
    HTNode ht[2 * N - 1];          /* 用于存放树中所有结点 */
    HCode hcd[N];                  /* 用于存放字符的哈夫曼编码 */
    while(1)
    {
        printf("1—输入待传送的字符信息 2—编码 3—发送 4—接收并译码 0—退出\n");
        scanf("%d", &x);
        switch(x)
        {
            case 1: printf("请输入要发送的字符串信息:");
                     scanf("%s", st);
                     break;
            case 2: n = Stat(st, cnt, sr);
                     CreatHufmTree(ht, n, sr, cnt);
                     HufmCode(ht, hcd, n);
                     break;
            case 3: t = 0;
                     for(j = 0; st[j] != '\0'; j++)
                     {
                         for(i = 1; i <= n; i++)
                         {
                             if(ht[i].data == st[j])
                             {
                                 for(k = hcd[i].start; k <= n; k++)
                                 {
                                     bm[t] = hcd[i].bits[k];
                                     t++;
                                 }
                                 break;
                             }
                         }
                         bm[t] = '\0';
                         printf("发送完毕! \n");
                         break;
            case 4: printf("接收到的编码信息为:");

```

```

        puts(bm);
        printf("译码后的结果为:");
        TsCode(bm,ht,n);
        printf("\n");
        break;
    case 0: exit(0);
}
}
}

```

(5) 运行测试

编译运行后显示:

1—输入待传送的字符信息 2—编码 3—发送 4—接收并译码 0—退出

输入:1(回车)

请输入要发送的字符串信息:(输入)goodmorningmum(回车)

1—输入待传送的字符信息 2—编码 3—发送 4—接收并译码 0—退出

输入:2(回车)

输出哈夫曼编码:

d:1100

g:100

i:1101

m:00

n:101

o:01

r:1110

u:1111

1—输入待传送的字符信息 2—编码 3—发送 4—接收并译码 0—退出

输入:3(回车)

发送完毕!

1—输入待传送的字符信息 2—编码 3—发送 4—接收并译码 0—退出

输入:4(回车)

接收到的编码信息为:10001011110000011110101110110110000111100

译码后的结果为:goodmorningmum

1—输入待传送的字符信息 2—编码 3—发送 4—接收并译码 0—退出

输入:0(回车)

退出程序

5.2.3 六皇后问题求解及演示

案例描述

编写一程序,求解六皇后问题:在 6×6 的方格棋盘上,放置6个皇后棋子,每个棋子都有6个可选择的位置,但在任何时刻,棋盘的合法布局都必须满足3个约束条件,即要求任何两个皇后棋子都不占据棋盘上的同一行、同一列或者同一对角线。要求以图形的方式显示出所有的合法布局。

设计分析

很多实际问题不是根据某种确定的计算法则,而是利用试探和回溯的搜索技术来求解的。回溯法也是递归过程的一种重要方法,它的求解实质上是一个先序遍历“状态树”的过程,只是这棵树不是遍历前预先建立的,而是隐含在遍历过程中的。

在本设计中,我们利用回溯法来求解。求所有合法布局的过程就是在上述约束条件下先序遍历“状态树”的过程。具体操作为:在遍历过程中,判断棋盘上是否已经得到一个完整的布局,若是,则输出该布局;否则依次先序遍历满足约束条件的各棵子树,即首先判断该子树根的布局是否合法,若合法,则先序遍历该子树,否则剪去该子树分支。

其算法描述如下:

```
void Trial(int i,int n){
    /* 进入本函数时,在 $n \times n$ 棋盘前 $i-1$ 行放置了互不攻击的 $i-1$ 个棋子。 */
    /* 现在从第 $i$ 行起继续为后续棋子选择合适位置。当 $i > n$ 时,求得一个合法布局,输出之。 */
    if( $i > n$ )输出棋盘的当前布局;
    else for( $j=1; j \leq n; ++j$ ){
        在第 $i$ 行第 $j$ 列放置一个棋子;
        if(当前布局合法) Trial( $i+1,n$ );
        移走第 $i$ 行第 $j$ 列的棋子;
    }
}
```

设计方案

(1)主要算法:

```
int Judge(int y,int x,int r[]) /* 判断第 $x$ 列的第 $y$ 行上能否放置皇后 */
{
    int j=1;
    while( $j < x$ ) /* 1~ $x-1$ 列已经放置了符合约束条件的皇后 */
    {
        if( $r[j] == y \parallel (abs(r[j]-y) == abs(j-x))$ )
            /* 判断第 $j$ 列皇后是否在第 $y$ 行,或者其是否与第 $x$ 列第 $y$ 行的位置在同一对角线上 */
            return 0; /* 不符合约束条件,返回0 */
        j++;
    }
    return 1; /* 符合约束条件,返回1 */
}
```

```

void Lay(int x,int r[])
{
    int i;
    if(x>6)                                /* 若6列中的皇后都已经放置完毕,则输出合法布局 */
    {
        getch();
        ChessBoard();
        DrawChess(r);
    }
    else
        for(i=1;i<=6;i++)                /* 在x列上试探每一个位置是否符合约束条件 */
            if(Judge(i,x,r))              /* 下一棵要遍历的子树由当前的判断结果决定 */
            {
                r[x]=i;
                Lay(x+1,r);                /* 进入下一层递归 */
            }
}

```

(2)程序实现

```

#include "stdio.h"
#include "stdlib.h"
#include "conio.h"
#include "dos.h"
#include "graphics.h"
void ChessBoard()                          /* 画棋盘 */
{
    int i,j;
    for(j=0;j<6;j++)
        for(i=0;i<6;i++)
            if(j%2==0&&i%2==0 || j%2!=0&&i%2!=0)
            {
                setfillstyle(SOLID_FILL, GREEN);
                setcolor(GREEN);
                rectangle(i*20,20+j*20,(i+1)*20,20+(j+1)*20);
                floodfill(i*20+10,20+j*20+10, GREEN);
            }
    else
    {
        setfillstyle(SOLID_FILL, WHITE);
        setcolor(WHITE);
        rectangle(i*20,20+j*20,(i+1)*20,20+(j+1)*20);
        floodfill(i*20+10,20+j*20+10, WHITE);
    }
}

```

```

void DrawChess(int r[])          /* 在棋盘上画出符合布局的棋子 */
{
    int i;
    for(i=1;i<=6;i++)
    {
        setfillstyle(SOLID_FILL,YELLOW);
        setcolor(YELLOW);
        circle((i-1)*20+10,r[i]*20+10,8);
        floodfill((i-1)*20+10,r[i]*20+10,YELLOW);
    }
}

main()
{
    int r[6];
    int gdriver=DETECT,gmode;
    initgraph(&gdriver,&gmode,"d:\\tc");
    Lay(1,r);
    getch();
}

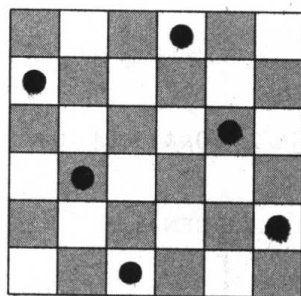
```

(3) 运行测试

编译运行后显示结果为:

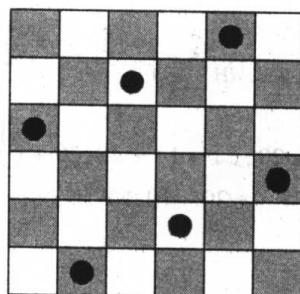
输入:按任意键

显示图形:

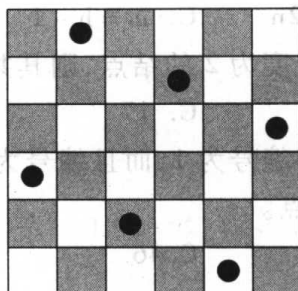


输入:按任意键

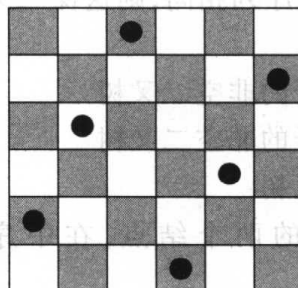
显示图形:



输入:按任意键
显示图形:



输入:按任意键
显示图形:



5.3 实训项目

项目内容

求一棵二叉树上结点的路径,即求出从根结点到指定结点所经过的所有结点的序列。

设计要求

首先建立一棵采用链式结构存储的二叉树,然后任意输入一个结点信息,要求程序能求出从根结点到此结点之间的路径并输出。

项目提示

在查找过程中,采用后序遍历二叉树,因为后序遍历是最后访问根结点的。将按后序遍历查到的从指定结点到根结点的整个序列存入一个栈中,最后从栈顶依次输出其中的所有结点,即为所求的路径。

5.4 基本题

5.4.1 选择题

1. 二叉树的第 i 层最多有_____个结点。

A. 2^i

B. $2i$

C. 2^{i-1}

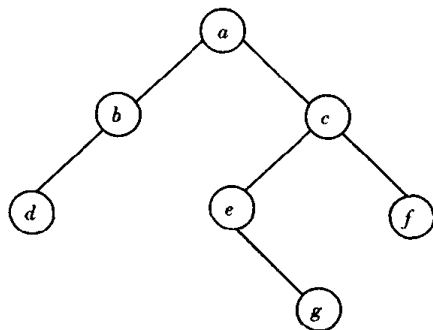
D. $2^i - 1$

2. 对于一棵满二叉树, 高度为 h , 共有 n 个结点, 其中有 m 个叶子结点, 则_____。
- A. $n = h + m$ B. $h + m = 2n$ C. $m = h - 1$ D. $n = 2^h - 1$
3. 在一棵二叉树中, 共有 16 个度为 2 的结点, 则其共有_____个叶子结点。
- A. 15 B. 16 C. 17 D. 18
4. 一棵完全二叉树中根结点的编号为 1, 而且编号为 23 的结点有左孩子但没有右孩子, 则此树中共有_____个结点。
- A. 24 B. 45 C. 46 D. 47
5. 下述编码哪一组不是前缀码_____。
- A. 00, 01, 10, 11 B. 0, 1, 00, 11
C. 0, 10, 110, 111 D. 1, 01, 001, 000
6. 某二叉树的中序序列和后序序列相同, 则这棵二叉树必然是_____。
- A. 空树
B. 空树或任一结点均无左孩子的非空二叉树
C. 空树或任一结点均无右孩子的非空二叉树
D. 空树或仅有一个结点的二叉树
7. 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_____。
- A. n 在 m 的右边 B. n 是 m 的祖先 C. n 在 m 的左边 D. n 是 m 的子孙
8. 二叉排序树查找时, 当二叉排序树为_____时, 效率最优。
- A. 线索树 B. Huffman 树 C. 完全二叉树 D. 平衡二叉树

5.4.2 填空题

1. 高度为 h 的完全二叉树至少有_____个结点, 至多有_____个结点。

2. 对右图的二叉树进行遍历, 其先序序列为_____, 中序序列为_____, 后序序列为_____。



3. 已知一棵二叉树的先序序列和中序序列分别为 ABDGHCEFI 和 GDHBAECIF, 请画出此二叉树。
已知一棵二叉树的中序序列和后序序列分别为 BDCEAFHG 和 DECBHGFA, 请画出此二叉树。

4. 已知一棵完全二叉树的第 7 层有 10 个叶子结点, 则整个二叉树的结点数最多是_____个。

5. 一棵完全二叉树共有 1001 个结点, 其中叶子结点的个数是_____。

6. 下列算法是输出一棵二叉树的第 i 层 (根结点为第一层) 的所有结点的值。

```
typedef struct Node{
    int data;
```

```

    struct Node *lchild,*rchild;
}BitNode;
void level(BitNode *T,int i)
{
    if(T==NULL)
        return;
    if(i==1)
    {
        printf("%d\n",T->data);
        return;
    }
    level(____);
    level(____);
}

```

5.5 习题解析

1. 已知一棵树边的集合为 $\{ \langle i,m \rangle, \langle i,n \rangle, \langle e,i \rangle, \langle b,e \rangle, \langle b,d \rangle, \langle a,b \rangle, \langle g,j \rangle, \langle g,k \rangle, \langle c,g \rangle, \langle c,f \rangle, \langle h,l \rangle, \langle c,h \rangle, \langle a,c \rangle \}$ 。

请画出这棵树,并回答下列问题:

- (1) 哪个是根结点?
- (2) 哪些是叶子结点?
- (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 为根的子树深度是多少?

解 所画树如图 5.9 所示。

- (1) a 是根结点。
- (2) d, f, j, k, l, m, n 是叶子结点。
- (3) c 是结点 g 的双亲。
- (4) a, c 是结点 g 的祖先。
- (5) j, k 是结点 g 的孩子。
- (6) i, m, n 是结点 e 的子孙。
- (7) d 是结点 e 的兄弟; g, h 是结点 f 的兄弟。
- (8) 结点 b 和 n 的层次号分别是 2 和 5。

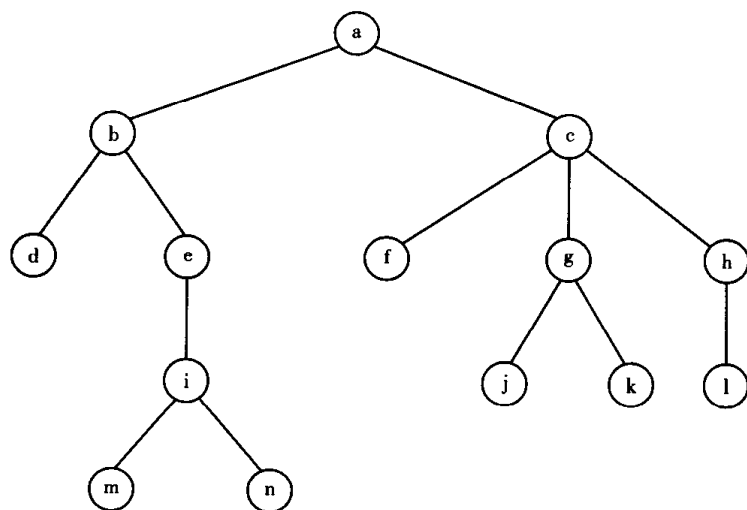


图 5.9 树

(9) 树的深度是 5。

(10) 以结点 c 为根的子树深度是 3。

2. 一棵度为 2 的树与一棵二叉树有何区别?

解 一棵度为 2 的树中存在有两个分支的结点, 该结点的两个分支没有左右之分; 一棵二叉树的结点也可有两个分支, 但两分支有左右之分, 且左右分支不能交换。

3. 试分别画出具有 3 个结点的树和 3 个结点的二叉树的所有不同形态。

解 (1) 具有 3 个结点的树的所有不同形态(共两种)如图 5.10 所示。

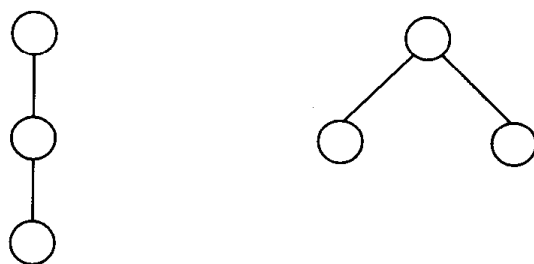


图 5.10 具有 3 个结点的树的不同形态

(2) 具有 3 个结点的二叉树的所有不同形态(共五种)如图 5.11 所示。

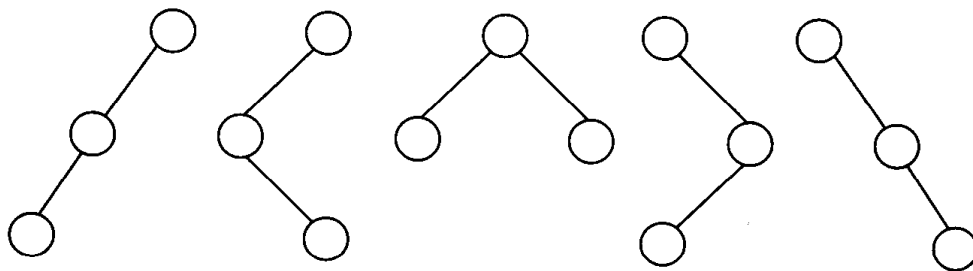


图 5.11 具有 3 个结点的二叉树的不同形态

4. 已知一棵度为 k 的树中有 n_1 个度为 1 的结点, n_2 个度为 2 的结点, $\dots$, n_k 个度为 k

的结点,问该树中有多少个叶子结点?

解 设度为 0 的结点(即叶子结点)数目为 n_0 , 树中总分支数为 b , 树中总结点数为 n , 则有:

$$\text{从度的角度看: } n = n_0 + n_1 + n_2 + \cdots + n_k \quad (1)$$

从分支的角度看有: $n = 1 + b$ (树中每个结点除根结点外都对应一个分支)

$$\text{而 } b = 0 \times n_0 + 1 \times n_1 + 2 \times n_2 + \cdots + k \times n_k$$

$$\text{则 } n = 1 + 0 \times n_0 + 1 \times n_1 + 2 \times n_2 + \cdots + k \times n_k \quad (2)$$

$$\text{合并(1),(2)得: } n_0 = 1 + 0 \times n_1 + 1 \times n_2 + \cdots + (k-1) \times n_k$$

$$\text{整理后可得树中叶子结点数目为: } n_0 = 1 + \sum_{i=2}^k (i-1)n_i$$

5. 一棵深度为 H 的满 k 叉树有如下性质:第 H 层上的结点都是叶子结点,其余各层上每个结点都有 k 棵非空子树。如果按层次自上而下,从左到右的顺序从 1 开始对全部结点编号,问:

(1) 各层的结点数目是多少?

(2) 编号为 n 的结点的双亲结点(若存在)的编号是多少?

(3) 编号为 n 的结点的第 i 个孩子结点(若存在)的编号是多少?

(4) 编号为 n 的结点有右兄弟的条件是什么? 其右兄弟的编号是多少?

解 (1) 第 $i(1, 2, \dots, H)$ 层上的结点数目是: k^{i-1} 个结点。

很明显,第 1 层的结点数为 1,而且对于任意的 $1 \leq i \leq H-1$,第 $i+1$ 层的结点数是第 i 层的结点数的 k 倍。如第 1 层为 k^0 ,第 2 层为 $k \times k^0 = k^1$,一般的就有第 i 层的结点数为 k^{i-1} , $1 \leq i \leq H$ 。

(2) 编号为 n 的结点的双亲结点(若存在)的编号是: $(k^{L-2} - 1)/(k-1) + \lceil j/k \rceil$, 或者是: $(k^{L-2} - 1)/(k-1) + \lceil (n - (k^{L-1} - 1)/(k-1))/k \rceil$

$$\text{其中: } L = \lceil (\log_2(n(k-1) + 1))/(\log_2 k) \rceil - 1$$

推导过程如下:

设参数 m_L 为满 k 叉树从第 1 层到第 L 层的总结点数, $L = 1, 2, \dots, H$ 。则有:

$$m_L = 1 + k + k^2 + \cdots + k^{L-1} = (k^L - 1)/(k-1)$$

对于满 k 叉树中的任意第 L 层的第 j 个结点,设其按层次顺序的结点编号为 n ,那么,就有:

$$n = m_{L-1} + j$$

其中, $L = 1, 2, \dots, H; j = 1, 2, \dots, K^{L-1}; m_0 = 0, 1 \leq n \leq m_H$ 。

由于 n 是定位在严格单调上升的数列 $m_1, m_2, \dots, m_H$ 中,便可以由 n 找到 L ,使得 $m_{L-1} < n \leq m_L$,然后求得 $j = n - m_{L-1}$ 。

因为树中有双亲结点的结点编号 n 满足 $1 < n \leq m_H$,对于任意给定的编号为 n 的结点,按上面所述,可以确定其所在的层次 L 和在该层中的结点序号 j ,也就可以确定其双亲结点是第 $L-1$ 层的第 $\lceil j/k \rceil$ ($\lceil j/k \rceil$ 表示上取整,即取不小于 j/k 的最小整数)个结点,那么,编号为 n 的结点的双亲结点(若存在)的编号为: $m_{L-2} + \lceil j/k \rceil = (k^{L-2} - 1)/(k-1) + \lceil j/k \rceil$, 或者为: $(k^{L-2} - 1)/(k-1) + \lceil (n - (k^{L-1} - 1)/(k-1))/k \rceil$ 。

(3) 编号为 n 的结点的第 i 个孩子结点(若存在)的编号是:

$$(k^L - 1)/(k - 1) + (j - 1)k + i。$$

或者是: $(k^L - 1)/(k - 1) + ((n - (k^{L-1} - 1)/(k - 1)) - 1)k + i。$

其中: $L = \lceil (\log_2(n(k - 1) + 1)) / (\log_2 k) \rceil - 1。$

推导过程如下:

很明显,编号为 n 的结点有第 i 个儿子结点的必要且充分的条件是 $1 \leq n \leq m_{H-1}$, 且 $1 \leq i \leq k。$

对于任意给定的 $1 \leq n \leq m_{H-1}$ 和 $1 \leq i \leq k$, 与解答(2)一样, 我们可以确定编号 n 的结点所在的层次 L 和在该层中的结点序号 j , 从而可以确定编号为 n 的结点的第 i 个孩子结点是第 $L+1$ 层的第 $(j-1)k + i$ 个结点。整理后得: 编号为 n 的结点的第 i 个孩子结点(若存在)的编号 = $m_L + (j-1)k + i = (k^L - 1)/(k - 1) + (j-1)k + i。$

(4) 编号为 $n(1 \leq n \leq m_H)$ 的结点有右兄弟的条件是: 当且仅当 $(n-1) \bmod k \neq 0。$

在上述条件满足时, 编号为 n 的结点的右兄弟的编号是: $n+1。$

推导过程如下:

为了确定编号为 n 的结点有右兄弟的条件, 可以先确定它没有右兄弟的条件, 然后取该条件的否定便是所要求的条件。当然, 这里必须以 $1 \leq n \leq m_H$ 为前提。显然, 编号为 n 的结点没有右兄弟当且仅当它是其父结点的最右儿子(即第 k 个儿子)。

这等价于 $(n-1) \bmod k = 0。$

事实上, 如果编号为 n 的结点没有右兄弟, 不妨设它是第 L 层的第 j 个结点, 那么, j 一定是 k 的正整数倍, 即 $j \bmod k = 0$, 而

$$n = m_{L-1} + j = 1 + \sum_{i=1}^{L-2} k^i + h = 1 + k \sum_{i=0}^{L-3} k^i + j$$

因此

$$n - 1 = k \sum_{i=0}^{L-3} k^i + j$$

从而

$$(n-1) \bmod k = j \bmod k = 0$$

反之, 如果 $(n-1) \bmod k = 0$, 则 $n-1$ 是 k 的非负整数倍, 即存在整数 $t, 0 \leq t \leq m_{H-1}$, 使得 $n-1 = t \times k。$

不妨设 $t = m_{L-1} + j, 1 \leq j \leq k_L, 1 \leq L \leq H-1$, 那么

$$\begin{aligned} n &= 1 + t \times k \\ &= 1 + (m_{L-1} + j)k \\ &= 1 + m_{L-1} \times k + j \times k \\ &= m_L + j \times k。 \end{aligned}$$

这表明编号为 n 的结点是第 $L+1$ 层的第 $j \times k$ 个结点。由于 $j \times k$ 是 k 的正整数倍, 因此, 该结点一定是其父结点的第 k 个儿子, 从而它没有右兄弟。

于是, 编号为 $n(1 \leq n \leq m_H)$ 的结点有右兄弟当且仅当

$$(n-1) \bmod k \neq 0。$$

至于在此条件满足时, 编号为 n 的结点的右兄弟的编号, 不言而喻是 $n+1。$

6. 一棵含有 n 个结点的 k 叉树,可能达到的最大深度和最小深度各为多少?

解 一棵含有 n 个结点的 k 叉树,如果构造成每一层都只有一个结点,则其达到的深度为最大,即深度为 n ;而一棵含有 n 个结点的 k 叉树,如果构造成完全 k 叉树的话,其深度为最小,即 $\lceil \log_k n(k-1) \rceil + 1$ 。

推导过程如下:

设一个完全 k 叉树的深度为 L ,则前 $L-1$ 层是满 k 叉树,设参数 m_{L-1} 为满 k 叉树从第 1 层到第 $L-1$ 层的总结点数,则有

$$m_{L-1} = 1 + k + k^2 + \cdots + k^{L-2} = (k^{L-1} - 1) / (k - 1)$$

就有

$$(k^{L-1} - 1) / (k - 1) < n \leq (k^L - 1) / (k - 1)$$

$$k^{L-1} - 1 < n(k - 1) \leq k^L - 1$$

$$k^{L-1} \leq n(k - 1) < k^L$$

对此式两边同时取对数有: $L - 1 \leq \log_k n(k - 1) < L$

因为 L 是整数,所以有: $L - 1 = \lceil \log_k n(k - 1) \rceil$

最后得: $L = \lceil \log_k n(k - 1) \rceil + 1$

7. 对第 3 题所得各种形态的二叉树,分别写出先序、中序和后序遍历的序列。

解 将第 3 题所得各种形态的二叉树加上字母 a, b, c, 如图 5.12 所示。

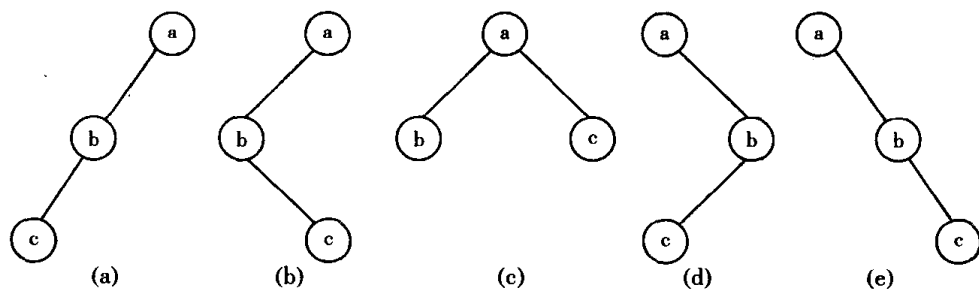


图 5.12 各种形态的二叉树

(a): 先序遍历: a, b, c

中序遍历: c, b, a

后序遍历: c, b, a

(c): 先序遍历: a, b, c

中序遍历: b, a, c

后序遍历: b, c, a

(e): 先序遍历: a, b, c

中序遍历: a, b, c

后序遍历: c, b, a

(b): 先序遍历: a, b, c

中序遍历: b, c, a

后序遍历: c, b, a

(d): 先序遍历: a, b, c

中序遍历: a, c, b

后序遍历: c, b, a

8. 找出所有满足下列条件的二叉树:

(1) 在先序遍历和中序遍历时,得到的结点访问序列相同;

(2) 在后序遍历和中序遍历时,得到的结点访问序列相同;

(3) 在先序遍历和后序遍历时,得到的结点访问序列相同。

解 (1) 在先序遍历和中序遍历时,得到的结点访问序列相同的二叉树是这样的一

类二叉树:树中的每一个结点都只有右子树,而没有左子树。

(2)在后序遍历和中序遍历时,得到的结点访问序列相同的二叉树是这样的一类二叉树:树中的每一个结点都只有左子树,而没有右子树。

(3)在先序遍历和后序遍历时,得到的结点访问序列相同的二叉树是这样的一棵二叉树:它只有一个根结点。

9. 假设一棵二叉树的先序序列为 EBADCFHGIKJ,中序序列为 ABCDEFGHIJK。请写出该二叉树的后序遍历序列。

解 有两种方法,第一种方法是先根据二叉树的先序序列和中序序列画出这棵二叉树,再写出该二叉树的后序遍历序列;第二种方法是对一棵二叉树的各结点在给出的先序序列和中序序列中的序列分布位置进行分析,从而写出该二叉树的后序遍历序列。

如按题目给出的一棵二叉树的先序序列为 EBADCFHGIKJ,中序序列为 ABCDEFGHIJK 进行分析,在先序序列中结点 E 就是树根结点,那么,在中序序列中,结点 E 将中序序列分成两部分,E 的左边 ABCD 是 E 的左子树,E 的右边 FGHJK 是 E 的右子树。那么,E 在后序遍历序列中一定处于最后位置,而 E 的左子树和 E 的右子树都在 E 的前面,且 E 的左子树结点又都在 E 的右子树结点前面。然后再依此方法,分析 E 的左子树和 E 的右子树。

如此可得该二叉树的后序遍历序列为:ACDBGJKIHFE。

10. 假设一棵二叉树的中序序列为 DCBGEAHFIK 和后序序列为 DCEGBFHKIA。请写出该二叉树的先序遍历序列。

解 方法类似于 5.9 题。该二叉树的先序遍历序列为:ABCDGEIHFJK。

11. 试编写算法判断两棵二叉树是否等价。如果 T1 和 T2 都是空的二叉树或者 T1 和 T2 的根结点的值相同,并且 T1 的左子树与 T2 的左子树是等价的,T1 的右子树与 T2 的右子树是等价的,则称二叉树 T1 和 T2 是等价的。

解 算法由主函数和创建二叉树、判断两棵二叉树是否等价三个函数组成。

程序如下:

```
#include <stdio.h>
typedef struct node
{
    int data;
    struct node *lchild;
    struct node *rchild;
} BTNODE, * BINTREE;          /* 定义二叉链表指针类型 */
int flag=0;

void createbintree(BINTREE *t) /* 输入二叉树的前序遍历序列,创建二叉树 */
{
    int a;
    scanf("%d",&a);
    if(a==0)
```

```

    *t = NULL;
else
{
    *t = (BTNODE *)malloc(sizeof(BTNODE));
    (*t) -> data = a;
    createbintree(&(*t) -> lchild);
    createbintree(&(*t) -> rchild);
}
}

void equal(BINTREE t1, BINTREE t2) /* 判断二叉树 t1 和 t2 是否等价, * /
/* flag=0 等价, flag=1 不等价, * /
{
    if(t1 && t2) /* t1 和 t2 都不为空 * /
    {
        if(t1 -> data == t2 -> data)
        {
            equal(t1 -> lchild, t2 -> lchild);
            equal(t1 -> rchild, t2 -> rchild);
        }
        else
            flag = 1;
    }
    else
        if((t1 && !t2) || (!t1 && t2))
            flag = 1;
}

main() /* 主函数 * /
{
    BINTREE t1 = NULL;
    BINTREE t2 = NULL;
    printf("\nplease input nodes of T1 BINTREE:\n");
    createbintree(&t1);
    printf("\nplease input nodes of T2 BINTREE:\n");
    createbintree(&t2);
    equal(t1, t2);
    if(flag == 0)
        printf("two trees are equal!");
    else
        printf("two trees are not equal!");
}

```

程序第一次执行输出情况为:

please input nodes of T1 BINTREE:

1 0 0(按回车)

please input nodes of T2 BINTREE:

1 0 0(按回车)

two trees are equal!

程序第二次执行输出情况为:

please input nodes of T1 BINTREE:

1 0 0(按回车)

please input nodes of T2 BINTREE:

1 2 0 0 0(按回车)

two trees are not equal!

12. 试编写算法交换以二叉链表作存储结构的二叉树中所有结点的左、右子树。

解 算法由主函数和创建二叉树、交换二叉树左右子树、输出二叉树中序遍历序列四个函数组成。

程序如下:

```
#include <stdio.h>
typedef struct node {
    int data;
    struct node *lchild;
    struct node *rchild;
} BTNODE, * BINTREE;          /* 定义二叉链表指针类型 */

void createbintree(BINTREE *t) /* 输入二叉树的前序遍历序列,创建二叉树 */
{
    int a;
    scanf("%d",&a);
    if(a == 0)
        *t = NULL;
    else
    {
        *t = (BTNODE *)malloc(sizeof(BTNODE));
        (*t) -> data = a;
        createbintree(&((*t) -> lchild));
        createbintree(&((*t) -> rchild));
    }
}

void exchange(BINTREE t)      /* 交换二叉树左右子树 */
{
    BINTREE p;
```

```

    if(t!=NULL)
    {
        exchange(t->lchild);
        exchange(t->rchild);
        p=t->lchild;
        t->lchild=t->rchild;
        t->rchild=p;
    }
}

void inorder(BINTREE t)          /* 输出二叉树中序遍历序列 */
{
    if(t)
    {
        inorder(t->lchild);
        printf("%d",t->data);
        inorder(t->rchild);
    }
}

main()                          /* 主函数 */
{
    BINTREE t=NULL;
    printf("\nplease input nodes of BINTREE:\n");
    createbintree(&t);
    printf("the inorder is: ");
    inorder(t);                  /* 交换前,中序遍历 */
    exchange(t);
    printf("\n");
    printf("the inorder is: ");
    inorder(t);                  /* 交换后,再中序遍历 */
}

```

程序执行输出情况为:

please input nodes of BINTREE:

1 2 4 0 0 5 0 0 3 0 6 0 0 (按回车)

the inorder is: 425136

/* 交换前,中序遍历 */

the inorder is: 631524

/* 交换二叉树中所有结点的左、右子树后,再中序遍历 */

13. 已知二叉排序树以二叉链表作存储结构,试编写算法按从大到小的顺序输出二叉排序树的各结点。

解 算法由主函数和创建二叉排序树、在二叉排序树中插入结点、中序遍历二叉排序树、RNL 方式的中序遍历二叉排序树五个函数组成。其中 RNL 方式的中序遍历二叉排序

树函数类似于中序遍历二叉排序树算法,区别在于该算法在中序遍历二叉排序树时,先访问右子树,再访问左子树,以实现二叉排序树按从大到小的顺序输出。

程序如下:

```
#include <stdio.h>
#define QUEUESIZE 20
typedef struct binode {          /* 定义结点数据类型 */
    int data;
    struct binode *lchild, *rchild;
} BTNODE, *BINTREE;
BINTREE insertbst(BINTREE s, BINTREE t) /* 在二叉排序树中插入结点 */
{
    if(t == NULL)
        t = s;                  /* 如果二叉排序树为空树,则插入的结点为根结点 */
    else
        if(s->data < t->data)     /* 如果该结点的数据小于根结点的数据则插入左子树 */
            t->lchild = insertbst(s, t->lchild);
        else
            /* 否则插入右子树 */
            t->rchild = insertbst(s, t->rchild);
    return t;
}

BINTREE createordbt()           /* 创建二叉排序树,输入0结束 */
{
    BINTREE s, t;
    int x;
    t = NULL;
    scanf("%d", &x);             /* 输入结点数据 */
    while(x != 0)
    {
        s = (BINTREE)malloc(sizeof(BTNODE));
        s->data = x;              /* 为生成的结点赋值 */
        s->lchild = NULL;
        s->rchild = NULL;
        t = insertbst(s, t);     /* 调用插入函数 */
        scanf("%d", &x);
    }
    return t;
}

void inorder(BINTREE t)         /* 中序遍历二叉排序树,结点输出序列为由小到大 */
{

```

```

    if(t!=NULL)
    {
        inorder(t->lchild);
        printf("%d",t->data);    /* 打印结点数值 */
        inorder(t->rchild);
    }
}

void rlinorder(BINTREE t)    /* RNL 方式的中序遍历二叉排序树,结点输出序列为由大到小 */
{
    if(t!=NULL)
    {
        rlinorder(t->rchild);
        printf("%d",t->data);    /* 打印结点数值 */
        rlinorder(t->lchild);
    }
}

main()
{
    BINTREE root;
    printf("\n");
    root=createordbt();    /* 调用创建二叉排序树函数 */
    inorder(root);    /* 调用中序遍历二叉排序树函数 */
    printf("\n");
    rlinorder(root);    /* 调用 RNL 方式中序遍历二叉排序树函数 */
}

```

程序执行输出情况为:

```

3 5 6 2 8 5 0(按回车)    /* 创建二叉排序树,输入整型数值的序列,以0结束 */
235568    /* 按 LNR 方式中序遍历二叉排序树,数据序列由小到大 */
865532    /* 按 RNL 方式中序遍历二叉排序树,数据序列由大到小 */

```

14. 画出和下列已知序列对应的树 T:

树的先根访问序列为 GFKDAIEBCHJ;

树的后根访问序列为 DIAEKFCJHBG。

解 本题目可以先画出与树 T 对应的二叉树,然后,再转换二叉树为树 T。

由于树 T 的先根序列等价于对应二叉树 B 的先序序列,而树 T 的后根序列等价于对应二叉树 B 的中序序列,那么,本题就转换为已知二叉树 B 的先序序列为 GFKDAIEBCHJ,中序序列为 DIAEKFCJHBG,画出这一棵二叉树 B。画这一棵二叉树 B 的方法可以参考第 9 题。二叉树 B 画好后,再按二叉树转换为树的规则,将其转换为树 T。结果如图 5.13 所示。

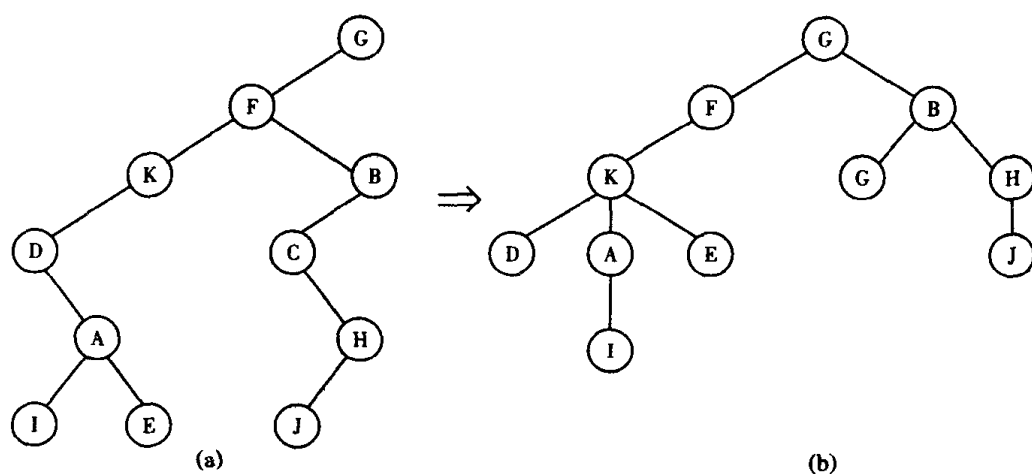


图 5.13 将二叉树转换为树

15. 给出图 5.14 中森林的先根、后根遍历结点序列, 然后画出此森林对应的二叉树。

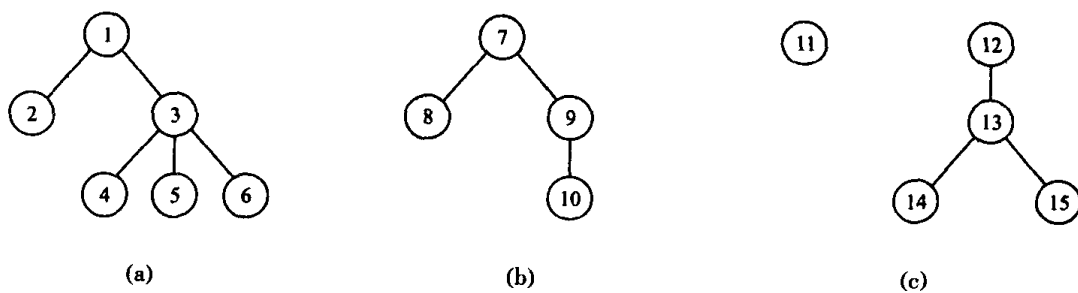


图 5.14 森林

解 对森林的先根遍历或后根遍历就是从左到右对组成森林的各树进行先根遍历或后根遍历。

森林的先根遍历结点序列: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

森林的后根遍历结点序列: 2, 4, 5, 6, 3, 1, 8, 10, 9, 7, 11, 14, 15, 13, 12

此森林对应的二叉树如图 5.15 所示。

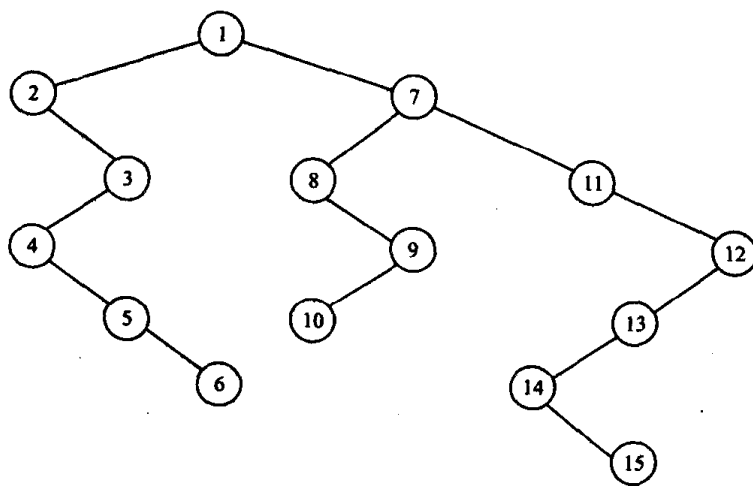


图 5.15 森林对应的二叉树

16. 假设用于通信的电文仅由 8 个字母组成, 字母在电文中出现的频率分别为 7, 19, 2, 6, 32, 3, 21, 10。试为这 8 个字母设计哈夫曼编码。使用 0~7 的二进制表示形式是另一种编码方案, 对于上述实例, 比较两种方案的优缺点。

解 设这 8 个字母是: A, B, C, D, E, F, G, H, 它们所对应的权值分别为 (7, 19, 2, 6, 32, 3, 21, 10)。可得哈夫曼树如图 5.16 所示:

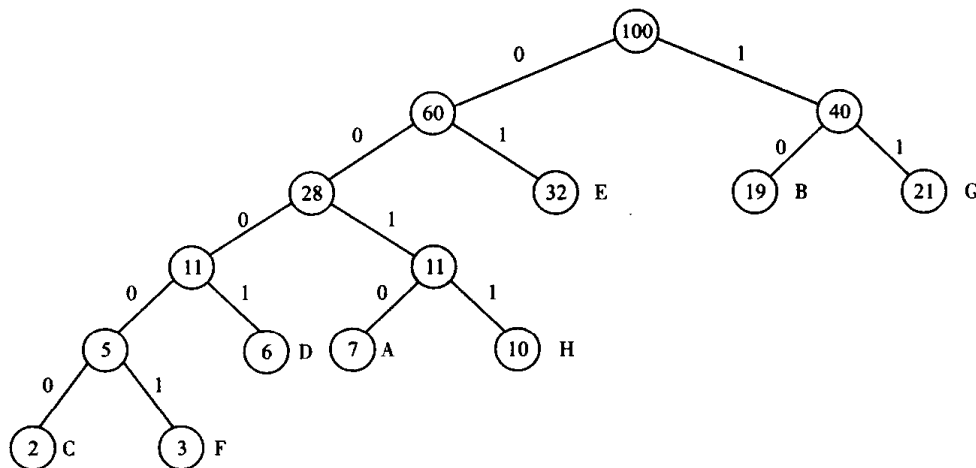


图 5.16 哈夫曼树

由图可知 8 个字母的哈夫曼编码为:

A:0010	B:10	C:00000	D:0001
E:01	F:00001	G:11	H:0011

使用 0~7 的二进制形式表示的编码为:

A:000	B:001	C:010	D:011
E:100	F:101	G:110	H:111

两种方案的优缺点如下:

对于上述实例, 若电文长度为 100 字母, 则:

哈夫曼编码长度为: $7 * 4 + 19 * 2 + 2 * 5 + 6 * 4 + 32 * 2 + 3 * 5 + 21 * 2 + 10 * 4 = 261$ (个二进制位)。

0~7 的二进制形式编码长度为: $100 * 3 = 300$ (个二进制位)。

可见, 电文长度同为 100 字母时, 哈夫曼编码长度比 0~7 的二进制形式编码长度要短。

但是, 若从接收到电文后进行译码的角度看, 0~7 的二进制形式编码的电文译码简单, 接收方只要按每 3 位一组进行分割, 便可译码成一个字母; 收到哈夫曼编码的电文, 译码则相对复杂, 接收方须根据对应的哈夫曼树实现。

第6章

图

6.1 基础知识篇

学习要点:

- 掌握图的基本概念及其存储结构
- 掌握图的深度优先遍历和广度优先遍历的方法和算法实现
- 掌握最小生成树的算法
- 掌握最短路径的两个经典算法:迪杰斯特拉(Dijkstra)和弗洛伊德(Floyed)算法
- 掌握拓扑排序的概念,会求拓扑序列
- 了解关键路径

本章实训是围绕数据结构中的图形结构展开的,基于图的基本概念及图的两种存储结构展开算法描述。针对图的深度优先遍历和广度优先遍历算法,以及最小生成树、最短路径、拓扑结构等应用问题进行研究,在实训案例中提出典型的应用问题并进行分析解决。

6.1.1 图的基本概念

1. 图(Graph)

图(Graph)是一种数据结构。图形结构都是由顶点和边组成的,记作 $G = (V, E)$, 其中, V 是顶点的非空有限集合, E 是连接 V 中两个不同顶点的边的有限集合。

2. 无向图(Undigraph)和有向图(Digraph)

如果图中每条边都是没有方向的,称此图为无向图。无向图的边用圆括号括起的两个相关顶点来表示。例如无向图 6.1 中 (V_1, V_2) , 且 (V_1, V_2) 和 (V_2, V_1) 表示同一条边。

如果图中每条边都是有方向的,即每条边在图示时均用箭头表示方向,称此图为有向图。有向图的边是用尖括号括起的两个相关顶点来表示的。例如有向图 6.2 中的 $\langle V_1, V_2 \rangle$, 也称作是一条弧,其中 V_2 称为弧头, V_1 称为弧尾。 $\langle V_2, V_1 \rangle$ 与 $\langle V_1, V_2 \rangle$ 表示的是不同的边。

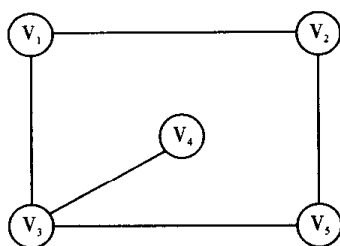


图 6.1 无向图

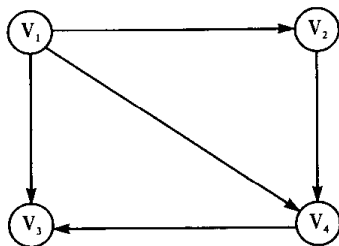


图 6.2 有向图

3. 度 (Degree)、入度 (Indegree) 和出度 (Outdegree)

对于无向图, 顶点 V 的度是指和 V 相关联的边的数目。例如图 6.1 中顶点 V_1 的度为 2, 顶点 V_3 的度为 3。

对于有向图, 以顶点 V 为头的弧的数目称为 V 的入度, 记作 $ID(V)$; 以顶点 V 为尾的弧的数目称为 V 的出度, 记作 $OD(V)$ 。顶点 V 的度记作 $TD(V) = ID(V) + OD(V)$ 。如图 6.2 所示, 顶点 V_2 的入度为 1, 出度为 1, 度为 2; 顶点 V_4 的入度为 2, 出度为 1, 度为 3。

4. 无向完全图 (Completed Graph) 和有向完全图 (Completed Digraph)

若一个无向图有 n 个顶点, 且每个顶点与其他 $n-1$ 个顶点之间都有边相连, 即有 $n(n-1)/2$ 条边。这样的图称为无向完全图。同理, 若一个有向图有 n 个顶点, 且有 $n(n-1)$ 条弧, 这样的图称为有向完全图。很显然, 图 6.3 是无向完全图, 图 6.4 是有向完全图。

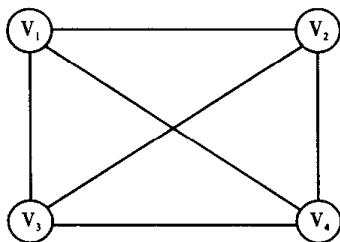


图 6.3 无向完全图

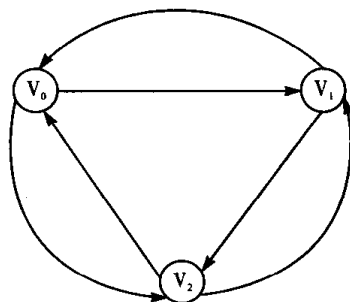


图 6.4 有向完全图

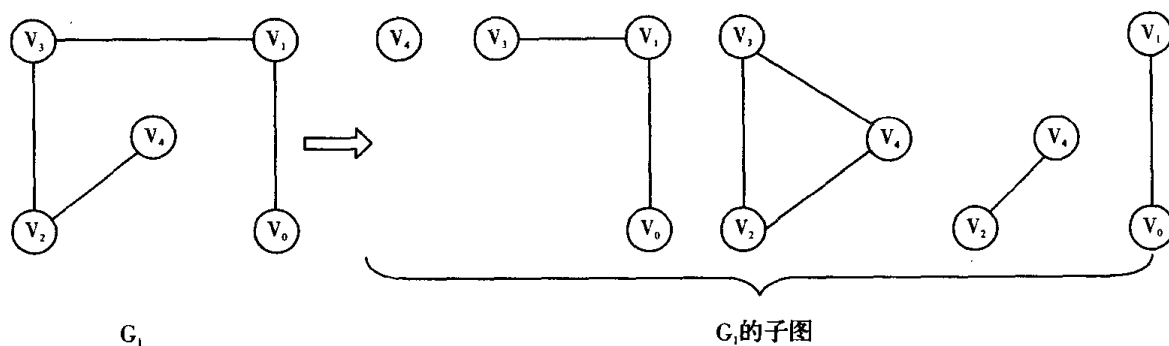
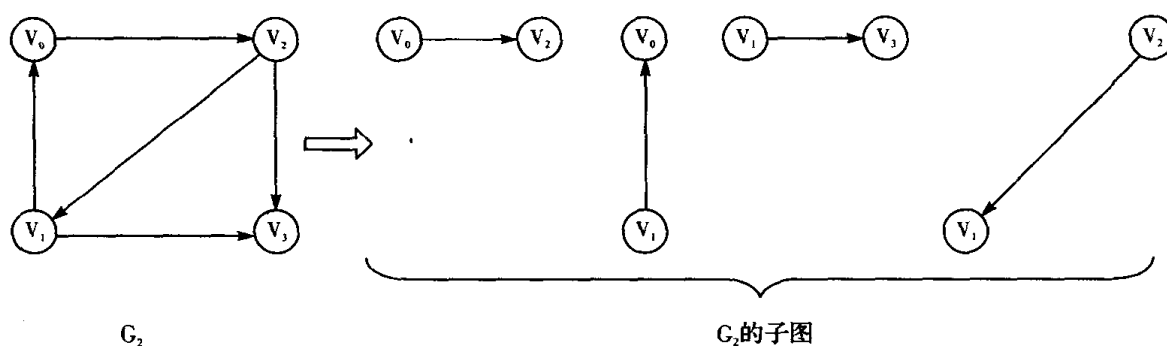
5. 子图 (Subgraph) 和路径 (Path)

对于图 $G = (V, E)$, $G' = (V', E')$, 若有 $V' \subseteq V, E' \subseteq E$, 则称 G' 是 G 的子图。

在无向图 G 中, 若存在顶点序列 $(V_p, V_1, V_2, \dots, V_n, V_q)$, 且 $(V_p, V_1), (V_1, V_2), \dots, (V_n, V_q)$ 属于 $E(G)$ 。则称顶点 V_p 到 V_q 存在一条路径, 路径上边的数目称为路径长度。

例如图 6.5 中的 G_1 图, (V_0, V_1, V_3, V_2) 是无向图 G_1 的一条路径, 其路径长度为 3。但 (V_0, V_1, V_2, V_3) 则不是图 G_1 的一条路径, 因为 (V_1, V_2) 不是图 G_1 的一条边。

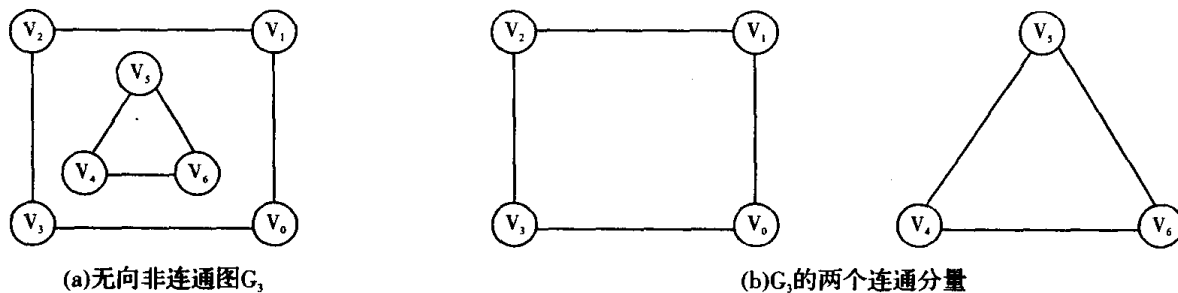
对于有向图, 其路径也是有向的, 路径由弧组成。例如图 6.6 中的 G_2 图, (V_0, V_2, V_1) 是有向图 G_2 的一条路径, 其路径长度为 2。而 (V_0, V_1, V_2) 不是图 G_2 的一条路径, 因为 $\langle V_0, V_1 \rangle$ 和 $\langle V_1, V_2 \rangle$ 不是图 G_2 的弧。

图 6.5 G_1 及其子图图 6.6 G_7 及其子图

6. 连通图(Connected Graph)、连通分量(Connected Component)、强连通图和强连通分量

在无向图 G 中,若从 V_i 到 V_j 有路径,则称 V_i 和 V_j 是连通的。若 G 中任意两顶点都是连通的,则称 G 是连通图。对于有向图,若有向图 G 中每一对不同顶点 V_i 、 V_j 之间都有从 V_i 到 V_j 和从 V_j 到 V_i 的路径,则称 G 为强连通图。

图 6.3 是连通图,图 6.4 是强连通图。而图 6.7 不是连通图,图 6.8 也不是强连通图。

图 6.7 无向非连通图 G_3 和它的两个连通分量

连通分量指的是无向图 G 中的极大连通子图。图 6.7 中的 G_3 有两个连通分量。强连通分量指的是有向图 G 中的极大强连通子图。图 6.8 中的 G_4 有两个强连通分量。

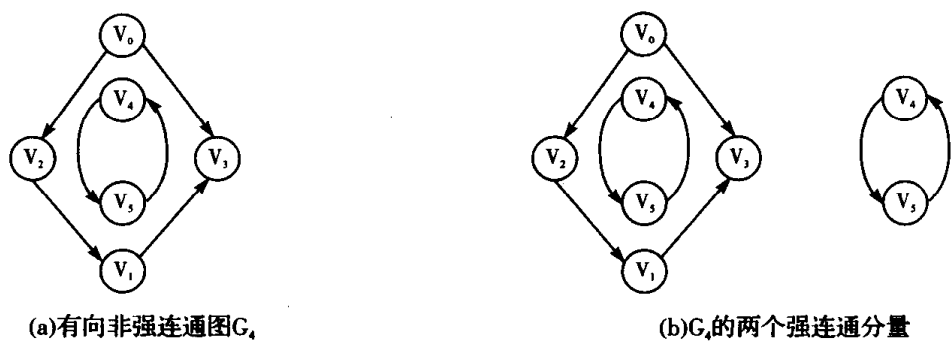


图 6.8 有向非强连通图 G_4 和它的两个强连通分量

7. 网 (Net)
若在图中的每条边上加上具有某种含义的权值,则这种带权值的图称为网。

6.1.2 图的存储结构

图的结构比较复杂,其存储结构也有多种形式,使用何种存储结构一般取决于具体应用。图的存储形式如下:

图的存储结构 { 邻接矩阵 } 两种最基本、最常用的存储结构
 { 邻接表
 { 十字链表
 { 多重链表

下面我们分别介绍一下邻接矩阵和邻接表。

1. 邻接矩阵

图的邻接矩阵表示是指用矩阵表示图中顶点间的相邻关系,也可用一个二维数组来表示。

设图 $G = (V, E)$, 其邻接矩阵定义如下:

$$A[i][j] = \begin{cases} 1 & \text{若 } (V_i, V_j) \in E(G) \text{ 或 } \langle V_i, V_j \rangle \in E(G) \\ 0 & \text{其他} \end{cases}$$

若图 $G = (V, E)$ 为带权的网,则其邻接矩阵定义如下:

$$A[i][j] = \begin{cases} W_{ij} & \text{若 } (V_i, V_j) \in E(G) \text{ 或 } \langle V_i, V_j \rangle \in E(G) \\ 0 & \text{若 } (V_i, V_j) \notin E(G) \text{ 或 } \langle V_i, V_j \rangle \notin E(G) \\ \infty & \text{其他} \end{cases}$$

图的邻接矩阵表示法如图 6.9 所示。

数据类型定义如下:

```
#define MAXVEX                                /* 顶点的最大个数即邻接矩阵的最大阶数 */
typedef char vextype;                         /* 顶点的数据类型 */
typedef int arctype;                          /* 边的数据类型 */
typedef struct                                /* 图的邻接矩阵结构定义 */
{
    vextype vex[MAXVEX];                     /* 顶点向量,存放顶点的信息 */
```

```

arctype arc[MAXVEX][MAXVEX];    /* 矩阵,存放边的信息 */
int vexnum,arcnum;               /* 图中顶点数和边数 */
} Mgraph;

```

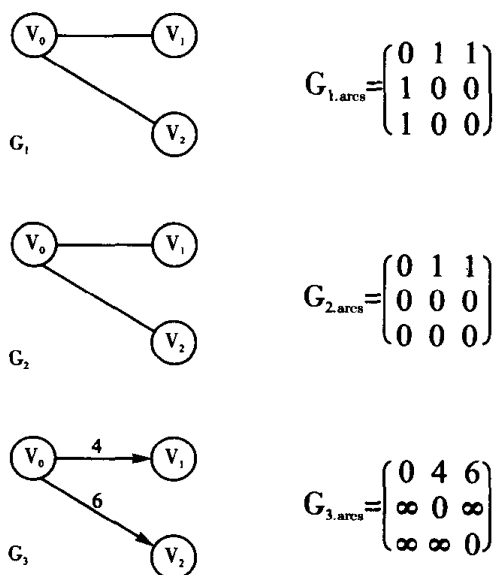


图 6.9 图的邻接矩阵存储示例

2. 邻接表

对于图 G 中的每个顶点 V_i , 把所有邻接 V_i 的顶点链接成一个单链表, 这个单链表就称为顶点 V_i 的邻接表。图的邻接表结点结构含有两部分: 邻接点域和链域。邻接点域存放某个顶点 V_i 的邻接表, 链域存放邻接表的所有表头结点。邻接表的结点结构如图 6.10 所示:

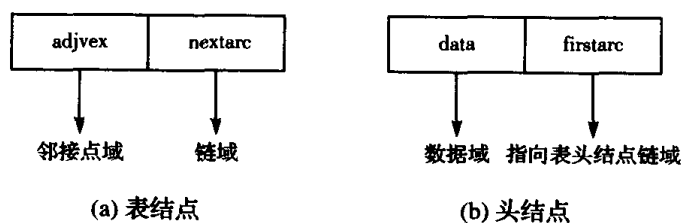
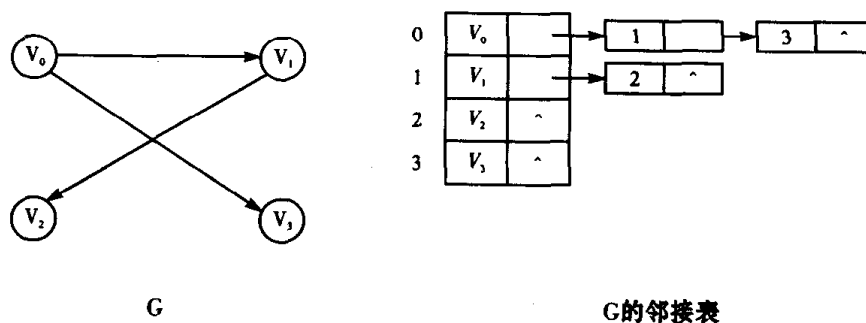


图 6.10 邻接表结点结构

图的邻接表示例如图 6.11 所示。

图 6.11 G 的邻接表示例

数据类型定义如下:

```
#define MAXVEX                /* 顶点的最大个数即邻接矩阵的最大阶数 */
typedef char vextype;        /* 顶点的数据类型 */
typedef struct node
{
    int adjvex;               /* 邻接点的序号 */
    struct node *nextarc;     /* 链域,指向下一个邻接点 */
} arcnode;                   /* 邻接表的表结点定义,即邻接点域定义 */
typedef struct
{
    vextype data;             /* 顶点数据域 */
    arcnode *firstarc;        /* 表头结点指针域 */
} vexnode;                   /* 邻接表的头结点定义,即链域定义 */
```

关于以上两种存储结构的说明:

(1) 邻接矩阵并非图的顺序存储结构(图没有顺序存储结构),它只是借助了数组这种数据类型来表示图中顶点间的相连关系。

(2) 图的邻接表不惟一,它与建立邻接表的算法和顶点对的输入顺序有关。

6.1.3 图的基本操作

1. 建立图的邻接矩阵(以无向图为例)

```
void create_Mgraph (Mgraph *g)
{
    int i,j,k,n,v1,v2;
    int vexnum,arcnum;
    scanf("%d",&n);           /* 输入顶点数 */
    g->vexnum=n;
    for(i=1;i<=n;i++)
        g->vex[i]=i;          /* 假设顶点信息为顶点序号 */
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            g->arc[i][j]=0;    /* 矩阵各元素赋初值0 */
    for(k=1;k<=arcnum;k++)
    {
        scanf("%d%d%d",&v1,&v2,&weight); /* 输入顶点对,顶点序号从1开始 */
        g->arc[v1][v2]=weight;
        g->arc[v2][v1]=weight;
    }
}
```

2. 建立图的邻接表(以有向图为例)

```
void create_adjlist(vexnode g)
{
    int i,v1,v2,vexnum,arcnum;
```

```

    arcnode *s;
    printf("输入顶点数:");
    scanf("%d",&vexnum);
    getchar();
    printf("输入顶点信息:");
    for(i=1;i<=vexnum;i++)
    {
        g[i].data=getchar();          /* 邻接表指针初始化 */
        g[i].firstarc=NULL;
    }
    printf("输入边数:");
    scanf("%d",&arcnum);
    printf("输入边的顶点对:");
    for(i=1;i<=arcnum;i++)
    {
        scanf("%d%d",&V1,&V2);      /* 输入边的顶点对,顶点序号从1开始 */
        s=(arcnode *)malloc(sizeof(arcnode)); /* 生成表结点 */
        s->adjvex=V2;
        s->nextarc=g[V1].firstarc;
        g[V1].firstarc=s;             /* 将s插到顶点V1的邻接表表头 */
    }
}

```

3. 图的深度优先搜索

图的深度优先搜索类似于树的先根遍历,具体过程是:首先访问出发顶点 V_0 ,接着选择一个与 V_0 相邻且未被访问过的顶点 V_i 访问,再从 V_i 开始进行深度优先搜索。当到达一个其所有相邻的顶点都已被访问过的顶点时,就从最后访问的顶点开始,依次向前退,回到尚有邻接顶点未曾访问过的顶点 V_k ,并从 V_k 开始进行深度优先搜索。这个过程进行到所有顶点都被访问到为止。递归算法如下所示:

```

void DFS(vexnode g,int v0)          /* 图用邻接表存储,v0为搜索起始结点 */
{
    int v;
    arcnode *p;
    printf("%c",g[v0].data);        /* 访问顶点 v0 */
    visit[v0]=1;                    /* 数组 visit 标记已访问过的顶点 */
    p=g[v0].firstarc;               /* p 指向 v0 对应的边表中的表头结点 */
    while(p)
    {
        v=p->adjvex;
        if (visit[v]==0)
            DFS(g,v);                /* 递归调用深度优先搜索算法 */
        p=p->nextarc;
    }
}

```

```
p = p -> nextarc;
```

4. 图的广度优先搜索

图的广度优先搜索类似于树的层次遍历,具体过程:首先访问出发顶点 V_0 ,然后访问与 V_0 相邻的全部顶点 $V_1, V_2, \dots, V_i$,再依次访问与 $V_1, V_2, \dots, V_i$ 邻接的所有未被访问过的顶点。依此类推,直到图中所有顶点都被访问到为止。非递归算法如下所示:

```
void BFS(vexnode g, int v0)                /* 图采用邻接表存储, v0 为搜索起始结点 */
{
    int u, w;
    arcnode *p;
    int q[MAXVEX], front = -1, rear = -1;    /* 设置队列并初始化 */
    printf("%c", g[v0].data);
    visit[v0] = 1;                          /* visit 数组标记已访问过的顶点 */
    q[ ++rear ] = v0;                        /* 访问过的顶点入队 */
    while (front != rear)
    {
        u = q[ ++front ];                  /* 队首元素出队 */
        p = g[u].firstarc;
        while(p)                            /* while 循环让同层顶点的相邻顶点依次出队列,再访问下层相邻顶点 */
        {
            w = p -> adjvex;
            if (!visit[w])
            {
                printf("%c", g[w].data);    /* 访问未访问过的邻接点 */
                visit[w] = 1;
                q[ ++rear ] = w;
            }
            p = p -> nextarc;
        }
    }
}
```

5. 最小生成树

连通图 $G = (V, E)$ 的生成树 (Spanning Tree) T 是一个包含 G 中所有顶点的子图,它是可连接所有顶点且路径惟一的树形结构,并且不会形成回路。

生成树是不惟一的,其中边的权值总和最小的生成树称为最小生成树 (Minimum Cost Spanning Tree)。主要应用于建立使用最少代价连接所有地点等实际问题中。

求最小生成树有两种著名算法:Kruskal 算法和 Prim 算法 (分别称为 K 氏、P 氏算法)。

Kruskal 算法:

Kruskal (克鲁斯卡尔) 算法是每次选取权值最小的边,然后检查是否形成回路,形成

回路的边不能取用。因为是由小到大取边以形成生成树,所以得到的生成树一定为最小生成树(MST)。设一个有 n 个顶点的无向连通图,其 Kruskal 算法的求解步骤如下:

(1) 将边的权值按由小到大排序。

(2) 从所有未遍历的边中取出最小权值的边,记录此边已遍历,检查是否形成回路:

① 若形成回路,此边不能加入 MST 中,回到步骤(2)。

② 若未形成回路,此边加入 MST 中,如果边数已达 $n-1$ 条则继续步骤(3),否则回到步骤(2)。

(3) 生成 MST,操作结束。

Kruskal 伪算法描述如下:

/* 输入:图 $G=(V,E)$, n 个顶点, $w(E_i)$ 为 E_i 上的权值 */

/* 输出: T , T 是图 G 的最小生成树 */

```
Kruskal ( )
{
    while ( (T 的边数 <= n-1) && (E 边的个数不为 0) )
    {
        从 E 中选出最小权值的边  $E_i$ 
        从 E 中删除  $E_i$ 
        if(  $E_i$  加入 T 中不会形成回路 )
            加入  $E_i$  到 T 中
    }
    if( T 的边数 < (n-1) )
        输出 "G 无最小生成树";
    else
        输出 T;
}
```

Prim 算法:

Prim(普里姆)算法可以避免回路的检查,算法是从某顶点 V 出发,列出顶点 V 所有邻接点的边,选择最小的边 (V,U) 加入到最小生成树中,然后标记 (V,U) 边已遍历,再从顶点 U 出发,在其所有连接边中(除 (U,V) 边之外),再选出最小的边加入到 MST。依此类推,直到找到 $n-1$ 条边,即可形成最小生成树。

设无向图 $G=(V,E)$, T_v 为最小生成树的顶点集合, E 是作用边集合, T_e 是最小生成树的边集合。Prim 算法步骤如下:

(1) 选择从某一顶点 V_1 开始。

(2) 将 V_1 的所有边加入 E 中,将 V_1 加入 T_v 中。

(3) 从 E 中找出最小权值的边 (V_1,U_1) 加入 T_e 中,并从 E 中删除 (V_1,U_1) 。

(4) 将 U_1 加入 T_v 中,如果 T_v 顶点数等于 n ,则跳到步骤(6)。

(5) 将 U_1 对应到 $\{V - T_v\}$ 顶点的所有边加入 E 中,回到步骤(3)。

(6) (T_v, T_e) 是最小生成树,结束。

Prim 的伪算法描述如下:

```
/* 输入:图  $G=(V,E)$ ,  $n$  个顶点,  $W(E_i)$  为  $E_i$  上的权值 */
/* 输出:  $T$ ,  $T$  是图  $G$  的最小生成树 */
```

```
Prim()
{
    Tv = V1
    加入顶点 V1 对  $\{V - Tv\}$  的所有边到 E 中
    while (Tv 的顶点数 < n)
    {
        选出 E 中权值最小的边  $E_i$  且删除之
         $V_i$  加入到 Tv 中, 边加入到 Te 中
        加入顶点 U 对  $\{V - Tv\}$  的所有边到 E 中
    }
    if (Tv 的顶点 < n)
        输出 "G 无最小生成树";
    else
        输出 T;
}
```

6. 最短路径问题

最短路径问题的典型应用是道路交通网的距离问题。

单源点的最短路径

采用 Dijkstra 算法的求解步骤如下:

- (1) 初始时, $s = \{V\}$, V 为源点, U 包含其他顶点;
- (2) 从 U 中选距离 V 最小的顶点 w , w 加入 s (该距离就是 v 到 w 的最短路径);
- (3) 以 w 为新的中间点, 修改 U 中各顶点距离。若经过 w 能缩短从源点 V 到顶点 U 的距离, 则修改顶点 U 的相关距离值;
- (4) 重复步骤 (2)、(3), 直到 s 中含有所有顶点, 算法结束。

Dijkstra 算法如下:

```
/* 输入:图  $G=(V,E)$ ,  $V = \{0, 1, \dots, n-1\}$  */
/* MAX 表示最大顶点数; MAXCOST 表示顶点间没有边 */
/* 输出:输出  $V_0$  到其他各顶点的最短距离 */
void Dijkstra (int cost[MAX][MAX], int n, int v0)
{
    int dist[MAX], path[MAX];
    int s[MAX];
    int mins, i, j, w;
    for (i = 0; i < n; i++)
    {
        dist[i] = cost[v0][i];           /* 距离初始值 */
        s[i] = 0;                       /* s[] 初始值 */
    }
}
```

```

s[v0] = 1;                                     /* 源点 v0 加入 s 中 */
for(i = 0; i < n; i++)                          /* 求出所有顶点的最短路径 */
{
    mins = MAXCOST;
    for(j = 0; j < n; j++)                      /* 选择不在 s 内且具有最小距离的顶点 w */
        if(s[j] == 0 && dist[j] < mins)
        {
            w = j;
            mins = dist[j];
        }
    s[w] = 1;                                    /* 顶点 w 加入 s 中 */
    for(j = 0; j < n; j++)                      /* w 作为中间点,修正不在 s 内的顶点的距离 */
        if(s[j] == 0)
        {
            d = dist[w] + cost[w][j];
            dist[j] = min(d, dist[j]);          /* 选择 d 与 dist[j] 中较小的赋值给 dist[j] */
        }
    }

printf("源点到其他各顶点的最短路径长度:\n");
for(i = 0; i < n; i++)                          /* 显示最短路径长度 */
    if(s[i] == 1)
        printf("从顶点 %d 到顶点 %d 的最短路径为 %d\n", v0, i, dist[i]);
    else printf("从顶点 %d 到顶点 %d 无路径\n", v0, i);
}

```

下面为 Dijkstra 算法的示例:

(1) 已知:

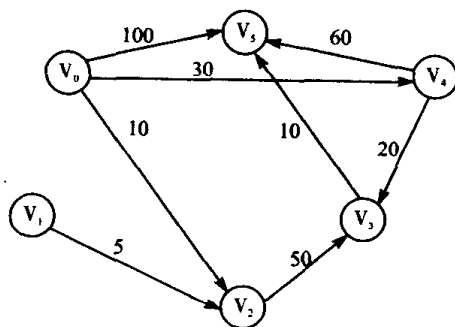


图 6.12 Dijkstra 算法示例图

(2) 求解:

从 V_0 到各终点的 dist 值和最短路径的求解过程

终点	I = 1	I = 2	I = 3	I = 4	I = 5
V_1	∞	∞	∞	∞	∞
V_2	10 (V_0, V_2)				
V_3	∞	60 (V_0, V_2, V_3)	50 (V_0, V_4, V_3)		
V_4	30 (V_0, V_4)	30 (V_0, V_4)			
V_5	100 (V_0, V_5)	100 (V_0, V_5)	90 (V_0, V_4, V_5)	60 (V_0, V_4, V_3, V_5)	
$W = V_j$	V_2	V_4	V_3	V_5	
S	(V_0, V_2)	(V_0, V_2, V_4)	(V_0, V_2, V_3, V_4)	(V_0, V_2, V_3, V_4, V_5)	

每对顶点的最短路径

每对顶点的最短路径可用 Floyd 算法, 主要思想是逐次按 $0, 1, 2, \dots, n-1$ 的次序加入顶点 0 、顶点 1 、 $\dots$ 、顶点 $n-1$ 作为转接点。计算各阶段时的最短路径, 我们以矩阵 A^{-1} 代表最原始的图邻接矩阵, A^0 代表加入顶点 0 的每对顶点的最短路径结果, 依此类推, 当我们求得 A^{n-1} 时, 即代表所有转接点都已处理过了, A^{n-1} 邻接数组即是 Floyd 算法的结果。

Floyd 算法如下:

```

/* 输入: 图  $G=(V, E)$ ,  $V=\{0, 1, \dots, n-1\}$  */
/* MAX 表示最大顶点数; MAXCOST 表示顶点间没有边 */
/* 输出: 任意两点  $i, j$  最短路径矩阵  $A$  */
void Floyd (int cost[MAX][MAX], int n)
{
    int A[MAX][MAX], path[MAX][MAX];
    int i, j, k;
    for(i=0; i<n; i++)                               /* 原始邻接数组 */
        for(j=0; j<n; j++)
        {
            A[i][j] = cost[i][j];
            path[i][j] = -1;
        }
    for(k=0; k<n; k++)                                /* 求出各顶点间的最短路径 */
        for(i=0; i<n; i++)
            for(j=0; j<n; j++)
                if(A[i][j] > A[i][k] + A[k][j])

```

```

    }
    A[i][j] = A[i][k] + A[k][j];
    path[i][j] = k;
}

printf("各顶点间最短路径长度:\n");          /* 显示最短路径长度 */
for(i=0;i<n;i++)
for(j=0;j<n;j++)
    if(A[i][j] != MAXCOST)
        printf("从顶点% d到顶点% d的最短路径为% d,转接点为% d\n",i,j,A[i][j],
            path[i][j]);
}

```

下面为 Floyed 的算法示例：

(1) 已知：

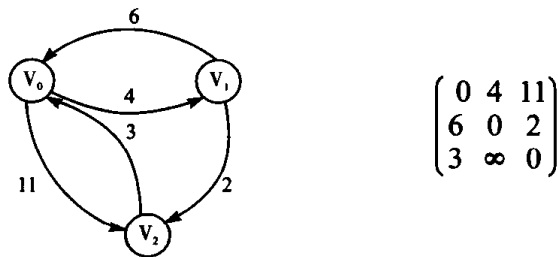


图 6.13 Floyed 算法示例图

(2) 求解：

$$A^{-1} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

$$A^0 = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

$$A^1 = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

$$A^2 = \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

$$p^{-1} = \begin{bmatrix} -1 & 1 & 2 \\ 0 & -1 & 2 \\ 0 & -1 & -1 \end{bmatrix}$$

$$p^0 = \begin{bmatrix} -1 & 1 & 2 \\ 0 & -1 & 2 \\ 0 & 0 & -1 \end{bmatrix}$$

$$p^1 = \begin{bmatrix} -1 & 1 & 1 \\ 0 & -1 & 2 \\ 0 & 0 & -1 \end{bmatrix}$$

$$p^2 = \begin{bmatrix} -1 & 1 & 1 \\ 2 & -1 & 2 \\ 0 & 0 & -1 \end{bmatrix}$$

7. 拓扑结构

图形结构除了可以表示顶点与边的关系、最短路径的关系之外，还可以表示某些事件的前后次序，即必须按照一定的次序，先完成某项或某几项事件后才能进行下一个事件。这样的应用很多，例如学校开设课程的承接问题、建筑工程的先后进度问题等。

AOV 网络是用顶点表示事件、用有向边表示事件间前后关系的有向图，用于处理拓扑序列问题。主要思想是维持顶点的前驱和后继关系，先找到没有前驱点的顶点，输出此

顶点,将其连接的边都删除,反复这个过程,直到所有的顶点都输出为止。

拓扑序列的伪算法描述如下:

```

/* 输入:图  $G=(V,E)$  */
/* 输出:G 的拓扑序列 */
topol()
{
    while (G 中仍有顶点)
    {
        If (所有的顶点都有前驱)
            Return /* 表示有回路,不能拓扑排序 */
        任选没有前驱点的顶点  $v$ 
        输出顶点  $v$ 
        删除顶点  $v$  及所有从顶点  $v$  发出的边
    }
}

```

8. 关键路径

通常利用 AOE 网来处理关键路径问题。AOE 网中,以顶点表示事件,弧表示活动,弧上的权值表示活动的开销(即活动持续时间)。AOE 网表示一项工程,用于研究完成整个工程至少需要多少时间和影响工程进度的关键活动问题。从源点到汇点的最长路径称为关键路径,关键路径上的所有活动都是关键活动。

关键路径的求解步骤如下:

(1)从源点 V_1 出发,对 AOE 网进行拓扑排序,求出各顶点的最早发生时间 $V_e[i]$ 。若网中有环,则无关键路径,算法结束。

(2)从汇点 V_n 出发,令 $V_L[n-1] = V_e[n-1]$,按照逆拓扑排序求出各顶点的最迟发生时间 V_L 。

(3)根据各顶点的 V_e 和 V_L 值,求出每个活动边 a_i 的最早开始时间 $e(i)$ 和最迟开始时间 $L(i)$ 。若某活动满足 $e(i) = L(i)$,则是关键活动。

6.2 实训篇

实训目的:

图是一种复杂的数据结构,我们前面研究的线性结构、树形结构都可以看成简单的图。图的应用十分广泛,很多问题都可以用图形结构来解决。例如:交通网、资源调度、网络拓扑分析等问题。

- 实训中理解图的概念和图的各种类型
- 掌握图的两种存储结构
- 理解实训项目中的算法要点,掌握图的实际应用

6.2.1 图的遍历问题

案例描述

已知 8 个城市的交通路线图如图 6.14 所示,从沈阳出发,要到其他 7 个城市推销产品。给出到达每一个城市的一组城市顺序。

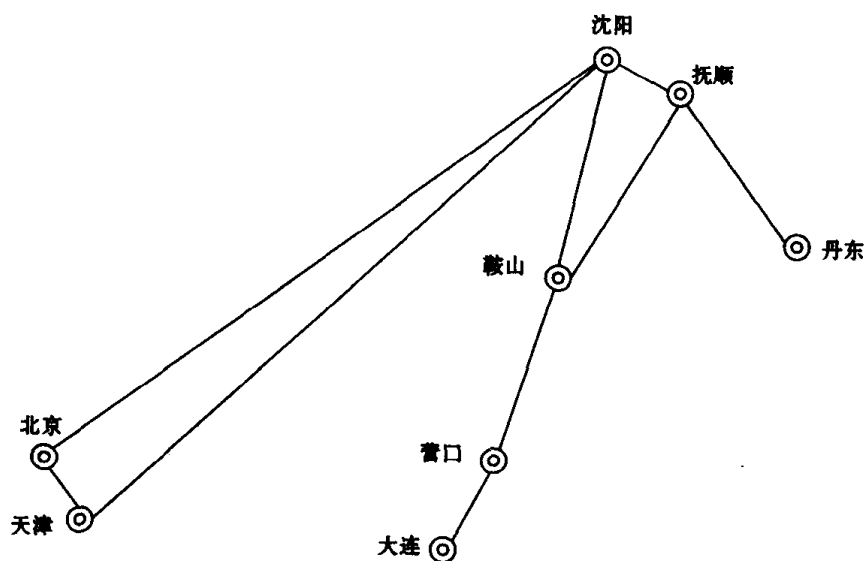


图 6.14 8 城市交通图

设计分析

该问题是求从沈阳出发到各城市的序列,就是求遍历各城市的方法,即为图的遍历问题。

设计方案

图的遍历有广度优先遍历和深度优先遍历两种算法。在这里我们利用数据结构中图的邻接矩阵存储方式以及图的深度优先遍历的算法来解决该问题。

结构定义:

```
typedef char vexntype[LEN];           /* 顶点数据类型 */
typedef int edgetype;                 /* 边数据类型 */
typedef struct
{
    vexntype vex[VEXN];
    edgetype arc[VEXN][VEXN];
    int vexn, arcn;
} Mgraph;                             /* 顶点个数和边数 */
/* 图的邻接矩阵表示结构 */
```

算法实现与分析:

```
#include <stdio.h>
#include <stdlib.h>
#define NULL 0
```

```

#define VEXN 30                                /* 顶点的最大个数 */
#define LEN 10                                  /* 城市名称的最大长度 */
typedef char vexntype[LEN];                    /* 顶点数据类型 */
typedef int edgetype;                           /* 边数据类型 */
typedef struct
{
    vexntype vex[VEXN];
    edgetype arc[VEXN][VEXN];
    int vexn, arcn;
} Mgraph;                                       /* 顶点个数和边数 */
int visit[VEXN];                               /* 图的邻接矩阵表示结构 */
Mgraph City_Graph();                           /* 遍历标志数组 */
void DFS_M();                                  /* 定义函数类型 */

Mgraph City_Graph()
{
    int i, j, k;
    Mgraph G;
    printf("\n Input vex number ");
    scanf("% d", &G.vexn);                    /* 输入顶点数目 */
    printf("Input edge number:");
    scanf("% d", &G.arcn);                     /* 输入连接各顶点的总边数 */
    printf("Input % d vex information such as Shenyang Beijing and so on:\n",
           G.vexn);
    for(i=0; i<=G.vexn; i++)
        scanf("% s", &G.vex[i]);
    for(i=0; i<vexn; i++)
        for(j=0; j<vexn; j++)
            G.arc[i][j]=0;
    for(k=0; k<G.arcn; k++)                    /* 输入各顶点距离 */
    {
        printf("Input edge % d such as i,j:", k+1);
        scanf("% d,% d", &i, &j);
        while (i<1 || i>G.vexn || j<1 || j>G.vexn) /* 输入各顶点距离 */
        {
            printf("Error vexn Number, input(i,j) again:");
            scanf("% d,% d", &i, &j);
        }
        G.arc[i-1][j-1]=1;
        G.arc[j-1][i-1]=1;
    }
}

```

```

    return G;
}
void DFS_G(Mgraph G,int i)                                /* 输出深度优先遍历序列 */
{
    int j;
    printf("%s",G.vex[i]);
    visit[i]=1;
    for(j=0;j<G.vexn;j++)
        if ((G.arc[i][j]==1)&&(!visit[j]))
            DFS_G(G,j);
}
main()
{
    Mgraph G;
    G=City_Graph();                                       /* 调用输入顶点和边的函数 */
    printf("\n 深度优先搜索序列为:\n");
    DFS_G(G,0);                                           /* 调用查询最短路径函数 */
}

```

运行测试:

< 输入 >

Input vex number:8

Input edge number:9

Input 8 vex information such as Shenyang Beijing and so on:

沈阳 北京 天津 大连 营口 鞍山 抚顺 丹东

Input edge 1 such as i,j:1,2

Input edge 2 such as i,j:1,3

Input edge 3 such as i,j:2,3

Input edge 4 such as i,j:4,5

Input edge 5 such as i,j:5,6

Input edge 6 such as i,j:6,7

Input edge 7 such as i,j:7,8

Input edge 8 such as i,j:1,6

Input edge 9 such as i,j:1,7

< 输出 >

深度优先搜索序列为:

沈阳 北京 天津 鞍山 营口 大连 抚顺 丹东

6.2.2 所有顶点对最短路径问题**案例描述**

医院选址:4个村庄之间的交通图如图6.15所示,村庄之间的距离如图中各边上的权值所示。现在要从这4个村庄中选择一个村庄建一所医院,问这所医院应建在哪个村

庄,才能使离医院最远的村庄到医院最近。

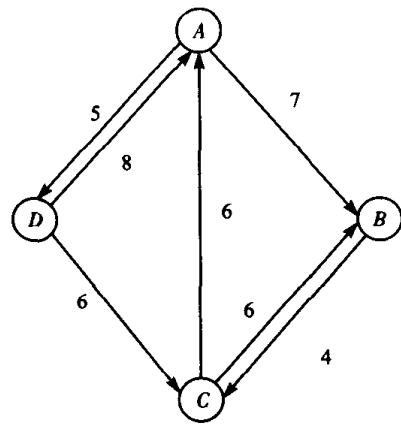


图 6.15 4 村庄交通图

设计分析

用有向加权图表示的交通图中,有向边 $\langle V_i, V_j \rangle$ 表示第 i 个村庄和第 j 个村庄之间有道路,边上的权表示这条道路的长度。该问题实质上是求解任意两顶点间的最短路径问题。

设计方案

- (1)运用 Floyed 算法求出有向图中各顶点对的最短路径。
- (2)根据步骤(1)结果,求出每个顶点到其他顶点的最短路径的最大值。
- (3)最大值最小的顶点作为医院所在村庄。

结构定义:

```
typedef char vextype; /* 顶点数据类型 */
typedef int edgetype; /* 边数据类型 */
typedef struct
{
    vextype vex[MAXVEX];
    edgetype arc[MAXVEX][MAXVEX];
    int vexnum, arcnum;
} Mgraph; /* 顶点数和边数 */
/* 图的邻接矩阵表示结构定义 */
```

算法实现与分析:

```
#include <stdio.h>
#include <stdlib.h>
#define NULL 0
#define MAXVEX 20 /* 图的最大顶点个数 */
#define MAXINT 1000 /* 极大值 */
typedef char vextype; /* 顶点数据类型 */
typedef int edgetype; /* 边数据类型 */
typedef struct
{
    vextype vex[MAXVEX];
```

```

    edgetype arc[MAXVEX][ MAXVEX];
    int vexnum,arcnum;                                /* 顶点数和边数 */
    Mgraph;                                             /* 图的邻接矩阵表示结构定义 */
    int Path(Mgraph G);
    Mgraph Create_Mgraph();
    void shortpath(Mgraph G,edgetype A[MAXVEX][ MAXVEX]);

void shortpath(Mgraph G,edgetype A[MAXVEX][ MAXVEX])
{
    /* 多源最短路径的 Floyed 算法,结果矩阵为 A */
    int k=0,i,j;
    for(i=0;i<G.vexnum;i++)
        for(j=0;j<G.vexnum;j++)
            A[i][j]=G.arcs[i][j];
    for(k=0;k<G.vexnum;k++)
    {
        for(i=0;i<G.vexnum;i++)
            for(j=0;j<G.vexnum;j++)
                if (A[i][j]>A[i][k]+A[k][j])
                    A[i][j]=A[i][k]+A[k][j];        /* 修正最短路径 */
    }
}

int Path(Mgraph G)
{
    /* 求图 G 中到其他顶点最短路径的最大值最小的顶点,返回顶点的下标 */
    int i,j,k,ki;
    int maxindex[MAXVEX];
    edgetype A[MAXVEX][MAXVEX];
    shortpath(G,A);
    for(j=0;j<G.vexnum;j++)
    {
        /* 求矩阵各列的最大值的行下标 maxindex[j] */
        k=0;
        for(i=1;i<G.vexnum;i++)
            if(A[i][j]>A[k][j])
                k=i;
        maxindex[j]=k;
    }
    k=0;ki=maxindex[k];
    for(j=1;j<G.vexnum;j++)
    {
        /* 求矩阵各列的最大值最小的列下标 k */
        i=maxindex[j];
        if(A[ki][k]>A[i][j])

```

```

    }
    k = j; ki = maxindex[k];
    }
}
return k;
}

main()
{
    int i;
    Mgraph G;
    G = Create_Mgraph();

    /* 求图 G 中到其他顶点最短路径的最大值最小的顶点 i */
    i = Path(G);
    printf("Solution is: % c", G.vex[i]);
}

Mgraph Create_Mgraph() /* 创建有向图的邻接矩阵算法 */
{
    int i, j, k;
    edgetype v; /* 边的权值 */
    Mgraph G;
    printf("\nInput vex number:");
    scanf("% d", &G.vexnum); /* 顶点数 */
    printf("Input edge number:");
    scanf("% d", &G.arcnum);
    printf("Input % d vex information such as ABCD: ", G.vexnum);
    scanf("% s", G.vex); /* 顶点的字符表示 */
    for(i = 0; i < MAXVEX; ++i)
    {
        for(j = 0; j < MAXVEX; ++j)
        {
            G.arcs[i][j] = MAXINT; /* 置初始距离值 */
            G.arcs[i][i] = 0; /* 顶点到本身的距离值为 0 */
        }
    }
    for(k = 1; k <= G.arcnum; k++)
    {
        printf("Input edge % d such as i, j: ", k); /* 输入对应边起点序号和终点序号 */
        scanf("% d, % d", &i, &j);
        while(i < 1 || i > G.vexnum || j < 1 || j > G.vexnum)
        {
            printf("Error vex number, try(i, j) again");
            scanf("% d, % d", &i, &j);
        }
    }
}

```

```

        printf("Input edge value:");
        scanf("%d",&v);
        G.arc[i-1][j-1]=v;
    }
    return G;
}

```

/* 输入边的权值 */
/* 有向图矩阵存储表示 */

运行测试**<输入>**

```

Input vex number:4
Input edge number:7
Input 4 vex information such as ABCD:BCAD
Input edge 1 such as i,j:1,2
Input edge value:4
Input edge 2 such as i,j:2,3
Input edge value:6
Input edge 3 such as i,j:3,4
Input edge value:5
Input edge 4 such as i,j:4,2
Input edge value:6
Input edge 5 such as i,j:4,3
Input edge value:8
Input edge 6 such as i,j:3,1
Input edge value:7
Input edge 7 such as i,j:2,1
Input edge value:6

```

<输出>

```

Solution is:A

```

6.3 实训项目

项目内容

某公司拟建一个局域网,需要连接6栋楼(1,2,3,4,5,6),其地理分布如图6.16所示,各边上的权值代表楼宇间距离。

设计要求

根据图6.16中的楼宇分布,设计出布线造价最小的局域网络图,要求连接所有的顶点,并且总的布线费用最低。

项目提示

要想布线造价最小,可将图中各顶点间的距离看作其费用。这样就可以用Prim算法构造一棵最小生成树来解决问题。

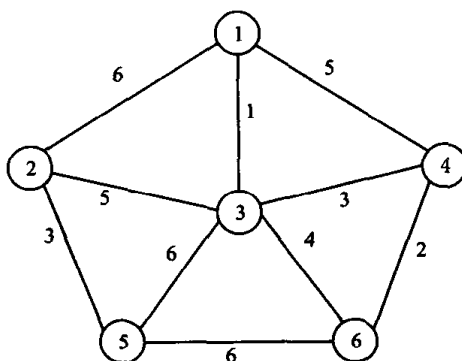


图 6.16 楼宇分布图

6.4 基本题

6.4.1 选择题

1. 对某个无向图的邻接矩阵来说,_____。

- A. 第 i 行上的非零元素个数和第 i 列的非零元素个数一定相等
- B. 矩阵中的非零元素个数等于图中的边数
- C. 第 i 行、第 i 列上非零元素总数等于顶点 V_i 的度数
- D. 矩阵中非全零行的行数等于图中的顶点数

2. 已知一个图,如图 6.17 所示,若从顶点 V_1 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;按广度搜索法进行遍历,则可能得到的一种顶点序列为 ②。

① A. $V_1, V_2, V_3, V_5, V_6, V_4$

B. $V_1, V_3, V_5, V_6, V_2, V_4$

C. $V_1, V_5, V_2, V_3, V_6, V_4$

D. $V_1, V_5, V_4, V_6, V_3, V_2$

② A. $V_1, V_2, V_3, V_4, V_5, V_6$

B. $V_1, V_2, V_3, V_5, V_6, V_4$

C. $V_1, V_5, V_2, V_3, V_6, V_4$

D. $V_1, V_3, V_6, V_4, V_5, V_2$

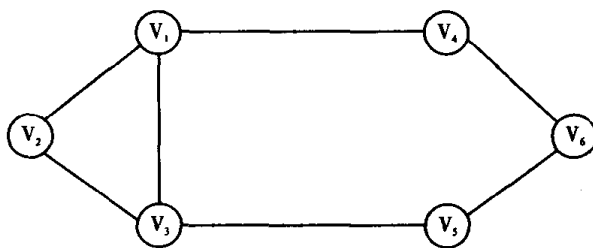


图 6.17 练习 2 示意图

3. 已知一有向图的邻接表存储结构如图 6.18 所示。

(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_____。

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

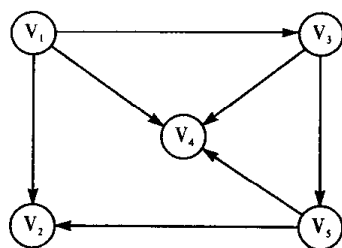


图 6.18 练习 3 示意图

(2) 根据有向图的广度优先遍历算法, 从顶点 V_1 出发, 所得到的顶点序列是_____。

- 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

(3) 求每个顶点的度, 有向图还要求入度和出度。

(4) 给出图的邻接矩阵。

(5) 给出图的邻接表。

4. 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_____。

- A. 先序遍历 B. 中序遍历 C. 后序遍历 D. 按层遍历

5. 以下说法中不正确的是_____。

- A. 无向图中的极大连通子图称为连通分量
B. 连通图的广度优先搜索中一般要采用队列来暂存刚访问过的顶点
C. 图的深度优先搜索中一般要采用栈来暂存刚访问过的顶点
D. 有向图的遍历不可采用广度优先搜索方法

6. 关键路径是事件结点网络中_____。

- A. 从源点到汇点的最长路径 B. 从源点到汇点的最短路径
C. 最长的回路 D. 最短的回路

7. 在一个图中, 所有顶点的度数之和等于所有边数的_____倍。

- A. $1/2$ B. 1 C. 2 D. 4

8. 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_____倍。

- A. $1/2$ B. 1 C. 2 D. 4

9. 一个有 n 个顶点的无向图最多有_____条边。

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

10. 具有四个顶点的无向完全图有_____条边。

- A. 6 B. 12 C. 16 D. 20

11. 具有 6 个顶点的无向图至少应有_____条边才能确保是一个连通图。

- A. 5 B. 6 C. 7 D. 8

12. 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_____条边。

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

13. 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

_____。

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

14. 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. $n+e$

6.4.2 填空题

1. 一个图的_____表示法是惟一的,而_____表示法是不惟一的。
2. 在有 n 个顶点的有向图中,每个顶点的度最大可达_____。
3. 对 n 个顶点的连通图来说,它的生成树一定有_____条边。
4. 设有向图 G 如图 6.19 所示。

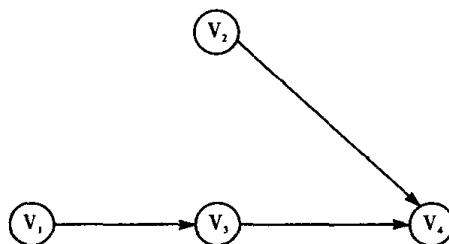


图 6.19 练习 4 示意图

- 写出所有的拓扑序列_____;
 添加边_____后,则仅可能有惟一的拓扑序列。
5. 对于图 6.20,按照迪杰斯特拉 (Dijkstra) 算法求出顶点 0 到其他各顶点的最短路径。

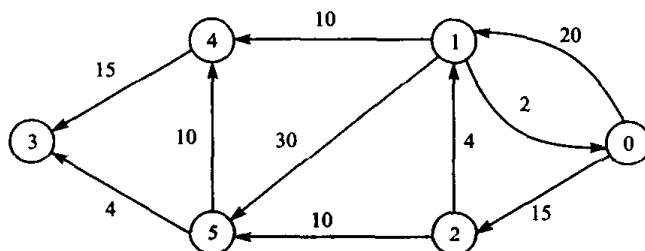


图 6.20 练习 5 示意图

$0 \rightarrow 1$: _____; $0 \rightarrow 2$: _____; $0 \rightarrow 3$: _____; $0 \rightarrow 4$: _____;
 $0 \rightarrow 5$: _____。

6. 建立无向图的邻接矩阵

```
void Create_Mgraph (Mgraph * g)
{
    int i,j,k,n,v1,v2;
    int vexnum,arcnum;
    scanf("%d",&n);
```

```

g->vexnum = n;
for(i=1; i<=n; i++)
    g->vex[i] = ①;
for(i=1; i<=n; i++)
    for(j=1; j<=n; j++)
        ② = 0;
for(k=1; k<=arcnum; k++)
{
    scanf("%d %d %d", &v1, &v2, &weight);
    g->arc[v1][v2] = weight;
    g->arc[v2][v1] = weight;
}
}

```

7. 图的深度优先遍历算法

```

void DFS(vexnode g, int v0)
{
    int v;
    arcnode *p;
    printf("%c", g[v0].data);
    visit[v0] = 1;
    p = g[v0].firstarc;
    while(p)
    {
        v = p->adjvex;
        if (visit[v] == 0)
            _____;
        p = p->nextarc;
    }
}

```

6.5 习题解析

1. 对于图 6.21 和图 6.22, 分别求:

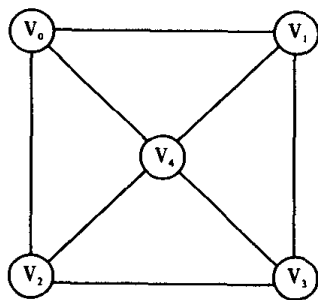


图 6.21 无向图

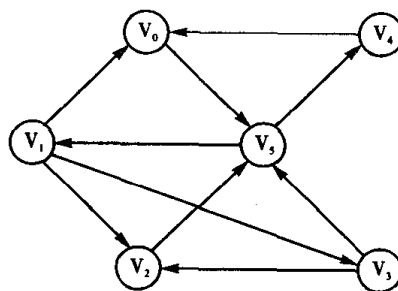


图 6.22 有向图

- (1) 每个顶点的度,有向图还要求入度和出度。
- (2) 给出一条从 V_0 到 V_3 的简单路径。
- (3) 给出图的邻接矩阵。
- (4) 给出图的邻接表。

解 对于图 6.21,

- (1) 每个顶点的度分别为: V_0, V_1, V_2, V_3 均为 3, V_4 为 4。
- (2) 从 V_0 到 V_3 的一条简单路径为: $(V_0, V_1, V_4, V_2, V_3)$ 。
- (3) 图的邻接矩阵: 图 6.21 的邻接矩阵为 A_1 。

$$A_1 = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

- (4) 图的邻接表(某顶点的邻接点按序号大小以队列方式加入)如图 6.23 所示。

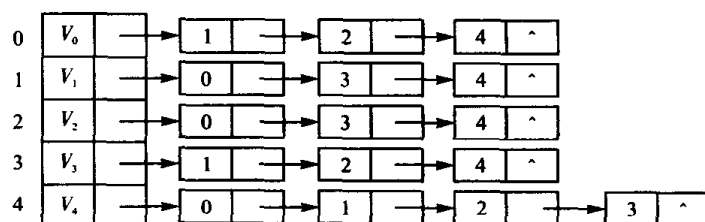


图 6.23 图 6.21 的邻接表

对于图 6.22

- (1) 每个顶点的度,入度和出度分别为:

顶点	度	入度	出度
V_0	3	2	1
V_1	4	1	3
V_2	3	2	1
V_3	3	1	2
V_4	2	1	1
V_5	5	3	2

- (2) 从 V_0 到 V_3 的一条简单路径为: (V_0, V_5, V_1, V_3) 。
- (3) 图的邻接矩阵: 图 6.22 的邻接矩阵为 A_2 。

$$A_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

(4) 图的邻接表(某顶点的邻接点按序号大小以队列方式加入)如图 6.24 所示。

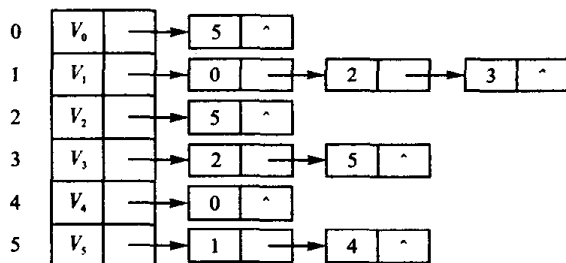


图 6.24 图 6.22 的邻接表

2. 对于图 6.21 和图 6.22, 分别求:

(1) 从顶点 0 出发, 采用深度优先搜索算法进行遍历所得到的搜索序列及其生成树。

(2) 从顶点 0 出发, 采用广度优先搜索算法进行遍历所得到的搜索序列及其生成树。

解 对于图 6.21, 因为一个图的深度优先搜索序列 (DFS 序列) 和广度优先搜索序列 (BFS 序列) 可能不惟一, 它与算法的存储结构密切相关, 所以, 本题解的依据是题目 1 中的第 (4) 小题图 6.21 的邻接表存储结构。

(1) 从顶点 V_0 出发, 采用深度优先搜索算法进行遍历所得到的搜索序列为: $V_0 V_1 V_3 V_2 V_4$ 。

从顶点 V_0 出发, 采用深度优先搜索算法得到的生成树如图 6.25 所示。

(2) 从顶点 V_0 出发, 采用广度优先搜索算法进行遍历所得到的搜索序列为: $V_0 V_1 V_2 V_4 V_3$ 。

从顶点 V_0 出发, 采用广度优先搜索算法得到的生成树如图 6.26 所示。

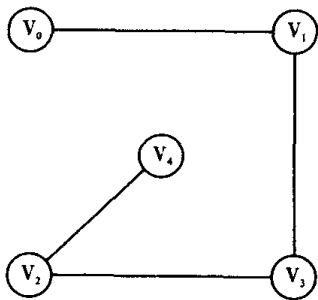


图 6.25 采用深度优先搜索算法得到的生成树

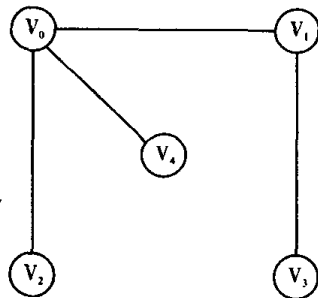


图 6.26 采用广度优先搜索算法得到的生成树

对于图 6.22, 同样, 本题解的依据是题目 1 中的第 (4) 小题图 6.22 的邻接表存储结构。

(1) 从顶点 V_0 出发, 采用深度优先搜索算法进行遍历所得到的搜索序列为: $V_0 V_5 V_1 V_2 V_3 V_4$ 。

从顶点 V_0 出发, 采用深度优先搜索算法进行遍历所得到的生成树如图 6.27 所示。

(2) 从顶点 V_0 出发, 采用广度优先搜索算法进行遍历所得到的搜索序列为: $V_0 V_5 V_1 V_4 V_2 V_3$ 。

从顶点 V_0 出发,采用广度优先搜索算法进行遍历所得到的生成树如图 6.28 所示。

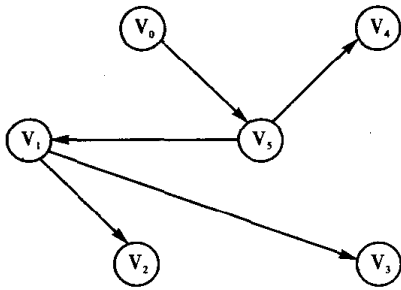


图 6.27 采用深度优先搜索算法得到的生成树

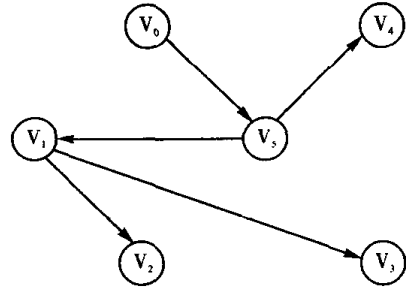


图 6.28 采用广度优先搜索算法得到的生成树

3. 对于图 6.29 所示的邻接表,写出从顶点 V_0 出发的深度优先搜索序列和从顶点 V_0 出发的广度优先搜索序列。

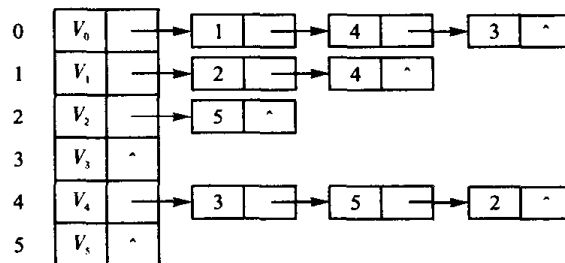


图 6.29 邻接表

解 从顶点 V_0 出发的深度优先搜索序列为: $V_0V_1V_2V_5V_4V_3$ 。

从顶点 V_0 出发的广度优先搜索序列为: $V_0V_1V_4V_3V_2V_5$ 。

4. 对于图 6.30,试写出两种拓扑有序序列。

解 两种拓扑有序序列如下:(1) $V_0V_1V_4V_5V_2V_3$;

(2) $V_0V_1V_5V_4V_2V_3$ 。

5. 对于图 6.31,试利用克鲁斯卡尔算法(Kruskal)求图的最小生成树,并写出其构造过程。

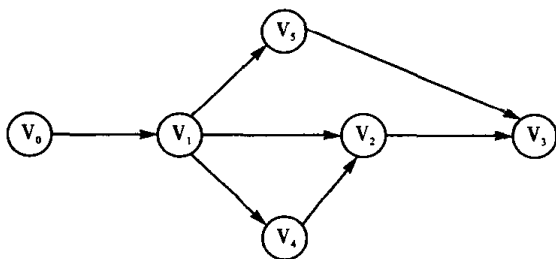


图 6.30 第4题示意图

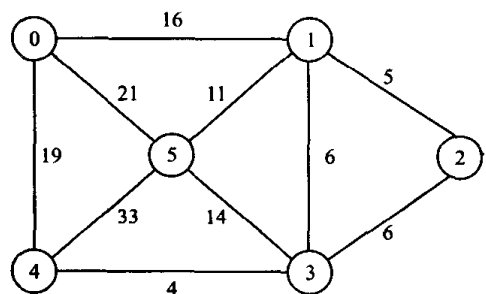


图 6.31 第5题示意图

解 利用克鲁斯卡尔算法,图 6.31 的最小生成树构造过程如图 6.32 所示。

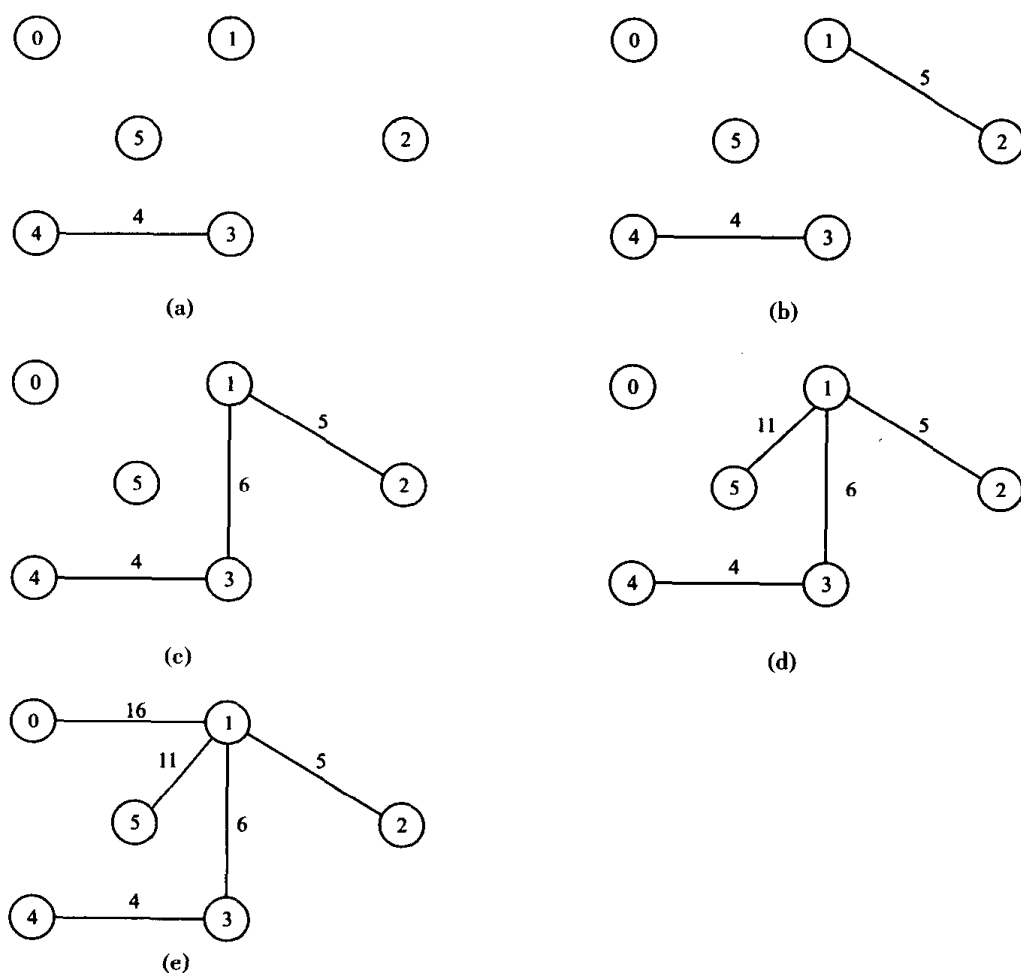


图 6.32 最小生成树构造过程

6. 对于图 6.33, 按迪杰斯特拉 (Dijkstra) 算法求从顶点 0 到其他各顶点的最短路径, 并给出辅助数组中值的变化过程。

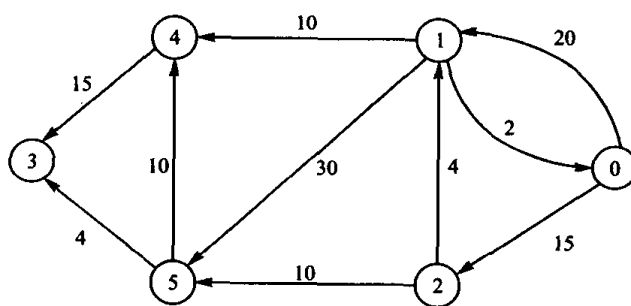


图 6.33 第 6 题示意图

解 图 6.33 的邻接矩阵 A 如下:

$$A = \begin{bmatrix} \infty & 20 & 15 & \infty & \infty & \infty \\ 2 & \infty & \infty & \infty & 10 & 30 \\ \infty & 4 & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 15 & \infty & \infty \\ \infty & \infty & \infty & 4 & 10 & \infty \end{bmatrix}$$

用迪杰斯特拉算法求从顶点 0 到其他各顶点的最短路径的过程如图 6.34 所示。

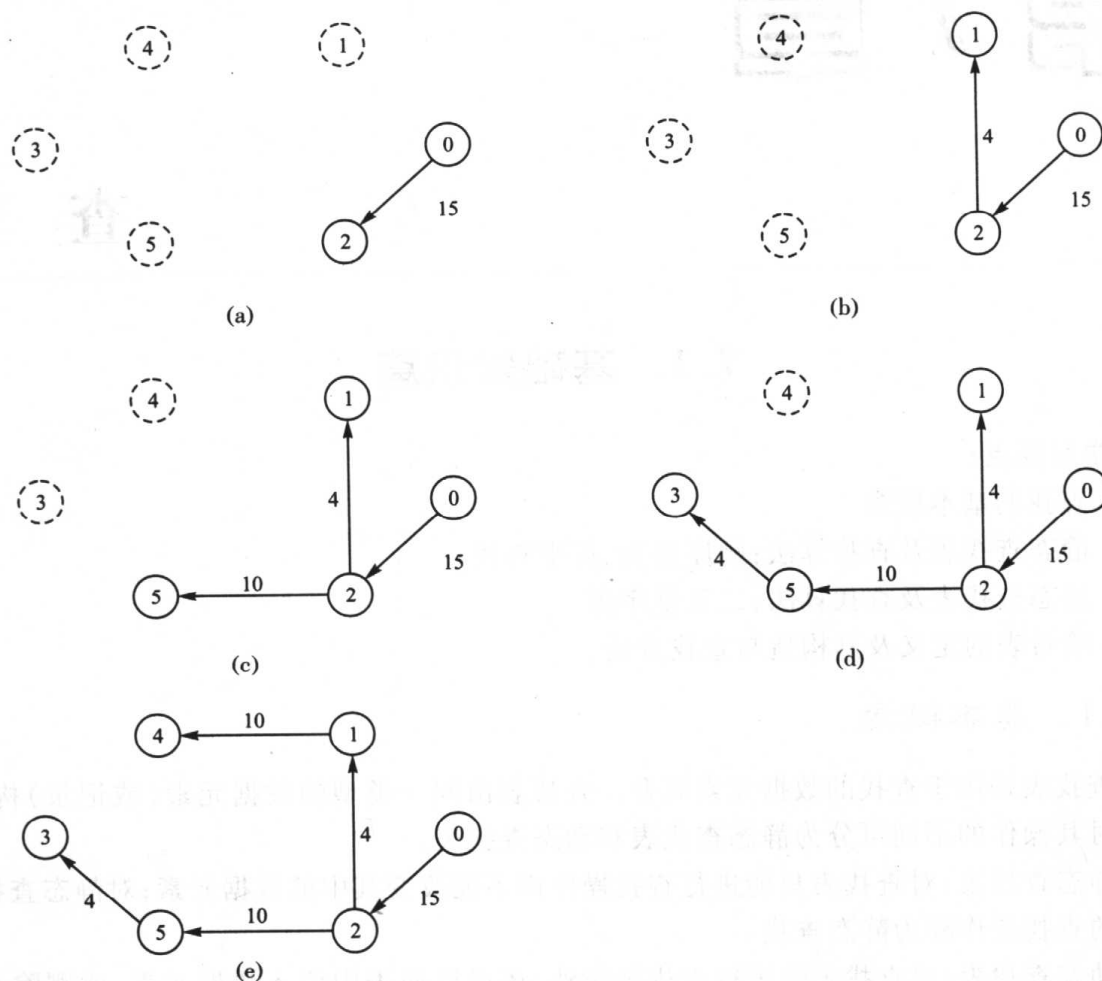


图 6.34 求最短路径示意图

辅助数组 $dist[]$ 初始化值由邻接矩阵 A 的第一行获得。

顶点 0 到其他各顶点的最短路径 (S) 及各辅助数组中值的变化过程如图 6.35 所示。

$S[]$		$dist[]$						$path[]$					
		0	1	2	3	4	5	0	1	2	3	4	5
{0}	$S[0] = 1$	∞	20	15	∞	∞	∞	0	0				
{0,2}	$S[2] = 1$	∞	19	15	∞	∞	25		2	0			2
{0,2,1}	$S[1] = 1$	∞	19	15	∞	29	25		2	0		1	2
{0,2,1,5}	$S[5] = 1$	∞	19	15	29	29	25		2	0	5	1	2
{0,2,1,5,3}	$S[3] = 1$	∞	19	15	29	29	25		2	0	5	1	2
{0,2,1,5,3,4}	$S[4] = 1$	∞	19	15	29	29	25		2	0	5	1	2

图 6.35 最短路径变化过程示意图

第7章

查 找

7.1 基础知识篇

学习要点:

- 查找的基本概念
- 静态查找表及查找算法:顺序查找、折半查找
- 动态查找表及查找算法:二叉排序树
- 哈希表的定义及其构造与查找方法

7.1.1 基本概念

查找表是用于查找的数据元素集合。查找表由同一类型的数据元素(或记录)构成,根据对其操作的不同可分为静态查找表和动态查找表。

静态查找表:对查找表只能进行查找操作而不能改变其中的数据元素;对静态查找表进行的查找操作称为静态查找。

动态查找表:对查找表除进行查找操作外,还可以向表中插入数据元素,或删除表中数据元素;对动态查找表进行的查找操作称为动态查找。

7.1.2 静态查找

1. 顺序查找

(1) 顺序查找的基本思想

顺序查找是一种最简单的查找方法。其基本思想是将查找表看作是一个线性表,可以是顺序表,也可以是链表,用查找条件中给定的关键字值依次与查找表中数据元素的关键字值进行比较,若某个记录的关键字值与给定值相等,则查找成功,返回该记录的存储位置,反之查找失败,返回查找失败标志。

(2) 顺序表的顺序查找

下面是顺序表的类型定义:

```
#define MAXSIZE 100                /* 用于定义表的空间大小 */
typedef struct {
    KeyType key;                    /* 关键字域 */
    :                               /* 其他域 */
} ElemType;
```

静态查找表是数据元素的线性表,可以采用顺序存储结构,也可以采用链式存储结构。

顺序表的存储结构定义:

```
typedef struct {
    ElemType *elem;           /* 顺序表的基址 */
    int length;               /* 表长度 */
} SSTable;
```

下面是顺序表的查找算法:

```
int Search_Seq(SSTable ST,KeyType k)
/* 在顺序表 ST 中查找关键字值等于 k 的记录,若找到,则函数返回值为该元素的位置;否则,返回 0 */
{
    ST.elem[0].key = k;       /* "哨兵" */
    for(i = ST.length; ST.elem[i].key != k; i--); /* 从后向前查找 */
    return i;                 /* 返回元素在顺序表中的位置,若没找到,i 为 0 */
}
```

2. 折半查找

(1) 折半查找的基本思想

折半查找要求查找表用顺序存储结构存放且各数据元素按关键字有序(升序或降序)排列,也就是说折半查找只适用于对有序顺序表进行操作。

折半查找的基本思想是:首先以整个查找表作为查找范围,用查找条件中给定值 k 与中间位置数据元素的关键字比较,若相等,则查找成功,否则,根据比较结果缩小查找范围,如果 k 的值小于关键字的值,根据查找表的有序性可知查找的数据元素只有可能在表的前半部分(设表升序排列),即在左半部分子表中,所以继续对左子表进行折半查找;若 k 的值大于中间结点的关键字值,则可以判定查找的数据元素只有可能在表的后半部分,即在右半部分子表中,所以应该继续对右子表进行折半查找。每进行一次折半查找,要么查找成功,结束查找,要么将查找范围缩小一半,如此重复,直到查找成功或查找范围缩小为空即查找失败为止。

(2) 折半查找算法

假设查找表存放在数组 $a[n]$ 中,且升序排列,查找关键字值为 k 。

折半查找的主要步骤为:

(a) 置初始查找范围: $low = 1, high = n$;

(b) 求查找范围中间项: $mid = (low + high) / 2$;

(c) 将指定的关键字值 k 与中间项 $a[mid].key$ 比较:

若相等,查找成功;

若小于,查找范围的低端数据元素指针 low 不变,高端数据元素指针 $high$ 更新为 $mid - 1$;

若大于,查找范围的高端数据元素指针 $high$ 不变,低端数据元素指针 low 更新为 $mid + 1$;

(d) 重复步骤(b)、(c),直到查找成功或查找范围为空($low > high$)即查找失败为止;

(e) 如果查找成功, 返回找到元素的存放位置, 即当前的中间项位置指针 `mid`; 否则返回查找失败标志。

其算法如下:

```
int Search_Bin(SSTable ST, KeyType key)
/* 在有序表 ST 中折半查找关键字等于 key 的元素, 若找到, 则函数返回该元素的位置; 否则, 返回 0 */
{
    low = 1; high = ST.length;          /* 设置查找上、下界初值 */
    while(low <= high)
    {
        mid = (low + high) / 2;
        if(ST.elem[mid].key == key)
            return mid;                  /* 查找成功, 返回元素位置 */
        else if(key < ST.elem[mid].key)
            high = mid - 1;               /* 继续在前半区查找 */
        else low = mid + 1;               /* 继续在后半区查找 */
    }
    return 0;                             /* 查找不到, 返回 0 */
}
```

7.1.3 动态查找

1. 二叉排序树

(1) 二叉排序树的概念

二叉排序树是一种常用的动态查找表, 它是一棵二叉树, 或者为空, 或者具有如下性质:

- (a) 任一非终端结点若有左孩子, 则该结点的关键字值大于其左孩子结点的关键字值;
- (b) 任一非终端结点若有右孩子, 则该结点的关键字值小于其右孩子结点的关键字值。

对一棵二叉排序树进行中序遍历, 可以得到一个按关键字排序的有序序列, 这是二叉排序树的一个重要特征。

(2) 二叉排序树的查找

从排序树的结构定义中可以看到: 一棵非空二叉排序树中根结点的关键字值大于其左子树上所有结点的关键字值, 而小于其右子树上所有结点的关键字值, 所以在二叉排序树中查找一个关键字值为 k 的结点的基本思想是: 用给定值 k 与根结点关键字值比较, 如果相等, 则查找成功, 如果 k 小于根结点的值, 则要找的结点只可能在左子树中, 所以继续在左子树中查找, 否则将继续在右子树中查找, 依此方法查找下去, 直至查找成功或查找失败为止。二叉排序树查找的过程描述如下:

- (a) 若二叉树为空树, 则查找失败;
- (b) 将给定值 k 与根结点的关键字值比较, 若相等, 则查找成功;

(c) 若给定值 k 小于根结点的关键字值,则在左子树中继续查找;

(d) 否则,在右子树中继续查找。

二叉排序树查找过程的描述是一个递归过程,若用链式存储结构存储,其查找操作的递归算法如下所示:

```
BiTree SearchBST( BiTree T , KeyType key)
{ /* 在根指针为 T 的二叉排序树上查找一个关键字值为 k 的结点,
   若查找成功则返回指向该结点的指针,否则返回空指针 * /
   if ( T == NULL || T->data == key) return T;
   /* 若查找到,则返回的 T 为指向该结点的指针,若没查找到,返回的 T 为 NULL * /
   else if ( k < T->data) return SearchBST(T->lchild,key);
                                   /* 在左子树中继续搜索 * /
   else return SearchBST(T->rchild,key); /* 在右子树中继续搜索 * /
}
```

这个过程也可以用非递归算法实现,算法描述如下:

```
BiTree Search_BST(BiTree T,KeyType k)
{
   BiTree p = T; /* 指针 p 指向根结点,搜索从根结点开始 * /
   while(p != NULL && p->data != k)
   {
      if ( k < p->data) p = p->lchild; /* 在左子树中继续搜索 * /
      else p = p->rchild; /* 在右子树中继续搜索 * /
   }
   return (p);
   /* 若查找到,则返回的 p 为指向该结点的指针,若没查到,返回的 p 为 NULL * /
}
```

(3) 二叉排序树结点的插入

在一棵二叉排序树中插入一个结点可以用一个递归的过程实现,即:若二叉排序树为空,则新结点作为二叉排序树的根结点;否则,若给定结点的关键字值小于根结点的关键字值,则插入在左子树上;若给定结点的关键字值大于根结点的值,则插入在右子树上。

下面是二叉排序树的插入算法:

```
BiTree InsertBST(BiTree T , BitNode * s)
{ /* 在 T 所指的二叉排序树中插入结点 s,使其保持为二叉排序树,返回插入后的指向树根结点的指针 * /
   s->lchild = s->rchild = NULL;
   if(T == NULL) return s;
   BiTree p = T;
   while(p)
   {
      if(s->data == p->data)
         return T; /* 已有该结点,插入失败 * /
      else if(s->data < p->data)
```

```

    { pr = p; p = p->lchild; }
    else { pr = p; p = p->rchild; }
}
/* 无相同的结点,将结点 s 作为结点 pr 的孩子 */
if(s->data < pr->data)
    pr->lchild = s;                /* 将 s 作为 pr 的左孩子 */
else
    pr->rchild = s;                /* 将 s 作为 pr 的右孩子 */
return T;
}

```

利用二叉排序树的插入算法,可以很容易地实现创建二叉排序树的操作,其基本思想为:由一棵空二叉树开始,经过一系列的查找插入操作生成一棵二叉排序树。

2. 哈希表

(1) 哈希表的概念

哈希表查找方法的基本思想:将记录的存储位置与它的关键字之间建立一个确定的关系 H ,使每个关键字对应一个惟一的存储位置,在查找时,只需要根据对应关系计算出给定关键字值 k 所对应的值 $H(k)$,就可以得到记录的存储位置。

哈希函数:将记录的关键字值与记录的存储位置对应起来的关系 H 称为哈希函数, $H(k)$ 的值称为哈希地址。

哈希表:根据哈希函数建立的表。以记录的关键字值为自变量,根据哈希函数,计算出对应的哈希地址,并在此存储该记录的内容。当对记录进行查找时,首先根据给定的关键字值,用同一个哈希函数计算出给定关键字值对应的存储地址,随后进行访问。哈希表既是一种存储方法,又是一种查找方法,通常将这种查找方法称为哈希查找。

冲突:不同的关键字值对应相同的存储地址的情况。具有相同函数值的关键字值称为同义词。在实际应用中冲突是不可能完全避免的,但可以通过一些办法减少冲突。

(2) 哈希函数的构造

建立哈希表,关键是构造哈希函数。其原则是尽可能地使任意一组关键字的哈希地址均匀地分布在整個地址空间中,即用任意关键字作为哈希函数的自变量得到的计算结果随机分布,以便减少冲突发生的可能性。

常用的哈希函数的构造方法有:

(a) 直接定址法

取关键字或关键字的某个线性函数为哈希地址。即

$$H(key) = key \text{ 或 } H(key) = a * key + b。$$

其中 a, b 为常数,调整 a 与 b 的值可以使哈希地址的取值范围与存储空间范围一致。

(b) 质数取余法

取关键字被某个不大于哈希表表长 n 的质数 m 整除后所得的余数为哈希地址。即

$$H(key) = key \% m \quad (m < n, \text{ 设其中 } n \text{ 为哈希表长})。$$

质数取余法计算简单,适用范围大,但是整数 m 的选择很重要,如果选择不当,会产生较多同义词,使哈希表中有较多的冲突。

(c) 平方取中法

取关键字平方后的中间几位为哈希地址。由于平方后的中间几位数与原关键字的每一位数字都相关,只要原关键字的分布是随机的,以平方后的中间几位数作为哈希地址一定也是随机分布的。

(d) 折叠法

把关键字折叠成位数相同的几部分,然后取这几部分的叠加作为哈希地址。在关键字位数较多,且每一位上数字的分布基本均匀时,采用折叠法得到的哈希地址比较均匀。

(3) 处理冲突的方法

常用的处理冲突的方法有:

(a) 开放定址法

当发生冲突时,产生一个探测序列,沿着此序列去寻找可以存放记录的空闲单元。常用的探测方法有:线性探测再散列;二次探测再散列;伪随机探测再散列。

(b) 链地址法

将所有关键字是同义词的记录链接成一个线性链表,将其表头链接在由哈希函数确定的哈希地址所指示的存储单元中。

(c) 再哈希法

当发生冲突时,用另一个哈希函数计算另一个哈希地址,如果再发生冲突,再使用另一个哈希函数,直至不发生冲突为止。这种方法要求预先设置一个哈希函数的序列。

(d) 溢出区法

除基本的存储区外(称为基本表),另外建立一个公共溢出区(称为溢出表),当发生冲突时,将记录存到这个公共溢出区中。

(4) 哈希表查找及其分析

哈希表的查找过程与哈希表的构造过程基本一致,查找过程中需要比较的次数取决于以下三个因素:哈希函数、处理冲突的方法和哈希表的装填因子。其中,哈希表的装填因子定义为:

$$\alpha = \frac{\text{表中填入的记录数}}{\text{哈希表的长度}}$$

装填因子越小,表中填入的记录就越少,发生冲突的可能性就会小。反之,表中已填入的记录越多,再填充记录时,发生冲突的可能性就越大,继而查找时进行关键字的比较次数就越多。

7.2 实训篇

实训目的:

- 顺序查找及折半查找的算法实现
- 二叉排序树的建立、查找及中序遍历输出
- 散列表的构造和查找算法,包括哈希函数的建立、处理冲突的方法

7.2.1 手机短信内容中电话号码和手机号码的识别与提取

案例描述

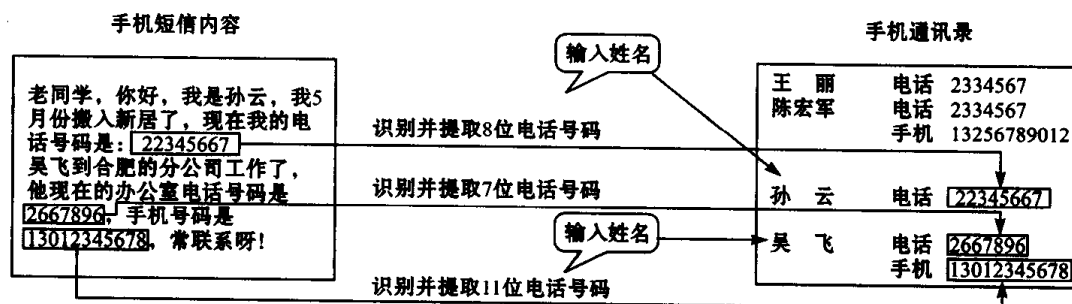
在使用手机收发短信时,收到的短信内容中常会包含对方发来的电话号码或手机号码,为了方便用户能直接提取其中的号码并存入到其手机的通讯录中,现要求开发手机系统软件中的一个子功能,实现从手机短信内容中识别和提取电话号码(7位或8位)和手机号码(11位),并将其存入通讯录。

设计分析

要从手机短信的内容中识别出电话号码或手机号码,必须从短信的第一个字符开始查找,找到第一个数值型字符('0'~'9'),然后依次判断其后的字符,若其后有连续的6个或7个数值型字符,则将其识别成电话号码并提取,若其后有连续的10个数值型字符,则将其识别成手机号码并提取。继续向后搜索直到整个短信内容查找完毕。

因为短信的内容是没有规律的,所以查找时只能用顺序查找方法。

短信的内容以文本文件的形式提供,通讯录中的信息采用文本文件的形式保存。



设计方案

(1) 存储结构类型定义

短信的存储结构类型定义:

```
typedef struct {
    char word[200];          /* 短信内容 */
    int length;              /* 短信长度 */
} Message;
```

通讯录中记录的存储结构类型定义:

```
typedef struct {
    char name[8];            /* 姓名 */
    char phone[11];          /* 电话号码或手机号码 */
} Note;
```

(2) 主要算法

/* 从存放短信的文件中将短信内容取出,放入顺序表中 */

```
void Read(Message *l)
{
    FILE *fp;
    int i=0;
```

```

fp = fopen("d:\\tc\\swj\\message.txt", "r");
if(!fp)
    printf("\n 短信无法打开! (Can not open the message!);");
else
{
    l -> length = 0;                /* 置顺序表长度初值 */
    while(!feof(fp))                /* 将短信内容全部读入顺序表中 */
    {
        fscanf(fp, "%c", &l -> word[i]);
        i++;
        l -> length++;
    }
}
fclose(fp);

/* 从存放短信的顺序表中自动识别 7 位、8 位的电话号码, 或 11 位的手机号码 */
int Search_Seq(Message *st, Note *log)
    /* 通过 w 记录本次查询结束位置的下个单元, 作为下次查询的开始位置 */
{
    int k, j, t, i = 0;
    for(i = w; i < st -> length && !Judge(st -> word[i]); i++);
    /* 找到第一个数值型字符 */
    for(k = i + 1; k <= i + 6; k++)
        if(!Judge(st -> word[k])) { w = k + 1; return 0; }
    /* 判断后 6 个字符是否为数值型字符 */
    if(!Judge(st -> word[k]))
        /* 前 7 个字符为数值型, 判断下个字符是否为非数值型, 若是则识别出一个 7 位的电话号码 */
        {
            for(j = 0, t = i; t <= i + 6; t++) /* 将 7 位电话号码读入 */
            {
                log -> phone[j] = st -> word[t];
                j++;
            }
            log -> phone[j] = '\0';
            w = k + 1;                /* w 指向本次查询结束点的下一个位置 */
            return 7;                /* 返回值为 7, 说明本次识别的是一个 7 位电话号码 */
        }

    else if(!Judge(st -> word[k + 1]))
        /* 前 8 个字符为数值型, 判断下个字符是否为非数值型, 若是则识别出一个 8 位的电话号码 */
        {
            for(j = 0, t = i; t <= i + 7; t++) /* 将 8 位电话号码读入 */

```

```

    {
        log->phone[j] = st->word[t];
        j++;
    }
    log->phone[j] = '\0';
    w = k + 1;
    return 8;                /* 返回值为 8,说明本次识别的是一个 8 位的电话号码 */
}
else if(Judge(st->word[k+2])&&Judge(st->word[k+3])&&
        !Judge(st->word[k+4]))
/* 前 9 个字符为数值型,判断第 10、11 个字符是否为数值型且第 12 个字符是否为非数值型,若是则识别出一个 11 位的手机号码 */
{
    for(j=0,t=i;t<=i+10;t++)
    {
        log->phone[j] = st->word[t];
        j++;
    }
    log->phone[j] = '\0';
    w = k + 4;
    return 11;              /* 返回值为 11,说明本次识别的是一个 11 位的手机号码 */
}
}

```

(3) 实现程序

```

#include "string.h"
#include "conio.h"
#include "stdio.h"
typedef struct {
    char word[100];          /* 短信内容 */
    int length;              /* 短信长度 */
} Message;
typedef struct {
    char name[8];            /* 姓名 */
    char phone[11];          /* 电话号码或手机号码 */
} Note;
int w;                      /* 全局变量,在读文件时用于记录当前读到的位置 */

int Judge(char c)           /* 判断一个字符是否为数值型字符 */
{
    if(c >= '0' && c <= '9') return 1;
    else return 0;
}

```

```

main()
{
    int i,k,s;
    char name[8];
    FILE * fp;
    Message st;
    Note log;
    Read(&st);
    printf("%s",st.word);
    fp = fopen("d:\\tc\\c\\note.txt","wb");
    printf("\n");
    w=0;
    while(w<st.length)
    {
        k = Search_Seq(&st,&log);
        /* 循环识别短信中所包含的所有号码,直到剩余未被判断的字符个数小于7个 */
        if(k==0) continue; /* 若本次没有识别出号码,直接进入下一次 */
        for(i=0;i<k;i++)
            printf("%c",log.phone[i]);
        printf("\n");
        printf("请输入姓名:");
        scanf("%s",log.name);
        fprintf(fp,"%s %s ",log.name,log.phone); /* 将姓名和号码写入文件中 */
    }
    fclose(fp);
}

```

(4) 运行测试

文本文件 message.txt 的内容为:

老同学,你好,我是孙云,我5月份搬入新居了,现在我的电话号码是:
22345667,吴飞到合肥的分公司工作了,他现在的办公室电话号码是 2667896,
手机号码是 13012345678,常联系呀!

编译运行后显示结果为:

老同学,你好,我是孙云,我5月份搬入新居了,现在我的电话号码是:22345667,吴飞到合肥的分公司工作了,他现在的办公室电话号码是 2667896,手机号码是 13012345678,常联系呀!

识别出:

22345667

请输入姓名:孙云(回车)

2667896

请输入姓名:吴飞(回车)

13012345678

请输入姓名:吴飞(回车)

程序运行结束后文件 note.txt 的内容为:

孙云	22345667	吴飞	2667896	吴飞	13012345678
----	----------	----	---------	----	-------------

7.2.2 数据结构教材中各功能函数的存放与查找

案例描述

在数据结构教材的每一章中有许多各种结构的基本操作函数,为了便于查找各函数,了解其功能,现要求设计一个系统来放置各函数及其功能说明,并能按照函数名快速查找到相应的函数及其功能说明,从而方便用户了解各函数的具体功能,对学习数据结构这门课程起到辅助作用。

设计分析

由于各函数名之间的差异较大,因此在采用散列法实现对它们的存储及查找时,应该设计合理的散列函数,以减少在存储和查找过程中冲突的发生,当发生冲突时,可以采用线性探测法等措施来解决冲突。存储后对各函数的查找采用按名查找。

哈希函数的选择: $f(key) = (key \text{ 的第 } 1、2、3 \text{ 个字母的 ASCII 码之和} - 96 * 3) \% 35$

其中若这三个字母有大写的,则首先将其转换成小写。

各函数名和功能说明如表 7-1 所示:

表 7-1 函数名和功能说明对应表

函数名	功能说明
AddPolyn	多项式加法
BinsertSort	折半插入排序
Bubble_Sort	冒泡排序
BfsTraverse	广度优先遍历图
CreatePolyn	建立一元多项式链表
CreateGraph	构造图的数组存储结构
CopyGList	复制广义表
Dfs	深度优先遍历图
DeQueue	删除非空队列
DestroyQueue	销毁链队列
EnQueue	向队列插入新元素
Gettop	返回栈顶元素
HeapSort	堆排序
InitQueue	构造一个空队列
Index	返回子串在主串中的位置
InsertBST	向二叉排序树中插入新结点

续表

函数名	功能说明
ListDelete	删除链表中的一个结点
ListInsert	向链表插入一个新结点
LocateElem	在链表中查找一个元素
MergeList	归并两个有序线性表
Msort	归并排序
Pop	删除栈顶元素
Push	向栈顶插入新元素
QuickSort	快速排序
QueueLength	返回队列长度
RadixSort	基数排序
SubString	返回定长子串
Search_Bin	折半查找
SelectSort	简单选择排序
Trial	求 n 皇后问题的合法布局

通过哈希函数计算和线性探测法解决冲突后,得到的哈希表如下图所示:

0	1	2	3	4	5	6	7	8
InitQueue	EnQueue	Index	LocateElem	MergeList	ListDelete	ListInsert	InsertBST	QueueLength
info	info	info	info	info	info	info	info	info
9	10	11	12	13	14	15	16	17
AddPolyn	SubString	Search_Bin	Msort	Pop	HeapSort	QuickSort	SelectSort	Trial
info	info	info	info	info	info	info	info	info
18	19	20	21	22	23	24	25	
#####	#####	#####	Push	#####	RadixSort	#####	BinsertSort	
			info		info		info	
26	27	28	29	30	31	32	33	34
Bubble_Sort	BfsTraverse	CreatePolyn	CreateGraph	Dfs	DeQueue	DestroyQueue	Cettop	CopyGList
info	info	info	info	info	info	info	info	info

设计方案

(1) 存储结构类型定义

函数信息的存储结构类型定义:

```
typedef struct
{
    char name[13];           /* 函数名 */
    char info[30];          /* 函数功能说明 */
} Func;
```

函数库的存储结构类型定义:

```
typedef struct
{
    Func data[35];
    int length;             /* 函数库的长度 */
} HashTable;
```

(2) 主要算法

```
int HashFun(char key[])    /* 用于计算哈希地址的哈希函数 */
{
    int p,i;
    char k[3];
    for(i=0;i<3;i++)
        k[i]=tolower(key[i]); /* 若参与哈希函数运算的字符是大写,将其转换成小写 */
    p=(k[0]+k[1]+k[2]-96*3)%35; /* 通过哈希函数计算地址 */
    return p;                /* 返回哈希地址 */
}
```

```
void InsertHash(HashTable *st,Func x) /* 将 x 放入散列表中 */
{
    int i,p;
    p=i=HashFun(x.name); /* 计算 x 的哈希地址 */
    while(strcmp(st->data[i].name,XX)&&(i+1)%MAXSIZE!=p)
        /* 有冲突,向下探测 */
        i=(i+1)%MAXSIZE; /* 用线性探测法重新计算地址 */
    if((i+1)%MAXSIZE==p) return; /* 若没有探测到空单元,插入失败,返回 */
    else
        st->data[i]=x; /* 将数据存入找到的空单元,插入成功 */
}
```

```
void CreateHash(HashTable *l) /* 建立哈希表 */
{
    FILE *fp;
    Func x;
    int i;
    char key[13],sh[30];
    for(i=0;i<MAXSIZE;i++) /* 给所有单元赋初值 */
```

```

    strcpy(l->data[i].name,XX,12);
    l->data[i].name[12]='\0';
}
fp=fopen("d:\\tc\\swj\\func.txt","r");
if(!fp) printf("无法打开数据文件! \n");
else
{
    while(!feof(fp))                /* 从文件中每读出一个数据,就将其插入到哈希表中 */
    {
        fscanf(fp,"%s%s\n",key,sh);
        strcpy(x.name,key,13);
        strcpy(x.info,sh,30);
        InsertHash(l,x);
    }
    fclose(fp);
}

int SearchHash(HashTable st,char key[]) /* 在哈希表中进行查询 */
{
    int i,p;
    i=p=HashFun(key);
    if(p<0) return -1;                /* 防止输入错误的查询关键字,如只输入两个字母 */
    while(strcmp(st.data[i].name,key)&&((i+1)%MAXSIZE)!=p)
        i=(i+1)%MAXSIZE;             /* 若没查找到,则重新计算地址 */
    if(strcmp(st.data[i].name,key)==0)
        return i;                    /* 若查找到,则返回其下标,查找成功 */
    else if((i+1)%MAXSIZE==p)
        return -1;                   /* 搜索完整个表后若没查找到,则返回-1表示查找失败 */
}

```

(3) 程序实现

```

#include "stdio.h"
#include "string.h"
#include "conio.h"
#include "ctype.h"
#define XX "#####"
#define MAXSIZE 35                /* 哈希表的长度 */
main()
{
    HashTable st;
    char key[13];

```

```

int k,i;
CreateHash(&st);
while(getch()!='0')
{
    puts("\n 请输入要查询的函数名:(按0退出)");
    scanf("%s",key);
    k = SearchHash(st,key);
    if(k)
        printf("%s\n",st.data[k].info);
    else
        printf("无此函数");
}
}

```

(4) 运行测试

文本文件 func.txt 的内容为表 7-1 的内容。

编译运行后显示结果为:

提示:请输入要查询的函数名:(按0退出)

输入:AddPolyn

输出:多项式加法

提示:请输入要查询的函数名:(按0退出)

输入:CreatePolyn

输出:建立一元多项式链表

提示:请输入要查询的函数名:(按0退出)

输入:Concat

输出:无此函数

提示:请输入要查询的函数名:(按0退出)

输入:Msort

输出:归并排序

提示:请输入要查询的函数名:(按0退出)

输入:0

退出程序

7.2.3 二叉排序树的使用

案例描述

编写一程序来统计一个文本文件中各字符出现的次数,再对它们字符的 ASCII 码进行排序,建立一棵二叉排序树,然后按照字符大小依次输出字符的信息:字符本身和出现次数,并可以按照用户的要求查询某字符的出现次数。

设计分析

在本设计中,首先从文件中一边读字符,一边按其大小将其插入到二叉排序树中,当所有的字符插入完毕,二叉排序树即建立完成。然后对二叉排序树进行中序遍历,便可按

字符大小输出字符及其出现次数,对于用户要求查询特定字符的出现次数,可在二叉排序树中进行查找,找到后输出。

设计方案

(1) 存储结构类型定义

字符信息的存储类型定义:

```
typedef struct node
{
    char val;                /* 字符 */
    int count;               /* 出现次数 */
    struct node *lchild, *rchild; /* 左、右孩子指针 */
} BitNode, * BinTree;
```

(2) 主要算法

```
void CreatBinTree(BinTree *t, char ch) /* 按字符的大小建立一棵二叉排序树 */
{
    BitNode *ptr, *p;
    p = NULL;
    ptr = *t;
    while(ptr != NULL)
        if(ch == ptr->val)
        {
            ptr->count++;
            return;
        }
        else
        {
            p = ptr;
            ptr = ch > ptr->val ? ptr->rchild : ptr->lchild;
        }

    ptr = (BitNode *)malloc(sizeof(BitNode));
    ptr->lchild = ptr->rchild = NULL;
    ptr->val = ch;
    ptr->count = 1;
    if(p == NULL) *t = ptr;
    else if(ch > p->val) p->rchild = ptr;
    else p->lchild = ptr;
}

void Travel_tree(BinTree t) /* 中序遍历二叉排序树各结点的递归算法 */
{
    if(t == NULL) return;
    travel_tree(t->lchild);
```

```

    printf("%4c,%9d\n",t->val,t->count);
    travel_tree(t->rchild);
}

BitNode * Search_BST(BinTree t,char k) /* 在二叉排序树中查找字符 */
{
    BitNode * p;
    p = BitNode * t; /* 指针 p 指向根结点,搜索从根结点开始 */
    while(p != NULL && p->val != k)
    {
        if (k < p->val) p = p->lchild; /* 在左子树中继续搜索 */
        else p = p->rchild; /* 在右子树中继续搜索 */
    }
    return (p); /* 若查找到,则返回指向该结点的指针 p,若没查找到,返回为 NULL */
}

```

(3) 程序实现

```

#include "stdio.h"
#include "malloc.h"
typedef struct node
{
    char val; /* 字符 */
    int count; /* 出现次数 */
    struct node * lchild, * rchild; /* 左、右孩子指针 */
} BitNode, * BinTree;

main()
{
    FILE * fp;
    BitNode * p;
    BinTree root = NULL;
    char c;
    fp = fopen("d:\\tc\\swj\\text.txt","r");
    if(!fp)
        printf("\n 数据文件无法打开!");
    while(!feof(fp))
    {
        fscanf(fp,"%c\n",&c);
        c = tolower(c); /* 将大写字母转换成小写 */
        CreatBinTree(&root,c);
    }
    printf("字符 出现次数\n");
    Travel_tree(root);
}

```

```

do
{
    printf("请输入要查询的字符:(输入'#'退出)\n");
    scanf("\n% c",&c);
    c=tolower(c);                /* 若输入的待查询的字母为大写,将其转换成小写 */
    p=Search_BST(root,c);
    printf("出现次数为:% d\n",p->count);
}while(c!='#');
fclose(fp);
}

```

(4) 运行测试

文本文件 text.txt 的内容为:

An instruction is made up of operations that specify the function to be performed and operands that represent the data to be operated on.

编译运行后显示结果为:

字符	出现次数
.	1
a	10
b	2
c	3
d	6
e	15
f	4
h	4
i	6
m	2
n	10
o	11
p	7
r	8
s	6
t	15
u	3
y	1

提示:请输入要查询的字符:(输入'#'退出)

输入:m

显示结果:出现次数为:2

提示:请输入要查询的字符:(输入'#'退出)

输入:e

显示结果:出现次数为:15

提示:请输入要查询的字符:(输入'#'退出)

输入:E

显示结果:出现次数为:15

提示:请输入要查询的字符:(输入'#'退出)

输入:b

显示结果:出现次数为:2

提示:请输入要查询的字符:(输入'#'退出)

输入:A

显示结果:出现次数为:10

提示:请输入要查询的字符:(输入'#'退出)

输入:#

退出程序

7.3 实训项目

项目内容

设计一程序实现公司人事档案的查询。一个公司有多个部门,将公司所有职员的档案按部门依次存放,部门间是有序的,而每个部门内部职员的档案是无序存放的。其中职员档案的主关键字是编号,每个编号占五位,前两位为部门编号,后三位为职员在部门中的员工编号。现在要求设计一程序实现对员工的档案按照编号进行快速查询。

设计要求

按照输入的编号快速查询出相应职员的档案,并输出。

项目提示

部门间的档案是有序的,部门内的档案存放是无序的,因此可以按部门建立档案索引表,存储各个部门员工档案的起止位置。查找时,根据输入编号的前两位,在档案索引表中找到其部门档案的起止位置,然后按编号的后三位在此范围内顺序查找。

7.4 基本题

7.4.1 选择题

1. 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_____。

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

2. 对线性表进行二分查找时,要求线性表必须_____。

A. 以链式方式存储
B. 以顺序方式存储

- C. 以链式方式存储且结点按关键字排序
 D. 以顺序方式存储且结点必须按关键字排序
3. 已知一个有序顺序表为(11,15,23,35,45,56,66,85,89,106,127),当二分查找值为89的元素时,需要_____次比较即可查找成功。
 A. 1 B. 2 C. 3 D. 4
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_____。
 A. $O(n^2)$ B. $O(n\log_2 n)$ C. $O(n)$ D. $O(\log_2 n)$
5. 哈希查找中 k 个关键字具有同一哈希值,若用线性探测法将这 k 个关键字对应的记录存入哈希表中,至少要进行_____次探测。
 A. k B. $k+1$ C. $k(k-1)/2$ D. $1+k(k+1)/2$
6. 设哈希表长 $m=14$,哈希函数 $H(key)=key\%11$ 。表中已经有4个结点:
 $addr(15)=4$ $addr(38)=5$ $addr(61)=6$ $addr(84)=7$
 其余地址为空,若采用二次探测再散列解决冲突,关键字为49的结点的地址为_____。
 A. 8 B. 3 C. 5 D. 9

7.4.2 填空题

1. 采用二分查找算法,当 n 较大时,其平均查找长度 ASL 近似为_____。
2. 长度为255的表,采用分块查找法,每块的最佳长度是_____。
3. 若有序顺序表中有1000个元素,用折半法查找时,最大的比较次数是_____。
4. 二叉排序数采用_____遍历可以得到结点的有序序列。
5. 假设在有序顺序表 $A[1..20]$ 上进行二分查找,比较一次查找成功的结点数_____ ,比较两次查找成功的结点数_____ ,比较三次查找成功的结点数_____ ,比较四次查找成功的结点数_____ ,比较五次查找成功的结点数_____ ,平均查找长度为_____。
6. 已知一组关键字为(18,25,63,50,42,32,90,66),按哈希函数 $H(key)=key\%9$ 和线性探测冲突构造哈希表,在每个关键字的查找概率相同的情况下,查找成功时的平均查找长度为_____。
7. 下列算法实现在二叉排序树上进行查找,请在空格处填上适当的语句,完成上述功能。

```

BitNode *bstsearch(BitNode *t,KeyType k)
{
    if(t==NULL)
        return NULL;
    else
        while(t!=NULL)
        {
            if(t->key==k) ① ;
        }
    }

```

```

        if(t->key > k) ②;
        else ③;
    }
}

```

7.5 习题解析

1. 试述顺序查找、二分法查找和分块查找对被查找的表中的元素有什么要求,并求对长度为 n 的表,分别按三种方法进行查找时的平均查找长度。

解 对于顺序查找,表中元素可以按任何方式存放;而采用二分法查找时要求表中元素必须是有序的,而且须以顺序方式进行存储;若要利用分块查找,则要求表中元素须“块”间有序,但每一块内的元素可以按任意方式存放。

平均查找长度:顺序查找时查找成功的平均查找长度为 $(n+1)/2$,而采用二分法查找时,其平均查找长度为 $\log_2(n+1)-1$ (当 n 较大时)。分块查找的平均查找长度的大小与其确定所在块所采用的查找方法有关。若用顺序查找确定所在块,则平均查找长度为 $(n/s + s)/2 + 1$ (或 $(s^2 + 2s + n)/2s$);若用二分法查找确定所在的块,平均查找长度为 $\log_2(n/s + 1) + s/2$,其中 s 为每块含有的元素个数。

2. 用 C 语言写出顺序查找的程序,要求当查找不成功时在表尾插入新元素。

解 算法由主函数和顺序查找表、输出表三个函数组成。

程序如下:

```

#define MAXITEM 20                                /* 定义一常量 MAXITEM 为 20,为顺序表的大小 */
typedef struct element                             /* 顺序表的定义 */
{
    int key;                                       /* 结点关键字域 */
}sqlist[MAXITEM];

int seqsrch(sqlist r,int k,int *n)                /* 在顺序表 r 中查找关键字为 k 的记录 */
{
    int i=0;                                       /* 置下标 i 为 0 */
    while((i < *n)&&(r[i].key!=k))
        i++;
    if(i==( *n))
    {
        r[i].key=k;                               /* 未找到,在顺序表末尾插入 k */
        ( *n)++;                                  /* 顺序表长度加 1 */
    }
    return(i);                                    /* 返回下标值 */
}

void print(sqlist r,int *n)                       /* 输出顺序表 */

```

```

    {
        int i;
        for(i=0;i<(*n);i++)
        {
            printf("%d ",r[i].key);
        }
        printf("\n");
    }

main()
{
    sqliist a={7,3,5,8,10,4,12};          /* 顺序表有 7 个元素 */
    int i,k,m,*n;
    *n=m=7;                               /* 顺序表长度为 7 */
    k=20;                                  /* 可以在第二次运行时置 k=10 */
    print(a,n);                            /* 第一次输出顺序表 */
    i=seqsrch(a,k,n);                      /* 找 k 元素 */
    if(i<m)
        printf("found, k=%d and i=%d\n",k,i);
    else
        printf("not found, k=%d and i=%d\n",k,i);
    print(a,n);                            /* 第二次输出顺序表 */
}

```

第一次输出结果为:

7 3 5 8 10 4 12

not found, k=20 and i=7

7 3 5 8 10 4 12 20

第二次输出结果为(修改 k=10):

7 3 5 8 10 4 12

found, k=10 and i=4

7 3 5 8 10 4 12

3. 设链表中的数据元素结构为关键字域 key 和指向下一个元素的指针。试对此链表写出顺序查找的算法。若查找不成功,请将待查关键字 k 插入表尾。

解 算法由主函数和创建链表、顺序查找链表、输出链表四个函数组成。

程序如下:

```

#include<stdio.h>
typedef struct node
{
    int key;
    struct node *next;
}Node;

```



```

void print(Node *head)                                /* 输出线性链表各元素 */
{
    Node *p;
    p = head;
    while(p != NULL)
    {
        printf("%d ", p->key);
        p = p->next;
    }
    printf("\n");
}

main()
{
    Node *a;
    int k = 6, p;                                     /* p 为 k 的结点序号 */
    a = create();
    print(a);                                           /* 第一次输出线性链表 */
    p = seqsrch(a, k);
    printf("position = %d\n", p);                       /* 输出 k 的位置 p */
    print(a);                                           /* 第二次输出线性链表 */
}

```

第一次输出结果为:

5(按回车)

1 2 3 4 5(按回车)

-1 1 2 3 4 5

position = 7

-1 1 2 3 4 5 6

第二次输出结果为(k 仍然为 6):

5(按回车)

2 3 4 5 6(按回车)

-1 2 3 4 5 6

position = 6

-1 2 3 4 5 6

4. 若有一个由 17 个元素组成的有序表, 现利用二分法查找有序表的元素, 问查找成功时, 最少比较几次? 最多比较几次?

解 依题意, 可以构造一棵有序完全二叉树, 共 17 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 2 个结点。查找从根结点开始, 沿其左或右向下一层继续查找, 或查找成功或查找失败。因此, 查找成功时, 最少比较 1 次, 最多比较 5 次。

5. 已知如下 11 个数据元素的有序表(6,14,19,21,36,57,63,76,81,89,93),请画出查找键值为 21(成功)和 85(失败)的查找过程。

解 对有序表(设有序表为 A,其中某关键字为 $A[i].key$)的查找可采用二分法查找。

设变量 low、high 和 mid 分别表示待查有序表的下界、上界和中间位置,即:

$mid = (low + high)/2$ 。初值 $low = 1, high = 11$,具体如下:

序号:	1	2	3	4	5	6	7	8	9	10	11
关键字:	6	14	19	21	36	57	63	76	81	89	93
变量:	$\uparrow low$										$\uparrow high$

(1) 以下给出键值 $key = 21$ 的查找过程:

由初始 $low = 1, high = 11$,求得 $mid = 6$,如下所示:

序号:	1	2	3	4	5	6	7	8	9	10	11
关键字:	6	14	19	21	36	57	63	76	81	89	93
变量:	$\uparrow low$					$\uparrow mid$					$\uparrow high$

因为 $A[mid].key = A[6].key > key$,即 $57 > 21$,所以,继续在前半区进行查找。 low 值不变, $high = mid - 1 = 5, mid = (1 + 5)/2 = 3$ 。如下所示:

序号:	1	2	3	4	5	6	7	8	9	10	11
关键字:	6	14	19	21	36	57	63	76	81	89	93
变量:	$\uparrow low$		$\uparrow mid$		$\uparrow high$						

又因为 $A[mid].key = A[3].key < key$,即 $19 < 21$,所以,继续在后半区进行查找。 $high$ 值不变, $low = mid + 1 = 3 + 1 = 4, mid = (4 + 5)/2 = 4$ 。如下所示:

序号:	1	2	3	4	5	6	7	8	9	10	11
关键字:	6	14	19	21	36	57	63	76	81	89	93
变量:				$\uparrow low$	$\uparrow high$						
				$\uparrow mid$							

由于 $A[mid].key = A[4].key = 21$,此值与给定关键字 21 相等,查找成功。

(2) 以下给出键值 $key = 85$ 的查找过程:

由初始 $low = 1, high = 11$,求得 $mid = 6$,如下所示:

序号:	1	2	3	4	5	6	7	8	9	10	11
关键字:	6	14	19	21	36	57	63	76	81	89	93
变量:	$\uparrow low$					$\uparrow mid$					$\uparrow high$

因为 $A[mid].key = A[6].key < key$,即 $57 < 85$,所以,继续在后半区进行查找。 $high$ 值不变, $low = mid + 1 = 7, mid = (7 + 11)/2 = 9$ 。如下所示:

序号:	1	2	3	4	5	6	7	8	9	10	11
关键字:	6	14	19	21	36	57	63	76	81	89	93
变量:							$\uparrow low$		$\uparrow mid$		$\uparrow high$

又因为 $A[mid].key = A[9].key < key$, 即 $81 < 85$, 所以, 继续在后半区进行查找。
high 值不变, $low = mid + 1 = 10$, $mid = (10 + 11)/2 = 10$ 。如下所示:

序号:	1	2	3	4	5	6	7	8	9	10	11
关键字:	6	14	19	21	36	57	63	76	81	89	93
变量:										↑ low	↑ high
										↑ mid	

因为 $A[mid].key = A[10].key > key$, 即 $89 > 85$, 所以, 继续在前半区进行查找。low 值不变, $high = mid - 1 = 10 - 1 = 9$, $mid = (10 + 9)/2 = 9$ 。如下所示:

序号:	1	2	3	4	5	6	7	8	9	10	11
关键字:	6	14	19	21	36	57	63	76	81	89	93
变量:									↑ high	↑ low	
									↑ mid		

此刻, 由于下界 $low >$ 上界 $high$, 则说明表中没有关键字值等于 85 的元素, 查找失败。

6. 给定表 (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Oct, Nov, Dec)。设取散列函数 $H(x) = i/2$, 其中 i 为键值中第一个字母在英语字母表中的序号, 要求:

- (1) 画出相应的链地址法解决冲突的散列表。
- (2) 画出相应的线性探测法解决冲突的散列表。

解 (1) 散列表用链表实现, 用链地址法解决散列冲突:

因为, 英文字母共 26 个, 散列函数 $H(x) = i/2$, 其中 i 为键值中第一个字母在英语字母表中的序号, 所以 $H(x)$ 最大值为 13。由此可设计用于指向单链表的散列表表头数组 $HT[0 \dots 13]$ 。

根据散列函数 $H(x) = i/2$, 计算给定表中 11 个键值的散列函数值如下:

$$H(\text{Jan}) = 10/2 = 5$$

$$H(\text{Feb}) = 6/2 = 3$$

$$H(\text{Mar}) = 13/2 = 6$$

$$H(\text{Apr}) = 1/2 = 0$$

$$H(\text{May}) = 13/2 = 6$$

$$H(\text{Jun}) = 10/2 = 5$$

$$H(\text{Jul}) = 10/2 = 5$$

$$H(\text{Aug}) = 1/2 = 0$$

$$H(\text{Oct}) = 15/2 = 7$$

$$H(\text{Nov}) = 14/2 = 7$$

$$H(\text{Dec}) = 4/2 = 2$$

用前插法将键值 x 插到计算后的第 $H(x)$ (即 $i/2$) 个单链表上。结果如图 7.1 所示。

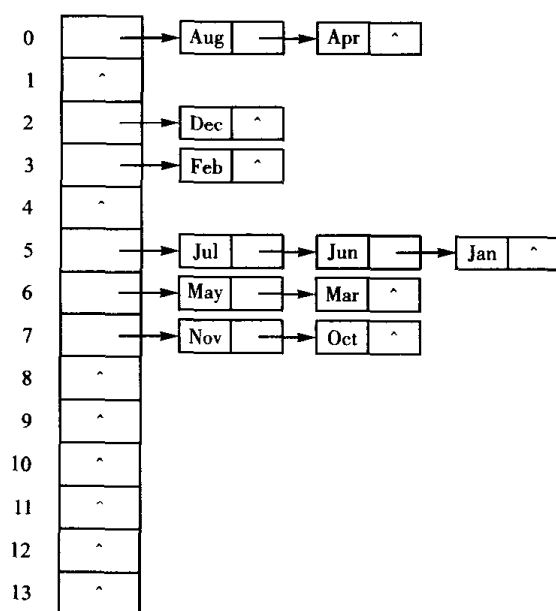


图 7.1 用链地址法解决散列冲突

(2) 散列表用顺序表实现,用线性探测法解决散列冲突:

为了减少散列冲突,令装填因子 $\alpha = 0.75$, 因为 $n = 11$, 散列表长 $m = n/\alpha = 11/0.75 = 14.67$, 取整数为 15, 即散列表为 $HT[0 \dots 14]$ 。

根据散列函数 $H(x) = i/2$, 计算给定表中 11 个键值的散列函数值, 即散列地址 (计算结果如上一小题); 若地址是开放的 (可用), 则插入新键值; 否则, 求下一个开放地址, 再插入新键值。结果如图 7.2 所示。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Apr	Aug	Dec	Feb		Jan	Mar	May	Jun	Jul	Oct	Nov			
1	2	1	1		1	1	2	4	5	4	5			

图 7.2 用线性探测法解决散列冲突

7. 采用分块查找时, 若线性表中共有 625 个元素, 且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, 求每块应分多少个结点最佳。

解 每块应分结点最佳数目可依据公式 $d = \sqrt{n} = \sqrt{625} = 25$ 求得, 每块应分 25 个结点为最佳。

8. 设散列表的长度为 13, 散列函数为 $H(K) = K \% 13$, 给定的关键字序列为 19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79。试画出分别用链地址法和线性探测法解决冲突时所构造的散列表, 并求等概率下这两种方法的成功和不成功的平均查找长度。

解 (1) 散列表用链表实现, 用链地址法解决散列冲突:

根据题目给定的散列函数 $H(K) = K \% 13$, 其值域为 $0 \sim 12$, 可设计用于指向单链表的散列表表头数组 $HT[0 \dots 12]$ 。

根据散列函数 $H(K) = K \% 13$, 计算给定关键字序列的散列函数值如下:

$$H(19) = 19 \% 13 = 6$$

$$H(14) = 14 \% 13 = 1$$

$$H(23) = 23 \% 13 = 10$$

$$H(1) = 1 \% 13 = 1$$

$$H(68) = 68 \% 13 = 3$$

$$H(20) = 20 \% 13 = 7$$

$$H(84) = 84 \% 13 = 6$$

$$H(27) = 27 \% 13 = 1$$

$$H(55) = 55 \% 13 = 3$$

$$H(11) = 11 \% 13 = 11$$

$$H(10) = 10 \% 13 = 10$$

$$H(79) = 79 \% 13 = 1$$

用前插法将关键字 K 插到计算后的第 $H(K)$ (即 $K \% 13$) 个单链表上。结果如图 7.3 所示。

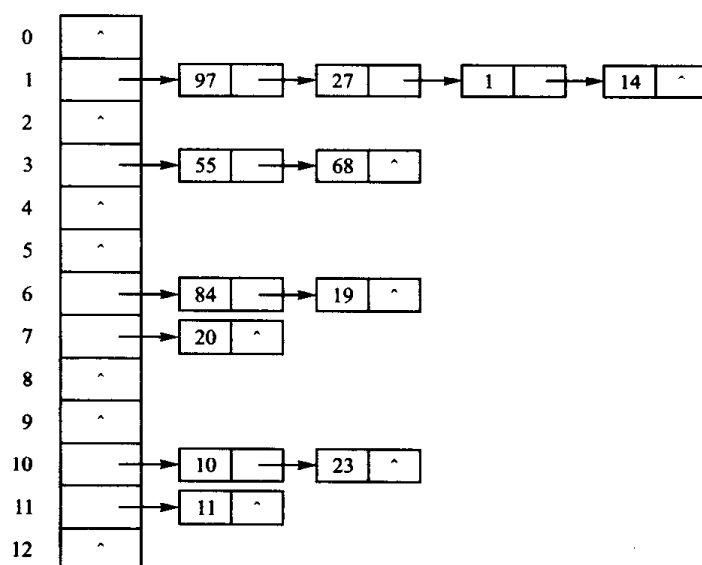


图 7.3 用链地址法解决散列冲突示意图

等概率下链地址法查找成功的平均查找长度:

$$LEN_{link} = (1 \times 6 + 2 \times 4 + 3 \times 1 + 4 \times 1) / 12 = 21 / 12 = 1.75$$

等概率下链地址法查找不成功的平均查找长度:

主要与装填因子有关,其次与待查值有关。

(2) 散列表用顺序表实现,用线性探测法解决散列冲突:

为了减少散列冲突,令装填因子 $\alpha = 0.75$, 因为 $n = 12$, 散列表长 $m = n / \alpha = 12 / 0.75 = 16$, 即散列表为 $HT[0 \dots 15]$ 。

根据散列函数 $H(K) = K \% 13$, 计算给定表中 12 个关键字序列的散列函数值, 即散列地址(计算结果如上一小题); 若地址是开放的(可用), 则插入新关键字; 否则, 求下一个开放地址, 再插入新关键字。结果如图 7.4 所示。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	1	68	27	55	19	20	84	79	23	11	10			
	1	2	1	4	3	1	1	3	9	1	1	3			

图 7.4 用线性探测法解决散列冲突

等概率下线性探测法查找成功的平均查找长度：

$$LEN_{hash} = (1 + 2 + 1 + 4 + 3 + 1 + 1 + 3 + 9 + 1 + 1 + 3) / 12 = 30 / 12 = 2.5$$

等概率下线性探测法查找不成功的平均查找长度：

主要与装填因子有关,其次与待查值有关。

第 8 章

排 序

8.1 基础知识篇

学习要点:

- 排序的基本概念和种类
- 了解和掌握五类基本排序:插入排序、交换排序、选择排序、归并排序及基数排序的基本思想和算法实现
- 了解各种排序算法的时间复杂度、空间复杂度和稳定性,能正确分析它们的性能及适用范围

8.1.1 排序的基本概念

排序是数据处理中一种重要的操作,如何进行高效率的排序,是程序设计中所面临的重要问题。

关键字:是数据元素中的某个数据项。如果某个数据项可以惟一地确定一个数据元素,就将其称为主关键字;否则,称为次关键字。

排序:是把一组无序的数据元素按照关键字值递增(或递减)重新排列。如果排序依据的是主关键字,排序的结果将是惟一的。

排序算法的稳定性:如果在待排序的记录序列中有多个关键字值相同的数据元素,经过排序后,这些数据元素的相对次序保持不变,则称这种排序算法是稳定的,否则称之为不稳定的。

内部排序与外部排序:根据在排序过程中待排序的所有数据元素是否全部被放置在内存中,可将排序方法分为内部排序和外部排序两大类。内部排序是指在排序的整个过程中,待排序的所有数据元素全部被放置在内存中;外部排序是指由于待排序的数据元素个数太多,不能同时放置在内存中,而需要将一部分数据元素放置在内存,另一部分数据元素放置在外存中,排序过程中需要在内外存之间多次交换数据才能得到排序的结果。本章只讨论常用的内部排序方法。

排序的基本方法:内部排序主要有 5 种方法,插入排序、交换排序、选择排序、归并排序和基数排序。

每一种内部排序均可在不同的存储结构上实现。通常有三种存储结构:

(1)以顺序表作为存储结构:排序过程是对各记录进行位置重排,即通过比较和判

断,把记录移到合适的位置。

(2) 以链表作为存储结构:排序过程无须移动记录,仅需修改指针即可。

(3) 采用辅助表排序:有的排序方法难以在链表上实现,此时若仍需要避免排序过程中的记录移动,可以建立一个辅助表,例如,由关键字和指向记录的指针组成的索引表。这样,排序过程中只需要对这个辅助表的表目进行物理重排,即只移动辅助表目而不移动记录本身。

8.1.2 插入排序

1. 直接插入排序

直接插入排序是一种比较简单的排序方法。它的基本思想是依次将记录序列中的每一个记录插入到有序段中,使有序段的长度不断地扩大。

其算法如下:

```
void InsertSort(Sqlist *L)                /* L 为指向顺序表的指针 */
{
    for(i=2; i<=L->length; ++i)
        if(L->r[i].key<L->r[i-1].key)    /* 若满足小于,则将 r[i] 插入有序子表 */
        {
            L->r[0] = L->r[i];            /* 将 r[i] 存入哨兵 r[0] 中 */
            for(j=i-1; L->r[0].key<L->r[j].key; --j)
                L->r[j+1] = L->r[j];      /* 记录后移 */
            L->r[j+1] = L->r[0];          /* 插入到正确位置 */
        }
}
```

当待排序记录较少时,直接插入排序速度较快,但是,当待排序的记录数量较大时,大量的比较和移动操作将使直接插入排序算法的效率降低;然而,当待排序的数据元素基本有序时,直接插入排序过程中记录的移动次数大大减少,从而效率会有所提高。插入排序是一种稳定的排序方法。

2. 希尔排序

希尔排序又称为缩小增量排序,其基本思想是将待排序的记录划分成几组,从而减少参与直接插入排序的数据量,当经过几次分组排序后,记录的排列已经基本有序,这个时候再对所有的记录实施直接插入排序。

具体步骤可以描述如下:假设待排序的记录为 n 个,先取整数 $d < n$,例如,取 $d = \lfloor n/2 \rfloor$ ($\lfloor n/2 \rfloor$ 表示不大于 $n/2$ 的最大整数),将所有距离为 d 的记录构成一组,从而将整个待排序记录序列分割成为 d 个子序列,对每个分组分别进行直接插入排序,然后再缩小间隔 d ,例如取 $d = d/2$,重复上述的分组,再对每个分组分别进行直接插入排序,直到最后取 $d = 1$,即将所有记录放在一组进行一次直接插入排序,最终将所有记录排列成按关键字有序的序列。

其算法如下:

```
void ShellInsert(Sqlist *L, int dk)        /* L 为指向顺序表的指针, dk 为增量 */
```

```

/* 对顺序表 L 做一趟插入排序 */
{
    for(i = dk + 1; i <= L.length; ++i)
        if(L->r[i].key < L->r[i - dk].key) /* 需将 r[i] 插入有序增量子表 */
        {
            L->r[0] = L->r[i];          /* 将 r[i] 暂存在 r[0] 中 */
            for(j = i - dk; j > 0 && L->r[0].key < L->r[j].key; j -= dk)
                L->r[j + dk] = L->r[j]; /* 记录后移, 查找插入位置 */
            L->r[j + dk] = L->r[0];      /* 插入 */
        }
}

void ShellSort(Sqlist *L, int dlta[], int t)
/* 按增量序列 dlta[0..t-1] 对顺序表 L 做希尔排序 */
{
    for(k = 0; k < t; ++k)
        ShellInsert(L, dlta[k]);      /* 一趟增量为 dlta[k] 的插入排序 */
}

```

8.1.3 交换排序

交换排序是指在排序过程中, 两两比较待排序的关键字, 若其次序相反, 则进行交换, 直到没有逆序的记录为止。

1. 冒泡排序

冒泡排序是交换排序中一种简单的排序方法。它的基本思想是对所有相邻记录的关键字值进行比较, 如果是逆序 ($a[j] > a[j+1]$), 则将其交换, 最终达到有序化。对由 n 个记录组成的记录序列, 最多经过 $(n-1)$ 趟冒泡排序, 就可以使记录序列成为有序序列。

其算法如下:

```

void Bubble(Sqlist *L, int n)          /* L 为指向顺序表的指针 */
{
    for(i = n - 1, change = TRUE; i >= 1 && change; --i) /* 控制共进行 n-1 趟排序 */
    {
        change = FALSE;                /* 每趟排序前 change 置 FALSE */
        for(j = 0; j < i; ++j)          /* 进行第 i 趟排序 */
            if(L->r[j] > L->r[j + 1])    /* 判断相邻的两记录是否为逆序 */
            {
                temp = L->r[j];          /* 是逆序, 交换两记录的位置 */
                L->r[j] = L->r[j + 1];
                L->r[j + 1] = temp;
                change = TRUE;           /* 标志位置 TRUE */
            }
    }
}

```

冒泡排序比较简单,当初始序列基本有序时,冒泡排序有较高的效率,反之效率较低;冒泡排序只需要一个记录的辅助空间来作为记录交换的中间暂存单元;冒泡排序是一种稳定的排序方法。

2. 快速排序

快速排序是由冒泡排序改进而得到的,它的基本思想是:通过比较关键字值交换记录,以某个记录为界(该记录称为支点),将待排序列分成两部分。其中,一部分所有记录的关键字值都大于等于支点记录的关键字值,另一部分所有记录的关键字值都小于支点记录的关键字值。将待排序列按关键字以支点记录分成两部分的过程,称为一次划分。

其算法如下:

```
int Partition(Sqlist *L,int low,int high)
/* 交换顺序表 L 中子表 r[low..high] 的记录,支点记录到位,并返回其所在的位置,此时在它之
前的记录均不大于它。 */
{
    L->r[0] = L->r[low];          /* 用子表的第一个记录作支点记录,将其暂存于 r[0] 中 */
    pivotkey = L->r[low].key;      /* 支点记录关键字 */
    while(low < high)
    {
        /* 从表的两端交替地向中间扫描 */
        while(low < high && L->r[high].key >= pivotkey) --high;
        L->r[low] = L->r[high];     /* 将比支点记录小的记录移到低端 */
        while(low < high && L->r[low].key <= pivotkey) ++low;
        L->r[high] = L->r[low];     /* 将比支点记录大的记录移到高端 */
    }
    L->r[low] = L->r[0];           /* 支点记录到位 */
    return low;                  /* 返回支点位置 */
}
```

到目前为止,快速排序是平均速度最快的一种排序方法,但当原始记录基本有序或基本逆序时,每一趟的排序有可能只将其余记录分成一部分,这样就降低了时间效率,所以快速排序适用于原始记录排列杂乱无章的情况,对各部分不断划分,直到整个序列按关键字有序。快速排序是一种不稳定的排序方法。

8.1.4 选择排序

选择排序的基本思想是:每一趟从待排序的记录中选出关键字值最小的记录,放在已排好序的子表的最后,直到全部记录排序完毕。选择排序有直接选择排序和堆排序两种方法。

1. 直接选择排序

直接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值最小的记录,并和第 i 个记录交换位置,直到整个序列按关键字有序。

其算法如下:

```
void SelectSort(Sqlist *L)
```

```

for(i=1; i<L->length; ++i)    /* 选择第 i 小的记录, 并交换到位 */
{
    j = SelectMinKey(L,i); /* 在 L->r[i..L.length] 中选择关键字值最小的记录 */
    if(i!=j)
    {
        temp = L->r[i];          /* 第 j 条记录与第 i 条记录交换位置 */
        L->r[i] = L->r[j];
        L->r[j] = temp;
    }
}

```

直接选择排序是一种不稳定的排序方法。

2. 堆排序

可以借助完全二叉树来描述堆。在一棵完全二叉树中, 如果所有结点的值均小于 (或均大于) 其左、右孩子结点的值, 按层对该二叉树进行遍历时得到的结点序列就是一个堆。

堆排序的基本思想是: 首先将待排序的记录序列构造为一个堆, 此时, 选出了堆中所有记录的最小者或最大者, 然后将它从堆中移走, 并将剩余的记录再调整成堆, 这样又找出了次小 (或次大) 的记录, 依此类推, 直到堆中只有一个记录为止, 得到的出堆序列就是有序序列。

其算法如下:

```

typedef SqList HeapType;          /* 堆采用顺序表存储结构 */
void HeapAdjust(HeapType *H, int s, int m) /* 将序列调整成一个大顶堆 */
{
    rc = H->r[s];
    for(j=2*s; j<=m; j*=2)
    {
        /* 沿 key 较大的孩子结点向下筛选 */
        if((j<m)&&(H->r[j].key<=H->r[j+1].key))
            ++j;
        /* j 为 key 较大的记录下标 */
        if(rc.key>H->r[j].key) break;
        /* rc 应插入到位置 s 上 */
        H->r[s] = H->r[j];
        s = j;
    }
    H->r[s] = rc;
    /* 插入 */
}

void HeapSort(HeapType *H, int k) /* 对顺序表 H 进行堆排序 */
{
    for(i=H->length/2; i>0; i--)
        HeapAdjust(H, i, H->length);
    for(i=H->length; i>1; i--)
        /* 把表建成大顶堆 */

```

```

    }
    temp = H->r[1];      /* 将堆顶记录和当前未经排序的子序列中的最后一个记录交换 */
    H->r[1] = H->r[i];
    H->r[i] = temp;
    HeapAdjust(H,1,i-1);      /* 将 H->r[1..i-1] 重新调整成堆 */
}
}

```

堆排序法对于大量记录的排序效率是较高的。堆排序是一种不稳定的排序方法。

8.1.5 归并排序

归并排序是另一类排序方法。所谓归并是指将两个或两个以上的有序表合并成一个新的有序表。归并排序的基本思想是将一个具有 n 个待排序记录的序列看成是 n 个长度为 1 的有序序列, 然后进行两两归并, 如此重复, 直至得到一个长度为 n 的有序序列为止。

其算法如下:

```

void Msort(RcdType SR[], RcdType TR1[], int s, int t)
/* 将 SR[s..t] 归并为有序序列 TR1[s..t] */
{
    if(s == t) TR1[s] = SR[s];
    else
    {
        m = (s + t) / 2;      /* 将 SR[s..t] 平分为 SR[s..m] 和 SR[m+1..t] */
        Msort(SR, TR2, s, m); /* 递归地将 SR[s..m] 归并为有序的 TR2[s..m] */
        Msort(SR, TR2, m + 1, t); /* 递归地将 SR[m+1..t] 归并为有序的 TR2[m+1..t] */
        Msort(TR2, TR1, s, t); /* 将 TR2[s..m] 和 TR2[m+1..t] 归并到 TR1[s..t] */
    }
}

void MergeSort(Sqlist *L) /* 对顺序表 L 作归并排序 */
{
    Msort(L->r, L->r, 1, L->length);
}

```

归并排序是一种稳定的排序方法。

8.1.6 基数排序

基数排序是一种基于多关键字的排序方法, 是将单关键字按基数 (RADIX) 分成“多关键字” (d 个) 进行排序的方法。从最低位关键字起, 按关键字的不同值将序列中的记录“分配”到 RADIX 个队列中, 然后再“收集”。如此重复 d 次即可。链式基数排序是用 RADIX 个链队列作为分配队列, 关键字相同的记录存入同一个链队列中, 收集则是将各链队列按关键字大小顺序链接起来。

基数排序适用于待排序的记录数目较多, 而其关键字位数较少, 且关键字每一位的基数相同的情况。若待排序记录的关键字有 d 位就需要进行 d 次“分配”与“收集”, 即共执

行 d 趟排序,因此,若 d 值较大,基数排序的时间效率就会随之降低。基数排序是一种稳定的排序方法。

8.1.7 内部排序法的比较和选择

表 8-1 内部排序法的比较

排序方法	平均时间	最坏情况	辅助存储
简单排序	$O(n^2)$	$O(n^2)$	$O(1)$
快速排序	$O(n \log_2 n)$	$O(n^2)$	$O(n \log_2 n)$
堆排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$
归并排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n)$
基数排序	$O(d(n + rd))$	$O(d(n + rd))$	$O(rd)$

说明:简单排序包括直接插入排序和冒泡排序。

选择排序方法时需考虑的因素有:

- (1) 待排序的记录个数 n ;
- (2) 记录本身的大小;
- (3) 关键字的分布情况;
- (4) 对排序稳定性的要求;
- (5) 语言工具的条件,辅助空间的大小等。

依据以上因素,在排序时可根据具体情况做出以下选择:

(1) 当待排序的一组记录数目 n 较小时,可采用插入排序和选择排序。由于插入排序所需的记录移动次数比直接选择排序多,因此当记录本身信息量较大时,用选择排序较好。

(2) 若 n 较大,则采用快速排序、堆排序或归并排序。

(3) 若待排序记录按关键字基本有序时,则适于采用直接插入排序或冒泡排序。

(4) 当 n 很大,而关键字位数较少时,采用链式基数排序较好。

8.2 实训篇

实训目的:

- 熟练应用插入排序、交换排序、选择排序、归并排序及基数排序法对数据记录序列进行升序或降序排序
- 利用直接插入排序法实现对链表的排序

8.2.1 药店的药品销售统计系统

案例描述

设计一系统,实现医药公司定期对各药品的销售记录进行统计,并按药品编号、单价、销售量或销售额做出排序。

设计分析

在本设计中,首先从数据文件中读出各药品的信息记录,存储在顺序表中。各药品的信息包括:药品编号、药品名称、单价、销售量、销售额。其中药品编号共4位,采用字母和数字混合编号,如:A125,前一位为大写字母,后三位为数字,按药品编号进行排序时,可采用基数排序法。按药品的单价、销售量或销售额进行排序时,可采用多种排序方法,如直接插入排序、冒泡排序、快速排序、直接选择排序等方法。在本设计中,对单价的排序采用冒泡排序法,对销售量的排序采用快速排序法,对销售额的排序采用堆排序法。

设计方案

(1) 药品信息的元素类型定义:

```
typedef struct node {
    char num[4];           /* 药品编号 */
    char name[10];        /* 药品名称 */
    float price;           /* 单价 */
    int count;             /* 销售量 */
    float sale;            /* 销售额 */
} DataType;
```

存储药品信息的顺序表的定义:

```
typedef struct {
    DataType r[MAXSIZE];
    int length;
} SequeList;
```

(2) 按药品编号升序排列采用基数排序法,其药品编号为字母和数字混合编号,因此要分别对1位字母和3位数字进行基数排序。其算法如下:

```
void distribute_num(DataType r[], int i, ArrType_num f, ArrType_num e)
/* 一趟数字字符分配函数 */
{
    int p, j, k;
    for(j=0; j<RADIX_num; ++j)
        f[j]=0;           /* 对各队列的队首结点初始化 */
    for(p=r[0].next; p; p=r[p].next) /* 每趟对链表中的所有关键字进行分离 */
    {
        j=r[p].num[i]*48;    /* 将该位数值型字符转换成其表达的数值 */
        if(! f[j]) f[j]=p;   /* 若对应队列为空,则插入队首 */
        else r[e[j]].next=p; /* 若对应队列非空,则插入队尾 */
        e[j]=p;
    }
}

void collect_num(DataType r[], ArrType_num f, ArrType_num e)
/* 一趟数字字符收集函数 */
{

```

```

int j,t;
for(j=0;! f[j];++j);          /* 查找第一个非空队列 */
r[0].next = f[j];              /* 将第一个非空队列链入 */
t = e[j];
while(j < RADIX_num - 1)       /* 将其他非空队列依次链入 */
{
    for(j = j + 1; j < RADIX_num - 1 && ! f[j]; ++j);
    if(f[j]) { r[t].next = f[j]; t = e[j]; }
}
r[t].next = 0;
}

void distribute_char(DataType r[],int i,ArrType_char f,ArrType_char e)
    /* 一趟字母字符分配函数 */
{
    int p,j,k;
    for(j=0;j < RADIX_char; ++j)
        f[j] = 0;              /* 对各队列的队首结点初始化 */
    for(p = r[0].next; p; p = r[p].next) /* 每趟对链表中的所有关键字进行分离 */
    {
        j = r[p].num[i] % 65;    /* 将该位字母字符转换成其表达的数值 */
        if(! f[j]) f[j] = p;    /* 若对应队列为空,则插入队首 */
        else r[e[j]].next = p;   /* 若对应队列非空,则插入队尾 */
        e[j] = p;
    }
}

void collect_char(DataType r[],ArrType_char f,ArrType_char e)
    /* 一趟字母字符收集函数 */
{
    int j,t;
    for(j=0;! f[j];++j);        /* 查找第一个非空队列 */
    r[0].next = f[j];           /* 将第一个非空队列链入 */
    t = e[j];
    while(j < RADIX_char - 1)    /* 将其他非空队列依次链入 */
    {
        for(j = j + 1; j < RADIX_char - 1 && ! f[j]; ++j);
        if(f[j]) { r[t].next = f[j]; t = e[j]; }
    }
    r[t].next = 0;
}

void RadixSort(SequenList *L) /* 对表 L 做基数排序 */

```

```

{
    int i,j;
    ArrType_num fn,en;
    ArrType_char fc,ec;
    for(i=0;i<L->length;i++) /* 将顺序表改造为静态链表,0号单元作为头结点 */
        L->r[i].next = i+1;
    L->r[L->length].next = 0;

    for(i=3;i>=1;--i)
    {
        /* 按最低位优先次序对各关键字进行分配和收集,先做低3位数字字符部分 */
        distribute_num(L->r,i,fn,en);
        collect_num(L->r,fn,en);
    }

    /* 对最高位的大写字母字符进行分配和收集 */
    distribute_char(L->r,0,fc,ec);
    collect_char(L->r,fc,ec);
}

```

(3) 按销售量降序排列采用快速排序法。其算法如下:

```

/* 一趟快速排序算法 */
int Partition(SequenList *H,int s,int t)
/* 将基准记录移到正确位置,并返回其所在位置 */
{
    int x,i,j;
    DataType rp;
    i=s; j=t; x=H->r[i].count; rp=H->r[i]; /* 基准记录暂存入rp,关键字存入x */
    while(i<j) /* 从序列的两端交替向中间扫描 */
    {
        while(i<j&&H->r[j].count<=x) j--; /* 扫描比基准记录小的位置 */
        H->r[i]=H->r[j]; /* 将比基准记录小的记录移到低端 */
        while(i<j&&H->r[i].count>=x) i++; /* 扫描比基准记录大的位置 */
        H->r[j]=H->r[i]; /* 将比基准记录大的记录移到高端 */
    }
    H->r[i]=rp; /* 基准记录到位 */
    return i; /* 返回基准记录位置 */
}

/* 快速排序递归算法 */
void Qsort(SequenList *H,int s,int t)
{
    int k;
    if(s<t) /* 长度大于1 */
    {

```

```

    k=Partition(H,s,t);          /* 调用一趟快速排序算法,将 r[s]...r[t]一分为二 */
    Qsort(H,s,k-1);              /* 对低端子序列递归排序,k 是支点位置 */
    Qsort(H,k+1,t);              /* 对高端子序列递归排序 */
}
}

```

(4) 按单价降序排列采用冒泡排序法。其算法如下:

```

void BubbleSort(SequenList *H)
{
    int i,j,mark;
    DataType temp;
    for(i=1;i<H->length;i++) /* 控制共进行 H->length-1 趟排序 */
    {
        mark=0;                /* 每趟排序前 mark 置 0 */
        for(j=1;j<=H->length-i;j++) /* 进行第 i 趟排序 */
            if(H->r[j].price>H->r[j+1].price) /* 判断相邻的两记录是否为逆序 */
            {
                temp=H->r[j];H->r[j]=H->r[j+1];H->r[j+1]=temp;
                /* 交换两记录的位置 */
                mark=1;          /* 标志位置 1 */
            }
        if(mark==0)              /* mark 为 0,说明本次没有记录交换,则整个排序结束 */
            break;
    }
}

```

(5) 按销售额降序排列采用堆排序法。其算法如下:

```

/* 堆的调整 */
void HeapAdjust(SequenList *H,int s,int m) /* 将序列调整成一个堆 */
{
    int j;
    DataType rc;
    rc=H->r[s];
    for(j=2*s;j<=m;j*=2) /* 沿 key 较小的孩子结点向下筛选 */
    {
        if((j<m)&&(H->r[j].sale>=H->r[j+1].sale)) ++j;
        /* j 为 sale 较小的记录下标 */
        if(rc.sale<H->r[j].sale) break; /* rc 应插入到位置 s 上 */
        H->r[s]=H->r[j];
        s=j;
    }
    H->r[s]=rc; /* 插入 */
}

```

```

void HeapSort(SequenList *H)                                /* 对顺序表 H 进行堆排序 */
{
    DataType temp;
    int i,n,m;
    n = H->length;
    for(i = n/2; i > 0; i--)                                /* 把表建成小顶堆 */
        HeapAdjust(H,i,n);
    for(i = H->length; i > 1; i--)
    {
        temp = H->r[1];          /* 将堆顶记录和当前未经排序子序列中的最后一个记录交换 */
        H->r[1] = H->r[i];
        H->r[i] = temp;
        HeapAdjust(H,1,i-1);    /* 将 H->r[1..i-1] 重新调整成堆 */
    }
}

```

(6) 程序实现

```

main()
{
    SequenList H;
    int i,k,x,t=0;
    H.length=0;
    Creat(&H);
    while(1)
    {
        printf("1.按药品编号排序 2.按销售量排序 3.按单价排序 4.按销售额排序 0.退出\n");
        scanf("%d",&x);
        switch(x)
        {
            case 1: RadixSort(&H);                          /* 按药品编号排序 */
                    for(k = H.r[0].next; k != H.r[k].next)
                    {
                        printf("%s %s %f %d %f\n",H.r[k].num,H.r[k].name,
                                H.r[i].price,H.r[i].count,H.r[i].sale);
                        t++;
                        if(t%20==0) getch();                 /* 输入 20 条记录后暂停,按任意键继续 */
                    }
                    break;
            case 2: Qsort(&H,1,H.length);                   /* 按销售量排序 */
                    for(i=1; i<=H.length; i++)
                    {
                        printf("%s %s %d\n",H.r[i].num,H.r[i].name,H.r[i].count);
                        if(i%20==0) getch();
                    }
        }
    }
}

```

```

        break;
case 3: BubbleSort(&H);                /* 按单价排序 */
        for(i=1;i<=H.length;i++)
        {
            printf("%s %s %f\n",H.r[i].num,H.r[i].name,H.r[i].price);
            if(i%20==0) getch();
        }
        break;
case 4: HeapSort(&H);                  /* 按销售额排序 */
        for(i=1;i<=H.length;i++)
        {
            printf("%s %s %f\n",H.r[i].num,H.r[i].name,H.r[i].sale);
            if(i%20==0) getch();
        }
        break;
case 0: exit(0);                      /* 退出 */
}
}

```

(7) 运行测试

数据文件 yaopin.txt 的内容如下:

药品编号	药品名称	单价	销售量	销售额
B137	左旋多巴	10	8	80
E107	乳果糖	2	35	70
D118	氮霉素	2	50	100
C129	甲地孕酮	3	17	51
B112	倍他司汀	5	32	160
A103	硝酸甘油	3	76	228
D110	头孢米诺	12	87	1044
A153	降压灵	2	121	242
D120	诺氟沙星	6	56	336
B106	甘露醇	2	72	144
D101	青霉素	3	98	294
C101	氮芥	7	12	84
A110	复方利血平	9	49	441
C108	甲氨蝶呤	4	11	44
Z019	鱼腥草注射剂	2	59	118
Y071	鱼腥草素	1	26	26
Z059	丹参	15	55	825
Z006	香丹	13	80	1040
Y822	复方土槿皮	3	19	57
Y517	天麻素	17	51	867
W078	乙酰天麻素	19	12	228

编译运行后显示:

1.按药品编号排序 2.按销售量排序 3.按单价排序 4.按销售额排序 0.退出
输入 1(回车)

显示:

A103	硝酸甘油	3	76	228
A110	复方利血平	9	49	441
A153	降压灵	2	121	242
B106	甘露醇	2	72	144
B112	倍他司汀	5	32	160
B137	左旋多巴	10	8	80
C101	氮芥	7	12	84
C108	甲氨蝶呤	4	11	44
C129	甲地孕酮	3	17	51
D101	青霉素	3	98	294
D110	头孢米诺	12	87	1044
D118	氮霉素	2	50	100
D120	诺氟沙星	6	56	336
E107	乳果糖	2	35	70
W078	乙酰天麻素	19	12	228
Y071	鱼腥草素	1	26	26
Y517	天麻素	17	51	867
Y822	复方土槿皮	3	19	57
Z006	香丹	13	80	1040
Z019	鱼腥草注射剂	2	59	118
Z059	丹参	15	55	825

1.按药品编号排序 2.按销售量排序 3.按单价排序 4.按销售额排序 0.退出
输入 2(回车)

显示:

A153	降压灵	121
D101	青霉素	98
D110	头孢米诺	87
Z006	香丹	80
A103	硝酸甘油	76
B106	甘露醇	72
Z019	鱼腥草注射剂	59
D120	诺氟沙星	56
Z059	丹参	55
Y517	天麻素	51
D118	氮霉素	50
A110	复方利血平	49
E107	乳果糖	35
B112	倍他司汀	32

Y071	鱼腥草素	26
Y822	复方土槿皮	19
C129	甲地孕酮	17
C101	氮芥	12
W078	乙酰天麻素	12
C108	甲氨蝶呤	11
B137	左旋多巴	8

1.按药品编号排序 2.按销售量排序 3.按单价排序 4.按销售额排序 0.退出

输入 3(回车)

显示:

Y071	鱼腥草素	1
A153	降压灵	2
B106	甘露醇	2
Z019	鱼腥草注射剂	2
D118	氮霉素	2
E107	乳果糖	2
D101	青霉素	3
A103	硝酸甘油	3
Y822	复方土槿皮	3
C129	甲地孕酮	3
C108	甲氨蝶呤	4
B112	倍他司汀	5
D120	诺氟沙星	6
C101	氮芥	7
A110	复方利血平	9
B137	左旋多巴	10
D110	头孢米诺	12
Z006	香丹	13
Z059	丹参	15
Y517	天麻素	17
W078	乙酰天麻素	19

1.按药品编号排序 2.按销售量排序 3.按单价排序 4.按销售额排序 0.退出

输入 4(回车)

显示:

D110	头孢米诺	1044
Z006	香丹	1040
Y517	天麻素	867
Z059	丹参	825
A110	复方利血平	441
D120	诺氟沙星	336
D101	青霉素	294
A153	降压灵	242

A103	硝酸甘油	228
w078	乙酰天麻素	228
B112	倍他司汀	160
B106	甘露醇	144
Z019	鱼腥草注射剂	118
D118	氮霉素	100
C101	氮地芥	84
B137	左旋多巴	80
E107	乳果糖	70
Y822	复方土槿皮	57
C129	甲地孕酮	51
C108	甲氨蝶呤	44
Y071	鱼腥草素	26

1.按药品编号排序 2.按销售量排序 3.按单价排序 4.按销售额排序 0.退出

输入0(回车)

退出程序

8.2.2 学生成绩排名

案例描述

设计一程序将文件中的学生信息读出,按读出的先后顺序建立一链表,然后对链表中的学生记录按总分降序排序。

设计分析

教材中的大部分排序方法都是以顺序存储为基础的,因为对顺序表中的记录可以按下标直接定位,所以对其排序比较方便。在本例中要对链表进行排序,对链表中的记录只能采用顺序读取,不能直接定位到某条记录,所以大部分排序方法不能用于链表的排序,因此在本例中只能采用直接插入排序法。

具体过程为:以链表中的第一个结点作为初始的有序序列,从第二个结点开始进行插入,在插入前首先判断它的关键字是否大于前一个结点的关键字,若是,则不需要进行插入操作;若其关键字小于前一个结点的关键字,则从表头向后搜索有序序列,找到其应该插入的位置,然后修改指针,完成插入。按照以上步骤插入第三、第四、第五……结点,直到所有的结点都插入完毕,则整个排序结束。

在本设计中,整个排序过程完全利用初始链表中的结点,不需要开辟新的结点空间。

设计方案

(1) 存储类型的定义

学生信息存储类型的定义:

```
typedef struct student
{
    char num[6];           /* 学号 */
    char name[8];          /* 姓名 */
    int tax;               /* 名次 */
}
```

```

int english;          /* 英语成绩 */
int math;             /* 数学成绩 */
int score;            /* 总分 */
struct student *next; /* 指针域 */
|Node, *LinkList;

```

(2) 初始链表的建立

依次从文件中读出学生的信息,每读出一个,为其创建一个结点,然后将其插入到链表的尾部,直到所有的记录都被读出,相应的链表建立完毕。

```

LinkList CreateList() /* 从文件中依次读出学生信息,创建结点,建立链表 */
{
    LinkList head,p,s;
    FILE *fp;
    head=(Node *)malloc(sizeof(Node)); /* 创建头结点 */
    head->next=NULL;
    p=head;
    fp=fopen("d:\\tc\\c\\student.txt","r");
    if(!fp)
        printf("\n 数据文件无法打开!");
    else
    {
        while(!feof(fp))
        {
            s=(Node *)malloc(sizeof(Node)); /* 创建学生信息结点 */
            fscanf(fp,"%s %s %d %d\n",s->num,s->name,&(s->english),&(s->math));
            /* 将学生的学号、姓名、英语和数学成绩读出,并存入结点中 */
            s->score=s->english+s->math; /* 总分为英语和数学成绩之和 */
            s->next=p->next; /* 将新建立的结点插入链表尾部 */
            p->next=s;
            p=s;
        }
        fclose(fp);
    }
    return head; /* 返回指向头结点的指针 */
}

```

(3) 利用直接插入排序法对链表按总分进行降序排列

```

void InsertSort(LinkList head) /* 对链表按总分用直接插入排序法进行降序排序 */
{
    LinkList p,q,s,t;
    p=head->next;
    for(q=p->next;q!=NULL;)
    {

```

```

    if(q->score > p->score)
        /* 若当前结点的总分大于前一结点的总分,则从链表头开始向后扫描已存在的有序序列,查找其应该插入的位置,然后将其插入 */
        {
            for(s = head; s->next->score > q->score; s = s->next);
            t = q->next;
            q->next = s->next;
            s->next = q;
            p->next = NULL;
            q = t;
            /* q 指向下一个待插入的记录 */
        }
    else { p = q; q = q->next; }
        /* 若当前结点的总分小于前一结点的总分,则不需进行插入操作 */
    }
}

```

(4) 对所有的记录按总分排名次。在计算某个记录的名次时,要与前一条记录进行比较,若总分相同,则名次与前一条记录的名次相同。因此,计算完某条记录的名次后,要将这条记录的地址保留。

```

void PlaceList(LinkList head)
/* 按总分进行排名,总分相同则名次也相同 */
{
    LinkList p, q;
    int i = 1;
    p = head->next;
    p->tax = 1;
    /* 第一个结点的名次为 1 */
    q = p;
    p = p->next;
    while(p != NULL)
    /* 对剩下的结点进行排名 */
    {
        if(p->score == q->score)
            /* 若当前待排名结点的总分与前一结点的总分相同,则其名次也与前一结点相同 */
            p->tax = q->tax;
        else
        {
            i++;
            /* 名次加 1 */
            p->tax = i;
        }
        q = p;
        /* 将当前记录的地址保留在 q 中 */
        p = p->next;
        /* p 指向下一条待排名的记录 */
    }
}

```

(5) 程序实现

```

#include "stdio.h"
#include "stdlib.h"
typedef struct student{
    char num[6];
    char name[8];
    int tax;
    int english;
    int math;
    int score;
    struct student * next;
}Node, * LinkList;

void main()
{
    LinkList head,p;
    head=CreateList();
    InsertSort(head);           /* 排序 */
    PlaceList(head);           /* 排名次 */
    for(p=head->next;p!=NULL;p=p->next) /* 输出排序和排名后的学生记录 */
        printf("%d %s %s %d %d %d\n",p->tax,p->num,p->name,p->english,
            p->math,p->score);
}

```

(6) 运行测试

数据文件 student.txt 的内容为:

10000	王强	90	75
10001	李盛	53	60
10002	张国利	76	83
10003	刘丽丽	65	82
10004	陈俊	85	90
10005	赵文文	77	82
10006	钱浩然	93	95
10007	宋容容	82	77
10008	周军	66	77
10009	郑发亮	81	59
10010	朱静	73	88

编译运行后显示结果为:

1	10006	钱浩然	93	95	188
2	10004	陈俊	85	90	175
3	10000	王强	90	75	165
4	10010	朱静	73	88	161

5	10007	宋容容	82	77	159
5	10005	赵文文	77	82	159
5	10002	张国利	76	83	159
6	10003	刘丽丽	65	82	147
7	10008	周军	66	77	143
8	10009	郑发亮	81	59	140
9	10001	李盛	53	60	113

8.2.3 电视大赛观众投票及排名系统

案例描述

在很多电视大赛中,通常当选手表演结束后,现场观众通过手中的按键对参赛选手进行投票,然后对选手获得的票数进行统计,从高到低进行降序排列,从而自动产生冠军、亚军和季军。现要求编写一程序模拟实现上述系统的功能。

设计分析

在本例中,首先输入参赛选手的人数(范围为1~9个),然后根据人数通过 malloc 函数来开辟存放选手信息的顺序表。将选手的编号和姓名依次存入顺序表单元中,观众通过按键进行投票,按“1”表示为1号选手投票,按“2”表示为2号选手投票,依次类推,以按“0”作为投票结束标志。投票结束后进行排序,在此采用希尔排序,然后为每个选手计算名次,得票相同的名次也相同,其处理方法与实训8.2.2一致。

设计方案

(1) 存储类型的定义

参赛选手信息存储类型的定义:

```
typedef struct node
{
    char name[8];           /* 选手姓名 */
    int num;                /* 选手编号 */
    int score;              /* 选手得分 */
    int tax;                /* 选手名次 */
} Node;
```

(2) 开辟空间用于构造存放选手信息的顺序表 R: $R = (\text{Node} *) \text{malloc}(n * \text{sizeof}(\text{Node}))$; 其中 n 为参赛选手的人数,在此采用动态空间分配,而不是在开始时直接开辟静态数组,这样是为了避免空间的不足或浪费。

在投票时,按“1”为1号选手投票的实现:

```
scanf("%c",&k);           /* 观众按键 */
R[k-48].score = R[k-48].score + 1;
```

若观众输入的是“1”,则“1”-48即为“1”的ASCII码值-48=1,因此可以实现对1号选手的票数加1,即 $R[1] = R[1] + 1$ 。对其他选手的投票与此相同。

(3) 采用希尔排序法对所有选手的票数进行降序排列。

```
void ShellInsert(Node r[],int dk,int length)
```

/* 希尔排序,dk 为第一个步长值,length 为待排序表的长度 */

```

{
    int i,j;
    while(dk >= 1)                /* 步长 dk >= 1,对每组进行直接插入排序 */
    {
        for(i = dk + 1; i <= length; i++) /* 对每组进行直接插入排序 */
        {
            r[0] = r[i];           /* 记录 r[i] 暂存入 r[0] 中 */
            j = i - dk;             /* 确定每组中记录 r[i] 的前一位置 */
            while(j >= 1 && r[0].score > r[j].score) /* 在组中查找插入位置 */
            {
                r[j + dk] = r[j]; /* 记录后移 */
                j = j - dk;        /* 记录位置前移一个步长 */
            }
            r[j + dk] = r[0];      /* 插入记录 */
        }
        dk = dk / 2;              /* 缩小步长值 */
    }
}

```

(4) 程序实现

```

#include "stdio.h"
#include "stdlib.h"
typedef struct node {
    char name[8];           /* 选手姓名 */
    int num;                /* 选手编号 */
    int score;              /* 选手得分 */
    int tax;                /* 选手名次 */
} Node;

main()
{
    int n,i,j,t;
    char k;
    Node * R;
    printf("请输入选手人数:");
    scanf("%d",&n);
    R = (Node *)malloc(n * sizeof(Node)); /* 开辟用于存放选手信息的顺序空间 */
    for(i = 1; i <= n; i++)
    {
        printf("请输入选手姓名:");
        scanf("%s",R[i].name);
        R[i].num = i;        /* 输入次序即为选手的编号 */
    }
}

```

```

    R[i].score=0;                /* 得分的初始值为0 */
}
for(i=1;i<=n;i++)
    printf("%d号选手:%s\n",R[i].num,R[i].name);
printf("请观众为选手投票:");
do                                /* 对选手进行投票,以'0'结束 */
{
    scanf("%c",&k);
    R[k-48].score=R[k-48].score+1;
}while(k!='0');
for(i=1;i<=n;i++)
    printf("%d号选手—%s 得分:%d\n",R[i].num,R[i].name,R[i].score);
ShellInsert(R,n/2,n);            /* 对选手按得分进行希尔排序,初始步长为n/2 */
R[1].tax=1;                      /* 计算每个选手的名次,得分相同,名次相同 */
j=1;
t=2;
for(i=2;i<=n;i++)
{
    if(R[i].score==R[j].score)
        R[i].tax=R[j].tax;
    else
    { R[i].tax=t; t++; }
    j=i;
}
printf("*** 电视大赛获奖名单:***\n");
for(i=1;R[i].tax<=3&&i<=n;i++)
    /* 输出名次为1-3的选手,分别为冠、亚、季军 */
{
    if(R[i].tax==1)
        printf("冠军—%d号选手:%s 得分%d\n",R[i].num,R[i].name,R[i].score);
    if(R[i].tax==2)
        printf("亚军—%d号选手:%s 得分%d\n",R[i].num,R[i].name,R[i].score);
    if(R[i].tax==3)
        printf("季军—%d号选手:%s 得分%d\n",R[i].num,R[i].name,R[i].score);
}
}

```

(5) 运行测试

编译运行后显示:

请输入选手人数:7(回车)

请输入选手姓名:张丽丽(回车)

请输入选手姓名:何杰(回车)

```

请输入选手姓名: 毛毛 (回车)
请输入选手姓名: 刘学琼 (回车)
请输入选手姓名: 林雨萱 (回车)
请输入选手姓名: 曹睿睿 (回车)
请输入选手姓名: 王楠囡 (回车)
1 号选手: 张丽丽
2 号选手: 何杰
3 号选手: 毛毛
4 号选手: 刘学琼
5 号选手: 林雨萱
6 号选手: 曹睿睿
7 号选手: 王楠囡
请观众为选手投票: 2 7 6 1 2 5 4 3 2 2 5 7 6 6 1 4 5 5 7 7 1 1 1 7 3 4 4 3 3 2 1 2 2 2 7 1 1 3 4 5 5 5
2 5 5 2 4 1 2 4 3 5 7 1 1 5 2 3 3 4 3 3 2 4 2 4 3 5 3 6 5 6 1 2 3 4 5 4 6 5 7 4 5 3 5 4 2 0 (回车)
1 号选手—张丽丽 得分: 12
2 号选手—何杰 得分: 16
3 号选手—毛毛 得分: 14
4 号选手—刘学琼 得分: 14
5 号选手—林雨萱 得分: 17
6 号选手—曹睿睿 得分: 6
7 号选手—王楠囡 得分: 8
* * * 电视大赛获奖名单: * * *
冠军—5 号选手: 林雨萱 得分 17
亚军—2 号选手: 何杰 得分 16
季军—4 号选手: 刘学琼 得分 14
季军—3 号选手: 毛毛 得分 14
    
```

8.3 实训项目

项目内容

设计一程序实现对列车班次信息进行排序和查找。列车的班次为 5 位编码, 如 K1038, 第一位为大写字母, 后四位为数字。要求程序能按照列车的编号对所有班次的列车进行排序, 然后可以按编号查询相应列车的信息: 起点站、终点站、到达时间、票价。还可以按起点站和终点站来查询列车信息。

设计要求

按照输入的列车编号快速查询出其信息, 并输出。

项目提示

可采用基数排序法对所有班次的列车信息进行排序, 利用二分查找法对排好序的列车信息按编号进行快速查找, 按起点站和终点站来查询列车信息时可以采用顺序查找方法。

8.4 基本题

8.4.1 选择题

- 下列排序算法中,_____算法可能在初始数据有序时,花费的时间反而最多。
A. 堆排序 B. 冒泡排序 C. 快速排序 D. 插入排序
- 对包含 N 个元素的散列表进行检索,平均查找长度为_____。
A. $O(\log_2 N)$ B. $O(N)$
C. 不直接依赖于 N D. 上述说法都不对
- 用 n 个键值构造一棵二叉排序树,最低高度为_____。
A. $n/2$ B. n C. $\lfloor \log_2 n \rfloor$ D. $\lfloor \log_2 n \rfloor + 1$
- 在各种排序方法中,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)的一端的方法是_____。
A. 插入排序 B. 希尔排序 C. 选择排序 D. 归并排序
- 一组记录的关键字为(46,79,56,38,40,84),则利用堆排序的方法建立的初始堆为_____。
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
- 对具有 8 个元素的序列(49,38,65,97,76,13,27,50),按升序排序,采用快速排序法时第一趟的结果为_____。
A. 13,65,38,97,76,49,27,50 B. 13,27,38,49,50,65,76,97
C. 97,76,65,50,49,38,27,13 D. 13,38,65,97,76,49,27,50

8.4.2 填空题

- 在插入排序、选择排序、快速排序和归并排序中,平均查找时间最少的是_____,要求存储量最大的是_____。
- 用冒泡法对 n 个关键字排序,在最好的情况下,只需做_____次比较和_____次移动;在最坏的情况下,要做_____次比较。
- 在快速排序和堆排序中,若待排序记录序列接近正序或逆序,则应该选用_____,若待排序记录序列无序,则应该选用_____。
- 设顺序表中有 1000 个元素,用折半查找时,最大比较次数为_____,最小比较次数为_____。
- 已知关键字序列为(20,15,14,18,21,36,40,10),采用快速排序法对其排序,第一趟排序后的关键字序列为_____。
- 在下面冒泡排序算法中填入适当的语句,使该算法在发现有序能及时停止排序。

```
void Bubble(Rectype R[])
```

```

{
    int i,j,exchang;
    Rectype temp;
    i = 1;
    do{
        exchang = False;
        for(j = n; j >= ①; j --)
            if(R[j] < R[j - 1])
            {
                temp = R[j - 1];
                R[j - 1] = R[j];
                R[j] = temp;
                ②;
            }
        ③;
    } while(④);
}

```

8.5 习题解析

1. 已知有 8 个整数:1,7,3,2,0,5,6,8,分别用下列方法进行排序,编写程序。

- (1) 直接插入排序;
- (2) 折半插入排序;
- (3) 希尔排序。

解 算法由主函数和直接插入排序、折半插入排序、希尔排序、输出五个函数组成。
程序如下:

```

#include <stdio.h>
#define LENGTH 8
void InsertSort(int r[],int n)          /* 直接插入排序 */
{
    int i,j;
    for(i = 2; i < n; ++i)               /* 认为第一个数有序,i 从 2 开始 */
        if(r[i] < r[i - 1])             /* 若小于,将 r[i]插入有序序列中 */
        {
            r[0] = r[i];                /* r[i]的值放入监视哨中 */
            for(j = i - 1; r[0] < r[j]; --j)
                r[j + 1] = r[j];         /* 记录后移 */
            r[j + 1] = r[0];             /* 插入到正确位置 */
        }
    printdata(r,n);
}

```

```
void BinSort(int r[],int n)
```

```
{
    int i,j,low,high,m;
    for(i=2;i<n;++i)
    {
        r[0]=r[i];
        low=1;
        high=i-1;
        while(low<=high)
        {
            m=(low+high)/2;
            if(r[0]<r[m])
                high=m-1;
            else
                low=m+1;
        }
        for(j=i-1;j>=high+1;--j)
            r[j+1]=r[j];
        r[high+1]=r[0];
    }
    printdata(r,n);
}
```

```
void ShellSort(int r[],int n)
```

```
{
    int i,j,d;
    d=n/2;
    while(d>=1)
    {
        for(i=d;i<n;i++)
        {
            r[0]=r[i];
            j=i-d;
            while((j>0)&&(r[0]<r[j]))
            {
                r[j+d]=r[j];
                j=j-d;
            }
            r[j+d]=r[0];
        }
        d=d/2;
    }
}
```

```
/* 折半插入排序 */
```

```
/* 定义变量,其中 low,high 表示查找的上下界 */
```

```
/* 认为第一个数有序,i 从 2 开始 */
```

```
/* 将 r[i] 暂时存入 r[0] 中 */
```

```
/* 置有序序列区间的初值 */
```

```
/* 从 r[low] 到 r[high] 折半查找插入位置 */
```

```
/* 折半,取中间位置送 m */
```

```
/* 插入位置在低半区 */
```

```
/* 插入位置在高半区 */
```

```
/* 插入位置以后的记录后移 */
```

```
/* 插入记录 */
```

```
/* 希尔排序 */
```

```
/* 取第一个步长值 */
```

```
/* 步长 d>=1 */
```

```
/* 对每组进行直接插入排序 */
```

```
/* 记录 r[i] 暂存入 r[0] 中 */
```

```
/* 确定每组中的记录 r[i] 前一个位置 */
```

```
/* 在组中查找插入位置 */
```

```
/* 记录后移 */
```

```
/* 记录位置前移一个步长 */
```

```
/* 插入记录 */
```

```
/* 缩小步长值 */
```

```

    printdata(r,n);
}

void printdata(int r[],int n)
{
    int i;
    for(i=1;i<n;i++)
        printf("%d",r[i]);
}

main()                                /* 主程序 */
{
    int r[LENGTH+1] = {0,1,7,3,2,0,5,6,8}; /* 定义数组并赋初值 */
    int r1[LENGTH+1];                     /* 暂用数组 */
    int i,n=LENGTH+1;
    for(i=0;i<=LENGTH;i++)                /* 复制数组 */
        r1[i] = r[i];
    printf("\nInsertSort output: ");
    InsertSort(r1,n);                     /* 直接插入排序并输出 */
    for(i=0;i<LENGTH;i++)                  /* 复制数组 */
        r1[i] = r[i];
    printf("\nBinSort output: ");
    BinSort(r1,n);                         /* 折半插入排序并输出 */
    for(i=0;i<LENGTH;i++)                  /* 复制数组 */
        r1[i] = r[i];
    printf("\nShellSort output: ");
    ShellSort(r1,n);                       /* 希尔排序并输出 */
}

```

输出结果为:

InsertSort output: 0 1 2 3 5 6 7 8

BinSort output: 0 1 2 3 5 6 7 8

ShellSort output: 0 1 2 3 5 6 7 8

2. 对无序序列:(70,73,69,23,93,18,11,68),分别用下列方法进行排序,编写程序。

(1) 快速排序;

(2) 简单选择排序。

解 算法由主函数和快速排序递归算法、简单选择排序、输出四个函数组成。

程序如下:

```

#include <stdio.h>
#define LENGTH 8

void QuickSort(int r[],int m,int n)      /* 快速排序递归算法 */

```

```

{
    int i,j;
    int temp;
    if(m<n)
    {
        i=m;
        j=n;
        temp=r[i];
        while(i!=j)
        {
            while((r[j]>=temp)&&(i<j)) j--; /* 扫描比基准记录小的位置 */
            if(i<j)
            {
                r[i]=r[j]; /* 将比基准记录小的记录移到低端 */
                i++;
            }
            while((r[i]<=temp)&&(i<j)) i++; /* 扫描比基准记录大的位置 */
            if(i<j)
            {
                r[j]=r[i]; /* 将比基准记录大的记录移到高端 */
                j--;
            }
        }
        r[i]=temp; /* 基准记录到位 */
        if(m<j) QuickSort(r,m,i-1); /* 对低端子序列递归排序 */
        if(i<n) QuickSort(r,i+1,n); /* 对高端子序列递归排序 */
    }
}

void SelectSort(int r[],int n) /* 直接选择排序 */
{
    int i,j,min;
    int swap;
    for(i=0;i<n;i++)
    {
        min=i;
        for(j=i+1;j<n;j++)
            if(r[j]<r[min])
                min=j; /* 存放相对较小关键字的下标 */
        if(min!=i)
        {
            swap=r[min]; /* 交换记录 */
            r[min]=r[i];
            r[i]=swap;
        }
    }
}

```

```

        r[min] = r[i];
        r[i] = swap;
    }
}

printrdata(r,n);
}

void printrdata(int r[],int n)
{
    int i;
    for(i=0;i<n;i++)
        printf("%d ",r[i]);
}

void main()                                /* 主程序 */
{
    int r[LENGTH] = {70,73,69,23,93,18,11,68}; /* 定义数组并赋初值 */
    int r1[LENGTH];                          /* 暂用数组 */
    int i,m=0,n=LENGTH;
    for(i=0;i<LENGTH;i++)                  /* 复制数组 */
        r1[i] = r[i];
    printf("\nQuickSort output: ");
    QuickSort(r,m,n-1);                    /* 快速排序递归算法 */
    printrdata(r,n);                        /* 快速排序后输出 */
    for(i=0;i<LENGTH;i++)
        r1[i] = r[i];
    printf("\nSelectSort output: ");
    SelectSort(r,n);                       /* 直接选择排序并输出 */
}

```

输出结果为:

QuickSort output: 11 18 23 68 69 70 73 93

SelectSort output: 11 18 23 68 69 70 73 93

3. 判别以下序列是否为堆(大顶堆或小顶堆)。如果不是,则把它调整为堆。

(1) (100,86,48,73,35,39,42,57,66,21)

(2) (12,70,33,65,24,56,48,92,86,33)

(3) (103,97,56,38,66,23,42,12,30,52,6,20)

(4) (5,56,20,23,40,38,29,61,35,76,28,100)

解 (1)将序列(100,86,48,73,35,39,42,57,66,21)构造表如下:

序号	1	2	3	4	5	6	7	8	9	10
结点值	100	86	48	73	35	39	42	57	66	21
左、右儿子	86,48	73,35	39,42	57,66	21					

表中序号与结点之间有这样的关系:当某结点的序号为偶数时,其二分之一值为该结点的父亲结点的序号;当某结点的序号(序号>1)为奇数时,其减一后的二分之一值为该结点的父亲结点的序号。观察表中序号1到5的结点,发现父结点的值均大于其左右儿子的值,故该序列是大顶堆。

(2)将序列(12,70,33,65,24,56,48,92,86,33)构造成二叉树如图8.1所示:观察图(a),因为 $12 < 70$,而 $70 > 24$,所以不是堆。
对图(a)进行堆调整,过程见图(b),结果见图(c)。

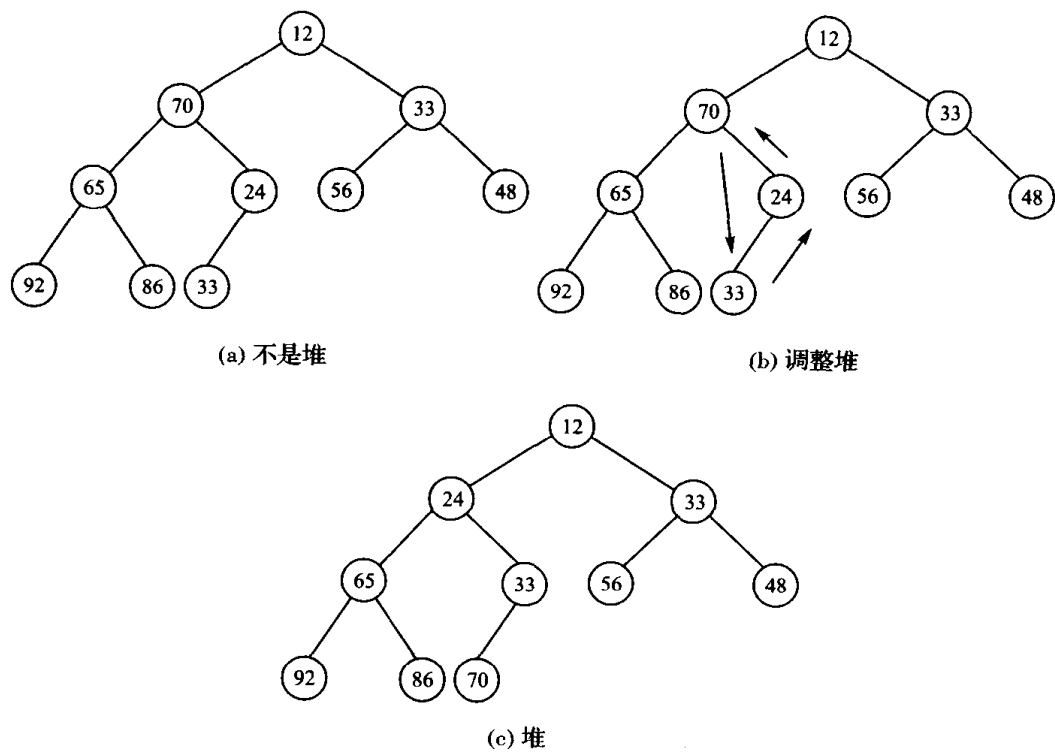


图8.1 堆调整过程

(3)将序列(103,97,56,38,66,23,42,12,30,52,6,20)构造表如下:

序号	1	2	3	4	5	6	7	8	9	10	11	12
结点值	103	97	56	38	66	23	42	12	30	52	6	20
左、右儿子	97,56	38,66	23,42	12,30	52,6	20						

观察表中序号1到6的结点,发现父结点的值均大于其左右儿子的值,故该序列是大顶堆。

(4)将序列(5,56,20,23,40,38,29,61,35,76,28,100)构造成二叉树如图8.2所示:

观察图 8.2, 因为 $5 < 56$, 而 $56 > 23$, 所以不是堆。

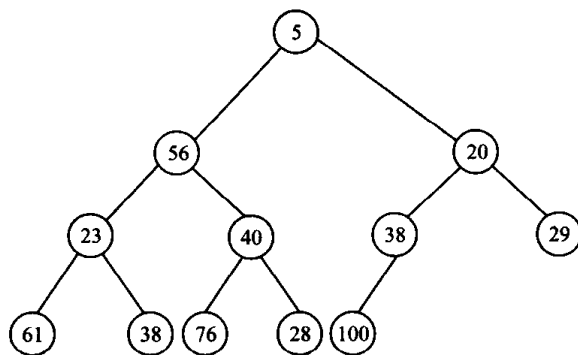


图 8.2 不是堆

对图 8.2 进行堆调整, 过程见图 8.3, 结果见图 8.4。

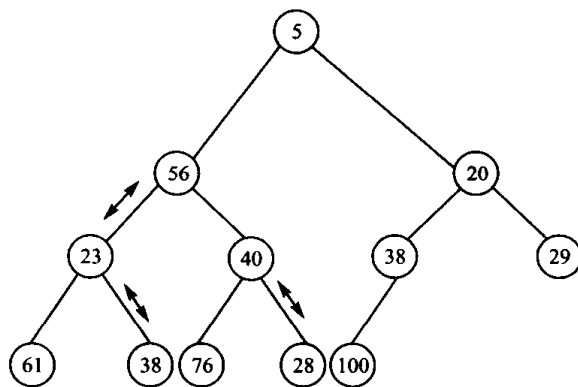


图 8.3 调整堆

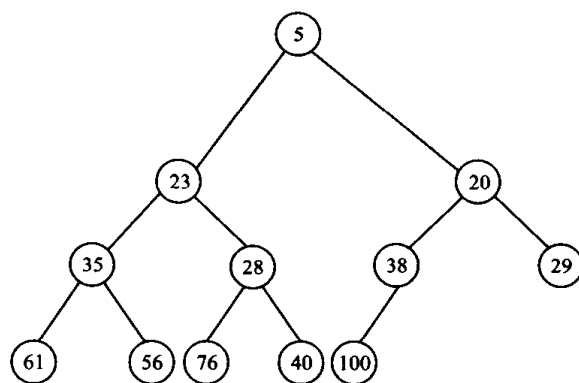


图 8.4 堆

4. 把第 3 题的第(2)小题按递增次序进行堆排序, 编写程序。

解 题目实际为对无序序列(12, 70, 33, 65, 24, 56, 48, 92, 86, 33)按递增次序进行堆排序, 编写程序。

算法由主函数和建堆、堆排序算法、输出四个函数组成。

程序如下:

```
#define LENGTH 10                /* 定义符号常量表示数组长度 */
void HeapAdjust(int r[],int i,int n) /* 把 r[i]...r[n]建成大顶堆的算法 */
{
    int j;
    r[0] = r[i];
    for(j = 2 * i; j <= n; j *= 2) /* 沿关键字较大的孩子结点向下搜索调整 */
    {
        if((j < n) && (r[j] <= r[j + 1]))
            ++j; /* 若右孩子大于左孩子,则 j 为右孩子下标 */
        if(r[0] > r[j])
            break;
        r[i] = r[j]; /* 将 r[j]调到父结点的位置 */
        i = j;
    }
    r[i] = r[0]; /* 将 r[0]插入到最终位置 */
}

void HeapSort(int r[],int n) /* 堆排序算法 */
{
    int i,x;
    for(i = LENGTH/2; i > 0; --i) /* 将待排序序列调整为大顶堆 */
        HeapAdjust(r,i,LENGTH); /* 调用建堆算法函数 */
    for(i = LENGTH; i > 1; --i)
    {
        x = r[1];
        r[1] = r[i];
        r[i] = x; /* 堆顶元素和最后的记录相交换 */
        HeapAdjust(r,1,i - 1); /* 调用建堆算法函数,进行堆调整 */
    }
    printdata(r,n);
}

void printdata(int r[],int n)
{
    int i;
    for(i = 1; i <= n; i++)
        printf("%d ",r[i]);
}

void main()
{
```

```
int r[LENGTH+1] = {0,12,70,33,65,24,56,48,92,86,33};  
/* 给数组赋值,0 元素值无意义 */  
  
printf("\nHeapSort output:");  
HeapSort(r,LENGTH);  
}
```

输出结果为:

HeapSort output: 12 24 33 33 48 56 65 70 86 92

5. 本章的排序算法中,哪些是稳定的? 哪些是不稳定的?

解 直接插入排序、冒泡排序、归并排序和基数排序是稳定的,而希尔排序、直接选择排序、快速排序和堆排序是不稳定的。

附录

基本题参考答案

第2章 线性表

2.4.1 选择题

1. D 2. A 3. A 4. B 5. D

2.4.2 填空题

1. $\text{head} \rightarrow \text{next} == \text{NULL}$

2. $\text{p} \rightarrow \text{next} == \text{head}$

3. $\text{p} \rightarrow \text{next}, \text{s}$

4. $\text{p} \rightarrow \text{next} = \text{q} \rightarrow \text{next}$

5. $\text{p} \rightarrow \text{next}, \text{s} \rightarrow \text{data}, \text{t}$

6. $\text{LA_rear} \rightarrow \text{next} = \text{LB_rear} \rightarrow \text{next} \rightarrow \text{next}, \text{LB_rear} \rightarrow \text{next} = \text{p}$

7. ① $i < n$ ② $j < m$

8. ① NULL ② $\text{p} \neq \text{NULL}$ ③ $\text{q} = \text{p}$

9. ① $\text{h} = \text{b}$ ② $\text{q} \rightarrow \text{next} = \text{b}$ ③ $\text{q} \rightarrow \text{next} = \text{p}$

10. ① $\text{q} \rightarrow \text{next} = \text{p}$ ② q ③ $\&\text{u}, \text{p} \rightarrow \text{val}$ ④ $\text{p} \neq \text{NULL}; \text{p} = \text{p} \rightarrow \text{next}$ ⑤ $\text{p} = \text{u}$

第3章 栈和队列

3.4.1 选择题

1. B A 2. C 3. C 4. C 5. A

3.4.2 填空题

1. 提示:利用栈“后进先出”的特点可得到,答案略。

2. (1) $\text{ABC} + * \text{D} + \text{E} * \text{FG} * -$

(2) $\text{ABD} - * \text{HDE} + / \text{SN} / \text{T} * -$

(3) $\text{AC} - \text{BD} + * \text{EF} - \text{GH} + / +$

3. 略

4. $\text{s} \rightarrow \text{top} == \text{MAXSIZE} - 1$ $\text{s} \rightarrow \text{data}[\text{s} \rightarrow \text{top}]$

5. $\text{s} == \text{NULL}$ $\text{p} \rightarrow \text{next}$

6. $\text{cq} \rightarrow \text{front} == \text{cq} \rightarrow \text{rear}$ $(\text{cq} \rightarrow \text{front} + 1) \% \text{MAXSIZE}$

第4章 串

4.4.1 选择题

1. B 2. D 3. B

4.4.2 填空题

1. 顺序存储方式 链式存储方式

2. 零个字符的串, 零

3. 答案略

4. 答案略

5. $s \rightarrow \text{curlen} + t \rightarrow \text{curlen}$

6. $s \rightarrow \text{ch}[i]$

7. $s.\text{ch}[\text{pos} + i - 1]$

第5章 树

5.4.1 选择题

1. C 2. D 3. C 4. C 5. B 6. C 7. C 8. D

5.4.2 填空题

1. 2^{h-1} , $2^h - 1$

2. a b d c e g f, d b a e g c f, d b g e f c a

3. 略

4. 73

5. 501

6. $T \rightarrow \text{lchild}, i - 1$ $T \rightarrow \text{rchild}, i - 1$

第6章 图

6.4.1 选择题

1. A 2. ①A ②A 3. (1)C (2)A (3)~(5)略 4. A 5. D 6. A 7. C
8. B 9. C 10. A 11. A 12. C 13. D 14. ①A ②C

6.4.2 填空题

1. 邻接矩阵 邻接表

2. $2(n-1)$

3. $n-1$

4. (1) V_1, V_2, V_3, V_4 V_1, V_3, V_2, V_4 V_2, V_1, V_3, V_4

(2) $\langle V_3, V_2 \rangle$

5. 下表是按迪杰斯特拉(Dijkstra)算法求出从顶点0到其他各顶点的最短路径,并给出了辅助数组中值的变化过程。

终点	从 V_0 到各终点的 dist 值和最短路径				
V_1	20 $\langle V_0, V_1 \rangle$	19 $\langle V_0, V_2, V_1 \rangle$			
V_2	15 $\langle V_0, V_2 \rangle$				
V_3				29 $\langle V_0, V_2, V_5, V_3 \rangle$	
V_4			29 $\langle V_0, V_2, V_1, V_4 \rangle$	29 $\langle V_0, V_2, V_1, V_4 \rangle$	29 $\langle V_0, V_2, V_1, V_4 \rangle$
V_5		25 $\langle V_0, V_2, V_5 \rangle$	25 $\langle V_0, V_2, V_5 \rangle$		
W	$W = V_2$	$W = V_1$	$W = V_5$	$W = V_3$	$W = V_4$

6. ① i ② $g \rightarrow \text{arc}[i][j]$

7. DFS(g, v)

第 7 章 查找

7.4.1 选择题

1. C 2. D 3. B 4. D 5. C 6. D

7.4.2 填空题

1. $\log_2(n+1) - 1$

2. 15 3. 11

4. 中序

5. 1, 2, 4, 8, 5, 3.7

6. 1.75

7. ① return(t)

② return(bstsearch($t \rightarrow \text{lchild}, k$))

③ return(bstsearch($t \rightarrow \text{rchild}, k$))

第 8 章 排序

8.4.1 选择题

1. C 2. C 3. D 4. C 5. B 6. B

8.4.2 填空题

1. 快速排序 归并排序

2. $n-1$ 0 $n(n-1)/2$

3. 堆排序 快速排序

4. 11 1

5. 10, 15, 14, 18, 20, 36, 40, 21

6. ① $i+1$ ② $\text{exchang} = \text{True}$ ③ $i = i+1$ ④ $\text{exchang} == \text{True}$

参考文献

- 1 严蔚敏,吴伟民. 数据结构(C语言版). 北京:清华大学出版社,1997
- 2 陈雁. 数据结构. 北京:高等教育出版社,2001
- 3 李春葆. 数据结构习题与解析(C语言版). 北京:清华大学出版社,2002
- 4 苏光奎. 李春葆编著 数据结构导学. 北京:清华大学出版社,2002
- 5 孙巧萍. 数据结构实训教程. 北京:科学出版社,2003
- 6 张海藩. 软件工程导论. 北京:清华大学出版社,2003
- 7 苏仕华等. 数据结构课程设计. 北京:机械工业出版社,2005
- 8 安训国,刘俞. 数据结构(第三版). 大连:大连理工大学出版社,2006