21 世纪高等学校电子信息类专业规划教材

数据结构实验教程

高晓兵　张凤琴　高晓军　万　能　编著

清 华 大 学 出 版 社
北京交通大学出版社
·北京·

内 容 简 介

《数据结构实验教程》是为了让学生能够尽快地掌握数据结构中的各种算法而编写的。本教材所写的算法具有程序结构清晰、可读性强、符合软件工程的规范要求等特点，所有的程序均在 VC 调试环境下运行通过，如果要运行程序，则仅需要编译一下便可。如果需要在 TURBO C 环境下运行，则仅需要将“//”注释修改一下便可。本书在数据结构的每个知识点上均给出了多个实验项目，且在每个实验项目中包括实验项目、任务分析、程序构思、源程序、测试数据、注意事项及思考问题等。在最后一章中给出了两个实际问题，着重分析了解决的思路、模块划分、重点难点等。本书共分 9 章，包括线性表、栈与队列、串、数组、树和二叉树、图、查找、排序和文件。

本书是清华大学出版社和北京交通大学出版社出版的《数据结构》教材（张凤琴主编）的配套实验教材，也可作为其他数据结构的实验教材及软件水平考试、计算机等级考试的上机指导、程序员编写算法的参考书。

图书在版编目（CIP）数据

数据结构实验教程/高晓兵等编著. —北京：清华大学出版社；北京交通大学出版社，2006.5
（21 世纪高等学校电子信息类专业规划教材）
ISBN 7 - 81082 - 782 - 0

Ⅰ. 数… Ⅱ. 高… Ⅲ. 数据结构 - 实验 - 高等学校 - 教材 Ⅳ. TP311.12 - 33

中国版本图书馆 CIP 数据核字（2006）第 048582 号

责任编辑：杨伟　　特邀编辑：吴炳林
出 版 者：清 华 大 学 出 版 社　　邮编：100084　　电话：010 - 62776969
　　北京交通大学出版社　　邮编：100044　　电话：010 - 51686414
印 刷 者：北京鑫海金澳胶印有限公司
发 行 者：新华书店总店北京发行所
开　　本：185 × 260　　印张：11.75　　字数：277 千字
版　　次：2006 年 6 月第 1 版　　2006 年 6 月第 1 次印刷
书　　号：ISBN 7 - 81082 - 782 - 0/TP · 276
印　　数：1 - 4 000 册　　定价：18.00 元

目　　录

第 1 章　线　性　表

本章的典型实验主要练习线性表的顺序、链式存储的基本操作和初步应用，分为线性表的顺序存储实验、线性表的单链表实验和线性表应用实验三类，共 5 个实验内容。

1.1　知识点概述

线性表是最基本最常用的数据结构，它有且仅有一个开始结点，该结点没有前驱且仅有一个后继；有且仅有一个终端结点，该结点没有后继且仅有一个前驱；其他所有结点都是内部结点，并且都有一个前驱和一个后继。一个线性表中的数据元素应具有相同的描述性质，即属于同一个数据对象。

线性表基本操作有初始化、判断表空、求表长、插入元素、删除元素等。

在实际应用中，必须将线性表中的数据存放在计算机中。常用的存储方式有两种：顺序存储和链式存储。顺序存储是用一块连续的地址存储单元依次存放线性表中的数据元素，使得数据元素的逻辑上的相邻关系与物理上的相邻关系一致。链式存储是用任意的存储单元存放线性表中的数据元素，这些单元可以分散在内存中的任一位置上，使得在表示数据结构时不能再用物理上相邻关系，而必须在存储每个元素的同时，也要存储元素之间的逻辑关系。这种存储单元只有在需要的时候才申请，不用事先分配；同样在使用完毕，应立即释放。

顺序存储的线性表又称顺序表，其特点是可以方便地存取表中任一元素；无需为表示元素间的逻辑关系而增加额外的存储空间；插入和删除运算时需移动大量元素，其效率较低；在长度变化较大的线性表预分配空间时，必须按最大空间分配，存储空间得不到充分利用；一旦分配好空间，表的容量难以扩充。

链式存储的线性表又称链表，其特点是查找表中任一元素时需从头结点的指针域开始逐步向后（前）查找；每个结点需要增加指针域；动态分配存储空间，存储空间得到了充分利用；容易插入和删除数据元素。

1.2　线性表的顺序存储实验

【实验 1.1】　顺序表基本操作的设计与实现

实现顺序表的基本操作，包括顺序表的建立、查找、求长度、查找前驱、插入、删除、输出等函数。

第 1 步：任务分析。

完成顺序表的建立、查找、求长度、查找前驱、插入、删除、输出等函数功能，有助于更好的理解顺序表的概念和用法。上述这些函数都是线性表的基本操作，根据这些基本操作，可构成其他更复杂的操作。

第 2 步：程序构思。

顺序表的建立，首先要建立一块连续存储的空间。在 Turbo C 或 C++ 中由于数组采用顺序存储，所以使用数组这种数据结构来描述线性表的顺序存储结构。顺序表的存储结构描述包含存储空间和实际长度两个重要的数据，这两个重要数据在实际实现时可以灵活变化，参见思考描述。

第 3 步：源程序。

```
1        //调试环境： Visual C++ 6.0
2
3        //-----库文件和预设定义
4        #include <stdlib.h>
5        #include <stdio.h>
6
7        #define OK              1
8        #define ERROR           0
9        #define  OVERFLOW       -1
10
11       #define List_INIT_SPACE     100     //存储空间初始分配量
12       #define List_INC_SPACE      10      //存储空间分配增量
13
14       typedef int ElemType;               //指定顺序表中数据类型
15
16       typedef  struct
17       {
18          ElemType *elem;                  //存储空间基址
19          int     length;                  //当前长度
20          int     listsize; //当前分配的存储容量(以 sizeof(ElemType)为单位)
21       }Sq_List;
22
23       /*****------------------------------    *****/
24       // 函数名：   Sq_ListInit (Sq_List &L)
25       // 参数：     (传入) SqList L,顺序表结构体 L,存储线性表相关信息(& 相当于传
                      入 L 的地址)
26       // 返回值：   int 型,返回 1 表示创建成功,0 表示失败
27       // 功能：     初始化一个空顺序表
28       /*****------------------------------    *****/
29       int Sq_ListInit ( Sq_List &L)
30       {
31           //在内存中分配空间
32           L.elem = ( ElemType *) malloc (List_INIT_SPACE *sizeof ( Elem-
Type));
33           if (! L.elem)  exit (OVERFLOW);     //存储分配失败
34
```

```
35          // 构造一个空的线性表 L   //
36          L.length =0;
37          L.listsize =List_INC_SPACE;          //初始存储容量
38          return  OK;
39      }//函数 Sq_ListInit 结束
40
41      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
42      // 函数名：   LocateElem(Sq_List L,ElemType e)
43      // 参数：     (传入) Sq_List L,顺序表
44      //            (传入) ElemType e,定位元素
45      // 返回值：   int 型,返回定位位置,0 表示失败
46      // 功能：     在顺序表中定位元素
47      /* * * * - - - - - - - - - - - - - - - - - - - -   * * * * */
48      int LocateElem(Sq_List L,int e)
49      {
50         int i =1;
51         //定义线性表指针
52         int *p =L.elem;
53         //查找元素 e
54         while(i <=L.length&& *p ++!=e)
55            ++i;
56
57         if(i <=L.length)
58            return i;
59         else
60            return ERROR;
61      }
62      /*求顺序表的长度*/
63      int GetListLength(Sq_List L)
64      {
65          return L.length;
66      }
67
68      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
69      // 函数名：   Sq_Listlnsert ( Sq_List &L, int i, ElemType e )
70      // 参数：     (传入) Sq_List &L     顺序表
71      //            (传入) int i 插入位置
72      //            (传入) ElemType e  插入元素
73      // 返回值：   1 表示成功,0 表示操作失败
74      // 功能：     在顺序表 L 中的第 i 个位置前插入新元素 e
75      // 备注：     i 的合法取值为 1≤i≤线性表长度 +1
76      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
77      int Sq_Listlnsert ( Sq_List &L, int i, ElemType e )
```

```
78      {
79      //判断位置是否合法
80            if ( i < 1 ||i > L.length +1)
81            {
82               printf("i 的值不合法!\n");
83              return 0;
84          }
85
86          //超出空间进行再分配
87          if ( L.length >=L.listsize )
88          {
89              int * newspace;
90      newspace = ( ElemType * ) realloc ( L.elem, (L.listsize +List_INC_
        SPACE) * sizeof ( ElemType ));
91              if (! newspace)  exit (OVERFLOW);          //存储分配失败
92              L.elem = newspace;                         //新基址
93              L.listsize + =List_INC_SPACE;              //增加存储容量
94          }
95
96          int *p, *q;              //定义指向线性表位置 i 和尾的指针
97          q = &(L.elem[i -1]);     //q 指针指向插入位置 i 的前一个
98          for(p = &(L.elem[L.length -1]);p >=q; --p)
99                * (p +1) = * p;    //插入元素之后的元素右移
100         *q = e;                  //把元素 e 放在位置 i 处
101         ++L.length;              //线性表长度增 1
102         return  OK;
103     }//函数 Sq_ListInsert 结束
104
105     /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
106     // 函数名：  Sq_ListDelete ( Sq_List &L, int i, ElemType &e )
107     // 参数：    (传入) Sq_List &L     顺序表
108     //           (传入) int i          删除位置
109     //           (传出) ElemType &e 删除元素
110     // 返回值：  1 表示成功,0 表示操作失败
111     // 功能：    在顺序线性表 L 中删除第 i 个元素,用 e 返回其值
112     // 备注：    i 的合法取值为 1≤i≤线性表长度
113     /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
114     int Sq_ListDelete ( Sq_List &L, int i, ElemType &e )
115     {
116         //判断位置是否合法
117         if ( i < 1 ||i > L.length +1)
118         {
119             printf("i 的值不合法!\n");
```

```
120              return 0;
121         }
122
123         int *p,*q;                              //定义指向线性表位置 i 和尾的
指针
124            p=& (L.elem[i-1]);                   //p 为被删除元素的位置
125            e=* p;                               //取删除元素的值
126            q=L.elem + L.length-1;               //q 指针指向线性表最后一个
元素
127            for (++p; p<=q; ++p)
128                   *(p-1)=*p;                    //被删除元素之后的元素左移
129            --L.length;                          //线性表长度减 1
130            return  OK;
131        }//函数 Sq_ListDelete 结束
132
133      //-----------测试程序-----------
134        void main()
135        {
136            Sq_List R;//实参定义
137            int flag;
138            //为了判断调用成功与否,可以检查 flag 的值,或者见插入操作的调用方法
139            //  if(flag==1)  printf("成功");
140            //  else  printf("失败");
141            flag=Sq_ListInit(R);
142
143            //调用方法
144            int i=1;
145            int elem;
146            printf("the length of current list is %d\n", GetListLength
(R));
147            printf("input the element:");
148            scanf("%d", &elem);
149            if( Sq_ListInsert(R,i,elem) )    printf("successed!\n");
150            printf("the length of current list is %d\n", GetListLength
(R));
151            //Sq_ListInsert(R,i,elem);    //或者直接调用
152        }
```

第 4 步：测试数据。

运行结果：

```
the length of current list is 0
input the element:45
successed!
```

```
the length of current list is 1
```

注意：

从这个实验开始，程序使用了 malloc **函数，此函数向系统请求分配内存空间，其参数为申请的内存空间的大小，通常由求字节长度的** sizeof **函数计算得到空间大小。申请空间成功时返回指向该空间起始地址的指针（由于** malloc **默认返回** void **类型指针，所以要强制类型转换），否则返回** NULL **指针。使用此函数，必须包含对应的头文件** stdlib. h**。**

思考题：

采用直观的数组定义来实现顺序表，规定数组中下标为 0 **的元素存储顺序表的实际长度，这时顺序表存储的元素从下标** 1 **开始。具体定义举例如下：**

```
#define MAX 100
int A[MAX +1];        /* 存储空间初始分配量为 100，下标访问从 1 - 100 */
A[0] =10;             /* 顺序表当前长度为 10 */
```

试根据上述定义，完成顺序表的基本操作。

【实验 1.2**】　顺序表基本操作应用实验**

应用题目：

⑴ 有一个表元素按值递增排列的顺序表，编写一个函数实现删除顺序表中多余的值相同元素；

⑵ 有一个顺序表，编写一个在顺序表中查找最大和最小值元素的函数，并分析其时间复杂度 T (n) **。**

第 1 **步：任务分析。**

这两个题目涉及顺序表元素的删除、访问等，重点在于在顺序表的基础上领悟程序算法设计。

第 2 **步：程序构思。**

⑴ 由于表中元素按照其值非递减排列，值相同的元素必为相邻的元素，故程序实现时，依次比较相邻的两个元素，若值相等，删除其中一个，否则继续向后查找。

⑵ 依次扫描一遍表中所有元素，比较查找最大和最小值。

第 3 **步：源程序。**

```
1        /* * * * - - - - - - - - - - - - - - - - - - - - - -    * * * * */
2        // 函数名：   delete(List A, int n)
3        // 参数：     List A 顺序表, int n 顺序表长度
4        // 返回值：   无
5        // 功能：     删除顺序表中多余的值相同元素
6        // 备注：     顺序表用一维数组实现，故 List A 可以理解为 int A[N]
7        /* * * * - - - - - - - - - - - - - - - - - - - - - -    * * * * */
8        void delete(List A, int n)
9        {
10           int i =0, j;
11           while(i <=n -2)
```

```
12              if(A[i]!=A[i+1])i++;     /*元素值不等,继续查找*/
13              else;
14              {
15              for(j=(i+2);j<=n;j++)A[j-1]=A[j];  /*删除第i+1个元
素*/
```

```
16                  n--;    /*表长度减1*/
17              }/* end if */
18      }
19
20      /*****------------------------*****/
21      //函数名：  maxmin(List A, int n)
22      //参数：    List A顺序表, int n顺序表长度
23      //返回值：  无
24      //功能：    查找最大和最小值元素
25      //备注：    顺序表用一维数组实现,故List A可以理解为int A[N]
26      /*****------------------------*****/
27      void maxmin(List A, int n)
28      {
29          int max,min,i;
30          max=A[0];min=A[0];
31          for(i=1;i<n;i++)
32          {
33              if(A[i]>max)    max=A[i];
34              else if(A[i]<min)    min=A[i];
35          }
36          printf("max=%d, min=%d\n", max, min);
37      }
```

第4步：测试。

请读者自行完成main()函数。

思考题：

第二个题目函数关于时间复杂度T(n)的讨论，请从下面两种特殊情况分别考虑程序的比较次数。

⑴ 最坏情况(顺序表递减排列)下的比较次数；

⑵ 最好情况(顺序表递增排列)下的比较次数。

1.3　线性表的链表实验

【实验1.3】　单链表的的设计与实现

实现单链表的基本操作，包括链表的建立与释放、查找、求长度、查找后继、插入、删除、输出等函数。

第1步：任务分析。

单链表的建立、查找、求长度、查找前驱、插入、删除、输出等是链表的基本操作，正确的实现它们，需要首先从存储形式上理解链表。单链表的存储结构描述包含数据域(存储数据元素值)和指针域(存储地址，指向链表下一个节点的指针)两部分，这两部分构成一个结点，具体用C语言的结构体来描述。链表的操作主要是要清楚指针的变化情况，明确头指

针、当前指针的指向，为了更好的理解和掌握，可以以图示的形式进行。请参见教材相关部分。

第 2 步：程序构思。

1. 单链表的建立

⑴ 先建立头节点 head，将头节点的指针域置为空。

⑵ 新建一个节点 p，把此新节点链接到单链表的尾端（p - > next 设为空）或者始端。

① p - > next 指向头节点指向的下一个节点 head - > next。

② head - > next 指向 p。

2. 单链表的插入

⑴ 新建一个节点 p，指定插入位置。

⑵ 从单链表头开始查找节点位置。

① 找到该位置，则 p - > next 指向当前位置的下一个节点，当前位置节点指向新建节点 p。

② 没有找到该位置，插入操作失败。

3. 单链表的删除

⑴ 指定删除位置。

⑵ 从单链表头开始查找节点位置。

① 找到该位置，则删除该位置的节点。

② 没有找到该位置，删除操作失败。

第 3 步：源程序。

```
1        //调试环境：Visual C++ 6.0
2
3        //-----库文件和预设定义
4        #include <stdlib.h>
5        #include <stdio.h>
6        #define NULL 0
7
8        typedef int ElemType;//指定单链表中数据类型
9
10       //单链表存储结构定义
11       typedef struct LNode
12       {    ElemType data;        //数据域
13            struct LNode *next;  //指针域
14       }LNode, *LinkList;
15
16       /*****------------------------    *****/
17       // 单链表建立方法一（函数用返回值得到表头指针）
18       // 函数名：    CreateOne(int n)
19       // 参数：      （传入）int n，传入线性表结点数量
20       // 作用：      建立一个空线性表
```

```
21        // 返回值:    LNode * 型返回结构体指针，即得到建立的线性表头指针
22        /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
23        LNode * CreateOne(int n)
24        {
25          int i;
26          LNode * head; /* 定义头结点指针 */
27          LNode *p; /* 定义新结点指针 */
28
29          /* 建立带头结点的线性链表 */
30          head = (LNode *)malloc(sizeof(LNode));
31          head -> next = NULL;
32
33          printf("Please input the data for LinkList Nodes:\n");
34          for(i = n;i > 0; --i)
35          {
36              p = (LNode *)malloc(sizeof(LNode)); /* 为新结点申请空间,即创建一
                新结点 */
37              scanf("%d",&p -> data); /* 新结点赋值 */
38              /* 新结点插入到表头 */
39              p -> next = head -> next;
40              head -> next = p;
41          }
42          return head; /* 返回头结点指针,即可以得到单链表的地址 */
43        }
44
45        /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
46        // 单链表建立方法二(函数无返回值)
47        // 函数名:    CreateTwo(LNode * head, int n)
48        // 参数:      (传入)LNode * head 传入一个链表指针
49        //            (传入)int n,传入线性表结点数量
50        // 作用:      建立一个空线性表
51        // 返回值:    无
52        /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
53        void CreateTwo(LinkList &head, int n)
54        {
55          int i;
56          LNode *p; /* 定义新结点指针 */
57
58          /* 建立带头结点的线性链表 */
59          head = (LinkList)malloc(sizeof(LNode));
60          head -> next = NULL;
61
62          printf("Please input the data for LinkList Nodes:\n");
```

```
63          for(i=n;i>0;--i)
64          {
65              p=(LNode*)malloc(sizeof(LNode));/*为新结点申请空间,即创建一
                新结点*/
66              scanf("%d",&p->data); /*新结点赋值*/
67              /*新结点插入到表头*/
68              p->next=head->next;
69              head->next=p;
70          }
71      }
72
73      /*****------------------------    *****/
74      //函数名:    InsertNode(LNode* L,int i, ElemType e)
75      //参数:      (传入)LNode* L,传入线性表头指针 L
76      //           (传入)int i 插入位置
77      //           (传入)ElemType e 插入元素
78      //作用:      线性表中插入一个元素
79      //返回值:    int 型,返回 1 表示操作成功,0 表示失败
80      /*****------------------------    *****/
81      int InsertNode(LNode* L,int i,int e)
82      {
83          LNode *p=L;/*定义一个指向第一个结点的指针*/
84          int j=0;
85
86            /*顺指针向后查找,直到 p 指向第 i 个结点*/
87          while(p&&j<i-1)
88          {
89              p=p->next;
90               ++j;
91            }
92            /*插入位置合法性判断*/
93          if(!p||j>i-1)
94          {
95                printf("Error! The location is illegal!");
96              return 0;
97          }
98
99          LNode *s;
100         s=(LNode *)malloc(sizeof(LNode));/*建立新结点*/
101         s->data=e;/*新结点赋值*/
102
103         /*插入结点*/
104         s->next=p->next;
```

```
105          p->next=s;
106
107          return 1;
108      }
109
110      /*****------------------------*****/
111      // 函数名:    DeleteNode(LinkList &L, int i, ElemType &e)
112      // 参数:      (传入)LinkList &L,线性表头指针 L 的地址
113      //            (传入)int i 删除位置
114      //            (传出)ElemType &e 存储删除结点元素的值
115      // 作用:      线性表中删除一个元素
116      // 返回值:    ElemType 型返回删除结点元素的值
117      /*****------------------------*****/
118      ElemType DeleteNode(LinkList &L,int i,int &e)
119      {
120         LNode *p;
121         p=L;/*定义一个指向第 i 个结点的指针 p*/
122         LNode*q;/*暂时存放待删除结点*/
123         int j=0;
124
125         /*顺指针向后查找,直到 p 指向第 i 个结点*/
126         while(p->next&&j<i-1)
127         {
128            p=p->next;
129            ++j;
130         }
131         /*删除位置合法性判断*/
132         if(!p||j>i-1)
133         {
134            printf(" element is not exist !");
135            return(0);
136         }
137         q=p->next;
138         p->next=q->next; //删除第 i 个结点
139         e=q->data;
140         free(q);
141         return (e); /*返回删除结点元素的值*/
142      }
143
144      /*****------------------------*****/
145      // 函数名:    DisplayList(LinkList &L, int i, ElemType &e)
146      // 参数:      (传入)LinkList &L,线性表头指针 L 的地址
147      //            (传入)int i 显示位置
```

```
148     //              (传出)ElemType &e 存储显示结点元素的值
149     // 作用:        显示线性表中所有元素
150     // 返回值:      无
151     /* * * * - - - - - - - - - - - - - - - - - - - - - -    * * * * */
152     void DisplayList(LinkList &L)
153     {
154         LNode *p;
155         p=L->next;/*定义一个指向第 i 个结点的指针 p*/
156         while(p!=NULL)
157         {
158             printf("%d ", p->data);
159             p=p->next;
160         }
161         printf("\n");
162     }
163     //- - - - - - - - - -测试程序- - - - - - - - - -
164     void main()
165     {
166        LNode* L1;
167        int NodeNum;
168        printf("Please input the Init LinkNode Number:\n");
169        scanf("%d", &NodeNum);
170        L1=CreateOne(NodeNum);
171        //LNode *L2;
172        //CreateTwo(L2, NodeNum);//也可以这样调用生成 L2 链表
173
174        //输出当前单链表内容
175        printf("the current L1 is:");
176        DisplayList(L1);
177
178        //调用
179        int result;/*为了判断调用成功与否,可以定义 result 以备检查*/
180        result=InsertNode(L1, 2,88);
181        if(result)
182            printf("success to insert!\n");
183
184        //输出当前单链表内容
185        printf("the current L1 is:");
186        DisplayList(L1);
187     }
```

第 4 步：测试数据

运行结果：

```
Please input the Init LinkNode Number:
3
Please input the data for LinkList Nodes:
23 42 55
the current L1 is: 55 42 23
success to insert!
the current L1 is:55 88 88 42 23
```

1.4　线性表应用实验

【实验 1.4】　集合的并交差运算

完成实现集合的并交差等运算。

第 1 步：任务分析。

结合线性表的基本操作，完成两个集合（线性表）元素的并交差等运算。实践的重点在于线性表存储形式的深刻理解和线性表基本操作的熟练运用等。

第 2 步：程序构思。

集合并运算的实现步骤如下。

⑴ 设置初始状态。当线性表 La 和 Lb 非空时，指针 pa 和 pb 分别指向线性表 La 和 Lb 中的第 1 个元素；为合并后的新线性表 Lc 分配内存空间，使用指针 pc 指向新线性表 Lc。

⑵ 在 C 中插入新结点，并修改指针。分两种情况：若 ∗pa≤∗pb，则将 pa 所指向的值赋给 pc 所指向的，同时使得 pc 和 pa 向后移动一个元素；若 ∗pa ≥∗pb，则将 pb 所指向的值赋给 pc 所指向的，同时使得 pc 和 pb 向后移动一个元素。

⑶ 插入剩余段。

⑷ 释放单链表 Lb 的头结点。

第 3 步：源程序。

```
1        //调试环境：Visual C++6.0
2
3        //-----库文件和预设定义
4        #include <stdlib.h>
5        #include <stdio.h>
6
7        #define OK          1
8        #define ERROR       0
9        #define OVERFLOW    -1
10
11       #define List_INIT_SPACE    100    //存储空间初始分配量
12       #define List_INC_SPACE     10     //存储空间分配增量
13
```

```
typedef  struct
{
   int   *elem;    //存储空间基址
   int  length;    //当前长度
   int listsize; //当前分配的存储容量 (以 sizeof(ElemType)为单位)
}Sq_List;

/*****------------------------*****/
// 函数名：  Sq_ListInit (Sq_List &L)
// 参数：    (传入) SqList L,顺序表结构体 L,存储线性表相关信息
// 返回值：  int 型,返回 1 表示创建成功,0 表示失败
// 功能：    初始化一个空顺序表
/*****------------------------*****/
int Sq_ListInit (Sq_List &L)
{
   //在内存中分配空间
   L.elem = (int *) malloc (List_INIT_SPACE *sizeof (int));
   if (!L.elem)  exit (OVERFLOW);     //存储分配失败

   // 构造一个空的线性表 L  //
   L.length =0;
   L.listsize =List_INC_SPACE;         //初始存储容量
   return  OK;
}//函数 Sq_ListInit 结束

/*****------------------------*****/
// 函数名：  Sq_Listlnsert (Sq_List &L, int i, int e)
// 参数：    (传入) Sq_List &L 顺序表
//           (传入) int i 插入位置
//           (传入) ElemType e 插入元素
// 返回值：  1 表示成功,0 表示操作失败
// 功能：    在顺序表 L 中的第 i 个位置前插入新元素 e
// 备注：    i 的合法取值为 1≤i≤线性表长度 +1
/*****------------------------*****/
int Sq_Listlnsert (Sq_List &L, int i, int e)
{
      //判断位置是否合法
      if ( i < 1 ||i > L.length +1)
      {
       printf("i 的值不合法!\n");
       return 0;
```

```
55          }
56
57          //超出空间进行再分配
58          if ( L.length >=L.listsize )
59          {
60              int * newspace;
61              newspace = (int *) realloc ( L.elem, (L.listsize + List_INC_
                SPACE) * sizeof (int));
62              if (! newspace)  exit (OVERFLOW);    //存储分配失败
63              L.elem = newspace;                   //新基址
64              L.listsize + = List_INC_SPACE;       //增加存储容量
65          }
66
67          int *p, *q;            //定义指向线性表位置 i 和尾的指针
68          q = &(L.elem[i -1]);   //q 指针指向插入位置 i 的前一个
69          for(p = &(L.elem[L.length -1]);p >= q; --p)
70          *(p +1) = *p;          //插入元素之后的元素右移
71          *q = e;                //把元素 e 放在位置 i 处
72          ++L.length;            //线性表长度增 1
73          return  OK;
74      }
75
76      /* 功能：  显示顺序线性表 L 中的各个元素值  */
77      void PrintList(Sq_List L)
78      {
79          for(int i =0;i <L.length;i ++)
80              printf("%d ", *L.elem ++);
81          printf("\n");
82      }
83      /* * * * - - - - - - - - - - - - - - - - - - - - - -  * * * * */
84      // 函数名：  Sq_ListMerge(Sq_List La, Sq_List Lb, Sq_List &Lc)
85      // 参数：    (传入) Sq_List La    顺序表 La
86      //           (传入) Sq_List La    顺序表 Lb
87      //           (传出) Sq_List Lc    合并后的顺序表 Lc
88      // 功能：    合并两顺序表 (顺序表数据元素按值非递减有序排列)
89      /* * * * - - - - - - - - - - - - - - - - - - - - - -  * * * * */
90      void Sq_ListMerge(Sq_List La, Sq_List Lb, Sq_List &Lc)
91      {
92          int *pa, *pb, *pc;       //定义指向顺序表头部的指针
93          int *qa, *qb;            //定义指向顺序表尾部的指针
94          pa = La.elem;
```

```
95          pb=Lb.elem;
96          qa=La.elem + La.length - 1;
97          qb=Lb.elem + Lb.length - 1;
98
99          //分配合并后新顺序表的空间并定义其参数
100         Lc.listsize=Lc.length=La.length + Lb.length;
101         pc=Lc.elem=(int *)malloc(Lc.listsize*sizeof(int));
102         if(!Lc.elem)exit(OVERFLOW);    //存储分配失败
103
104         while (pa <=qa && pb <=qb)    //归并
105         {
106             if (*pa <= *pb) *pc++ = *pa++;
107             else *pc++ = *pb++;
108         }
109         while (pa <=qa) *pc++ = *pa++;    //插入 La 的剩余元素
110         while (pb <=qb) *pc++ = *pb++;    //插入 Lb 的剩余元素
111     }
112
113     //- - - - - - - - - - -测试程序- - - - - - - - - - - -
114     void main()
115     {
116         Sq_List La, Lb, Lc;
117         //建立顺序表 La
118         if (Sq_ListInit(La))
119         {
120             Sq_ListInsert(La, 1, 3);
121             Sq_ListInsert(La, 2, 5);
122             Sq_ListInsert(La, 3, 8);
123             Sq_ListInsert(La, 4, 11);
124             PrintList(La);
125         }
126         else printf("create failed!\n");
127         //建立顺序表 Lb
128         if (Sq_ListInit(Lb))
129         {
130             Sq_ListInsert(Lb, 1, 2);
131             Sq_ListInsert(Lb, 2, 6);
132             Sq_ListInsert(Lb, 3, 8);
133             Sq_ListInsert(Lb, 4, 9);
134             Sq_ListInsert(Lb, 5, 11);
135             Sq_ListInsert(Lb, 6, 15);
```

```
136             PrintList(Lb);
137         }
138         else printf("create failed!\n");
139
140         Sq_ListMerge(La, Lb, Lc);
141         PrintList(Lc);
142     }
```

第 4 步：测试数据。

运行结果：

```
3 5 8 11
2 6 8 9 11 15
2 3 5 6 8 8 9 11 11 15
```

思考题：

1. 请参照集合并的实现函数，自行实现集合交、差的运算。
2. 请使用单链表的存储形式实现集合的并交差运算。

【实验 1.5】　两个一元多项式相加实验

用线性表的存储形式实现两个一元多项式相加。

第 1 步：任务分析。

用线性表的存储形式实现两个一元多项式相加，首先要考虑多项式的存储形式。若只对多项式进行“求值”等不改变多项式的系数和指数的运算，则采用类似于顺序表的顺序存储结构即可，否则应采用链式存储表示。接下来要了解其运算规则，根据一元多项式相加的运算规则：对于两个一元多项式中所有指数相同的项，对应指数相加，若其和不为零，则构成“和多项式”中的一项；对于两个一元多项式中所有指数不相同的项，则分别复抄到“和多项式”中去。根据以上两点，本实验采用链式存储形式实现两个一元多项式相加运算。

第 2 步：程序构思。

两个一元多项式相加程序的构思，实际首先是通过程序对其运算规则的描述。对其规则描述如下。

假设指针 qa 和 qb 分别指向多项式 A 和多项式 B 中当前进行比较的某个结点，则比较两个结点中的指数项，有下列三种情况：

⑴ 指针 qa 所指结点的指数值小于指针 qb 所指结点的指数值，则应摘取 qa 指针所指结点插入到“和多项式”链表中去；

⑵ 指针 qa 所指结点的指数值大于指针 qb 所指结点的指数值，则应摘取指针 qb 所指结点插入到“和多项式”链表中去；

⑶ 指针 qa 所指结点的指数值等于指针 qb 所指结点的指数值，则将两个结点中的系数相加，若和数不为零，则修改 qa 所指结点的系数值，同时释放 qb 所指结点，反之，从多项式 A 的链表中删除相应结点，并释放指针 qa 和 qb 所指结点。

一元多项式相加过程如图 1-1 所示。

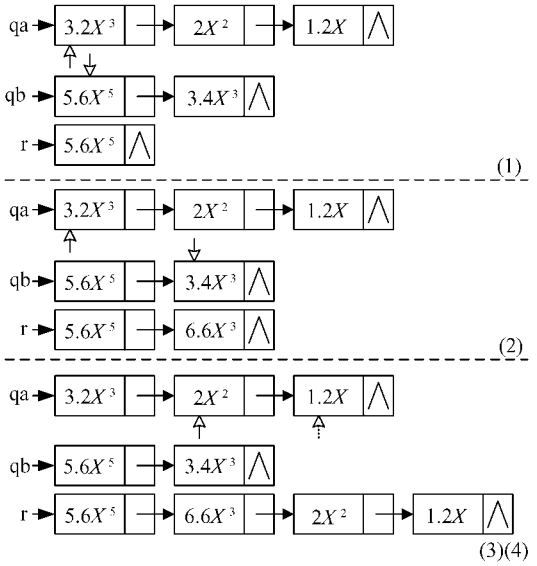


图 1-1　一元多项式相加过程示意图

第 3 步：源程序。

```
//调试环境：Visual C++6.0

//-----库文件和预设定义
#include <stdlib.h>
#include <stdio.h>

typedef  struct
{
     float coef;        //系数
     int expn;          //指数
}term;
typedef  struct LNode
{
     term  data;        //多项式的值
     struct LNode *next;
}LNode, *LinkList;

typedef LinkList polynomial;    //用带头节点的链表表示多项式

int cmp(term a, term b)
{
     int flag;
     if (a.expn < b.expn)    flag = -1;
```

```
24            else if (a.expn = b.expn)    flag = 0;
25            else    flag = 1;
26
27            return flag;
28        }
29
30        /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
31        // 函数名：  CreatPolyn(polynomial &P, int m)
32        // 参数：    (传入) int m,多项式的项数
33        //           (传出) polynomial P,多项式链表
34        // 功能：    建立多项式链表
35        /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
36        void CreatPolyn(polynomial &P, int m)
37        {
38            polynomial r, s;
39            term para;
40            P = (LNode *)malloc(sizeof(LNode));
41            r = P;
42            for(int i = 0; i < m; i ++)
43            {
44                s = (LNode *)malloc(sizeof(LNode));
45                printf("输入系数 c 和指数 e:");
46                scanf("%f %d", &para.coef, &para.expn);
47                s -> data.coef = para.coef;
48                s -> data.expn = para.expn;
49                r -> next = s;    /* 把 s 链接到 r 后面 */
50                r = s;
51            }
52            r -> next = NULL;
53        }
54
55        /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
56        // 函数名：  AddPolyn(polynomial &pa, polynomial &pb)
57        // 参数：    (传入) polynomial &pa,多项式 pa
58        //           (传入) polynomial &pb,多项式 pb
59        // 功能：    多项式相加
60        // 返回值：  两多项式相加得到的新多项式
61        /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
62        polynomial AddPolyn(polynomial &pa, polynomial &pb)
63        {
64            polynomial newp, p, q, s, r;
65            float sum;
66            p = pa -> next;
```

```
67          q=pb->next;
68          newp=(LNode *)malloc(sizeof(LNode));
69          r=newp;
70          while (p && q)    //pa 和 pb 均非空
71          {
72              switch(cmp(p->data, q->data))    //比较两多项式指数
73              {
74              case -1:    //多项式 pa 当前节点的指数值小
75                  s=(LNode *)malloc(sizeof(LNode));
76                  s->data.coef=p->data.coef;
77                  s->data.expn=p->data.expn;
78                  r->next=s;
79                  r=s;
80                  q=q->next;
81                  break;
82              case 0:    //两多项式指数值相等
83                  sum=p->data.coef + q->data.coef;
84                  if (sum!=0.0)
85                  {
86                      s=(LNode *)malloc(sizeof(LNode));
87                      s->data.coef=sum;
88                      s->data.expn=p->data.expn;
89                      r->next=s;
90                      r=s;
91                  }
92                  p=p->next;
93                  q=q->next;
94                  break;
95              case 1:    //多项式 pb 当前节点的指数值小
96                  s=(LNode *)malloc(sizeof(LNode));
97                  s->data.coef=q->data.coef;
98                  s->data.expn=q->data.expn;
99                  r->next=s;    //把 s 链接到新表 newp 中
100                 r=s;
101                 p=p->next;
102                 break;
103             }//switch
104         }//while
105
106         //链接 pa 中剩余节点
107         while (p)
108         {
109             s=(LNode *)malloc(sizeof(LNode));
```

```
110             s->data.coef=p->data.coef;
111             s->data.expn=p->data.expn;
112             r->next=s;    //把 s 链接到新表 newp 中
113             r=s;
114             p=p->next;
115         }
116         //链接 pb 中剩余节点
117         while (q)
118         {
119             s=(LNode *)malloc(sizeof(LNode));
120             s->data.coef=q->data.coef;
121             s->data.expn=q->data.expn;
122             r->next=s;
123             r=s;
124             q=q->next;
125         }
126         r->next=NULL;    //最后节点的置空,表示结束
127
128         return newp;
129     }
130
131     /*****------------------------    *****/
132     // 函数名:   PrintPolyn(polynomial p)
133     // 参数:     (传入) polynomial p,多项式链表
134     // 功能:     输出多项式链表
135     /*****------------------------    *****/
136     void PrintPolyn(polynomial p)
137     {
138         polynomial s;
139         s=p->next;
140         while (s)
141         {
142             //输出系数和指数
143             printf("%.2f(%d) ", s->data.coef,s->data.expn);
144             s=s->next;
145         }
146         printf("\n");
147     }
148     void main()
149     {
150         int m, n;
151         polynomial p, q;
152         printf("请输入多项式 pa 的项数:");
```

```
153          scanf("%d", &m);
154          CreatPolyn(p, m);
155          printf("请输入多项式 pb 的项数：");
156          scanf("%d", &n);
157          CreatPolyn(q, n);
158          PrintPolyn(p);
159          PrintPolyn(q);
160          PrintPolyn( AddPolyn(p, q) );
161      }
```

第 4 步：测试数据（以图 1-1 给出的数据作为测试数据）。

运行结果：

```
请输入多项式 pa 的项数：3
输入系数 c 和指数 e:3.2 3
输入系数 c 和指数 e:2 2
输入系数 c 和指数 e:1.2 1
请输入多项式 pb 的项数：2
输入系数 c 和指数 e:5.6 5
输入系数 c 和指数 e:3.4 3
3.2(3) 2(2) 1.2(1)
5.6(5) 3.4(3)
5.60(5) 6.60(3) 2.00(2) 1.20(1)
```

思考题：

1. **上述程序中多项式加法运算函数最终是生成一个新多项式链表进行存储，请修改程序使之把计算结果保存在第一个多项式链表（如 pa）中。**

2. **请参照多项式加法运算实现多项式乘法运算。**

1.5　小　　结

通过本章的练习，掌握线性表的逻辑结构特征，顺序存储和链式存储的特点以及在两种存储结构下基本操作的实现。

第2章　栈与队列

本章的典型实验主要练习栈和队列的几种不同存储形式的基本操作及其实际应用，特别是栈的实际应用，分为栈与队列的基本操作实验和栈与队列应用实验两类，共9个实验内容。

2.1　知识点概述

栈和队列是两种重要的线性结构，从数据结构上来看，栈与队列也是线性表，不同之处在于线性表的插入和删除操作可以在线性表的任意位置进行，但栈与队列的插入和删除操作是受限的。

栈是一种受限的线性表，其限制在于对它所有的插入和删除都限定在表的同一端进行，这一端称为栈顶，另一端则称为栈底。如果表中没有元素时称为空栈，通常将插入元素的过程叫做进栈，删除元素的过程叫做出栈。栈的修改是按后进先出的原则进行的，因此，栈又称为后进先出（Last In First Out，LIFO）的线性表。栈的基本操作有置空栈、读栈顶元素、进栈和出栈。

栈的顺序存储结构，又称为顺序栈，它是利用一组连续的地址存储单元依次存入从栈底到栈顶的数据元素。由于栈底的位置是不变的，栈顶的位置随着进栈、出栈的操作而变化的，因此设置指针 top 来指示栈顶元素在顺序栈中的位置。通常以 top = 0 表示空栈，栈的链式存储结构称为链栈，即用链表来存储栈。在实际应用中，更多的是使用顺序栈，除应用程序外，更主要地是 F 用于系统程序（如编译系统、操作系统），例如过程调用和返回、表达式求值等。

队列也是一种操作受限的线性表，其限制在于对它的所有插入都在表的一端进行，所有的删除都在表的另一端进行。进行插入的一端称为队列的尾，简称队尾，进行删除的一端称为队列的头，简称队头。当队列中没有元素时称为空队。通常在队列中把元素的插入称入队，把元素的删除称为出队。在队列中最早入队的元素也最早离开，所以又称为先进先出的线性表（First In First Out，FIFO）。队列的基本操作有 :建空队、置空队、判队空否、入队、出队和读队头元素等。用链式存储的队列简称为链队列，一个链队列需要两个分别指示队头和队尾的指针（分别称为头指针和尾指针）才能唯一确定。为了操作方便起见，给链队列添加一个头结点，并令头指针指向头结点。显然，当头指针和尾指针均指向头结点时链队列为空。链队列的插入操作和删除操作需要修改尾指针或头指针。

队列的顺序存储结构，除了用一组地址连续的存储单元依次存放从队列头到队列尾的元素之外，需要附设两个指针 front 和 rear，用于指示队列头元素和队列尾元素的位置。初始化建空队列时，令 Q. front = Q. rear = 0，每当插入新的队列尾元素时，“尾指针增 1”；每当删除队列头元素时，“头指针增 1”。在实际操作的过程中，队列中的实际可用空间并未占满，但尾指针已经指示到队列空间的最末端，即出现了“假溢出”现象。为了解决该问题，通常

使用循环队列,但无论队列处于“空”或“满”时,头指针与尾指针均相等,无法判别队列空间是“空”还是“满”。此时主要有三种处理方法:其一是另设一个标志位以区别队列是“空”还是“满”;其二是另设一个计数器用以判别队列是“空”还是“满”;其三是少用一个元素空间,约定以“队列头指针在队列尾指针的下一位置(指环状的下一位置)上”作为队列呈“满”状态的标志。

2.2　栈与队列的基本操作实验

【实验 2.1】　链栈的设计与实现

实现栈的链表存储的基本操作。

第 1 步:任务分析。

链表栈的基本操作包括栈的建立、求长度、取栈顶元素、入栈、出栈、判断栈是否空等具体操作。

第 2 步:程序构思。

栈的链式存储结构称为链栈,即用链表来存储栈。由于栈的操作是受限的,出栈与进栈都是在表的一端进行,因此程序就是实现特殊的链表操作,数据元素的存储与不带头结点的单链表相似,用一指针 top 指向单链表的第一个结点,插入和删除操作都在 top 端进行。

第 3 步:源程序。

```
1        //调试环境:Visual C++ 6.0
2
3        //-----库文件和预设定义
4        #include <stdlib.h>
5        #include <stdio.h>
6
7        #define Stack_Length 6
8        #define OK 1
9        #define ERROR 0
10
11       typedef int SElemType;
12       //存储形式
13       typedef struct SNode
14       {    SElemType data;
15            struct SNode *next;
16       }SNode,*LinkStack;
17
18       /*****--------------------------    *****/
19       // 函数名:    CreateTwo(LNode* head, int n)
20       // 参数:      (传入)LNode* head  传入一个链表指针
21       //            (传入)int n,传入线性表结点数量
22
```

```
23      // 作用:      建立一个空栈
24      // 返回值:    无
25      /* * * * - - - - - - - - - - - - - - - - - - - - - -   * * * * */
26      void CreateTwo(LinkStack &head, int n)
27      {
28        int i;
29        SNode *p;/*定义新结点指针*/
30
31        /*建立带头结点的线性链表*/
32        head=(LinkStack)malloc(sizeof(SNode));
33        head->next=NULL;
34
35        printf("Please input the data for LinkList Nodes: \n");
36        for(i=n;i>0;--i)
37        {
38            p=(SNode*)malloc(sizeof(SNode));/*为新结点申请空间,即创建一
              新结点*/
39            scanf("%d",&p->data); /*新结点赋值*/
40            /*新结点插入到表头*/
41            p->next=head->next;
42            head->next=p;
43        }
44      }
45
46      /* * * * - - - - - - - - - - - - - - - - - - - - - -   * * * * */
47      // 函数名:    Push(LinkStack &top,SElemType e)
48      // 参数:      (传入)LinkStack &top,栈顶指针
49      //            (传入)SElemType e   入栈元素
50      // 作用:      入栈
51      // 返回值:    int 型,返回1表示入栈成功,0表示失败
52     /* * * * - - - - - - - - - - - - - - - - - - - - - -   * * * * */
53      int Push(LinkStack &top,SElemType e)
54      {
55          SNode *q;
56          q=(LinkStack)malloc(sizeof(SNode));/*创建入栈元素结点*/
57
58          /*创建结点失败处理*/
59          if(!q)
60          {
61              printf("Overfolow !\n");
62              return (ERROR);
63          }
64          q->data=e;
```

```
65          q->next=top->next;/*入栈元素结点插入栈顶*/
66          top->next=q;
67
68          return (OK);
69      }
70
71  /*****--------------------------    *****/
72      //函数名:    Pop(LinkStack &top,SElemType &e)
73      //参数:      (传入)LinkStack &top,栈顶指针
74      //           (传出)SElemType e   出栈元素
75        //作用:    出栈
76        //返回值: int 型,返回 1 表示出栈成功,0 表示失败
77        /*****--------------------------    *****/
78        int Pop(LinkStack &top,SElemType &e)
79        {
80          SNode *q;/*定义临时存储栈顶结点的指针*/
81            if(!top->next)
82            {
83              printf("error");
84                return (ERROR);
85            }
86            e=top->next->data;
87            q=top->next;
88            top->next=q->next;/*删除栈顶元素*/
89            free(q);
90            return (OK);
91        }
92
93      //--------测试程序--------
94        void main()
95        {
96            int e;
97            /*建立一个栈*/
98            LinkStack top;
99            /*使用建立方法二*/
100           CreateTwo(top, 3);
101
102           /*显示栈元素*/
103           LinkStack p;
104           printf("\nThe old LinkStack is (top to bottom) : \n");
105           p=top;
106           while(p->next)
107           {
```

```
108                p = p -> next;
109                printf("%d ", p -> data);
110             }
111           printf("\nPlease input the data to push : ");
112           scanf("%d", &e);
113
114            /* 入栈操作 */
115            if(Push(top,e))
116                  printf("success to push");
117
118           /* 显示栈元素 验证入栈操作成功否 */
119           printf("\nThe new LinkStack is : \n");
120           while(top -> next)
121           {
122               top = top -> next;
123               printf("%d ", top -> data);
124           }
125
126       }
```

第 4 步：测试数据。

运行结果：

```
Please input the data for LinkList Nodes:
6 45 8
The old LinkStack is (top to bottom) :
8 45 6
Please input the data to push : 22
success to push
The new LinkStack is :
22 8 45 6
```

【实验 2.2】　循环队列的设计与实现

实现顺序存储的队列 (循环队列) 的基本操作。

第 1 步：任务分析。

循环队列的基本操作包括队列的建立、入队、出队、取队列元素、判断队列是否空等具体操作。

第 2 步：程序构思。

循环队列将顺序表的元素空间看作一个首尾相接的圆环，在线性圆环上设置队尾 rear 端和队头 front 端，限制在队尾 rear 端进行插入操作，在队头 front 端进行删除操作。整个流程描述如下。

(1) 入队操作。将队尾指针向前移：rear + 1，**若 rear 小于或者等于队列的最大索引值**

MaxSize - 1,则将数据存入 rear 所指的数组元素中,否则无法完成入队操作。

⑵ 出队操作。① 检查队列是否有数据。若头指针 front 等于尾指针 rear,则表示队列中无数据,若头指针 front 不等于 rear,则将队头指针向前移动 :front + 1。② 取出队头指针所指的数组元素内容。

第 3 步:源程序。

```
1        //调试环境:Visual C++6.0
2
3        //-----库文件和预设定义
4        #include <stdlib.h>
5        #include <stdio.h>
6
7        #define OVERFLOW -1
8        #define OK 1
9        #define ERROR 0
10
11       //队列存储结构
12       #define MAXSIZE 100        //最大队列长度
13       typedef  struct
14       {
15           int *elem;     //队列存储空间
16           int front;     //头指针,指向队头元素
17           int rear;      //尾指针,指向队尾元素的下一个位置
18       }SqQueue;
19
20       /*****------------------------    *****/
21       // 函数名:   InitQueue(SqQueue &Q)
22       // 参数:     (传出) SqQueue &Q,循环队列
23       // 功能:     构造一个空队列
24       /*****------------------------    *****/
25       int InitQueue(SqQueue &Q)
26       {
27       Q.elem=(int *)malloc(MAXSIZE * sizeof(int));
28       if (!Q.elem)exit(OVERFLOW);
29       Q.front=Q.rear=-1;     //设定头指针和尾指针的指向为空
30
31       for(int i=0; i<MAXSIZE; i++)     //清除数组内容
32           Q.elem[i]=-1;
33
34           return OK;
35       }
36
37       /* 返回 Q 的元素个数,即为队列的长度 */
```

```
38      int QueueLength(SqQueue Q)
39      {
40          return (Q.rear - Q.front + MAXSIZE) %MAXSIZE;
41      }
42      void Display(SqQueue Q)
43      {
44          for(int i =0; i <=QueueLength(Q); i ++)
45              if(Q.elem[i] != -1)printf("%d ", Q.elem[i]);
46          printf("\n");
47      }
48
49      /* * * * - - - - - - - - - - - - - - - - - - - - - - -    * * * * */
50      // 函数名：  EnQueue(SqQueue &Q, int e)
51      // 参数：    (传入) SqQueue &Q,循环队列
52      //           (传入) int e,入队元素
53      // 功能：    队列入队操作
54    /* * * * - - - - - - - - - - - - - - - - - - - - - - -    * * * * */
55      int EnQueue(SqQueue &Q, int e)
56      {
57          Q.rear = (Q.rear +1) %MAXSIZE;
58          if (Q.rear ==Q.front)    return ERROR;    //检查队列是否已满
59          Q.elem[Q.rear] =e;       //插入元素 e 为 Q 的队尾元素
60          return OK;
61      }
62
63     /* * * * - - - - - - - - - - - - - - - - - - - - - - -    * * * * */
64      // 函数名：  DeQueue(SqQueue &Q, int &e)
65      // 参数：    (传入) SqQueue &Q,循环队列
66      //           (传出) int e,出队元素
67      // 功能：    队列出队操作
68    /* * * * - - - - - - - - - - - - - - - - - - - - - - -    * * * * */
69      int DeQueue(SqQueue &Q, int &e)
70      {
71          if (Q.front ==Q.rear)return ERROR;    //检查队列是否已空
72          e =Q.elem[Q.front +1];    //删除 Q 的队头元素,并存放在 e 中
73          Q.elem[Q.front +1] = -1;
74          Q.front = (Q.front +1) %MAXSIZE;
75          return OK;
76      }
77
78      void main()
79      {
80          SqQueue Q;
```

```
81          InitQueue(Q);
82          int elem, e;
83          printf("Please input the data of queue(exit for 0):\n");
84          scanf("%d", &elem);
85          while (elem !=0)
86          {
87              EnQueue(Q, elem);
88              scanf("%d", &elem);
89          }
90          printf("The current queue is: ");
91          Display(Q);
92          DeQueue(Q, e);
93          printf("DeQueue() be excuted: ");
94          Display(Q);
95      }
```

第 4 步：测试数据。

运行结果：

```
Please input the data of queue(exit for 0):
1 2 3 4 5 6 0
The current queue is: 1 2 3 4 5 6
DeQueue( ) be excuted: 2 3 4 5 6
```

思考题：

请根据以上构思和具体实现，给出队列判空和判满的条件，并考虑在这种循环队列的实现方式中最多存放多少个元素。

【实验 2.3】　链队列的设计

实现链表存储的队列（链队列）的基本操作

第 1 步：任务分析。

链队列的基本操作包括队列的建立、入队、出队、取队列元素、判断队列是否空等具体操作，由于是特殊的单链表操作，故本实验是按照队列的思想对单链表操作的应用。

第 2 步：程序构思。

为链队列添加一个头结点，并指定两个分别指示队头和队尾的指针（分别称为头指针和尾指针），并令头指针指向头结点。当头指针和尾指针均指向头结点时链队列为空。链队列的操作即为单链表的插入和删除操作的特殊情况，只是同时需要修改尾指针或头指针，具体插入删除操作请参考第 1 章线性表基本操作实验部分。

第 3 步：源程序。

```
1       //调试环境：Visual C++6.0
2       //-----库文件和预设定义
3       #include <stdlib.h>
4       #include <stdio.h>
```

```
#define OVERFLOW -1
# define OK 1
# define ERROR 0

//链表节点定义
typedef struct QNode
{
    int data;
    struct QNode *next;
}QNode, *QueuePtr;

//队列头指针和尾指针结构体定义
typedef struct
{
    QueuePtr front;
    QueuePtr rear;
}LinkQueue;

/*****------------------------*****/
// 函数名:    InitQueue(LinkQueue &Q)
// 参数:      (传入)LinkQueue Q,队列指针
// 作用:      初始化一个空队列
// 返回值:    int 型,返回 1 表示创建成功,0 表示失败
/*****------------------------*****/
int InitQueue(LinkQueue &Q)
{
    Q.front = Q.rear = (QueuePtr)malloc(sizeof(QNode));
    if(! Q.front)    exit(OVERFLOW);
    Q.front -> next = NULL;
    return OK;
}

/*****------------------------*****/
// 函数名:    DestroyQueue(LinkQueue &Q)
// 参数:      (传入)LinkQueue Q,队列指针
// 作用:      销毁一个队列
// 返回值:    int 型,返回 1 表示销毁成功,0 表示失败
/*****------------------------*****/
int DestroyQueue(LinkQueue &Q)
{
    while(Q.front)
    {
        Q.rear = Q.front -> next;
```

```
48              free(Q.front);
49              Q.front = Q.rear;
50          }
51          return OK;
52      }
53
54      /* 队列出队入队操作 - - - - - - - - - - - - - - - */
55      //队尾插入元素
56      int EnQueue(LinkQueue &Q, int e)
57      {
58          QueuePtr p;
59          p = (QueuePtr)malloc(sizeof(QNode));
60          if( !p )exit(OVERFLOW);
61          p -> data = e;
62          p -> next = NULL;
63          Q.rear -> next = NULL;
64          Q.rear = p;
65          return OK;
66      }
67      //删除队头元素
68      int DeQueue(LinkQueue &Q, int &e)
69      {
70          QueuePtr p;
71          p = (QueuePtr)malloc(sizeof(QNode));
72          if(Q.front == Q.rear)return ERROR;
73          p = Q.front -> next;
74          e = p -> data;
75          Q.front -> next = p -> next;
76          if(Q.rear == p)Q.rear = Q.front;
77          free(p);
78          return OK;
79      }
80      /* * * * - - - - - - - - - - - - - - - - - - - - - -    * * * * */
81
82      void main()
83      {
84          LinkQueue Q;
85          if(InitQueue(Q))
86              printf("Initial Success !\n");
87          if(DestroyQueue(Q))
88              printf("Destroy Success !\n");
89      }
```

第 4 步：测试数据。

运行结果：

```
Initial Success !
Destroy Success !
```

2.3　栈与队列的应用

【实验 2.4】　用栈模拟队列的设计与实现

用栈模拟实现队列的基本操作。

第 1 步：任务分析。

用栈模拟实现队列的基本操作，关键在于模拟队列先进先出的特点。本实验首先要使用栈的基本操作，然后设计合适的栈来存储队列元素模拟队列操作。

第 2 步：程序构思。

设计使用两个栈模拟队列的操作，一个栈 (s1) 用于插入元素，另一个栈 (s2) 用于删除元素。每次删除元素时将前一个栈的所有元素读出然后入第二个栈。使用的栈的基本操作包括出栈、入栈、判定栈是否为空等 3 个操作。

第 3 步：源程序。

```
1       //调试环境：Visual C ++6.0
2
3       //- - - - -库文件和预设定义
4       #include <stdlib.h>
5       #include <stdio.h>
6
7       # define Stack_Length 6
8       # define OK 1
9       # define ERROR 0
10      /*   - - - - - - - - - -链栈基本操作- - - - - - - - - -  */
11      typedef struct SNode
12      {
13          int data;
14          struct SNode *next;
15      }SNode, *LinkStack;
16      void Create(LinkStack &head)
17      {
18          head = (LinkStack)malloc(sizeof(SNode));
19          head->next = NULL;
20      }
21      int Push(LinkStack &top, int e)
22      {
```

```
23          SNode *q;
24          q=(LinkStack)malloc(sizeof(SNode));/*创建入栈元素结点*/
25
26          /*创建结点失败处理*/
27          if(!q)
28          {
29              printf("Overfolow!\n");
30              return (ERROR);
31          }
32          q->data=e;
33          q->next=top->next;/*入栈元素结点插入栈顶*/
34          top->next=q;
35
36          return (OK);
37      }
38      int Pop(LinkStack &top)
39      {
40          int e;
41          SNode *q;/*定义临时存储栈顶结点的指针*/
42          if(!top->next)
43          {
44              printf("error");
45              return (ERROR);
46          }
47          e=top->next->data;
48          q=top->next;
49          top->next=q->next;/*删除栈顶元素*/
50          free(q);
51          return e;
52      }
53      int Empty(LinkStack &top)
54      {
55          if(top->next!=NULL)return OK;
56          else return ERROR;
57      }
58      void Display(LinkStack &top)
59      {
60          SNode *temp;
61          temp=top;
62
63          while(temp->next)
64          {
65              temp=temp->next;
```

```
66              printf("%d ", temp ->data);
67          }
68          printf("\n");
69      }
70      /*   - - - - - - - - - - - - -链栈基本操作 */
71
72      /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
73      // 函数名：   EnQueue(LinkStack &s1, int e)
74      // 参数：     (传入) LinkStack &s1, 队列(用栈模拟)
75      //            (传入) int e,入队元素
76      // 功能：     队列入队操作
77      /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
78      void EnQueue(LinkStack &s1, int e)
79      {
80          Push(s1, e);
81      }
82
83      /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
84      // 函数名：   DeQueue(LinkStack &s1, LinkStack &s2, int &e)
85      // 参数：     (传入) LinkStack &s1,,队列(用栈模拟)
86      //            (传入) LinkStack &s2,,辅助栈
87      //            (传出) int e,出队元素
88      // 功能：     队列出队操作
89      /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
90      void DeQueue(LinkStack &s1, LinkStack &s2, int &e)
91      {
92          s2 ->next =NULL;    //将 s2 清空
93          while( Empty(s1) )Push(s2, Pop(s1));    //s1 所有元素退栈并入栈 s2
94          e =Pop(s2);    //将 s1 的栈顶元素退栈并赋值给 e
95          while( Empty(s2))Push(s1, Pop(s2));    //s2 所有元素退栈并入栈 s1
96      }
97      void main()
98      {
99          int elem;
100         int e;
101         LinkStack s1, s2;
102         Create(s1);
103         Create(s2);
104         printf("Please input the data of queue(exit for 0):\n");
105         scanf("%d", &elem);
106         while (elem !=0)
107         {
108             EnQueue(s1, elem);
```

```
109             scanf("%d", &elem);
110         }
111         printf("The current queue is: ");
112         Display(s1);
113         DeQueue(s1, s2, e);
114         printf("DeQueue() be excuted: ");
115         Display(s1);
116         DeQueue(s1, s2, e);
117         printf("DeQueue() be excuted: ");
118         Display(s1);
119     }
```

第 4 步：测试数据

运行结果：

```
Please input the data of queue(exit for 0):
1 2 3 4 5 6 0
The current queue is: 6 5 4 3 2 1
DeQueue() be excuted: 6 5 4 3 2
DeQueue() be excuted: 6 5 4 3
```

【实验 2.5】　用栈排序的设计与实现

对一组数据用栈的形式进行排序操作。

第 1 步：任务分析。

为了用栈完成数据排序操作，首先要实现栈的基本操作，主要包括出栈和入栈操作。然后应用出栈和入栈的操作，完成数据排序，最终把数据按顺序存入栈中。

第 2 步：程序构思。

由于排序过程重复出栈入栈操作以及大小比较，故适合设计成递归过程，描述的递归过程如下。

比较数组元素 a 和栈顶元素 b；

若 a > b，栈顶元素出栈，若栈不为空则再比较数组元素和栈顶元素，否则 a 元素入栈，然后出栈元素入栈。

否则，数组元素小于栈顶元素，数组元素入栈。

递归结束条件：

当栈为空时候，数组当前元素入栈，结束递归调用。

第 3 步：源程序。

```
1       //调试环境：Visual C++6.0
2
3       //-----库文件和预设定义
4       #include <stdlib.h>
5       #include <stdio.h>
6
```

```
/*  -------链栈基本操作-------- */
typedef struct stack
{
    int data;
    struct stack *link;
}sStack,*sNode;
sStack *createStack()
{
    sStack* p;
    p=(sStack*)malloc(sizeof(sStack));
    p->link=NULL;
    return p;
}
void push(sNode &head, int e)
{
    sStack* p;
    p=(sStack*)malloc(sizeof(sStack));
    p->data=e;
    p->link=head;
    head=p;
}
void pop(sNode &head)
{
    sStack* p;
    int e;
    if(head==NULL)exit(0);
    e=head->data;
    p=head;
    head=head->link;
    free(p);
}
int getStackTop(sNode &head)
{
    return head->data;
}
int isEmpty(sNode &head)
{
    if(head==NULL)
        return 0;
    else return 1;
}
/*  ----------链栈基本操作 */

```

```
/*****------------------------*****/
// 函数名：  compare(sNode &head, int a, int b)
// 参数：    (传入) sNode &head, 排序用栈
//           (传入) int a, 待比较数组元素
//           (传入) int b, 栈顶元素
// 功能：    两数字比较排序
/*****------------------------*****/
void compare(sNode &head, int a, int b)
{
    int t;
    if(a > b)
    {
        pop(head);    //数组元素大于栈顶元素,栈顶元素出栈
        if( isEmpty(head) )
        {
            t = getStackTop(head);    //取栈顶元素
            compare(head, a, t);    //比较数组元素和栈顶元素
        }
        else push(head, a);
        push(head, b);
    }
    else push(head, a);        //数组元素小于栈顶元素,数组元素入栈
}

void main()
{
    int temp;
    int a[100];/*predefine the size of array is 100*/
    int num;

    sNode pStack;
    sNode head;

    /*input the values*/
    printf("input the number of array:\n");
    scanf("%d", &num);
    printf("input the value of %d number of array:\n",num);
    for(int i =0; i <num; i ++)
        scanf("%d", &a[i]);

    pStack = createStack();
    head = pStack ->link;

```

```
93          for(int j =0;j <num;j ++)
94          {
95              if(isEmpty(head))
96              {
97                  //取栈顶元素,和未排序的下一元素进行排序
98                  temp =getStackTop(head);
99                  compare(head, a[j], temp);
100             }
101             else push(head, a[j]);    //第一次执行时使数组第一个元素入栈
102         }
103
104         printf("sort by asc:\n");
105         for(int k =0; head!=NULL; k ++, head =head ->link)
106             printf("%d ", head ->data);
107         printf("\n");
108     }
```

第 4 步：测试数据。

运行结果：

```
input the number of array:
5
input the value of 5 number of array:
12 0 63 5 13
sort by asc:
0 5 12 13 63
```

【实验 2.6】　算术表达式求值的设计与实现

输入一个算术表达式，实现计算其值的程序。

第 1 步：任务分析。

为了完成算术表达式求值运算，要用栈存储用户输入的由操作数 (operand)、运算符 (operator) 和界限符 (delimiter) 组成的算术表达式，重点在于分离操作数和运算符，关键在于判定两个运算符之间的优先关系。

第 2 步：程序构思。

使用两个工作栈。运算符栈 OPTR，用以寄存运算符；操作数栈 OPND，用以寄存操作数或运算结果。算法的基本思想是：

⑴ 首先置操作数栈为空栈，表达式起始符“#”为运算符栈的栈底元素；

⑵ 依次读入表达式中每个字符，若是操作数则进 OPND 栈；若是运算符，则和 OPTR 栈的栈顶运算符比较优先权后作相应操作，直至整个表达式求值完毕 (即 OPTR 栈的栈顶元素和当前读入的字符均为“#”)。

算法中调用了两个函数。其中 Compare 是判定运算符栈的栈顶运算符 $\gamma 1$ 与读入的运算符 $\gamma 2$ 之间优先关系的函数，运算符之间的优先关系被定义存储为一个 7×7 数组中；

Operate为进行二元运算 aγb 的函数，如果是编译表达式，则产生这个运算的一组相应指令并返回存放结果的中间变量名；如果是解释执行表达式，则直接进行该运算，并返回运算的结果。

算术表达式求值过程如图 2-1 所示。

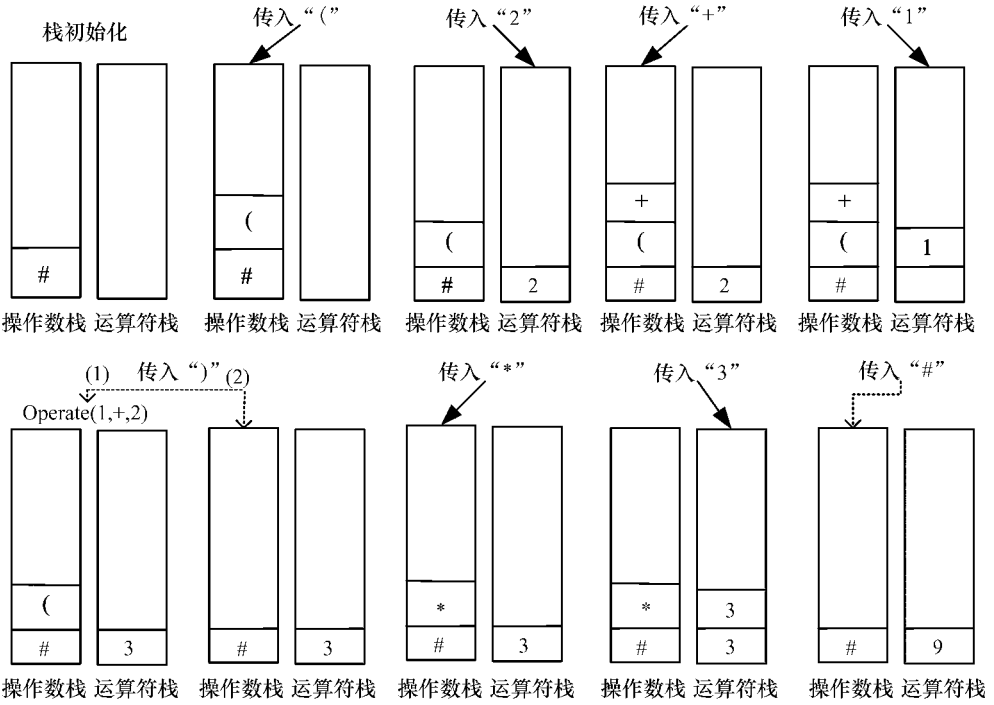


图 2-1　算术表达式求值过程示意图

第 3 步：源程序。

```
#include "stdlib.h"
#include "stdio.h"
#define OK 0
#define ERROR -1
//定义栈结构体类型
typedef char element;
typedef struct SNode
{
    element data;
    struct SNode *next;
}SNode,*LinkStack;

/*****------------------------------*****/
// 函数名：  Create_linkStack/Push/Pop/GetTop
// 作用：    创建一个栈/执行出栈操作/执行进栈操作/取栈顶元素值，
/*****------------------------------*****/
```

```
17      void Create_linkStack(LinkStack &L)
18      {
19          L = (LinkStack)malloc(sizeof(SNode));
20          if(!L)
21              printf("shenqingshibai");
22          L -> next = NULL;
23      }
24      int Push(LinkStack &top,element e)
25      {
26          SNode * q;
27          /* 创建入栈元素结点 */
28          q = (LinkStack)malloc(sizeof(SNode));
29          if(! q)
30          {
31              printf("Overflow ! \n");
32              return (ERROR);
33          }
34          q -> data = e;
35          q -> next = top -> next; /* 入栈元素结点插入栈顶 */
36          top -> next = q;
37          return (OK);
38      }
39      char Pop(LinkStack &top,element &e)
40      {
41          /* 定义临时存储栈顶结点的指针 */
42          SNode * q;
43          if(!top)
44          {
45              printf("error!");
46              return (ERROR);
47          }
48          e = top -> next -> data;
49          ////////////////////////////////////
50          q = top -> next;
51          top -> next = q -> next;
52          //正确处理头结点
53          //原：q = top;
54          //    top = top -> next;
55          ////////////////////////////////////
56          free(q);
57          return e;
58      }
59      char GetTop(LinkStack &top)
```

```
60      {
61          element e;
62          e = top -> next -> data;
63          //top = top -> next;    //gettop 是访问栈元素的,不能改变栈
64          return e;
65      }
66      /* * * * - - - - - - - - - - - - - - - - - - - - - -   * * * * */
67
68      /* * * * - - - - - - - - - - - - - - - - - - - - - -   * * * * */
69      // 函数名:    In(char c)
70      // 参数:      (传入)char c,字符
71      // 作用:      判断一个字符是否是运算符或#
72      // 返回值:    整型值,0 代表是,1 代表不是
73      /* * * * - - - - - - - - - - - - - - - - - - - - - -   * * * * */
74      int In(char c)
75      {
76          int i;
77          int flag = 1;
78          char OP[] = {" + - * / () #"};
79          for(i = 0;i < 7;i ++)
80              if(c == OP[i])flag = 0;
81
82              return flag;
83      }
84
85      /* * * * - - - - - - - - - - - - - - - - - - - - - -   * * * * */
86      // 函数名:    In(char c)
87      // 参数:      (传入)char c,字符
88      // 作用:      将字符转换为索引数字
89      // 返回值:    整型值,按照自定义,完成字符到指定索引的转换
90      /* * * * - - - - - - - - - - - - - - - - - - - - - -   * * * * */
91      int contrast(char ch)
92      {
93          switch(ch)
94          {
95          case '+':  return 0;break;
96          case '-':  return 1;break;
97          case '*':  return 2;break;
98          case '/':  return 3;break;
99          case '(':  return 4;break;
100         case ')':  return 5;break;
101         case '#':  return 6;break;
102         default :  return -2;
```

```
103         }
104     }
105
106     /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
107     // 函数名:    PriorTable(int m,int n)
108     // 参数:      (传入)int m,索引数字,代表一个运算符
109     //            (传入)int n,索引数字,代表一个运算符
110     // 作用:      建立运算符对照表(二维数组),并查找指定字符的关系(1,0, -1)
111     // 返回值:    整型值,按照自定义,1 表示大于关系,0 表示等于关系, -1 表示小于
                   关系
112     /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
113     int PriorTable(int m,int n)
114     {
115         int a[7][7];//运算符间的关系存储在二维数组中
116         int i,j;
117
118         //定义优先关系:1 表示大于关系,0 表示等于关系, -1 表示小于关系
119         for(i =0;i <7;i ++)
120         {
121             for(j =0;j <7;j ++)
122                 a[i][j] =1;
123         }
124         a[6][6] =a[4][5] =0;
125         for(j =0;j <5;j ++)
126         {
127             a[6][j] = -1;
128             a[4][j] = -1;
129         }
130         a[0][2] =a[0][3] =a[0][4] = -1;
131         a[1][2] =a[1][3] =a[1][4] = -1;
132         a[2][4] = -1;
133         a[3][4] = -1;
134
135         return a[m][n];     //返回指定索引数字的运算符的优先关系(1,0, -1)
136     }
137
138     /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
139     // 函数名:    Compare(char opr,char c)
140     // 参数:      (传入)char opr,栈顶运算符
141     //            (传入)char c,输入运算符
142     // 作用:      根据字符间的关系(1,0, -1)得到栈顶运算符与输入的运算符关系
                   (>, =, <)
143     // 返回值:    字符型,得到大于/等于/小于关系
```

```
/*****------------------------*****/
char Compare(char opr,char c)
{
    int m,n;
    char tag;
    m=contrast(opr);
    n=contrast(c);
    if(PriorTable(m,n)==1)
        tag='>';
    if(PriorTable(m,n)==0)
        tag='=';
    if(PriorTable(m,n)== -1)
        tag='<';

    return tag;
}

/*****------------------------*****/
// 函数名:    Operate(char a,char theta,char b)
// 参数:      (传入)char a,变量
//            (传入)char theta,输入运算符
//            (传入)char b,变量
// 作用:      求出表达式的值
// 返回值:    字符型,得到运算结果
/*****------------------------*****/
char Operate(char a,char theta,char b)
{
    int t=0;
    switch(theta)
    {
    case '+':   t=(a-'0')+(b-'0');break;//need break;
    case '-':   t=(a-'0')-(b-'0');break;
    case '*':   t=(a-'0')*(b-'0');break;
    case '/':   t=(a-'0')/(b-'0');break;
    default :  printf("error33");
    }

    return (t+'0');
}

/*****------------------------*****/
// 函数名:  EvaluateExpression()
// 作用:    算术表达式求值的处理
```

```
187    // 返回值： 字符型,得到整个表达式结果
188    /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
189    char EvaluateExpression()
190    {
191        char c;
192        char theta; //存储运算符
193        char a =0,b =0;
194        LinkStack OPTR; //运算符栈
195        LinkStack OPND; //操作数栈
196        Create_linkStack(OPTR);
197        Push(OPTR,'#');
198        Create_linkStack(OPND);
199        printf("请输入表达式:\n");
200        c =getchar();    //取得表达式
201        while(c!='#'||GetTop(OPTR)!='#')
202        {
203            int temp;
204            temp =In(c); //判定是否是运算符
205            if(temp)
206            {
207                //不是运算符进操作数栈
208                Push(OPND,c);
209                c =getchar();
210            }
211            else
212            {
213                switch (Compare(GetTop(OPTR),c))
214                {
215                case '<':   //栈顶元素优先级低
216                    Push(OPTR,c);
217                    c =getchar();
218                    break;
219                case '=':   //脱括号并接收下一字符
220                    Pop(OPTR,c);
221                    c =getchar();
222                    break;
223                case '>':   //退栈并将运算结果入栈
224                    {
225                        theta =GetTop(OPTR);
226                        Pop(OPTR,theta);
227                        Pop(OPND,b);
228                        Pop(OPND,a);
229                        Push(OPND,Operate(a, theta, b));
```

```
                    break;
                }
            }//switch
        }//else
    }//while

    return GetTop(OPND);    //返回运算结果
}

//演示调用计算
void main()
{
      printf("%c", EvaluateExpression());
}
```

第 4 步：测试数据。

运行结果：

```
请输入表达式：
(2 +1) * 3#
9
```

思考题：

1. 上述程序中运算符的优先关系是通过建立索引数字得到的，请思考还能以什么形式实现，并编写程序替换上述 Compare () 函数。

2. 上述程序是把输入的字符视作字符处理的，所以不能对多位数进行计算。请思考修改上述程序，使之实现多位数计算。（提示：把输入的多位数组合成一个数字变量入栈，或者建立数字操作数栈。）

【实验 2.7】　汉诺塔问题

用计算机模拟解决 n 阶汉诺塔问题。

第 1 步：任务分析。

由于 n 阶汉诺塔问题，将一个圆盘和将 n - 1 个圆盘从一个塔座移动到另外一个塔座具有相同的特征，故使用基于栈应用的递归求解比较方便。

第 2 步：程序构思。

递归部分：

(1) 将编号为 n 的圆盘上的 n - 1 个圆盘从塔座 x 移至塔座 y 上；

(2) 将编号为 n 的圆盘从塔座 x 移至塔座 z 上；

(3) 将塔座 y 上的 n - 1 个圆盘移至塔座 z 上。

递归结束条件：

当 n = 1 时，将编号为 1 的圆盘从塔座 x 移至塔座 z 上。

汉罗塔移动园盘的过程如图 2-2 所示。

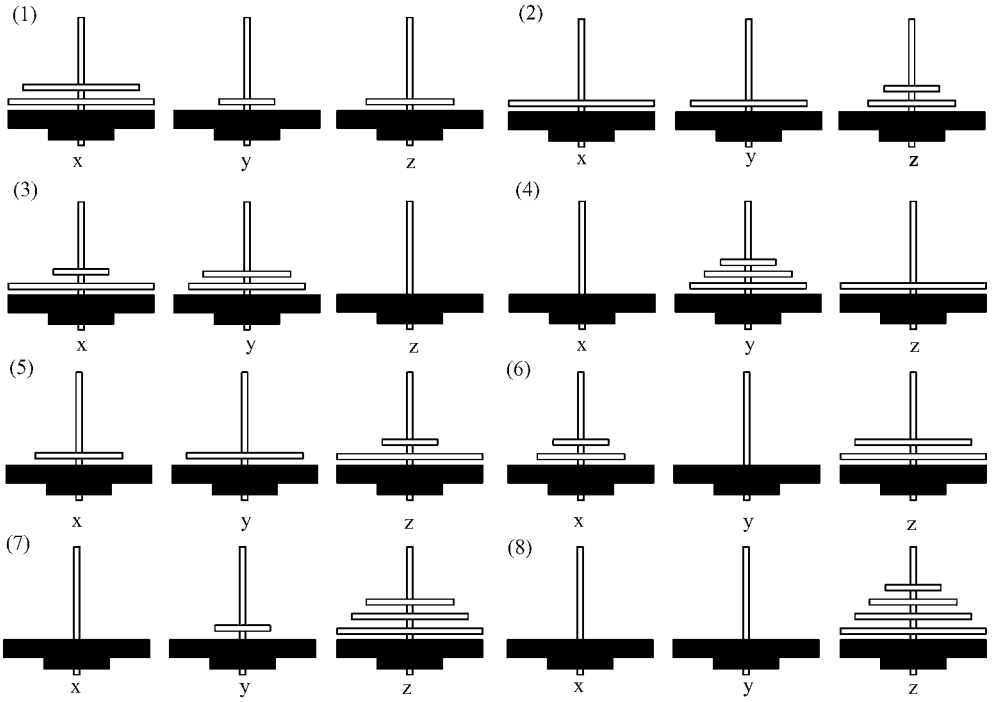


图 2-2　汉罗塔移动过程示意图

第 3 步：源程序。

```
#include <stdio.h>

/*****------------------------*****/
// 函数名：  move(char x, int n, char z)
// 参数：    (传入)char x, z 圆盘移动位置
//           (传入)int n 圆盘编号
// 功能：    圆盘移动操作
/*****------------------------*****/
void move(char x, int n, char z)
{
    printf("move disk %i from %c to %c\n", n, x, z);
}

/*****------------------------*****/
// 函数名：  Hanoi(int n, char x, char y, char z)
// 参数：    (传入)int n 圆盘编号
//           (传入)char x, y, z 塔座位置
// 功能：    n 阶 Hanoi 问题求解
/*****------------------------*****/
void Hanoi(int n, char x, char y, char z)
{
```

```
22      if (n ==1)
23          move(x, 1, z);          //将编号为 1 的圆盘从 x 移动到 z
24      else
25      {
26          Hanoi(n -1, x, z, y);   //将 x 上编号为 n -1 的圆盘移到 y,z
27          move(x, n, z);          //将编号为 n 的圆盘从 x 移动到 z
28          Hanoi(n -1, y, x, z);   //将 y 上编号为 n -1 的圆盘移到 z,x
29      }
30    }
31    //测试主程序
32    void main()
33    {
34        Hanoi(4, 'x', 'y', 'z');
35    }
```

第 4 步：测试数据 (如图 2-1 的示意过程)。

运行结果：

```
move disk 1 from x to y
move disk 2 from x to z
move disk 1 from y to z
move disk 3 from x to y
move disk 1 from z to x
move disk 2 from z to y
move disk 1 from x to y
move disk 4 from x to z
move disk 1 from y to z
move disk 2 from y to x
move disk 1 from z to x
move disk 3 from y to z
move disk 1 from x to y
move disk 2 from x to z
move disk 1 from y to z
```

思考题：

1. **汉诺塔问题实际上是递归以栈的形式实现。请参照教材理解函数“层次”运行时配合的递归工作栈。**

2. **请按照递归实现一个字符串的倒序输出。**

提示：

```
1     void magic()
2     {
3         char ch;
4         scanf("%c", &ch);
5         if(ch != 'q')
```

```
6         {
7             magic();
8             printf("%c", ch);
9         }
10    }
```

【实验 2.8】　迷宫求解问题

迷宫路径求解。

第 1 步：任务分析。

编制程序给出一条通过迷宫的路径或报告一个无可通过路径的信息。

第 2 步：程序构思。

迷宫求解思路：寻找一条通过迷宫的路径，必须进行试探性搜索，只要有路可走就前进一步，无路可走时，退回一步，重新选择未走的可走之路，如此继续，直至到达出口或者返回入口（无法通过迷宫）。

⑴ 入口点入栈，开始进行试探性搜索。

⑵ 出栈一个点，使之为当前点。

⑶ 依次试探当前点的每个方向：

① 如果有路则输出路径；

② 如果走到没走过的点，作标记表明已走过，当前点入栈，使下一点转换成当前点。

迷宫求解过程如图 2-3 所示。

1	1	1	1	1	1	1	1	1	1	1
1	0	1	0	0	1	1	1	0	0	1
1	0	0	0	0	0	1	0	0	1	1
1	0	1	1	1	0	0	0	1	1	1
1	0	0	0	1	0	1	1	0	1	1
1	1	0	0	1	0	1	1	0	0	1
1	1	1	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1	1	1

(a) 初始状态入口 (1,1), 出口 (6,9)

1	1	1	1	1	1	1	1	1	1	1
1	0	1	0	0	1	1	1	0	0	1
1	0	0	0	0	0	1	0	0	1	1
1	0	1	1	1	0	0	0	1	1	1
1	0	0	0	1	0	1	1	0	1	1
1	1	0	0	1	0	1	1	0	0	1
1	1	1	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1	1	1

(b) 求解路径

图 2-3　迷宫求解示意图

第 3 步：源程序。

```
1         #include <stdio.h>
2         #include <stdlib.h>
```

```
#define OVERFLOW -1

#define MAX 100     //栈中最大元素个数
#define N 11     //长度

typedef struct
{
    int  x;    // 行下标
    int  y;    //列下标
    int  d;    //运动方向
}Data;

//顺序栈类型定义
typedef struct
{
    int  pos;        //指示栈顶位置
    Data  data[MAX];
}SNode, *Stack;
//顺序栈基本操作 - - - - - - - - - - - -
Stack InitStack()
{
    Stack pStack;
    pStack = (Stack)malloc(sizeof(SNode));
    if (!pStack)exit(OVERFLOW);
    pStack ->pos = -1;
    return pStack;
}
int IsEmpty(Stack stack)
{
    return stack ->pos == -1;
}
/* 在栈中压入一元素 x */
void  Push(Stack pStack, Data x )
{
    if(pStack ->pos >=MAX -1) exit(OVERFLOW);
    else
    {
        pStack ->pos ++;
        pStack ->data[pStack ->pos] =x;
    }
}
/*删除栈顶元素*/
void  Pop(Stack pStack)
```

```
46      {
47          if (pStack->pos == -1 )exit(OVERFLOW);
48          else
49              pStack->pos--;
50      }
51      /*求栈顶元素的值*/
52      Data  GetTop(Stack pStack )
53      {
54          return (pStack->data[pStack->pos]);
55      }
56      //- - - - - - - - - - - - - - - - - - - - - -
57
58      /*设置栈元素的值*/
59      Data SetStackElem(int x, int y, int d)
60      {
61          Data element;
62          element.x=x;
63          element.y=y;
64          element.d=d;
65          return element;
66      }
67      /*输出栈元素具体的值*/
68      void DisplayPath(Stack pStack)
69      {
70          Data element;
71          printf("The path is:\n");
72          while(!IsEmpty(pStack))
73          {
74              element=GetTop(pStack);
75              Pop(pStack);
76              printf("the node is: %d %d \n", element.x, element.y);
77          }
78      }
79      /*****- - - - - - - - - - - - - - - - - - - - - -    *****/
80      // 函数名：  MazePath(int maze[][N], int direction[][2], int x1,
                     int y1, int x2, int y2)
81      // 参数：    (传入) int maze[][N] 迷宫
82      //           (传入) int direction[][2] 表示方向(上/下/左/右/左上/左
                     下/右上/右下)
83      //           (传入) int x1, int y1 入口
84      //           (传入) int x2, int y2 出口
85      // 返回值：  1 表示成功,0 表示操作失败
86      // 功能：    迷宫 maze[M][N] 中求从入口 maze[x1][y1] 到出口 maze[x2]
```

```
                    [y2]的一条路径
87      // 备注：     其中 1 <= x1,x2 <= M - 2 , 1 <= y1,y2 <= N - 2
88      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
89      void MazePath(int maze[][N], int direction[][2], int x1, int y1,
        int x2, int y2)
90      {
91          int i, j, k, g, h;
92          Stack pStack;
93          Data element;
94          pStack = InitStack();
95          maze[x1][y1] = 2;      //从入口开始进入,作标记
96            Push(pStack, SetStackElem(x1, y1, -1));    //入口点进栈
97
98          while ( !IsEmpty(pStack))    //走不通时,一步步回退
99            {
100             element = GetTop(pStack);
101             Pop(pStack);
102
103             i = element.x;
104               j = element.y;
105
106              //依次试探每个方向
107            for (k = element.d + 1; k <= 3; k ++)
108              {
109               g = i + direction[k][0];
110                 h = j + direction[k][1];
111               if (g == x2 && h == y2 && maze[g][h] == 0)    //走到出口点
112                 {
113                     DisplayPath(pStack);    //打印路径
114                     return;
115                 }
116               if (maze[g][h] == 0)    //走到没走过的点
117                 {
118                     maze[g][h] = 2;     //作标记
119                     Push(pStack, SetStackElem(i, j, k));    //进栈
120                     //下一点转换成当前点
121                     i = g;
122                     j = h;
123                     k = -1;
124                 }
125             }
126         }
127         //栈退完未找到路径
```

```
128          printf("The path has not been found.\n");
129        }
130
131        void main()
132        {
133            int direction[][2]={0,1,1,0,0,-1,-1,0};
134            int maze[][N]={
135                1,1,1,1,1,1,1,1,1,1,
136                1,0,1,0,0,1,1,1,0,0,1,
137                1,0,0,0,0,0,1,0,0,1,1,
138                1,0,1,1,1,0,0,0,1,1,1,
139                1,0,0,0,1,0,1,1,0,1,1,
140                1,1,0,0,1,0,1,1,0,0,1,
141                1,1,1,0,0,0,0,0,0,0,1,
142                1,1,1,1,1,1,1,1,1,1,1
143            };
144            MazePath(maze,direction,1,1,6,9);
145        }
```

第 4 步：测试数据（路径如图 2-3（b）所示）。

运行结果：

```
The path is:
the node is: 6 7
the node is: 6 6
the node is: 6 5
the node is: 5 5
the node is: 4 5
the node is: 3 5
the node is: 2 5
the node is: 2 4
the node is: 2 3
the node is: 2 2
the node is: 2 1
the node is: 1 1
```

思考题：

1. **参照迷宫求解程序，实现顺序栈的基本操作。**

2. **请运用递归解迷宫问题。**

提示：

递归执行部分：

判断传入坐标是否可走，如果可走，递归调用判断向上、右上、下、右下等八个方向的坐标是否可走。如果可走，返回一个标记，如果不可走则标记已经走过。

递归结束条件：

当已经走到出口时结束。

【实验 2.9】　八皇后问题

在一个 8 ×8 的棋盘里放置 8 个皇后,要求每个皇后两两之间不相“冲”(在每一横列、竖列、斜列只有一个皇后)。

第 1 步:任务分析。

根据不冲突条件编制程序列出符合题意的共 92 组解。首先明确互不冲突的处理分类:

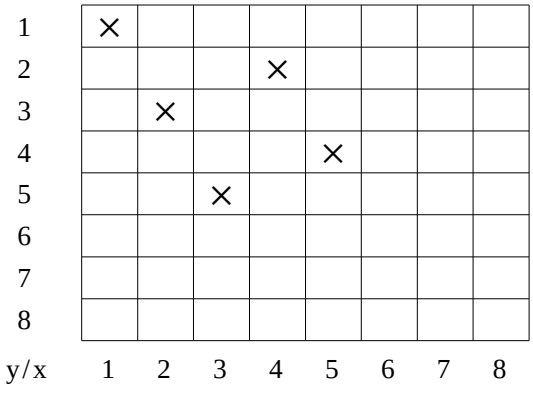
① 每一列放一个皇后,不会造成列上的冲突;

② 当第 i 行被某个皇后占领后,则同一行上的所有空格都不能再放皇后;

③ 当第 i 个皇后占领了第 j 列后,在同一对角线上的两个方向上的所有点不能再放皇后。

第 2 步:程序构思。

根据以上分析,建立存放当前皇后放置位置的栈,搜寻如下:



(1) 皇后由 (1,1) 开始放置,将皇后 y 坐标入栈,stack [1] = 1,top = 1

(2) 找第 top + 1 (j = top + 1) 皇后的安全位置:由 newq = 1 (y 坐标 = 1) 开始往下找,每次检查新皇后坐标与旧皇后 (在 stack 中) 是否相冲突,根据 x,y 坐标判断此位置是否可以放置新皇后。设定新皇后 (j, newq),旧皇后 (m, stack [m]),检查不安全的情况:

同一列 (y):newq == stack [m]

斜角 (考虑有两个方向):| j - m| == | newq - stack [m] |

若安全,将皇后 y 坐标入栈,top = top + 1;stack [top] = newq;

若不安全,往下找:newq ++ ;

(3) if newq > 8,表示 8 个位置皆无安全位置,则回溯,出栈旧皇后 y 坐标,继续往下找。

newq = stack [top];top - - ;

newq ++ ;

(4) top == 8 :求得一组解。

第 3 步:源程序。

```
1       #include "stdio.h"
2       #define M 8
3
```

```
4       char queen[M+1][M+1];
5       int stack[M+2];
6       int top=1;
7
8       /*****------------------------*****/
9       // 函数名：  Display()
10      // 功能：    显示八皇后问题的正解,给出棋子位置和棋盘形状
11      /*****------------------------*****/
12      void Display()
13      {
14          int i, j;
15          for(i=1; i<=top; i++)
16              printf("(%d, %d)",stack[i],i);
17          printf("\n");
18
19          //绘制合适的棋盘
20          for(i=1;i<=M;i++)
21              for(j=1;j<=M;j++)
22                  queen[i][j]='0';
23              for(i=1;i<=top;i++)
24                  queen[stack[i]][i]='*';
25              for(i=1;i<=M;i++)
26              {
27                  printf(" ");
28                  for(j=1;j<=M;j++)
29                      printf("%c",queen[i][j]);
30                  printf("\n");
31              }
32      }
33
34      /*****------------------------*****/
35      // 函数名：  CheckPos(int x,int y)
36      //参数：    x,y 棋盘坐标
37      // 功能：    根据 x,y 坐标判断此位置是否可以放置
38      //返回值：  1 表示可以,0 表示不可以
39      /*****------------------------*****/
40      int CheckPos(int x,int y)
41      {
42          int i, flag=1;
43          for(i=1; flag&&i<=top; i++)
44              if(stack[i]==x)
45                  flag=0;
46              for(i=1; flag&&i<=top; i++)
```

```
47                if ((x-stack[i]==y-i)||(stack[i]-x==y-i))
48                    flag=0;
49
50            return flag;
51        }
52
53        void main()
54        {
55            int pos, flag=1;
56            stack[top]=1;
57            pos=1;
58            while(flag) // 标志 1 循环
59            {
60                while(pos <=M)
61                {
62                    if (CheckPos(pos, top+1)) //此位置是否可放
63                        break;
64                    else
65                        pos++; //不可放,找下一个位置
66                }
67                if(pos <=M)
68                {
69                stack[++top]=pos; //找到放置位置且在棋盘内部,把此位置存入到栈中
70                    if(top==M) // 栈满了
71                    {
72                        Display();//显示
73                        pos=stack[top--]+1; //这里是寻找下一种可能的方案。
74                    }
75                    else
76                        pos=1; //找到后,接着找 ++top,l=1;
77                }
78                else
79                {
80                    pos=stack[top--]+1; //没找到,回朔
81                    if(top==0 && pos>M)
82                        flag=0;
83                }
84            }
85        }
```

第 4 步：测试数据。

测试数据示意图如图 2-4 所示。

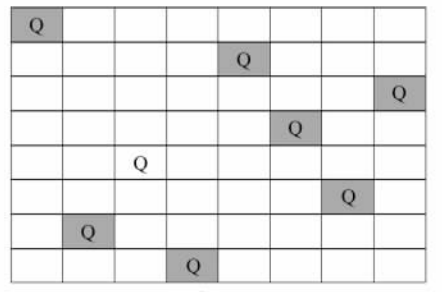


图 2-4　八皇后问题示意图

运行部分结果：

```
(8, 1)(2, 2)(5, 3)(3, 4)(1, 5)(7, 6)(4, 7)(6, 8)
0  0  0  0  *  0  0  0
0  *  0  0  0  0  0  0
0  0  0  *  0  0  0  0
0  0  0  0  0  0  *  0
0  0  *  0  0  0  0  0
0  0  0  0  0  0  0  *
0  0  0  0  0  *  0  0
*  0  0  0  0  0  0  0
(8, 1)(3, 2)(1, 3)(6, 4)(2, 5)(5, 6)(7, 7)(4, 8)
0  0  *  0  0  0  0  0
0  0  0  0  *  0  0  0
0  *  0  0  0  0  0  0
0  0  0  0  0  0  0  *
0  0  0  0  0  *  0  0
0  0  0  *  0  0  0  0
0  0  0  0  0  0  *  0
*  0  0  0  0  0  0  0
```

2.4　小　　结

通过顺序栈和链栈的存储实现，了解掌握栈的操作特点以及栈的实际应用 通过循环队列和链队列的存储实现，了解掌握队列的操作特点以及队列的实际应用。

第 3 章　串

本章的典型实验主要练习串的基本操作及其初步应用。典型实验分为串的基本操作实验和串的应用实验两类，共 6 个实验内容。

3.1　知识点概述

串是一种特殊的线性表，它的每个结点仅由一个字符组成。串是计算机中主要的非数值处理的对象，在计算机文字编辑、词法分析等方面有着广泛的应用。串的基本运算有：串赋值、复制串、判断串空否、求串长、串联接、求子串、子串位置、串比较、串置换、插入和删除等。串的存储除熟知的顺序存储（分为压缩格式和非压缩格式。）和链式存储之外，还有根据关键码组织索引表的索引存储（堆存储）。

3.2　串的基本操作实验

【实验 3.1】　字符串复制实验

编程实现字符串复制。

第 1 步：任务分析。

静态存储的字符串的复制问题的程序实现是字符数组的基本操作，重要的在于确认新的字符串的长度。

第 2 步：程序构思。

⑴ 扫描源字符串，逐一复制到目的字符串；

⑵ 指定目的字符串的长度。

第 3 步：源程序。

```
1       //调试环境：Visual C ++ 6.0
2       #include <stdio.h>
3       //-----库文件和预设定义
4       #define MAXSTRLEN 255
5       typedef unsigned char SString[MAXSTRLEN +1];
6
7       /*****------------------------------*****/
8       // 函数名：    strLengh(SString S)
9       // 参数：      (传入)SString S,字符串
10      // 作用：      得到串的长度,放于下标为 0 的位置
11      // 返回值：    无
12      /*****------------------------------*****/
```

```
13      void strLengh(SString S)
14      {
15          int m;
16          for(m=1;S[m]!='\0';m++);
17          S[0]=m-1;
18      }
19
20      /*****------------------------------*****/
21      // 函数名:    StringCopy(SString &T,SString S)
22      // 参数:      (传出)SString T,字符串
23      //            (传入)SString S,字符串
24      // 作用:      串赋值(生成和 S 一样的字符串 T)
25      // 返回值:    无
26      /*****------------------------------*****/
27      int StringCopy(SString &T,SString S)
28      {
29          int i;
30          for(i=1;i<=S[0];i++)
31              T[i]=S[i];  //T[1...len]=S[1...len]
32          T[i]='\0';
33          T[0]=S[0];
34          return T[0];
35      }
36
37      void main()
38      {
39          SString T;
40          unsigned char S[MAXSTRLEN+1];//或者 S 定义为   SString S;
41          printf("Please input the main String:");
42          scanf("%s",S+1);
43
44          //得到串的长度,放于下标为 0 的位置
45          strLengh(S);
46
47          printf("\nThe copy string T is: %s(length=%d)\n",T+1,String-
            Copy(T,S));
48      }
```

第 4 步：测试数据。

运行结果：

```
Please input the main String:
ILoveCoding
The copy string T is:
```

```
ILoveCoding(length=11)
```

思考题：

根据教材提供的算法 4.3 实现动态顺序存储串的字符串复制函数

【实验 3.2】　求子串实验

编程实现求字符串的子串。

第 1 步：任务分析。

静态存储的字符串求子串问题的程序实现关键在于如何在主串中查找子串，具体实现是字符数组的基本操作。

第 2 步：程序构思。

⑴ 从 pos 位置开始取串 S 放到新串 Sub 中；

⑵ 手工添加字符串结束标记'h0'。

第 3 步：源程序。

```
1       //调试环境：Visual C++6.0
2       #include <stdio.h>
3       //-----库文件和预设定义
4       #define MAXSTRLEN 255
5       typedef unsigned char SString[MAXSTRLEN+1];
6       /*****------------------------*****/
7       // 函数名：    strLengh(SString S)
8       // 参数：      (传入)SString S,字符串
9       // 作用：      得到串的长度,放于下标为 0 的位置
10      // 返回值：    无
11      /*****------------------------*****/
12      void strLengh(SString S)
13      {
14          int m;
15          for(m=1;S[m]!='\0';m++);
16          S[0]=m-1;
17      }
18
19      /*****------------------------*****/
20      // 函数名：    SubString(SString &Sub,SString S,int pos,int len)
21      // 参数：      (传出)SString Sub,子字符串
22      //             (传入)SString S,字符串
23      //             (传入)int pos,字符串中子字符串起始位置
24      //             (传入)int len,长度字符串
25      // 作用：      求子串(寻找串 S 的子串,位置从 S 的 pos 个字符开始,长度为 len 的子
                       串)
26      // 返回值：    无
27      /*****------------------------*****/
```

```
int SubString(SString &Sub,SString S,int pos,int len)
{
    int i;
    if(pos<1 ||pos>S[0] ||len<0 ||len>S[0]-pos+1)
    {
        printf("Error!positon or length is out of range\n");
        return 0;
    }
    //从 pos 位置开始在串 S 中取子字符串放到新串 Sub 中
    for(i=1;i<=len;i++)
        Sub[i]=S[pos+i-1];

    Sub[i]='\0';          //手工添加字符串结束标记'\0'
    Sub[0]=len;           //在 Sub[0]处存放字符串长度
    return 1;
}

void main()
{
    SString Sub;
    int pos,len;
    unsigned char S[MAXSTRLEN+1];//或者 S 定义为 SString S;
    printf("Please input the main String:\n");
    scanf("%s",S+1);
    strLengh(S);
    printf("Please input the position of SubString:");
    scanf("%d",&pos);
    printf("Please input the length of SubString:");
    scanf("%d",&len);
    if(SubString(Sub,S,pos,len))
        printf("\nThe SubString is: %s\nSubString.length=%d\n",
        Sub+1,Sub[0]);
}
```

第4步：测试数据。

运行结果：

```
Please input the main String:
codingisok
Please input the position of SubString:2
Please input the length of SubString:3
The SubString is: odi
SubString.length=3
```

思考题：

根据教材提供的算法 4.6 实现动态顺序存储串的字符串求子串函数

【实验 3.3】　字符串连接实验

编程实现字符串连接程序。

第 1 步：任务分析。

静态存储的字符串求连接问题的程序实现关键在于如何把要连接串连在主串之后，具体实现的关键在于字符串长度的处理。

第 2 步：程序构思。

⑴ 将串接在源串后面；

⑵ 给新串加上结束符。

参见具体程序请考虑串没有被截断和分别被截断的情况。

第 3 步：源程序。

```
1       //调试环境：Visual C ++ 6.0
2       #include <stdio.h>
3       //-----库文件和预设定义
4       #define MAXSTRLEN 255
5       typedef unsigned char SString[MAXSTRLEN+1];
6
7       /*****------------------------------*****/
8       // 函数名：    Concat(SString &T,SString S1,SString S2)
9       // 参数：      (传出)SString T,新字符串
10      //             (传入)SString S1,字符串 1
11      //             (传入)SString S2,字符串 2
12      // 作用：      字符串连接(将字符串 S1 和 S2 连接成新字符串 T)
13      // 返回值：    返回 1 表示串没有被截断,0 表示串被截断
14      /*****------------------------------*****/
15      int Concat(SString &T,SString S1,SString S2)
16      {
17
18          int i,k;
19          int uncut;    //标记字符串是否被截断
20
21          //没有截断
22          if (S1[0]+S2[0]<=MAXSTRLEN)
23          {
24              //先复制串 S1
25              for(i=1;i<=S1[0];i++)
26                  T[i]=S1[i];
27              //将串 S2 接在后面
28              for(k=1,i=S1[0]+1;i<=S1[0]+S2[0];i++)
```

```
29                T[i]=S2[k++];
30
31            T[i]='\0';  //给新串加上结束符
32            T[0]=S1[0]+S2[0];  //计算新串的长度
33            uncut=1;
34        }
35
36        //串 S2 将被截断
37        else if(S1[0]<MAXSTRLEN)
38        {
39            for(i=1;i<=S1[0];i++)    //先把 S1 放入 T 中
40                T[i]=S1[i];
41            for(k=1,i=S1[0]+1;i<=MAXSTRLEN;i++) // S2 截断,将 S2 部分放入
              T 中
42                T[i]=S2[k++];
43            T[i]='\0';    //添加字符串结束标记
44            T[0]=MAXSTRLEN;
45            uncut=0;
46        }
47        //T[0]==S[0]==MAXSTRLEN
48        else if(S1[0]=MAXSTRLEN)
49        {
50            for(i=0;i<=MAXSTRLEN;i++)    //把 S1 放入 T 中
51                T[i]=S1[i];
52            T[i]='\0';    //添加字符串结束标记
53            uncut=0;
54        }
55        //S1[0]>MAXSTRLEN,串 S1 被截断
56        else
57        {
58            for(i=1;i<=MAXSTRLEN;i++)    // S1 截断,将 S1 部分放入 T 中
59                T[i]=S1[i];
60            T[i]='\0';        //添加字符串结束标记
61            T[0]=MAXSTRLEN;
62            uncut=0;
63        }
64
65        return (uncut);
66    }
67    /*****------------------------    *****/
68    // 函数名:    strLengh(SString S)
69    // 参数:      (传入)SString S,字符串
70    // 作用:      得到串的长度,放于下标为 0 的位置
```

```
71      // 返回值:    无
72      /*****----------------------------*****/
73      void strLengh(SString S)
74      {
75          int m;
76          for(m=1;S[m]!='\0';m++);
77          S[0]=m-1;
78      }
79
80      void main()
81      {
82          SString T;
83          int j=0,k=0;
84          SString S1,S2;
85          printf("Please input the String S1>: ");
86          scanf("%s",S1+1);    //跳过下标为0的元素
87          printf("Please input the String S2: ");
88          scanf("%s",S2+1);    //跳过下标为0的元素
89
90          //得到两个串的长度,放于下标为0的位置
91          strLengh(S1);
92          strLengh(S2);
93
94          /* 或使用计算长度的函数strlen */
95          /* strlen的参数是char*,我们定义的S1和S2是unsigned char,故不能
            如下使用 */
96          /* S1[0]=strlen(S1+1); */
97          /* S2[0]=strlen(S2+1); */
98
99          Concat(T,S1,S2);
100         //输出子串和新串
101         printf("\nS1=%s\n",S1+1);
102         printf("S2=%s\n",S2+1);
103         printf("T=%s\n\n",T+1);
104         printf("S1[0]=%u\n",S1[0]);
105         printf("S2[0]=%u\n",S2[0]);
106         printf("Length of New string T is: %u \n",T[0]);
107
108         //输出新字符串是否被截断的信息
109         if(Concat(T,S1,S2))
110             printf("字串没有被截断\n");
111         else printf("字串被截断\n");
112     }
```

第4步：测试数据

运行结果：

```
Please input the String S1 >: ilovecoding
Please input the String S2: verymuch
S1 =ilovecoding
S2 =verymuch
T =ilovecodingverymuch
S1[0] =11
S2[0] =8
Length of New string T is: 19
```

字串没有被截断

思考题：

根据教材提供的算法4.5实现动态顺序存储串的字符串连接函数

【实验3.4】　字符串模式匹配BF实验

编程实现字符串模式匹配程序。

第1步：任务分析。

通过字符串模式匹配程序理解布鲁特—福斯算法。

第2步：程序构思。

从主串S的第pos个字符起和模式的第一个字符相比较，若相等，则继续逐个比较后续字符；否则从主串的下一个字符起再重新和模式的字符比较。依次类推，直至模式T中的每个字符依次和主串S中的一个连续的字符序列相等，此时匹配成功，定位函数返回和模式T中第一个字符相等的字符在主串S中的序号。否则匹配不成功，定位函数返回零。

第3步：源程序。

```
1          //调试环境：Visual C++6.0
2          #include <stdio.h>
3          //-----库文件和预设定义
4          #define MAXSTRLEN 255
5          typedef unsigned char SString[MAXSTRLEN+1];
6
7          /*****------------------------------*****/
8          // 函数名:    Index(SString S,SString T,int pos)
9          // 参数:      (传出)SString S,字符串
10         //            (传入)SString T,字符串
11         //            (传入)int pos,定位初试位置
12         // 作用:      求子串位置
13         // 返回值:    整型,定位成功返回位置,否则返回0
14         /*****------------------------------*****/
```

```
15      int Index(SString S,SString T,int pos)
16      {
17          int i =pos,j =1;
18          while(i < =S[0] && j < =T[0])  //在 S 和 T 的长度之内开始查找
19          {
20              if(S[i] ==T[j])    //若相等,继续比较
21              {
22                  ++i;
23                  ++j;
24              }
25              else    //    否则从 S 的下一个字符开始重新和 T 的字符进行比较
26              {
27                 i =i -j +2;
28                 j =1;   //指针后退重新开始匹配
29              }
30          }
31
32          if(j >T[0]) return i -T[0];
33          else return 0;
34      }
35
36      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
37      // 函数名:    strLengh(SString S)
38      // 参数:      (传入)SString S,字符串
39      // 作用:      得到串的长度,放于下标为 0 的位置
40      // 返回值:    无
41      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
42      void strLengh(SString S)
43      {
44          int m;
45          for(m =1;S[m]!='\0';m ++);
46          S[0] =m -1;
47      }
48
49      void main()
50      {
51          SString S,T;
52          int pos;  //  1 < =pos < =S[0] -T[0] +1;
53          int r;
54          printf("输入主串 S: ");
55          scanf("%s",S +1);    //跳过下标为 0 的元素
56          printf("输入模式串 T: ");
57          scanf("%s",T +1);    //跳过下标为 0 的元素
```

```
58            printf("输入起始位置 pos: ");
59            scanf("%d",&pos);
60
61            //得到两个串的长度,放于下标为 0 的位置
62            strLengh(S);
63            strLengh(T);
64
65            if(r = Index(S,T,pos))
66                printf("模式串在主串中的位置为: %d\n",r);
67            else printf("匹配失败!");
68        }
```

第 4 步：测试数据。

运行结果：

```
输入主串 S: ilovecoding
输入模式串 T: coding
输入起始位置 pos: 1
模式串在主串中的位置为: 6
```

思考题：

1. **请利用串的五种基本操作（串赋值、求串长、串连接、求子串、串比较）组合完成实现串的定位函数**

2. **根据教材提供的算法 4.5 实现动态顺序存储串的字符串定位函数**

【实验 3.5】　字符串模式匹配 KMP 实验

编程实现字符串模式匹配程序。

第 1 步：任务分析。

在上节字符串模式匹配实验的基础上用克努特 - 莫里斯 - 普拉特操作，即改进型的模式匹配 KMP 算法实现字符串的模式匹配。

第 2 步：程序构思。

KMP 算法的改进在于：每当一趟匹配过程中出现字符比较不相等的时候，不需要回溯 i 指针，而是利用已经得到的“部分匹配”的结果将模式向右“滑动”尽可能远的一段距离后，继续进行比较。具体实现过程描述如下。

⑴ 假设以指针 i 和 j 分别指示主串 S 和模式 T 中正待比较的字符，并令 i 的初值为 pos，j 的初值为 1。

⑵ 如果在匹配的过程中 S [i] = T [j]，i 和 j 分别增 1；否则，i 不变，而 j 退到 next [j] 的位置再比较。

⑶ 再比较，若相等，则指针各自增 1，否则 j 再退到下一个 next [j] 的位置上，依次类推，直至出现下面两种可能：

- 一种可能是 j 退到某个 next 值（next [next […next [j]]]）时字符比较相等，则指针各自增 1 继续进行匹配。

- **另一种可能是 j 退到 next [j] 为 0 (即模式 T 的第一个字符“失配”),则此事需要将模式 T 继续向右移动一个位置,即从主串的下一个字符 S [i + 1] 起和模式 T 重新开始匹配。**

第 3 步:源程序。

```
//调试环境:Visual C ++6.0
#include <stdio.h>
//- - - - -库文件和预设定义
#define MAXSTRLEN 255
typedef unsigned char SString[MAXSTRLEN +1];

/* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
// 函数名:    strLengh(SString S)
// 参数:      (传入)SString S,字符串
// 作用:      得到串的长度,放于下标为 0 的位置
// 返回值:    无
/* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
void strLength(SString S)
{
    int m;
    for(m =1;S[m]!='\0';m ++);
    S[0] =m -1;
}

/* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
// 函数名:    get_next(SString T, int *next)
// 参数:      (传入)SString T,子字符串
//            (传出) int *next,模式 T 的 next 函数
// 作用:      求模式 T 的 next 函数值并存入数组 next
// 注意:      串 S 和 T 采用定长顺序存储,且 T 非空
// 返回值:    无
/* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
void get_next(SString T,int *next)
{
    //求模式 T 的 next 函数值并存入数组 next,为后面进行模式匹配做准备
    int j,k;
    j =1;
    next[1] = -1;
    k = -1;          //设置初值

    while (j < T[0])
    {
        if (k == -1 ||T[j] ==T[k])
```

```
39              {
40                  //若 k = -1 或 T[j] ==T[k],令 next[j +1] = k +1
41                  ++j; ++k;
42                  next[j] = k;
43              }
44              else next[j] = k;    //若 k 大于 0 或 T[j]不等于 T[k],令 k = next
                [k]
45          }
46      }
47
48      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
49      // 函数名:    Index_KMP(SString S,SString T,int pos,int next[])
50      // 参数:      (传出)SString S,主字符串
51      //            (传入)SString T,子字符串
52      //            (传入)int pos,定位初试位置
53      //            (传入)int next[],模式 T 的 next 函数
54      // 作用:      利用模式 T 的 next 函数求模式 T 在主串 S 中第 pos 个字符后的位置
55      // 注意:      串 S 和 T 采用定长顺序存储,且 T 非空
56      // 返回值:    整型,定位成功返回位置,否则返回 0
57      /* * * * - - - - - - - - - - - - - - - - - - - - - -   * * * * */
58      int Index_KMP(SString S,SString T,int pos,int *next)
59      {
60          int i,j;
61          i =pos;  j =1;     //设置初值
62
63          get_next(T,next);    //计算模式串 T 的 next 数组
64
65          while (i <S[0] && j <T[0])
66          {
67              if (j == -1 ||S[i] ==T[j])//继续比较后继字符
68              {
69                  ++i;
70                  ++j;
71              }
72              else j =next[j];        //模式向右移动
73          }//end_while
74          if (j >=T[0]) return (i -T[0]);    //匹配成功
75          else return 0;             //匹配不成功
76      }
77
78      void main()
79      {
80          SString S,T;
```

```
81          int pos;  //  1 <=pos <=S[0] -T[0] +1;
82          int next[MAXSTRLEN];
83          int *p =&next[1];
84          int r;
85          printf("输入主串 S: ");
86          scanf("%s",S +1);    //跳过下标为 0 的元素
87          printf("输入模式串 T: ");
88          scanf("%s",T +1);    //跳过下标为 0 的元素
89          printf("输入起始位置 pos: ");
90          scanf("%d",&pos);
91
92          //得到两个串的长度,放于下标为 0 的位置
93          strLength(S);    //求主串 S 的长度
94          strLength(T);    //求模式 T 的长度
95
96          if(r =Index_KMP(S,T,pos,p))
97              printf("模式串在主串中的位置为: %d\n",r);
98          else printf("匹配失败!");
99      }
```

第 4 步：测试数据。

运行结果：

```
输入主串 S: ilovecoding
输入模式串 T: coding
输入起始位置 pos: 1
模式串在主串中的位置为: 6
```

3.3　串的应用

【实验 3.6】　串查找与替换

根据串的基本操作完成串查找和替换操作。

第 1 步：任务分析。

串查找与替换的实现，是根据串的基本操作组合完成的，故熟练掌握串的基本操作是实现串查找与替换的前提，程序构思省略，请读者自行思考。

第 2 步：源程序。

```
1           #include <stdio.h>
2           #define MAXSTRLEN 255
3           typedef unsigned char SString[MAXSTRLEN +1];
4           /*得到串的长度*/
```

```
int strLengh(SString S)
{
    int m;
    for(m=0;S[m]!='\0';m++);
      return m;
}
/*字符串复制*/
void StringCopy(SString &T,SString S)
{
      int i;
      for(i=1;i<=strLengh(S);i++)
      T[i]=S[i];  //T[1...len]=S[1...len]
      T[i]='\0';
}

/*****------------------------------*****/
//函数名:   Replace(SString &s,SString s1,SString s2)
//参数:     (传入)SString s,字符串
//          (传入)SString s1,待查找的字符串
//          (传入)SString s2,替换字符串
//作用:     字符串查找与替换
//返回值:   无
/*****------------------------------*****/
void Replace(SString &s,SString s1,SString s2)
{
    SString t;
    int i,j,f,k=0;
    if(strLengh(s) >=strLengh(s1))
    {
        for(i=0;i<=strLengh(s);)
        {
            if(i <=strLengh(s)-strLengh(s1))
            {
                /*查找*/
                f=1;
                for(j=0;j<strLengh(s1);j++)
                    if(s[i+j]!=s1[j])
                    {
                        f=0;
                        break;
                    }
```

```
46                  }
47                  else f =0;
48                  if(f ==1)
49                  {
50                      /* 替换 */
51                      for(j =0;j <strLengh(s2);j ++,k ++)
52                          t[k] =s2[j];
53                      i =i +strLengh(s1);
54                  }
55                  else
56                  {
57                      /* 复制一个字符 */
58                      t[k] =s[i];
59                      k ++;
60                      i ++;
61                  }
62
63              }//for
64          }//if
65          StringCopy(s,t);/* 复制字符串 */
66      }
67
68      void main()
69      {
70          SString str;
71          SString temp,substr;
72          printf("请输入主串 :");
73          scanf("%s",str);
74          printf("请输入查找的字符串 :");
75          scanf("%s",temp);
76          printf("请输入替换的字符串 :");
77          scanf("%s",substr);
78          Replace(str,temp,substr);
79          printf("新字符串为 :");
80          printf("%s\n",str);
81      }
```

第 3 步：测试数据。

运行结果：

请输入主串：abcdef
请输入查找的字符串：d

请输入替换的字符串：xyz123
新字符串为：abcxyz123ef

思考题：

查找替换功能的实现请读者参考串的匹配、复制等基本操作。

3.4 小　结

串的顺序存储结构类似于线性表的顺序存储结构，串的链式存储结构采用的是链表方式存储串值字符序列。根据常见的串的两种存储表示，串连接、求子串、子串位置定位等基本运算就易于实现。此外，串的模式匹配算法也是学习的一个重点。

第 4 章　数　　组

本章的典型实验主要练习数组结构的实际应用，特别是多维数组的存储以及特殊矩阵的存储共 3 个实验内容。

4.1　知识点概述

数组是大家非常熟悉的一种数据结构，大多数程序设计语言都把数组定为固有类型。而多维数组和广义表是对线性表的推广，它们在逻辑结构、存储结构、表示形式、实际应用等方面都有各自不同的特点。数组的存储结构有顺序存储结构和数组的压缩存储技术，常见的有三元组顺序表和十字链表。

4.2　数组结构应用实验

【实验 4.1】　矩阵运算的设计与实现

矩阵基本运算的实现。

第 1 步：任务分析。

应用数组结构实现矩阵的转置和加减乘等运算。

第 2 步：程序构思。

对于 $m \times n$ 的矩阵 $A = \{a(i,j) \mid 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n\}$ 和 $B = \{a(i,j) \mid 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n\}$，它们的加减结果矩阵 C，其元素 $c(i,j) = a(i,j) \pm b(i,j)$。

$A * B = C$（其中 A 为 $m \times n$ 矩阵，B 为 $n \times t$ 矩阵），其结果矩阵 C 的元素 $c(i,j) = \sum_{k=1}^{n} a(i,k) * b(k,j)$（$1 \leqslant k \leqslant n, 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant t$）。

对于 $m \times n$ 的矩阵 $A = \{a(i,j) \mid 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant m\}$，它的转置矩阵 C 是一个 $n \times m$ 的矩阵，且 $C(i,j) = A(j,i)$。

第 3 步：源程序。

```
#include <stdio.h>
#define M 4
/*****------------------------------*****/
// 函数名:    MatrixAdd(int m1[M][M],intm2[M][M],int &result[M][M])
// 参数:      (传入)int m1[M][M],m1[M][M] 矩阵 m1 和 m2
//            (传入)int result[M][M],矩阵加计算结果
// 作用:      两矩阵相加
```

```
8       // 备注:      计算公式为 result[i][j] =m1[i][j] + m2[i][j]
9       /* * * * - - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
10      void MatrixAdd(int m1[M][M],int  m2[M][M],int &result[M][M])
11      {
12          int i,j;
13          for (i =0;i <4;i ++)
14          {
15              for (j =0;j <4;j ++)
16                  result[i][j] =m1[i][j] + m2[i][j];
17          }
18      }
19      void MatrixTrams(int m1[M][M],int &result[M][M])
20      {      //将矩阵转置
21          int i,j;
22          for (i =0;i <4;i ++)
23          {
24              for (j =0;j <4;j ++)
25                  result[i][j] =m1[j][i];
26          }
27      }
28      /* * * * - - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
29      // 函数名:    MatrixPlus(int m1[M][M],int m2[M][M],int result[M]
                       [M])
30      // 参数:      (传入)int m1[M][M],m1[M][M] 矩阵 m1 和 m2
31      //            (传入)int result[M][M],矩阵乘计算结果
32      // 作用:      两矩阵相乘
33      // 备注:      计算公式为 result[i][j] + =m1[i][k] * m2[k][j]
34      /* * * * - - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
35      void MatrixPlus(int m1[M][M],int  m2[M][M],int result[M][M])
36      {
37          int i,j;
38          for (i =0;i <4;i ++)
39          {
40              for (j =0;j <4;j ++)
41              {
42                  result[i][j] =0;
43                  for(int k =0;k <4;k ++)
44                      result[i][j] + =m1[i][k] * m2[k][j];
45              }
46          }
47      }
48      /* 输出矩阵值 */
49      void Display(int result[M][M])
```

```
    {
        int i,j;
        printf("the operating result of Matrix:\n");
        for(i =0;i <M;i ++)
        {
            for(j =0;j <M;j ++)
                printf("%d ",result[i][j]);
            printf("\n");
        }
    }
    void main()
    {
        int A[M][M];
        int B[M][M];
        int i,j;
        for(i =0;i <M;i ++)
            for(j =0;j <M;j ++)
                scanf("%d",&A[i][j]);
        for(i =0;i <M;i ++)
            for(j =0;j <M;j ++)
                scanf("%d",&B[i][j]);

        int result[M][M];
        MatrixAdd(A,B,result);
        Display(result);
        MatrixPlus(A,B,result);
        Display(result);
    }
```

第 4 步：测试数据。

运行结果：

```
input the matrix:
1 1 1 1
2 2 2 2
1 1 1 1
2 2 2 2
input the other matrix:
2 2 2 2
1 1 1 1
2 2 2 2
1 1 1 1
the add operating result of Matrix:
3 3 3 3
```

```
3 3 3 3
3 3 3 3
3 3 3 3
the plus operating result of Matrix:
6   6   6   6
12 12  12  12
6   6   6   6
12 12  12  12
```

思考题：

请按照实验所示思路和程序，完成矩阵的其他操作。

【实验 4.2】　矩阵排序实验

根据矩阵的存储形式，完成实现矩阵排序。

第 1 步：任务分析。

对于一个 m × n 的矩阵，使其每一行元素按非减的次序排列。

第 2 步：程序构思。

矩阵在存储中常用二维数组，对于一些特殊的矩阵，一般按其规律压缩存储在一维向量中。本实验的 m × n 的矩阵按一般情况处理，故采用二维数组的存储形式。矩阵的每一行视作一个一维数组，对其排序采用冒泡排序算法，冒泡排序设计实现请参考第 8 章的实验 8.3。

第 3 步：源程序。

```
#include <stdio.h>
#define M 4

/*****------------------------------------*****/
// 函数名：    BubblesSort(int list[],int n)
// 参数：      (传入) int list[],待排序数组
//             (传入) int n,数组长度
// 功能：      使用冒泡排序对数据进行排序
/*****------------------------------------*****/
void BubblesSort(int list[],int n)
{
    int i,j ,k;
    for(i =0;i <n -1;i ++)
    {
        for(j =n -1;j >i;j - -)
        {
            if (list[j] < list[j -1])//比较前后大小
            {
                //两数进行交换
                k =list[j];
```

```
                    list[j] = list[j -1];
                    list[j -1] = k;
                }
            }
        }
    }

    /* 输出矩阵值 */
    void Display(int result[M][M])
    {
        int i,j;
        printf("the operating result of Matrix:\n");
        for(i =0;i <M;i ++)
        {
            for(j =0;j <M;j ++)
                printf("%d ",result[i][j]);
            printf("\n");
        }
    }

    void main()
    {
        int A[M][M];
        int i,j;
        printf("input the data of matrix:\n");
        for(i =0;i <M;i ++)
            for(j =0;j <M;j ++)
                scanf("%d",&A[i][j]);

        for(i =0;i <M;i ++)
            BubblesSort(A[i],M);
        Display(A);
    }
```

第 4 步：测试数据。

运行结果：

```
input the data of matrix:
1 2 3 4
24 54 3 2
2 3 4 1
5 4 56 3
the operating result of Matrix:
1 2 3 4
```

```
2 3 24 54
1 2 3 4
3 4 5 56
```

【实验 4.3】　稀疏矩阵运算的设计与实现

稀疏矩阵运算的实现。

第 1 步：任务分析。

在上节矩阵基本运算的基础上，实现稀疏矩阵的基本操作。

第 2 步：程序构思。

按照压缩存储的概念，只存储稀疏矩阵的非零元素。由于稀疏矩阵的非零元素分布无规律，因此，为了实现矩阵的各种运算，除了存储非零元素的值之外，还必须同时记下它所在行和列的位置 (i,j) 。这样，稀疏矩阵中的每一个非零元素 a_{ij} 就用三部分表示：元素的值 a_{ij}，行号 i 和列号 j，这三部分信息组织在一起 (i,j,a_{ij}) ——称为三元组。这个三元组唯一确定了矩阵 A 的一个非零元素。由此，稀疏矩阵可由其行列数及表示非零元素的三元组唯一确定。

第 3 步：源程序。

```
#include <stdio.h>
#include <string.h>
#define OK 1
#define MAX 10 //用户自定义三元组最大长度

//定义三元组表
typedef struct
{
    int i,j;    //非零元素的行下标和列下标
    int v;      //非零元素的值
}TriTupleNode;

typedef struct
{
    TriTupleNode data[MAX];    ////非零元素三元组表
    int m;    //矩阵行数
    int n;    //矩阵列数
    int t;    //三元组表长度(非零元素个数)
}TSMatrix;

/*****------------------------------*****/
// 函数名:    InitTriTupleNode (TSMatrix *a)
// 参数:      (传入)TSMatrix *a 稀疏矩阵指针
// 作用:      建立稀疏矩阵的三元组表
/*****------------------------------*****/
```

```
26      void InitTriTupleNode (TSMatrix *a)
27      {
28          int i,j,k,val,maxrow,maxcol;
29          maxrow =0;
30          maxcol =0;
31          k =1;
32          while(i!= -1 && j!= -1) /*rol = -1&&col = -1 结束输入*/
33          {
34              printf("input row col val: ");
35              scanf("%d %d %d",&i,&j,&val);
36
37              //存储非零元素的位置和值
38              a ->data[k].i =i;
39              a ->data[k].j =j;
40              a ->data[k].v =val;
41              if (maxrow <i) maxrow =i;
42              if (maxcol <j) maxcol =j;
43              k ++;
44          }
45          //设置矩阵 a 的行列参数以及非零元素个数
46          a ->m =maxrow;
47          a ->n =maxcol;
48          a ->t =k;
49      }
50      /*****------------------------------------*****/
51      // 函数名:    ShowMatrix(TSMatrix *a)
52      // 参数:      (传入)TSMatrix *a 稀疏矩阵指针
53      // 作用:      输出稀疏矩阵
54      /*****------------------------------------*****/
55      void ShowMatrix(TSMatrix *a)
56      {
57          int p,q;
58          int t =1;
59          for(p =1;p <=a ->m;p ++)
60          {
61              for(q =1;q <=a ->n;q ++)
62                  if (a ->data[t].i ==p && a ->data[t].j ==q) //打印非零元素
63                  {
64                      printf("%d  ",a ->data[t].v);
65                      t ++;
66                  }
67                  else printf("0  ");    //否则打印输出 0
68              printf("\n"  );
```

```
69          }//for
70      }
71
72      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
73      // 函数名:    TransposeSMatrix(TSMatrix *a,TSMatrix *b)
74      // 参数:      (传入)TSMatrix *a,*b 稀疏矩阵 a,转置矩阵 b
75      // 作用:      转置稀疏矩阵
76      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
77      void TransposeSMatrix(TSMatrix *a,TSMatrix *b)
78      {
79          int q,col,p;
80          //设置矩阵 a 的转置矩阵 b 的行列参数以及非零元素个数
81          b->m=a->n;
82          b->n=a->m;
83          b->t=a->t;
84          if(b->t)
85          {
86              q=1;
87              for(col=1;col<=a->n;col++)    //从 a 的第一行开始起扫描
88                  for(p=0;p<=a->t;p++)
89                      if(a->data[p].j==col)
90                      {
91                          //将每个三元组中的 i 和 j 相互调换
92                          b->data[q].i=a->data[p].j;
93                          b->data[q].j=a->data[p].i;
94                          b->data[q].v=a->data[p].v;
95                          ++q;
96                      }
97          }//if
98      }
99      void main()
100     {
101         TSMatrix a,b;
102         TSMatrix *pa=&a,*pb=&b;
103         InitTriTupleNode(pa);
104         printf("稀疏矩阵转置前:\n");
105         ShowMatrix(pa);
106         printf("稀疏矩阵转置后:\n");
107         TransposeSMatrix(pa,pb);
108         ShowMatrix(pb);
109     }
```

第 4 步：测试数据。

运行结果：

```
input row col val: 1 1 1
input row col val: 1 2 2
input row col val: 2 1 4
input row col val: 2 2 3
input row col val: 2 4 7
input row col val: 3 4 8
input row col val: -1 -1 0
```

稀疏矩阵转置前：

```
1   2   0   0
4   3   0   7
0   0   0   8
```

稀疏矩阵转置后：

```
1   4   0
2   3   0
0   0   0
0   7   8
```

思考题：

1. **参考教材关于稀疏矩阵的快速转置算法** 5.3**，实现稀疏矩阵的快速转置。**

2. **参照实验** 4.1 **矩阵运算和实验二稀疏矩阵运算实现两个稀疏矩阵的相加运算函数。**

3. **参考本实验稀疏矩阵的存储方法结合教材建立十字链表的算法** 5.4**，实现稀疏矩阵的十字链表存储表示。**

4.3 小　　结

通过对本章的实验学习，掌握数组的存储方法和存取特点，了解掌握特殊矩阵和稀疏矩阵的存储，掌握广义表的表示形式等，并对数组的压缩存储技术——三元组顺序表和十字链表有所了解。

第 5 章　树和二叉树

本章的典型实验主要练习二叉树的存储实现及二叉树的基本操作，以哈夫曼编码的设计与实现进一步掌握二叉树的实际应用。典型实验分为二叉树结构实验和二叉树应用实验两类，共 6 个实验内容。

5.1　知识点概述

1. 定义

树 (Tree) 形结构是一类很重要的非线性数据结构。树是 n (n≥0) 个结点的有限集。在任意一棵非空树中：有且仅有一个特定的称为根 (Root) 的结点，该结点没有前驱；当 n > 1 时，其余结点可分为 m (m > 0) 个互不相交的有限集 T_1，T_2，…，T_m，其中每一个集合本身又是一棵树，并且称为根的子树 (Subtree)。树的基本操作有：初始化操作、求根函数、求双亲函数、求孩子结点函数、求深度函数、求结点值函数、结点赋值函数、插入子树操作、删除子树操作、遍历操作等。

二叉树 (Binary Tree) 又称二分树或二元树，是另一种重要的树形结构。二叉树是 n (n≥0) 个结点的有限集，它或为空树 (n = 0)，或由一个根结点和两棵分别称为左子树和右子树的互不相交的二叉树 (它们也是结点的集合) 构成。二叉树是一种度小于 2 的有序树，它的特点是每个结点至多只有二棵子树，并且二叉树的子树有左右之分，其次序不能任意颠倒。二叉树有 5 种基本形态：空二叉树、仅有根结点的二叉树、右子树为空的二叉树、左子树为空的二叉树和左右子树均非空的二叉树。

如果一棵二叉树的任何结点，或者是树叶，或者恰有两棵非空子树，则此二叉树称为满二叉树。这种树的特点是每一层上结点的个数都达到了最大。因此满二叉树有如下性质：叶结点的个数等于分支结点数加 1。

如果一棵二叉树最多只有最下面的两层结点的度数小于等于 2，并且最下面一层的结点都集中在该层最左边的若干位置上，则该二叉树称为完全二叉树。一棵满二叉树必然是一棵完全二叉树，但一棵完全二叉树并不一定是一棵满二叉树。

2. 二叉树的性质

二叉树的基本性质有：

性质 1　二叉树的第 i 层上至多有 2^{i-1} 个结点 (i≥1)。

性质 2　深度为 h 的二叉树中至多含有 $2^h - 1$ 个结点。

性质 3　若在任意一棵二叉树中，有 n_0 个叶子结点，有 n_2 个度为 2 的结点，则必有

$$n_0 = n_2 + 1$$

性质 4　具有 n 个结点的完全二叉树深度为 $\lfloor \log_2^n \rfloor + 1$ (“$\lfloor X \rfloor$”表示不大于 X 的最大整数)。

性质 5　若对有 n 个结点的完全二叉树进行顺序编号 (1 ≤ i ≤ n)，那么，对于编号为

i (i≥1) 的结点：

① 当 i = 1 时，该结点为根，它无双亲结点。

② 当 i > 1 时，该结点的双亲结点的编号为 ⌊i/2⌋。

③ 若 2i≤n，则有编号为 2i 的左孩子，否则没有左孩子。

④ 若 (2i + 1) ≤n，则有编号为 2i + 1 的右孩子，否则没有右孩子。

3. 遍历

二叉树的遍历方式有三种，分别为 NLR：先序遍历；LNR：中序遍历；LRN：后序遍历。

树的遍历方式有：先序遍历和后序遍历。

4. 森林与二叉树转换

森林转换成二叉树的一般步骤：第一步加线，在各兄弟结点之间加一条连线 (对于森林而言，所有树的根结点可视为兄弟结点)。第二步断线，对每个结点，除了其最左边的一个孩子外，将该结点与其余孩子之间的连线断开。第三步旋转，从水平方向顺时针旋转 45 度。

二叉树转换成森林或树分为三步：第一步加线，若某结点 i 是其双亲结点的左孩子，则将该结点 i 的右孩子以及连续的沿着右孩子的右链不断搜索到的所有右孩子，都分别与结点 i 的双亲结点用线连接。第二步断线，将原二叉树中所有双亲结点与其右孩子及其右子孙结点的连线断开。第三步旋转，从水平方向逆时针旋转 45 度。

5. 存储结构

树的常用存储结构：①双亲存储法是一种顺序存储方法，该方法以一组连续的存储空间存储树的结点，同时在每个结点中附设一个“指示器”指示其在双亲“链表”中的位置；②孩子存储法，该方法在每个结点设置多个指针域，其中每一个指针指向一棵子树的根结点，结点一般结构有两种形式，定长结点型和不定长结点型；③孩子链表法，该方法是把每个结点的孩子结点排列起来，看成是一个线性表，且以单链表作存储结构，则 n 个结点就有 n 个孩子链表 (叶子的孩子链表为空表)，而 n 个头指针又组成一个线性表，为了便于查找，可采用顺序存储结构。

二叉树的存储结构与树的存储结构分为链式和顺序存储结构。

链式存储结构通常用具有两个指针域的链表作为二叉树的存储结构，其中每个结点由数据域 Data、左指针域 Llink 和右指针域 Rlink 组成。两个指针域分别指向该结点的左、右孩子。若某结点没有左孩子或右孩子，则对应的指针域为空。

顺序存储结构是把所有结点按照一定的次序顺序存储到一片连续的存储单元中。适当安排这些结点的线性序列，可以使结点在线性序列的相互位置反映出二叉树结构的部分信息，通常情况使用完全二叉树的顺序存储法。

6. 线索二叉树

含有 n 个结点的二叉树，若用二叉链表存储，在 2n 个指针域中只用到了 n - 1 个指针域，因此，利用这些空指针域来存储结点的线性前驱和后继的信息：若结点有左子树，则其 Llink 域指示其左孩子，否则令 Llink 域指示其直接前驱；若结点有右子树，则其 Rlink 域指示其右孩子，否则令域指示其直接后继。为表示结点有无子树，则在结点中还需要增加两个左标志域 LTag 和右标志域 RTag。

当 LTag = 0 时，表示 Llink 指示结点的左孩子。

当 LTag = 1 时，表示 Llink 指示结点的直接前驱。

当 RTag = 0 时，表示 Rlink 指示结点的右孩子。

当 RTag = 1 时，表示 Rlink 指示结点的直接后继。

以上述结点结构构成的二叉链表作为二叉树的存储结构称作线索链表，指向结点直接前驱和直接后继的指针称为线索，加上线索的二叉树即为线索二叉树，对二叉树以某种规则进行遍历将其变为线索二叉树的过程称为线索化。

对二叉树进行不同规则的遍历，得到的结点序列不同，由此产生的线索二叉树也不同，所以有先序线索二叉树、中序线索二叉树和后序线索二叉树之分。

7. 哈夫曼树

哈夫曼树（Huffman）又称最优树，是一类带权路径最短的树，是树形结构的典型应用实例之一。哈夫曼树定义：假设有 n 个权值 $\{W_1, W_2, \cdots, W_n\}$，可构造多个有 n 个叶子结点的二叉树，每个叶子结点带权 W_i，而其中带权路径长度 WPL 最小的树称为最优二叉树或哈夫曼树。

构造出哈夫曼树步骤：

⑴ 根据给定的 n 个权值 $\{W_1, W_2, \cdots, W_n\}$ 构成 n 棵二叉树的集合 $F = \{T_1, T_2, \cdots, T_n\}$，其中每棵二叉树中只有一个带权为 W_i 的根结点。

⑵ 在 F 中选择两棵根结点最小的树作为左、右子树构造一棵新的二叉树，且置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。

⑶ 将新的二叉树加入 F 中，去除原来两棵根结点权值最小的树。

⑷ 重复步骤 ⑵ 和 ⑶ 直到 F 中只含有一棵树为止，这棵树就是哈夫曼树。

5.2　二叉树结构实验

【实验 5.1】　数组存储二叉树实验

用数组的形式实现二叉树的存储。

第 1 步：任务分析。

用数组形式存储二叉树，关键在于把数组的下标形式和二叉树的节点对应起来。参见下述程序构思。

第 2 步：程序构思。

二叉树子结点表示：假设一父结点编号为 n，则左子结点为 2 * n，右子结点为 2 * n + 1。故数组表示的二叉树建立步骤如下。

⑴ 以第一个输入的元素作为二叉树的根结点。

⑵ 依次将输入元素与根结点做比较。

- if（元素 > 根结点）　该元素向根结点右子树移动，如右子树为空，则存储，否则就重复比较，直至找到空结点进行存储。
- else　元素向根结点左子树移动，如左子树为空，则存储，否则就重复比较，直至找到空结点进行存储。

数组存储二叉树的过程如图 5-1 所示。

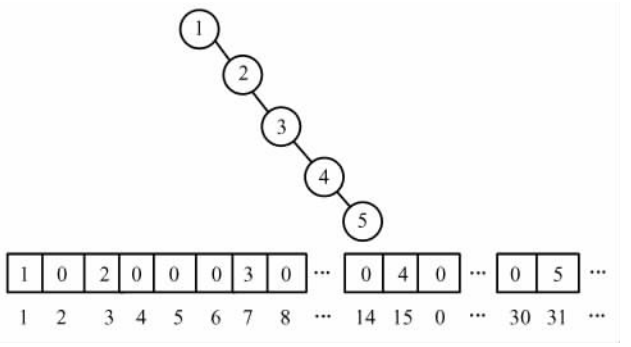


图 5-1　数组存储二叉树过程

第 3 步：源程序。

```
#include <stdio.h>
#define MAX 30

/*****------------------------------------*****/
// 函数名：  CreateTree(int btree[],int list[],int n)
// 参数：    (传入)int btree[]  二叉树
//           (传入)int list[]  数据数组
//           (传入)int n  数据数量
// 功能：    建立二叉树
/*****------------------------------------*****/
void CreateTree(int *bt,int list[],int n)
{
    int i;
    int level;

    for(i=0;i<MAX;i++)
        bt[i]=0;

    bt[0]=list[0];    //建立二叉树根结点
    for(i=1;i<n;i++)
    {
        level=1;    //从第一层开始建立
        while (bt[level]!=0)        //判断是否有子树存在
        {
            if (list[i] < bt[level])    //建立左子树
                level=level*2;
            else    level=level*2 + 1;    //建立右子树
        }
        bt[level]=list[i];    //结点赋值
    }
```

```
    }
    //测试主程序
    void main()
    {
        int count,i;
        int btree[MAX];
        int nodelist[MAX];

        printf("input the number of elements(n<50):\n");
        scanf("%d",&count);
        printf("input elements:\n");
        for(i=0;i<count;i++)
            scanf("%d",&nodelist[i]);
        CreateTree(btree,nodelist,count);
        printf("the binary tree is:\n");
        for(i=0;i<MAX;i++)
            printf("%d ",btree[i]);
        printf("\n");

    }
```

第 4 步：测试数据 (参见图 5-1)。

```
input the number of elements(n<50):
5
input elements:
1 2 3 4 5
the binary tree is:
1 2 0 3 0 0 0 4 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

思考题：

除了用数组的形式存储二叉树之外，由于二叉树结点最多只能有两个子结点，故可以定义结构体类型的存储方式实现二叉树的结构数组表示。请参考如上二叉树数组实现过程，完成二叉树的结构数组表示。

提示：

结构数组类型定义：

```
//二叉树结构
typedef struct btree
{
    int data;    //结点数据内容
    int left;    //存放左子树位置
    int right;   //存放右子树位置
}BtreeNode;
//定义二叉树结构数组
```

```
BtreeNode btree[MAX];
```

二叉树的结构数组表示参考函数：

```
1        void CreateTree(btree * bt,int list,int n)
2        {
3            int i;
4            int level;
5            int pos;    //左子树 -1,右子树 1
6
7            bt[0].data = list[0];    //建立根结点
8            for(i =1;i <n;i ++)
9            {
10               bt[i].data = list[i];    //元素存入结点
11               level =0;               //从根结点开始
12               pos =0;
13               while (pos ==0)        //查找结点位置
14               {
15                   if (bt[level].data < list[i])//是右子树
16                   {
17                       if (bt[level].right != -1)//是否有下一层
18                           level =bt[level].right;
19                       else
20                           pos = -1;//设为右子树
21                   }
22                   else//是左子树
23                   {
24                       if (bt[level].left != -1)//是否有下一层
25                           level =bt[level].left;
26                       else
27                           pos =1;//设为左子树
28                   }
29               }
30               //连接左右子树
31               if (pos ==1)
32                   bt[level].left = i;
33               else
34                   bt[level].right = i;
35           }
36       }
```

【实验 5.2】　链表存储二叉树实验

二叉树的链表存储实现。

第 1 步：任务分析。

二叉树的链表存储实现，基本操作包括二叉树的建立与释放、中序遍历、输出、测深度等。

第 2 步：程序构思。

链表存储二叉树通常用具有两个指针域的链表作为二叉树的存储结构，其中每个结点由数据域 Data、左指针域 Llink 和右指针域 Rlink 组成。两个指针域分别指向该结点的左、右孩子。若某结点没有左孩子或右孩子，则对应的指针域为空。最后，还需要一个链表的头指针指向根结点。

本次实验算法描述参见教材，注意二叉树建立、遍历函数的递归和非递归的分别实现，请先复习有关栈的操作。

第 3 步：源程序。

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 50
//二叉树链表存储结构
typedef struct btnode
{
    int  Data;    //结点数据内容
    struct btnode  * Llink;    //左子树指针
    struct btnode  * Rlink;    //右子树指针
}btnode, *btreetype;

/*****------------------------------------*****/
// 函数名：  CreateTree(int n)
// 参数：    (传入)int n   数据数量
// 返回值：  返回二叉树(根结点)指针
// 功能：    建立二叉树
/*****------------------------------------*****/
btreetype CreateTree(int n)
{
    int i;
    btreetype root = NULL;

    for(i = 0;i < n;i ++)
    {
        btreetype newNode;
        btreetype currentNode;
        btreetype parentNode;

        //建立新结点
        newNode = (btreetype)malloc(sizeof(btnode));
```

```
31                scanf("%d",&newNode->Data);/*新结点赋值*/
32                newNode->Llink=NULL;
33                newNode->Rlink=NULL;
34
35                currentNode=root;    //存储当前结点指针
36                if (currentNode==NULL) root=newNode;//新结点作为二叉树
                  根结点
37                else
38                {
39                    while (currentNode!=NULL)
40                    {
41                        parentNode=currentNode;    //存储当前结点的父结点
42                        if (newNode->Data < currentNode->Data) //比较结
                          点数值大小
43                            currentNode=currentNode->Llink;    //左子树
44                        else
45                            currentNode=currentNode->Rlink;    //右子树
46                    }
47                    //根据数值大小连接父结点和子结点
48                    if (newNode->Data < parentNode->Data)
49                        parentNode->Llink=newNode;
50                    else
51                        parentNode->Rlink=newNode;
52                }//else
53            }//for
54            return root;
55        }
56
57        /*****------------------------*****/
58        // 函数名：  OutputTree(btreetype &root)
59        // 参数：    (传入)btreetype &root  二叉树指针
60        // 功能：    输出二叉树
61        /*****------------------------*****/
62        void OutputTree(btreetype &root)
63        {
64            btreetype p;
65
66            //打印左子树
67            p=root->Llink;
68            printf("建立的二叉树的左子树为:");
69            while (p!=NULL)
70            {
71                  printf("%d ",p->Data);
```

```
72                  p=p->Llink;
73          }
74          //打印右子树
75          p=root->Rlink;
76          printf("\n建立的二叉树的右子树为:");
77          while (p!=NULL)
78          {
79                  printf("%d ",p->Data);
80                  p=p->Rlink;
81          }
82      }
83
84      //测试主程序
85      void main()
86      {
87          btreetype btree;
88          int count;
89          printf("input the number of elements:\n");
90          scanf("%d",&count);
91          printf("input data(num=%d):\n",count);
92          btree=CreateTree(9);
93          OutputTree(btree);
94      }
```

第 4 步：测试数据。

当测试数据为 45,2,68,1,23,5,12,9,6 时,所建立的二叉排序树的过程如图 5-2 所示。

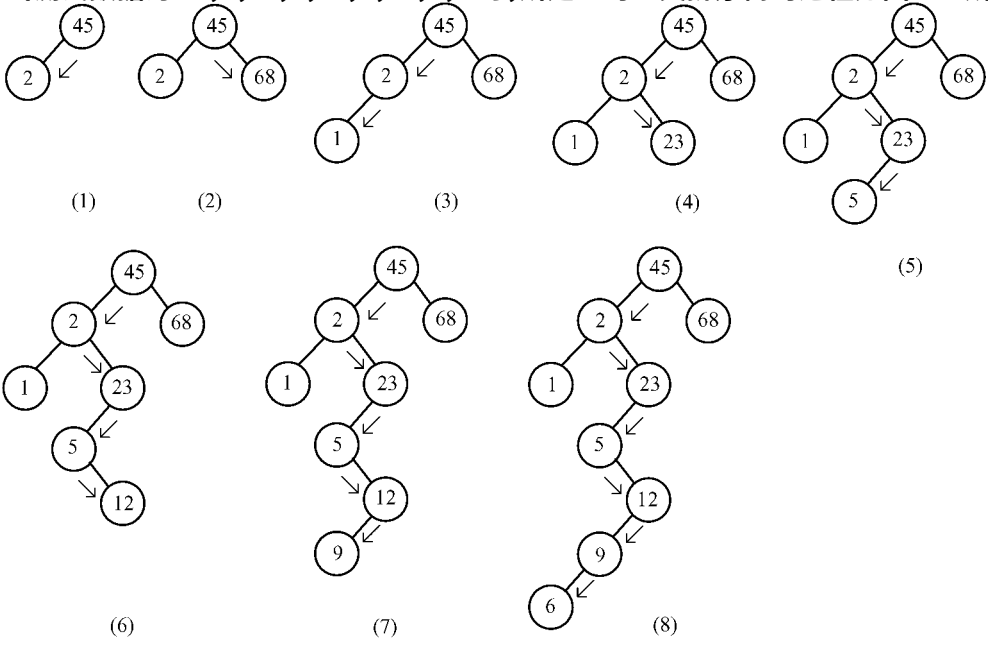


图 5-2　链表结构存储二叉树过程

运行结果：

```
input the number of elements:
9
input data(num=9):
45 2 68 1 23 5 12 9 6
建立的二叉树的左子树为:2 1
建立的二叉树的右子树为:68
```

思考题：

1. 结合数组存储二叉树的知识，考虑如何将二叉树的数组结构转换成链表结构（提示：用递归法进行建立）。

2. 请对比上述输出函数 OutputTree（）输出二叉树节点值和实际二叉树节点值情况，参考二叉树的遍历实验，体会如何完整地遍历一颗二叉树。

【实验 5.3】　计算二叉树的深度实验

计算二叉树的高度。

第 1 步：任务分析。

使用递归或者非递归的方式实现二叉树的遍历。

第 2 步：程序构思。

空二叉树的高度为 0，对于非空二叉树，其深度根据定义等于根结点的左子树和右子树中深度较大者的子树的深度加 1。有如下的递归公式：

$$\begin{cases} \text{depth}(\text{tree}) = 1 & \text{tree 为空} \\ \text{depth}(\text{tree}) = \max(\text{depth}(\text{tree->Llink}), \text{depth}(\text{tree->Rlink})) + 1 & \text{其他情况} \end{cases}$$

第 3 步：源程序。

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 50
//二叉树链表存储结构
typedef struct btnode
{
    int  Data;    //结点数据内容
    struct btnode   * Llink;    //左子树指针
    struct btnode   * Rlink;    //右子树指针
}btnode, *btreetype;

/*****------------------------*****/
// 函数名：  CreateTree(int n)
// 参数：    (传入)int n   数据数量
// 返回值：  返回二叉树(根结点)指针
// 功能：    建立二叉树
/*****------------------------*****/
```

```
18      btreetype CreateTree(int n)
19      {
20          int i;
21          btreetype root = NULL;
22
23          for(i = 0;i < n;i ++)
24          {
25                  btreetype newNode;
26                  btreetype currentNode;
27                  btreetype parentNode;
28
29                  //建立新结点
30                  newNode = (btreetype)malloc(sizeof(btnode));
31                  scanf("%d",&newNode -> Data);/ * 新结点赋值 * /
32                  newNode -> Llink = NULL;
33                  newNode -> Rlink = NULL;
34
35                  currentNode = root;    //存储当前结点指针
36              if (currentNode == NULL) root = newNode;//新结点作为二叉树根
                结点
37                  else
38                  {
39                      while (currentNode != NULL)
40                      {
41                          parentNode = currentNode;    //存储当前结点的父结点
42                          if (newNode -> Data < currentNode -> Data)
                                                         //比较结点数值大小
43                              currentNode = currentNode -> Llink;    //左子树
44                              else
45                              currentNode = currentNode -> Rlink;    //右子树
46                      }
47                      //根据数值大小连接父结点和子结点
48                      if (newNode -> Data < parentNode -> Data)
49                          parentNode -> Llink = newNode;
50                      else
51                          parentNode -> Rlink = newNode;
52                  }//else
53          }//for
54          return root;
55      }
56
57      / * * * * - - - - - - - - - - - - - - - - - - - - - - - -    * * * * * /
58      // 函数名：  depth(btreetype &root)
```

```
59      // 参数：      (传入)btreetype &root   二叉树指针
60      // 功能：      输出二叉树深度
61      /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
62      int depth(btreetype &root)
63      {
64          btreetype p;
65          p = root;
66
67          int dep1;
68          int dep2;
69          if (root == NULL)    return 0;
70          else
71          {
72              dep1 = depth(p -> Llink);    //测左子树的深度
73              dep2 = depth(p -> Rlink);    //测右子树的深度
74
75              //二叉树的深度为左子树或右子树的最大深度加 1
76              if (dep1 > dep2)    return (dep1 + 1);
77              else return (dep2 + 1);
78          }//else
79      }
80
81      //测试主程序
82      void main()
83      {
84          btreetype btree;
85          int count;
86          printf("input the number of elements:\n");
87          scanf("%d",&count);
88          printf("input data(num = %d):\n",count);
89          btree = CreateTree(9);
90          printf("\n 建立的二叉树的深度为:%d\n",depth(btree));
91      }
```

第 4 步：测试数据（参见图 5-2）。

运行结果：

```
input the number of elements:
9
input data(num = 9):
45 2 68 1 23 5 12 9 6
建立的二叉树的深度为:7
```

思考题：

请参考上述程序二叉树深度的实现,写出求二叉树宽度的算法(二叉树的某层上具有的结点数最多,其总数定义为宽度)。

【实验5.4】　二叉排序树的判定实验

判定一棵二叉树是否是二叉排序树。

第1步:任务分析。

根据二叉排序树的定义判定一棵二叉树是否是二叉排序树。空的二叉树也是二叉排序树,对于非空二叉树要求如下:

⑴ 若它的左子树非空,则左子树上所有结点的值均小于根结点的值;

⑵ 若它的右子树非空,则右子树上所有结点的值均大于根结点的值;

⑶ 左、右子树本身又是一棵二叉排序树。

第2步:程序构思。

根据判定定义,拟采用递归的形式。

递归部分:

⑴ 判定根结点是否为空;

⑵ 若左子树不为空,判断左子树结点是否小于根结点值。

⑶ 若右子树不为空,判断右子树结点是否大于根结点值

递归结束条件:

左右子树为空。

第3步:源程序。

```
1       //判断是否是排序二叉树
2       #include <stdio.h>
3       #include <stdlib.h>
4       #define MAX 50
5       //二叉树链表存储结构
6       typedef struct btnode
7       {
8          int  Data;    //结点数据内容
9          struct btnode   * Llink;    //左子树指针
10         struct btnode   * Rlink;    //右子树指针
11      }btnode,*btreetype;
12
13      /*****------------------------    *****/
14      // 函数名:   CreateTree(int n)
15      // 参数:     (传入)int n   数据数量
16      // 返回值:   返回二叉树(根结点)指针
17      // 功能:     建立二叉树
18      /*****------------------------    *****/
19      btreetype CreateTree(int n)
20      {
```

```
21          int i;
22          btreetype root = NULL;
23
24          for(i = 0;i < n;i ++)
25          {
26                  btreetype newNode;
27                  btreetype currentNode;
28                  btreetype parentNode;
29
30                  //建立新结点
31                  newNode = (btreetype)malloc(sizeof(btnode));
32                  scanf("%d",&newNode -> Data);/ * 新结点赋值 * /
33                  newNode -> Llink = NULL;
34                  newNode -> Rlink = NULL;
35
36                  currentNode = root;     //存储当前结点指针
37                  if (currentNode == NULL) root = newNode;//新结点作为二叉树根
                    结点
38                  else
39                  {
40                      while (currentNode != NULL)
41                      {
42                          parentNode = currentNode;     //存储当前结点的父结点
43                          if (newNode -> Data < currentNode -> Data)
                                                          //比较结点数值大小
44                              currentNode = currentNode -> Llink;     //左子树
45                          else
46                              currentNode = currentNode -> Rlink;     //右子树
47                      }
48                      //根据数值大小连接父结点和子结点
49                      if (newNode -> Data < parentNode -> Data)
50                          parentNode -> Llink = newNode;
51                      else
52                          parentNode -> Rlink = newNode;
53                  }//else
54          }//for
55          return root;
56      }
57
58      / * * * * - - - - - - - - - - - - - - - - - - - - - - - -    * * * * * /
59      // 函数名：   Check(btreetype &root)
60      // 参数：     (传入)btreetype &root 二叉树指针
61      // 功能：     判定一棵二叉树是否是二叉排序树
```

```
/* * * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * * /
int Check(btreetype &root)
{
    btreetype p;
    p = root;

    //1 进行比较判别
    if(p == NULL)//当前结点是二叉排序树(为空),结束判定
        return 1;
    if(p -> Llink && (p -> Llink -> Data > p -> Data))
        return 0;
    if(p -> Rlink && (p -> Rlink -> Data < p -> Data))
        return 0;
    //2 进行递归判别
    return(Check(p -> Llink) && Check(p -> Rlink));

}
//测试主程序
void main()
{
    btreetype btree;
    int count;
    printf("input the number of elements:\n");
    scanf("%d",&count);
    printf("input data(num = %d):\n",count);
    btree = CreateTree(count);
    if(Check(btree))    printf("建立的二叉树是二叉排序树\n");
    else printf("建立的二叉树不是二叉排序树\n");
}
```

第 4 步：测试数据 (参见图 5-2)。

运行结果：

```
input the number of elements:
9
input data(num = 9):
45 2 68 1 23 5 12 9 6
```

建立的二叉树是二叉排序树

思考题：

1. **根据判定实验的算法，思考完成二叉排序树的查找函数。**

2. **根据二叉排序树的判定和查找函数，结合本实验中二叉树的生成函数，思考完成二叉排序树的生成函数。**

3. **假设二叉树用链表实现，根据定义给出算法，判定给定的二叉树是否为完全二叉树。**

4. 请参考第 7 章实验 7.3 中的二叉排序树的实际应用。

【实验 5.5】　二叉树的遍历实验

二叉树的遍历实现。

第 1 步：任务分析。

二叉树是非线性结构，遍历时是先访问根结点还是先访问子树，是先访问左子树还是先访问右子树必须有所规定，这就是遍历规则。采用不同的遍历规则会产生不同的遍历结果，因此对二叉树进行遍历时，必须设定遍历规则。根据遍历规则采用递归或者非递归的方式实现二叉树的遍历。

第 2 步：程序构思。

分先序遍历、中序遍历和后序遍历三种情况考虑。

先序遍历，当二叉树非空时按以下顺序遍历，否则结束操作：

⑴ 访问根结点；

⑵ 按先序遍历规则遍历左子树；

⑶ 按先序遍历规则遍历右子树。

中序遍历，当二叉树非空时按以下顺序遍历，否则结束操作：

⑴ 按中序遍历规则遍历左子树；

⑵ 访问根结点；

⑶ 按中序遍历规则遍历右子树。

后序遍历，当二叉树非空时按以下顺序遍历，否则结束操作：

⑴ 按后序遍历规则遍历左子树；

⑵ 按后序遍历规则遍历右子树；

⑶ 访问根结点。

采用非递归算法实现先序、中序、后序遍历时要用到栈结构。根据中序遍历二叉树的递归定义，转换成非递归函数时用栈报销返回的结点，先扫描根结点的所有左结点并入栈，出栈一个结点，访问之，然后扫描该结点的右结点并入栈再扫描该右结点的所有左结点并入栈，如此这样，直到栈空为止。具体实现参见源程序部分。

第 3 步：源程序。

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 50
//二叉树链表存储结构
typedef struct btnode
{
    int  Data;    //结点数据内容
    struct btnode   * Llink;    //左子树指针
    struct btnode   * Rlink;    //右子树指针
}btnode,*btreetype;

/* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
```

```
13      // 函数名：  OutputTree(btreetype &root)
14      // 参数：    (传入)btreetype &root   二叉树指针
15      // 功能：    输出二叉树
16      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
17      void OutputTree(btreetype &root)
18      {
19          btreetype p;
20
21          //打印左子树
22          p = root -> Llink;
23          printf("建立的二叉树的左子树为:");
24          while (p != NULL)
25          {
26              printf("%d ",p -> Data);
27              p = p -> Llink;
28          }
29          //打印右子树
30          p = root -> Rlink;
31          printf("\n 建立的二叉树的右子树为:");
32          while (p != NULL)
33          {
34              printf("%d ",p -> Data);
35              p = p -> Rlink;
36          }
37      }
38
39      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
40      // 函数名：  PreOrder(btreetype &root)
41      // 参数：    (传入)btreetype &root 二叉树指针
42      // 功能：    先序遍历二叉树
43      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
44      void PreOrder(btreetype &root)
45      {
46          btreetype p;
47          p = root;
48          if (p != NULL)
49          {
50              printf("%d ",p -> Data);
51              PreOrder(p -> Llink);    //递归处理左子树
52              PreOrder(p -> Rlink);    //递归处理右子树
53          }
54      }
55
```

```
/*****------------------------------*****/
// 函数名：  InOrder(btreetype &root)
// 参数：    (传入)btreetype &root 二叉树指针
// 功能：    中序遍历二叉树(递归方式)
/*****------------------------------*****/
void InOrder(btreetype &root)
{
    btreetype p;
    p=root;
    if (p!=NULL)
    {
        InOrder(p->Llink);    //递归处理左子树
        printf("%d ",p->Data);
        InOrder(p->Rlink);    //递归处理右子树
    }
}
/*****------------------------------*****/
// 函数名：  InOrder(btreetype &root)
// 参数：    (传入)btreetype &root  二叉树指针
// 功能：    中序遍历二叉树(非递归方式)
/*****------------------------------*****/
void InOrder_Norecursion(btreetype &root)
{
    btreetype stack[MAX];
    btreetype p;
    int top=0;
    p=root;
    do
    {
        while(p!=NULL)    //扫描左结点
        {
            top++;
            stack[top]=p;
            p=p->Llink;
         }
         if (top > 0)
         {
            p=stack[top];    //p 所指结点为无左子树或其左子树已遍历过
            top--;
            printf("%d ",p->Data);
            p=p->Rlink;    //扫描右结点
         }
     }while(p!=NULL ||top !=0);
```

```
99      }
100
101     /*****------------------------*****/
102     // 函数名：  PostOrder(btreetype &root)
103     // 参数：    (传入)btreetype &root 二叉树指针
104     // 功能：    后先序遍历二叉树
105     /*****------------------------*****/
106     void PostOrder(btreetype &root)
107     {
108          btreetype p;
109          p=root;
110          if (p!=NULL)
111          {
112              PreOrder(p->Llink);    //递归处理左子树
113              PreOrder(p->Rlink);    //递归处理右子树
114              printf("%d ",p->Data);
115          }
116     }
117
118     /*****------------------------*****/
119     // 函数名：  CreateTree(int n)
120     // 参数：    (传入)int n 数据数量
121     // 返回值：  返回二叉树(根结点)指针
122     // 功能：    建立二叉树
123     /*****------------------------*****/
124     btreetype CreateTree(int n)
125     {
126         int i;
127         btreetype root=NULL;
128
129         for(i=0;i<n;i++)
130         {
131             btreetype newNode;
132             btreetype currentNode;
133             btreetype parentNode;
134
135             //建立新结点
136             newNode=(btreetype)malloc(sizeof(btnode));
137             scanf("%d",&newNode->Data);/*新结点赋值*/
138             newNode->Llink=NULL;
139             newNode->Rlink=NULL;
140
141             currentNode=root;    //存储当前结点指针
```

```
142             if (currentNode == NULL) root = newNode;//以新结点作为二叉树根
                结点
143             else
144             {
145                 while (currentNode != NULL)
146                 {
147                     parentNode = currentNode;    //存储当前结点的父结点
148                     if (newNode -> Data < currentNode -> Data)
                                                     //比较结点数值大小
149                         currentNode = currentNode -> Llink;    //左子树
150                     else
151                         currentNode = currentNode -> Rlink;    //右子树
152                 }
153                 //根据数值大小连接父结点和子结点
154                 if (newNode -> Data < parentNode -> Data)
155                     parentNode -> Llink = newNode;
156                 else
157                     parentNode -> Rlink = newNode;
158             }//else
159         }//for
160         return root;
161     }
162
163     //测试主程序
164     void main()
165     {
166         btreetype btree;
167         int count;
168         printf("input the number of elements:\n");
169         scanf("%d",&count);
170         printf("input data(num = %d):\n",count);
171         btree = CreateTree( count );
172         //二叉树的各种遍历 - - - - - - - - -
173         printf("\n 先序遍历建立的二叉树:");
174         PreOrder(btree);
175         printf("\n 中序遍历建立的二叉树(递归方式):");
176         InOrder(btree);
177         printf("\n 中序遍历建立的二叉树(非递归方式):");
178         InOrder_Norecursion(btree);
179         printf("\n 后序遍历建立的二叉树:");
180         PostOrder(btree);
181         //- - - - - - - - - - - - - - - - - - - -
182     }
```

第 4 步：测试数据 (参见图 5-2)。

运行结果：

```
input the number of elements:
9
input data(num = 9):
45 2 68 1 23 5 12 9 6
```

先序遍历建立的二叉树:45 2 1 23 5 12 9 6 68

中序遍历建立的二叉树 (递归方式):1 2 5 6 9 12 23 45 68

中序遍历建立的二叉树 (非递归方式):1 2 5 6 9 12 23 45 68

后序遍历建立的二叉树:2 1 23 5 12 9 6 68 45

思考题：

1. **请参考上述程序中遍历的非递归调用，实现另外两种遍历的非递归调用。**
2. **结合二叉树的遍历，请实现统计一颗二叉树的结点数的函数。**

5.3　二叉树应用

【实验 5.6】　哈夫曼编码的设计与实现

哈夫曼编码程序的实现。

第 1 步：任务分析。

哈夫曼编码译码的实现，首要的是根据给定的 n 个权值构造哈夫曼树。通过遍历此二叉树完成哈夫曼编码。

第 2 步：程序构思。

构造哈夫曼树：

⑴ 将给定的 n 个权值 $\{W_1, W_2, \ldots, W_n\}$ 作为 n 个根结点的权值构造一个具有 n 棵二叉树的森林 $\{T_1, T_2, \ldots, T_n\}$，其中每棵二叉树只有一个根结点；

⑵ 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；

⑶ 在森林中，将上面选择的这两棵根权值最小的二叉树从森林中删除，并将刚刚新构造的二叉树加入到森林中；

⑷ 重复上面 (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。

编码过程：

哈夫曼树从根到每个叶子都有一条路径，对路径上的各分支约定指向左子树根的分支编码为“0”，指向右子树根的分支编码为“1”，从根到每个叶子相应路径上的“0”和“1”组成的序列就是这个叶子 (字符) 的编码。

哈夫曼树结点的存储结构：

```
typedef struct
{
```

```
int data;           //结点值
int Weight;         //权重
int Flag;           //标识是否待构节点,是的话用 0 表示,否则用 1 表示
int Parent;         //父结点
int LChild,RChild; //左右结点
}hnodetype;
```

- Weight 存放每个结点的权值,初始时只有表示字符的叶结点有权值,在哈夫曼树建立的过程中,各分支结点的权值产生并填入其中。
- Parent 存放每个结点的双亲在数组中的序号,初始时该字段值为 0,在哈夫曼树建立的过程中,逐步填入。
- Llink 与 Rlink 分别存储每个结点左右子女在数组中的序号,初始时该字段值也为 0,在哈夫曼树建立的过程中,逐步填入。

第 3 步:源程序。

```
#define MAXVALUE 10000
#define MAXLEAF 30
#define MAXNODE 59
#define MAXBIT 10
#include <stdio.h>

//哈夫曼树结构
typedef struct
{
    int data;    //结点值
    int Weight;    //权重
    int Flag;    //标识是否待构节点,是的话用 0 表示,否则 1 表示
    int Parent;    //父结点
    int LChild;    //左结点
    int RChild;    //右结点
}hnodetype;

typedef struct
{
    int Bit[MAXBIT];
    int Start;
}hcodetype;

/*****************************    *****/
// 函数名:  InitHaffman(hnodetype HuffNode[],hcodetype HuffCode
            [],int n)
// 参数:    (传入)hnodetype HuffNode[]  哈夫曼树结点
//          (传入)hcodetype HuffCode[]  哈夫曼树编码树结点
```

```
28      //            (传入)int n   结点数量
29      // 功能：     哈夫曼结点初始化
30      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
31      void InitHaffman(hnodetype HuffNode[],hcodetype HuffCode[],int
        n)
32      {
33          int i;
34          //把生成的节点初始化,把指向父亲的指针,左孩子、右孩子的指针都先置空
35          for(i=0;i<2*n-1;i++)
36          {
37              HuffNode[i].Weight=0;
38              HuffNode[i].Parent=0;
39              HuffNode[i].Flag=0;
40              HuffNode[i].LChild=-1;
41              HuffNode[i].RChild=-1;
42          }
43          for(i=0;i<n;i++)
44          {
45              getchar();
46              printf("输入第%d个叶结点值:",i+1);
47              scanf("%c",&HuffNode[i].data);
48              printf("输入对应结点权值:");
49              scanf("%d",&HuffNode[i].Weight);
50          }
51      }
52
53      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
54      // 函数名：   OutputHaffman(hnodetype HuffNode[],hcodetype Huff-
                      Code[],int n)
55      // 参数：     (传入)hnodetype HuffNode[]  哈夫曼树结点
56      //            (传入)hcodetype HuffCode[]  哈夫曼树编码树结点
57      //            (传入)int n   结点数量
58      // 功能：     输出哈夫曼编码
59      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
60      void OutputHaffman(hnodetype HuffNode[],hcodetype HuffCode[],
        int n)
61      {
62          int i,j;
63          printf("%d个叶结点对应编码为:\n",n);
64          for(i=0;i<n;i++)
65          {
66              printf("%c----",HuffNode[i].data);
67              for(j=HuffCode[i].Start+1;j<n;j++)
```

```
68                  printf("%d",HuffCode[i].Bit[j]);
69              printf("\n");
70          }
71      }
72
73      /*****------------------------------*****/
74      // 函数名：  Haffman(hnodetype HuffNode[],hcodetype HuffCode[],
                     int n)
75      // 参数：    (传入)hnodetype HuffNode[]  哈夫曼树结点
76      //           (传入)hcodetype HuffCode[]  哈夫曼树编码树结点
77      //           (传入)int n   结点数量
78      // 功能：    构造哈夫曼树,根据树生成哈夫曼编码
79      /*****------------------------------*****/
80      void Haffman(hnodetype HuffNode[],hcodetype HuffCode[],int n)
81      {
82          int i,j,m1,m2,x1,x2,c,p;
83          hcodetype cd;
84
85           for(i=0;i<n-1;i++)    //构造哈夫曼树
86           {
87              //根据哈夫曼树的构成过程,始终选择最小权值的两个节点构成一棵二叉树
88              m1=m2=MAXVALUE;
89              //x1 和 x2 为最小权重的两个结点位置
90              x1=x2=0;
91              //循环从 flag 为 0 的节点中找到一个,供下面取最小值
92              for(j=0;j<n+i;j++)
93                  {
94                  if(HuffNode[j].Weight<m1 && HuffNode[j].Flag==0)
95                  {
96                      m2=m1;
97                      x2=x1;
98                      m1=HuffNode[j].Weight;
99                      x1=j;    //记下 x1 的地址
100                 }
101                 else if(HuffNode[j].Weight<m2 && HuffNode[j].Flag=
=0)
102                 {
103                     m2=HuffNode[j].Weight;
104                     x2=j;    //记下 x2 的地址
105                 }
106             }//for
107
108 //把找到的两个节点按照哈夫曼树的规则构建成一个二叉树,x1 为左孩子,x2 为右孩子
```

```
109             HuffNode[x1].Parent = n + i;
110             HuffNode[x2].Parent = n + i;
111             HuffNode[x1].Flag = 1;    //将 x1 的下标置 1
112             HuffNode[x2].Flag = 1;    //将 x2 的下标置 1
113             HuffNode[n + i].Weight = HuffNode[x1].Weight + HuffNode
                [x2].Weight;
114             HuffNode[n + i].LChild = x1;
115             HuffNode[n + i].RChild = x2;
116         }//for
117
118         for(i = 0;i < n;i ++)    //根据哈夫曼树生成哈夫曼编码
119         {
120             cd.Start = n - 1;
121             c = i;
122             p = HuffNode[c].Parent;
123
124             while(p != 0)//当父节点不为根节点的时候,逆序往上找
125             {
126                 //如果当前是左孩子,则编码为 0
127                 if(HuffNode[p].LChild == c)cd.Bit[cd.Start] = 0;
128                 //当前是右孩子的话编码为 1
129                 else cd.Bit[cd.Start] = 1;
130                 cd.Start --;
131                 c = p;
132                 p = HuffNode[c].Parent;
133             }
134
135             for(j = cd.Start + 1;j < n;j ++)
136             {
137                 HuffCode[i].Bit[j] = cd.Bit[j];
138                 HuffCode[i].Start = cd.Start;
139             }
140         }//for
141     }//end of Haffman
142
143     //测试主程序
144     void main()
145     {
146          hnodetype HuffNode[MAXNODE];
147          hcodetype HuffCode[MAXLEAF];
148          int n;
149          printf("输入叶结点个数 n:\n");
150          scanf("%d",&n);
```

```
151
152             InitHaffman(HuffNode,HuffCode,n);
153             Haffman(HuffNode,HuffCode,n);
154             OutputHaffman(HuffNode,HuffCode,n);
155       }
```

第 4 步：测试数据。

哈夫曼树的 6 个结点为 a、b、c、d、e、f，其权值分别为 3，9，12，3，2，71 哈夫曼树的构造过程如图 5-3 所示，哈夫曼编码如图 5-4 所示。

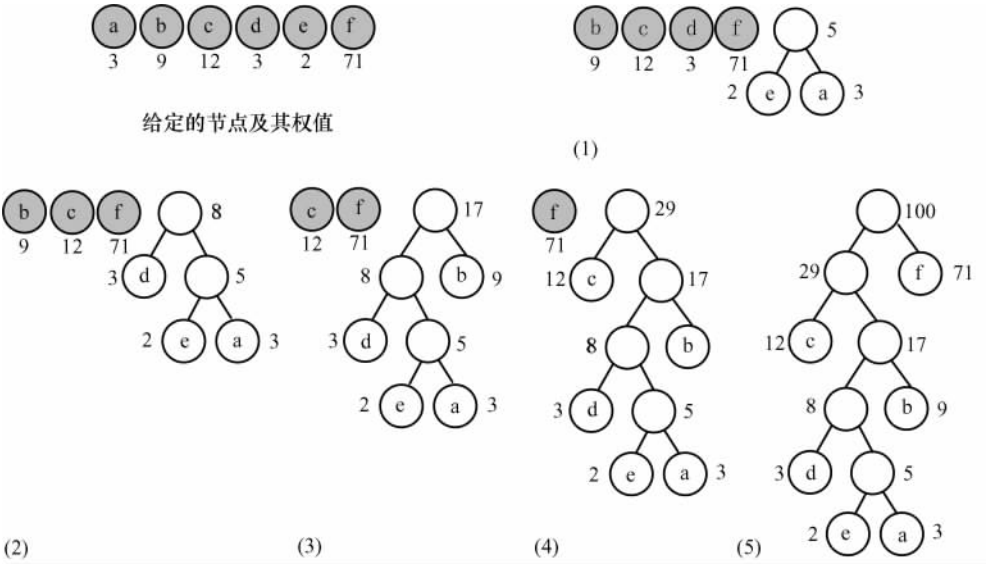


图 5-3　哈夫曼树构造过程

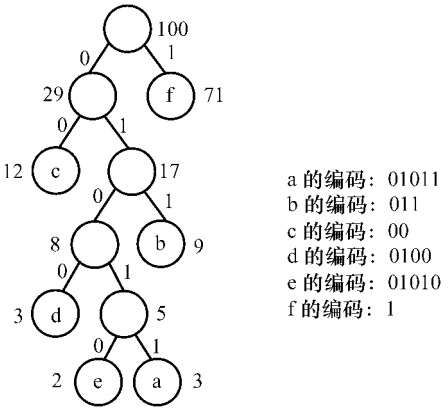


图 5-4　哈夫曼树及各字符的哈夫曼编码

运行结果：

输入叶结点个数 n：

6

输入第 1 **个叶结点值**：a

输入对应结点权值:3
输入第 2 个叶结点值:b
输入对应结点权值:9
输入第 3 个叶结点值:c
输入对应结点权值:12
输入第 4 个叶结点值:d
输入对应结点权值:3
输入第 5 个叶结点值:e
输入对应结点权值:2
输入第 6 个叶结点值:f
输入对应结点权值:71
6 个叶结点对应编码为:
a——01011
b——011
c——00
d——0100
e——01010
f——1

思考题:

请参考程序实现哈夫曼译码的实现函数。

5.4 小　　结

通过实验掌握二叉树的数组存储结构和链表存储结构,掌握二叉树的三种遍历方法,掌握线索二叉树的表示、建立与遍历,掌握哈夫曼树的构造与应用。

第 6 章　图

本章的典型实验主要练习图的两种表示方法和图的遍历，在此基础上应用图结构解决经典的实际应用，分为图结构基本操作实验和图结构应用实验两类，共 4 个实验内容。

6.1　知识点概述

1. 定义

图是指由若干个节点和若干条边构成的结构，是一种重要的非线性数据结构。

图 G 是一个有序对 (V,E)，记作 G = (V,E)，V 是一个非空有限集合，它的元素称为顶点；E 是由 V 中两个元素 v_i，v_j 组成的子集的集合，E 的元素称为边。

若顶点的偶对是无序的，则称此图为无向图。无向图中的一条边记为 (v_i，v_j)。

若顶点的偶对是有序的，则称此图为有向图，有向图中的一条有向边又称为弧，记为 < v_i，v_j >，表示从顶点 v_i 出发到顶点 v_j 存在一条弧，其中，v_i 称之为弧尾，v_j 称之为弧头。

无向完全图：如果无向图中每对顶点之间都有一条边，则称这样的无向图称为无向完全图，显然，有 n 个顶点的无向完全图具有 n (n - 1) /2 条边。

有向完全图：如果有向图中每对顶点 V 和 W 之间都有两条边 < v,w > 和 < w,v >，则称这样的有向图为有向完全图，显然，有 n 个顶点的有向完全图具有 n (n - 1) 条边。有向完全图和无向完全图具有对称性。

子图：假设有两个图 G 和 G'，G = (V,E)，G' = (V',E') 且满足 V' 是 V 的子集，E' 是 E 的子集，则称 G' 为 G 的子图。

邻接点：对于无向图，如果存在边 (v,w)，则顶点 v 和顶点 w 互为邻接点。

顶点的度：对于无向图，顶点的度是以该顶点相关联的边的数目，在有向图中，顶点的度为该顶点的入度和出度之和。

顶点的入度和出度：入度指以该顶点为弧头的弧的数目；出度则指以该顶点为孤尾的弧的数目。

路径：对于无向图 G = (V,E)，若存在着一个顶点序列 v_p，v_{i1}，v_{i2}，…，v_{in}，v_q，使得 (v_p，v_{i1}) (v_{i1}，v_{i2})，…，(v_{in}，v_q) 均属于 E 中，则顶点序列 v_p，v_{i1}，v_{i2}，…，v_{in}，v_q 称为 v_p 到 v_q 之间的一条路径 (有向图亦然)。

路径长度：路径长度定义为路径上边的数目。

连通图：在无向图中，若每一对顶点之间都有路径，则称此图为连通图。

图的操作主要有构造、销毁、查找、遍历、求路径等。

2. 存储结构

图的存储结构有顺序存储和链式存储。由于图的结构比较复杂，任意两个顶点之间都可能存在联系，因此很难以数据元素在存储区中的物理位置来表示元素之间的关系，而必须显式的存储结点之间的关系 (即边) 的信息。一般有两种方法，一是用边的集合表示图；二

是用链接关系表示图。实际中邻接矩阵就属于顺序存储，邻接链表、十字链表、多重链表属于链式存储。

3. 图的遍历

图的遍历与树的遍历类似，也是从某个顶点出发，沿着某条搜索路径对图中所有顶点各作一次访问。若给定的图是连通图，则从图中任一顶点出发顺着边可以访问到该图的所有顶点。但是，图的遍历比树的遍历复杂得多，这是因为图中的任一顶点都可能和其余顶点相邻接，故在访问了某个顶点之后，可能顺着某条回路又回到了该顶点。为了避免重复访问同一个顶点，必须记住每个顶点是否被访问过。根据搜索的不同规则，我们将遍历图的方法分为两种：深度优先遍历和广度优先遍历。

深度优先遍历 DFS (Depth_First_Search) 基本思想是：

(1) 选择一个起始点 v_0 出发，并访问之；

(2) 依次从 v_0 的未访问过的邻接点出发，深度优先遍历图，直到图中与 v_0 有路径相通的顶点都被访问过为止；

(3) 如此时图中尚有顶点未被访问过，则另选图中一个未访问过的顶点作起始点，重复步骤上述过程，直到所有顶点都被访问过为止。

广度优先遍历其的基本思想是：

(1) 选择一个起始点 v_0，并访问之；

(2) 从 v_0 出发，依次访问 v_0 的未被访问过的邻接顶点 v_1，v_2，…，v_k，然后依次从 v_1，v_2，…，v_k，出发，访问各自未被访问过的邻接顶点；

(3) 重复步骤 (2) 直到图中所有顶点都被访问过为止。

4. 最小生成树

在无向图中，连通图 G 的一个子图如果是一棵包含 G 的所有顶点的树，由于有 n 个顶点的连通图至少有 n - 1 条边，而所有包含 n - 1 条边及 n 个顶点的连通图就是该图的生成树。也称该生成树是连通图的极小连通子图。可得出结论：一个生成树中任意两个顶点间，只有一条通路。对于带权图而言，图中的边往往具有不同的权值，因此由不同边形成的不同生成树其权值总和往往会不同，其中代价最小的生成树称为最小生成树，也称为最小代价生成树。

产生最小生成树的方法有著名的普里姆 (Prim) 算法和克鲁斯卡尔 (Kruskal) 算法。

普里姆算法的思路：

假设 N = (V,E) 是带权连通网，T 是 N 上最小生成树中边的集合，U 是最小生成树中顶点的集合，执行以下操作：

(1) 令 $T = \{\}$，$U = \{u_0\}$ (u_0 是 V 中任意顶点)。

(2) 从 E 中选取代价最少的边 (V_x,V_y)，其中 $V_x \in U$，$V_y \in V - U$。

(3) 将边 (V_x,V_y) 加入 T 中，将顶点 V_y 加入 U 中。

(4) 重复步骤 (2) 和 (3)，直到 $U = V$ 为止。

克鲁斯卡尔算法的思路：

假设 N = (V, {E}) 是带权连通网，则令 T 是 N 上最小生成树，执行以下操作：

(1) 令 T = (V, {})，即 T 中有 n 个顶点组成的非连通图，每一个顶点自成一个连通分量。

⑵ 从 E 中找一条代价最少的边 (V_x, V_y)，如果该边的两个顶点落在 T 中不同的连通分量上，则将其并入集合T 中，否则舍去该边。

⑶ 重复步骤 ⑵，直到所有的顶点均在同一个连通分量上为止。

5. 图的应用

1) 拓扑排序

在有向图中，用顶点表示活动，有向边表示活动的优先顺序，这种有向图叫做用顶点表示活动的网络 (Active On Vertex Network) 简称为 AOV 网，AOV 网是一个有向无环无权图。实现拓扑排序步骤为：

⑴ 在 AOV 网中选择一个入度为 0 的顶点且输出之；

⑵ 从 AOV 网中删除此顶点及以该顶点为尾的所有有向边；

⑶ 重复 ⑴ 和 ⑵ 两步，直到 AOV 网中所有顶点都被输出或网中不存在入度为 0 的顶点。

2) 关键路径

有向图中，用顶点表示事件，有向边表示活动，边权表示活动持续时间的网称为用边表示活动的网络 (Active On Edge Network)，简称为 AOE 网，AOE 网是一个带权的无环有向图。求关键路径的基本步骤为：

⑴ 求每个活动 a_i 最早开始的时间 E (i)；

⑵ 求每个活动 a_i 的最迟开始时间 L (i)；

⑶ 求完成每个活动 a_i 的时间余量；

⑷ 确定出所有的关键活动，从而得出关键路径。即显然当 a_i 的时间余量 = L (i) - E (i) = 0，a_i 就是关键活动，由关键活动组成的路径就是关键路径。

3) 最短路径

交通网络可以用带权的图表示，图中顶点表示城市，边代表城市间的公路 (或铁路或航线等)，边上的权表示两个城市间的距离，或表示通过这段路程所花费的时间或者费用等。任意两个城市间的一条花费时间最短或费用最少的路径称为最短路径。求最短路径的算法有迪杰斯特拉算法和弗洛伊德算法。

迪杰斯特拉算法的基本思想是：把图中顶点分成两组，其中第一组 S_0 表示尚未确定最短路径的顶点的集合，第二组 S_1 表示已求出最短路径的顶点集合。按最短路径长度递增次序逐个把第一组 S_0 中的顶点加入到第二组 S_1 中。直至从 v_0 出发可以到达的所有顶点都在 S_1 中。在这一过程中，总保持从 v_0 到 S_1 中各顶点的最短路径长度都不大于从 v_0 到 S_0 中任何顶点的最短路径长度。每次以一个顶点为源点，重复执行迪杰斯特拉算法 n 次。

弗洛伊德算法的基本思想是：设求从顶点 v_i 到顶点 v_j 的最短路径，如果从 v_i 到 v_j 有弧，则从 v_i 到 v_j 存在着一条长度为 cost [i] [j] 的路径，该路径不一定是最短路径，需要进行 n 次试探。首先考虑当 v_0 作为中间顶点时，求从 v_i 到 v_j 的最短路径，其次当 v_0 和 v_1 作为中间顶点时，求从 v_i 到 v_j 的最短路径 …… 依次直到 n 个顶点都作为中间顶点时，得到的各顶点之间的最短路径即为最终求得的任意一对顶点之间的最短路径。

6.2 图结构基本操作实验

【实验6.1】 图的邻接矩阵表示和邻接表表示相互转换实验

实现图的邻接矩阵表示和邻接表表示相互转换程序。

第1步：任务分析。

本实验包括要实现图的邻接矩阵表示和邻接表表示两种形式，然后根据各自的定义进行转换。

第2步：程序构思。

图的邻接矩阵表示和邻接表表示相互转换，实质是数组存储形式和链表存储的转换，关键在于对图的邻接矩阵表示和邻接表表示两种形式的深刻理解和掌握。实现思路请参见实现程序的注释部分。

第3步：源程序。

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 20

//图的邻接表存储表示----------------
typedef struct ArcNode
{
    int adjvex;                 //弧的邻接顶点
    char info;                  //邻接点信息
    struct ArcNode *nextarc;    //指向下一条弧的指针
}ArcNode;

typedef struct Vnode
{
    char data;                  //结点信息
    ArcNode *link;
}Vnode, AdjList[MAX+1];

typedef struct
{
    AdjList vertices;
    int vexnum;                 //顶点数
    int arcnum;                 //边数
}ALGraph;
//----------------------

//图的邻接矩阵表示---------
typedef struct
{
```

```
30      int n;                          // 图的顶点个数
31      char vexs[MAX+1];               //顶点信息
32      int arcs[MAX+1][MAX+1];         //边信息
33  }AdjMatrix;
34  //-----------------
35
36  /*****------------------------------*****/
37  // 函数名：  AdjListToAdjMatrix(AdjList gl, AdjMatrix &gm, int n)
38  // 参数：    (传入) AdjList gl 图的邻接表
39  //           (传入) int n 顶点个数
40  //           (传出) AdjMatrix gm 图的邻接矩阵
41  // 功能：    将图的邻接表表示转换为邻接矩阵表示
42  /*****------------------------------*****/
45  void  AdjListToAdjMatrix(AdjList gl, AdjMatrix &gm, int n)
46  {
47      int i, j;
48      ArcNode *p;
49
50      for (i=1; i<=MAX; i++)
51          for (j=1; j<=MAX; j++)
52              gm.arcs[i][j]=0;
53
54      for (i=1; i<=n; i++)
55      {
56          p=gl[i].link; //取第一个邻接点
57          while (p!=NULL)//下一个邻接点
58          {
59              gm.arcs[i][p->adjvex]=1;
60              p=p->nextarc;
61          }//while
62      }//for
63  }
64
65  /*****------------------------------*****/
66  // 函数名：  AdjMatrixToAdjList( AdjMatrix gm, AdjList &gl, int n)
67  // 参数：    (传入) AdjMatrix gm 图的邻接矩阵
68  //           (传入) int n 顶点个数
69  //           (传出) AdjList gl 图的邻接表
70  // 功能：    将图的邻接矩阵表示法转换为邻接表表示
71  /*****------------------------------*****/
72  void AdjMatrixToAdjList( AdjMatrix gm, AdjList &gl, int n)
73  {
74      int i, j;
75      ArcNode *p;
76
```

```
77        //邻接表表头向量初始化
78        for (i =1; i <=n; i ++)
79        {
80            gl[i].data =0;
81            gl[i].link =NULL;
82         }
83
84         for (i =1; i <=n; i ++)
85            for (j =1; j <=n; j ++)
86               if (gm.arcs[i][j] ==1)
87               {
88                  p =(ArcNode *)malloc(sizeof(ArcNode));//申请结点空间
89                  p ->adjvex =j;//顶点 i 的邻接点是 j
90                  p ->nextarc =gl[i].link;
91                  gl[i].link =p;//链入顶点 i 的邻接点链表中
92               }
93     }
94     /* * * * - - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
95     // 函数名：  CreateGraph(ALGraph &G)
96     // 参数：    (传入) ALGraph G 图结构体
97     // 功能：    建立图的邻接表表示,并计算了顶点的入度
98     /* * * * - - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
99     void CreateGraph(ALGraph &G)
100    {
101       int i, s, d;
102       ArcNode *p;
103
104       //输入结点信息
105       for(i =1; i <=G.vexnum; i ++)
106        {
107          getchar();
108          printf("第%d 个结点信息(char 型):", i);
109          scanf("%c", &G.vertices[i].data);
110          G.vertices[i].link =NULL;
111       }
112
113       //输入边的信息
114       for(i =1; i <=G.arcnum; i ++)
115        {
116          printf("第%d 条边 - - - -起点序号,终点序号:", i);
117          scanf("%d %d", &s, &d);
118          p =(ArcNode *)malloc(sizeof(ArcNode));
119          p ->adjvex =d;
120          p ->info =G.vertices[d].data;//存储边的权值等信息,此处以顶点数据
             代替
```

```
121         p->nextarc=G.vertices[s].link;    //p 插入顶点 s 的邻接表中
122         G.vertices[s].link=p;
123       }
124   }
125   /*****------------------------------*****/
126   // 函 数 名：    OutputAdjList ( ALGraph &G )、 OutputAdjMatrix
                     (AdjMatrix gm,
                     int n)
127   // 参数：        (传入) ALGraph G 图结构体
128   // 功能：        输出图的邻接表表示、输出图的矩阵表示
129   /*****------------------------------*****/
130   void OutputAdjList(ALGraph &G)
131   {
132         int i;
133         ArcNode *p;
134         printf("图的邻接表如下:\n");
135         for(i=1; i<=G.vexnum; i++)
136         {
137            printf("[%c]", G.vertices[i].data);
138            p=G.vertices[i].link;
139            while (p!=NULL)
140            {
141               printf("--->(%d %c)", p->adjvex, p->info);
142               p=p->nextarc;
143            }
144            printf("\n");
145         }//for
146   }
147   void OutputAdjMatrix(AdjMatrix gm, int n)
148   {
149         printf("图的邻接矩阵如下:\n");
150         for(int i=1;i<=n; i++)
151         {
152            for(int j=1;j<=n; j++)
153            printf("%d ", gm.arcs[i][j]);
154            printf("\n");
155         }
156   }
157
158   void main()
159   {
160         ALGraph G;
161         AdjMatrix gm;
162         printf("输入结点数和边数:");
163         scanf("%d %d", &G.vexnum, &G.arcnum);
```

```
164             CreateGraph(G);
165
166             AdjListToAdjMatrix(G.vertices, gm, G.vexnum);
167
168             //输出对比转换结果
169             OutputGraph(G);
170             OutputAdjMatrix(gm, G.vexnum);
171
172     }
```

第4步：测试数据。

测试数据如图6-1所示。

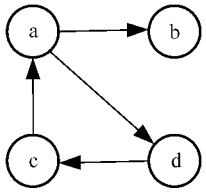


图6-1　测试数据的图结构

运行结果：

```
输入结点数和边数:4 4
第1个结点信息(char型):a
第2个结点信息(char型):b
第3个结点信息(char型):c
第4个结点信息(char型):d
第1条边——起点序号,终点序号:1 2
第2条边——起点序号,终点序号:1 3
第3条边——起点序号,终点序号:3 4
第4条边——起点序号,终点序号:4 1
图的邻接表如下:
[a] - - -> (3 c) - - -> (2 b)
[b]
[c] - - -> (4 d)
[d] - - -> (1 a)
图的邻接矩阵如下:
0 1 1 0
0 0 0 0
0 0 0 1
1 0 0 0
```

思考题：

上述程序完成了邻接表到邻接矩阵的转换过程，请参照教材上采用邻接矩阵表示图的算法生成图的邻接矩阵表示，并参照上述AdjMatrixToAdjList()函数实现此邻接矩阵到邻接表的转换。

【实验 6.2】　图的遍历实验

图的遍历与树的遍历类似，也是从某个顶点出发，沿着某条搜索路径对图中所有顶点各作一次访问。试用程序完成图的遍历实验。

第 1 步：任务分析。

图的遍历是从图中某个顶点出发，沿某条搜索路径对图中的所有结点进行访问，而且仅访问一次。遍历的流程图如图 6-2 所示。根据搜索的不同规则，遍历图的方法分为两种：深度优先遍历（DFS）和广度优先遍历（BFS）。

第 2 步：程序构思。

深度优先遍历其算法基本思想是：

⑴ 选择一个起始点 v_0 出发，并访问之；

⑵ 依次从 v_0 的未访问过的邻接点出发，深度优先遍历图，直到图中与 v_0 有路径相通的顶点都被访问过为止；

⑶ 如此时图中尚有顶点未被访问过，则另选图中一个未访问过的顶点作起始点，重复步骤上述过程，直到所有顶点都被访问过为止。

用流程图表示，如图 6-3 所示。

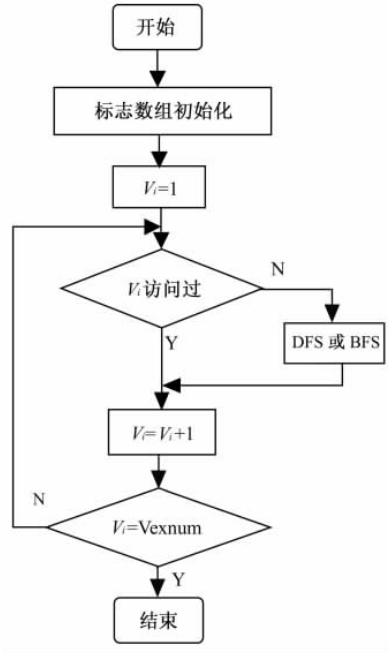


图 6-2　图的遍历流程

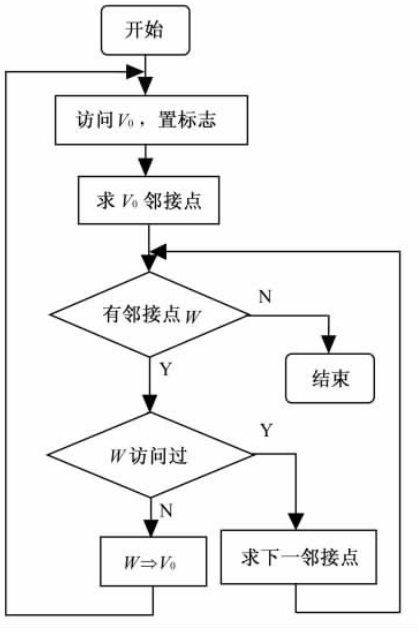


图 6-3　深度优先遍历流程图

广度优先遍历其算法基本思想是：

⑴ 选择一个起始点 v_0，并访问之；

⑵ 从 v_0 出发，依次访问 v_0 的未被访问过的邻接顶点 v_1，v_2，…，v_k，然后依次从 v_1，v_2，…，v_k，出发，访问各自来被访问过的邻接顶点；

⑶ 重复步骤 ⑵ 直到图中所有顶点都被访问过为止。

用流程图表示，如图 6-4 所示。

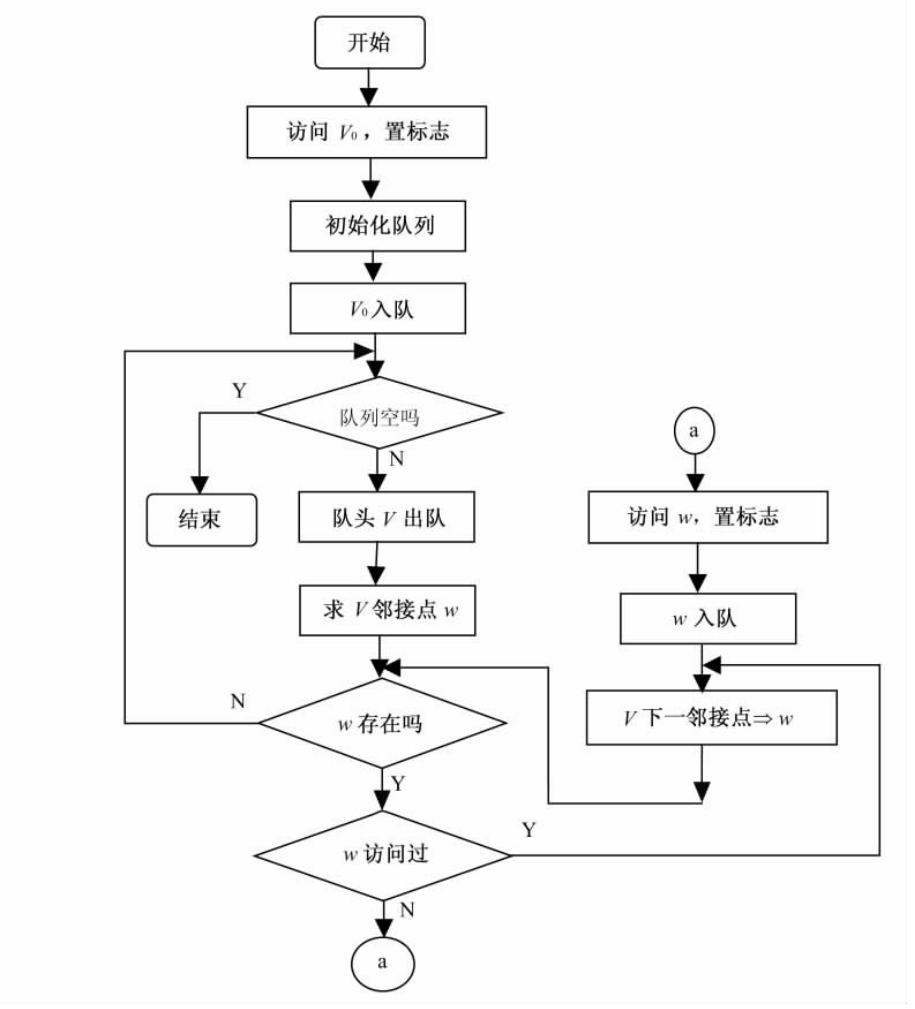


图 6-4　广度优先遍历流程图

第 3 步：源程序。

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 20

//图的邻接表存储表示 - - - - - - - -
typedef struct ArcNode
{
    int adjvex;                 //弧的邻接顶点
    char info;                  //邻接点信息
    struct ArcNode *nextarc;    //指向下一条弧的指针
 }ArcNode;

```

```
typedef struct Vnode
{
    char data;                    //结点信息
    ArcNode * link;
}Vnode, AdjList[MAX +1];

typedef struct
{
    AdjList vertices;
       int vexnum;                //顶点数
       int arcnum;                //边数
}ALGraph;
//- - - - - - - - - - - - - - - -

void CreateGraph(ALGraph &G)
{
    int i, s, d;
    ArcNode * p;

    //输入结点信息
    for(i =1; i <=G.vexnum; i ++)
    {
        getchar();
        printf("第%d 个结点信息(char 型):", i);
        scanf("%c", &G.vertices[i].data);
        G.vertices[i].link =NULL;
    }

    //输入边的信息
    for(i =1; i <=G.arcnum; i ++)
    {
        printf("第%d 条边 - - - -起点序号,终点序号:", i);
        scanf("%d %d", &s, &d);
        p =(ArcNode *)malloc(sizeof(ArcNode));
        p ->adjvex =d;
        p ->info =G.vertices[d].data;//存储边的权值等信息,此处以顶点数据
        代替
        p ->nextarc =G.vertices[s].link;    //p 插入顶点 s 的邻接表中
        G.vertices[s].link =p;
    }
}

//深度遍历图的 DFS 算法实现
```

```
55      void DFS(AdjList g,int v,int visited[])
56      {
57          ArcNode *w;
58          int i;
59          printf("%d  ", v);
60          visited[v]=1;
61          w=g[v].link;
62          while(w!=NULL)
63          {
64              i=w->adjvex;
65              if(visited[i]==0)
66                  DFS(g, i, visited);
67              w=w->nextarc;
68          }
69      }
70      //广度优先遍历图的 BFS 算法实现
71      void BFS(AdjList g,int v,int visited[])
72      {
73          int qu[MAX], f=0, r=0;
74          ArcNode *p;
75          printf("%d  ", v);
76          visited[v]=1;
77          qu[0]=v;
78          while(f<=r)
79          {
80              v=qu[f++];
81              p=g[v].link;
82              while(p!=NULL)
83              {
84                  v=p->adjvex;
85                  if(visited[v]==0)
86                  {
87                      visited[v]=1;
88                      printf("%d ", v);
89                      qu[++r]=v;
90                  }
91                  p=p->nextarc;
92              }
93          }
94      }
95
96      /*****------------------------------   *****/
97      // 函数名：  Traver(AdjList g,int n, int id)
```

```
98      // 参数：     (传入) AdjList G 图结构体
99      //           (传入) int n,图结点数
100      //          (传入) int id,遍历方法选择 (1 深度遍历 2 广度遍历)
101      // 功能：    图的遍历
102      /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
103      void Traver(AdjList g,int n, int id)
104      {
105          int i;
106          static int visited[MAX];
107          for(i =1; i <=n; i ++)
108              visited[i] =0;
109          for(i =1; i <=n; i ++)
110              if(visited[i] ==0)
111              {
112                  if(id ==1)    DFS(g, i, visited);
113                  if(id ==2)    BFS(g, i, visited);
114              }
115      }
116
117      void main()
118      {
119          ALGraph G;
120            int id;
121          printf("输入结点数和边数:");
122          scanf("%d %d", &G.vexnum, &G.arcnum);
123          CreateGraph(G);
124
125            printf("选择深度遍历或者广度遍历:(1 DFS, 2 BFS)");
126            scanf("%d", &id);
127          Traver(G.vertices, G.vexnum, id);
128      }
```

第 4 步：测试数据。

测试数据如图 6-5 所示。

运行结果：

```
输入结点数和边数:6 10
第 1 个结点信息(char 型):a
第 2 个结点信息(char 型):b
第 3 个结点信息(char 型):c
第 4 个结点信息(char 型):d
第 5 个结点信息(char 型):e
第 6 个结点信息(char 型):f
第 1 条边 - - - -起点序号,终点序号:1 2
```

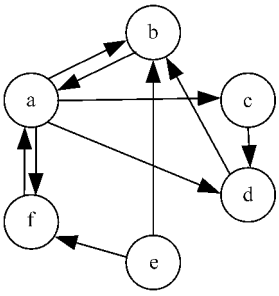


图 6-5　测试数据的图结构

```
第 2 条边 - - - - 起点序号,终点序号:2 1
第 3 条边 - - - - 起点序号,终点序号:1 3
第 4 条边 - - - - 起点序号,终点序号:1 4
第 5 条边 - - - - 起点序号,终点序号:3 4
第 6 条边 - - - - 起点序号,终点序号: 4 2
第 7 条边 - - - - 起点序号,终点序号:1 6
第 8 条边 - - - - 起点序号,终点序号:6 1
第 9 条边 - - - - 起点序号,终点序号:5 6
第 10 条边 - - - - 起点序号,终点序号:5 2
选择深度遍历或者广度遍历:(1 DFS, 2 BFS)
2
1 6 4 3 2 5
```

思考题：

根据以上图的遍历实现算法，编写算法判别以邻接表方式存储的有向图中是否存在由顶点 V_i 到顶点 V_j 的路径 $(i \neq j)$，并完成程序进行验证。

6.3　图结构应用

【实验 6.3】　拓扑排序的设计与实现

有向图的拓扑排序。

第 1 步：任务分析。

有向图的拓扑排序操作涉及有向图的建立，拓扑排序法，图的遍历等操作，请先熟悉有向图的邻接表表示法，本实验对有向图采用邻接表建立。

第 2 步：程序构思。

在有向图的邻接表表示法的基础上，本实验对有向图采用邻接表建立，并实现有向图的建立，拓扑排序法，图的遍历等操作。

拓扑排序的实现步骤如下：

⑴ 在 AOV 网中选择一个入度为 0 的顶点且输出之；

⑵ 从 AOV 网中删除此顶点及以该顶点为尾的所有有向边；

⑶ 重复 ⑴ 和 ⑵ 两步，直到 AOV 网中所有顶点都被输出或网中不存在入度为 0 的

顶点。

从拓扑排序步骤可知，若在第 (3) 步中，网中所有顶点都被输出，则表明网中无有向环，拓扑排序成功。若仅输出部分顶点，网中已不存在入度为 0 的顶点，则表明网中含有向环，拓扑排序失败。

具体的算法流程请参见图 6-6 拓扑排序流程图。需要注意的是在实现拓扑排序之前，需预先得到顶点的入度，否则无法进行下一步。本实验需要掌握的图的基本操作有有向图的邻接表建立，邻接表顶点出度入度计算，有向图的遍历等。

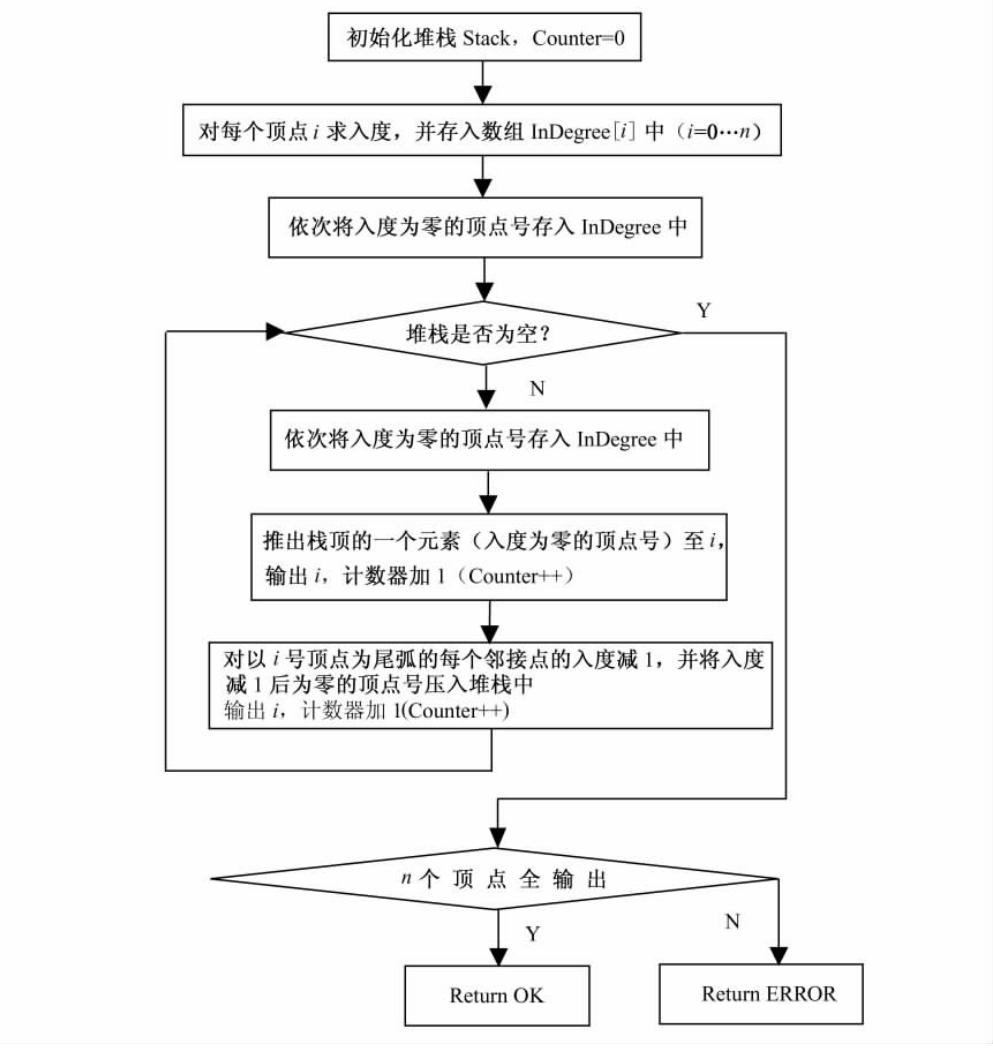


图 6-6　拓扑排序流程图

第 3 步：源程序。

```
1          #include <stdio.h>
2          #include <stdlib.h>
3          #define MAX 50
4          typedef struct ArcNode
```

```
{
        int adjvex;    //弧邻接的顶点
        char info;     //弧相关信息
        struct ArcNode *nextarc;    //指向下一条弧的指针
}ArcNode;

typedef struct Vnode
{
        char data;          //结点信息
        int inDegree;       //入度
        int OutDegree;      //出度
        ArcNode *link;      //指向第一个和该顶点相连的弧
}Vnode, AdjList[MAX+1];

//图结构体
typedef struct
{
        AdjList vertices;
        int vexnum;    //顶点数
        int arcnum;    //边数
}ALGraph;

/*****------------------------------*****/
// 函数名：  CreateGraph(ALGraph &G)
// 参数：    (传入) ALGraph G 图结构体
// 功能：    建立图的邻接表表示,并计算了顶点的入度
/*****------------------------------*****/
void CreateGraph(ALGraph &G)
{
    int i, j, s, d;
    ArcNode *p;
    ArcNode *t;

    //输入结点信息
    for(i=1; i<=G.vexnum; i++)
    {
        getchar();
        printf("第%d个结点信息(char型):", i);
        scanf("%c", &G.vertices[i].data);
        G.vertices[i].inDegree=0;
        G.vertices[i].OutDegree=0;
        G.vertices[i].link=NULL;
    }
```

```
48
49          //输入边的信息
50          for(i =1; i <=G.arcnum; i ++)
51          {
52              printf("第%d 条边 - - - -起点序号,终点序号:", i);
53              scanf("%d %d", &s, &d);
54              p = (ArcNode * )malloc(sizeof(ArcNode));
55              p -> adjvex = d;
56              p -> info = G.vertices[d].data;     //存储边的权值等信息,此处以顶
                                                    //点数据代替
57              p -> nextarc = G.vertices[s].link;    //p 插入顶点 s 的邻接表中
58              G.vertices[s].link =p;
59          }
60
61          //计算顶点的入度
62          for (i =1; i <=G.vexnum; i ++)
63          {
64              int counter =0;
65              for (j =1; j <=G.vexnum; j ++)   //求顶点 i 的入度要遍历整个邻
                                                 //接表
66              {
67                  if(j !=i)           //顶点 i 的邻接链表不必计算
68                  {
69                      t =G.vertices[j].link;    //取顶点 i 的邻接表
70                      while (t)//遍历整个邻接表
71                      {
72                          if ( t -> adjvex ==i)    counter ++;
73                          t =t -> nextarc;
74                      }
75                  }//if
76              }//for
77              G.vertices[i].inDegree =counter;
78           }
79      }
80
81      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -    * * * * */
82      // 函数名:   OutputGraph(ALGraph &G)
83      // 参数:     (传入) ALGraph G 图结构体
84      // 功能:     输出图的邻接表表示
85      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -    * * * * */
86      void OutputGraph(ALGraph &G)
```

```
87      {
88              int i;
89              ArcNode *p;
90              printf("遍历图的结果如下:\n");
91              for(i=1; i<=G.vexnum; i++)
92              {
93                  printf("[%c(%d)]", G.vertices[i].data, G.vertices
                    [i].inDegree);
94                  p=G.vertices[i].link;
95                  while (p!=NULL)
96                  {
97                      printf("--->(%d %c)", p->adjvex, p->info);
98                      p=p->nextarc;
99                  }
100                 printf("\n");
101             }//for
102     }
103
104     /*****------------------------------*****/
105     // 函数名:  TopoSort(ALGraph &G)
106     // 参数:    (传入) ALGraph G 邻接表
107     // 功能:    拓扑排序
108     /*****------------------------------*****/
109     void TopoSort(ALGraph &G)
110     {
111             int i, j, k;
112             ArcNode *p;
113             int ghead=0;    //ghead 是入度为 0 的顶点链表的头指针
114             int m=0;    //m 指示输出的顶点个数
115                 for(i=1; i<=G.vexnum; i++)// 建立入度为 0 的顶点链表
116                 {
117                 if(G.vertices[i].inDegree==0)
118                 {
119                     G.vertices[i].inDegree=ghead;
120                     ghead=i;
121                     while(ghead!=0)//while1
122                     {
123                         j=ghead;
124
125                         //在链表中删除入度为 0 的顶点,顶点序号为 j
126                         ghead=G.vertices[i].inDegree;
127                         p=G.vertices[j].link;
128                         printf("%c ", j);    // 输出顶点 j 并计数
```

```
129                         m ++;
130                         while(p != NULL)//while2
131                         {
132                             k = p -> adjvex;
133
134                             //以顶点 k 为尾的弧头的入度减 1
135                             G.vertices[k].inDegree = G.vertices[k].in-
                                Degree - 1;
136                             if (G.vertices[k].inDegree == 0)
137                             {
138                                G.vertices[k].inDegree = ghead;//新顶点
                                                                入链,
                                                                //入 度
                                                                为 0
139                                p = p -> nextarc;   // 找下一条弧
140                             }
141                         }//while2
142                     }//while1
143                 }//if
144             }//for
145             if (m < G.vexnum)  printf("网中有环! \n"); // 输出顶点数不足
n
146     }
147
148     void main()
149     {
150             ALGraph G;
151
152             printf("输入结点数和边数:");
153             scanf("%d %d", &G.vexnum, &G.arcnum);
154
155             CreateGraph(G);
156             TopoSort(G);
157             OutputGraph(G);
158     }
```

第 4 步：测试数据。

测试数据如图 6-7 所示。

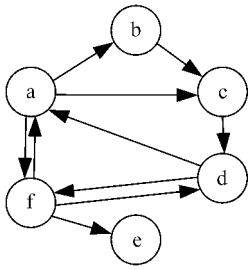


图 6-7　拓扑排序的图结构

运行结果：

```
输入结点数和边数:6 10
第 1 个结点信息(char 型):a
第 2 个结点信息(char 型):b
第 3 个结点信息(char 型):c
第 4 个结点信息(char 型):d
第 5 个结点信息(char 型):e
第 6 个结点信息(char 型):f
第 1 条边——起点序号,终点序号:1 2
第 2 条边——起点序号,终点序号:2 3
第 3 条边——起点序号,终点序号:3 4
第 4 条边——起点序号,终点序号:4 6
第 5 条边——起点序号,终点序号:6 4
第 6 条边——起点序号,终点序号:6 5
第 7 条边——起点序号,终点序号:1 3
第 8 条边——起点序号,终点序号:4 1
第 9 条边——起点序号,终点序号:1 6
第 10 条边——起点序号,终点序号:6 1
网中有环!
遍历图的结果如下：
[a(2)] - - -> (6 f) - - -> (3 c) - - -> (2 b)
[b(1)] - - -> (3 c)
[c(2)] - - -> (4 d)
[d(2)] - - -> (1 a) - - -> (6 f)
[e(1)]
[f(2)] - - -> (1 a) - - -> (5 e) - - -> (4 d)
```

思考题：

1. 图的邻接表表示是不唯一的，请通过上述程序验证。

2. 在有向图的邻接表中，求顶点的出度容易，只要简单地在该顶点的邻接点链表中查结点个数即可。请修改建立有向图邻接表的函数，为每一结点计算其出度，放置在 OutDegree 中。

3. 请参考教材关于图的遍历算法，更改上述程序中关于图的遍历的函数 OutputGraph（），

分别实现图的深度优先搜索和广度优先搜索。

【实验 6.4】　最短路径的设计与实现

最短路径求解。

第 1 步：任务分析。

最短路径求解包括迪杰斯特拉 (Dijkstra) 算法的实现，图的邻接数组表示等。

第 2 步：程序构思。

迪杰斯特拉算法的基本思想如下。

(1) 初始化数组 dist、path、S。其中 dist [i] = cost [0] [i]，设S 为已找到的从源点出发的最短路径上的终点集合，则初始时S 为空集。path [i] 均为空 (i = 1…n)。

(2) 选择当前从 v_0 出发的一条最短路径的终点 v_i，即：dist [i] = min { dist [i] | $v_i \in V - S$}。

(3) 将 v_i 并入集合S 中，即令S = S ∪ {i}，将 v_i 并入集合 path [i] 中。

(4) 修改从 v_0 出发到集合V - S 上所有顶点 v_k 可达的最短路径长度。修改规则如下：如果 dist [i] + cost [i] [k] < dist [k]，则令 dist [k] = dist [i] + cost [i] [k]。

(5) 重复步骤 (2) - (4) 直到所有顶点均在 S 中为止，即可求得从 v_0 到图中其余顶点的最短路径。

参照上述算法思想，程序实现做如下处理：

(1) 把图形建立成加权边图的邻接数组表示；

(2) 申明两个一维数组来来分别记录已查找的数组和顶点间距离总和的变化；

(3) 取原先的距离总和与新键入顶点后的距离总和的最小值作为新顶点的距离总和，计算距离总和的变化情况。

第 3 步：源程序。

```
1     #include < stdio.h >
2     #define MAX 9999
3     #define MAXVertex 100
4     typedef char VexType;
5     typedef float AdjType;
6
7     typedef struct
8     {
9        int n;                                    // 图的顶点个数
10       VexType vexs[MAXVertex];                  //顶点信息
11       AdjType arcs[MAXVertex][MAXVertex]; //边信息
12    }GraphMatrix;
13
14    typedef struct
15    {
16       VexType vertex;                           //顶点信息
17       AdjType length;                           //最短路径长度
```

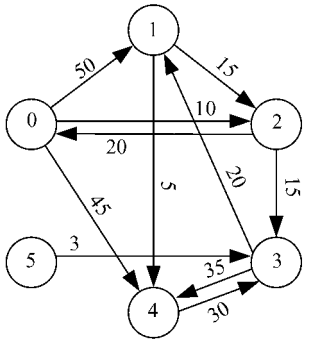
```
    int prevex;                                  //v_i 的前趋顶点
}Path;

Path dist[6];                                    //距离总和

/*****------------------------------------*****/
// 函数名：  init(GraphMatrix &graph, Path dist[])
// 参数：    (传入) GraphMatrix graph   图结构体
//           (传入) Path dist[]   距离总和
// 功能：    初始化集合中的距离值
/*****------------------------------------*****/
void init(GraphMatrix graph, Path dist[])
{
    int i;
    dist[0].length =0;
    dist[0].prevex =0;
    dist[0].vertex =graph.vexs[0];

    graph.arcs[0][0] =1;      //表示顶点 v0 在集合 U 中

    //初始化集合 V-U 中顶点的距离值
    for(i =1; i <graph.n; i ++)
    {
        dist[i].length =graph.arcs[0][i];
        dist[i].vertex =graph.vexs[i];
        if (dist[i].length !=MAX)
            dist[i].prevex =0;
        else  dist[i].prevex = -1;
    }
}

/*****------------------------------------*****/
// 函数名：  dijkstra(GraphMatrix graph, Path dist[])
// 参数：    (传入) GraphMatrix graph   图结构体
//           (传入) Path dist[]   距离总和
// 功能：    Dijkstra 算法求最短路径
/*****------------------------------------*****/
void dijkstra(GraphMatrix graph, Path dist[])
{
    int i,j,minvex;
    AdjType min;
    init(graph,dist); //初始化,此时集合 U 中只有顶点 v0
    for(i =1; i < graph.n; i ++)
```

```
61        {
62            min = MAX;
63            minvex = 0;
64            for (j = 1; j < graph.n; j ++) //在 V - U 中选出距离值最小顶点
65            {
66                if(graph.arcs[j][j] == 0 && dist[j].length < min)
67                {
68                    min = dist[j].length;
69                    minvex = j;
70                }
71
72                //从 v0 没有路径可以通往集合 V - U 中的顶点
73                if(minvex == 0) break;
74                graph.arcs[minvex][minvex] = 1;
75                //调整集合 V - U 中的顶点的最短路径
76                for (j = 1; j < graph.n; j ++)
77                {
78                    if(graph.arcs[j][j] == 1)  continue;
79                    if(dist[j].length > dist[minvex].length + graph.arcs
                      [minvex][j])
80                    {
81                        dist[j].length = dist[minvex].length + graph.arcs
                          [minvex][j];
82                        dist[j].prevex = minvex;
83                    }
84                }
85            }
86        }
87    }
88
89    /* * * * * - - - - - - - - - - - - - - - - - - - - - - - - -    * * * * * /
90    // 函数名：   initgraph(GraphMatrix &graph, int n)
91    // 参数：     (传入) GraphMatrix graph 图结构体
92    // 功能：     建立图的邻接矩阵表示
93    /* * * * * - - - - - - - - - - - - - - - - - - - - - - - - -    * * * * * /
94    void initgraph(GraphMatrix &graph, int n)
95    {
96        int i,j;
97        for (i = 0; i < graph.n; i ++)
98            for (j = 0; j < graph.n; j ++)
99                graph.arcs[i][j] = (i == j ? 0 : MAX);
100           graph.arcs[0][1] = 50;
101           graph.arcs[0][2] = 10;
```

```
102          graph.arcs[1][2]=15;
103          graph.arcs[1][4]=5;
104          graph.arcs[2][0]=20;
105          graph.arcs[2][3]=15;
106          graph.arcs[3][1]=20;
107          graph.arcs[3][4]=35;
108          graph.arcs[4][3]=30;
109          graph.arcs[5][3]=3;
110          graph.arcs[0][4]=45;
111    }
112
113    //测试主程序
114    void main()
115    {
116        int i;
117       GraphMatrix graph;
118
119        initgraph(graph, 6);//6 为图中顶点个数
120        dijkstra(graph, dist);
121        for (i=0; i < graph.n; i++)
122           printf("(%.0f %d) ", dist[i].length,dist[i].prevex);
123    }
```

第 4 步：测试数据。

测试数据如图 6-8 所示。



从 0 到 0，最短路径 0，经过 0 个节点
从 0 到 1，最短路径 45，经过 3 个节点 (0,2,3)
从 0 到 2，最短路径 10，经过 0 个节点
从 0 到 3，最短路径 25，经过 2 个节点 (0,2)
从 0 到 4，最短路径 45，经过 0 个节点
从 0 到 5，无最短路径

图 6-8　查找最短路径的图结构

```
(0 0)   (45 3)   (10 0)   (25 2)   (45 0)   (100000000 -1)
```

思考题：

1. **结合程序理解加权边图的邻接数组表示。**
2. **思考实现求最短路径的弗洛伊德 (Floyd) 方法。**

6.4 小　　结

通过实验掌握图的有关概念和图的邻接矩阵存储表示和邻接表存储表示，掌握图的遍历算法及其简单应用，根据图的相关知识理解掌握最小生成树、最短路径、拓扑排序和关键路径等的基本思想等。

第 7 章　查　　找

本章的典型实验为查找操作基本实验，共 4 个实验内容。

7.1　知识点概述

1. 定义

查找是数据结构和计算机各种算法中最常用的基本算法。假设被查找的对象是存储在某种数据结构（如线性表）中的 n 个结点，在其中查找满足给定条件（如关键字 = m）的结点，如找到满足条件的结点，则称查找成功，否则称查找失败。

查找表：是由同一类型的数据元素（或记录）构成的集合，其中每个数据元素又由若干个数据项组成，并假设每个数据元素都有一个能唯一标识它的关键字。对查找表经常进行的操作有：查询某个“特定的”数据元素是否在查找表中；检索某个“特定的”数据元素的各种属性；在查找表中插入一个数据元素；从查找表中删去某个数据元素。

静态查找表：若对查找表只对“特定的”数据元素进行查找，则称此类查找表为静态查找表。

动态查找表：若在查找过程中同时插入查找表中不存在的数据元素，或者从查找表中删除已存在的某个数据元素，则称此类表为动态查找表。

查找：在一个含有众多的数据元素（或记录）的查找表中找出某个“特定的”数据元素（或记录）。

关键字：关键字是数据元素（或记录）中某个数据项的值，用它可以标识（识别）一个数据元素（或记录）。若此关键字可以唯一地标识一个记录，则称此关键字为主关键字。反之，称用以识别若干记录的关键字为次关键字。

查找成功：根据给定的某个值，在查找表中确定一个其关键字等于给定值的记录或数据元素，若表中存在这样的一个记录，则称查找是成功的，此时查找的结果为给出整个记录的信息，或指示该记录在查找表中的位置。

查找不成功：若表中不存在关键字等于给定值的记录，则称查找不成功，此时查找的结果可给出一个“空”记录或“空”指针。

例如，当用计算机处理大学入学考试成绩时，全部考生的成绩可以用教材中表 8-1 所示表的结构储存在计算机中，表中每一行为一个记录，考生的准考证号为记录的关键字。

平均查找长度（Average Search Length）：查找运算的主要操作是关键字的比较，所以通常把查找过程中对关键字需要执行的平均比较次数称为平均查找长度，作为衡量一个查找算法效率优劣的标准。

2. 静态查找表

⑴ 顺序表的查找。顺序查找的基本思想是：从表的一端开始，顺序扫描线性表，依次将扫描到的结点关键字和给定值 K 相比较。若当前扫描到的结点关键字与 K 相等，则查找

成功;若扫描结束后,仍未找到关键字等于 K 的结点,则查找失败。

⑵ 有序表的查找。如果顺序表中的数据元素按关键字有序排列,即以有序表表示静态查找表时,则可进行“折半查找”,又称二分查找。其基本思想如下。

① 确定查找区间 R [low,high] 的中点位置 mid = (low + high) /2。

② 将待查的 K 值与 R [mid] . key 比较:若相等,则查找成功并返回此位置,否则须确定新的查找区间,继续二分查找。若 R [mid] . key > K,则确定新的查找区间是左子表 R [low , mid - 1]。若 R [mid] . key < K,则确定新的查找区间是右子表 R [mid + 1,high]。

③ 重复②直至找到关键字为 K 的结点,或者直至当前的查找区间为空 (即查找失败) 时为止。

⑶ 静态树表的查找。已知一个按关键字有序的记录序列 ($r_1, r_{1+1}, \cdots, r_h$),其中:$r_1.key < r_{1+1}.key < \cdots < r_h.key$,与每个记录相应的权值为:$w_1, w_{1+1}, \cdots, w_h$,现构造一个二叉树,使这棵二叉树的带权内路径长度 PH 值在所有具有相同权值的二叉树种近似为最小,称这类二叉树为次优查找树。构造次优查找树的方法是:首先在待查找的有序序列中取第 i 个记录构造根结点,使得:

$$\Delta P_i = \left| \sum_{j=i+1}^{h} W_j - \sum_{j=1}^{i-1} W_j \right|$$

取最小值,然后分别对子序列 $\{r_1, r_{1+1}, \cdots, r_{i-1}\}$ 和 $\{r_{i+1}, \cdots, r_h\}$ 构造两棵次优查找树,并分别设为根结点 r_i 的左右子树。重复上述步骤直至将所有的结点均构造到次优查找树中。

⑷ 分块查找。分块查找又称索引顺序查找。它的基本思想是:把一个有 n 个记录的线性表划分为若干个块,另建一个索引表记录每个块最后一个记录的地址号及该块中最大的关键字值。在这里,每块中的元素可以是无序的,但要保证第 i 块的最大值不大于第 i + 1 块的最小值。当需要查找该表中某一个特定记录时可分两步完成。第一步找索引表确定该记录应在哪一块中,第二步根据第一步的结果在块中进行顺序查找。

3. 动态查找表

⑴ 二叉排序树。二叉排序树 (又称二叉查找树) 或者是一棵空树,或者是一棵同时满足下列条件的二叉树:若它的左子树不空,则左子树上所有结点的键值均小于根结点的键值;或若它的右子树不空,则右子树上所有结点的键值均大于根结点的键值;或它的左、右子树也分别为二叉排序树。

二叉排序树的查找过程为:即当二叉排序树不空时,首先将给定值和根结点的关键字比较,若相等,则查找成功,否则将依据给定值和根结点的关键字之间的大小关系,分别在左子树或右子树上继续进行查找。

⑵ 平衡二叉树。一棵平衡二叉排序树 (简称 AVL 树) 或者是一棵空树,或者是一棵任一结点的左子树与右子树的高度至多差 1 的二叉排序树。当一棵二叉排序树在插入新结点的过程中如果出现了不平衡,经过调整后仍能保持二叉排序树的特性。查找过程与二叉排序树的类似。

⑵ B - 树和 B + 树。当查找的文件较大,且存放在磁盘等直接存取设备中时,为了减少查找过程中对磁盘的读写次数,提高查找效率,基于直接存取设备的读写操作其特征是以“页”为单位。

B - 树的查找过程 (B + 树与此类似) 为:首先在树中查找 K,若找到则直接返回 (假设不

处理相同关键字的插入）；否则查找操作必失败于某个叶子上，然后将 K 插入该叶子中。若该叶子结点原来是非满（指 keynum < Max，即结点中原有的关键字总数小于 m - 1）的，则插入 K 后并未破坏 B - 树的性质，故插入 K 后即完成了插入操作；若该结点原为满，则 K 插入后 keynum = m，违反 B - 树性质（3），故须调整使其维持 B - 树性质不变。

4. 哈希查找

根据设定的哈希函数 H（key）和所选中的处理冲突的方法，将一组关键字映射到一个有限的、地址连续的地址集（区间）上，并以关键字在地址集中的“像”作为相应记录在表中的存储位置，这种表被称为哈希表，这一映射的过程亦被称为“散列”。

构造哈希函数的方法有：直接定址法、数字分析法、平方取中法、折叠法、除留余数法和随机数法。

在哈希表的建表过程中，若对于某个哈希函数，两个或两个以上的关键字映射的哈希地址相同，即 H（key1）= H（key2）（key1 ≠ key2），则发生冲突。常用的处理冲突的方法有开放地址法和链地址法。

在哈希表上进行查找的过程和哈希造表的过程基本一致。给定 Key 值，根据造表时设定的哈希函数求得哈希地址，若表中此位置上为空，没有记录，则查找不成功；否则比较关键字，若和给定值相等，则查找成功；否则根据造表时设定的处理冲突的方法找“下一地址”，直至哈希表中某个位置为“空”或者表中所填记录的关键字等于给定值时为止。

7.2　查 找 实 验

【实验 7.1】　顺序查找的设计与实现

顺序查找的实现。

第 1 步：任务分析。

查找实验包含基本查找函数的实现和改进的查找算法实现。

第 2 步：程序构思。

顺序查找的基本思想是：从表的一端开始，顺序扫描线性表，依次将扫描到的结点关键字和给定值 K 相比较。若当前扫描到的结点关键字与 K 相等，则查找成功；若扫描结束后，仍未找到关键字等于 K 的结点，则查找失败。

为了更好的理解查找技术的用关键字查找数据项内容，此段程序定义的数组元素的类型如下：

```
typedef struct
{
char key;        // 关键字项
int  data;       // 其他数据项
}SequenList;     // 数据元素类型
```

说明：在后续的查找技术中，程序使用的关键字为数组的下标，不再单独定义关键字，以方便程序的书写和阅读。

第 3 步：源程序。

```
#include "stdio.h"
/* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
// 函数名：  SequenSearch(SequenList L[], int n, char key)
// 参数：    (传入) SequenList L[],顺序表 L(结构体数组)
//           (传入) int n,查找范围
//           (传入) char key,查找的关键字
// 返回值：  int 型返回查找到的位置，-1 表示没有找到
// 功能：    顺序表查找
/* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
int SequenSearch(SequenList L[], int n, char key)
{
    int i =1;
    int position;    //查找到的位置
    int counter =1;  //查找次数积数
    while ( i <=n )
    {
        if ( L[i -1].key == key )    //查找到数据时候,记录此位置
        {
            position =i;
            break;
        }
        else    position = -1;

        i ++;
        counter ++;
    }
    printf("Search times =%d\n", counter);//输出查找次数

    return position;
}
   void main()
   {
      SequenList s[4];
      char temp;
      s[0].key ='b';    s[0].data =111;
      s[1].key ='c';    s[1].data =222;
      s[2].key ='a';    s[2].data =333;
      s[3].key ='d';    s[3].data =444;
      printf("please input the key value('a','b','c','d'):\n");
      scanf("%c", &temp);//输入关键字
      int pos =SequenSearch(s, 4, temp);
```

```
42          if(pos!= -1)
43          //输出位置和内容
44          printf("the position is %d, it's value is %d\n", pos, s[pos -1].
            data);
45          else printf("cannot found! \n");
46      }
```

第 4 步：测试数据。

运行结果：

```
please input the key value('a','b','c','d'):
d
Search times =4
the position is 4, it's value is 444
```

思考题：

请结合教材关于顺序查找的改进算法实现其函数。参考函数说明如下：

```
1       int SequenSearch(SequenList L[], int n, char key)
2       {
3         int i =1;
4         int position;       //查找到的位置
5         L[n -1].key = key;  //在数组的最后一位设置监视哨
6         while ( 1 )
7         {
8            if ( L[i -1].key == key )    //查找到数据时候
9            {
10               //找到的位置是不是设置的监视哨,不是的话,
11               //表示找到数据,反之,表示没有查找到数据
12               if (n -1 != i -1)
13               {
14                  position = i;
15                  break;
16               }
17               else return -1;
18           }
19           else    position = -1;
20
21           i ++;
22           counter ++;
23        }
24
25        return position;
26    }
```

备注：使用改进的顺序查找算法的时候由于使用了监视哨，所以在定义数组的时候需要

给监视哨预留存储位置。本段函数是在高下标处设置,所以数组定义为 L [0] - L [n - 1],在 L [n] 处设置监视哨,此时数组长度为 n +1。

【实验 7.2】　折半查找的设计与实现

折半查找的实现。

第 1 步:任务分析。

如果顺序表中的数据元素按关键字有序排列,即以有序表表示静态查找表时,则可进行“折半查找”,又称二分查找。本节在上节的基础上改进实现折半查找。

第 2 步:程序构思。

基本思想是:

(1) 首先确定查找区间 R [low,high] 的中点位置 mid = (low + high) /2:

(2) 然后将待查的 K 值与 R [mid] . key 比较:

- 若 k == r [mid] . key,查找成功
- 若 k < r [mid] . key,则 high = mid - 1
- 若 k > r [mid] . key,则 low = mid +1

(3) 重复上述操作,或查找成功,或直至 low > high 时,查找失败。

第 3 步:源程序。

```
#include <stdio.h>
#define N 10

//数据结构定义:
typedef struct
{
    int *elem;
    int length;
}SSTable;
/*****************************  *****/
// 函数名:   Search_Bin ( SSTable ST, int key )
// 参数:     (传入) SSTable ST,顺序表 L
//           (传入) int key,查找的关键字
// 返回值:   int 型,若找到,则函数值为该元素在表中的位置,否则为 0
// 功能:     在有序表 ST 中折半查找其关键字等于 key 的数据元素
/*****************************  *****/
int Search_Bin ( SSTable &ST, int key )
{
    int counter =1;        //查找次数
    int low =1;            //左边界变量
    int mid;               //中间位置变量
    int high =ST.length;   //右边界变量,置区间初值
    while (low <=high)
    {
```

```
25              mid = (low + high) / 2;
26              if (key == ST.elem[mid])
27                  return mid;                    //找到待查元素
28              else
29              {
30                  if (key < ST.elem[mid])
31                      high = mid - 1;//继续在前半区间进行查找
32                  else
33                      low = mid + 1;        //继续在后半区间进行查找
34              }
35              counter ++;
36          }
37          printf("Search times = %d\n", counter);//输出查找次数
38
39          return 0;                              //顺序表中不存在待查元素
40      }
41
42      void main()
43      {
44          int kvalue;//待查找的变量
45
46          SSTable L;
47          L.elem = (int *)malloc(N * sizeof(int));
48          L.length = N;
49          for(int i = 0; i < N; i ++)    scanf("%d", &L.elem[i]);
50
51          printf("input a key value:\n");
52          scanf("%d", &kvalue);
53          int pos = Search_Bin(L, kvalue);
54          if(pos) printf("position is %d, the value is %d\n", pos +1, L.
            elem[pos]);
55          else printf("not found! \n");
56      }
```

第 4 步：测试数据。

运行结果：

```
input data array:
11 22 33 44 55 66 77 88 99 100
input a key value:
66
position is 6, the value is 66
```

思考题：

试着使用递归方式实现折半查找程序。参考函数如下：

```
int Search_Recur ( SSTable &ST, int low, int high, int key )
{
    int mid;        //中间位置变量
    if(low > high)    return 0;  //递归结束
    else
    {
        mid = (low + high) / 2;
        if (key == ST.elem[mid])
            return mid;           //找到待查元素
        else
        {
            if (key < ST.elem[mid])
                //继续在前半区间进行查找
                return Search_Recur(ST, low, mid -1, key);
            else
                //继续在后半区间进行查找
                return Search_Recur(ST, mid +1, high, key);
        }
    }

    return 0;        //顺序表中不存在待查元素
}
```

备注：调用此函数时候需首先指定左边界（初始为 1）和右边界（初始为数组的长度 length）。

思考：如果要统计递归方式下折半查找的次数，如何在程序中加上查找次数计数器 counter。

【实验 7.3】　二叉排序树的设计与实现

二叉排序树的实现。

第 1 步：任务分析。

二叉排序树是一种特殊的、增加了限制条件的二叉树（因此它的存储结构及其类型定义与二叉树相同）。

第 2 步：程序构思。

二叉排序树的查找过程如下：即当二叉排序树不空时，首先将给定值和根结点的关键字比较，若相等，则查找成功，否则将依据给定值和根结点的关键字之间的大小关系，分别在左子树或右子树上继续进行查找。

第 3 步：源程序。

```
#include <stdio.h>
typedef struct bstnode
```

```
    {
        int key;
        struct bstnode *lchild, *rchild;
    }BSTNode;
    /*****------------------------*****/
    // 函数名：  Search(BSTNode* bt, int key)
    // 参数：    (传入) BSTNode* bt,二叉查找树
    //           (传入) int key,查找的关键字
    // 返回值：  BSTNode 指针,若找到,返回此结点,否则为 0
    // 功能：    在二叉查找树中查找其关键字等于 key 的数据元素
    /*****------------------------*****/
    BSTNode * Search(BSTree &bt, int key)
    {
        int counter=0;
        BSTree p;
        p=bt;
        while(p!=NULL)
        {
            counter++;
            if (p->key==key)return p; //查找成功,返回查找到的结点
            else
            {
                if (key < p->key)p=p->lchild; //在左子树中继续查找
                else p=p->rchild; //在右子树中继续查找
            }
        }
        printf("Search times=%d\n", counter);//输出查找次数

        return 0; //查找失败
    }
    /*****------------------------*****/
    // 函数名：  CreateTree(BSTree &root, int data[])
    // 参数：    (传入) BSTree &root,二叉查找树根结点
    //           (传入) int data[],数据
    // 返回值：  无
    // 功能：    建立二叉查找树
    /*****------------------------*****/
    void CreateTree(BSTree &root, int data[])
    {
    BSTree p;//新建结点
    BSTree current;//当前结点
    BSTree father;//父结点
    int i;
```

```
46
47      for(i =0; i < N; i ++)
48      {
49          //创建一个新结点,并指定结点内容,左子树和右子树置空
50          p = (BSTree)malloc(sizeof(BSTNode));
51          p -> key = data[i];
52          p -> lchild = NULL;
53          p -> rchild = NULL;
54
55          if(root == NULL)root = p;//根结点为空
56          else
57          {
58              current = root;//目前的位置在根结点
59              while (current != NULL)//当前结点为最底端时结束循环
60              {
61                  father = current;//记录父结点
62
63                  //当前结点数据大于等于输入数据,则此数据送往左子树
64                  if(current -> key > = data[i])
65                      current = current -> lchild;
66                  //当前结点数据小于输入数据,则此数据送往右子树
67                  else
68                      current = current -> rchild;
69              }
70
71                  //连起父与子结点
72                  if (father -> key > data[i])
73                      father -> lchild = p;
74                  else father -> rchild = p;
75              }
76          }
77      }
78      void main()
79      {
80          BSTree root = NULL;
81          BSTree p;
82          int data[N];
83          int key;
84          printf("input the data of the tree:\n");
85          for(int i =0; i <N; i ++)
86          {
87              scanf("%d", &data[i]);
88          }
```

```
89            CreateTree(root, data);
90
91            printf("input the key value:\n");
92            scanf("%d", &key);
93
94            p = Search(root, key);
95
96            if(p)printf("the founded value is %d\n", p -> key);
97            else printf("not found\n");
98        }
```

第 4 步：测试数据。

测试数据形成的二叉树如图 7-1 所示。

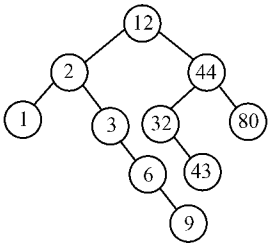


图 7-1　测试数据构成的二叉树

运行结果：

```
input the data of the tree:
12  2 3 44 32 43 6  80  9 1
input a key value:
12
search times = 1
the founded value is 12
```

【实验 7.4】　哈希查找的设计与实现

实现哈希查找程序。

第 1 步：任务分析。

实现哈希查找，包括首先要根据哈希函数构造哈希地址，然后根据哈希地址进行查找。

第 2 步：程序构思。

哈希表查找的过程为根据给定 Key 值，按照造表时设定的哈希函数求得哈希地址，若表中此位置上为空，没有记录，则查找不成功；否则比较关键字，若和给定值相等，则查找成功；否则根据造表时设定的处理冲突的方法找“下一地址”，直至哈希表中某个位置为“空”或者表中所填记录的关键字等于给定值时为止。

整个哈希查找过程可以描述如下：

(1) 选择合适的哈希函数 H (key) = key % p (p 为小于或等于哈希表长的最大质数，本程序取 13)；

⑵ 计算各个关键字的直接哈希函数值；

⑶ 根据处理冲突的方法建立哈希表，并输出；

⑷ 在哈希表中进行查找，输出查找的结果，以及所需和记录关键字比较的次数，并计算和输出在等概率情况下查找成功的平均查找长度。

如果用流程图表示，如图 7-2 所示。

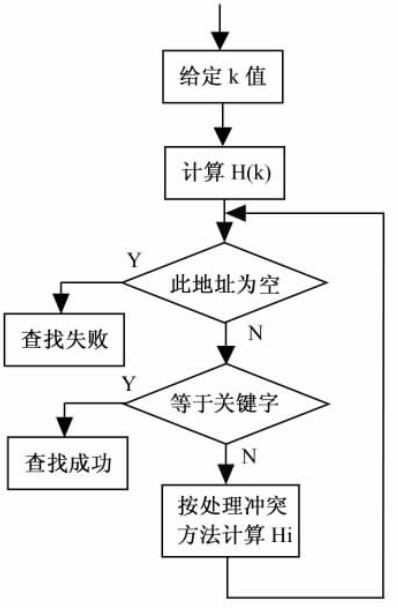


图 7-2　哈希查找的流程图

哈希表的构造：

有一组关键字 {19,01,23,14,55,20,84,27,68,11,10,77}，采用哈希函数 H (key) = key%13，采用线性探测再散列的方法解决冲突，在 0 - 18 的散列地址空间构造哈希表。

线性探测再散列的下一地址计算公式为：

d1 = H (key)

dj + 1 = (dj + 1) % m，j = 1，2，…

m = 19，进行计算如下：

H (19) = 19%13 = 6

H (01) = 01%13 = 1

H (23) = 23%13 = 10

H (14) = 19%13 = 1 (冲突)

H (14) = (1 + 1) %19 = 2

H (55) = 55%13 = 3

H (20) = 20%13 = 7

H (84) = 84%13 = 6 (冲突)

H (84) = (6 + 1) %19 = 7 (冲突)

H (84) = (7 + 1) % 19 = 8
H (27) = 27% 13 = 1 **(冲突)**
H (27) = (1 + 1) % 19 = 2 **(冲突)**
H (27) = (2 + 1) % 19 = 3 **(冲突)**
H (27) = (3 + 1) % 19 = 4
H (68) = 68% 13 = 3 **(冲突)**
H (68) = (3 + 1) % 19 = 4 **(冲突)**
H (68) = (4 + 1) % 19 = 5
H (11) = 11% 13 = 11
H (10) = 10% 13 = 10 **(冲突)**
H (10) = (10 + 1) % 19 = 11 **(冲突)**
H (10) = (11 + 1) % 19 = 12
H (77) = 77% 13 = 12 **(冲突)**
H (77) = (12 + 1) % 19 = 13

按照上述计算,得到各关键字对应的地址为：

address (01) = 1
address (14) = 2
address (55) = 3
address (27) = 4
address (68) = 5
address (19) = 6
address (20) = 7
address (84) = 8
address (23) = 10
address (11) = 11
address (10) = 12
address (77) = 13

其他地址为空。

第 3 步：源程序。

```
#include <stdlib.h>
#include <stdio.h>
typedef struct
{
      int key;
}HTable, *Htype;

/*****------------------------    *****/
// 函数名：  Hash(int key,int mod)
// 返回值：  哈希地址
// 功能：    构造除留余数法的哈希函数
```

```
12        // 备注：     为了使哈希函数简单,哈希函数 H(Key) = Key MOD 13
13        /* * * * - - - - - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
14        int Hash(int key,int mod)
15        {
16              return  key%13;
17        }
18
19        /* * * * - - - - - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
20        // 函数名：   Createhash(Htype &Hash, Htype ST, int n, int m)
21        // 参数：     (传入) Htype &ST 数据表
22        //            (传入) int n, int m 数据集合 ST 和哈希表的长度
23        // 功能：     根据数据表 ST,构造哈希表
24        /* * * * - - - - - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
25        void Createhash(Htype &HashTable, Htype ST, int n, int m)
26        {
27            int i,j;
28            HashTable = (Htype)malloc(m * sizeof(HTable));
29            for (i = 0;i < m;i ++)
30                HashTable[i].key = NULL;  //初始化哈希为空表
31            for (i = 0;i < n;i ++)
32            {
33                j = Hash(ST[i].key, m);    //获得直接哈希地址
34                while (HashTable[j].key! = NULL)
35                    j = (j + 1) %m;    //用线性探测再散列技术确定存放位置
36                HashTable[j].key = ST[i].key;    //将元素存入哈希表的相应位
置 * /
37            }
38        }
39
40        /* * * * - - - - - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
41        // 函数名：   Search(Htype &Hash, int key, int m,int times)
42        // 参数：     (传入) Htype 数据表
43        //            (传入) int key, int m     查找的关键字和哈希表的长度
44        //            (传出) int times     记录比较次数
45        // 返回值：   int 型,若查找成功,返回其在哈希表中的位置,否则返回 -1
46        // 功能：     在表长为 m 的哈希表中查找关键字等于 key 的元素
47        /* * * * - - - - - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
48        int Search(Htype HashTable, int key, int m, int &times)
49        {
50            int i;
51            times = 1;
52            i = Hash(key,m);
53            while (HashTable[i].key! = 0 && HashTable[i].key! = key)
```

```
54          {
55              i ++;
56              ++times;
57          }
58          if (HashTable[i].key ==0) return -1;
59          else return i;
60      }
61      void main()
62      {
63          Htype ST;
64          Htype HASH;
65          int key;
66          int i,n,m,times;
67          char ch;
68
69          //输入关键字集合元素个数和哈希表长
70          printf("Please input length of data:\n");
71          scanf("%d",&n);
72          printf("Please input length of hash table:\n");
73          scanf("%d",&m);
74
75          //输入关键字集合元素
76          ST = (Htype)malloc(n * sizeof(HTable));
77          printf("Please input %d data:\n",n);
78          for (i =0;i <n;i ++)
79              scanf("%d",&ST[i].key);
80          //建立哈希表
81          Createhash(HASH,ST,n,m);
82
83          //输出已建立的哈希表
84          printf("The hash table is\n");
85          for (i =0;i <m;i ++) printf("%5d",i);
86          printf("\n");
87          for (i =0;i <m;i ++) printf("%5d",HASH[i].key);
88
89          /*哈希表的查找,可进行多次查找*/
90          do {
91              printf("Input the key you want to search:\n");
92              scanf("%d",&key);
93              i =Search(HASH,key,m,times);
94
95              //查找成功
96              if (i!= -1)
```

```
97              {
98                  printf("the position is %d\n",i);
99                  printf("Search times is %d\n",times);
100               }
101               //查找失败
102               else
103               {
104                   printf("not found! \n");
105                   printf("Search times is %d\n",times);
106               }
107
108               //是否继续查找 yes or no
109             printf("Continue(y/n):\n");
110         }while ((ch=getchar())=='y' ||(ch=getchar())=='Y') ;
111     }
```

第 4 步：测试数据。

运行结果：

```
Please input length of data :
12
Please input length of hash table :
15
Please input 12 data:
19 14 23 1 68 20 84 27 55 11 10 79
The hash table is
   0    1    2    3    4    5    6    7    8    9    10   11   12
   0   14    1   68   27   55   19   20   84   79   23   11   10
Input the key you want to search:27
the position is 4
Search times is 4
Continue(y/n):
```

7.3 小　　结

通过实验，了解掌握常见查找算法的算法思路和程序实现，并了解常见查找算法的平均查找长度、查找效率等，对查找算法进行简单的特点总结归纳，体会时间复杂度等概念。

第 8 章　排　　序

本章的典型实验分为插入排序实验、交换排序实验和选择排序实验，共 6 个实验内容。

8.1　知识点概述

排序是将一个数据元素集合或序列重新排列成一个按数据元素某个项值有序的序列，通常是按照某种规则把数据的顺序重新整理。排序方法各种各样，主要有插入排序、交换排序、选择排序、归并排序、基数排序等。

1. 插入排序

插入排序（Insertion sort）的基本思想是：每次将一个待排序的记录，按其关键字插入到已排好序的序列的适当位置，直到全部记录插完为止。常用插入排序方法有 4 种：直接插入排序、折半插入排序、表插入排序和希尔排序。

⑴ 直接插入排序。直接插入排序是一种最简单的排序方法，算法的基本思想为：对 n 个等待排序的结点序列 $\{R_1, R_2, \cdots, R_n\}$，假定已有 i - 1 个结点 $R_1, R_2, R_3, \cdots, R_{i-1}$ 排好序，所以各结点的关键字之间存在不等式是：$K_1 <= K_2 <= \cdots <= K_{i-1}$，对下一个要插入的结点 R_i，首先将 R_i 的值送到存放到 R_0 中，然后用 R_0 的关键字值 K_0 依次与排好序的结点序列中的 $R_{i-1}, R_{i-2}, \cdots, R_0$ 的关键字进行比较，若 $K_0 < K_j$（$j = i - 1, i - 2, \cdots, 1$），则 R_j 后移一个位置，否则停止比较和移动，最后，将 R_0（即原来待插入的记录 R_i）移到 $j + 1$ 的位置上。

⑵ 折半插入排序。折半插入排序的算法思想与直接插入排序的算法类似，仅是在将要确定插入的元素的插入位置时，在已排好序的有序表中不断地用折半来确定插入位置，即一次比较，通过待插入记录与有序表居中的记录按关键码比较，将有序表一分为二，下次比较在其中一个有序子表中进行，将子表又一分为二，这样继续下去，直到要比较的子表中只有一个记录时，比较一次便确定了插入位置。

⑶ 表插入排序。表插入排序是通过链接指针，按关键码的大小，实现从小到大的链接过程，为此需增设一个指针项。操作方法与直接插入排序类似，所不同的是直接插入排序要移动记录，而表插入排序是修改链接指针。

⑷ 希尔排序。希尔排序又称“缩小增量排序”，它对直接插入排序进行了较大的改进，其算法基本思想为：先取定一个小于 n 的整数 d_1 作为第一个增量，把序列的全部记录分成 d_1 个组，将所有距离为 d_1 倍数的记录放在同一个组中，在各组内进行直接插入排序；然后取第二个增量 $d_2 < d_1$；不断重复上述分组和排序工作，直至所取的增量 $d_i = 1$（$d_i < d_{i-1} < \cdots < d_2 < d_1$），即到所有记录放在同一组中进行直接插入排序为止。

2. 交换排序

交换排序的基本思想是：两两比较待排序记录的关键字，发现两个记录的次序相反时即进行交换，直至没有反序的记录为止。常用有两种交换排序，即冒泡排序和快速排序。

⑴ 冒泡排序。冒泡排序算法的基本思想：对每两个相邻的关键字进行比较，若为逆序（即 $K_{j-1} > K_j$），则将 R_{j-1}、R_j 两个记录交换位置，这样的操作反复进行，直至全部记录都比较、交换完毕为上。如此经过一趟冒泡排序之后，就将关键字最大的记录安排在最后一个记录的位置上。然后，对后 n - 1 个记录重复进行同样的操作，则将具有次大关键字的记录安排在倒数二个记录的位置上。重复以上过程，直至没有记录需要交换为止。至此，整个序列的记录按关键字由小到大的顺序排列完毕。

⑵ 快速排序。快速排序的基本思想是：通过比较关键码、交换记录，以某个记录为界（该记录称为支点，该记录的关键字为 pivot - key），将待排序列分成两部分。其中，一部分所有记录的关键字大于等于支点记录的关键字，另一部分所有记录的关键字小于支点记录的关键字。我们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。对各部分不断划分，直至每一部分中只剩下一个记录为止，整个序列才能按关键字有序排列。

3. 选择排序

选择排序的基本思想是每一趟从待排序列中选取一个关键码最小的记录，也即第一趟从 n 个记录中选取关键码最小的记录，第二趟从剩下的 n - 1 个记录中选取关键码最小的记录，直到整个序列的记录选完。主要有三种选择排序方法，即直接选择排序、树形选择排序和堆排序。

⑴ 直接选择排序。直接选择排序的算法基本思想：通过比较，首先在所有记录中选出关键字最小的记录，把它与第一个记录交换，然后在其余的记录中再选出关键字次小的记录与第二个记录交换，依此类推，直至所有的记录成为有序序列。

⑵ 树形选择排序。树形选择排序又称锦标赛排序，它的具体做法是：首先对 n 个记录的关键字进行两两比较，取出 $\lceil n/2 \rceil$ 个较小的关键字作为比较结果保存下来，然后对这 $\lceil n/2 \rceil$ 个关键字再进行比较……如此重复，直至比较出最小的关键字为止。此过程可用一棵有 n 个叶结点的完全二叉树来表示。

⑶ 堆排序。堆排序的基本思想是：对一组待排序记录的关键字，首先把它们按堆的定义排成一个序列（称为建堆），从而输出堆顶的最小关键字（在此我们利用小根堆来排序）。然后将剩下的结点重新调整为一个堆，便得到次小的关键字。反复进行下去，直到只剩下一个结点为止，便能将全部关键字排成有序序列。

4. 归并排序

归并排序（Merge Sort）是利用“归并”技术来进行排序。所谓归并是指将若干个已排序的子文件合并成一个有序的文件。简单的归并是将两个有序的子文件合并成一个有序的文件。对于一个有 n 个待排序的无序序列，可以看成是由 n 个长度为 1 的有序子序列组成的序列，然后进行两两归并，得到 $\lceil n/2 \rceil$ 个长度为 2 或 1 的有序子序列，再两两归并……如此重复，直到最后形成包含 n 个记录的有序序列为止。这种总是反复将两个有序序列归并成一个有序序列的排序方法称为两路归并排序。

5. 基数排序

基数排序是按组成关键字的各个位的值来实现的排序的。采用基数排序法需要使用一批桶（或箱子），依次扫描待排序的记录，把关键字相同的记录全都装入到同一个桶中（分

配),然后按序号依次将各非空的桶首尾连接起来(收集)。所以这种方法又称为桶排序列。算法的基本思想:设置若干个桶(个数取决于关键字的取值范围),依次扫描待排序的记录 R[1],R[1],…,R[n],把关键字等于 k 的记录全都装入到第 k 个箱子里(分配),然后按序号依次将各非空的箱子首尾连接起来。

8.2　插入排序实验

【实验 8.1】　直接插入排序的设计与实现

实现直接插入排序程序

第 1 步:任务分析。

本节需要首先理解插入排序的基本思想,然后以数组为数据存储形式,实现直接插入排序。

第 2 步:程序构思。

插入排序(Insertion sort)的基本思想是:每次将一个待排序的记录,按其关键字插入到已排好序的序列的适当位置,直到全部记录插完为止。

第 3 步:源程序。

```
#include <stdio.h>
#define MAX 50

/*****************************   *****/
// 函数名:  InputData(int list[], int n) OutputData(int list[], int n)
// 功能:    建立待排序数据表,输出数据表内容
/*****************************   *****/
void InputData(int list[], int n)
{
    printf("input data:\n");
    for(int i=0; i<n; i++)
        scanf("%d", &list[i]);
}
void OutputData(int list[], int n)
{
    printf("\nthe current soring is:");
    for(int k=0; k<n; k++)
        printf("%d ", list[k]);
}

/*****************************   *****/
```

```
22      // 函数名：  InsertSort(int list[], int n)
23      // 参数：    (传入) int list[],待排序数组
24      //           (传入) int n,数组长度
25      // 功能：    使用插入排序对数据进行排序
26      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -  * * * * */
27      void InsertSort(int list[], int n)
28      {
29          int i, j;
30          int temp;
31          for(i =0; i <n; i ++)
32          {
33              temp = list[i];    //temp 是监视哨
34              j = i -1;
35              while (temp < list[j])    //进行元素移动,以便腾出一个位置插入
                                          //list[i]
36              {
37                  list[j +1] = list[j];
38                  j --;
39              }
40              list[j +1] = temp;    //在 j +1 位置处插入 list[i]
41              OutputData(list, n);  //将每趟排序结果输出
42          }
43      }
44
45      void main()
46      {
47          int num;
48          int list[MAX];
49          printf("input length of the list(n <50): \n");
50          scanf("%d", &num);
51
52          InputData(list, num);
53          InsertSort(list, num);
54      }
```

第 4 步：测试数据。

运行结果：

```
input length of the list(n <50):
6
input data:
23 43 4 6 2 12
```

```
the current soring is:23 43 4 6 2 12
the current soring is:23 43 4 6 2 12
the current soring is:4 23 43 6 2 12
the current soring is:4 6 23 43 2 12
the current soring is:2 4 6 23 43 12
the current soring is:2 4 6 12 23 43
```

思考题：

在熟悉掌握插入排序的基础上，结合下述折半插入排序算法描述，写出折半插入排序函数，并进行程序运行验证。

折半插入排序程序构思：

① low = 1；high = i - 1；　　// **有序表长度为** i - 1，**第** i **个记录为待插入记录**

　　//**设置有序表区间，待插入记录送辅助单元**

② **若** low > high，**得到插入位置，转**⑤

③ low≤high，m = (low + high) /2；

　　// **取表的中点，并将表一分为二，确定待插入区间**

④ **若** r [0] . key < r [m] . key，**则** high = m - 1；　　//**插入位置在低半区**

否则，low = m + 1；　　// **插入位置在高半区**

转②

⑤ high + 1 **即为待插入位置，从** i - 1 **到** high + 1 **的记录，逐个后移，然后将** r [0] **放置在** r [high + 1] **位置，完成一趟插入排序。**

【实验 8.2**】　希尔排序的设计与实现**

实现希尔排序程序。

第 1 **步：任务分析。**

按照希尔排序算法实现希尔排序。

第 2 **步：程序构思。**

希尔排序是先将待排序记录的序列按某个间隔长度分割成为若干个子序列分别进行直接插入排序，重复进行数据分割，每次分割的间隔长度逐步缩小，直到最后间隔长度为 1 **时，对全体记录进行一次直接插入排序，这时所有记录按关键字有序，排序结束。具体排序过程如下：**

⑴ **选择一个步长序列** $t_1, t_2, \cdots, t_k$，**其中** $t_i < t_i + 1$，$t_k = 1$；

⑵ **按步长序列个数** k，**对序列进行** k **趟排序；**

⑶ **每趟排序，根据对应的步长** t_i，**将待排序列分割成若干长度为** m **的子序列，分别对各子系列进行直接插入排序。当步长因子为** 1 **时，整个序列作为一个表来处理，表长度即为整个序列的长度。**

第 3 **步：源程序。**

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

```
#define MAX 50

/*****------------------------------*****/
// 函数名：  InputData(int list[], int n)  OutputData(int list[], int n)
// 功能：    建立待排序数据表,输出数据表内容
/*****------------------------------*****/
void InputData(int list[], int n)
{
    printf("input data:\n");
    for(int i=0; i<n; i++)
        scanf("%d", &list[i]);
}
void OutputData(int list[], int n)
{
    printf("\nthe current soring is:");
    for(int k=0; k<n; k++)
        printf("%d ", list[k]);
}

/*****------------------------------*****/
// 函数名：  ShellSort(int list[], int n)
// 参数：    (传入) int list[],待排序数组
//           (传入) int n,数组长度
// 功能：    使用希尔排序对数据进行排序
/*****------------------------------*****/
void ShellSort(int list[], int n)
{
    int i, pos, length, temp;
    int change;    //记录数值是否有交换
    length=n/2;
    while (length!=0)    //数组仍然可以分割
    {
        //对各个分割的集合进行处理
        for(i=length; i<n; i++)
        {
            change=0;
            temp=list[i];
            pos=i - length;        //计算进行处理的位置

            while (temp < list[pos] && pos >=0 && i <=n)
```

```
                {
                    list[pos + length] = list[pos];  //在集合内比较后交换
                    pos = pos - length;      //计算下一个进行处理的位置
                        change = 1;        //记录数值进行了交换
                }
                list[pos + length] = temp;    //与最后的交换

                if (change != 0)OutputData(list, n); //将每趟排序结果输出
            }

            length = length/2;    //减小增量
        }
    }

    void main()
    {
        int num;
        int list[MAX];
        printf("input length of the list(n <50):\n");
        scanf("%d", &num);

        InputData(list, num);
        ShellSort(list, num);
    }
```

第 4 步：测试数据。

运行结果：

```
input length of the list(n <50):
10
input data:
48 36 60 80 75 12 26 58 36 08
the current soring is:12 36 60 80 75 48 26 58 36 8
the current soring is:12 26 60 80 75 48 36 58 36 8
the current soring is:12 26 58 80 75 48 36 60 36 8
the current soring is:12 26 58 36 75 48 36 60 80 8
the current soring is:12 26 58 36 8 48 36 60 80 75
the current soring is:8 26 12 36 58 48 36 60 80 75
the current soring is:8 26 12 36 36 48 58 60 80 75
the current soring is:8 12 26 36 36 48 58 60 80 75
the current soring is:8 12 26 36 36 48 58 60 75 80
```

思考题：

请结合教材，实现链表存储的希尔排序，并将希尔排序实现函数用 ShellInsert（）和 ShellSort（）完成。

提示函数（请补充建立链表函数和主函数）：

```
typedef struct SqList
{
    int *elem;
    int length;
}SqList, *Sqtpye;

void ShellInsert(Sqtpye &L,int dk)
{
    int i, j;
    SqList *p;
    p=L;
    //一趟增量为 dk 的插入排序，dk 为步长因子
    for(i=dk+1;i<=p->length;i++)
    {
        if(p->elem[i] < p->elem[i-dk])
        {
            //小于时，需 elem[i]将插入有序表
            p->elem[0]=p->elem[i]; //为统一算法设置监测
            for(j=i-dk;j>0&&p->elem[0] < p->elem[j];j=j-dk)
                p->elem[j+dk]=p->elem[j]; //记录后移
                p->elem[j+dk]=p->elem[0]; //插入到正确位置
        }
    }
}
void ShellSort(Sqtpye &L,int dlta[],int t)
{
    int k;
    //按增量序列 dlta[0,1…,t-1]对顺序表 L 作希尔排序
    for (k=0;k<t;k++)      // t 为排序趟数
        ShellInsert(L,dlta[k]); //一趟增量为 dlta[k]的插入排序
}
```

8.3　交换排序实验

【实验 8.3】　冒泡排序的设计与实现

实现冒泡排序程序。

第 1 步：任务分析。

按照冒泡排序算法实现冒泡排序。

第 2 步：程序构思。

冒泡排序是通过相邻元素之间的比较和交换使排序码较大的元素逐渐向序列的尾部移动。冒泡排序的一趟冒泡过程：设 1 < j≤n，r [1]，r [2]，…，r [j] 为待排序列，通过两两比较、交换，重新安排存放顺序，使得 r [j] 是序列中关键码最大的记录。对 n 个待排序的表，第一趟冒泡得到一个关键码最大的记录 r [n]，第二趟冒泡对 n - 1 个待排序的表，再得到一个关键码最大的记录 r [n - 1]，如此重复，直到 n 个记录按关键码有序的表，整个排序结束。

算法描述：

⑴ j = n；　　//从 n 记录的表开始

⑵ if (j < 2) 排序结束；

⑶ i = 1；　　//一趟冒泡，设置从第 1 个记录开始进行两两比较；

⑷ if (i≥j)

　{一趟冒泡结束，j = j - 1；冒泡表的记录数 - 1，转 ⑵ }

else 转 ⑸ ；

⑸ for (i = 1；i <= j - n；i + +)

　if (r [i]. key > r [i + 1]. key) 将 r [i] 与 r [i + 1] 互换；

⑹ 循环结束，转 ⑷ 。

第 3 步：源程序。

```
#include <stdio.h>
#define MAX 50

/*****------------------------------*****/
// 函数名：  InputData(int list[], int n)  OutputData(int list[],
            int n)
// 功能：    建立待排序数据表，输出数据表内容
/*****------------------------------*****/
void InputData(int list[], int n)
{
    printf("input data:\n");
    for(int i =0; i <n; i ++)
        scanf("%d", &list[i]);
}
void OutputData(int list[], int n)
{
    printf("\nthe current soring is:");
    for(int k =0; k <n; k ++)
        printf("%d ", list[k]);
}
```

```
20
21      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
22      // 函数名：  BubblesSort(int list[], int n)
23      // 参数：    (传入) int list[],待排序数组
24      //           (传入) int n,数组长度
25      // 功能：    使用冒泡排序对数据进行排序
26      /* * * * - - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
27      void BubblesSort(int list[], int n)
28      {
29          int i, j , k;
30          for(i =0; i <n -1; i ++)
31          {
32                 for(j =n -1; j >i; j - -)
33                 {
34                     if (list[j] < list[j -1])//比较前后大小
35                     {
36                         //两数进行交换
37                         k =list[j];
38                         list[j] =list[j -1];
39                         list[j -1] =k;
40
41                         OutputData(list, n);//将每趟排序结果输出
42                     }
43                 }
44          }
45      }
46
47      void main()
48      {
49          int num;
50          int list[MAX];
51          printf("input length of the list(n <50):\n");
52          scanf("%d", &num);
53
54          InputData(list, num);
55          BubblesSort(list, num);
56      }
```

第 4 步：测试数据。

运行结果：

```
input length of the list(n <50):
```

```
6
input data:
2 34 1 56 43 7 56 90
the current soring is:2 34 1 56 7 43
the current soring is:2 34 1 7 56 43
the current soring is:2 1 34 7 56 43
the current soring is:1 2 34 7 56 43
the current soring is:1 2 34 7 43 56
the current soring is:1 2 7 34 43 56
```

思考题：

改进冒泡排序，试编制一个程序，对一个待排序的数据集合进行奇偶转换排序。（说明：奇偶排序是指第一趟对所有奇数的 i，将 a［i］和 a［i＋1］进行比较，第二趟是对所有偶数的 i，将 a［i］和进行比较 a［i＋1］，每次比较时，若 a［i］＞a［i＋1］，则将二者交换，重复上述二趟过程的交换进行，直到整个数组有序。）

【实验 8.4】　快速排序的设计与实现

实现快速排序程序。

第 1 步：任务分析。

按照快速排序算法实现快速排序。

第 2 步：程序构思。

快速排序的基本思想是：通过比较关键码、交换记录，以某个记录为界（该记录称为支点，该记录的关键字为 pivot - key），将待排序列分成两部分。其中，一部分所有记录的关键字大于等于支点记录的关键字，另一部分所有记录的关键字小于支点记录的关键字。我们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。对各部分不断划分，直至每一部分中只剩下一个记录为止，整个序列才能按关键字有序排列。

第 3 步：源程序。

```
1        #include <stdio.h>
2        #define MAX 50
3
4        /* * * * - - - - - - - - - - - - - - - - - - - - - - - -    * * * * */
5        // 函数名：  InputData(int list[], int n)  OutputData(int list[],
                     int n)
6        // 功能：    建立待排序数据表，输出数据表内容
7        /* * * * - - - - - - - - - - - - - - - - - - - - - - - -    * * * * */
8        void InputData(int list[], int n)
9        {
10           printf("input data:\n");
11           for(int i =0; i <n; i ++)
12               scanf("%d", &list[i]);
```

```
13      }
14      void OutputData(int list[], int n)
15      {
16          printf("\nthe current soring is:");
17          for(int k=0; k<n; k++)
18              printf("%d ", list[k]);
19      }
20
21      /*****------------------------------   *****/
22      // 函数名：  QuickSort(int list[], int n)
23      // 参数：    (传入) int list[],待排序数组
24      //           (传入) int n,数组长度
25      // 功能：    使用快速排序对数据进行排序
26      /*****------------------------------   *****/
27      void QuickSort(int list[], int start, int end)
28      {
29          int i=start;
30          int j=end;    //设置左右指针
31          int temp;
32          int value=list[start]; //取得最左边的元素
33          if (start < end)
34          {
35              do
36              {
37                  //从左向右找比支点大的值
38                  while (i <=end && list[i] <=value)i++;
39                  //从右向左找比支点小的值
40                  while (j > start && list[j] >=value)j--;
41
42                  if (i <j )//交换 list[i]和 list[j]的值
43                  {
44                      temp=list[i];
45                      list[i]=list[j];
46                      list[j]=temp;
47                  }
48
49              }while(i < j);
50
51              //交换 list[start]和 list[j]的值
52              temp=list[start];
53              list[start]=list[j];
```

```
54                list[j] = temp;
55
56                OutputData(list, 8);//将每趟排序结果输出
57
58                QuickSort(list, start, j -1);
59                QuickSort(list, j +1, end);
60            }
61        }
62
63        void main()
64        {
65            int num;
66            int list[MAX];
67            printf("input length of the list(n <50):\n");
68            scanf("%d", &num);
69
70            InputData(list, num);
71            QuickSort(list, 0, num -1);
72        }
```

第 4 步：测试数据。

运行结果：

```
input length of the list(n <50):
8
input data:
48 36 60 80 75 12 26 36
the current soring is:12 36 36 26 48 75 80 60
the current soring is:12 36 36 26 48 75 80 60
the current soring is:12 26 36 36 48 75 80 60
the current soring is:12 26 36 36 48 75 80 60
the current soring is:12 26 36 36 48 60 75 80
```

思考题：

1. **针对快速排序法的单趟排序，如何在程序中添加代码输出当前一次划分结果呢？**

2. **本程序是递归方式的快速排序，请结合数据结构的栈的知识，设计非递归方式的快速排序程序（提示：用栈保存要排序数据的起止点，从堆栈中弹出要排序数据范围。例如栈操作“**stack［++top］=1；stack［++top］=j；**”就相当于** QuickSort（list，1，j）。

8.4　选择排序实验

【实验 8.5】　直接选择排序的设计与实现

实现直接选择排序程序。

第 1 步：任务分析。

按照直接选择排序算法实现直接选择排序。

第 2 步：程序构思。

直接选择排序的具体做法是 通过比较，首先在所有记录中选出关键字最小的记录，把它与第一个记录交换，然后在其余的记录中再选出关键字次小的记录与第二个记录交换，依此举推，直至所有的记录成为有序序列。

第 3 步：源程序。

```
#include <stdio.h>
#define MAX 50

/*****************************   *****/
// 函数名： InputData(int list[], int n)  OutputData(int list[],
            int n)
// 功能：   建立待排序数据表，输出数据表内容
/*****************************   *****/
void InputData(int list[], int n)
{
    printf("input data:\n");
    for(int i=0; i<n; i++)
        scanf("%d", &list[i]);
}
void OutputData(int list[], int n)
{
    printf("\nthe current soring is:");
    for(int k=0; k<n; k++)
        printf("%d ", list[k]);
}

/*****************************   *****/
// 函数名：  SelectSort(int list[], int n)
// 参数：    (传入) int list[],待排序数组
//           (传入) int n,数组长度
// 功能：    使用选择排序对数据进行排序
/*****************************   *****/
void SelectSort(int list[], int n)
{
    int i, j , k, temp;
    for(i=0; i<n; i++)
```

```
        {
            k=i;
            for(j=i+1; j<n; j++)
            {
                if(list[j] < list[k])k=j;    //用 k 指出每趟在每个区段的
                                             //最小元素

                //交换 list[k]和 list[i]
                temp=list[i];
                list[i]=list[k];
                list[k]=temp;
            }
            OutputData(list, n);  //将每趟排序结果输出
        }
    }

    void main()
    {
        int num;
        int list[MAX];
        printf("input length of the list(n<50):\n");
        scanf("%d", &num);

        InputData(list, num);
        SelectSort(list, num);
    }
```

第 4 步：测试数据。

运行结果：

```
input length of the list(n<50):
8
input data:
34 48 39 21 56 17 80 52
the current soring is:17 48 39 21 56 34 80 52
the current soring is:17 21 48 39 56 34 80 52
the current soring is:17 21 34 39 56 48 80 52
the current soring is:17 21 34 39 56 48 80 52
the current soring is:17 21 34 39 48 56 80 52
the current soring is:17 21 34 39 48 52 80 56
the current soring is:17 21 34 39 48 52 56 80
the current soring is:17 21 34 39 48 52 56 80
```

思考题：

试采用单链表作为存储结构,实现选择排序函数。

【实验 8.6】　堆排序的设计与实现

实现堆排序程序。

第 1 步:任务分析。

按照堆排序算法实现堆排序。

第 2 步:程序构思。

结合教材第 9 章堆排序的例 9.9 具体分析以及算法示例。主要步骤如下:

⑴ 将待排序的数据存入二叉树数组中;

⑵ 将二叉树转成最大堆;

⑶ 堆的最大值和数组最后一个元素交换;

⑷ 其余元素进行堆重建,并打印目前排序结果;

⑸ 重复 ⑶、⑷ 步,直到所有值均已排序完成。

第 3 步:源程序。

```
#include <stdio.h>
#define MAX 50 //排序表的最大容量

/*****************************  *****/
// 函数名:  InputData(int list[], int n)  OutputData(int list[],
            int n)
// 参数:    (传入)(int list[]  顺序表
// 功能:    输入顺序表数据,显示表中所有元素
/*****************************  *****/
void InputData(int list[], int n)
{
    int i;
    printf("input data:\n");
    for(i =1; i <=n; i ++)
        scanf("%d",&list[i]);
}
void OutputData(int list[], int n)
{
    int i;
    printf("the sorted list is:\n");
    for(i =1; i <=n; i ++)
        printf("%d ", list[i]);
    printf("\n");
}
/*****************************  *****/
// 函数名:   HeapAdjust(int list[], int s, int m)
// 参数:     (传入)int list[]  待排序顺序表
```

```
27        //              (传入)int s, m   调整调整范围
28        // 功能：       已知 list[s..m]中除 list[s]之外均满足堆的定义,调整 list
[s]
29        //              使 list[s..m]成为一个大顶堆
30        /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
31        void HeapAdjust(int list[], int s, int m)
32        {
33            int j, temp;
34            temp = list[s];
35            for(j = 2 * s; j <= m; j * = 2) //沿关键字较大的结点向下筛选
36            {
37              if(j < m && list[j] < list[j + 1]) ++j; //j 为关键字较大的记录的
                                                        //下标
38              if(temp > = list[j]) break; //temp 应插入在位置 s 上
39              list[s] = list[j];
40              s = j;
41            }
42            list[s] = temp; //插入
43        }
44        /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
45        // 函数名：   HeapSort(int list[], int n)
46        // 参数：     (传入)int list[]  待排序顺序表
47        //            (传入)int n   顺序表长度
48        // 功能：     对顺序表 L 做堆排序
49        /* * * * - - - - - - - - - - - - - - - - - - - - - - -   * * * * */
50        void HeapSort(int list[], int n)
51        {
52            int i,t;
53            for(i = n/2; i > 0; i --) //把 list[1..n]建成大顶堆
54                HeapAdjust(list, i, n);
55            for(i = n; i > 1; i --)
56            {
57                t = list[1]; //将堆顶记录和当前未经排序子序列 list[1..i]
58                list[1] = list[i]; //中的最后一个记录相互交换
59                list[i] = t;
60                HeapAdjust(list,1,i - 1); //将 list[1..i - 1]重新调整为大顶堆
61            }
62        }
63
64        void main()
65        {
66            int list[MAX];//声明表 list
67            int num;
```

```
68
69          printf("input length of the list(n<50):\n");
70          scanf("%d", &num);
71
72          InputData(list, num);     //输入待排序列
73          HeapSort(list, num);      //堆排序
74          OutputData(list, num);    //显示排序结果
75      }
```

第 4 步：测试数据

待排序的二叉树数据表示如图 8-1 所示，二叉树建立最大堆过程如图 8-2 所示。堆的排序过程如图 8-3 所示。

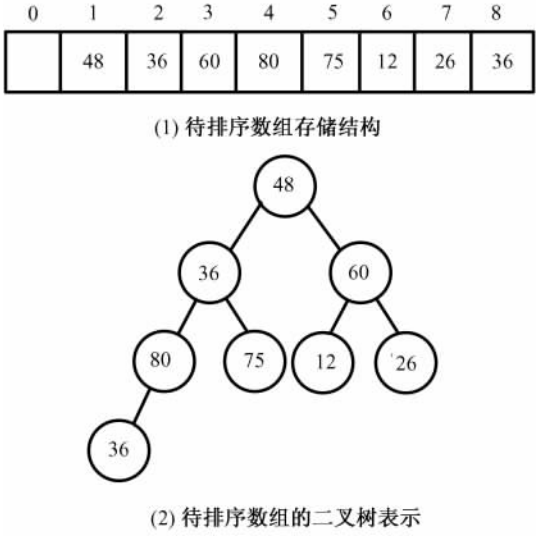


图 8-1　堆排序的二叉树组表示

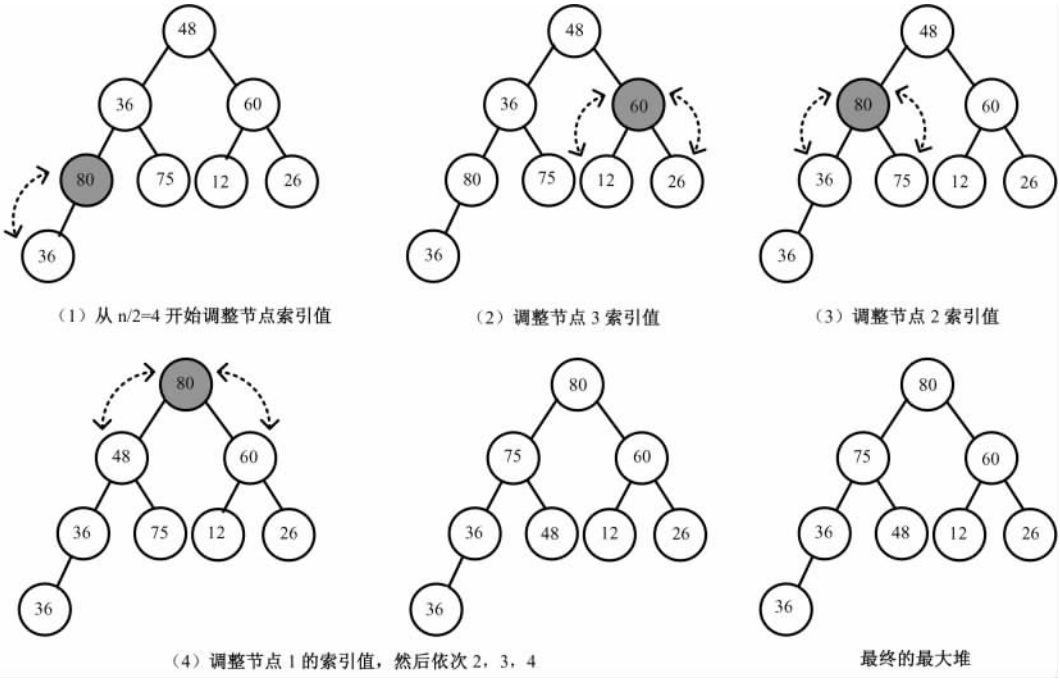


图 8-2　二叉树建立最大堆过程示意

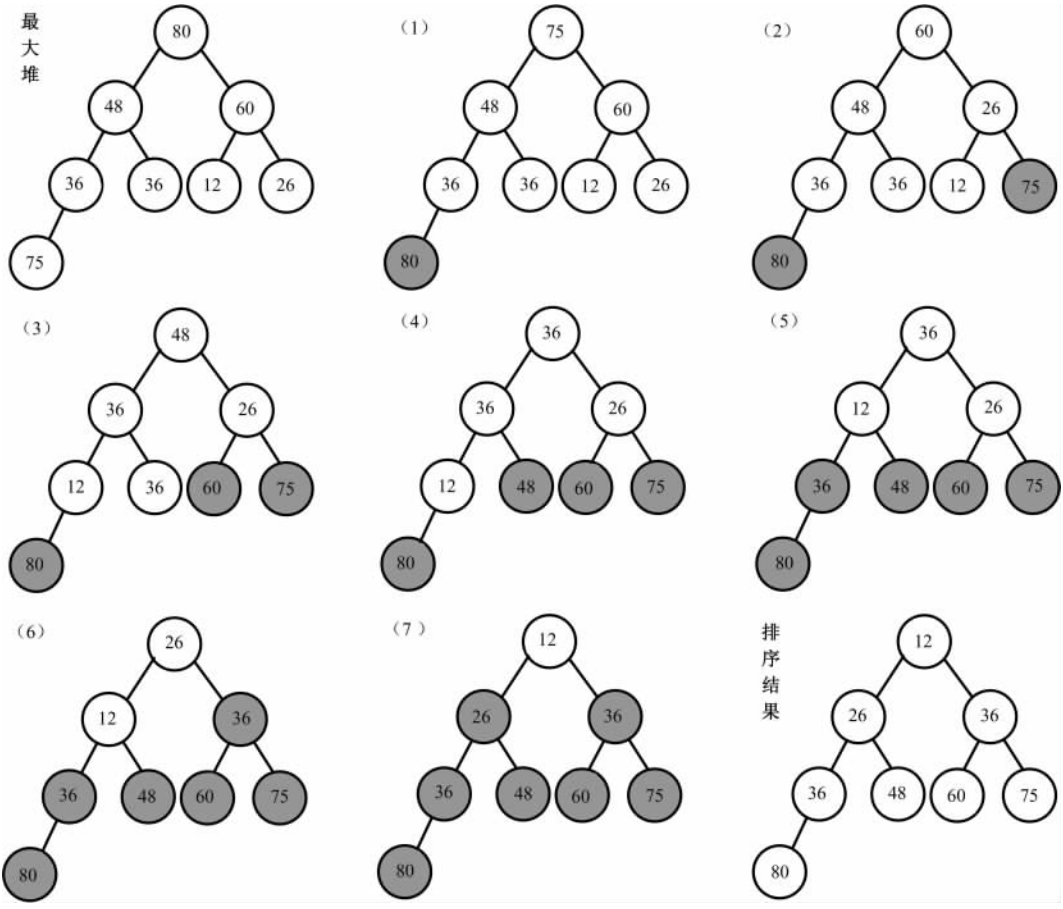


图 8-3　堆排序过程 (取出根节点,再排成堆) 示意

运行结果：

```
input length of the list(n <50):
8
input data:
48 36 60 80 75 12 26 36
the sorted list is:
12 26 36 36 48 60 75 80
```

8.5 小　　结

通过实验,了解掌握常见排序算法的思路和程序实现,并了解常见排序算法的时间、空间的性能分析和稳定性等,对各种排序算法进行简单的特点总结归纳,掌握适用情况。

第9章 文　　件

本章的典型实验主要是数据结构内容的综合练习，不仅包括文件操作，而且是前几章的综合应用，共2个实验内容。经过前述实践的练习，本章侧重从问题分析、算法考虑，数据结构设计、模块划分等扩展读者综合应用数据结构知识解决实际问题的能力。

9.1 知识点概述

和表类似，文件是由大量性质相同的记录组成的集合。一般称存储在二级存储器（外存储器）中的记录集合为文件。本章在文件的表示方法和各种运算实现方法的基础上，利用几类综合实验，不仅完成文件操作的基本练习，而且对前述数据结构内容进行了练习，达到综合应用的目的。

9.2 综 合 实 验

【实验9.1】 班级个人信息管理程序

本程序目的在于运用数据结构相关知识存储管理班级个人信息。

1. 概述

如果用文件来存储班级成员的各种信息，达到用计算机管理班级成员的花名册，不仅方便快捷，便于作一些分析统计应用，而且在班级个人信息管理的基础上，还能扩充添加与个人相关的课程信息、成绩信息等，将使得班级个人管理工作变得轻松而有趣。本实验的目的就在于建立这种信息管理的基础，实现个人信息管理，读者可以在此基础上进行扩充。

2. 程序构思

1）数据结构设计

每个人包括的信息一般不能由一个数据类型进行存储，而是由若干不同类型的字段组合存储的。为了达到这样的目的，以结构体的形式存储个人信息就是理想的选择。个人信息包括姓名、学号、年龄、班级、宿舍、电话等。可以建立如下的结构体类型：

```
typedef struct
{
char name[20];          //姓名
char id[10];            //学号
int age;                //年龄
char address[20];       //宿舍
char tel[15];           //电话
}PersonInfo;
```

每一个个体组成了丰富多彩的班级大家庭。如何来存储这些数据呢？经过前面的实验内容，很容易就考虑到用数组的形式，当然为了更好的动态组织数据，用链表处理也是一个不错的选择，请读者自行选择。班级成员信息可以用 Person Info person [MAX] 定义结构体数组来存储。考虑到班级是班级成员共有的相同的属性，可以把这些组合到一起，形成如下的班级结构体类型：

```
typedef struct
{
    PersonInfo person[MAX];     //个人信息结构体数组
    char classid[10];           //班号
    int count;                  //班级成员总数
}ClassInfo;
```

举例：当存储一个班级成员信息时，定义 ClassInfo classNo1，实际个人信息存储在 classNo1. person这个结构体数组中。

2）模块设计

以下分析引导读者按照模块化的思想处理分析问题，每个模块不一定只有一个函数实现该模块的功能，所以实现函数由用户自定义名称，选择合适的参数等。

⑴ 文件读写操作模块。由于班级个人信息相关数据存储到文件当中，以方便程序再次调用。所以本程序涉及文件的新建、修改、删除等操作。

OpenFile (char * filename)；　//将保存在指定文件中的所有个人数据按照已知的格式定义读入到 classNo1 变量中。

SaveFile (char * filename)；　//将 classNo1 变量中的全部数据保存到指定的文件当中，写入格式由读者自行设定。

⑵ 个人信息录入模块。该模块负责班级个人信息的录入工作，把用户录入的姓名、年龄、学号、班级等个人信息资料存储在班级个人信息结构体变量中。

⑶ 个人信息显示模块。该模块根据用户指定条件和范围，比如学号、姓名，或者全部显示，显示班级个人的完整信息。

⑷ 个人信息查询模块。该模块负责根据用户输入的学号、姓名等查询条件，在班级个人信息数组中查找，如果找到显示完整的个人信息，否则显示“未找到”。

⑸ 个人信息修改模块。该模块接收用户输入的一条记录的学号或者姓名等条件，在班级个人信息数组中查找，如果找到则显示该记录的原数据并提示用户输入新的数据以代替原有数据，否则显示“未找到”。

⑹ 个人信息删除模块。该模块根据用户指定的学号或者姓名等条件，在班级个人信息数组中查找，如果找到则显示该记录完整信息，并提示用户是否选择删除，如果选择删除则在数组中删除该记录。

3. 界面设计

为了简单起见，使用 C 语言提供的输出函数绘制供用户选择的菜单，并响应用户的选择。

4. 源程序略。

【实验 9.2】　《我的课表》程序的设计与实现

1. 概述

大学课程多而且上课地点不固定,为了能准时在正确地点上课,需要经常查询自己所在班的课程。如果能用利用计算机管理课表,方便查询自己的课程并且提供诸如日程安排等的时间提示等,上课就会变得愉快而有趣。《我的课表》小程序就是为这样的目的而诞生的。

为了能达到这样的目的,需要实现以下功能:根据不同用户的课表不同定义、管理、维护个人课表;根据不同检索信息查询课表功能;上课时间提醒功能;消息显示功能;用户管理功能。

2. 数据结构设计

如何存储、组织、处理等是完成这些功能的基础。为了能管理课表,实现恰到好处的提醒,课表相关数据必须长期保存。以文件的形式保存是一种简单的实现方法。按照一定格式以文件的形式保存课表相关数据信息,既方便用户自定义课表信息,也方便程序读入修改相关数据。为此,需要定义文件格式,参考定义如下(本着方便用户直接修改数据文件,方便程序读写,读者可以自行修改)。

⑴ 课表文件定义。数据行第一行存储周一数据,第二行存储周二数据,以此类推。每一行由分号分隔为四部分存储每天四大节课。如果没有数据,记做 NULL;如有数据,按照“周次,班号,上课地点”的格式连成字符串,由数据读取模块负责处理;课表文件以 END 作为文件结尾标志。举例如下:

```
NULL; NULL; 2 -10,215401,409; NULL;
NULL; NULL; 10 -16,215409,409; NULL;
2 -16,215406,409; 11 -17,215407,409; NULL; NULL;
9 -16,215403,409; 11 -16,215401,409; NULL; NULL;
2 -10,215407,409;13 -16,215409,409; 13 -16,215409,409; 10 -17,215403,409;
END
```

⑵ 课程信息文件定义。文件格式定于如下:

“班号　学时　课程名称
 班号　学时　课程名称
 END”

⑶ 时间结构体定义。时间结构体定义如下:

```
typedef struct
{
    int year,  month,  day;
    int hour,  minute,  second;
}DATE;
```

⑷ 课程结构体定义。当每个课程信息的存储解决后,整个课表数据就能在此基础上利用数组、链表等数据结构相关知识建立。

```
typedef struct
{
DATE date;
char name[20];          //课程名称
char address[20];       //上课地点
}Lesson;
```

3. **模块设计**

根据程序描述以及功能侧重，建立以用户为中心的模块划分，如图 9-1 **所示。**

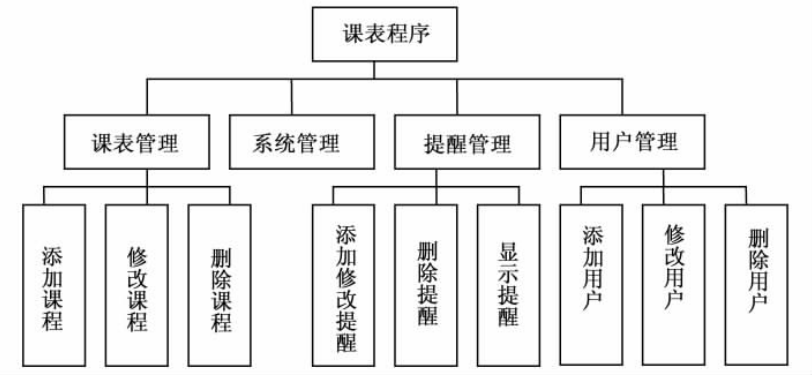


图 9-1　**以用户为中心的模块划分**

⑴ 课表管理。课表管理模块主要完成对课表课程的添加、修改、删除等操作。课表和用户是对应的，不同的用户有不同的课表。而课程的添加、修改、删除等都是在它所属的某个课表中完成的。为此在课表管理模块中，用课表的编号 ID **和用户、课程联系起来。**

课程添加、修改、删除操作函数分别为：

```
void  AddLesson(Lesson less, int tableID);
void  UpdateLesson(Lesson less, int tableID);
void  DeleteLesson(Lesson less, int tableID);
```

课表添加、删除操作函数分别为：

```
int  AddNewTable();    //返回新建课表的 ID
void  DeleteTable();
```

由于课表相关数据是存储到自定义的课表文件中，以方便程序再次调用，所以课表添加删除操作中涉及文件操作，课程操作中涉及文件内容读取操作，具体根据文件格式定义实现。文件创建和删除操作函数分别为：

```
bool  NewFile(char * filename);
bool  DelFile(char * filename);
```

⑵ 系统管理。系统管理包括系统登录以及对程序初始化数据准备过程中相关变量的设定，包括提前提醒日期和时间、默认用户和学期第一周设定等。设定相关变量函数为：

```
void  SetDayRemind(int day, int min);    //提前几天提醒、提前多少分钟提醒
```

```
void  SetDefaultUser();
void  SetFirstWeek();
```

与系统登录及相关的函数模块定义如下：

```
void  Login();          //负责处理用户登录,并完成课表关联
void  ShowMenu();       //显示系统功能菜单,响应用户调用
void  ShowToday();      //显示当日提醒等相关数据
```

⑶ 提醒管理。在提醒管理中,要根据用户身份读取课表信息,然后根据设定的某天(或者前一天,提前提醒日期、时间用户可以设定)所在的周次、日期、时间判定当前是否有该上的课程,给出相关提醒。相关函数模块有：

```
bool  ReadTable(char * user);          //读取其课表信息
void  ShowMessage();                   //显示提醒
bool  CheckAlarm();                    //检测是否有提醒
```

提醒的基本操作,包括建立和删除：

```
void  CreateAlarm();
void  DelAlarm();
```

⑷ 用户管理。此模块负责对用户进行管理,包括用户的添加、修改和删除。添加一个用户的完整过程不仅包括完成用户的基本信息添加,而且包括其课表的关联,如果该用户没有关联课表,应提醒建立或者修改已存在的内容。与此有关的函数模块有：

```
void NewUser(char *name);
void UpdateUser(char *name);
void DelUser(char *name);
```

当定义好基本的函数模块后,运行如图 9-2 所描述的整个程序的流程。

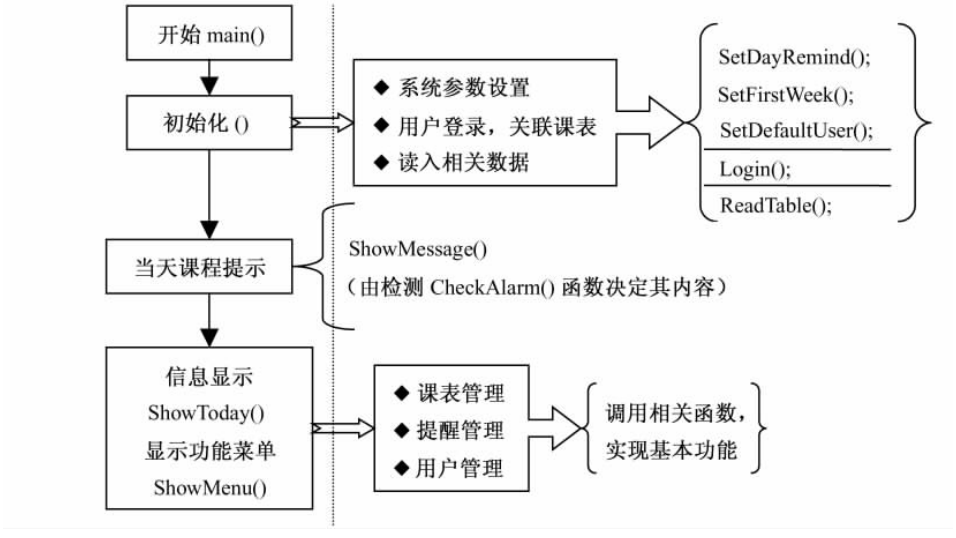


图 9-2　程序流程

4. 源程序

全部代码略,这里仅给出读取课表数据文件的实现函数。读者可体会文件格式的定义和具体文件操作的关联。其他文件处理请参考如下函数。

```
bool  ReadTable(char * user)
{
    //负责把字符串按照课程和周次处理
    lessonIndex=0;
    weekIndex=0;

    //读入课程表信息
    FILE *file=NULL;
    //根据传入的 user 的值,可以选择其关联的课表数据文件
    file=fopen("data.txt","rb");
    if(file==NULL)
    {
        return true;
    }

    char inf[80];
    while(fscanf(file,"%s",inf)&&(strcmp(inf,"END"))!=0)
    {
        if(strcmp(inf,"NULL;")==0)//如果是 NULL
        {
            //不把信息放到课程结构表中
        }
        else//把对应的信息放到对应的结构表中
        {
            ReadInformation(inf);    //具体处理每一节课程信息
        }

        lessonIndex++;
        if(lessionIndex==4)
        {
            weekIndex++;
            lessonIndex=0;
        }
    }

    fclose(file);

    return true;
}
```

思考题：

完成程序编码后，需要进行大量的测试检验程序的界面、功能等。测试时候需要注重输入的数据的边界性及其他情况，例如需考虑到课表数据文件不存在的情况等，这些对于一个完整、健壮的程序来说都是需要注意的问题。

9.3 小　结

通过综合实验，学习掌握如何把一个实际问题程序化，包括程序功能的规划，模块的设计，具体函数的实现，编码测试等，并以此达到综合掌握运用数据结构相关知识的能力。