实验 2 链表

设计并实现一个简单的学生成绩管理系统。学生成绩表信息包括：学号、姓名、各科课程成绩（语文、数学、英语、政治）和总分。用带头结点的单链表管理学生成绩表，每个学生的信息依次从键盘输入，并根据需要进行插入、删除、排序和输出等操作。该学生成绩管理系统的功能如下：

（1）输入学生信息，按输入顺序建立一个带头结点的单链表，直到输入学号为 0 结束。

（2）在单链表末尾追加一个学生的信息，并输出结果。

（3）输入一个学生姓名，在链表中进行查找，如果存在，显示该生的所有信息；如果不存在，显示提示信息“查无此人”。

（4）输入一个学生学号，如果链表中存在该学生信息，则将其删除。

（5）将学生成绩按总成绩从高到低进行排序，并输出结果。

根据题目要求，用带头结点的单链表存储学生成绩表，则每个学生信息的结构和单链表的类型定义，以及实现学生成绩管理的完整程序如下：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define NULL 0
#define NUM    4
#define len sizeof(linklist)

typedef struct stud                         /*学生类型定义*/
{     char no[10];                          /*学生的学号*/
      char name[8];                         /*学生的姓名*/
      float score[NUM];                     /*学生的各科成绩*/
      float total;                          /*学生的总分*/
      struct stud *next;                    /*结点的指针域*/
}student,linklist;                          /*学生的类型名*/

char    course[8][10]={"","学号","姓名","语文","数学","英语","政治","总分"};

int menu();                                 /*菜单函数*/
void printlist(linklist *head);             /*输出学生成绩表函数*/
linklist *create_list();                    /*创建按输入顺序排列的成绩表函数*/
linklist *findnode(linklist *head,char *name); /*查找指定学生姓名的函数*/
void print_node(linklist *p);               /*输出当前结点学生信息*/
int insert_data(linklist *head);            /*在末尾插入学生信息函数*/
int delete_no(linklist *head,char *no);     /*删除指定学号的函数*/
void sort(linklist *head);                  /*排序-按学号升序排列函数*/

void main( )                                /*学生成绩管理系统主函数*/
{     linklist *head,*p;                    /*学生成绩表表头指针 head,临时指针变量 p*/
      char name[8], no[10];
      int select,isdo;

      do
```

```
    {   system("cls");                          /*调用系统清屏函数*/
        select = menu();
        switch (select)
        {   case 1: head=create_list();     /*创建函数*/
                    if(head == NULL)
                    {   printf("\n\t\t 空间不足!程序执行结束.....\n\n");
                        system("pause");
                        return;
                    }
                    break;
            case 2: isdo = insert_data(head);          /*插入函数*/
                    if (isdo == 1)
                        printf("\n\t\t 追加学生信息成功!");
                    else
                        printf("\n\t\t 空间不足，追加学生信息失败!");

                    break;
            case 3: printf("\n\t\t 请输入要查找的学生姓名：\n\t\t");
                    fflush(stdin);
                    gets(name);
                    p = findnode(head,name);      /*查找函数*/
                    if(p != NULL)
                        print_node(p);
                    else
                        printf("\n\t\t\t 查无此人!\n\n");
                    break;
            case 4: printf("\n\n\t\t 请输入要删除学生的学号:\n\t\t");
                    fflush(stdin);
                    gets(no);
                    isdo = delete_no(head,no);     /*删除函数*/
                    if (isdo == 0)
                        printf("\n\t 学号为 %s 的学生不存在,删除失败!\n",no);
                    break;
            case 5: sort(head);                         /*按总分排序函数*/
                    break;
            case 6: printlist(head);                    /*输出函数*/
                    break;
            case 0: printf("\n\n\n\t\t\t 谢谢使用！再见...\n\n");
                    exit(0);
        }
        printf("\n\n");
        system("pause");
    }while(select != 0);
}/*main*/

int menu( )                                         /*菜单显示函数*/
```

```
{    int n;char c;

     printf("\n\n\t\t 学生成绩管理系统：\n");
     printf("\n\t\t\t1. 建立学生成绩表");
     printf("\n\t\t\t2. 追加学生成绩信息");
     printf("\n\t\t\t3. 按姓名查找");
     printf("\n\t\t\t4. 删除指定学号的学生信息");
     printf("\n\t\t\t5. 排序(按总分降序排列)");
     printf("\n\t\t\t6. 输出");
     printf("\n\t\t\t0. 退出");

     do
     {    fflush(stdin);
          printf("\n\n\t\t 请输入数字 0～6 选择功能:");
          c=getchar();n = c-48;
          if (n<0||n>6)
               printf("\t\t\t 输入选项错误！请重新输入选项");

     }while(n<0||n>6);
     return n;
}/*menu*/

linklist *create_list()          /*尾插法创建带头结点的学生成绩单链表*/
{    linklist *head,*p,*r;
     int i;
     head = (linklist*)malloc(len);
     if(head == NULL)
     {
          return head;
     }
     head->total = 999;                         /*输入头结点的特殊数据*/
     head->next = NULL;
     r = head;
     while(1)
     {
          p = (linklist*)malloc(len);
          if(p == NULL)
               return head;
          printf("\n\n\t\t 请输入学生信息，直到输入学号为'0'时结束:\n\n");
          fflush(stdin);                        /*清除输入缓冲区*/
          printf("\n\t\t 学号: ");
          gets( p->no);                         /*输入学生的学号*/
          if (strcmp(p->no,"0") != 0)           /*若输入不为"0"，则建立该学生信息结点*/
          {    printf("\t\t 姓名: ");
               gets( p->name );                 /*输入学生的姓名*/
               p->total = 0;
```

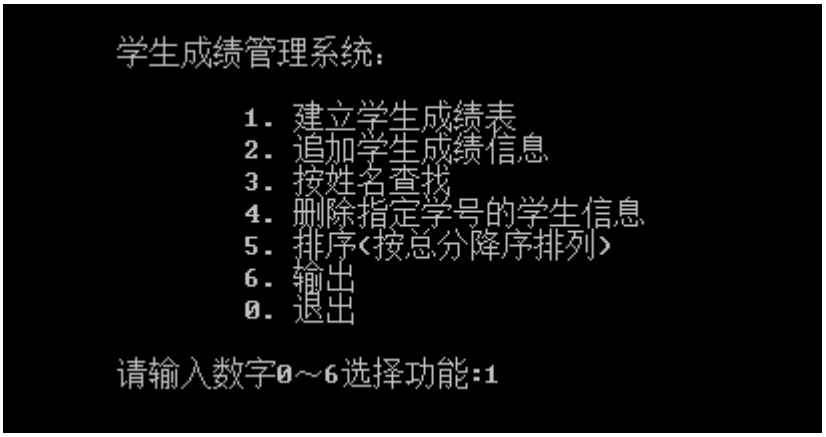
```
                for (i = 0; i < NUM;i ++)
                {     printf("\t\t%s: ", course[i+3]);
                      scanf("%f", &p->score[i]);    /*输入学生各科成绩*/
                      p->total +=    p->score[i];     /*计算总分*/
                }
                p->next = NULL;                     /*将新结点的指针域置空*/
          }
          else
          {     free(p);                            /*释放空间*/
                return head;                        /*返回学生成绩表的头指针*/
          }
          r->next = p;
          r = p;
     }
}/*linklist *create_list()*/

void printlist(linklist *head)                      /*输出学生成绩表*/
{     linklist *p;
      int i;

      p = head->next;
      if (p == NULL)
      {     printf("\n\n\t\t\t 目前学生成绩表没有数据，请输入数据！\n");
            return;
      }
      printf("\n\n");
      for (i = 1; i <= NUM+2; i++)                  /*输出学生成绩表标题*/
            printf("%-10s",course[i]);
      printf("总分\n");
      p = head->next;
      while(p != NULL)                              /*输出所有学生的信息*/
      {     printf("%-10s%-10s", p->no,p->name);
            for(i = 1; i <= NUM; i++)
                  printf("%-10.1f", p->score[i]);
            printf("%-10.1f\n", p->total);
            p = p->next;
      }
      printf("\n\n\n\t\t\t");
}/*printlist*/

void print_node(linklist *p)                        /*输出单个学生信息*/
{     int i;
      printf("\n 学生信息如下：\n");
      for (i = 1; i <= NUM + 2; i++)                /*输出学生成绩表标题*/
            printf("%-10s",course[i]);
      printf("总分\n");
```

```
    printf("%-10s%-10s", p->no,p->name);
    for(i = 1; i <= NUM; i++)
        printf("%-10.1f", p->score[i]);
    printf("%-10.1f\n\n", p->total);
}/*print_node*/

int insert_data(linklist *head)                    /*在末尾插入学生信息函数*/
{
    linklist *r = head,*p;
    int i;
    while(r->next != NULL)                         /*定位指针指向最后一个结点*/
    {
        r = r->next;
    }

    p = (linklist*)malloc(len);
    if(p == NULL)
        return 0;
    printf("\n\n\t\t 请输入要追加的学生信息:\n\n");
    fflush(stdin);                                 /*清除输入缓冲区*/
    printf("\n\t\t 学号: ");
    gets( p->no);                                  /*输入学生的学号*/
    while(strcmp(p->no,"0") == 0)
    {
        printf("\n 学号不能为 0，请重新输入：\n");
        gets( p->no);
    }
    printf("\t\t 姓名: ");
    gets( p->name );                               /*输入学生的姓名*/
    p->total = 0;
    for (i = 0; i < NUM;i ++)
    {   printf("\t\t%s: ", course[i+3]);
            scanf("%f", &p->score[i]);             /*输入学生各科成绩*/
            p->total +=   p->score[i];             /*计算总分*/
    }
    p->next = NULL;                                /*将新结点的指针域置空*/
    r->next = p;
    return 1;
}/*insert_data*/

linklist *findnode(linklist *head,char *name)
{   linklist *p;
    p = head->next;
    while(p != NULL && strcmp(p->name,name) != 0)
    {
        p = p->next;                               /*查找指定结点的前驱结点*/
```

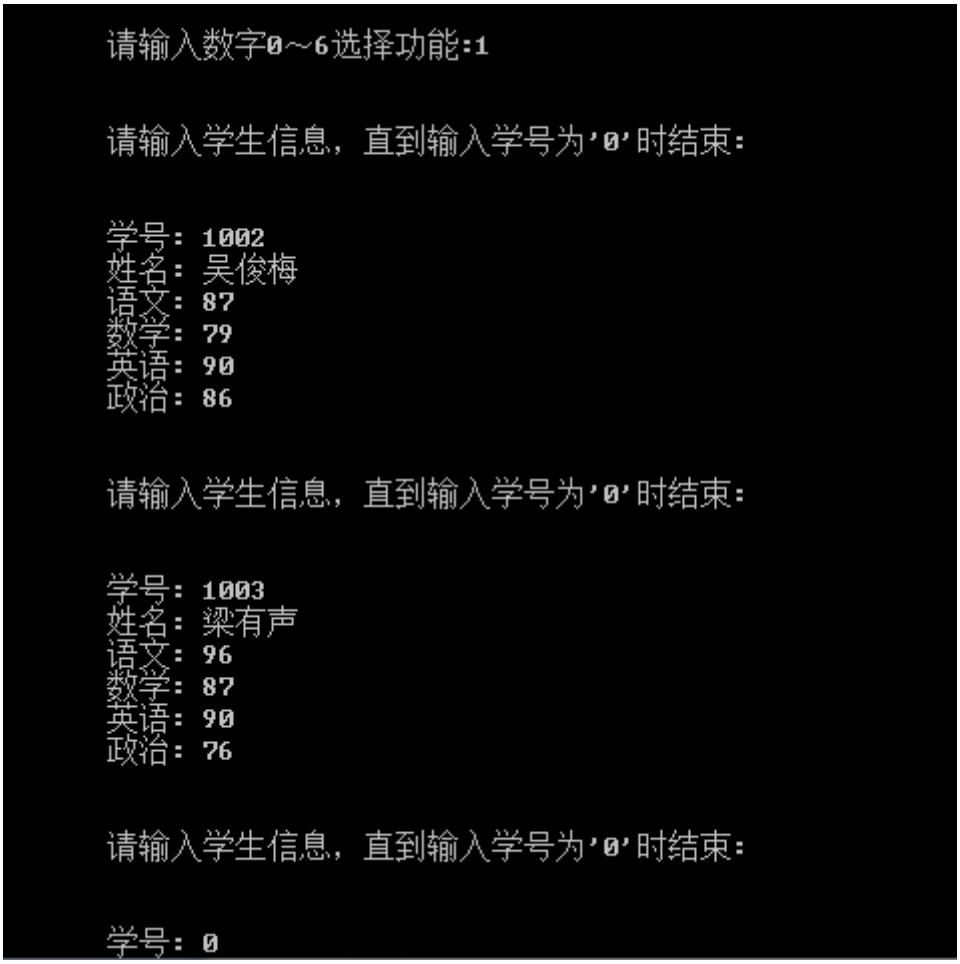
```
        }
    return p;                                   /*返回指定结点的前驱结点*/
}/*findnode*/

int delete_no(linklist *head,char *no)          /*删除指定学号的函数*/
{   linklist *p = head->next, *q = head;
    char c;
    while(p != NULL && strcmp(p->no,no) != 0)
    {   q = p;                                  /*查找指定结点的前驱结点*/
        p = p->next;
    }
    if (p != NULL)                              /*找到该结点，则从学生成绩表中删除*/
    {   print_node(p);
        printf("\n\t\t 确实删除吗？(Y/N): ");
        fflush(stdin);
        c=getchar();
        if (c == 'Y' || c == 'y')
        {   q->next = p->next;                  /*修改指针,删除指定结点*/
            free(p);                            /*释放删除结点所占用的内存*/
            return 1;                           /*删除成功,返回成功标志 1*/
         }
         else return 2;                         /*不删除结点,返回标志 2*/
    }
    else
    {
        return 0;                               /*删除失败,返回失败标志 0*/
    }
}/*delete_no*/

void sort(linklist *head)                       /*按总分降序排列*/
{   linklist *p,*q,*r,*t;
    int flag = 1 ;
    while (flag)
    {   p=head;
        q=head->next;
        flag = 0;
        while (q != NULL)
        {
            r = q->next;
            if (r == NULL)
                break;
            else if(q->total < r->total )
            {   t = r->next;
                q->next = t;
                r->next = q;
                p->next = r;
```

```
                    p = r;
                    q = p->next;
                    r = q->next;
                    flag = 1;
                }
                else
                {   p = q;
                    q = p->next;
                }
            }
        }
    }/*sort*/
```

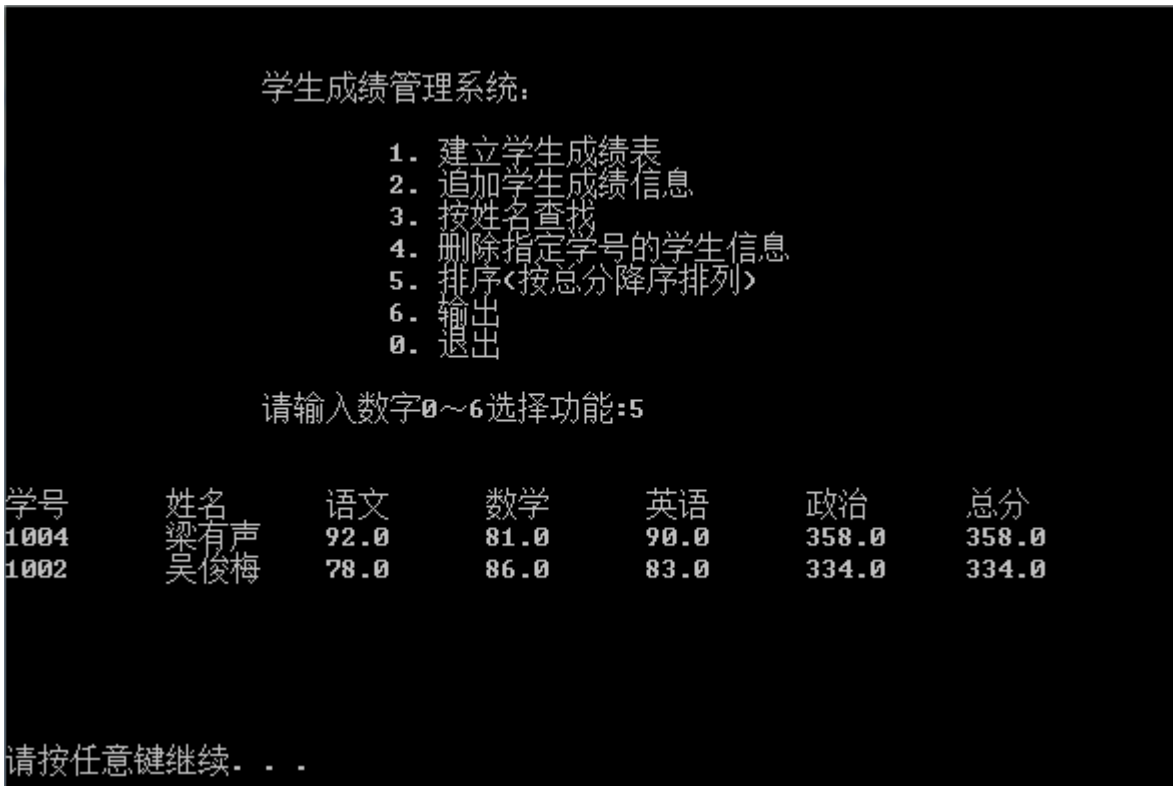
程序运行结果如实验图 2-1 所示。



(a) 运行初始界面—菜单项选择



(b) 输入学生信息按总分降序排列的学生成绩表



(c) 输按总分降序排列的学生成绩表

实验图 2-1 实验示例---简单的学生成绩管理系统

实验 4 队列

根据对表达式求值的算法描述，下面给出具体的 C 语言实现代码，其中包括两个文件，SequenStack.c 和 CalculateExpression.c。其中，SequenStack.c 文件中包含了对顺序栈的定义和其基本运算的实现代码。而 CalculateExpression.c 文件通过包含 SequenStack.c 文件，通过栈来具体实现对所输入的中缀表达式的求值过程。为了方便起见，代码中的栈为一个存储字符类型的顺序栈，可以用来存储操作数、运算符或运算结果。因此，要求输入的表达式为一个字符串，其中操作数只能为一位整数，并且运算符为基本的加、减、乘、除四则运算，可以包含嵌套括号，其表达式的计算结果也只能为一位整数。具体代码如下：

CalculateExpression.c 文件的代码。

```
#include <stdio.h>
#include<string.h>
#include "SequenStack.c"

int changeCharToData(char c);
char changeDataToChar(int x);
void calculate(SequenStack *S, char operator);
int arithmeticalOperate(int operand1, int operand2, char operator);
int evaluatePostfixExpression(char *express);
char *transIntoPostfixExpression(char *express);

void main()
{    char *infixExpression;                    /* 中缀表达式指针 */
     char *postfixExpression;                  /* 后缀表达式指针 */
     int result = 0;                           /* 表达式的计算结果 */
     infixExpression = (char *)malloc(sizeof(char));

     printf("请输入中缀表达式:");
     gets(infixExpression);
     postfixExpression = transIntoPostfixExpression(infixExpression);
          /* 将中缀表达式转换为后缀表达式 */
     printf("转换为后缀表达式: %s\n", postfixExpression);

     result = evaluatePostfixExpression(postfixExpression);   /* 根据后缀表达式求值 */
     printf("表达式的计算结果: %d\n", result);

}

/*根据后缀表达式 express 计算表达式的值，并返回计算的结果*/
int evaluatePostfixExpression(char *express)
{    char oprand;                              /*存储后缀表达式的操作数或运算符*/
     SequenStack *S;                           /*存储运算结果*/
     char result;
     int i;
     S = Init_SequenStack( );
```

```
    for(i=0;express[i]!='\0';i++)
    {   oprand = express[i];
        if(oprand>='0'&&oprand<='9')
        {   Push_SequenStack(S, oprand);    /*如果为操作数，则将操作数入栈 S*/
        }
        else                    /*否则为运算符，则调用 calculate 函数进行求值计算*/
        {   calculate(S, oprand);
        }
    }
    /*栈 S 中存放的是表达式的计算结果，通过出栈操作，赋值给 result 变量，返回调用函数*/
    Pop_SequenStack(S, &result);
    return changeCharToData(result);
}

/* 将输入的中缀表达式转换为后缀表达式 */
char * transIntoPostfixExpression(char *express)
{   char oprand;                        /*存储操作数或运算符*/
    char operator;
    SequenStack *S;                     /*存储运算符*/
    char *postfixExpress;               /*后缀表达式指针*/
    int offset = 0;
    int i,len;
    S = Init_SequenStack( );
    len = strlen(express);
    postfixExpress = (char *)malloc(len*sizeof( char ) );
    /*遍历中缀表达式，如果字符为数字，直接存入后缀表达式，否则，根据运算符的优先级，进行出栈或入栈操作*/
    for(i = 0; express[i] != '\0'; i++)
    {   oprand = express[i];
        switch(oprand)
        {   case '+':   /*如果为'+','-'，则将栈中'('前的运算符出栈并存入后缀表达式*/
            case '-':
                while(!SequenStack_Empty(S))
                {   GetTop_SequenStack(S, &operator);
                    if(operator != '(' )
                    {   Pop_SequenStack(S, &operator);
                        postfixExpress[offset++] = operator;
                    }
                    else
                    {   break;
                    }
                }
                Push_SequenStack(S, oprand);
                break;
            case '*':
                /*如果为'*','/'，则将栈顶为'*','/'的运算符出栈并存入后缀表达式*/
```

```
                case '/':
                    while(!SequenStack_Empty(S))
                    {   GetTop_SequenStack(S, &operator);
                        if(operator=='*'||operator=='/')
                        {   Pop_SequenStack(S, &operator);
                            postfixExpress[offset++] = operator;
                        }
                        else
                        {   break;
                        }
                    }
                    Push_SequenStack(S, oprand);
                    break;
                case '(':                           /*如果为'('，则将'('入栈*/
                    Push_SequenStack(S, oprand);
                    break;
                case ')':  /*如果为')'，则将栈中'('前的运算符出栈并存入后缀表达式*/
                    while(!SequenStack_Empty(S))
                    {   GetTop_SequenStack(S, &operator);
                        if(operator!='(')
                        {   Pop_SequenStack(S, &operator);
                            postfixExpress[offset++] = operator;
                        }
                        else
                        {   Pop_SequenStack(S, &operator);
                            break;
                        }
                    }
                    break;
                default:                    /*如果为操作数，则将操作数存入后缀表达式 */
                    postfixExpress[offset++] = oprand;
            } /*switch*/
        } /*for*/
        /*遍历完中缀表达式后，将栈中仍有的运算符出栈，并存入后缀表达式*/
        while(!SequenStack_Empty(S))
        {   Pop_SequenStack(S, &operator);
            postfixExpress[offset++] = operator;
        }
        postfixExpress[offset] = '\0';

        return postfixExpress;
}

/* 将字符转换为数字 */
int changeCharToData(char c)
{   return c-'0';
```

```
}

/*  将数字转换为字符  */
char changeDataToChar(int x)
{     return x+48;
}
/*  根据操作符进行求值计算  */
void calculate(SequenStack *S, char operator )
{     char operand1, operand2;      /*两个字符变量分别用来存储操作数 1 和操作数 2*/
      int result;                                   /*存储计算结果*/
      Pop_SequenStack(S, &operand2);
      Pop_SequenStack(S, &operand1);
      /*根据运算符 operator，调用 arithmeticalOperate  函数进行四则运算*/
      result = arithmeticalOperate(changeCharToData(operand1),
                changeCharToData( operand2), operator);          /*将此次运算结果入栈*/
      Push_SequenStack(S, changeDataToChar(result));
}
/*  根据操作符和操作数进行四则运算  */
int arithmeticalOperate(int operand1, int operand2, char operator)
{     int result;                                   /*存储计算结果*/
      switch( operator )                            /*根据运算符选择相应的四则运算*/
      {           case '+': result = operand1 + operand2; break;
                  case '-': result = operand1 - operand2; break;
                  case '*': result = operand1 * operand2; break;
                  case '/': result = operand1 / operand2; break;
      }
      return result;                                /*返回运算结果*/
}
```

下面为 SequenStack.c 的代码。

```
#include <stdio.h>
#define    MAXSIZE    1024                        /*顺序栈可能的最大长度，假设为 1024 */
typedef    char    elemtype;                      /*elemtype 可为任意类型，假设为 char*/
typedef    struct    SequenStack
{     elemtype    data[MAXSIZE];                 /*定义顺序栈为一维数组*/
      int    top;                                 /*top 为栈顶指针*/
}SequenStack;                                    /*顺序栈的结构类型为 SequenStack */

/*初始化顺序栈*/
SequenStack * Init_SequenStack( )
{     SequenStack    *S;                         /*定义顺序栈指针变量*/
      S = (SequenStack *) malloc ( sizeof( SequenStack ) );       /*申请分配内存空间*/
      S->top = -1;                                /*设置顺序栈的栈顶指针*/
      return S;                                   /*返回顺序栈的首地址*/
}
/*判断顺序栈是否空*/
```

```
int SequenStack_Empty(SequenStack *S)
{    if ( S->top == - 1 )                    /*检查栈顶指针的值*/
     {    return 1;                          /*栈 S 为空，函数返回 1*/
     }
     else
     {    return 0;                          /*栈 S 为不空，函数返回 0*/
     }
}
/*判断顺序栈是否满*/
int SequenStack_Full(SequenStack *S)
{
     if ( S->top + 1 == MAXSIZE )            /*检查栈顶指针的值*/
     {    return 1;                          /*栈 S 为满，函数返回 1*/
     }
     else
     {    return 0;                          /*栈 S 为不满，函数返回 0*/
     }
}
/*入栈*/
int Push_SequenStack(SequenStack *S, elemtype x )
{    if ( S->top >= MAXSIZE - 1 )            /*检查顺序栈的长度*/
     {    printf("overflow\n");              /*输出溢出错误信息*/
          return 0;                          /*插入失败，函数返回 0*/
     }
     S->top++;                               /*将栈顶指针加 1*/
     S->data[S->top] = x;                    /*在数组中插入 x，成为新的栈顶元素*/
     return 1;                               /*插入成功，函数返回 1*/
}
/*出栈*/
int   Pop_SequenStack(SequenStack   *S, elemtype *x)
{    if (S->top == - 1)                      /*检查顺序栈的长度*/
     {    printf("error\n");                 /*栈为空，输出出错信息*/
          return 0;                          /*删除失败，函数返回 0*/
     }
     else
     {    S->top--;                          /*栈顶指针 top 的值减 1*/
          *x = S->data[S->top+1];            /*将原栈顶数据元素赋值给 x*/
          return 1;                          /*删除成功，函数返回 1*/
     }
}
/*读取栈顶数据元素*/
int   GetTop_SequenStack(SequenStack   *S, elemtype *x)
{    if (S->top == - 1)                      /*检查顺序栈的长度*/
     {    printf("error\n");                 /*栈为空，输出出错信息*/
          return 0;                          /*查找失败，函数返回 0*/
     }
```

```
    else
    {   *x = S->data[S->top];                    /*将栈顶数据元素赋值给 x*/
        return 1;                                /*查找成功，函数返回 1*/
    }
}
```

实验 5 串、数组和广义表

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

本题的关键是通过给定的行号 i 和列号 j 找出原矩阵的对应元素值，通过设计一个 value 函数，其功能是当在三元组表示中找到要找的元素时返回其元素值，找不到时原该位置元素为 0，因此返回 0。然后利用该函数计算出 C 的行号 i 和列号 j 处的元素值，若该值不为 0，则存入其三元组表示的矩阵中，否则不存入。根据题目要求完整程序如下：

```
#include <stdio.h>
#include <stdlib.h>
#define MAX   40      /*非零元个数的最大值*/
typedef struct
{   int i,j;          /*非零元的行下标和列下标*/
    int e;            /*该非零元的数值*/
}Triple;

typedef struct
{   Triple data[MAX+1];      /*非零元三元组表,data[0]未用*/
    int mu,nu,tu;            /*矩阵的行数，列数，非零元的个数*/
    int rpos[MAX+1];         /*各行第一个非零元的位置表*/
}Matrix;

/*稀疏矩阵的初始化函数*/
void InitMatrix(Matrix *W)
{   int i, p, num[MAX], rpos[MAX], r;
    printf("please input the number of row,col and fei ling yuan:\n");
    /*输入矩阵的行数，列数非零元的个数*/
    scanf("%d, %d, %d", &W->mu, &W->nu, &W->tu);
    printf("please input the data:\n");
    for(i = 1;i <= W->tu; i++)              /*输入矩阵的行号，列号和非零元的数值*/
    scanf("%d, %d, %d", &W->data[i].i, &W->data[i].j, &W->data[i].e);
    /*求矩阵 W 各行第一个非零元在 W.data 中的位置*/
    for(r = 1;r <= W->mu; r++)
        num[r] = 0;                         /*初始化 W 每一行的非零元的个数为 0*/
    for(p = 1;p <= W->tu; p++)
        num[W->data[p].i] = num[W->data[p].i] + 1;
    W->rpos[1] = 1;
    for(r = 2; r <= W->mu + 1; r++)
        W->rpos[r] = W->rpos[r - 1] + num[r - 1];
}
```

```
/*用三元组表示的稀疏矩阵 M 和 N 相乘的函数*/
void MulMatrix(Matrix M, Matrix N, Matrix *Q)
{    int t, p, q, tp, k, arrow, brow, ccol, ctemp[MAX], num[MAX];
     if(M.nu != N.mu)   printf("error");/*如果 M 矩阵的列数不等于 N 矩阵的行数输出错误信息*/
     /* 初始化 Q 的行数，列数和非零元的个数*/
     Q->mu = M.mu;
     Q->nu = N.nu;
     Q->tu = 0;
     if(M.tu * N.tu != 0)                    /*如果 Q 是非零矩阵继续下面的运行*/
     {    for(arrow = 1; arrow <= M.mu; ++arrow)
          {    for(k = 1; k <= M.mu; k ++)
                    ctemp[k] = 0;         /*将累加器清零*/
               Q->rpos[arrow] = Q->tu + 1;
               if(arrow < M.mu)   tp = M.rpos[arrow + 1];
               else {      tp = M.tu + 1;}
               /*对当前行中每一个非零元找到对应元在 N 中的位置*/
               for(p = M.rpos[arrow]; p < tp; ++p)
               {
                    brow = M.data[p].j;
                    if(brow < N.mu)   t = N.rpos[brow + 1];
                    else {      t = N.tu + 1;}
                    for(q = N.rpos[brow]; q < t; ++q)
                    {
                         ccol = N.data[q].j;         /*乘积元在 Q 中的列号*/
                         ctemp[ccol] += M.data[p].e * N.data[q].e;
                    }
               }                    /*求得 Q 中第 arrow 行的非零元*/
               for(ccol = 1; ccol <= Q->nu; ++ccol)
                    if( ctemp[ccol] )
                    {    /*压缩存储该行的非零元*/
                         /*若 Q 中的非零元的个数超出最大范围输出错误的信息*/
                         if(++Q->tu > MAX)       printf("error");
                         Q->data[Q->tu].i = arrow;
                         Q->data[Q->tu].j = ccol;
                         Q->data[Q->tu].e = ctemp[ccol];
                    }
          }
     }
     /*求出 Q 中每一行非零元在 Q.data 中的位置*/
     for(t = 1; t <= (*Q).tu; t++)
          ++num[(*Q).data[t].i];
     (*Q).rpos[1] = 1;
     for(t = 2; t <= (*Q).mu; t++)
          (*Q).rpos[t] = (*Q).rpos[t - 1] + num[t - 1];
}
```

```
/* 矩阵的输出函数*/
void output(Matrix *P)
{    int i, j, t, k = 0;
     t = 1;
     printf(" the array   is: \n");
     for(i = 1; i <= P->mu; i ++)
     {    for(j = 1; j <= P->nu; j ++)
          {    if(P->data[t].i == i && P->data[t].j == j)
               {    printf("%4d", P->data[t].e); t++; }
               else printf("%4d", k);
          }
          printf("\n");
     }
 }

void main( )
{    int i;
     Matrix A,B,C;/*定义 Matrix 类型的三个矩阵 A，B，C*/
     printf("1 mult array,A*B=C\n");              /*使用说明*/
     printf("2 exit\n");
     while(1)
     {    printf("please select item to operarting:");
          scanf("%d", &i);
          switch(i)                               /*用 switch 语句实现各种功能的转换*/
          {    case 1:   InitMatrix(&A);
                         output(&A);
                         InitMatrix(&B);
                         output(&B);
                         MulMatrix(A, B, &C);
                         output(&C);
                         break;
               case 2:   exit(0);
                         break;
          }
     }
}
```

实验 6 树和二叉树

本实验示例要求输入一串电文字符实现哈夫曼编码，再对哈夫曼编码生成的代码串进行译码，输出电文字符串。

要实现本设计的要求，必须实现以下功能：

（1） 电文字符串中字符的个数及每个字符出现频率的统计

（2） 哈夫曼树的建立

（3） 哈夫曼编码的生成

（4） 对电文编码

（5） 电文的译码

由于配套教材中给出了哈夫曼树的建立和哈夫曼编码的算法，下面简单分析电文字符串的统计：

假设电文字符串只包含英文字母（不区分大小写），该算法的实现思想是：定义一个含有 26 个元素的临时整型数组，用来存储每个字母出现的次数。首先将电文中的字母全部转换为大写字母，因为大小字母的 ASCII 码与整数 1~26 之间相差 64，因此在算法中使用字母减去 64 作为统计数组的下标对号入座，然后用一个循环判断统计好的各类字符个数的数组元素是否为零，若不为零，表示该字符在电文中出现，则将其值存入一个数组对应的元素中，同时将其对应的字符也存入另一个字符数组的元素中。

完整的程序代码如下：

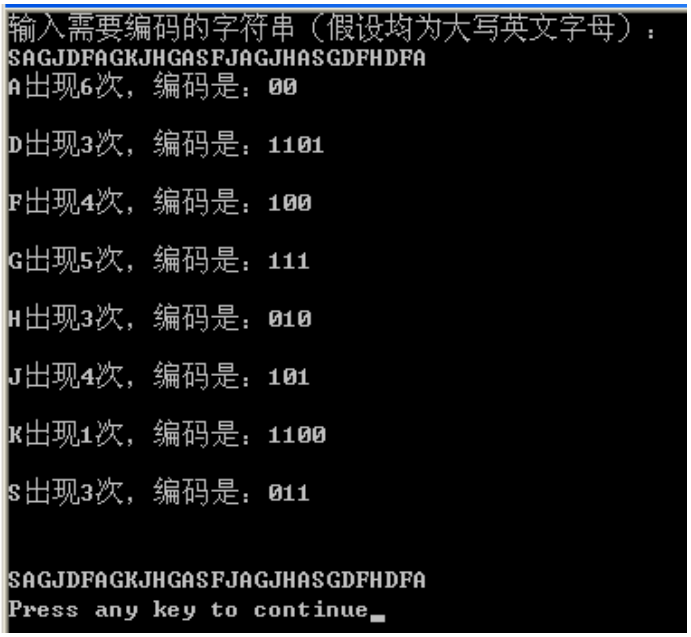
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
# define MAXBIT 10                    /*每个字符编码的最大长度*/
# define MAXVALUE 1000                /*哈夫曼树中叶子结点的最大权值*/
#define MAXSIZE 26                    /*电文中出现的字符种类的最大值*/
typedef struct HNode                  /*定义哈夫曼树的结点结构*/

{    int weight;
     int parent , lchild , rchild;
}HNode,*HTree;

typedef char** HCode;                 /*哈夫曼编码*/

/*统计电文中的字符种类及每种字符出现的次数*/
int Count(char *s, int cnt[], char str[])
{    /*s 指向电文字符串，cnt 存储电文中字符出现的次数，str 存储电文中出现的字符*/
     char *p;
     int i, j, k;
     int temp[26];                          /*temp 是临时数组，用于统计每个字符出现的次数*/
     for(i = 0; i < 26; i ++)    temp[i] = 0;
     for(p = s; *p != '\0'; p ++)           /*统计各种字符出现的次数*/
          if(*p >= 'A' && *p <= 'Z')
          {
               k = (*p) - 65;
               temp[k]++;
          }
     for(i = 0, j = 0; i < 26; i ++)        /*统计电文中出现的字符*/
          if(temp[i] != 0)
          {    str[j] = i + 65;            /*将出现的字符存入字符数组*/
               cnt[j] = temp[i];           /*相应字符出现的次数作为该字符的权值存储到 cn*/
               j ++;
          }
     str[j] = '\0';
     return j;                              /*返回电文中字符的种类，即哈夫曼树种叶子结点个数*/
```

```
}

/*构造哈夫曼树并生成叶子结点的前缀编码*/
void HuffmanCoding(HTree *HT, HCode *HC, int *w , int n)
{    /* w 存储 n 个字符的权值，构造哈夫曼树 HT，并求出 n 个字符的哈夫曼编码 HC。*/
     int m;                          /*哈夫曼树中结点个数*/
     int m1, m2, x1, x2;             /*m1、m2 存储两个最小权值，x1、x2 存储对应的下标*/
     int i, j, start;

     char * cd;
     int c, f;
     HNode *p;

     if (n <= 1) return;
     /*哈夫曼树的构造*/
     m= 2*n - 1;
     *HT = (HNode *)malloc (m*sizeof(HNode) );
     for(p =*HT, i = 0; i < n; ++ i, ++ p, ++w)          /*初始化叶子结点信息*/
     {    p->weight = *w;    p->lchild = -1; p->rchild = -1; p->parent = -1;}
     for( ; i < m; ++ i, ++ p)                           /*初始化分支结点信息*/
     {    p->weight = 0; p->lchild = -1; p->rchild = -1; p->parent = -1;}
     for(i = n; i < m; ++ i)                             /*构造哈夫曼树 */
          {    m1 = m2 = MAXVALUE;
          x1 = x2 = 0;
          for(j = 0; j < i; ++ j)                        /*寻找 parent 为-1 且权值最小的两棵子树*/
          {    if((*HT)[j].parent == -1 && (*HT)[j].weight < m1)
               {    m2 = m1; x2 = x1; m1 = (*HT)[j].weight; x1 = j;
               }
               else if((*HT)[j].parent == -1 && (*HT)[j].weight < m2)
               {    m2 = (*HT)[j].weight; x2 = j;
               }
          }
          /*合并成一棵新的子树*/
          (*HT)[x1].parent = i; (*HT)[x2].parent = i;
          (*HT)[i].lchild = x1; (*HT)[i].rchild = x2;
          (*HT)[i].weight = m1 + m2;
     }
     /*生成字符的前缀编码*/
     *HC = ( HCode ) malloc ( n * sizeof(char*) );
     cd = (char *) malloc( n * sizeof(char) );
     cd[n-1] = '\0';
     for(i = 0; i < n; ++ i)                             /* 从叶子到根逆向求每个字符的哈夫曼编码 */
     {    start = n -1;
          for( c = i, f = (*HT)[i].parent; f != -1; c = f, f = (*HT)[f].parent)
               if((*HT)[f].lchild == c) cd[--start] = '0';
               else        cd[--start] = '1';
```

```
            (*HC)[i] = ( char*) malloc((n-start)*sizeof(char));      /*为第 i 个字符编码分配空间 */
            strcpy((*HC)[i], &cd[start]);                            /*从 cd 复制编码串到 HC[i] */
        }
        free(cd);
}

/*对电文进行编码并写入文件*/
void Coding(HCode HC, char *s, char str[])
{    /*s 指向电文字符串，str 存储电文中出现的字符，HC 字符的哈夫曼编码表*/
        int i,j;
        char *cp;
        FILE *fp;
        fp = fopen("\\codefile.txt", "w");
        while(*s)                                /*对电文中的字符逐一生成编码并写入文件*/
        {
            for(i = 0; i < MAXSIZE; i ++)
                if(str[i] == *s)
                {    for(j = 0, cp = HC[i]; j < strlen(HC[i]); j ++, cp ++)
                        fputc(*cp, fp);
                     break;
                }
            s++;
        }
        fclose(fp);
}

/*输出电文中每个字符出现的次数及其前缀编码*/
void Print(HCode HC, char str[], int cn[], int n)
{
        int i;
        for(i = 0; i < n; i ++)
        {    printf("%c 出现%d 次，编码是：", str[i], cn[i]);
            puts(HC[i]);
            putchar('\n');
        }
        return;

}

/*对编码文件进行译码，恢复原电文内容*/
char *Decode(HCode HC, char str[], int num)
{
        FILE *fp;
        char s[254];                             /*假设原文本文件不超过 254 个字符*/
        char *p;
        static char cd[MAXBIT + 1];
```

```
    int i, j, k = 0, cjs;
    fp = fopen("\\codefile.txt","r");
    while(!feof(fp))
    {   cjs = 0;
        for(i = 0; i < MAXSIZE && cjs == 0 && !feof(fp); i ++)
        {   cd[i] = ' '; cd[i + 1] = '\0';
            cd[i] = fgetc(fp);
            for(j = 0; j < num; j ++)
                if(strcmp(HC[j], cd) == 0)
                {   s[k] = str[j];
                    k ++;
                    cjs = 1;
                    break;
                }
        }
    }
    s[k] = '\0';
    p = s;
    return p;
}

/*主函数*/
void main()
{
    char st[254], *s, str[26];
    int cn[26];
    int num;
    HNode *HT;
    HCode HC;
    printf("输入需要编码的字符串（假设均为大写英文字母）：\n");
    gets(st);
    num = Count(st, cn, str);
    HuffmanCoding(&HT, &HC, cn, num);
    Print(HC, str, cn, num);
    Coding(HC, st,str);
    printf("\n");
    s = Decode(HC, str, num);
    printf("%s\n", s);
}
```

程序运行结果如实验图 6-2 所示。



实验图 6-2 程序运行结果

实验 7　图

有全国 25 个主要城市的公路交通图如实验图 7-4 所示，顶点表示城市，边表示城市之间有公路相连，边上的权值表示城市之间公路的长度。

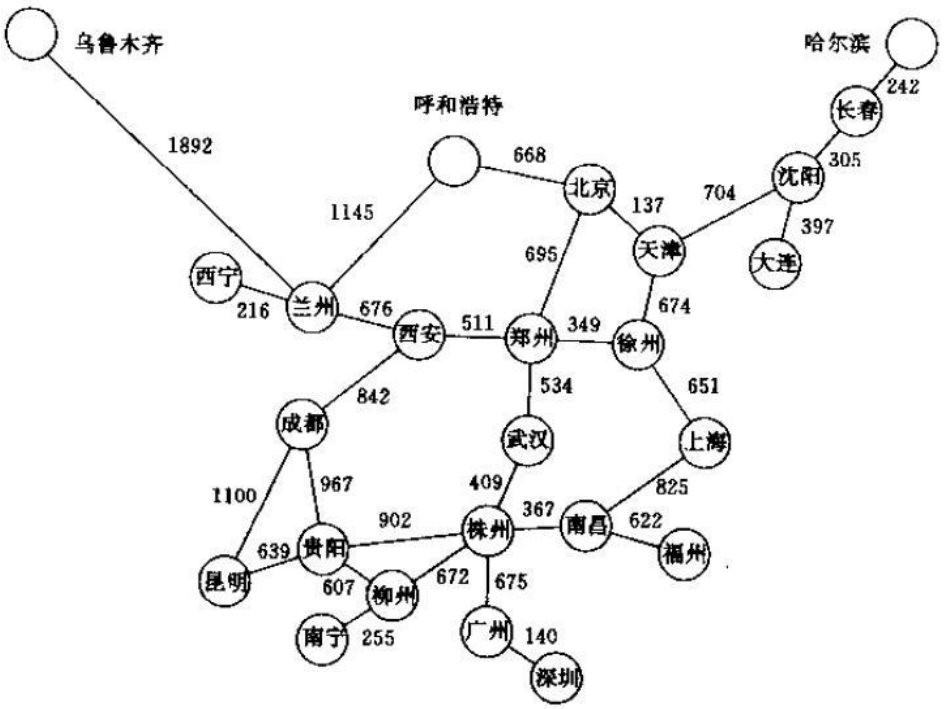
编程解决下列问题：

（1）输入城市信息和城市之间公路的信息，建立图的邻接矩阵存储结构。

（2）为了使所有城市之间能够通信，将沿公路铺设光纤，编程给出合理的铺设方案，使光纤总耗费最小。

（3）根据上图建立简单的交通咨询系统，当用户输入任意出发城市后，能够输出到其它所有城市的合理的行车路线，使路程长度最短。

分析：第二个问题实际上是求最小生成树的问题，选用普里姆算法解决；第三个问题实际上是求单源最短路径问题，采用迪杰斯特拉算法解决。



实验图 7-4 城市公路交通图

完整程序代码如下：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MaxVertexNum 25                 /*最大顶点数设为 100*/
#define INF 32767                       /*INF 表示∞*/
typedef int EdgeType;                   /*边的权值设为整型*/
typedef struct                          /*邻接矩阵类型定义*/
{    char citys[MaxVertexNum][10];                          /*顶点表*/
     EdgeType edges[MaxVertexNum][MaxVertexNum];  /*邻接矩阵，即边表*/
     int n, e;                                              /*顶点数和边数*/
}MGraph;                                                    /*Maragh 是以邻接矩阵存储的图类型*/

int CityID(MGraph *G,char CityName[])
{   /*返回城市名 CityName 在邻接矩阵 G 中序号，没有改城市名返回-1*/
    int i;
    for(i = 0; i < G -> n; i ++)
    {    if(strcmp(CityName, G->citys[i]) == 0)
             break;
    }
    if(i == G->n)
         return -1;
    else
         return i;
}

void CreateMGraph(MGraph *G)
```

```
{    /*建立交通图的邻接矩阵存储函数*/
     int i, j, k, l;
     char t[10], m[10];
     printf("\n 请输入城市信息，输入 q 退出：\n");
     for(i = 0; ; i ++)
     {    printf("\n 城市名称%d：", i);
          scanf("%s", t);
          if(strcmp(t, "q") == 0)
               break;
          if(CityID(G, t) >= 0)              /*检查重复输入*/
          {    printf("\n 已有此城市!\n");
               i --;
               continue;
          }
          strcpy(G->citys[i], t);
          G->n = i + 1;                      /*记录当前输入城市数目*/
     }
     printf("\n 请输入公路的信息，输入 q 退出：\n");
     for(i = 0; ; i ++)
     {    printf("\n 起始城市：");
          scanf("%s", t);
          if(strcmp(t, "q") == 0)
               break;
          printf("终止城市：");
          scanf("%s", m);
          if(strcmp(m, "q") == 0)
               break;
          printf("公路长度：");
          scanf("%d", &l);
          if(strcmp(t, "q") == 0)
               break;
          j = CityID(G, t);
          k = CityID(G, m);
          if(-1 == j || -1 == k)             /*检查城市名称输入正确性*/
          {    printf("\n 城市名称输入有误！\n");
               continue;
          }
          G->edges[j][k] = l;/*因为是无向图，所以邻接矩阵对称位置都要记录边的权值*/
          G->edges[k][j] = l;
     }
     G->e = i;                               /*记录边的数目*/
}

typedef struct                /*定义 closedge 数组类型，作为普里姆算法的辅助数组类型*/
{
     int adjvex;
```

```
    int lowcost;
}Closedge;

void Prim(MGraph *G,int v)
{   /*输出最小生成树的普里姆算法。*/
    /*G 为图的邻接矩阵存储，v 是第一个进入集合 U 中的顶点的序号*/
    /*算法将依次输出需要铺设光缆的公路*/
    int k, j, i, minCost;
    Closedge closedge[MaxVertexNum];   /*定义辅助数组*/
    closedge[v].lowcost = 0;
    for (j = 0; j < G->n; j ++)              /*初始化 closedge 数组*/
        if (j != v)
        {   closedge[j].adjvex = v;
            closedge[j].lowcost = G->edges[v][j];
        }
    for (i = 1; i < G->n; i ++)              /*依次将顶点加入到集合 U 中*/
    {   for(j = 0; j < G->n; j++)            /*定位第一个没有加入到 U 中的顶点*/
            if(closedge[j].lowcost != 0)
            {   k = j;
                break;
            }
        minCost = closedge[k].lowcost;   /*定位 V-U 集合中 lowcost 值最小的顶点*/
        for (j = 0; j < G->n; j ++)
            if (closedge[j].lowcost < minCost && closedge[j].lowcost != 0)
            {   minCost = closedge[j].lowcost;
                k = j;
            }
        printf("(%s,%s)\n", G->citys[closedge[k].adjvex], G->citys[k]);
                                             /*输出新加入到树中的边*/
        closedge[k].lowcost = 0;             /*将该顶点加入到集合 U*/
        for (j = 0; j < G->n; j ++)          /*更新 closedge 数组的内容*/
            if (G->edges[k][j] < closedge[j].lowcost)
            {   closedge[j].adjvex = k;
                closedge[j].lowcost = G->edges[k][j];
            }
    }
     getchar();
}

void dispath(MGraph *G,int dist[], int path[], int s[], int v)
{   /*输出最短路径的算法，最短路径将由终点到起点倒着输出*/
    int i, k;
    for(i = 0; i < G->n; i ++)
        if(s[i] == 1)                        /*S 中顶点*/
        {   k = i;
            printf("\n%s 到 %s 的最短路径为:", G->citys[v], G->citys[i]);
```

```
            while(k != v)
            {   printf("%s<-", G->citys[k]);
                k = path[k];
            }
            printf("%s 路径长度为:%d\n", G->citys[v], dist[i]);
        }
        else
            printf("%s<-%s 不存在路径\n",G->citys[i],G->citys[v]);
}

void dijkstra(MGraph *G, int v)
{   /*迪杰斯特拉算法求最短路径，G 为指向图的邻接矩阵的指针，v 是源点的序号*/
    int dist[MaxVertexNum], path[MaxVertexNum];
    int s[MaxVertexNum];    /*S 数组用来标记已经找到最短路径的顶点*/
    int mindis;
    int i, j, k;
    for(i = 0;i < G->n; i ++)
    {   dist[i] = G->edges[v][i];           /*距离初始化*/
        s[i] = 0;                           /*s 数组初始化，0 表示未找到最短路径*/
        if(G->edges[v][i] < 32767)          /*路径初始化*/
            path[i] = v;
        else
            path[i] = -1;
    }
    s[v] = 1;                               /*源点 v 放入 S 中*/
    for(i=0;i<G->n;i++)         /*重复，直到求出 v 到其余所有顶点的最短路径*/
    {   mindis = INF;
        k = v;
        for(j = 0; j < G->n; j ++)          /*从 V-S 中选取具有最小距离的顶点 v k*/
        {   if(s[j] == 0 && dist[j] < mindis)
            {   k = j;
                mindis = dist[j];
            }
        }
        s[k] = 1;                           /*将顶点 k 加入 S 中*/
        for(j = 0; j < G->n; j ++)          /*修改 V-S 中顶点的距离 dist[j]*/
        {   if(s[j] == 0 && G->edges[k][j] < 32767 && dist[k] + G->edges[k][j] < dist[j])
            {   dist[j] = dist[k] + G->edges[k][j];
                path[j] = k;
            }
        }
    }
    dispath(G, dist, path, s, v);   /*输出最短路径*/
}

void consult(MGraph *G)
```

```
{    /*交通咨询函数，调用迪杰斯特拉算法*/
     char c[10];
     int i;
     printf("\n 请输入出发城市：");
     scanf("%s", c);
     i = CityID(G, c);
     if(-1 == i)
     {    printf("\n 找不到该城市！");
          getchar();
          return;
     }
     dijkstra(G, i);
     getchar();
}

int menu( )                                   /*菜单显示函数*/
{    int n;char c;
     printf("\n\n\t\t 城市公路交通系统：\n");
     printf("\n\t\t\t1. 建立城市公路交通图");
     printf("\n\t\t\t2. 光纤铺设解决方案");
     printf("\n\t\t\t3. 交通咨询");
     printf("\n\t\t\t0. 退出");
     do
     {    fflush(stdin);
          printf("\n\n\t\t 请输入数字 0～3 选择功能:");
          c = getchar();n = c - 48;
          if (n < 0 || n > 3)
               printf("\t\t\t 输入选项错误！请重新输入选项");
     }while(n < 0 || n > 3);
     return n;
}/*menu*/

void main()
{    int select, i, j;
     char c[10];
     MGraph *G;                                    /*初始化邻接矩阵*/
     G = (MGraph *)malloc(sizeof(MGraph));
     G->n = 0;
     G->e = 0;
     for(i = 0; i < MaxVertexNum; i ++)
          for(j = 0; j < MaxVertexNum; j ++)
               if(i != j)
                    G->edges[i][j] = INF;
               else
                    G->edges[i][j] = 0;
     do
```

```
    {   system("cls");
        select = menu();
        switch (select)
        {   case 1: CreateMGraph(G); break;              /*创建邻接矩阵*/
            case 2: Prim(G, 0); break;                   /*给出光缆铺设方案*/
            case 3: consult(G);break;                    /*交通咨询*/
            case 0: printf("\n\n\n\t\t\t 谢谢使用！再见...\n");getchar();
                    exit(0);
        }
        getchar();
    }while(select != 0);
}
```

实验 8 排序

由于配套教材已经给出了大部分排序算法的源码，因此此次实验主要考察对算法的理解及综合运用能力，由于需要每种算法对关键字的比较及其移动次数，就需要在每个算法中当进行相关操作时进行计数。下面给出计数实现的一个简单思路：

1．为了程序可读性，定义如下的常量，表示每种排序算法：

```
#define   INSERT_SORT 0
#define   SHELL_SORT 1
#define   BUBBLE_SORT 2
#define   QUICK_SORT 3
#define   SIMPLE_SELECTION_SORT 4
#define   HEAP_SORT 5
#define   MERGING_SORT 6
```

2．再定义如下常量表示比较的类别：

```
#define   MOVING_COUNT 0                 /*移动次数*/
#define   COMPARING_COUNT 1              /*比较次数*/
```

3．定义一个二维数组用于计数，为了操作方便，可以把该数组定义为全局变量。当然在计数前要注意数组必须初始化为 0。

```
int   result[7][2];
```

4．经过上面一系列的准备，统计计数变得简单起来，只需要在算每次进行比较或交换的时候执行如下操作：

```
/*表示插入排序移动次数增 1，其它计数类似*/
  result[INSERT_SORT][ MOVING_COUNT]++;
```

5．为了使算法间的比较结果更加客观，建议待排序记录的长度不低于 50，用至少 5 组数据进行测试，移动和比较次数可以取平均值。

6．最终输出的统计表格也就是输出二维数组的内容即可。

下面讨论随机序列如何生成。

1．随机数生成用到一个叫 rand 的函数，该函数声明在 stdlib.h 文件中，因此在程序开始需要包含头文件：

```
#include<stdlib.h>
```

2．rand 的函数说明如下：

函数原型：int rand(void)

函数功能：返回 0 到 RAND_MAX 之间的随机整数，RAND_MAX 通常为 32767。

3．返回 1-10 之间的随机整数。

当需要返回某个范围内的随机整数，可以用如下操作：

```
int m = rand() ;          /*取得随机数在 0 到 RAND_MAX 之间*/
m = m%10;                 /*此时 m 在 0-9 之间*/
m=m+1;                    /*此时 m 在 1-10 之间*/
```

程序源代码如下：

```
#include<stdlib.h>
#include <stdio.h>
#define   N   50
#define   INSERT_SORT 0                    /*定义调试算法的常量*/
#define   SHELL_SORT 1
#define   BUBBLE_SORT 2
#define   QUICK_SORT 3
#define   SIMPLE_SELECTION_SORT 4
#define   HEAP_SORT 5
#define   MERGING_SORT 6
#define   MOVING_COUNT 0                   /*定义表示统计类别的常量*/
#define   COMPARING_COUNT 1
/*定义全局二维数组，一共 7 种算法，2 种统计类别（移动和比较次数）*/
int   result[7][2];

void InsertionSort(int * pData, int n)                    /*直接插入排序*/
{
    int temp;
    for(int i = 1;i < n; i ++)                    /*对于 N 个关键字，进行 N-1 次比较*/
    {
        temp = pData[i];                                  /*保存待插入关键字*/
       /*往后移动关键字，以便找到正确的插入位置*/
       for (int j = i - 1 ; j >= 0 && temp < pData[j] ; j--)
       {
          pData[j + 1] = pData[j];
          result[INSERT_SORT][MOVING_COUNT]++;  /*更改比较和移动次数*/
          result[INSERT_SORT][COMPARING_COUNT]++;
       }
       pData[j + 1] = temp;
       result[INSERT_SORT][MOVING_COUNT]++;
    }
}

void ShellSort(int * pData, int n, int delta)             /*一趟希尔排序*/
{
    int temp;
    for(int i = 1; i < n; i = i + delta)
    {
     temp = pData[i];
     for (int j = i - delta ; j >= 0 && temp < pData[j] ; j = j - delta)
```

```
        {
            result[SHELL_SORT][MOVING_COUNT]++;     /*更改比较和移动次数*/
            result[SHELL_SORT][COMPARING_COUNT]++;
            pData[j + delta] = pData[j];
         }
         pData[j + delta] = temp;
         result[SHELL_SORT][MOVING_COUNT]++;
    }
}

void Shell(int * pData, int n)                          /*希尔排序*/
{   int k = 4;
    while (k >= 1)
    {   ShellSort(pData, n, k);
        k = k / 2;
    }
}

void BubbleSort(int a[], int n)                         /*冒泡排序*/
{   int temp;
    /*根据结论 1：n 个关键字，最多需要 n-1 次冒泡处理*/
    for (int i = 1; i <= n-1; i ++)
    {   for(int j = 1; j <= n - i; j ++)
        /* 根据结论 3：对于 n 个关键字，在第 i 趟中，进行 n-i 次比较*/
        {   result[BUBBLE_SORT][COMPARING_COUNT]++;
            if(a[j-1]>a[j])
            {   temp = a[j]; a[j] = a[j - 1];a[j - 1] = temp;}          /*交换前后 2 个关键字*/
                result[BUBBLE_SORT][MOVING_COUNT] += 3;
        }
    }
}

int   Partition(int a[], int i, int j)                  /*一趟快速排序*/
{   int temp = a[i];                                    /*选择首元素为划分元素*/
    while (i < j)                                       /*根据结论 2，当 i>=j 结束本次划分*/
    {   while(a[j] >= temp && i < j)                    /*从后往前找到第一个小于划分元素的元素*/
        {   j --;
            result[QUICK_SORT][COMPARING_COUNT]++;
        }
        if(i < j)                                       /*若找到则移动元素，并把下标 i 增 1*/
        {   a[i++] = a[j];
            result[QUICK_SORT][MOVING_COUNT]++;
        }
        while(a[i] <= temp && i < j)                    /*从前往后找到第一个大于划分元素的元素*/
        {   i++;
            result[QUICK_SORT][COMPARING_COUNT]++;
```

```
            }
            if(i < j)                           /*若找到则移动元素，并把下标 i 减 1*/
            {   a[j--] = a[i];
                result[QUICK_SORT][MOVING_COUNT]++;
            }
        }
        a[i] = temp;                            /*划分完毕，最后设置把划分元素置于目标位置 */
        result[QUICK_SORT][MOVING_COUNT]++;
        return i;                               /*返回划分元素所在下标*/
}

void QuickSort(int a[], int i, int j)           /*快速排序*/
{   int k;
    if(i < j)                   /*程序采用递归实现，这里是递归的结束条件*/
    {   k = Partition(a, i, j);                 /*调用划分函数，把序列分成两部分*/
        QuickSort(a, i, k - 1);                 /*对前半部分，进行快速排序*/
        QuickSort(a, k + 1, j);                 /*对后半部分，进行快速排序*/
    }
}

void SimpleSelectionSort(int a[],int n)         /*简单选择排序*/
{   int min;                                    /*min 存放最小元素的下标*/
    int temp;
    for (int i = 0; i < n - 1; i ++)
    {   min = i;                                /*赋予初始值 */
        for (int j = i + 1; j < n; j ++)
        if(a[j] < a[min])                       /*找寻最小值*/
        {   min = j;
            result[SIMPLE_SELECTION_SORT][COMPARING_COUNT]++;
        }
        temp = a[i]; a[i] = a[min]; a[min] = temp;          /*交换，交换计数为 3 次*/
        result[SIMPLE_SELECTION_SORT][MOVING_COUNT]+=3;
    }
}

#define   LeftChild(i) (2 * (i) +1 )            /*定义一个宏，便于取得左孩子下标*/

void BuildDown(int a[], int n, int rootIndex)           /*一次堆调整*/
{
    int root = a[rootIndex];                            /*取出根结点数据*/
    int childIndex = LeftChild(rootIndex);              /*取出左孩子的下标 */
    while(childIndex < n )                              /*childIndex 小于 n 表示调整并未结束*/
    {   /*不是最后一个结点，表示 root 还有右孩，同时保证右孩子大于左孩子*/
        if (childIndex != n - 1 && a[childIndex + 1] > a[childIndex])
        {   childIndex ++;                              /*使得 childIndex 指向较大的孩子*/
            result[HEAP_SORT ][COMPARING_COUNT]++;   }
```

```
        if (root < a[childIndex])
        {
            a[rootIndex] = a[childIndex];          /*较大数据往上移动*/
            result[HEAP_SORT ][MOVING_COUNT]++;
            rootIndex = childIndex;                /*设定新的根结点下标*/
            childIndex = LeftChild(rootIndex);     /*设定新的左孩子下标*/
        }
        else                                       /*否则调整结束 */
            break;
    }
    a[rootIndex] = root;                           /*将根结点置于目标位置*/
}

void HeapSort(int a[], int n)                      /*堆排序*/
{   int temp;
    /*进行排序前，需要把随机序列构建成堆的结构。*/
    for (int rootIndex = (n-2)/2 ; rootIndex >= 0 ; rootIndex-- )
    BuildDown(a, n, rootIndex);
    for (int i = n - 1; i >= 0; i --)
    {   temp = a[0]; a[0] = a[i]; a[i] = temp;    /*把根结点（最大值）交换到子序列末尾*/
        result[HEAP_SORT ][COMPARING_COUNT] += 3;
        BuildDown(a, i, 0);                        /*重新调整，构建堆结构*/
    }
}

/* 合并两个有序序列，用于归并排序*/
void Merge(int a[], int s1, int e1, int s2, int e2, int b[])
{   int k = s1 ;   int i = s1;
    while ((s1 <= e1) && (s2 <= e2))               /*当 2 分组都不为空时*/
    {   result[MERGING_SORT][COMPARING_COUNT]++;
        if(a[s1] <= a[s2])
        {   b[k++] = a[s1++];
            result[MERGING_SORT][MOVING_COUNT]++;
        }
        else
        {   b[k++] = a[s2++];
            result[MERGING_SORT][MOVING_COUNT]++;
        }
    }
    while (s1 <= e1)            /* 若 s1 分组有剩余数据，则直接移动到辅助数组*/
    {   b[k++] = a[s1++];
        result[MERGING_SORT][MOVING_COUNT]++;
    }
    while (s2 <= e2)            /* 若 s2 分组有剩余数据，则直接移动到辅助数组*/
    {   b[k++] = a[s2++];
        result[MERGING_SORT][MOVING_COUNT]++;
```

```
        }
        k --;
        while (k >= i)
        {     a[k] = b[k]; k--;    result[MERGING_SORT][MOVING_COUNT]++;       }
        /*把辅助空间内数据拷贝到原序列*/
    }

    /*归并排序  对数组 a 中的数据 a[i]-a[j]进行归并排序，排序用到辅助空间 b*/
    void MergeSort(int a[], int i, int j, int b[])
    {     int k ;
        if (i < j )                              /*当分组内数据个数大于 1 时，才需要排序*/
        {     k = (i + j ) / 2;                  /*找到分组中点*/
            MergeSort(a, i, k, b);               /*对左半段进行归并排序*/
            MergeSort(a, k + 1, j, b);           /*对右半段进行归并排序*/
            Merge(a, i, k, k + 1, j, b);         /*合并左右两段*/
        }
    }

/*生成随机序列，存放到数组 a 中，序列长度为 length,序列最大值为 max，最小值为 min*/
    void Generate_Random(int a[],int length,int max,int min)
    {     for(int i = 0; i < length; i ++)
            {     a[i] = rand() % (max – min + 1);          a[i] += min;
            }
    }

    /*主函数*/
    void main()
    {     int i, j;
        int data[N],back[N];
        for ( i = 0; i < 7; i ++)          /*初始化统计结果*/
          for( j = 0 ; j < 2; j ++)
              result[i][j] = 0;
        /*打印表头*/
        printf("\t\t 比较次数\t 移动次数\t 比较次数(有序)\t 移动次数(有序)\n");
        /*分别执行 7 中排序算法*/
        Generate_Random(data, 50, 10, 1);
        InsertionSort(data, 50);
        Generate_Random(data, 50, 10, 1);
        Shell(data, 50);
        Generate_Random(data, 50, 10, 1);
        BubbleSort(data, 50);

        Generate_Random(data, 50, 10, 1);
        QuickSort(data, 0, 49);
        Generate_Random(data, 50, 10, 1);
        SimpleSelectionSort(data, 50);
```

```
    Generate_Random(data, 50, 10, 1);
    HeapSort(data, 50);
    Generate_Random(data, 50, 10, 1);
    MergeSort(data,0,49,back);
    /*再定义一个二维数组，用于暂存统计结果*/
    int result_back[7][2];
    /*暂存统计结果*/
    for ( i = 0; i < 7; i ++)
       for( j = 0 ; j < 2; j ++)
           {   result_back[i][j] = result[i][j];
               result[i][j] = 0;
            }
    /*重新执行排序算法，计算当序列有序时的移动、比较次数*/
    InsertionSort(data, 50);
    Shell(data, 50);
    BubbleSort(data, 50);
    QuickSort(data, 0, 49);
    SimpleSelectionSort(data, 50);
    HeapSort(data, 50);
    MergeSort(data, 0, 49, back);
    char * name[7] ={"插入排序","希尔排序","冒泡排序","快速排序","简选排序","堆排序   ","归
                    并排序"};
    /*打印输出统计结果*/
    for ( i = 0; i < 7; i ++)
    {   printf("%s", name[i]);
        for( j = 0 ; j < 2; j ++)   printf("\t   %d\t", result_back[i][j]);
        for( j = 0 ; j < 2; j ++)   printf("\t   %d\t", result[i][j]);
        printf("\n");
    }
}
```

以下是程序执行后的输出结果，由于随机序列不同，统计结果会与实际执行程序的结果有出入。

```
                比较次数        移动次数        比较次数(有序)  移动次数(有序)
插入排序        613             564             49              0
希尔排序        486             399             87              0
冒泡排序        3675            1225            3675            1225
快速排序        96              294             49              1225
简选排序        147             81              147             0
堆排序          176             221             207             235
归并排序        572             216             572             153
Press any key to continue
```

实验 9 查找

本章安排的实验示例是一个综合性的查找算法比较程序，该程序中包含顺序查找、折半查找、二叉排序树、平衡二叉树四种查找算法，通过对随机序列的排序，程序打印输出下表：

排序算法	平均查找长度

顺序查找	
折半查找	
二叉排序树	
平衡二叉树	

在实现此程序时会遇到若干问题，其中待排序随机序列的产生、如何进行元素比较次数的计数（该计数用于计算平均查找长度）等问题的解决方法，参照实验 7 即可。下面给出程序源码：

```
#include <stdio.h>
#include <stdlib.h>
#define SEQ_SEARCH 0                    /*定义 4 个常量，表示各个查找算法*/
#define BINARY_SEARCH 1
#define BST_SEARCH 2
#define BALANCE_SEARCH 3
int ASL[4];                             /*存放 4 个算法的屏幕内查找长度*/

/*顺序查找，找到长度为 length 的数组 a 中的数据 key
  成功返回元素所在数组下标，否则返回-1
*/
int SeqSearch(int a[], int length, int key)
{    for(int i = 0; i < length; i ++)
     {    ASL[SEQ_SEARCH]++;
          if(a[i] == key)
          return i;
     }
     return -1;
}

/*二分查找，找到长度为 length 的数组 a 中的数据 key,
成功返回元素所在数组下标，否则返回-1*/
int BinarySearch(int a[], int n, int key)
{    int l = 0;   int h = n-1;   int m;
     while (l <= h)
     { m = (l + h) / 2;                      /* m 为中间元素下标*/
          ASL[BINARY_SEARCH]++;
          if( key == a[m])   return m;       /*相等，则找到*/
          ASL[BINARY_SEARCH]++;
          if( key < a[m])                    /*小于，在左半侧查找*/
               h = m - 1;
          else
               l = m + 1;                    /*大于，再右半侧查找*/
     }
     return -1;
}
#define   NULL    0
```

```
/*定义节点结构，为二叉排序树做准备*/
typedef struct Node                         /* 树中结点定义*/
{   int key;                                /* 关键字*/
    struct Node * pLeft;                    /* 指向左子树的指针 */
    struct Node * pRight;                   /* 指向右子树的指针 */
    int bf;                                 /*平衡因子，用以平衡二叉树*/
}Node;

/*
描述：      二叉排序树搜索算法
输入参数：  pRoot：子树中根结点指针
key：       被查找关键字
pParentNode：pRoot 的父结点，初始时 pRoot 指向根结点，其父结点为 NULL
输出参数：  pKeyNode : 查找成功时返回关键字所在结点的指针，不成功时，返回查找路径上不为空的最后一个结点，当 pKeyNode 返回 NULL 时，表示此时二叉排序树为一颗空树
返回：  查找成功返回 1，查找失败返回 0
*/
int SearchBST(Node * pRoot,int key, Node **pKeyNode, Node **pParentNode)
{   *pKeyNode = pRoot;
    *pParentNode = NULL;
    int found = 0;
    while(*pKeyNode)                    /*当节点不为空*/
    {   if(key > (*pKeyNode)->key)      /*大于则在右子树中查找*/
        {   ASL[BST_SEARCH]++;
            *pParentNode = *pKeyNode;
            *pKeyNode = (*pKeyNode)->pRight;
        }
        else if(key < (*pKeyNode)->key) /*小于则在左子树中查找*/
        {   ASL[BST_SEARCH]++;
            *pParentNode = *pKeyNode;
            *pKeyNode = (*pKeyNode)->pLeft;
        }
        else { return 1;}
    }                                   /*end while*/
    return 0;
}

/*
描述：      二叉排序树搜索算法，注意和上一个函数的不同，上一个函数在构建树的过程中使用，当插入新数据时，都要预选查看树中是否含有该数据，若没有才进行插入该函数在进行单纯查找时使用
输入参数：  pRoot：子树中根结点指针
key：       被查找关键字
返回：      查找成功返回值为 1，查找失败返回 0
*/
```

```
int SearchBST(Node * pRoot, int key)
{
    int found = false;
    Node * pKeyNode = pRoot;
    while(pKeyNode)
    {   if(key > pKeyNode->key)
        {   ASL[BST_SEARCH]++;
            pKeyNode = pKeyNode->pRight;
        }
        else if(key < pKeyNode->key)
        {
            ASL[BST_SEARCH]++;
            pKeyNode = pKeyNode->pLeft;
        }
        else { return true;}
    }
    return 0;
}

/*
描述：      平衡二叉树搜索算法，和上个函数基本过程一样，不同之处仅在于查找长度的计
            数
输入参数：  pRoot：子树中根结点指针
key：       被查找关键字
返回：      查找成功返回值为 1，查找失败返回 0
*/
int SearchBalanceBST(Node * pRoot, int key)
{   int found = 0;
    Node * pKeyNode = pRoot;
    while(pKeyNode)
    {   if(key > pKeyNode->key)
        {   ASL[BALANCE_SEARCH]++;
            pKeyNode = pKeyNode->pRight;
        }
        else if(key < pKeyNode->key)
        {
            ASL[BALANCE_SEARCH]++;
            pKeyNode = pKeyNode->pLeft;
        }
        else { return true;}
    }
    return false;
}

/*
描述：      二叉排序树插入算法
```

```
输入参数： key：被插入关键字
输入输出参数：pRoot：插入后新的二叉排序树的根结点指针
返回　　：　　插入成功返回 1，插入失败返回 0
*/
int InsertBST(Node ** pRoot, int key)
{
    Node * pKeyNode;            /*定义存放新的关键字的结点*/
    Node * pParentNode = NULL;
    /*如果该关键字已在树中，则插入失败*/
    if(SearchBST(*pRoot,key,&pKeyNode, &pParentNode)) return 0;
    /*若不在树中，则首先为新结点分配空间*/
    Node * pNewNode = (Node*)malloc(sizeof(Node));
    pNewNode->key = key;                                /*为新结点赋初值*/
    pNewNode->pLeft = pNewNode->pRight = NULL;
    /*若树为空树，则新结点为插入后的根结点*/
    if(*pRoot == NULL) (*pRoot) = pNewNode;
    /*比结点关键字小，插入到左孩子*/
    else if(key < pParentNode->key) pParentNode->pLeft = pNewNode;
    else pParentNode->pRight = pNewNode;
    return 1;
}

/*以下 4 个函数和平衡处理有关，请务必参照配套教材讲述该算法的执行过程，单纯阅读源代码，很难对算法流程充分理解*/
void RightRotate(Node **pNode)
{   Node * pAxis = (*pNode)->pLeft;
    (*pNode)->pLeft = pAxis->pRight;
    pAxis->pRight = *pNode;
    *pNode = pAxis;
}

void LeftRotate(Node ** pNode)
{   Node * pAxis = (*pNode)->pRight;
    (*pNode)->pRight = pAxis->pLeft;
    pAxis->pLeft = *pNode;
    *pNode = pAxis;
}

void LL_LR_Balance(Node ** pNode)
{
    Node * pLeft = (*pNode)->pLeft;
    switch(pLeft->bf)
    {
        case 1:
            (*pNode)->bf = pLeft->bf = 0;
            RightRotate(pNode);
```

```
                break;
            case -1:
                Node * pRight = pLeft->pRight;
                switch(pRight->bf)
                {
                    case 0:
                    (*pNode)->bf = pLeft->bf = 0;
                    break;
                    case 1:
                        pRight->bf = pLeft->bf = 0;
                        (*pNode)->bf = -1;
                        break;
                    case -1:
                        (*pNode)->bf = pRight->bf = 0;
                        pLeft->bf = 1;
                        break;
                }
                LeftRotate(&(*pNode)->pLeft);
                RightRotate(pNode);
                break;
        }
}

void RR_RL_Balance(Node ** pNode)
{
    Node * pRight = (*pNode)->pRight;
    switch(pRight->bf)
    {
        case -1:
            (*pNode)->bf = pRight->bf = 0;
            LeftRotate(pNode);
            break;
        case 1:
            Node * pLeft = pRight->pLeft;
            switch(pLeft->bf)
            {
                case 0:
                    (*pNode)->bf = pRight->bf = 0;
                    break;
                case 1:
                    (*pNode)->bf = pLeft->bf = 0;
                    pRight->bf = -1;
                    break;
                case -1:
                    pRight->bf = pLeft->bf = 0;
                    (*pNode)->bf = 1;
```

```
                    break;
                }
                RightRotate(&(*pNode)->pRight);
                LeftRotate(pNode);
                break;
        }
}

/*
描述：          平衡二叉树插入算法
输入参数：      pRoot：根结点指针
key: 被插入的关键字
chain:表示插入结点后，是否引起调整的连锁反应，初始为 0
返回：插入成功返回 1，插入失败返回 0
*/
int InsertBalanceBST(Node **pRoot, int key, int* chain)
{
  if((*pRoot) == NULL) /*表示未在树中找到 key，直接生成新的结点，用于存储 key*/
  {   *pRoot = (Node * )malloc(sizeof(Node));
      (*pRoot)->bf = 0;
      (*pRoot)->pLeft = (*pRoot)->pRight = NULL;
      (*pRoot)->key = key;
      *chain =1;                    /*插入新的结点引起树的高度变化*/
  }
  else
  {       if(key == (*pRoot)->key)      /*树中含有相同关键字，则插入失败*/
          { *chain = 0; return 0;}
    /*key 小于当前结点的 key，则递归在当前结点的左子树插入*/
          if(key < (*pRoot)->key)
          {     if(!InsertBalanceBST( &(*pRoot)->pLeft,key,chain)) return 0;
                if(*chain)              /*如果树的高度发上变化（插入新结点）*/
                {     /*此处在递归调用退出时调用，表示新结点在 pRoot 左子树插入*/
                      switch((*pRoot)->bf)
                      { /*若 pRoot 的 bf 等于 0，在其左子树插入新结点，则 bf 变为 1，树的高度增
        加，会导致平衡被破坏的连锁反应（*chain = 1）*/
                            case 0:   (*pRoot)->bf = 1;*chain = 1; break;
                        /*若 pRoot 的 bf 等于 1，在其左子树插入新结点，则平衡被破坏，需要进行
        调整，调整完后，不会导致树的高度增加，则反应结束（*chain=0）*/
                            case 1:   LL_LR_Balance(pRoot);   *chain = 0; break;
                       /*若 pRoot 的 bf 等于-1，在其左子树插入新结点，则 bf 变为 0，树的平衡没有
                被破坏，反应结束（*chain = 0）*/
                            case -1: (*pRoot)->bf = 0; chain = false;break;
                      }
                }
    }
    else        /*以下处理，与在左子树插入类似*/
```

```
        {     if(!InsertBalanceBST( &(*pRoot)->pRight, key, chain)) return 0;
              if(*chain)
               { switch((*pRoot)->bf)
                    {     case 0:    (*pRoot)->bf = -1;*chain = true; break;
                          case 1:    (*pRoot)->bf = 0; *chain = false; break;
                          case -1: RR_RL_Balance(pRoot); *chain = false; break;
              }
              }
        }
     }
     return 1;
  }
/*主函数*/
void main()
{
        int data[50];          /*测试数据*/
        int back[2];           /*临时数据存储区*/
        int temp;
        int i, j;
        /*产生递增序列*/
        for(i = 0; i < 50; i ++) data[i] = i;
         /*该循环目的是产生乱序的随机序列，之所以不直接调用 Generate_Random 函数，是因为
  防止产生重复数据，影响查询结果的统计*/
        for ( i = 0; i < 5000; i ++)
        {
              Generate_Random(back, 2, 49, 1);      /*该函数作用参看实验 8*/
              temp = data[back[0]];
              data[back[0]] = data[back[1]];
              data[back[1]] = temp;
        }
        Node * pTree = NULL;                        /*指向二分查找树的指针*/
        Node * pBTree = NULL;                       /*指向平衡二叉树的指针*/
        int chain = 0;
        for( i = 0; i < 50; i++)
        { /*分别构建二分查找树和平衡二叉树*/
              InsertBST(&pTree, data[i]);
              InsertBalanceBST(&pBTree, data[i], &chain);
        }
        /*重新构建有序序列，以便二分查找使用*/
        for(i = 0; i < 50; i ++)    data[i] = i;
        for(i = 0; i < 4; i ++)     ASL[i] = 0;                 /*初始化查找长度*/
        for (i = 49; i >= 0; i --)
        {     /*分别调用 4 中查找算法*/
              SeqSearch(data, 50, data[i]);
              BinarySearch(data, 50, data[i]);
              SearchBST(pTree, data[i]);
```

```
            SearchBalanceBST(pBTree, data[i]);
        }
    /*输出结果*/
    char *name1[ ] = {"顺序查找", "二分查找", "二分查找树", "平衡二叉树"};
    for (i = 0; i < 4; i++)
    {
        printf("\t %s \t %f\n", name1[i], (float)ASL[i] / 50);
    }
}
```

以下是程序的运行结果：