第三章 上机实验

本实验教材将按照教材的章节去安排实验，教师在使用过程中可以根据需要适当挑选其中的若干实验要求学生上机完成。对于每个实验，本实验教材都给出了一些代码示例，供学生参考。

3.1 实验一：线性表

3.1.1 背景知识

线性表是一直简单且应用广泛的基本结构，其特点是在非空的数据元素集合中，元素之间在逻辑上存在一个序偶关系，除了第一个元素外，都有一个直接前驱，除了最后一个元素外，都有一个直接后继。

线性表的抽象数据类型定义如下：

```
ADT List{
    数据对象：D={ai|ai∈ElemSet,i=1,2,...,n, n≥0,ElemSet为元素集合}
    数据关系：R={<ai-1,ai>|ai-1,ai ∈D,i=1,2,...,n}
    基本操作：
    InitList(&L);   //构造空线性表
    DestroyList(&L);   //销毁线性表
    ClearList(&L);     //将L置空
    ListEmpty(L);      //检查L是否为空
    ListLength(L);     //返回L中元素个数
    GetElem(L,i,&e);   //返回L中第i个元素赋予e
    LocateItem(L,e);   //返回L中e的位置
    PriorElem(L,cur_e,&pre_e); //将L中cur_e的前一个元素赋予pre_e
    NextElem(L,cur_e,&next_e); //将L中cur_e的后一个元素赋予next_e
    ListInsert(&L,i,e);    //在L中i位置前插入e
    Listdelete(&L,i,&e);   //删除L中第i个元素，并赋予e
    ListTravers(L);        //依次输出L中元素。
}
```

3.1.2 实验目的

- ✧ 熟悉线性表的定义及其顺序和链式存储结构
- ✧ 掌握工程中的头文件、实现文件和主文件之间的相互关系。
- ✧ 熟悉对线性表的一些基本操作和具体的函数定义。
- ✧ 熟悉C++环境中工程文件的使用。

3.1.3 实验要求

- ✧ 认真阅读本实验内容中给出的程序。
- ✧ 熟悉C++程序的基本结构，会创建工程文件。
- ✧ 在计算机上实现对线性表的基本操作文件（实现文件）。
- ✧ 编写自己的主文件实现实验中的“动手与实践”部分内容。

3.1.4 实验内容

3.1.4.1 示例一:整型元素顺序表操作。

1、建立一个头文件sqlist1.h，内容如下：

```
#define LIST_INIT_SIZE 100
#define LIST_INC_SIZE 20
typedef int ElemType;   //ElemType 定义为int类型
typedef struct{
   ElemType  * elem;
   int          listsize;
   int          length;
}SqList;
//初始化线性表
bool InitList_sq(SqList &L,int msize=LIST_INIT_SIZE);
//销毁线性表
```

```
void DestroyList_sq(SqList &L);
//清空线性表
void ClearList_sq(SqList &L);
//判断线性表是否为空
bool ListEmpty_sq(SqList L);
//判断线性表是否满
bool ListFull_sq(SqList L);
//求线性表长度
int ListLength_sq(SqList L);
//查找元素
int LocateItem_sq(SqList L,ElemType e);
//获取元素
bool GetItem_sq(SqList L,int i,ElemType &e);
//插入元素
bool ListInsert_sq(SqList &L,int i,ElemType e);
//删除元素
bool ListDelete_sq(SqList &L,int i,ElemType &e);
//遍历元素
void ListTraverse_sq(SqList L);
//扩展线性表
bool Increment(SqList &L,int inc_size=LIST_INC_SIZE);
```

2、建立一个实现文件sqlist1.cpp，内容如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include "sqlist1.h"

//初始化线性表
bool InitList_sq(SqList &L,int msize)
{
    L.elem=new ElemType[msize];
    if(!L.elem){cerr<<"分配内存错误!"<<endl;return false;}
    L.listsize=msize;
    L.length=0;
    return true;
}
```

```
//销毁线性表
void DestroyList_sq(SqList &L)
{
    delete [] L.elem;
    L.listsize=0;
    L.length=0;
}

//清空线性表
void ClearList_sq(SqList &L)
{
    L.length=0;
}
//判断线性表是否为空
bool ListEmpty_sq(SqList L)
{
    return (L.length==0);
}
//判断线性表是否满
bool ListFull_sq(SqList L)
{
    return(L.length==L.listsize);
}
//求线性表长度
int ListLength_sq(SqList L)
{
    return L.length;
}

//查找元素
int LocateItem_sq(SqList L,ElemType e)
{
    int i;
    for(i=0;i<L.length;i++)     //依次查找每个元素
        if(L.elem[i]==e)return i+1; //找到位序为i的元素
    return 0;
}
//获取元素
```

```
bool GetItem_sq(SqList L,int i,ElemType &e)
{

    if(i<1||i>L.length+1){printf("i值非法!");return false;}
    e=L.elem[i-1];
    return true;
}
//插入元素
bool ListInsert_sq(SqList &L,int i,ElemType e)
{
    int j;

    if(i<1||i>L.length+1){printf("i值非法!");return false;}
    if(ListFull_sq(L))  //线性表已满
    if(Increment(L))return false;   //扩展失败
    for(j=L.length-1;j>=i-1;j--)
       L.elem[j+1]=L.elem[j];
    L.elem[i-1]=e;
    ++L.length;
    return true;
}
//删除元素
bool ListDelete_sq(SqList &L,int i,ElemType &e)
{
    int j;
    if(i<1||i>L.length) {printf("i值非法!");return false;}
    if(ListEmpty_sq(L))return false;
    e=L.elem[i-1];
    for(j=i;j<=L.length-1;j++)
    L.elem[j-1]=L.elem[j];
    --L.length;
    return true;
}
//遍历线性表
void ListTraverse_sq(SqList L)
{
    int i;
    for(i=0;i<L.length;i++)
```

```
    cout<<L.elem[i]<<' ';
    cout<<endl;
}
bool Increment(SqList &L,int inc_size)
{
    //增加顺序表L的容量为listsize+inc_size
    int i;

    ElemType *a;
    a=new ElemType[L.listsize+inc_size];  // 给a指针动态分配内存
    if(!a)return false;           //分配失败
    for(i=0;i<L.length;i++)a[i]=L.elem[i];
    delete [] L.elem;
    L.elem=a;
    L.listsize+=inc_size;
    return true;
}
```

3、建立主程序exam1_1.cpp。

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include "sqlist1.h"

void main()
{
    SqList L;
    ElemType e;
    int i;

    cout<<"1)初始化顺序表L"<<endl;
    InitList_sq(L);
    cout<<"2)从键盘输入5个整型元素并插入线性表L: ";
    for (i=1;i<=5;i++){
        cin>>e;
        ListInsert_sq(L,i,e);
    }
```

```
    cout<<"3)输出线性表L:"<<endl;
    ListTraverse_sq(L);
    cout<<"4)顺序表L长度="<<ListLength_sq(L)<<endl;
    cout<<"5)顺序表L"<<(ListEmpty_sq(L)?"空":"非空")<<endl;

    cout<<"6)请输入顺序表L中待查元素位序:";
    cin>>i;
    if(GetItem_sq(L,i,e))
        cout<<"顺序表L的第"<<i<<"个元素是:"<<e<<endl;
    else cout<<"输入数据非法! ";

    cout<<"7)请输入线性表L中待查元素:";
    cin>>e;
    if((i=LocateItem_sq(L,e)))
        cout<<"元素"<<e<<"在顺序表L的位序是"<<i<<endl;
    else cout<<"元素"<<e<<"不在在顺序表L中"<<endl;

    cout<<"8)输入要插入顺序表L的位置和元素:";
    cin>>i>>e;
    cout<<(ListInsert_sq(L,i,e)?"插入成功!":"插入失败!")<<endl;
    cout<<"插入后输出线性表L:";
    ListTraverse_sq(L);

    cout<<"9)输入要删除顺序表L的元素的位置:";
    cin>>i;
    cout<<(ListDelete_sq(L,i,e)?"删除成功!":"删除失败!")<<endl;
    cout<<"删除后输出线性表L:";
    ListTraverse_sq(L);

    cout<<"10)销毁线性表L."<<endl;
    DestroyList_sq(L);
}
```

示例一的运行结果如下图：

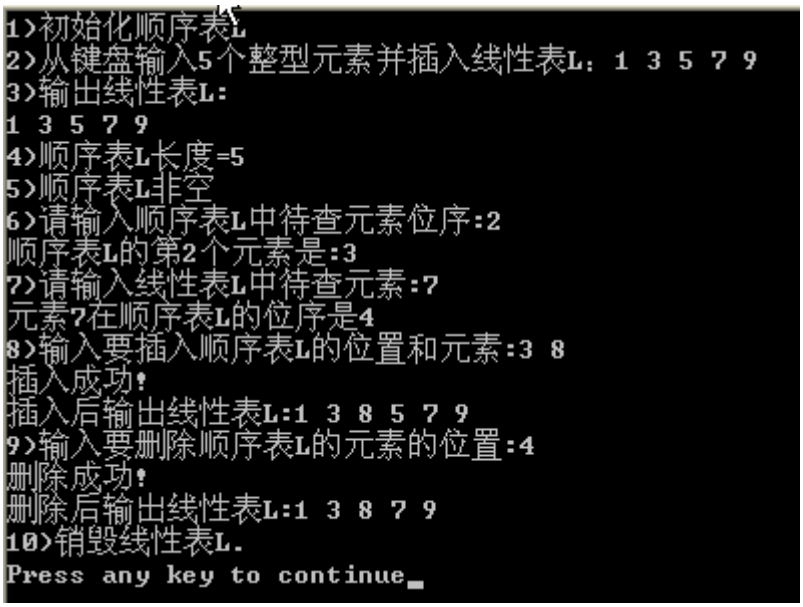

图3.1 示例一运行结果

3.1.4.2 示例二:结构体元素顺序表操作。

下面的示例是针对ElemType为结构体的顺序表的操作演示。要特别注意C++中运算符重载的应用。

1、建立一个头文件sqlist2.h，内容如下：

```
#define LIST_INIT_SIZE 100
#define LIST_INC_SIZE 20
struct student{
char    id[10];
char    name[10];
int     age;
};
typedef student ElemType;   //ElemType 定义为结构体student类型
typedef struct{
    ElemType  * elem;
    int          listsize;
    int          length;
}SqList;
```

```
//初始化线性表
bool InitList_sq(SqList &L,int msize=LIST_INIT_SIZE);
//销毁线性表
void DestroyList_sq(SqList &L);
//清空线性表
void ClearList_sq(SqList &L);
//判断线性表是否为空
bool ListEmpty_sq(SqList L);
//判断线性表是否满
bool ListFull_sq(SqList L);
//求线性表长度
int ListLength_sq(SqList L);
//查找元素
int LocateItem_sq(SqList L,ElemType e);
//获取元素
bool GetItem_sq(SqList L,int i,ElemType &e);
//插入元素
bool ListInsert_sq(SqList &L,int i,ElemType e);
//删除元素
bool ListDelete_sq(SqList &L,int i,ElemType &e);
//遍历元素
void ListTraverse_sq(SqList L);
//扩展线性表
bool Increment(SqList &L,int inc_size=LIST_INC_SIZE);
//比较元素相等
bool operator == (const ElemType r1, const ElemType r2);
//比较元素大小
bool operator < (const ElemType r1, const ElemType r2);
//输出元素
ostream & operator << (ostream & ostr, const ElemType r);
//输出元素
istream & operator >> (istream & istr, ElemType &r);
```

2、建立一个实现文件sqlist2.cpp，内容如下：

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

```
#include <string.h>
#include "sqlist2.h"


//初始化线性表
bool InitList_sq(SqList &L,int msize)
{
    L.elem=new ElemType[msize];
    if(!L.elem){cerr<<"分配内存错误!"<<endl;return false;}
    L.listsize=msize;
    L.length=0;
    return true;
}

//销毁线性表
void DestroyList_sq(SqList &L)
{
    delete [] L.elem;
    L.listsize=0;
    L.length=0;
}

//清空线性表
void ClearList_sq(SqList &L)
{
    L.length=0;
}
//判断线性表是否为空
bool ListEmpty_sq(SqList L)
{
    return (L.length==0);
}
//判断线性表是否满
bool ListFull_sq(SqList L)
{
    return(L.length==L.listsize);
}
//求线性表长度
```

```
int ListLength_sq(SqList L)
{
    return L.length;
}

//查找元素
int LocateItem_sq(SqList L,ElemType e)
{
    int i;
    for(i=0;i<L.length;i++)     //依次查找每个元素
        if(L.elem[i]==e)return i+1; //找到位序为i的元素
    return 0;
}
//获取元素
bool GetItem_sq(SqList L,int i,ElemType &e)
{

    if(i<1||i>L.length+1){printf("i值非法!");return false;}
    e=L.elem[i-1];
    return true;
}
//插入元素
bool ListInsert_sq(SqList &L,int i,ElemType e)
{
    int j;

    if(i<1||i>L.length+1){printf("i值非法!");return false;}
    if(ListFull_sq(L))  //线性表已满
    if(Increment(L))return false;   //扩展失败
    for(j=L.length-1;j>=i-1;j--)
       L.elem[j+1]=L.elem[j];
    L.elem[i-1]=e;
    ++L.length;
    return true;
}
//删除元素
bool ListDelete_sq(SqList &L,int i,ElemType &e)
{
```

```
    int j;
    if(i<1||i>L.length) {printf("i值非法!");return false;}
    if(ListEmpty_sq(L))return false;
    e=L.elem[i-1];
    for(j=i;j<=L.length-1;j++)
    L.elem[j-1]=L.elem[j];
    --L.length;
    return true;
}
//遍历线性表
void ListTraverse_sq(SqList L)
{
    int i;
    for(i=0;i<L.length;i++)
    cout<<L.elem[i]<<endl;
    cout<<endl;
}

bool Increment(SqList &L,int inc_size)
{
    //增加顺序表L的容量为listsize+inc_size
    int i;

    ElemType *a;
    a=new ElemType[L.listsize+inc_size];  // 给a指针动态分配内存
    if(!a)return false;           //分配失败
    for(i=0;i<L.length;i++)a[i]=L.elem[i];
    delete [] L.elem;
    L.elem=a;
    L.listsize+=inc_size;
    return true;
}
//比较元素相等
bool operator == (const ElemType r1, const ElemType r2)
{
    return (strcmp(r1.id,r2.id)==0);
}
//比较元素大小
```

```
bool operator < (const ElemType r1, const ElemType r2)
{
    return strcmp(r1.id,r2.id)<0;
}
//输出元素
ostream & operator << (ostream & ostr, const ElemType r)
{
    ostr.setf(ios::left);
    ostr<<setw(10)<<r.id<<setw(10)<<r.name<<r.age;
    return ostr;
}
//输出元素
istream & operator >> (istream & istr, ElemType &r)
{
    istr>>r.id>>r.name>>r.age;
    return istr;
}
```

3、建立主程序exam1_2.cpp。

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include "sqlist2.h"

void main()
{
    SqList L;
    ElemType e;
    int i;

    cout<<"1)初始化顺序表L"<<endl;
    InitList_sq(L);
    cout<<"2)从键盘输入3个结构体元素并插入线性表L: "<<endl;
    for (i=1;i<=3;i++){
        cin>>e;
        ListInsert_sq(L,i,e);
    }
    cout<<"3)输出线性表L:"<<endl;
```

```
    ListTraverse_sq(L);
    cout<<"4)顺序表L长度="<<ListLength_sq(L)<<endl;
    cout<<"5)顺序表L"<<(ListEmpty_sq(L)?"空":"非空")<<endl;

    cout<<"6)请输入顺序表L中待查元素位序:";
    cin>>i;
    if(GetItem_sq(L,i,e))
        cout<<"顺序表L的第"<<i<<"个元素是:"<<e<<endl;
    else cout<<"输入数据非法! ";

    cout<<"7)请输入线性表L中待查元素:";
    cin>>e;
    if((i=LocateItem_sq(L,e)))
        cout<<"元素"<<e<<"在顺序表L的位序是"<<i<<endl;
    else cout<<"元素"<<e<<"不在在顺序表L中"<<endl;

    cout<<"8)输入要插入顺序表L的位置和元素:";
    cin>>i>>e;
    cout<<(ListInsert_sq(L,i,e)?"插入成功!":"插入失败!")<<endl;
    cout<<"插入后输出线性表L:"<<endl;
    ListTraverse_sq(L);

    cout<<"9)输入要删除顺序表L的元素的位置:";
    cin>>i;
    cout<<(ListDelete_sq(L,i,e)?"删除成功!":"删除失败!")<<endl;
    cout<<"删除后输出线性表L:"<<endl;
    ListTraverse_sq(L);

    cout<<"10)销毁线性表L."<<endl;
    DestroyList_sq(L);
}
```

示例二的运行结果如下图：

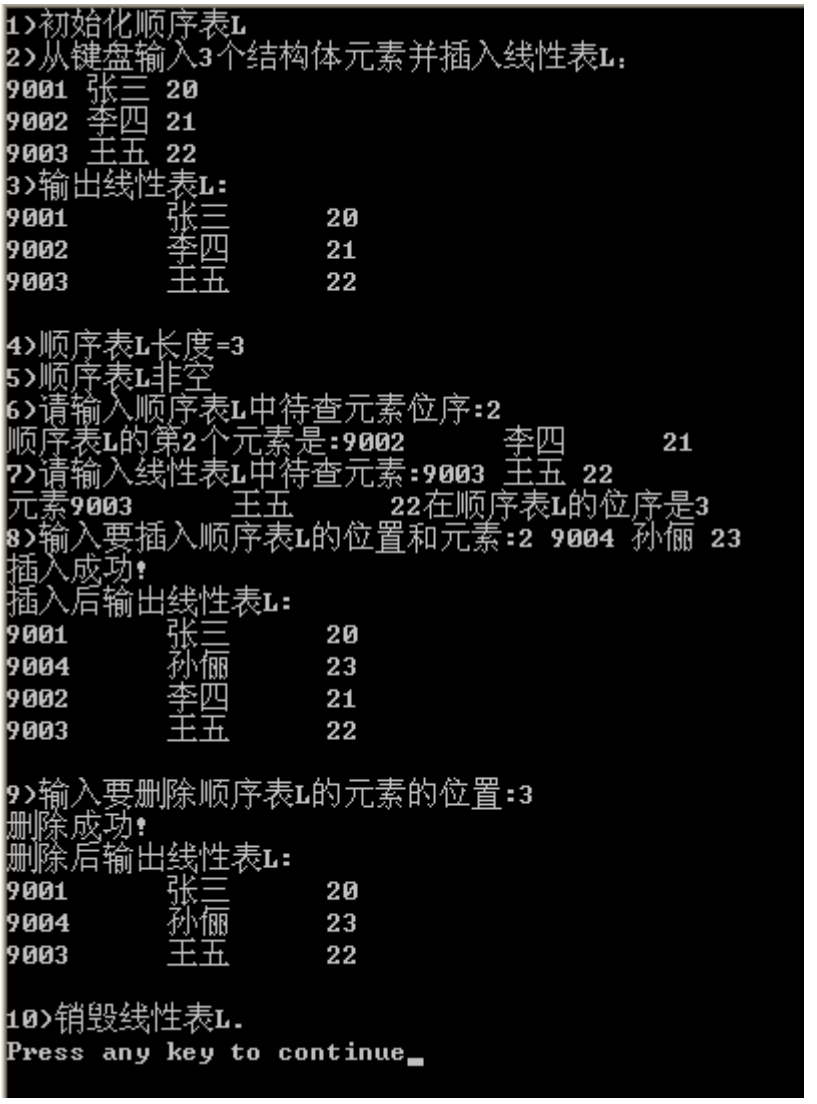

图3.2 示例二运行结果

3.1.4.3 示例三:单链表的实现与操作。

下面的示例给出了不带头结点的单链表的实现与操作。

1、建立一个头文件linklist.h，内容如下：

```
#define LIST_INIT_SIZE 100
#define LIST_INC_SIZE 20
typedef char ElemType;   //ElemType 定义为char类型
```

```
typedef struct LNode{
    ElemType    data;
    struct LNode  *next;
}LNode,*LinkList;
//初始化链表
void InitList_L(LinkList &L);
//销毁链表
void DestroyList_L(LinkList &L);
//判断链表是否为空
bool ListEmpty_L(LinkList L);
//求链表长度
int ListLength_L(LinkList L);
//查找元素
int LocateItem_L (LinkList L,ElemType e);
//获取元素
bool GetItem_L(LinkList L,int i,ElemType &e);
//插入元素
bool ListInsert_L(LinkList &L,int i,ElemType e);
//删除元素
bool ListDelete_L(LinkList &L,int i,ElemType &e);
//遍历元素
void ListTraverse_L(LinkList L);
```

2、建立一个实现文件linklist.cpp，内容如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <iomanip.h>
#include <string.h>
#include "linklist.h"

//初始化链表
void InitList_L(LinkList &L)
{
    L=NULL;
}
//销毁链表
void DestroyList_L(LinkList &L)
{
```

```
    LNode *p;
    while(L){
        p=L;
        L=L->next;
        delete p;
    }
}
//判断链表是否为空
bool ListEmpty_L(LinkList L)
{
    return (L==NULL);
}
//求链表长度
int ListLength_L(LinkList L)
{
    LNode * p=L;
    int k=0;
    while(p){p=p->next;k++;}
    return k;
}
//查找元素
int LocateItem_L(LinkList L,ElemType e)
{
    int j=1;
    LNode * p=L;
    while(p&&p->data!=e){p=p->next;j++;}
    if(p)return j;
    else return 0;
}
//获取元素
bool GetItem_L(LinkList L,int i,ElemType &e)
{
    int j=1;
    LNode * p=L;

    if(i<1){printf("i值非法!");return false;}
    while(j<i&&p){
        j++;
```

```
        p=p->next;
    }
    if(p)
        {e=p->data;return true;}
    else
        return false;
}
//插入元素
bool ListInsert_L(LinkList &L,int i,ElemType e)
{
    int    j=1;
    LNode  *p=L,*s;

    if(i<1){printf("i值非法!");return false;}
    while(j<i-1&&p){p=p->next;j++;}       //p指向i-1结点
    if(!p&&i!=1){
        cout<<"未找到i-1结点!"<<endl;
        return false;
    }

    s=new LNode;
    s->data=e;
    if(i==1){
        s->next=L;
        L=s;
    }
    else{
        s->next=p->next;
        p->next=s;
    }
    return true;
}
//删除元素
bool ListDelete_L(LinkList &L,int i,ElemType &e)
{
    int j=1;
    LNode *p=L,*q;
```

```
    if(i<1) {printf("i值非法!");return false;}
    while(j<i-1&&p){p=p->next;j++;}       //p指向i-1结点
    if(!p||!(p->next)){
        cout<<"未找到i-1/i结点!"<<endl;
        return false;
    }
    if(i==1){           //删除第一个结点
        q=L;
        L=L->next;
        e=q->data;
    }
    else{
        q=p->next;
        p->next=q->next;
        e=q->data;
    }
    delete q;
    return true;
}
//遍历链表
void ListTraverse_L(LinkList L)
{
    LNode *p=L;
    while(p){
        cout<<p->data<<' ';
        p=p->next;
    }
    cout<<endl;
}
```

3、建立主程序exam1_3.cpp。

```
#include <stdio.h>
#include <stdlib.h>
#include <iomanip.h>
#include "linklist.h"

void main()
{
```

```
    LinkList L;
    ElemType e;
    int i;

    cout<<"1)初始化单链表L"<<endl;
    InitList_L(L);
    cout<<"2)顺序插入链表L元素a、b、c、d、e."<<endl;
    for (i=1;i<=5;i++)
       ListInsert_L(L,i,'a'+i-1);
    cout<<"3)输出单链表L:"<<endl;
    ListTraverse_L(L);
    cout<<"4)单链表L长度="<<ListLength_L(L)<<endl;
    cout<<"5)单链表L"<<(ListEmpty_L(L)?"空":"非空")<<endl;
    if(GetItem_L(L,3,e))
    cout<<"6)单链表L中第3个元素是:"<<e<<endl;
    cout<<"7)元素'b'在单链表中的位序
是:"<<LocateItem_L(L,'b')<<endl;
    cout<<"8)在单链表L中第4位置插入元素'f'."<<endl;
    cout<<(ListInsert_L(L,4,'f')?"插入成功!":"插入失败!")<<endl;
    cout<<"插入后输出单链表表L:"<<endl;
    ListTraverse_L(L);
    cout<<"9)删除单链表L中第3个元素.";
    cout<<(ListDelete_L(L,3,e)?"删除成功!":"删除失败!")<<endl;
    cout<<"删除后输出单链表L:"<<endl;
    ListTraverse_L(L);
    cout<<"10)销毁单链表L."<<endl;
    DestroyList_L(L);
}
```

示例三的运行结果如下图：

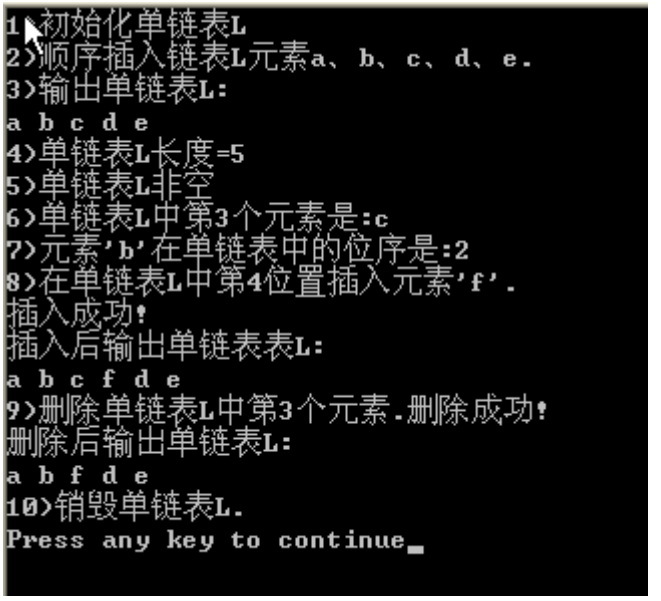

图3.3 示例三运行结果

3.1.4.4 动手与实践

- 修改示例三中主程序exam3.cpp，使之能够接受交互输入。
- 修改示例三中ElemType类型为结构体类型，重新实现linklist.cpp。
- 约瑟夫问题求解。

1）**内容：**

约瑟夫(Joseph)问题的一种描述是:编号为1,2,..., n 的n 个人按顺时针方向围坐一圈， 每人持有一个密码(正整数)。一开始选任一个正整数作为报数上限值m， 从第一个人开始按顺时针方向自1开始顺序报数，报到m时停止报数。报m的人出列，将它的密码作为新的m值，再从下个人开始新一轮报数，如此反复，直到剩下最后一人则为获胜者。试设计一个程序求出出列顺序。

2）**要求：**

利用单向循环链表存储结构模拟此过程， 按照出列的顺序印出各人的编号。

3） **测试数据：**

n=7，7 个人的密码依次为：3，1，7，2，4，8，4 。m的初值为20，则正确的出列顺序应为6，1，4，7，2，3，5。

4）**输入输出：**

输入数据：建立输入处理输入数据，输入n输入以及每个人的密码；m的初值。

输出形式：建立一个输出函数，输出正确的序列。

3.2 实验二：栈与队列

3.2.1 背景知识

栈和队列可以看做是一种特殊的线性表，他们都属于操作受限的线性表。从逻辑上来看，栈是限定仅在一端进行插入和删除操作的线性表，允许插入和删除的一端称为栈顶（top），另一端称为栈底（bottom）；队列则是限定在一端插入另一端删除的线性表，删除的一端称队首(front),插入的一端称为队尾（rear）。

栈的插入和删除是按后进先出的原则（LIFO，last in first out），队列的插入和删除是按先进先出的原则（FIFO，first in first out）。

栈的抽象数据类型定义如下：

```
ADT Stack{
    数据对象：D={ai|ai∈ElemSet,i=1,2,...,n, n≥0,ElemSet为元素集合}
    数据关系：R={<ai-1,ai>|ai-1,ai ∈D,i=1,2,...,n}
    基本操作：
    InitStack(&S);   //构造空栈
    DestroyStack(&S);   //销毁栈
    ClearStack(&S);     //将栈置空
    StackEmpty(S);      //检查栈是否为空
    StackLength(S);     //返回栈中元素个数
    GetTop(S,&e);   //返回栈顶元素赋予e
    Push(S,e);   //插入e为新的栈顶元素
    Pop(S,&e); //删除栈顶元素并赋予e
```

```
    StackTravers(S);          //依次输出栈中元素。
}
```

队列的抽象数据类型定义如下：

```
ADT Queue{
    数据对象：D={ai|ai∈ElemSet,i=1,2,...,n, n≥0,ElemSet为元素集合}
    数据关系：R={<ai-1,ai>|ai-1,ai ∈D,i=1,2,...,n}
    基本操作：
    InitQueue(&Q);   //构造空队列
    DestroyQueue(&Q);   //销毁队列
    ClearQueue(&Q);     //将队列置空
    QueueEmpty(Q);      //检查队列是否为空
    QueueLength(Q);     //返回队列中元素个数
    GetHead(Q,&e);   //返回队首元素赋予e
    EnQueue(Q,e);   //在队尾插入新元素e
    DeQueue(Q,&e）; //删除队首元素并赋予e
    QueueTravers(Q);   //依次输出队列中元素。
}
```

3.2.2 实验目的

- ✧ 熟悉栈和队列的定义及其顺序和链式存储结构。
- ✧ 掌握栈的LIFO和队列的FIFO特点。
- ✧ 熟悉对栈和队列的一些基本操作和及其具体实现。
- ✧ 通过应用示例，进一步熟悉和掌握栈和队列的实践应用。

3.2.3 实验要求

- ✧ 认真阅读本实验内容中给出的示例程序。
- ✧ 在计算机上输入示例程序中关于栈和队列的实现程序。
- ✧ 调试和运行示例程序。
- ✧ 编写程序实现实验中的“动手与实践”部分要求。

3.2.4 实验内容

3.2.4.1 示例一:顺序栈的操作。

1、建立一个头文件sqstack.h，内容如下：

```
#define STACK_INIT_SIZE 100
typedef char SElemType;   //ElemType 定义为char类型
typedef struct{
    SElemType  * elem;
    int          stacksize;
    int          top;
}SqStack;
//初始化栈
bool InitStack_sq(SqStack &S,int msize=STACK_INIT_SIZE);
//销毁栈
void DestroyStack_sq(SqStack &S);
//清空栈
void ClearStack_sq(SqStack &S);
//判断栈是否为空
bool StackEmpty_sq(SqStack S);
//判断栈是否满
bool StackFull_sq(SqStack S);
//求栈长度
int StackLength_sq(SqStack S);
//获取栈顶元素
bool GetTop_sq(SqStack S,SElemType &e);
//元素入栈
bool Push_sq(SqStack &S, SElemType e);
//元素出栈
bool Pop_sq(SqStack &S,SElemType &e);
//遍历元素
void StackTraverse_sq(SqStack S);
```

2、建立一个实现文件sqstack.cpp，内容如下：

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

```
#include <stdlib.h>
#include <iostream.h>
#include "sqstack.h"

//初始化栈
bool InitStack_sq(SqStack &S,int msize)
{
    S.elem=new SElemType[msize];
           // 给elem指针动态分配msize长度的数组
    if(!S.elem){cerr<<"分配内存错误!"<<endl;return false;}
    S.stacksize=msize;        //顺序栈的最大容量
    S.top=-1;                // 顺序栈初始时空栈
    return true;
}
//销毁栈
void DestroyStack_sq(SqStack &S)
{
    delete [] S.elem;
    S.stacksize=0;
    S.top=-1;
}

//清空栈
void ClearStack_sq(SqStack &S)
{
    S.top=-1;
}
//判断栈是否为空
bool StackEmpty_sq(SqStack S)
{
    return (S.top==-1);
}
//判断栈是否满
bool StackFull_sq(SqStack S)
{
    return(S.top==S.stacksize-1);
}
//求栈长度
```

```
int StackLength_sq(SqStack S)
{
    return S.top+1;
}
//获取栈顶元素
bool GetTop_sq(SqStack S,SElemType &e)
{
    if(S.top==-1)return false;
    e=S.elem[S.top];
    return true;
}
//元素入栈
bool Push_sq(SqStack &S, SElemType e)
{
    if(StackFull_sq(S))return false;
    S.elem[++S.top]=e;
    return true;
}
//元素出栈
bool Pop_sq(SqStack &S,SElemType &e)
{
    if(StackEmpty_sq(S))return false;
    e=S.elem[S.top--];
    return true;
}
//遍历元素
void StackTraverse_sq(SqStack S)
{
    int i;
    for(i=0;i<=S.top;i++)
        cout<<S.elem[i]<<' ';
    cout<<endl;
}
```

3、建立主程序exam2_1.cpp。

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

```
#include <iomanip.h>
#include "sqstack.h"

void main()
{
    SqStack S;
    SElemType e;
    int i;

    cout<<"1)初始化栈S"<<endl;
    InitStack_sq(S);
    cout<<"2)依次进栈元素a、b、c、d、e."<<endl;
    for (i=0;i<5;i++)
        Push_sq(S,'a'+i);
    cout<<"3)遍历输出栈S:"<<endl;
    StackTraverse_sq(S);
    cout<<"4)栈S长度="<<StackLength_sq(S)<<endl;
    cout<<"5)栈S"<<(StackEmpty_sq(S)?"空":"非空")<<endl;
    if(GetTop_sq(S,e))
        cout<<"6)栈顶元素是:"<<e<<endl;
    cout<<"7)入栈元素'f'."<<endl;
    cout<<(Push_sq(S,'f')?"入栈成功!":"入栈失败!")<<endl;
    cout<<"入栈后输出栈全部元素:"<<endl;
    StackTraverse_sq(S);
    cout<<"8)出栈一个元素.";
    cout<<(Pop_sq(S,e)?"出栈成功!":"出栈失败!")<<endl;
    cout<<"出栈后输出栈:"<<endl;
    StackTraverse_sq(S);

    cout<<"9)元素依次出栈:";
    while(!StackEmpty_sq(S)){Pop_sq(S,e);cout<<e<<' ';}
    cout<<endl;
    cout<<"栈S"<<(StackEmpty_sq(S)?"空":"非空")<<endl;

    cout<<"10)销毁栈S."<<endl;
    DestroyStack_sq(S);
}
```

示例一的运行结果如下图：

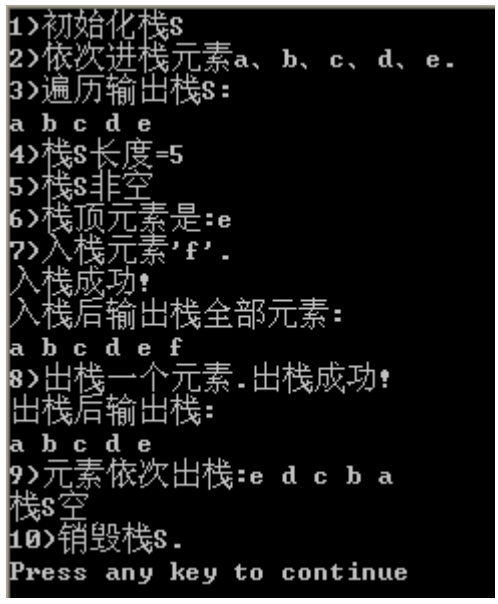

图3.4 示例一运行结果

3.2.4.2 示例二:链队列的操作。

因链队列用到了有关链表的方法，因此本示例的工程需要包含有linklist.h。

1、建立一个头文件linkqueue.h，内容如下：

```
#define QUEUE_INIT_SIZE 100
#include "linklist.h"

typedef char QElemType;   //ElemType 定义为char类型
typedef LinkList Queueptr;  //结点指针
typedef struct{
    Queueptr  front;
    Queueptr  rear;
}LinkQueue;              //链队列定义

//初始化队列
bool InitQueue_L(LinkQueue &Q);
//销毁队列
void DestroyQueue_L(LinkQueue &Q);
```

```
//清空队列
void ClearQueue_L(LinkQueue &Q);
//判断队列是否为空
bool QueueEmpty_L(LinkQueue Q);
//求队列长度
int QueueLength_L(LinkQueue Q);
//获取队首元素
bool GetHead_L(LinkQueue Q,QElemType &e);
//元素入队列
bool EnQueue_L(LinkQueue &Q, QElemType e);
//元素出队列
bool DeQueue_L(LinkQueue &Q,QElemType &e);
//遍历元素
void QueueTraverse_L(LinkQueue Q);
```

2、建立一个实现文件linkqueue.cpp，内容如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include "linkqueue.h"

//初始队列
bool InitQueue_L(LinkQueue &Q)
{
    Q.front=Q.rear=new LNode;
    // 给elem指针动态分配msize长度的数组
    if(!Q.front){cerr<<"分配内存错误!"<<endl;return false;}
    return true;
}
//销毁队列
void DestroyQueue_L(LinkQueue &Q)
{
    while(Q.front){
        Q.rear=Q.front->next;
        delete Q.front;
        Q.front=Q.rear;
    }
```

```
}

//清空队列
void ClearQueue_L(LinkQueue &Q)
{
    QElemType e;
    while(Q.front!=Q.rear)DeQueue_L(Q,e);
}
//判断队列是否为空
bool QueueEmpty_L(LinkQueue Q)
{
    return (Q.front==Q.rear);
}
//求队列长度
int QueueLength_L(LinkQueue Q)
{
    int i=0;
    Queueptr p=Q.front;

    while(p!=Q.rear){i++;p=p->next;}
    return i;
}
//获取队首元素
bool GetHead_L(LinkQueue Q,QElemType &e)
{
    if(Q.front==Q.rear)return false;
    e=Q.front->next->data;
    return true;
}
//元素入栈
bool EnQueue_L(LinkQueue &Q, QElemType e)
{
    Queueptr p=new LNode;
    if(!p)return false;
    p->data=e;p->next=NULL;
    Q.rear->next=p;
    Q.rear=p;
    return true;
```

```
}
//元素出队列
bool DeQueue_L(LinkQueue &Q,QElemType &e)
{
    if(Q.front==Q.rear)return false;
    Queueptr p=Q.front->next;
    Q.front->next=p->next;
    e=p->data;
    if(Q.rear==p)Q.rear=Q.front;
    delete p;
    return true;
}
//遍历元素
void QueueTraverse_L(LinkQueue Q)
{
    Queueptr p=Q.front->next;
    while(p){cout<<p->data<<' ';p=p->next;}
    cout<<endl;
}
```

3、建立主程序exam2_2.cpp。

```
#include <stdio.h>
#include <stdlib.h>
#include <iomanip.h>
#include "LinkQueue.h"

void main()
{
    LinkQueue Q;
    QElemType e;
    int i;

    cout<<"1)初始化队列Q"<<endl;
    InitQueue_L(Q);
    cout<<"2)依次入队列元素a、b、c、d、e."<<endl;
    for (i=0;i<5;i++)
        EnQueue_L(Q,'a'+i);
```

```
    cout<<"3)输出队列Q:"<<endl;
    QueueTraverse_L(Q);
    cout<<"4)队列Q长度="<<QueueLength_L(Q)<<endl;
    cout<<"5)队列Q"<<(QueueEmpty_L(Q)?"空":"非空")<<endl;
    if(GetHead_L(Q,e))
        cout<<"6)队首元素是:"<<e<<endl;
    cout<<"7)入队列元素'f'."<<endl;
    cout<<(EnQueue_L(Q,'f')?"入队列成功!":"入队列失败!")<<endl;
    cout<<"入队列后输出队列全部元素:"<<endl;
    QueueTraverse_L(Q);
    cout<<"8)出队列一个元素.";
    cout<<(DeQueue_L(Q,e)?"出队列成功!":"出队列失败!")<<endl;
    cout<<"出队列后输出全部元素:"<<endl;
    QueueTraverse_L(Q);

    cout<<"9)元素依次出队列:";
    while(!QueueEmpty_L(Q)){DeQueue_L(Q,e);cout<<e<<' ';}
    cout<<endl;
    cout<<"队列Q"<<(QueueEmpty_L(Q)?"空":"非空")<<endl;

    cout<<"10)销毁队列Q."<<endl;
    DestroyQueue_L(Q);
}
```

示例二的运行结果如下图：

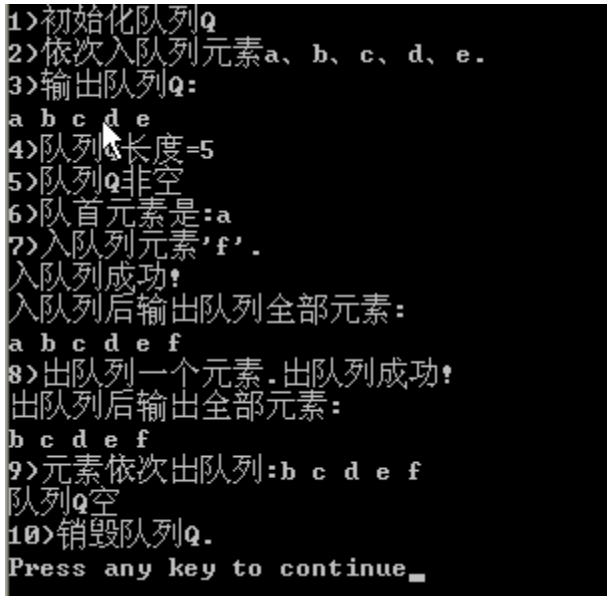

图3.5 示例二运行结果

3.2.4.3 动手与实践

- 编写链栈的实现文件linkstack.cpp，并编写验证主程序。
- 编写顺序队列的实现文件sqqueue.cpp，并编写验证主程序。
- 停车场问题。

1）**内容：**

设停车场是一个可停放 n 辆汽车的狭长通道，且只有一个大门可供汽车进出。汽车在停车场内按车辆到达时间的先后顺序，依次由北向南排列（大门在最南端，最先到达的在最北端），若停车场内已经停满 n辆车，那么后来的车只能在场外等候，一旦有车开走，则等候在第一位的车即可开入（这是一个队列设长度为m）；当停车场内某辆车需要开出，则在它之后的车辆必须给它让道，当这辆车驶出停车场后，其他车辆按序入栈。每辆车按时间收费。

2）**要求：**

以栈模拟停车场，以队列模拟车场外的便道，按照从终端读入数据的序列进行模拟管理。每一组输入数据包括三个数据：汽车的“到达”（'A'表示）或“离去”（'D'表示）信息，汽车标识（牌照号）以及到达或离去的时刻。对每一组输入数据进行操作后的输出信息为：若是车辆到达，则输出汽车在停车场内或者便道上的停车位置；若是车辆离去，则输出汽车在停车场停留的时间和应缴纳的费用（便道上不收费）。栈以顺序结构实现，队列以链表结构实现。

3）**测试数据：**

设 n＝3，m=4,停车价格为p=2。输入数据为：

('A',101,5),('A',102,10),('D',101,15),('A',103,20),('A',104,25),('A',105,30),('D',102,35),('D',104,40),('E',0,0)。其中'A' 表示到达，'D'表示离开，'E'表示结束。时间为相对分钟数。

4）**输入输出：**

输入数据：程序接受5个命令，分别是：到达（'A'，车牌号，时间）；离去（'D'，车牌号，时间）；停车场（P，0，0）显示停车场的车；候车场（W，0，0）显示候车场的车；退出（E，0，0）退出程序。

输出数据：对于车辆到达，要输出汽车在停车场内或者便道上的停车位置；对于车辆离去，则输出汽车在停车场停留的时间和应缴纳的费用（便道上不收费）。

3.3 实验三：串与数组

3.3.1 背景知识

字符串是计算机中很重要的一种数据对象。随着语言处理的发展，也就产生了字符串处理。这样字符串作为一种数据类型在越来越多的程序设计语言中出现。C语言中也有专门的字符串操作函数。

字符串的抽象数据类型定义如下：

```
ADT String{
    数据对象：D={ai|ai∈CharSet,i=1,2,...,n, n≥0 }
    数据关系：R={<ai-1,ai>|ai-1,ai ∈D,i=1,2,...,n}
    基本操作：
    StrAssign (&T,chars);   //生成一个值为chars的串T
    StrCopy(&T,S);   //串S复制到T
    StrEmpty(S);      //检查字符串是否为空串
    StrLength(S);     //返回字符串长度
    StrCompare(S,T);   //比较字符串大小
    StrConcat(&T,S1,S2);   //S1、S2连接成新串T
    SubString(&Sub,S,pos,len); //从pos位置开始取S的长度为len子串
    Index(S,T,pos);          //在S串中查找子串T的位置
    Replace(&S,T,V);   //把S中T子串替换为V
    StrInsert(&S,pos,T) //把T插入S中pos位置
    StrDelete(&S,pos,len)  //删除S中pos开始的长度为len的子串
    DestroyString(&S)       //销毁串S
```

```
}
```

3.3.2 实验目的

- ✧ 通过对串的特点分析，掌握串的主要存储结构。
- ✧ 在串具体的存储结构基础上，实现串的基本操作。
- ✧ 通过应用示例，进一步熟悉和掌握字符串的应用。

3.3.3 实验要求

- ✧ 认真阅读本实验内容中给出的示例程序。
- ✧ 在计算机上输入示例程序中字符串操作的实现程序。
- ✧ 调试和运行示例程序。
- ✧ 编写程序实现实验中的“动手与实践”部分要求。

3.3.4 实验内容

3.3.4.1 示例一:顺序串的各种操作。

C语言中的字符串是以'\0'结束的字符数组来表示。这里我们使用另一种静态存储方法，即首字符存储字符串的长度，我们定义为SString。在本示例中我们使用该存储结构来实现字符串的各种操作。

1、建立一个头文件sstring.h，内容如下：

```
#define MAX_STR_LEN 254
typedef char SString[MAX_STR_LEN +1];
void StrAssign(SString &T,char *cstr);
void StrCopy(SString &T,SString S);
bool StrEmpty (SString S);
int StrLength(SString S);
int StrCompare(SString S,SString T);  //比较字符串大小
```

```
bool StrConcat(SString &T, SString S1, SString S2);
bool SubString(SString &Sub, SString S,int pos,int len);
int Index(SString S, SString T,int pos);  //在S串中查找子串T的位置
bool Replace(SString &S,SString T,SString V);//把S中T子串替换为V
bool StrInsert(SString &S,int pos, SString T);
bool StrDelete(SString &S,int pos,int len);
void StrPrint(SString S);       //输出串S
```

2、建立一个实现文件sstring.cpp，内容如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include "sstring.h"


void StrAssign(SString &T,char *cstr)
{
    int i;
    for(i=0;cstr[i]!='\0';i++)T[i+1]=cstr[i];
    T[0]=i;
}
void StrCopy(SString &T,SString S)
{
    int i;
    for(i=0;i<=S[0];i++)T[i]=S[i];
}
bool StrEmpty (SString S)
{
    return (S[0]==0);
}
int StrLength(SString S)
{
    return S[0];
}
int StrCompare(SString S,SString T)   //比较字符串大小
{
    int i;
```

```
    for(i=1;i<=S[0]&&i<=T[0];i++)
        if(S[i]!=T[i])return (S[i]-T[i]); //<0表示S小；>0表示S大
    return (S[0]-T[0]);               //=0表示S=T
}
bool StrConcat(SString &T, SString S1, SString S2)
{
    int i;

    if(S1[0]+S2[0]>MAX_STR_LEN)return false;
    for(i=1;i<=S1[0];i++)T[i]=S1[i];
    for(i=1;i<=S2[0];i++)T[i+S1[0]]=S2[i];
    T[0]=S1[0]+S2[0];
    return true;
}

bool SubString(SString &Sub, SString S,int pos,int len)
{
    int i;
    if(pos<1||pos>S[0]||len<0||len>S[0]-pos+1)return false;
    for(i=1;i<=len;i++)
        Sub[i]=S[pos+i-1];
    Sub[0]=len;
    return true;
}

int Index(SString S, SString T,int pos)  //在S串中查找子串T的位置
{
    int i,j;

    if(pos<1||pos>S[0])return 0;
    i=pos;
    j=1;
    while(i<=S[0]&&j<=T[0])
        if(S[i]==T[j])
            {i++;j++;}
        else
            {i=i-j+2;j=1;}   //指针回溯
    if(j>T[0])
```

```
        return i-T[0];   //定位成功
    else
        return 0;     //子串不存在
}
bool Replace(SString &S,SString T,SString V)//把S中T子串替换为V
{
    int i=1;

    if(StrEmpty(T))return false;
    do{
        i=Index(S,T,i);
        if(i){
            StrDelete(S,i,StrLength(T));
            if(!StrInsert(S,i,V))return false;
        }
    }while(i);
    return true;
}

bool StrInsert(SString &S,int pos, SString T)
{
    int i,tlen;

    if(pos<1||pos>S[0]+1) return false;       //参数pos非法
    if(S[0]+T[0]>MAX_STR_LEN)return false;      // 溢出
    tlen=StrLength(T);
    for(i=S[0];i>=pos;i--)
        S[i+tlen]=S[i];
    for(i=pos;i<pos+T[0];i++)
        S[i]=T[i-pos+1];
    S[0]+=T[0];
    return true;
}
bool StrDelete(SString &S,int pos,int len)
{
    int i;
    if(pos<1||pos>S[0]-len+1) return false;       //参数pos非法
    for(i=pos+len;i<=S[0];i++)
```

```
        S[i-len]=S[i];
    S[0]-=len;
    return true;
}

void StrPrint(SString S)        //输出串S
{
    int i;
    for(i=1;i<=S[0];i++)
        cout<<S[i];
    cout<<endl;
}
```

3、建立主程序exam3_1.cpp。

```
#include <stdio.h>
#include <stdlib.h>
#include <iomanip.h>
#include "sstring.h"

void main()
{
    SString s,s1,s2,s3;

    cout<<"1)建立串s和s1"<<endl;
    StrAssign(s,"abcdefghigklmnopq");
    StrAssign(s1,"xyz");
    cout<<"2)输出串s=";StrPrint(s);
    cout<<"输出串s1=";StrPrint(s1);
    cout<<"3)串s的长度="<<StrLength(s)<<endl;
    cout<<"4)串s"<<(StrEmpty(s)?"空":"非空")<<endl;
    if(StrConcat(s2,s,s1)){
        cout<<"5)串s和s1的串接是";StrPrint(s2);
    }
    SubString(s3,s,5,4);
    cout<<"6)串s的第5个字符开始的4个字符的子串是";StrPrint(s3);
    cout<<"7)串efgh在s中的位置为:"<<Index(s,s3,1)<<endl;
    cout<<"8)将串s中子串efgh替换为xyz"<<endl;
```

```
    Replace(s,s3,s1);
    cout<<"输出替换后的s:";StrPrint(s);

    cout<<"9)在串s中第15位置插入efgh"<<endl;
    StrInsert(s,15,s3);
    cout<<"输出插入后的s:";StrPrint(s);

    StrDelete(s,10,5);
    cout<<"10)删除s中第10位字符开始的5个字符，输出为:";StrPrint(s);
}
```

示例一的运行结果如下图：

图3.6 示例一运行结果

3.3.4.2 动手与实践

- 编写C语言存储结构的字符串操作实现文件，并编写验证主程序。
- 关键词检索。

1）**内容：**

实现类似Unix下grep命令的程序。在一个文件中查找某个关键词，并把出现该关键词的行及行号显示出来。

2）**要求：**

使用C语言的字符串存储结构来实现字符串的操作，编写函数index实现在一个传中查找子串的功能。然后从文件中每次读入一行，作为一个主串看待，然后查找是否存在待查找的关键词（子串），如果有则显示该行内容及行号，否则继续处理下一行。

3） **测试数据：**

任意一个文本文件，文件中任意一词语作为关键词。

4）**输入输出：**

输入数据：屏幕输入或命令行给出文本文件名、关键词。

输出数据：屏幕输出文本文件中出现关键词的行及行号。

3.4 实验四：树和二叉树

3.4.1 背景知识

树形结构式一种很重要的非线性数据结构。而二叉树又是树中特殊的一类，易于在计算机中存储和处理，因此针对二叉树的处理较为常用。直观的来看，树是以分支关系定义的层次结构。树实现了对象之间一对多的联系。本实验主要学习二叉树。

二叉树的抽象数据类型定义如下：

```
ADT BinaryTree{
    数据对象：D={ai | ai∈ElemSet,i=1,2,...,n, n≥0 }
    数据关系：R={
              若D=Φ，则二叉树为空数；否则满足
              1)D中存在唯一的根元素root，它没有前驱；
              2)若D-{root}=ϕ，则R=ϕ；否则存在D-{root}的一个划分Dl和Dr,
                且Dr∩Dl=ϕ;
              3)若Dl≠ϕ，则Dl中存在唯一元素xl,<root, xl>∈R,且存在Dl上的
                关系Rl⊂R；同样若Dr≠ϕ，则Dr中存在唯一元素xr,<root, xr>∈R,
                且存在Dr上的关系Rr⊂R;R={<root, xl>,<root, xr>,Rl,Rr}
              4)(Dl, Rl)和(Dr, Rr)也是一颗符合定义的二叉树。
    基本操作：
    InitBiTree(&T)          //初始化空二叉树
```

```
    DestroyBiTree(&T)      //销毁二叉树
    CreateBiTree(&T)     //创建二叉树
    ClearTree(&T)        //清空二叉树
    TreeEmpty(T)        //判断是否空树
    TreeDepth(T)      //返回二叉树深度
    Root(T)           //返回二叉树树根
    Parent(T,x)         //返回T中x的双亲
    LeftChild(T,x)     //返回T中x的左孩子
    RightChild(T,x)     //返回T中x的右孩子
    LeftSibling(T,x)   //返回T中x的左兄弟
    RightSibling(T,x)   //返回T中x的右兄弟
    InsertChild(&T,x,LR,c) //插入二叉树子树
    DeleteChild(&T,x,LR)    //删除二叉树子树
    TraverseTree(T,visit())        //变量二叉树
}
```

3.4.2 实验目的

- ✧ 通过对二叉树特点的分析，掌握二叉树的主要存储结构。
- ✧ 在二叉树具体的存储结构基础上，实现二叉树的基本操作。
- ✧ 能针对二叉树的具体应用选择相应的存储结构。
- ✧ 通过应用示例，进一步掌握递归算法的设计方法。

3.4.3 实验要求

- ✧ 认真阅读本实验内容中给出的示例程序。
- ✧ 在计算机上输入示例程序中关于二叉树的实现程序。
- ✧ 调试和运行示例程序。
- ✧ 编写程序实现实验中的“动手与实践”部分要求。

3.4.4 实验内容

3.4.4.1 示例一:二叉树的各种操作。

在本实验中，我们采用二叉链表的存储方式来存储二叉树，二叉树的每个结点是个结构，定义为BiTNode。因二叉树的层次遍历以及寻找双亲结点等操作需要用到队列，因此本示例的工程文件中需要包含linkqueue.h以及其所依赖的linklist.h。

在本例中需要进入队列的是树的结点指针，因此需要对前面linklist.h和linkqueue做适当修改。在linklist.h中需要将链表的结点元素类型定义为BiTree类型，当然需要包含BiTree的定义头文件bitree.h，改变如下：

```
typedef char ElemType;
```

修改为：

```
#include "bitree.h"
typedef BiTree ElemType;
```

同样在linkqueue.h中需要修改队列的结点类型：

```
typedef char QElemType;
```

修改为：

```
typedef BiTree QElemType;
```

此外本示例工程还需要包含linkqueue.cpp,以便可以直接使用链队列的实现。

1、建立一个头文件bitree.h，内容如下：

```
typedef char TElemType;

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

void InitBiTree(BiTree &T);          //初始化空二叉树
void DestroyBiTree(BiTree &T);      //销毁二叉树
void CreateBiTree(BiTree &T) ;     //创建二叉树
```

```
bool BiTreeEmpty(BiTree T);       //判断是否空树
int  BiTreeDepth(BiTree T);     //返回二叉树深度
BiTNode * Value(BiTree T,TElemType e);  //返回T树中值为e的结点指针
BiTNode * Parent(BiTree T,BiTNode * x);        //返回T中x的双亲
BiTNode * LeftSibling(BiTree T,BiTNode *x);  //返回T中x的左兄弟
BiTNode * RightSibling(BiTree T,BiTNode * x);  //返回T中x的右兄弟
void TraverseTree(BiTree T, int mark);        //遍历二叉树
```

2、建立一个实现文件bitree.cpp，内容如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
//#include <strstrea.h>
#include "linkqueue.h"

void InitBiTree(BiTree &T)         //初始化空二叉树
{
    T=NULL;
}
void DestroyBiTree(BiTree &T)     //销毁二叉树
{
    if(!T) return;
    if(T->lchild)
        DestroyBiTree(T->lchild);
    if(T->rchild)
        DestroyBiTree(T->rchild);
    delete T;
    T=NULL;
}
void CreateBiTree(BiTree &T)    //创建二叉树
{
    //输入扩展二叉树先序序列，创建二叉树
    TElemType  e;
    cin>>e;
    if(e=='#')T=NULL;
    else{
        T=new BiTNode;
```

```
        T->data=e;
        CreateBiTree(T->lchild);
        CreateBiTree(T->rchild);
    }
}
bool BiTreeEmpty(BiTree T)       //判断是否空树
{
    return (T==NULL);
}
int  BiTreeDepth(BiTree T)     //返回二叉树深度
{
    int hr,hl;
    if(!T) return 0;

    hl=BiTreeDepth(T->lchild);
    hr=BiTreeDepth(T->rchild);
    return (hl>hr?hl+1:hr+1);
}
BiTNode * Value(BiTree T,TElemType e) //返回T树中值为e的结点指针
{
    LinkQueue Q;
    QElemType  a;
    if(!T) return NULL;
    InitQueue_L(Q);
    EnQueue_L(Q,T);
    while(!QueueEmpty_L(Q))
    {
        DeQueue_L(Q,a);
        if(a->data==e)return a;
        if(a->lchild)EnQueue_L(Q,a->lchild);
        if(a->rchild)EnQueue_L(Q,a->rchild);

    }
    return NULL;
}
BiTNode * Parent(BiTree T,BiTNode * x)  //返回T中x的双亲
{
    LinkQueue Q;
```

```
    QElemType  a,b=NULL;

    if(!T) return NULL;
    InitQueue_L(Q);
    EnQueue_L(Q,T);
    while(!QueueEmpty_L(Q))
    {
        DeQueue_L(Q,a);
        if(a->lchild && a->lchild==x||a->rchild && a->rchild==x){
            b=a;break;
        }
        else{
            if(a->lchild)EnQueue_L(Q,a->lchild);
            if(a->rchild)EnQueue_L(Q,a->rchild);
        }
    }
    DestroyQueue_L(Q);
    return b;
}
BiTNode * LeftSibling(BiTree T,BiTNode *x)  //返回T中x的左兄弟
{
    BiTNode *p;
    p=Parent(T,x);
    if(p)return p->lchild;
    else return NULL;
}
BiTNode * RightSibling(BiTree T,BiTNode * x) //返回T中x的右兄弟
{
    BiTNode *p;
    p=Parent(T,x);
    if(p)return p->rchild;
    else return NULL;
}
void TraverseTree(BiTree T, int mark)       //遍历二叉树
{
    //mark=1、2、3、4 分别表示先序、中序、后序和层次遍历
    LinkQueue Q;
    QElemType  a;
```

```
    if(mark==1){
        if(!T)return;
        cout<<T->data<<' ';
        TraverseTree(T->lchild,mark);
        TraverseTree(T->rchild,mark);
    }
    if(mark==2){
        if(!T)return;
        TraverseTree(T->lchild,mark);
        cout<<T->data<<' ';
        TraverseTree(T->rchild,mark);
    }
    if(mark==3){
        if(!T)return;
        TraverseTree(T->lchild,mark);
        TraverseTree(T->rchild,mark);
        cout<<T->data<<' ';
    }
    if(mark==4){
        if(!T)return;

        InitQueue_L(Q);
        EnQueue_L(Q,T);
        while(!QueueEmpty_L(Q))
        {
            DeQueue_L(Q,a);
            cout<<a->data;
            if(a->lchild)EnQueue_L(Q,a->lchild);
            if(a->rchild)EnQueue_L(Q,a->rchild);
        }
        DestroyQueue_L(Q);
    }
}
```

3、建立主程序exam4_1.cpp。

```
#include <stdio.h>
#include <stdlib.h>
#include <iomanip.h>
#include "linkqueue.h"
void main()
{
    BiTree T;
    cout<<"1)初始化二叉树T."<<endl;
    InitBiTree(T);
    cout<<"2)创建二叉树，请输入扩展先序序列(如AB#DF###C#E##):";
    CreateBiTree(T);
    cout<<"3)二叉树T的深度="<<BiTreeDepth(T)<<endl;
    cout<<"4)二叉树T"<<(BiTreeEmpty(T)?"空":"非空")<<endl;
    cout<<"5)二叉树T的先序遍历序列:";
    TraverseTree(T, 1);
    cout<<endl;
    cout<<"6)二叉树T的中序遍历序列:";
    TraverseTree(T, 2);
    cout<<endl;
    cout<<"7)二叉树T的后序遍历序列:";
    TraverseTree(T, 3);
    cout<<endl;
    cout<<"8)二叉树T的层次序遍历序列:";
    TraverseTree(T, 4);
    cout<<endl;
    cout<<"9)二叉树T中'D'的双亲是:";
    cout<<(Parent(T,Value(T,'D')))->data<<endl;
    cout<<"10)二叉树T中'C'的左兄弟是:";
    cout<<(LeftSibling(T,Value(T,'C')))->data<<endl;
    cout<<"11)二叉树T中'B'的右兄弟是:";
    cout<<(RightSibling(T,Value(T,'B')))->data<<endl;
}
```

示例一的运行结果如下图：

图3.7 示例一运行结果

3.4.4.2 动手与实践

☞ huffman编解码。

1）**内容：**

利用 Huffman 编码进行通信可以大大提高信道的利用率，缩短信息传输时间，降低传输成本。但是，这要求在发送端通过一个编码系统对待传数据进行预先编码，在接收端进行解码。对于双工信道（即可以双向传输信息的信道），每端都需要一个完整的编/解码系统。

2）**要求：**

一个完整的huffman编解码系统应该具有以下功能：

初始化（**I**nitialization）。从终端读入字符集大小n，以及n个字符和n个权值，建立Huffman 树，并将它存入hfmTree 中。

编码（**E**ncoding）。利用已经建好的Huffman树（如果不在内存，则应从文件hfmTree中读取），对文件ToBeTran中的正文进行编码，然后将结果存入文件CodeFile中。

解码（**D**ecoding）。利用已经建立好的Huffman树将文件CodeFile中的代码进行解码，结果存入TextFile中。

打印代码文件（Print）。将文件CodeFile以紧凑的格式显示在终端上，每行 50 个代码。同时将此字符形式的编码文件写入文件CodePrint中。

打印Huffman树（Tree Printing）。将已经在内存中的Huffman树以直观的形式（树或者凹入的形式）显示在终端上，同时将此字符形式的Huffman树写入文件TreePrint中。

3）**测试数据：**

用下表给出的字符集和频度的实际统计数据建立Huffman树，并对以下报文进行编码和译码：“THIS PROGRAM IS MY FAVORITE”。

字符		A	B	C	D	E	F	G	H	I	J	K	L	M
频度	186	64	13	22	32	103	21	15	47	57	1	5	32	20
字符	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
频度	57	63	15	1	48	51	80	23	8	18	1	16	1	

4）**输入输出：**

- 字符集大小 n，n个字符和 n个权值均从终端读入，初始化后的huffman树存储在hfmTree文件中，待编码文件为ToBeTran,编码结果以文本的方式存储在文件CodeFile中，解码文件存在TextFile中,打印的编码和赫夫曼树分别存储在CodePrint和TreePrint文件中。
- 用户界面可以设计为“菜单”方式：显示上述功能符号，再加上一个退出功能“Q”，表示退出（quit）。用户键入一个选择功能符，此功能执行完毕后再显示此菜单，直至某次用户选择了 Q为止。

3.5 实验五：图

3.5.1 背景知识

图是一种比线性表和树更为复杂的数据结构。在图结构中，结点和结点之间是多对多的关系，任何两个结点都可能发生关系。图的应用也极其广泛，在最小生成树、最短路径和拓扑排序等应用方面，图有着重要的作用。

图的抽象数据类型定义如下：

```
ADT Graph{
    数据对象：V是同类数据元素的非空有限集，称顶点集。
    数据关系：R={<v_i,v_j>|v_i,v_j∈V且<v_i,v_j>表示从v_i到v_j得弧}
    基本操作：
    CreateGraph(&G,V,VR)  //创建图G
    DestroyGraph(&G)      //销毁图G
    LocateVex(G,u)     //返回u在G中的位置
    GetVex(G,v)        //返回G中v顶点的值
    PutVex(&G,v,value)    //为G中v结点赋值为value
    FirstAdjVex(G,v)      //返回G中v的第一个邻接点
    NextAdjVex(G,v,w)     //返回G中相对于w的v的下一个邻接点
    InsertVex(&G,u)       //插入结点
    DeleteVex(&G,v)       //删除结点
    InsertArc(&G,v,w)     //在G中v和w之间插入弧<v,w>
    DeleteArc(&G,v,w)     //在G中删除弧<v,w>
    DFSTraverse(G,v,visit())     //深度优先遍历
    BFSTraverse(G,v,visit())     //广度优先遍历
}
```

3.5.2 实验目的

- ✧ 通过对图特点的分析，掌握图的主要存储结构。
- ✧ 掌握图的几种常见存储结构下基本操作的实现。
- ✧ 通过图的遍历操作，进一步理解图存储结构的特点。

- 通过应用示例，学会使用图的遍历来解决问题。

3.5.3 实验要求

- 认真阅读本实验内容中给出的示例程序。
- 在计算机上输入示例程序中关于图的实现程序。
- 调试和运行示例程序。
- 编写程序实现实验中的“动手与实践”部分要求。

3.5.4 实验内容

3.5.4.1 示例一:图的基本操作

图的存储结构有邻接矩阵和邻接表两种方式，在本实验中，我们采用邻接表的存储结构来实现图的各种操作。因图的广度优先遍历操作需要用到队列，因此本示例的工程文件中需要包含linkqueue.h以及其所依赖的linklist.h。

同前一节，在本例中需要进入队列的是图的结点指针，因此需要对前面linklist.h和linkqueue做适当修改。在linklist.h中需要将链表的结点元素类型定义为结点的位置坐标类型即整型，改变如下：

```
typedef char ElemType;
```

修改为：

```
typedef int ElemType;
```

同样在linkqueue.h中需要修改队列的结点类型：

```
typedef char QElemType;
```

修改为：

```
typedef int QElemType;
```

此外本示例工程还需要包含linkqueue.cpp,以便可以直接使用链队列的实现。

1、建立一个头文件graph.h，内容如下：

```
#define MAX_VEX_NUM 20
typedef char VexType;
typedef enum{DG,DN,AG,AN} GraphKind;

typedef struct ArcNode{
    int adjvex;
    int weight;
    struct ArcNode  *nextarc;
}ArcNode,*ArcLink;

typedef struct VexNode{
    VexType   data;
    struct ArcNode   *firstarc;
}VexNode,AdjList[MAX_VEX_NUM];

typedef struct {
    AdjList   vertices;
    int VexNum,ArcNum;
    int kind;
}ALGraph;

void CreateGraph(ALGraph &G);  //创建图G
int LocateVex(ALGraph G, VexType u);  //返回u在G中的位置
VexType GetVex(ALGraph G,int v);       //返回G中v顶点的值
void PutVex(ALGraph &G,int v,VexType value); //为G中v赋值为value
int FirstAdjVex(ALGraph G,int v);   //返回G中v的第一个邻接点
int NextAdjVex(ALGraph G,int v,int w);
                            //返回G中相对于w的v的下一个邻接点
bool InsertArc(ALGraph &G,VexType v1,VexType v2); //在G中插入弧
bool DeleteArc(ALGraph &G,VexType v1,VexType v2); //在G中删除弧
void DFSTraverse(ALGraph G);    //深度优先遍历
void BFSTraverse(ALGraph G);    //广度优先遍历
```

2、建立一个实现文件graph.cpp，内容如下：

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

```
#include "linkqueue.h"
#include "graph.h"

void CreateGraph(ALGraph &G)  //创建图G
{
    int i,j,k;
    VexType  u,v;
    ArcNode *p;

    cout<<"请输入图的顶点数、弧数和类型(DG=0,DN=1,AG=2,AN=3):";
    cin>>G.VexNum>>G.ArcNum>>G.kind;
    cout<<"请输入第"<<G.VexNum<<"个结点的元素值,空格分隔:";
    for(i=0;i<G.VexNum;i++){
        cin>>G.vertices[i].data;      //顶点赋值
        G.vertices[i].firstarc=NULL;
    }
    for(k=0;k<G.ArcNum;k++){
        cout<<"输入第"<<k+1<<"个弧的两个顶点元素(空格分隔):";
        cin>>u>>v;
        i=LocateVex(G,u);
        j=LocateVex(G,v);
        p=new ArcNode;
        p->adjvex=j;
        p->nextarc=G.vertices[i].firstarc;
        G.vertices[i].firstarc=p;
        if(G.kind==AG){
            p=new ArcNode;
            p->adjvex=i;
            p->nextarc=G.vertices[j].firstarc;
            G.vertices[j].firstarc=p;
        }//if
    }//for
}

int LocateVex(ALGraph G, VexType u)    //返回u在G中的位置
{
    int i;
    for(i=0;i<G.VexNum;i++)
```

```
        if(G.vertices[i].data==u)return i;
    return -1;
}
VexType GetVex(ALGraph G,int v)       //返回G中v顶点的值
{
    return G.vertices[v].data;
}
void PutVex(ALGraph &G,int v,VexType value)   //为G中v结点赋值为
value
{
    G.vertices[v].data=value;
}
int FirstAdjVex(ALGraph G,int v)     //返回G中v的第一个邻接点
{
    if(G.vertices[v].firstarc)
        return G.vertices[v].firstarc->adjvex;
    else
        return -1;
}
int NextAdjVex(ALGraph G,int v,int w)    //返回G中相对于w的v的下一
个邻接点
{
    ArcNode *p;
    p=G.vertices[v].firstarc;
    while(p&&p->adjvex!=w)p=p->nextarc;
    if(!p||!p->nextarc)
        return -1;
    else
        return p->nextarc->adjvex;
}
bool InsertArc(ALGraph &G,VexType v1,VexType v2)    //插入弧
{
    ArcNode *p;
    int i,j;
    i=LocateVex(G,v1);
    j=LocateVex(G,v2);
    if(i==-1||j==-1)return false;
    p=new ArcNode;
```

```
    p->adjvex=j;
    p->nextarc=G.vertices[i].firstarc;
    G.vertices[i].firstarc=p;
    if(G.kind==AG){
        p=new ArcNode;
        p->adjvex=i;
        p->nextarc=G.vertices[j].firstarc;
        G.vertices[j].firstarc=p;
    }
    return true;
}
bool DeleteArc(ALGraph &G,VexType v1,VexType v2)   //在G中删除弧
{
    ArcNode *p,*q;
    int i,j;

    i=LocateVex(G,v1);
    j=LocateVex(G,v2);
    if(i==-1||j==-1)return false;

    p=G.vertices[i].firstarc;
    while(p->nextarc&&p->nextarc->adjvex!=j)p=p->nextarc;
    if(!p->nextarc)
        return false;    //v1 v2之间没有弧存在
    else{
        q=p->nextarc;
        p->nextarc=q->nextarc;
        delete q;
    }
    if(G.kind==AG){
        p=G.vertices[j].firstarc;
        while(p->nextarc&&p->nextarc->adjvex!=i)p=p->nextarc;
        if(!p->nextarc)
            return false;    //v2 v1之间没有弧存在
        else{
            q=p->nextarc;
            p->nextarc=q->nextarc;
            delete q;
```

```
        }
    }
    return true;
}
bool visited[MAX_VEX_NUM];
void DFS(ALGraph G,int v)
{
    int w;
    cout<<G.vertices[v].data;  //visit 内容
    visited[v]=true;
    for(w=FirstAdjVex(G,v);w!=-1;w=NextAdjVex(G,v,w))
        if(!visited[w])DFS(G,w);
}
void DFSTraverse(ALGraph G)    //深度优先遍历
{
    int i,v;
    for(i=0;i<G.VexNum;i++)visited[i]=false;
    for(v=0;v<G.VexNum;v++)
        if(!visited[v])DFS(G,v);
    cout<<endl;
}
void BFSTraverse(ALGraph G)    //广度优先遍历
{
    int i,v,u,w;
    LinkQueue Q;

    for(i=0;i<G.VexNum;i++)visited[i]=false;
    InitQueue_L(Q);
    for(v=0;v<G.VexNum;v++){               //防止非连通图
        if(visited[v])continue;
        cout<<G.vertices[v].data;   //visit内容
        visited[v]=true;
        EnQueue_L(Q,v);
        while(!QueueEmpty_L(Q)){
            DeQueue_L(Q,u);
            for(w=FirstAdjVex(G,u);w!=-1;w=NextAdjVex(G,u,w)){
                if(visited[w])continue;
                cout<<G.vertices[w].data;
```

```
                visited[w]=true;
                EnQueue_L(Q,w);
            }//for
        }//while
    }//for
    cout<<endl;
}
```

3、建立主程序exam5_1.cpp。

本示例主程序中输入的图逻辑结构如下：

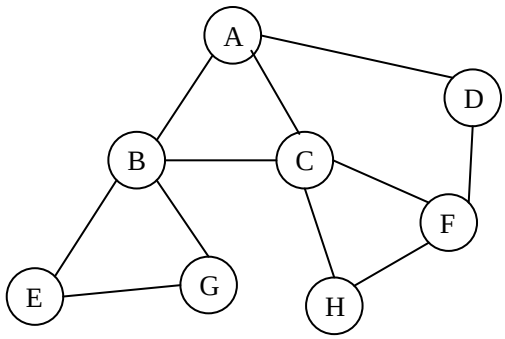

图3.8 示例一的图逻辑结构

```
#include <stdio.h>
#include <stdlib.h>
#include <iomanip.h>
#include "graph.h"
void main()
{
    ALGraph G;
    cout<<"1)创建图G."<<endl;
    CreateGraph(G);
    cout<<"2)广度优先遍历G:";
    BFSTraverse(G);
    cout<<"3)深度优先遍历G:";
    DFSTraverse(G);
    cout<<"4)元素'C'在图G中的存储位置坐标是:"<< LocateVex(G,'C')<<
endl;
```

```
    cout<<"5)图G中坐标是1的结点元素为"<<GetVex(G,1)<<endl;
    cout<<"6)把图G中坐标为3的顶点元素值设置为'Z'."<<endl;
    PutVex(G,3,'Z');
    cout<<"7)删除元素BC之间的弧."<<endl;
    DeleteArc(G,'B','C');
    cout<<"8)在元素AD之间插入弧."<<endl;
    InsertArc(G,'A','D');
    cout<<"9)广度优先遍历G:";
    BFSTraverse(G);
    cout<<"10)深度优先遍历G:";
    DFSTraverse(G);
}
```

示例一的运行结果如下图：

```
1)创建图G.
请输入图的顶点数、弧数和类型(DG=0,DN=1,AG=2,AN=3):8 11 2
请输入第8个结点的元素值,空格分隔:A B C D E F G H
输入第1个弧的两个顶点元素(空格分隔):A D
输入第2个弧的两个顶点元素(空格分隔):A C
输入第3个弧的两个顶点元素(空格分隔):A B
输入第4个弧的两个顶点元素(空格分隔):B G
输入第5个弧的两个顶点元素(空格分隔):B E
输入第6个弧的两个顶点元素(空格分隔):B C
输入第7个弧的两个顶点元素(空格分隔):C H
输入第8个弧的两个顶点元素(空格分隔):C F
输入第9个弧的两个顶点元素(空格分隔):D F
输入第10个弧的两个顶点元素(空格分隔):E G
输入第11个弧的两个顶点元素(空格分隔):F H
2)广度优先遍历G:ABCDEGFH
3)深度优先遍历G:ABCFHDEG
4)元素'C'在图G中的存储位置坐标是:2
5)图G中坐标是1的结点元素为B
6)把图G中坐标为3的顶点元素值设置为'Z'.
7)删除元素BC之间的弧.
8)在元素AD之间插入弧.
9)广度优先遍历G:ABCZEGFH
10)深度优先遍历G:ABCFHZEG
Press any key to continue
```

图3.9 示例一运行结果

3.5.4.2 动手与实践

☞ 管道铺设施工的最佳方案。

1）**内容：**

需要在某个城市n个居民小区之间铺设煤气管道，则在这n个居民小区之间只需要铺设n-1条管道即可。假设任意两个小区之间都可以铺设管道，但由于地理环境不同，所需要的费用也不尽相同。选择最优的方案能使总投资尽可能小，这个问题即为求无向网的最小生成树。

2）**要求：**

在可能假设的m条管道中，选取n-1条管道，使得既能连通n个小区，又能使总投资最小。每条管道的费用以网中该边的权值形式给出，网的存储采用邻接表的结构。

3）**测试数据：**

使用下图给出的无线网数据作为程序的输入，求出最佳铺设方案。右侧是给出的参考解。

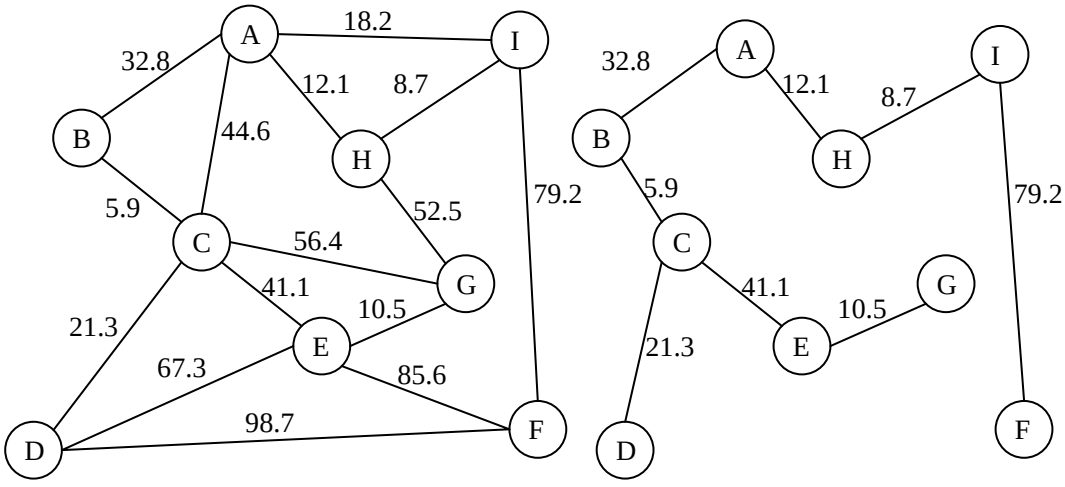

图3.10 小区煤气管道铺设网及其参考解

4）**输入输出：**

参考示例中图的创建方式，从键盘或文件读入上图中的无向网，以顶点对(i,j)的形式输出最小生成树的边。

3.6 实验六：查找表

3.6.1 背景知识

查找表是应用广泛的数据结构，有由同一类型的元素构成的集合。集合中元素之间存在着完全松散的关系。对查找表的操作一般包括：在表中查询某个特定的元素；在查找表中插入心元素；在查找表中删除某个数据元素。为了使得查找的效率更有效，我们往往对查找表内的数据加以约束，这就构成了一些特定的结构，如有序表、二叉排序树、哈希表等，相应的算法也就有了二分查找、二叉树查找、哈希查找等。本实验主要给出链接散列哈希查找表的存储结构及其实现。

哈希查找表的抽象数据类型定义如下：

```
ADT HashList{
    数据对象：D是相同类型元素构成的结合。
    数据关系：R={集合内的元素之间是松散关系}
    基本操作：
    InitHashList(&HT)  //创建哈希表
    DestroyHashList(&HT)      //销毁哈希表
    SearchHT(HT,e)       //查找元素
    InsertHT(&HT,e)       //插入元素
    DeleteHT(&HT,&e)       //删除元素
    TraverseHT(HT)      //遍历哈希表
}
```

3.6.2 实验目的

- ✧ 熟悉哈希表的有关概念。
- ✧ 掌握哈希表链接散列处理冲突的方法及其存储结构。
- ✧ 掌握哈希表链接处理冲突存储结构下基本操作的实现。

3.6.3 实验要求

- ✧ 认真阅读本实验内容中给出的示例程序。
- ✧ 在计算机上输入示例程序中关于哈希表的实现程序。
- ✧ 调试和运行示例程序。
- ✧ 编写程序实现实验中的“动手与实践”部分要求。

3.6.4 实验内容

3.6.4.1 示例一:哈希表的基本操作

哈希表的散列有开散列和闭散列两大类。本示例主要介绍开散列的存储结构及其操作的实现。开散列通过一个指针数组（指针的指针）来保存每个哈希值的链表头指针，具有相同哈希值的原始以一个单链表组织起来，头指针即存储在指针数组里。

1、建立一个头文件hash.h，内容如下：

```
typedef int ElemType;
typedef struct LNode{
    ElemType      data;
    struct LNode  *next;
}LNode,**ppLNode;
typedef struct{
    ppLNode head;
    int    length;
}LinkHashList;

int HashFunc(LinkHashList HT,ElemType e);  //哈希函数
void InitHashList(LinkHashList &HT,int m);  //创建哈希表
void DestroyHashList(LinkHashList &HT);     //销毁哈希表
void InsertHT(LinkHashList &HT,ElemType e);      //插入元素
void DeleteHT(LinkHashList &HT,ElemType &e);      //删除元素
LNode *  SearchHT(LinkHashList HT,ElemType e);      //查找元素
void TraverseHT(LinkHashList HT);     //遍历哈希表
```

2、建立一个实现文件hash.cpp，内容如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include "hash.h"

int HashFunc(LinkHashList HT,ElemType e)
{
    return e%HT.length;
}

void InitHashList(LinkHashList &HT,int m)  //创建哈希表
{
    HT.head=new LNode *[m];
    HT.length=m;
    for(int i=0;i<m;i++)HT.head[i]=NULL;
}
void DestroyHashList(LinkHashList &HT)     //销毁哈希表
{
    LNode *p;
    for(int i=0;i<HT.length;i++){
        p=HT.head[i];
        while(!p){
            HT.head[i]=p->next;
            delete p;
            p=HT.head[i];
        }
    }//for
    delete [] HT.head;
    HT.head=NULL;
}
void InsertHT(LinkHashList &HT,ElemType e)      //插入元素
{
    int d=HashFunc(HT,e);
    LNode *p=new LNode;
    p->data=e;
    p->next=HT.head[d];
```

```
    HT.head[d]=p;
}
void DeleteHT(LinkHashList &HT,ElemType &e)      //删除元素
{
    int d=HashFunc(HT,e);
    LNode *p=HT.head[d];
    if(!p)return;
    if(p->data==e){          //第一个结点即是e
        HT.head[d]=p->next;
        delete p;
        return;
    }
    LNode *q=p->next;
    while(!q&&q->data!=e){
        p=q;
        q=q->next;
    }
    if(q){
        p->next=q->next;
        delete q;
    }
}
LNode * SearchHT(LinkHashList HT,ElemType e)      //查找元素
{
    int d=HashFunc(HT,e);
    LNode *p=HT.head[d];
    while(p&&p->data!=e)p=p->next;
    return p;
}
void TraverseHT(LinkHashList HT)    //遍历哈希表
{
    LNode *p;
    for(int i=0;i<HT.length;i++){
        p=HT.head[i];
        cout<<"["<<i<<"]";
        while(p){
            cout<<p->data<<"->";
            p=p->next;
```

```
        }
        cout<<endl;
    }//for
}
```

3、建立主程序exam6_1.cpp。

本示例主程序中输入查找表元素序列为(7,15,20,31,48,53,64,76,82,99)。哈希表长度为11，哈希函数H(key)=key%11。

```
#include <stdio.h>
#include <stdlib.h>
#include <iomanip.h>
#include "hash.h"
void main()
{
    LinkHashList ht;
    int m,e;
    LNode *p;

    cout<<"1)创建哈希表,请输入哈希表的长度: ";
    cin>>m;
    InitHashList(ht,m);
    cout<<"2)请输入要插入哈希表的一组整数，以-1作为结束:";
    do{
        cin>>e;
        if(e==-1)break;
        InsertHT(ht,e);
    }while(1);

    cout<<"3)遍历哈希表:"<<endl;
    TraverseHT(ht);
    cout<<"4)输入要删除的元素:";
    cin>>e;
    DeleteHT(ht,e);
    cout<<"5)遍历哈希表:"<<endl;
    TraverseHT(ht);
    cout<<"4)输入要查找的元素:";
    cin>>e;
```

```
    p=SearchHT(ht,e);
    if(p)cout<<"查找成功"<<p->data<<endl;
    else cout<<"没有查找到！"<<endl;

}
```

示例一的运行结果如下图：

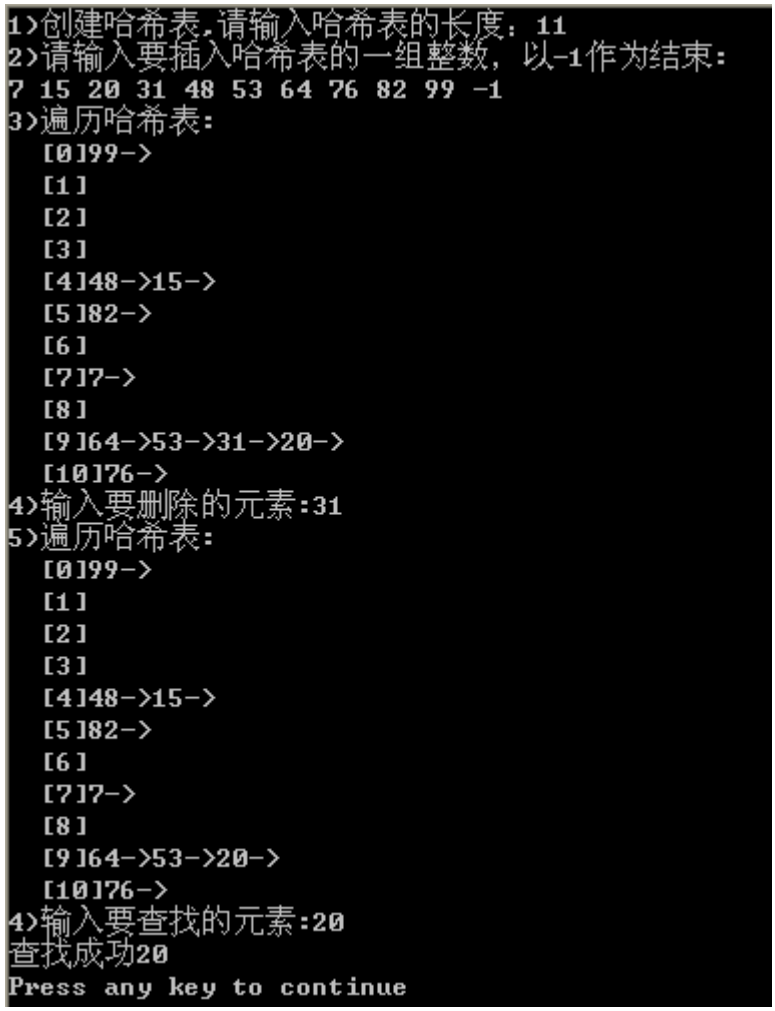

图3.11 示例一运行结果

3.6.4.2 动手与实践

☞ 利用哈希表统计两源程序的相似性。

1）**内容：**

对于两个 C 语言的源程序清单，用哈希表的方法分别统计两程序中使用C语言关键字的情况，并最终按定量的计算结果，得出两份源程序的相似性。

2）**要求与提示：**

C 语言关键字的哈希表可以自建，也可以采用下面的哈希函数昨晚参考：

Hash(key)=(key第一个字符序号*100+key最后一个字符序号)%41

表长m取43。此题的工作主要是扫描给定的源程序，累计在每个源程序中C语言关键字出现的频度。为保证查找效率，建议自建哈希表的平均查找长度不大于2。 扫描两个源程序所统计的所有关键字不同频度， 可以得到两个向量。如下面简单的例子所示：

关键字	void	int		for	char		if	else		while
程序1中 关键字频度	4	3		4	3		7	0		2
程序2中 关键字频度	4	2		5	4		5	2		1
哈希地址	0	1	2	3	4	5	6	7	8	9

根据程序1和程序2中关键字出现的频度，可提取到两个程序的特征向量X1和X2，其中

$X1=(4\ 3\ 0\ 4\ 3\ 0\ 7\ 0\ 0\ 2)^T$

$X2=(4\ 2\ 0\ 5\ 4\ 0\ 5\ 2\ 0\ 1)^T$

一般情况下，可以通过计算向量Xi和Xj的相似值来判断对应两个程序的相似性，相似值的判别函数计算公式为：

$$S(X_i,X_j)=\frac{X_i^T}{|X_i|\cdot|X_j|} \qquad (3\text{-}1)$$

其中，$|X_i|=\sqrt{X^T\cdot X_i}$ 。$S(X_i,X_j)$的值介于[0,1]之间，也称广义余弦，即$S(X_i,X_j)=COS\theta$。$X_i=X_j$ 时，显见$S(X_i,X_j)=1$，$\theta=0$；X_iX_j 差别很大时，

$S(X_i,X_j)$接近0，θ接近π/2。如X1=(1 0)^T,X2=(0 1)^T,则$S(X_i,X_j)=0$，θ=π/2。可以用下面的二维的图示来直观地表示向量的相似程度：

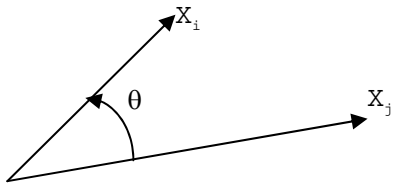

图3.12 向量相似度示意图

有些情况下，还需要做进一步的考虑，如下图所示：

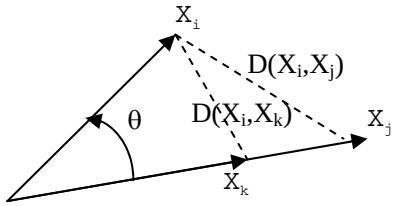

图3.13 向量几何距离

从图中看出， 尽管$S(X_i,X_j)$和$S(X_i,X_k)$的值是一样的，但直观上X_i与X_k更相似。因此当S值接近1 的时候，为避免误判相似性（可能是夹角很小，模值很大的向量），应当再次计算之间的“几何距离” $D(X_i,X_k)$。其计算公式为：

$$D(X_i,X_k)=\mid X_i-X_k\mid=\sqrt{(X_i-X_k)^T(X_i-X_k)} \qquad (3\text{-}2)$$

最后的相似性判别计算可分两步完成：

第一步 用式（3-1）计算S，把接近 1的保留，抛弃接近0的情况（把不相似的排除）；

第二步 对保留下来的特征向量，再用式（3-2）计算D，如D值也比较小，说明两者对应的程序确实可能相似（慎重肯定相似的）。

S和D的值达到什么门限才能决定取舍？需要积累经验，选择合适的阈值。

3） **测试数据：**

做几个编译和运行都无误的C程序，程序之间有相近的和差别大的，用上述方

法求S，并对比差异程度。

4）**输入输出：**

输入为若干个C源程序，输出为程序间的相似度以及向量的几何距离。