数据结构
实践教程

前言

数据结构是计算机专业的必修。主干课程之一，它旨在使读者学会分析研究数据对象的特性，学会数据的组织方法，以便选择合适的数据逻辑结构和存储结构，以及相应的运算（操作），把现实世界中的问题转化为计算机内部的表示和处理，这是一个良好的程序设计技能训练的过程。在整个教学或学习过程中，解题能力和技巧的训练是一个重要的环节。为了帮助教师讲授“数据结构”，满足指导和评价“课程设计”的需要，为了帮助和指导读者更好地学习数据结构这门课程，我们特编写了这本《数据结构实践教程》辅助教材，旨在弥补课堂教学和实验中的不足，帮助学生充分理解和巩固所学的基本概念、原理和方法，达到融会贯通、举一反三的目的。

实践证明，理解课程内容与较好地解决实际问题之间存在着明显差距，而算法设计完成的质量与基本的程序设计素质的培养是密切相关的。要想理解和巩固所学的基本概念。原理和方法，牢固地掌握所学的基本知识。基本技能，达到融会贯通。举一反三的目的，就必须多做。多练。多见（见多识广）。正是为了达到上述目的，书中用一些实际的应用，对一些重要的数据结构和算法进行解读。经过循序渐进地训练，就可以使读者掌握更多的程序设计技巧和方法，提高分析。解决问题的能力。

本书根据学生的基础知识和兴趣爱好将内容分为基础篇和提高篇两个部分。第一部分基础篇 精选出适当的、与实际生活结合密切的课程设计实例加以分析实现。第二部分提高篇旨在使读者通过运用数据结构知识及复杂算法去解决现实世界中的一些实际问题。

本书依据数据结构课程教学大纲要求，同时又独立于具体的教科书，既重视实践应用，又重视理论分析，本书的主要特点有：

- 本书精选出来的实例项目经典、实用、具有一定的趣味性，其内容丰富、涉及面广、难易适当，能给读者以启发，达到让读者掌握相关知识和开阔视野的目的
- 为了提高学生分析问题、解决问题的能力，本书对实例项目进行分析，其设计思路清晰流畅，值得参考。
- 本书不仅仅是对照数据结构课程教学大纲举些例子说明数据结构能解决什么问题，而是通过分析具体的实例项目，得到对数据组织关系的需求，从而选择某个数据结构适应一些特定的问题和算法，并说明使用这种数据结构的优缺点。
- 所有实例项目都给出了参考算法和源程序代码并在 Turbo C 和 VisualC++6.0 环境下运行通过。

由于作者水平有限、时间仓促，本书难免存在一些缺点和错误，恳请广大读者及同行们批评指正。

目　录

第一部分 基础篇

第一章 线性表

线性表是数据结构中最简单、最常用的一种线性结构，也是学习数据结构全部内容的基础，其掌握的好坏直接影响着后继知识的学习。本章通过四个模拟项目来学习线性表的顺序和链式存储结构，首先通过使用有关数组的操作实现学生成绩管理，其次通过使用有关线性链表的操作实现考试报名管理，然后通过使用循环链表的操作实现约瑟夫生者死者游戏。

1.1 学生成绩管理

1.1.1 项目简介

学生成绩管理是学校教务部门日常工作的重要组成部分，其处理信息量很大。本项目是对学生成绩管理的简单模拟，用菜单选择方式完成下列功能：输入学生数据；输出学生数据；学生数据查询；添加学生数据；修改学生数据；删除学生数据。

1.1.2 设计思路

本项目的实质是完成对学生成绩信息的建立、查找、插入、修改、删除等功能，可以首先定义项目的数据结构，然后将每个功能写成一个函数来完成对数据的操作，最后完成主函数以验证各个函数功能并得出运行结果。

1.1.3 数据结构

本项目的数据是一组学生的成绩信息，每条学生的成绩信息由学号、姓名和成绩组成，这组学生的成绩信息具有相同特性，属于同一数据对象，相邻数据元素之间存在序偶关系。由此可以看出，这些数据具有线性表中数据元素的性质，所以该系统的数据采用线性表来存储。

顺序表是线性表的顺序存储结构，是指用一组连续的内存单元依次存放线性表的数据元素。在顺序存储结构下，逻辑关系相邻的两个元素在物理位置上也相邻，这是顺序表的特点。本项目可以采用顺序表的线性表顺序存储结构。

若一个数据元素仅占一个存储单元，则其存储方式参见图 1-1。

从图 1-1 中可见，第 i 个数据元素的地址为

Loc(ai)=loc(a1)+(i-1)

假设线性表中每个元素占用 k 个存储单元，那么在顺序表中，线性表的第 i 个元素的存储位置与第 1 个元素的存储位置的关系是

Loc(ai)=loc(a1)+(i-1)*k

这里 Loc(ai)是第 i 个元素的存储位置，loc(a1)是第 1 个元素的存储位置，也称为线性表的基址。显然，顺序表便于进行随机访问，故线性表的顺序存储结构是一种随机存储结构。

顺序表适宜于做查找这样的静态操作；顺序存储的优点是存储密度大，存储空间利用率高。缺点是插入或删除元素时不方便。

由于 C 语言的数组类型也有随机存储的特点，一维数组的机内表示就是顺序结构。因此，可用 C 语言的一维数组实现线性表的顺序存储。数组实现线性表的顺序存储的优点是可以随机存取表中任一元素 O(1)，存储空间使用紧凑；缺点是在插入，删除某一元素时，需要移动大量元素 O(n)，预先分配空间需按最大空间分配，利用不充分，表容量难以扩充。

用结构体类型定义每个学生数据，故该数组中的每个数据的结构可描述为：

```
typedef struct STU
{    char stuno[10];          //学号
     char name[10];           //姓名
     float score;             //成绩
} ElemType;
```

1.1.4 程序清单

```
#include<iostream.h>
#include<iomanip.h>
#include<malloc.h>
#include<string.h>

#define MaxListSize 20
#define EQUAL 1

typedef struct STU{
   char stuno [10];
   char name [10];
   float score;
}ElemType;
```

```
class List
{private:
   //线性表的数组表示
   ElemType elem[MaxListSize];
   int length;
   int MaxSize;
 public:
   //输入学生数据
   void init(List **L,int ms);
   //删除所有学生数据
   void DestroyList(List &L){free(&L);}
   //将顺序表置为空表
   void ClearList(){length=0;}
   //判断顺序表是否为空表
   bool ListEmpty()
     {return length==0;}
   //判断顺序表是否为满
   bool ListFull()
     {return length==MaxSize;}
   //删除某个学生数据
   bool ListDelete(int,ElemType &e);
   //遍历顺序表
   void ListTraverse();
   //返回顺序表的长度
   int ListLength();
   //学生数据查询
   void GetElem(int,ElemType *);
   //修改学生数据
   bool UpdateList(ElemType& e,ElemType);
   //添加学生数据
   bool ListInsert(int,ElemType &);
   //对学生数据按升序或降序输出
   void printlist(int);
};
```

```
void List::init(List **L,int ms)
{*L=(List *)malloc(sizeof(List));
 (*L)->length=0;
 (*L)->MaxSize=ms;
}
int List::ListLength()
{return length;}

bool List::ListDelete(int mark,ElemType &e)
{int i,j;
 if(ListEmpty()) return false;
 if(mark>0) {   //删除表头元素
      e=elem[0];
      for(i=1; i<length; i++)
        elem[i-1]=elem[i];}
 else     //删除表尾元素
    if(mark<0) e=elem[length-1];
    else {   //删除值为 e 的元素
      for(i=0;i<length;i++)
        if(strcmp(elem[i].name,e.name)==0) break;
        if(i>=length) return false;
        else e=elem[i];
        for(j=i+1;j<length;j++)
          elem[j-1]=elem[j];}
 length--;
 return true;
}
void List::ListTraverse()
{for(int i=0;i<length;i++)
    {cout<<setw(8)<<elem[i].name;
      cout<<setw(10)<<elem[i].stuno;
      cout<<setw(9)<<elem[i].age;
      cout<<setw(8)<<elem[i].score<<endl;}
}
void List::GetElem(int i,ElemType *e)
```

```
{*e=elem[i];}

bool List::EqualList(ElemType *e1,ElemType *e2)
{ if (strcmp(e1->name,e2->name))
      return false;
   if (strcmp(e1->stuno,e2->stuno))
      return false;
   if (e1->age!=e2->age)
      return false;
   if (e1->score!=e2->score)
      return false;
   return true;
}
bool List::Less_EqualList(ElemType *e1,ElemType *e2)
{ if(strcmp(e1->name,e2->name)<=0) return true;
   else return false;
}
bool List::LocateElem(ElemType e,int type)
{ int i;
   switch (type)
      { case EQUAL:
      for(i=0;i<length;i++)
         if(EqualList(&elem[i],&e))
            return true;
      break;
         default:break;}
   return false;
}
//修改学生数据
bool List::UpdateList(ElemType& e,ElemType e1)
{for(int i=0;i<length;i++)
    if(strcmp(elem[i].name,e.name)==0) {
       elem[i]=e1;return true;}
 return false;
}
```

```
bool List::ListInsert(int i,ElemType &e)
{ElemType *p,*q;
 if(i<1||i>length+1) return false;
 q=&elem[i-1];
 for(p=&elem[length-1];p>=q;--p)
    *(p+1)=*p;
 *q=e;
 ++length;
 return true;
}
//对学生成绩按升序或降序输出
void List::printlist(int mark)
{int* b=new int[length];
 int i,k;
 cout<<"      姓名        学号          成绩\n";
 if(mark!=0){
  for(i=0; i<length;i++) b[i]=i;
  for(i=0; i<length;i++) {k=i;
    for(int j=i+1;j<length;j++) {
      if(mark==1&&elem[b[j]].score<elem[b[k]].score) k=j;
      if(mark==-1&&elem[b[k]].score<elem[b[j]].score) k=j;}
      if(k!=i) {int x=b[i];b[i]=b[k];b[k]=x;}}
  for(int i=0;i<length;i++)
  {cout<<setw(8)<<elem[b[i]].name;
    cout<<setw(10)<<elem[b[i]].stuno;
    cout<<setw(9)<<elem[b[i]].age;
    cout<<setw(8)<<elem[b[i]].score<<endl;}}
 else {
  for(i=0;i<length;i++)
    {cout<<setw(8)<<elem[i].name;
      cout<<setw(10)<<elem[i].stuno;
      cout<<setw(9)<<elem[i].age;
      cout<<setw(8)<<elem[i].score<<endl;}}
}
```

```
void main()
{ cout<<"linelist1m.cpp 运行结果:\n";
   ElemType e,e1,e2,e3,e4,e5,e6;
   List *La,*Lb,*Lc;
   int k;
   cout<<"首先调用插入函数.\n";
   La->init(&La,4);
   strcpy(e1.name,"stu1");
   strcpy(e1.stuno,"100001");
   e1.age=22;
   e1.score=88;
   La->ListInsert(1,e1);
   strcpy(e2.name,"stu2");
   strcpy(e2.stuno,"100002");
   e2.age=21;
   e2.score=79;
   La->ListInsert(2,e2);
   strcpy(e3.name,"stu3");
   strcpy(e3.stuno,"100003");
   e3.age=19;
   e3.score=87;
   La->ListInsert(3,e3);
   La->printlist(0);
   cout<<"表 La 长:"<<La->ListLength()<<endl;
   cin.get();

   Lb->init(&Lb,4);
   strcpy(e4.name,"zmofun");
   strcpy(e4.stuno,"100001");
   e4.age=20;
   e4.score=94;
   Lb->ListInsert(1,e4);
   strcpy(e5.name,"bobjin");
   strcpy(e5.stuno,"100002");
   e5.age=23;
```

```
    e5.score=69;
    Lb->ListInsert(2,e5);
    strcpy(e6.name,"stu1");
    strcpy(e6.stuno,"100001");
    e6.age=22;
    e6.score=88;
    Lb->ListInsert(3,e6);
    Lb->printlist(0);
    cout<<"表 Lb 长:"<<Lb->ListLength()<<endl;
    cin.get();

    k=Lc->ListDelete(-1,e6);
    if(k==0) cout<<"删除失败!\n";
    else cout<<"删除成功!\n";
    cout<<"输出表 Lc:\n";
    Lc->printlist(0);
    cin.get();
    cout<<"按成绩升序输出表 Lc\n";
    Lc->printlist(1);cin.get();
    cout<<"按成绩降序输出表 Lc\n";
    Lc->printlist(-1);cin.get();
}
```

1.1.5 运行结果

首先建立学生信息管理，输出结果为：

姓名　　学号　　成绩

Stu1　　100001　　80

Stu2　　100002　　91

Stu3　　100003　　56

其次查询学号为 100002 的学生的成绩，输出结果为：

91

再次调用插入函数，插入 Stu4 成功！输入结果为：

姓名　　学号　　成绩

Stu1　　100001　　80

Stu2　　100002　　91

Stu3　　100003　　56

Stu4　　100004　　75

最后删除 Stu2 成果！输出结果为：

姓名　　学号　　成绩

Stu1　　100001　　80

Stu3　　100003　　56

Stu4　　100004　　75

查询不及格的学生，输出结果为：

Stu3　　100003　　56

1.2 考试报名管理

1.2.1 项目简介

考试报名工作给各高校报名工作带来了新的挑战，给教务管理部门增加了很大的工作量，报名数据手工录入既费时又会不可避免地出现错误，同时也给不少学生以可乘之机。本项目是对考试报名管理的简单模拟，用菜单选择方式完成下列功能：输入考生信息；输出考生信息；查询考生信息；添加考生信息；修改考生信息；删除考生信息。

1.2.2 设计思路

本项目的实质是完成对考生信息的建立、查找、插入、修改、删除等功能，可以首先定义项目的数据结构，然后将每个功能写成一个函数来完成对数据的操作，最后完成主函数以验证各个函数功能并得出运行结果。

1.2.3 数据结构

本项目的数据是一组考生信息，每条考生信息由准考证号、姓名、性别、年龄、报考类别等信息组成，这组考生信息具有相同特性，属于同一数据对象，相邻数据元素之间存在序偶关系。由此可以看出，这些数据也具有线性表中数据元素的性质，所以该系统的数据可以采用线性表来存储。

从上一节的例子中可见，线性表的顺序存储结构的特点是逻辑关系相邻的两个元素在物理位置上也相邻，因此可以随机存储表中任一元素，它的存储位置可用一个简单、直观的公式来表示。然而，从另一个方面来看，这个特点也铸成了这种存储结构的弱点：在做插入或

删除操作时，需要移动大量元素。为克服这一缺点，我们引入另一种存储形式－－链式存储。链式存储是线性表的另一种表示方法，由于它不要求逻辑上相邻的元素在物理位置上也相邻，因此它没有顺序存储结构的弱点，但同时也失去了顺序表可随机存取的特点。

链式存储的优点是插入或删除元素时很方便，使用灵活。缺点是存储密度小，存储空间利用率低。事实上，链表插入、删除运算的快捷是以空间代价来换取时间。

顺序表适宜于做查找这样的静态操作；链表宜于做插入、删除这样的动态操作。若线性表的长度变化不大，且其主要操作是查找，则采用顺序表；若线性表的长度变化较大，且其主要操作是插入、删除操作，则采用链表。

本项目对考生数据主要进行插入、删除、修改等操作，所以采用链式存储结构比较适合。用结构体类型定义每个考生信息，故该单链表中的每个结点的结构可描述为：

```
typedef struct examinee
{    char examno[10];          //准考证号
     char name[10];            //姓名
     char sex;
     float age;
char examtype[5];              //成绩
} ElemType;
```

1.2.4 程序清单

```
//单链表的类定义 linklist3.h
#ifndef linklist3H
#define linklist3H
#define LEN 30
//定义 ElemType 为 int
typedef int ElemType;
//单链表中结点的类型
typedef struct LNode{
   ElemType data;//值域
   LNode *next;   //指针域
}LNode;
class LinkList{
   LNode *head;
 public:
//构造函数
```

```
    LinkList();
//析构函数
    ~LinkList();
//清空单链表
    void ClearList();
//求单链表长度
    int ListSize();
//检查单链表是否为空
    bool ListEmpty();
//返回单链表中指定序号的结点值
    ElemType GetElem(int pos);
//遍历单链表
    void TraverseList(void f(ElemType &));
//从单链表中查找元素
    bool FindList(ElemType& item);
//更新单链表中的给定元素
    bool UpdateList(const ElemType& item,ElemType e);
//向单链表插入元素,mark=0 插在表首,否则插在表尾
    void InsertList(ElemType item,int mark);
//从单链表中删除元素 , mark 为要删除的第几个元素
    bool DeleteList(ElemType& item,int mark);
//对单链表进行有序排列  mark>0 升序,否则降序
    void pailie(int mark=1);
//对单链表进行有序输出,mark=0 不排序,mark>0 升序,mark<0 降序
    void OrderOutputList(int mark=0);
};
#endif

//linklist3.cpp
#include "linklist3.h"
LinkList::LinkList()//构造函数
{head=new LNode;
 head->next=NULL;
}
LinkList::~LinkList()//析构函数
```

```
{LNode *p=head->next ,*q;
 while(p)
   {q=p->next ;
    free(p);
    p=q;
   }
}
void LinkList::ClearList()//清空单链表
{LNode*p=head->next ,*q;
 while(p)
   {q=p->next;
    free(p);
    p=q;
   }
 head->next =NULL;
}
int LinkList::ListSize()//求单链表长度
{LNode*p=head->next ;
 int i=0;
 while(p)
   {i++;
    p=p->next ;}
 return i;
}
bool LinkList::ListEmpty()//检查单链表是否为空
{return ListSize()==0;}
//返回单链表中指定序号的结点值
ElemType LinkList::GetElem(int pos)
{LNode*p=head->next ;
 int i=1;
 while(p)
   {if(i++==pos)return p->data ;
    p=p->next ;
   }
 return head->data ;
```

```
}
void LinkList::TraverseList(void f(ElemType &))//遍历单链表
{LNode*p=head->next ;
 while(p)
   {f(p->data );
    p=p->next ;}
}
bool LinkList::FindList(ElemType& item)//从单链表中查找元素
{LNode*p=head->next ;
 while(p)
   {if(p->data==item)return 1;
    p=p->next ;}
 return 0;
}
//更新单链表中的给定元素
bool LinkList::UpdateList(const ElemType &item,ElemType e)
{LNode*p=head->next ;
 bool flag=0;
 while(p)
   {if(p->data==item)
      {p->data=e;
       flag=1;}
    p=p->next ;}
 return flag;
}
//向单链表插入元素
void LinkList::InsertList(ElemType item,int mark)
{LNode *q= new LNode;
 q->data = item;
 if(mark==0)
   {q->next = head->next ;
    head->next=q;
    return;}
 LNode *p=head;
 while(p->next)
```

```
    {p=p->next ;}
  q->next =NULL;
  p->next =q;
}
//从单链表中删除元素
bool LinkList::DeleteList(ElemType& item,int mark)
{if(ListEmpty()||mark<1||mark>ListSize())return 0;
  LNode *p=head,*q;
  for(int i=0;i<mark-1;i++)
     p=p->next;
  item=p->next->data;
  q=p->next->next ;
  free(p->next );
  p->next=q;
  return 1;
}
//对单链表进行有序排列 mark>0 升序,否则降序
void LinkList::pailie(int mark)
{ElemType a[LEN+1];
  LNode *p=head->next;
  int k ;
  for(k=1;p!=NULL;k++,p=p->next )
     a[k]=p->data;
  k--;
  for(int i=1;i<k;i++)
    for(int j=1;j<=k-i;j++)
      {int t;
       if( mark>0&&a[j]>a[j+1]||mark<0&&a[j]<a[j+1])
         {t=a[j+1];
          a[j+1]=a[j];
          a[j]=t;}}
  p=head->next;
  for(int j=1;j<=k;j++,p=p->next )
    p->data=a[j];
}
```

```
//对单链表进行有序输出
void LinkList::OrderOutputList(int mark)
{ElemType a[LEN+1];
  LNode *p=head->next;
  int k ;
  for( k=1;p!=NULL;k++,p=p->next )
     a[k]=p->data;
  k--;
  for(int i=1;i<k;i++)
    for(int j=1;j<=k-i;j++)
      {int t;
       if( mark>0&&a[j]>a[j+1]||mark<0&&a[j]<a[j+1])
         {t=a[j+1];
          a[j+1]=a[j];
          a[j]=t;}}
  for(int j=1;j<=k;j++)
    cout<<a[j]<<"    ";
}

#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
//#include<stdio.h>
#include"linklist3.cpp"
void ff(int &a)//用于遍历的函数
{cout<<a<<"    ";}
void main()
{cout<<"\nlinklist3m.cpp 运行结果:\n";
  int init_size,seed,xu;
  cout<<"首先请构造单链表 list";
  cout<<"\n 初始化长度(1--30):";
  cin>>init_size;
  seed=150;
  cout<<"是否排序:(=0 不排序,=1 升序,=-1 降序):";
  cin>>xu;
```

```
cout<<"\n 单链表 list 构造成功!"<<"\n 它是:";
  list.TraverseList(ff);
  cout<<"\n 它为空吗?(1:是;0:不是):"<<list.ListEmpty();
  cout<<"\n 长度为:"<<list.ListSize() ;
  int i;
  cout<<"\n 请输入你想得到第几个元素的值(1--"<<init_size<<"):";
  cin>>i;
  cout<<"单链表 list 中第"<<i<<"的值是"<<list.GetElem(i);
  int it;
  cout<<"\n 请输入你想删除第几个元素的值(1--"<<init_size<<"):";
  cin>>i;
  list.DeleteList(it,i);
  cout<<"\n 单链表 list 删除第"<<i<<"个元素"<<"\""<<it<<"\""<<"后变为:";
  list.TraverseList(ff);//对单链表 list 每个数进行遍历.
  int news,olds;
  cout<<"\n 请输入要被修改的元素:"; cin>>olds;
  cout<<"请输入修改后要变成的元素:";cin>>news;
  list.UpdateList(olds,news);
  cout<<"\n 修改后单链表 list 变为:";
  list.TraverseList(ff);
  cout<<"\n 下面请构造单链表 list2";
  cout<<"\n 请输入单链表 list2 初始化长度(1--30):";
  cin>>init_size;
  seed=120;
  cout<<"请选择是否排序:(=0 不排序,=1 升序,=-1 降序):";
  cin>>xu;
cout<<"\n 按回车键结束...";
  cin.get();cin.get();}
```

1.2.5 运行结果

1.3 约瑟夫生者死者游戏

1.3.1 项目简介

约瑟夫生者死者游戏的大意是：30 个旅客同乘一条船，因为严重超载，加上风高浪大，危险万分；因此船长告诉乘客，只有将全船一半的旅客投入海中，其余人才能幸免遇难。无奈，大家只得同意这种办法，并议定 30 个人围成一圈，由第一个人开始，依次报数，数到第 9 人，便把他投入大海中，然后从他的下一个人数起，数到第 9 人，再将他投入大海，如此循环，直到剩下 15 个乘客为止。问哪些位置是将被扔下大海的位置。

1.3.2 设计思路

本游戏的数学建模如下：假设 n 个旅客排成一个环形，依次顺序编号 1，2，…，n。从某个指定的第 1 号开始，沿环计数，每数到第 m 个人就让其出列，且从下一个人开始重新计数，继续进行下去。这个过程一直进行到剩下 k 个旅客为止。

本游戏的要求用户输入的内容包括：

1. 旅客的个数，也就是 n 的值；

2. 离开旅客的间隔数，也就是 m 的值；

3. 所有旅客的序号作为一组数据要求存放在某种数据结构中。

本游戏要求输出的内容是包括

1. 离开旅客的序号；

2. 剩余旅客的序号；

所以，根据上面的模型分析及输入输出参数分析，可以定义一种数据结构后进行算法实现。

1.3.3 数据结构

为了解决这一问题，可以用长度为 30 的数组作为线性存储结构，并把该数组看成是一个首尾相接的环形结构，那么每投入大海一个乘客，就要在该数组的相应位置做一个删除标记，该单元以后就不再作为计数单元。这样做不仅算法较为复杂，而且效率低，还要移动大量的元素。用单循环链表解决这一问题，实现的方法相对要简单得多。首先要定义链表结点，单循环链表的结点结构与一般的结点结构完全相同，只是数据域用一个整数来表示位置；然后将它们组成具有 30 个结点的单循环链表。接下来从位置为 1 的结点开始数，数到第 8 个结点，就将下一个结点从循环链表中删去，然后再从删去结点的下一个结点开始数起，数到第 8 个结点，再将其下一个结点删去，如此进行下去，直到剩下 15 个结点为止。

为了不失一般性，将 30 改为一个任意输入的正整数 n，而报数上限（原为 9）也为一个任选的正整数 k。这样该算法描述如下：

(1) 创建含有 n 个结点的单循环链表；

(2) 生着与死者的选择：

p 指向链表的第一个结点，初始 i 置为 1；

while(i<=n/2)　//删除一半的结点

{　从 p 指向的结点沿链前进 k-1 步；

删除第 k 个结点（q 所指向的结点）；

p 指向 q 的下一个结点；

输出其位置 q->data;

i 自增 1；

}

(3) 输出所有生者的位置。

1.3.4 程序清单

```
LinkList InitRing(int n, LinkList R)                    //尾插入法建立单循环链表函数
{
    ListNode *p, *q;
    int I;
    R=q=(LinkNode *)malloc(sizeof(LinkNode));
    for(i=1;i<n;i++){
        p=(LinkNode *)malloc(sizeof(LinkNode));
        q->data=i;
        q->next=p;
        q=p;
    }
    p->data=n;
    p->next=R;
    R=p;
    return R;
}
LinkList DeleteDeath(int n, int k, LinkList R)              //生者与死者的选择
{
    int i, j;
```

```
    ListNode *p, *q;
    p=R;
    for(i=1; i<n/2; i++){      //删除一半结点
        for(j=1; j<k-1; j++) //沿链前进 k-1 步
            p=p->next;
        q=p->next;
        p->next=q->next;
        printf("%4d", q->data);
        free(q);
}
R=p; return R;
    }
    void OutRing(int n, LinkList R){                    //输出所有生者
        int i;
        LinkNode *p;
        p=R;
        for(i=1;i<=n/2; i++, p=p->next){
            printf("%4d", p->data)
        }
    }
```

有了上述算法分析和设计之后，实现就比较简单了。首先要定义一个链表结构类型，然后编写一个主函数调用上面已定义好的函数即可。主函数的源程序如下：

```
#include<stdio.h>
#include<stdlib.h>
typedef struct node{
    int data;
    struct node * next;
}ListNode;
typedef ListNode * LinkList;
void main(){
    LinkList R;
    int n,k;
    LinkList InitRing(int n, LinkList R);
    LinkList DeleteDeath(int n, int k, LinkList R);
    void OutRing(int n, LinkList R);
```

```
    printf("总人数 n. 报数上限 k=");
    scanf("%d%d",&n, &k);
    R=InitRing(n, R);
    R=DeleteDeath(n, k, R);
    OutRing(n, R);
}
```

1.3.5 运行结果

编译运行上述程序，提示：总人数 n. 报数上限 k=

输入 30 和 9 后并“回车”可得出如下结果：

9 18 27 6 16 26 7 19 30 12 24 8 22 5 23

21 25 28 29 1 2 3 4 10 11 13 14 15 17 20

1.4 约瑟夫双向生死游戏

1.4.1 项目简介

约瑟夫双向生死游戏是在约瑟夫生者死者游戏的基础上，正向计数后反向计数，然后再正向计数。具体描述如下：30 个旅客同乘一条船，因为严重超载，加上风高浪大，危险万分；因此船长告诉乘客，只有将全船一半的旅客投入海中，其余人才能幸免遇难。无奈，大家只得同意这种办法，并议定 30 个人围成一圈，由第一个人开始，顺时针依次报数，数到第 9 人，便把他投入大海中，然后从他的下一个人数起，逆时针数到第 5 人，将他投入大海，然后从他逆时针的下一个人数起，顺时针数到第 9 人，再将他投入大海，如此循环，直到剩下15 个乘客为止。问哪些位置是将被扔下大海的位置。

1.4.2 设计思路

本游戏的数学建模如下：假设 n 个旅客排成一个环形，依次顺序编号 1，2，…，n。从某个指定的第 1 号开始，沿环计数，数到第 m 个人就让其出列，然后从第 m+1 个人反向计数到 m-k+1 个人，让其出列，然后从 m-k 个人开始重新正向沿环计数，再数 m 个人后让其出列，然后再反向数 k 个人后让其出列。这个过程一直进行到剩下 q 个旅客为止。

本游戏的要求用户输入的内容包括：

1. 旅客的个数，也就是 n 的值；

2. 正向离开旅客的间隔数，也就是 m 的值；

3. 反向离开旅客的间隔数，也就是 k 的值；

4. 所有旅客的序号作为一组数据要求存放在某种数据结构中。

本游戏要求输出的内容是包括

1. 离开旅客的序号；

2. 剩余旅客的序号；

所以，根据上面的模型分析及输入输出参数分析，可以定义一种数据结构后进行算法实现。

1.4.4 数据结构

约瑟夫双向生死游戏如果用单循环链表作为线性存储结构，就只能正向计数结点，反向计数比较困难，算法较为复杂，而且效率低。用双向循环链表解决这一问题，实现的方法相对要简单得多。

为了不失一般性，将 30 改为一个任意输入的正整数 n，而正向报数上限（原为 9）也为一个任选的正整数 m，正向报数上限（原为 5）也为一个任选的正整数 k，。这样该算法描述如下：

(1) 创建含有 n 个结点的双向循环链表；

(2) 生着与死者的选择：

p 指向链表的第一个结点，初始 i 置为 1；

while(i<=n/2)　//删除一半的结点

{　从 p 指向的结点沿链前进 m-1 步；

删除第 m 个结点（q 所指向的结点）；

p 指向 q 的下一个结点；

输出其位置 q->data;

i 自增 1；

从 p 指向的结点沿链后退 k-1 步；

删除第 k 个结点（q 所指向的结点）；

p 指向 q 的上一个结点；

输出其位置 q->data;

i 自增 1；

}

(3) 输出所有生者的位置。

1.4.4 程序清单

```
//双向循环链表的类定义 dcirlinkl.h
typedef int ElemType;
//双向链表结点的类型定义
typedef struct DuLNode {
   ElemType    data;
   struct DuLNode *prior;//左指针
   struct DuLNode *next;//右指针
}DuLNode;
#define LEN 20

class DuLinkList
{private:
   DuLNode *head;//指向表头的指针
   DuLNode *curr;//当前结点指针
   int count;//  双向循环链表的结点个数
 public:
//构造函数
   DuLinkList();
//析构函数
   ~DuLinkList(){delete head;}
//创建有序或无序的带头结点的双向循环链表
   DuLNode *CreateCLinkL(int,int,int mark=0);
//清空单循环链表
   void ClearCList();
//求双向循环链表长度
   int CListSize();
//检查双向循环链表是否为空
   bool CListEmpty();
//返回指向第 pos 个结点的指针
   DuLNode *Index(int pos);
//返回双向循环链表中指定序号的结点值
   ElemType GetElem(int pos);
//遍历双向循环链表
```

```
    void TraverseCList();
//当前指针 curr 指向 pos 结点并返回 curr
    DuLNode *Reset(int pos=0);
//当前指针 curr 指向下一结点并返回
    DuLNode *Next();
//当前指针 curr 指向上一结点并返回
    DuLNode *Prior();
//  判双向循环链表当前指针 curr==head  否
    bool EndOCList();
//判双向循环链表当前指针 curr->next 是否到达表尾
    bool EndCList();
//判双向循环链表当前指针 curr->prior 是否到达表尾
    bool PEndCList();
//删除 curr->next 所指结点,并返回所删结点的 data
    ElemType DeleteNt();
//从双向循环链表中查找元素
    bool FindCList(ElemType& item);
//更新双向循环链表中的给定元素
    bool UpdateCList(const ElemType &item,ElemType &e);
//向链表中第 pos 个结点前插入域值为 item 的新结点
    void InsertCLfront(const ElemType& item,int pos);
//向链表中第 pos 个结点后插入域值为 item 的新结点
    void InsertCLafter(const ElemType& item,int pos);
//从链表中删除第 pos 个结点并返回被删结点的 data
    ElemType DeleteCList(int pos);
};

//双向循环链表的实现 dcirlinkl.cpp
#include<iostream.h>
#include<stdlib.h>
#include"dcirlinkl.h"
//构造函数
DuLinkList::DuLinkList()
{head=new DuLNode;
 head->prior=head;
```

```
  head->next=head;
  curr=NULL;
  count=0;
}
//创建有序或无序的带头结点的双向循环链表
DuLNode *DuLinkList::CreateCLinkL(int n,int m,int mark)
{ElemType x,a[LEN];
  srand(m);
  for(int i=0;i<n;i++) a[i]=rand()%100;
  for(i=0;i<n-1;i++)
    {int k=i;
      for(int j=i+1;j<n;j++)
        if(a[k]>a[j]) k=j;
      if(k!=i)
      {x=a[k];a[k]=a[i];a[i]=x;}}
  DuLNode *p;
  head=new DuLNode;
  head->prior=NULL;
  head->next=curr=p=new DuLNode;
  curr->prior=head;
  for(i=0;i<n;i++){
    if(mark==1) p->data=a[i];//升序
    else
      if(mark==-1) p->data=a[n-1-i];//降序
      else p->data=rand()%100;//无序
    if(i<n-1){curr=curr->next=new DuLNode;
      curr->prior=p;p=curr;}
    count++;}
  head->prior=curr;
  curr->next=head;
  return head;
}
//清空双向循环链表
void DuLinkList::ClearCList()
{DuLNode *cp,*np;
```

```
  cp=head->next;
  while(cp!=head)
  {np=cp->next;delete cp;cp=np;}
  head=NULL;
}
//求双向循环链表长度
int DuLinkList::CListSize()
{DuLNode* p=head->next;
  int i=0;
  while(p!=head)
    {i++;p=p->next;}
  return i;
}
//检查双向循环链表是否为空
bool DuLinkList::CListEmpty()
{return head->next==head;}
//返回指向第 pos 个结点的指针
DuLNode *DuLinkList::Index(int pos)
{if(pos<1)
    {cerr<<"pos is out range!"<<endl;exit(1);}
  DuLNode* p=head->next;
  int i=0;
  while(p!=head)
    {i++;
      if(i==pos) break;
      p=p->next;}
  if(p!=head) return p;
  else {cerr<<"pos is out range!"<<endl;
    return NULL;}
}
//返回双向循环链表中指定序号的结点值
ElemType DuLinkList::GetElem(int pos)
{if(pos<1)
  {cerr<<"pos is out range!"<<endl;exit(1);}
  DuLNode* p=head->next;
```

```
  int i=0;
  while(p!=head)
    {i++;
     if(i==pos) break;
     p=p->next;}
  if(p!=head) return p->data;
  else {cerr<<"pos is out range!"<<endl;
    return pos;}
}
//遍历双向循环链表
void DuLinkList::TraverseCList()
{DuLNode *p=head->next;
  while(p!=head)
    {cout<<setw(4)<<p->data;
     p=p->next;}
  cout<<endl;
}
//当前指针 curr 指向 pos 结点并返回 curr
DuLNode *DuLinkList::Reset(int pos)
{DuLNode* p=curr=head->next;
  int i=-1;
  while(p!=head)
    {i++;
     if(i==pos) break;
     p=p->next;curr=curr->next;}
  return curr;
}
//当前指针 curr 指向下一结点并返回
DuLNode *DuLinkList::Next()
{curr=curr->next;
  return curr;
}
//当前指针 curr 指向上一结点并返回
DuLNode *DuLinkList::Prior()
{curr=curr->prior;
```

```
  return curr;
}
// 判双向循环链表当前指针 curr==head 否
bool DuLinkList::EndOCList()
{return curr==head;}
//判双向循环链表当前指针 curr->next 是否到达表尾
bool DuLinkList::EndCList()
{return curr->next==head;}
//判双向循环链表当前指针 curr->prior 是否到达表尾
bool DuLinkList::PEndCList()
{return curr->prior==head;}
//删除 curr->next 所指结点,并返回所删结点的 data
ElemType DuLinkList::DeleteNt()
{DuLNode *p=curr->next;
  curr->next=p->next;
  curr->next->next->prior=p->prior;
  ElemType data=p->data;
  delete p;
  count--;
  return data;
}
//从双向循环链表中查找元素
bool DuLinkList::FindCList(ElemType& item)
{DuLNode* p=head->next;
  while(p!=head)
    if(p->data==item)
      {item=p->data;return true;}
    else p=p->next;
  return false;
}
//更新双向循环链表中的给定元素
bool DuLinkList::UpdateCList(const ElemType &item,ElemType &e)
{DuLNode* p=head->next;
  while(p!=head)    //查找元素
    if(p->data==item) break;
```

```
    else p=p->next;
  if(p==head) return false;
  else {    //更新元素
    p->data=e;return true;}
}
//向链表中第 pos 个结点前插入域值为 item 的新结点
void DuLinkList::InsertCLfront(const ElemType& item,int pos)
{DuLNode *newP=new DuLNode;
  newP->data=item;
  DuLNode* p=head->next;
  int i=0;
  while(p!=head)
    {i++;
      if(i==pos) break;
      p=p->next;}
  newP->prior=p->prior;
  p->prior->next=newP;
  newP->next=p;
  p->prior=newP;
  count++;
}
//向链表中第 pos 个结点后插入域值为 item 的新结点
void DuLinkList::InsertCLafter(const ElemType& item,int pos)
{DuLNode *newP=new DuLNode;
  newP->data=item;
  DuLNode* p=head->next;
  int i=-1;
  while(p!=head)
    {i++;
      if(i==pos) break;
      p=p->next;}
  newP->prior=p->prior;
  p->prior->next=newP;
  newP->next=p;
  p->prior=newP;
```

```
  count++;
}
//从链表中删除第 pos 个结点并返回被删结点的 data
ElemType DuLinkList::DeleteCList(int pos)
{if(pos<1)
   {cerr<<"pos is out range!"<<endl;exit(1);}
  DuLNode *p=head->next;
  ElemType data;
  int i=0;
  while(p!=head)
    {i++;
     if(i==pos) break;
     p=p->next;}
  if(p!=head)
    {data=p->data;
     p->prior->next=p->next;
     p->next->prior=p->prior;
     delete []p;count--;return data;}
  else return pos;
}

//双向循环链表的测试与应用 dcirlinklm.cpp
#include<iomanip.h>
#include "dcirlinkl.cpp"
void main()
{cout<<"dcirlinklm.cpp 运行结果:\n";
  int m=150,i,n=10,x,it;
  DuLinkList p,t,q,mylink;
  p.CreateCLinkL(n,m,1);
  if(p.CListEmpty()) cout<<"双向循环链表 p 空!\n";
  else cout<<"双向循环链表 p 非空!\n";
  cout<<"双向循环链表 p(升序):\n";
  p.TraverseCList();
  if(p.CListEmpty()) cout<<"双向循环链表 p 空!\n";
  else cout<<"双向循环链表 p 非空!\n";
```

```
if(p.EndCList()) cout<<"双向循环链表 p 满!\n";
else cout<<"双向循环链表 p 非满!\n";
cout<<"双向循环链表 t(无序):\n";
t.CreateCLinkL(n-2,m);
t.TraverseCList();
cout<<"双向循环链表 t 的长度:"<<t.CListSize()<<endl;
cout<<"双向循环链表 q(降序):\n";
q.CreateCLinkL(n,m,-1);
q.TraverseCList();
cout<<"双向循环链表 q 的长度:"<<q.CListSize()<<endl;
cout<<"链表 q 的第 1 个元素:"<<q.GetElem(1)<<endl;
cout<<"链表 q 的第 1 个元素地址:"<<q.Index(1)<<endl;
cout<<"链表 q 的第 5 个元素:"<<q.GetElem(5)<<endl;
cout<<"链表 q 的第 5 个元素地址:"<<q.Index(5)<<endl;
cout<<"链表 q 的第 10 个元素:"<<q.GetElem(10)<<endl;
cout<<"链表 q 的第 10 个元素地址:"<<q.Index(10)<<endl;
cout<<"链表 q 的 curr->next 所指元素地址:"<<q.Next()<<endl;
x=65;it=66;
if(q.FindCList(x)) cout<<x<<"查找成功!\n";
else cout<<x<<"查找不成功!\n";
if(q.UpdateCList(x,it)) cout<<x<<"更新成功!\n";
else cout<<x<<"更新不成功!\n";
cout<<"更新后双向循环链表 q:\n";
q.TraverseCList();
cout<<"插入后双向循环链表 q:\n";
it=100;q.InsertCLfront(it,1);
q.TraverseCList();
cout<<"插入后双向循环链表 q:\n";
it=101;q.InsertCLfront(it,5);
q.TraverseCList();
cout<<"插入后双向循环链表 q:\n";
it=102;q.InsertCLfront(it,13);
q.TraverseCList();
cout<<"插入后 q 表长:"<<q.CListSize()<<endl;
cout<<"第 1 个数:"<<q.DeleteCList(1)<<"删除成功!\n";
```

```
    cout<<"删除后 q 表长:"<<q.CListSize()<<endl;
    q.TraverseCList();
    cout<<"第 5 个数:"<<q.DeleteCList(5)<<"删除成功!\n";
    cout<<"删除后 q 表长:"<<q.CListSize()<<endl;
    q.TraverseCList();
    cout<<"第 11 个数:"<<q.DeleteCList(11)<<"删除成功!\n";
    cout<<"删除后 q 表长:"<<q.CListSize()<<endl;
    q.TraverseCList();
    cout<<"删除的数为:"<<q.DeleteNt()<<endl;
    cout<<"删除后 q 表长:"<<q.CListSize()<<endl;
    q.TraverseCList();
  cin.get();cin.get();}
```

1.4.5 运行结果

第二章 栈与队列

栈和队列是两种重要的线性结构。从数据结构角度上看，栈和队列也是线性表，其特殊性在于栈和队列的基本操作是线性表操作的子集，它们是受限的线性表，因此，可称为限定性的数据结构。但从数据类型角度看，它们是和线性表大不相同的两类重要的抽象数据类型。由于它们广泛应用在各种软件系统中，因此在面向对象的程序设计中，它们是多型数据类型。本章通过迷宫旅行游戏、八皇后问题和停车场管理三个项目来学习栈和队列的定义、表示方法和实现。

2.1 迷宫旅行游戏

2.1.1 项目简介

迷宫只有两个门，一个门叫入口，另一个门叫出口。一个骑士骑马从入口走进迷宫，迷宫中设置很多墙壁，对前进方向形成了多处障碍。骑士需要在迷宫中寻找通路以到达出口。

2.1.2 设计思路

迷宫问题的求解过程可以采用回溯法即在一定的约束条件下试探地搜索前进，若前进中受阻，则及时回头纠正错误另择通路继续搜索的方法。从入口出发，按某一方向向前探索，若能走通（未走过的），即某处可达，则到达新点，否则试探下一方向；若所有的方向均没有通路，则沿原路返回前一点，换下一个方向再继续试探，直到所有可能的通路都探索到，或找到一条通路，或无路可走又返回到入口点。

在求解过程中，为了保证在到达某一点后不能向前继续行走（无路）时，能正确返回前一点以便继续从下一个方向向前试探，则需要在试探过程中保存所能够到达的每一点的下标及从该点前进的方向，当找到出口时试探过程就结束了。

为了确保程序能够终止，调整时，必须保证曾被放弃过的填数序列不被再次试验，即要求按某种有序模型生成填数序列。给解的候选者设定一个被检验的顺序，按这个顺序逐一生成候选者并检验。

2.1.3 数据结构

迷宫问题是栈应用的一个典型例子。通过前面分析，我们知道在试探过程中为了能够沿着原路逆序回退，就需要一种数据结构来保存试探过程中曾走过的点的下标及从该点前进的方向，在不能继续走下去时就要退回前一点继续试探下一个方向，栈底元素是入口，栈顶元

素是回退的第一站，也即后走过的点先退回，先走过的点后退回，与栈的“后进选出，先进后出”特点一致，故在该问题求解的程序中可以采用栈这种数据结构。在迷宫有通路时，栈中保存的点逆序连起来就是一条迷宫的通路，否则栈中没有通路。

2.1.4 程序清单

程序提示：用二维数组表示二维迷宫中各个点是否有通路，在二维迷宫里面，从出发点开始，每个点按四邻域计算，按照右、上、左、下的顺序搜索一下落脚点，有路则走，无路即退回前点再从下一个方向搜索，即可构成一有序模型。栈用顺序结构实现。

```
//求解迷宫问题 maze.cpp
#include <iostream.h>
#include <stdlib.h>
#include <time.h>

enum Direction{DOWN,RIGHT,UP,LEFT};
const int ROWS=8,COLS=10;

void mazeTraversal(char [][COLS],const int,const int,int,int,int);
void mazeGenerator(char [][COLS],int *,int *);
void printMaze(const char[][COLS]);
bool validMove(const char [][COLS],int,int);
bool coordsAreEdge(int,int);

void main()
{cout<<"maze.cpp 运行结果:\n";
 cout<<"迷宫问题求解:\n";
 char maze[ROWS][COLS];
 int xStart,yStart,x,y;
 srand(time(0));
 for(int loop=0;loop<ROWS;++loop )
  for(int loop2=0;loop2<COLS;++loop2 )
       maze[loop][loop2]='#';
 mazeGenerator(maze,&xStart,&yStart);
 x=xStart;//开始行
 y=yStart;//开始列
```

```
 mazeTraversal(maze,xStart,yStart,x,y,RIGHT);
 printMaze(maze);cin.get();
}
void mazeTraversal(char maze[][COLS],const int xCoord,const int yCoord,int row,int col,int direction)
{static bool flag=false;//开始位置标志变量
 maze[row][col]='x';   //在当前位置插入 x
 if(coordsAreEdge(row,col)&&row!=xCoord&&col!=yCoord) {cout<<endl<<"成功走出迷宫!\n";return;}
 else if(row==xCoord&&col==yCoord&&flag) {
  cout<<"\n 返回迷宫开始位置.\n";return;}
 else {flag=true;
  for(int move=direction,count=0;count<4;++count,++move,move%=4)
   switch(move) {
     case DOWN://向下移动
       if(validMove(maze,row+1,col)) {
         mazeTraversal(maze,xCoord,yCoord,row+1,col,LEFT);
         return;}
       break;
     case RIGHT://向右移动
       if( validMove(maze,row,col+1)) {
         mazeTraversal(maze,xCoord,yCoord,row,col+1,DOWN);
         return;}
       break;
     case UP://向上移动
       if(validMove(maze,row-1,col)) {
         mazeTraversal(maze,xCoord,yCoord,row-1,col,RIGHT);
         return;}
       break;
     case LEFT://向左移动
       if(validMove(maze,row,col-1)) {
         mazeTraversal(maze,xCoord,yCoord,row,col-1,UP);
         return;}
       break;}
}}
```

```
//有效移动
bool validMove(const char maze[][COLS],int r,int c)
{return(r>=0&&r<=ROWS-1&&c>=0&&c<=COLS-1&&maze[r][c]!='#');}

bool coordsAreEdge( int x, int y )
{if((x==0||x==ROWS-1)&&(y>=0&&y<=COLS-1))
      return true;
 else if((y==0||y==COLS-1)&&(x>=0&&x<=ROWS-1))
      return true;
 else return false;
}
void printMaze(const char maze[][COLS] )
{for(int x=0;x<ROWS;++x) {
    for(int y=0;y<COLS;++y)
       cout<<maze[x][y]<<' ';
    cout<<'\n';}
 cout<<endl;
}
void mazeGenerator(char maze[][COLS],int *xPtr,int *yPtr)
{   int a,x,y,entry,exit;
    do {
        entry=rand()%4;
        exit=rand()%4;
    }while(entry==exit);
    // 确定入口位置
    if(entry==0) {
        *xPtr=1+rand()%(ROWS-2);//避免死角
        *yPtr=0;
        maze[*xPtr][*yPtr]='0';}
    else if(entry==1) {
        *xPtr=0;
        *yPtr=1+rand()%(COLS-2);
        maze[*xPtr][*yPtr]='0';}
    else if(entry==2) {
        *xPtr=1+rand()%(ROWS-2);
```

```
            *yPtr=COLS-1;
            maze[*xPtr][*yPtr]='0';}
        else {
            *xPtr=ROWS-1;
            *yPtr=1+rand()%(COLS-2);
            maze[*xPtr][*yPtr]='0';}
        //确定出口位置
        if(exit==0) {
            a=1+rand()%(ROWS-2);
            maze[a][0]='0';}
        else if(exit==1) {
            a=1+rand()%(COLS-2);
            maze[0][a]='0';}
        else if(exit==2) {
            a=1+rand()%(ROWS-2);
            maze[a][COLS-1]='0';}
        else {
            a=1+rand()%(COLS-2);
            maze[ROWS-1][a]='0';}
        for(int loop=1;loop<(ROWS-2)*(COLS-2);++loop) {
            x=1+rand()%(ROWS-2);//添加圆点到迷宫
            y=1+rand()%(COLS-2);
            maze[x][y]='0';}
    }
```

2.1.5 运行结果

迷宫问题求解:

返回迷宫开始位置.

0 # # # # # #

0 # # 0 0 0 # #

0 0 0 # # # x x

0 0 # # # x # x

x # x x # x x

###xxxxxxx#
###x#xxx##
##########

2.2 八皇后问题

2.2.1 项目简介

八皇后问题是一个古老而著名的问题，是回溯算法的典型例题。该问题是十九世纪著名的数学家高斯 1850 年提出：在 8×8 格的国际象棋棋盘上，安放八个皇后，要求没有一个皇后能够“吃掉”任何其他一个皇后，即任意两个皇后都不能处于同一行、同一列或同一条对角线上，求解有多少种摆法。

高斯认为有 76 种方案。1854 年在柏林的象棋杂志上不同的作者发表了 40 种不同的解，后来有人用图论的方法得出结论，有 92 种摆法。

2.2.2 设计思路

八皇后在棋盘上分布的各种可能的格局数目非常大，约等于 2^{32} 种，但是，可以将一些明显不满足问题要求的格局排除掉。由于任意两个皇后不能同行，即每一行只能放置一个皇后，因此将第 i 个皇后放置在第 i 行上。这样在放置第 i 个皇后时，只要考虑它与前 i-1 个皇后处于不同列和不同对角线位置上即可。

解决该问题采用回溯法。首先将第一个皇后放于第一行第一列，然后依次在下一行上放置下一个皇后，直到八个皇后全放置安全。在放置每一个皇后时，都依次对每一列进行检测，首先检测待第一列是否与已放置的皇后冲突，如不冲突，则将皇后放置在该列，否则，选择该行的下一列进行检没。如整行的八列都冲突，则回到上一行，重新选择位置，依次类推。

2.2.3 数据结构

八皇后问题是栈应用的另一个典型例子。通过前面分析，我们知道在对八个皇后的位置进行试探的过程中，可能遇到在某一行上的所有位置都不安全的情况，这时就要退回到上一行，重新摆放在上一行放置的皇后。为了能够对上一行的皇后继续寻找下一个安全位置，就必须记得该皇后目前所在的位置，即需要一种数据结构来保存从第一行开始的每一行上的皇后所在的位置。在放置进行不下去时，已经放置的皇后中后放置的皇后位置先被纠正，先放置的皇后后被纠正，与栈的“后进选出，先进后出”特点一致，故在该问题求解的程序中可以采用栈这种数据结构。在八个皇后都放置安全时，栈中保存的数据就是八个皇后在八行上的列位置。

2.2.4 程序清单

程序提示：用二维数组表示 8×8 格的国际象棋棋盘。栈用顺序结构实现。

```
#include<iostream.h>
#include<stdio.h>
int line[8],answer=0;
void show()//显示摆放的结果.
{
 int i,j;
 for(i=0;i<8;i++)
 {
  for(j=0;j<8;j++)
  {
   if(line[i]==j)
    cout<<"Q";
   else
    cout<<"*";
  }
  cout<<endl;
 }
     answer++;
 cout<<endl;
 cout<<answer<<endl;
 getchar();
}
int Judge(int t)//判断摆放的位置是否正确,不正确返回 1,正确返回 0.
{
 int i,n=0;
 for(i=0;i<t;i++)
 {
  if(line[i]==line[t])
  {n=1;break;}
  if(line[i]+i==line[t]+t)
  {n=1;break;}
```

```
 if(line[i]-i==line[t]-t)
  {n=1;break;}
 }
 return n;
}
void control(int n)//主要控制函数.
{
 int t=8;
 for(line[n]=0;line[n]<t;line[n]++)
 {
  if(Judge(n))
   continue;
  else
   if(n!=7)
    control(n+1);
   else
    show();
 }
}
int main()//主函数.
{
 control(0);
 cout<<answer<<endl;
 return 0;
}
```

2.2.5 运行结果

```
Q*******
****Q***
*******Q
*****Q**
**Q*****
******Q*
*Q******
```

Q*

1

2.3 停车场管理

2.3.1 项目简介

设停车场是一个可以停放 n 辆汽车的南北方向的狭长通道，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，最先到达的第一辆车停放在车场的最北端），若车场内已停满 n 辆车，那么后来的车只能在门外的便道上等候，一旦有车开走，则排在便道上的第一辆车即可开入；当停车场内某辆车要离开时，在它之后进入的车辆必须先退出车场为它让路，待该辆车开出大门外，其它车辆再按原次序进入车场，每辆停放在车场的车在它离开停车场时必须按它停留的时间长短交纳费用。试为停车场编制按上述要求进行管理的模拟程序。要求程序输出每辆车到达后的停车位置(停车场或便道上)，以及某辆车离开停车场时应缴纳的费用和它在停车场内停留的时间。

2.3.2 设计思路

停车场的管理流程如下：

①当车辆要进入停车场时，检查停车场是否已满，如果未满则车辆进入停车场；如果停车场已满，则车辆进入便道等候。

②当车辆要求出栈时，先让在它之后进入停车场的车辆退出停车场为它让路，再让该车退出停车场，让路的所有车辆再按其原来进入停车场的次序进入停车场。之后，再检查在便道上是否有车等候，有车则让最先等待的那辆车进入停车场。

2.3.3 数据结构

由于停车场只有一个大门，当停车场内某辆车要离开时，在它之后进入的车辆必须先退出车场为它让路，先进停车场的后退出，后进车场的先退出，符合栈的“后进先出，先进后出”的操作特点，因此，可以用一个栈来模拟停车场。而当停车场满后，继续来到的其它车辆只能停在便道上，根据便道停车的特点，先排队的车辆先离开便道进入停车场，符合队列的“先进先出，后进后出”的操作特点，因此，可以用一个队列来模拟便道。排在停车场中间的车辆可以提出离开停车场，并且停车场内在要离开的车辆之后到达的车辆都必须先离开停车场为它让路，然后这些车辆依原来到达停车场的次序进入停车场，因此在前面已设的一个栈和一个队列的基础上，还需要有一个地方保存为了让路离开停车场的车辆，由于先退出停车场的后进入停车场，所以很显然保存让路车辆的场地也应该用一个栈来模拟。因此，本题求解过程中需用到两个栈和一个队列。栈以顺序结构实现，队列以链表结构实现。

2.3.4 程序清单

程序提示：以栈模拟停车场，以队列模拟车场外的便道，按照从终端读入的输入数据序列进行模拟管理。每一组输入数据包括三个数据项：汽车“到达”或“离去”信息、汽车牌照号码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便道上的停车位置；若是车辆离去，则输出汽车在停车场内停留的时间和应交纳的费用（在便道上停留的时间不收费）。

```
#include<stdio.h>
#include <stdlib.h>
#include<iostream.h>
#include<string.h>
#include<math.h>
#define size 1          //停车场位置数
//模拟停车场的堆栈的性质；
typedef struct zanlind{
  int number;  //汽车车号
  int ar_time;  //汽车到达时间
}zanInode;
typedef struct{
  zanInode  *base;  //停车场的堆栈底
 zanInode  *top;  //停车场的堆栈顶
 int stacksize_curren;
}stackhead;
//堆栈的基本操作；
void initstack(stackhead &L)  //构造一个空栈
{
  L.base=(zanInode*)malloc(size*sizeof(zanlind));
  if(!L.base) exit(0);
  L.top=L.base;
  L.stacksize_curren=0;
}
void push(stackhead &L,zanInode e) //把元素 e 压入 s 栈
{
  *L.top++=e;
  L.stacksize_curren++;
```

```
}
void pop(stackhead &L,zanInode &e)   //把元素 e 弹出 s 栈
{
  if(L.top==L.base)
  {
     cout<<"停车场为空 !!";
     return;
  }
  e=*--L.top;
  L.stacksize_curren--;
}
//模拟便道的队列的性质；
typedef struct duilie{
  int number;    //汽车车号
  int ar_time;    //汽车到达时间
 struct duilie *next;
}*queueptr;
typedef struct{
  queueptr front;  //便道的队列的对头
  queueptr rear;  //便道的队列的队尾
  int length;
}linkqueue;
//队列的基本操作；
void initqueue(linkqueue &q) //构造一个空队列
{
  q.front=q.rear=(queueptr)malloc(sizeof(duilie));
  if(!q.front||!q.rear)
     exit(0);
  q.front->next=NULL;
  q.length=0;
}
void enqueue(linkqueue &q,int number,int ar_time) //把元素的插入队列（属性为 number，
ar_time）
{
  queueptr p;
```

```
  p=(queueptr)malloc(sizeof(duilie));
  if(!p) exit(0);
  p->number=number;
  p->ar_time=ar_time;
  p->next=NULL;
  q.rear->next=p;
  q.rear=p;
  q.length++;
}
void popqueue(linkqueue &q,queueptr &w) //把元素的插入队列（属性为 number，ar_time）
{
  queueptr p;
  if(q.front==q.rear)
  {
     cout<<"停车场的通道为空！!"<<endl;
     return;
  }
  p=q.front->next;
  w=p;
  q.front->next=p->next;
  q.length--;
  if(q.rear==p) q.front=q.rear;

}
void jinru(stackhead &st,linkqueue &q)   //对进入停车场的汽车的处理；
{
  int number,time_a;
  cout<<"车牌为：";
  cin>>number;
  cout<<"进场的时刻:";
  cin>>time_a;
  if(st.stacksize_curren<2)
  {
    zanInode e;
    e.number=number;
```

```
    e.ar_time=time_a;
   push(st,e);
    cout<< " 该车已进入停车场在: "<<st.stacksize_curren<<" 号车道"<<endl<<endl;
  }
  else
  {
   enqueue(q,number,time_a);
   cout<<"停车场已满，该车先停在便道的第"<<q.length<<"个位置上"<<endl;
   }

}

void likai(stackhead &st,stackhead &sl,linkqueue &q) //对离开的汽车的处理；
{                              //st 堆栈为停车场，sl 堆栈为倒车场
  int number,time_d,flag=1,money,arrivaltime;    //q 为便道队列
  cout<<"车牌为：";
  cin>>number;
  cout<<"出场的时刻：";
  cin>>time_d;
  zanInode e,q_to_s;
  queueptr w;
  while(flag)  //找到要开出的车，并弹出停车场栈
  {
 pop(st,e);
 push(sl,e);
   if(e.number==number)
   {
    flag=0;
     money=(time_d-e.ar_time)*2;
     arrivaltime=e.ar_time;
   }
  }
  pop(sl,e);  //把临时堆栈的第一辆车（要离开的）去掉；
  while(sl.stacksize_curren) //把倒车场的车倒回停车场
  {
```

```
    pop(sl,e);
    push(st,e);
  }
  if(st.stacksize_curren<2&&q.length!=0) //停车场有空位，便道上的车开进入停车场
  {
  popqueue(q,w);
  q_to_s.ar_time=time_d;
  q_to_s.number=w->number;
 push(st,q_to_s);
  cout<<" 车 牌 为 "<<q_to_s.number<<" 的 车 已 从 通 道 进 入 停 车 场 , 所 在 的 停 车 位 为:"<<st.stacksize_curren<<endl<<endl;
  }

 cout<<"\n          收      据"<<endl;
  cout<<"          ======================= 车牌号: "<<number<<endl;
  cout<<"=========================================================="<<endl;
  cout<<"|进车场时刻 | 出车场时刻 | 停留时间 | 应付（元）|"<<endl;
 cout<<"=========================================================="<<endl;
  cout<<"|  "<<arrivaltime<<"  |   "<<time_d<<"  |  "<<time_d-arrivaltime<<"   |   "<<m
oney<<"   |"<<endl;
  cout<<"----------------------------------------------------"<<endl<<endl;

}
void main()
{
int m=100;
char flag;          //进入或离开的标识；
stackhead sting,slinshi;  //停车场和临时倒车场堆栈的定义；
linkqueue   line;     //队列的定义；
initstack(sting);     //构造停车场堆栈 sting
initstack(slinshi);     //构造倒车场堆栈 slinshi
initqueue(line);      //构造便道队列 line
while(m)
{
   cout<<"\n              ** 停车场管理程序 **"<<endl;
```

```
    cout<<"=========================================================="<<endl;
    cout<<"**                          **"<<endl;
    cout<<"**   A --- 汽车 进 车场   D --- 汽车 出 车场   **"<<endl;
    cout<<"**                          **"<<endl;
    cout<<"**          E --- 退出   程序         **"<<endl;
    cout<<"=========================================================="<<endl;
    cout<<" 请选择 :(A,D,E): ";
    cin>>flag;
    switch(flag)
    {
    case 'A': jinru(sting,line);break;       //汽车进车场
    case 'D': likai(sting,slinshi,line);break; //汽车出车场
    case 'E': exit(0);
    }
    m--;
}
}
```

2.3.5 运行结果

第三章 串、数组和广义表

3.1 单词检索统计程序

3.1.1 项目简介

给定一个文本文件，要求统计给定单词在文本中出现的总次数，并检索输出某个单词出现在文本中的行号、在该行中出现的次数以及位置。

3.1.2 设计思路

本项目的设计要求可以分为三个部分实现：其一，建立一个文本文件，文件名由用户用键盘输入；其二，给定单词计数，输入一个不含空格的单词，统计输出该单词在文本中的出现次数；其三，检索给定单词，输入一个单词，检索并输出该单词所在的行号、该行中出现的次数以及在该行中的相应位置。

1. 建立文件的实现思路是：

(1) 定义一个串变量；

(2) 定义文本文件；

(3) 输入文件名，打开该文件；

(4) 循环读入文本行，写入文本文件，其过程如下：

```
while(不是文件输入结束){
    读入一文本行至串变量；
    串变量写入文件；
    输入是否结束输入标志；
}
```

(5) 关闭文件。

2. 给定单词计数的实现思路是：

该功能需要用到模式匹配算法，逐行扫描文本文件。匹配一个，计数器加 1，直到整个文件扫描结束；然后输出单词出现的次数。

串是非数值处理中的主要对象，如在信息检索、文本编辑、符号处理等许多领域，得到越来越广泛的应用。在串的基本操作中，在主串中查找模式串的模式匹配算法是文本处理中最常用、最重要的操作之一，称为模式匹配或串匹配，就是求子串在主串中首次出现的位置。朴素模式匹配算法的基本思路是将给定字串与主串从第一个字符开始比较，找到首次与子串完全匹配的子串为止，并记住该位置。但为了实现统计子串出现的个数，不仅需要从主串的第一个字符位置开始比较，而且需要从主串的任一位置检索匹配字符串。

其实现过程如下：

(1) 输入要检索的文本文件名，打开相应的文件；

(2) 输入要检索统计的单词；

(3) 循环读文本文件，读入一行，将其送入定义好的串中，并求该串的实际长度，调用串匹配函数进行计数。具体描述如下：

```
while(不是文件结束){
    读入一行并到串中；
    求出串长度；
    模式匹配函数计数；
}
```

(4) 关闭文件，输出统计结果。

3. 检索单词出现在文本文件中的行号、次数及其位置的实现思路是：

这个设计要求同上一个设计类似，但是要相对复杂一些。其实现过程描述如下：

(1) 输入要检索的文本文件名，打开相应的文件；

(2) 输入要检索统计的单词；

(3) 行计数器置初值 0；

(4)
```
while(不是文件结束){
    读入一行到指定串中；
    求出串长度；
    行单词计数器 0；
    调用模式匹配函数匹配单词定位、该行匹配单词计数；
    行号计数器加 1；
    if(行单词计数器!=0)输出行号、该行有匹配单词的个数以及相应的位置；
}
```

4. 主控菜单程序的结构

该部分内容如下：

(1) 头文件包含；

(2) 菜单选项包括：

1. 建立文件
2. 单词计数
3. 单词定位
4. 退出程序

(3) 选择 1-4 执行相应的操作，其他字符为非法。

3.1.3 数据结构

如果在程序设计语言中，串只是作为输入或输出的常量出现，则只需存储此串的串值，即字符序列即可.但在多数非数值处理的程序中，串也以变量的形式出现。串有三种机内表示方法：定长顺序存储表示、堆分配存储表示和串的块链存储表示。定长顺序存储表示类似于线性表的顺序存储结构，用一组地址连续的存储单元存储串值的字符序列。在串的定长顺序存储结构中，按照予定义的大小，为每个定义的串变量分配一个固定长度的存储区，则可用定长数组描述。

——串的定长顺序存储表示——

#define MAXSTRLEN 255 //用户可在 255 以内定义最大串长

typedef unsigned char SString [MAXSTRLEN+1]; //0 号单元存放串的长度

串的实际长度可在这预定义长度的范围内随意，超过于定义长度的串值则被舍去，称之为"截断"。对串长有两种表示方法:一是如上述定义描述的那样，以下标为 0 的数组分量存放串的实际长度，如 PASCAL 语言中的串类型采用这种表示方法;二是在串值后面加一个不计入串长的结束标记字符，如在有的 C 语言中以"\0"表示串值的终结。此时的串长为隐含值，显然不便于进行某些串操作。在这种存储结构表示时实现串求子串操作如下：

求子串 SubStrmg(&Sub,S,pos,len)过程即为复制字符序列的过程，将串 S 中从第 pos 个字符开始长度为 len 的字符序列复制到串 Sub 中。显然，本操作不会有需截断的情况，但有可能产生用户给出的参数不符合操作的初始条件，当参数非法时，返回 ERROR。其算法描述如算法 4.3 所示。

```
StatusSub String (Sstring &Sub,Sstring S,int pos,int len){
//用 Sub 返回串 s 第 pos 个字符起长度为 len 的子串。其中 1≤pos≤StrLength(S)且
0≤len≤StrLength(S)-pos+1.
if(poss[0] || lenS[0]-pos+1) return ERROR;
Sub[1..len] = S[pos..pos+len-1];
Sub[0] =len;
return OK;
} //End of SubString
```

由此可见，在顺序存储结构中，实现串操作的原操作为“字符序列的复制”，操作的时间复杂度基于复制的字符序列的长度。

本项目主要实现子串的检索和定位操作，所以用定长顺序存储表示比较简单易行。

3.1.4 程序清单

```
//字符串的模式匹配 Findstr.cpp
#include<iostream.h>
```

```
#include<iomanip.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define MAXSTRLEN 64

void GetNext(char T[MAXSTRLEN],int (&next)[64])
{int i,j;i=1;next[1]=j=0;
  while(i<(int)T[0])
  if(j==0||T[i]==T[j])
    {++i;++j;next[i]=j;}
  else j=next[j];
  for(j=1;j<(int)T[0];j++)
  {printf("next[%d]=%-3d",j,next[j]);
   if((j)%5==0) printf("\n");}
  cout<<endl;
}
void GetNext(char T[MAXSTRLEN],int (&next)[64],int m)
{int i,j;i=0;next[0]=-1;
  for(j=1;j<m;j++)
  {i=next[j-1];
  while(T[j]!=T[i+1]&&i>=0) i=next[i];
  if(T[j]==T[i+1]) next[j]=i+1;
  else next[j]=-1;}
  for(j=0;j<=m;j++)
    {printf("next[%d]=%-3d",j,next[j]);
     if((j+1)%5==0) printf("\n");}
  cout<<endl;
}
void GetNextVal(char T[MAXSTRLEN],int (&next)[64])
{int i,j;i=1;next[1]=j=0;
  while(i<(int)T[0])
  if(j==0||T[i]==T[j])
    {++i;++j;
     if(T[i]!=T[j]) next[i]=j;
```

```
      else next[i]=next[j];}
    else j=next[j];
  for(i=1;i<(int)T[0];i++)
    {printf("next[%d]=%-3d",i,next[i]);
      if(i%5==0) cout<<endl;}
  cout<<endl;
}
int IndexKMP(char S[MAXSTRLEN],char T[MAXSTRLEN],int (&next)[64])
{int i,j,n,m;i=j=1;
  n=(int)S[0];m=(int)T[0];
  while((i<n||j<m)&&T[j]!='\0'&&S[i]!='\0')
    if(j==0||S[i]==T[j]) {++i;++j;}
    else j=next[j];
  if(j>=m) return i+1-m;
  else return 0;
}
int IndexKMP(char S[MAXSTRLEN],char T[MAXSTRLEN],int (&next)[64],int pos)
{int i,j;i=pos;j=1;
  while((i<(int)S[0]||j<(int)T[0])&&T[j]!='\0'&&S[i]!='\0')
    if(j==0||S[i]==T[j]) {++i;++j;}
    else j=next[j];
  if(j>=(int)T[0]) return i+1-(int)T[0];
  else return 0;
}
int IndexKMP(char S[MAXSTRLEN],char T[MAXSTRLEN],int (&next)[64],int n,int m)
{int i,j;i=j=0;
  while(i<n&&j<m)
    if(S[i]==T[j]) {++i;++j;}
    else if(j==0) i++;
            else j=next[j-1]+1;
  if(j<m) return -1;
  else return i-m+1;
}
int IndexBF(char S[MAXSTRLEN],char T[MAXSTRLEN],int pos)
{int i,j;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;}
  if(j>T[0]) return i-T[0];
  else return 0;
}
void main()
{printf("Findstr.cpp 运行结果:\n");
  int Index,N,M,next[64]={0};
  char s[MAXSTRLEN],t[MAXSTRLEN];
  printf("GetNext-IndexKMP 的结果:\n");
  printf("输入主串 s:");gets(s);
  printf("输入模式串 t:");gets(t);
  N=strlen(s);M=strlen(t);
  printf("主串 s 长=%d\n",N);
  printf("  模式串 t 长=%d\n",M);
  GetNext(t,next,M);
  Index=IndexKMP(s,t,next,N,M);
  if(Index!=-1)
    printf("模式串在主串的位置从第%d 个字符开始\n",Index);
  else printf("主串 s 中不含模式串 t\n");

  printf("GetNext-IndexKMP 的结果:\n");
  s[0]=N;t[0]=M;
  GetNext(t,next);
  Index=IndexKMP(s,t,next,1);
  if(Index)
    printf("模式串在主串的位置从第%d 个字符开始\n",Index);
  else printf("主串 s 中不含模式串 t\n");

  printf("GetNextVal-IndexKMP 的结果:\n");
  GetNextVal(t,next);
  Index=IndexKMP(s,t,next,1);
  if(Index)
    printf("模式串在主串的位置从第%d 个字符开始\n",Index);
```

```
else printf("主串 s 中不含模式串 t\n");

printf("GetNext-IndexKMP 的结果:\n");
GetNext(t,next);
Index=IndexKMP(s,t,next);
if(Index)
   printf("模式串 t 在主串 s 中的位置从第%d 个字符开始\n",Index);
else printf("主串 s 中不含模式串 t\n");

printf("IndexBF 的结果:\n");
Index=IndexBF(s,t,1);
if(Index)
   printf("模式串 t 在主串 s 中的位置从第%d 个字符开始\n",Index);
else printf("主串 s 中不含模式串 t\n");
cin.get();}
```

3.1.5 运行结果

3.2 Internet 网络通路管理

3.2.1 项目简介

本项目是对 Interert 网络通路管理的简单模拟，以完成建立 Interert 网络通路、修改 Interert 网络通路信息和删除 Interert 网络通路信息等功能。

3.2.2 设计思路

本项目的实质是完成对 Interert 网络通路信息的建立、查找、插入、修改、删除等功能，可以首先定义项目的数据结构，然后将每个功能写成一个函数来完成对数据的操作，最后完成主函数以验证各个函数功能并得出运行结果。

3.2.3 数据结构

Interert 网络通路中的主机之间关系可以是任意的，任意两个站点之间都可能相关。而在图形结构中，结点之间的关系可以是任意的，图中任意两个数据元素之间都可能相关。所以

可以用图形结构来表示 n 个主机以及主机之间可能设置的 Interert 网络通路，其中网的顶点表示网络通路中的主机，边表示两个主机之间的网络通路，赋予边的权值表示相应的距离。

可以用一维数组存储图中顶点的信息，用矩阵表示图中各顶点之间的相邻关系。即用两个数组分别存储数据元素（顶点）的信息和数据元素之间的关系（边或弧）的信息。

假设图中顶点数为 n，网的邻接矩阵的定义为，当 v_i 到 v_j 有弧相邻接时，$a_{i,j}$ 的值应为该弧上的权值，否则为∞。将图的顶点信息存储在一个一维数组中，并将它的邻接矩阵存储在一个二维数组中即构成图的数组表示。

图的数组(邻接矩阵)存储表示

```
const INFINITY = INT_MAX;                // 最大值∞
const MAX_VERTEX_NUM = 20;               // 最大顶点个数
typedef enum {DG, DN, AG, AN} GraphKind;
                                         // 类型标志{有向图,有向网,无向图,无向网}
typedef struct ArcCell {                 // 弧的定义
  VRType adj;                            // VRType 是顶点关系类型。
                                         // 对无权图，用 1 或 0 表示相邻否；对带权图，则为权值类型。
  InfoType *info;                        // 该弧相关信息的指针
} AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
typedef struct {                         // 图的定义
  VertexType vexs[MAX_VERTEX_NUM]; // 顶点信息
  AdjMatrix arcs;                        // 表示顶点之间关系的二维数组
  int vexnum, arcnum;                    // 图的当前顶点数和弧(边)数
  GraphKind kind;                        // 图的种类标志
} MGraph;
```

3.2.4 程序清单

```
//图的相关数据类型的定义 graph.h
//最多顶点数
const int MaxV=10;
//最大权值
const int MaxValue=99;
//定义邻接表中的边结点类型
struct edgenode {
  int adjvex;     //邻接点域
  int weight;     //权值域
```

```
  edgenode* next;//指向下一个边结点的链域
};
//定义邻接表类型
typedef edgenode** adjlist;
//邻接矩阵类定义
class AdjMatrix
{private:
   char g[MaxV];//顶点信息数组
   int size;//当前顶点数
   int GA[MaxV][MaxV];//定义邻接矩阵 GA
   int numE;//当前边数
 public:
   //构造函数,初始化图的邻接矩阵
   AdjMatrix(int n,int k2);
   //判断图空否
   bool GraphEmpty() {return size==0;}
   //取当前顶点数
   int NumV() {return size;}
   //取当前边数
   int NumEdges() {return numE;}
   //取顶点 i 的值
   char GetValue(const int i);
   //取弧<v1,v2>的权
   int GetWeight(const int v1,const int v2);
   //在位置 pos 处插入顶点 V
   void InsertV(const char &V,int pos);
   //插入弧<v1,v2>,权为 weight
   void InsertEdge(const int v1,const int v2,int weight);
   //删除顶点 i 与顶点 i 相关的所有边
   char DeleteVE(const int i);
   //删除弧<v1,v2>
   void DeleteEdge(const int v1,const int v2);
   //建立图的邻接矩阵
   void CreateMatrix(int n, int k1,int k2);
   //k1 为 0 则无向否则为有向,k2 为 0 则无权否则为有权
```

```
    //从初始点 vi 出发深度优先搜索由邻接矩阵表示的图
    void dfsMatrix(bool*& visited,int i,int n,int k2);
    //从初始点 vi 出发广度优先搜索由邻接矩阵表示的图
    void bfsMatrix(bool*& visited,int i,int n,int k2);
    //由图的邻接矩阵得到图的邻接表
    void graphChange(adjlist &GL,int n,int k2);
    //检查输入的边序号是否越界,若越界则重输
    void Check(int n,int& i,int& j);
    //由图的邻接矩阵建立图
    void Creatgraph(int n,int k2);
    //对非连通图进行深度优先搜索
    void dfsMatrix(int n,int k2);
    //对非连通图进行广度优先搜索
    void bfsMatrix(int n,int k2);
};

//图的相关运算的实现 graph.cpp
#include"graph.h"
//构造函数,初始化图的邻接矩阵
AdjMatrix::AdjMatrix(int n,int k2)
{int i,j;
  if(k2==0){//初始化无(有)向无权图
    for(i=0;i<n;i++)
      for(j=0;j<n;j++)
        GA[i][j]=0;}
  else {//初始化无(有)向有权图
    for(i=0;i<n;i++)
      for(j=0;j<n;j++)
        if(i==j) GA[i][j]=0;
        else GA[i][j]=MaxValue;}
  size=numE=0;
}
//建立图的邻接矩阵
void AdjMatrix::CreateMatrix(int n,int k1,int k2)
//k1 为 0 则无向否则为有向,k2 为 0 则无权否则为有权
```

```
{int i,j,k,e,w;
 cout<<"输入图的总边数:";
    cin>>e;
 if(k1==0 && k2==0) { //建立无向无权图
  cout<<"输入"<<e<<"条无向无权边的起点和终点序号!"<<endl;
  for(k=1; k<=e; k++) {
   cin>>i>>j;
   Check(n,i,j);
   GA[i][j]=GA[j][i]=1;}
  }
 else if(k1==0 && k2!=0) { //建立无向有权图
   cout<<"输入"<<e<<"条无向带权边的起点和终点序号及权值!"<<endl;
   for(k=1; k<=e; k++) {
     cin>>i>>j>>w;
     Check(n,i,j);
     GA[i][j]=GA[j][i]=w;}
  }
  else if(k1!=0 && k2==0) { //建立有向无权图
     cout<<"输入"<<e<<"条有向无权边的起点和终点序号!"<<endl;
     for(k=1; k<=e; k++) {
       cin>>i>>j;
       Check(n,i,j);
       GA[i][j]=1;}
  }
  else if(k1!=0 && k2!=0) { //建立有向有权图
     cout<<"输入"<<e<<"条有向有权边的起点和终点序号及权值!"<<endl;
     for(k=1; k<=e; k++) {
       cin>>i>>j>>w;
       Check(n,i,j);
       GA[i][j]=w;}}
  numE=e;
  cout<<"创建后的邻接矩阵:\n";
  for(i=0;i<n;i++)
  {for(j=0;j<n;j++)
     cout<<setw(4)<<GA[i][j];
```

```
    cout<<endl;}
}
//从初始点 vi 出发深度优先搜索由邻接矩阵表示的图
void AdjMatrix::dfsMatrix(bool*& visited,int i,int n,int k2)
{cout<<g[i]<<':'<<i<<"    ";
 visited[i]=true;              //标记 vi 已被访问过
 for(int j=0; j<n; j++)    //依次搜索 vi 的每个邻接点
  if(k2==0)
   {if(i!=j&&GA[i][j]!=0&&!visited[j])
      dfsMatrix(visited,j,n,k2);}
  else
   if(i!=j&&GA[i][j]!=MaxValue&&!visited[j])
      dfsMatrix(visited,j,n,k2);
}
//从初始点 vi 出发广度优先搜索由邻接矩阵表示的图
void AdjMatrix::bfsMatrix(bool*& visited,int i,int n,int k2)
{const int MaxLength=30;
 //定义一个队列 q,其元素类型应为整型
 int q[MaxLength]={0};
 //定义队首和队尾指针
 int front=0,rear=0;
 //访问初始点 vi
 cout<<g[i]<<':'<<i<<"    ";
 //标记初始点 vi 已访问过
 visited[i]=true;
  //将已访问过的初始点序号 i 入队
 q[++rear]=i;
  //当队列非空时进行循环处理
 while(front!=rear) {
  //删除队首元素,第一次执行时 k 的值为 i
  front=(front+1)%MaxLength;
  int k=q[front];
  //依次搜索 vk 的每一个可能的邻接点
  for(int j=0;j<n;j++)
   if(k2==0)
```

```
        {if(k!=j&&GA[k][j]!=0&&!visited[j])
         {//访问一个未被访问过的邻接点 vj
          cout<<g[j]<<':'<<j<<"    ";
          visited[j]=true;         //标记 vj 已访问过
          rear=(rear+1)%MaxLength;//顶点序号 j 入队
          q[rear]=j;
         }
        }
      else
        if(k!=j&&GA[k][j]!=MaxValue&&!visited[j])
         {//访问一个未被访问过的邻接点 vj
          cout<<g[j]<<':'<<j<<"    ";
          visited[j]=true;     //标记 vj 已访问过
          rear=(rear+1)%MaxLength;//顶点序号 j 入队
          q[rear]=j;
         }
}}
//检查输入的边序号是否越界,若越界则重输
void AdjMatrix::Check(int n,int& i,int& j)
{while(1) {
   if(i<0||i>=n||j<0||j>=n)
      cout<<"输入有误,请重输!";
   else return;
   cin>>i>>j;
  }
}
//由图的邻接矩阵得到图的邻接表
void AdjMatrix::graphChange(adjlist &GL,int n,int k2)
{int i,j;
 if(k2==0)
 {for(i=0;i<n;i++){
  for(j=0;j<n;j++)
     if(GA[i][j]!=0) {
       edgenode* p=new edgenode;
       p->adjvex=j;
```

```
        p->next=GL[i];GL[i]=p;
        cout<<'('<<i<<','<<p->adjvex<<") ";}
    cout<<endl;}}
  else {
    for(i=0;i<n;i++){
      for(j=0;j<n;j++)
        if(GA[i][j]!=0 && GA[i][j]!=MaxValue) {
          edgenode* p=new edgenode;
          p->adjvex=j;p->weight=GA[i][j];
          p->next=GL[i];GL[i]=p;
          cout<<'('<<i<<','<<p->adjvex<<','<<p->weight<<") ";}
      cout<<endl;}
}}
//由图的邻接矩阵建立图
void AdjMatrix::Creatgraph(int n,int k2)
{int i,j,k,m=0;
  if(k2==0)
  {for(i=0;i<n;i++){
      k=i;
    for(j=0;j<n;j++)
        if(GA[i][j]!=0)
          if(k==i&&m<n)
            {g[m]='A'+m;size++;
             cout<<g[m]<<'('<<i<<','<<j<<") ";
             m++;
            }
    }
    cout<<endl;}
  else {
    for(i=0;i<n;i++){
      k=i;
      for(j=0;j<n;j++)
        if(GA[i][j]!=0 && GA[i][j]!=MaxValue)
          if(k==i&&m<n)
            {g[m]='A'+m;size++;
```

```
            cout<<g[m]<<'('<<i<<','<<j<<','<<GA[i][j]<<") ";
            m++;
          }
    }
  cout<<endl;}
  g[n]='\0';
}
//取顶点 i 的值
char AdjMatrix::GetValue(const int i)
{if(i<0||i>size)
    {cerr<<"参数 i 越界!\n";exit(1);}
  return g[i];
}
//取弧<v1,v2>的权
int AdjMatrix::GetWeight(const int v1,const int v2)
{if(v1<0||v1>size||v2<0||v2>size)
    {cerr<<"参数 v1 或 v2 越界!\n";exit(1);}
  return GA[v1][v2];
}
//在位置 pos 处插入顶点 V
void AdjMatrix::InsertV(const char &V,int pos)
{int i;
  if(size==MaxV)
    {cerr<<"表已满,无法插入!\n";exit(1);}
  if(pos<0||pos>size)
    {cerr<<"参数 pos 越界!\n";exit(1);}
  for(i=size;i>pos;i--) g[i]=g[i-1];
  g[pos]=V;
  size++;
}
//插入弧<v1,v2>,权为 weight
void AdjMatrix::InsertEdge(const int v1,const int v2,int weight)
{if(v1<0||v1>size||v2<0||v2>size)
    {cerr<<"参数 v1 或 v2 越界!\n";exit(1);}
  GA[v1][v2]=weight;
```

```
  numE++;
}
//删除顶点 v 与顶点 v 相关的所有边
char AdjMatrix::DeleteVE(const int v)
{for(int i=0;i<size;i++)
   for(int j=0;j<size;j++)
     if((i==v||j==v)&&GA[i][j]>0&&GA[i][j]<MaxValue)
       {GA[i][j]=MaxValue;
        numE--;}
  if(size==0)
    {cerr<<"表已空,无元素可删!\n";exit(1);}
  if(v<0||v>size-1)
    {cerr<<"参数 v 越界!\n";exit(1);}
  char temp=g[v];
  for(i=v;i<size-1;i++) g[i]=g[i+1];
  size--;
  g[size]='\0';
  return temp;
}
//删除弧<v1,v2>
void AdjMatrix::DeleteEdge(const int v1,const int v2)
{if(v1<0||v1>size||v2<0||v2>size||v1==v2)
    {cerr<<"参数 v1 或 v2 出错!\n";exit(1);}
  GA[v1][v2]=MaxValue;
  numE--;
}
//对非连通图进行深度优先搜索
void AdjMatrix::dfsMatrix(int n,int k2)
{bool *vis=new bool[NumV()];
  for(int i=0;i<NumV();i++) vis[i]=false;
  for(i=0;i<NumV();i++)
    if(!vis[i]) dfsMatrix(vis,i,n,k2);
  delete []vis;
}
//对非连通图进行广度优先搜索
```

```
void AdjMatrix::bfsMatrix(int n,int k2)
{bool *vis=new bool[NumV()];
  for(int i=0;i<NumV();i++) vis[i]=false;
  for(i=0;i<NumV();i++)
    if(!vis[i]) bfsMatrix(vis,i,n,k2);
  delete []vis;
}


//图的相关运算的测试 graphM.cpp
#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
#include "graph.cpp"
void main()
{cout<<"graphM.cpp 运行结果:\n";
  //定义图的点数及搜索起始点序号等
  int n,k,i;
  //k1 为 0 则无向否则为有向,k2 为 0 则无权否则为有权
  int k1,k2;
  //标记已访问过的点
  bool *vis;
  //定义邻接表
  adjlist AL;
  cout<<"输入图的点数 n=";cin>>n;
  AL=new edgenode*[n];
  vis=new bool[n];
  if(!vis) {cout<<"申请堆内存失败!\n";exit(1);}
  for(i=0;i<n;i++)
    vis[i]=false;
  cout<<"输入选择无向(权)与有向(权)图的值 k1,k2:";
  cin>>k1>>k2;
  //定义邻接矩阵
  AdjMatrix A(n,k2);
  A.CreateMatrix(n,k1,k2);
```

```
 cout<<"出发点 Vk 的序号=";cin>>k;
 cout<<"\n 输出邻接矩阵相应图的每个顶点:\n";
 A.Creatgraph(n,k2);
 cout<<"当前的顶点数为:"<<A.NumV()<<endl;
 cout<<"当前的边数为:"<<A.NumEdges()<<endl;

 cout<<"图的深度优先搜索顺序:\n";
 A.dfsMatrix(vis,k,n,k2);
 for(i=0;i<n;i++) vis[i]=false;
 cout<<"\n 图的广度优先搜索顺序:\n";
 A.bfsMatrix(vis,k,n,k2);

 cout<<"\n 输出邻接表的每个邻接点:\n";
 for(i=0;i<n;i++) vis[i]=false;
 A.graphChange(AL,n,k2);
 delete []vis;

 A.DeleteEdge(0,2);
 A.DeleteEdge(2,0);

 cout<<"当前的顶点数为:"<<A.NumV()<<endl;
 cout<<"当前的边数为:"<<A.NumEdges()<<endl;
 cout<<"图的深度优先搜索顺序:\n";
 A.dfsMatrix(n,k2);
 cout<<"\n 图的广度优先搜索顺序:\n";
 A.bfsMatrix(n,k2);
 cin.get();cin.get();
}
```

3.2.5 运行结果

graphM.cpp 运行结果:

```
输入图的点数 n=7
输入选择无向(权)与有向(权)图的值 k1,k2:0 1
输入图的总边数:12
```

输入 12 条无向带权边的起点和终点序号及权值!

0 1 1 0 2 1 1 3 1 1 4 1 2 5 1 2 6 1

1 0 1 2 0 1 3 1 1 4 1 1 5 2 1 6 2 1

创建后的邻接矩阵:

```
  0   1   1  99  99  99  99
  1   0  99   1   1  99  99
  1  99   0  99  99   1   1
 99   1  99   0  99  99  99
 99   1  99  99   0  99  99
 99  99   1  99  99   0  99
 99  99   1  99  99  99   0
```

出发点 Vk 的序号=0

输出邻接矩阵相应图的每个顶点:

A(0,1,1) B(0,2,1) C(1,0,1) D(1,3,1) E(1,4,1) F(2,0,1) G(2,5,1)

当前的顶点数为:7

当前的边数为:12

图的深度优先搜索顺序:

A:0 B:1 D:3 E:4 C:2 F:5 G:6

图的广度优先搜索顺序:

A:0 B:1 C:2 D:3 E:4 F:5 G:6

输出邻接表的每个邻接点:

(0,1,1) (0,2,1)

(1,0,1) (1,3,1) (1,4,1)

(2,0,1) (2,5,1) (2,6,1)

(3,1,1)

(4,1,1)

(5,2,1)

(6,2,1)

当前的顶点数为:7

当前的边数为:10

图的深度优先搜索顺序:

A:0 B:1 D:3 E:4 C:2 F:5 G:6

图的广度优先搜索顺序:

A:0 B:1 D:3 E:4 C:2 F:5 G:6

第四章 树和二叉树

树型结构是一类重要的非线性数据结构。其中以树和二叉树最为常用，直观看来，树是以分支关系定义的层次结构。树结构在客观世界中广泛存在，如人类社会的族谱和各种社会组织机构都可以用树来形象表示。树在计算机领域中也得到广泛应用，如在编译程序中，可用树来表示源程序的语法结构。又如在数据库系统中，树型结构也是信息的重要组织形式之一。本章通过三个模拟项目来学习树及二叉树的存储结构及其各种操作，首先通过使用有关树的操作实现家谱管理，其次通过使用有关线索二叉树的操作实现图书目录分类管理，最后通过哈夫曼树的应用实现图像压缩编码优化。

4.1 家谱管理

4.1.1 项目简介

家谱（或称族谱）是一种以表谱形式，记载一个以血缘关系为主体的家族世系繁衍和重要人物事迹的特殊图书体裁。家谱是中国特有的文化遗产，是中华民族的三大文献（国史，地志，族谱）之一，属珍贵的人文资料，对于历史学、民俗学、人口学、社会学和经济学的深入研究，均有其不可替代的独特功能。本项目对家谱管理进行简单的模拟，以实现查看祖先和子孙个人信息、插入家族成员、删除家族成员等功能。

4.1.2 设计思路

本项目的实质是完成对家谱成员信息的建立、查找、插入、修改、删除等功能，可以首先定义家族成员的数据结构，然后将每个功能写成一个函数来完成对数据的操作，最后完成主函数以验证各个函数功能并得出运行结果。

4.1.3 数据结构

因为家族中的成员之间存在一个对多个的层次结构关系，所以不能用上面讲的线性表来表示。家谱从形状上看像一颗倒长的树，所以用树结构来表示家谱比较适合。树型结构是一类非常重要的非线性数据结构，直观看来，树是以分支关系定义的层次结构。

可以二叉链表作为树的存储结构，链表中的两个链域分别指向该结点的第一个孩子结点和下一个兄弟结点，固该表示法又称二叉树表示法，或孩子兄弟表示法。孩子兄弟表示法是一种链式存储结构，它通过描述每个结点的一个孩子和兄弟信息来反映结点之间的层次关系，其具体的结点结构为：

firstChild	data	nextSibling

其中，firstchild 为指向该结点第一个孩子的指针，nextsibling 为指向该结点下一个兄弟，elem 是数据元素内容，举例如下：

其存储形式定义如下：

```
typedef struct CSLinklist{
    Elemtype data;
    struct CSLinklist *firstchild,*nextsibling;
} CSLinklist,*CSTree;
```

4.1.4 程序清单

```
//树的孩子兄弟表示法为存储结构的结构体 Tree.h
template<class T> class Tree;
template<class T> struct TreeNode
{friend class Tree<T>;//树类为友元
 private:
   TreeNode<T> *firstChild;//第一个孩子结点指针域
   TreeNode<T> *nextSibling;//下一个兄弟结点指针域
 public:
   T data;//数据域
//构造函数
   TreeNode(T value,TreeNode<T> *fc=NULL,
      TreeNode<T> *ns=NULL):data(value),
        firstChild(fc),nextSibling(ns){}
//访问指针域的成员函数
   TreeNode<T>* &FirstChild()
     {return firstChild;}
   TreeNode<T>* &NextSibling()
     {return nextSibling;}
};
//树类
template<class T> class Tree
{private:
   TreeNode<T> *root;//根结点指针
```

```
  TreeNode<T> *curr;//当前结点指针
  //显示以 t 为先根结点的树的数据域
  void PreOrderTree(TreeNode<T> *&t);
  //显示以 t 为后根结点的树的数据域
  void PosOrderTree(TreeNode<T> *&t);
  //使当前结点为 t 所指结点
  int Current(TreeNode<T> *&t);
  //在树 root 中回溯查找结点 s 的双亲结点
  TreeNode<T> *SearchParent(TreeNode<T> *&root,TreeNode<T> *&s);
 public:
  //构造函数与析构函数
  Tree(){root=curr=NULL;}
  ~Tree(){DeleteSubTree(root);}
  //使根结点为当前结点
  int Root();
  //使当前结点的双亲结点为当前结点
  int Parent();
  //使当前结点的第一个孩子结点为当前结点
  int FirstChild();
  //使当前结点的兄弟结点为当前结点
  int NextSibling();
  //把 valve 插入到当前结点的最后一个结点
  void InsertChild(T value);
  //删除以 t 为根结点的子树
  void DeleteSubTree(TreeNode<T> *&t);
  //删除当前结点的第 i 个孩子结点
  int DeleteChild(int i);
  //删除以 root 为根结点的子树的第 i 个孩子结点
  int DeleteChild1(int i);
  //按先根遍历次序显示树的数据域值
  void DisplayTree();
  //按后根遍历次序显示树的数据域值
  void DisplayTree1();
};
```

```
   //树类的实现 Tree.cpp
template<class T>
void Tree<T>::DeleteSubTree(TreeNode<T> *&t)
{if(t==NULL) return;
 TreeNode<T> *q=t->firstChild,*p;
 while(q!=NULL)
 {p=q->nextSibling;
  DeleteSubTree(q);
  q=p;}
  printf("释放:%2c",t->data);
  delete t;
}
template<class T>
int Tree<T>::Current(TreeNode<T> *&t)
{if(t==NULL) return 0;
 curr=t;
 return 1;
}
template<class T>
int Tree<T>::Root()
{if(root==NULL)
   {curr=NULL;
    return 0;}
 return Current(root);
}
template<class T>
int Tree<T>::FirstChild()
{if(curr!=NULL&&curr->firstChild!=NULL)
   return Current(curr->firstChild);
 else return 0;
}
template<class T>
int Tree<T>::NextSibling()
{if(curr!=NULL&&curr->nextSibling!=NULL)
   return Current(curr->nextSibling);
```

```
  else return 0;
}
template<class T>
int Tree<T>::Parent()
{if(curr==NULL)
    {curr=root;
     return 0;}
  TreeNode<T> *p=SearchParent(root,curr);
  if(p==NULL) return 0;
  else return Current(p);
}
template<class T>
TreeNode<T> *Tree<T>::SearchParent(TreeNode<T> *&root,TreeNode<T> *&s)
{if(root==NULL) return NULL;
  if(root->FirstChild()==s||root->NextSibling()==s)
    return root;
  TreeNode<T> *p;
  if((p=SearchParent(root->FirstChild(),s))!=NULL) return p;
  if((p=SearchParent(root->NextSibling(),s))!=NULL) return p;
  return NULL;
}
template<class T>
void Tree<T>::InsertChild(T value)
{TreeNode<T> *newNode=new TreeNode<T> (value);
  if(root==NULL)    //当为空树时
    {root=curr=newNode;
     return;}
  if(curr->firstChild==NULL)//当当前结点无孩子时
    curr->firstChild=newNode;
  else                              //当当前结点有孩子时
    {TreeNode<T> *p=curr->firstChild;
     while(p->nextSibling!=NULL) p=p->nextSibling;
     p->nextSibling=newNode;
    }
    Current(newNode);//使新建立的结点成为当前结点
```

```
}
template<class T>
int Tree<T>::DeleteChild(int i)
{TreeNode<T> *r=NULL;
 if(i==1)                //当删除当前结点的第一棵子树时
 {r=curr->firstChild;
  if(r==NULL) return 0;//要删除子树为空时返回
  curr->firstChild=r->nextSibling;//脱链要删除的子树
 }
 else {                  //当删除当前结点的其他子树时
  int k=1;
  TreeNode<T> *p=curr->firstChild;
  while(p!=NULL&&k<=i-1)//寻找要删除子树的指针
  {p=p->nextSibling;
   k++;}
  if(p!=NULL)//寻找到要删除的子树的指针
  {r=p->nextSibling;
   if(r!=NULL)
     p->nextSibling=r->nextSibling;
   else return 0;
  }
  else return 0;
 }
 DeleteSubTree(r);
 return 1;
}
template<class T>
int Tree<T>::DeleteChild1(int i)
{if(root==NULL) return 0;//当为空树时
 TreeNode<T> *r=NULL,*q=root->firstChild;
 if(i==1&&q!=NULL) //当第一结点有孩子时
   {r=root->firstChild;
    root->firstChild=r->nextSibling;//脱链要删除的子树
   }
 else                    //要删除第一结点外的其他子树时
```

```
    {int k=1;
     TreeNode<T> *p=root->firstChild;
     while(p!=NULL&&k<=i-1)//寻找要删除子树的指针
     {p=p->nextSibling;
      k++;
     }
    if(p!=NULL)       //寻找到要删除的子树的指针
     {r=p->nextSibling;
      if(r!=NULL)
        p->nextSibling=r->nextSibling;//脱链要删除的子树
      else return 0;}
    else return 0;
  }
  DeleteSubTree(r);//调用函数执行删除
  return 1;
}
template<class T>
void Tree<T>::PreOrderTree(TreeNode<T> *&t)
{if(t==NULL) return;
  cout<<setw(2)<<t->data;//显示根结点数据
  if(t->firstChild!=NULL)//先根遍历子树
    PreOrderTree(t->firstChild);
  if(t->nextSibling!=NULL)
    PreOrderTree(t->nextSibling);
}
template<class T>
void Tree<T>::DisplayTree()
{PreOrderTree(root);}

template<class T>
void Tree<T>::DisplayTree1()
{PosOrderTree(root);}

template<class T>
void Tree<T>::PosOrderTree(TreeNode<T> *&t)
```

```
{if(t==NULL) return;
 if(t->firstChild!=NULL)//后根遍历子树
   PosOrderTree(t->firstChild);
 printf("%2c",t->data);//显示根结点数据
 if(t->nextSibling!=NULL)
   PosOrderTree(t->nextSibling);
}
//树类相关操作的测试 TreeM.cpp
#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
#include<conio.h>
#include<stdio.h>
#include "Tree.h"
#include "Tree.cpp"
void main()
{printf("TreeM.cpp 运行结果:\n");
 int i;
 Tree<char> t;
 t.InsertChild('A');
 for(i=0;i<7;i++)
 {t.Root();
  if(i>=3&&i<5) t.FirstChild();
  t.InsertChild('B'+i);
 }
 printf("按后根遍历显示的结点次序为:\n");
 t.DisplayTree1();
 int k;
 printf("\n 输入欲删除第几个结点(k):");scanf("%d",&k);
 if(t.DeleteChild1(k))
    printf("\n 第%d 个孩子结点,删除成功!\n",k);
 else printf("第%d 个孩子结点,删除失败!\n",k);
 printf("按先根遍历显示的结点次序为:\n");
 t.DisplayTree();
 cin.get();
```

```
  printf("\n 析构函数按后根遍历释放结点的次序为:\n");
  cin.get();
}
```

4.1.5 运行结果

```
按后根遍历显示的结点次序为:
 E F B C D G H A
输入欲删除第几个结点(k):3
释放: G
第 3 个孩子结点,删除成功!
按先根遍历显示的结点次序为:
 A B E F C D H
```

4.2 表达式求值问题

4.2.1 项目简介

表达式求值是程序设计语言编译中的一个最基本问题，就是将一个表达式转化为逆波兰表达式并求值。具体要求是以字符序列的形式从终端输入语法正确的、不含变量的整数表达式，并利用给定的优先关系实现对算术四则混合表达式的求值，并演示在求值过程中运算符栈、操作数栈、输入字符和主要操作变化过程。

要把一个表达式翻译成正确求值的一个机器指令序列，或者直接对表达式求值，首先要能正确解释表达式。任何一个表达式都是由操作符（operand）、运算符（operator）和界限符（delimiter）组成，我们成它们为单词。一般的，操作数既可以是常数，也可以是被说明为变量或常量的标识符；运算符可以分为算术运算符、关系运算符和逻辑运算符 3 类；基本界限符有左右括号和表达式结束符等。为了叙述的简洁，我们仅仅讨论简单算术表达式的求值问题。这种表达式只包括加、减、乘、除 4 种运算符。

人们在书写表达式时通常采用的是“中缀”表达形式，也就是将运算符放在两个操作数中间，用这种“中缀”形式表示的表达式称为中缀表达式。但是，这种表达式表示形式对计算机处理来说是不大合适的。对于表达式的表示还有另一种形式，称之为“后缀表达式”，即将运算符紧跟在两个操作数的后面。这种表达式比较适合计算机的处理方式，因此要用计算机来处理、计算表达式的问题，首先要将中缀表达式转化成后缀表达式，又成为逆波兰表达式。

4.2.2 设计思路

为了实现表达式求值，可以首先读入原表达式（包括括号）并创建对应二叉树，其次对二叉树进行前序遍历、中序遍历、后续遍历（非递归），并输出逆波兰表达式，最后求解原表达式的值，同时对非法表达式格式能予以判断。

设 str1=（a+b)*((c+d)*e+f*h*g)；str2 是目的串。先用一个堆栈存储 operand，遇到数字就不做处理直接放到目的串中，遇到 operaor 则将其与堆栈顶元素比较优先级，决定是否送到输出串中，这样处理之后能得到一个后续的排列，再建立二叉树，当遇到 ab+这样的建立左右接点为 a 和 b 的根为+的树并返回根接点，当遇到操作符前面只有一个操作数的如 c*这样的情况就建立右子接点为 c 的根接点为*的，左结点为上次操作返回的子树的根接点（这种情况下就是合并树的情况）。

非递归方法进行树的遍历需要用到栈结构。首先调用 stack 的 ADT 中的 creatEmptyStack() 函数来建立一个空栈（注意，栈元素的类型是指向树结点的指针）。

首先将根节点压入栈中。然后进入循环。

如果栈不空，

弹出栈顶元素，

进行访问，

压入右子树（节点指针），

压入左子树（节点指针），

循环。

由于需要同时用到树和栈两种抽象数据类型并且需要进行互动操作，在头文件中需要加入#ifndef#define#endif 宏命令来避免出现重复定义的问题。

4.2.3 数据结构

用二叉树的结构来存储表达式，后续遍历二叉树即可得到逆波兰表达式，也可以通过树的合并操作完成表达式的计算。表达式二叉树结点类型如下。

```
typedef    struct nodeTag
union{
   int opnd;
  char optr;
}val;
struct    nodeTag *left;
struct    nodeTag *right;
}treeNode;
```

4.2.4 程序清单

```
//Tree.h
#ifndef _TREE_H_
#define _TREE_H_ //防止反复包含

#include<iostream>
#include<string>
#include<math.h>
#include<sstream>
using namespace std;
/*********************************************************/
bool IsOperator(string mystring) //验证操作符
{
if(mystring=="-"||mystring==" "||mystring=="/"||mystring=="*"||mystring=="^")
return(true);
else
return(false);
}
bool IsOperator(char ops)//重载
{
if(ops==' '||ops=='-'||ops=='*'||ops=='/'||ops=='^'||ops=='('||ops==')')
return(true);
else
return(false);
}
/*********************************************************/
class node_type//结点类
{
public:
string data;
node_type *left_child;
node_type *right_child;
node_type(string k)
{
```

```
data=k;
left_child=NULL;
right_child=NULL;
}
};
/*************************************************************/
class binary_tree //二叉树类开始
{
public:
node_type *root;
void print(node_type *r);
binary_tree(void){root=NULL;}
void inprint(node_type *p)
{
if(p!=NULL)
{

inprint(p->left_child);
cout<<p->data<<" ";
inprint(p->right_child);
}
}
void deprint(node_type *p)
{
if(p!=NULL)
{

deprint(p->right_child);
cout<<p->data<<" ";
deprint(p->left_child);
}
}
void print(void){print(root);}
void inprint(void){print(root);}
void evaluate()
```

```
{
evaluate(root);
}
void evaluate(node_type *prt)
{
if(IsOperator(prt->data)&&!IsOperator(prt->left_child->data)&&!IsOperator(prt-
>right_child->data))
{
int num=0;
int num1=atoi(prt->left_child->data.c_str()); //C 类型的字符串  常量不可改变
int num2=atoi(prt->right_child->data.c_str()); //同上
if(prt->data==" ")
num=num1 num2;
else if(prt->data=="-")
num=num1-num2;
else if(prt->data=="*")
num=num1*num2;
else if(prt->data=="/")
num=num1/num2;
else if(prt->data=="^")
num=pow(num1,num2);
cout<<num<<'\t'; //打印中间结果
stringstream bob;
bob<<num;
string suzzy(bob.str());
prt->data=suzzy;
prt->left_child=NULL;
prt->right_child=NULL;

}
else if(prt->left_child==NULL&&prt->right_child==NULL);
else
{
evaluate(prt->left_child);
evaluate(prt->right_child);
```

```
evaluate(prt);
}
}
void clear_help(node_type *rt) //删除者
{
if(rt!=NULL)
{
clear_help(rt->left_child);
clear_help(rt->right_child);
delete rt;
}
}
void clear() //删除整个二叉树。
{
clear_help(root);
}
/*************************************************************/
void count_help(node_type *rt,int &a) //计数者//引用可以带回改变值。
{
a=0;
int f1,f2;
if(rt!=NULL)
{
count_help(rt->left_child,f1);
count_help(rt->right_child,f2);
a=f1 f2 1;
}
}
int counter() //返回结点的个数
{
int a;
count_help(root,a);
return a;
}
bool compare(string str1,string str2) //比较两结点的大小。若 str1 大于 str2 返回 TRUE。否
```

```
则为 FALSE。
{
int i=0;
while(str1[i]&&str2[i])
{
if((int)str1[i]>(int)str2[i])
return true;
else if((int)str1[i]<(int)str2[i])
return false;
else i ;
}
if(str1[i])return true;
else return false;
}
/*************************************************************/
void inserter(node_type *&p,string data) //插入者。
{
if(p==NULL)
{
p=new node_type(data);
p->left_child=NULL;
p->right_child=NULL;
}
else if(compare(p->data,data))
inserter(p->left_child,data);
else inserter(p->right_child,data);
}

void found(node_type *&p,node_type *&p1) //对一个树排序，并建新树。依靠插入者。
{
if(p!=NULL)
{
found(p->left_child,p1);
found(p->right_child,p1);
inserter(p1,p->data);
```

```
}
}
binary_tree order(binary_tree aa) //排序的函数
{
binary_tree bb;
found(aa.root,bb.root);
return bb;
}
binary_tree order()
{
return order(*this);
}
}; //二叉树类结束。


node_type *build_node(string x) //建立一个结点
{
node_type *new_node;
new_node=new node_type(x);
return(new_node);
}
void binary_tree::print(node_type *p)//打印
{
if(p!=NULL)
{
print(p->left_child);
print(p->right_child);
cout<<p->data<<" ";
}
}
bool IsOperand(char ch)//验证数据
{
if((ch>='0')&&(ch<='9'))
return true;
else
```

```
return false;
}
/*********************************************************/
bool addition(char OperatorA,char OperatorB) //A=B 返回 TRUE.
{
if(OperatorA==OperatorB||(OperatorA=='*'&&OperatorB=='/')||(OperatorA=='/'&&OperatorB
=='*')||(OperatorA==' '&&OperatorB=='-')||(OperatorA=='-'&&OperatorB==' '))
return true;
else return false;
}

bool TakesPrecedence(char OperatorA,char OperatorB) //判别符号的优先级。A>B，返回为
TRUE。
{
if(OperatorA=='(')
return false;
else if(OperatorB=='(')
return false;
else if(OperatorB==')')
return true;
else if(addition(OperatorA,OperatorB))
return false;
else if(OperatorA=='^')
return true;
else if(OperatorB=='^')
return false;
else if((OperatorA=='*')||(OperatorA=='/'))
return true;
else if((OperatorB=='*')||(OperatorB=='/'))
return false;
else if((OperatorA==' ')||(OperatorA=='-'))
return true;
else return false;
}
/*********************************************************/
```

```
void copy(node_type *&r1,node_type *r2) //拷贝整个二叉树
{
if(r2==NULL)
r1=NULL;
else
{
r1=build_node(r2->data);
copy(r1->left_child,r2->left_child);
copy(r1->right_child,r2->right_child);
}
}
#endif //条件编绎结束


/************************ main.cpp************************************/
#include<iostream>
#include<string>
#include<stack>
#include "Tree.h"
/*************************** Checks if expression if ok ********************************/
bool isok(string exp) //此函数验证式子是否正确，即是否符合运算规则。
{
char check;
int error=0;
int lb=0;
int rb=0;
if(exp.size()==1)return false;
else      if((IsOperator(exp[0])||IsOperator(exp[exp.size()-1]))&&exp[0]!='('&&exp[exp.size()-
1]!=')') //此处若不加，在遇到某些式子时，会出现非法操作。
return false;
for(int m=0;m<exp.size();m )
{
check=exp[m];
if(IsOperand(check)); //如果是数字，跳过，不管。
else if(IsOperator(check))
```

```
{
if(check==')')
{
rb ;
if(IsOperator(exp[m 1])&&(exp[m 1]==' '||exp[m 1]=='-'||exp[m 1]=='*'||exp[m 1]=='/'||exp[m
1]=='^'||exp[m 1]==')'))
{
m ;
if(exp[m]==')')
rb ;
}
else if(IsOperator(exp[m 1]))
error   ;
}
else if(check=='(')
{
lb ;
if(exp[m 1]=='(')
{
m ;
lb ;
}
else if(IsOperator(exp[m 1]))
error   ;
}
else
{
if(IsOperator(exp[m 1])&&exp[m 1]=='(')
{
m ;
lb ;
}
else if(IsOperator(exp[m 1]))
error   ;
}
```

```
}
else
error   ;
}
if(error==0&&lb==rb)
return(true);
else
return(false);
}
/*************************************************************/
int main() //主函数开始
{
binary_tree etree;
stack<binary_tree>NodeStack;
stack<char>OpStack;
string infix;
char choice='y';
char c;

while(choice=='y'||choice=='Y')
{
cout<<"\n\n 请输入表达式，不要带空格;\n";
cin>>infix;
cout<<"--------------------------------------------------------------------------------"<<'\n';
cout<<"表达式为: "<<infix<<'\n';

if(isok(infix))
{
for(int i=0;i<infix.size();i )
{
c=infix[i];
if(IsOperand(c))
{
string tempstring;
tempstring=tempstring c;
```

```
while(i 1<infix.size()&&IsOperand(infix[i 1]))
{
tempstring=tempstring infix[ i];
}
binary_tree temp;
temp.root=build_node(tempstring);
NodeStack.push(temp);
}
/*********************************************************/
else if(c==' '||c=='-'||c=='*'||c=='/'||c=='^')
{
if(OpStack.empty())
OpStack.push(c);
else if(OpStack.top()=='(')
OpStack.push(c);
else if(TakesPrecedence(c,OpStack.top()))
OpStack.push(c);
else
{
while(!OpStack.empty()&&(TakesPrecedence(OpStack.top(),c)||addition(OpStack.top(),c)))
{
binary_tree temp_tree;
string thisstring="";
thisstring=thisstring OpStack.top();
OpStack.pop();
etree.root=build_node(thisstring);
copy(temp_tree.root,NodeStack.top().root);
NodeStack.pop();
etree.root->right_child=temp_tree.root;
temp_tree.root=NULL;
copy(temp_tree.root,NodeStack.top().root);
etree.root->left_child=temp_tree.root;
NodeStack.pop();
temp_tree.root=NULL;
copy(temp_tree.root,etree.root);
```

```
NodeStack.push(temp_tree);
etree.root=NULL;
}
OpStack.push(c);
}

}
else if(c=='(')
OpStack.push(c);
else if(c==')')
{
while(OpStack.top()!='(')
{
binary_tree temp_tree;
string thisstring="";
thisstring=thisstring OpStack.top();
OpStack.pop();
etree.root=build_node(thisstring);
copy(temp_tree.root,NodeStack.top().root);
NodeStack.pop();
etree.root->right_child=temp_tree.root;
temp_tree.root=NULL;
copy(temp_tree.root,NodeStack.top().root);
etree.root->left_child=temp_tree.root;
NodeStack.pop();
temp_tree.root=NULL;
copy(temp_tree.root,etree.root);
NodeStack.push(temp_tree);
etree.root=NULL;
}
OpStack.pop();
}
}
/*********************************************************/
while(!OpStack.empty())
```

```
{
binary_tree temp_tree;
string thisstring="";
thisstring=thisstring OpStack.top();
OpStack.pop();
etree.root=build_node(thisstring);
copy(temp_tree.root,NodeStack.top().root);
NodeStack.pop();
etree.root->right_child=temp_tree.root;
temp_tree.root=NULL;
copy(temp_tree.root,NodeStack.top().root);
etree.root->left_child=temp_tree.root;
NodeStack.pop();
temp_tree.root=NULL;
copy(temp_tree.root,etree.root);
NodeStack.push(temp_tree);
if(!OpStack.empty())
{
etree.root=NULL;
}
}
cout<<"打印结点如下: ";
etree.print();
binary_tree temp;
copy(temp.root,etree.root);
cout<<'\n';
cout<<"结点个数为："<<etree.counter()<<'\n';
cout<<"以下是，中间的计算结果："<<'\n';
etree.evaluate();
cout<<'\n';
cout<<"结果是: ";
cout<<etree.root->data<<'\n';
cout<<"--------------------------------------------------------------------------------"<<'\n';
cout<<"是不要进行二叉树排序，并显示？若是升序点 A，若是降序点 B,若不是点 C"<<'\n';
char c1;
```

```
cin>>c1;
if(c1=='A'||c1=='a')
{
binary_tree temp1;
copy(temp1.root,temp.order().root); //此处代码无需再改
temp1.inprint(temp1.root);//前边程序有错,此处 TEMP1 为空.所以没有输出.
}
else if(c1=='B'||c1=='b')
{
binary_tree temp1;
copy(temp1.root,temp.order().root);
temp1.deprint(temp1.root);
}

cout<<"\n\nRun Program again?Enter<y/n>:";
cin>>choice;
}
else
{
cout<<"************************************************"<<'\n';
cout<<"ERROR:Invalid Expression"<<'\n';
cout<<"************************************************"<<'\n';
cout<<"\n\nRun Program again?Enter<y/n>:";
cin>>choice;
}
}
return 0;
}
//环境 VC6.0
```

4.2.5 运行结果

```
表达式为: 1 {2 3^[4-(1 1)]}
打印结点如下: 1 2 3 4 1 1 - ^
结点个数为：11
```

以下是，中间的计算结果：

2 2 9 11 12

结果是: 12

4.3 图像压缩编码优化

4.3.1 项目简介

信息时代，人们对使用计算机获取信息、处理信息的依赖性越来越高。计算机系统面临的是数值、文字、语言、音乐、图形、动画、静图像、电视视频图像等多种媒体。数字化的视频和音频信号的数量之大是惊人的，对于电视画面的分辨率 640×480 的彩色图像，30 帧/s，则一秒钟的数据量为：640×480×24×30＝221　12M，所以播放时，需要 221Mbps 的通信回路。存储时，1 张 CD 可存 640M，则仅可以存放 2　89s 的数据。多媒体信息中，视频信息是一种比较特殊的媒体，数据量极大，信息丰富，并以与时间密切相关的流的形式存在。因此，视频数据的表达、组织、存储和传输都有很大难度。解决的基础在于对视频数据进行压缩。自 1948 年 Oliver 提出了 PCM 编码理论，编码技术日趋成熟，已经设计和应用了许多压缩算法和技术，如在 H.261、JPEG 和 MPEG 编码标准中的应用等。数据压缩目前的主要目标是较大的压缩比、较快的压缩解压速度以及尽可能好的图象还原质量，而对于压缩数据的处理如数据组织、检索、重构等还并没有较好的考虑，也没有一个比较完整的解决方案，因此在这方面仍有许多工作要做。最基本的是要有比较合理高效的压缩算法。

大数据量的图像信息会给存储器的存储容量，通信干线信道的带宽，以及计算机的处理速度增加极大的压力。单纯靠增加存储器容量，提高信道带宽以及计算机的处理速度等方法来解决这个问题是不现实的，这时就要考虑压缩。压缩的关键在于编码，如果在对数据进行编码时，对于常见的数据，编码器输出较短的码字；而对于少见的数据则用较长的码字表示，就能够实现压缩。

4.3.2 设计思路

假设一个文件中出现了 8 种符号 S0，SQ，S2，S3，S4，S5，S6，S7，那么每种符号要编码，至少需要 3bit。假设编码成 000，001，010，011，100，101，110，111。那么符号序列 S0S1S7S0S1S6S2S2S3S4S5S0S0S1 编码后变成 000001111000001110010010011100101000000001，共用了 42bit。我们发现 S0，S1，S2 这 3 个符号出现的频率比较大，其它符号出现的频率比较小，我们采用这样的编码方案：S0 到 S7 的码辽分别 01，11，101，0000，0001，0010，0011，100，那么上述符号序列变成 011110001110011101101000000010010010111，共用了 39bit。尽管有些码字如 S3，S4，S5，S6 变长了(由 3 位变成 4 位)，但使用频繁的几个码字如 S0，S1 变短了，所以实现了压缩。对于

上述的编码可能导致解码出现非单值性：比如说，如果 S0 的码字为 01，S2 的码字为 011，那么当序列中出现 011 时，你不知道是 S0 的码字后面跟了个 1，还是完整的一个 S2 的码字。因此，编码必须保证较短的编码决不能是较长编码的前缀。符合这种要求的编码称之为前缀编码。

4.3.3 数据结构

要构造符合这样的二进制编码体系，可以通过二叉树来实现。

1)首先统计出每个符合出现的频率，上例 SO 到 S7 的出现频率分别为 4/14，3/14，2/14，1/14，1/14，1/14，1/14，1/14。

2)从左到右把上述频率按从小到大的顺序排列。

3)每一次选出最小的两个值，作为二叉树的两个叶子节点，将和作为它们的根节点，这两个叶子节点不再参与比较，新的根节点参与比较。

4)重复(3)，直到最后得到和为 1 的根节点。

将形成的二叉树的左节点标 0，右节点标 1。把从最上面的根节点到最下面的叶子节点路径中遇到的 0，1 序列串起来，得到各个符号的编码。

可以看到，符号只能出现在树叶上，任何一个字符的路径都不会是另一字符路径的前缀路径，这样，前缀编码也就构造成功了。

这样一棵二叉树称之为 Huffman 树，常用于最佳判定，它是最优二叉树，是一种带权路径长度最短的二叉树。所谓树的带权路径长度，就是树中所有的叶结点的权值乘上其到根结点的路径长度(若根结点为 0 层，叶结点到根结点的路径长度为叶结点的层数)。树的带权路径长度记为：WPL＝(W1*L1＋W2*L2＋W3*L3＋…＋Wn*Ln)，N 个权值 Wi(i＝1，2，…n)构成一棵有 N 个叶结点的二叉树，相应的叶结点的路径长度为 Li(i＝1，2，…n)。Huffman 树得出的 WPL 值最小。

在对一幅大小为 100，672bytes 8 位 BMP 图像文件进行 Huffman 编码过程中，作者按照以下步骤实现了的压缩和解压缩算法。

1)扫描位图文件的全部数据(对应用于调色板的编码)，完成数据频度的统计。

2)依据数据出现的频度建立哈夫曼树。

3)将哈夫曼树的信息写入输出文件(压缩后文件)，以备解压缩时使用。

4)进行第二遍扫描，将原文件所有编码数据转化为哈夫曼编码，保存到输出文件。解压缩则为逆过程，以下是编码和解码的实现算法。

a)定义数据结构 Node 如下：

Struct Node

{long freq；//该节点符号的频率值，初值为 0

int parent；//该节点父节点的序号，初值为－1

int right；//该节点右子节点的序号，初值为－1

int left；//该节点左子节点的序号，初值为－1

｜Bmp tree［511］

说明：

之所以有 511 个节点，是因为每个字节可表示的符号个数为 256 个(对应于 256 种颜色)二叉树有 256 个叶节点，根据二叉树的性质总节点数为 2·256－1＝511 个节点。这里用 0～255 个元素来依次对应 256 种颜色。由第 256 以后的元素来依次对应形成的各个父节点的信息，即父节点的编号从 256 开始。

b)按照前述的压缩步骤，先对欲压缩文件的各个符号的使用次数进行统计，填充于 bmp tree［0～255］·freq 项内；在已有的节点中找出频率最低的两个节点，给出它们的父节点，将两个节点号填充于父节点的 right，left，将父节点号填充于两个节点的 Parent 内。重复步骤直到根节点，建树工作完成。

建树完成后进行编码，对每个符号从符号的父节点开始。若节点的父节点值不为－1，则一直进行下去，直到树根。回溯过程中遇左出 0，遇右出 1 输出编码。

Huffman 编码递归过程如下：

Void Bmp Huff Code(int node,int child)

if{Bmp tree［node］　Parent!＝－1}；//父节点为－1 的节点是树根

Bmp Huff Code(Bmp tree［node］　parent，node)；//若不为－1 则递归

if(child≠－1)；//若不为叶节点

{if(child＝bmp tree［node］　right)；//右子节点，输出“1”

outputbit(1)；

else if(child＝bmp tree［node］　left)；//左子节点，输出“0”

Outputbit(0)；

}

c)解码时从树根开始，遇 1 取右节点，遇 0 取左节点，直到找到节点号小于 256 的节点(叶节点)。

HuHnun 解码过程如下：

Int Expand Huffman(Void)

{int node＝Root-node-leaf；//解码从根节点开始。

do+

{Head Flag＝Getonebit()；//从编码串中读取一位。

if(Head Flag＝0)；//若为“0”，

{node＝Huffman-Tree［(node-256)*2］；}//取当前节点的左节点号；

else if(Head Flag＝1)若为“1”

{node＝Huffman-tree［)node-256)*2＋1］；}//取当前节点的右节点号；

}While(node>＝256)；//节点号大于 256 继续循环

}expanddata-buffer［counter＋＋］＝node；//输出解码得到一个字节

压缩后的文件大小为 48，431bytes，压缩比为 48　1%，解压缩后的数据读出的图像正常。

4.3.4 程序清单

```
//赫夫曼树与赫夫曼编码
//Huffman2.cpp
#include<iostream.h>
#include<iomanip.h>
#include<string.h>
#include<stdlib.h>
const int MaxV=100;//初始设定的最大权值
const int MaxBit=15;//初始设定的最大编码位数
const int MaxN=15;//初始设定的最大结点数
//赫夫曼树的结点结构
typedef struct
{int weight;//权值
 int parent;//双亲结点下标
 int left;//左孩子下标
 int right;//右孩子下标
}HTNode,*HuffmanTree;
typedef char **HuffmanCode;
//类定义
class HuffmanT
{public:
   //构造函数
   HuffmanT(HuffmanTree &,HuffmanCode &);
   //创建权值数组为 w 的赫夫曼树 HT,并求出 n 个字符的赫夫曼编码 HC
   void MakeHufm(int *w,int n);
 private:
   HuffmanTree HT;
   HuffmanCode HC;
};
//类实现
```

```
HuffmanT::HuffmanT(HuffmanTree &h1,HuffmanCode &h2)
{HT=h1;HC=h2;}
void HuffmanT::MakeHufm(int *w,int n)
{int m,m1,m2,x1,x2,i,j,start,c,f;
 HuffmanTree p;
 //赫夫曼树 HuffTree 的初始化
 if(n<=1) return;
 m=2*n-1;
 for(p=HT,i=0;i<n;++i,++p,++w)
   {p->weight=*w;p->parent=0;
    p->left=0;p->right=0;}
 for(;i<m;++i,++p)
   {p->weight=0;p->parent=0;
    p->left=0;p->right=0;}
 for(i=n;i<m;++i)//建赫夫曼树
 {m1=m1=MaxV;
  x1=x2=0;
  for(j=0;j<i;j++)
  {if(HT[j].weight<m1&&HT[j].parent==0)
     {m2=m1;
      x2=x1;
      m1=HT[j].weight;
      x1=j;
     }
    else if(HT[j].weight<m2&&HT[j].parent==0)
     {m2=HT[j].weight;
      x2=j;
     }
  }
  //将找出的两棵权值最小的子树合并为一棵子树
  HT[x1].parent=i;HT[x2].parent=i;
  HT[i].left=x1;HT[i].right=x2;
  HT[i].weight=HT[x1].weight+HT[x2].weight;
 }
 //从叶子到根逆向求每个字符的赫夫曼编码
```

```
  //分配求编码的工作空间
  char *cd=(char *)malloc(n*sizeof(char));
  cd[n-1]='\0';//编码结束符
  for(i=0;i<n;++i)//逐个字符求赫夫曼编码
  {start=n-1;//编码结束符位置
   //从叶子到根逆向求编码
   for(c=i,f=HT[i].parent;f!=0;c=f,f=HT[f].parent)
     if(HT[f].left==c) cd[--start]='0';
     else cd[--start]='1';
   //为第 i 个字符编码分配空间
   HC[i]=(char *)malloc((n-start)*sizeof(char));
   strcpy(HC[i],&cd[start]);//复制编码
  }
  free(cd);
}
//赫夫曼编码问题的测试
void main()
{cout<<"Huffman2.cpp 运行结果:\n";
  int i,n=4,q[4]={1,3,5,7};
  HuffmanTree ht=(HuffmanTree)malloc((2*n)*sizeof(HTNode));
  char **hc=(HuffmanCode)malloc((n+1)*sizeof(char *));
  HuffmanT t(ht,hc);//创建类对象 t
  t.MakeHufm(q,n);
  //输出每个叶结点的赫夫曼编码
  for(i=0;i<n;i++)
  {cout<<"weight="<<ht[i].weight;
   cout<<"    Code="<<hc[i]<<'\n';
  }
  cin.get();}
```

4.3.5 运行结果

第五章 图

图是一种较线性表和树更为复杂的数据结构，也是日常生活中应用广泛的结构之一。在线性表中，数据元素之间仅仅有线性关系，每个数据元素只有一个直接前驱和一个直接后继；在树形结构中，数据元素之间有着明显的层次关系，并且每一层上的数据元素可能和下一层中多个元素（即其孩子结点）相关，但只能和上一层中的一个元素（即其双亲结点相关）；而在图形结构中，结点之间的关系可以是任意的，图中任意两个数据元素之间都可能相关。由此，图的应用极为广泛，特别是近年来的迅速发展，已渗入到诸如语言学、逻辑学、物理、化学、电讯工程、计算机科学以及数学的其他分支中。

5.1 公交线路管理

5.1.1 项目简介

本项目是对公交车线路信息的简单模拟，以完成建立公交路线信息、修改公交路线信息和删除公交路线信息等功能。

5.1.2 设计思路

本项目的实质是完成对公交线路信息的建立、查找、插入、修改、删除等功能，可以首先定义项目的数据结构，然后将每个功能写成一个函数来完成对数据的操作，最后完成主函数以验证各个函数功能并得出运行结果。

5.1.3 数据结构

公交站点之间的关系可以是任意的，任意两个站点之间都可能相关。而在图形结构中，结点之间的关系可以是任意的，图中任意两个数据元素之间都可能相关。所以可以用图形结构来表示 n 个公交站点之间以及站点之间可能设置的公交路线，其中网的顶点表示公交站点，边表示两个站点之间的路线，赋予边的权值表示相应的距离。

因为公交路线是有一定的连续关系的，如果想输出从某一个起始点开始到某一终点结束的公交路线，就需要找到从某一顶点开始的第一个邻接点和下一个邻接点。因为在邻接表中容易找到任一顶点的第一个邻接点和下一个邻接点，所以本项目使用了图的邻接表存储结构。邻接表是图的一种链式存储结构。在邻接表中，对图的每一个顶点建立一个单链表，第 i 个单链表中的结点表示依附于顶点 v_i 的边（对有向图是以顶点 v_i 为尾的弧）。每个结点由三个域组成，其中邻接点域(adjvex)指示与顶点 v_i 邻接的点在图中的位置，链域(nextarc)指示下一条边或弧的结点；数据域(info)存储和边或弧相关的信息，如权值等。每个链表上附设一个表头结点，在表头结点中，除了设有链域(firstarc)指向链表中第一个结点之外，还设有存储

顶点 v_i 的名或其它有关信息的数据域(data)。这些表头结点通常以顺序结构的形式存储，以便随机访问任意顶点的链表。图的邻接表存储表示结构可形式的说明如下：

```
#define MAX_VERTEX_NUM   20
typedef struct ArcNode{     //弧的结构
int                  adjvex;   //该弧所指向的顶点的位置
struct ArcNode   *nextarc;   //指向下一条弧的指针
InfoType           *info;   //该弧相关信息的指针
}ArcNode;
typedef struct VNode{       //顶点结构
VertexType     data;   //顶点信息
ArcNode       *firstarc;   //指向第一条依附该顶点的弧的指针
}VNode,   AdjList[MAX_VERTEX_NUM];
typedef struct {       //图的结构
AdjList     vertices;
int          vexnum,   arcnum;   //图的当前顶点数和弧数
int          kind;   //图的种类标志
}ALGraph;
```

5.1.4 程序清单

```
//图的相关数据类型的定义 graph1.h
//最多顶点数
const int MaxV=10;
//定义邻接表中的边结点类型
struct edgenode {
  int adjvex;        //邻接点域
  int weight;        //权值域
  edgenode* next; //指向下一个边结点的链域
  edgenode(){}
  edgenode(int d,int w):adjvex(d),weight(w),next(NULL){}
};
struct Top //顶点数组的元素类型
{char data;//顶点数据
  edgenode *adj;//邻接表指针
};
```

```
struct RCW
{int row;
 int col;
 int weight;
};
//邻接表的类定义
class AdjAdjoin
{private:
   Top g[MaxV];//顶点数组
   int size;     //顶点个数
   int numE;     //当前边的条数
 public:
   edgenode **GL;//定义邻接表
   //构造函数
   AdjAdjoin() {}
   //构造函数,初始化图的邻接表
   AdjAdjoin(edgenode **gl,int n);
   //判断图空否
   bool GraphEmpty() {return size==0;}
   //取当前顶点数
   int NumV() {return size;}
   //取当前边数
   int NumEdges() {return numE;}
   //取顶点 i 的值
   char GetValue(const int i);
   //取弧<v1,v2>的权
   int GetWeight(const int v1,const int v2);
   //在位置 pos 处插入顶点 V
   void InsertV(const char &V);
   //插入弧<v1,v2>,权为 weight
   void InsertEdge(const int v1,const int v2,int weight);
   //删除顶点 i 与顶点 i 相关的所有边
   void DeleteVE(const int v);
   //删除弧<v1,v2>
   void DeleteEdge(const int v1,const int v2);
```

```
    //删除图的邻接表
    void DeleteAdjoin(int n);
    //建立图
    void CreatGraph(char V[],int n,RCW E[],int e);
    //建立图的邻接表
    void CreateAdjoin(int n,int k1,int k2,RCW rcw[]);
    //从初始点 vi 出发深度优先搜索由邻接表 GL 表示的图
    void dfsAdjoin(bool*& visited,int i,int n);
    //从初始点 vi 出发广度优先搜索由邻接表 GL 表示的图
    void bfsAdjoin(bool*& visited,int i,int n);
    //检查输入的边序号是否越界,若越界则重输
    void Check(int n,int& i,int& j);
      //对非连通图进行深度优先搜索
    void dfsAdjoin(int n);
    //对非连通图进行广度优先搜索
    void bfsAdjoin(int n);
};


//图的相关运算的实现 graph1.cpp
#include"graph1.h"
//构造函数,初始化图的邻接表
AdjAdjoin::AdjAdjoin(edgenode **gL,int n)
{GL=gL;
  for(int i=0;i<n;i++)
    {g[i].adj=GL[i]=NULL;}
  size=numE=0;
}
//建立图的邻接表
void AdjAdjoin::CreateAdjoin(int n,int k1,int k2,RCW rcw[])
{int i,j,k,e,w;
  cout<<"输入图的总边数:";
    cin>>e;
  if(k1==0 && k2==0) {    //建立无向无权图
      cout<<"输入"<<e
```

```
        <<"条无向无权边的起点和终点序号!"<<endl;
    for(k=1; k<=e; k++) {
      cin>>i>>j;
      Check(n,i,j);
    //向序号 i 的单链表的表头插入一个边结点
    edgenode *p=new edgenode;
    p->adjvex=j; p->weight=1;
    p->next=GL[i];
    GL[i]=p;//向序号 j 的单链表的表头插入一个边结点
    p=new edgenode;
    p->adjvex=i; p->weight=1;
    cout<<'('<<p->adjvex<<','<<j<<','<<p->weight<<")\n";
    p->next=GL[j];
    GL[j]=p;}
}
else if(k1==0 && k2!=0) {   //建立无向有权图
    cout<<"输入"<<e
        <<"条无向带权边的起点和终点序号及权值!"<<endl;
    for(k=0; k<e; k++) {
      i=rcw[k].row;j=rcw[k].col;w=rcw[k].weight;
      //cin>>i>>j>>w;
      Check(n,i,j);
    //向序号 i 的单链表的表头插入一个边结点
    edgenode *p=new edgenode;
    p->adjvex=j; p->weight=w;
    p->next=GL[i];
    GL[i]=p;
    //向序号 j 的单链表的表头插入一个边结点
    p=new edgenode;
    p->adjvex=i;p->weight=w;
    p->next=GL[j];
    GL[j]=p;}
}
else if(k1!=0&&k2==0) {   //建立有向无权图
    cout<<"输入"<<e<<"条有向无权边的起点和终点序号!"<<endl;
```

```
    for(k=1; k<=e; k++) {
      cin>>i>>j;
      Check(n,i,j);
      //向序号 i 的单链表的表头插入一个边结点
      edgenode* p=new edgenode;
      p->adjvex=j; p->weight=1;
      p->next=GL[i];
      GL[i]=p;}
  }
  else if(k1!=0&&k2!=0) {   //建立有向有权图
    cout<<"输入"<<e
        <<"条有向有权边的起点和终点序号及权值!"<<endl;
    for(k=1; k<=e; k++) {
      cin>>i>>j>>w;
      Check(n,i,j);
      edgenode* p=new edgenode;
      p->adjvex=j; p->weight=w;
      p->next=GL[i];
      GL[i]=p;}}
  numE=e;size=n;
}
//从初始点 vi 出发深度优先搜索由邻接表 GL 表示的图
void AdjAdjoin::dfsAdjoin(bool*& visited,int i,int n)
{cout<<g[i].data<<':'<<i<<"    ";
  visited[i]=true;
  //取 vi 邻接表的表头指针
  edgenode *p=GL[i];
  //依次搜索 vi 的每个邻接点
  while (p!=NULL) {
    int j=p->adjvex;//j 为 vi 的一个邻接点序号
    if(!visited[j])
    dfsAdjoin(visited,j,n);
    p=p->next;   //使 p 指向 vi 单链表的下一个边结点
}}
//从初始点 vi 出发广度优先搜索由邻接表 GL 表示的图
```

```
void AdjAdjoin::bfsAdjoin(bool*& visited,int i,int n)
{const int    MaxLength=30;
 //定义一个队列 q,其元素类型应为整型
 int q[MaxLength]={0};
 //定义队首和队尾指针
 int front=0, rear=0;
 //访问初始点 vi
 cout<<g[i].data<<':'<<i<<"    ";
 //标记初始点 vi 已访问过
 visited[i]=true;
 //将已访问过的初始点序号 i 入队
 q[++rear]=i;
 //当队列非空时进行循环处理
 while(front!=rear) {
  //删除队首元素，第一次执行时 k 的值为 i
  front=(front+1)%MaxLength;
  int k=q[front];
  //取 vk 邻接表的表头指针
  edgenode* p=GL[k];
  while(p!=NULL)
     {//依次搜索 vk 的每一个邻接点
      int j=p->adjvex;   //vj 为 vk 的一个邻接点
      if(!visited[j]) { //若 vj 没有被访问过则进行处理
         cout<<g[j].data<<':'<<j<<"    ";
         visited[j]=true;
         rear=(rear+1)%MaxLength;//顶点序号 j 入队
         q[rear]=j;}
         p=p->next; //使 p 指向 vk 邻接表的下一个边结点
}}}
//检查输入的边序号是否越界,若越界则重输
void AdjAdjoin::Check(int n,int& i,int& j)
{while(1) {
  if(i<0||i>=n||j<0||j>=n)
     cout<<"输入有误,请重输!";
  else return;
```

```
   cin>>i>>j;
  }
}
//取顶点 i 的值
char AdjAdjoin::GetValue(const int i)
{if(i<0||i>size)
   {cerr<<"参数 i 越界!\n";exit(1);}
  return g[i].data;
}
//取弧<v1,v2>的权
int AdjAdjoin::GetWeight(const int v1,const int v2)
{if(v1<0||v1>size||v2<0||v2>size)
   {cerr<<"参数 v1 或 v2 越界!\n";exit(1);}
  edgenode *p=g[v1].adj;
  while(p!=NULL&&p->adjvex<v2) p=p->next;
  if(v2!=p->adjvex)
   {cerr<<"边<v1,v2>不存在!\n";exit(1);}
  return p->weight;
}
//在位置 pos 处插入顶点 V
void AdjAdjoin::InsertV(const char &V)
{g[size].data=V;
  size++;
}
//插入弧<v1,v2>,权为 weight
void AdjAdjoin::InsertEdge(const int v1,const int v2,int weight)
{if(v1<0||v1>size||v2<0||v2>size)
   {cerr<<"参数 v1 或 v2 越界!\n";exit(1);}
  edgenode *q=new edgenode(v2,weight);
  //q->adjvex=v2;q->weight=weight;
  if(g[v1].adj==NULL) //第一次插入
    g[v1].adj=q;
  else                          //非第一次插入
   {edgenode *curr=g[v1].adj,*pre=NULL;
     while(curr!=NULL&&curr->adjvex<v2)
```

```
    {pre=curr;
     curr=curr->next;
    }
    if(pre==NULL)       //在第一个结点前插入
     {q->next=g[v1].adj;
      g[v1].adj=q;
     }
    else                //在其他位置插入
     {q->next=pre->next;
      pre->next=q;
     }
   }
  numE++;
}
//删除顶点 v 与顶点 v 相关的所有边
void AdjAdjoin::DeleteVE(const int v)
{edgenode *pre,*curr;
 for(int i=0;i<size;i++)
 {pre=NULL;          //删除顶点 v 的入边
  curr=g[i].adj;
  while(curr!=NULL&&curr->adjvex<v)
  {pre=curr;
   curr=curr->next;
  }
  if(pre==NULL&&curr->adjvex==v)
   {g[i].adj=curr->next; //该出边结点是链表的第一结点时
    delete curr;
    numE--;
   }
  else if(curr!=NULL&&curr->adjvex==v)
   {pre->next=curr->next;//该出边结点是链表的其他结点时
    delete curr;
    numE--;
   }
 }
```

```
  edgenode *p=g[v].adj,*q;
  for(i=v;i<size-1;i++)
    g[i]=g[i+1]; //删除数组的顶点 v 元素
  numE--;
  while(p!=NULL)//删除顶点 v 的所有出边
  {q=p->next;
    delete p;       //释放空间
    p=q;
    numE--;
  }
}
//删除弧<v1,v2>
void AdjAdjoin::DeleteEdge(const int v1,const int v2)
{if(v1<0||v1>size||v2<0||v2>size||v1==v2)
    {cerr<<"参数 v1 或 v2 出错!\n";exit(1);}
  edgenode *curr=g[v1].adj,*pre=NULL;
    while(curr!=NULL&&curr->adjvex<v2)
    {pre=curr;
      curr=curr->next;
    }
    if(pre==NULL&&curr->adjvex==v2)//要删除的结点是链表的第一结点
      {g[v1].adj=curr->next;
        delete curr;
        numE--;
      }
    else if(curr!=NULL&&curr->adjvex==v2)//不是链表的第一结点
      {pre->next=curr->next;
        delete curr;
        numE--;
      }
    else
      {cerr<<"边<v1,v2>不存在!\n";exit(1);}
}
//删除图的邻接表
void AdjAdjoin::DeleteAdjoin(int n)
```

```
{int i;
 edgenode* p;
 for(i=0;i<n;i++)
 {p=GL[i];
  while(p!=NULL)
    {GL[i]=p->next;
     delete p;p=GL[i];
    }
 }
 delete []GL;
}
//对非连通图进行深度优先搜索
void AdjAdjoin::dfsAdjoin(int n)
{bool *vis=new bool[NumV()];
 for(int i=0;i<NumV();i++) vis[i]=false;
 for(i=0;i<NumV();i++)
  if(!vis[i]) dfsAdjoin(vis,i,n);
 delete []vis;
}
//对非连通图进行广度优先搜索
void AdjAdjoin::bfsAdjoin(int n)
{bool *vis=new bool[NumV()];
 for(int i=0;i<NumV();i++) vis[i]=false;
 for(i=0;i<NumV();i++)
  if(!vis[i]) bfsAdjoin(vis,i,n);
 delete []vis;
}
void AdjAdjoin::CreatGraph(char V[],int n,RCW E[],int e)
{for(int i=0;i<n;i++) InsertV(V[i]);
 for(int k=0;k<e;k++)
  InsertEdge(E[k].row,E[k].col,E[k].weight);
 cout<<"输出建立的图:\n";
 for(i=0;i<n;i++)
  cout<<g[i].data<<"    ";
 cout<<endl;
```

```
}


//图的相关运算的测试 graph1M.cpp
#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
#include "graph1.cpp"
void main()
{cout<<"graph1M.cpp 运行结果:\n";
 char a[]={'A','B','C','D','E','F','G'};
 RCW rcw[]={{0,1,1},{0,2,1},{1,3,1},{1,4,1},{2,5,1},
   {2,6,1},{1,0,1},{2,0,1},{3,1,1},{4,1,1},{5,2,1},{6,2,1}};
 //定义图的点数及搜索起始点序号等
 int n,k,i;
 //k1 为 0 则无向否则为有向,k2 为 0 则无权否则为有权
 int k1,k2;
 //标记已访问过的点
 bool *vis;
 cout<<"输入图的点数 n=";cin>>n;
 vis=new bool[n];
 if(!vis) {cout<<"申请堆内存失败!\n";exit(1);}
 for(i=0;i<n;i++) vis[i]=false;
 cout<<"输入选择无向(权)与有向(权)图的值 k1,k2:";
 cin>>k1>>k2;
 edgenode **gl=new edgenode*[n];
 AdjAdjoin B(gl,n);
 B.CreatGraph(a,n,rcw,12);
 cout<<"创建邻接表:\n";
 B.CreateAdjoin(n,k1,k2,rcw);
 cout<<"出发点 Vk 的序号=";cin>>k;
 cout<<"当前的顶点数为:"<<B.NumV()<<endl;
 cout<<"当前的边数为:"<<B.NumEdges()<<endl;
 cout<<"表的深度优先搜索顺序:\n";
 B.dfsAdjoin(vis,k,n);cout<<endl;
```

```
    cout<<"表的广度优先搜索顺序:\n";
    for(i=0;i<n;i++) vis[i]=false;
    B.bfsAdjoin(vis,k,n);cout<<endl;

    B.DeleteEdge(0,2);
    B.DeleteEdge(2,0);
    cout<<"当前的顶点数为:"<<B.NumV()<<endl;
    cout<<"当前的边数为:"<<B.NumEdges()<<endl;
    cout<<"表的深度优先搜索顺序:\n";
    B.dfsAdjoin(n);cout<<endl;
    cout<<"表的广度优先搜索顺序:\n";
    for(i=0;i<n;i++) vis[i]=false;
    B.bfsAdjoin(n);cout<<endl;
    B.DeleteAdjoin(n);
    cin.get();cin.get();
}
```

5.1.5 运行结果

5.2 导航最短路径查询

5.2.1 项目简介

设计一个交通咨询系统，能让旅客咨询从任一个城市顶点到另一个城市顶点之间的最短路径问题。设计分三个部分，一是建立交通网络图的存储结构；二是解决单源最短路径问题；最后再实现两个城市顶点之间的最短路径问题。

最短路径问题的提出随着计算机的普及以及地理信息科学的发展，GIS 因其强大的功能得到日益广泛和深入的应用。网络分析作为 GIS 最主要的功能之一，在电子导航、交通旅游、城市规划以及电力、通讯等各种管网、管线的布局设计中发挥了重要的作用。而网络分析中最基本和关键的问题是最短路径问题[34]。

最短路径不仅仅指一般地理意义上的距离最短，还可以引申到其他的度量，如时间、费用、线路容量等。相应地，最短路径问题就成为最快路径问题、最低费用问题等。由于最短路径问题在实际中常用于汽车导航系统以及各种应急系统等（110 报警、119 火警以及医疗救护系统），这些系统一般要求计算出到出事地点的最佳路线的时间一般在 1s-3s，在行车过程

中还需要实时计算出车辆前方的行驶路线，这就决定了最短路径问题的实现应该是高效率的[36]。最优路径问题不仅包括最短路径问题，还有可能涉及到最少时间问题、最少收费（存在收费公路）问题、或者是几个问题的综合，这时将必须考虑道路级别、道路流量、道路穿越代价（如红灯平均等待时间）等诸多因素。但是必须指出的是，一般来说最优路径在距离上应该是最短的，但最短路径在行驶时间和能源消耗的意义上未必是最优的[35]。其实，无论是距离最短、时间最快还是费用最低，它们的核心算法都是最短路径算法。

5.2.2 设计思路

单源最短路径算法的主要代表之一是 Dijkstra（迪杰斯特拉）算法。该算法是目前多数系统解决最短路径问题采用的理论基础，在每一步都选择局部最优解，以期望产生一个全局最优解[40]。

Dijksira 算法的基本思路是：对于图 G=(V,E)，V 是包含 n 个顶点的顶点集，E 是包含 m 条弧的弧集，(v, w)是 E 中从 v 到 w 的弧，c(v, w)是弧(v, w)的非负权值，设 s 为 V 中的顶点，t 为 V 中可由 s 到达的顶点，则求解从 s 至 t 的具有最小弧权值和的最短路径搜索过程如下：

(1) 将 v 中的顶点分为 3 类：已标记点、未标记点、已扫描点。将 s 初始化为已标记点，其它顶点为未标记点。为每个顶点 v 都建立一个权值 d 和后向顶点指针 p，并将 d 初始化如下：d(v)=0，v=s；d(v)=∞，v≠s。

(2) 重复进行扫描操作：从所有已标记点中选择一个具有最小权值的顶点 v 并将其设为已扫描点，然后检测每个以 v 为顶点的弧(v, w)，若满足 d(v) + c(v, w) < d(w) 则将顶点 v 设为已标记点，并令 d(w) = d(v) + c(v, w), p(w) = v。

(3) 若终点 t 被设为已扫描点，则搜索结束。由 t 开始遍历后向顶点指针 P 直至起点 s，即获得最短路径解。

5.2.3 数据结构

（1）定义一个数组 min_dist，它的每个数组元素 min_dist[i]表示当前所找到的从始点 vi 到每个终点 vj 的最短路径的长度。它的初态为：若从 vi 到 vj 有边，则 min_dist[j]为边的权值；否则置 min_dist[i]为∞。定义一个数组 path，其元素 path[k]（0≤k≤n-1）用以记录 vi 到 vk 最短路径中 vk 的直接前驱结点序号，如果 vi 到 vk 存在边，则 path[k]初值为 i。定义一个数组 W，存储任意两点之间边的权值。

（2）查找 min(min_dist[j]，j∈V-S)，设 min_dist[k]最小，将 k 加入 S 中。修改对于 V-S 中的任一点 vj，min_dist[j]=min(min_dist[k]+w[k][j], min_dist[j]) 且 path[j]=k。

（3）重复上一步，直到 V-S 为空。

在算法设计时，用一个 tag 数组来记录某个顶点是否已计算过最短距离，如果 tag[k]=0，则 vk∈V-S，否则 vk∈S。初始值除 tag[i]=1 以外，所有值均为 0。

5.2.4 程序清单

```
/*****************************************************************/
/*程序名：samp10_7．c                                            */
/*程序功能：求带权图中一个指定顶点到其它所有顶点之间             */
/*的最短矩离和最短路径                                           */
/*****************************************************************/
#include"stdio.h"
#include"stdlib.h"
#include"graph_define.h"
#defineMAXNUM32767
/*****************************************************************/
/*函 数 名：min_distance1                                        */
/*函数功能：求顶点 i 到其余各顶点的最短距离                      */
/*入口参数：  w-- 图的带权邻接矩阵                               */
/*min_dist -- 顶点 i 到其它各个顶点的距离                        */
/*path -- 最短路径                                               */
/*i-- 顶点 i                                                     */
/*n-- 总的边数                                                   */
/*返 回 值：无                                                   */
/*****************************************************************/
void min_distance1 (int w[][MAX_VERTEX_NUM], int min_dist[],int path[], int i, int n)
{
    int j,k,t;
    unsigned min;
    int tag[MAX_VERTEX_NUM];
    for(j=0;j<n;j++ )
        tag[j] = 0;
        tag[i]=1;
    for(j=0;j<n;j++)
        if (min_dist[j] != MAXNUM )
            path[j] = i;
            t=1;
            while(t==1)
            {
```

```
                min=MAXNUM;
        for(j=0;j<n;j++)
            if(tag[j]==0&&min>=min_dist[j])
            {
                min=min_dist[j];
                k=j;
            }
            tag[k]=1;
        for(j=0;j<n;j++)
            if(min_dist[j]> min_dist[k]+w[k][j])
            {
                min_dist[j]= min_dist[k]+w[k][j];
                path[j]=k;
            }
            t=0;
        for(j=0;j<n;j++)
            if(tag[j]==0) t=1;
            }
    }
```

5.2.5 运行结果

例如，图 10-36 所示一个有向带权图邻接矩阵为：

图 10-36　一个有向带权图及其邻接矩阵

若施行 Dijkstra 算法，则从 v0 到其余各顶点的最短路径，以及运算过程中 min_dist 向量的变化状况，如下表所示：

图 10-2　有向图示例

5.3 电网建设造价计算

5.3.1 项目简介

假设一个城市有 n 个小区，要实现 n 个小区之间的电网都能够相互接通，构造这个城市n个小区之间的电网，使总工程造价最低。请设计一个能满足要求的造价方案。

5.3.2 设计思路

在每个小区之间都可以设置一条电网线路，相应的都要付出一点经济代价。n 个小区之间最多可以有 n(n-1)/2 条线路，选择其中的 n-1 条使总的耗费最少。可以用连通网来表示 n 个城市之间以及 n 个城市之间可能设置的电网线路，其中网的顶点表示小区，边表示两个小区之间的线路，赋予边的权值表示相应的代价。对于 n 个顶点的连通网可以建立许多不同的生成树，每一颗生成树都可以是一个电路网。现在，我们要选择总耗费最少的生成树，就是构造连通网的最小代价生成树的问题，一颗生成树的代价就是树上各边的代价之和。

设 G=(V, E)是具有 n 个顶点的网络，T=(U, TE)为 G 的最小生成树，U 是 T 的顶点集合，TE 是 T 的边集合。Prim 算法的基本思想是：首先从集合 V 中任取一顶点（例如去顶点 v0）放入集合 U 中，这时 U={ v0}，TE=NULL。然后找出所有一个顶点在集合 U 里，另一个顶点在集合 V-U 里的边，使权(u, v)(u∈U, v∈V-U)最小，将该边放入 TE，并将顶点 v 加入集合 U。重复上诉操作直到 U=V 为止。这时 TE 中有 n-1 条边，T=(U, TE)就是 G 的一颗最小生成树。

5.3.3 数据结构

假设图采用邻接矩阵表示法表示，用一对顶点的下标（在顶点表中的下标）表示一条边，定义如下：

```
typedef struct{
int start_vex, stop vex;          //边的起点和终点
AdjType weight;           //边的权
}Edge;
```

在构造最小生成树的过程中定义一个类型为 Edge 的数组 mst: Edge mst[n-1]；其中，n 为网络中顶点的个数，算法结束时，mst 中存放求出的最小生成树的 n-1 条边。

可以用带权的无向图（即无向网）表示这 n 个小区之间的电网连接，其中顶点表示小区，权值表示城市之间电网建设的造价，构造一个无向网的最小生成树即是满足要求的最低电网连接造价方案。

5.3.4 程序清单

```
//利用普里姆算法求出用邻接
//矩阵表示的图的最小生成树
//图的相关数据类型的定义 graph2.h
//最多顶点数
const int MaxV=10;
//最大权值
const int MaxValue=99;
//定义边集数组中的元素类型
struct RCW
{int row,col;
 int weight;
};
//类定义
class adjMList
{private:
   int numE;//当前边数
   int GA[MaxV][MaxV];//定义邻接矩阵
 public:
   //构造函数,初始化图的邻接矩阵与边集数组
   adjMList(RCW GE[],int n,int e);
   //建立无向带权图的邻接矩阵
   void CreateMatrix(int n,int &e,RCW r[]);
   //输出边集数组中的每条边
   void OutputEdgeSet(RCW ge[],int e);
   //根据图的邻接矩阵生成图的边集数组
   void ChangeEdgeSet(RCW GE[],int n,int e);
   //按升序排列图的边集数组
   void SortEdgeSet(RCW GE[],int e);
   //利用普里姆算法从顶点 v0 出发求出用邻接矩阵 GA 表
   //示的图的最小生成树,最小生成树的边集存于数组 CT 中
   void Prim(RCW CT[],int n);
   //检查输入的边序号是否越界,若越界则重输
   void Check(int n, int& i, int& j);
```

```
};

//图的运算的实现文件 graph2.cpp
#include"graph2.h"
//构造函数,初始化图的邻接矩阵与边集数组
adjMList::adjMList(RCW GE[],int n,int e)
{int i,j;
  for(i=0; i<n; i++)
  for(j=0; j<n; j++)
    if(i==j) GA[i][j]=0;
    else GA[i][j]=MaxValue;
  for(i=0;i<e;i++) GE[i].weight=0;
  numE=0;
}
//输出边集数组中的每条边
void adjMList::OutputEdgeSet(RCW ge[],int e)
{int i,k=0;
  cout<<"{";
  for(i=0; i<=e-2; i++)
    if(ge[i].weight>0){k++;
      cout<<'('<<ge[i].row<<','<<ge[i].col;
      cout<<','<<ge[i].weight<<") ";
      if(k%5==0) cout<<endl;}
    if(e>0&&ge[i].weight>0) {
      cout<<'('<<ge[e-1].row<<','<<ge[e-1].col;
      cout<<','<<ge[e-1].weight<<')';}
      cout<<'}'<<endl;
}
//建立无向带权图的邻接矩阵
void adjMList::CreateMatrix(int n,int &e,RCW r[])
{int i,j,k=0,w;
  cout<<"依次输入无向带权图的每条边的起点和终点"<<endl;
  cout<<"序号及权值!直到输入权值为 0 的边为止!"<<endl;
  do {i=r[k].row;j=r[k].col;w=r[k].weight;
        //cin>>i>>j>>w;
```

```
        Check(n,i,j);
        if(k==e-1) break;
        GA[i][j]=GA[j][i]=w;k++;
    }while(1);
  numE=e=k;
  cout<<"邻接矩阵:\n";
  for(i=0;i<n;i++)
  {for(j=0;j<n;j++)
     cout<<setw(4)<<GA[i][j];
     cout<<endl;}
}
//检查输入的边序号是否越界,若越界则重输
void adjMList::Check(int n, int& i, int& j)
{while(1) {
     if(i<0 || i>=n ||j<0 || j>=n)
       cout<<"输入有误,请重输!";
     else return;
   cin>>i>>j;}
}
//根据图的邻接矩阵生成图的边集数组
void adjMList::ChangeEdgeSet(RCW GE[],int n,int e)
{//假定只考虑无向图的情况
  int i,j,k=0;
  for(i=0; i<n; i++)
  for(j=i+1; j<n; j++)
    if(GA[i][j]!=0 && GA[i][j]!=MaxValue)
     {if(k==e) {cout<<"数组 GE 下标越界!\n";
        exit(1);}
       GE[k].row=i;
       GE[k].col=j;
       GE[k].weight=GA[i][j];
       k++;
     }
}
//按升序排列图的边集数组
```

```
void adjMList::SortEdgeSet(RCW GE[],int e)
{int i,j;
 RCW x;
 for(i=1; i<=e-1; i++)
   {x=GE[i];
    for(j=i-1; j>=0; j--)
      if(x.weight<GE[j].weight) GE[j+1]=GE[j];
      else break;
    GE[j+1]=x;
   }
}
//利用普里姆算法从顶点 v0 出发求出用邻接矩阵 GA 表示
//的图的最小生成树,最小生成树的边集存于数组 CT 中
void adjMList::Prim(RCW CT[],int n)
{int i,j, k, min, t, m, w;
   //给 CT 赋初值，对应为 v0 依次到其余各顶点的边
 for(i=0; i<n-1; i++)
   {CT[i].row=0;
    CT[i].col=i+1;
    CT[i].weight=GA[0][i+1];}
    //进行 n-1 次循环，每次求出最小生成树中的第 k 条边
 for(k=1; k<n; k++)
   {//从 CT[k-1]~CT[n-2]中查找最短边 CT[m]
    min=MaxValue;
    m=k-1;
    for(j=k-1; j<n-1; j++)
      if(CT[j].weight<min) {
        min=CT[j].weight;
        m=j;}
   //把最短边对调到第 k-1 下标位置
   RCW temp=CT[k-1];
   CT[k-1]=CT[m];
   CT[m]=temp;
   //把新并入最小生成树 T 中的顶点序号赋给 j
   j=CT[k-1].col;
```

```
    //修改有关边，使 T 中到 T 外的每一个顶点各保持
    //一条到目前为止最短的边
    for(i=k; i<n-1; i++) {
      t=CT[i].col;
      w=GA[j][t];
      if(w<CT[i].weight) {
        CT[i].weight=w;
        CT[i].row=j;
}}}}


//图的相关运算的测试 graph2M.cpp
#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
#include "graph2.cpp"
void main()
{cout<<"graph2M.cpp 运行结果:\n";
  RCW rcw[20]={{0,1,50},{1,0,50},{0,2,60},{2,0,60},
    {1,3,65},{3,1,65},{1,4,40},{4,1,40},{2,3,52},
    {3,2,52},{2,6,45},{6,2,45},{3,4,50},{4,3,50},
    {3,5,30},{5,3,30},{3,6,42},{6,3,42},{4,5,70},{5,4,70}};
  int n,k;//定义图的点数及边数等
  cout<<"输入图的点数 n=";cin>>n;
  cout<<"输入图的边数 k=";cin>>k;
  static RCW AE[30],BE[30];//定义边集数组
  adjMList A(AE,n,k);
  A.CreateMatrix(n,k,rcw);
  cout<<"输出边集数组中的每条边:\n";
  A.ChangeEdgeSet(AE,n,k);
  A.OutputEdgeSet(AE,k);
  cout<<"输出按升序排列的图的边集数组:\n";
  A.SortEdgeSet(AE,k);
  A.OutputEdgeSet(AE,k);
  A.Prim(BE,n);
```

```
cout<<"输出最小生成树的边集数组:\n";
A.OutputEdgeSet(BE,k);
cin.get();cin.get();}
```

5.3.5 运行结果

```
输入图的点数 n=7
输入图的边数 k=20
依次输入无向带权图的每条边的起点和终点
序号及权值!直到输入权值为 0 的边为止!
邻接矩阵:
  0  50  60  99  99  99  99
 50   0  99  65  40  99  99
 60  99   0  52  99  99  45
 99  65  52   0  50  30  42
 99  40  99  50   0  70  99
 99  99  99  30  70   0  99
 99  99  45  42  99  99   0
输出边集数组中的每条边:
{(0,1,50) (0,2,60) (1,3,65) (1,4,40) (2,3,52) (2,6,45) (3,4,50) (3,5,30) (3,6,42) (4,5,70) }
输出按升序排列的图的边集数组:
{(3,5,30) (1,4,40) (3,6,42) (2,6,45) (0,1,50) (3,4,50) (2,3,52) (0,2,60) (1,3,65) (4,5,70)}
输出最小生成树的边集数组:
{(0,1,50) (1,4,40) (4,3,50) (3,5,30) (3,6,42) (6,2,45) }
```

5.4 软件工程进度规划

5.4.1 项目简介

设计一个软件，需要进行用户需求分析、系统需求确认、系统概要设计、设计用例场景、系统的详细设计、数据库详细设计、编码、单元测试、集成测试、系统测试、维护等活动。用户需求分析需要在系统需求确认之前完成，系统的系统的详细设计必须在系统的概要设计、设计系统用例和设计用例场景之前完成。

如表所示，是一系列活动之间的关系。

表 系统活动之间的关系

活动代码	活动名称	先需活动
A1	用户需求分析	无
A2	系统需求确认	A1
A3	系统概要设计	A2
A4	设计用例场景	无
A5	系统的详细设计	A3, A4
A6	数据库详细设计	A3
A7	编码	A5, A6
A8	单元测试	A7
A9	集成测试	A8
A10	系统测试	A7
A11	维护	A11

图所示是设计一个软件的 AOV 网示意图。

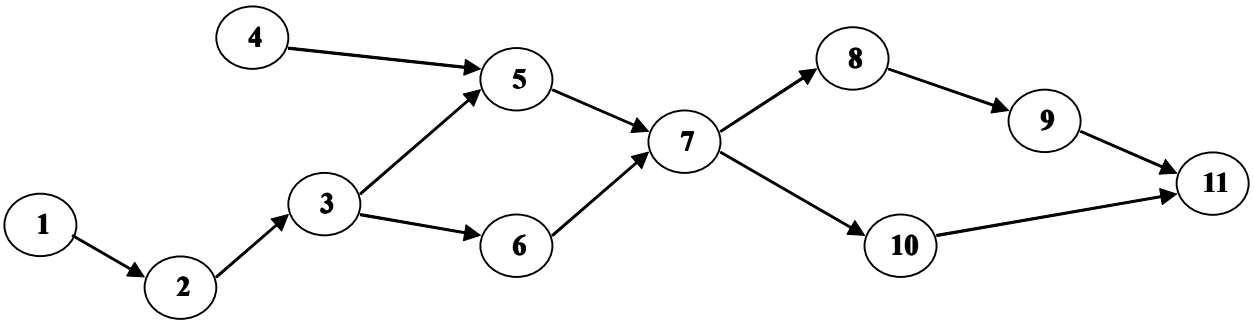

图 软件设计流程的 AOV 网

请设计算法判断该软件设计流程是否有回路，若无请给出该软件设计 AOV 网的拓朴序列。

5.4.2 设计思路

拓扑排序(topological sort)是求解网络问题所需的主要算法。管理技术如计划评审技术 PERT(Performance Evaluation And Review Technique)和关键路径法 CPM(Critical Path Method)都应用这一算法。通常，软件开发、施工过程、生产流程、程序流程等都可作为一个工程。一个工程可分成若干子工程，子工程常称为活动(activity)。因此要完成整个工程，必须完成所有的活动。活动的执行常常伴随着某些先决条件，一些活动必须先于另一些活动被完成。

AOV 网络代表的领先关系应当是一种拟序关系，它具有传递性(transitive)和反自反性

(irreflexive)。如果这种关系不是反自反的，就意味着要求一个活动必须在它自己开始之前就完成。这显然是荒谬的，这类工程是不可实施的。如果给定了一个 AOV 网络，我们所关心的事情之一，是要确定由此网络的各边所规定的领先关系是否是反自反的，也就是说，该 AOV 网络中是否包含任何有向回路。一般地，它应当是一个有向无环图(DAG)。

一个拓扑序列(topological order)是 AOV 网络中顶点的线性序列，使得对图中任意两个顶点 i 和 j，i 是 j 的前驱结点，则在线性序列中 i 先于 j。

拓扑排序算法可描述如下：

(1) 在图中选择一个入度为零的顶点，并输出之；

(2) 从图中删除该顶点及其所有出边(以该顶点为尾的有向边)；

(3) 重复(1)和(2)，直到所有顶点都已列出，或者直到剩下的图中再也没有入度为零的顶点为止。后者表示图中包含有向回路。

5.4.3 数据结构

拓扑排序可以在不同的存储结构上实现，与遍历运算相似，邻接表方法在这里更有效。拓扑排序算法包括两个基本操作：(1)决定一个顶点是否入度为零；(2)删除一个顶点的所有出边。如果我们对每个顶点的直接前驱予以计数，使用一个数组 InDgree 保存每个顶点的入度，即 InDgree[i]为顶点 i 的入度，则基本操作(1)很容易实现。而基本操作(2)在使用邻接表表示时，一般会比邻接矩阵更有效。在邻接矩阵的情况下，必须处理与该顶点有关的整行元素(n 个)，而邻接表只需处理在邻接矩阵中非零的那些顶点。

5.4.4 程序清单

```
//图类结构体定义与相关操作 graph4.h
typedef struct
{char *data;
 int *visited;
 float **edge;
 int max,size;
}Graph;
//初始化图
void SetGraph(Graph *G,int n)
{int i,j;
 G->data=new char[n];
 G->visited=new int[n];
 G->edge=(float **)malloc(n*sizeof(float *));
```

```
  for(i=0;i<n;i++)
    G->edge[i]=(float *)malloc(n*sizeof(float));
  for(i=0;i<n;i++) G->visited[i]=0;
  for(i=0;i<n;i++)
    for(j=0;j<n;j++) G->edge[i][j]=0;
  G->max=n;
  G->size=0;
}
//构造图
void MakeGraph(Graph *G,RCW r[],int n,int e)
{int m=0;
  while(m<n)
    {if(G->size==G->max)
      {cout<<"Graph is full!\n";
        exit(1);}
      G->data[G->size]='a'+m;
      G->size++;m++;}
    //插入弧
    for(int p=0;p<e;p++)
    {int i,j,k;
      for(k=0;k<n;k++)
        {if(r[p].w1==G->data[k]) i=k;
          if(r[p].w2==G->data[k]) j=k;}
      G->edge[i][j]=r[p].w;}
}

//拓扑排序 topSort.cpp
#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
typedef struct
{char w1,w2;
  float w;
}RCW;
#include "graph4.h"
```

```
typedef struct
{int *data;
 int max,top;
}Stack;
void TopSort(Graph *G)
{int i,j,n,d,count=0,*D;
 Stack S;
 if(G->size==0) return;
 n=G->size;
 S.data=new int[n];
 S.max=n;S.top=-1;
 D=new int[n];
 for(j=0;j<n;j++)
   {d=0;
    for(i=0;i<n;i++)
      if(G->edge[i][j]!=0) d++;
      D[j]=d;
      if(d==0)
        {if(S.top==S.max-1)
           {cout<<"Stack is full!\n";exit(1);}
         S.top++;
         S.data[S.top]=j;
        }
   }
 while(!(S.top==-1))
 {if(S.top==-1)
     {cout<<"Pop an empty stack!\n";exit(1);}
  S.top--;
  i=S.data[S.top+1];
  cout<<G->data[i]<<' ';
  count++;
  for(j=0;j<n;j++)
  if(G->edge[i][j]!=0)
  {D[j]--;
   if(D[j]==0)
```

```
      {if(S.top==S.max-1)
          {cout<<"Stack is full!\n";exit(1);}
        S.top++;
        S.data[S.top]=j;
 }}}
 if(count<n)
  cout<<"\nThere is a cycle.";
 free(D);
 free(S.data);
}
void main()
{cout<<"topSort.cpp 运行结果:\n";
 Graph G;int n=6,e=8;//n 为顶点数,e 为边数
 RCW rcw[8]={{'a','b',1},{'a','d',1},{'a','e',1},{'b','f',1},
                  {'c','b',1},{'c','f',1},{'e','d',1},{'e','f',1}};
 SetGraph(&G,n);
 MakeGraph(&G,rcw,n,e);
 TopSort(&G);
 free(G.data);//释放空间
 free(G.visited);
 for(int i=0;i<G.max;i++) free(G.edge[i]);
 free(G.edge);cin.get();}


/*程序功能：对有向图 G 进行拓朴排序，输出拓朴排序序列
#include“stdio.h”
#include“graph_define.h”
/*****************************************************************/
/*函 数 名：topo_sort                                              */
/*函数功能：拓扑排序                                               */
/*入口参数： a-- 邻接矩阵                                          */
/*vex-- 存放拓扑排序顶点序列                                       */
/*vexno -- 顶点序号                                                */
/*n-- 图的顶点数                                                   */
/*返 回 值：拓扑排序成功返回 1，否则返回 0                         */
```

```
/*************************************************************/
int topo_sort(int a[][MAX_VERTEX_NUM],int vex[], int vexno[], int n)
{
    int i,j,k,tag;
    if (n==0) return 1;
        for (i=0;i<n;i++)
        {
            tag=1;
            for (j=0;j<n;j++)
            if (a[j][i]!=0) tag=0;
                if (tag==1)
                {
                    j=0;
                    while (vex[j]!=-1) j++;
                    vex[j]=vexno[i];
            for (j=i;j<n-1;j++)
                for (k=0;k<n;k++)
                    a[j][k]=a[j+1][k];
            for (j=i;j<n-1;j++)
                for (k=0;k<n;k++)
                    a[k][j]=a[k][j+1];
            for (j=i;j<n-1;j++)
                vexno[j]=vexno[j+1];
                return topo_sort(a,vex, vexno, n-1);
                }
        }
        return 0;
}
/*************************************************************/
/*函 数 名：main                                             */
/*函数功能：主函数                                           */
/*入口参数：无                                               */
/*返 回 值：无                                               */
/*************************************************************/
voidmain()
```

```
{
    Lgraph *g;
    int vex[MAX_VERTEX_NUM], a[MAX_VERTEX_NUM][ MAX_VERTEX_NUM],
    vexno[MAX_VERTEX_NUM];
    int i,j,n,k;
    Edge_node *p;
    for (i=0;i< MAX_VERTEX_NUM;i++)
        vex[i]=-1;
        g=(Lgraph *)malloc(sizeof(Lgraph));
        create_graph1(g);/*建立有向图*/
        n=g->n;
    for(i=0; i<n;i++)
        for(j=0;j<n;j++)
            a[i][j]=0;
    for (i=0;i<n;i++)
    {
        p=g->adjlist[i].firstedge;
        while(p!=NULL)
        {
            j=p->adjvex;
            a[i][j]=p->weight;
            p=p->next;
        }
    }
    for (i=0;i<n;i++)
        vexno[i]=i;
        k=topo_sort(a,vex,vexno, n);
        if (k==0)
            printf("该图有回路，拓朴排序失败\n");
        else
        {
            printf("拓朴排序序列：");
    for(i=0;i<n-1;i++)
        printf(" %s->",g->adjlist[vex[i]].vertex);
        printf(" %s\n",g->adjlist[vex[n-1]].vertex);
```

```
        }
    }
```

5.4.5 运行结果

第二部分 综合篇

1.1 景区旅游信息管理系统

1.1.1 项目需求

在旅游景区，经常会遇到游客打听从一个景点到另一个景点的最短路径和最短距离，这类游客不喜欢按照导游图的线路来游览，而是挑选自己感兴趣的景点游览。为于帮助这类游客信息查询，就需要计算出所有景点之间最短路径和最短距离。算法采用迪杰斯特拉算法或弗洛伊德算法均可。建立一个景区旅游信息管理系统，实现的主要功能包括制订旅游景点导游线路策略和制订景区道路铺设策略。

任务中景点分布是一个无向带权连通图，图中边的权值是景点之间的距离。

（1）景区旅游信息管理系统中制订旅游景点导游线路策略，首先通过遍历景点，给出一个入口景点，建立一个导游线路图，导游线路图用有向图表示。遍历采用深度优先策略，这也比较符合游客心理。

（2）为了使导游线路图能够优化，可通过拓朴排序判断图中有无回路，若有回路，则打印输出回路中的景点，供人工优化。

（3）在导游线路图中，还为一些不愿按线路走的游客提供信息服务，比如从一个景点到另一个景点的最短路径和最短距离。在本线路图中将输出任意景点间的最短路径和最短距离。

（4）在景区建设中，道路建设是其中一个重要内容。道路建设首先要保证能连通所有景点，但又要花最小的代价，可以通过求最小生成树来解决这个问题。本任务中假设修建道路的代价只与它的里程相关。

因此归纳起来，本任务有如下功能模块：

创建景区景点分布图；

输出景区景点分布图（邻接矩阵）

输出导游线路图；

判断导游线路图有无回路；

求两个景点间的最短路径和最短距离；

输出道路修建规划图。

主程序用菜单选项供用户选择功能模块。

1.1.2 设计流程

主程序采用设计主菜单调用若干功能模块，同时在主程序中定义两个邻接链表类型变量 G 和 G1，作为调用子函数的参数。

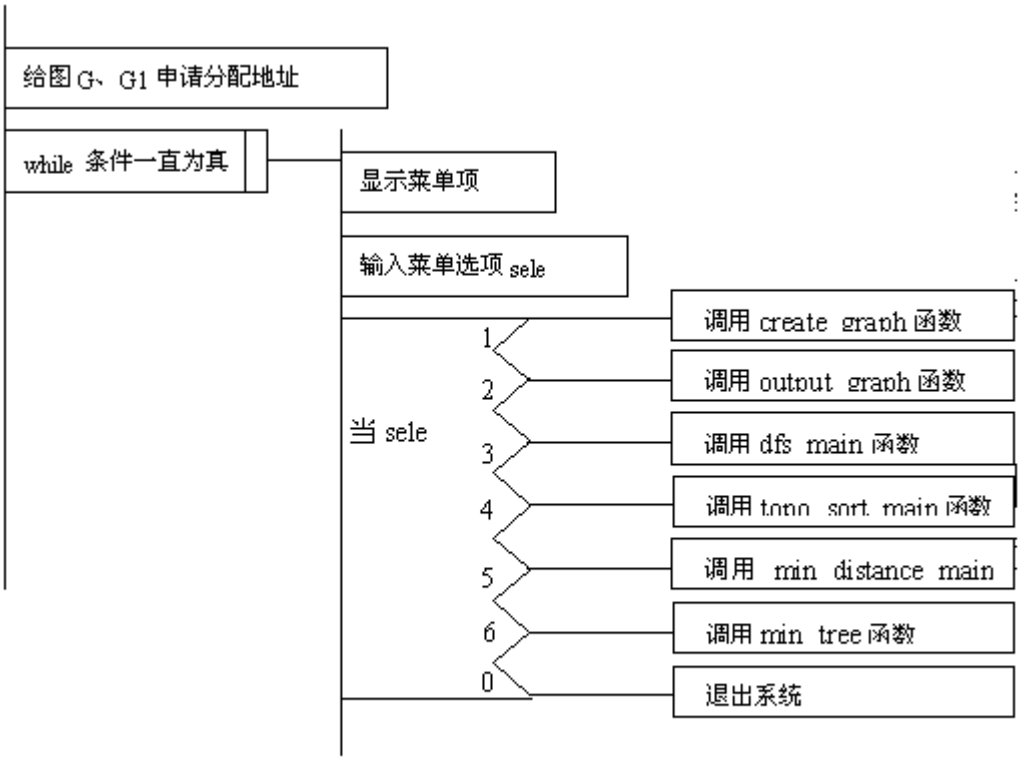

图 10-41 主程序流程图

建图子模块建立无向带权图，输入顶点信息和边的信息，输出邻接链表 G。由于是无向边，输入一条边时构建两条边。

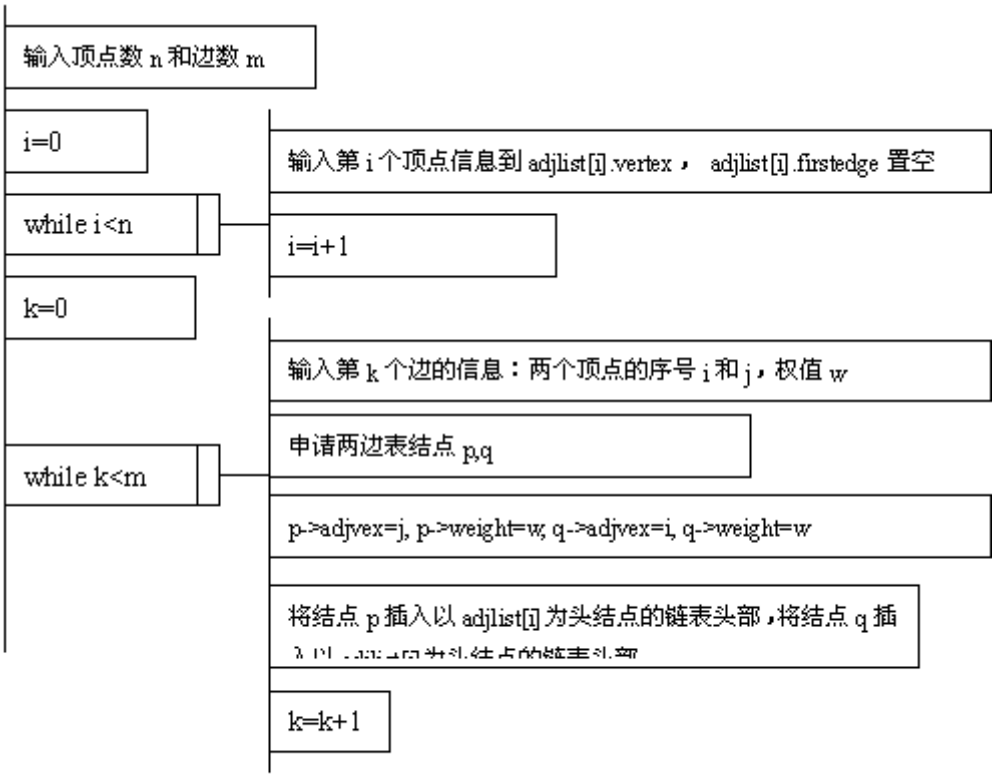

图 10-26　例 15-2 的流程图

输出图子模块：从邻接链表 g 转换成邻接矩阵 a，并输出邻接矩阵 a。图中边的权值∞用 32767 表示。

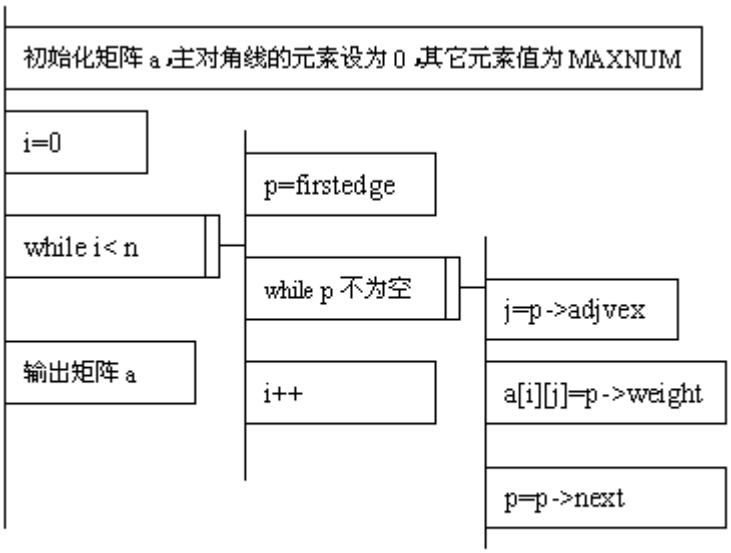

图 10-42 输出图邻接矩阵的流程图

遍历子模块：通过遍历图 G，只得到遍历的顶点序列。我们先将顶点序列存在数组 vex

中，然后再转换成导游线路存入数组 vex1 中，最后生成导游线路图 G1（同样用邻接链表存储，供拓朴排序用）。将遍历顶点序列转换成导游线路。

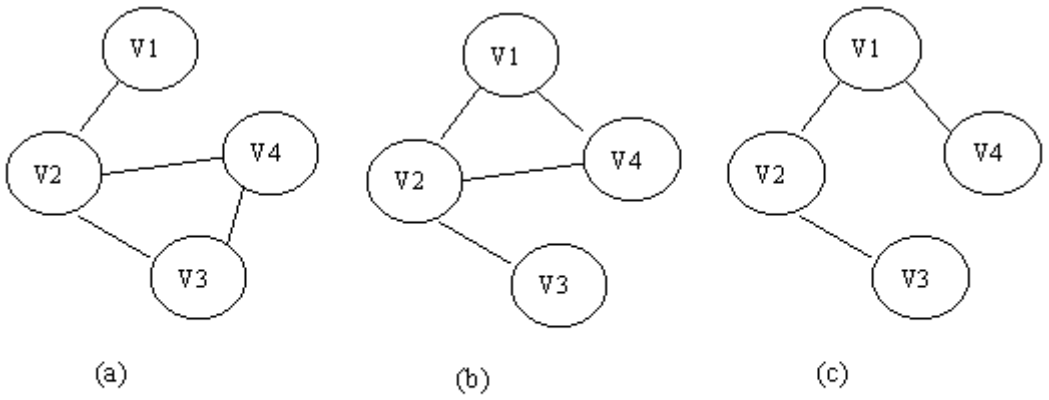

图 10-43 例 10-6 的无向图

图 10-43（a）（b）（c）三个无向图的深度优先搜索遍历的结果均为 v1→v2→v3→v4。但它们的导游线路图却不同。图（a）的导游线路图为 v1→v2→v3→v4，与遍历结果相同。

图（b）的导游线路图为 v1→v2→v3→v2→v4，图（c）的导游线路图为 v1→v2→v3→v2→v1→v4。

遍历结点序列与导游线路图转换的策略：

设遍历结果为 v1→v2→…→vi→vi+1→…→vn

对于结点 vi 和 vi+1，如果 vi 和 vi+1 存在边，则直接转换。

否则，加入边 vi→vi-1，如果 vi-1 和 vi+1 存在边，则加入边 vi-1→vi+1。

再否则，加入边 vi-1→vi-2，如果 vi-2 和 vi+1 存在边，则加入边 vi-2→vi+1。

如果 vi-2 和 vi+1 还不存在边，继续回溯，一定能找到某个整数 k（因为景点分布图是连通图），使得 vi-k 和 vi+1 存在边，则加入边 vi-k→vi+1。

在本任务中，转换后的线路图存于数组 vex1 中。

流程图见 10-29。

拓朴排序子模块流程图，见图 10-39 源程序，参见 10.7 节的 samp10-8.c。

求最短路径子模块流程图：见 10-34。源程序，参见 10.6 节的 samp10-6.c。

求最小生成树子模块流程图：见 19-33。源程序，参见 10.6 节的 samp10-5.c。

1.1.3 数据结构

景点的信息包括景点的名称和近邻景点之间的通路和距离。用邻接链表存储景点分布图的信息。

（带权无向）图的邻接链表

```
/*****************************************************************/
/*程序功能：建立一个旅游景区管理系统，实现旅游路线选择          */
/*景区道路优化等功能                                             */
/*****************************************************************/
#include“stdio.h”
#include“stdlib.h”
#include “string.h”
#define MAX_EDGE_NUM100/*定义图的最大边数*/
#define MAX_VERTEX_NUM 20
#define MAXNUM 32767
typedef char Vertex_type[10];
typedef struct node/*边表结点*/
{
    int adjvex;/*邻接点域*/
    int weight;
    struct node* next;/*指向下一个邻接点的指针域*/
}Edge_node;
typedef struct /*顶点表结点*/
{
    Vertex_type vertex;/*顶点域，存放景点名称*/
    Edge_node* firstedge;/*边表头指针*/
}Vertex_node;
typedef struct
{
    Vertex_node adjlist[MAX_VERTEX_NUM];/*邻接表*/
    int n,m;/*顶点数和边数*/
}Lgraph;
```

边的类型定义

在求最小生成树时，用到边的定义。

```
typedef struct
{
    inti; /*顶点 vi 的序号*/
    int j; /*顶点 vi 的序号*/
    intweight;
```

```
} Edge_type;
```

1.1.4 程序清单

主程序源程序

```
/*****************************************************************/
    /*函 数 名：main                                                 */
    /*入口参数：无                                                   */
    /*返 回 值：无                                                   */
    /*****************************************************************/
    voidmain()
    {
        Lgraph *g, *g1;
        int sele;
        void create_graph();
        void output_graph();
        void dfs_main();
        void topo_sort_main();
        void min_distance_main();
        void min_tree();
        g=(Lgraph *)malloc(sizeof(Lgraph));
        g->m=0;
        g1=(Lgraph *)malloc(sizeof(Lgraph));
        while(1)
        {
            system("cls");
            printf("\n\n******景区旅游管理信息系统******\n");
            printf("1.输入景点分布图\n");
            printf("2.输出景点分布图邻接矩阵\n");
            printf("3.生成导游线路图\n");
            printf("4.输出导游线路图中回路\n");
            printf("5.求两景点间最短路径和最短距离\n");
            printf("6.输出景区道路修建规划图\n");
            printf("0.退出\n");
            printf("请选择功能序号:");
```

```
            scanf(“%d”,&sele);
            printf("\n\n*****************************************************\n\n");
            switch(sele)
            {
                case 1: create_graph(g); break;
                case 2: output_graph(g);break;
                case 3: dfs_main(g,g1);break;
                case 4: topo_sort_main(g1);break;
                case 5: min_distance_main(g);break;
                case 6: min_tree(g); break;
                case 0: exit(0);
            }
            getchar();
            printf("按回车键继续.....");
            getchar();
        }
    }
```

建图子模块源程序参见 10.3 节的 create_graph()函数。

输出图子模块

```
/*****************************************************************/
    /*函 数 名：output_graph                                        */
    /*函数功能：输出图 G 的邻接矩阵                                  */
    /*入口参数：g --- 邻接链表                                       */
    /*返 回 值：无                                                  */
    /*****************************************************************/
    void output_graph(Lgraph *g)
    {
        int i,j,n;
        inta[MAX_VERTEX_NUM][ MAX_VERTEX_NUM];
        Edge_node *p;
        if(g->n==0)
        {
            printf(“景点分布图未输入，无法输出！\n”);
```

```
            return;
        }
        for(i=0;i<g->n;i++)
            for(j=0;j<g->n;j++)
                if (i==j)
                    a[i][j]=0;
                else
                    a[i][j]=MAXNUM;
        for (i=0;i< g->n;i++)
        {
            p=g->adjlist[i].firstedge;
            while (p!=NULL)
            {
                j=p->adjvex;
                a[i][j]=p->weight;
                p=p->next;
            }
        }
        printf("景点分布图邻接矩阵为:\n\n");
        printf("%8s "," ");
        for (i=0;i<g->n;i++)
            printf("%8s ",g->adjlist[i].vertex);
        for (i=0;i<g->n;i++)
            {printf("%8s ",g->adjlist[i].vertex);
            for(j=0;j<g->n;j++)
                printf("%9d",a[i][j]);
                printf("\n");
        }
    }
```

遍历子模块

```
/*****************************************************************/
    /*函 数 名：dfs_main                                          */
    /*函数功能：生成导游线路图                                     */
    /*入口参数：g-- 景点分布图                                     */
```

```
/*g1 — 导游线路图                                                    */
/*返 回 值：无                                                       */
/*****************************************************************/
voiddfs_main(Lgraph *g,Lgraph *g1)
{
    int visited[MAX_VERTEX_NUM];
    int x,i;
    int vex[MAX_VERTEX_NUM];
    int j, k, i1, tag;
    int vex1[MAX_VERTEX_NUM];
    Edge_node *p, *q;
    for (i=0;i< MAX_VERTEX_NUM;i++)
        visited[i]=0;
        if(g->n==0)
        {
            printf("景点分布图未输入，无法生成导游线图！\n");
            return;
        }
        do
        {
            printf("请输入口景点序号：");
            scanf("%d",&x);
            if (x>=1&&x<=g->n)
            {
                x--;
                break;
            }
            else
                printf("景点号输入有误，请重新输入！\n");
        }while(1);
        j=0;
        dfs(g,x,visited,vex,&j);//每次调用时，j 初始化为 0
        /*构建游览线路，存放在数组 Vex1*/
        i1=0;
    for(i=0;i< g->n-1;i++)
```

```
{
    j=vex[i+1];
    tag=1;
    k=0;
    while (tag)
    {
        vex1[i1++] = vex[i+k];
        p=g->adjlist[vex[i+k] ].firstedge;
        while (p!=NULL && p->adjvex!=j)/*判断 vi+k 与 vj 之间有没有边*/
        p=p->next;
        if (p==NULL)
            k--;/*若 vi+k 与 vj 之间没有边回溯*/
        else
            tag=0;
    }
}
vex1[i1++]=j;
/*建立游览线路图的邻接链表 G1，供拓朴排序用*/
for (i=0;i<g->n;i++)
{
    strcpy( g1->adjlist[i].vertex, g->adjlist[i].vertex );
    g1->adjlist[i].firstedge=NULL;
}
for (k=0;k<i1-1;k++)
{
    i=vex1[k];
    j=vex1[k+1];
    p=(Edge_node *)malloc(sizeof(Edge_node));
    p->adjvex=j;
    p-> weight=1; /*建立游览线路图时，不考虑边的权值*/
    q=g1->adjlist[i].firstedge;
    g1->adjlist[i].firstedge=p;
    p->next=q;
}
g1->n=g->n;
```

```
    g1->m=i1-1;
    /*输出游览线路*/
    printf("游览线路为\n：");
    for (k=0;k<i1-1;k++)
    {
        i=vex1[k];
        printf("%s->",g->adjlist[i].vertex);
    }
    printf("%s\n",g->adjlist[vex1[i1-1] ].vertex);
}
/*****************************************************************/
/*函 数 名：dfs                                                  */
/*函数功能：以 Vi 为出发点对邻接表存储的图 G 进行 DFS 搜索          */
/*入口参数：g-- 图 G 的邻接表存储                                 */
/*i— 顶点 Vi                                                     */
/*visited— 标志顶点是否已被访问的数组                            */
/*vex— 存放遍历时所经过的顶点                                    */
/*j— 存放位置                                                    */
/*返 回 值：无                                                   */
/*****************************************************************/
voiddfs(Lgraph *g,int i, int visited[], int vex[] .int *j)
{
    int k;
    static j=0;
    Edge_node *p;
    printf("visit vertex:V%s\n",g->adjlist[i].vertex);/*访问顶点 Vi*/
    vex[*j]=i; /*遍历结点 vi，存入数组 Vex 中*/
    *j=*j+1;
    visited[i]=1;/*标记 Vi 已访问*/
    p=g->adjlist[i].firstedge;/*取 Vi 边表的头指针*/
    while (p) /*依次搜索 Vi 的邻接点 Vj，j=p->adjva*/
    {
        k= p->adjvex;
        if (!visited[k])/*若 Vj 尚未访问，则以 Vj 为出发点向纵深搜索*/
            dfs(g,k, visited, vex,j);
```

```
            p=p->next;/*找 Vi 的下一个邻接点*/
        }
    }
```

在本任务中，通过 10.4 节的遍历输出景区旅游导游线路，该线路从入口出发，经过所有旅游景点后到达出口。

在输出的导游线图中，有可能出现回路，即一个景点经过两次及两次以上。为了实现导游线路图的优化，首先要判断图中有没有回路，若有，回路由哪几个景点组成。然后尽可能消除回路。比如说可以通过景区之间多修建道路来消除回路。当然，如果确实存在困难，不能彻底消除回路，也最好使回路经过的景点最少。

其中，判断导游线路图有无回路，可用拓朴排序来解决，若有回路则输出回路中的景点 。要实现这一功能，可直接使用上述程序，存在回路时，输出未能拓朴排序的顶点。

```
main()
    {
        Lgraph *g;
        int vex[MAX_VERTEX_NUM], a[MAX_VERTEX_NUM][ MAX_VERTEX_NUM],
            vexno[MAX_VERTEX_NUM];
        int i,j,n,k;
        Edge_node *p;
        for (i=0;i< MAX_VERTEX_NUM;i++)
            vex[i]=-1;
        g=(Lgraph *)malloc(sizeof(Lgraph));
        create_graph1(g);/*建立有向图*/
        n=g->n;
        for(i=0; i<n;i++)
            for(j=0;j<n;j++)
                a[i][j]=0;
        for (i=0;i<n;i++)
        {
            p=g->adjlist[i].firstedge;
            while(p!=NULL)
            {
                j=p->adjvex;
                a[i][j]=p->weight;
```

```
            p=p->next;
        }
    }
    for (i=0;i<n;i++)
        vexno[i]=i;
    k=topo_sort(a,vex,vexno, n);
    if(k==0)
    {
        printf("该图有回路，回路中的景点为：\n");
        for (i=0;i<n;i++)
        {
            tag=1;
            for (j=0;vex[j]!=-1&&tag;j++)
                if (i==vex[j])
                    tag=0;
            if (tag==1)
                printf(" %s",g->adjlist[vex[i]].vertex);
        }
    }
    else
    printf("该图没有回路");
}
```

1.1.5 运行测试