《算法与数据结构》实验指导书

2006 年 8 月 25 日

长江大学电子信息学院计算机系列课程组

目　录

0 绪　论

本书是以《算法与数据结构》课程教学大纲和实验教学大纲为指导，分别从实验（包含：基本实验和扩展实验）的六个方面（实验目的、实验内容、实验指导、参考程序、实验步骤以及思考题）来组织内容。其中实验目的强调了每个实验要掌握的内容和要达到的层次；实验内容是对基本实验和扩展实验的问题进行描述，并且对实现要求进行了规划；实验指导对整个实验的结构进行了组织和指导；而参考程序则列出了实验要求中，特别是一些重要算法的参考描述；实验步骤则是指导学生进行实验的具体操作；思考题则是布置给学生对同类实验的一种思考和提示。

在参考程序中，主要实现了数据结构定义，基本操作和算法的验证。其中：在所有实验中我们建议采用 Visual C++作为实验的开发环境,也可以用 BC++,TC++环境，如果用 C 环境，那么后面的参考程序中,凡是涉及到“引用”内容的，均要改为“指针”。用 VC++作为实验开发环境，首先建立 Win32 Console Application 工程,其目的是在 Windows 图形环境中模拟 DOS 字符方式。由于工程是以一组相关的文件组织的，将严格地按照文件内容的不同功能进行细致的划分，为了便于大家认真理解本指导书的内容，特对各类文件的名称作了如下的规定：

文件名的标识基本以每个实验的内容作为依据，如顺序表的存储结构定义文件为 SeqListDef.h。这些程序文件主要包含了如下 4 类：

（1） pubuse.h　是几乎所有实验中都涉及到的，包含了一些常量定义，系统函数原型声明和类型（Status）重定义,结果状态代码等。在本文中的内容对某一个实验可能没有完全用到，但对本课程的其他实验可能有用；

（2） 数据结构定义：以____Def.h 为文件名；

（3） 基本操作和算法：以____Algo.h 为文件名；

（4） 调用基本操作的主程序：以_____Use.cpp 为文件名。

实验一：线性表的存储结构定义及基本操作（必做：基本 2 学时，扩展 4 学时）

一、实验目的：

. 掌握线性表的逻辑特征

. 掌握线性表顺序存储结构的特点，熟练掌握顺序表的基本运算

. 熟练掌握线性表的链式存储结构定义及基本操作

. 理解循环链表和双链表的特点和基本运算

. 加深对顺序存储数据结构的理解和链式存储数据结构的理解，逐步培养解决实际问题的编程能力

二、实验内容：

（一）基本实验内容（顺序表）：

建立顺序表，完成顺序表的基本操作：初始化、插入、删除、逆转、输出、销毁，置空表、求表长、查找元素、判线性表是否为空；

1. 问题描述：利用顺序表,设计一组输入数据（假定为一组整数），能够对顺序表进行如下操作：

. 创建一个新的顺序表，实现动态空间分配的初始化；

. 根据顺序表结点的位置插入一个新结点(位置插入)，也可以根据给定的值进行插入(值插入)，形成有序顺序表；

. 根据顺序表结点的位置删除一个结点(位置删除)，也可以根据给定的值删除对应的第一个结点，或者删除指定值的所有结点(值删除)；

. 利用最少的空间实现顺序表元素的逆转；

. 实现顺序表的各个元素的输出；

. 彻底销毁顺序线性表，回收所分配的空间；

. 对顺序线性表的所有元素删除，置为空表；

. 返回其数据元素个数；

. 按序号查找，根据顺序表的特点，可以随机存取，直接可以定位于第 i 个结点，查找该元素的值，对查找结果进行返回；

. 按值查找，根据给定数据元素的值，只能顺序比较，查找该元素的位置，对查找结果进行返回；

. 判断顺序表中是否有元素存在，对判断结果进行返回；

. 编写主程序，实现对各不同的算法调用。

2．实现要求：

对顺序表的各项操作一定要编写成为C（C++）语言函数，组合成模块化的形式，每个算法的实现要从时间复杂度和空间复杂度上进行评价；

. “初始化算法”的操作结果：构造一个空的顺序线性表。对顺序表的空间进行动态管理，实现动态分配、回收和增加存储空间；

. “位置插入算法” 的初始条件：顺序线性表L已存在，给定的元素位置为i，且1≤i≤ListLength(L)+1 ；

操作结果：在L中第i个位置之前插入新的数据元素e，L的长度加1；

. “位置删除算法”的初始条件：顺序线性表L已存在，1≤i≤ListLength(L) ；

操作结果：删除L的第i个数据元素，并用e返回其值，L的长度减1 ；

. “逆转算法”的初始条件：顺序线性表L已存在；

操作结果：依次对L的每个数据元素进行交换，为了使用最少的额外空间，对顺序表的元素进行交换；

. “输出算法”的初始条件：顺序线性表L已存在；

操作结果：依次对L的每个数据元素进行输出；

. “销毁算法”初始条件：顺序线性表L已存在；

操作结果：销毁顺序线性表L；

. “置空表算法”初始条件：顺序线性表L已存在；

操作结果：将L重置为空表；

. “求表长算法”初始条件：顺序线性表L已存在；

操作结果：返回L中数据元素个数；

. “按序号查找算法”初始条件：顺序线性表L已存在，元素位置为i，且1≤i≤ListLength(L)

操作结果：返回L中第i个数据元素的值

. “按值查找算法”初始条件：顺序线性表L已存在，元素值为e；

操作结果：返回L中数据元素值为e的元素位置；

. “判表空算法”初始条件：顺序线性表L已存在；

操作结果：若L为空表，则返回TRUE，否则返回FALSE；

分析：修改输入数据，预期输出并验证输出的结果，加深对有关算法的理解。

(二）基本实验内容（链表）：

建立单链表，完成链表（带表头结点）的基本操作：建立链表、插入、删除、查找、输出、求前驱、

求后继、两个有序链表的合并操作。

其他基本操作还有销毁链表、将链表置为空表、求链表的长度、获取某位置结点的内容、搜索结点。

1. 问题描述：

利用线性表的链式存储结构,设计一组输入数据（假定为一组整数），能够对单链表进行如下操作：

. 初始化一个带表头结点的空链表；

. 创建一个单链表是从无到有地建立起一个链表，即一个一个地输入各结点数据，并建立起前后相互链接的关系。又分为逆位序(插在表头)输入 n 个元素的值和正位序(插在表尾)输入 n 个元素的值；

. 插入结点可以根据给定位置进行插入（位置插入），也可以根据结点的值插入到已知的链表中（值插入），且保持结点的数据按原来的递增次序排列，形成有序链表。

. 删除结点可以根据给定位置进行删除（位置删除），也可以把链表中查找结点的值为搜索对象的结点全部删除（值删除）；

. 输出单链表的内容是将链表中各结点的数据依次显示，直到链表尾结点；

. 编写主程序，实现对各不同的算法调用。

其它的操作算法描述略。

2. 实现要求：

对链表的各项操作一定要编写成为 C（C++）语言函数，组合成模块化的形式，还要针对每个算法的实现从时间复杂度和空间复杂度上进行评价。

- “初始化算法”的操作结果：构造一个空的线性表 L，产生头结点,并使 L 指向此头结点；
- **“**建立链表算法” 初始条件：空链存在；

  操作结果：选择逆位序或正位序的方法，建立一个单链表，并且返回完成的结果；
- “链表（位置）插入算法”初始条件：已知单链表 L 存在；

  操作结果：在带头结点的单链线性表 L 中第 i 个位置之前插入元素 e；
- “链表（位置）删除算法”初始条件：已知单链表 L 存在；

  操作结果：在带头结点的单链线性表 L 中，删除第 i 个元素,并由 e 返回其值；

. “输出算法”初始条件：链表 L 已存在；

操作结果：依次输出链表的各个结点的值；

（三）扩展实验内容（顺序表）

查前驱元素、查后继元素、顺序表合并等.

1. 问题描述：

. 根据给定元素的值，求出前驱元素；

. 根据给定元素的值，求出后继元素；

. 对已建好的两个顺序表进行合并操作，若原线性表中元素非递减有序排列,要求合并后的结果还是有序(有序合并)；对于原顺序表中元素无序排列的合并只是完成 A=A∪B（无序合并），要求同样的数据元素只出现一次。

. 修改主程序，实现对各不同的算法调用。

2．实现要求：

. “查前驱元素算法” 初始条件：顺序线性表 L 已存在 ；

操作结果：若数据元素存在且不是第一个，则返回前驱，否则操作失败；

. “查后继元素算法” 初始条件：顺序线性表 L 已存在 ；

操作结果：若数据元素存在且不是最后一个，则返回后继，否则操作失败；

. “无序合并算法”的初始条件：已知线性表 La 和 Lb；

操作结果：将所有在线性表 Lb 中但不在 La 中的数据元素插入到 La 中；

. “有序合并算法”的初始条件： 已知线性表 La 和 Lb 中的数据元素按值非递减排列；

操作结果： 归并 La 和 Lb 得到新的线性表 Lc,Lc 的数据元素也按值非递减排列；

（四）扩展实验内容（链表）

1．问题描述：

. 求前驱结点是根据给定结点的值，在单链表中搜索其当前结点的后继结点值为给定的值，将当前结点返回；

. 求后继结点是根据给定结点的值，在单链表中搜索其当前结点的值为给定的值，将后继结点返回；

. 两个有序链表的合并是分别将两个单链表的结点依次插入到第 3 个单链表中，继续保持结点有序；

2．实现要求：

. “求前驱算法”初始条件: 线性表 L 已存在；

操作结果: 若 cur_e 是 L 的数据元素,且不是第一个,则用 pre_e 返回它的前驱；

. “求后继算法”初始条件: 线性表 L 已存在；

操作结果: 若 cur_e 是 L 的数据元素,且不是最后一个,则用 next_e 返回它的后继；

. “两个有序链表的合并算法”初始条件: 线性表单链线性表 La 和 Lb 的元素按值非递减排列；

操作结果:归并 La 和 Lb 得到新的单链表。

三、实验指导

一个简单程序通常主要由三部分构成：

1、常量定义（#define），类型重定义(typedef)及函数原型(#include)声明；还有针对每一种数据结

构的类型定义，由于本实验是要求实现对线性表的顺序存储和链式存储两种存储结构的定义，因此不同的物理存储结构的定义和其基本操作最好是以独立的文件存在，因此本实验将顺序表和链表可分解为两个不同的实验部分。

2、算法函数，对于顺序表，每一个函数具有独立的功能，组合成为模块的形式，如 ListInit_Sq、ListInsert_Sq、ListDelete_Sq、 ListReverse_Sq、ListPrint_Sq 等；对于链表，每一个函数具有独立的功能，组合成为模块的形式，如 ListInit_Link、ListInsert_Link、ListDelete_Link、ListTraverse_Link、PriorElem_Link、NextElem_Link、GetElem_Link、MergeList_Link 等；

3、主函数（main）。

【说明 1】——顺序表的定义与操作

1．可以将这三部分组合在一个文件中，**也可以按照项目的方式（多个不同的文件组合在一起，共同完成一个功能）进行组织（本课程的实验强烈推荐这种形式**），如将本课程后续所有算法几乎都要使用的常量定义(#define)和系统函数原型定义(#include)声明组合成一个文件，存储为一个头文件(**取名为 pubuse.h**)，只需建立一次，以后凡涉及到相关的内容，只需在你的文件的前面加上一条#include “pubuse. h” 即可，无需重复输入。

2．对于类型定义，由于每一种数据结构的定义都不一样，因此要进行专门的类型定义，也可以将数据结构的定义组合成为一个文件，存储为一个头文件（如线性表的顺序存储结构定义取名为 seqlistDef. h），只需建立一次，以后凡涉及到关于顺序表操作的相关内容，一定要在你的文件的前面加上一条#include “seqlistDef. h” 即可，无需重复进行顺序存储结构的定义。

3．关于实验内容要完成的操作，一般都以独立的算法出现，关于某类数据结构的各种不同算法函数组合在一个文件之中，存储为一个头文件（如取名为 seqlistAlgo. h），以后凡涉及到要使用本文件中的顺序表算法，一定要在你的文件的前面加上一条#include “seqlistAlgo. h” 即可，无需重复定义类似的算法，就象使用系统函数一样，只需进行函数原型的声明即可。

4．还应包含一个 main 函数，作为整个程序的执行入口处，定义测试的数据，完成各种算法的调用执行，对算法的执行过程和结果进行判断和输出控制，存储为一个源文件（如取名为 seqlistUse. cpp）。

5．为了对实际问题更加通用和适应，在算法中使用的数据类型为 ElemType，为了与实际数据类型一致，一般要将 ElemType 用 typedef 重定义：如实际问题中顺序表的每个元素是整型，则使用 typedef int ElemType；定义语句；如实际问题中顺序表的每个元素是单精度实数的话，使用 typedef float ElemType；定义语句。

【说明 2】——链表的定义与操作

1．根据一个完整的 C 语言组成，将实验内容组合成如上所述的三个组成部分，功能清晰明了，其常

量定义和常用系统函数原型声明的文件“pubuse.h”代码复用，不需另行建立。

2．建立一个单链表类型定义的头文件，如取名为：linklistDef.h，以后凡使用该文件定义的类型，就只需使用包含语句#include “linklistDef.h”即可。

3．关于实验内容要完成的操作，一般都以独立的算法出现，建立一个单链表的基本算法文件,将各种不同算法组合在一个文件之中，存储为一个头文件如取名为：linklistAlgo.h，后凡涉及到要使用本文件中的单链表算法，一定要在你的文件的前面加上一条#include “linklistAlgo.h” 即可，实现的算法函数，如ListInit_Link、ListInsert_Link、ListDelete_Link、 ListTraverse_Link、PriorElem_Link、NextElem_Link、GetElem_Link、MergeList_Link 等；

4．还应包含一个 main 函数，作为整个程序的执行入口处，定义测试的数据，完成各种算法的调用执行，对算法的执行过程和结果进行判断和输出控制，存储为一个源文件（如取名为 linklistUse. cpp）。

四、基本实验的参考程序

（一）线性表的顺序存储结构的定义及其基本操作的参考程序（顺序表）

(1) 文件 1： pubuse. h 是公共使用的常量定义和系统函数调用声明，以后每个实验中几乎都涉及到此文件。

```
#include<string. h>
#include<ctype. h>
#include<malloc. h>              /* malloc()等  */
#include<limits. h>              /* INT_MAX 等  */
#include<stdio. h>               /* EOF(=^Z 或 F6),NULL */
#include<stdlib. h>              /* atoi() */
#include<io. h>                  /* eof() */
#include<math. h>                /* floor(),ceil(),abs() */
#include<process. h>             /* exit() */
/*  函数结果状态代码  */
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
/* #define OVERFLOW -2  因为在 math. h 中已定义 OVERFLOW 的值为 3,故去掉此行  */
```

```
typedef int Status;          /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */
typedef int Boolean;         /* Boolean 是布尔类型,其值是 TRUE 或 FALSE */
```

(2) 文件 2：seqlistDef. h 进行线性表的动态分配顺序存储结构的表示

```
#define LIST_INIT_SIZE 10 /* 线性表存储空间的初始分配量 */
#define LISTINCREMENT 2 /* 线性表存储空间的分配增量 */
typedef struct
{
   ElemType *elem;    /* 存储空间基址 */
   int length;        /* 当前长度 */
   int listsize;      /* 当前分配的存储容量(以 sizeof(ElemType)为单位) */
}SqList;
```

(3)文件 3：seqlistAlgo. h 进行线性表顺序存储结构的基本实验算法定义

```
Status ListInit_Sq(SqList &L)              /* 算法 2. 3 */
{ /* 操作结果：构造一个空的顺序线性表 */
   L. elem=(ElemType*)malloc(LIST_INIT_SIZE*sizeof(ElemType));
   if(!L. elem)
      exit(OVERFLOW);  /* 存储分配失败 */
   L. length=0;  /* 空表长度为 0 */
   L. listsize=LIST_INIT_SIZE;  /* 初始存储容量 */
   return OK;
}
Status ListInsert_Sq(SqList &L,int i,ElemType e)    /* 算法 2. 4 */
{ /* 初始条件：顺序线性表 L 已存在，1≤i≤ListLength(L)+1 */
   /* 操作结果：在 L 中第 i 个位置之前插入新的数据元素 e，L 的长度加 1 */
   ElemType *newbase,*q,*p;
   if(i<1||i>L. length+1) /* i 值不合法 */
      return ERROR;
   if(L. length>=L. listsize) /* 当前存储空间已满,增加分配 */
   {
      newbase=(ElemType *)realloc(L. elem,(L. listsize+LISTINCREMENT)*sizeof(ElemType));
      if(!newbase)
```

```
        exit(OVERFLOW);                 /* 存储分配失败 */
      L. elem=newbase;                  /* 新基址 */
      L. listsize+=LISTINCREMENT;       /* 增加存储容量 */
    }
    q=L. elem+i-1;                      /* q 为插入位置 */
    for(p=L. elem+L. length-1; p>=q; --p) /* 插入位置及之后的元素右移 */
      *(p+1)=*p;
    *q=e;                               /* 插入 e */
    ++L. length;                        /* 表长增 1 */
    return OK;
}

Status ListDelete_Sq(SqList &L,int i,ElemType *e)       /* 算法 2. 5 */
{ /* 初始条件：顺序线性表 L 已存在，1≤i≤ListLength(L) */
    /* 操作结果：删除 L 的第 i 个数据元素，并用 e 返回其值，L 的长度减 1 */
    ElemType *p,*q;
    if(i<1||i>L. length)                /* i 值不合法 */
      return ERROR;
    p=L. elem+i-1;                      /* p 为被删除元素的位置 */
    *e=*p;                              /* 被删除元素的值赋给 e */
    q=L. elem+L. length-1;              /* 表尾元素的位置 */
    for(++p; p<=q; ++p)                 /* 被删除元素之后的元素左移 */
      *(p-1)=*p;
    L. length--;                        /* 表长减 1 */
    return OK;
}

Status ListReverse_Sq(SqList &L)
{ /* 初始条件：顺序线性表 L 已存在 */
    /* 操作结果：依次对 L 的数据元素成对交换*/
    ElemType    t;
```

```
    int i;
    for(i=0；i<L. length/2；i++)
       {t=L. elem[i]；  L. elem[i]= L. elem[L. length-i-1]；  L. elem[L. length-i-1]=t；    }
    printf("\n")；
    return OK；
 }

Status ListPrint_Sq(SqList L)
{   /* 初始条件：顺序线性表 L 已存在 */
    /* 操作结果：依次对 L 的数据元素输出*/
    int i；
    printf("\n")；
    for(i=0；i<L. length；i++)
       printf("%d   ", L. elem[i])；
    return OK；
}
/*   还有一些算法，同学们自己补充完整    */
```

(4)文件 4：seqlistUse. cpp　进行线性表顺序存储结构的基本算法验证

```
#include"pubuse. h"          /* 实现通用常量的定义，常用系统函数的声明 */
typedef int ElemType；        /*实现一组整数的操作,将 int 型特定义为通用的 ElemType 类型名*/
#include"seqlistDef. h"      /* 采用线性表的动态分配顺序存储结构定义 */
#include"seqlistAlgo. h"     /* 采用顺序表的基本算法定义 */

void main()
 {
    SqList L；
    Status i；
    int j；
    ElemType t；
    /*    首先一定要初始化顺序表   */
    i=ListInit_Sq(L)；
```

```
    if(i==1) /* 创建空表 L 成功 */
    for(j=1；j<=5；j++) /* 在表 L 中插入 5 个元素，每个元素的值分别为 2，4，6，8，10 */
            i=ListInsert_Sq(L,j,2*j)；
     ListPrint_Sq(L)；    /*检验一下插入的结果，输出表 L 的内容 */

     ListInsert_Sq(L,2,20)；/* 随机指定插入点位置，假设在第二个元素前插入新的元素，其值为 20 */
     ListDelete_Sq(L,4,&t)；/* 随机指定删除点位置，假设对第四个元素进行删除 */
     printf("\n The Deleted value is %d",t)；/* 检验一下删除点元素的值 */
     ListPrint_Sq(L)；/* 检验一下插入和删除后的结果，输出表 La 的内容 */
     ListReverse_Sq(L)；/* 将顺序表 La 的所有元素进行逆序    */
     ListPrint_Sq(L)；/* 检验一下逆序的结果，输出表 L 的内容 */
}
```

（二）线性表的链式存储结构的定义及其基本操作的参考程序（链表）

1. 文件 linklistDef.h 是线性表的单链表存储结构的定义

```
struct LNode
 {
   ElemType data;
   struct LNode *next;
 };
 typedef struct LNode *LinkList;  /* 另一种定义 LinkList 的方法 */
```

2. 文件 linklistAlgo.h 是单链表的基本算法描述

```
/*  以下的算法均是针对带表头结点的单链表  */
Status ListInit_Link(LinkList &L)
 { /* 操作结果：构造一个空的线性表 L */
    L=(LinkList)malloc(sizeof(struct LNode))；  /* 产生头结点,并使 L 指向此头结点 */
    if(!L) /* 存储分配失败 */
      exit(OVERFLOW)；
    L->next=NULL；  /* 指针域为空 */
    return OK；
 }
```

```
Status ListDestroy_Link (LinkList L)
{ /* 初始条件：线性表 L 已存在。*/
/* 操作结果：销毁线性表 L */
    LinkList q;
    while(L)
    {
      q=L->next;
      free(L);
      L=q;
    }
    return OK;
}

Status ListClear_Link (LinkList L)          /* 不改变 L */
{ /* 初始条件：线性表 L 已存在。*/
/*  操作结果：将 L 重置为空表 */
    LinkList p,q;
    p=L->next;  /* p 指向第一个结点 */
    while(p) /* 没到表尾 */
    {
      q=p->next;
      free(p);
      p=q;
    }
    L->next=NULL;  /* 头结点指针域为空 */
    return OK;
}

Status ListEmpty_Link (LinkList L)
{ /* 初始条件：线性表 L 已存在。*/
  /* 操作结果：若 L 为空表，则返回 TRUE，否则返回 FALSE */
```

```
    if(L->next) /*  非空  */
      return FALSE；
    else
      return TRUE；
 }

 int ListLength_Link (LinkList L)
 { /*  初始条件：线性表 L 已存在。*/
/*  操作结果：返回 L 中数据元素个数  */
    int i=0；
    LinkList p=L->next；   /* p 指向第一个结点  */
    while(p) /*  没到表尾  */
    {
      i++；
      p=p->next；
    }
    return i；
 }

 Status GetElem_Link (LinkList L,int i,ElemType &e)          /*  算法 2.8 */
 { /*  初始条件: L 为带头结点的单链表的头指针*/
/*  操作结果：当第 i 个元素存在时,其值赋给 e 并返回 OK,否则返回 ERROR */
    int j=1；   /* j 为计数器  */
    LinkList p=L->next；   /* p 指向第一个结点  */
    while(p&&j<i) /*  顺指针向后查找,直到 p 指向第 i 个元素或 p 为空  */
    {
      p=p->next；
      j++；
    }
    if(!p||j>i)                    /*  第 i 个元素不存在  */
      return ERROR；
```

```
    e=p->data;                /* 取第 i 个元素 */
  return OK;
}

int LocateElem_Link (LinkList L,ElemType e,Status(*compare)(ElemType,ElemType))
{ /* 初始条件: 线性表 L 已存在,compare()是数据元素判定函数(满足为 1,否则为 0) */
  /* 操作结果: 返回 L 中第 1 个与 e 满足关系 compare()的数据元素的位序。 */
  /*                若这样的数据元素不存在,则返回值为 0 */
  int i=0;
  LinkList p=L->next;
  while(p)
  {
    i++;
    if(compare(p->data,e)) /* 找到这样的数据元素 */
      return i;
    p=p->next;
  }
  return 0;
}

Status ListInsert_Link (LinkList L,int i,ElemType e)    /* 算法 2.9,不改变 L */
{ /* 在带头结点的单链线性表 L 中第 i 个位置之前插入元素 e */
  int j=0;
  LinkList p=L,s;
  while(p&&j<i-1) /* 寻找第 i-1 个结点 */
  {
    p=p->next;
    j++;
  }
  if(!p||j>i-1) /* i 小于 1 或者大于表长 */
    return ERROR;
```

```
    s=(LinkList)malloc(sizeof(struct LNode)); /* 生成新结点 */
    s->data=e; /* 插入 L 中 */
    s->next=p->next;
    p->next=s;
    return OK;
}

Status ListDelete_Link (LinkList L,int i,ElemType &e) /* 算法 2.10,不改变 L */
{ /* 在带头结点的单链线性表 L 中，删除第 i 个元素,并由 e 返回其值 */
    int j=0;
    LinkList p=L,q;
    while(p->next&&j<i-1) /* 寻找第 i 个结点,并令 p 指向其前趋 */
    {
      p=p->next;
      j++;
    }
    if(!p->next||j>i-1) /* 删除位置不合理 */
      return ERROR;
    q=p->next; /* 删除并释放结点 */
    p->next=q->next;
    e=q->data;
    free(q);
    return OK;
}

Status ListTraverse_Link (LinkList L)
 { /* 初始条件：线性表 L 已存在 */
    /* 操作结果:依次对 L 的每个数据元素的值进行输出 */
    LinkList p=L->next;
    while(p)
    {
```

```
        printf("%d",p->data);
        p=p->next;
    }
    printf("\n");
    return OK;
}
void CreateList_Link (LinkList &L,int n) /* 算法 2.11 */
{ /* 逆位序(插在表头)输入 n 个元素的值，建立带表头结构的单链线性表 L */
    int i;
    LinkList p;
    L=(LinkList)malloc(sizeof(struct LNode));
    L->next=NULL;  /* 先建立一个带头结点的单链表 */
    printf("请输入%d 个数据\n",n);
    for(i=n; i>0; --i)
    {
        p=(LinkList)malloc(sizeof(struct LNode));  /* 生成新结点 */
        scanf("%d",&p->data);  /* 输入元素值 */
        p->next=L->next;  /* 插入到表头 */
        L->next=p;
    }
}

void CreateList2_Link (LinkList &L,int n)
{ /* 正位序(插在表尾)输入 n 个元素的值，建立带表头结构的单链线性表 */
    int i;
    LinkList p,q;
    L=(LinkList)malloc(sizeof(struct LNode));  /* 生成头结点 */
    L->next=NULL;
    q=L;
    printf("请输入%d 个数据\n",n);
    for(i=1; i<=n; i++)
```

```
    {
      p=(LinkList)malloc(sizeof(struct LNode))；
      scanf("%d",&p->data)；
      q->next=p；
      q=q->next；
    }
   p->next=NULL；
}

Status PriorElem_Link (LinkList L,ElemType cur_e,ElemType &pre_e)     /* 为扩展实验的内容 */
{ /* 初始条件: 线性表 L 已存在 */
   /* 操作结果: 若 cur_e 是 L 的数据元素,且不是第一个,则用 pre_e 返回它的前驱, */
   /*           返回 OK；否则操作失败,pre_e 无定义,返回 INFEASIBLE */
   LinkList q,p=L->next；  /* p 指向第一个结点 */
   while(p->next) /* p 所指结点有后继 */
   {
      q=p->next；  /* q 为 p 的后继 */
      if(q->data==cur_e)
      {
         pre_e=p->data；
         return OK；
      }
      p=q；  /* p 向后移 */
   }
   return INFEASIBLE；
}

Status NextElem_Link (LinkList L,ElemType cur_e,ElemType &next_e)     /* 为扩展实验的内容 */
{ /* 初始条件：线性表 L 已存在 */
   /* 操作结果：若 cur_e 是 L 的数据元素，且不是最后一个，则用 next_e 返回它的后继， */
   /*   返回 OK；否则操作失败，next_e 无定义，返回 INFEASIBLE */
```

```
  LinkList p=L->next;  /* p 指向第一个结点 */
  while(p->next) /* p 所指结点有后继 */
  {
    if(p->data==cur_e)
    { next_e=p->next->data;
      return OK;
    }
    p=p->next;
  }
  return INFEASIBLE;
}

void MergeList_Link (LinkList &La,LinkList &Lb,LinkList &Lc)    /* 算法 2.12 为扩展实验的内容*/
{ /* 已知单链线性表 La 和 Lb 的元素按值非递减排列。 */
  /* 归并 La 和 Lb 得到新的单链线性表 Lc，Lc 的元素也按值非递减排列 */
  LinkList pa=La->next,pb=Lb->next,pc;
  Lc=pc=La;      /* 用 La 的头结点作为 Lc 的头结点 */
  while(pa&&pb)
    if(pa->data<=pb->data)
    {
      pc->next=pa;
      pc=pa;
      pa=pa->next;
    }
    else
    {
      pc->next=pb;
      pc=pb;
      pb=pb->next;
    }
  pc->next=pa?pa:pb;       /* 插入剩余段 */
```

```
    free(Lb);                /* 释放 Lb 的头结点 */
    Lb=NULL;
 }
```

3．将测试数据和主函数新定义一个文件,如取名为:linlistUse.cpp.

```
/* linlistUse.cpp 主要实现算法2.9、2.10、2.11、2.12的程序 */
#include"pubuse.h"
 typedef int ElemType;
 #include"linklistDef.h"
 #include"linklistAlgo.h"

 void main()
 {
   int n=5;
   LinkList La,Lb,Lc;
   int i;
   ElemType e;

   printf("按非递减顺序, ");
   CreateList2_Link (La,n);  /* 正位序输入 n 个元素的值 ,建立一个单链表*/
   printf("La=");            /* 输出链表 La 的内容 */
   ListTraverse_Link (La);

   printf("按非递增顺序, ");
   CreateList_Link (Lb,n);  /* 逆位序输入 n 个元素的值 */
   printf("Lb=");           /* 输出链表 Lb 的内容 */
   ListTraverse_Link (Lb);

   MergeList_Link (La,Lb,Lc);  /* 按非递减顺序归并 La 和 Lb,得到新表 Lc */

   printf("Lc=");              /* 输出链表 Lc 的内容 */
```

```
    ListTraverse_Link (Lc);

    /*  算法 2.9 的测试 */
    printf("\n enter insert location and value : ");
    scanf("%d %d",&i,&e);
    ListInsert_Link (Lc,i,e);
    /* 在 Lc 链表中第 i 个结点处插入一个新的结点，其值为 e; */
    printf("Lc=");              /* 输出链表 Lc 的内容 */
    ListTraverse_Link (Lc);

    /*  算法 2.10 的测试 */
    printf("\n enter deletd location : ");
    scanf("%d",&i);
    ListDelete_Link (Lc,i,e);          /* 在 Lc 链表中删除第 i 个结点，其值为返回给 e 变量*/
    printf("The Deleted e=%d\n",e);    /* 输出被删结点的内容 */
    printf("Lc=");                     /* 输出链表 Lc 的内容 */
    ListTraverse_Link (Lc);
}
```

五、实验环境和实验步骤

实验环境： 利用 Visual C++集成开发环境进行本实验的操作。

实验步骤－－顺序表的定义与操作：

1．启动 VC++；

2. 新建工程/Win32 Console Application，选择输入位置：如 “d:\”，输入工程的名称：如 “SeqlistDemo”；按“确定”按钮，选择“An Empty Project”，再按“完成”按钮，

3．新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“pubuse. h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

4. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“seqlistDef. h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

5. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“seqlistAlgo. h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

6. 新建文件/C++ Source File，选中“添加到工程的复选按钮”，输入文件名“seqlistUse. cpp”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

7．按 F7 键，或  工具图标进行工程的建立，如有错误，根据错误显示区中的提示，改正错误，重新建立应用程序；

8．按 Ctrl+F5 键，或  工具图标进行工程的执行。

实验步骤－－链表的定义与操作：

1．启动 VC++;

2.新建工程/Win32 Console Application，选择输入位置：如 “d:\”，输入工程的名称：如 “linklistDemo”；按“确定”按钮，选择“An Empty Project”，再按“完成”按钮，

3．加载实验一中的 pubuse.h 选中菜单的“project”－>“add to project” --->“files”选择已存在文件，确定，然后一定将文件 pubuse.h 拷贝到所建的工程目录下；

4. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“linklistDef.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

5. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“linklistAlgo.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

6. 新建文件/C++ Source File，选中“添加到工程的复选按钮”，输入文件名“linklistUse.cpp”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

7．构件、调试、运行与实验一相同，在此不再复述；

六、思考题

1．如果将所有的常量定义、系统函数声明、类型定义、算法定义以及主函数全部写在一个文件中，试比较其文件的组织方式。

2．如果将顺序表的存储分配由动态分配设计为静态分配，其算法该如何修改，比较动态和静态分配的结果对那些算法的操作有影响？

3．要求以较高的效率实现删除线性表中元素值在 x 到 y(X 和 Y 自定)之间的所有元素，写出算法。

【参考解题思路】在线性表中设置两个初值为 O 的下标变量 i 和 j，其中，i 为比较元素的下标，j 为赋值元素的下标。依次取线性表中下标为 i 的元素与 x 和 y 比较，假若是 x 到 y 之外的元素，则赋值给下标为

J 的元素。这种算法比删除一个元素后立即移动其后面的元素的效率高得多。

4．利用单向链表完成一个班级的一个学期的所有课程的管理：能够增加、删除、修改学生的成绩记录。

5．对于同样的一组整数，比较线性表的两种不同存储结构的特点和使用场合。

实验二：线性表的综合应用（选做： 2学时）

一、实验目的：

. 掌握顺序表和链表的概念，学会对问题进行分析，选择恰当的逻辑结构和物理结构

. 加深对顺序表和链表的理解，培养解决实际问题的编程能力

二、实验内容:

实现一元稀疏多项式的表示及基本操作（建立、销毁、输出、加法、减法、乘法等操作）；

1．问题描述：

. 一元多项式一定要包含系数项和指数项的描述，对一元多项式的基本运算，可应用两个有序链表合并的思想进行。

2．实现要求：

. “建立多项式算法”操作结果： 输入 m 项的系数和指数，建立一元多项式 P；

. “销毁多项式算法”初始条件： 一元多项式 p 已存在；

操作结果：销毁一元多项式

. “输出多项式算法”初始条件： 一元多项式 p 已存在；

操作结果：打印一元多项式 p；

. “多项式加法算法” 初始条件：两个多项式的 Pa, Pb 已存在 ；

操作结果：pa = pa + pb，并销毁 pb；

. “多项式减法算法” 初始条件：两个多项式的 Pa, Pb 已存在 ；

操作结果：pa = pa － pb，并销毁 pb；

. “多项式乘法算法” 初始条件：两个多项式的 Pa, Pb 已存在 ；

操作结果：pa = pa * pb，并销毁 pb；

. “求多项式项数算法” 初始条件：一元多项式 p 已存在；

操作结果：返回 p 中的项数；

三、编程指导

1．由于一元多项式包含系数项和指数项，其系数为 float 型，指数项为 int 型，往往将其两项组合成为一个结构元素类型，将多项式看成是一个有序表，则多项式定义中的各个操作均可利用有序表操作来完成。定义抽象类型数据 polynomial：

```
typedef   struct{          //项的表示，多项式的项作为 LinkList 的数据元素
            float coef;       //系数
            int expn;         //指数
   }term, ElemType;    //两个类型名：term 用于本 ADT，ElemType 为 LinkList 的数据对象名
typedef LinkList polynomial;  //用带表头结点的有序链表表示多项式；
```

再根据算法的描述和实现要求，分别实现如下算法：CreatePolyn、 DestroyPolyn、PrintPolyn、 AddPolyn、SubtractPolyn、 MultiplyPolyn、 PolynLength，可以组合成在一个头文件之中，如取名 Polyn.h。

2．对单向循环链表的类型定义与单链表相同，无需重复定义，只是对单向循环链表的判断，要对最后一个结点作特殊的处理，建链表时最后一个结点的指针域不能是 p->next=NULL，而是指向第一个结点，即 p->next=head；p=head；

3．对双向循环链表的操作，要从数据结构类型定义重新开始，对其操作的算法均要重写，如插入和删除算法，比单链要灵活，可以通过修改向前、向后指针以实现。 其定义、实现及应用与单链表的过程相似，在此不复述。

四、参考程序

1. 文件 Polyn.h 为一元多项式类型定义的带头结点的线性链表类型定义和基本操作描述

```
typedef struct LNode /*  结点类型  */
{
   ElemType data;
   struct LNode *next;
}LNode,*Link,*Position;

typedef struct LinkList /*  链表类型  */
{
   Link head,tail;        /*  分别指向线性链表中的头结点和最后一个结点  */
   int len;               /*  指示线性链表中数据元素的个数  */
```

```
    }LinkList;
   typedef    struct{              //项的表示，多项式的项作为 LinkList 的数据元素
                  float coef;       //系数
                  int expn;          //指数
        }term, ElemType;     //两个类型名：term 用于本 ADT，ElemType 为 LinkList 的数据对象名
   typedef LinkList polynomial;   //用带表头结点的有序链表表示多项式;


   int cmp(term a , term   b) //比较多项式的项，a 的指数值<b 的指数值  返回-1
                              //a 的指数值 〉b 的指数值  返回 0
                              //a 的指数值 =b 的指数值  返回 1
 {   /* 算法略   */    }
void CreatePolyn(polynomial &p, int m)
{ polynomial h;
   ElemType e;
//输入 m 项的系数和指数，建立表示一元多项式的有序链表 P
InitList(p);
h=p;
e.coef = 0.0;
e.expn = -1;
//SetCurElem(P, e)                  //置头结点的数据元素
for(i=1;   i<= m  ;   i++)          //依次输入 m 个非零项
  {
    scanf(“%f%d”,&e.coef, &e.expn);
    if(! LocateElem(p, e, (*cmp)() )     //当前链表中不存在该指数项
          ListInsert_Link(p,i e);   //在第一个大于插入项指数的数据项前插入/新的数据项
    }
}
```

2．准备测试数据，对 main 函数 和其他算法改写，请同学们自我完成。

五、实验步骤

实验环境： 利用 Visual C++集成开发环境进行本实验的操作。

六、思考题：

1．设计一元稀疏多项式简单计数器

（1） 输入并建立多项式

（2） 输出多项式，输出形式为整数序列：n，c1，e1，c2，e2……cn，en，其中 n 是多项式的项数，ci，ei 分别为第 i 项的系数和指数。序列按指数降序排列。

（3） 多项式 a 和 b 相加，建立多项式 a+b，输出相加的多项式。

（4） 多项式 a 和 b 相减，建立多项式 a-b，输出相减的多项式。

用带表头结点的单链表存储多项式。

测试数据为：

（1）$(2x+5x^{8}-3.1x^{11})+(7-5x^{8}+11x^{9})$

（2） $(6x^{-3}-x+4.4x^{2}-1.2x^{9})-(-6x^{-3}+5.4x^{2}+7.8x^{15})$

（3）$(x+x^{2}+x^{3})+0$

（4）$(x+x^{3})-(-x-x^{3})$

2．实现两个链表的合并

基本功能要求：

（1）建立两个链表 A 和 B，链表元素个数分别为 m 和 n 个。

（2）假设元素分别为(x1,x2,…xm)，和(y1,y2, …yn)。把它们合并成一个线性表 C，使得：

当 m>=n 时，C=x1,y1,x2,y2,…xn,yn,…,xm

当 n>m 时，C=y1,x1,y2,x2,…ym,xm,…,yn

输出线性表 C。

3．实现约瑟夫环

问题描述：编号为 1，2… n 的 n 个人按顺时针方向围坐一圈，每人持有一个密码（正整数）。一开始任选一个正整数作为报数的上限值 m，从第一个人开始按顺时针方向自 1 开始顺序报数，报到 m 时停止报数，报 m 的人出列，将他的密码作为新的 m 值，从他的顺时针方向上的下一个开始重新从 1 报数，如此下去，直至所有人全部出列为止，设计一个程序求出出列顺序。

基本功能要求：

（1）利用单循环链表作为存储结构模拟此过程；

（2）键盘输入总人数、初始报数上限值 m 及各人密码；

（3）按照出列顺序输出各人的编号。

实验三：栈和队列的定义及基本操作（必做： 2学时）

一、实验目的：

. 熟练掌握栈和队列的特点

. 掌握栈的定义和基本操作，熟练掌握顺序栈的操作及应用

. 掌握对列的定义和基本操作，熟练掌握链式队列的操作及应用, 掌握环形队列的入队和出队等基本操作

. 加深对栈结构和队列结构的理解，逐步培养解决实际问题的编程能力

二、实验内容:

定义顺序栈，完成栈的基本操作：空栈、入栈、出栈、取栈顶元素；

实现十进制数与八进制数的转换，十进制数与十六进制数的转换和任意进制之间的转换；

定义链式队列，完成队列的基本操作：入队和出队；

1. 问题描述：

（1）利用栈的顺序存储结构,设计一组输入数据（假定为一组整数），能够对顺序栈进行如下操作：

. 初始化一个空栈，分配一段连续的存储空间，且设定好栈顶和栈底；

. 完成一个元素的入栈操作，修改栈顶指针；

. 完成一个元素的出栈操作，修改栈顶指针；

. 读取栈顶指针所指向的元素的值；

. 将十进制数 N 和其它 d 进制数的转换是计算机实现计算的基本问题，其解决方案很多，其中最简单方法基于下列原理：即除 d 取余法。例如：（1348）10=（2504）8

N	N div 8	N mod 8
1348	168	4
168	21	0
21	2	5
2	0	2

从中我们可以看出，最先产生的余数 4 是转换结果的最低位，这正好符合栈的特性即后进先出的特性。所以可以用顺序栈来模拟这个过程。以此来实现十进制数与八进制数的转换，十进制数与十六进制数的转换；

. 编写主程序，实现对各不同的算法调用。

（2）利用队列的链式存储结构,设计一组输入数据（假定为一组整数），能够对链式队列进行如下操作：

. 初始化一个空队列，形成一个带表头结点的空队；

. 完成一个元素的入队操作，修改队尾指针；

. 完成一个元素的出队操作，修改队头指针；

. 修改主程序，实现对各不同的算法调用。

其他算法的描述省略，参见实现要求说明。

2．实现要求：

对顺序栈的各项操作一定要编写成为 C（C++）语言函数，组合成模块化的形式，每个算法的实现要从时间复杂度和空间复杂度上进行评价。

.“初始化栈算法”操作结果：构造一个空栈 S；

.“销毁栈算法”操作结果：销毁栈 S，S 不再存在；

.“置空栈算法” 操作结果：把 S 置为空栈；

.“判是否空栈算法” 操作结果：若栈 S 为空栈，则返回 TRUE，否则返回 FALSE；

.“求栈的长度算法” 操作结果：返回 S 的元素个数，即栈的长度；

.“取栈顶元素算法” 操作结果：若栈不空，则用 e 返回 S 的栈顶元素，并返回 OK；否则返回 ERROR；

.“入栈算法”操作结果：插入元素 e 为新的栈顶元素

.“出栈算法”操作结果：若栈不空，则删除 S 的栈顶元素，用 e 返回其值，并返回 OK；否则返回 ERROR

.“实现十进制数与八进制数的转换算法”操作结果：输入任意一个非负的十进制数，输出对应的八进制数；

.“实现十进制数与十六进制数的转换算法”操作结果：输入任意一个非负的十进制数，输出对应的十六进制数；

对链式队列的各项操作一定要编写成为 C（C++）语言函数，组合成模块化的形式，每个算法的实现要从时间复杂度和空间复杂度上进行评价。

.“初始化空队算法”操作结果：构造一个空队列 Q；

.“销毁队列算法”操作结果：销毁队列 Q(无论空否均可)；

.“空队列算法” 操作结果：将 Q 清为空队列；

.“判队列是否为空算法” 操作结果：若 Q 为空队列,则返回 TRUE,否则返回 FALSE；

.“求队列的长度算法” 操作结果：求队列的长度，返回队列中结点的个数；

.“取队头元素算法” 操作结果：若队列不空,则用 e 返回 Q 的队头元素,并返回 OK,否则返回 ERROR；

. “入队算法”操作结果：插入元素 e 为 Q 的新的队尾元素；

.“出队算法”操作结果：若队列不空,删除 Q 的队头元素,用 e 返回其值,并返回 OK,否则返回 ERROR；

三、实验指导

（一）顺序栈的实验指导

1．首先将顺序栈存储结构定义放在一个头文件：如取名为 SqStackDef.h。

2．将顺序栈的基本操作算法也集中放在一个文件之中，如取名为 SqStackAlgo.h。如：InitStack、DestroyStack、ClearStack、StackEmpty、StackLength、GetTop、Push、Pop、conversion10_8、conversion10_16 等。

3．将函数的测试和主函数组合成一个文件，如取名为 SqStackUse.cpp。

（二）链式队列的实验指导

1．首先将链式队列的存储结构定义放在一个头文件：如取名为 LinkQueueDef.h。

2．将链式队列的基本操作算法也集中放在一个文件之中，如取名为 LinkQueueAlgo.h。如：InitQueue、DestroyQueue、ClearQueue、QueueEmpty、QueueLength、GetHead_Q、EnQueue、 DeQueue、QueueTraverse 等。

3．将函数的测试和主函数组合成一个文件，如取名为 LinkQueueUse.cpp。

四、参考程序

（一）顺序栈

1．文件 SqStackDef.h 中实现了栈的顺序存储表示

```
#define STACK_INIT_SIZE 10          /*  存储空间初始分配量  */
#define STACKINCREMENT 2          /*  存储空间分配增量  */
typedef struct SqStack
{
   SElemType *base;          /*  在栈构造之前和销毁之后，base 的值为 NULL */
   SElemType *top;           /*  栈顶指针  */
   int stacksize;              /*  当前已分配的存储空间，以元素为单位  */
}SqStack;   /*  顺序栈  */
```

2． 文件 SqStackAlgo.h 中实现顺序栈的基本操作（存储结构由 SqStackDef.h 定义）

```
Status InitStack(SqStack &S)
{ /* 构造一个空栈 S */
   S.base=(SElemType *)malloc(STACK_INIT_SIZE*sizeof(SElemType));
   if(!S.base)
      exit(OVERFLOW);  /* 存储分配失败 */
   S.top=S.base;
   S.stacksize=STACK_INIT_SIZE;
   return OK;
}

Status DestroyStack(SqStack &S)
{ /* 销毁栈 S，S 不再存在 */
   free(S.base);
   S.base=NULL;
   S.top=NULL;
   S.stacksize=0;
   return OK;
}

Status ClearStack(SqStack &S)
{ /* 把 S 置为空栈 */
   S.top=S.base;
   return OK;
}

Status StackEmpty(SqStack S)
{ /* 若栈 S 为空栈，则返回 TRUE，否则返回 FALSE */
   if(S.top==S.base)
      return TRUE;
   else
      return FALSE;
```

```
}

int StackLength(SqStack S)
{ /* 返回 S 的元素个数，即栈的长度 */
  return S.top-S.base；
}

Status GetTop(SqStack S,SElemType &e)
{ /* 若栈不空，则用 e 返回 S 的栈顶元素，并返回 OK；否则返回 ERROR */
  if(S.top>S.base)
  {
    e=*(S.top-1)；
    return OK；
  }
  else
    return ERROR；
}

Status Push(SqStack &S,SElemType e)
{ /* 插入元素 e 为新的栈顶元素 */
  if(S.top-S.base>=S.stacksize) /* 栈满，追加存储空间 */
  {
    S.base=(SElemType *)realloc(S.base,(S.stacksize+STACKINCREMENT)*sizeof(SElemType))；
    if(!S.base)
      exit(OVERFLOW)； /* 存储分配失败 */
    S.top=S.base+S.stacksize；
    S.stacksize+=STACKINCREMENT；
  }
  *(S.top)++=e；
  return OK；
}
```

```
Status Pop(SqStack &S,SElemType &e)
{ /* 若栈不空，则删除 S 的栈顶元素，用 e 返回其值，并返回 OK；否则返回 ERROR */
  if(S.top==S.base)
    return ERROR；
  e=*--S.top；
  return OK；
}

Status StackTraverse(SqStack S,Status(*visit)(SElemType))
{ /* 从栈底到栈顶依次对栈中每个元素调用函数 visit()。 */
  /* 一旦 visit()失败，则操作失败 */
  while(S.top>S.base)
    visit(*S.base++)；
  printf("\n")；
  return OK；
}
void conversion10_8()          /* 算法 3.1 */
{ /* 对于输入的任意一个非负十进制整数，打印输出与其等值的八进制数 */
  SqStack s；
  unsigned n； /* 非负整数 */
  SElemType e；
  InitStack(s)； /* 初始化栈 */
  printf("Enter an number(>=0): ")；
  scanf("%u",&n)； /* 输入非负十进制整数 n */
  while(n) /* 当 n 不等于 0 */
  {
    Push(s,n%8)； /* 入栈 n 除以 8 的余数(8 进制的低位) */
    n=n/8；
  }
  while(!StackEmpty(s)) /* 当栈不空 */
```

```
    {
      Pop(s,e);  /* 弹出栈顶元素且赋值给 e */
      printf("%d",e);  /* 输出 e */
    }
    printf("\n");
}

void conversion10_16()
{ /* 对于输入的任意一个非负 10 进制整数，打印输出与其等值的 16 进制数 */
    SqStack s;
    unsigned n;  /* 非负整数 */
    SElemType e;
    InitStack(s);  /* 初始化栈 */
    printf("Enter an number(>=0): ");
    scanf("%u",&n);  /* 输入非负十进制整数 n */
    while(n) /* 当 n 不等于 0 */
    {
      Push(s,n%16);  /* 入栈 n 除以 16 的余数(16 进制的低位) */
      n=n/16;
    }
    while(!StackEmpty(s)) /* 当栈不空 */
    {
      Pop(s,e);  /* 弹出栈顶元素且赋值给 e */
      if(e<=9)
        printf("%d",e);
      else
        printf("%c",e+55);
    }
    printf("\n");
}
```

3．在 SqStackUse.cpp 文件中，测试算法 3.1 的调用，其中间接调用了顺序栈的其他基本算法。

```
typedef int SElemType;              /* 定义栈元素类型为整型 */
#include "pubuse.h"                 /* 常量定义与系统函数原型声明，与实验一中的相同*/
#include "SqStackDef.h"             /* 采用顺序栈的类型定义 */
#include "SqStackAlgo.h"            /* 利用顺序栈的基本操作 */


void main()
{   conversion10_8();       /* 十进制数到八进制转换的验证    */
   conversion10_16();       /* 十进制数到十六进制转换的验证    */
}
```

（二）链式队列

1．文件 LinkQueue.h 中实现单链队列－－队列的链式存储结构的表示。

```
typedef struct QNode
{
   QElemType data;
   struct QNode *next;
}QNode,*QueuePtr;


typedef struct
{
   QueuePtr front,rear;   /* 队头、队尾指针 */
}LinkQueue;
```

2．文件 LinkQueueAlgo.h 中实现的链队列的基本算法，其存储结构由 LinkQueueDef.h 定义。

```
Status InitQueue(LinkQueue &Q)
{ /* 构造一个空队列 Q */
   Q.front=Q.rear=(QueuePtr)malloc(sizeof(QNode));
   if(!Q.front)
      exit(OVERFLOW);
   Q.front->next=NULL;
```

```
    return OK；
}

Status DestroyQueue(LinkQueue &Q)
{ /*  销毁队列 Q(无论空否均可) */
    while(Q.front)
    {
        Q.rear=Q.front->next；
        free(Q.front)；
        Q.front=Q.rear；
    }
    return OK；
}

Status ClearQueue(LinkQueue &Q)
{ /*  将 Q 清为空队列  */
    QueuePtr p,q；
    Q.rear=Q.front；
    p=Q.front->next；
    Q.front->next=NULL；
    while(p)
    {
        q=p；
        p=p->next；
        free(q)；
    }
    return OK；
}

Status QueueEmpty(LinkQueue Q)
{ /*  若 Q 为空队列,则返回 TRUE,否则返回 FALSE */
```

```
    if(Q.front==Q.rear)
        return TRUE；
    else
        return FALSE；
}

int QueueLength(LinkQueue Q)
{ /*  求队列的长度  */
    int i=0；
    QueuePtr p；
    p=Q.front；
    while(Q.rear!=p)
    {
        i++；
        p=p->next；
    }
    return i；
}

Status GetHead_Q(LinkQueue Q,QElemType &e)
{ /*  若队列不空,则用 e 返回 Q 的队头元素,并返回 OK,否则返回 ERROR */
    QueuePtr p；
    if(Q.front==Q.rear)
        return ERROR；
    p=Q.front->next；
    e=p->data；
    return OK；
}

Status EnQueue(LinkQueue &Q,QElemType e)
{ /*  插入元素 e 为 Q 的新的队尾元素  */
```

```
    QueuePtr p=(QueuePtr)malloc(sizeof(QNode));
    if(!p) /*  存储分配失败  */
        exit(OVERFLOW);
    p->data=e;
    p->next=NULL;
    Q.rear->next=p;
    Q.rear=p;
    return OK;
}

Status DeQueue(LinkQueue &Q,QElemType &e)
{ /*  若队列不空,删除 Q 的队头元素,用 e 返回其值,并返回 OK,否则返回 ERROR */
    QueuePtr p;
    if(Q.front==Q.rear)
        return ERROR;
    p=Q.front->next;
    e=p->data;
    Q.front->next=p->next;
    if(Q.rear==p)
        Q.rear=Q.front;
    free(p);
    return OK;
}

Status QueueTraverse(LinkQueue Q,void(*vi)(QElemType))
{ /*  从队头到队尾依次对队列 Q 中每个元素调用函数 vi()。一旦 vi 失败,则操作失败  */
    QueuePtr p;
    p=Q.front->next;
    while(p)
    {
        vi(p->data);
```

```
    p=p->next;
  }
  printf("\n");
  return OK;
}
```

3．文件 LinkQueueUse.cpp 中包含检验 LinkQueueAlgo.h 中关于链式队列基本操作的声明、测试数据和主函数。

```
#include "pubuse.h"              /* 与实验一的意义相同 */
typedef int QElemType;       /* 假设链式队列中的结点是一组整数   */
#include "linkqueuedef.h"
#include "linkqueuealgo.h"

void visit(QElemType i)
{    printf("%d ",i);   }

void main()
{
  int i;
  QElemType d;
  LinkQueue q;
  i=InitQueue(q);
  if(i)
    printf("成功地构造了一个空队列!\n");
  printf("是否空队列？%d(1:空  0:否)   ",QueueEmpty(q));
  printf("队列的长度为%d\n",QueueLength(q));
  EnQueue(q,-5);
  EnQueue(q,5);
  EnQueue(q,10);
  printf("插入 3 个元素(-5,5,10)后,队列的长度为%d\n",QueueLength(q));
  printf("是否空队列？%d(1:空  0:否)   ",QueueEmpty(q));
  printf("队列的元素依次为：");
```

```
    QueueTraverse(q,visit);
    i=GetHead_Q(q,d);
    if(i==OK)
       printf("队头元素是：%d\n",d);
    DeQueue(q,d);
    printf("删除了队头元素%d\n",d);
    i=GetHead_Q(q,d);
    if(i==OK)
       printf("新的队头元素是：%d\n",d);
    ClearQueue(q);
    printf("清空队列后,q.front=%u q.rear=%u q.front->next=%u\n",q.front,q.rear,q.front->next);
    DestroyQueue(q);
    printf("销毁队列后,q.front=%u q.rear=%u\n",q.front, q.rear);
}
```

五、实验环境和实验步骤

实验环境： 利用 Visual C++集成开发环境进行本实验的操作。

（一）基本实验的实验步骤：(顺序栈的定义以及应用)

1．启动 VC++;

2.新建工程/Win32 Console Application，选择输入位置：如 “d:\”，输入工程的名称：如 “SqStackDemo”；按“确定”按钮，选择“An Empty Project”，再按“完成”按钮，

3．加载实验一中的 pubuse.h 选中菜单的“project”→“add to project”→ “files”选择已存在文件，确定，然后一定将文件 pubuse.h 拷贝到所建的工程目录下；

4. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“SqStackDef.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

5. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“SqStackAlgo.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

6. 新建文件/C++ Source File，选中“添加到工程的复选按钮”，输入文件名“SqStackUse.cpp”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

7．构件、调试、运行与实验一相同，在此不再复述；

（二）基本实验的实验步骤：(链式队列定义以及应用)

1．启动 VC++;

2.新建工程/Win32 Console Application，选择输入位置：如 “d:\”，输入工程的名称：如 “LinkQueueDemo”; 按“确定”按钮，选择“An Empty Project”，再按“完成”按钮，

3．加载实验一中的 pubuse.h 选中菜单的”project”--“add to project”-- “files”选择已存在文件，确定，然后一定将文件 pubuse.h 拷贝到所建的工程目录下；

4. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“LinkQueueDef.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

5. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“LinkQueueAlgo.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

6. 新建文件/C++ Source File，选中“添加到工程的复选按钮”，输入文件名“LinkQueueUse.cpp”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

7．构件、调试、运行与实验一相同，在此不再复述；

六、思考题

1．利用一个堆栈，将一个线性表中的元素按逆序重新存放。例如原来的顺序为 12，8，6，4，2，要求改为 2，4，6，8，12。

[实验说明]

设原始数据已存入数组 a 中，堆栈为 stack ， 已清空，栈指针为 top ， 初始 top=0 。 首先从线性表第 1 个元素开始，依次将其元素压入栈中，然后将栈中元素依次弹出，重新放入数组 a 中。

2. 设数组 QU[0 ，mo-1] 中存放循环队列的元素。编写能向该循环队列插入一个结点数据和删除一个结点数据的程序。

[实验说明]

(1) 队列的特点是在队尾入队，在队首出队。在循环队列中，最初队列为空时队首指针 front 和队尾指针 rear 都指向同一位置，当有元素入队时，由于是循环的，所以 rear 位置前移，即：

QU. rear = (QU. rear + 1) % mo

将插入元素放到 rear 的新位置上。

(2)当有元素出队时，先将 front 前移一个位置，即：

QU. front = (QU. front + 1) % mo

将 front 新位置的元素取出即可。

实验四：栈和队列的综合应用（选做： 2 学时）

一、实验目的：

. 熟悉栈的定义和栈的基本操作

. 熟悉队列的定义和栈的基本操作

. 掌握递归和非递归算法的实现技术和实际应用

. 加深对栈结构的理解，培养解决实际问题的编程能力。

二、实验内容:

（一）基本实验内容：

实现 Hanoi 塔的问题；

完成迷宫问题或马踏棋盘问题求解。

1．问题描述：汉诺塔（ Hanoi），传说在创世纪时，在一个叫 Brahma 的寺庙里，有三个柱子，其中一柱上有 64 个盘子从小到大依次叠放，僧侣的工作是将这 64 个盘子从一根柱子移到另一个柱子上。

移动时的规则： 每次只能移动一个盘子； 只能小盘子在大盘子上面； 可以使用任一柱子。

当工作做完之后，就标志着世界末日到来。

2．实现要求：

设三根柱子分别为 x，y,z,盘子在 x 柱上，要移到 z 柱上。

1、当 n=1 时，盘子直接从 x 柱移到 z 柱上；

2、当 n>1 时，则①设法将前 n-1 个盘子借助 z，从 x 移到 y 柱上，把盘子 n 从 x 移到 z 柱上；②把 n -1 个盘子从 y 移到 z 柱上。

迷宫求解的问题描述：请参考教材和参考程序；

三、参考程序

（一）实现 Hanoi 塔问题(只需建立如下的一个文件 hanoi.cpp 即可)

```
#include<stdio.h>
int c=0; /* 全局变量，搬动次数 */
```

```
void move(char x,int n,char z)
{ /* 第 n 个圆盘从塔座 x 搬到塔座 z */
  printf("第%i 步: 将%i 号盘从%c 移到%c\n",++c,n,x,z);
}

void hanoi(int n,char x,char y,char z) /* 算法 3.5 */
{ /* 将塔座 x 上按直径由小到大且自上而下编号为 1 至 n 的 n 个圆盘 */
  /* 按规则搬到塔座 z 上。y 可用作辅助塔座 */
  if(n==1)
    move(x,1,z); /* 将编号为 1 的圆盘从 x 移到 z */
  else
  {
    hanoi(n-1,x,z,y); /* 将 x 上编号为 1 至 n-1 的圆盘移到 y，z 作辅助塔 */
    move(x,n,z); /* 将编号为 n 的圆盘从 x 移到 z */
    hanoi(n-1,y,x,z); /* 将 y 上编号为 1 至 n-1 的圆盘移到 z，x 作辅助塔 */
  }
}

void main()
{
  int n;
  printf("3 个塔座为 a、b、c，圆盘最初在 a 座，借助 b 座移到 c 座。请输入圆盘数：");
  scanf("%d",&n);
  hanoi(n,'a','b','c');
}
```

（二）迷宫问题参考程序(只需建立如下的一个文件 maze.cpp 即可)

```
/* 用递归函数求解迷宫问题(求出所有解) */
#include<stdio.h>
```

```
struct PosType /* 迷宫坐标位置类型 */
{
  int x; /* 行值 */
  int y; /* 列值 */
};

#define MAXLENGTH 25 /* 设迷宫的最大行列为 25 */
typedef int MazeType[MAXLENGTH][MAXLENGTH]; /* [行][列] */

/* 全局变量 */
struct PosType end; /* 迷宫终点位置 */
MazeType m;   /* 迷宫数组 */
int x,y;    /* 迷宫行数，列数 */

/* 定义墙元素值为 0,可通过路径为-1,通过路径为足迹 */

void Print(int x,int y)
{ /* 输出解 */
  int i,j;
  for(i=0;  i<x;  i++)
  {
    for(j=0;j<y;j++)
      printf("%3d",m[i][j]);
    printf("\n");
  }
  printf("\n");
}

void Try(struct PosType cur,int curstep)
{ /* 由当前位置 cur、当前步骤 curstep 试探下一点 */
  int i;
```

```
  struct PosType next; /* 下一个位置 */
  struct PosType direc[4]={{0,1},{1,0},{0,-1},{-1,0}}; /* {行增量,列增量} */
  /* 移动方向,依次为东南西北 */
  for(i=0;i<=3;i++) /* 依次试探东南西北四个方向 */
  {
    next.x=cur.x+direc[i].x;
    next.y=cur.y+direc[i].y;
    if(m[next.x][next.y]==-1) /* 是通路 */
    {
      m[next.x][next.y]=++curstep;
      if(next.x!=end.x||next.y!=end.y) /* 没到终点 */
        Try(next,curstep); /* 试探下一点(递归调用) */
      else
        Print(x,y); /* 输出结果 */
      m[next.x][next.y]=-1; /* 恢复为通路，试探下一条路 */
      curstep--;
    }
  }
}

void main()
{
  struct PosType begin;
  int i,j,x1,y1;
  printf("请输入迷宫的行数,列数(包括外墙)：");
  scanf("%d,%d",&x,&y);
  for(i=0;i<x;i++) /* 定义周边值为0(同墙) */
  {
    m[0][i]=0; /* 行周边 */
    m[x-1][i]=0;
  }
```

```
  for(j=1;j<y-1;j++)
  {
    m[j][0]=0;  /* 列周边 */
    m[j][y-1]=0;
  }
  for(i=1;i<x-1;i++)
    for(j=1;j<y-1;j++)
      m[i][j]=-1;   /* 定义通道初值为-1 */
  printf("请输入迷宫内墙单元数：");
  scanf("%d",&j);
  if(j)
    printf("请依次输入迷宫内墙每个单元的行数,列数：\n");
  for(i=1;i<=j;i++)
  {
    scanf("%d,%d",&x1,&y1);
    m[x1][y1]=0;
  }
  printf("迷宫结构如下:\n");
  Print(x,y);
  printf("请输入起点的行数,列数：");
  scanf("%d,%d",&begin.x,&begin.y);
  printf("请输入终点的行数,列数：");
  scanf("%d,%d",&end.x,&end.y);
  m[begin.x][begin.y]=1;
  Try(begin,1); /* 由第一步起点试探起 */
}
```

四、实验环境和实验步骤

实验环境： 利用 Visual C++集成开发环境进行本实验的操作。

hanoi 塔实验步骤：

1．启动 VC++;

2. 新建文件/C++ Source File，选中“添加到工程的复选按钮”，输入文件名“hanoi.cpp”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

3. 按照提示，创建一个缺省工程；

4. 构件、调试、运行与实验一相同，在此不再复述；

迷宫求解实验步骤：

1．启动 VC++;

2. 新建文件/C++ Source File，选中“添加到工程的复选按钮”，输入文件名“maze.cpp”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

3. 按照提示，创建一个缺省工程；

4. 构件、调试、运行与实验一相同.

实验五：二叉树的定义及基本操作 （必做：基本 2 学时，扩展 4 学时）

一、实验目的：

. 熟练掌握二叉树的二叉链表存储结构

. 掌握二叉树的非线性和递归性特点

. 熟练掌握二叉树的递归遍历操作的实现方法，掌握二叉树的非递归遍历操作的实现

. 掌握线索二叉树的定义和基本操作

. 加深对二叉树结构和性质的理解，逐步培养解决实际问题的编程能力

二、实验内容:

（一）基本实验内容：

. 定义二叉树的链式存储结构；

. 实现二叉树的基本操作：建空树、销毁二叉树、生成二叉树(先序,中序或后序)、判二叉树是否为空、求二叉树的深度、求二叉树的根等基本算法；

. 实现二叉树的递归(先序、中序或后序)遍历算法；

1．问题描述：

利用二叉树的链式存储结构，设计一组输入数据（假定为一组整数或一组字符），能够对二叉树进行如下操作：

. 创建一棵空二叉树；

. 对一棵存在的二叉树进行销毁；

. 根据输入某种遍历次序输入二叉树中结点的值，依序建立二叉树；

. 判断某棵二叉树是否为空；

. 求二叉树的深度；

. 求二叉树的根结点，若为空二叉树，则返回一特殊值；

. 二叉树的遍历，即按某种方式访问二叉树中的所有结点，并使每个结点恰好被访问一次；

. 编写主程序，实现对各不同的算法调用；

其他算法的描述省略，参见实现要求说明。

2．实现要求：

.“构造空二叉树算法”操作结果：构造一个空二叉树 T；

.“销毁二叉树算法”初始条件：二叉树 T 存在；

操作结果：销毁二叉树 T ；

.“创建二叉树算法”初始条件:可以根据先序、中序和后序输入二叉树中结点的值（可为字符型或整型）；

操作结果：以选择的某种次序建立二叉树 T ；

.“判二叉树是否为空算法”初始条件：二叉树 T 存在；

操作结果：若 T 为空二叉树,则返回 TRUE,否则 FALSE ；

.“求二叉树的深度算法” 初始条件：二叉树 T 存在；

操作结果：返回 T 的深度；

. “求二叉树的根算法”初始条件：二叉树 T 存在；

操作结果：返回 T 的根；

. “先序递归遍历算法”初始条件：二叉树 T 存在,Visit 是对结点操作的应用函数；

操作结果：先序递归遍历 T,对每个结点调用函数 Visit 一次且仅一次；

. “中序递归遍历算法”初始条件：二叉树 T 存在,Visit 是对结点操作的应用函数；

操作结果：中序递归遍历 T,对每个结点调用函数 Visit 一次且仅一次；

. “后序递归遍历算法”初始条件：二叉树 T 存在,Visit 是对结点操作的应用函数；

操作结果：后序递归遍历 T,对每个结点调用函数 Visit 一次且仅一次；

（二）扩展实验内容：

利用二叉树的链式存储结构，设计一组输入数据（假定为一组整数或一组字符），能够对二叉树进行如下操作：

. 求某一个结点的双亲结点，求某一个结点的左孩子（或右孩子）结点；求某一个结点的左兄弟（或右兄弟）算法；

. 利用栈，实现二叉树的非递归(先序、中序或后序)遍历算法；

. 利用队列，实现层序递归遍历二叉树；

. **定义线索二叉树的链式存储结构**，建立线索二叉树，实现线索二叉树的插入和删除操作；

1. 问题描述：

. 求二叉树中某个指定结点的父结点，当指定结点为根时，返回一特殊值；

. 求二叉树中某个指定结点的左孩子结点，当指定结点没有左孩子时，返回一特殊值；

. 求二叉树中某个指定结点的右孩子结点，当指定结点没有右孩子时，返回一特殊值；

. 实现中序非递归遍历二叉树算法一定采用二叉链表存储结构，并且仿照递归算法执行过程中递归工作栈的状态变化状况直接实现栈的操作，写出相应的非递归算法；中序和后序类似；

. 编写主程序，实现对各不同的算法调用。

2．实现要求：

.“求双亲算法”初始条件：二叉树 T 存在,e 是 T 中某个结点；

操作结果：若 e 是 T 的非根结点,则返回它的双亲,否则返回＂空＂；

.“求左孩子算法”初始条件：二叉树 T 存在,e 是 T 中某个结点；

操作结果：返回 e 的左孩子。若 e 无左孩子,则返回＂空＂ ；

.“求右孩子算法”初始条件：二叉树 T 存在,e 是 T 中某个结点；

操作结果：返回 e 的右孩子。若 e 无右孩子,则返回＂空＂ ；

. “求左兄弟算法”初始条件：二叉树 T 存在,e 是 T 中某个结点；

操作结果：返回 e 的左兄弟。若 e 是 T 的左孩子或无左兄弟,则返回＂空＂；

. “求右兄弟算法”初始条件：二叉树 T 存在,e 是 T 中某个结点；

操作结果：返回 e 的右兄弟。若 e 是 T 的右孩子或无右兄弟,则返回＂空＂；

. “孩子插入算法”初始条件：二叉树 T 存在,p 指向 T 中某个结点,LR 为 0 或 1,非空二叉树 c 与 T 不相交且右子树为空；

操作结果：根据 LR 为 0 或 1,插入 c 为 T 中 p 所指结点的左或右子树。p 所指结点的原有左或右子树则成为 c 的右子树；

. “删除孩子算法”初始条件：二叉树 T 存在,p 指向 T 中某个结点,LR 为 0 或 1 ；

操作结果：根据 LR 为 0 或 1,删除 T 中 p 所指结点的左或右子树；

. “中序非递归遍历二叉树算法”初始条件：二叉树 T 存在,Visit 是对结点操作的应用函数；

操作结果：中序非递归遍历 T(利用栈),对每个结点调用函数 Visit 一次且仅一次；

. “层序递归遍历二叉树算法初始条件：二叉树 T 存在,Visit 是对结点操作的应用函数；

操作结果：层序递归遍历 T(利用队列),对每个结点调用函数 Visit 一次且仅一次；

三、实验指导

（一）基本实验指导

1．首先将二叉树的链式存储结构定义放在一个头文件：如取名为 BinTreeDef.h。

2．将二叉树的基本操作算法也集中放在一个文件之中，如取名为 BinTreeAlgo.h。包含关于二叉树的链式结构操作的一些基本算法，如：InitBiTree、DestroyBiTree、CreateBiTree、BiTreeEmpty、BiTreeDepth、Root、

PreOrderTraverse、InOrderTraverse 等。

3．将函数的测试和主函数组合成一个文件，如取名为 BinTreeUse.cpp。

（二）扩展实验指导

1．用栈实现二叉树先序遍历的非递归程序。

【思想】非递归遍历二叉树的程序中，要用栈来保存遍历经过的路径，才能访问到二叉树的每一个结点。先序遍历二叉树的顺序是“根、左、右”，所以，在对二叉树进行先序遍历时，从二叉树的根结点开始，在沿左子树向前走的过程中，将所遇结点进栈，并退栈访问之，并将其左、右子树进栈，当前进到最左端无法再走下去时，则退回，按退回的顺序进入其右子树进行遍历，如此重复直到树中的所有结点都访问完毕为止。

【算法优化】对于先序遍历加进适当判断可以减少进出栈的次数：访问一个结点之后，仅当该结点左、右子树都不空时才把结点的右孩子推进栈。这样可以节省算法的时间与空间开销。

后序非递归算法比较复杂，每个结点要到它们左、右子树都遍历完时才得以访问，所以在去遍历它的左、右子树之前都需要进栈。当它出栈时，需要判断是从左子树回来（即刚遍历完左子树，此时需要去遍历右子树），还是从右子树回来（即刚遍历完右子树，此时需要去访问这个结点）。因此，进栈的结点需要伴随着一个标记 tag 。

tag = 1 表明遍历它的左子树
tag = 2 表明遍历它的右子树

(2) 由于用栈直接进行二叉树非递归遍历算法的描述，凡涉及栈的定义和基本操作算法，可以调用实验 3_1 中栈的算法，无需重新编写；

(3) 由于用队列直接进行二叉树层序递归遍历算法的描述，凡涉及队列的定义和基本操作算法，可以调用实验 3_2 中队列的算法，无需重新编写；

2. 二叉树的扩展操作实验

(1) 首先将二叉树的链式存储结构定义放在一个头文件：如取名为 BinTreeDef.h。

(2) 将二叉树的扩展操作算法也集中放在一个文件之中，也可以与基本操作算法放在同一个文件 BinTreeAlgo.h 之中。包含关于二叉树的链式结构的一些扩展操作算法，如：InOrderTraverse1、InOrderTraverse2、LevelOrderTraverse 等。

(3) 将函数的测试和主函数组合成一个文件，如取名为 BinTreeUse.cpp。

(4）其中二叉树非递归遍历算法直接调用了顺序栈的基本算法，可以直接使用实验 3_1 的栈的操作算法，要在程序中务必包含#include “SqStackDef.h” 和 #include “SqStackAlgo.h”，还要在工程中将这 2 个文件添

加进去；

(5) 二叉树层序递归遍历算法直接调用了队列的基本算法, 可以直接使用实验 3_2 的链式队列的操作算法, 要在程序中务必包含#include “LinkQueueDef.h” 和 #include “LinkQueueAlgo.h”， 还要在工程中将这 2 个文件添加进去；

3．线索二叉树的实验

(1) 首先将线索二叉树的链式存储结构定义放在一个头文件：如取名为 ThreadBinTreeDef.h。

(2) 将线索二叉树的扩展操作算法也集中放在一个文件之中，如取名为 ThreadBinTreeAlgo.h。包含关于线索二叉树的链式结构操作的一些扩展操作，如：InOrderTraverse1、InOrderTraverse、LevelOrderTraverse 等。

(3) 将函数的测试和主函数组合成一个文件，如取名为 ThreadBinTreeUse.cpp。

四、参考程序

（一）基本实验的参考程序

1、文件 pubuse.h，与实验一相同；

2．文件 BinTreeDef.h 中实现了二叉树的链式存储结构定义

```
typedef struct BiTNode
 {
    TElemType data;
    struct BiTNode *lchild,*rchild;   /*  左右孩子指针  */
 }BiTNode,*BiTree;
```

3． 文件 BinTreeAlgo.h 中实现二叉树的基本操作（存储结构由 BinTreeDef.h 定义）

```
/* BinTreeAlgo.h  二叉树的二叉链表存储(存储结构由 BinTreeDef.h 定义)的基本操作  */
 Status InitBiTree(BiTree &T)
 { /*  操作结果: 构造空二叉树 T */
    T=NULL;
    return OK;
 }

 void DestroyBiTree(BiTree &T)
 { /*  初始条件: 二叉树 T 存在。操作结果: 销毁二叉树 T */
    if(T) /*  非空树  */
```

```
  {
    if(T->lchild) /* 有左孩子 */
      DestroyBiTree(T->lchild);  /* 销毁左孩子子树 */
    if(T->rchild) /* 有右孩子 */
      DestroyBiTree(T->rchild);  /* 销毁右孩子子树 */
    free(T);  /* 释放根结点 */
    T=NULL;  /* 空指针赋 0 */
  }
}
#define ClearBiTree DestroyBiTree
void CreateBiTree(BiTree &T)
{ /* 算法 6.4:按先序次序输入二叉树中结点的值（可为字符型或整型，在主程中 */
  /* 定义），构造二叉链表表示的二叉树 T。变量 Nil 表示空（子）树。有改动 */
  TElemType ch;
#ifdef CHAR
  scanf("%c",&ch);
#endif
#ifdef INT
  scanf("%d",&ch);
#endif
  if(ch==Nil) /* 空 */
    T=NULL;
  else
  {
    T=(BiTree)malloc(sizeof(BiTNode));
    if(!T)
      exit(OVERFLOW);
    T->data=ch;  /* 生成根结点 */
    CreateBiTree(T->lchild);  /* 构造左子树 */
    CreateBiTree(T->rchild);  /* 构造右子树 */
  }
```

```
}
Status BiTreeEmpty(BiTree T)
{ /* 初始条件: 二叉树 T 存在 */
  /* 操作结果: 若 T 为空二叉树,则返回 TRUE,否则 FALSE */
  if(T)
    return FALSE;
  else
    return TRUE;
}
int BiTreeDepth(BiTree T)
{ /* 初始条件: 二叉树 T 存在。操作结果: 返回 T 的深度 */
  int i,j;
  if(!T)
    return 0;
  if(T->lchild)
    i=BiTreeDepth(T->lchild);
  else
    i=0;
  if(T->rchild)
    j=BiTreeDepth(T->rchild);
  else
    j=0;
  return i>j?i+1:j+1;
}
TElemType Root(BiTree T)
{ /* 初始条件: 二叉树 T 存在。操作结果: 返回 T 的根 */
  if(BiTreeEmpty(T))
    return Nil;
  else
    return T->data;
}
```

```
TElemType Value(BiTree p)
{ /* 初始条件: 二叉树 T 存在，p 指向 T 中某个结点 */
  /* 操作结果: 返回 p 所指结点的值 */
  return p->data；
}
void Assign(BiTree p,TElemType value)
{ /* 给 p 所指结点赋值为 value */
  p->data=value；
}
void PreOrderTraverse(BiTree T,Status(*Visit)(TElemType))
{ /* 初始条件: 二叉树 T 存在,Visit 是对结点操作的应用函数。算法 6.1，有改动 */
  /* 操作结果: 先序递归遍历 T,对每个结点调用函数 Visit 一次且仅一次 */
  if(T) /* T 不空 */
  {
    Visit(T->data)； /* 先访问根结点 */
    PreOrderTraverse(T->lchild,Visit)； /* 再先序遍历左子树 */
    PreOrderTraverse(T->rchild,Visit)； /* 最后先序遍历右子树 */
  }
}
void InOrderTraverse(BiTree T,Status(*Visit)(TElemType))
{ /* 初始条件: 二叉树 T 存在,Visit 是对结点操作的应用函数 */
  /* 操作结果: 中序递归遍历 T,对每个结点调用函数 Visit 一次且仅一次 */
  if(T)
  {
    InOrderTraverse(T->lchild,Visit)； /* 先中序遍历左子树 */
    Visit(T->data)； /* 再访问根结点 */
    InOrderTraverse(T->rchild,Visit)； /* 最后中序遍历右子树 */
  }
}
void PostOrderTraverse(BiTree T,Status(*Visit)(TElemType))
{ /* 初始条件: 二叉树 T 存在,Visit 是对结点操作的应用函数 */
```

```
    /* 操作结果: 后序递归遍历 T,对每个结点调用函数 Visit 一次且仅一次 */
    if(T) /* T 不空 */
    {
      PostOrderTraverse(T->lchild,Visit);  /* 先后序遍历左子树 */
      PostOrderTraverse(T->rchild,Visit);  /* 再后序遍历右子树 */
      Visit(T->data);  /* 最后访问根结点 */
    }
}
```

4．在文件 BinTreeUse.cpp 中，测试二叉树基本算法的调用

```
/* BinTreeUse.cpp    检验 BinTreeAlgo.h 的主程序,利用条件编译选择数据类型(另一种方法) */
/* 二叉树的数据类型可以以是字符型，也可以是整型，可在程序中使用条件编译作出判断和控制 */
#define CHAR          /* 字符型， 本例是采用字符型作为数据类型    */
/* #define INT        /* 整型（二者选一） */
#include"pubuse.h"    /* 与实验一的意义相同 */

#ifdef CHAR
  typedef char TElemType;
  TElemType Nil=' ';          /* 字符型以空格符为空 */
#endif
#ifdef INT
  typedef int TElemType;
  TElemType Nil=0;            /* 整型以 0 为空 */
#endif

#include"BinTreeDef.h"    /*  二叉树链式存储结构定义    */
#include"BinTreeAlgo.h"   /*  二叉树基本算法和扩展算法定义      */

Status visitT(TElemType e)
{
#ifdef CHAR
```

```
    printf("%c ",e);
#endif
#ifdef INT
    printf("%d ",e);
#endif
    return OK;
}

void main()
{ int i;
    BiTree T,p,c;
    TElemType e1,e2;
    /*      1---基本实验算法的验证      */
    InitBiTree(T);
    printf("构造空二叉树后,树空否？%d(1:是  0:否)  树的深度=%d\n",BiTreeEmpty(T),BiTreeDepth(T));
    e1=Root(T);
    if(e1!=Nil)
#ifdef CHAR
        printf("二叉树的根为: %c\n",e1);
#endif
#ifdef INT
        printf("二叉树的根为: %d\n",e1);
#endif
    else
        printf("树空，无根\n");
#ifdef CHAR
    printf("请先序输入二叉树(如:ab 三个空格表示 a 为根结点,b 为左子树的二叉树)\n");
#endif
#ifdef INT
    printf("请先序输入二叉树(如:1 2 0 0 0 表示 1 为根结点,2 为左子树的二叉树)\n");
#endif
```

```
    CreateBiTree(T);
    printf("建立二叉树后,树空否？%d(1:是 0:否) 树的深度=%d\n",BiTreeEmpty(T),BiTreeDepth(T));
    e1=Root(T);
    if(e1!=Nil)
#ifdef CHAR
      printf("二叉树的根为: %c\n",e1);
#endif
#ifdef INT
      printf("二叉树的根为: %d\n",e1);
#endif
    else
      printf("树空，无根\n");

    printf("中序递归遍历二叉树:\n");
    InOrderTraverse(T,visitT);

    printf("后序递归遍历二叉树:\n");
    PostOrderTraverse(T,visitT);
}
```

（二）扩展实验的参考程序

1．保持二叉树的链式存储类型定义文件不变，存储结构由 BinTreeDef.h 定义。

2．将扩展操作的实验算法可以添加到 BinTreeAlgo.h 文件（也可单独新建一个文件，将扩展实验的算法另作一个文件存在），包含非递归遍历算法，层序递归算法等。

```
Status InOrderTraverse1(BiTree T,Status(*Visit)(TElemType))
{ /* 采用二叉链表存储结构，Visit 是对数据元素操作的应用函数。算法 6.3 */
  /* 中序遍历二叉树 T 的非递归算法(利用栈)，对每个数据元素调用函数 Visit */
  SqStack S;
  InitStack(S);
  while(T||!StackEmpty(S))
```

```
  {
    if(T)
    { /* 根指针进栈,遍历左子树 */
      Push(S,T);
      T=T->lchild;
    }
    else
    { /* 根指针退栈,访问根结点,遍历右子树 */
      Pop(S,T);
      if(!Visit(T->data))
        return ERROR;
      T=T->rchild;
    }
  }
  printf("\n");
  return OK;
}

Status InOrderTraverse2(BiTree T,Status(*Visit)(TElemType))
{ /* 采用二叉链表存储结构，Visit 是对数据元素操作的应用函数。算法 6.2 */
  /* 中序遍历二叉树 T 的非递归算法(利用栈)，对每个数据元素调用函数 Visit */
  SqStack S;
  BiTree p;
  InitStack(S);
  Push(S,T);  /* 根指针进栈 */
  while(!StackEmpty(S))
  {
    while(GetTop(S,p)&&p)
      Push(S,p->lchild);  /* 向左走到尽头 */
    Pop(S,p);  /* 空指针退栈 */
    if(!StackEmpty(S))
```

```
    { /* 访问结点,向右一步 */
      Pop(S,p);
      if(!Visit(p->data))
        return ERROR;
      Push(S,p->rchild);
    }
  }
  printf("\n");
  return OK;
}
/*  注意：以下算法使用队列    */
void LevelOrderTraverse(BiTree T,Status(*Visit)(TElemType))
{ /* 初始条件：二叉树 T 存在,Visit 是对结点操作的应用函数 */
  /* 操作结果：层序递归遍历 T(利用队列),对每个结点调用函数 Visit 一次且仅一次 */
  LinkQueue q;
  QElemType a;
  if(T)
  {
    InitQueue(q);
    EnQueue(q,T);
    while(!QueueEmpty(q))
    {
      DeQueue(q,a);
      Visit(a->data);
      if(a->lchild!=NULL)
        EnQueue(q,a->lchild);
      if(a->rchild!=NULL)
        EnQueue(q,a->rchild);
    }
    printf("\n");
  }
```

}

还有一些算法，同学们自己可以根据实际需要，写出独立的算法。

3．扩展实验算法的验证，可以修改 BinTreeUse.cpp 文件内容，但由于要使用栈和队列技术以及基本操作，要达到数据类型名一致，添加如下定义：

```
typedef BiTree SElemType;    /*  设栈元素为二叉树的指针类型  */
typedef BiTree QElemType;    /*设队列元素为二叉树的指针类型*/
```

为了正确引用栈的函数，添加实验 2-1 中的基本算法包含说明：

```
#include"SqStackDef.h"
#include"SqStackAlgo.h"
```

为了正确引用队列的函数，添加实验 2-2 中的基本算法包含说明：

```
#include"LinkQueueDef.h"
#include"LinkQueueAlgo.h"
```

在主函数中，添加一些关于扩展实验的测试数据和算法调用，以验证二叉树的扩展实验内容。例如如下代码：

```
printf("\n 中序非递归遍历二叉树:\n");
InOrderTraverse1(T,visitT);
printf("\n 中序非递归遍历二叉树(另一种方法):\n");
InOrderTraverse2(T,visitT);
printf("\n 层序遍历二叉树:\n");
LevelOrderTraverse(T,visitT);
```

（三）线索二叉树的参考程序

1．文件 ThreadBinTreeDef.h 中，实现了线索二叉树的链式存储结构定义

```
typedef enum{Link,Thread}PointerTag;  /* Link(0):指针,Thread(1):线索  */
 typedef struct BiThrNode
 {
    TElemType data;
    struct BiThrNode *lchild,*rchild;  /*  左右孩子指针  */
    PointerTag LTag,RTag;  /*  左右标志  */
 }BiThrNode,*BiThrTree;
```

2．文件 ThreadBinTreeAlgo.h 中，实现线索二叉树的基本操作

```
Status CreateBiThrTree(BiThrTree &T)
{ /* 按先序输入二叉线索树中结点的值,构造二叉线索树 T */
  /* 0(整型)/空格(字符型)表示空结点 */
  TElemType h;
#if CHAR
  scanf("%c",&h);
#else
  scanf("%d",&h);
#endif
  if(h==Nil)
    T=NULL;
  else
  {
    T=(BiThrTree)malloc(sizeof(BiThrNode));
    if(!T)
      exit(OVERFLOW);
    T->data=h;  /* 生成根结点(先序) */
    CreateBiThrTree(T->lchild);  /* 递归构造左子树 */
    if(T->lchild) /* 有左孩子 */
      T->LTag=Link;
    CreateBiThrTree(T->rchild);  /* 递归构造右子树 */
    if(T->rchild) /* 有右孩子 */
      T->RTag=Link;
  }
  return OK;
}

BiThrTree pre;  /* 全局变量,始终指向刚刚访问过的结点 */
void InThreading(BiThrTree p)
{ /* 中序遍历进行中序线索化。算法 6.7 */
```

```
  if(p)
  {
    InThreading(p->lchild);  /* 递归左子树线索化 */
    if(!p->lchild) /* 没有左孩子 */
    {
      p->LTag=Thread;  /* 前驱线索 */
      p->lchild=pre;  /* 左孩子指针指向前驱 */
    }
    if(!pre->rchild) /* 前驱没有右孩子 */
    {
      pre->RTag=Thread;  /* 后继线索 */
      pre->rchild=p;  /* 前驱右孩子指针指向后继(当前结点 p) */
    }
    pre=p;  /* 保持 pre 指向 p 的前驱 */
    InThreading(p->rchild);  /* 递归右子树线索化 */
  }
}

Status InOrderThreading(BiThrTree &Thrt,BiThrTree T)
{ /* 中序遍历二叉树 T,并将其中序线索化,Thrt 指向头结点。算法 6.6 */
  Thrt=(BiThrTree)malloc(sizeof(BiThrNode));
  if(!Thrt)
    exit(OVERFLOW);
  Thrt->LTag=Link;  /* 建头结点 */
  Thrt->RTag=Thread;
  Thrt->rchild=Thrt;  /* 右指针回指 */
  if(!T) /* 若二叉树空，则左指针回指 */
    Thrt->lchild=Thrt;
  else
  {
    Thrt->lchild=T;
```

```
    pre=Thrt;
    InThreading(T);  /* 中序遍历进行中序线索化 */
    pre->rchild=Thrt;
    pre->RTag=Thread;  /* 最后一个结点线索化 */
    Thrt->rchild=pre;
  }
  return OK;
}

Status InOrderTraverse_Thr(BiThrTree T,Status(*Visit)(TElemType))
{ /* 中序遍历二叉线索树 T(头结点)的非递归算法。算法 6.5 */
  BiThrTree p;
  p=T->lchild;  /* p 指向根结点 */
  while(p!=T)
  { /* 空树或遍历结束时,p==T */
    while(p->LTag==Link)
      p=p->lchild;
    if(!Visit(p->data)) /* 访问其左子树为空的结点 */
      return ERROR;
    while(p->RTag==Thread&&p->rchild!=T)
    {
      p=p->rchild;
      Visit(p->data);  /* 访问后继结点 */
    }
    p=p->rchild;
  }
  return OK;
}
```

3．文件 ThreadBinTreeUse.cpp 是检验 ThreadBinTreeAlgo.h 的主程序

```
#define CHAR 1              /* 字符型 */
/*#define CHAR 0          /* 整型(二者选一) */
```

```
#if CHAR
   typedef char TElemType；
   TElemType Nil=' '；        /* 字符型以空格符为空 */
#else
   typedef int TElemType；
   TElemType Nil=0；          /* 整型以 0 为空 */
#endif
#include"pubuse.h"                    /* 与实验一的意义相同 */
#include"ThreadBinTreeDef.h"     /*   线索二叉树的存储结构定义   */
#include"ThreadBinTreeAlgo.h"   /*线索二叉树的基本操作   */

Status vi(TElemType c)
{   #if CHAR
   printf("%c ",c)；
#else
   printf("%d ",c)；
#endif
   return OK；
}

void main()
{
   BiThrTree H,T；
#if CHAR
   printf("请按先序输入二叉树(如:ab 三个空格表示 a 为根结点,b 为左子树的二叉树)\n")；
#else
   printf("请按先序输入二叉树(如:1 2 0 0 0 表示 1 为根结点,2 为左子树的二叉树)\n")；
#endif
   CreateBiThrTree(T)；                /* 按先序产生二叉树 */
   InOrderThreading(H,T)；             /* 中序遍历，并中序线索化二叉树 */
   printf("中序遍历(输出)二叉线索树:\n")；
```

```
    InOrderTraverse_Thr(H,vi);          /* 中序遍历(输出)二叉线索树 */
    printf("\n");
}
```

五、实验环境和实验步骤

实验环境： 利用 Visual C++集成开发环境进行本实验的操作。

（一）基本实验的实验步骤：

1．启动 VC++;

2. 新建工程/Win32 Console Application，选择输入位置：如 “d:\”，输入工程的名称：如 “BinTreeDemo”；按“确定”按钮，选择“An Empty Project”，再按“完成”按钮，

3．加载实验一中的 pubuse.h 选中菜单的”project”—>“add to project”—> “files”选择已存在文件，确定，然后一定将文件 pubuse.h 拷贝到所建的工程目录下；

4. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“BinTreeDef.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

5. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“BinTreeAlgo.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

6. 新建文件/C++ Source File，选中“添加到工程的复选按钮”，输入文件名“BinTreeUse.cpp”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

7．构件、调试、运行与实验一相同，在此不再复述；

（二）二叉链表扩展实验的实验步骤：

实验步骤与实验一相似，将所涉及的所有文件组合成一个工程，如没有的文件采用新建的方式，原来已经生成过的文件可以采用添加到工程的方式，二叉树基本算法所建的的文件之外，还要添加 4 个头文件：SqQueueDef.h, SqQueueAlgo.h,LinkStackDef.h 和 LinkStackAlgo.h，另外再修改 BinTreeUse.cpp 中的主函数，实现扩展算法的调用和验证。

（三）线索二叉树的实验步骤：

1．启动 VC++;

2. 新建工程/Win32 Console Application，选择输入位置：如 “d:\”，输入工程的名称：如

“ThreadBinTreeDemo”；按“确定”按钮，选择“An Empty Project”，再按“完成”按钮，

3．加载实验一中的 pubuse.h 选中菜单的”project”→“add to project”→ “files”选择已存在文件，确定，然后一定将文件 pubuse.h 拷贝到所建的工程目录下；

4. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“ThreadBinTreeDef.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

5. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“ThreadBinTreeAlgo.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

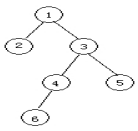
6. 新建文件/C++ Source File，选中“添加到工程的复选按钮”，输入文件名“ThreadBinTreeUse.cpp”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

7．构件、调试、运行与实验一相同，在此不再复述；

六、思考题

1．主函数中设计一个选项菜单，可选择执行建立二叉树，先序、中序、后序遍历二叉树； 先序、中序、后序（非递归）遍历二叉树；层次遍历二叉树。

2．采用链式存储结构建立二叉树，并按先序输入二叉树的结点序列。例如建立如图所示的二叉树， 建立时准备好按先序输入的结点序列。



实验六：赫夫曼编码及其应用（选做：基本 2 学时，扩展 2 学时）

一、实验目的：

. 掌握赫夫曼树的概念、存储结构

. 掌握建立赫夫曼树和赫夫曼编码的方法及带权路径长度的计算

. 熟练掌握二叉树的应用

二、实验内容:

（一）基本实验内容

实现赫夫曼树的生成，完成赫夫曼编码的输出；

1．问题描述：

利用动态分配数组存储赫夫曼树,设计一组输入数据（假定为一组整数），能够对其进行如下操作：

. 创建一个新的顺序表，实现动态空间分配的初始化;

. 对输入的数据构造成一棵 Huffman 树；

. 根据生成的 Huffman 树进行 Huffman 编码;

. 实现对输入数据的 Huffman 编码输出;

. 编写主程序，实现对各不同的算法调用。

2．实现要求：

对赫夫曼编码的各项操作一定要编写成为 C（C++）语言函数，组合成模块化的形式，每个算法的实现要从时间复杂度和空间复杂度上进行评价；

.“初始化算法”的操作结果：构造一个空的 Huffman 树顺序线性表。对顺序表的空间进行动态管理，实现动态分配、回收和增加存储空间；

.“生成 Huffman 树算法”初始条件：顺序线性表 L 已存在，且线性表里已经存在需要进行 Huffman 算法的数据；

操作结果：对原始数据按照 Huffman 树的要求输出；

.“Huffman 编码算法”初始条件：存在一棵 Huffman 树；

操作结果：对 Huffman 树的各个结点进行 Huffman 编码并输出；

分析: 修改输入数据，预期输出并验证输出的结果，加深对有关算法的理解。

（二）扩展实验内容：

完成一组码字的赫夫曼编码编码及解码(以一个文本文件的内容为例).

1. 问题描述：

试设计一种编码方案，要求对一个文本文件进行压缩编码，然后对该文件再进行译制码操作，对比原文件和编码文件进行压缩率的分析。

2. 实现要求：

首先对文本信息进行分析统计，计算出各字符出现的概率，然后以概率作为权值，应用赫夫曼编码的思想进行编解码实现。

三、实验指导

（一）基本实验指导

1．首先将赫夫曼树和赫夫曼编码的存储定义放在一个头文件：如取名为 HuffermanDef.h。

2．将建立赫夫曼树和求赫夫曼编码的算法以及测试数据和主函数也集中放在一个文件之中，如取名为 HuffermanUse.cpp。包含求赫夫曼编码的算法以及相关的算法，如：HuffmanCoding 等。

四、参考程序

（一）基本实验的参考程序

1．文件 HuffermanDef.h 是赫夫曼树和赫夫曼编码的存储表示

```
typedef struct
{
   unsigned int weight;
   unsigned int parent,lchild,rchild;
}HTNode,*HuffmanTree;            /* 动态分配数组存储赫夫曼树 */
typedef char **HuffmanCode;     /* 动态分配数组存储赫夫曼编码表 */
```

2． 文件 HuffermanUse.cpp 是求赫夫曼编码，实现书中算法 6.12 的程序

```
/* HuffermanUse.cpp 求赫夫曼编码。实现算法 6.12 的程序 */
#include"pubuse.h"
#include"HuffermanDef.h"
```

```
int min1(HuffmanTree t,int i)
{ /*  函数 void select()调用  */
   int j,flag;
   unsigned int k=UINT_MAX; /*  取 k 为不小于可能的值  */
   for(j=1;j<=i;j++)
      if(t[j].weight<k&&t[j].parent==0)
         k=t[j].weight,flag=j;
   t[flag].parent=1;
   return flag;
}

void select(HuffmanTree t,int i,int *s1,int *s2)
{ /* s1 为最小的两个值中序号小的那个  */
   int j;
   *s1=min1(t,i);
   *s2=min1(t,i);
   if(*s1>*s2)
   {
      j=*s1;
      *s1=*s2;
      *s2=j;
   }
}

void HuffmanCoding(HuffmanTree &HT,HuffmanCode &HC,int *w,int n) /*  算法 6.12 */
{ /* w 存放 n 个字符的权值(均>0),构造赫夫曼树 HT,并求出 n 个字符的赫夫曼编码 HC */
   int m,i,s1,s2,start;
   unsigned c,f;
   HuffmanTree p;
   char *cd;
```

```
if(n<=1)
   return;
m=2*n-1;
HT=(HuffmanTree)malloc((m+1)*sizeof(HTNode)); /* 0 号单元未用 */
for(p=HT+1,i=1;i<=n;++i,++p,++w)
{
   (*p).weight=*w;
   (*p).parent=0;
   (*p).lchild=0;
   (*p).rchild=0;
}
for(;i<=m;++i,++p)
   (*p).parent=0;
for(i=n+1;i<=m;++i) /* 建赫夫曼树 */
{ /* 在 HT[1~i-1]中选择 parent 为 0 且 weight 最小的两个结点,其序号分别为 s1 和 s2 */
   select(HT,i-1,&s1,&s2);
   HT[s1].parent=HT[s2].parent=i;
   HT[i].lchild=s1;
   HT[i].rchild=s2;
   HT[i].weight=HT[s1].weight+HT[s2].weight;
}
/* 从叶子到根逆向求每个字符的赫夫曼编码 */
HC=(HuffmanCode)malloc((n+1)*sizeof(char*));
/* 分配 n 个字符编码的头指针向量([0]不用) */
cd=(char*)malloc(n*sizeof(char)); /* 分配求编码的工作空间 */
cd[n-1]='\0'; /* 编码结束符 */
for(i=1;i<=n;i++)
{ /* 逐个字符求赫夫曼编码 */
   start=n-1; /* 编码结束符位置 */
   for(c=i,f=HT[i].parent;f!=0;c=f,f=HT[f].parent)
   /* 从叶子到根逆向求编码 */
```

```
            if(HT[f].lchild==c)
                cd[--start]='0';
            else
                cd[--start]='1';
        HC[i]=(char*)malloc((n-start)*sizeof(char));
        /*  为第 i 个字符编码分配空间  */
        strcpy(HC[i],&cd[start]); /*  从 cd 复制编码(串)到 HC */
    }
    free(cd); /*  释放工作空间  */
}

void main()
{
    HuffmanTree HT;
    HuffmanCode HC;
    int *w,n,i;
    printf("请输入权值的个数(>1)：");
    scanf("%d",&n);
    w=(int*)malloc(n*sizeof(int));
    printf("请依次输入%d 个权值(整型)：\n",n);
    for(i=0;i<=n-1;i++)
        scanf("%d",w+i);
    HuffmanCoding(HT,HC,w,n);
    printf("赫夫曼编码为：\n");
    for(i=1;i<=n;i++)
        puts(HC[i]);
}
```

五、实验环境和实验步骤

实验环境： 利用 Visual C++集成开发环境进行本实验的操作。

实验步骤：

1．启动 VC++;

2. 新建工程/Win32 Console Application，选择输入位置：如 “d:\”，输入工程的名称：如 “HuffmanDemo”; 按“确定”按钮，选择“An Empty Project”，再按“完成”按钮，

3．加载实验一中的 pubuse.h 选中菜单的”project”→“add to project”→ “files”选择已存在文件，确定，然后一定将文件 pubuse.h 拷贝到所建的工程目录下；

4. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“HuffmanDef.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

5. 新建文件/C++ Source File，选中“添加到工程的复选按钮”，输入文件名“HuffmanUse.cpp”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

6．构件、调试、运行与实验一相同，在此不再复述；

六、思考题

1．如何改进以上 Huffman 算法，提高编码效率？

2．用 Huffman 编码的思想对不同格式的文件进行编码，比较压缩率，分析一下其应用的意义。

实验七：图及其应用（选做： 2 学时）

一、实验目的：

. 熟练掌握图的两种存储结构(邻接矩阵和邻接表)的表示方法

. 掌握图的基本运算及应用

. 加深对图的理解，逐步培养解决实际问题的编程能力

二、实验内容:

. 采用邻接表或邻接矩阵方式存储图，实现图的深度遍历和广度遍历；

. 用广度优先搜索方法找出从一顶点到另一顶点边数最少的路径。

1．问题描述：利用邻接表存储结构，设计一种图（有向或无向），并能够对其进行如下操作：

. 创建一个可以随机确定结点数和弧（有向或无向）数的图;

. 根据图结点的序号，得到该结点的值；

. 根据图结点的位置的第一个邻接顶点的序号，以及下一个邻接顶点的序号;

. 实现从第 v 个顶点出发对图进行深度优先递归遍历;

. 实现对图作深度优先遍历;

. 实现对图进行广度优先非递归遍历;

. 编写主程序，实现对各不同的算法调用。

2．实现要求：（以邻接表存储形式为例）编写图的基本操作函数:：

对图的各项操作一定要编写成为 C（C++）语言函数，组合成模块化的形式，每个算法的实现要从时间复杂度和空间复杂度上进行评价。

.“建立图的邻接表算法”：CreateGraph(ALGraph *G)

操作结果：采用邻接表存储结构,构造没有相关信息的图 G

.“邻接表表示的图的递归深度优先遍历算法”：DFSTraverse(ALGraph G,void(*Visit)(char*))

初始条件：图 G 已经存在；

操作结果：返回图的按深度遍历的结果。

.“邻接表表示的图的广度优先遍历算法”： BFSTraverse(ALGraph G,void(*Visit)(char*))

初始条件：图 G 已经存在；

操作结果：返回图的按广度遍历的结果。

.“邻接表从某个结点开始的广度优先遍历算法”：BFS(ALGraph G, int v)

初始条件：图 G 已经存在；

操作结果：返回图从某个结点开始的按广度遍历的结果。

分析： 修改输入数据，预期输出并验证输出的结果，加深对有关算法的理解。

三、实验指导:

本实验以图的邻接表存储结构为例，要求完成基本要求，同时对无向图，有向网，无向网也一并实现其相关的操作，课后同学们可以用邻接矩阵式存储结构完成以上操作。

1． 首先将图的链接存储结构定义放在一个头文件：如取名为 ALGraphDef.h。

2． 链接表式存储图的基本操作也放在一个文件中 ALGraphAlgo.h.

3．将函数的测试和主函数组合成一个文件，如取名为文件 ALGraphUse.cpp.

由于要用到队列技术，对于队列的数据结构定义和基本操作函数的描述就可以借助实验二的结果，不必重写。

四、参考程序:

1．文件 pubuse.h 头文件；

2．文件 LinkQueueDef.h 是实验 3 中的链式队列的存储结构；

3．文件 LinkQueueAlgo.cpp 是实验 3 中的链式队列的基本操作；

4．文件 ALGraphDef.h 定义了图的链接存储结构（以邻接表存储表示）

```
#define MAX_VERTEX_NUM 20
 typedef enum{DG,DN,AG,AN}GraphKind; /* {有向图,有向网,无向图,无向网} */

 typedef struct ArcNode
 {
    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; /*  网的权值指针）  */
 }ArcNode; /*  表结点  */

 typedef struct
```

```
{
   VertexType data; /* 顶点信息 */
   ArcNode *firstarc; /* 第一个表结点的地址,指向第一条依附该顶点的弧的指针 */
}VNode,AdjList[MAX_VERTEX_NUM]; /* 头结点 */

typedef struct
{
   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．文件 ALGraphAlgo.h 定义了链接表式存储图的基本操作

```
/* ALGraphAlgo.cpp 图的邻接表存储(存储结构由 ALGraphDef.h 定义)的基本操作 */
int LocateVex(ALGraph G,VertexType u)
{ /* 初始条件: 图 G 存在,u 和 G 中顶点有相同特征 */
   /* 操作结果: 若 G 中存在顶点 u,则返回该顶点在图中位置;否则返回-1 */
   int i;
   for(i=0;i<G.vexnum;++i)
      if(strcmp(u,G.vertices[i].data)==0)
         return i;
   return -1;
}

Status CreateGraph(ALGraph &G)
{ /* 采用邻接表存储结构,构造没有相关信息的图 G(用一个函数构造 4 种图) */
   int i,j,k;
   int w; /* 权值 */
   VertexType va,vb;
   ArcNode *p;
   printf("请输入图的类型(有向图:0,有向网:1,无向图:2,无向网:3): ");
   scanf("%d",&(G.kind));
```

```
printf("请输入图的顶点数,边数: ");
scanf("%d,%d",&(G.vexnum),&(G.arcnum));
printf("请输入%d 个顶点的值(<%d 个字符):\n",G.vexnum,MAX_NAME);
for(i=0;i<G.vexnum;++i) /* 构造顶点向量 */
{
   scanf("%s",G.vertices[i].data);
   G.vertices[i].firstarc=NULL;
}
if(G.kind==1||G.kind==3) /* 网 */
   printf("请顺序输入每条弧(边)的权值、弧尾和弧头(以空格作为间隔):\n");
else /* 图 */
   printf("请顺序输入每条弧(边)的弧尾和弧头(以空格作为间隔):\n");
for(k=0;k<G.arcnum;++k) /* 构造表结点链表 */
{
   if(G.kind==1||G.kind==3) /* 网 */
      scanf("%d%s%s",&w,va,vb);
   else /* 图 */
      scanf("%s%s",va,vb);
   i=LocateVex(G,va); /* 弧尾 */
   j=LocateVex(G,vb); /* 弧头 */
   p=(ArcNode*)malloc(sizeof(ArcNode));
   p->adjvex=j;
   if(G.kind==1||G.kind==3) /* 网 */
   {
      p->info=(int *)malloc(sizeof(int));
      *(p->info)=w;
   }
   else
      p->info=NULL; /* 图 */
   p->nextarc=G.vertices[i].firstarc; /* 插在表头 */
   G.vertices[i].firstarc=p;
```

```
      if(G.kind>=2) /* 无向图或网,产生第二个表结点 */
      {
        p=(ArcNode*)malloc(sizeof(ArcNode));
        p->adjvex=i;
        if(G.kind==3) /* 无向网 */
        {
          p->info=(int*)malloc(sizeof(int));
          *(p->info)=w;
        }
        else
          p->info=NULL; /* 无向图 */
        p->nextarc=G.vertices[j].firstarc; /* 插在表头 */
        G.vertices[j].firstarc=p;
      }
    }
    return OK;
}

void DestroyGraph(ALGraph &G)
{ /* 初始条件: 图 G 存在。操作结果: 销毁图 G */
    int i;
    ArcNode *p,*q;
    G.vexnum=0;
    G.arcnum=0;
    for(i=0;i<G.vexnum;++i)
    {
      p=G.vertices[i].firstarc;
      while(p)
      {
        q=p->nextarc;
        if(G.kind%2) /* 网 */
```

```
            free(p->info);
          free(p);
          p=q;
       }
    }
}

VertexType* GetVex(ALGraph G,int v)
{ /*  初始条件: 图 G 存在,v 是 G 中某个顶点的序号。操作结果: 返回 v 的值  */
   if(v>=G.vexnum||v<0)
      exit(ERROR);
   return &G.vertices[v].data;
}

 int FirstAdjVex(ALGraph G,VertexType v)
{ /*  初始条件: 图 G 存在,v 是 G 中某个顶点  */
   /*  操作结果: 返回 v 的第一个邻接顶点的序号。若顶点在 G 中没有邻接顶点,则返回-1 */
   ArcNode *p;
   int v1;
   v1=LocateVex(G,v); /* v1 为顶点 v 在图 G 中的序号  */
   p=G.vertices[v1].firstarc;
   if(p)
      return p->adjvex;
   else
      return -1;
}

int NextAdjVex(ALGraph G,VertexType v,VertexType w)
{ /*  初始条件: 图 G 存在,v 是 G 中某个顶点,w 是 v 的邻接顶点  */
   /*  操作结果: 返回 v 的(相对于 w 的)下一个邻接顶点的序号。  */
   /*                若 w 是 v 的最后一个邻接点,则返回-1 */
```

```
  ArcNode *p;
  int v1,w1;
  v1=LocateVex(G,v); /* v1 为顶点 v 在图 G 中的序号 */
  w1=LocateVex(G,w); /* w1 为顶点 w 在图 G 中的序号 */
  p=G.vertices[v1].firstarc;
  while(p&&p->adjvex!=w1) /* 指针 p 不空且所指表结点不是 w */
    p=p->nextarc;
  if(!p||!p->nextarc) /* 没找到 w 或 w 是最后一个邻接点 */
    return -1;
  else /* p->adjvex==w */
    return p->nextarc->adjvex; /* 返回 v 的(相对于 w 的)下一个邻接顶点的序号 */
}

Boolean visited[MAX_VERTEX_NUM]; /* 访问标志数组(全局量) */
void(*VisitFunc)(char* v); /* 函数变量(全局量) */
void DFS(ALGraph G,int v)
{ /* 从第 v 个顶点出发递归地深度优先遍历图 G。算法 7.5 */
  int w;
  VertexType v1,w1;
  strcpy(v1,*GetVex(G,v));
  visited[v]=TRUE; /* 设置访问标志为 TRUE(已访问) */
  VisitFunc(G.vertices[v].data); /* 访问第 v 个顶点 */
  for(w=FirstAdjVex(G,v1);w>=0;w=NextAdjVex(G,v1,strcpy(w1,*GetVex(G,w))))
    if(!visited[w])
      DFS(G,w); /* 对 v 的尚未访问的邻接点 w 递归调用 DFS */
}

void DFSTraverse(ALGraph G,void(*Visit)(char*))
{ /* 对图 G 作深度优先遍历。算法 7.4 */
  int v;
  VisitFunc=Visit; /* 使用全局变量 VisitFunc,使 DFS 不必设函数指针参数 */
```

```
  for(v=0;v<G.vexnum;v++)
    visited[v]=FALSE; /* 访问标志数组初始化 */
  for(v=0;v<G.vexnum;v++)
    if(!visited[v])
      DFS(G,v); /* 对尚未访问的顶点调用 DFS */
  printf("\n");
}

typedef int QElemType; /* 队列类型 */

#include"LinkQueueDef.h"
#include"LinkQueueAlgo.h"

void BFSTraverse(ALGraph G,void(*Visit)(char*))
{/*按广度优先非递归遍历图 G。使用辅助队列 Q 和访问标志数组 visited。算法 7.6 */
  int v,u,w;
  VertexType u1,w1;
  LinkQueue Q;
  for(v=0;v<G.vexnum;++v)
    visited[v]=FALSE; /* 置初值 */
  InitQueue(Q); /* 置空的辅助队列 Q */
  for(v=0;v<G.vexnum;v++) /* 如果是连通图,只 v=0 就遍历全图 */
    if(!visited[v]) /* v 尚未访问 */
    {
      visited[v]=TRUE;
      Visit(G.vertices[v].data);
      EnQueue(Q,v); /* v 入队列 */
      while(!QueueEmpty(Q)) /* 队列不空 */
      {
        DeQueue(Q,u); /* 队头元素出队并置为 u */
        strcpy(u1,*GetVex(G,u));
```

```
            for(w=FirstAdjVex(G,u1);w>=0;w=NextAdjVex(G,u1,strcpy(w1,*GetVex(G,w))))
               if(!visited[w]) /* w 为 u 的尚未访问的邻接顶点 */
               {
                  visited[w]=TRUE;
                  Visit(G.vertices[w].data);
                  EnQueue(Q,w); /* w 入队 */
               }
         }
      }
   printf("\n");
}

void Display(ALGraph G)
{ /* 输出图的邻接矩阵 G */
   int i;
   ArcNode *p;
   switch(G.kind)
   { case DG: printf("有向图\n");            break;
      case DN: printf("有向网\n");           break;
      case AG: printf("无向图\n");            break;
      case AN: printf("无向网\n");
   }
   printf("%d 个顶点：\n",G.vexnum);
   for(i=0;i<G.vexnum;++i)
      printf("%s ",G.vertices[i].data);
   printf("\n%d 条弧(边):\n",G.arcnum);
   for(i=0;i<G.vexnum;i++)
   {
      p=G.vertices[i].firstarc;
      while(p)
      {
```

```
        if(G.kind<=1) /* 有向 */
        {
          printf("%s→%s ",G.vertices[i].data,G.vertices[p->adjvex].data);
          if(G.kind==DN) /* 网 */
            printf(":%d ",*(p->info));
        }
        else /* 无向(避免输出两次) */
        {
          if(i<p->adjvex)
          {
            printf("%s－%s ",G.vertices[i].data,G.vertices[p->adjvex].data);
            if(G.kind==AN) /* 网 */
              printf(":%d ",*(p->info));
          }
        }
        p=p->nextarc;
      }
      printf("\n");
    }
  }
```

6．文件 ALGraphUse.cpp 是检验 ALGraphAlgo.h 的主程序

```
#include "pubuse.h"
#define MAX_NAME 3 /* 顶点字符串的最大长度+1 */
typedef int InfoType; /* 存放网的权值 */
typedef char VertexType[MAX_NAME]; /* 字符串类型 */
#include"ALGraphDef.h"
#include"ALGraphAlgo.h"

void print(char *i)
{
  printf("%s ",i);
```

```
}

void main()
{
    int i,j,k,n;
    ALGraph g;
    VertexType v1,v2;
    printf("请选择有向图\n");
    CreateGraph(g);
    Display(g);
    printf("深度优先搜索的结果:\n");
    DFSTraverse(g,print);
    printf("广度优先搜索的结果:\n");
    BFSTraverse(g,print);

    DestroyGraph(g);      /* 销毁图  */
}
```

五、实验环境和实验步骤

实验环境：利用 Visual C++集成开发环境进行本实验的操作。

实验步骤：

1．启动 VC++;

2．新建工程/Win32 Console Application，选择输入位置：如“d:\”，输入工程的名称：如“ALGraphDemo”；按“确定”按钮，选择“An Empty Project”，再按“完成”按钮；

3．加载实验一中的 pubuse.h 选中菜单的“project”→“add to project”→“files”选择已存在文件，确定，然后一定将文件 pubuse.h 拷贝到所建的工程目录下；

4．加载实验三中的 LinkQueueDef.h 和 LinkQueueAlgo.h，选中菜单的“project”→“add to project”→“files”选择以上两个已存在文件，确定，然后一定将文件 LinkQueueDef.h 和 LinkQueueAlgo.h 拷贝到所建的工程目录下；

5．新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“ALGraphDef.h”，按“确

定”按钮，在显示的代码编辑区内输入如上的参考程序；

6．新建文件/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“ALGraphAlgo.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

7．新建文件/C++ Source File，选中“添加到工程的复选按钮”，输入文件名“ALGraphUse.cpp”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

8．构件、调试、运行与实验一相同。

六、思考题

实验八：最短路径和关键路径的研究与实现（选做：2 学时）

一、实验目的：

. 掌握图的邻接矩阵、邻接表的表示方法

. 掌握迪杰斯特拉和弗洛伊德的最短路径算法

. 理解拓扑排序，掌握关键路径的算法

. 加深对图的理解，逐步培养解决实际问题的编程能力

二、实验内容:

最短路径和关键路径的实现.

1．问题描述：建立一个网，完成以下操作：

. 选择一种网的存储结构，初始化网;

. 用迪杰斯特拉算法或弗洛伊德算法实现点与点之间的最短路径

. 建立一个 AOE-网,实现关键路径算法.

2．实现要求：

. AOE-网:带权的有向无环网，顶点代表事件，弧代表活动，权代表时间；

. 关键路径:AOE-网中路径长度最长的路径。

三、实验指导:

本实验以图的邻接表存储结构为例，要求完成基本要求，同时对无向图，有向网，无向网也一并实现其相关的操作，课后同学们可以用邻接矩阵式存储结构完成以上操作。

3． 首先将图的链接存储结构定义放在一个头文件：如取名为 ALGraphDef.h。

4． 链接表式存储图的基本操作也放在一个文件中 ALGraphAlgo.h.

3．将函数的测试和主函数组合成一个文件，如取名为文件 ALGraphUse.cpp.

由于要用到队列技术，对于队列的数据结构定义和基本操作函数的描述就可以借助实验三的结果，不必重写。

四、参考程序:

1、最短路径

（1）文件 MatrixGraphDef.h 是定义建立图的邻接矩阵的存储结构

```
#define INFINITY INT_MAX /* 用整型最大值代替∞ */
#define MAX_VERTEX_NUM 20 /* 最大顶点个数 */
typedef enum{DG,DN,AG,AN}GraphKind; /* {有向图,有向网,无向图,无向网} */
typedef struct
{
   VRType adj; /* 顶点关系类型。对无权图，用 1(是)或 0(否)表示相邻否； */
               /* 对带权图，c 则为权值类型 */
   InfoType *info; /* 该弧相关信息的指针(可无) */
}ArcCell,AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
typedef struct
{
   VertexType vexs[MAX_VERTEX_NUM]; /* 顶点向量 */
   AdjMatrix arcs; /* 邻接矩阵 */
   int vexnum,arcnum; /* 图的当前顶点数和弧数 */
   GraphKind kind; /* 图的种类标志 */
}MGraph;
```

（2）文件 MatrixGraphAlgo.h 是描述图的基本操作（矩阵邻接表的存储结构在文件 MatrixGraphDef.h 中定义）

```
int LocateVex(MGraph G,VertexType u)
{ /* 初始条件:图 G 存在,u 和 G 中顶点有相同特征 */
   /* 操作结果:若 G 中存在顶点 u,则返回该顶点在图中位置;否则返回-1 */
   int i;
   for(i=0;i<G.vexnum;++i)
      if(strcmp(u,G.vexs[i])==0)
         return i;
   return -1;
}
```

```
Status CreateDN(MGraph &G)
{ /* 采用数组(邻接矩阵)表示法,构造有向网 G */
  int i,j,k,w,IncInfo;
  char s[MAX_INFO],*info;
  VertexType va,vb;
  printf("请输入有向网 G 的顶点数,弧数,弧是否含其它信息(是:1,否:0): ");
  scanf("%d,%d,%d",&G.vexnum,&G.arcnum,&IncInfo);
  printf("请输入%d 个顶点的值(<%d 个字符):\n",G.vexnum,MAX_NAME);
  for(i=0;i<G.vexnum;++i) /* 构造顶点向量 */
    scanf("%s",G.vexs[i]);
  for(i=0;i<G.vexnum;++i) /* 初始化邻接矩阵 */
    for(j=0;j<G.vexnum;++j)
    {
      G.arcs[i][j].adj=INFINITY; /* 网 */
      G.arcs[i][j].info=NULL;
    }
  printf("请输入%d 条弧的弧尾 弧头 权值(以空格作为间隔): \n",G.arcnum);
  for(k=0;k<G.arcnum;++k)
  {
    scanf("%s%s%d%*c",va,vb,&w);  /* %*c 吃掉回车符 */
    i=LocateVex(G,va);
    j=LocateVex(G,vb);
    G.arcs[i][j].adj=w; /* 有向网 */
    if(IncInfo)
    {
      printf("请输入该弧的相关信息(<%d 个字符): ",MAX_INFO);
      gets(s);
      w=strlen(s);
      if(w)
      {
        info=(char*)malloc((w+1)*sizeof(char));
```

```
            strcpy(info,s);
            G.arcs[i][j].info=info; /* 有向 */
          }
        }
      }
    G.kind=DN;
    return OK;
}
```

(3) 文件 ShorestPathAlgo.h 包含了一个最短路径算法（迪杰斯特拉算法）的实现

```
    #define MAX_NAME 5 /* 顶点字符串的最大长度+1 */
    #define MAX_INFO 20 /* 相关信息字符串的最大长度+1 */
    typedef int VRType;
    typedef char InfoType;
    typedef char VertexType[MAX_NAME];
    typedef int PathMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
    typedef int ShortPathTable[MAX_VERTEX_NUM];
   void ShortestPath_DIJ(MGraph G,int v0,PathMatrix P,ShortPathTable D)
   { /* 用 Dijkstra 算法求有向网 G 的 v0 顶点到其余顶点 v 的最短路径 P[v]及带权长度 */
     /* D[v]。若 P[v][w]为 TRUE,则 w 是从 v0 到 v 当前求得最短路径上的顶点。 */
     /* final[v]为 TRUE 当且仅当 v∈S,即已经求得从 v0 到 v 的最短路径 算法 7.15 */
     int v,w,i,j,min;
     Status final[MAX_VERTEX_NUM];
     for(v=0;v<G.vexnum;++v)
     {
       final[v]=FALSE;
       D[v]=G.arcs[v0][v].adj;
       for(w=0;w<G.vexnum;++w)
         P[v][w]=FALSE; /* 设空路径 */
       if(D[v]<INFINITY)
       {
         P[v][v0]=TRUE;
```

```
            P[v][v]=TRUE;
        }
    }
    D[v0]=0;
    final[v0]=TRUE; /* 初始化,v0 顶点属于 S 集 */
    for(i=1;i<G.vexnum;++i) /* 其余 G.vexnum-1 个顶点 */
    { /* 开始主循环,每次求得 v0 到某个 v 顶点的最短路径,并加 v 到 S 集 */
        min=INFINITY; /* 当前所知离 v0 顶点的最近距离 */
        for(w=0;w<G.vexnum;++w)
            if(!final[w]) /* w 顶点在 V-S 中 */
        if(D[w]<min)
        {
            v=w;
            min=D[w];
        } /* w 顶点离 v0 顶点更近 */
        final[v]=TRUE; /* 离 v0 顶点最近的 v 加入 S 集 */
        for(w=0;w<G.vexnum;++w) /* 更新当前最短路径及距离 */
        {
            if(!final[w]&&min<INFINITY&&G.arcs[v][w].adj<INFINITY&&(min+G.arcs[v][w].adj<D[w]))
            { /* 修改 D[w]和 P[w],w∈V-S */
                D[w]=min+G.arcs[v][w].adj;
                for(j=0;j<G.vexnum;++j)
                    P[w][j]=P[v][j];
                P[w][w]=TRUE;
            }
        }
    }
}
```

【说明】算法的测试和其他算法等代码由同学们自己完成。

2、关键路径：

【说明】下面列出判断有向网关键路径的主要算法描述，在求关键路径中要借助与栈的特性和相关操作，

其他算法和测试代码需要同学们补充完成。

```
#define MAX_NAME 5 /*  顶点字符串的最大长度+1 */
 typedef int InfoType;
 typedef char VertexType[MAX_NAME]; /*  字符串类型  */

 int ve[MAX_VERTEX_NUM]; /*  全局变量(用于算法 7.13 和算法 7.14) */

 void FindInDegree(ALGraph G,int indegree[])
 { /*  求顶点的入度，算法 7.12、7.13 调用  */
    int i;
    ArcNode *p;
    for(i=0;i<G.vexnum;i++)
       indegree[i]=0; /*  赋初值  */
    for(i=0;i<G.vexnum;i++)
    {
       p=G.vertices[i].firstarc;
       while(p)
       {
          indegree[p->adjvex]++;
          p=p->nextarc;
       }
    }
 }

typedef int SElemType; /*  栈类型  */
Status TopologicalOrder(ALGraph G,SqStack &T)
 { /*  算法 7.13   有向网 G 采用邻接表存储结构,求各顶点事件的最早发生时间 ve */
    /* (全局变量)。T 为拓扑序列顶点栈,S 为零入度顶点栈。若 G 无回路,则用栈 T */
    /*  返回 G 的一个拓扑序列,且函数值为 OK,否则为 ERROR */
    int j,k,count,indegree[MAX_VERTEX_NUM];
    SqStack S;
```

```
ArcNode *p;
FindInDegree(G,indegree);/*对各顶点求入度 indegree[0..vernum-1] */
InitStack(S); /* 初始化栈 */
for(j=0;j<G.vexnum;++j) /* 建零入度顶点栈 S */
   if(!indegree[j])
      Push(S,j); /* 入度为 0 者进栈 */
InitStack(T); /* 初始化拓扑序列顶点栈 */
count=0; /* 对输出顶点计数 */
for(j=0;j<G.vexnum;++j) /* 初始化 ve[]=0 (最小值) */
   ve[j]=0;
while(!StackEmpty(S))
{ /* 栈不空 */
   Pop(S,j);
   Push(T,j); /* j 号顶点入 T 栈并计数 */
   ++count;
   for(p=G.vertices[j].firstarc;p;p=p->nextarc)
   { /* 对 j 号顶点的每个邻接点的入度减 1 */
      k=p->adjvex;
      if(--indegree[k]==0) /* 若入度减为 0,则入栈 */
         Push(S,k);
      if(ve[j]+*(p->info)>ve[k])
         ve[k]=ve[j]+*(p->info);
   }
}
if(count<G.vexnum)
{
   printf("此有向网有回路\n");
   return ERROR;
}
else
   return OK;
```

```
}

Status CriticalPath(ALGraph G)
{ /* 算法 7.14 G 为有向网,输出 G 的各项关键活动 */
  int vl[MAX_VERTEX_NUM];
  SqStack T;
  int i,j,k,ee,el;
  ArcNode *p;
  char dut,tag;
  if(!TopologicalOrder(G,&T)) /* 产生有向环 */
    return ERROR;
  j=ve[0];
  for(i=1;i<G.vexnum;i++) /* j=Max(ve[]) 完成点的值 */
    if(ve[i]>j)
      j=ve[i];
  for(i=0;i<G.vexnum;i++) /* 初始化顶点事件的最迟发生时间(最大值) */
    vl[i]=j; /* 完成点的最早发生时间 */
  while(!StackEmpty(T)) /* 按拓扑逆序求各顶点的 vl 值 */
    for(Pop(&T,&j),p=G.vertices[j].firstarc;p;p=p->nextarc)
    {
      k=p->adjvex;
      dut=*(p->info); /* dut<j,k> */
      if(vl[k]-dut<vl[j])
        vl[j]=vl[k]-dut;
    }
  printf(" j   k   dut   ee   el   tag\n");
  for(j=0;j<G.vexnum;++j) /* 求 ee,el 和关键活动 */
    for(p=G.vertices[j].firstarc;p;p=p->nextarc)
    {
      k=p->adjvex;
      dut=*(p->info);
```

```
            ee=ve[j];
            el=vl[k]-dut;
            tag=(ee==el)?'*':' ';
            printf("%2d %2d %3d %3d %3d       %c\n",j,k,dut,ee,el,tag); /* 输出关键活动 */
        }
    printf("关键活动为:\n");
    for(j=0;j<G.vexnum;++j) /* 同上 */
        for(p=G.vertices[j].firstarc;p;p=p->nextarc)
        {
            k=p->adjvex;
            dut=*(p->info);
            if(ve[j]==vl[k]-dut)
                printf("%s→%s\n",G.vertices[j].data,G.vertices[k].data); /* 输出关键活动 */
        }
    return OK;
}
```

五、实验环境和实验步骤

实验环境：利用 Visual C++集成开发环境进行本实验的操作。

六、思考题

1．校园导游图编程要求：

(1) 设计学校的校园平面图，所含景点不少于 10 个。以图中顶点表示校内各景点，存放景点名称、代号、简介等信息；以边表示路径，存放路径长度等相关信息。

(2)为来访客人提供图中任意景点相关信息的查询。

(3)为来访客人提供图中任意景点的问路查询，即查询任意两个景点之间的一条最短的简单路径。

2. 最少换车次数问题：设某城市有 n 个车站，并有 m 条公交线路连接这些车站．设这些公交车都是单向的，这 n 个车站被顺序编号为 0----n-1;

【实现要求】 编号程序，输入该城市的公交线路数、车站个数、以及各公文线路上的各站编号。

实现求得从站 i 出发乘公交车至站 n-1 的最少换车次数。

【编程提示】 利用输入信息构建一张有向图 G(用邻接矩阵 g 表示)，有向图的顶点是车站，若有某条公文线路经 i 站能到达 j 站，就在顶点 i 到顶点 j 之间设置一条权为 1 的有向边(i，j)．这样，从站 x 至站 y 的最少上车次数便对应于图 G 中从点；至点 y 的最短路径长度．而程序要求的换车次数就是上车次数减 1．

实验九：查找和排序算法的实现 （选做：基本 2 学时，扩展 4 学时）

一、实验目的

. 掌握有序表、无序表查找的基本思想及存储、运算的实现

. 熟练掌握常用排序算法的基本思想及实现

. 深刻理解各种算法的特点，并加以灵活应用

. 加深对查找和排序的理解，逐步培养解决实际问题的编程能力

二、实验内容

(一) 基本实验内容：

. 建立一个无序表并实现其上的顺序查找；

. 建立一个有序表并实现其上的折半查找；

. 实现插入排序、起泡排序、快速排序和希尔排序的基本算法；

1．问题描述：利用顺序表,设计一组输入数据（假定为一组整数），能够对顺序表进行如下操作：

. 创建一个新的顺序表，实现动态空间分配的初始化，并将一组随机数组存储到创建的顺序表中；

. 用顺序查找法查询该数组中某个值；

. 用插入、起泡或希尔排序法对随机数进行排序；

. 利用折半查找法查询排序后的顺序表中的某个数据；

. 实现以上算法的元素输出；

. 编写主程序，实现对各不同的算法调用。

2．实现要求：

. 对以上算法一定要编写成为 C（C++）语言函数，组合成模块化的形式；

. “初始化算法”的操作结果：构造一个空的顺序线性表。对顺序表的空间进行动态管理，实现动态分配、回收和增加存储空间；

. “折半、顺序查找算法” 初始条件：顺序线性表 L 已存在；

操作结果：给定的元素 s，如果元素 s 在线性表中则返回其下标 i（1≤i≤L.length），如果元素 s 不在该线性表中，则返回 0 ；

. “选择选择、起泡或希尔排序算法” 初始条件：顺序线性表 L 已存在；

操作结果：对顺序表中的元素按非增或非减的顺序 ；

分析： 修改输入数据，预期输出并验证输出的结果，加深对有关算法的理解。

算法设计思想：

“顺序查找”：从表的最后或第一个记录开始，逐个进行记录的关键字和给定值的比较，若某个记录的关键字和给定值相等，则查找成功返回该关键字的下标；反之查找失败返回 0。

“折半查找”：只适合与有序表的查找；首先确定待查找的范围或区间，然后逐步缩小查找范围：

（1） 设指针 low、high 分别指向待查范围的下界和上界，指针 mid 指向中间位置

mid=int（（low+high）/2）；

（2） 令中间元素的关键字 L.r(mid).key 与给定值 s 相比较：

若 L.r(mid).key<s 则令 low=mid+1

若 L.r(mid).key>s 则令 high=mid-1

若 L.r(mid).key=s 则返回 mid

（3） 重复（2）步直到 low>high，说明表中没有关键字等于 s，查找不成功。

“排序”：一般需要下列两种操作：

（a） 比较两个关键字的大小；

（b） 将记录从一个位置移到另一个位置，实现排序功能。

（二）扩展实验内容：

. 构造哈希函数，实现哈希查找；

. 建立二叉排序树并在其上查找指定关键字。

1．实现要求：

. 创建一个 hash 空表，实现动态空间分配的初始化；

. 构造一个 hash 函数和某种处理冲突的方法将数据元素存放到 hash 表中；

. 设计相应的 hash 查找函数对 hash 表中的元素进行查找；

. 设计二叉树排序算法对数据元素进行排序和相应的查找算法。

2．算法设计：

- hash 构造函数:可选用除留余数法(H(key)=key MOD m)或其他方法；
- 处理冲突:可选开放定址法(Hi=(H(key)+di)MOD m,i=1,2,…,k(k<m-i))或其他方法；
- 二叉排序树:(1) 若左子树不空,则左子树上所有结点的值均小于其根结点的值；

(2) 若右子树不空,则右子树上所有结点的值均大于其根结点的值；

(3) 左右子树均为二叉排序树。

. 二叉排序树的查找:让给定值先与二叉排序树的根结点比较：等于则返回跟结点;大于则与右子树比较；小于则与左子树比较。

三、实验指导

程序通常主要由三部分构成：

1、常量定义（#define），类型定义(typedef)及函数原型声明(#include)；

2、算法函数，即 List_Search_Sq(顺序查找)、List_BinSearch_Sq(折半查找)、InsertSort(插入排序)、BInsertSort（折半插入排序)、P2_InsertSort（2_路插入排序)、ShellSort（希尔排序）等；

3、主函数 main（），调用算法函数完成相应的操作。

四、参考程序

（一） 基本实验的参考程序

1．**文件 pubuse.h** 同实验一

2．文件 compare.h 是对两个数值型关键字的比较约定宏定义

```
#define EQ(a,b) ((a)==(b))
#define LT(a,b) ((a)<(b))
#define LQ(a,b) ((a)<=(b))
```

3．文件 ElemDef.h 是对待查询/排序记录的数据类型定义

```
#define MAXSIZE 20       /* 一个用作示例的小顺序表的最大长度 */
typedef int KeyType;     /* 定义关键字类型为整型 */
typedef int InfoType;
typedef struct
{   KeyType key;              /* 关键字项 */
   InfoType otherinfo; /* 其它数据项，具体类型在主程中定义 */
}sortstruct;                  /* 记录类型 */

typedef struct
```

```
{ sortstruct *r; /* r[0]闲置或用作哨兵单元 */
  int length; /* 顺序表长度 */
}SqList; /* 顺序表类型 */
```

4．文件 SeqBinSearchAlgo.h 中包含对静态查找表的基本操作（元素的类型存储结构在 ElemDef.h 定义）

主要有：顺序查找，折半查找

```
Status List_Creat_Sq(SqList &L,int n)
{
  L.r=(sortstruct *)calloc(n+1,sizeof(sortstruct));
  if(!L.r)
    return ERROR;
  return OK;
}

void Ascend(SqList &L)
{ /* 重建静态查找表为按关键字非降序排序 */
  int i,j,k;
  for(i=1;i<L.length;i++)
  {
    k=i;
    L.r[0]=L.r[i]; /* 待比较值存[0]单元 */
    for(j=i+1;j<=L.length;j++)
      if LT(L.r[j].key,L.r[0].key)
      {
        k=j;
        L.r[0]=L.r[j];
      }
    if(k!=i) /* 有更小的值则交换 */
    {
      L.r[k]=L.r[i];
      L.r[i]=L.r[0];
```

```
    }
  }
}

Status List_Creat_Ord(SqList &L,int n)
{ /* 操作结果: 构造一个含 n 个数据元素的静态按关键字非降序查找表 L */
  /* 数据来自全局数组 r */
  Status f;
  f=List_Creat_Sq(L,n);
  if(f)
    Ascend(L);
  return f;
}

Status List_Destroy_Sq(SqList &L)
{ /* 初始条件: 静态查找表 L 存在。操作结果: 销毁表 L */
  free(L.r);
  L.r=NULL;
  L.length=0;
  return OK;
}

int List_Search_Sq(SqList L,KeyType key)/*顺序查找*/
{ /* 在顺序表 L 中顺序查找其关键字等于 key 的数据元素。若找到，则函数值为 */
  /* 该元素在表中的位置，否则为 0。算法 9.1 */
  int i;
  L.r[0].key=key; /* 哨兵 */
  for(i=L.length;!EQ(L.r[i].key,key);--i); /* 从后往前找 */
  return i; /* 找不到时，i 为 0 */
}
```

```
int List_BinSearch_Sq(SqList L,KeyType key)
{ /* 在有序表 L 中折半查找其关键字等于 key 的数据元素。若找到，则函数值为 */
  /* 该元素在表中的位置，否则为 0。算法 9.2 */
  int low,high,mid,i=0;
  low=1; /* 置区间初值 */
  high=L.length;
  while(low<=high)
  {
    mid=(low+high)/2;
    if EQ(key,L.r[mid].key)  /* 找到待查元素 */
      return i+1;
    else if LT(key,L.r[mid].key)
      high=mid-1; /* 继续在前半区间进行查找 */
    else
      low=mid+1; /* 继续在后半区间进行查找 */
   i++;
  }
  return i;    /* 顺序表中不存在待查元素 */
}

Status Traverse(SqList L,void(*Visit)(sortstruct))
{ /* 初始条件: 静态查找表 L 存在，Visit()是对元素操作的应用函数 */
  /* 操作结果: 按顺序对 L 的每个元素调用函数 Visit()一次且仅一次。 */
  /* 一旦 Visit()失败，则操作失败 */
  sortstruct *p;
  int i;
  p=++L.r; /* p 指向第一个元素 */
  for(i=1;i<=L.length;i++)
    Visit(*p++);
  return OK;
}
```

5．文件 InsertSortAlgo.h 中实现顺序表插入排序的函数(3 个)

```
void InsertSort(SqList &L)
{ /* 对顺序表 L 作直接插入排序。算法 10.1 */
  int i,j;
  for(i=2;i<=L.length;++i)
    if LT(L.r[i].key,L.r[i-1].key) /* "<",需将 L.r[i]插入有序子表 */
    {
      L.r[0]=L.r[i]; /* 复制为哨兵 */
      for(j=i-1;LT(L.r[0].key,L.r[j].key);--j)
    L.r[j+1]=L.r[j]; /* 记录后移 */
      L.r[j+1]=L.r[0]; /* 插入到正确位置 */
    }
}

void BInsertSort(SqList &L)
{ /* 对顺序表 L 作折半插入排序。算法 10.2 */
  int i,j,m,low,high;
  for(i=2;i<=L.length;++i)
  {
    L.r[0]=L.r[i]; /* 将 L.r[i]暂存到 L.r[0] */
    low=1;
    high=i-1;
    while(low<=high)
    { /* 在 r[low..high]中折半查找有序插入的位置 */
      m=(low+high)/2; /* 折半 */
      if LT(L.r[0].key,L.r[m].key)
        high=m-1; /* 插入点在低半区 */
      else
        low=m+1; /* 插入点在高半区 */
    }
    for(j=i-1;j>=high+1;--j)
```

```
        L.r[j+1]=L.r[j]; /* 记录后移 */
      L.r[high+1]=L.r[0]; /* 插入 */
    }
}

void P2_InsertSort(SqList &L)
{ /* 2_路插入排序 */
   int i,j,first,final;
   sortstruct *d;
   d=(sortstruct*)malloc(L.length*sizeof(sortstruct)); /* 生成 L.length 个记录的临时空间 */
   d[0]=L.r[1]; /* 设 L 的第 1 个记录为 d 中排好序的记录(在位置[0]) */
   first=final=0; /* first、final 分别指示 d 中排好序的记录的第 1 个和最后 1 个记录的位置 */
   for(i=2;i<=L.length;++i)
   { /* 依次将 L 的第 2 个～最后 1 个记录插入 d 中 */
     if(L.r[i].key<d[first].key)
     { /* 待插记录小于 d 中最小值，插到 d[first]之前(不需移动 d 数组的元素) */
       first=(first-1+L.length)%L.length; /* 设 d 为循环向量 */
       d[first]=L.r[i];
     }
     else if(L.r[i].key>d[final].key)
     { /* 待插记录大于 d 中最大值，插到 d[final]之后(不需移动 d 数组的元素) */
       final=final+1;
       d[final]=L.r[i];
     }
     else
     { /* 待插记录大于 d 中最小值，小于 d 中最大值，插到 d 的中间(需要移动 d 数组的元素) */
       j=final++; /* 移动 d 的尾部元素以便按序插入记录 */
       while(L.r[i].key<d[j].key)
       {
         d[(j+1)%L.length]=d[j];
         j=(j-1+L.length)%L.length;
```

```
        }
        d[j+1]=L.r[i];
      }
    }
    for(i=1;i<=L.length;i++) /* 把 d 赋给 L.r */
      L.r[i]=d[(i+first-1)%L.length]; /* 线性关系 */
}
```

6．文件 ShellSortAlgo.h 实现希尔排序操作算法

```
void ShellInsert(SqList &L,int dk)
{ /* 对顺序表 L 作一趟希尔插入排序。本算法是和一趟直接插入排序相比，作了以下修改： */
  /* 1.前后记录位置的增量是 dk,而不是 1; */
  /* 2.r[0]只是暂存单元,不是哨兵。当 j<=0 时,插入位置已找到。算法 10.4 */
  int i,j;
  for(i=dk+1;i<=L.length;++i)
    if LT(L.r[i].key,L.r[i-dk].key)
    { /* 需将 L.r[i]插入有序增量子表 */
      L.r[0]=L.r[i]; /* 暂存在 L.r[0] */
      for(j=i-dk;j>0&&LT(L.r[0].key,L.r[j].key);j-=dk)
        L.r[j+dk]=L.r[j]; /* 记录后移，查找插入位置 */
      L.r[j+dk]=L.r[0]; /* 插入 */
    }
}

void print(SqList L)
{
  int i;
  for(i=1;i<=L.length;i++)
    printf("%d ",L.r[i].key);
  printf("\n");
}
```

```
void ShellSort(SqList &L,int dlta[],int t)
{ /* 按增量序列 dlta[0..t-1]对顺序表 L 作希尔排序。算法 10.5 */
   int k;
   for(k=0;k<t;++k)
   {
      ShellInsert(L,dlta[k]); /* 一趟增量为 dlta[k]的插入排序 */
      printf("第%d 趟排序结果: ",k+1);
      print(L);
   }
}
```

7．文件 SearchSortUse.cpp 是检验各种查找，排序算法的程序

```
/* 检验 ShellSortAlgo.h、InsertSortAlgo.h 和 SeqBinSearchAlgo.h 的程序 */
  #include"pubuse.h"          /* 通用常量定义和系统函数原型声明   */
 #include"compare.h"          /* 两个数值型关键字的比较约定宏定义*/
 #include"ElemDef.h"         /*是对待查询/排序记录的数据类型定义   */
 #include "ShellSortAlgo.h"         /*实现希尔排序操作算法*/
 #include "InsertSortAlgo.h"         /* 顺序表插入排序的函数*/
 #include "SeqBinSearchAlgo.h"     /*静态查找表的基本操作*/

/*typedef int InfoType; 定义其它数据项的类型 */
#define N 10
 #define T 3

 void print1(SqList L)
 { //依次输出顺序表每个元素得关键字项和其它信息项
   int i;
   for(i=1;i<=L.length;i++)
      printf("(%d,%d)",L.r[i].key,L.r[i].otherinfo);
   printf("\n");
 }
```

```
void main()
{
   sortstruct d[N]={{49,1},{38,2},{65,3},{97,4},{76,5},{13,6},{27,7},{49,8},{55,9},{4,10}};
   SqList l1,l2,l3,l4,st;
   int i,s,j;
   int dt[T]={5,3,1};

    List_Creat_Sq(st,N); /*  由全局数组产生静态查找表 st */

   for(i=0;i<N;i++) /*  给 l1.r 赋值  */
      st.r[i+1]=d[i];
   st.length=N;

   l4=l2=l3=l1=st; /*  复制顺序表 l2、l3 与 l1 相同  */
   printf("排序前:\n");
   print1(st);
   printf("\n");

   printf("请输入待查找的关键字: ");
   scanf("%d",&s);

   i=List_Search_Sq(st,s); /*  顺序查找  */
   if(i)
      printf("%d\n",(*(st.r+i)).otherinfo);
   else
      printf("没找到\n");

   InsertSort(l1);
   printf("直接插入排序后:\n");
   print1(l1);
```

```
    printf("\n") ;

    BInsertSort(l2);
    printf("折半插入排序后:\n");
    print1(l2);
    printf("\n") ;

    P2_InsertSort(l3);
    printf("2_路插入排序后:\n");
    print1(l3);
    printf("\n") ;

    ShellSort(l4,dt,T);
    printf("shell 排序后: ");
    print1(l4);
    printf("\n");

    printf("请输入待折半查找的关键字: ");
    scanf("%d",&s);
    j=List_BinSearch_Sq(l4,s);
    if(j)
       printf("查找了%d 次！ \n",j);
    else
       printf("没找到\n");
}
```

（二） 扩展实验的参考程序

1．建立 hash 表结构 HashTableDef.h

```
int hashsize[]={11,19,29,37}; /* 哈希表容量递增表，一个合适的素数序列 */
  int m=0; /* 哈希表表长，全局变量 */
```

```
typedef struct
{
   ElemType *elem; /* 数据元素存储基址，动态分配数组 */
   int count; /* 当前数据元素个数 */
   int sizeindex; /* hashsize[sizeindex]为当前容量 */
}HashTable;

#define SUCCESS 1
#define UNSUCCESS 0
#define DUPLICATE -1
```

2．建立 Hash 基本操作 HashTableAlgo.h

```
/*  建立 Hash 基本操作 HashTableAlgo.h    */
Status InitHashTable(HashTable &H)
{ /* 操作结果: 构造一个空的哈希表 */
   int i;
   H.count=0; /* 当前元素个数为 0 */
   H.sizeindex=0; /* 初始存储容量为 hashsize[0] */
   m=hashsize[0];
   H.elem=(ElemType*)malloc(m*sizeof(ElemType));
   if(!H.elem)
      exit(OVERFLOW); /* 存储分配失败 */
   for(i=0;i<m;i++)
      H.elem[i].key=NULLKEY; /* 未填记录的标志 */
   return OK;
}

void DestroyHashTable(HashTable &H)
{ /* 初始条件: 哈希表 H 存在。操作结果: 销毁哈希表 H */
   free(H.elem);
   H.elem=NULL;
   H.count=0;
```

```
    H.sizeindex=0;
}

unsigned Hash(KeyType K)
{ /*  一个简单的哈希函数(m 为表长，全局变量) */
    return K%m;
}

void collision(int &p,int d) /*  线性探测再散列  */
{ /*  开放定址法处理冲突  */
    p=(p+d)%m;
}

Status SearchHash(HashTable H,KeyType K,int &p,int &c)
{ /*  在开放定址哈希表 H 中查找关键码为 K 的元素,若查找成功,以 p 指示待查数据  */
    /*  元素在表中位置,并返回 SUCCESS;否则,以 p 指示插入位置,并返回 UNSUCCESS */
    /* c 用以计冲突次数，其初值置零，供建表插入时参考。算法 9.17 */
    p=Hash(K); /*  求得哈希地址  */
    while(H.elem[p].key!=NULLKEY&&!EQ(K,H.elem[p].key))
    { /*  该位置中填有记录．并且关键字不相等  */
        c++;
        if(c<m)
            collision(p,c); /*  求得下一探查地址 p */
        else
            break;
    }
    if EQ(K,H.elem[p].key)
        return SUCCESS; /*  查找成功，p 返回待查数据元素位置  */
    else
        return UNSUCCESS; /*  查找不成功(H.elem[p].key==NULLKEY)，p 返回的是插入位置  */
}
```

```
Status InsertHash(HashTable &H,ElemType e)
{ /* 查找不成功时插入数据元素 e 到开放定址哈希表 H 中，并返回 OK； */
  /* 若冲突次数过大，则重建哈希表。算法 9.18 */
  int c,p;
  c=0;
  if(SearchHash(H,e.key,p,c)) /* 表中已有与 e 有相同关键字的元素 */
    return DUPLICATE;
  else if(c<hashsize[H.sizeindex]/2) /* 冲突次数 c 未达到上限,(c 的阀值可调) */
  { /* 插入 e */
    H.elem[p]=e;
    ++H.count;
    return OK;
  }
  else
    RecreateHashTable(H); /* 重建哈希表 */
  return ERROR;
}

void RecreateHashTable(HashTable &H) /* 重建哈希表 */
{ /* 重建哈希表 */
  int i,count=H.count;
  ElemType *p,*elem=(ElemType*)malloc(count*sizeof(ElemType));
  p=elem;
  printf("重建哈希表\n");
  for(i=0;i<m;i++) /* 保存原有的数据到 elem 中 */
    if((H.elem+i)->key!=NULLKEY) /* 该单元有数据 */
      *p++=*(H.elem+i);
  H.count=0;
  H.sizeindex++; /* 增大存储容量 */
  m=hashsize[H.sizeindex];
```

```
  p=(ElemType*)realloc(H.elem,m*sizeof(ElemType));
  if(!p)
    exit(OVERFLOW); /* 存储分配失败 */
  H.elem=p;
  for(i=0;i<m;i++)
    H.elem[i].key=NULLKEY; /* 未填记录的标志(初始化) */
  for(p=elem;p<elem+count;p++) /* 将原有的数据按照新的表长插入到重建的哈希表中 */
    InsertHash(H,*p);
}

void TraverseHash(HashTable H,void(*Vi)(int,ElemType))
{ /* 按哈希地址的顺序遍历哈希表 */
  int i;
  printf("哈希地址 0～%d\n",m-1);
  for(i=0;i<m;i++)
    if(H.elem[i].key!=NULLKEY) /* 有数据 */
      Vi(i,H.elem[i]);
}

Status Find(HashTable H,KeyType K,int &p)
{ /* 在开放定址哈希表 H 中查找关键码为 K 的元素,若查找成功,以 p 指示待查数据 */
  /* 元素在表中位置,并返回 SUCCESS;否则,返回 UNSUCCESS */
  int c=0;
  p=Hash(K); /* 求得哈希地址 */
  while(H.elem[p].key!=NULLKEY&&!EQ(K,H.elem[p].key))
  { /* 该位置中填有记录．并且关键字不相等 */
    c++;
    if(c<m)
      collision(p,c); /* 求得下一探查地址 p */
    else
      return UNSUCCESS; /* 查找不成功(H.elem[p].key==NULLKEY) */
```

```
    }
    if EQ(K,H.elem[p].key)
      return SUCCESS; /*  查找成功，p 返回待查数据元素位置  */
    else
      return UNSUCCESS; /*  查找不成功(H.elem[p].key==NULLKEY) */
  }
```

3．建立 HashtableUse.cpp　实现对 Hash 函数生成算法的验证

```
#include "pubuse.h"
#include "HashTableDef.h"
#include "HashTableAlgo.h"
#define N 10

 void print(int p,ElemType r)
 {
   printf("address=%d (%d,%d)\n",p,r.key,r.ord);
 }

 void main()
 {
   ElemType r[N]={{17,1},{60,2},{29,3},{38,4},{1,5},{2,6},{3,7},{4,8},{60,9},{13,10}};
   HashTable h;
   int i,p;
   Status j;
   KeyType k;
   InitHashTable(h);
   for(i=0;i<N-1;i++)
   { /*  插入前 N-1 个记录  */
     j=InsertHash(h,r[i]);
     if(j==DUPLICATE)
       printf("表中已有关键字为%d 的记录，无法再插入记录(%d,%d)\n",r[i].key,r[i].key,r[i].ord);
   }
```

```
    printf("按哈希地址的顺序遍历哈希表:\n");
    TraverseHash(h,print);
    printf("请输入待查找记录的关键字: ");
    scanf("%d",&k);
    j=Find(h,k,p);
    if(j==SUCCESS)
      print(p,h.elem[p]);
    else
      printf("没找到\n");
    j=InsertHash(h,r[i]); /* 插入第 N 个记录 */
    if(j==ERROR) /* 重建哈希表 */
      j=InsertHash(h,r[i]); /* 重建哈希表后重新插入第 N 个记录 */
    printf("按哈希地址的顺序遍历重建后的哈希表:\n");
    TraverseHash(h,print);
    printf("请输入待查找记录的关键字: ");
    scanf("%d",&k);
    j=Find(h,k,p);
    if(j==SUCCESS)
      print(p,h.elem[p]);
    else
      printf("没找到\n");
    DestroyHashTable(h);
}
```

五、实验环境和实验步骤

实验环境：利用 Visual C++集成开发环境进行本实验的操作。

实验步骤：

1．启动 VC++;

2.新建工程/Win32 Console Application，选择输入位置：如 “d:\”，输入工程的名称：如 “SearchSortDemo”；按“确定”按钮，选择“An Empty Project”，再按“完成”按钮，

3．加载实验一中的 pubuse.h 选中菜单的”project”--“add to project”-- “files”选择已存在文件，确定，然后一定将文件 pubuse.h 拷贝到所建的工程目录下；

4. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“compare.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

5. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“ElemDef.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

6. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“SeqBinSearchAlgo.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

7. 新建文件/C/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“InsertSortAlgo.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

8. 新建文件/C++ Header File，选中“添加到工程的复选按钮”，输入文件名“ShellSortAlgo.h”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

9. 新建文件/C++ Source File，选中“添加到工程的复选按钮”，输入文件名“SearchSortUse.cpp”，按“确定”按钮，在显示的代码编辑区内输入如上的参考程序；

10．构件、调试、运行与实验一相同，在此不再复述；

六、思考题

1. 编程实现对学生成绩的信息管理，现有某班学生的各科考试成绩数据信息，要求对同学生的各科成绩进行累加，按学生的总成绩排名次和单科成绩排序，并且能对学生的单科和总分成绩进行查询并输出查询结果。

2．比较各种查找算法的效率，试对上一题用不同的查找算法进行实验。

3．设计对一组整数的 Hash 查找方案和冲突解决方案，对每一种方案进行分析，选出一种最佳的组合。