

华中科技大学

课程实验报告

课程名称： 数据结构实验

专业班级 CS2310

学 号 U202315719

姓 名 郭金环

指导教师 郑渤龙

报告日期 2024 年 6 月 16 日

计算机科学与技术学院

目 录

1	基于链式存储结构的线性表实现.....	1
1.1	问题描述	1
1.2	系统设计	1
1.3	系统实现	5
1.4	系统测试	21
1.5	实验小结	34
2	基于邻接表的图实现	36
2.1	问题描述	36
2.2	系统设计	36
2.3	系统实现	40
2.4	系统测试	57
2.5	实验小结	68
3	课程的收获和建议	70
3.1	基于顺序存储结构的线性表实现	70
3.2	基于链式存储结构的线性表实现	70
3.3	基于二叉链表的二叉树实现	71
3.4	基于邻接表的图实现	71
4	附录 A 基于顺序存储结构线性表实现的源程序	73
5	附录 B 基于链式存储结构线性表实现的源程序	99
6	附录 C 基于二叉链表二叉树实现的源程序	125
7	附录 D 基于邻接表图实现的源程序	159

1 基于链式存储结构的线性表实现

1.1 问题描述

实验要求：

通过实验达到：

1. 加深对线性表的概念、基本运算的理解；
2. 熟练掌握线性表的逻辑结构与物理结构的关系；
3. 物理结构采用单链表, 熟练掌握线性表的基本运算的实现。

通过学习，我学到具有链接存储结构的线性表用一组地址任意的存储单元存放线性表中的数据元素，逻辑上相邻的元素在物理上不要求也相邻，不能随机存取。基于链式存储结构的线性表具有以下优势：

1. 动态分配内存：链表对于处理不可预知数量的数据非常灵活，可以根据需要动态调整存储空间，避免了固定容量的限制。
2. 插入和删除效率高：链表的节点之间通过指针连接，因此在链表中插入和删除节点的操作相对较快。只需修改指针的指向，不需要移动节点的位置，时间复杂度为 $O(1)$ 。而在数组中插入和删除元素通常需要移动其他元素，时间复杂度为 $O(n)$ 。
3. 空间利用率高：链表在存储空间上不需要连续的内存块。每个节点可以在任意位置创建，只需要指向下一个节点的指针即可。这样可以更加灵活地利用内存，减少空间浪费。
4. 数据长度可变：链表的长度可以随时改变，可以根据需要动态增加或减少节点。在需要频繁地改变数据长度的情况下，链表是一个便捷的选择。

同时，基于链式存储结构的线性表也存在需要更多内存，遍历困难、不易于查询等问题。

1.2 系统设计

1.2.1 头文件和预定义

头文件定义如下

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

预定义常量和类型表达式

```
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
typedef int status;
typedef int ElemType;
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
typedef struct LNode{ // 单链表（链式结构）结点的定义
    ElemType data;
    struct LNode *next;
}LNode,*LinkList;
typedef struct{ // 线性表的管理表定义
    struct { char name[30];
            LinkList L;
    } elem[10];
    int length;
    int listsize;
}LISTS;
```

1.2.2 演示系统菜单的结构

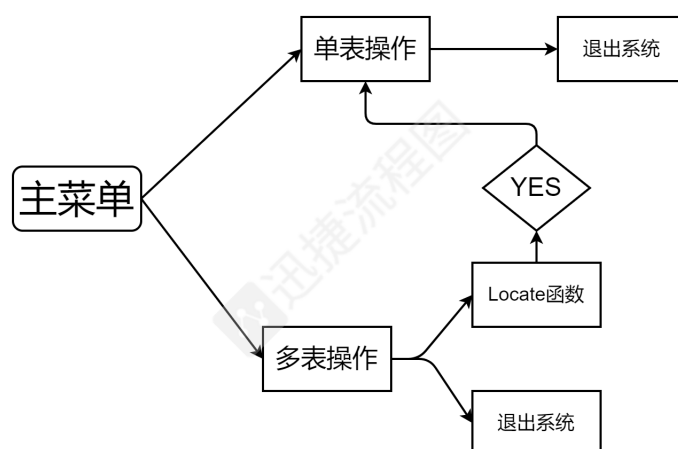


图 1-1 菜单的结构

演示系统共有 17 个功能，其中 1-16 为单表的初始化，销毁，清空，判空，求表长，获得元素，查找元素，获得前驱、后继，插入，删除，遍历共 12 个基本功能和翻转、求倒数第 n 个结点，排序，以文件形式保存共 4 个附加功能。

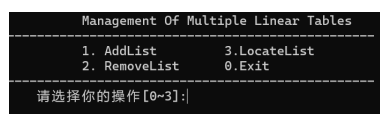


图 1-2 多表管理

功能 17 为多表操作，管理多个线性表的查找、添加、移除等功能，可通过功能 3 自由切换到管理的每个表，并可单独对某线性表进行单线性表的所有操作。

1.2.3 基本功能函数

1. 初始化表：函数名称是 `InitList(L)`；初始条件是线性表 L 不存在；操作结果是构造一个空的线性表；
2. 销毁表：函数名称是 `DestroyList(L)`；初始条件是线性表 L 已存在；操作结果是销毁线性表 L ；
3. 清空表：函数名称是 `ClearList(L)`；初始条件是线性表 L 已存在；操作结果是将 L 重置为空表；
4. 判定空表：函数名称是 `ListEmpty(L)`；初始条件是线性表 L 已存在；操作结果是若 L 为空表则返回 `TRUE`, 否则返回 `FALSE`；

5. 求表长: 函数名称是 `ListLength(L)`; 初始条件是线性表已存在; 操作结果是返回 `L` 中数据元素的个数;
6. 获得元素: 函数名称是 `GetElem(L,i,e)`; 初始条件是线性表已存在, $1 \leq i \leq \text{ListLength}(L)$; 操作结果是用 `e` 返回 `L` 中第 `i` 个数据元素的值;
7. 找元素: 函数名称是 `LocateElem(L,e,compare())`; 初始条件是线性表已存在; 操作结果是返回 `L` 中第 1 个与 `e` 满足关系 `compare` \cap 关系的数据元素的位置, 若这样的数据元素不存在, 则返回值为 0;
8. 获得前驱: 函数名称是 `PriorElem(L,cur,pre)`; 初始条件是线性表 `L` 已存在; 操作结果是若 `cur` 是 `L` 的数据元素, 且不是第一个, 则用 `pre` 返回它的前驱, 否则操作失败, `pre` 无定义;
9. 获得后继: 函数名称是 `NextElem(L,cur,next)`; 初始条件是线性表 `L` 已存在; 操作结果是若 `cur` 是 `L` 的数据元素, 且不是最后一个, 则用 `next` 返回它的后继, 否则操作失败, `next` 无定义;
10. 插入元素: 函数名称是 `ListInsert(L,i,e)`; 初始条件是线性表 `L` 已存在, $1 \leq i \leq \text{ListLength}(L)+1$; 操作结果是在 `L` 的第 `i` 个位置之前插入新的数据元素 `e`;
11. 删除元素: 函数名称是 `ListDelete(L,i,e)`; 初始条件是线性表 `L` 已存在且非空, $1 \leq i \leq \text{ListLength}(L)$; 操作结果: 删除 `L` 的第 `i` 个数据元素, 用 `e` 返回其值;
12. 遍历表: 函数名称是 `ListTraverse(L,visit())`, 初始条件是线性表 `L` 已存在; 操作结果是依次对 `L` 的每个数据元素调用函数 `visit()`;

1.2.4 附加功能函数

1. 链表翻转: 函数名称是 `reverseList(L)`, 初始条件是线性表 `L` 已存在; 操作结果是将 `L` 翻转;
2. 删除链表的倒数第 `n` 个结点: 函数名称是 `RemoveNthFromEnd(L,n)`; 初始条件是线性表 `L` 已存在且非空, 操作结果是该链表中倒数第 `n` 个节点;
3. 链表排序: 函数名称是 `sortList(L)`, 初始条件是线性表 `L` 已存在; 操作结果是将 `L` 由小到大排序;
4. 实现线性表的文件形式保存: 其中, \square 需要设计文件数据记录格式, 以高效保存线性表数据逻辑结构 (D,R) 的完整信息; \square 需要设计线性表文件保存和加载操作合理模式。

- (a) 文件写入：函数名称是 `SaveList(L,FileName)`；初始条件是线性表 `L` 已存在；操作结果是将 `L` 的元素写到名称为 `FileName` 的文件中。
 - (b) 文件读出：函数名称是 `LoadList(L,FileName)`；初始条件是线性表 `L` 不存在；操作结果是将文件 `FileName` 中的元素读到表 `L` 中。
5. 实现多个线性表管理：设计相应的数据结构管理多个线性表的查找、添加、移除等功能。
- (a) 增加线性表：函数名称是 `AddList(Lists, ListName)`；初始条件是名称为 `ListName` 的线性表不存在于线性表集合中；操作结果是在 `List`s 中创建一个名称为 `ListName` 的线性表并初始化。
 - (b) 移除线性表：函数名称是 `RemoveList(Lists, ListName)`；初始条件是名称为 `ListName` 的线性表存在于线性表集合中；操作结果是将该线性表移除。
 - (c) 查找线性表：函数名称是 `LocateList(Lists, ListName)`；初始条件是名称为 `ListName` 的线性表存在于线性表集合中；操作结果是返回该线性表在 `List`s 中的逻辑索引。

1.3 系统实现

`op` 初始化为 1，整个菜单通过变量 `op` 获取所选择的功能选项，1 至 17 分别为不同的功能。通过 `switch` 语句执行不同的功能，每次执行完该功能后通过 `break` 跳出 `switch` 语句，继续执行 `while` 循环，重新读取 `op`。通过外层的 `while` 循环实现多次选择并执行功能，直到输入 0，循环结束，退出演示系统。对于多表操作，与单表操作类似，输入 0 后退出多表操作，回到单表操作系统。

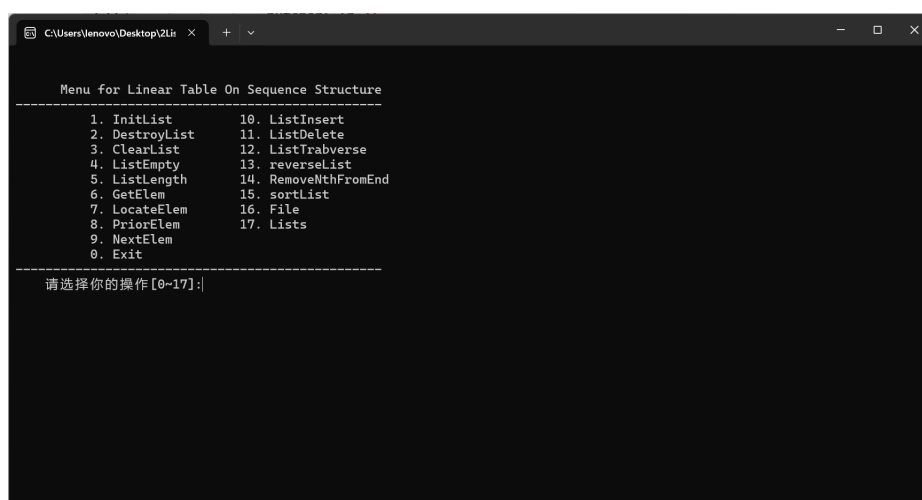


图 1-3 演示系统

1.3.1 初始化表

输入: 顺序表

输出: 函数执行状态

算法思想: 如果顺序表 L 不存在, 则为 L 结点分配存储空间, 将 L->next 结点置空, 返回状态 OK; 否则, 返回状态 INFEASIBLE。

```
status InitList(LinkList &L)
{
    if (L==NULL){
        L=(LinkList )malloc(sizeof(LinkList));
        L->next=NULL;
        return OK;
    }
    else return INFEASIBLE;
}
```

1.3.2 销毁表

输入: 顺序表

输出: 函数执行状态

算法思想: 若 L 为 NULL, 返回 INFEASIBLE。否则用 while 循环依次释放所有节点的储存空间, 再将 L 置为 NULL, 返回 OK。

```
status DestroyList(LinkList &L)
{
    if(L==NULL) return INFEASIBLE;
    else {
        LinkList p=L,temp;
        while(p){
            temp=p->next;
            free(p);
            p=temp;
        }
        L=NULL;
        return OK;
    }
}
```

1.3.3 清空表

输入: 顺序表

输出: 函数执行状态

算法思想: 若 L 为 NULL, 返回 INFEASIBLE。否则用 while 循环释放所有节点的储存空间, 再将 L->next 置为 NULL, 返回 OK。

```
status ClearList(LinkList &L)
{
    if(L==NULL) return INFEASIBLE;
    else {
        LinkList p=L->next,temp;
        while(p){
            temp=p->next;
            free(p);
        }
        L->next=NULL;
        return OK;
    }
}
```

```
                p=temp;
            }
            L->next=NULL;
            return OK;
        }
    }
```

1.3.4 判定空表

输入: 顺序表

输出: 函数执行状态

算法思想: 若 L 为 NULL, 返回 INFEASIBLE。否则判断 L->next 是否为 NULL, 若为 NULL, 则链表为空, 返回 TRUE, 否则链表不为空, 返回 FALSE。

```
status ListEmpty(LinkList L)
{
    if(L==NULL) return INFEASIBLE;
    else {
        if(L->next==NULL) return TRUE;
        else return FALSE;
    }
}
```

1.3.5 求表长

输入: 顺序表

输出: 顺序表的长度

算法思想: 如果顺序表 L 不存在, 则返回 INFEASIBLE; 如果 L 存在, 从头开始循环遍历: 若指针不为空则移动到下一个结点, 长度加 1。

```
int ListLength(LinkList L)
{
    if(L==NULL) return INFEASIBLE;
    else {
```

```
        int i=0;
        while(L){
            L=L->next;
            i++;
        }
        return i-1;
    }
}
```

1.3.6 获得元素

输入: 顺序表

输出: L 中第 i 个数据元素的值 e

算法思想: 若 L 为 NULL, 返回 INFEASIBLE。若 i 小于 1 或大于链表的长度, 则返回 ERROR。否则遍历至第 i 个元素, 返回其数据域的值。

```
status GetElem(LinkList L, int i, ElemType &e)
{
    if(L==NULL) return INFEASIBLE;
    else {
        int j=0;
        for(j; j<i&&L!=NULL; j++){
            L=L->next;
        }
        if(i==0||L==NULL) return ERROR;
        else {
            e=L->data;
            return OK;
        }
    }
}
```

1.3.7 查找元素

输入: 顺序表

输出: L 中第 1 个与 e 相等的元素的序号

算法思想: 若 L 为 NULL, 返回 INFEASIBLE。否则遍历链表, 查找是否有值与 e 相同的元素, 若找到相应元素则返回此元素在链表中的位置序号 i, 否则返回 ERROR。

```
status LocateElem(LinkList L, ElemType e)
{
    if(L==NULL) return INFEASIBLE;
    else {
        int i=1;
        while(L){
            if(L->data==e) return i-1;
            L=L->next;
            i++;
        }
        if(i==0||L==NULL) return ERROR;
        else {
            e=L->data;
            return OK;
        }
    }
}
```

1.3.8 获得前驱

输入: 顺序表 L, 元素 e, 引用参数 pre。

输出: 函数的执行状态。

算法思想: 如果顺序表 L 不存在, 则返回 INFEASIBLE; 否则遍历顺序表, 若当前节点的后继值为 e, 则将当前节点赋给 pre, 返回 OK, 若未找到返回 ERROR。

```
status PriorElem(LinkList L, ElemType e, ElemType &pre)
```

```
{  
    if(L==NULL) return INFEASIBLE;  
    if(L->next==NULL) return ERROR;  
    LinkList p1=L->next, p2=p1->next;  
    while(p2){  
        if(p2->data==e){  
            pre=p1->data;  
            return OK;  
        }  
        p1=p1->next;  
        p2=p1->next;  
    }  
    return ERROR;  
}
```

1.3.9 获得后继

输入: 顺序表 L, 元素 e, 引用参数 next。

输出: 函数的执行状态.

算法思想: 如果顺序表 L 不存在, 则返回 INFEASIBLE; 否则遍历顺序表, 若当前节点的值为 e, 则将当前节点的后继节点赋值给 next, 返回 OK, 否则返回 ERROR

```
status NextElem(LinkList L, ElemType e, ElemType &next)  
{  
    if(L==NULL) return INFEASIBLE;  
    if(L->next==NULL) return ERROR;  
    LinkList p1=L->next, p2=p1->next;  
    while(p2){  
        if(p1->data==e){  
            next=p2->data;  
            return OK;  
        }  
    }  
}
```

```
    }
    p1=p1->next;
    p2=p1->next;
}
return ERROR;
}
```

1.3.10 插入元素

输入: 顺序表 L, 插入位置 i, 插入元素 e。

输出: 函数的执行状态。

算法的思想描述: 如果顺序表 L 不存在, 则返回 INFEASIBLE; 否则判断 i 值是否符合为正数, 不为正则返回 ERROR; 如果 i 值合法, 增加一个新的结点用于存储插入元素, 将插入位置前的结点指向新插入的节点, 将新节点指向插入位置后一个节点, 返回 OK。

```
status ListInsert(LinkList &L,int i,ElemType e)
{
    if(L==NULL) return INFEASIBLE;
    if(i<=0) return ERROR;
    LinkList p1=L,insert=(LinkList) malloc(sizeof(LNode));
    for(int j=0;j<i-1;j++){
        if(p1->next==NULL) return ERROR;
        p1=p1->next;
    }
    insert->data=e;
    insert->next=p1->next;
    p1->next=insert;
    return OK;
}
```

1.3.11 删除元素

输入: 顺序表 L, 删除位置 i, 删除元素 e 的地址。

输出: 函数的执行状态。

算法的思想描述: 如果顺序表 L 不存在, 则返回 INFEASIBLE; 否则判断 i 值是否符合为正数, 不为正则返回 ERROR; 如果 i 值合法, 则遍历至链表第 i-1 个元素, 将其指针域置为其后继的后继, 数据赋给 e, 最后释放其后继节点, 返回 OK。

```
status ListDelete(LinkList &L,int i,ElemType &e)
{
    if(L==NULL) return INFEASIBLE;
    if(i==0) return ERROR;
    LinkList p1=L,p2=L->next;
    int j;
    for(int j=0;j<i-1;j++){
        if(p1->next==NULL) return ERROR;
        p1=p1->next;
        p2=p1->next;
    }
    if(p2==NULL) return ERROR;
    e=p2->data;
    p1->next=p2->next;
    free(p2);
    return OK;
}
```

1.3.12 遍历表

输入: 顺序表 L

输出: 函数的执行状态。

算法的思想描述: 如果顺序表 L 不存在, 则返回 INFEASIBLE; 否则遍历 L 并输出每个结点的值

```
status ListTraverse(LinkList L)
{
    if(L==NULL) return INFEASIBLE;
    LinkList p=L->next;
    if(p==NULL) return OK;
    printf("%d",p->data);
    p=p->next;
    while(p){
        printf("□%d",p->data);
        p=p->next;
    }
    return OK;
}
```

1.3.13 链表翻转

输入: 顺序表 L

输出: 翻转后的头结点。

算法的思想描述: 如果链表为空或者只有一个节点, 则直接返回头节点; 否则使用循环遍历链表, 每次迭代中, 将 `cur` 的 `next` 指针指向 `pre`, 然后更新 `pre`, `cur`, `next` 指针, 循环结束后, 将头节点的 `next` 指针指向 `pre`, 完成链表的反转操作, 返回反转后的链表头节点。

```
LinkList reverseList(LinkList &head) {
    if (head == NULL || head->next == NULL) {
        return head;
    }

    LinkList pre = NULL, cur = head->next, next = NULL;
    while (cur != NULL) {
        next = cur->next;
        cur->next = pre;
    }
```



```

        pre = cur;
        cur = next;
    }
    head->next = pre;

    return head;
}
    
```

1.3.14 删除链表的倒数第 n 个结点

输入: 顺序表 L, 序号 n

输出: 函数执行状态

算法思想: 如流程图

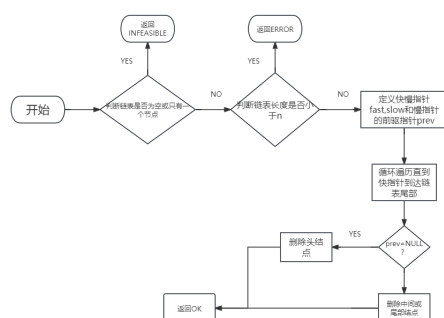


图 1-4 删除链表的倒数第 n 个结点

```

status RemoveNthFromEnd(LinkList &L, int n) {
    if (L == NULL || L->next == NULL) {
        return INFEASIBLE;
    }

    LinkList fast = L->next, slow = L->next;
    for (int i = 0; i < n; i++) {
        if (fast == NULL) {
            return ERROR;
        }
        fast = fast->next;
    }
    
```

```
    }

    LinkList prev = NULL;
    while (fast != NULL) {
        prev = slow;
        slow = slow->next;
        fast = fast->next;
    }

    if (prev == NULL) {
        LinkList temp = L->next;
        L->next = L->next->next;
        free(temp);
    } else {
        prev->next = slow->next;
        free(slow);
    }
    return OK;
}
```

1.3.15 链表排序

输入: 顺序表 L

输出: 函数执行状态

算法思想: 采用插入排序的方法对链表内元素进行大小排序

```
status sortList(LinkList &L) {
    if (L == NULL || L->next == NULL) {
        return INFEASIBLE;
    }

    LinkList p = L->next, q = NULL;
```

```
ElemType temp;
while (p != NULL) {
    q = p->next;
    while (q != NULL) {
        if (p->data > q->data) {
            temp = p->data;
            p->data = q->data;
            q->data = temp;
        }
        q = q->next;
    }
    p = p->next;
}
return OK;
}
```

1.3.16 写文件

输入: 顺序表 L

输出: 函数执行状态

算法思想: 打开文件, 若 L 为 NULL, 返回 INFEASIBLE。否则遍历链表将所有元素的数据域写入文件, 关闭文件, 返回 OK

```
status SaveList(LinkList L, char FileName[])
{
    FILE *fp;
    fp = fopen(FileName, "w");
    if (fp == NULL || L == NULL) {
        return INFEASIBLE;
    }
    LinkList p = L->next;
    if (p == NULL) return OK;
```

```
        while(p) {
            fprintf(fp, "%d", p->data);
            p=p->next;
        }
        fclose(fp);
        return OK;
    }
```

1.3.17 读文件

输入: 顺序表 L

输出: 函数执行状态

算法思想: 打开文件, 若 L 不为 NULL, 返回 INFEASIBLE。否则从文件中每读入一个数据创建一个节点, 并置为上一个结点的后继, 直到读取完所有数据, 关闭文件, 返回 OK。

```
status LoadList(LinkList &L, char FileName[])
{
    FILE *fp;
    fp = fopen(FileName, "r");
    if (fp == NULL || L!=NULL) {
        return INFEASIBLE;
    }
    int value;
    L=(LinkList) malloc(sizeof(LNode));
    LinkList p=L;
    while (fscanf(fp, "%d", &value) != EOF) {
        p->next=(LinkList) malloc(sizeof(LNode));
        p=p->next;
        p->data = value;
    }
    p->next=NULL;
```

```
        fclose(fp);  
        return OK;  
        /***** End 2 *****/  
    }
```

1.3.18 多个线性表的添加

输入: 多表 Lists

输出: 函数执行状态

算法思想: 将读入的表名复制给要添加的表, 将新加的表置为 NULL, 多表长度加 1, 返回 OK。

```
status AddList(LISTS &Lists, char ListName[])  
{  
    strcpy(Lists.elem[Lists.length].name, ListName);  
    Lists.elem[Lists.length].L= NULL;  
    InitList(Lists.elem[Lists.length].L);  
    Lists.length++;  
    return OK;  
}  
}
```

1.3.19 多个线性表的移除

输入: 多表 Lists

输出: 函数执行状态

算法思想: 在 Lists 中查找名称为 ListName 的线性表, 如果可以找到, 将此表销毁, 将其之后的表依次向前移动并将多表的长度减一, 返回 OK, 否则返回 ERROR

```
status RemoveList(LISTS &Lists, char ListName[])  
{  
    for(int i=0; i<Lists.length; i++){
```

```
        if(strcmp(ListName, Lists.elem[i].name)==0){
            DestroyList(Lists.elem[i].L);
            for(int j=i; j<Lists.length-1; j++){
                strcpy(Lists.elem[j].name,
                    Lists.elem[j+1].name);
                Lists.elem[j].L = Lists.elem[j+1].L;
            }
            Lists.length--;
            return OK;
        }
    }
    return ERROR;
}
```

1.3.20 多个线性表的查找

输入: 多表 Lists

输出: 函数执行状态

算法思想: 在 Lists 中查找名称为 ListName 的线性表, 如果可以找到, 返回线性表的逻辑索引, 否则返回 0。

```
int LocateList(LISTS Lists, char ListName[])
{
    for(int i=0; i<Lists.length; i++){
        if(strcmp(ListName, Lists.elem[i].name)==0)
            return i+1;
    }
    return 0;
}
```

1.3.21 对多线性表的单表操作

输入: Lists.elem[n].

输出: 无

算法思想: 将主函数中 case1-16 整合成一个函数

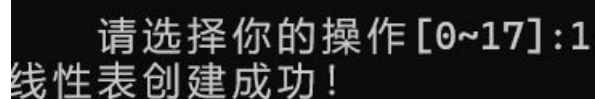
1.4 系统测试

1.4.1 InitList 测试

线性表未创建过

预测结果: 线性表创建成功!

实际结果:



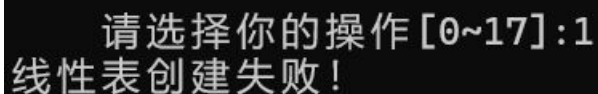
```
请选择你的操作[0~17]:1  
线性表创建成功!
```

图 1-5 测试 1

线性表已创建过

预测结果: 线性表创建失败!

实际结果:



```
请选择你的操作[0~17]:1  
线性表创建失败!
```

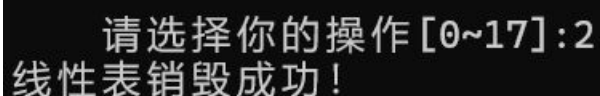
图 1-6 测试 2

1.4.2 DestroyList 测试

线性表存在

预测结果: 线性表销毁成功!

实际结果:



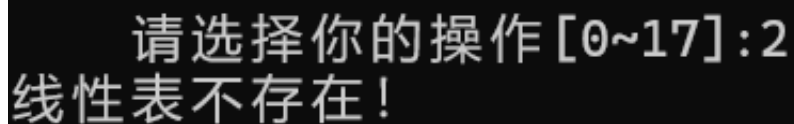
```
请选择你的操作[0~17]:2  
线性表销毁成功!
```

图 1-7 测试 1

线性表不存在

预测结果：线性表不存在！

实际结果：



```
请选择你的操作[0~17]:2
线性表不存在!
```

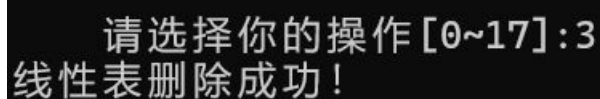
图 1-8 测试 2

1.4.3 ClearList 测试

线性表存在

预测结果：线性表删除成功！

实际结果：



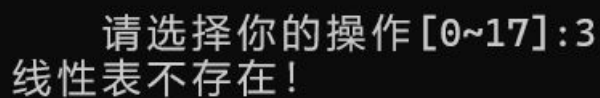
```
请选择你的操作[0~17]:3
线性表删除成功!
```

图 1-9 测试 1

线性表不存在

预测结果：线性表不存在！

实际结果：



```
请选择你的操作[0~17]:3
线性表不存在!
```

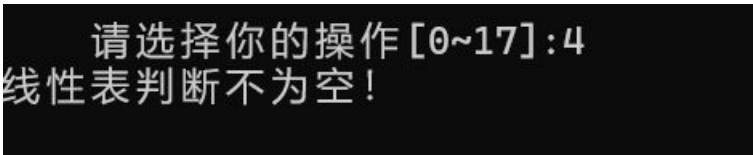
图 1-10 测试 2

1.4.4 ListEmpty 测试

线性表存在且不为空

预测结果：线性表判断不为空！

实际结果：



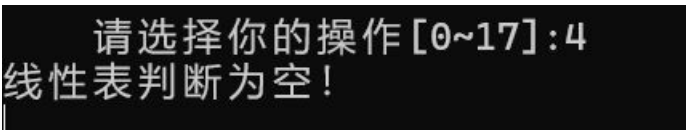
请选择你的操作[0~17]:4
线性表判断不为空!

图 1-11 测试 1

线性表存在且为空

预测结果：线性表判断为空！

实际结果：



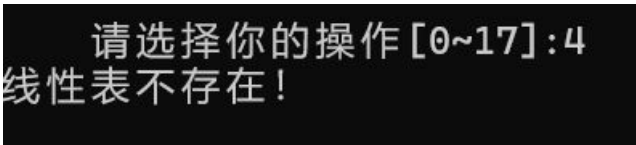
请选择你的操作[0~17]:4
线性表判断为空!

图 1-12 测试 2

线性表不存在

预测结果：线性表不存在！

实际结果：



请选择你的操作[0~17]:4
线性表不存在!

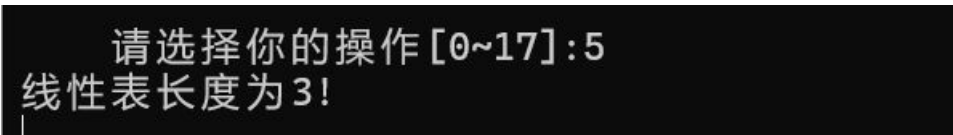
图 1-13 测试 3

1.4.5 ListLength 测试

线性表存在（输入 1 3 5）

预测结果：线性表长度为 3!

实际结果：



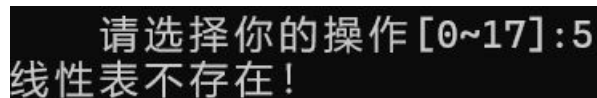
请选择你的操作[0~17]:5
线性表长度为3!

图 1-14 测试 1

线性表不存在

预测结果：线性表不存在！

实际结果:



```
请选择你的操作[0~17]:5  
线性表不存在!
```

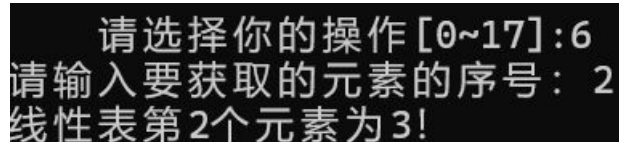
图 1-15 测试 2

1.4.6 GetElem 测试

线性表存在（输入线性表为 1 3 5，输入序号 2）

预测结果：线性表第 2 个元素为 3！

实际结果:



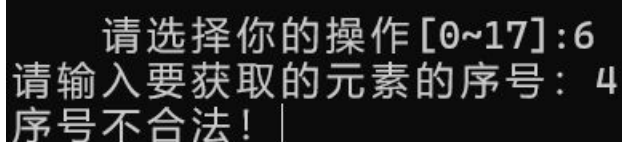
```
请选择你的操作[0~17]:6  
请输入要获取的元素的序号: 2  
线性表第2个元素为3!
```

图 1-16 测试 1

线性表存在（输入线性表为 1 3 5，输入序号 4）

预测结果：序号不合法！

实际结果:



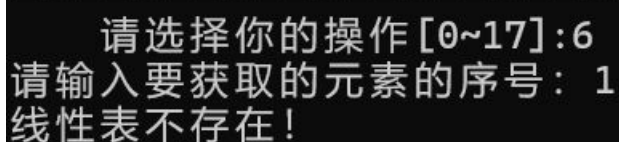
```
请选择你的操作[0~17]:6  
请输入要获取的元素的序号: 4  
序号不合法! |
```

图 1-17 测试 2

线性表不存在

预测结果：线性表不存在！

实际结果:



```
请选择你的操作[0~17]:6  
请输入要获取的元素的序号: 1  
线性表不存在!
```

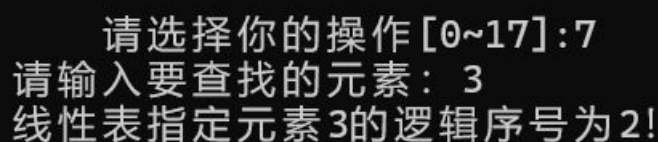
图 1-18 测试 3

1.4.7 LocateElem 测试

线性表存在（输入线性表为 1 3 5，输入元素 3）

预测结果：线性表指定元素 3 的逻辑序号为 2！

实际结果：



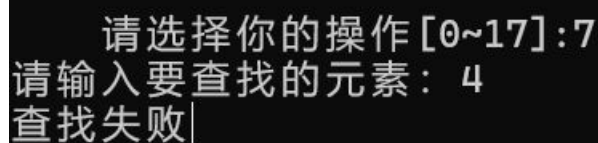
```
请选择你的操作[0~17]:7
请输入要查找的元素: 3
线性表指定元素3的逻辑序号为2!
```

图 1-19 测试 1

线性表存在（输入线性表为 1 3 5，输入元素 4）

预测结果：查找失败

实际结果：



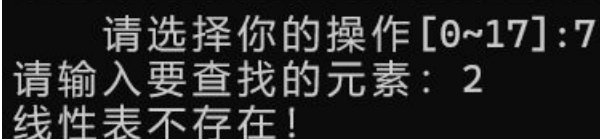
```
请选择你的操作[0~17]:7
请输入要查找的元素: 4
查找失败
```

图 1-20 测试 2

线性表不存在

预测结果：线性表不存在！

实际结果：



```
请选择你的操作[0~17]:7
请输入要查找的元素: 2
线性表不存在!
```

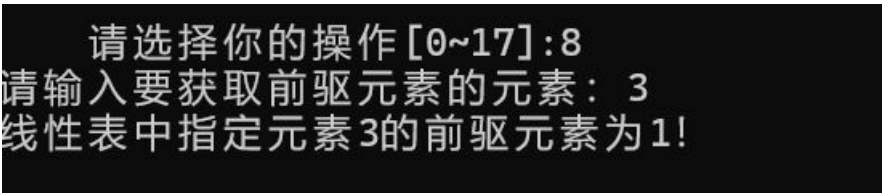
图 1-21 测试 3

1.4.8 PriorElem 测试

线性表存在（输入线性表为 1 3 5，输入元素 3）

预测结果：线性表中指定元素 3 的前驱元素为 1！

实际结果：



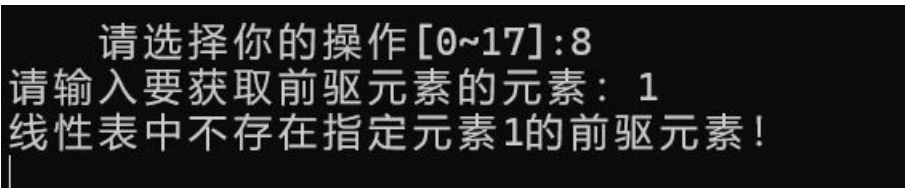
```
请选择你的操作[0~17]:8
请输入要获取前驱元素的元素: 3
线性表中指定元素3的前驱元素为1!
```

图 1-22 测试 1

线性表存在（输入线性表为 1 3 5，输入元素 1）

预测结果：线性表中不存在指定元素 1 的前驱元素！

实际结果：



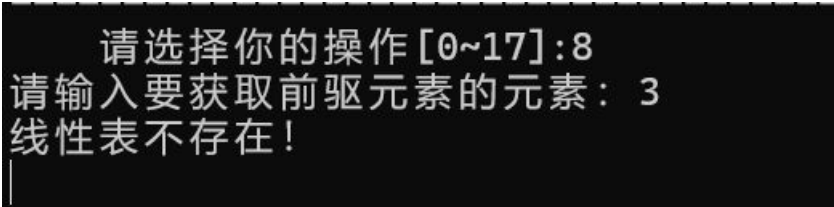
```
请选择你的操作[0~17]:8
请输入要获取前驱元素的元素: 1
线性表中不存在指定元素1的前驱元素!
```

图 1-23 测试 2

线性表不存在

预测结果：线性表不存在！

实际结果：



```
请选择你的操作[0~17]:8
请输入要获取前驱元素的元素: 3
线性表不存在!
```

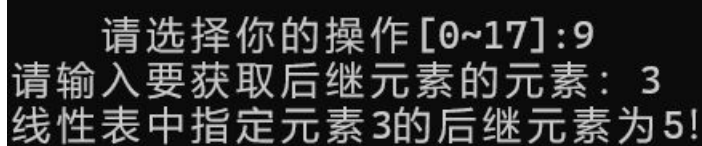
图 1-24 测试 3

1.4.9 NextElem 测试

线性表存在（输入线性表为 1 3 5，输入元素 3）

预测结果：线性表中指定元素 3 的后继元素为 5！

实际结果：



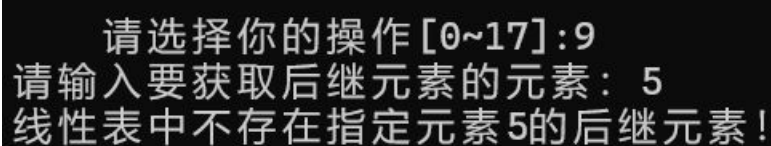
```
请选择你的操作[0~17]:9
请输入要获取后继元素的元素: 3
线性表中指定元素3的后继元素为5!
```

图 1-25 测试 1

线性表存在（输入线性表为 1 3 5，输入元素 1）

预测结果：线性表中不存在指定元素 5 的后继元素！

实际结果：



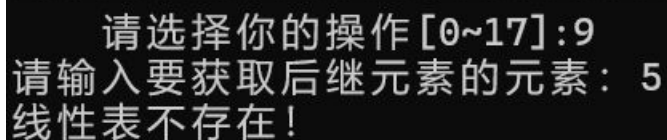
```
请选择你的操作[0~17]:9
请输入要获取后继元素的元素: 5
线性表中不存在指定元素5的后继元素!
```

图 1-26 测试 2

线性表不存在

预测结果：线性表不存在！

实际结果：



```
请选择你的操作[0~17]:9
请输入要获取后继元素的元素: 5
线性表不存在!
```

图 1-27 测试 3

1.4.10 ListInsert 测试

线性表存在且输入序号合法如序号 1 元素 1

预测结果：线性表元素插入成功！

新线性表为：1

实际结果：

```
请选择你的操作[0~17]:10
请输入要在线性表中插入元素的序号和要插入的元素: 1 1
线性表元素插入成功!
新线性表为: 1
```

图 1-28 测试 1

线性表存在但序号不合法如 0

预测结果: 线性表元素插入失败!

实际结果:

```
请选择你的操作[0~17]:10
请输入要在线性表中插入元素的序号和要插入的元素: 0 3
线性表元素插入失败!
```

图 1-29 测试 2

线性表不存在

预测结果: 线性表不存在!

实际结果:

```
请选择你的操作[0~17]:10
请输入要在线性表中插入元素的序号和要插入的元素: 1 1
线性表不存在!
```

图 1-30 测试 3

1.4.11 ListDelete 测试

线性表存在且输入序号合法如线性表 1 3 5 输入 3

预测结果: 线性表元素删除成功!

删除元素为: 5

新线性表为: 1 3

实际结果:

```
请选择你的操作[0~17]:11
请输入要在线性表中删除的元素序号: 3
线性表元素删除成功!
删除元素为: 5
新线性表为: 1 3
```

图 1-31 测试 1

线性表存在但序号不合法如 0

预测结果: 线性表元素删除失败!

实际结果:

```
请选择你的操作[0~17]:11
请输入要在线性表中删除的元素序号: 0
线性表元素删除失败!
```

图 1-32 测试 2

线性表不存在

预测结果: 线性表不存在!

实际结果:

```
请选择你的操作[0~17]:11
请输入要在线性表中删除的元素序号: 1
线性表不存在!
```

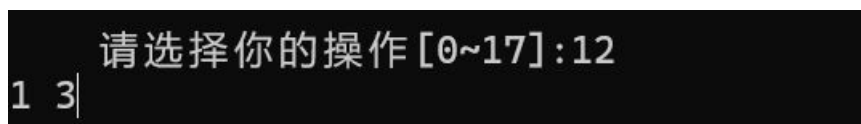
图 1-33 测试 3

1.4.12 ListTraverse 测试

线性表存在如线性表 1 3

预测结果: 1 3

实际结果:



请选择你的操作[0~17]:12
1 3|

图 1-34 测试 1

线性表不存在

预测结果：线性表不存在！

实际结果：



请选择你的操作[0~17]:12
线性表不存在！
|

图 1-35 测试 2

1.4.13 reverseList 测试

线性表存在如线性表 1 3

预测结果：反转后的线性表为：3 1

实际结果：



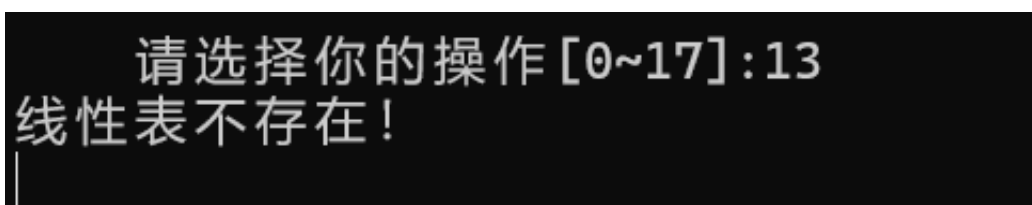
请选择你的操作[0~17]:13
反转后的线性表为：3 1
|

图 1-36 测试 1

线性表不存在

预测结果：线性表不存在！

实际结果：



请选择你的操作[0~17]:13
线性表不存在！
|

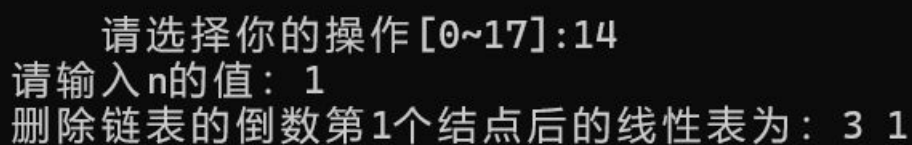
图 1-37 测试 2

1.4.14 RemoveNthFromEnd 测试

线性表存在如线性表 3 1 5 且序号合法输入 1

预测结果：删除链表的倒数第 1 个结点后的线性表为：3 1

实际结果：



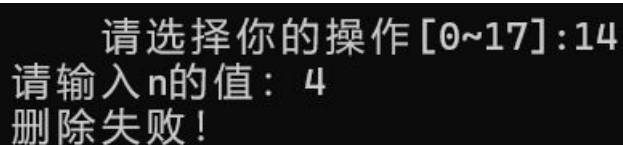
```
请选择你的操作[0~17]:14
请输入n的值: 1
删除链表的倒数第1个结点后的线性表为: 3 1
```

图 1-38 测试 1

序号不合法

预测结果：删除失败

实际结果：



```
请选择你的操作[0~17]:14
请输入n的值: 4
删除失败!
```

图 1-39 测试 2

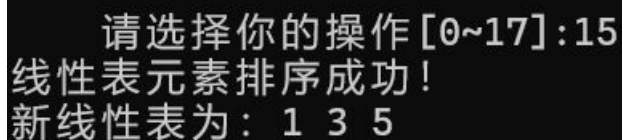
1.4.15 sortList 测试

线性表存在线性表 3 1 5

预测结果：线性表元素排序成功！

新线性表为：1 3 5

实际结果：



```
请选择你的操作[0~17]:15
线性表元素排序成功!
新线性表为: 1 3 5
```

图 1-40 测试 1

线性表不存在

预测结果：线性表不存在！

实际结果:

```
请选择你的操作[0~17]:15
线性表不存在!
```

图 1-41 测试 2

1.4.16 FILE 测试

线性表存在线性表 1 3 5

预测结果: 线性表判断存在! 对线性表进行写文件操作! txt 文件中存有 1 3

5 实际结果:

```
请选择你的操作[0~17]:16
线性表判断存在! 对线性表进行写文件操作!
```

图 1-42 测试 1



图 1-43 测试 1

线性表不存在

预测结果: 线性表不存在!

实际结果:

```
请选择你的操作[0~17]:16
线性表不存在! 对空表进行读文件操作! 读得线性表为:
1 3 5|
```

图 1-44 测试 2

1.4.17 Lists 测试

AddList 添加两个线性表 a 1 3 5 和 b 2 4 6

预测结果：添加后的多线性表为：a 1 3 5

b 2 4 6

实际结果：

```
请选择你的操作[0~17]:17
Management Of Multiple Linear Tables
-----
1. AddList          3. LocateList
2. RemoveList       0. Exit
-----

请选择你的操作[0~3]:1
请输入要添加的线性表的个数: 2
a 1 3 5 0
b 2 4 6 0
添加后的多线性表为: a 1 3 5
b 2 4 6
```

图 1-45 测试 1

RemoveList 删除 a 表

预测结果：删除后的线性表为：

b 2 4 6

```
Management Of Multiple Linear Tables
-----
1. AddList          3. LocateList
2. RemoveList       0. Exit
-----

请选择你的操作[0~3]:2
请输入要删除的线性表的名字: a
删除后的线性表为:
b 2 4 6
|
```

图 1-46 测试 2

LocateList 多线性表含 b 2 4 6 查找 b 表

预测结果: b 2 4 6

实际结果: b 2 4 6

是否对所查到的线性表进行单表操作? (Y or N)

此时输入 Y 可进入对查到的线性表的单表操作

```
Management Of Multiple Linear Tables
-----
1. AddList          3. LocateList
2. RemoveList       0. Exit
-----
请选择你的操作[0~3]:3
请输入要查找的线性表的名字: b
b 2 4 6
是否对所查到的线性表进行单表操作? (Y or N)
|
```

图 1-47 测试 3

LocateList 多线性表含 b 2 4 6 查找 a 表预测结果: 查找失败

实际结果:

```
Management Of Multiple Linear Tables
-----
1. AddList          3. LocateList
2. RemoveList       0. Exit
-----
请选择你的操作[0~3]:3
请输入要查找的线性表的名字: a
查找失败|
```

图 1-48 测试 4

1.5 实验小结

通过本实验,我进一步理解了线性表的链式存储结构的优点和缺点,链式存储的线性表可以根据实际需要动态地分配内存空间,使得数据结构更加灵活和动态,不像顺序存储需要预先确定大小。在插入和删除元素时具有较高的效率,

只需修改指针指向即可完成操作。链式存储结构相比于顺序存储结构存在一定的额外空间开销，因为需要为每个节点维护一个指针域，这会增加一定的存储空间消耗，但相对灵活性和效率来说是可以接受的。

通过不同函数的编写，我了解到在实现链式存储结构时，指针操作非常重要，正确的指针操作能够保证数据结构的正确性和稳定性，而错误的指针操作可能导致内存泄漏或指针丢失等问题。如何有条理地更改指针的指向是一大难点，比如在插入新结点时，必须先让新节点的指针域指向 `p` 指针所指元素的 `next`，然后才能让 `p` 指向新节点。稍有不慎就会造成错误。基础功能对我来说难度平平，附加功能的编写进一步磨练了我的编程能力。

通过演示系统菜单的构建，我了解了主函数和子函数的关系，以及如何搭建一个可以调用不同模块的系统，让我对系统整体设计有了更深的认识。

这个实验让我对数据结构有了更深入的了解和认识，同时也提高了我的编程能力。在未来的学习中，我将继续努力提升自己的编程能力，利用这些基础功能的实现和理解，充分发挥编程能力的作用。

2 基于邻接表的图实现

2.1 问题描述

实验要求：

通过实验达到：

1. 加深对图的概念、基本运算的理解；
2. 熟练掌握图的逻辑结构与物理结构的关系；
3. 以邻接表作为物理结构，熟练掌握图基本运算的实现。。

通过学习，我学到邻接表是图或网络数据结构的一种重要表示方法，它可以有效地表示网络拓扑结构，在图论和网络分析中有着重要的应用，为复杂的图网络数据的快速查询和统计提供了可靠而有效的解决方案。基于邻接表的图实现具有以下优势：

1. 数据存储的方便性：邻接表的表示方法能够以很方便的方式存储复杂的图数据，其中每个顶点会有非常多的相邻顶点，而邻接表可以有效地表示每个顶点的所有邻接点，使得数据的存储更加方便。
2. 操作的便捷性：在操作邻接表时，可以实现对复杂的图网络数据的快速查询、统计，这样使得在处理大规模图数据时，变得更加方便。
3. 图计算的效率：在使用邻接表来描述网络拓扑结构时，可以更加关注每个顶点之间的邻接关系，特别是在计算机图形学中，可以使用邻接表来快速计算图形网络的最佳路径等，极大地提升了图计算的效率。
4. 实现的简易性：邻接表的实现比较简单，只需要实现一组简单的操作语句，就可以完成对图的描述，而且邻接表只需要一个数组，节省了存储空间，使实现变得更加简单。

2.2 系统设计

2.2.1 头文件和预定义

头文件定义如下

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

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

预定义常量和类型表达式

```
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
#define MAX_VERTEX_NUM 20

typedef struct
{
    KeyType key;
    char others[20];
} VertexType;

typedef struct ArcNode
{
    int adjvex;
    struct ArcNode *nextarc;
} ArcNode;

typedef struct VNode
{
    VertexType data;
    ArcNode *firstarc;
} VNode, AdjList[MAX_VERTEX_NUM];

typedef struct
{
    AdjList vertices;
    int vexnum, arcnum;
    GraphKind kind;
```

```
} ALGraph;
```

2.2.2 演示系统菜单的结构

演示系统共有 17 个功能，其中 1-16 为单图的创建，销毁，查找顶点，顶点赋值，获得第一邻接点，获得下一邻接点，插入顶点，插入弧、删除弧，深度优先搜索遍历，广度优先搜索遍历共 12 个基本功能和距离小于 k 的顶点集合、顶点间最短路径和长度，图的连通分量，以文件形式保存共 4 个附加功能。

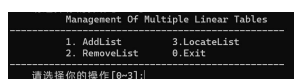


图 2-1 多图管理

功能 17 为多图操作，管理多个图的查找、添加、移除等功能，可通过功能 3 自由切换到管理的每个图，并可单独对某图进行单线性表的所有操作。

2.2.3 基本功能函数

1. 创建图：函数名称是 `CreateCraph(G,V,VR)`；初始条件是 V 是图的顶点集， VR 是图的关系集；操作结果是按 V 和 VR 的定义构造图 G ；
2. 销毁图：函数名称是 `DestroyGraph(G)`；初始条件图 G 已存在；操作结果是销毁图 G ；
3. 查找顶点：函数名称是 `LocateVex(G,u)`；初始条件是图 G 存在， u 是和 G 中顶点关键字类型相同的给定值；操作结果是若 u 在图 G 中存在，返回关键字为 u 的顶点位置序号（简称位序），否则返回其它表示“不存在”的信息；
4. 顶点赋值：函数名称是 `PutVex (G,u,value)`；初始条件是图 G 存在， u 是和 G 中顶点关键字类型相同的给定值；操作结果是对关键字为 u 的顶点赋值 $value$ ；
5. 获得第一邻接点：函数名称是 `FirstAdjVex(G, u)`；初始条件是图 G 存在， u 是 G 中顶点的位序；操作结果是返回 u 对应顶点的第一个邻接顶点位序，如果 u 的顶点没有邻接顶点，否则返回其它表示“不存在”的信息；
6. 获得下一邻接点：函数名称是 `NextAdjVex(G, v, w)`；初始条件是图 G 存在， v 和 w 是 G 中两个顶点的位序， v 对应 G 的一个顶点， w 对应 v 的邻接顶点；操作结果是返回 v 的（相对于 w ）下一个邻接顶点的位序，如果 w 是最后

一个邻接顶点，返回其它表示“不存在”的信息；

7. 插入顶点：函数名称是 `InsertVex(G,v)`；初始条件是图 G 存在， v 和 G 中的顶点具有相同特征；操作结果是在图 G 中增加新顶点 v 。（在这里也保持顶点关键字的唯一性）；
8. 删除顶点：函数名称是 `DeleteVex(G,v)`；初始条件是图 G 存在， v 是和 G 中顶点关键字类型相同的给定值；操作结果是在图 G 中删除关键字 v 对应的顶点以及相关的弧；
9. 插入弧：函数名称是 `InsertArc(G,v,w)`；初始条件是图 G 存在， v 、 w 是和 G 中顶点关键字类型相同的给定值；操作结果是在图 G 中增加弧 $\langle v,w \rangle$ ，如果图 G 是无向图，还需要增加 $\langle w,v \rangle$ ；
10. 删除弧：函数名称是 `DeleteArc(G,v,w)`；初始条件是图 G 存在， v 、 w 是和 G 中顶点关键字类型相同的给定值；操作结果是在图 G 中删除弧 $\langle v,w \rangle$ ，如果图 G 是无向图，还需要删除 $\langle w,v \rangle$ ；
11. 深度优先搜索遍历：函数名称是 `DFS_Traverse(G,visit())`；初始条件是图 G 存在；操作结果是图 G 进行深度优先搜索遍历，依次对图中的每一个顶点使用函数 `visit` 访问一次，且仅访问一次；
12. 广度优先搜索遍历：函数名称是 `BFS_Traverse(G,visit())`；初始条件是图 G 存在；操作结果是图 G 进行广度优先搜索遍历，依次对图中的每一个顶点使用函数 `visit` 访问一次，且仅访问一次；

2.2.4 附加功能函数

1. 距离小于 k 的顶点集合：函数名称是 `VerticesSetLessThanK(G,v,k)`，初始条件是图 G 存在；操作结果是返回与顶点 v 距离小于 k 的顶点集合
2. 顶点间最短路径和长度：函数名称是 `ShortestPathLength(G,v,w)`；初始条件是图 G 存在；操作结果是返回顶点 v 与顶点 w 的最短路径的长度；
3. 图的连通分量：函数名称是 `ConnectedComponentsNums(G)`，初始条件是图 G 存在；操作结果是返回图 G 的所有连通分量的个数；
4. 实现图的文件形式保存：其中，□ 需要设计文件数据记录格式，以高效保存图的数据逻辑结构 (D,R) 的完整信息；□ 需要设计图文件保存和加载操作合理模式。

(a) 文件写入：函数名称是 `SaveGraph(G,FileName)`；初始条件是线性表 G

已存在；操作结果是将 G 的元素写到名称为 FileName 的文件中。

(b) 文件读出：函数名称是 LoadGraph(G,FileName)；初始条件是线性表 G 不存在；操作结果是将文件 FileName 中的元素读到表 G 中。

5. 实现多个线性表管理：设计相应的数据结构管理多个线性表的查找、添加、移除等功能。

(a) 增加图：函数名称是 Addg(G, ListName)；初始条件是名称为 ListName 的图不存在于图集合中；操作结果是在 Lists 中创建一个名称为 ListName 的图并初始化。

(b) 移除图：函数名称是 Removeg(G, ListName)；初始条件是名称为 ListName 的图存在于图集合中；操作结果是将该图移除。

(c) 查找图：函数名称是 Locateg(Lists, ListName)；初始条件是名称为 ListName 的图存在于图集合中；操作结果是返回该图在 Lists 中的逻辑索引。

2.3 系统实现

op 初始化为 1，整个菜单通过变量 op 获取所选择的功能选项，1 至 17 分别为不同的功能。通过 switch 语句执行不同的功能，每次执行完该功能后通过 break 跳出 switch 语句，继续执行 while 循环，重新读取 op。通过外层的 while 循环实现多次选择并执行功能，直到输入 0，循环结束，退出演示系统。对于多表操作，与单表操作类似，输入 0 后退出多表操作，回到单表操作系统。

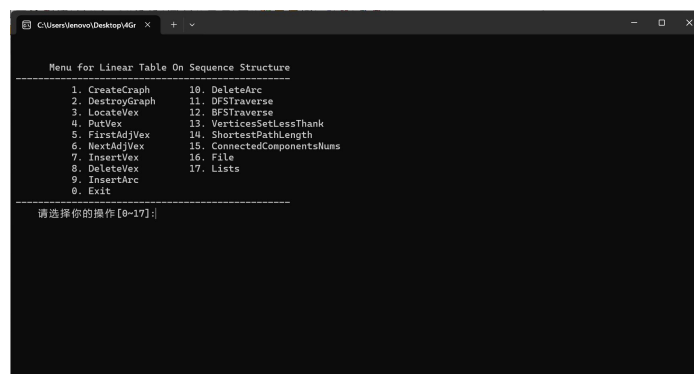


图 2-2 演示系统

2.3.1 创建图

输入: 图 G, 顶点集 V[], 边集 VR[]

输出: 函数执行状态

算法思想: 如若当前顶点序列的关键字不为-1 执行 while 循环: 如果当前关键字未出现则更新标记数组并继续, 否则返回 ERROR, 在邻接表添加新顶点, 令表头结点为 NULL, 更新顶点数, 检查是否超过最大数目 MAXVERTEXNUM, 超过则返回 ERROR。循环结束如果 vexnum=0, 即没有顶点, 则返回 ERROR, 否则令 G.vexnum=vexnum。当当前关系序列不为 (-1,-1) 时执行 while 循环。用 checksame 函数检查关键字是否重复

```
status CreateCraph(ALGraph &G, VertexType V[], KeyType VR[][2]) {
    int i = 0;
    while (V[i].key != -1) {
        i++;
    }
    if(i==0) return ERROR;
    if(i-1>=MAX_VERTEX_NUM) return ERROR;
    if(checksame(V)==ERROR) return ERROR;
    G.vexnum = i;
    for (i = 0; i < G.vexnum; i++) {
        G.vertices[i].data = V[i];
        G.vertices[i].firstarc = NULL;
    }
    i = 0;
    while (VR[i][0] != -1)
    {
        int a=locatei(G,VR[i][0]),b=locatei(G,VR[i][1]);
        if(!(a>=0&&b>=0)) return ERROR;
        ArcNode *q=(ArcNode*)malloc(sizeof(ArcNode)) ;
        q->adjvex=b;
        q->nextarc=G.vertices[a].firstarc;
    }
}
```

```
G.vertices[a].firstarc=q;
q=(ArcNode*)malloc(sizeof(ArcNode)) ;
q->adjvex=a;
q->nextarc=G.vertices[b].firstarc;
G.vertices[b].firstarc=q;
i++;
}
G.arcnum = i;
return OK;    }
```

2.3.2 销毁图

输入: 图 G

输出: 函数执行状态

算法思想: 记录首节点与下一结点, 遍历完一个顶点的所有邻接点, 循环删除

```
status DestroyGraph(ALGraph &G)
{
    ArcNode *p=NULL,*q=NULL;
    for(int k=0;k<G.vexnum;k++)
    {
        if(G.vertices[k].firstarc!=NULL)
        {
            p=G.vertices[k].firstarc;
            while(p!=NULL)
            {
                q=p->nextarc;
                free(p);
                p=q;
            }
            G.vertices[k].firstarc=NULL;
        }
    }
}
```

```
    }  
    }  
    G.arcnum=0;  
    G.vexnum=0;  
    return OK;  
}
```

2.3.3 查找顶点

输入: 图 G, 要查找顶点的关键字

输出: 顶点的位序

算法思想: 用 for 循环遍历邻接表, 如果当前顶点关键字与所找关键字相等, 则返回当前顶点序号。否则返回-1

```
int LocateVex(ALGraph G,KeyType u){  
    int i=0;  
    for(i;i<G.vexnum;i++){  
  
        if(G.vertices[i].data.key==u) return i;  
    }  
    return -1;  
}
```

2.3.4 顶点赋值

输入: 图 G, 要修改顶点的关键字, 修改成的值

输出: 函数执行状态

算法思想: 定位需要赋值的顶点的位置, 将该顶点的关键字和值域重新赋值。

```
status PutVex(ALGraph &G,KeyType u,VertexType value){  
    int i=LocateVex(G,u);  
    if(i==-1) return ERROR;  
    for(int k=0;k<G.vexnum;k++){  
        if(G.vertices[k].data.key==value.key)
```

```
        return ERROR;
    }
    G.vertices[i].data=value;
    return OK;
}
```

2.3.5 获得第一邻接点

输入: 图 G, 要查找第一邻接点的顶点的关键字

输出: 邻接点位序

算法思想: 定位需要查找第一邻接点的顶点的位置, 如果此顶点有第一邻接点则返回其邻接点, 否则返回-1

```
int FirstAdjVex(ALGraph G,KeyType u){
    int i=0;
    for(i;i<G.vexnum;i++)
    {
        if(G.vertices[i].data.key==u)
            return G.vertices[i].firstarc->adjvex;
    }
    return -1;
}
```

2.3.6 获得下一邻接点

输入: 图 G, 要查找下一邻接点的顶点的关键字

输出: 邻接点位序

算法思想: 定位需要查找下一邻接点的顶点的位置, 用 while 循环遍历该顶点的所有邻接点, 找到另一个顶点, 用 p 指向该邻接点。如果 p == NULL || p->nextarc == NULL, 即 v 与 w 不相邻, 或 v 相对 w 无下一邻接点, 返回-1。否则返回 return p->nextarc->adjvex。

```
int NextAdjVex(ALGraph G,KeyType v,KeyType w){
    int i=LocateVex(G,v),j=LocateVex(G,w);
```

```
while (i != -1 && j != -1)
{
    ArcNode *p = G.vertices[i].firstarc;
    while (p->adjvex != j && p)
    {
        p = p->nextarc;
    }
    if (p->nextarc)
    {
        return p->nextarc->adjvex;
    }
    else return -1;
}
return -1;
}
```

2.3.7 插入顶点

输入: 图 G, 要插入的顶点

输出: 函数执行状态

算法思想: 定位需要插入的顶点, 如果要插入顶点的关键字已出现, 则返回 ERROR。如果 $G.vexnum == MAXVERTEXNUM$, 则返回 ERROR。否则插入新顶点, 更新顶点数, 返回 OK。

```
status InsertVex(ALGraph &G, VertexType v){
    if (LocateVex(G, v.key) != -1) return ERROR; \
    if (G.vexnum == MAX_VERTEX_NUM) return ERROR;
    G.vertices[G.vexnum].data = v;
    G.vertices[G.vexnum].firstarc = NULL;
    G.vexnum++;
    return OK;
}
```

2.3.8 删除顶点

输入: 图 G, 要删除的顶点

输出: 函数执行状态

算法思想: 定位需要删除的顶点, 如果需要删除的顶点不存在, 则返回 ERROR。否则循环删除所有与此顶点相连的其他顶点中的弧

```
status DeleteVex(ALGraph &G,KeyType v)
{
    if(G.vexnum==1 || G.vexnum==0) return ERROR;
    int i=LocateVex(G,v);
    if(i==-1) return ERROR;
    ArcNode *p=G.vertices[i].firstarc,
    *q=NULL,*temp=NULL;
    while(p){
        int j=p->adjvex;
        q=G.vertices[j].firstarc;
        if(q->adjvex==i){
            temp=q;
            G.vertices[j].firstarc=q->nextarc;
            free0(temp);
        }
        else {
            while(q->nextarc->adjvex!=i)
                q=q->nextarc;
            temp=q->nextarc;
            q->nextarc=temp->nextarc;
            free0(temp);
        }
        temp=p->nextarc;
        free0(p);
        p=temp;
    }
}
```



```

        G. arcnum--;
    }
    for (int k=i; k<G. vexnum; k++)
        G. vertices[k]=G. vertices[k+1];
    G. vexnum--;
    for (int k=0; k<G. vexnum; k++){
        p=G. vertices[k]. firstarc;
        while (p!=NULL)
        {
            if (p->adjvex>i)
                p->adjvex--;
            p=p->nextarc;
        }
    }
    return OK;    }
    
```

2.3.9 插入弧

输入: 图 G, 要插入弧的两个端点的关键字

输出: 函数执行状态

算法思想: 如流程图所示

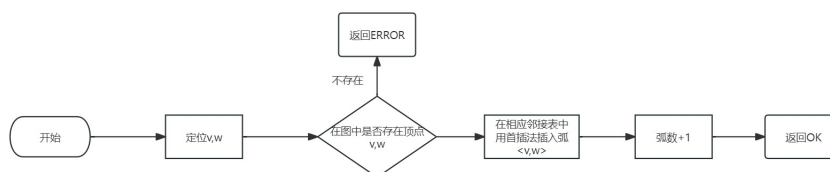


图 2-3 删除链表的倒数第 n 个结点

```

status InsertArc (ALGraph &G,KeyType v,KeyType w)
{
    int a=LocateVex (G,v) ,b=LocateVex (G,w);
    if (a== -1||b== -1) return ERROR;
    ArcNode *p=G. vertices[a]. firstarc;
    
```

```
    while (p!=NULL){
        if (p->adjvex==b) return ERROR;
        p=p->nextarc;
    }
    p=(ArcNode*)malloc(sizeof(ArcNode));
    p->adjvex=b;
    p->nextarc=G.vertices[a].firstarc;
    G.vertices[a].firstarc=p;
    p=(ArcNode*)malloc(sizeof(ArcNode));
    p->adjvex=a;
    p->nextarc=G.vertices[b].firstarc;
    G.vertices[b].firstarc=p;
    G.arcnum++;
    return OK;
}
```

2.3.10 删除弧

输入: 图 G, 要删除弧的两个顶点的关键字

输出: 函数执行状态

算法思想: 定位顶点, 若顶点不存在则返回 ERROR, 否则分别在这两个关键字后的链表中寻找并删除对应结点的弧, 返回 OK。

```
status DeleteArc(ALGraph& G, KeyType v, KeyType w)
{
    int s1 = Get_index(G, v);
    int s2 = Get_index(G, w);
    if (s1 == -1 || s2 == -1) return ERROR;
    ArcNode* q = G.vertices[s1].firstarc;
    int flag = 0;
    while (q) {
        if (q->adjvex == s2) flag = 1;
```

```
        q = q->nextarc;
    }
    if (flag == 0) return ERROR;
    DeleteArc_A_Node(G.vertices[s1], s2);
    DeleteArc_A_Node(G.vertices[s2], s1);
    --G.arcnum;
    return OK;
}
```

2.3.11 深度优先搜索遍历

输入: 图 G

输出: 函数执行状态

算法思想: 使用了一个 flag 数组来标记顶点是否已被访问过, 通过循环遍历所有顶点, 并对未被访问过的顶点进行深度优先搜索, 实现了图的 DFS 遍历

```
status DFSTraverse(ALGraph &G, void (*visit)(VertexType)){
    int flag[MAX_VERTEX_NUM]={0};
    for(int i=0; i<G.vexnum; i++) flag[i]=0;
    for(int i=0; i<G.vexnum; i++){
        if(!flag[i]){
            visit(G.vertices[i].data);
            flag[i]++;
            ArcNode *p=G.vertices[i].firstarc;
            while(p){
                int k=LocateVex
                (G, G.vertices[p->adjvex].data.key);
                if(k==-1) break;
                else if(flag[k]!=0){
                    if(p->nextarc==NULL)
                        break;
                    p=p->nextarc;
                }
            }
        }
    }
}
```

```
        }
        else {
            i=k;
            visit(G.vertices[i].data);
            flag[i]++;
            p=G.vertices[i].firstarc;
        }
    }
}
else continue;
}
return OK;
}
```

2.3.12 广度优先搜索遍历

输入: 图 G

输出: 函数执行状态

算法思想: 使用了一个 flag 数组来标记顶点是否已被访问过, 通过循环遍历所有顶点, 并对未被访问过的顶点进行广度优先搜索, 实现了图的 BFS 遍历

```
status BFSTraverse(ALGraph &G, void (*visit)(VertexType)) {
    int flag[MAX_VERTEX_NUM];
    int i;
    for (i = 0; i < G.vexnum; i++) {
        flag[i] = 0; // 初始化所有顶点为未访问状态
    }
    for (i = 0; i < G.vexnum; i++) {
        if (!flag[i]) {
            flag[i] = 1;
            visit(G.vertices[i].data); // 访问顶点
            ArcNode *p=G.vertices[i].firstarc;
```

```
        while(p){
            int k=LocateVex
            (G,G.vertices[p->adjvex].data.key);
            if(k==-1) break;
            else if(flag[k]!=0){
                if(p->nextarc==NULL) break;
                p=p->nextarc;
            }
            else {
                visit(G.vertices[k].data);
                flag[k]++;
                p=p->nextarc;
            }
        }
    }
}
return OK; }
```

2.3.13 距离小于 k 的顶点集合

输入: 图 G, 顶点 v, 距离 k

输出: 函数执行状态

算法思想: 遍历图 G 中的每一个顶点, 计算顶点 v 到当前顶点的最短路径长度。如果顶点 v 到当前顶点的最短路径长度小于 k, 并且当前顶点不等于顶点 v 本身, 则对当前顶点进行访问操作。继续遍历直到遍历完所有顶点。

```
void VerticesSetLessThank(ALGraph G,KeyType v,KeyType k){
    int i,j;
    for(i=0;i<G.vexnum;i++)
    {
        j=ShortestPathLength(G,v,G.vertices[i].data.key);
        if(j<k&&G.vertices[i].data.key!=v)
```

```
        visit(G.vertices[i].data);
    }
}
```

2.3.14 顶点间最短路径和长度

输入: 图 G, 两个顶点

输出: 函数执行状态

算法思想: 利用广度优先搜索的思想, 通过逐层遍历顶点的邻接顶点, 并更新最短路径长度, 最终找到从顶点 v 到顶点 w 的最短路径长度

```
status ShortestPathLength(ALGraph G,KeyType v,KeyType w){
    int i,j,n;
    VertexType top;
    top.key=v;
    Linkqueue Q;
    InitQueue(Q);
    for(i=0;i<G.vexnum;i++)
        distance[G.vertices[i].data.key]=20;
    distance[v]=0;
    int k=LocateVex(G,v);
    enqueue(Q,G.vertices[k].data);
    while(!QueueEmpty(Q))
    {
        dequeue(Q,top);
        if(top.key==w)break;
        for(j=FirstAdjVex(G,top.key);j>=0;
            j=NextAdjVex(G,top.key,G.vertices[j].data.key))
        {
            if(distance[G.vertices[j].data.key]==20)
            {
                distance[G.vertices[j].data.key]=

```

```
        distance[top.key]+1;
        enqueue(Q,G.vertices[j].data);
    }
}
}
n=distance[w];
return n; }
```

2.3.15 图的连通分量

输入: 图 G

输出: 函数执行状态

算法思想: 通过深度优先搜索遍历图中的各个连通分量, 并对每个未被访问过的顶点进行 DFS 搜索, 以此统计图中的连通分量个数

```
status ConnectedComponentsNums(ALGraph G)
{
    int count=0,i;
    for(i=0;i<G.vexnum;i++)
        mark[G.vertices[i].data.key]=FALSE;
    for(i=0;i<G.vexnum;i++)
    {
        if(!mark[G.vertices[i].data.key])
        {
            dfs(G,G.vertices[i].data.key);
            count++;
        }
    }
    return count;
}
```

2.3.16 写文件

输入: 图 G

输出: 函数执行状态

算法思想: 打开文件, 遍历图将所有顶点的数据域写入文件, 关闭文件, 返回 OK

```
status SaveGraph(ALGraph G, char FileName[]) {
    FILE *fp;
    fp = fopen(FileName, "w");
    if (fp == NULL || G.vexnum==0) return ERROR;
    for (int i=0; i<G.vexnum; i++){
        fprintf(fp, "%d%s", G.vertices[i].data.key,
            G.vertices[i].data.others);
        ArcNode *p=G.vertices[i].firstarc;
        while(p){
            fprintf(fp, "%d", p->adjvex);
            p=p->nextarc;
        }
        fprintf(fp, "%d\n", -1);
    }
    fclose(fp);
    return OK;
}
```

2.3.17 读文件

输入: 图 G

输出: 函数执行状态

算法思想: 从文件中读取数据, 构建图的邻接表结构。逐行读取顶点和邻接顶点信息, 然后将它们添加到图中。最终形成一个表示图的邻接关系的数据结构。

```
status SaveGraph(ALGraph G, char FileName[]) {
```



```
FILE *fp;
fp = fopen(FileName, "w");
if (fp == NULL || G.vexnum==0) return ERROR;
for(int i=0; i<G.vexnum; i++){
    fprintf(fp, "%d%s", G.vertices[i].data.key,
    G.vertices[i].data.others);
    ArcNode *p=G.vertices[i].firstarc;
    while(p){
        fprintf(fp, "%d", p->adjvex);
        p=p->nextarc;
    }
    fprintf(fp, "%d\n", -1);
}
fclose(fp);
return OK;
}
```

2.3.18 多个图的添加

输入: 多图表 Lists

输出: 函数执行状态

算法思想: 将读入的图名复制给要添加的图, 多图表长度加 1, 返回 OK

```
status Addg(LISTS &Lists, char ListName[])
{

    strcpy(Lists.elem[Lists.length].name, ListName);
    Lists.length++;
    return OK;
}
```

2.3.19 多个线性表的移除

输入: 多图表 Lists

输出: 函数执行状态

算法思想: 在 Lists 中查找名称为 ListName 的图, 如果可以找到, 将此图销毁, 将其之后的表依次向前移动并将多表的长度减一, 返回 OK, 否则返回 ERROR

```
status Removeg(LISTS &Lists, char ListName[])
{
    for (int i=0; i<Lists.length; i++){
        if (strcmp(ListName, Lists.elem[i].name)==0){
            for (int j=i; j<Lists.length-1; j++){
                strcpy(Lists.elem[j].name,
                    Lists.elem[j+1].name);
                Lists.elem[j].G = Lists.elem[j+1].G;
            }
            Lists.length--;
            return OK;
        }
    }
    return ERROR;
}
```

2.3.20 多个线性表的查找

输入: 多图表 Lists

输出: 函数执行状态

算法思想: 在 Lists 中查找名称为 ListName 的图, 如果可以找到, 返回逻辑索引, 否则返回 0。

```
int Locateg(LISTS Lists, char ListName[])
{
    for (int i=0; i<Lists.length; i++){
        if (strcmp(ListName, Lists.elem[i].name)==0)
```

```
        return i+1;
    }
    return 0;
}
```

2.3.21 对多线性表的单表操作

输入:Lists.elem[n].

输出: 无

算法思想: 将主函数中 case1-16 整合成一个函数

2.4 系统测试

2.4.1 CreateCraph 测试

输入: 5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1

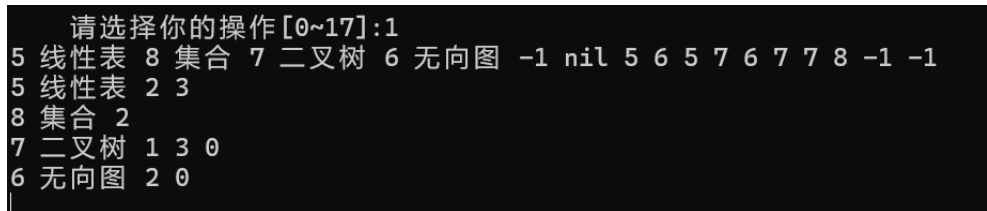
预测结果: 5 线性表 2 3

8 集合 2

7 二叉树 1 3 0

6 无向图 2 0

实际结果:



```
请选择你的操作[0~17]:1
5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1
5 线性表 2 3
8 集合 2
7 二叉树 1 3 0
6 无向图 2 0
```

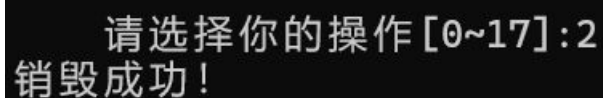
图 2-4 测试

2.4.2 DestroyGraph 测试

输入: 5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1

预测结果: 销毁成功!

实际结果:



```
请选择你的操作[0~17]:2
销毁成功!
```

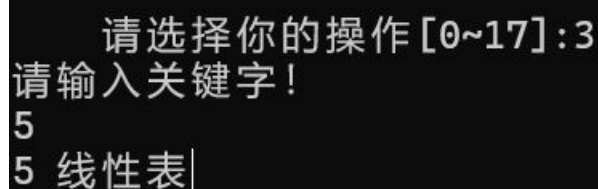
图 2-5 测试

2.4.3 LocateVex 测试

输入：(5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1) 5

预测结果：5 线性表

实际结果：



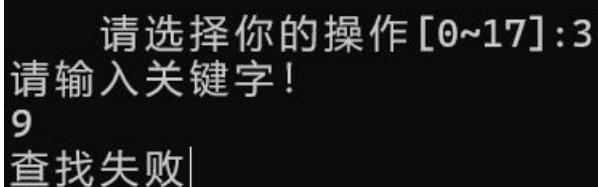
```
请选择你的操作[0~17]:3
请输入关键字!
5
5 线性表|
```

图 2-6 测试 1

输入：(5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1) 9

预测结果：查找失败

实际结果：



```
请选择你的操作[0~17]:3
请输入关键字!
9
查找失败|
```

图 2-7 测试 2

2.4.4 PutVex 测试

输入：(5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1) 8 9 小集合

预测结果：5 线性表 9 小集合 7 二叉树 6 无向图

实际结果：

```
请选择你的操作[0~17]:4
请输入关键字和赋值结点!
8 9 小集合
5 线性表 9 小集合 7 二叉树 6 无向图|
```

图 2-8 测试 1

输入：(5 线性表 9 小集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1) 0 10
集合

预测结果：赋值操作失败

实际结果：

```
请选择你的操作[0~17]:4
请输入关键字和赋值结点!
0 10 集合
赋值操作失败
```

图 2-9 测试 2

2.4.5 FirstAdjVex 测试

输入：(5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1) 5

预测结果：7 二叉树

实际结果：

```
请选择你的操作[0~17]:
5
请输入关键字!
5
7 二叉树|
```

图 2-10 测试

2.4.6 FirstAdjVex 测试

输入：(5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1) 5

预测结果：7 二叉树

实际结果:

```
      请选择你的操作[0~17]:  
5  
请输入关键字!  
5  
7 二叉树|
```

图 2-11 测试

2.4.7 NextAdjVex 测试

输入: (5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1) 5 6

预测结果: 查找失败!

实际结果:

```
      请选择你的操作[0~17]:6  
请输入要查找顶点和相对顶点的关键字!  
5 6  
查找失败!
```

图 2-12 测试 1

输入: (5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1) 7 8

预测结果: 7 二叉树

实际结果:

```
      请选择你的操作[0~17]:6  
请输入要查找顶点和相对顶点的关键字!  
7 8  
顶点7的邻接顶点相对于8的下一邻接顶点的位序是3  
其值为 6 无向图  
|
```

图 2-13 测试 2

2.4.8 InsertVex 测试

输入: (5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1) 9 新增

预测结果：5 线性表 2 3

8 集合 2

7 二叉树 1 3 0

6 无向图 2 0

9 新增

实际结果：

```
      请选择你的操作[0~17]:7
请输入新增的顶点!
9 新增
5 线性表 2 3
8 集合 2
7 二叉树 1 3 0
6 无向图 2 0
9 新增
```

图 2-14 测试 1

输入：（5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1 ） 8 新增

预测结果：插入操作失败

实际结果：

```
      请选择你的操作[0~17]:7
请输入新增的顶点!
8 新增
插入操作失败|
```

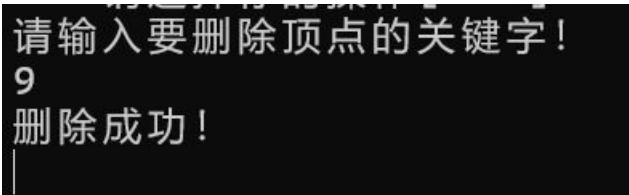
图 2-15 测试 2

2.4.9 DeleteVex 测试

输入：9

预测结果：删除成功

实际结果：



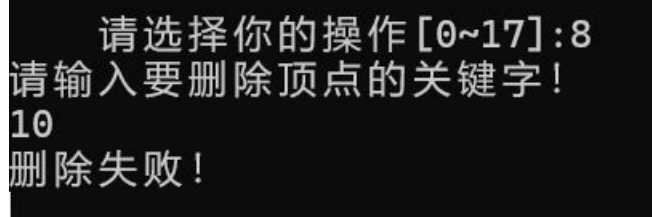
```
请输入要删除顶点的关键字!  
9  
删除成功!
```

图 2-16 测试 1

输入: 10

预测结果: 删除失败

实际结果:



```
请选择你的操作[0~17]:8  
请输入要删除顶点的关键字!  
10  
删除失败!
```

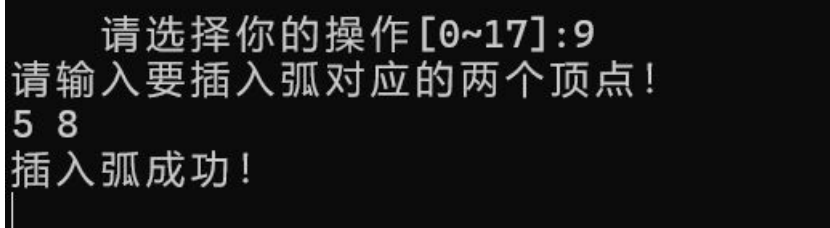
图 2-17 测试 2

2.4.10 InsertArc 测试

输入: 5 8

预测结果: 插入弧成功!

实际结果:



```
请选择你的操作[0~17]:9  
请输入要插入弧对应的两个顶点!  
5 8  
插入弧成功!
```

图 2-18 测试 1

输入: 5 6

预测结果: 插入弧失败!

实际结果:


```
请选择你的操作[0~17]:9
请输入要插入弧对应的两个顶点!
5 6
插入弧失败!
```

图 2-19 测试 2

2.4.11 DeleteArc 测试

输入: (5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1) 5 6

预测结果: 删除弧成功!

实际结果:

```
请选择你的操作[0~17]:10
请输入要删除弧对应的两个顶点!
5 6
删除弧成功!
```

图 2-20 测试 1

输入: 5 8

预测结果: 删除弧失败!

实际结果:

```
请选择你的操作[0~17]:10
请输入要删除弧对应的两个顶点!
5 8
删除弧失败!
```

图 2-21 测试 2

2.4.12 DFSTraverse 测试

预测结果: 5 线性表 7 二叉树 8 集合 6 无向图

实际结果:

```
请选择你的操作[0~17]:11
5 线性表 7 二叉树 8 集合 6 无向图|
```

图 2-22 测试

2.4.13 BFSTraverse 测试

预测结果：5 线性表 7 二叉树 6 无向图 8 集合

实际结果：

```
请选择你的操作[0~17]:12
5 线性表 7 二叉树 6 无向图 8 集合|
```

图 2-23 测试

2.4.14 VerticesSetLessThanK 测试

输入：5 2

预测结果：与顶点距离小于 2 的顶点有

7 二叉树 6 无向图

实际结果：

```
请选择你的操作[0~17]:13
请输入顶点的关键字及查找距离！
5 2
与顶点距离小于2的顶点有
7 二叉树 6 无向图|
```

图 2-24 测试

2.4.15 ShortestPathLength 测试

输入：5 8

预测结果：这两个顶点间的最短路径是 2

实际结果：

```
请选择你的操作[0~17]:14
请输入两个顶点的关键字!
5 8
这两个顶点间的最短路径是2|
```

图 2-25 测试

2.4.16 ConnectedComponentsNums 测试

预测结果：图的连通分量有 1 个！

实际结果：

```
请选择你的操作[0~17]:15
图的连通分量有1个！
```

图 2-26 测试

2.4.17 FILE 测试

预测结果：已将图存储到文件中从文件中读得 G1 为：

5 线性表 2 3

8 集合 2

7 二叉树 1 3 0

6 无向图 2 0

实际结果：

```
请选择你的操作[0~17]:16
已将图存储到文件中从文件中读得G1为：
5 线性表 2 3
8 集合 2
7 二叉树 1 3 0
6 无向图 2 0
|
```

图 2-27 测试

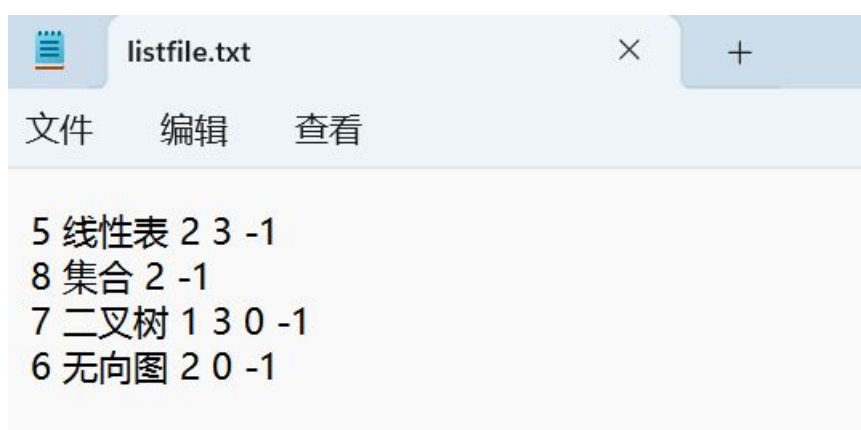


图 2-28 测试

2.4.18 LISTS 测试

Addg 添加两个图 a 5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1
-1 和 b 5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1

预测结果：添加后的多图表为：a 5 线性表 2 3

8 集合 2

7 二叉树 1 3 0

6 无向图 2 0

b 5 线性表 2 3

8 集合 2

7 二叉树 1 3 0

6 无向图 2 0

实际结果：

```

Management Of Multiple Linear Tables
-----
1. AddList          3. LocateList
2. RemoveList       0. Exit
-----

请选择你的操作[0~3]:1
请输入要添加的图的个数: 2
a 5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1
b 5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1
添加后的多图表为: a 5 线性表 2 3
8 集合 2
7 二叉树 1 3 0
6 无向图 2 0
b 5 线性表 2 3
8 集合 2
7 二叉树 1 3 0
6 无向图 2 0

```

图 2-29 测试 1

Removeg 删除 b 图

预测结果: 添加后的多图表为: a 5 线性表 2 3

8 集合 2

7 二叉树 1 3 0

6 无向图 2 0

实际结果:

```

Management Of Multiple Linear Tables
-----
1. AddList          3. LocateList
2. RemoveList       0. Exit
-----

请选择你的操作[0~3]:2
请输入要删除的线性表的名字: b
删除后的线性表为:
a 5 线性表 2 3
8 集合 2
7 二叉树 1 3 0
6 无向图 2 0

```

图 2-30 测试 2

locateg 定位 a 图

预测结果：a 5 线性表 2 3

8 集合 2

7 二叉树 1 3 0

6 无向图 2 0

是否对所查到的线性表进行单表操作？(Y or N)

此时输入 Y 可进入对查到的线性表的单表操作

实际结果：

```
Management Of Multiple Linear Tables
-----
1. AddList          3. LocateList
2. RemoveList       0. Exit
-----
    请选择你的操作[0~3]:3
    请输入要查找的图的名字: a
a 5 线性表 2 3
8 集合 2
7 二叉树 1 3 0
6 无向图 2 0

    是否对所查到的线性表进行单表操作？ (Y or N)
|
```

图 2-31 测试 3

2.5 实验小结

通过基于邻接表的图实现实验，我首先了解到了邻接表这种图的表示方法。邻接表是一种通过链表来表示图中顶点之间关系的数据结构，每个顶点都有一个对应的链表，链表中存储了与该顶点相邻的其他顶点。这种表示方法在处理稀疏图时具有优势，可以更有效地存储和访问图的信息。

通过实现基于邻接表的图，我学会了如何构建图的数据结构并实现相关操作。逐个读取顶点和邻接顶点的信息，并将它们添加到对应的链表中，从而构建起图的邻接关系，让我们能够更直观地理解图的结构和顶点之间的连接关系。

在实验中，我还学会了如何遍历图的顶点和边，以及如何实现图的深度优先

搜索和广度优先搜索算法。这些算法对于解决图相关问题非常重要，而通过实际实现它们，我更加深入地理解了它们的工作原理和应用场景。

总的来说，通过实验基于邻接表的图实现，我不仅加深了对图数据结构的理解，还提升了编程能力和算法思维，提升了局部和统一考虑问题的思维方式。这些收获对在未来解决复杂问题和设计高效算法打下了坚实基础。

3 课程的收获和建议

通过数据结构实验课，我学到了怎么样从一个实际问题出发，建立模型，找到相应的存储结构和实现方法，实际运行，反复调试和修改，最终实现功能，我提升了局部和统一考虑问题的思维方式，我对基于顺序存储结构的线性表实现、基于链式存储结构的线性表实现、基于二叉链表的二叉树实现、基于邻接表的图实现这四种存储方式有了更深刻的理解，实验课的实践过程令我受益匪浅

3.1 基于顺序存储结构的线性表实现

通过这一专题，我了解到顺序存储结构的线性表是一种通过数组来表示和存储数据元素的数据结构。我们了解了顺序存储结构线性表的特点，其中包括线性表元素在内存中的连续存储、通过下标访问元素的高效性等。

通过实验，我学会了如何在实际编程中实现顺序存储结构的线性表，并存储线性表的元素，实现插入、删除、查找等操作。在实验过程中，我们也了解了线性表的一些基本算法，比如排序，求最大连续和的子数组等，以及如何在顺序存储结构上实现这些算法。

实验后，我加深了对顺序存储结构线性表的理解，包括其特点和操作方法。通过实际编程实现线性表的操作，我提升了对数据结构和算法的理解和应用能力。我还学会了如何分析和设计算法，以及如何在实际问题中应用数据结构来解决问题。

3.2 基于链式存储结构的线性表实现

通过这一专题，我学习了基于链式存储结构的线性表的实现方法。链式存储结构是一种通过节点之间的指针来连接数据元素的数据结构，每个节点中存储数据元素和指向下一个节点的指针。这种存储结构的优点是可以动态地分配内存空间，灵活地插入和删除元素，适用于不确定大小的线性表。

通过实验，我了解了如何在实际编程中实现基于链式存储结构的线性表，并进行相关操作。我们可以通过定义节点结构体来表示线性表的节点，通过指针来连接各个节点，实现插入、删除、查找等操作。在实验过程中，我们也学会了如何遍历链表、反转链表等基本操作。

通过实验，我收获了很多。我加深了对链式存储结构线性表的理解，包括节点之间的链接关系和操作方法。我对链表的各种操作变得更加熟练，我深刻理解到了顺序存储结构和链式存储结构的区别，并学会在不同的场景之下进行选择并且灵活运用他们。

3.3 基于二叉链表的二叉树实现

基于二叉链表的二叉树实现专题是我认为的四个专题中最难的一个，为此我改变了我的演示系统的结构，由原本的 case1 到 16 是单表，case17 是多表，再由多表中的 locate 函数进入单表，改变为函数从一开始就进入多表，可以选择进入单表。此专题的任务很多，由于我一开始对二叉树并不熟练，所以花了很长时间才完成这一部分。

通过学习这一部分，我了解到基于二叉链表的二叉树是一种通过指针连接各个节点来表示二叉树的数据结构。在这种实现方式中，每个节点包含数据元素以及指向其左子节点和右子节点的指针。这种存储结构的特点包括：可以动态地分配内存空间来创建节点，易于实现插入、删除和遍历等操作，适用于各种二叉树的表示和操作。

通过实验，我学习了如何在实际编程中实现基于二叉链表的二叉树，并进行相关操作。可以通过定义节点结构体来表示二叉树的节点，通过指针来连接各个节点，实现插入、删除、查找等操作。在实验过程中，我也学会了如何遍历二叉树、实现二叉树的各种算法，比如前序遍历、中序遍历、后序遍历等。最令我头疼的是第一题二叉树不同定义下的创建，也即先序创建，先序加中序创建和后序加中序创建，我耗费了很长时间才把这三种都写出来，但最后并没有检查这个，令我感到很遗憾。

3.4 基于邻接表的图实现

通过本专题，我了解到基于邻接表的图是一种常用的图的表示方法，通过使用链式结构存储图中每个顶点的邻接点，可以有效地表示稀疏图。在邻接表中，每个顶点都有一个链表，链表中存储了与该顶点相邻的其他顶点。这种存储结构的特点包括：节省空间，适用于稀疏图；方便查找某个顶点的邻接点；便于插入和删除边。

通过实验，我学习了如何在实际编程中实现基于邻接表的图，并进行相关操作。可以通过使用链表来表示图中的每个顶点及其邻接点，实现插入边、删除边、查找邻接点等操作。在实验过程中，我也学会了如何遍历图、实现图的各种算法，比如深度优先搜索（DFS）和广度优先搜索（BFS）等。在做完二叉树专题后，我感觉我对图这一专题做起来已经比较得心应手了。

我加深了对邻接表的图的理解，包括顶点之间的连接关系和操作方法。我还学会了如何分析和设计图相关的算法，以及如何在实际问题中应用图来解决问题。

4 附录 A 基于顺序存储结构线性表实现的源程序

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

#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2

typedef int status;
typedef int ElemType; //数据元素类型定义

/*-----page 22 on textbook -----*/
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
typedef struct{ //顺序表（顺序结构）的定义
    ElemType * elem;
    int length;
    int listsize;
}SqlList;
typedef struct{ //线性表的管理表定义
    struct { char name[30];
        SqlList L;
    } elem[10];
    int length;
    int listsize;
}LISTS;
```

```
/*-----page 19 on textbook -----*/
status InitList(SqList& L)
// 线性表L不存在，构造一个空的线性表，返回OK，否则返回INFEASIBLE。
{
// 请在这里补充代码，完成本关任务
/***** Begin *****/
if(L.elem==NULL){
L.elem=(ElemType *) malloc(sizeof(ElemType)*LIST_INIT_SIZE);
L.length=0;
L.listsize=LIST_INIT_SIZE;
return OK;
}
else return INFEASIBLE;

/***** End *****/
}

status DestroyList(SqList& L)
// 如果线性表L存在，销毁线性表L，释放数据元素的空间，返回OK，否则返回INFEASIBLE。
{
// 请在这里补充代码，完成本关任务
/***** Begin *****/
if(L.elem==NULL) return INFEASIBLE;
else{
free(L.elem);
L.elem=NULL;
int length;
int listsize;
return OK;
}

/***** End *****/
}
```

```
}

status ClearList(SqList& L)
// 如果线性表L存在，删除线性表L中的所有元素，返回OK，否则返回INFEASIBLE。
{
// 请在这里补充代码，完成本关任务
/***** Begin *****/
if(L.elem==NULL) return INFEASIBLE;
else{
L.length=0;
free(L.elem);
L.elem=(ElemType *) malloc(sizeof(ElemType));
return OK;
}
/***** End *****/
}

status ListEmpty(SqList L)
/* 如果线性表L存在，判断线性表L是否为空，空就返回TRUE，否则返回FALSE；
如果线性表L不存在，返回INFEASIBLE。*/
{
// 请在这里补充代码，完成本关任务
/***** Begin *****/
if(L.elem==NULL) return INFEASIBLE;
else{
if(!L.length) return TRUE;
else return FALSE;
}
/***** End *****/
}

status ListLength(SqList L)
// 如果线性表L存在，返回线性表L的长度，否则返回INFEASIBLE。
```

```
{
// 请在这里补充代码，完成本关任务

/***** Begin *****/
if(L.elem==NULL) return INFEASIBLE;
else return L.length;

/***** End *****/
}

status GetElem(SqList L,int i,ElemType &e)
// 如果线性表L存在，获取线性表L的第i个元素，保存在e中，返回OK；
// 如果i不合法，返回ERROR；如果线性表L不存在，返回INFEASIBLE。
{
// 请在这里补充代码，完成本关任务

/***** Begin *****/
if(L.elem==NULL) return INFEASIBLE;
else if(i<1||i>L.length) return ERROR;
else{
e=L.elem[i-1];
return OK;
}

/***** End *****/
}

int LocateElem(SqList L,ElemType e)
// 如果线性表L存在，查找元素e在线性表L中的位置序号并返回该序号；
// 如果e不存在，返回0；当线性表L不存在时，返回INFEASIBLE（即-1）。
{
// 请在这里补充代码，完成本关任务

/***** Begin *****/
if(L.elem==NULL) return INFEASIBLE;
else{
```

```
for(int i=0;i<=L.length;i++){
    if(e==L.elem[i]) return i+1;
}
return ERROR;
}

/***** End *****/

}

status PriorElem(SqList L,ElemType e,ElemType &pre)
// 如果线性表L存在, 获取线性表L中元素e的前驱, 保存在pre中, 返回OK;
// 如果没有前驱, 返回ERROR; 如果线性表L不存在, 返回INFEASIBLE。
{
    // 请在这里补充代码, 完成本关任务
    /***** Begin *****/
    if(L.elem==NULL) return INFEASIBLE;
    else{
        int i=0;
        for(i;i<=L.length;i++){
            if(e==L.elem[i]) break;
        }
        if(i==0||e!=L.elem[i]) return ERROR;
        else{
            pre=L.elem[i-1];
            return OK;
        }
    }

    /***** End *****/

}

status NextElem(SqList L,ElemType e,ElemType &next)
// 如果线性表L存在, 获取线性表L元素e的后继, 保存在next中, 返回OK;
```

如果没有后继, 返回ERROR; 如果线性表L不存在, 返回INFEASIBLE。

```
{
// 请在这里补充代码, 完成本关任务
/***** Begin *****/
if(L.elem==NULL) return INFEASIBLE;
else{
int i=0;
for(i;i<L.length;i++){
if(e==L.elem[i]) break;
}
if(i>=L.length-1) return ERROR;
else{
next=L.elem[i+1];
return OK;
}
}
/***** End *****/
}

status ListInsert(SqList &L,int i,ElemType e)
// 如果线性表L存在, 将元素e插入到线性表L的第i个元素之前, 返回OK;
// 当插入位置不正确时, 返回ERROR; 如果线性表L不存在, 返回INFEASIBLE。
{
// 请在这里补充代码, 完成本关任务
/***** Begin *****/
if(L.elem==NULL) return INFEASIBLE;
else{
if(i>=1&& i<=L.length+1){
if(L.length >= L.listsize){
L.listsize+=LISTINCREMENT;
L.elem=(ElemType *) realloc(L.elem,sizeof(ElemType)*L.listsize);
}
}
```



```
for(int j=L.length;j>=i;j--){
    L.elem[j]=L.elem[j-1];
}
L.elem[i-1]=e;
L.length++;
return OK;
}
else return ERROR;
}
/***** End *****/
}

status ListDelete(SqList &L,int i,ElemType &e)
// 如果线性表L存在，删除线性表L的第i个元素，并保存在e中，返回OK；
// 当删除位置不正确时，返回ERROR；如果线性表L不存在，返回INFEASIBLE。
{
    // 请在这里补充代码，完成本关任务
    /***** Begin *****/
    if(L.elem==NULL) return INFEASIBLE;
    else{
        if(i>=1&&i<=L.length){
            e=L.elem[i-1];
            for(int j=i-1;j<L.length-1;j++){
                L.elem[j]=L.elem[j+1];
            }

            L.length--;
            return OK;
        }
        else return ERROR;
    }
}
```

```
/****** End *****/
}

status ListTraverse(SqList L)
// 如果线性表L存在，依次显示线性表中的元素，每个元素间空一格，返回OK；
// 如果线性表L不存在，返回INFEASIBLE。
{
// 请在这里补充代码，完成本关任务
/****** Begin *****/
if(L.elem==NULL) return INFEASIBLE;
else{
if(!L.length) return OK;
else{
for(int i=0;i<L.length;i++){
if(i<L.length-1) printf("%d ",L.elem[i]);
else printf("%d",L.elem[i]);
}
return OK;
}
}

/****** End *****/
}

int MaxSubArray(SqList L) {
// 检查线性表是否非空
if (L.length == 0) {
return 0;
}

int maxSum = L.elem[0]; // 全局最大和
int currentSum = L.elem[0]; // 当前子数组的最大和
```

```
// 从第二个元素开始遍历数组
for (int i = 1; i < L.length; i++) {
// 更新当前子数组的最大和
currentSum = (currentSum > 0) ? currentSum + L.elem[i] : L.elem[i];

// 更新全局最大和
if (currentSum > maxSum) {
maxSum = currentSum;
}
}

// 返回最大和
return maxSum;
}

#define MAX_SIZE 10001

int SubArrayNum(SqList L, int k) {
// 检查线性表是否非空
if (L.length == 0) {
return 0;
}

int count = 0; // 计数器
int sum = 0; // 前缀和
int prefixSum[MAX_SIZE] = {0}; // 用于存储前缀和及其出现次数

// 遍历线性表，计算前缀和
for (int i = 0; i < L.length; i++) {
sum += L.elem[i];

// 如果前缀和等于 k，直接将计数器加一
```

```
if (sum == k) {
    count++;
}

// 如果前缀和减去 k 在前缀和表中存在，则说明存在连续子数组和为 k
if (sum - k >= 0 && sum - k < MAX_SIZE) {
    count += prefixSum[sum - k];
}

// 更新前缀和表
prefixSum[sum]++;
}

// 返回结果
return count;
}

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void sortList(Sqlist *L) {
    int i, j;
    for (i = 0; i < L->length - 1; i++) {
        for (j = 0; j < L->length - 1 - i; j++) {
            if (L->elem[j] > L->elem[j + 1]) {
                swap(&L->elem[j], &L->elem[j + 1]);
            }
        }
    }
}
```

```
}

status SaveList(Sqlist L,char FileName[])
// 如果线性表L存在, 将线性表L的元素写到FileName文件中, 返回OK
{
// 请在这里补充代码, 完成本关任务
/***** Begin *****/
FILE *fp;
fp = fopen(FileName, "w");
if (fp == NULL||L.elem==NULL) {
return INFEASIBLE;
}
for (int i = 0; i < L.length; i++) {
fprintf(fp, "%d ", L.elem[i]);
}
fclose(fp);
return OK;
/***** End *****/
}

status LoadList(Sqlist &L,char FileName[])
// 如果线性表L不存在, 将FileName文件中的数据读入到线性表L中, 返回OK
{
// 请在这里补充代码, 完成本关任务
/***** Begin *****/
FILE *fp;
fp = fopen(FileName, "r");
if (fp == NULL||L.elem!=NULL) {
return INFEASIBLE;
}
L.elem=(ElemType *) malloc(sizeof(ElemType)*LIST_INIT_SIZE);
int value;
```

```
int i = 0;
while (fscanf(fp, "%d", &value) != EOF) {
    L.elem[i] = value;
    i++;
}
L.length = i;
fclose(fp);
return OK;
/***** End *****/
}

status AddList(LISTS &Lists, char ListName[])
// 只需要在Lists中增加一个名称为ListName的空线性表，线性表数据又后台测试程序插入。
{
    // 请在这里补充代码，完成本关任务
    /***** Begin *****/
    strcpy(Lists.elem[Lists.length].name, ListName);
    Lists.elem[Lists.length].L.elem = NULL;
    InitList(Lists.elem[Lists.length].L);
    Lists.length++;
    return OK;
    /***** End *****/
}

status RemoveList(LISTS &Lists, char ListName[])
// Lists中删除一个名称为ListName的线性表
{
    // 请在这里补充代码，完成本关任务
    /***** Begin *****/
    for(int i=0; i<Lists.length; i++){
        if(strcmp(ListName, Lists.elem[i].name)==0){
            DestroyList(Lists.elem[i].L);
            for(int j=i; j<Lists.length-1; j++){
```

```
strcpy(Lists.elem[j].name, Lists.elem[j+1].name);
Lists.elem[j].L = Lists.elem[j+1].L;
}

Lists.length--;
return OK;
}
}
return ERROR;
/***** End *****/
}

int LocateList(LISTS Lists, char ListName[])
// 在Lists中查找一个名称为ListName的线性表，成功返回逻辑序号，否则返回0
{
// 请在这里补充代码，完成本关任务
/***** Begin *****/
for(int i=0; i<Lists.length; i++){
if(strcmp(ListName, Lists.elem[i].name)==0)
return i+1;
}
return 0;

/***** End *****/
}

void singleListOperation(SqList &L){
int op=1, op2=1, i, j, e, pre, next;
while(op){
printf("\n\n");
printf("      Menu for Linear Table On Sequence Structure \n");
printf("-----\n");
printf("      1. InitList          10. ListInsert \n");
```

```
printf("      2. DestroyList      11. ListDelete \n");
printf("      3. ClearList      12. ListTraverse \n");
printf("      4. ListEmpty      13. MaxSubArray \n");
printf("      5. ListLength      14. SubArrayNum \n");
printf("      6. GetElem      15. sortList \n");
printf("      7. LocateElem      16. File \n");
printf("      8. PriorElem      \n");
printf("      9. NextElem      \n");
printf("      0. Exit\n");

printf("-----\n");
printf("      请选择你的操作[0~17]:");
scanf("%d",&op);
switch(op){
case 1:

getchar();getchar();
break;
case 2:
if(DestroyList(L)==INFEASIBLE) printf("线性表不存在! \n");
else printf("线性表销毁成功! \n");
getchar();getchar();
break;
case 3:
if(ClearList(L)==INFEASIBLE) printf("线性表不存在! \n");
else printf("线性表删除成功! \n");
getchar();getchar();
break;
case 4:
if(ListEmpty(L)==INFEASIBLE) printf("线性表不存在! \n");
else if(ListEmpty(L)==TRUE) printf("线性表判断为空! \n");
else printf("线性表判断不为空! \n");
```



```
getchar();getchar();
break;
case 5:
i=ListLength(L);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else printf("线性表长度为%d!\n",i);
getchar();getchar();
break;
case 6:
printf("请输入要获取的元素的序号: ");
scanf("%d",&j);
i=GetElem(L,j,e);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(j<=L.length) printf("线性表第%d个元素为%d! \n",j,e);
else printf("序号超限");
getchar();getchar();
break;
case 7:
printf("请输入要查找的元素: ");
scanf("%d",&e);
i=LocateElem(L,e);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(i==ERROR) printf("查找失败");
else printf("线性表指定%d元素的逻辑序号为%d! \n",e,j);
getchar();getchar();
break;
case 8:
printf("请输入要获取前驱元素的元素: ");
scanf("%d",&e);
i=PriorElem(L,e,pre);
if(i==INFEASIBLE) printf("线性表不存在! \n");
```

```
else if(i==ERROR) printf("线性表中不存在指定元素%d的前驱元素! \n",e);
else printf("线性表中指定元素%d的前驱元素为%d! \n",e,pre);
getchar();getchar();
break;
case 9:
printf("请输入要获取后继元素的元素: ");
scanf("%d",&e);
i=NextElem(L,e,next);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(i==ERROR) printf("线性表中不存在指定元素%d的后继元素! \n",e);
else printf("线性表中指定元素%d的后继元素为%d! \n",e,next);
getchar();getchar();
break;
case 10:
printf("请输入要在线性表中插入元素的序号和要插入的元素: ");
scanf("%d%d",&j,&e);
i=ListInsert(L,j,e);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(i==ERROR) printf("线性表元素插入失败! \n");
else{
printf("线性表元素插入成功! \n新线性表为: ");
for(i=0;i<L.length;i++)
printf("%d ",L.elem[i]);
}
getchar();getchar();
break;
case 11:
printf("请输入要在线性表中删除的元素: ");
scanf("%d",&j);
i=ListDelete(L,j,e);
if(i==INFEASIBLE) printf("线性表不存在! \n");
```

```
else if(i==ERROR) printf("线性表元素删除失败！\n");
else{
printf("线性表元素删除成功！\n删除元素为： %d\n新线性表为： ",e);
for(i=0;i<L.length;i++)
printf("%d ",L.elem[i]);
}
getchar();getchar();
break;
case 12:
if(!ListTraverse(L)) printf("线性表是空表！\n");
getchar();getchar();
break;
case 13:
i=MaxSubArray(L);
if(i==INFEASIBLE) printf("线性表不存在！\n");
else printf("连续子数组的最大和为： %d",i);
getchar();getchar();
break;
case 14:
printf("请输入连续子数组的和k: ");
scanf("%d",&j);
i=SubArrayNum(L,j);
if(i==INFEASIBLE) printf("线性表不存在！\n");
else printf("该线性表中和为%d的连续子数组的个数为： %d",j,i);
getchar();getchar();
break;
case 15:
sortList(&L);
if(L.elem==NULL) printf("线性表不存在！\n");
else{
printf("线性表元素排序成功！\n新线性表为： ",e);
```

```
for(i=0;i<L.length;i++)
printf("%d ",L.elem[i]);
}
getchar();getchar();
break;
case 16:
FILE *fp;
i=ListEmpty(L);
if(i==INFEASIBLE){
printf("线性表不存在！对空表进行读文件操作！读得线性表为：\n");
LoadList(L,"C:\\Users\\lenovo\\Desktop\\listfile.txt");

for(i=0;i<L.length;i++)
printf("%d ",L.elem[i]);
}
else{
printf("线性表判断存在！对线性表进行写文件操作！\n");
SaveList(L,"C:\\Users\\lenovo\\Desktop\\listfile.txt");

}

getchar();getchar();
break;
case 0:
break;
}
}
printf("对多表的单表操作结束！\n");
}
/*-----*/
int main(void){
```

```
SqList L;  int op=1,op2=1,i,j,e,pre,next;
L.elem=NULL;
while(op){
printf("\n\n");
printf("          Menu for Linear Table On Sequence Structure \n");
printf("-----\n");
printf("          1. InitList          10. ListInsert \n");
printf("          2. DestroyList       11. ListDelete \n");
printf("          3. ClearList         12. ListTraverse \n");
printf("          4. ListEmpty         13. MaxSubArray \n");
printf("          5. ListLength        14. SubArrayNum \n");
printf("          6. GetElem           15. sortList \n");
printf("          7. LocateElem        16. File \n");
printf("          8. PriorElem         17. Lists\n");
printf("          9. NextElem          \n");
printf("          0. Exit\n");
printf("-----\n");
printf("      请选择你的操作[0~17]:");
scanf("%d",&op);
switch(op){
case 1:
//printf("\n----IntiList功能待实现! \n");

if(InitList(L)==OK) printf("线性表创建成功! \n");
else printf("线性表创建失败! \n");
getchar();getchar();
break;
case 2:
if(DestroyList(L)==INFEASIBLE) printf("线性表不存在! \n");
else printf("线性表销毁成功! \n");
getchar();getchar();
```

```
break;
case 3:
if(ClearList(L)==INFEASIBLE) printf("线性表不存在! \n");
else printf("线性表删除成功! \n");
getchar();getchar();
break;
case 4:
if(ListEmpty(L)==INFEASIBLE) printf("线性表不存在! \n");
else if(ListEmpty(L)==TRUE) printf("线性表判断为空! \n");
else printf("线性表判断不为空! \n");
getchar();getchar();
break;
case 5:
i=ListLength(L);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else printf("线性表长度为%d!\n",i);
getchar();getchar();
break;
case 6:
printf("请输入要获取的元素的序号: ");
scanf("%d",&j);
i=GetElem(L,j,e);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(j<=L.length) printf("线性表第%d个元素为%d! \n",j,e);
else printf("序号超限");
getchar();getchar();
break;
case 7:
printf("请输入要查找的元素: ");
scanf("%d",&e);
i=LocateElem(L,e);
```

```
if(i==INFEASIBLE) printf("线性表不存在！\n");
else if(i==ERROR) printf("查找失败");
else printf("线性表指定%d元素的逻辑序号为%d！\n",e,j);
getchar();getchar();
break;
case 8:
printf("请输入要获取前驱元素的元素：");
scanf("%d",&e);
i=PriorElem(L,e,pre);
if(i==INFEASIBLE) printf("线性表不存在！\n");
else if(i==ERROR) printf("线性表中不存在指定元素%d的前驱元素！\n",e);
else printf("线性表中指定元素%d的前驱元素为%d！\n",e,pre);
getchar();getchar();
break;
case 9:
printf("请输入要获取后继元素的元素：");
scanf("%d",&e);
i=NextElem(L,e,next);
if(i==INFEASIBLE) printf("线性表不存在！\n");
else if(i==ERROR) printf("线性表中不存在指定元素%d的后继元素！\n",e);
else printf("线性表中指定元素%d的后继元素为%d！\n",e,next);
getchar();getchar();
break;
case 10:
printf("请输入要在线性表中插入元素的序号和要插入的元素：");
scanf("%d%d",&j,&e);
i=ListInsert(L,j,e);
if(i==INFEASIBLE) printf("线性表不存在！\n");
else if(i==ERROR) printf("线性表元素插入失败！\n");
else{
printf("线性表元素插入成功！\n新线性表为：");
```

```
for(i=0;i<L.length;i++)
printf("%d ",L.elem[i]);
}
getchar();getchar();
break;
case 11:
printf("请输入要在线性表中删除的元素序号: ");
scanf("%d",&j);
i=ListDelete(L,j,e);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(i==ERROR) printf("线性表元素删除失败! \n");
else{
printf("线性表元素删除成功! \n删除元素为: %d\n新线性表为: ",e);
for(i=0;i<L.length;i++)
printf("%d ",L.elem[i]);
}
getchar();getchar();
break;
case 12:
ListTraverse(L);
getchar();getchar();
break;
case 13:
i=MaxSubArray(L);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else printf("连续子数组的最大和为: %d",i);
getchar();getchar();
break;
case 14:
printf("请输入连续子数组的和k: ");
scanf("%d",&j);
```



```
i=SubArrayNum(L,j);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else printf("该线性表中和为%d的连续子数组的个数为: %d",j,i);
getchar();getchar();
break;
case 15:
sortList(&L);
if(L.elem==NULL) printf("线性表不存在! \n");
else{
printf("线性表元素排序成功! \n新线性表为: ",e);
for(i=0;i<L.length;i++)
printf("%d ",L.elem[i]);
}
getchar();getchar();
break;
case 16:
FILE *fp;
i=ListEmpty(L);
if(i==INFEASIBLE){
printf("线性表不存在! 对空表进行读文件操作! 读得线性表为: \n");
LoadList(L,"C:\\Users\\lenovo\\Desktop\\listfile.txt");

for(i=0;i<L.length;i++)
printf("%d ",L.elem[i]);
}
else{
printf("线性表判断存在! 对线性表进行写文件操作! \n");
SaveList(L,"C:\\Users\\lenovo\\Desktop\\listfile.txt");

}
```

```
getchar();getchar();
break;
case 17:
LISTS Lists;
Lists.length=0;
while(op2){

printf("          Management Of Multiple Linear Tables  \n");
printf("-----\n");
printf("      1. AddList          3.LocateList \n");
printf("      2. RemoveList       0.Exit\n");
printf("-----\n");
printf("    请选择你的操作[0~3]:");

scanf("%d",&op2);
char name[30];
switch(op2){

case 1:
int n,e;

printf("请输入要添加的线性表的个数: ");
scanf("%d",&n);
while(n--)
{
scanf("%s",name);
AddList(Lists,name);
scanf("%d",&e);
while (e
```

```
{
ListInsert(Lists.elem[Lists.length-1].L,List elem
[Lists.length-1].L.length+1,e);
scanf("%d",&e);
}
}
printf("添加后的多线性表为: ");
for(n=0;n<Lists.length;n++)
{
printf("%s ",Lists.elem[n].name);
ListTraverse(Lists.elem[n].L);
putchar('\n');
}
getchar();getchar();
break;
case 2:
printf("请输入要删除的线性表的名字: ");
scanf("%s",name);
if (RemoveList(Lists,name)==OK){
printf("删除后的线性表为: \n");
for(n=0;n<Lists.length;n++)
{
printf("%s ",Lists.elem[n].name);
ListTraverse(Lists.elem[n].L);
putchar('\n');
}
}
else printf("删除失败");

getchar();getchar();
break;
```


5 附录 B 基于链式存储结构线性表实现的源程序

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
typedef int status;
typedef int ElemType;
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
typedef struct LNode{ //单链表（链式结构）结点的定义
    ElemType data;
    struct LNode *next;
}LNode,*LinkList;
typedef struct{ //线性表的管理表定义
    struct { char name[30];
        LinkList L;
    } elem[10];
    int length;
    int listsize;
}LISTS;

status InitList(LinkList &L)
// 线性表L不存在，构造一个空的线性表，返回OK，否则返回INFEASIBLE。
{
```

```
if (L==NULL){
L=(LinkList )malloc(sizeof(LinkList));
L->next=NULL;
return OK;
}
else return INFEASIBLE;
}

status DestroyList(LinkList &L)
// 如果线性表L存在, 销毁线性表L, 释放数据元素的空间, 返回OK, 否则返回INFEASIBLE。
{
if(L==NULL) return INFEASIBLE;
else{
LinkList p=L,temp;
while(p){
temp=p->next;
free(p);
p=temp;
}
L=NULL;
return OK;
}
}

status ClearList(LinkList &L)
// 如果线性表L存在, 删除线性表L中的所有元素, 返回OK, 否则返回INFEASIBLE。
{
if(L==NULL) return INFEASIBLE;
else{
LinkList p=L->next,temp;
while(p){
temp=p->next;
```

```
free(p);
p=temp;
}
L->next=NULL;
return OK;
}
}

status ListEmpty(LinkList L)
// 如果线性表L存在, 判断线性表L是否为空, 空就返回TRUE, 否则返回FALSE;
// 如果线性表L不存在, 返回INFEASIBLE。
{
if(L==NULL) return INFEASIBLE;
else{
if(L->next==NULL) return TRUE;
else return FALSE;
}
}

int ListLength(LinkList L)
// 如果线性表L存在, 返回线性表L的长度, 否则返回INFEASIBLE。
{
if(L==NULL) return INFEASIBLE;
else{
int i=0;
while(L){
L=L->next;
i++;
}
return i-1;
}
}

status GetElem(LinkList L,int i,ElemType &e)
```

// 如果线性表L存在，获取线性表L的第i个元素，保存在e中，返回OK；
如果i不合法，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

```
{  
if(L==NULL) return INFEASIBLE;  
else{  
int j=0;  
for(j;j<i&&L!=NULL;j++){  
L=L->next;  
}  
if(i==0||i>ListLength(L)) return ERROR;  
else{  
e=L->data;  
return OK;  
}  
}  
}
```

status LocateElem(LinkList L,ElemType e)

// 如果线性表L存在，查找元素e在线性表L中的位置序号；如果e不存在，
返回ERROR；当线性表L不存在时，返回INFEASIBLE。

```
{  
if(L==NULL) return INFEASIBLE;  
else{  
int i=1;  
while(L){  
if(L->data==e) return i-1;  
L=L->next;  
i++;  
}  
if(i==0||L==NULL) return ERROR;  
else{  
e=L->data;
```



```
return OK;
}
}
}

status PriorElem(LinkList L,ElemType e,ElemType &pre)
// 如果线性表L存在, 获取线性表L中元素e的前驱, 保存在pre中, 返回OK;
// 如果没有前驱, 返回ERROR; 如果线性表L不存在, 返回INFEASIBLE。
{
    if(L==NULL) return INFEASIBLE;
    if(L->next==NULL) return ERROR;
    LinkList p1=L->next,p2=p1->next;
    while(p2){
        if(p2->data==e){
            pre=p1->data;
            return OK;
        }
        p1=p1->next;
        p2=p1->next;
    }
    return ERROR;
}

status NextElem(LinkList L,ElemType e,ElemType &next)
// 如果线性表L存在, 获取线性表L元素e的后继, 保存在next中, 返回OK;
// 如果没有后继, 返回ERROR; 如果线性表L不存在, 返回INFEASIBLE。
{
    if(L==NULL) return INFEASIBLE;
    if(L->next==NULL) return ERROR;
    LinkList p1=L->next,p2=p1->next;
    while(p2){
        if(p1->data==e){
            next=p2->data;
        }
        p1=p1->next;
        p2=p1->next;
    }
    return ERROR;
}
```

```
return OK;
}
p1=p1->next;
p2=p1->next;
}
return ERROR;
}

status ListInsert(LinkList &L,int i,ElemType e)
// 如果线性表L存在，将元素e插入到线性表L的第i个元素之前，返回OK；
// 当插入位置不正确时，返回ERROR；如果线性表L不存在，返回INFEASIBLE。
{
    if(L==NULL) return INFEASIBLE;
    if(i<=0) return ERROR;
    LinkList p1=L,insert=(LinkList) malloc(sizeof(LNode));
    for(int j=0;j<i-1;j++){
        if(p1->next==NULL) return ERROR;
        p1=p1->next;
    }
    insert->data=e;
    insert->next=p1->next;
    p1->next=insert;
    return OK;
}

status ListDelete(LinkList &L,int i,ElemType &e)
// 如果线性表L存在，删除线性表L的第i个元素，并保存在e中，返回OK；
// 当删除位置不正确时，返回ERROR；如果线性表L不存在，返回INFEASIBLE。
{
    if(L==NULL) return INFEASIBLE;
    if(i==0) return ERROR;
    LinkList p1=L,p2=L->next;
    int j;
```

```
for(int j=0;j<i-1;j++){
    if(p1->next==NULL) return ERROR;
    p1=p1->next;
    p2=p1->next;
}
if(p2==NULL) return ERROR;
e=p2->data;
p1->next=p2->next;
free(p2);
return OK;
}

status ListTraverse(LinkList L)
// 如果线性表L存在, 依次显示线性表中的元素, 每个元素间空一格, 返回OK;
// 如果线性表L不存在, 返回INFEASIBLE。
{
    if(L==NULL) return INFEASIBLE;
    LinkList p=L->next;
    if(p==NULL) return OK;
    printf("%d",p->data);
    p=p->next;
    while(p){
        printf(" %d",p->data);
        p=p->next;
    }
    return OK;
}

LinkList reverseList(LinkList &head) {
    if (head == NULL || head->next == NULL) {
        return head;
    }
}
```

```
LinkedList pre = NULL, cur = head->next, next = NULL;
while (cur != NULL) {
    next = cur->next;
    cur->next = pre;
    pre = cur;
    cur = next;
}
head->next = pre;

return head;
}
```

```
status RemoveNthFromEnd(LinkedList &L, int n) {
    // 如果链表为空或只有一个节点，无法删除倒数第n个节点
    if (L == NULL || L->next == NULL) {
        return INFEASIBLE;
    }
```

```
    LinkedList fast = L->next, slow = L->next;
    // 快指针先移动n步
    for (int i = 0; i < n; i++) {
        if (fast == NULL) {
            return ERROR; // 链表长度小于n，删除失败
        }
        fast = fast->next;
    }
```

```
    LinkedList prev = NULL;
    // 快慢指针一起移动，直到快指针到达链表尾部
    while (fast != NULL) {
        prev = slow;
```

```
slow = slow->next;
fast = fast->next;
}

if (prev == NULL) {
// 如果要删除的是头节点
LinkedList temp = L->next;
L->next = L->next->next;
free(temp);
} else {
// 删除中间或尾部节点
prev->next = slow->next;
free(slow);
}
return OK; // 删除成功
}

status sortList(LinkedList &L) {
if (L == NULL || L->next == NULL) {
return INFEASIBLE;
}

LinkedList p = L->next, q = NULL;
ElemType temp;
while (p != NULL) {
q = p->next;
while (q != NULL) {
if (p->data > q->data) {
temp = p->data;
p->data = q->data;
q->data = temp;
```

```
}
q = q->next;
}
p = p->next;
}
return OK;
}

status SaveList(LinkList L,char FileName[])
// 如果线性表L存在, 将线性表L的元素写到FileName文件中, 返回OK,
// 否则返回INFEASIBLE。
{
FILE *fp;
fp = fopen(FileName, "w");
if (fp == NULL||L==NULL) {
return INFEASIBLE;
}
LinkList p=L->next;
if(p==NULL) return OK;
while(p) {
fprintf(fp, "%d ", p->data);
p=p->next;
}
fclose(fp);
return OK;
}

status LoadList(LinkList &L,char FileName[])
// 如果线性表L不存在, 将FileName文件中的数据读入到线性表L中,
// 返回OK, 否则返回INFEASIBLE。
{
FILE *fp;
fp = fopen(FileName, "r");
```

```
if (fp == NULL || L != NULL) {
return INFEASIBLE;
}

int value;
L=(LinkedList) malloc(sizeof(LNode));
LinkedList p=L;
while (fscanf(fp, "%d", &value) != EOF) {
p->next=(LinkedList) malloc(sizeof(LNode));
p=p->next;
p->data = value;
}
p->next=NULL;
fclose(fp);
return OK;
}
```

```
status AddList(LISTS &Lists, char ListName[])
// 只需要在Lists中增加一个名称为ListName的空线性表，线性表数据又后台测试程序插入。
{
strcpy(Lists.elem[Lists.length].name, ListName);
Lists.elem[Lists.length].L= NULL;
InitList(Lists.elem[Lists.length].L);
Lists.length++;
return OK;
}

status RemoveList(LISTS &Lists, char ListName[])
// Lists中删除一个名称为ListName的线性表
{
for(int i=0; i<Lists.length; i++){
if(strcmp(ListName, Lists.elem[i].name)==0){
```

```
DestroyList(Lists.elem[i].L);
for(int j=i; j<Lists.length-1; j++){
    strcpy(Lists.elem[j].name, Lists.elem[j+1].name);
    Lists.elem[j].L = Lists.elem[j+1].L;
}
Lists.length--;
return OK;
}
}
return ERROR;
}

int LocateList(LISTS Lists, char ListName[])
// 在Lists中查找一个名称为ListName的线性表，成功返回逻辑序号，否则返回0
{
    for(int i=0; i<Lists.length; i++){
        if(strcmp(ListName, Lists.elem[i].name)==0)
            return i+1;
    }
    return 0;
}

void singleListOperation(LinkList &L){
    int op=1, op2=1, i, j, e, pre, next;
    while(op){
        printf("\n\n");
        printf("      Menu for Linear Table On Sequence Structure \n");
        printf("-----\n");
        printf("      1. InitList          10. ListInsert \n");
        printf("      2. DestroyList       11. ListDelete \n");
        printf("      3. ClearList         12. ListTraverse \n");
        printf("      4. ListEmpty         13. reverseList \n");
        printf("      5. ListLength        14. RemoveNthFromEnd \n");
```



```
printf("        6. GetElem          15. sortList \n");
printf("        7. LocateElem       16. File \n");
printf("        8. PriorElem           \n");
printf("        9. NextElem            \n");
printf("        0. Exit\n");
printf("-----\n");
printf("    请选择你的操作[0~17]:");
scanf("%d",&op);
switch(op){
case 1:
//printf("\n----IntiList功能待实现! \n");

if(InitList(L)==OK) printf("线性表创建成功! \n");
else printf("线性表创建失败! \n");
getchar();getchar();
break;
case 2:
if(DestroyList(L)==INFEASIBLE) printf("线性表不存在! \n");
else printf("线性表销毁成功! \n");
getchar();getchar();
break;
case 3:
if(ClearList(L)==INFEASIBLE) printf("线性表不存在! \n");
else printf("线性表删除成功! \n");
getchar();getchar();
break;
case 4:
if(ListEmpty(L)==INFEASIBLE) printf("线性表不存在! \n");
else if(ListEmpty(L)==TRUE) printf("线性表判断为空! \n");
else printf("线性表判断不为空! \n");
getchar();getchar();
```

```
break;
case 5:
i=ListLength(L);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else printf("线性表长度为%d!\n",i);
getchar();getchar();
break;
case 6:
printf("请输入要获取的元素的序号: ");
scanf("%d",&j);
i=GetElem(L,j,e);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else printf("线性表第%d个元素为%d! \n",j,e);
getchar();getchar();
break;
case 7:
printf("请输入要查找的元素: ");
scanf("%d",&e);
i=LocateElem(L,e);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(i==ERROR) printf("查找失败");
else printf("线性表指定%d元素的逻辑序号为%d! \n",e,j);
getchar();getchar();
break;
case 8:
printf("请输入要获取前驱元素的元素: ");
scanf("%d",&e);
i=PriorElem(L,e,pre);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(i==ERROR) printf("线性表中不存在指定元素%d的前驱元素! \n",e);
else printf("线性表中指定元素%d的前驱元素为%d! \n",e,pre);
```

```
getchar();getchar();
break;
case 9:
printf("请输入要获取后继元素的元素: ");
scanf("%d",&e);
i=NextElem(L,e,next);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(i==ERROR) printf("线性表中不存在指定元素%d的后继元素! \n",e);
else printf("线性表中指定元素%d的后继元素为%d! \n",e,next);
getchar();getchar();
break;
case 10:
printf("请输入要在线性表中插入元素的序号和要插入的元素: ");
scanf("%d%d",&j,&e);
i=ListInsert(L,j,e);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(i==ERROR) printf("线性表元素插入失败! \n");
else{
printf("线性表元素插入成功! \n新线性表为: ");
ListTraverse(L);
putchar('\n');
}
getchar();getchar();
break;
case 11:
printf("请输入要在线性表中删除的元素序号: ");
scanf("%d",&j);
i=ListDelete(L,j,e);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(i==ERROR) printf("线性表元素删除失败! \n");
else{
```

```
printf("线性表元素删除成功！\n删除元素为：%d\n新线性表为：",e);
ListTraverse(L);
putchar('\n');
}
getchar();getchar();
break;
case 12:
if(L==NULL) printf("线性表不存在！\n");
else ListTraverse(L);
getchar();getchar();
break;
case 13:
if(ListEmpty(L)!=INFEASIBLE){
L=reverseList(L);
printf("反转后的线性表为：");
ListTraverse(L);putchar('\n');
}

else printf("线性表不存在！\n");

getchar();getchar();
break;
case 14:

printf("请输入n的值：");
scanf("%d",&j);
i=RemoveNthFromEnd(L,j);
if(i!=OK) printf("删除失败！\n");
else{
printf("删除链表的倒数第%d个结点后的线性表为：",j);
ListTraverse(L);
```

```
putchar('\n');
}
getchar();getchar();
break;
case 15:
sortList(L);
if(L==NULL) printf("线性表不存在! \n");
else{
printf("线性表元素排序成功! \n新线性表为: ");
ListTraverse(L);
putchar('\n');
}
getchar();getchar();
break;
case 16:
FILE *fp;
i=ListEmpty(L);
if(i==INFEASIBLE){
printf("线性表不存在! 对空表进行读文件操作! 读得线性表为: \n");
LoadList(L,"C:\\Users\\lenovo\\Desktop\\listfile.txt");
ListTraverse(L);
}

else{
printf("线性表判断存在! 对线性表进行写文件操作! \n");
SaveList(L,"C:\\Users\\lenovo\\Desktop\\listfile.txt");
}
getchar();getchar();
break;
case 0:
break;
```

```
}
}
printf("对多表的单表操作结束! \n");
}
/*-----*/
int main(void){
LinkList L=NULL;  int op=1,op2=1,i,j,e,pre,next;
while(op){
printf("\n\n");
printf("      Menu for Linear Table On Sequence Structure \n");
printf("-----\n");
printf("      1. InitList      10. ListInsert \n");
printf("      2. DestroyList   11. ListDelete \n");
printf("      3. ClearList     12. ListTraverse \n");
printf("      4. ListEmpty     13. reverseList \n");
printf("      5. ListLength    14. RemoveNthFromEnd \n");
printf("      6. GetElem       15. sortList \n");
printf("      7. LocateElem    16. File \n");
printf("      8. PriorElem     17. Lists\n");
printf("      9. NextElem      \n");
printf("      0. Exit\n");
printf("-----\n");
printf("      请选择你的操作[0~17]:");
scanf("%d",&op);
switch(op){
case 1:
//printf("\n----IntiList功能待实现! \n");

if(InitList(L)==OK) printf("线性表创建成功! \n");
else printf("线性表创建失败! \n");
getchar();getchar();
```

```
break;
case 2:
if(DestroyList(L)==INFEASIBLE) printf("线性表不存在! \n");
else printf("线性表销毁成功! \n");
getchar();getchar();
break;
case 3:
if(ClearList(L)==INFEASIBLE) printf("线性表不存在! \n");
else printf("线性表删除成功! \n");
getchar();getchar();
break;
case 4:
if(ListEmpty(L)==INFEASIBLE) printf("线性表不存在! \n");
else if(ListEmpty(L)==TRUE) printf("线性表判断为空! \n");
else printf("线性表判断不为空! \n");
getchar();getchar();
break;
case 5:
i=ListLength(L);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else printf("线性表长度为%d!\n",i);
getchar();getchar();
break;
case 6:
printf("请输入要获取的元素的序号: ");
scanf("%d",&j);
i=GetElem(L,j,e);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(i==ERROR) printf("序号不合法!");
else printf("线性表第%d个元素为%d! \n",j,e);
getchar();getchar();
```

```
break;
case 7:
printf("请输入要查找的元素: ");
scanf("%d",&e);
i=LocateElem(L,e);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(i==ERROR) printf("查找失败");
else printf("线性表指定元素%d的逻辑序号为%d! \n",e,i);
getchar();getchar();
break;
case 8:
printf("请输入要获取前驱元素的元素: ");
scanf("%d",&e);
i=PriorElem(L,e,pre);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(i==ERROR) printf("线性表中不存在指定元素%d的前驱元素! \n",e);
else printf("线性表中指定元素%d的前驱元素为%d! \n",e,pre);
getchar();getchar();
break;
case 9:
printf("请输入要获取后继元素的元素: ");
scanf("%d",&e);
i=NextElem(L,e,next);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(i==ERROR) printf("线性表中不存在指定元素%d的后继元素! \n",e);
else printf("线性表中指定元素%d的后继元素为%d! \n",e,next);
getchar();getchar();
break;
case 10:
printf("请输入要在线性表中插入元素的序号和要插入的元素: ");
scanf("%d%d",&j,&e);
```



```
i=ListInsert(L,j,e);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(i==ERROR) printf("线性表元素插入失败! \n");
else{
printf("线性表元素插入成功! \n新线性表为: ");
ListTraverse(L);
putchar('\n');
}
getchar();getchar();
break;
case 11:
printf("请输入要在线性表中删除的元素序号: ");
scanf("%d",&j);
i=ListDelete(L,j,e);
if(i==INFEASIBLE) printf("线性表不存在! \n");
else if(i==ERROR) printf("线性表元素删除失败! \n");
else{
printf("线性表元素删除成功! \n删除元素为: %d\n新线性表为: ",e);
ListTraverse(L);
putchar('\n');
}
getchar();getchar();
break;
case 12:
if(L==NULL) printf("线性表不存在! \n");
else ListTraverse(L);
getchar();getchar();
break;
case 13:
if(ListEmpty(L)!=INFEASIBLE){
L=reverseList(L);
```

```
printf("反转后的线性表为: ");
ListTraverse(L);putchar('\n');
}

else printf("线性表不存在! \n");

getchar();getchar();
break;
case 14:

printf("请输入n的值: ");
scanf("%d",&j);
i=RemoveNthFromEnd(L,j);
if(i!=OK) printf("删除失败! \n");
else{
printf("删除链表的倒数第%d个结点后的线性表为: ",j);
ListTraverse(L);
putchar('\n');
}
getchar();getchar();
break;
case 15:
sortList(L);
if(L==NULL) printf("线性表不存在! \n");
else{
printf("线性表元素排序成功! \n新线性表为: ");
ListTraverse(L);
putchar('\n');
}
getchar();getchar();
break;
```

```
case 16:
FILE *fp;
i=ListEmpty(L);
if(i==INFEASIBLE){
printf("线性表不存在！对空表进行读文件操作！读得线性表为：\n");
LoadList(L,"C:\\Users\\lenovo\\Desktop\\listfile.txt");
ListTraverse(L);
}
else{
printf("线性表判断存在！对线性表进行写文件操作！\n");
SaveList(L,"C:\\Users\\lenovo\\Desktop\\listfile.txt");
}
getchar();getchar();
break;
case 17:
LISTS Lists;
Lists.length=0;
while(op2){

printf("          Management Of Multiple Linear Tables  \n");
printf("-----\n");
printf("      1. AddList          3.LocateList \n");
printf("      2. RemoveList      0.Exit\n");
printf("-----\n");
printf("    请选择你的操作[0~3]:");
scanf("%d",&op2);
char name[30];
switch(op2){
case 1:
int n,e;
printf("请输入要添加的线性表的个数：");
```

```
scanf("%d",&n);
while(n--)
{
scanf("%s",name);
AddList(Lists,name);
scanf("%d",&e);
while (e)
{
ListInsert(Lists.elem[Lists.length-1].L,ListLength
(Lists.elem[Lists.length-1].L)+1,e);
scanf("%d",&e);
}
}
printf("添加后的多线性表为: ");
for(n=0;n<Lists.length;n++)
{
printf("%s ",Lists.elem[n].name);
ListTraverse(Lists.elem[n].L);
putchar('\n');
}
getchar();getchar();
break;
case 2:
printf("请输入要删除的线性表的名字: ");
scanf("%s",name);
if (RemoveList(Lists,name)==OK){
printf("删除后的线性表为: \n");
for(n=0;n<Lists.length;n++)
{
printf("%s ",Lists.elem[n].name);
ListTraverse(Lists.elem[n].L);
```

```
putchar('\n');
}
}
else printf("删除失败");
getchar();getchar();
break;
case 3:
printf("请输入要查找的线性表的名字: ");
scanf("%s",name);
if (n=LocateList(Lists,name))
{
printf("%s ",Lists.elem[n-1].name);
ListTraverse(Lists.elem[n-1].L);
printf("\n是否对所查到的线性表进行单表操作? (Y or N) \n");
char c;getchar();
scanf("%c",&c);
if(c=='Y') singleListOperation(Lists.elem[n-1].L);
putchar('\n');
}
else printf("查找失败");
getchar();getchar();
break;
case 0:
break;
}

}

case 0:
break;//end of switch
};//end of while
```

```
//end of main()  
printf("欢迎下次再使用本系统! \n");  
}
```

6 附录 C 基于二叉链表二叉树实现的源程序

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
#define REPEATED -3
#define MAX_BiTree_SIZE 1000
#define LIST_INIT_SIZE 50
#define NAMESIZE 50
#define LISTINCREMENT 10
typedef int status;
typedef int KeyType;
typedef struct {
    KeyType key;
    char others[20];
} TElemType; //二叉树结点类型定义
typedef struct BiTNode{ //二叉链表结点的定义
    TElemType data;
    struct BiTNode *lchild,*rchild;
} BiTNode, *BiTree;
typedef struct Stack{
    BiTree list[MAX_BiTree_SIZE];
    int top;
}Stack;
```

```
typedef struct{
char name[NAMESIZE];
BiTree root;
}NamedBiTree;//有名称二叉树的定义
typedef NamedBiTree ListElem;
typedef struct{ //多二叉树管理表定义
ListElem* elem;
int length;
int listsize;
}List;
//函数的声明
void visit(BiTree T);//访问结点
status keycheck(TElemType definition[]){
for(int i=0;definition[i].key!=-1;i++){
if(definition[i].key==0) continue;
for(int j=i+1;definition[j].key!=-1;j++){
if(definition[i].key==definition[j].key) return ERROR;
}
}
return OK;
}
status CreateBiTree1(BiTree &T, TElemType definition[])
{
if(keycheck(definition)==ERROR) return ERROR;
static int i = 0;
if(definition[0].key==0) return OK;
TElemType ch = definition[i];
if (ch.key == -1) {
T = NULL;
return OK;
}
```



```
T = (BiTree)malloc(sizeof(BiTNode));
if (!T) {
    exit(OVERFLOW);
}
T->data = ch;
T->lchild = NULL;
T->rchild = NULL;
i++;
if (definition[i].key != 0) {

    CreateBiTree1(T->lchild, definition);
}
i++;
if (definition[i].key != 0) {

    CreateBiTree1(T->rchild, definition);
}
return OK;
}

status CreateBiTree2(BiTree &T, TElemType preOrder[], TElemType inOrder[],
int preStart, int preEnd, int inStart, int inEnd) {
    if (preStart > preEnd || inStart > inEnd) { /
        / 当前序遍历序列或中序遍历序列为空时，返回空树
        T = NULL;
        return OK;
    }

    // 创建根节点，根据前序遍历序列的第一个元素确定
    T = (BiTree)malloc(sizeof(BiTNode));
    if (!T) {
```

```
exit(OVERFLOW);
}
T->data = preOrder[preStart];
T->lchild = NULL;
T->rchild = NULL;

// 在中序遍历序列中找到根节点的位置
int rootIndex;
for (rootIndex = inStart; rootIndex <= inEnd; rootIndex++) {
    if (inOrder[rootIndex].key == preOrder[preStart].key) {
        break;
    }
}

// 左子树的长度
int leftLength = rootIndex - inStart;

// 递归构建左子树
CreateBiTree2(T->lchild, preOrder, inOrder, preStart + 1, preStart +
leftLength, inStart, rootIndex - 1);

// 递归构建右子树
CreateBiTree2(T->rchild, preOrder, inOrder, preStart + leftLength +
1, preEnd, rootIndex + 1, inEnd);

return OK;
}

status CreateBiTree3(BiTree &T, TElemType inOrder[], TElemType postOrder[],
    int inStart, int inEnd, int postStart, int postEnd) {
    if (inStart == inEnd && postStart == postEnd) {
```

```
T = (BiTree)malloc(sizeof(BiTNode));
if (!T) {
    exit(OVERFLOW);
}
T->data = postOrder[postEnd];
// 当前节点为叶子节点，直接使用后序序列的最后一个元素作为节点数据
T->lchild = NULL;
T->rchild = NULL;
return OK;
}

if (inStart > inEnd || postStart > postEnd ||
    (inEnd - inStart) != (postEnd - postStart)) {
    T = NULL;
    return OK;
}

// 在后序序列中找到根节点
TElemType root = postOrder[postEnd];

// 在中序序列中找到根节点的位置
int rootIndex;
for (rootIndex = inStart; rootIndex <= inEnd; rootIndex++) {
    if (inOrder[rootIndex].key == root.key) {
        break;
    }
}

// 创建根节点
T = (BiTree)malloc(sizeof(BiTNode));
if (!T) {
```

```
exit(OVERFLOW);
}
T->data = root;
T->lchild = NULL;
T->rchild = NULL;

// 计算左子树和右子树的大小
int leftTreeSize = rootIndex - inStart;
int rightTreeSize = inEnd - rootIndex;

// 递归地构建左子树和右子树
if (leftTreeSize > 0) {
    CreateBiTree3(T->lchild, inOrder, postOrder, inStart,
        rootIndex - 1, postStart, postStart + leftTreeSize - 1);
}
if (rightTreeSize > 0) {
    CreateBiTree3(T->rchild, inOrder, postOrder, rootIndex + 1,
        inEnd, postStart + leftTreeSize, postEnd - 1);
}

return OK;
}

status CreateBiTree(BiTree &T,TElemType definition[]);
status isRepeat(TElemType definition[]);
status checkedCreateBiTree(BiTree &T,TElemType definition[], int &i);
status DestroyBiTree(BiTree &T);
status ClearBiTree(BiTree &T);
int BiTreeEmpty(BiTree T);
int BiTreeDepth(BiTree T);
BiTNode* LocateNode(BiTree T,KeyType e);
```

```
status Assign(BiTree &T,KeyType e,TElemType value);
BiTNode* GetSibling(BiTree T,KeyType e);
status InsertNode(BiTree &T,KeyType e,int LR,TElemType c);
BiTNode* FatherNode(BiTree &T, KeyType e, int &i);
status DeleteNode(BiTree &T, KeyType e);
status PreOrderTraverse(BiTree T,void (*visit)(BiTree));
status InOrderTraverse(BiTree T,void (*visit)(BiTree));
status PostOrderTraverse(BiTree T,void (*visit)(BiTree));
status LevelOrderTraverse(BiTree T, void (*visit)(BiTree));
void filePreTraverse(BiTree T, FILE *fp);
status SaveBiTree(BiTree T, char FileName[]);
void filePreCreateBiTree(BiTree &T, FILE *fp);
status LoadBiTree(BiTree &T, char FileName[]);
int MaxPathSum(BiTree &T);
status InvertTree(BiTree &T);
BiTNode* LowestCommonAncestor(BiTree &T, KeyType e1, KeyType e2);
status findAncestors(BiTree T, KeyType e, Stack &stack);
void SingleTree_Operation(BiTree &T);
status InitList(List &L);
status EnterTree(List &TreesList, char TreeName[NAMESIZE]);
status AddTree(List &TreesList, char TreeName[NAMESIZE]);
status RemoveTree(List &TreesList, char TreeName[NAMESIZE]);
int LocateTree(List &TreesList, char TreeName[NAMESIZE]);
//主函数
int main(void){
BiTree T = NULL;
int op=1, reValue, i;
char FileName[NAMESIZE], c, TreeName[NAMESIZE];
List TreesList;
InitList(TreesList);//创建二叉树管理表
void(*pvisit)(BiTree) = visit;//访问结点函数的指针
```

```
while(op){
system("cls");//清屏
printf("\n\n");
printf("    多二叉树管理菜单: \n");
printf("-----\n");
printf("1. SingleTree_Operation  2. EnterTreeByName 3. TreesTraverse\n");
printf("    4. ListLength      5. AddTree 6. RemoveTree\n");
printf("    7. LocateTreeByName    0. Exit\n");
printf("-----\n");
printf("    请选择你的操作[0~8]:");
scanf("%d",&op);
switch(op){
case 1:{
SingleTree_Operation(T);
break;
}
case 2:{
printf("请输入你需要单独操控的二叉树在管理表中的名称: ");
scanf("%s", &TreeName);
reValue = EnterTree(TreesList, TreeName);
if(reValue == ERROR) printf("不存在该二叉树! \n");
break;
}
case 3:{
if(TreesList.length==0) {
printf("管理表为空, 无二叉树可供遍历! \n");
break;
}
for(int i=0; i<TreesList.length; i++){
printf("\n序号为%d, 名称为%s的二叉树", i+1, TreesList.elem[i].name);
if(TreesList.elem[i].root==NULL){//空二叉树不予遍历
```

```
printf("为空树! ");
continue;
}

printf("\n前序遍历: ");PreOrderTraverse(TreesList.elem[i].root, pvisit);
printf("\n中序遍历: ");InOrderTraverse(TreesList.elem[i].root, pvisit);
printf("\n后序遍历: ");PostOrderTraverse(TreesList.elem[i].root, pvisit);
printf("\n层序遍历: ");LevelOrderTraverse(TreesList.elem[i].root, pvisit);
}

printf("\n所有二叉树遍历完成! \n");
break;
}

case 4:
printf("多二叉树管理表中有%d个二叉树。 \n", TreesList.length);
break;

case 5:
printf("请输入需要增加的二叉树的名称: ");
scanf("%s", TreeName);
reValue = AddTree(TreesList, TreeName); //增加二叉树
if(reValue == OK) printf("增加二叉树成功! \n");
else if(reValue == REPEATED) printf("已存在同名二叉树, 无法添加! \n");
else printf("增加二叉树出现错误! \n");
break;

case 6:
printf("请输入需要删除的二叉树的名称: ");
scanf("%s", TreeName);
reValue = RemoveTree(TreesList, TreeName);
if(reValue == OK) printf("删除二叉树成功! \n");
else printf("不存在该二叉树, 删除二叉树失败! \n");
break;

case 7:
printf("请输入需要查找的二叉树的名称: ");
```

```
scanf("%s", TreeName);
reValue = LocateTree(TreesList, TreeName);
if(reValue == 0)printf("所查找的二叉树不存在! \n");
else printf("名称为%s的二叉树在多线性表中是第%d个。 \n", TreeName, reValue);
break;
case 0:
break;
} //end of switch
getchar();getchar();//键入任意字符或回车结束循环
} //end of while
printf("欢迎下次再使用本系统! \n");
return 0;
}
```

//函数的定义

void visit(BiTree T)//访问结点

```
{
printf(" %d,%s",T->data.key,T->data.others);
}
```

status CreateBiTree(BiTree &T,TElemType definition[])

```
{
if(T) return INFEASIBLE;
int reValue = isRepeat(definition);
if(reValue) return ERROR;
int i = 0;
checkedCreateBiTree(T, definition, i);
return OK;
}
```

status isRepeat(TElemType definition[])// 查找是否存在重复关键字

```
{
int i = 0, j;
```



```
for(; definition[i].key!=-1; i++){
    if(definition[i].key==0)continue;// 关键字为0, 代表空结点, 直接跳过
    for(j=i+1; definition[j].key!=-1; j++){
        if(definition[j].key==definition[i].key)return TRUE;// 存在重复关键字
    }
}
return FALSE;
}

status checkedCreateBiTree(BiTree &T, TElemType definition[], int &i)
{
    if(definition[i].key==0){
        T = NULL;
        i++;
    }
    else{
        if(!(T = (BiTNode *)malloc(sizeof(BiTNode)))) exit(OVERFLOW);
        T->data.key = definition[i].key;
        strcpy(T->data.others, definition[i].others);
        i++;
        checkedCreateBiTree(T->lchild, definition, i);
        checkedCreateBiTree(T->rchild, definition, i);
    }
    return OK;
}

status DestroyBiTree(BiTree &T){
    if(T==NULL) return OK;
    DestroyBiTree(T->lchild);
    DestroyBiTree(T->rchild);
    free(T);
    return OK;
}
```

```
status ClearBiTree(BiTree &T){
    if(T == NULL){
        return OK;
    }
    ClearBiTree(T->lchild);
    ClearBiTree(T->rchild);
    free(T);
    T=NULL;
    return OK;
}

int BiTreeEmpty(BiTree T)
{
    if(T) return FALSE;
    else return TRUE;
}

int BiTreeDepth(BiTree T)
//求二叉树T的深度
{
    if(!T) return 0;
    int d=0,dr=0,dl=0;
    dl=BiTreeDepth(T->lchild);
    dr=BiTreeDepth(T->rchild);
    d=dl>dr?dl:dr;
    return d+1;
}

BiTNode* LocateNode(BiTree T,KeyType e){
    if(T==NULL) return NULL;
    if(T->data.key==e) return T;
    if(LocateNode(T->lchild,e)==NULL){
        if(LocateNode(T->rchild,e)!=NULL) return T->rchild;
```

```
}
else return T->lchild;
return NULL;
}

status Assign(BiTree &T,KeyType e,TElemType value)
//实现结点赋值。此题允许通过增加其它函数辅助实现本关任务
{
if(LocateNode(T,value.key)&&(value.key!=e)) return ERROR;
BiTree p=LocateNode(T,e);
if(p)
{
p->data=value;
return OK;
}
return ERROR;
}

BiTNode* GetSibling(BiTree T,KeyType e){
if(T == NULL || T->lchild == NULL || T->rchild == NULL) return NULL;
BiTree parent=T;
if(parent->lchild->data.key==e) return parent->rchild;
if(parent->rchild->data.key==e) return parent->lchild;
if(GetSibling(parent->lchild,e)==NULL){
if(GetSibling(parent->rchild,e)!=NULL) return GetSibling(parent->rchild,e);
}
else return GetSibling(parent->lchild,e);
return NULL;
}

status InsertNode(BiTree &T,KeyType e,int LR,TElemType c){
if (T == NULL)
{
BiTree neww = (BiTNode *)malloc(sizeof(BiTNode));
```

```
neww->data.key = c.key;
strcpy(neww->data.others, c.others);
neww->lchild = NULL;
neww->rchild = NULL;
T = neww;
return OK;
}
// 检查是否存在相同关键字的结点
if (LocateNode(T, c.key) != NULL)
{
return ERROR; // 存在相同关键字的结点, 返回ERROR
}
BiTree IT = LocateNode(T, e);
if (IT == NULL)
{
return ERROR;
}
BiTree neww = (BiTNode *)malloc(sizeof(BiTNode));
neww->data.key = c.key;
strcpy(neww->data.others, c.others);
neww->lchild = NULL;
neww->rchild = NULL;
if (LR == 0)
{
neww->rchild = IT->lchild;
IT->lchild = neww;
return OK;
}
else if (LR == 1)
{
neww->rchild = IT->rchild;
```

```
IT->rchild = neww;
```

```
return OK;
```

```
}
```

```
else if (LR == -1)
```

```
{
```

```
neww->rchild = T;
```

```
T = neww;
```

```
return OK;
```

```
}
```

```
return ERROR;
```

```
/****** End *****/
```

```
}
```

```
BiTNode* FatherNode(BiTree &T, KeyType e, int &i)// 寻找结点e的双亲结点
```

```
// i最后的值为0则结点e是左孩子, i最后的值为1则结点e是右孩子返回1, -1代表不存在
```

```
{
```

```
if(!T || T->data.key==e){// T为空树或根结点为e时, 结点e无双亲结点
```

```
i = -1;
```

```
return NULL;
```

```
}
```

```
// 左右孩子是否为结点e
```

```
if(T->lchild && T->lchild->data.key==e){
```

```
i = 0;
```

```
return T;
```

```
}
```

```
if(T->rchild && T->rchild->data.key==e){
```

```
i = 1;
```

```
return T;
```

```
}
```

```
// 左右孩子均不为结点e, 依次在左右子树中搜索
```

```
BiTNode* tmp;
tmp = FatherNode(T->lchild, e, i);
if(tmp) return tmp;
tmp = FatherNode(T->rchild, e, i);
if(tmp) return tmp;
//左右子树中均没有搜索到, 返回空指针
return NULL;
}

status DeleteNode(BiTree &T, KeyType e)//删除结点: 可以不查找父结点
{
    if(!T) return INFEASIBLE;
    BiTNode *tmp = LocateNode(T, e), *father;
    int i = -1;
    if(!tmp) return ERROR;// 没有找到关键字为e的结点, 无法删除 (包含空树)
    father = FatherNode(T, e, i);
    if(!tmp->lchild && !tmp->rchild){//
        if(i==-1 && T==tmp) T = NULL;//
        else if(father && i!=-1){
            if(i==0) father->lchild = NULL;
            else if(i==1) father->rchild = NULL;
        }
        free(tmp);
    }
    else if(tmp->lchild && tmp->rchild){//关键字为e的结点度为2
        BiTNode* LCrTree;
        if(T==tmp){
            T = tmp->lchild;
            LCrTree = tmp->lchild;
            while(LCrTree->rchild) LCrTree = LCrTree->rchild;
            LCrTree->rchild = tmp->rchild;
        }
    }
```

```
else if(father && i!=-1){//关键字为e的结点不是根结点

if(i==0) father->lchild = tmp->lchild;
else if(i==1) father->rchild = tmp->lchild;
LCrTree = tmp->lchild;
while(LCrTree->rchild) LCrTree = LCrTree->rchild;// 找到右孩子为空的结点
LCrTree->rchild = tmp->rchild;
}
free(tmp);
}
else if(tmp->lchild){// 左孩子结点非空，右孩子结点为空
if(T==tmp)// 关键字为e的结点是根结点
T = tmp->lchild;//用e的左子树代替被删除的e位置
else if(father && i!=-1){//关键字为e的结点不是根结点
if(i==0) father->lchild = tmp->lchild;
else if(i==1) father->rchild = tmp->lchild;
}
free(tmp);
}
else if(tmp->rchild){
if(T==tmp)
T = tmp->rchild;
else if(father && i!=-1){
if(i==0) father->lchild = tmp->rchild;
else if(i==1) father->rchild = tmp->rchild;
}
free(tmp);
}
return OK;
}
status PreOrderTraverse(BiTree T,void (*visit)(BiTree))//先序遍历二叉树T
```

```
{
if(!T) return OK;
visit(T);
PreOrderTraverse(T->lchild, visit);
PreOrderTraverse(T->rchild, visit);
return OK;
}

status InOrderTraverse(BiTree T,void (*visit)(BiTree))//中序遍历二叉树T
{
BiTree stack[MAX_BiTree_SIZE];
int top = 0;
stack[top++] = T;
while(top){
T = stack[top-1];
while(T){
T = T->lchild;
stack[top++] = T;
}
top--;
if(top){
T = stack[--top];
visit(T);
stack[top++] = T->rchild;
}
}
return OK;
}

status PostOrderTraverse(BiTree T,void (*visit)(BiTree))//后序遍历二叉树T
{
BiTree stack[MAX_BiTree_SIZE];
int top = 0;
```



```
stack[top++] = T;
while(top){
    T = stack[top-1];
    while(T){
        T = T->lchild;
        stack[top++] = T;
    }
    top--;
    if(top){
        T = stack[top-1];
        if(T->rchild==NULL){
            do{
                T = stack[--top];

            }while(top && (T == stack[top-1]->rchild));
        }
        if(top){
            stack[top] = stack[top-1]->rchild;
            top++;
        }
    }
    return OK;
}

status LevelOrderTraverse(BiTree T, void (*visit)(BiTree))
{
    BiTree Queue[MAX_BiTree_SIZE];
    int front=0, rear=0;
    if(T) Queue[rear++]=T;
    while(front != rear)
    {
```

```
T=Queue[front];
front=(front+1) % MAX_BiTree_SIZE;
visit(T);
if(T->lchild) {
Queue[rear]=T->lchild;
rear=(rear+1) % MAX_BiTree_SIZE;
}
if(T->rchild) {
Queue[rear]=T->rchild;
rear=(rear+1) % MAX_BiTree_SIZE;
}
}
return OK;
}

void filePreTraverse(BiTree T, FILE *fp)
{
if(T==NULL) fprintf(fp, "0,null ");
else {
fprintf(fp, "%d,%s ",T->data.key,T->data.others);
filePreTraverse(T->lchild, fp);
filePreTraverse(T->rchild, fp);
}
}

status SaveBiTree(BiTree T, char FileName[])
{
if(!T) return OK;
FILE * fp;
if((fp=fopen(FileName,"wb"))==NULL){
return ERROR;
}
filePreTraverse(T, fp);
```

```
fclose(fp);
return OK;
}

void filePreCreateBiTree(BiTree &T, FILE *fp)//基于先序遍历创建二叉树（文件）
{
    int tmpkey, reValue;
    char tmpOthers[50];
    reValue = fscanf(fp, "%d,%s ", &tmpkey, tmpOthers);// 读取数据
    if(reValue){
        if(tmpkey==0) T = NULL;
        else{
            if(!(T=(BiTNode *)malloc(sizeof(BiTNode)))) exit(OVERFLOW);//生成根结点
            T->data.key = tmpkey;//为根结点赋值
            strcpy(T->data.others, tmpOthers);
            filePreCreateBiTree(T->lchild, fp);//构造左子树
            filePreCreateBiTree(T->rchild, fp);//构造右子树
        }
    }
}

status LoadBiTree(BiTree &T, char FileName[])//读取文件中的二叉树
{
    if(T) return INFEASIBLE;//二叉树已存在
    FILE * fp;
    if((fp=fopen(FileName,"rb"))==NULL){
        return ERROR;// 文件打开失败
    }
    filePreCreateBiTree(T, fp);
    fclose(fp); // 关闭文件句柄
    return OK;// 读入完成
}

int MaxPathSum(BiTree &T)//返回根节点到叶子结点的最大路径和
```

```
{
if(!T) return 0;
int lSum = MaxPathSum(T->lchild), rSum = MaxPathSum(T->rchild);
if(lSum>rSum) return lSum+T->data.key;
else return rSum+T->data.key;
}

status InvertTree(BiTree &T)
{
if(!T) return OK;
BiTNode *tmp;
tmp = T->lchild;
T->lchild = T->rchild;
T->rchild = tmp;
InvertTree(T->lchild);
InvertTree(T->rchild);
return OK;
}

status findAncestors(BiTree T, KeyType e, Stack &stack)
{
stack.top = 0;
stack.list[stack.top++] = T;
while(stack.top){
T = stack.list[stack.top-1];
while(T){
T = T->lchild;
stack.list[stack.top++] = T;
}
stack.top--;
if(stack.top){
T = stack.list[stack.top-1];
if(T->rchild==NULL){
```

```
do{
T = stack.list[--stack.top];
if(T->data.key==e){
stack.list[stack.top++] = T;
return OK;
}

}while(stack.top && (T == stack.list[stack.top-1]->rchild));
}
if(stack.top){
stack.list[stack.top] = stack.list[stack.top-1]->rchild;
stack.top++;
}
}
return ERROR;
}

BiTNode* LowestCommonAncestor(BiTree &T, KeyType e1, KeyType e2)//最近公共祖先
{
Stack s1, s2;//为e1和e2定义两个指针栈
if(findAncestors(T, e1, s1)==ERROR || findAncestors(T, e2, s2)==ERROR)
return NULL;
int i = 0, top = (s1.top>s2.top)?s2.top:s1.top;
for(i=0;i<top && (s1.list[i]->data.key == s2.list[i]->data.key);i++);
return s2.list[i-1];
}

void SingleTree_Operation(BiTree &T){
int op=1, reValue, i;
KeyType e;
char FileName[NAMESIZE], c;
while(op){
```

华中科技大学课程实验报告

```
system("cls");//清屏
printf("\n\n");
printf("    单个二叉树操作菜单: \n");
printf("-----\n");
printf("    1. CreateBiTree 2. DestroyBiTree 3. ClearBiTree\n");
printf("    4. BiTreeEmpty 5. BiTreeDepth 6. LocateNode\n");
printf("    7. Assign 8. GetSibling 9. InsertNode\n");
printf("    10.DeleteNode 11.PreOrderTraverse 12.InOrderTraverse\n");
printf("    13.PostOrderTraverse 14.LevelOrderTraverse 15.MaxPathSum\n");
printf("    16.LowestCommonAncestor 17.InvertTree 18.SaveBiTree\n");
printf("    19.LoadBiTree 0. Return返\n");
printf("-----\n");
printf("    请选择你的操作[0~17]:");
scanf("%d",&op);
switch(op){
case 1:{
if(T) {
printf("二叉树非空, 请先销毁! \n");
break;
}
TElemType definition[100],preOrder[100],inOrder[100],postOrder[100];
int i = 0,j=0;
printf("请输入二叉树的定义(1): 前序遍历; (2) 前序+中序; (3) 后序+中序");
int op3,ans;
scanf("%d",&op3);
switch(op3){
case 1:{
do {
scanf("%d%s",&definition[i].key,definition[i].others);
} while (definition[i++].key!=-1);
ans=CreateBiTree1(T,definition);break;}
}
```

```
case 2:{
do {
scanf("%d%s",&preOrder[i].key,preOrder[i].others);
} while (preOrder[i++].key!=-1);
do {
scanf("%d%s",&inOrder[j].key,inOrder[j].others);
} while (inOrder[j++].key!=-1);
ans = CreateBiTree2(T, preOrder, inOrder, 0, i - 2, 0, j - 2);break;}
case 3:{
do {
scanf("%d%s",&postOrder[i].key,postOrder[i].others);
} while (postOrder[i++].key!=-1);
do {
scanf("%d%s",&inOrder[j].key,inOrder[j].others);
} while (inOrder[j++].key!=-1);
ans = CreateBiTree3(T, inOrder, postOrder, 0, i - 2, 0, j - 2);break;

}}
if (ans==OK) printf("二叉树创建成功");else printf("关键字不唯一");
getchar();getchar();
break;
}
case 2:
reValue = DestroyBiTree(T);
if(reValue == OK) printf("二叉树销毁成功! \n");
else if(reValue == INFEASIBLE) printf("二叉树为空! \n");
break;
case 3:
reValue = ClearBiTree(T);
if(reValue == OK) printf("二叉树销毁成功! \n");
else if(reValue == INFEASIBLE) printf("二叉树为空! \n");
```

```
break;
case 4:
reValue = BiTreeEmpty(T);
if(reValue == TRUE) printf("二叉树为空。\\n");
else if(reValue == FALSE) printf("二叉树非空。\\n");
break;
case 5:
reValue = BiTreeDepth(T);
if(reValue == 0) printf("二叉树为空，深度为0。\\n");
else printf("二叉树深度为%d。\\n", reValue);
break;
case 6:{
printf("请输入需要获取的结点的关键字：");
scanf("%d", &e);
BiTNode *pNode = LocateNode(T, e);
if(!pNode) printf("查找失败，二叉树中不存在关键字为%d的结点！\\n", e);
else printf("二叉树中关键字为%d的结点名称为%s。\\n", e, pNode->data.others);
break;
}
case 7:{
TElemType value;
printf("请输入需要赋值的结点的关键字：");
scanf("%d", &e);
printf("请输入需要赋值的结点的新关键字：");
scanf("%d", &value.key);
printf("请输入需要赋值的结点的新名称：");
scanf("%s", value.others);
reValue = Assign(T, e, value);
if(reValue == INFEASIBLE)
printf("二叉树为空，赋值失败！\\n");
else if(reValue == ERROR)
```



```
printf("二叉树中不存在关键字为%d的结点，赋值失败！\n", e);
else if(reValue == REPEATED)
printf("二叉树中已经存在关键字为%d的结点，赋值失败！\n", value.key);
else printf("赋值成功！\n");
break;
}
case 8:{
printf("请输入需要查找兄弟的结点关键字：");
scanf("%d", &e);
BiTNode *pNode = GetSibling(T,e);
if(!pNode) printf("查找失败，二叉树中不存在该元素的兄弟！\n");
else printf("二叉树中关键字为%d的兄弟结点是：名称为%s而关键字为%d的
结点。\\n", e, pNode->data.others, pNode->data.key);
break;
}
case 9:{
TElemType value;
int LR;
scanf("%d", &LR);
if(LR!=-1){
printf("请输入插入位置的结点关键字e：");
scanf("%d", &e);
}
else e=0;//LR为-1的时候，e可以取任何值(没有意义)，赋值为0
printf("请输入需要待插入的结点关键字：");
scanf("%d", &value.key);
printf("请输入需要待插入的结点名称：");
scanf("%s", value.others);
reValue = InsertNode(T, e, LR, value);
if(reValue == INFEASIBLE) printf("二叉树为空，插入失败！\\n");
else if(reValue == ERROR)
```

```
printf("二叉树中不存在关键字为%d的结点，插入失败！\n", e);
else if(reValue == REPEATED)
printf("二叉树中已经存在关键字为%d的结点，插入失败！\n", value.key);
else printf("插入成功！\n");
break;
}
case 10:
printf("请输入需要删除的结点关键字：");
scanf("%d", &e);
reValue = DeleteNode(T, e); //删除结点
if(reValue == INFEASIBLE)
printf("二叉树为空，删除失败！\n");
else if(reValue == ERROR)
printf("二叉树中不存在关键字为%d的结点，删除失败！\n", e);
else printf("删除成功！\n");
break;
case 11:{
if(!T) printf("二叉树为空！\n");
else {
printf("二叉树中全部结点先序遍历如下：\n");
PreOrderTraverse(T, visit); //先序遍历二叉树T
printf("遍历完成！\n");
}
break;
}
case 12:{
if(!T) printf("二叉树为空！\n");
else {
printf("二叉树中全部结点中序遍历如下：\n");
InOrderTraverse(T, visit); //中序遍历二叉树T
printf("遍历完成！\n");
```

```
}
break;
}
case 13:{
if(!T) printf("二叉树为空! \n");
else {
printf("二叉树中全部结点后序遍历如下: \n");
PostOrderTraverse(T, visit);//后序遍历二叉树T
printf("遍历完成! \n");
}
break;
}
case 14:{
if(!T) printf("二叉树为空! \n");
else {
printf("二叉树中全部结点层序遍历如下: \n");
LevelOrderTraverse(T, visit);//后序遍历二叉树T
printf("遍历完成! \n");
}
break;
}
case 15:{
if(!T)
printf("二叉树为空树! \n");
else//计算并输出根节点到叶子结点的最大路径和
printf("二叉树的根节点到叶子结点的最大路径和为%d", MaxPathSum(T));
break;
}
case 16:{
int e1, e2;
printf("请输入待查找公共祖先的两个结点的关键字（以空格间隔，以回车结尾）：");
```

```
scanf("%d %d", &e1, &e2);
if(!T){
printf("二叉树为空树! \n");
}
else{
BiTNode* tmp = LowestCommonAncestor(T, e1, e2); //查找最近公共祖先
if(!tmp){
printf("没有找到最近公共祖先! 结点不存在于二叉树中! \n");
break;
}
else printf("最近公共祖先是关键字为%d的结点,
名称为%s. \n", tmp->data.key, tmp->data.others);
}
break;
}
case 17:{
if(!T)
printf("二叉树为空树! \n");
else{
InvertTree(T); //所有节点的左右节点互换
printf("翻转成功! \n");
}
break;
}
case 18:{
if(!T) printf("二叉树为空树! \n");
else{

reValue = SaveBiTree(T, "C:\\Users\\lenovo\\Desktop\\listfile.txt");
if(reValue == ERROR) printf("文件打开失败! \n");
else {
```

```
printf("存储二叉树成功！\n");
}
}
break;
}
case 19:{

reValue = LoadBiTree(T, "C:\\Users\\lenovo\\Desktop\\listfile.txt");
if(reValue == INFEASIBLE) printf("二叉树已经存在，不可覆盖！\n");
else if(reValue == ERROR) printf("文件打开失败！\n");
else {
printf("读取二叉树成功！\n");
}
break;
}
case 0:
return;
} //end of switch
getchar();getchar();//键入任意字符清屏
} //end of while
return;
}
status InitList(List &L)//创建多线性表管理表
{
L.elem=(ListElem*)malloc(sizeof(ListElem)*LIST_INIT_SIZE);
L.listsize=LIST_INIT_SIZE;// 设置空间容量为默认初始化值
L.length=0;// 元素个数初始值为0
if (L.elem == NULL)
return OVERFLOW; // 内存分配失败
for(int i=0; i<LIST_INIT_SIZE; i++)
L.elem[i].root = NULL;// 设为空指针，避免野指针的出现导致非法访问
```

```
return OK;
}
status EnterTree(List &TreesList, char TreeName[NAMESIZE])//操控特定名称的二叉树
{
for(int i=0; i<TreesList.length; i++){
if(strcmp(TreeName, TreesList.elem[i].name)==0){
printf("你即将进入名称为%s的二叉树。\\n按回车键进入二叉树操控。\\n", TreeName);
getchar();getchar();
SingleTree_Operation(TreesList.elem[i].root);
return OK;
}
}
return ERROR;//没找到符合名称的二叉树
}
status AddTree(List &L, char ElemName[])// 在List尾部增加一个名称为ElemName的元素
{
if(LocateTree(L, ElemName))return REPEATED;
if(L.length >= L.listsize){// 溢出时扩充
ListElem *newelem = (ListElem*)realloc(L.elem,
(L.listsize+LISTINCREMENT)*sizeof(ListElem));
if(newelem==NULL)
return OVERFLOW;// 扩充失败
int i=L.listsize;
L.elem = newelem;
L.listsize += LISTINCREMENT;// 空间容量增加
for(i=0; i<LIST_INIT_SIZE; i++)
L.elem[i].root = NULL;// 设为空指针, 避免野指针的出现导致非法访问
return OK;
}
//复制字符串, 设置二叉树名称
strcpy(L.elem[L.length].name, ElemName);
```

```
L.elem[L.length].root = NULL;//空二叉树

printf("空树创建完成!是否输入先根遍历序列以继续创建? (是1, 否0) \n");
int tmp;
scanf("%d", &tmp);
if(!tmp){
L.length++;
return OK;
}
TElemType definition[100];
int i = 0;
for(i=0;i<100;i++){
definition[i].key = 0;//关键字置为0
}
i = 0;
do {
scanf("%d%s",&definition[i].key,definition[i].others);
} while (definition[i++].key!=-1);
int reValue = CreateBiTree(L.elem[L.length].root,definition);
L.length++;
if(reValue ==OK)printf("二叉树创建成功! \n");
else if(reValue == ERROR) printf("关键字不唯一! \n");
else if(reValue == INFEASIBLE) printf("二叉树非空, 请先销毁! \n");
return OK;
}

status RemoveTree(List &TreesList, char TreeName[NAMESIZE])//删除给定名称的二叉树
{
int len = TreesList.length, i, j, tmp;// 定义表长与循环变量
for (i = 0; i < len; i++) {
if (strcmp(TreesList.elem[i].name, TreeName) == 0) {// 比对
tmp = DestroyBiTree(TreesList.elem[i].root);
```

```
TreesList.elem[i].root = NULL;//根结点设为空指针
TreesList.elem[i].name[0] = '\0';// 清空二叉树名称
for (j = i; j < len; j++)// 将第i个元素和之后的元素前移
TreesList.elem[j] = TreesList.elem[j + 1];
TreesList.length--;// 管理表表长减一
return OK;
}
}
return ERROR;//未找到给定名称的二叉树
}

int LocateTree(List &ElemList, char ElemName[NAMESIZE])
{
    int i;
    for (i = 0; i < ElemList.length; i++) {
        if (strcmp(ElemList.elem[i].name, ElemName) == 0) {
            return i + 1;
        }
    }
    return 0;
}
```


7 附录 D 基于邻接表图实现的源程序

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
#define MAX_VERTEX_NUM 20
typedef int status;
typedef int KeyType;
typedef enum {DG,DN,UDG,UDN} GraphKind;
typedef struct {
    KeyType key;
    char others[20];
} VertexType; //顶点类型定义

typedef struct ArcNode {           //表结点类型定义
    int adjvex;                    //顶点位置编号
    struct ArcNode *nextarc;       //下一个表结点指针
} ArcNode;

typedef struct VNode{ //头结点及其数组类型定义
    VertexType data;              //顶点信息
    ArcNode *firstarc;            //指向第一条弧
} VNode,AdjList[MAX_VERTEX_NUM];
```

```
typedef struct { //邻接表的类型定义
AdjList vertices; //头结点数组
int vexnum,arcnum; //顶点数、弧数
GraphKind kind; //图的类型
} ALGraph;

status checksame(VertexType V[]){
int i=0;
int flag[10000]={0};
while(V[i].key!=-1)
{
if(flag[V[i].key]>0){return ERROR;}
flag[V[i].key]++;
i++;
}
return OK;
}

int locatei(ALGraph G,KeyType VR){
int i=0;
for(i;i<G.vexnum;i++)
{
if(G.vertices[i].data.key==VR) return i;
}
return -1;
}

status CreateCraph(ALGraph &G,VertexType V[],KeyType VR[][2]){
int i = 0;
while (V[i].key != -1) {
i++;
}
}
```

```
if(i==0) return ERROR;
if(i-1>=MAX_VERTEX_NUM) return ERROR;
if(checksame(V)==ERROR) return ERROR;
G.vexnum = i;
for (i = 0; i < G.vexnum; i++) {
    G.vertices[i].data = V[i];
    G.vertices[i].firstarc = NULL;
}
i = 0;
while(VR[i][0]!=-1)
{
    int a=locatei(G,VR[i][0]),b=locatei(G,VR[i][1]);
    if(!(a>=0&&b>=0)) return ERROR;
    ArcNode *q=(ArcNode*)malloc(sizeof(ArcNode)) ;
    q->adjvex=b;
    q->nextarc=G.vertices[a].firstarc;
    G.vertices[a].firstarc=q;
    q=(ArcNode*)malloc(sizeof(ArcNode)) ;
    q->adjvex=a;
    q->nextarc=G.vertices[b].firstarc;
    G.vertices[b].firstarc=q;
    i++;
}
G.arcnum = i;
return OK;
}
```

```
#define free free0
struct ptr{
    void *pused[100],*pfree[100];
```

```
int len_used,len_free;
} pm;
void free0(void *p){
pm.pfree[pm.len_free++]=p;
memset(p,0,sizeof(ArcNode));
free(p);
}

#undef free
status DestroyGraph(ALGraph &G)
/*销毁无向图G,删除G的全部顶点和边*/
{
ArcNode *p=NULL,*q=NULL;
for(int k=0;k<G.vexnum;k++)
{
if(G.vertices[k].firstarc!=NULL)
{
p=G.vertices[k].firstarc;
while(p!=NULL)
{
q=p->nextarc;
free(p);
p=q;
}
G.vertices[k].firstarc=NULL;
}
}
G.arcnum=0;
G.vexnum=0;
return OK;
}
```

```
int LocateVex(ALGraph G,KeyType u){
    int i=0;
    for(i;i<G.vexnum;i++)
    {
        if(G.vertices[i].data.key==u) return i;
    }
    return -1;
}
```

```
status PutVex(ALGraph &G,KeyType u,VertexType value){
    int i=LocateVex(G,u);
    if(i==-1) return ERROR;
    for(int k=0;k<G.vexnum;k++){
        if(G.vertices[k].data.key==value.key) return ERROR;
    }
    G.vertices[i].data=value;
    return OK;
}
```

```
int FirstAdjVex(ALGraph G,KeyType u){
    int i=0;
    for(i;i<G.vexnum;i++)
    {
        if(G.vertices[i].data.key==u) return G.vertices[i].firstarc->adjvex;
    }
    return -1;
}
```

```
int NextAdjVex(ALGraph G,KeyType v,KeyType w){
int i=LocateVex(G,v),j=LocateVex(G,w);
while(i!=-1&&j!=-1)
{
ArcNode *p=G.vertices[i].firstarc;
while(p->adjvex!=j&&p)
{
p=p->nextarc;
}
if(p->nextarc)
{
return p->nextarc->adjvex;
}
else return -1;
}
return -1;
}
```

```
status InsertVex(ALGraph &G,VertexType v){
if(LocateVex(G,v.key)!=-1) return ERROR;\
if(G.vexnum==MAX_VERTEX_NUM) return ERROR;
G.vertices[G.vexnum].data=v;
G.vertices[G.vexnum].firstarc=NULL;
G.vexnum++;
return OK;
}
```

```
status DeleteVex(ALGraph &G,KeyType v)
//在图G中删除关键字v对应的顶点以及相关的弧，成功返回OK,否则返回ERROR
{
```

```
// 请在这里补充代码，完成本关任务
/***** Begin *****/
if(G.vexnum==1 || G.vexnum==0) return ERROR;
int i=LocateVex(G,v);
if(i==-1) return ERROR;
ArcNode *p=G.vertices[i].firstarc,*q=NULL,*temp=NULL;
while(p){
    int j=p->adjvex;
    q=G.vertices[j].firstarc;
    if(q->adjvex==i){
        temp=q;
        G.vertices[j].firstarc=q->nextarc;
        free0(temp);
    }
    else{
        while(q->nextarc->adjvex!=i) q=q->nextarc;
        temp=q->nextarc;
        q->nextarc=temp->nextarc;
        free0(temp);
    }
    temp=p->nextarc;
    free0(p);
    p=temp;
    G.arcnum--;
}
for(int k=i;k<G.vexnum;k++)    G.vertices[k]=G.vertices[k+1];
G.vexnum--;
for(int k=0;k<G.vexnum;k++){
    p=G.vertices[k].firstarc;
    while(p!=NULL)
    {
```

```
if(p->adjvex>i)
p->adjvex--;
p=p->nextarc;
}
}
return OK;
/***** End *****/
}

int Get_index(ALGraph G, int key)
{
for (int i = 0; i < G.vexnum; i++)
if (G.vertices[i].data.key == key)
return i;
return -1;
}

void DeleteArc_A_Node(VNode& Node, int index)
{
ArcNode* p = Node.firstarc;
if (p == NULL)return;
if (p->adjvex == index)//首个链表节点就是
{
Node.firstarc = Node.firstarc->nextarc;
free(p);
p = NULL;
return;
}
ArcNode* q = Node.firstarc->nextarc;
for (; q != NULL;)
{
if (q->adjvex == index)
{
```



```
p->nextarc = q->nextarc;
free(q);
q = p->nextarc;
break;
}
else
{
p = p->nextarc;
q = q->nextarc;
}
}
}
```

```
status InsertArc(ALGraph& G, KeyType v, KeyType w)
//在图G中增加弧<v,w>, 成功返回OK, 否则返回ERROR
{
// 请在这里补充代码, 完成本关任务
/***** Begin *****/
int s1 = Get_index(G, v);
int s2 = Get_index(G, w);
if (s1 == -1 || s2 == -1) return ERROR;
ArcNode* q = G.vertices[s1].firstarc;
while (q) {
if (q->adjvex == s2) return ERROR;
q = q->nextarc;
}

ArcNode* p = (ArcNode*)malloc(sizeof(ArcNode));
p->adjvex = s2;
```

```
p->nextarc = G.vertices[s1].firstarc;
G.vertices[s1].firstarc = p;

p = (ArcNode*)malloc(sizeof(ArcNode));
p->adjvex = s1;
p->nextarc = G.vertices[s2].firstarc;
G.vertices[s2].firstarc = p;
++G.arcnum;
return OK;
/***** End *****/
}

status DeleteArc(ALGraph& G, KeyType v, KeyType w)
//在图G中删除弧<v,w>, 成功返回OK, 否则返回ERROR
{
    int s1 = Get_index(G, v);
    int s2 = Get_index(G, w);
    if (s1 == -1 || s2 == -1) return ERROR;
    ArcNode* q = G.vertices[s1].firstarc;
    int flag = 0;
    while (q) {
        if (q->adjvex == s2) flag = 1;
        q = q->nextarc;
    }
    if (flag == 0) return ERROR;
    DeleteArc_A_Node(G.vertices[s1], s2);
    DeleteArc_A_Node(G.vertices[s2], s1);
    --G.arcnum;
    return OK;
}
```

```
void visit(VertexType v)
{
printf(" %d %s",v.key,v.others);
}

status DFSTraverse(ALGraph &G,void (*visit)(VertexType)){
int flag[MAX_VERTEX_NUM]={0};
for(int i=0;i<G.vexnum;i++) flag[i]=0;
for(int i=0;i<G.vexnum;i++){
if(!flag[i]){
visit(G.vertices[i].data);
flag[i]++;
ArcNode *p=G.vertices[i].firstarc;
while(p){
int k=LocateVex(G,G.vertices[p->adjvex].data.key);
if(k==-1) break;
else if(flag[k]!=0){
if(p->nextarc==NULL) break;
p=p->nextarc;
}
else{
i=k;
visit(G.vertices[i].data);
flag[i]++;
p=G.vertices[i].firstarc;
}
}
}
else continue;
}
return OK;
```

```
}
```

```
typedef struct QNode{  
VertexType data;  
struct QNode *next;  
}QNode,*Queue;  
typedef struct{  
Queue front;  
Queue rear;  
}Linkqueue;
```

```
status BFSTraverse(ALGraph &G,void (*visit)(VertexType)){  
int flag[MAX_VERTEX_NUM];  
int i;  
for (i = 0; i < G.vexnum; i++) {  
flag[i] = 0; //初始化所有顶点为未访问状态  
}  
for (i = 0; i < G.vexnum; i++) {  
if (!flag[i]) {  
flag[i] = 1;  
visit(G.vertices[i].data); //访问顶点  
ArcNode *p=G.vertices[i].firstarc;  
while(p){  
int k=LocateVex(G,G.vertices[p->adjvex].data.key);  
if(k==-1) break;  
else if(flag[k]!=0){  
if(p->nextarc==NULL) break;  
p=p->nextarc;  
}  
else{  
visit(G.vertices[k].data);
```

```
flag[k]++;
p=p->nextarc;
}
}
}
}
return OK;
}

status InitQueue(Linkqueue &Q)
{
Q.front=Q.rear=(QNode *)malloc(sizeof(QNode));
if(!Q.front)return ERROR;
Q.front->next=NULL;
return OK;
}

status QueueEmpty(Linkqueue Q)
{
if(Q.front==Q.rear)return TRUE;
else return FALSE;
}

status enqueue(Linkqueue &Q,VertexType value)
{
Queue p=(Queue)malloc(sizeof(QNode));
if(!p)return ERROR;
p->data=value;
p->next=NULL;
Q.rear->next=p;
Q.rear=p;
return OK;
}

status dequeue(Linkqueue &Q,VertexType &value)
```

```
{
if(Q.front==Q.rear)return ERROR;
Queue p=Q.front->next;
value=p->data;
Q.front->next=p->next;
if(Q.rear==p)
Q.rear=Q.front;
free(p);
return OK;
}

int distance[21];
status ShortestPathLength(ALGraph G,KeyType v,KeyType w){
int i,j,n;
VertexType top;
top.key=v;
Linkqueue Q;
InitQueue(Q);
for(i=0;i<G.vexnum;i++)
distance[G.vertices[i].data.key]=20;
distance[v]=0;
int k=LocateVex(G,v);
enqueue(Q,G.vertices[k].data);
while(!QueueEmpty(Q))
{
deQueue(Q,top);
if(top.key==w)break;
for(j=FirstAdjVex(G,top.key);j>=0;j=NextAdjVex(G,top.key,G.vertices[j].data.key))/
{
if(distance[G.vertices[j].data.key]==20)
{
distance[G.vertices[j].data.key]=distance[top.key]+1;
```

```
enqueue(Q,G.vertices[j].data);
}
}
}
n=distance[w];
return n;
}
```

```
void VerticesSetLessThan(ALGraph G,KeyType v,KeyType k){
int i,j;
for(i=0;i<G.vexnum;i++)
{
j=ShortestPathLength(G,v,G.vertices[i].data.key);
if(j<k&&G.vertices[i].data.key!=v)
visit(G.vertices[i].data);
}
}
```

```
bool mark[20];
bool visited[MAX_VERTEX_NUM];
void dfs(ALGraph &G,KeyType v)
{
mark[v]=TRUE;
for(int w=FirstAdjVex(G,v);w>=0;w=NextAdjVex(G,v,G.vertices[w].data.key))
{
if(!mark[G.vertices[w].data.key])dfs(G,G.vertices[w].data.key);
}
}

status ConnectedComponentsNums(ALGraph G)//求图中连通分量个数
{
int count=0,i;
```

```
for(i=0;i<G.vexnum;i++)
mark[G.vertices[i].data.key]=FALSE;//标记数组记录关键字
for(i=0;i<G.vexnum;i++)
{
if(!mark[G.vertices[i].data.key])
{
dfs(G,G.vertices[i].data.key);
count++;
}
}
return count;
}
```

```
status SaveGraph(ALGraph G, char FileName[]){
FILE *fp;
fp = fopen(FileName, "w");
if (fp == NULL||G.vexnum==0) return ERROR;
for(int i=0;i<G.vexnum;i++){
fprintf(fp,"%d %s ",G.vertices[i].data.key,G.vertices[i].data.others);
ArcNode *p=G.vertices[i].firstarc;
while(p){
fprintf(fp,"%d ",p->adjvex);
p=p->nextarc;
}
fprintf(fp,"%d\n",-1);
}
fclose(fp);
return OK;
}
```



```
}
```

```
status LoadGraph(ALGraph &G, char FileName[]){
FILE *fp;
fp = fopen(FileName, "r");
if (fp == NULL) return ERROR;
VNode tempNode;
int i = 0;
G.arcnum = 0;
G.vexnum = 0;
while (fscanf(fp, "%d %s", &tempNode.data.key, tempNode.data.others) != EOF) {
G.vertices[i].data = tempNode.data;
G.vertices[i].firstarc = NULL;

ArcNode *p, *q;
int j = 0;
while (fscanf(fp, "%d", &j) != EOF && j != -1) {
p = (ArcNode*)malloc(sizeof(ArcNode));
p->adjvex = j;
p->nextarc = NULL;

if (G.vertices[i].firstarc == NULL) {
G.vertices[i].firstarc = p;
} else {
q = G.vertices[i].firstarc;
while (q->nextarc != NULL) {
q = q->nextarc;
}
q->nextarc = p;
}
}
```

```
G.arcnum++;
```

```
}
```

```
i++;
```

```
G.vexnum++;
```

```
}
```

```
G.arcnum = G.arcnum / 2;
```

```
fclose(fp);
```

```
return OK;
```

```
}
```

```
typedef struct{ //线性表的管理表定义
```

```
struct { char name[30];
```

```
ALGraph G;
```

```
} elem[10];
```

```
int length;
```

```
int listsize;
```

```
}LISTS;
```

```
status Addg(LISTS &Lists, char ListName[])
```

```
{
```

```
strcpy(Lists.elem[Lists.length].name, ListName); // 复制名称
```

```
Lists.length++;
```

```
return OK;
```

```
}
```

```
status Removeg(LISTS &Lists,char ListName[])
```

```
// Lists中删除一个名称为ListName的线性表
```

```
{
```

```
for(int i=0;i<Lists.length;i++){
```

```
if(strcmp(ListName,Lists.elem[i].name)==0){
```

```
for(int j=i; j<Lists.length-1; j++){
    strcpy(Lists.elem[j].name, Lists.elem[j+1].name);
    Lists.elem[j].G = Lists.elem[j+1].G;
}
Lists.length--;
return OK;
}
}
return ERROR;
}
```

```
int Locateg(LISTS Lists,char ListName[])
// 在Lists中查找一个名称为ListName的线性表，成功返回逻辑序号，否则返回0
{
    for(int i=0;i<Lists.length;i++){
        if(strcmp(ListName,Lists.elem[i].name)==0)
            return i+1;
    }
    return 0;
}

void singleGraphOperation(ALGraph &G){
    VertexType V[30];
    KeyType VR[100][2];
    int op=1,op2=1,i,j,u,e,pre,next;
    while(op){
        printf("\n\n");
        printf("      Menu for Linear Table On Sequence Structure \n");
        printf("-----\n");
        printf("      1. CreateCraph      10. DeleteArc \n");
        printf("      2. DestroyGraph     11. DFSTraverse \n");
```

```
printf("      3. LocateVex      12. BFSTraverse \n");
printf("      4. PutVex        13. VerticesSetLessThank \n");
printf("      5. FirstAdjVex      14. ShortestPathLength \n");
printf("      6. NextAdjVex      15. ConnectedComponentsNums \n");
printf("      7. InsertVex        16. File \n");
printf("      8. DeleteVex        17. Lists\n");
printf("      9. InsertArc        \n");
printf("      0. Exit\n");
printf("-----\n");
printf("      请选择你的操作[0~17]:");
scanf("%d",&op);
switch(op){
case 1:{
int i=0,j;
do {
scanf("%d%s",&V[i].key,V[i].others);
} while(V[i++].key!=-1);
i=0;
do {
scanf("%d%d",&VR[i][0],&VR[i][1]);
} while(VR[i++][0]!=-1);
if (CreateCraph(G,V,VR)==ERROR) printf("输入数据错，无法创建");
else
{
if (G.arcnum!=i-1) {
printf("边的数目错误！ \n");
}
for(j=0;j<G.vexnum;j++){
ArcNode *p=G.vertices[j].firstarc;
printf("%d %s",G.vertices[j].data.key,G.vertices[j].data.others);
while (p)
```

```
{
printf(" %d",p->adjvex);
p=p->nextarc;
}
printf("\n");}
}
getchar();getchar();
break;}
case 2:{
if(G.vexnum==0)printf("销毁失败，请先创建图！\n");
else{
if(DestroyGraph(G)==OK)printf("销毁成功！\n");
else printf("销毁失败，请先创建图！\n");
}
getchar();getchar();
break;}
case 3:{
printf("请输入关键字！\n");
scanf("%d",&u);
i=LocateVex(G,u);
if (i!=-1) printf("%d %s",G.vertices[i].data.key,G.vertices[i].data.others);
else printf("查找失败");
getchar();getchar();
break;}
case 4:{
printf("请输入关键字和赋值结点！\n");
scanf("%d",&u);
VertexType V[21],value;
scanf("%d%s",&value.key,value.others);
i=PutVex(G,u,value);
if (i==OK)
```

```
for(i=0;i<G.vexnum;i++)
printf(" %d %s",G.vertices[i].data.key,G.vertices[i].data.others);
else printf("赋值操作失败");
getchar();getchar();
break;}
case 5:{
printf("请输入关键字! \n");
scanf("%d",&u);
i=FirstAdjVex(G,u);
if (i!=-1) printf("%d %s",G.vertices[i].data.key,G.vertices[i].data.others);
else printf("查找失败");
getchar();getchar();
break;}
case 6:{
KeyType v,w;
if(G.vexnum==0)printf("图不存在! \n");
else{
printf("请输入要查找顶点和相对顶点的关键字! \n");
scanf("%d%d",&v,&w);
if(NextAdjVex(G,v,w)==-1)printf("查找失败! \n");
else {
int x=NextAdjVex(G,v,w);
printf("顶点%d的邻接顶点相对于%d的下一邻接顶点的位序是%d\n",v,w,x);
printf("其值为 %d %s\n",G.vertices[x].data.key,G.vertices[x].data.others);}}
getchar();getchar();
break;}
case 7:{
VertexType v;
printf("请输入新增的顶点! \n");
scanf("%d%s",&v.key,v.others);
i=InsertVex(G,v);
```

```
if (i==OK)
for(i=0;i<G.vexnum;i++)
{
ArcNode *p=G.vertices[i].firstarc;
printf("%d %s",G.vertices[i].data.key,G.vertices[i].data.others);
while (p)
{
printf(" %d",p->adjvex);
p=p->nextarc;
}
printf("\n");
}
else printf("插入操作失败");
getchar();getchar();
break;}
case 8:{
if(G.vexnum==0)printf("图不存在! \n");
else{
KeyType v;
printf("请输入要删除顶点的关键字! \n");
scanf("%d",&v);
if(DeleteVex(G,v)==OK)printf("删除成功! \n");
else printf("删除失败! \n");}
getchar();getchar();
break;}
case 9:{
KeyType v,w,arcs;
if(G.vexnum==0)printf("图不存在! \n");
else{
printf("请输入要插入弧对应的两个顶点! \n");
scanf("%d%d",&v,&w);
```

```
if(InsertArc(G,v,w)==OK) printf("插入弧成功！ \n");
else printf("插入弧失败！ \n");
}
getchar();getchar();
break;}
case 10:{
KeyType v,w;
printf("请输入关键词： \n");
int uu, vv;
scanf("%d%d", &uu, &vv);
int ans = DeleteArc(G, uu, vv);
if (ans == ERROR) printf("删除失败！ \n");
else printf("删除成功！ \n");

getchar();getchar();
break;}
case 11:{

if(G.vexnum==0)printf("图不存在！ \n");
else
DFSTraverse(G,visit);
getchar();getchar();
break;}
case 12:{
if(G.vexnum==0)printf("图不存在！ \n");
else
BFSTraverse(G,visit);
getchar();getchar();
break;}
case 13:{KeyType v,k;
printf("请输入顶点的关键字及查找距离！ \n");
```



```
scanf("%d%d",&v,&k);
printf("与顶点距离小于%d的顶点有\n",k);
VerticesSetLessThank(G,v,k);
getchar();getchar();
break;
}
case 14:
{KeyType v,w,k;
printf("请输入两个顶点的关键字!\n");
scanf("%d%d",&v,&w);
k=ShortestPathLength(G,v,w);
if(k!=20){
printf("这两个顶点间的最短路径是%d",k);
}
else
printf("这两个顶点之间不存在路! \n");
getchar();getchar();
break;
}
case 15:{
printf("图的连通分量有%d个! \n",ConnectedComponentsNums(G));
getchar();getchar();
break;}
case 16:{
FILE *fp;
ALGraph G1;
SaveGraph(G,"C:\\Users\\lenovo\\Desktop\\listfile.txt");
printf("已将图存储到文件中");
printf("从文件中读得G1为: \n");
LoadGraph(G1,"C:\\Users\\lenovo\\Desktop\\listfile.txt");
for(j=0;j<G1.vexnum;j++){
```

```
ArcNode *p=G1.vertices[j].firstarc;
printf("%d %s",G1.vertices[j].data.key,G1.vertices[j].data.others);
while (p)
{
printf(" %d",p->adjvex);
p=p->nextarc;
}
printf("\n");}

getchar();getchar();
break;}
case 0:{
break;
}
}
}
printf("对多图的单图操作结束! \n");
}

/*-----*/
int main(void){
ALGraph G;
VertexType V[30];
KeyType VR[100][2];
int op=1,op2=1,i,j,u,e,pre,next;
int flag=1;
while(op){
printf("\n\n");
printf("      Menu for Linear Table On Sequence Structure \n");
printf("-----\n");
printf("      1. CreateCraph      10. DeleteArc \n");
```

```
printf("      2. DestroyGraph      11. DFSTraverse \n");
printf("      3. LocateVex         12. BFSTraverse \n");
printf("      4. PutVex                13. VerticesSetLessThank \n");
printf("      5. FirstAdjVex           14. ShortestPathLength \n");
printf("      6. NextAdjVex            15. ConnectedComponentsNums \n");
printf("      7. InsertVex             16. File \n");
printf("      8. DeleteVex             17. Lists\n");
printf("      9. InsertArc             \n");
printf("      0. Exit\n");

printf("-----\n");
printf("    请选择你的操作[0~17]:");
scanf("%d",&op);
switch(op){
case 1:{
int i=0,j;
do {
scanf("%d%s",&V[i].key,V[i].others);
} while(V[i++].key!=-1);
i=0;
do {
scanf("%d%d",&VR[i][0],&VR[i][1]);
} while(VR[i++][0]!=-1);
if (CreateCraph(G,V,VR)==ERROR) printf("输入数据错，无法创建");
else
{
if (G.arcnum!=i-1) {
printf("边的数目错误! \n");
}
for(j=0;j<G.vexnum;j++){
ArcNode *p=G.vertices[j].firstarc;
printf("%d %s",G.vertices[j].data.key,G.vertices[j].data.others);
```

```
while (p)
{
printf(" %d",p->adjvex);
p=p->nextarc;
}
printf("\n");}
}
getchar();getchar();
break;}
case 2:{
if(G.vexnum==0)printf("销毁失败, 请先创建图! \n");
else{
if(DestroyGraph(G)==OK)printf("销毁成功! \n");
else printf("销毁失败, 请先创建图! \n");
}
getchar();getchar();
break;}
case 3:{
printf("请输入关键字! \n");
scanf("%d",&u);
i=LocateVex(G,u);
if (i!=-1) printf("%d %s",G.vertices[i].data.key,G.vertices[i].data.others);
else printf("查找失败");
getchar();getchar();
break;}
case 4:{
printf("请输入关键字和赋值结点! \n");
scanf("%d",&u);
VertexType V[21],value;
scanf("%d%s",&value.key,value.others);
i=PutVex(G,u,value);
```

```
if (i==OK)
for(i=0;i<G.vexnum;i++)
printf(" %d %s",G.vertices[i].data.key,G.vertices[i].data.others);
else printf("赋值操作失败");
getchar();getchar();
break;}
case 5:{
printf("请输入关键字! \n");
scanf("%d",&u);
i=FirstAdjVex(G,u);
if (i!=-1) printf("%d %s",G.vertices[i].data.key,G.vertices[i].data.others);
else printf("查找失败");
getchar();getchar();
break;}
case 6:{
KeyType v,w;
if(G.vexnum==0)printf("图不存在! \n");
else{
printf("请输入要查找顶点和相对顶点的关键字! \n");
scanf("%d%d",&v,&w);
if(NextAdjVex(G,v,w)==-1)printf("查找失败! \n");
else {
int x=NextAdjVex(G,v,w);
printf("顶点%d的邻接顶点相对于%d的下一邻接顶点的位序是%d\n",v,w,x);
printf("其值为 %d %s\n",G.vertices[x].data.key,G.vertices[x].data.others);}}
getchar();getchar();
break;}
case 7:{
VertexType v;
printf("请输入新增的顶点! \n");
scanf("%d%s",&v.key,v.others);
```

```
i=InsertVex(G,v);
if (i==OK)
for(i=0;i<G.vexnum;i++)
{
ArcNode *p=G.vertices[i].firstarc;
printf("%d %s",G.vertices[i].data.key,G.vertices[i].data.others);
while (p)
{
printf(" %d",p->adjvex);
p=p->nextarc;
}
printf("\n");
}
else printf("插入操作失败");
getchar();getchar();
break;}
case 8:{
if(G.vexnum==0)printf("图不存在! \n");
else{
KeyType v;
printf("请输入要删除顶点的关键字! \n");
scanf("%d",&v);
if(DeleteVex(G,v)==OK)printf("删除成功! \n");
else printf("删除失败! \n");}
getchar();getchar();
break;}
case 9:{
KeyType v,w,arcs;
if(G.vexnum==0)printf("图不存在! \n");
else{
printf("请输入要插入弧对应的两个顶点! \n");
```

```
scanf("%d%d",&v,&w);
if(InsertArc(G,v,w)==OK)printf("插入弧成功! \n");
else printf("插入弧失败! \n");
}
getchar();getchar();
break;}
case 10:{
KeyType v,w;
flag++;
if(G.vexnum==0)printf("图不存在! \n");
else{
printf("请输入要删除弧对应的两个顶点! \n");
scanf("%d%d",&v,&w);
if(DeleteArc(G,v,w)==OK)printf("删除弧成功! \n");
else printf("删除弧失败! \n");
}
getchar();getchar();
break;}
case 11:{

if(G.vexnum==0)printf("图不存在! \n");
else
DFSTraverse(G,visit);
getchar();getchar();
break;}
case 12:{
if(G.vexnum==0)printf("图不存在! \n");
else
BFSTraverse(G,visit);
getchar();getchar();
break;}
```

```
case 13:{KeyType v,k;
printf("请输入顶点的关键字及查找距离!\n");
scanf("%d%d",&v,&k);
printf("与顶点距离小于%d的顶点有\n",k);
VerticesSetLessThank(G,v,k);
getchar();getchar();
break;
}
case 14:
{KeyType v,w,k;
printf("请输入两个顶点的关键字!\n");
scanf("%d%d",&v,&w);
k=ShortestPathLength(G,v,w);
if(k!=20){
printf("这两个顶点间的最短路径是%d",k);
}
else
printf("这两个顶点之间不存在路!\n");
getchar();getchar();
break;
}
case 15:{
printf("图的连通分量有%d个!\n",ConnectedComponentsNums(G));

getchar();getchar();
break;}
case 16:{
FILE *fp;
ALGraph G1;
SaveGraph(G,"C:\\Users\\lenovo\\Desktop\\listfile.txt");
printf("已将图存储到文件中");
```



```
printf("从文件中读得G1为: \n");
LoadGraph(G1,"C:\\Users\\lenovo\\Desktop\\listfile.txt");
for(j=0;j<G1.vexnum;j++){
ArcNode *p=G1.vertices[j].firstarc;
printf("%d %s",G1.vertices[j].data.key,G1.vertices[j].data.others);
while (p)
{
printf(" %d",p->adjvex);
p=p->nextarc;
}
printf("\n");}

getchar();getchar();
break;}

case 17:
LISTS Lists;
Lists.length=0;
int flag[100];
for(int i=0;i<100;i++){
flag[i]=0;
}
while(op2){

printf("          Management Of Multiple Linear Tables  \n");
printf("-----\n");
printf("      1. AddList          3.LocateList \n");
printf("      2. RemoveList       0.Exit\n");
printf("-----\n");
printf("      请选择你的操作 [0~3]:");

scanf("%d",&op2);
```

```
char name[30];
switch(op2){

case 1:{
int n,e;
printf("请输入要添加的图的个数: ");
scanf("%d",&n);
while(n-->0)
{
scanf("%s",name);
Addg(Lists,name);
VertexType V[30];
KeyType VR[100][2];
int i=0,j;
do {
scanf("%d%s",&V[i].key,V[i].others);
} while(V[i++].key!=-1);
i=0;
do {
scanf("%d%d",&VR[i][0],&VR[i][1]);
} while(VR[i++][0]!=-1);
if (CreateCraph(Lists.elem[List.length-1].G,V,VR)==ERROR) flag[List.length-1]=1;

}

printf("添加后的多图表为: ");
for(n=0;n<Lists.length;n++)
{
printf("%s ",Lists.elem[n].name);
if(flag[n]==1) printf("输入数据错误\n");
else
```

```
for(j=0;j<Lists.elem[n].G.vexnum;j++)
{

ArcNode *p=Lists.elem[n].G.vertices[j].firstarc;
printf("%d %s",Lists.elem[n].G.vertices[j].data.key,Lists.elem[n].G.vertices[j].da
while (p)
{
printf(" %d",p->adjvex);
p=p->nextarc;
}

putchar('\n');
}}
getchar();getchar();
break;}
case 2:
printf("请输入要删除的线性表的名字: ");
scanf("%s",name);
if (Removeg(Lists,name)==OK){
printf("删除后的线性表为: \n");
for(int n=0;n<Lists.length;n++)
{
printf("%s ",Lists.elem[n].name);
if(flag[i]==1) printf("输入数据错误");
else
for(j=0;j<Lists.elem[n].G.vexnum;j++)
{

ArcNode *p=Lists.elem[n].G.vertices[j].firstarc;
printf("%d %s",Lists.elem[n].G.vertices[j].data.key,Lists.elem[n].G.vertices[j].da
while (p)
```

```
{
printf(" %d",p->adjvex);
p=p->nextarc;
}

putchar('\n');
}}
}
else printf("删除失败");

getchar();getchar();
break;
case 3:{
printf("请输入要查找的图的名字: ");
scanf("%s",name);
int n=Locateg(Lists,name);
if (n)
{
printf("%s ",Lists.elem[n-1].name);
if(flag[n-1]==1) printf("输入数据错误");
else
for(j=0;j<Lists.elem[n-1].G.vexnum;j++)
{

ArcNode *p=Lists.elem[n-1].G.vertices[j].firstarc;
printf("%d %s",Lists.elem[n-1].G.vertices[j].data.key,
Lists.elem[n-1].G.vertices[j].data.others);
while (p)
{
printf(" %d",p->adjvex);
p=p->nextarc;
```

```
}
putchar('\n');
}
printf("\n是否对所查到的线性表进行单表操作? (Y or N) \n");
char c;getchar();
scanf("%c",&c);
if(c=='Y') singleGraghOperation(Lists.elem[n-1].G);
putchar('\n');
}
else printf("查找失败");
getchar();getchar();
break;}
case 0:
break;
}

}

case 0:
break;//end of switch
};//end of while

}
printf("欢迎下次再使用本系统! \n");
}
```