

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 CS2206

学 号 U202215525

姓 名 冯子潇

指导教师 杨中

报告日期 2023 年 6 月 9 日

计算机科学与技术学院

目 录

1	第一章：基于顺序存储结构的线性表实现.....	1
1.1	问题描述	1
1.2	系统设计	4
1.3	系统实现	8
1.4	系统测试	16
1.5	实验小结	25
2	第二章：基于二叉链表的二叉树实现	26
2.1	问题描述	26
2.2	系统设计	29
2.3	系统实现	31
2.4	系统测试	34
2.5	实验小结	43
3	课程的收获和建议	45
3.1	基于顺序存储结构的线性表实现	45
3.2	基于链式存储结构的线性表实现	45
3.3	基于二叉链表的二叉树实现	45
3.4	基于邻接表图实现的源程序	46
	参考文献	47
	附录 A 基于顺序存储结构线性表实现的源程序	47
	附录 B 基于链式存储结构线性表实现的源程序	71
	附录 C 基于二叉链表二叉树实现的源程序	96
	附录 D 基于邻接表图实现的源程序	123

1 第一章：基于顺序存储结构的线性表实现

1.1 问题描述

1.1.1 希望通过实验达到：

- 加深对线性表的概念、基本运算的理解；
- 熟练掌握线性表的逻辑结构与物理结构的关系；
- 物理结构采用顺序表, 熟练掌握线性表的基本运算的实现。

1.1.2 具体实验目标：

依据最小完备性和常用性相结合的原则，以函数形式定义了线性表的 20 种基本运算，具体运算功能定义如下。

- 线性表的创建：

函数原型: `status InitList(Sqlist L);`

功能说明：如果线性表 L 不存在，操作结果是构造一个空的线性表，返回 OK，否则返回 INFEASIBLE。

- 销毁线性表：

函数原型: `status DestroyList(Sqlist L);`

功能说明：如果线性表 L 存在，该操作释放线性表的空间，使线性表成为未初始化状态，返回 OK；否则对于一个未初始的线性表，是不能进行销毁操作的，返回 INFEASIBLE。

- 清空线性表：

函数原型: `status ClearList(Sqlist L);`

功能说明：若线性表 L 不存在，返回 INFEASIBLE。否则清空线性表 L，返回 OK。

- 线性表判空：

函数原型: `status ListEmpty(Sqlist L);`

功能说明：若线性表 L 不存在，则返回 INFEASIBLE；若线性表 L 长度为

0, 则返回 TRUE; 不为 0 则返回 FALSE。

- 线性表长度:

函数原型: `status ListLength(SqList L);`

功能说明: 若线性表 L 不存在, 返回 INFEASIBLE; 否则返回线性表 L 的长度。

- 获取元素:

函数原型: `status GetElem(SqList L, int i, ElemType e);`

功能说明: 若线性表 L 不存在, 返回 INFEASIBLE; 若 $i < 1$ 或者 i 超过线性表 L 的长度, 则返回 ERROR; 若获取成功, 则将值赋给 `e`, 并返回 OK。

- 查找元素:

函数原型: `status LocateElem(SqList L, ElemType e);`

功能说明: 若线性表 L 不存在, 返回 INFEASIBLE; 若没有找到指定的元素 `e`, 则查找失败, 返回 ERROR; 若查找成功, 则返回元素逻辑序号 i 。

- 获取前驱元素:

函数原型: `status PriorElem(SqList L, ElemType e, ElemType pre);`

功能说明: 若线性表 L 不存在, 返回 INFEASIBLE; 若没有找到指定元素 `e` 的前驱, 则查找失败, 返回 ERROR; 若查找成功, 则将值赋给 `pre`, 并返回 OK。

- 获取后继元素:

函数原型: `status NextElem(SqList L, ElemType e, ElemType next);`

功能说明: 若线性表 L 不存在, 返回 INFEASIBLE; 若没有找到指定元素 `e` 的后继, 则查找失败, 返回 ERROR; 若查找成功, 则将值赋给 `next`, 并返回 OK。

- 插入元素:

函数原型: `status ListInsert(SqList L, int i, ElemType e);`

如果线性表 L 不存在, 返回 INFEASIBLE; 否则在线性表 L 的第 i 个元素前插入新的元素 `e`, 插入成功返回 OK, 失败返回 ERROR。

- 删除元素：

函数原型: `status ListDelete(Sqlist L,int i,ElemType e);`

功能说明：若线性表 L 不存在，返回 INFEASIBLE；否则删除线性表 L 的第 i 个元素，若删除成功则将删除的值赋给 e 并返回 OK，若删除失败则返回 ERROR。

- 遍历线性表：

函数原型: `status ListTraverse(Sqlist L);`

功能说明：若线性表 L 不存在，返回 INFEASIBLE；否则输出线性表的每一个元素，并返回 OK。

- 最大连续子数组和：

函数原型: `status MaxSubArray(Sqlist L);`

功能说明：若线性表 L 不存在或为空，返回 INFEASIBLE；否则找出一个具有最大和的连续子数组（子数组最少包含一个元素），输出其最大和，并返回 OK。

- 和为 K 的子数组：

函数原型: `status SubArrayNum(Sqlist L,int k);`

功能说明：若线性表 L 不存在，返回 INFEASIBLE；否则输出线性表中和为 k 的连续子数组的个数，并返回 OK。

- 顺序表排序：

函数原型: `status SubArrayNum(Sqlist L,int k);`

功能说明：若线性表 L 不存在，返回 INFEASIBLE；否则输出线性表中和为 k 的连续子数组的个数，并返回 OK。

- 和为 K 的子数组：

函数原型: `status sortList(Sqlist L);`

功能说明：若线性表 L 不存在，返回 INFEASIBLE；否则将 L 由小到大排

序，并返回 OK。

- 线性表读写文件：

- a. 函数原型: `status SaveList(Sqlist L,char FileName[]);`

- 功能说明：如果线性表 L 不存在，返回 INFEASIBLE；否则将线性表 L 的全部元素写入到文件名为 FileName 的文件中，返回 OK。

- b. 函数原型: `status LoadList(Sqlist L,char FileName[]);`

- 功能说明：如果线性表 L 存在，表示 L 中已经有数据，读入数据会覆盖原数据造成数据丢失，返回 INFEASIBLE；否则将文件名为 FileName 的数据读入到线性表 L 中，返回 OK。本实验不考虑用追加的方式读入文件数据追加到现有线性表中。

- 多线性表管理：

- a. 增加一个新线性表

- 函数原型: `status AddList(LISTS Lists,char ListName[]);`

- Lists 是一个以顺序表形式管理的线性表的集合，在集合中增加一个新的空线性表。增加成功返回 OK，否则返回 ERROR。

- b. 移除一个线性表

- 函数原型: `status RemoveList(LISTS Lists,char ListName[]);`

- Lists 是一个以顺序表形式管理的线性表的集合，在集合中查找名称为 ListName 的线性表，有则删除，返回 OK，无则返回 ERROR。

- c. 查找线性表

- 函数原型: `int LocateList(LISTS Lists,char ListName[]);`

- Lists 是一个以顺序表形式管理的线性表的集合，在集合中查找名称为 ListName 的线性表，有则返回线性表的逻辑序号，无则返回 0。

1.2 系统设计

1.2.1 总体设计

- 使用一个大的 while 语句实现整体功能；
- 使用 switch 语句实现菜单栏的操作选择；
- 在每个 case 语句先进行对表是否存在的判断；

1.2.2 有关常量和类型定义

- include "stdio.h"
- include "stdlib.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 LISTINITSIZE 100
- define LISTINCREMENT 10
- typedef int ElemType;

1.2.3 数据结构设计

- 线性表的顺序存储结构是用一段地址连续的存储单元依次存储线性表中的数据元素。

1.2.4 函数算法设计

1) InitList(SqList *L);

初始条件：线性表不存在

操作思路：给 L.elem 分配存储空间，初始化线性表的长度和分配空间。

2) DestroyList(SqList *L);

初始条件：线性表存在

操作思路：释放空间，设置表长为 0，分配空间为 0。

3) ClearList(SqList *L);

初始条件：线性表存在

操作思路：将每个元素设置为 0，表长设置为 0。

4) ListEmpty(SqList L);

初始条件：线性表存在

操作思路：检查表长是否为 0。

5) ListLength(SqList L);

初始条件：线性表存在

操作思路：返回表长。

6) GetElem(SqList L,int i,ElemType *e);

初始条件：线性表存在

操作思路：判断位置是否正确，若正确，则赋值。

7) LocateElem(SqList L,ElemType e);

初始条件：线性表存在

操作思路：遍历比较表中数值，若找到相等数值，返回位置。

8) PriorElem(SqList L,ElemType e,ElemType *pre);

初始条件：线性表存在

操作思路：遍历比较表中数值，若找到相等数值且位置不为 1，返回前一个位置。

9) NextElem(SqList L,ElemType e,ElemType *next);

初始条件：线性表存在

操作思路：遍历比较表中数值，若找到相等数值且位置不为表长，返回后一个位置。

10) ListInsert(SqList *L,int i,ElemType e);

初始条件：线性表存在

操作思路：检查位置是否正确，然后给新元素分配空间，插入元素，表长加一。

11) ListDelete(SqList *L,int i,ElemType *e);

初始条件：线性表存在

操作思路：检查位置是否正确，然后删掉该位置的元素，表长减一。

12) ListTraverse(SqList L);

初始条件：线性表存在

操作思路：遍历表中数值并依次输出。

13) maxSubArray(SqList *L);

初始条件：线性表存在

操作思路：设 $\text{sum}=0$ ，从前往后相加数值赋值给 sum ，若相加第 i 个数值后 sum 增大，则 sum 赋值给 max ；若 sum 减小，则 max 赋值给 sum 。

14) `subarraySum(SqList *L, int e);`

初始条件：线性表存在

操作思路：遍历表中数值，循环依次求相加之后数值的和进行比较，若与给定值相同，则数量加一。

15) `SortList(SqList *L);`

初始条件：线性表存在

操作思路：用冒泡排序进行排序（较大的不断往后交换）。

16) `SaveList(SqList L, char FileName[]);`

初始条件：线性表存在

操作思路：打开文件，将线性表写入文件，关闭文件。

17) `LoadList(SqList *L, char FileName[]);`

初始条件：线性表不存在

操作思路：打开文件，创建一个线性表，将文件写入线性表，关闭文件。

18) `AddList(LISTS *Lists, char ListName[]);`

操作思路：复制名字，给 `Lists.elem` 分配空间，表长加一；将 `L.elem` 地址赋值到 `Lists.elem[i]`。

19) `RemoveList(LISTS *Lists, char ListName[]);`

初始条件：线性表存在

操作思路：删除对应位置的表，释放空间，表长减一，返回 OK。

20) `LocateList(LISTS Lists, char ListName[]);`

初始条件：线性表存在

操作思路：依次遍历元素，比对名字，若名字完全一样，则返回位置，否则没有该表，返回 ERROR。

21) `change(SqList **L, int i);`

初始条件：线性表存在

操作思路：将 `L` 切换到 `Lists` 对应逻辑位置的表头。

1.3 系统实现

1.3.1 函数实现

- 1) InitList 如图 1-1：判断表是否存在，若存在返回 Infeasible；若不存在，给 L.elem 分配存储空间；否则，头地址 L 为存储空间基址, 表长为 0，表的存储容量为 100，返回 OK；否则返回 ERROR；

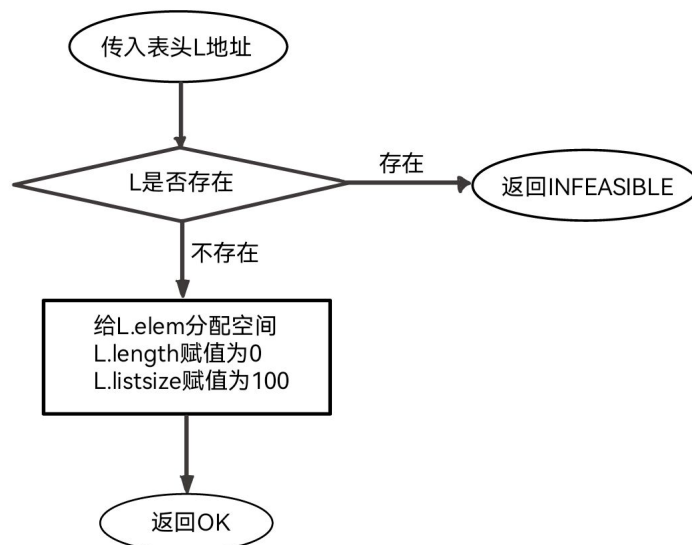


图 1-1 InitList

- 2) DestroyList 如图 1-2：判断表是否存在，若不存在返回 Infeasible；若存在，释放空间，首地址挂空，设置表长为 0，分配空间为 0，返回 OK。

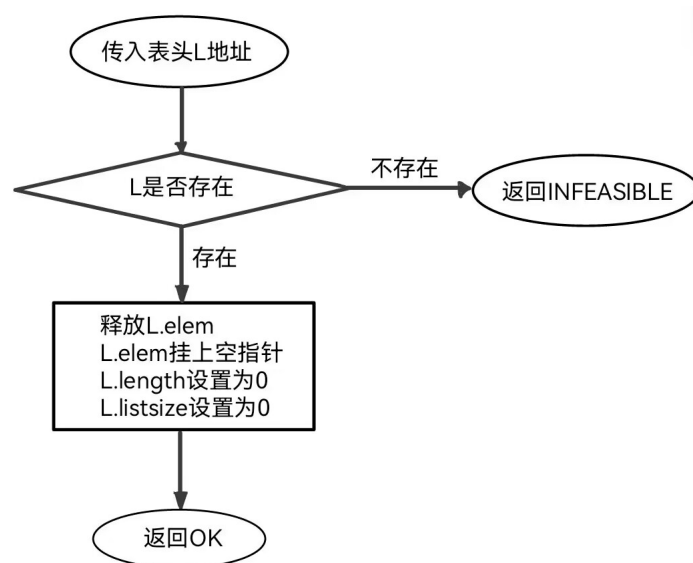


图 1-2 DestroyList

3) ClearList 如图 1-3: 判断表是否存在, 若不存在返回 Infeasible; 若存在, 将每个元素设置为 0, 表长设置为 0, 返回 OK。

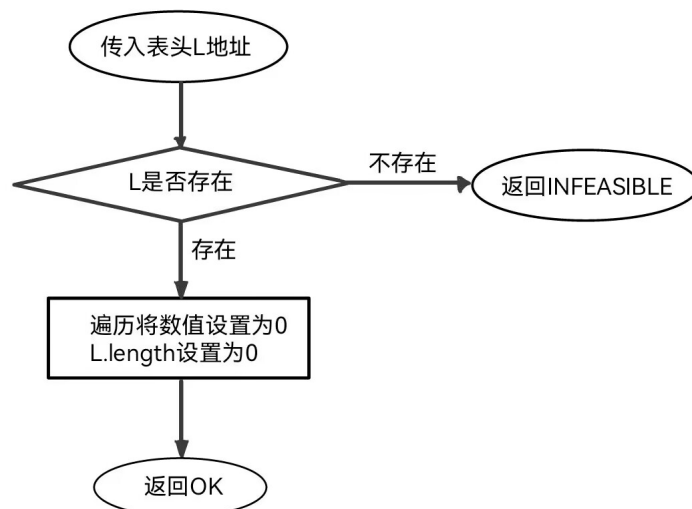


图 1-3 ClearList

4) ListEmpty 如图 1-4: 判断表是否存在, 若不存在返回 Infeasible; 若存在, 如果表长为 0, 返回 OK, 否则返回 ERROR。

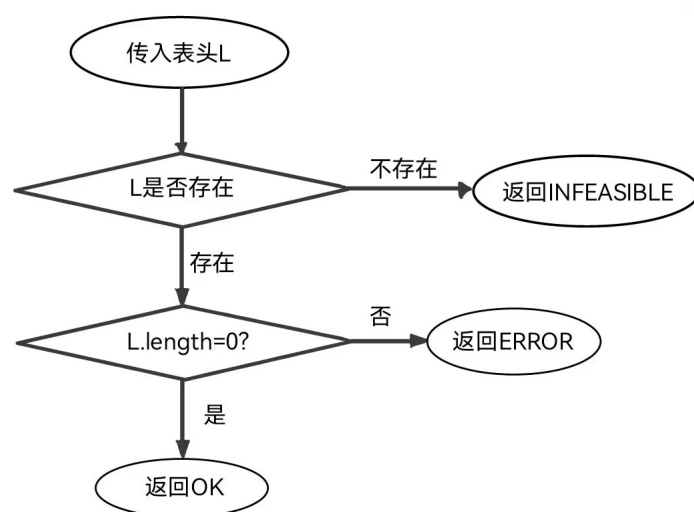


图 1-4 ListEmpty

5) ListLength 如图 1-5: 判断表是否存在, 若不存在返回 Infeasible; 若存在, 释放空间, 设置表长为 0, 分配空间为 0, 返回 OK。

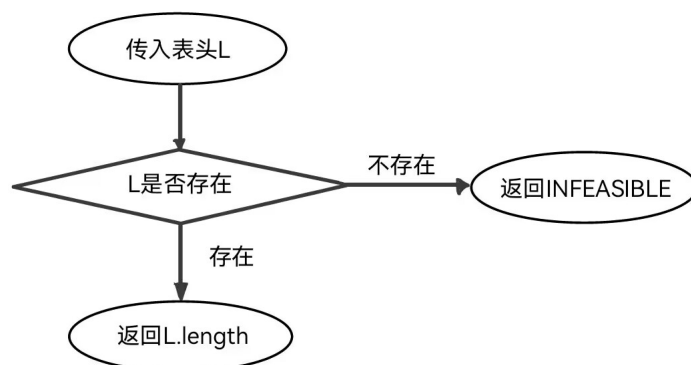


图 1-5 ListLength

6) GetElem 如图 1-6: 判断表是否存在, 若不存在返回 Infeasible; 若存在, 判断位置是否正确, 不正确返回 ERROR; 否则赋值, 返回 OK。

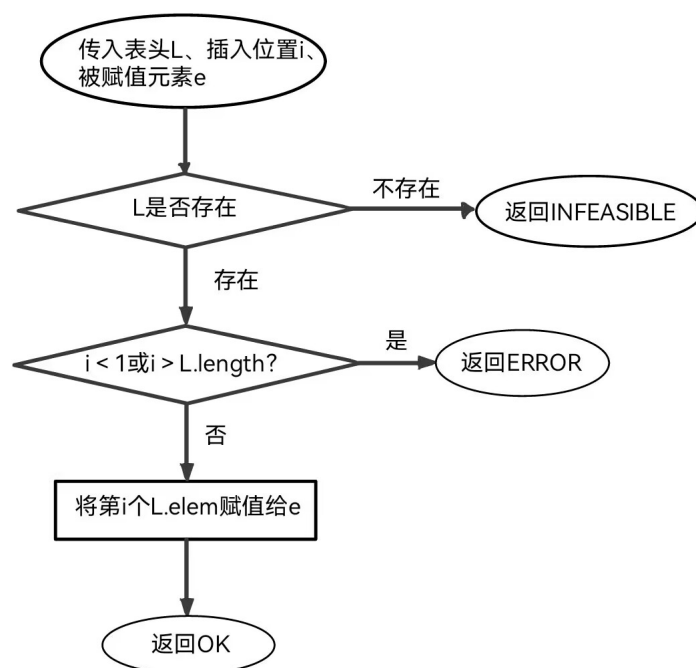


图 1-6 GetElem

7) LocateElem 如图 1-7: 判断表是否存在, 若不存在返回 Infeasible; 若存在, 遍历比较表中数值, 若找到相等数值, 返回位置, 否则返回 ERROR。

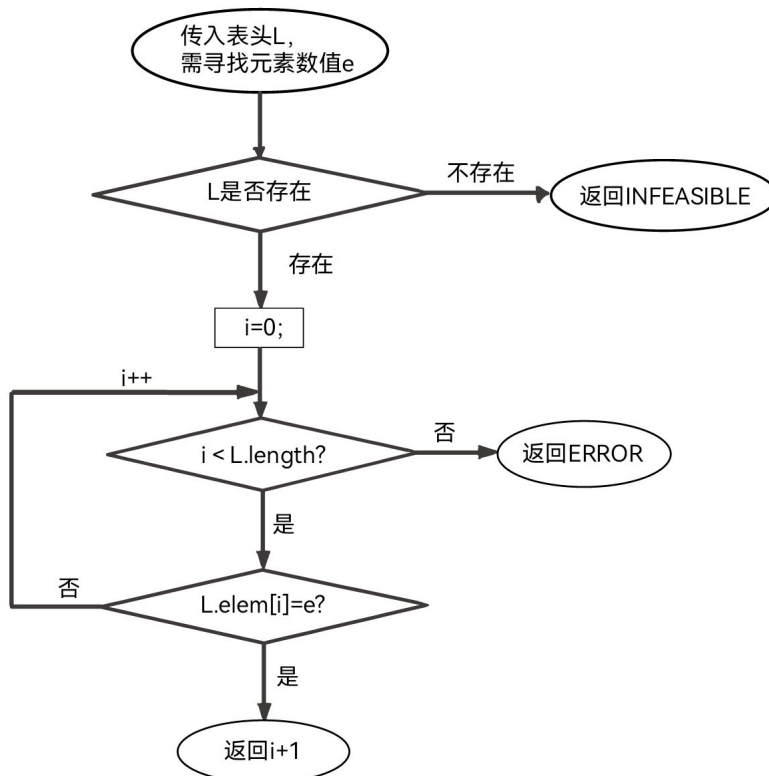


图 1-7 LocateElem

8) PriorElem 如图 1-8: 判断表是否存在, 若不存在返回 Infeasible; 若存在, 遍历比较表中数值, 若找到相等数值且位置不为 1, 返回前一个位置, 否则返回 ERROR。

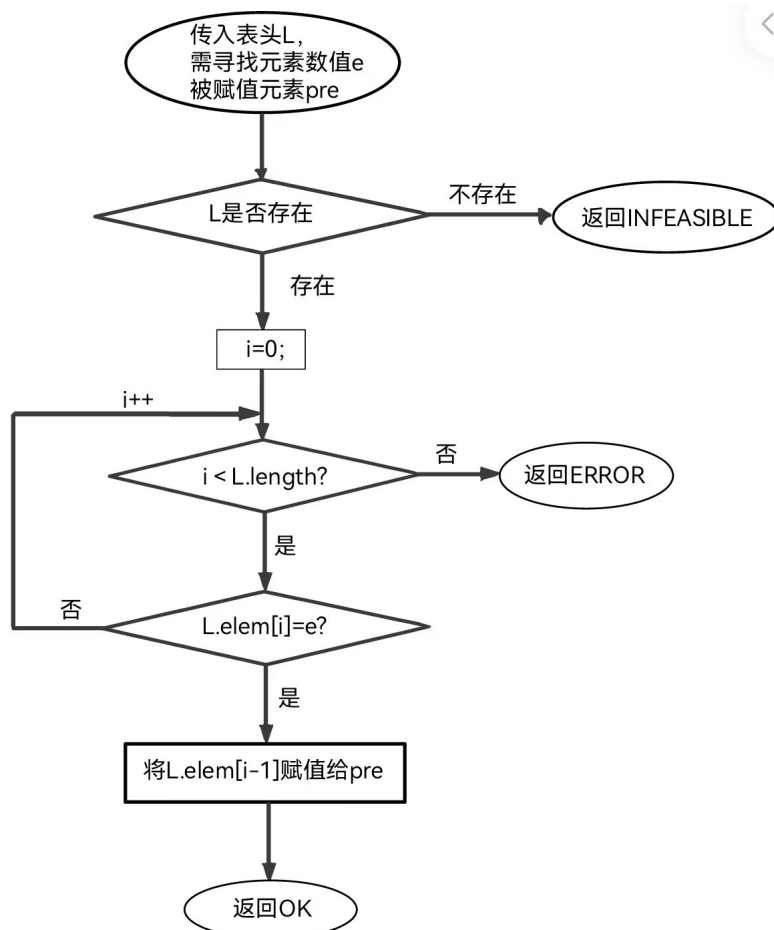


图 1-8 PriorElem

9) NextElem 如图 1-9: 判断表是否存在, 若不存在返回 Infeasible; 若存在, 遍历比较表中数值, 若找到相等数值且位置不为表长, 返回后一个位置, 否则返回 ERROR。

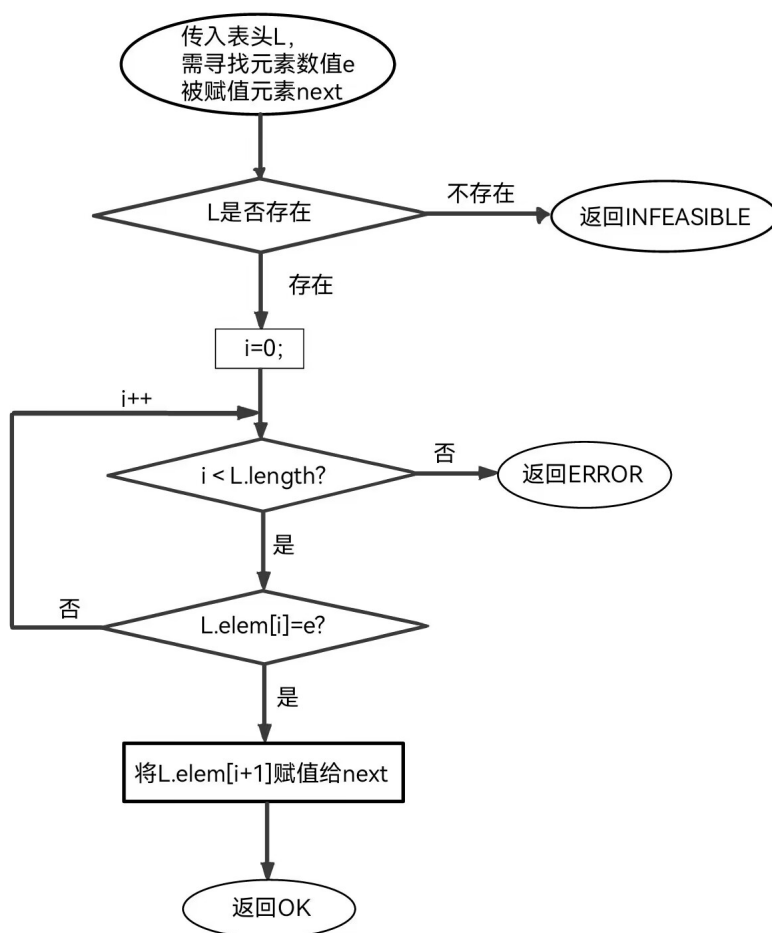


图 1-9 NextElem

- 10) ListInsert 如图 1-10: 判断表是否存在, 若不存在返回 Infeasible; 若存在, 判断位置是否正确, 若不正确, 返回 ERROR; 否则, 给新元素分配空间, 插入元素, 表长加一, 返回 OK。

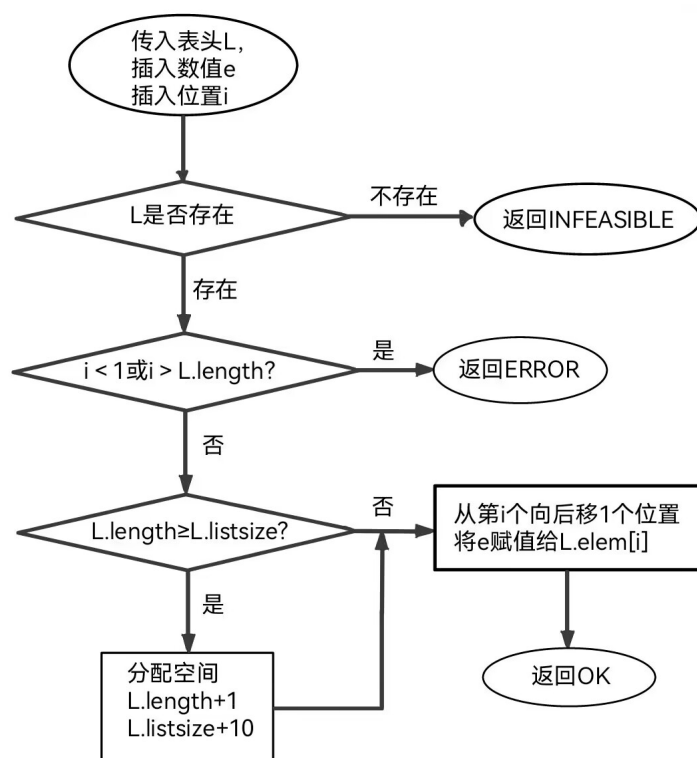


图 1-10 ListInsert

- 11) ListDelete 如图 1-11: 判断表是否存在, 若不存在返回 Infeasible; 若存在, 判断位置是否正确, 若不正确, 返回 ERROR; 否则, 删掉该位置的元素, 表长减一, 返回 OK。

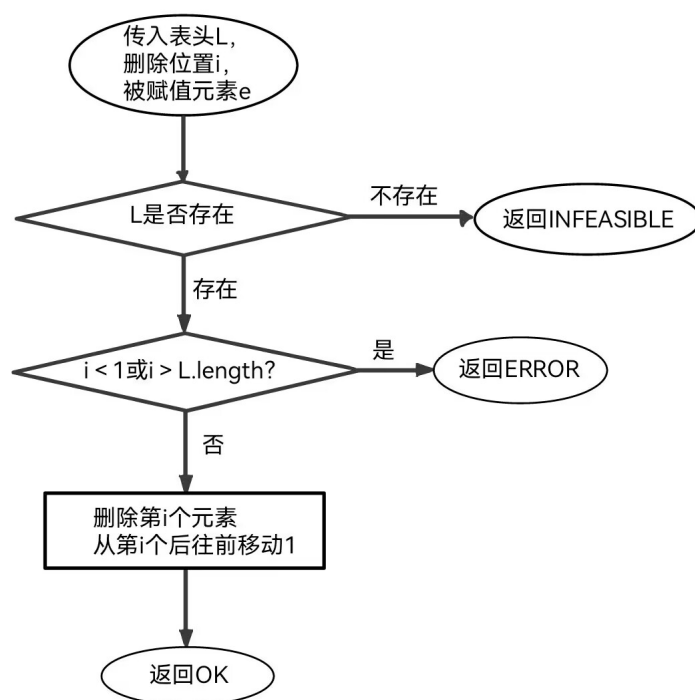


图 1-11 ListDelete

12) ListTraverse 如图 1-12: 判断表是否存在, 若不存在返回 Infeasible; 若存在, 遍历表中数值并 printf 输出, 返回 OK;

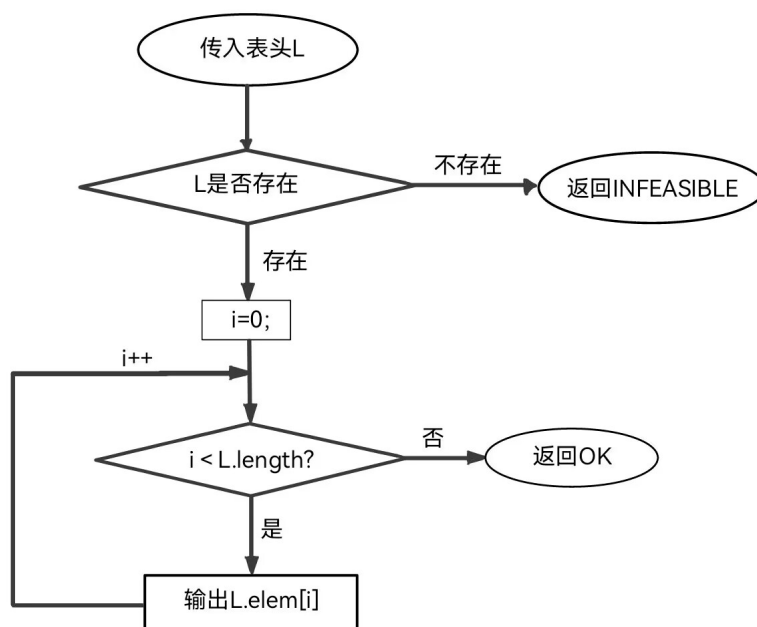


图 1-12 ListTraverse

13) maxSubArray: 判断表是否存在, 若不存在返回 Infeasible; 若存在, 设 sum=0,

从前往后相加数值赋值给 sum，若相加第 i 个数值后 sum 增大，则 sum 赋值给 max；若 sum 减小，则 max 赋值给 sum；最后返回 sum。

- 14) subarraySum: 判断表是否存在，若不存在返回 Infeasible；若存在，遍历表中数值，循环依次求相加之后数值的和进行比较，若与给定值相同，则数量加一；最后返回数量。
- 15) SortList: 判断表是否存在，若不存在返回 Infeasible；若存在，用冒泡排序进行排序（较大的不断往后交换），最后返回 OK。
- 16) SaveList: 判断表是否存在，若不存在返回 Infeasible；若存在，打开文件，将线性表写入文件，关闭文件，返回 OK。
- 17) LoadList: 判断表是否存在，若存在返回 Infeasible；若不存在，打开文件，创建一个线性表，将文件写入线性表，关闭文件，返回 OK。
- 18) AddList: 复制名字，给 Lists.elem 分配空间，表长加一；将 L.elem 地址赋值到 Lists.elem[i]。
- 19) RemoveList: 判断 Lists 是否存在，若存不在返回 Infeasible；若存在，删除对应位置的表，释放空间，表长减一，返回 OK。
- 20) LocateList: 判断 Lists 是否存在，若不存在返回 Infeasible；若存在，依次遍历元素，比对名字，若名字完全一样，则返回位置，否则没有该表，返回 ERROR。
- 21) change: 判断 Lists 是否存在，若不存在返回 Infeasible；若存在，将 L 切换到 Lists 对应逻辑位置。

1.3.2 源代码

详见附录 A 基于顺序存储结构线性表实现的源程序

1.4 系统测试

ps: 测试的线性表序列为 11 12 13。

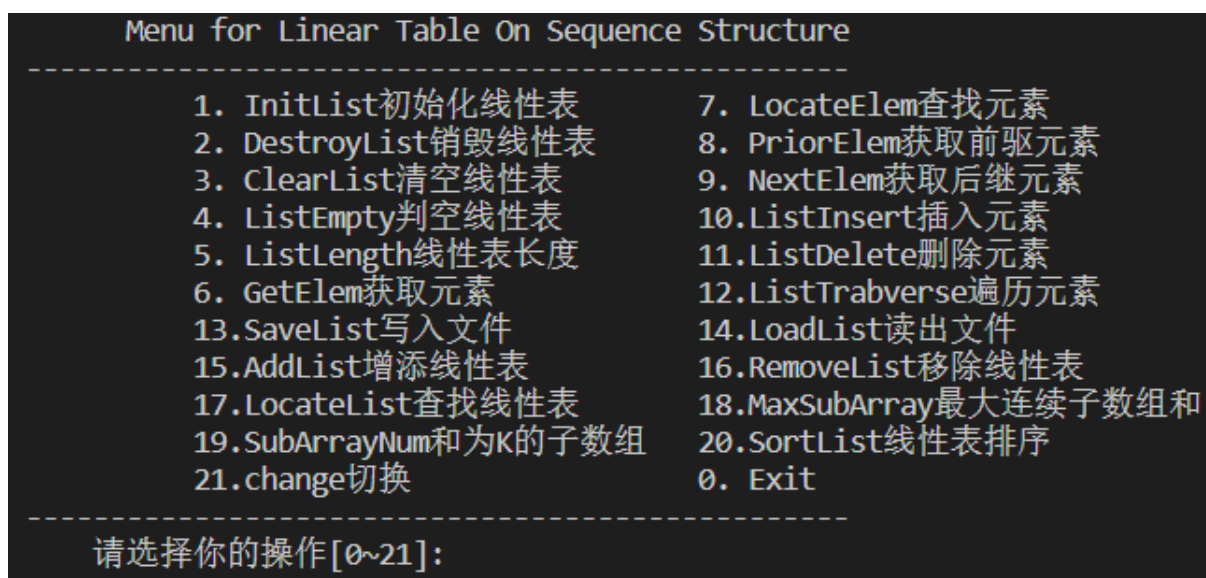


图 1-13 菜单

1) InitList 初始化线性表:

- 若线性表不存在，输出“已完成初始化”(如图 1-14)
- 若线性表存在，输出“线性表已存在”(如图 1-15)

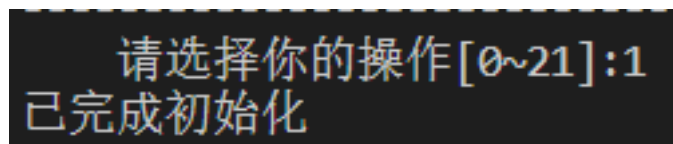


图 1-14 已完成初始化



图 1-15 线性表已存在

2) DestroyList:

- 若线性表不存在，输出“线性表不存在”(如图 1-16)
- 若线性表存在，输出“线性表已销毁”(如图 1-17)

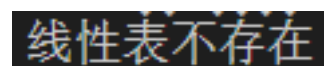
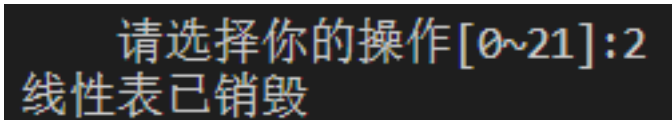


图 1-16 线性表不存在

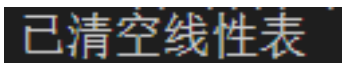


请选择你的操作[0~21]:2
线性表已销毁

图 1-17 线性表已销毁

3) ClearList 清空线性表:

- 若线性表不存在, 输出“线性表不存在”(如图 1-16)
- 若线性表存在, 输出“已清空线性表”(如图 1-18)



已清空线性表

图 1-18 清空线性表

4) ListEmpty 判空线性表:

- 若线性表不存在, 输出“线性表不存在”(如图 1-16)
- 若线性表存在, 且线性表不为空, 输出“线性表不为空”(如图 1-19)
- 若线性表存在, 且线性表为空, 输出“线性表为空”(如图 1-20)



线性表不为空

图 1-19 线性表不为空

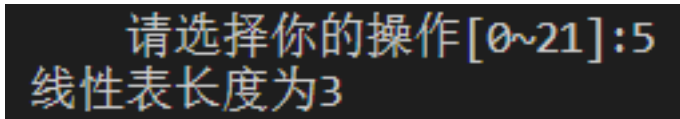


线性表为空

图 1-20 线性表为空

5) ListLength 线性表长度:

- 若线性表不存在, 输出“线性表不存在”(如图 1-16)
- 若线性表存在, 输出线性表的长度(如图 1-21)



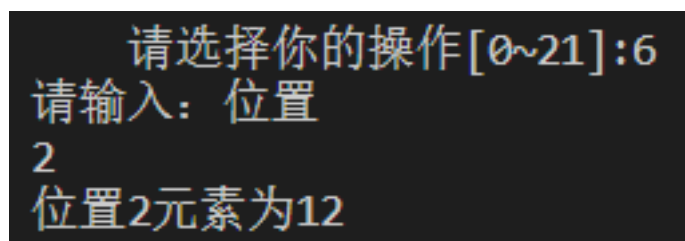
请选择你的操作[0~21]:5
线性表长度为3

图 1-21 线性表长度

6) GetElem 获取元素:

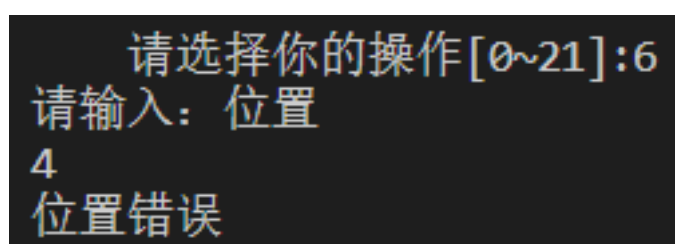
- 若线性表不存在, 输出“线性表不存在”(如图 1-16)

- 若线性表存在，位置正确，输出该位置元素 (如图 1-22)
- 若线性表存在，位置错误，输出“位置错误” (如图 1-23)



```
请选择你的操作[0~21]:6
请输入: 位置
2
位置2元素为12
```

图 1-22 获取元素成功

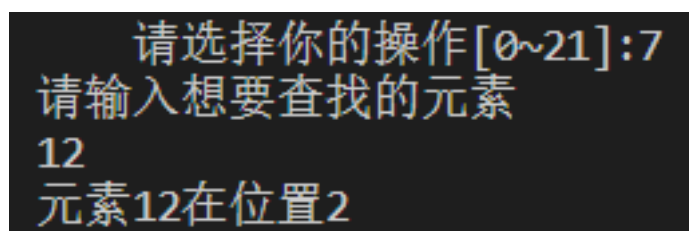


```
请选择你的操作[0~21]:6
请输入: 位置
4
位置错误
```

图 1-23 获取元素失败

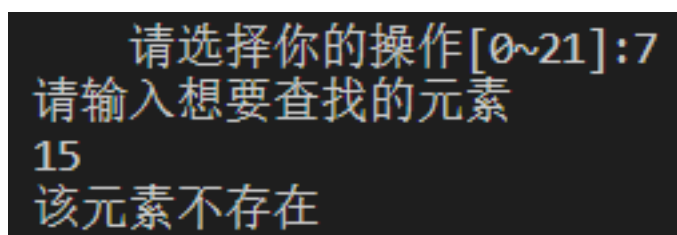
7) LocateElem 查找元素:

- 若线性表不存在，输出“线性表不存在” (如图 1-16)
- 若线性表存在，且元素存在，输出该元素的位置 (如图 1-24)
- 若线性表存在，且元素不存在，输出“该元素不存在” (如图 1-25)



```
请选择你的操作[0~21]:7
请输入想要查找的元素
12
元素12在位置2
```

图 1-24 查找元素成功

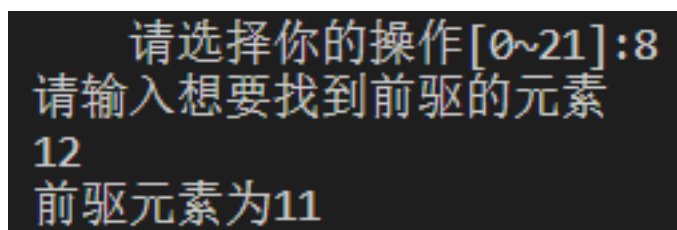


```
请选择你的操作[0~21]:7
请输入想要查找的元素
15
该元素不存在
```

图 1-25 查找元素失败

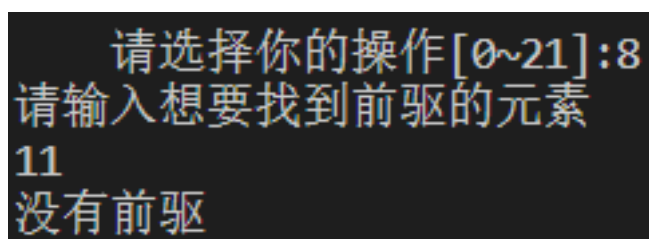
8) PriorElem 获取前驱元素：

- 若线性表不存在，输出“线性表不存在” (如图 1-16)
- 若线性表存在，元素前驱存在，输出该元素前驱 (如图 1-26)
- 若线性表存在，元素前驱不存在（包括元素不存在），输出“没有前驱” (如图 1-27)



```
请选择你的操作[0~21]:8
请输入想要找到前驱的元素
12
前驱元素为11
```

图 1-26 获取前驱成功

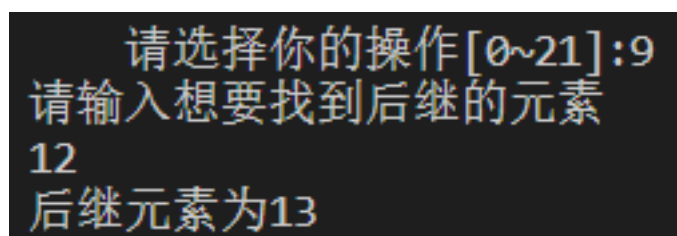


```
请选择你的操作[0~21]:8
请输入想要找到前驱的元素
11
没有前驱
```

图 1-27 获取前驱失败

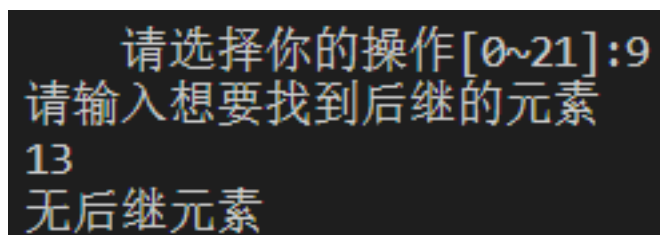
9) NextElem 获取后继元素：

- 若线性表不存在，输出“线性表不存在” (如图 1-16)
- 若线性表存在，元素后继存在，输出该元素后继 (如图 1-28)
- 若线性表存在，元素后继不存在（包括元素不存在），输出“无后继元素” (如图 1-29)



```
请选择你的操作[0~21]:9
请输入想要找到后继的元素
12
后继元素为13
```

图 1-28 获取后继成功

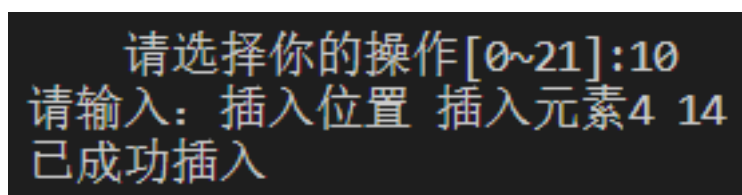


```
请选择你的操作[0~21]:9
请输入想要找到后继的元素
13
无后继元素
```

图 1-29 获取后继失败

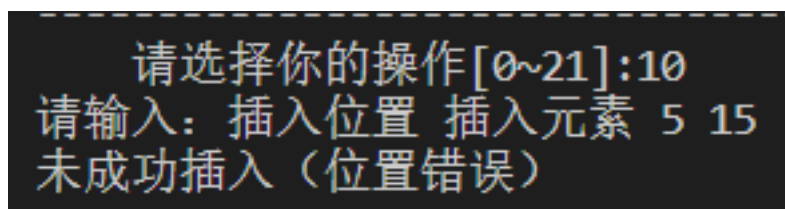
10) ListInsert 插入元素：

- 若线性表不存在，输出“线性表不存在”(如图 1-16)
- 若线性表存在，插入位置正确，输出“已成功插入”(如图 1-30)
- 若线性表存在，插入位置不正确，输出“未成功插入(位置错误)”(如图 1-31)



```
请选择你的操作[0~21]:10
请输入：插入位置 插入元素4 14
已成功插入
```

图 1-30 插入成功

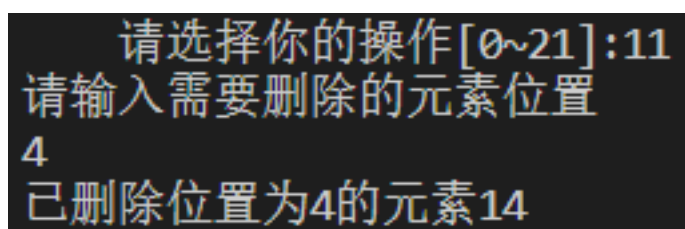


```
请选择你的操作[0~21]:10
请输入：插入位置 插入元素 5 15
未成功插入(位置错误)
```

图 1-31 插入失败

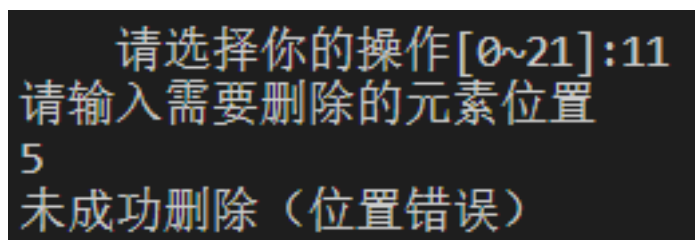
11) ListDelete 删除元素：

- 若线性表不存在，输出“线性表不存在”(如图 1-16)
- 若线性表存在，删除位置正确，输出“已删除第 位的元素”(如图 1-32)
- 若线性表存在，插入位置不正确，输出“未成功删除(位置错误)”(如图 1-33)



```
请选择你的操作[0~21]:11
请输入需要删除的元素位置
4
已删除位置为4的元素14
```

图 1-32 删除成功

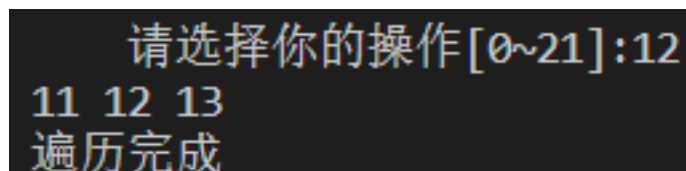


```
请选择你的操作[0~21]:11
请输入需要删除的元素位置
5
未成功删除（位置错误）
```

图 1-33 删除失败

12) ListTraverse 遍历元素：

- 若线性表不存在，输出“线性表不存在” (如图 1-16)
- 若线性表存在，输出所有元素 (如图 1-34)

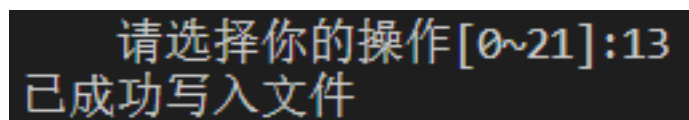


```
请选择你的操作[0~21]:12
11 12 13
遍历完成
```

图 1-34 遍历线性表

13) SaveList 写入文件：

- 若线性表不存在，输出“线性表不存在” (如图 1-16)
- 若线性表存在，将线性表写入文件，输出“已成功写入文件” (如图 1-35)



```
请选择你的操作[0~21]:13
已成功写入文件
```

图 1-35 写入文件

14) LoadList 读出文件：

- 若线性表存在，输出“线性表已存在” (如图 1-16)
- 若线性表存在，将线性表写入文件，输出“已成功读出文件到线性表”

(如图 1-36)

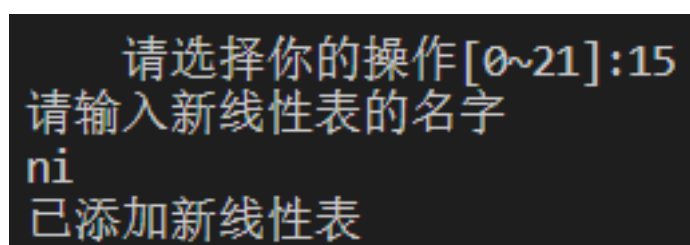


```
已成功 出文件到线性表
```

图 1-36 读出文件

15) AddList 增添线性表:

- 输入新线性表名称，输出“已成功添加新线性表” (如图 1-37)

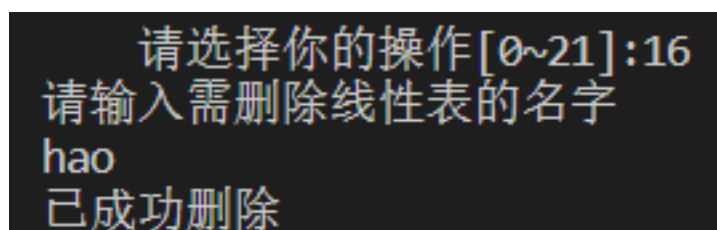


```
请选择你的操作[0~21]:15
请输入新线性表的名字
ni
已成功添加新线性表
```

图 1-37 添加新线性表

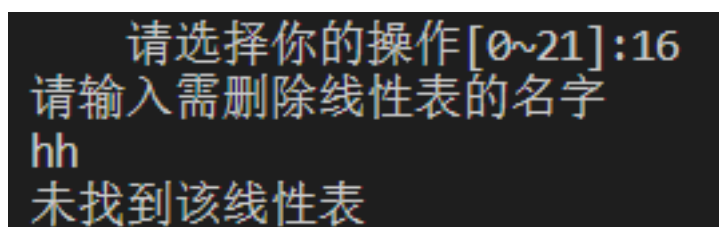
16) RemoveList 移除线性表:

- 若线性表不存在，输出“线性表不存在” (如图 1-16)
- 若线性表存在，输入名字，若找到，输出“已成功删除” (如图 1-38)
- 若线性表存在，输入名字，若未找到，输出“未找到该线性表” (如图 1-39)



```
请选择你的操作[0~21]:16
请输入需删除线性表的名字
hao
已成功删除
```

图 1-38 删除线性表成功

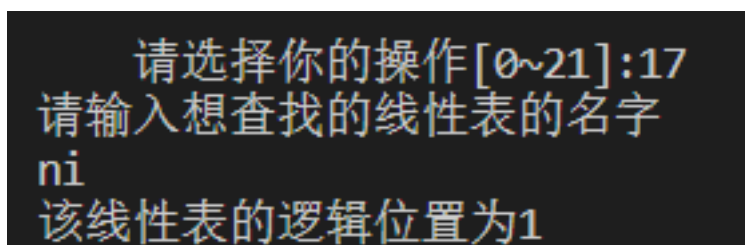


```
请选择你的操作[0~21]:16
请输入需删除线性表的名字
hh
未找到该线性表
```

图 1-39 删除线性表失败

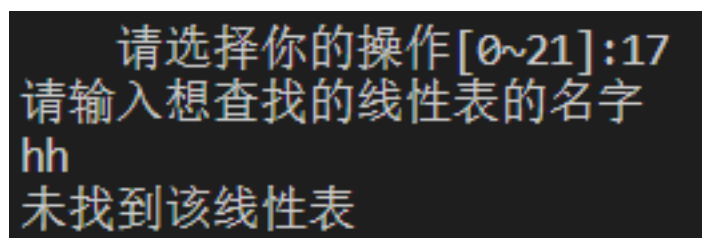
17) LocateList 查找线性表:

- 若线性表不存在，输出“线性表不存在”(如图 1-16)
- 若线性表存在，输入名字，若找到，输出逻辑位置(如图 1-40)
- 若线性表存在，输入名字，若未找到，输出“未找到该线性表”(如图 1-41)



```
请选择你的操作[0~21]:17
请输入想查找的线性表的名字
ni
该线性表的逻辑位置为1
```

图 1-40 查找线性表成功

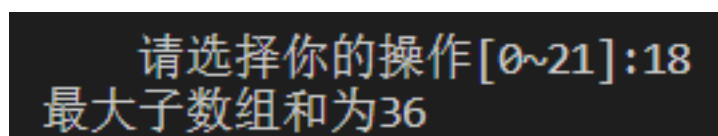


```
请选择你的操作[0~21]:17
请输入想查找的线性表的名字
hh
未找到该线性表
```

图 1-41 查找线性表失败

18) MaxSubArray 最大连续子数组和:

- 若线性表不存在，输出“线性表不存在”(如图 1-16)
- 若线性表存在，输入最大连续子数组和(如图 1-42)



```
请选择你的操作[0~21]:18
最大子数组和为36
```

图 1-42 最大连续子数组和

19) SubArrayNum 和为 K 的子数组个数:

- 若线性表不存在，输出“线性表不存在”(如图 1-16)
- 若线性表存在，输入和为 K 的子数组个数(如图 1-43)

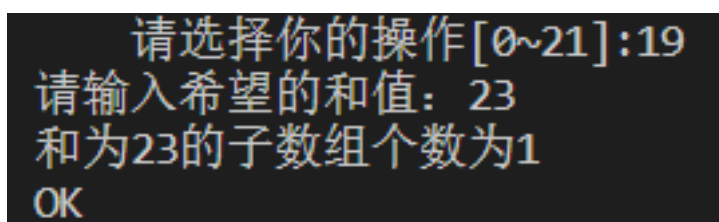


图 1-43 和子数组个数

20) SortList 线性表排序:

- 若线性表不存在, 输出“线性表不存在”(如图 1-16)
- 若线性表存在, 输入和为 K 的子数组个数(如图 1-44)

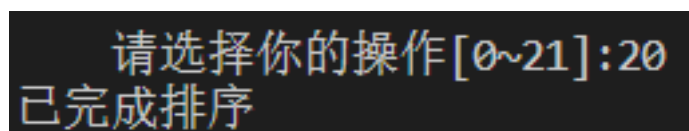


图 1-44 线性表排序

21) change 切换:

- 若线性表不存在, 输出“线性表不存在”(如图 1-16)
- 若线性表存在, 输入逻辑位置, 输出“成功切换”(如图 1-45)

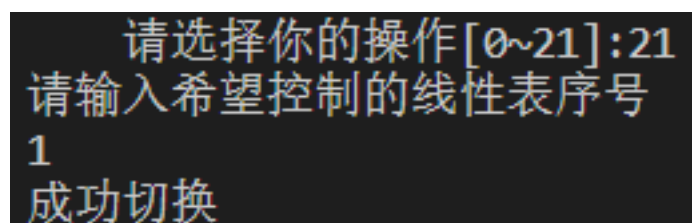


图 1-45 切换

1.5 实验小结

本次实验加深了对线性表的概念, 基本功能的理解, 实现了线性表的基本功能函数。对线性表的逻辑结构和物理结构和二者关系更加熟练。今后的学习应该多从数据结构的角度分析如何进行数据的处理、储存以及时间和空间复杂度, 并勤加练习。

2 第二章：基于二叉链表的二叉树实现

2.1 问题描述

2.1.1 希望通过实验达到：

- 加深对二叉树的基本概念、基本运算的理解；
- 熟练掌握二叉树的逻辑结构与物理结构的关系；
- 物理结构采用二叉树, 熟练掌握二叉树的基本运算的实现。

2.1.2 具体实验目标：

依据最小完备性和常用性相结合的原则，以函数形式定义了二叉树的多种基本运算，具体运算功能定义如下。

1) 二叉树的创建：

函数原型: `status CreateBiTree(BiTree T, TElemType definition[])`;

功能说明：根据带空子树的二叉树先序遍历序列 `definition`（注意该数组元素类型为 `TElemType`），构造一个如下图所示的二叉链表 `T`（要求二叉树 `T` 中各结点关键字具有唯一性）。根指针指向根结点，不需要在根结点之上再增加头结点。

2) 销毁二叉树：

函数原型: `status DestroyBTree(BiTree *T)`;

功能说明：将二叉树 `T` 置空，并释放所有结点的空间。

3) 二叉树清空：

函数原型: `status ClearBiTree(BiTree T)`;

功能说明：将二叉树所有结点值设置为 0。

4) 二叉树判空：

函数原型: `status BiTreeEmpty(BiTree T)`

功能说明：判断二叉树是否为空，若为空返回 `OK`，不为空返回 `ERROR`。

5) 求二叉树深度:

函数原型: `int BiTreeDepth(BiTree T);`

功能说明: 求二叉树 T 深度并返回深度值。

6) 查找结点:

函数原型: `BiTNode* LocateNode(BiTree T,KeyType e);`

功能说明: e 是和 T 中结点关键字类型相同的给定值; 根据 e 查找符合条件的结点, 返回该结点指针, 如无关键字为 e 的结点, 返回 NULL。

7) 结点赋值:

函数原型: `status Assign(BiTree T,KeyType e,TElemType value);`

e 是和 T 中结点关键字类型相同的给定值; 查找结点关键字等于 e 的结点, 将该结点值修改成 value, 返回 OK (要求结点关键字保持唯一性)。如果查找失败, 返回 ERROR。

8) 获得兄弟结点:

函数原型: `BiTNode* GetSibling(BiTree T,KeyType e);`

功能说明: e 是和 T 中结点关键字类型相同的给定值; 查找结点关键字等于 e 的结点的兄弟结点, 返回其兄弟结点指针。如果查找失败, 返回 NULL。

9) 插入结点:

函数原型: `status InsertNode(BiTree T,KeyType e,int LR,TElemType c);`

功能说明: e 是和 T 中结点关键字类型相同的给定值, LR 为 0 或 1, c 是待插入结点; 根据 LR 为 0 或者 1, 插入结点 c 到 T 中, 作为关键字为 e 的结点的左或右孩子结点, 结点 e 的原有左子树或右子树则为结点 c 的右子树, 返回 OK。如果插入失败, 返回 ERROR。特别地, 当 LR 为 -1 时, 作为根结点插入, 原根结点作为 c 的右子树。

10) 删除结点:

函数原型: `status DeleteNode(BiTree T,KeyType e);`

功能说明: e 是和 T 中结点关键字类型相同的给定值。删除 T 中关键字为 e

的结点；同时，根据该被删结点的度进行讨论：

如关键字为 e 的结点度为 0，删除即可；

如关键字为 e 的结点度为 1，用关键字为 e 的结点孩子代替被删除的 e 位置；

如关键字为 e 的结点度为 2，用 e 的左子树（简称为 LC）代替被删除的 e 位置，将 e 的右子树（简称为 RC）作为 LC 中最右节点的右子树。

成功删除结点后返回 OK，否则返回 ERROR。

11) 先序遍历：

函数原型: `status PreOrderTraverse(BiTree T,void (*visit)(BiTree));`

功能说明：对二叉树 T 进行先序遍历，Visit 是一个函数指针的形参（可使用该函数对结点操作），对每个结点调用函数 Visit 访问一次且一次。

12) 中序遍历：

函数原型: `status InOrderTraverse(BiTree T,void (*visit)(BiTree));`

功能说明：对二叉树 T 进行中序遍历，Visit 是一个函数指针的形参（可使用该函数对结点操作），对每个结点调用函数 Visit 访问一次且一次。

13) 后序遍历：

函数原型: `status PostOrderTraverse(BiTree T,void (*visit)(BiTree));`

功能说明：对二叉树 T 进行后序遍历，Visit 是一个函数指针的形参（可使用该函数对结点操作），对每个结点调用函数 Visit 访问一次且一次。

14) 按层遍历：

函数原型: `status LevelOrderTraverse(BiTree T,void (*visit)(BiTree));`

功能说明：对二叉树 T 进行先序遍历，Visit 是一个函数指针的形参（可使用该函数对结点操作），对每个结点调用函数 Visit 访问一次且一次。

15) 最大路径和：

函数原型: `status MaxPathSum(BiTree T);`

功能说明：求出并返回根节点到叶子节点的最大路径和。

16) 最近公共祖先:

函数原型: `status LowestCommonAncestor((BiTree T,int e1,int e2);`

功能说明: 求出并返回该二叉树中 `e1` 节点和 `e2` 节点的最近公共祖先。最近公共祖先可以是 `e1`, `e2` 中的某个节点。

17) 翻转二叉树:

函数原型: `status InvertTree(BiTree T);`

功能说明: 将 `T` 翻转, 使其所有节点的左右节点互换。

18) 写入文件:

函数原型: `status SaveBiTree(BiTree T,char FileName[]);`

功能说明: 函数 `SaveBiTree` 实现将二叉树 `T` 写到文件名为 `FileName` 的文件中, 返回 `OK`。

19) 读出文件:

函数原型: `status LoadBiTree(BiTree T,char FileName[]);`

功能说明: 函数 `LoadBiTree` 读入文件名为 `FileName` 的结点数据, 创建二叉树。

20) 添加新二叉树:

函数原型: `AddList(LISTS *Lists,char ListName[]);`

功能说明: 添加新的二叉以树实现多数控制。

21) 切换:

函数原型: `status change(BiTree **T,int i,LISTS *Lists);`

功能说明: 用于切换以操作不同的二叉树。

2.2 系统设计

2.2.1 总体设计

- 使用一个大的 `while` 语句实现整体功能;

- 使用 switch 语句实现菜单栏的操作选择;
- 在每个 case 语句先进行对表是否存在的判断;

2.2.2 有关常量和类型定义

- define TRUE 1
- define FALSE 0
- define OK 1
- define ERROR 0
- define INFEASIBLE -1
- define OVERFLOW -2
- typedef int status;
- typedef int KeyType;
- definition[] //用于存储输入树中的值
- gjz[] //用于读取并存储树中的各结点关键字的值

2.2.3 结构设计

- 二叉树结点类型定义

```
typedef struct
KeyType key;
char others[20];
TElemType;
```
- 二叉链表结点的定义

```
typedef struct BiTNode
TElemType data;
struct BiTNode *lchild,*rchild;
BiTNode, *BiTree;
```
- 控制二叉树的链表的定义

```
typedef struct //控制链表
struct char name[30];
BiTree Tr;
elem[10];
```



```
int length;  
int listsize;  
LISTS;
```

2.2.4 辅助函数

- `BiTNode* LocateFather(BiTree T,KeyType e)` //用于查找该结点的双亲结点
初始条件：二叉树存在
操作结果：返回寻找结点的双亲结点的地址。
- `void traverse(BiTree T)` //用于读取关键字
初始条件：二叉树存在
操作结果：将二叉树中各结点的值存储在 `gzj` 数组中。
- `void visit(BiTree T)` //访问结点
初始条件：二叉树存在
操作结果：将二叉树中各结点的值打印出来。
- `change(BiTree **T,int i,LISTS *Lists)` // 切换操作的树
初始条件：切换的树存在
操作结果：将 `T` 切换到 `Lists` 中对应的逻辑位置。

2.3 系统实现

2.3.1 函数实现

- 1) `CreateBiTree(BiTree T,TElemType definition[])` 创建二叉树：
判断树是否存在，若存在返回 `Infeasible`；若不存在，给 `T` 分配存储空间，若对应 `definition` 中值不为-1，将 `definition` 中存入的值输入树中，然后分别递归该结点的左右字数，返回 `OK`；否则将左右子树指针挂空，返回 `ERROR`；
- 2) `DestroyBTree(BiTree *T)` 销毁二叉树：
判断树是否存在，若不存在返回 `Infeasible`；若存在，分别递归左右子树释放空间，最后根节点挂空，返回 `OK`。
- 3) `status ClearBiTree(BiTree T)` 清空二叉树：
判断树是否存在，若不存在返回 `Infeasible`；若存在，递归左右子树，将各节点值设为 0，将 `definition` 中数值修改为 0，返回 `OK`，否则返回 `ERROR`。

4) status BiTreeEmpty(BiTree T) 判空二叉树:

判断树是否存在, 若不存在返回 Infeasible; 若存在, 将全局变量 flag 设置为 0, 递归左右子树判断结点数值是否为零, 若不为零, 则将全局变量 flag 设置为 1, 返回 OK。(最后通过 flag 判断是否为空)

5) BiTreeDepth(BiTree T) 二叉树深度:

判断树是否存在, 若不存在返回 ERROR; 若存在, 递归遍历左右子树深度, 比较二者深度, 返回较大的一个数值。

6) LocateNode(BiTree T,KeyType e) 获取二叉树:

判断树是否存在, 若不存在返回 NULL; 若存在, 判断该结点值是否等于 e, 相等返回该结点的地址; 否则分别遍历左右子树。

7) Assign(BiTree *T,KeyType e,TElemType value) 二叉树赋值:

先用 LocateNode 函数返回需要修改数值的结点, 判断结点是否存在, 若不存在返回 ERROR;

若存在, 用 traverse 将关键字存入 gjz 中, 检查新输入数值与 gjz 中数值是否重复, 重复返回 ERROR, 否则进行结点赋值, 返回 OK。

8) GetSibling(BiTree T,KeyType e) 获得兄弟节点:

判断树以及是否有左右子树存在, 若树不存在或同时无左右子树, 返回 NULL; 若树存在:

左子树存在且其结点数值等于 e, 返回右子树地址。

右子树存在且其结点数值等于 e, 返回作左子树地址。

分别递归遍历左右子树。

9) DeleteNode(BiTree *T,KeyType e) 删除二叉树结点:

先用 LocateNode 函数返回需要删除结点的结点, 用 LocateFather 函数返回需要删除结点的双亲结点, 判断两个结点是否存在, 若不存在返回 ERROR; 如关键字为 e 的同时没有左右子结点, 删除即可;

如关键字为 e 的有一个孩子结点, 用关键字为 e 的结点孩子代替被删除的 e 位置;

如关键字为 e 的同时又左右子结点, 用 e 的左子树 (简称为 LC) 代替被删除的 e 位置, 将 e 的右子树 (简称为 RC) 作为 LC 中最右节点的右子树。

成功删除结点后返回 OK, 否则返回 ERROR。

10) PreOrderTraverse(BiTree T,void (*visit)(BiTree)) 前序遍历:

判断树是否存在,若不存在返回 ERROR;若存在,通过 visit 函数访问该结点;递归左子树,递归右子树;返回 OK;

11) InOrderTraverse(BiTree T,void (*visit)(BiTree)) 中序遍历:

判断树是否存在,若不存在返回 ERROR;若存在,递归左子树;通过 visit 函数访问该结点;递归右子树;返回 OK;

12) PostOrderTraverse(BiTree T,void (*visit)(BiTree)) 后序遍历:

判断树是否存在,若不存在返回 ERROR;若存在,递归左子树,递归右子树;通过 visit 函数访问该结点;返回 OK;

13) LevelOrderTraverse(BiTree T, void (*visit)(BiTree)) 层序遍历:

设置全局变量 BiTreeA[] 存储各二叉树。将树的根节点存入 A 中,判断树的根节点是否存在,若不存在返回 ERROR;若存在,通过 visit 函数访问该结点;

判断是否有左子树,如果有,将左子节点存入 A 中。

判断是否有右子树,如果有,将右子节点存入 A 中。

递归访问 A 中各树,返回 OK。

14) MaxPathSum(BiTree T) 最大路径和:

判断表是否存在,若不存在返回 ERROR;若存在,分别递归遍历左子树和右子树;记录二者最大路径和,比较二者最大路径和,返回较大的一个值。

15) LowestCommonAncestor(BiTree T,KeyType e1,KeyType e2) 最近公共祖先:

判断树是否存在,若不存在返回 NULL;若存在:

情况一:通过 LocateNode 函数查找左子树 e1 对应结点存在,若通过 LocateNode 函数查找左子树 e2 对应结点存在,则递归遍历左子树;否则返回该结点的地址。

情况二:通过 LocateNode 函数查找左子树 e1 对应结点不存在,若通过 LocateNode 函数查找右子树 e2 对应结点存在,则递归遍历右子树;否则返回该结点的地址。

16) InvertTree(BiTree *T) 翻转二叉树:

判断树是否存在,若不存在返回 Infeasible;若存在,交换该结点的左右子树,然后递归左子树,递归右子树,最后返回 OK。

17) SaveBiTree(BiTree T, char FileName[]) 写入文件:

判断树是否存在,若不存在返回 Infeasible;若存在,打开文件,将二叉树写

入文件，关闭文件，返回 OK。

18) LoadBiTree(BiTree *T, char FileName[]) 读出文件：

判断树是否存在，若存在返回 Infeasible；若不存在，打开文件，创建一个二叉树，将文件写入线性表，关闭文件，返回 OK。

19) AddList: 复制名字，给 Lists.elem 分配空间，表长加一；将 T 地址赋值到 Lists.elem[i]。

2.3.2 源代码

详见附录 C 基于二叉链表二叉树实现的源程序

2.4 系统测试

ps: 测试的二叉树序列为 1 a, 2 b, 0 null, 0 null, 3 c, 4 d, 0 null, 0 null, 5 e, 0 null, 0 null, -1 null。



图 2-1 菜单 2

1) CreateBiTree 创建二叉树：

- 输出 “请输入带空子树的二叉树先序遍历序列：”
- 操作成功，输出 “二叉树创建成功！” (如图 2-2

```
请输入需要的操作：
1
请输入带空子树的二叉树先序遍历序列：
1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0
二叉树创建成功！
```

图 2-2 创建二叉树

2) DestroyBTree 销毁二叉树：

- 若二叉树不存在，输出“不能销毁不存在的二叉树！”(如图 2-3)
- 若二叉树存在，输出“已成功销毁二叉树。”(如图 2-4)

```
请输入需要的操作：
2
不能销毁不存在的二叉树！
请输入需要的操作：
```

图 2-3 二叉树不存在

```
请输入需要的操作：
2
已成功销毁二叉树。
请输入需要的操作：
```

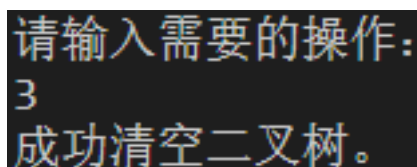
图 2-4 二叉树已销毁

3) ClearBiTree 清空二叉树：

- 若二叉树不存在，输出“不能对不存在的二叉树进行清空操作！”(如图 2-5)
- 若线性表存在，输出“已清空线性表”(如图 2-6)

```
请输入需要的操作：
3
不能对不存在的二叉树进行清空操作！
```

图 2-5 清空二叉树不存在

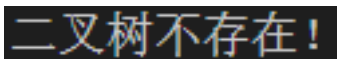


```
请输入需要的操作:  
3  
成功清空二叉树。
```

图 2-6 清空成功

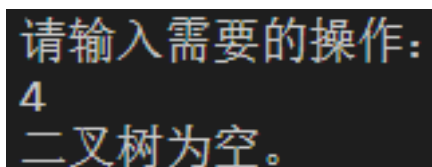
4) BiTreeEmpty 判空二叉树:

- 若二叉树不存在, 输出“二叉树不存在!” (如图 2-7)
- 若线性表存在, 二叉树为空, 输出“二叉树为空。” (如图 2-10)
- 若线性表存在, 二叉树不为空, 输出“二叉树不为空。” (如图 2-11)



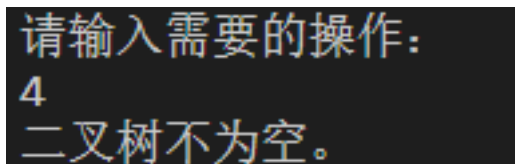
```
二叉树不存在!
```

图 2-7 二叉树不存在



```
请输入需要的操作:  
4  
二叉树为空。
```

图 2-8 判空为空

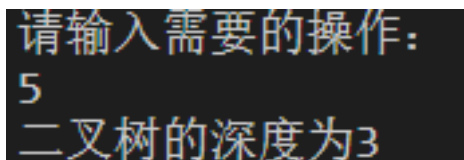


```
请输入需要的操作:  
4  
二叉树不为空。
```

图 2-9 判空不为空

5) BiTreeDepth 二叉树深度:

- 若二叉树不存在, 输出“二叉树不存在!” (如图 2-7)
- 若二叉树存在, 输出二叉树的深度 (如图 2-10)



```
请输入需要的操作:  
5  
二叉树的深度为3
```

图 2-10 二叉树深度

6) LocateNode 查找结点:

- 输出“请输入想要查找的结点的关键字:”
- 若结点存在, 输出”关键字为 的结点,”(如图 2-11)
- 若结点不存在, 输出“查找失败, 不存在关键字为 的结点”(如图 2-12)

```
请输入需要的操作:
6
请输入想要查找的结点的关键字: 3
关键字为3的结点3, c
```

图 2-11 查找结点成功

```
请输入需要的操作:
6
请输入想要查找的结点的关键字: 7
查找失败, 不存在关键字为7的结点
```

图 2-12 查找结点失败

7) Assign 结点赋值:

- 输出“请输入想要修改的结点的关键字:, 将其结点值修改为:”
- 若结点存在, 且关键字不重复, 输出”已将关键字为 的结点值修改为,”(如图 2-13)
- 若结点不存在, 或关键字重复, 输出“赋值操作失败!”(如图 2-14)

```
请输入需要的操作:
7
请输入想要修改的结点的关键字: 3
将其结点值修改为: 10 new
已将关键字为3的结点值修改为 10,new
```

图 2-13 赋值成功


```
请输入需要的操作：
7
请输入想要修改的结点的关键字： 3
将其结点值修改为： 2 b
赋值操作失败！
```

图 2-14 赋值失败

8) GetSibling 获得兄弟节点：

- 输出“请输入想要从二叉树中获得兄弟结点的关键字：”
- 若结点存在，且兄弟节点存在，输出”关键字为 的结点的兄弟结点为，”(如图 2-15)
- 若结点不存在，或无兄弟节点，输出“关键字为 的结点无兄弟结点”(如图 2-16)

```
请输入需要的操作：
8
请输入想要从二叉树中获得兄弟结点的关键字： 3
关键字为3的结点的兄弟结点为2,b
```

图 2-15 获取成功

```
请输入需要的操作：
8
请输入想要从二叉树中获得兄弟结点的关键字： 6
关键字为6的结点无兄弟结点
```

图 2-16 获取失败

9) InsertNode 插入结点：

- 输出“请输入插入结点的父亲的关键字：
插入结点作为 左孩子 (0)/右孩子 (1)/根节点 (-1):
插入结点的值:”
- 若结点存在，且关键字不重复，输出”插入成功!”(如图 2-17)
- 若结点存在，但关键字重复，输出“新增结点关键字重复，插入失败!”

(如图 2-18)

- 若结点不存在，输出“无法找到父亲结点，插入失败!” (如图 2-19)

```
请输入需要的操作:
9
请输入插入结点的父亲的关键字: 2
插入结点作为 左孩子(0)/右孩子(1)/根节点(-1): 0
插入结点的值: 10 new
插入成功!
```

图 2-17 插入成功

```
请输入需要的操作:
9
请输入插入结点的父亲的关键字: 2
插入结点作为 左孩子(0)/右孩子(1)/根节点(-1):
插入结点的值: 3 ee
新增结点关键字重复，插入失败!
```

图 2-18 插入失败（关键字重复）

```
请输入需要的操作:
9
请输入插入结点的父亲的关键字: 6
插入结点作为 左孩子(0)/右孩子(1)/根节点(-1):
插入结点的值: 9 ee
无法找到父亲结点，插入失败!
```

图 2-19 插入失败（结点不存在）

10) DeleteNode 删除节点:

- 输出“请输入想要在线二叉树中删除的结点关键字:”
- 若结点存在，输出”结点关键字为 10 的结点已从二叉树中删除。”(如图 2-20)
- 若结点不存在，输出“二叉树中不含有关键字为 的结点，删除失败!” (如图 2-21)

```
请输入需要的操作:
10
请输入想要在线二叉树中删除的结点关键字: 10
结点关键字为10的结点已从二叉树中删除。
```

图 2-20 删除成功

```
请输入需要的操作:
10
请输入想要在线二叉树中删除的结点关键字: 6
二叉树中不含有关键字为6的结点，删除失败！
```

图 2-21 删除失败

11) PreOrderTraverse 前序遍历:

- 若二叉树不存在，输出”不能遍历不存在的二叉树！”(如图 2-22)
- 若二叉树存在，打印出先序遍历的数值。(如图 2-23)

```
不能遍历不存在的二叉树！
```

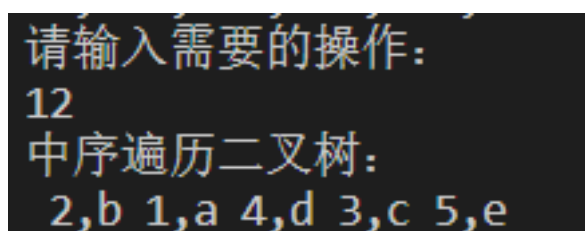
图 2-22 遍历失败

```
请输入需要的操作:
11
先序遍历二叉树:
1,a 2,b 3,c 4,d 5,e
```

图 2-23 先序遍历

12) InOrderTraverse 中序遍历:

- 若二叉树不存在，输出”不能遍历不存在的二叉树！”(如图 2-22)
- 若二叉树存在，打印出中序遍历的数值。(如图 2-24)

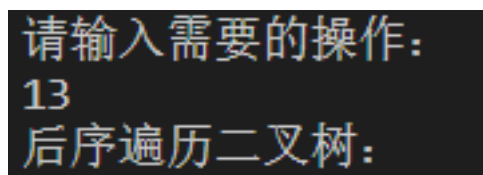


```
请输入需要的操作:  
12  
中序遍历二叉树:  
2,b 1,a 4,d 3,c 5,e
```

图 2-24 中序遍历

13) PostOrderTraverse 后序遍历:

- 若二叉树不存在，输出”不能遍历不存在的二叉树!”(如图 2-22)
- 若二叉树存在，打印出中序遍历的数值。(如图 2-25)

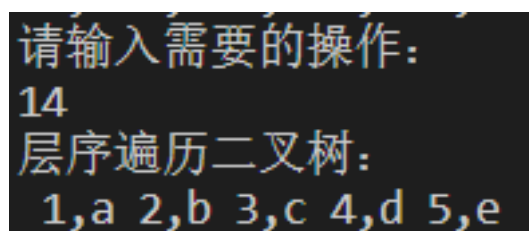


```
请输入需要的操作:  
13  
后序遍历二叉树:
```

图 2-25 后序遍历

14) LevelOrderTraverse 层序遍历:

- 若二叉树不存在，输出”不能遍历不存在的二叉树!”(如图 2-22)
- 若二叉树存在，打印出中序遍历的数值。(如图 2-26)

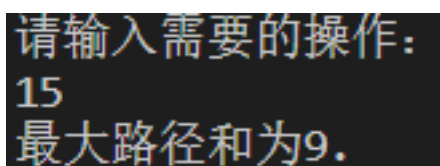


```
请输入需要的操作:  
14  
层序遍历二叉树:  
1,a 2,b 3,c 4,d 5,e
```

图 2-26 层序遍历

15) MaxPathSum 最大路径和:

- 若二叉树不存在，输出 “二叉树不存在!” (如图 2-7)
- 若二叉树存在，输出 “最大路径和为 ”。(如图 2-27)

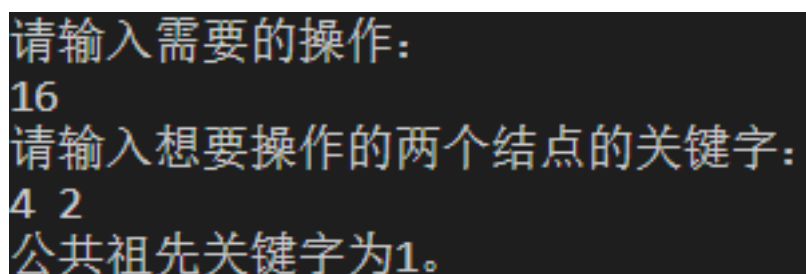


```
请输入需要的操作:  
15  
最大路径和为9.
```

图 2-27 最大路径和

16) LowestCommonAncestor 最近公共祖先:

- 若二叉树不存在, 输出 “二叉树不存在!” (如图 2-7)
- 若二叉树存在, 输出 “最大路径和为 ”。(如图 2-28)

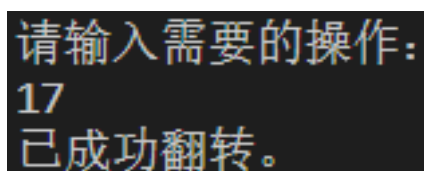


```
请输入需要的操作:  
16  
请输入想要操作的两个结点的关键字:  
4 2  
公共祖先关键字为1.
```

图 2-28 最近公共祖先

17) InvertTree 翻转二叉树:

- 若二叉树不存在, 输出 “二叉树不存在!” (如图 2-7)
- 若二叉树存在, 输出 “已成功翻转。”。(如图 2-29)

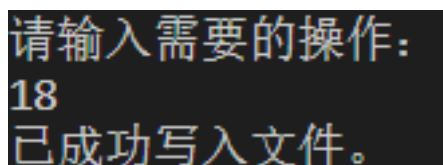


```
请输入需要的操作:  
17  
已成功翻转.
```

图 2-29 翻转成功

18) SaveBiTree 写入文件:

- 若二叉树不存在, 输出 “二叉树不存在!” (如图 2-7)
- 若二叉树存在, 将二叉树写入文件, 输出 “已成功写入文件。”。(如图 2-30)

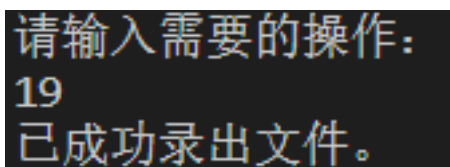


```
请输入需要的操作:
18
已成功写入文件。
```

图 2-30 写入文件成功

19) LoadBiTree 写出文件:

- 若二叉树不存在, 输出“二叉树已存在!”
- 若二叉树存在, 将二叉树写出文件, 输出“已成功录出文件。”。(如图 2-31)

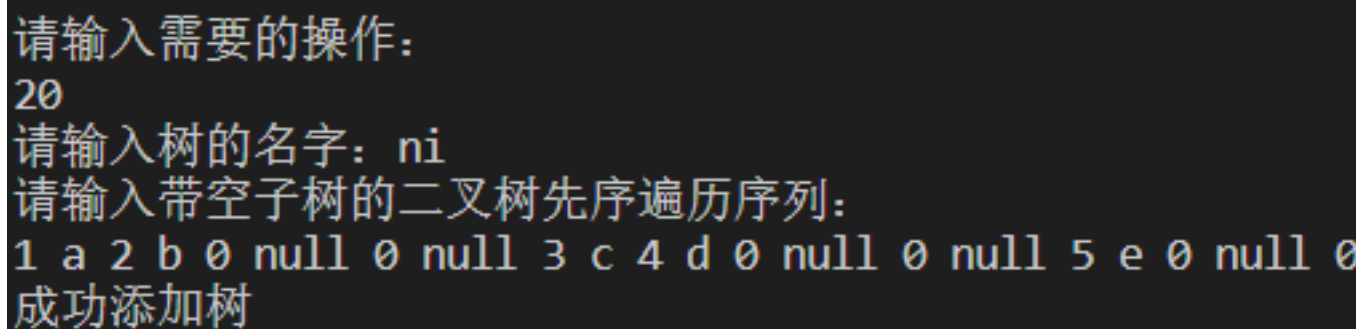


```
请输入需要的操作:
19
已成功录出文件。
```

图 2-31 写出文件成功

20) AddBiTree 增添二叉树:

- 输入新二叉树的名称, 输出“成功添加树”(如图 2-32)



```
请输入需要的操作:
20
请输入树的名字: ni
请输入带空子树的二叉树先序遍历序列:
1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0
成功添加树
```

图 2-32 添加新二叉树

2.5 实验小结

实验往后难度不断加大。二叉树比线性表要复杂许多, 尤其是多个函数功能需要用到递归的思路。在学习和完成函数功能的过程中经常需要查找资料或询问同学, 然后通过理解写代码。代码出现问题时, 通过调试, 监控变量, 能够高效的找出问题所在, 然后解决问题。熟悉了二叉树的逻辑结构和存储结构, 加强

了对递归的理解，受益匪浅。

3 课程的收获和建议

3.1 基于顺序存储结构的线性表实现

通过亲自编写基于顺序存储结构的线性表实现的功能函数，以及将其编写成完整的有多种功能的代码，让我熟悉了基于顺序存储结构的线性表的数据的逻辑结构与存储结构。在不断犯错和修改错误的过程中，加深了对顺序存储结构的认识，并学会更加灵活的处理问题。扩展了解决问题的方法储备，在遇到相似问题时，能够更快的解决。了解到了更多的实现特定功能的方法，并开始注意时间复杂度和空间复杂度。深刻体会了函数“健壮性”的要求，函数需要能够针对各种输入给与回应。这一部分的函数编写以及功能实现比较简单，非常适合新人上手，因此摆在第一个实现非常合适。

3.2 基于链式存储结构的线性表实现

通过亲自编写基于链式存储结构的线性表实现的功能函数，以及将其编写成完整的有多种功能的代码，让我熟悉了基于链式存储结构的线性表的数据的逻辑结构与存储结构。在不断犯错和修改错误的过程中，加深了对链式存储结构的认识，并学会更加灵活的处理问题。扩展了解决问题的方法储备，在遇到相似问题时，能够更快的解决。了解到了更多的实现特定功能的方法，并开始注意时间复杂度和空间复杂度。深刻体会了函数“健壮性”的要求，函数需要能够针对各种输入给与回应。链式存储结构在插入和删除结点时较为方便，只需要修改指针，大大降低了空间复杂度。完成了第一部分，即基于顺序存储结构的线性表实现，之后，再来实现基于链式存储结构的线性表，更能够深刻体会二者的存储结构的不同之处，感受二者的优缺点。

3.3 基于二叉链表的二叉树实现

通过亲自编写基于二叉链表的二叉树实现的功能函数，以及将其编写成完整的有多种功能的代码，让我熟悉了基于二叉链表的二叉树的数据的逻辑结构与存储结构。在不断犯错和修改错误的过程中，加深了二叉链表存储结构的认识，并学会更加灵活的处理问题。扩展了解决问题的方法储备，在遇到相似问题时，能够更快的解决。了解到了更多的实现特定功能的方法，并开始注意时间复杂度

和空间复杂度。深刻体会了函数“健壮性”的要求，函数需要能够针对各种输入给与回应。二叉树函数编写经常需要用到递归的思路，因此加深了我对递归的认识，是我能够更加熟练的应用递归实现目标函数。二叉树的实现相较前两部分难度较大，希望能够得到老师更多的指导。需要指出的是，二叉树部分，清空二叉树函数的要求，头歌和实验要求（word 文档）上不一样，导致产生错误，希望能够统一。最后二叉树是一个非常实用，且巧妙地逻辑结构，此函数的编写使我受益匪浅。

3.4 基于邻接表图实现的源程序

通过亲自编写基于邻接表图实现的源程序的功能函数，以及将其编写成完整的有多种功能的代码，让我熟悉了基于邻接表图的数据的逻辑结构与存储结构。在不断犯错和修改错误的过程中，加深了对邻接表的认识，并学会更加灵活的处理问题。扩展了解决问题的方法储备，在遇到相似问题时，能够更快的解决。了解到了更多的实现特定功能的方法，并开始注意时间复杂度和空间复杂度。深刻体会了函数“健壮性”的要求，函数需要能够针对各种输入给与回应。图的难度相较于前三部分大大加大，各项功能的实现对我来说十分复杂，经常发生我考虑不周全导致发生错误的情况。在不借助外力的情况下我无法独立完成，因此希望老师能够重点讲解这一部分的内容，或者给予关键思路并说明注意事项，或者给予一定的代码进行参照。总而言之，图是一个非常实用，且巧妙地逻辑结构，此函数的编写使我受益匪浅。

附录 A 基于顺序存储结构线性表实现的源程序

```
/* Linear Table On Sequence Structure */

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

/*—————page 16 on textbook —————*/

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

status InitList(SqList *K)
{
    if ((*K).elem==NULL) //是否存在?
    {
        (*K).elem=(ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
        if (!(*K).elem)
            exit(OVERFLOW);
        (*K).length=0;
        (*K).listsize=LIST_INIT_SIZE;
        return OK;
    }
    if ((*K).elem!=NULL)
        return INFEASIBLE;
}

status DestroyList(SqList *L)
```

```
{  
    if((*L).elem)  
    {  
        free((*L).elem); //  
        (*L).elem=NULL;  
        (*L).length=0;  
        (*L).listsize=0;  
        return OK;  
    }  
    return INFEASIBLE;  
}
```

```
status ClearList(SqList *L)  
{  
    if((*L).elem)  
    {  
        for(int i=0;i<=(*L).length;i++)  
        {  
            (*L).elem[i]=0;  
        }  
        (*L).length=0;  
        return OK;  
    }  
    else  
        return INFEASIBLE;  
}
```

```
status ListEmpty(SqList L)  
{  
    if(L.elem)  
    {
```

```
        if(L.length==0)
            return OK;
        else
            return ERROR;
    }
    else
        return INFEASIBLE;
}

status ListLength(SqList L)
{
    if(L.elem)
    {
        return L.length;
    }
    else
        return INFEASIBLE;
}

status GetElem(SqList L,int i,ElemType *e)
{
    if(L.elem)
    {
        if(i<1||i>L.length)
            return 0;
        (*e)=L.elem[i-1];
        return OK;
    }
    else
        return INFEASIBLE;
}
```

```
int LocateElem(SqList L,ElemType e)
{
    if(L.elem)
    {
        ElemType *p;
        int i=0;
        p=L.elem;
        for(;i<=L.length;i++)
        {
            if(*(p+i)==e)
                return i+1;
        }
        return ERROR;
    }
    return INFEASIBLE;
}

status PriorElem(SqList L,ElemType e,ElemType *pre)
{
    if(L.elem)
    {
        int i=0;
        ElemType *p;
        p=L.elem;
        for(;i<L.length;i++)
        {
            if(*(p+i)==e)
            {
                if(i==0)
                    return ERROR;
                else
```

```
        (*pre)=*(p+i-1);
        return OK;
    }
}

return ERROR;
}

return INFEASIBLE;
}

status NextElem(Sqlist L,ElemType e,ElemType *next)
{
    if(L.elem)
    {
        int i;
        ElemType *p;
        p=L.elem;
        for(i=0;i<L.length-1;i++)
        {
            if(*(p+i)==e)
            {
                (*next)=*(p+i+1);
                return OK;
            }
        }
        return ERROR;
    }
    return INFEASIBLE;
}

status ListInsert(Sqlist *L,int i,ElemType e)
{
```

```
    if((*L).elem)
    {
        if(i<1||i>(*L).length+1)
            return 0;
        if((*L).length>=(*L).listsize)
        {
            ElemType *newbase=(ElemType*)malloc(LIST_INIT_SIZE *sizeof(ElemType));
            if(!newbase)
                exit -2;
            (*L).listsize+=LISTINCREMENT;
        }
        ElemType *q=&((*L).elem[i-1]);
        for(int *p=&((*L).elem[(*L).length-1]);p>=q;--p)
        {
            *(p+1)=*p;
        }
        *q=e;
        ++(*L).length;
        return OK;
    }
    return INFEASIBLE;
}

status ListDelete(SqList *L,int i,ElemType *e)
{
    if((*L).elem)
    {
        if(i<1||i>(*L).length)
            return 0;
        ElemType *p;
        p=(*L).elem;
```

```
        (*e)=*(p+i-1);
        for(int j=i-1;j<(*L).length-1;j++)
        {
            *(p+j)=*(p+j+1);
        }
        (*L).length-=1;
        return OK;
    }
    return INFEASIBLE;
}

status ListTraverse(SqList L)
{
    if(L.elem)
    {int *p;
        for(p=L.elem;p<=&L.elem[L.length-1];p++)
        {
            printf("%d ",*p);
        }
        return OK;
    }
    return INFEASIBLE;
}

status SaveList(SqList L,char FileName[])
{
    if(L.elem)
    {
        FILE *fp;
        if((fp=fopen(FileName,"w"))==NULL)
        {
```

```
        return 0;
    }
    for(int i=0;i<L.length;i++)
    {
        fprintf(fp,"%d ",L.elem[i]);
    }
    fclose(fp);
    return OK;
}

return INFEASIBLE;
}

status LoadList(SqList *L,char FileName[])
{
    if((*L).elem==NULL)
    {

        FILE *fp;

        if((fp=fopen(FileName,"r"))==NULL)
            return 0;
        (*L).elem=(ElemType*)malloc((*L).listsize*sizeof(ElemType));
        (*L).length=1;
        (*L).listsize=100;
        for(int i=0;i<(*L).length;i++,(*L).length++)
        {
            fscanf(fp,"%d",&(*L).elem[i]);
            if((*L).elem[i]==0)
                break;
        }
        (*L).length-=1;
    }
}
```



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

status AddList(LISTS *Lists, char ListName[])
{
    strcpy((*Lists).elem[(*Lists).length].name, ListName);
    (*Lists).elem[(*Lists).length].L.elem = (ElemType*) malloc(LIST_INIT_SIZE * sizeof(ElemType));
    (*Lists).elem[(*Lists).length].L.length = 0;
    (*Lists).elem[(*Lists).length].L.listsize = LIST_INIT_SIZE;
    (*Lists).length++;
    return OK;
}

status RemoveList(LISTS *Lists, char ListName[])
{
    int i, *q, m;
    m = (*Lists).length;
    for(i = 0; i < (*Lists).length; i++)
    {
        for(i = 0; i < (*Lists).length; i++)
        {
            for(m = 0; ListName[m] != '\0'; m++)
            {
                if((*Lists).elem[i].name[m] == ListName[m])
                {
                    continue;
                }
            }
            else
            {
                // ... (code continues)
            }
        }
    }
}
```

```
        break;
    }
    if(ListName[m]=='\0')
    {
        for(int j=i;j<((*Lists).length-1);j++)
            (*Lists).elem[j]=(*Lists).elem[j+1];
        (*Lists).length--;
        return OK;
    }
}
return ERROR;
}
```

```
int LocateList(LISTS Lists,char ListName[])
{
    int i,m;
    for(i=0;i<Lists.length;i++)
    {
        for(m=0;ListName[m]!='\0';m++)
        {
            if(Lists.elem[i].name[m]==ListName[m])
            {
                continue;
            }
            else
                break;
        }
        if(ListName[m]=='\0')
            return i+1;
    }
}
```

```
    return 0;
}

int maxSubArray(SqList *L)
{
    int numsSize=ListLength(*L);
    int i = 0, sum = 0;
    int max = (*L).elem[0];
    for(i = 0; i < numsSize; i++) {
        sum += (*L).elem[i];
        if(max < sum)
            max = sum;
        if(sum < 0)
            sum = 0;
    }
    return max;
}

int subarraySum(SqList *L, int e)
{
    int numsSize=ListLength(*L);
    int sum=0;
    int key=0;
    for(int i=0;i<numsSize;i++)
    {
        sum=(*L).elem[i];
        if(sum==e)
        {
            key++;
        }
        for(int j=i+1;j<numsSize;j++)
```

```
{
    sum+=(*L).elem[j];
    if(sum==e)
    {
        key++;
    }
}
}
return key;
}

status SortList(SqList *L)
{
    if(!(*L).length)
    {
        return ERROR;
    }
    int i, j, k;
    int t;
    for(i = 0; i <(*L).length-1; ++i)
    {
        k = i;
        for(j = i+1; j <(*L).length; ++j)
            if((*L).elem[j] < (*L).elem[k])
                k = j;
        if(k != i)
        {
            t = (*L).elem[i];
            (*L).elem[i] = (*L).elem[k];
            (*L).elem[k] = t;
        }
    }
}
```

```
    }  
    for(i = 0; i < (*L).length; ++i)  
        return (*L).elem[i] ;  
}
```

```
status change(SqlList **K,int i,LISTS *Lists)  
{  
    if(i<=(*Lists).length)  
    {(*K)=&((*Lists).elem[i-1].L);  
    return OK;}  
    else  
    return ERROR;  
}
```

```
int main()  
{  
    FILE *fp;  
    LISTS Lists;  
    Lists.length=0;  
    SqlList *K;  
  
    int n, j,con=1,i,e;  
    while(con)  
{    if(con==1){  
        printf("          Menu for Linear Table On Sequence Structure \n");  
        printf("-----\n");  
        printf("          1. InitList 初始化线性表          7. LocateElem 查找元素\n");  
        printf("          2. DestroyList 销毁线性表          8. PriorElem 获取前驱元\n");  
        printf("          3. ClearList 清空线性表          9. NextElem 获取后继元\n");  
    }  
}
```

```

printf("          4. ListEmpty 判空线性表          10.ListInsert 插入元素\n");
printf("          5. ListLength 线性表长度          11.ListDelete 删除元素\n");
printf("          6. GetElem 获取元素          12.ListTraverse 遍历元素\n");
printf("          13.SaveList 写入文件          14.LoadList 读出文件\n");
printf("          15.AddList 增添线性表          16.RemoveList 移除线性表\n");
printf("          17.LocateList 查找线性表          18.MaxSubArray 最大连续子数组\n");
printf("          19.SubArrayNum 和为 K 的子数组          20.SortList 线性表排序\n");
printf("          21.change 切换          0. Exit\n");
printf("-----\n");
printf("    请选择你的操作 [0~21]:");

scanf("%d",&n);
switch(n)
{
    case 1:{
        j=InitList(K);
        if (j==INFEASIBLE)
        {
            printf("线性表已存在\n");
            break;
        }
        if (j==OK)
        {
            printf("已完成初始化\n");
        }
        else
            printf("未完成初始化\n");
        break;
    }

    case 2:{

```

```
j=DestroyList(K);
if (j==INFEASIBLE)
{
    printf("线性表不存在\n");
    break;
}
if (j==OK)
{
    printf("线性表已销毁\n");
}
else printf("未成功销毁\n");
break;
}
```

```
case 3:{
    j=ClearList(K);
    if (j==INFEASIBLE)
    {
        printf("线性表不存在\n");
        break;
    }
    if (j==OK)
    {
        printf("已清空线性表\n");
    }
    else printf("未成功清空\n");
    break;
}
```

```
case 4:{
    j=ListEmpty(*K);
```

```
    if (j==INFEASIBLE)
    {
        printf("线性表不存在\n");
        break;
    }
    if (j==OK)
    {
        printf("线性表为空\n");
    }
    else printf("线性表不为空\n");
    break;
}

case 5:{
    j=ListLength(*K);
    if (j==INFEASIBLE)
    {
        printf("线性表不存在\n");
    }
    else
    {printf("线性表长度为%d\n",j);}
    break;
}

case 6:{
    printf("请输入：位置\n");
    scanf("%d",&i);
    j=GetElem(*K,i,&e);
    if (j==INFEASIBLE)
    {
        printf("线性表不存在\n");
    }
}
```



```
        break;
    }
    if (j==OK)
    {
        printf("位置%d 元素为%d\n",i,e);
    }
    else printf("位置错误\n");
    break;
}
```

```
case 7:{
    printf("请输入想要查找的元素\n");
    scanf("%d",&e);
    j=LocateElem(*K,e);
    if (j==INFEASIBLE)
    {
        printf("线性表不存在\n");
        break;
    }
    else if(j==ERROR)
    {
        printf("该元素不存在\n");
    }
    else
    {
        printf("元素%d 在位置%d\n",e,j);
    }
    break;
}
```

```
case 8:{
```

```
int pre;
printf("请输入想要找到前驱的元素\n");
scanf("%d",&e);
j=PriorElem(*K,e,&pre);
if (j==INFEASIBLE)
{
    printf("线性表不存在\n");
    break;
}
if (j==OK)
{
    printf("前驱元素为%d\n",pre);
}
else printf("没有前驱\n");
break;
}
```

```
case 9:{
    int next;
    printf("请输入想要找到后继的元素\n");
    scanf("%d",&e);
    j=NextElem(*K,e,&next);
    if (j==INFEASIBLE)
    {
        printf("线性表不存在\n");
        break;
    }
    if (j==OK)
    {
        printf("后继元素为%d\n",next);
    }
}
```

```
        else printf("无后继元素\n");
        break;
    }

    case 10:{
        printf("请输入：插入位置 插入元素");
        scanf("%d %d",&i,&e);
        j=ListInsert(K,i,e);
        printf("%s\n", j==OK? "已成功插入\n" : j==ERROR? "未成功插入（位置错误）\n");
        break;
    }

    case 11:{
        printf("请输入需要删除的元素位置\n");
        scanf("%d",&i);
        j=ListDelete(K,i,&e);
        if (j==INFEASIBLE)
        {
            printf("线性表不存在\n");
            break;
        }
        if (j==OK)
        {
            printf("已删除位置为%d的元素%d\n",i,e);
        }
        else printf("未成功删除（位置错误）\n");
        break;
    }

    case 12:{
        j=ListTraverse(*K);
```

```
    if (j==INFEASIBLE)
    {
        printf("线性表不存在\n");
        break;
    }
    if (j==OK)
    {
        printf("\n 遍历完成\n");
    }
    else printf("未成功遍历\n");
    break;
}

case 13:{
    j=SaveList(*K,"C:\\Users\\ffsyd\\Desktop\\vscode\\ffsyd.txt");
    if (j==INFEASIBLE)
    {
        printf("线性表不存在\n");
        break;
    }
    if (j==OK)
    {
        printf("已成功写入文件\n");
    }
    else printf("未成功写入文件\n");
    break;
}

case 14:{
    j=LoadList(K,"C:\\Users\\ffsyd\\Desktop\\vscode\\ffsyd.txt");
    if (j==INFEASIBLE)
```

```
{
    printf("线性表已存在\n");
    break;
}
if (j==OK)
{
    printf("已成功读出文件到线性表\n");
}
else printf("未成功读出文件\n");
break;
}

case 15:{
    char name[30];
    printf("请输入新线性表的名字\n");
    scanf("%s",name);
    j=AddList(&Lists,name);
    if (j==OK)
    {
        printf("已添加新线性表\n");
    }
    else printf("未成功添加\n");
    break;
}

case 16:{
    char name[30];
    printf("请输入需删除线性表的名字\n");
    scanf("%s",name);
    j=RemoveList(&Lists,name);
```

```
        if (j==OK)
        {
            printf("已成功删除\n");
        }
        else printf("未找到该线性表\n");
        break;
    }

case 17:{
    char name[30];
    printf("请输入想查找的线性表的名字\n");
    scanf("%s",name);
    j=LocateList(Lists,name);
    if (j==INFEASIBLE)
    {
        printf("线性表不存在");
        break;
    }
    else if(j==ERROR)
    {
        printf("未找到该线性表\n");
    }
    else
    {
        printf("该线性表的逻辑位置为%d\n",j);
    }
    break;
}

case 18:{
    j=maxSubArray(K);
```

```
        printf("最大子数组和为%d\n",j);
        break;
    }

    case 19:{
        printf("请输入希望的和值: ");
        scanf("%d",&e);
        j=subarraySum(K,e);
        printf("和为%d的子数组个数为%d\nOK\n",e,j);
        break;
    }

    case 20:{
        if(SortList(K)!=ERROR)
            printf("已完成排序\n");
        else printf("未完成排序\n");
        break;}

    case 21:{
        printf("请输入希望控制的线性表序号\n");
        scanf("%d",&i);
        j=change(&K,i,&Lists);
        if(j==OK)
            printf("成功切换\n");
        else
            printf("未成功切换");
        break;
    }

    case 0:
        break;
}
```

```
}  
else  
{printf("ERROR");break;}  
}  
    return 0;  
}
```


附录 B 基于链式存储结构线性表实现的源程序

```
int main() {
#include<windows.h>
#include "def2.h"
#include <stdio.h>
#include <string.h>

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))//判断线性表存在
    {
        (*L)=(LNode *)malloc(sizeof(LIST_INIT_SIZE));//分配空间
        (*L)->next=NULL;
        return OK;
    }
    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))//判断线性表存在
    {
        LNode *p;
        while((*L))
        {
            p=(*L);
            (*L)=(*L)->next;//向后遍历
            free(p);//释放空间
        }
    }
}
```

```
        return OK;
    }
    return INFEASIBLE;
}
```

```
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))
    {
        LNode *p;
        while((*L)->next)
        {
            p=(*L)->next; //从第一个元素开始
            (*L)=(*L)->next; //向后遍历
            free(p); //释放空间
        }
        (*L)->next=NULL; //挂空指针
        return OK;
    }
    return INFEASIBLE;
}
```

```
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
```

```
{
    if(L)
    {
        if(L->next)
            return FALSE;
        return TRUE;
    }
}
```

```
    }  
    return INFEASIBLE;  
}
```

```
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)//判断存在  
    {  
        int i;  
        LNode *q;  
        q=L;  
        for(i=0;q->next!=NULL;i++)  
        {  
            q=q->next;  
        }  
        return i;  
    }  
    return INFEASIBLE;  
}
```

```
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  
{  
    if(L)//判断存在  
    {  
        LNode *q;  
        q=L;  
        int m;  
        for(m=0;q->next!=NULL;m++)  
        {
```

```
        q=q->next; //获取长度
    }
    int ListLength=m;
    if(i<1||i>ListLength) //判断位置是否正确, 是否超过长度或小于 1
        return ERROR;
    q=L;
    for(int j=0; j<i; j++) //找到位置
    {
        q=q->next;
    }
    (*e)=q->data; //赋值
    return OK;
}

return INFEASIBLE;
}
```

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

```
{
    if(L)
    {
        int i;
        LNode *q;
        q=L;
        for(i=0; q!=NULL; i++)
        {
            if(q->data==e)
                return i;
            q=q->next;
        }
    }
}
```

```
    return ERROR;

}

return INFEASIBLE;
}

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
{
    if(L)
    {
        int i;
        LNode *q,*pr;
        q=L->next;
        pr=L;
        for(i=0;pr->next!=NULL;i++)
        {
            if(q->data==e&& i!=0)//找到 e 的位置, 且位置不为 1
            {(*pre)=pr->data;//赋值
            return OK;}
            pr=q;
            q=q->next;//向后遍历
        }
        return ERROR;
    }
    return INFEASIBLE;
}

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,
{
```

```
    if(L)
    {
        LNode *q=L;
        int i;
        for(i=0;q->next!=NULL;i++)//不会遍历到最后一个
        {
            if(q->data==e)//找到 e 的位置
            {(*next)=q->next->data;
             return OK;}
            q=q->next;//向后遍历
        }
        return ERROR;
    }
    return INFEASIBLE;
}
```

```
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

```
{
    if(L)
    {
        LNode *q,*p;
        q=(*L);
        int m;
        for(m=1;q->next!=NULL;m++)//m 为线性表长度
        {
            q=q->next;
        }
        if(i<1||i>m)//判断位置是否正确
        return ERROR;
        int n;
```

```
q=(*L);
for(n=0;n<i-1;n++)//遍历到插入的位置
{
    q=q->next;
}
p=(LNode *)malloc(sizeof(LISTINCREMENT));//分配空间
p->data=e;
p->next=q->next;//新插入 p 的 next 连到插入位置前一个原指向的 next
q->next=p;//插入位置前的 next 指向 q
return OK;
}
return INFEASIBLE;
}
```

```
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

```
{
    if((*L))
    {
        LNode *q,*p;
        q=(*L);
        int m;
        for(m=0;q->next!=NULL;m++)//m 为长度
        {
            q=q->next;
        }
        if(i<1||i>m)//判断位置是否正确
        return ERROR;
        int n;
        q=(*L);
        for(n=0;n<i-1;n++)
```

```
{
    q=q->next; //找到删除的位置
}
p=q->next;
(*e)=p->data; //赋值
q->next=q->next->next; //直接跳过删除的元素，连到下一个 next
free(p); //释放空间
return OK;
}
return INFEASIBLE;
}
```

```
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
```

```
{
    if(L) //判断存在
    {
        LNode *q;
        q=L->next;
        if(L->next==NULL) //没有元素
        {
            return OK;
        }
        else{
            for(;q->next!=NULL;q=q->next) //只遍历到倒数第二位
                printf("%d ",q->data);
            printf("%d",q->data); //最后一个后面不加空格
            return OK;
        }
    }
    return INFEASIBLE;
}
```



```
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 INFEAS
{
    if(L)
    {
        FILE *fp;
        if((fp=fopen(FileName, "w"))==NULL)
        {
            return ERROR;
        }
        LNode *q=L->next;
        for(;q;q=q->next)
        {
            fprintf(fp, "%d ", q->data);
        }
        fclose(fp);
        return OK;
    }
    return INFEASIBLE;
}
```

```
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 IN
{
    if(!(*L))
    {
        FILE *fp;
        if((fp=fopen(FileName, "r"))==NULL)
        {
            return 0;
        }
        (*L)=(LNode *)malloc(sizeof(LIST_INIT_SIZE));
    }
}
```

```
LNode *q,*s;
q=(LNode *)malloc(sizeof(LISTINCREMENT));
s=(LNode *)malloc(sizeof(LISTINCREMENT));
(*L)->next=q;

while((fscanf(fp,"%d",&q->data))!=EOF)
{
    s=q;
    q=(LNode *)malloc(sizeof(LISTINCREMENT));
    s->next=q;
}
s->next=NULL;
fclose(fp);
return OK;
}
return INFEASIBLE;
}

status reverseList(LinkList L)//翻转链表, 成功返回 OK, 线性表不存在返回 INFEASIBLE
{
    if(L){//判断存在
        int length=(ListLength(L));//长度
        int a[100], i=0;
        LNode *q;
        q=L->next;
        for(;i<length;i++)
        {
            a[i]=q->data;//先读一遍, 存入数组 a 中
            q=q->next;
        }
        q=L->next;
```

```
    for(i=i-1;i>=0;i--)
    {
        q->data=a[i]; //再将 a 倒着存入
        q=q->next;
    }
    return OK;
}
else
    return INFEASIBLE;
}

status RemoveNthFromEnd(LinkList *L,int i)
{
    if((*L))
    {
        LNode *q,*p;
        q=(*L);
        int m=(ListLength((*L)));
        if(i<1||i>m) //判断位置是否正确
            return ERROR;
        int n=m-i;
        q=(*L);
        for(;n>0;n--)
        {
            q=q->next; //找到删除的位  $F$ ?, 此时 q 指向需删除数的前一个数
        }
        p=q->next;

        q->next=q->next->next; //直接跳过删除的元素, 连到下一个 next
        free(p); //释放空间
        return OK;
    }
}
```

```
    }  
    return INFEASIBLE;  
}  
  
status sortList(LinkList *L)  
{  
    if((*L)!=NULL)  
    {  
        {int i , num;//count 记录链表结点的个数, num 进行内层循环数  
        LinkList p, q,tail;  
        p = *L;  
        int count=ListLength(*L);  
        for(i = 0; i < count - 1; i++)//外层循环  
        {  
            num = count - i - 1;//记录内层循环需要的次数  
            q = (*L)->next;//令 q 指向第一个结点  
            p = q->next;//令 p 指向后一个结点  
            tail = (*L);//让 tail 始终指向 q 前一个结点, 方便交换  
            while(num--)//内层循环  
            {  
                if(q->data > p->data)//如果该结点的值大于后一个结点, 则交换  
                {  
                    q->next = p->next;  
                    p->next = q;  
                    tail->next = p;  
                }  
                tail = tail->next;  
                q = tail->next;  
                p = q->next;  
            }  
        }  
        return OK;  
    }  
    return INFEASIBLE;  
}
```

```
}
```

```
status change(LinkList **L,int i,LISTS *Lists)//将链表指针指向需要操作的链表
{
    if(i<=(*Lists).length)
    {(*L)=&((*Lists).elem[i-1].L);
    return OK;}
    else
    return ERROR;
}
```

```
status AddList(LISTS *Lists,char ListName[])
// 在 Lists 中增加一个名称为 ListName 的空线性表
{
    strcpy((*Lists).elem[(*Lists).length].name,ListName);
    (*Lists).elem[(*Lists).length].L=(LNode *)malloc(sizeof(LIST_INIT_SIZE));//分配
    (*Lists).elem[(*Lists).length].L->next=NULL;
    (*Lists).length+=1;
    return OK;
}
```

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

```
    for(m=0;ListName[m]!='\0';m++)//对照名称
    {
        if((*Lists).elem[i].name[m]==ListName[m])
        {
            continue;
        }
        else
            break;
    }
    if(ListName[m]=='\0')//若名称完全一样，此时对应字符为空
    {
        for(int j=i;j<((*Lists).length-1);j++)
            (*Lists).elem[j]=(*Lists).elem[j+1];
        (*Lists).length--;
        return OK;
    }
}

return ERROR;
}

int main()
{
    FILE *fp;
    LISTS Lists;
    Lists.length=0;
    LinkList *L;

    int i,j,con=1,n,e;
    while(con)
```

```
{  if(con==1){
    printf(
        "初始化----1\n"
        "销毁-----2\n"
        "清空-----3\n"
        "判空-----4\n"
        "长度-----5\n"
        "获取元素--6\n"
        "查找元素--7\n"
        "获取前驱--8\n"
        "获取后继--9\n"
        "插入-----10\n"
        "删除-----11\n"
        "遍历-----12\n"
        "写入文件-13\n"
        "写出文件-14\n"
        "翻转链表-15\n"
        "倒数删除-16\n"
        "排序-----17\n"
        "切换-----18\n"
        "添加线性表---19\n"
        "移除线性表---20\n"

    );
    scanf("%d",&n);
    switch(n)
    {
        case 1:{
            j = InitList(L);
            if (j==INFEASIBLE)
```

```
{  
    printf("线性表已存在\n");  
    break;  
}  
if (j==OK)  
{  
    printf("已完成初始化\n");  
}  
else  
printf("未成功初始化\n");  
break;  
}
```

```
case 2:{  
    j=DestroyList(L);  
    if (j==INFEASIBLE)  
    {  
        printf("线性表不存在\n");  
        break;  
    }  
    if (j==OK)  
    {  
        printf("已成功销毁\n");  
    }  
    else printf("未成功销毁\n");  
    break;  
}
```

```
case 3:{  
    j=ClearList(L);  
    if (j==INFEASIBLE)
```



```
{
    printf("线性表不存在\n");
    break;
}
if (j==OK)
{
    printf("已清空线性表\n");
}
else printf("未成功清空\n");
break;
}
```

```
case 4:{
    j=ListEmpty(*L);
    if (j==INFEASIBLE)
    {
        printf("线性表不存在\n");
        break;
    }
    if (j==OK)
    {
        printf("线性表为空\n");
    }
    else printf("线性表不为空\n");
    break;
}
```

```
case 5:{
    j=ListLength(*L);
    if (j==INFEASIBLE)
    {
```

```
        printf("线性表不存在\n");
    }
    else
    {printf("长度为%d\n",j);}
    break;
}

case 6:{
    printf("请输入: 位置\n");
    scanf("%d",&i);
    j=GetElem(*L,i,&e);
    if (j==INFEASIBLE)
    {
        printf("线性表不存在\n");
        break;
    }
    if (j==OK)
    {
        printf("位置%d 元素为%d\n",i,e);
    }
    else printf("位置错误\n");
    break;
}

case 7:{
    printf("请输入要查找的元素\n");
    scanf("%d",&e);
    j=LocateElem(*L,e);
    if (j==INFEASIBLE)
    {
        printf("线性表不存在\n");
```

```
        break;
    }
    else if(j==ERROR)
    {
        printf("该元素不存在\n");
    }
    else
    {
        printf("元素%d 在位置%d\n",e,j);
    }
    break;
}

case 8:{
    int pre;
    printf("请输入想要找到前驱的元素\n");
    scanf("%d",&e);
    j=PriorElem(*L,e,&pre);
    if (j==INFEASIBLE)
    {
        printf("线性表不存在\n");
        break;
    }
    if (j==OK)
    {
        printf("前驱元素为%d\n",pre);
    }
    else printf("没有前驱\n");
    break;
}
```

```
case 9:{
    int next;
    printf("请输入想要找到后继的元素\n");
    scanf("%d",&e);
    j=NextElem(*L,e,&next);
    if (j==INFEASIBLE)
    {
        printf("线性表不存在\n");
        break;
    }
    if (j==OK)
    {
        printf("后继元素为%d\n",next);
    }
    else printf("无后继元素\n");
    break;
}

case 10:{
    printf("请输入：插入位置 元素值\n");
    scanf("%d %d",&i,&e);
    j=ListInsert(L,i,e);
    printf("%s\n", j==OK? "已成功插元素" : j==ERROR? "未成功插入（位置错误）");
    break;
}

case 11:{
    printf("请输入需要删除的元素位置\n");
    scanf("%d",&i);
    j=ListDelete(L,i,&e);
    if (j==INFEASIBLE)
```

```
{
    printf("线性表不存在\n");
    break;
}
if (j==OK)
{
    printf("已删除位置为%d 的元为%d\n",i,e);
}
else printf("未成功删除 (位置错误) \n");
break;
}

case 12:{
    j=ListTraverse(*L);
    if (j==INFEASIBLE)
    {
        printf("线性表不存在\n");
        break;
    }
    if (j==OK)
    {
        printf("\n 遍历完成\n");
    }
    else printf("遍历错误\n");
    break;
}

case 13:{
    j=SaveList(*L,"C:\\Users\\ffsyd\\Desktop\\vscode\\ffsyd.txt");
    if (j==INFEASIBLE)
    {
```

```
        printf("线性表不存在\n");
        break;
    }
    if (j==OK)
    {
        printf("已成功读入文件\n");
    }
    else printf("未成功，出现错误\n");
    break;
}

case 14:{
    j=LoadList(L,"C:\\Users\\ffsyd\\Desktop\\vscode\\ffsyd.txt");
    if (j==INFEASIBLE)
    {
        printf("线性表不存在\n");
        break;
    }
    if (j==OK)
    {
        printf("已成功读出文件到线性表\n");
    }
    else printf("未成功，出现错误\n");
    break;
}

case 15:{
    j=reverseList(*L);
    if(j==OK)
        printf("已成功翻转\n");
    else
```

```
printf("线性表不存在\n");
break;
}

case 16:{
    printf("需要倒数第几个元素？\n");
    scanf("%d",&i);
    j=RemoveNthFromEnd(L,i);
    if (j==INFEASIBLE)
    {
        printf("线性表不存在\n");
        break;
    }
    if (j==OK)
    {

        printf("已删除倒数第%d个元素\n",i);
    }
    else printf("未成功删除（位置错误）\n");
    break;
}

case 17:{
    if(sortList(L)!=ERROR)
        printf("已完成排序\n");
    else printf("排序失败\n");
    break;}

case 18:{
    printf("请输入希望控制的线性表序号\n");
    scanf("%d",&i);
    j=change(&L,i,&Lists);
    if(j==OK)
```

```
printf("成功切换\n");
else
printf("未成功切换");
break;
}

case 19:{
    char name[30];
    printf("请输入新线性表的名字\n");
    scanf("%s",name);
    j=AddList(&Lists,name);
    if (j==OK)
    {
        printf("已添加新线性表\n");
    }
    else printf("未成功添加\n");
    break;
}

case 20:{
    char name[30];
    printf("请输入需删除线性表名字\n");
    scanf("%s",name);
    j=RemoveList(&Lists,name);
    if (j==OK)
    {
        printf("已成功删除\n");
    }
    else printf("未找到该线性表\n");
    break;
}
```



```
    }  
    printf("继续操作输 1, 否则输 0\n");  
    scanf("%d",&con);  
}  
else  
{printf("ERROR");break;}  
}  
    return 0;  
}  
}
```

附录 C 基于二叉链表二叉树实现的源程序

```
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
#pragma warning(disable:4996)
#pragma warning(disable:6031)

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{ //控制链表
    struct { char name[30];
            BiTree Tr;
    } elem[10];
    int length;
    int listsize;
}LISTS;
```

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

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

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

```
TElemType definition[50]={0};
```

```
BiTree A[100];
```

```
int gjz[20][100]={0}; //记录每棵树的关键字, 查重
```

```
int q=0, j=0, i, z;
```

```
int flag=0; //标记是否为空, 标记是否重复
```

```
int num=0; //记录控制的树的序号
```

```
char FileName[30] = { '\0' };
```

```
void traverse(BiTree T){
```

```
    if(T){
```

```
        gjz[num][z]=T->data.key;
```

```
        z++;
```

```
        traverse(T->lchild);
```

```
        traverse(T->rchild);
```

```
    }
```

```
}
```

```
status CreateBiTree(BiTree *T, TElemType definition[])
```

```
{ //根据带空枝的二叉树先根遍历序列 definition 构造一棵二叉树, 将根节点指针赋值给 T 并返回
```

```
    if(definition[q].key==0){
```

```
        (*T)=NULL;
```

```
        q++;
```

```
        return OK;
```

```
    }
```

```
    else if(definition[q].key==-1)
```

```
        return OK;
```

```
    else{
```

```
(*T)=(BiTree)malloc(sizeof(BiTNode));
(*T)->data.key=definition[q].key;
gjz[num][j]=definition[q].key;
for(int m=0;m<j;m++)
{
    if(gjz[num][m]==gjz[num][j])
        return ERROR;
}
j++; //j 是全局变量, 查找数组里的关键字
strcpy((*T)->data.others,definition[q++].others);
(*T)->lchild=NULL;
(*T)->rchild=NULL;
}
CreateBiTree(&(*T)->lchild,definition);
CreateBiTree(&(*T)->rchild,definition);
}

//销毁
status DestroyBTree(BiTree *T)
{
    if(*T)
    { //若当前结点存在
        if((*T)->lchild)
        { //则判断是否有左孩子, 若有则先销毁左孩子
            DestroyBTree(&(*T)->lchild); //递归销毁左孩子
        }
        if((*T)->rchild)
        { //无左孩子则判断是否有右孩子, 若有则销毁右孩子
            DestroyBTree(&(*T)->rchild); //递归销毁右孩子
        }
        free(*T); //若都没有则释放当前结点
    }
}
```

华中科技大学课程实验报告

```
        *T=NULL;    //将结点置空
        /*在通过 free 函数释放空间后，将指针立即置空*/
        return OK;
    }
    return INFEASIBLE;
}

//判空
/*status BiTreeEmpty(BiTree T)
{
    if(T)
        return FALSE;
    else
        return TRUE;
}*/
status BiTreeEmpty(BiTree T)
{
    if (T == NULL)
    {
        return INFEASIBLE;
    }
    BiTreeEmpty(T->lchild);
    BiTreeEmpty(T->rchild);
    if (T->data.key != 0)
    {
        flag = 1; //flag 是全局变量，进入函数前已设为 0，若不为空则变成 1
    }
    return OK;
}

/*status ClearBiTree(BiTree *T)
{

```

```
    if (!*T)
    {
        return INFEASIBLE;
    }

    ClearBiTree(&(*T)->lchild);
    ClearBiTree(&(*T)->rchild);
    free(*T);
    *T = NULL;
} */

status ClearBiTree(BiTree *T)
//将二叉树设置成空，并删除所有结点，释放结点空间
{
    if (*T == NULL)
    {
        return INFEASIBLE;
    }
    ClearBiTree(&(*T)->lchild);
    ClearBiTree(&(*T)->rchild);
    (*T)->data.key = 0;
    (*T)->data.others[0] = 0;
    return OK;
}

int BiTreeDepth(BiTree T)
//求二叉树 T 的深度
{
    int ans1, ansr, ans;
    if (T) {
        ans1 = BiTreeDepth(T->lchild);
        ansr = BiTreeDepth(T->rchild);
```

```
    ans=ansl>ansr?ansl:ansr;//哪个大返回哪个
    return ans+1;
}
return 0;
}
```

BiTNode* LocateNode(BiTree T,KeyType e)

//查找结点

```
{
    if(!T)//判断不存在
        return NULL;
    if(T->data.key==e)
        return T;
    BiTNode* an;
    if(T){
        an=LocateNode(T->lchild,e);
        if(an)//如果不是 NULL, 返回 ans
            return an;
        an=LocateNode(T->rchild,e);
        if(an)
            return an;
    }
    return an;
}
```

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

```
{
    BiTree target=LocateNode(*T,e);
    if(target==NULL)
        return ERROR;
}
```

```
traverse(*T);
for(int c=0;c<j;c++)
{
    if(value.key==gjz[num][c]&&target->data.key!=gjz[num][c])
        return ERROR;
}
target->data.key=value.key;
strcpy(target->data.others,value.others);
return OK;
}

BiTNode* GetSibling(BiTree T,KeyType e)
//实现获得兄弟结点
{
    if(!T||T->lchild==NULL&&T->rchild==NULL)
        return NULL;

    if(T->lchild&&T->lchild->data.key==e)
        return T->rchild;
    if(T->rchild&&T->rchild->data.key==e)//首先保证左右孩子的存在再讨论其 data
        return T->lchild;
    BiTNode* an=NULL;
    if(T){
        an=GetSibling(T->lchild,e);
        if(an)
            return an;
        an=GetSibling(T->rchild,e);
        if(an)
            return an;
    }
}
```



```
    return an;
}

status InsertNode(BiTree *T,KeyType e,int LR,TElemType c)
//插入结点。
{
    BiTree cur=LocateNode(*T,e);
    if(cur==NULL)
        return ERROR;
    z=0;
    traverse(*T);
    for(int m=0;m<z;m++)
    {
        if(gjz[num][m]==c.key)//检查关键字是否重复
        {flag=1;//标记有重复
        return ERROR;}
    }
    if(LR==0){//左节点
        BiTree p=(BiTree)malloc(sizeof(BiTNode));
        p->data=c;
        p->rchild=cur->lchild;
        p->lchild=NULL;
        cur->lchild=p;
    }
    else if(LR==1){//右节点
        BiTree p=(BiTree)malloc(sizeof(BiTNode));
        p->data=c;
        p->rchild=cur->rchild;
        p->lchild=NULL;
        cur->rchild=p;
    }
}
```

```
else if(LR==-1){//父节点
    BiTree p=(BiTree)malloc(sizeof(BiTNode));
    p->data.key=c.key;
    strcpy(p->data.others,c.others);
    p->rchild=*T;
    p->lchild=NULL;//不用的节点放 NULL
    *T=p;
}
return OK;
}

//删除节点
BiTNode* LocateFather(BiTree T,KeyType e)
//查找双亲结点
{
    if(!T)
        return NULL;
    if(T->lchild!=NULL&&T->lchild->data.key==e||T->rchild!=NULL&&T->rchild->data.k
        return T;
    BiTNode* an;
    if(T){
        an=LocateFather(T->lchild,e);
        if(an)
            return an;
        an=LocateFather(T->rchild,e);
        if(an)
            return an;
    }
    return an;
}

status DeleteNode(BiTree *T,KeyType e)
```

//删除结点。此题允许通过增加其它函数辅助实现本关任务

```
{
    BiTree cur=LocateFather(*T,e); //找到这个节点的父节点方便进行删除操作
    BiTree root=LocateNode(*T,e); //根节点特判
    if(cur==NULL&&root==NULL)
        return ERROR; //待删除节点不存在
    BiTree l;
    if(cur==NULL){ //父节点不存在 (空树)
        l=root;
        if(l->lchild==NULL&&l->rchild==NULL){ //左右子节点都不在
            free(l);
            *T=NULL;
        }
        else if(l->lchild==NULL&&l->rchild!=NULL){ //仅有右子节点
            *T=l->rchild;
            free(l);
        }
        else if(l->rchild==NULL&&l->lchild!=NULL){ //仅有左子节点
            *T=l->lchild;
            free(l);
        }
        else if(l->lchild&&l->rchild){ //左右子节点都存在
            *T=l->lchild;
            cur=l->lchild;
            while(cur->rchild!=NULL){
                cur=cur->rchild;
            }
            cur->rchild=l->rchild;
            free(l);
        }
        return OK;
    }
```

```
}
else{
    l=cur->lchild;//l 指向左子节点
    if(l!=NULL&&l->data.key==e){
        if(l->lchild==NULL&&l->rchild==NULL){//无子结点
            free(l);
            cur->lchild=NULL;
        }
        else if(l->lchild==NULL&&l->rchild!=NULL){//仅有右子节点
            cur->lchild=l->rchild;//将右子节点放上去
            free(l);
        }
        else if(l->rchild==NULL&&l->lchild!=NULL){//仅有左子节点
            cur->lchild=l->lchild;//将左子节点及其以下向上移
            free(l);
        }
        else if(l->lchild&&l->rchild){//都有
            cur->lchild=l->lchild;//左子节点及其以下上移
            cur=cur->lchild;
            while(cur->rchild!=NULL){
                cur=cur->rchild;//右子节点上移
            }
            cur->rchild=l->rchild;
            free(l);
        }
        return OK;
    }
    l=cur->rchild;//与上面的情况类比，道理相同
    if(l!=NULL&&l->data.key==e){
        if(l->lchild==NULL&&l->rchild==NULL){
            free(l);
        }
    }
}
```

```
        cur->rchild=NULL;
    }
    else if(l->lchild==NULL&&1->rchild!=NULL){
        cur->rchild=l->rchild;
        free(l);
    }
    else if(1->rchild==NULL&&1->lchild!=NULL){
        cur->rchild=l->lchild;
        free(l);
    }
    else if(1->lchild!=NULL&&1->rchild!=NULL){
        cur->rchild=l->lchild;
        cur=cur->rchild;
        while(cur->rchild!=NULL){
            cur=cur->rchild;
        }
        cur->rchild=l->rchild;
        free(l);
    }
    return OK;
}
return ERROR;
}
}
```

//四种遍历方法

void visit(BiTree T)//访问结点

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

status PreOrderTraverse(BiTree T,void (*visit)(BiTree))

//先序遍历二叉树 T

```
{
    if(!T)
        return 0;
    if(T){
        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

```
{
    if(!T)
        return 0;
    if(T){
        InOrderTraverse(T->lchild,visit);
        visit(T);
        InOrderTraverse(T->rchild,visit);
    }
    return OK;
}
status PostOrderTraverse(BiTree T,void (*visit)(BiTree))
```

//后序遍历二叉树 T

```
{
    if(!T)
        return 0;
    if(T){
```

```
        PostOrderTraverse(T->lchild,visit);
        //visit(T);
        PostOrderTraverse(T->rchild,visit);
        visit(T);
    }
    return OK;
}

status LevelOrderTraverse(BiTree T, void (*visit)(BiTree))
{
    A[0] = T;
    if (T == NULL) return ERROR;
    else
    {
        visit(T);i++;
        if (T->lchild) { A[z] = T->lchild;z++; }
        if (T->rchild) { A[z] = T->rchild;z++; }
        LevelOrderTraverse(A[i], visit);
        return OK;
    }
}

int max(int l,int r)
{
    if(l>r)
        return l;
    if(l<=r)
        return r;
}

int MPS(BiTree T)
{
```

```
    if(!T) return 0;
    int l=MPS(T->lchild);
    int r=MPS(T->rchild);
    return max(l,r)+T->data.key;
}

status MaxPathSum(BiTree T)
//最大路径和
{
    int ans=MPS(T);
    return ans;
}

BiTNode* LowestCommonAncestor(BiTree T,KeyType e1,KeyType e2)
//最近公共祖先
{
    if(!T) return NULL;
    if(LocateNode(T->lchild,e1))
    {
        if(LocateNode(T->lchild,e2)) return LowestCommonAncestor(T->lchild,e1,e2);
        return T;
    }
    else
    {
        if(LocateNode(T->rchild,e1)) return LowestCommonAncestor(T->rchild,e1,e2);
        return T;
    }
}

status InvertTree(BiTree *T)
//翻转二叉树
```



```
{  
    if(!*T) return INFEASIBLE;  
    BiTNode*t;//交换  
    t=(*T)->lchild;  
    (*T)->lchild=(*T)->rchild;  
    (*T)->rchild=t;  
    InvertTree(&(*T)->lchild);//往子节点遍历  
    InvertTree(&(*T)->rchild);  
    return OK;  
}
```

```
status SaveBiTree(BiTree T, char FileName[])  
//将二叉树的结点数据写入到文件 FileName 中  
{if(T){  
    FILE* fp;  
    fp = fopen(FileName, "w");  
    if (T == NULL) return ERROR;  
    int key = 0;  
    char ch[10];  
    ch[0] = '#';  
    BiTree st[50];  
    int top = 0;  
    st[top++] = T;  
    BiTree p;  
    while (top != 0) {  
        p = st[--top];  
        if (p != NULL) {  
            fprintf(fp, "%d ", p->data.key);  
            fprintf(fp, "%s ", p->data.others);  
            st[top++] = p->rchild;  
            st[top++] = p->lchild;  
        }  
    }  
}
```

```
    }
    else {
        fprintf(fp, "%d ", key);
        fprintf(fp, "%s ", ch);
    }
}
fclose(fp);
return OK;
}

return INFEASIBLE;
}

status LoadBiTree(BiTree *T, char FileName[])
//读入文件 FileName 的结点数据, 创建二叉树
{
    if (*T != NULL) return ERROR;
    FILE* fp;
    fp = fopen(FileName, "r");
    int key;
    char ch[10];
    int k = 0;
    TElemType dataa[50];
    while (1) {
        if (fscanf(fp, "%d ", &key) == EOF)
            break;
        fscanf(fp, "%s ", ch);
        dataa[k].key = key;
        strcpy(dataa[k].others, ch);
        k++;
    }
    dataa[k].key = -1;
```

```
j=0;q=0;
CreateBiTree(T, dataa);
fclose(fp);
return OK;
}

status AddList(LISTS *Lists, char ListName[])
// 添加树
{

strcpy((*Lists).elem[(*Lists).length].name, ListName);
i=0;j=0;q=0;
printf("请输入带空子树的二叉树先序遍历序列: \n");
do {
scanf("%d %s", &definition[i].key, definition[i].others);
} while (definition[i++].key != -1);
flag = CreateBiTree(&(*Lists).elem[(*Lists).length].Tr, definition);
(*Lists).length += 1;
if (flag == OK)
return OK;
if (flag == ERROR)
return ERROR;
}

status change(BiTree **T, int i, LISTS *Lists) // 选择需要控制的二叉树
{
if (i <= (*Lists).length)
{(*T) = &((*Lists).elem[i-1].Tr);
num = i-1;
return OK;}
else
```

```
    return ERROR;
}

int main(void)
{
    printf("      Menu for Linear Table On Sequence Structure \n");
    printf("-----\n");
    printf("      1. CreateBiTree 创建二叉树      2. DestroyBTree 销毁二叉树\n");
    printf("      3. ClearBiTree 清空二叉树      4. BiTreeEmpty 判空二叉树\n");
    printf("      5. BiTreeDepth 二叉树深度      6. LocateNode 查找结点\n");
    printf("      7. Assign      结点赋值      8. GetSibling 获得兄弟节点\n");
    printf("      9. InsertNode 插入结点      10. DeleteNode 删除节点\n");
    printf("      11. PreOrderTraverse 前序遍历      12. InOrderTraverse 中序遍历\n");
    printf("      13. PostOrderTraverse 后序遍历      14. LevelOrderTraverse 层序遍历\n");
    printf("      15. MaxPathSum 最大路径和      16. LowestCommonAncestor 最近公共祖先\n");
    printf("      17. InvertTree 翻转二叉树      \n");
    printf("      18. SaveBiTree 写入文件      19. LoadBiTree 写出文件\n");
    printf("      20. AddBiTree      21. RemoveBiTree\n");
    printf("      22. LocateBiTree      23. TreesTraverse\n");
    printf("      24. change 选择需要控制的树\n");
    printf("      0. Exit\n");
    printf("-----\n");
    LISTs Lists;
    Lists.length=0;
    TElemType value;
    BiTree *T;
    int op=1;//选择功能
    int fl,LR;//fl 记录函数返回值
    int e;//e 用于输入查找的结点的关键字
    BiTree T1,T2;
```

```
while (op) {
    printf("请输入需要的操作: \n");
    scanf("%d", &op);
    switch (op)
    {
    case 1: {
        for(int m=0;m<i;m++)
        {
            definition[m].key=0;
            for(int w=0;definition[m].others[w]!='\0';w++)
                definition[m].others[w]='\0';
        }//重置
        i=0;//i 是全局变量, 从头遍历 definition 存入数据
        j=0;//j 是全局变量, 查找数组 gjz 里的关键字, 因此应该重置
        q=0;//q 是全局变量, 在函数中从头遍历 definition 读出数据, 应该重置
        fl=0;
        printf("请输入带空子树的二叉树先序遍历序列: \n");
        do {
            scanf("%d %s",&definition[i].key,definition[i].others);
        } while (definition[i++].key!=-1);
        fl=CreateBiTree(T,definition);
        if (fl == OK)
            printf("二叉树创建成功! \n");
        else if (fl== ERROR)
            printf("关键字不唯一\n");
        else if (fl == OVERFLOW)
            printf("溢出! \n");
        break;
    }
    case 2 :{
        fl=DestroyBTree(T);
```

```
        if(fl==OK)
            printf("已成功销毁二叉树。\\n");
        if(fl==INFEASIBLE)
            printf("不能销毁不存在的二叉树! \\n");
        break;
    }

    case 3:{
        if (ClearBiTree(T) == OK)
            printf("成功清空二叉树。\\n");
        else
            printf("不能对不存在的二叉树进行清空操作! \\n");
        break;
    }

    case 4:{
        flag=0;//标志, 若不为空会变成 1
        fl=BiTreeEmpty(*T);
        if(fl==INFEASIBLE)
            printf("二叉树不存在! \\n");
        else if(flag==0)
            printf("二叉树为空。\\n");
        else
            printf("二叉树不为空。\\n");
        break;
    }

    case 5:{
        int depth = 0;
        depth=BiTreeDepth(*T);
        if(depth==0)
```

```
printf("二叉树不存在! \n");
else
printf("二叉树的深度为%d\n", depth);
break;
}
case 6:{
    T1 = NULL;
    printf("请输入想要查找的结点的关键字: ");
    scanf("%d", &e);
    T1=LocateNode(*T, e);
    if (T1) printf("关键字为%d的结点%d, %s\n", e, T1->data.key, T1->data.c
    else printf("查找失败, 不存在关键字为%d的结点\n", e);
    break;
}

case 7:{
    fl = 0; T1 = NULL;
    printf("请输入想要修改的结点的关键字: ");
    scanf("%d", &e);
    printf("将其结点值修改为: ");
    scanf("%d %s", &value.key, value.others);
    fl = Assign(T, e, value);
    if (fl == ERROR)
        printf("赋值操作失败! \n");
    else if (fl == OK)
        printf("已将关键字为%d的结点值修改为 %d,%s\n", e, value.key, value
    break;
}

case 8:{
    T1 = NULL; T2 = NULL;
```

```
printf("请输入想要从二叉树中获得兄弟结点的关键字: ");
scanf("%d", &e);
T2=GetSibling(*T, e);
if (T2) printf("关键字为%d 的结点的兄弟结点为%d,%s\n", e, T2->data.key,
else printf("关键字为%d 的结点无兄弟结点\n", e);
break;
}

case 9:{
    flag = 0;
    T1 = NULL;
    printf("请输入插入结点的父亲的关键字: ");
    scanf("%d", &e);
    printf("插入结点作为 左孩子 (0)/右孩子 (1)/根节点 (-1): ");
    scanf("%d", &LR);
    printf("插入结点的值: ");
    scanf("%d %s", &value.key, value.others);
    fl = InsertNode(T, e, LR, value);
    if (fl == OK)
        printf("插入成功! \n");
    else if (fl == ERROR && flag == 1)
        printf("新增结点关键字重复, 插入失败! \n");
    else if (fl == ERROR && flag == 0)
        printf("无法找到父亲结点, 插入失败! \n");
    break;
}

case 10:{
    fl = 0;
    T1 = NULL; T2 = NULL;
    printf("请输入想要在线二叉树中删除的结点关键字: ");
    scanf("%d", &e);
```



```
fl = DeleteNode(T, e);
if (fl == OK)
    printf("结点关键字为%d 的结点已从二叉树中删除。\\n", e);
else if (fl == ERROR)
    printf("二叉树中不含有关键字为%d 的结点，删除失败! \\n", e);
break;
}

case 11:{
    printf("先序遍历二叉树：\\n");
    fl = PreOrderTraverse(*T, visit);
    if (fl != OK)
        printf("不能遍历不存在的二叉树! \\n");
    else printf("\\n");
    break;
}

case 12:{
    printf("中序遍历二叉树：\\n");
    fl = InOrderTraverse(*T, visit);
    if (fl != OK)
        printf("不能遍历不存在的二叉树! \\n");
    else printf("\\n");
    break;
}

case 13:{
    printf("后序遍历二叉树：\\n");
    fl = PostOrderTraverse(*T, visit);
    if (fl != OK)
        printf("不能遍历不存在的二叉树! \\n");
    else printf("\\n");
    break;
}
```

```
    }  
    case 14:{  
        printf("层序遍历二叉树: \n");  
        for (i = 0;i < 100;i++)  
            A[i] = NULL;  
        i = 0;z = 1;  
        fl = LevelOrderTraverse(*T, visit);  
        if (fl != OK)  
            printf("不能遍历不存在的二叉树! \n");  
        else printf("\n");  
        break;  
    }  
  
    case 15:{  
        fl=MaxPathSum(*T);  
        printf("最大路径和为%d.\n",fl);  
        break;  
    }  
    case 16:{  
        int e1,e2;  
        printf("请输入想要操作的两个结点的关键字: \n");  
        scanf("%d %d",&e1,&e2);  
        BiTree R=LowestCommonAncestor(*T,e1,e2);  
        if(R==NULL)  
            printf("线性表不存在! \n");  
        else  
            printf("公共祖先关键字为%d. \n",R->data.key);  
        break;  
    }  
    case 17:{  
        fl=InvertTree(T);
```

```
        if(f1==INFEASIBLE)
            printf("二叉树不存在! \n");
        if(f1==OK)
            printf("已成功翻转. \n");
        break;
    }
    case 18:{
        fl=SaveBiTree(*T,"C:\\Users\\ffsyd\\Desktop\\vscode\\ffsyd.txt");
        if(f1==INFEASIBLE)
            printf("线性表不存在! \n");
        if(f1==OK)
            printf("已成功写入文件. \n");
        break;
    }

    case 19:{
        fl=LoadBiTree(T,"C:\\Users\\ffsyd\\Desktop\\vscode\\ffsyd.txt");
        if(f1==ERROR)
            printf("线性表已存在! \n");
        if(f1==OK)
            printf("已成功录出文件. \n");
        break;
    }

    case 20:{
        char name[30];
        printf("请输入树的名字: ");
        scanf("%s",name);

        j=AddList(&Lists,name);
        if (j==OK)
        {
```

```
        printf("成功添加树\n");
    }
    else printf("失败\n");
    break;
}
case 24:{
    printf("请输入需要操作的树的序号\n");
    scanf("%d",&i);

    fl=change(&T,i,&Lists);
    if(fl==OK)
        printf("已成功切换\n");
    else
        printf("未成功切换.\n");
    break;
}
case 0:
    break;
} //end of switch

} //end of while

} //end of main()
```

附录 D 基于邻接表图实现的源程序

```
#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 MAX_VERTEX_NUM 20
#define MAX_INT 32726 //权重最大值
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;
typedef struct{ //控制链表
    struct { char name[30];
            ALGraph G;
    } elem[10];
    int length;
    int listsize;
}LISTS;
int flag = 0;
int f[20] = {};
int a[20] = { 0 };
VertexType V[10][30] = {}; //顶点数组
KeyType VR[10][100][2] = {}; //弧数组
int i = 0, j; int num;
int first =0;

void visit(VertexType v) //访问节点
{
    printf(" %d %s", v.key, v.others);
}

int check(VertexType V[], KeyType VR[][2])
{//检查关键字是否重复, 若重复返回 0, 否则返回 1
    if (V[0].key == -1)
        return 1;
    for (int i = 0; V[i].key != -1; i++)
    {
        if (i >= 20)
            return 1;
        for (int j = i + 1; V[j].key != -1; j++)
```

```
        if (V[i].key == V[j].key)
            return 1;
    }
    for (int i = 0; VR[i][0] != -1; i++) //检查弧是否重复
        for (int j = i + 1; VR[j][0] != -1; j++)
            if (VR[i][0] == VR[j][0] && VR[i][1] == VR[j][1])
                return 1;
    return 0;
}

status insert(ALGraph *G, int x, int y)
{ //录入弧
    ArcNode* p = NULL; ArcNode* p1 = NULL;
    int i = 21, j = 21;
    for (int k = 0; k < (*G).vexnum; k++) //找到结点
    {
        if (x == (*G).vertices[k].data.key)
            i = k;
        if (y == (*G).vertices[k].data.key)
            j = k;
    }
    if (i == 21 || j == 21) //如果没找到, 返回 ERROR
        return ERROR;
    if ((*G).vertices[i].firstarc == NULL) //判断节点没有第一弧
    {
        (*G).vertices[i].firstarc = (ArcNode*)malloc(sizeof(ArcNode)); //分配空间
        if ((*G).vertices[i].firstarc != NULL)
        {
            (*G).vertices[i].firstarc->adjvex = j; //弧 i 指向 j
            (*G).vertices[i].firstarc->nextarc = NULL; //暂时无下一指针
        }
    }
}
```

```
else//有第一个弧
{
    if (!(p = (ArcNode*)malloc(sizeof(ArcNode))))
        return OVERFLOW;
    p->nextarc = (*G).vertices[i].firstarc;//插入新弧
    (*G).vertices[i].firstarc = p;
    p->adjvex = j;//弧 i 指向 j
}
if ((*G).vertices[j].firstarc == NULL)
{
    (*G).vertices[j].firstarc = (ArcNode*)malloc(sizeof(ArcNode));
    if ((*G).vertices[j].firstarc != NULL) {
        (*G).vertices[j].firstarc->adjvex = i;//弧 j 指向 i
        (*G).vertices[j].firstarc->nextarc = NULL;//暂时无下一指针
    }
}
else
{
    if (!(p = (ArcNode*)malloc(sizeof(ArcNode))))
        return OVERFLOW;
    p->nextarc = (*G).vertices[j].firstarc;//插入新弧
    (*G).vertices[j].firstarc = p;
    p->adjvex = i;
}
(*G).arcnum++;
return OK;
}

status CreateCraph(ALGraph *G,VertexType V[],KeyType VR[][2])
/*根据 V 和 VR 构造图 T 并返回 OK, 如果 V 和 VR 不正确, 返回 ERROR
如果有相同的关键字, 返回 ERROR. */
{
```



```
flag = check(V, VR); //检查顶点和弧是否重复
if (flag == 1)
return ERROR;
else
{
    int i = 0;
    for (i = 0; V[i].key != -1; i++)
    {
        (*G).vertices[i].data.key = V[i].key;
        strcpy((*G).vertices[i].data.others, V[i].others); //录入顶点
        (*G).vertices[i].firstarc = NULL; //弧全挂空
    }
    (*G).vexnum = i;
    (*G).arcnum = 0;
    for (i = 0; VR[i][0] != -1; i++)
    {
        int s = insert(G, VR[i][0], VR[i][1]); //录入弧
        if (s == ERROR)
            return ERROR;
    }
    return OK;
}
}

status DestroyGraph(ALGraph *G)
/*销毁无向图 G, 删除 G 的全部顶点和边*/
{
    if ((*G).vexnum == 0) //无顶点, 返回错误
        return ERROR;
    (*G).arcnum = 0;
    ArcNode* p1, * p2;
    int i = 0, j;
    for (i = 0; i < (*G).vexnum; i++) //销毁弧
```

```
{
    if ((*G).vertices[i].firstarc != NULL) //有弧
        p1 = (*G).vertices[i].firstarc;
    else
        p1 = NULL;
    do
    {
        p2 = p1;
        if(p1!=NULL){
            if (p1->nextarc == NULL)
                p1 = NULL;
            else
                p1 = p1->nextarc;
            free(p2); //释放
        }
    } while (p1);
    (*G).vertices[i].firstarc = NULL; //挂空
}

(*G).vexnum = 0;
return OK;
}

int LocateVex(ALGraph G,KeyType u)
//根据 u 在图 G 中查找顶点，查找成功返回位序，否则返回-1;
{
    int i = 0;
    while (i < G.vexnum)
    {
        if (u == G.vertices[i].data.key)
            return i;
        i++;
    }
}
```

```
    return -1;
}

status PutVex(ALGraph *G,KeyType u,VertexType value)
//根据  $u$  在图  $G$  中查找顶点, 查找成功将该顶点值修改成  $value$ , 返回  $OK$ ;
//如果查找失败或关键字不唯一, 返回  $ERROR$ 
{
    int i = 0, num = -1;
    while (i < (*G).vexnum)
    {
        if (u == (*G).vertices[i].data.key)
            num = i;
        if (value.key == (*G).vertices[i].data.key && value.key != u) //检查关键字
            return ERROR;
        i++;
    }
    if (num == -1)
        return ERROR;
    (*G).vertices[num].data = value; //修改
    return OK;
}

int FirstAdjVex(ALGraph G,KeyType u)
//根据  $u$  在图  $G$  中查找顶点, 查找成功返回顶点  $u$  的第一邻接顶点位序, 否则返回  $-1$ ;
{
    int i = 0, num = -1;
    while (i < G.vexnum)
    {
        if (u == G.vertices[i].data.key) //找到邻接点
        {
            num = i;
        }
    }
}
```

```
        break;
    }
    i++;
}

if (num == -1 || G.vertices[num].firstarc == NULL)
    return -1;
else
    return G.vertices[num].firstarc->adjvex;
}

int NextAdjVex(ALGraph G,KeyType v,KeyType w)
//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
{
    int i = 0, num1 = -1, num2 = -1;
    while (i < G.vexnum)
    {
        if (v == G.vertices[i].data.key) num1 = i; //找到邻接点
        if (w == G.vertices[i].data.key) num2 = i; //下一个邻接点
        i++;
    }
    if (num1 == -1 || num2 == -1)
        return -1;
    if (num1 == num2 || G.vertices[num1].firstarc == NULL) //不存在
        return -1;
    ArcNode* p1;
    p1 = G.vertices[num1].firstarc;
    while (p1->adjvex != num2) {
        p1 = p1->nextarc;
        if (p1 == NULL)
            return -1;
    }
    if (p1->nextarc == NULL)
```

```
    return -1;
else
    return p1->nextarc->adjvex;
}

status InsertVex(ALGraph *G,VertexType v)
//在图 G 中插入顶点 v, 成功返回 OK, 否则返回 ERROR
{
    int i = 0;
    while (i < (*G).vexnum) {
        if (v.key == (*G).vertices[i].data.key) //检查关键字是否重复
            return ERROR;
        i++;
        if (i >= 20)
            return ERROR;
    }
    (*G).vertices[i].data = v; //输入顶点
    (*G).vertices[i].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
{
    int i = 0, num = -1;
    while (i < (*G).vexnum) { //找到顶点
        if (v == (*G).vertices[i].data.key)
            num = i;
        i++;
    }
    if (num == -1) //未找到顶点
```

```
return ERROR;
if (num != -1 && (*G).vexnum == 1)
return ERROR;
ArcNode* p1, * p2;
for (i = 0; i < (*G).vexnum; i++)
{
    p1 = (*G).vertices[i].firstarc;
    p2 = p1;
    if (p1 == NULL) //没有弧
        continue;
    if (i == num) //已遍历到最后一个顶点
    {
        do{
            p2 = p1;
            p1 = p1->nextarc;
            free(p2); //释放
        } while (p1);
        (*G).vertices[i].firstarc = NULL;
        continue;
    }
    else
    {
        if (p1->adjvex == num) //改弧与顶点 v 关联
        {
            (*G).vertices[i].firstarc = p1->nextarc;
            free(p1);
            (*G).arcnum--;
        }
        else
        {
            p1 = p1->nextarc; //向相邻弧遍历
        }
    }
}
```

```
        while (p1)
        {
            if (p1->adjvex == num) //改弧与顶点  $v$  关联
            { p2->nextarc = p1->nextarc;
              free(p1);
              (*G).arcnum--;
              break;
            }
            else
            { p1 = p1->nextarc;
              p2 = p2->nextarc; }
        }
    }
}

for (i = num; i < (*G).vexnum; i++) //删除顶点
    (*G).vertices[i] = (*G).vertices[i + 1];
(*G).vexnum--;
for (i = 0; i < (*G).vexnum; i++)
{
    p1 = (*G).vertices[i].firstarc;
    while (p1)
    {
        if (p1->adjvex > num) p1->adjvex--; //修改弧中记录的顶点序号
        p1 = p1->nextarc;
    }
}

return OK;
}

status InsertArc(ALGraph *G, KeyType v, KeyType w)
//在图  $G$  中增加弧  $\langle v, w \rangle$ , 成功返回 OK, 否则返回 ERROR
```

```
{
int i = 0, num1 = -1, num2 = -1;
while (i < (*G).vexnum)
{ //找到两个顶点
    if (v == (*G).vertices[i].data.key)
        num1 = i;
    if (w == (*G).vertices[i].data.key)
        num2 = i;
    i++;
}

if (num1 == -1 || num2 == -1 || num1 == num2)
return ERROR; //未找到顶点, 或顶点一样

ArcNode* p1, * p2;

p1 = (*G).vertices[num1].firstarc;
while (p1) //插入弧
{
    if (p1->adjvex == num2)
        return ERROR;
    p1 = p1->nextarc;
}

p1 = (*G).vertices[num1].firstarc;
p2 = (ArcNode*)malloc(sizeof(ArcNode));
(*G).vertices[num1].firstarc = p2; //弧 v 指向 w
p2->nextarc = p1;
p2->adjvex = num2;

p1 = (*G).vertices[num2].firstarc;
p2 = (ArcNode*)malloc(sizeof(ArcNode));
```



```
(*G).vertices[num2].firstarc = p2; //弧 w 指向 v
p2->nextarc = p1;
p2->adjvex = num1;

(*G).arcnum++;
return OK;
}

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 pos1,pos2,a=0;
    ArcNode *p,*q;
    pos1=LocateVex(*G,v);
    pos2=LocateVex(*G,w);
    if(pos1==-1||pos2==-1) return ERROR; //关键字不存在
        //删一个顶点的弧
    p=(*G).vertices[pos1].firstarc;
    q=p;
    if(p==NULL) return ERROR; //有一个顶点为空, 即弧不存在
        if(p->adjvex==pos2){
            (*G).vertices[pos1].firstarc=p->nextarc;
            a=1;
            free(p);
        }
        else{
            p=p->nextarc;
            while(p){
                if(p->adjvex==pos2)
                {
                    q->nextarc=p->nextarc;
                    a=1;
                }
            }
        }
}
```

```
        free(p);
        break;
    }

    q=p;
    p=p->nextarc;
}

}

//删另一个顶点的弧
p=(*G).vertices[pos2].firstarc;
q=p;
if(p==NULL) return ERROR;
    if(p->adjvex==pos1){
        (*G).vertices[pos2].firstarc=p->nextarc;
        free(p);
    }
    else{
        p=p->nextarc;
        while(p){
            if(p->adjvex==pos1)
            {
                q->nextarc=p->nextarc;

                free(p);
                break;
            }

            q=p;
            p=p->nextarc;
        }
    }

    if(a==0) return ERROR; //没有该弧
    (*G).arcnum--;
return OK;
```

```
}
```

```
void DFS(ALGraph *G, int num, void (*visit)(VertexType))
```

```
{
```

```
    if (f[num]) return;
```

```
    ArcNode* p = NULL;
```

```
    f[num] = 1;
```

```
    visit((*G).vertices[num].data);
```

```
    p = (*G).vertices[num].firstarc;
```

```
    while (p) {
```

```
        num = p->adjvex; //先将该点后面的相邻的弧遍历完
```

```
        DFS(G, num, visit);
```

```
        p = p->nextarc;
```

```
    }
```

```
    return;
```

```
}
```

```
status DFSTraverse(ALGraph *G, void (*visit)(VertexType))
```

```
//对图 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
```

```
{
```

```
int i;
```

```
    for (i = 0; i < (*G).vexnum; i++) {
```

```
        DFS(G, i, visit);
```

```
    }
```

```
    return OK;
```

```
}
```

```
status BFSTraverse(ALGraph *G, void (*visit)(VertexType))
```

```
//对图 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
```

```
{
```

```
int i = 0;
```

```
    ArcNode* p1;
```

```
    while (i < (*G).vexnum)
```

```
{
    if (a[i] == 0) { visit((*G).vertices[i].data); a[i] = 1; }
    p1 = (*G).vertices[i].firstarc;
    while (p1)
    {
        if (a[p1->adjvex] == 0)
        { visit((*G).vertices[p1->adjvex].data);
          a[p1->adjvex] = 1; }
        p1 = p1->nextarc; //换到弧指向的点
    }
    i++;
}
return OK;
}
```

```
status SaveGraph(ALGraph G, char FileName[])
//将图的数据写入到文件 FileName 中
{
    FILE* fp;
    if ((fp = fopen(FileName, "w")) == NULL) return ERROR;
    fprintf(fp, "%d ", G.vexnum);
    fprintf(fp, "%d ", G.arcnum); //保存顶点数与弧数
    for (int i = 0; i < G.vexnum; i++) { //保存结点
        fprintf(fp, "%d ", G.vertices[i].data.key);
        fprintf(fp, "%s ", G.vertices[i].data.others);
    }
    int keyflag = -1;
    char othersflag[10] = "nil";
    fprintf(fp, "%d ", keyflag); //顶点结束
    fprintf(fp, "%s ", othersflag);
    int stack[50][2];
```

```
int top = 0;
int key1 = 0, key2 = 0;
ArcNode* p = NULL;
for (int i = 0; i < G.vexnum; i++) {
    p = G.vertices[i].firstarc;
    while (p) {
        stack[top][0] = G.vertices[i].data.key;
        stack[top][1] = G.vertices[p->adjvex].data.key;
        top++;
        p = p->nextarc;
    }
    while (top != 0) {
        key1 = stack[top - 1][0];
        key2 = stack[top - 1][1];
        top--;
        fprintf(fp, "%d ", key1);
        fprintf(fp, "%d ", key2);
    }
}
fprintf(fp, "%d ", keyflag); //弧结束
fprintf(fp, "%d ", keyflag);
fclose(fp);
return OK;
}

status createCraph(ALGraph *G, VertexType V[], KeyType VR[][2])
{
    int i = 0, j = 0;
    ArcNode* p;
    VertexType v;
    (*G).vexnum = 0;
    (*G).arcnum = 0;
```

```
int arrkey[50] = { 0 };
v.key = V[0].key;
strcpy(v.others, V[0].others);
while (v.key != -1) {
    for (int m = 0; m < (*G).vexnum; m++)
    {
        if (v.key == arrkey[m])
            return ERROR;
    }
    (*G).vertices[(*G).vexnum].data = v;
    (*G).vertices[(*G).vexnum].firstarc = NULL;
    arrkey[(*G).vexnum] = v.key;
    (*G).vexnum++;
    i++;
    v.key = V[i].key;
    strcpy(v.others, V[i].others);
}
if ((*G).vexnum > 20 || (*G).vexnum < 1)
return ERROR;
int a, b;
int k = 0;
a = VR[0][0];
b = VR[0][1];
while (a != -1 && b != -1) {
    (*G).arcnum++;
    int loca = LocateVex(*G, a);
    int locb = LocateVex(*G, b);
    if (loca == -1 || locb == -1) return ERROR;

    p = (ArcNode*)malloc(sizeof(ArcNode));
    p->nextarc = (*G).vertices[loca].firstarc;
```

```
p->adjvex = locb;
(*G).vertices[loca].firstarc = p;

k++;
a = VR[k][0];
b = VR[k][1];
}
return OK;
}

status LoadGraph(ALGraph *G, char FileName[])
//读入文件 FileName 的图数据, 创建图的邻接表
{
if ((*G).vexnum != 0) return INFEASIBLE;
FILE* fp;
fp = fopen(FileName, "r");
int vexnum = -1, arcnum = -1, k = 0;
VertexType V[25];
KeyType VR[200][2];
int vis[25][25] = { 0 };
int key1 = -1, key2 = -1;
fscanf(fp, "%d ", &vexnum);
fscanf(fp, "%d ", &arcnum);
for (int i = 0; i <= vexnum; i++) {
    fscanf(fp, "%d ", &V[i].key);
    fscanf(fp, "%s ", V[i].others);
}
fscanf(fp, "%d ", &key1);
fscanf(fp, "%d ", &key2);
while (key1 != -1 && key2 != -1) {
    VR[k][0] = key1;
    VR[k][1] = key2;
```

```
k++;
fscanf(fp, "%d ", &key1);
fscanf(fp, "%d ", &key2);
}
createCraph(G, V, VR);
return OK;
}

int ShortestPathLength(ALGraph G,int v,int w)//最短路径
{
    if(G.vexnum == 0 ) //图不存在
    {
        return INFEASIBLE;
    }
    int head = 0 ,tail =0;
    int record[100]={0};
    int arr[100][2];
    memset(arr,0,sizeof(arr));
    int i = 0; int flag = -1;
    int flagw = -1;
    for( ; i<G.vexnum ;i++)
    {
        if(G.vertices[i].data.key == v)
        {
            flag = i;//记录顶点 1
        }
        if(G.vertices[i].data.key == w)
        {
            flagw = i;//记录顶点 2
        }
    }
}
```



```
if(flag == -1 || flagw == -1)//顶点不存在
{
    return INFEASIBLE;
}
arr[head][0] = flag;

while (head <= tail)
{
    ArcNode *p = G.vertices[arr[head][0]].firstarc;
    if(G.vertices[arr[head][0]].data.key == w)
    {
        return arr[head][1];
    }
    while (p)
    {
        if(record[p->adjvex] == 0)
        {
            tail++;
            arr[tail][0] = p->adjvex;
            arr[tail][1] = arr[head][1]+1;//路径长度加一
            record[arr[head][0]]++;
        }
        p = p->nextarc;
    }
    head++;
}

return -1;
}

int flag16[100] = {0};
void dfs(ALGraph G, int nownode)
```

```
{
    flag16[nownode]=1;
    ArcNode *p =G.vertices[nownode].firstarc;
    while (p)
    {
        if(flag16[p->adjvex] == 0)
        {
            dfs(G,p->adjvex);
        }
        p = p->nextarc ;
    }
}

int ConnectedComponentsNums(ALGraph G)
{//连通分支数
    memset(flag16,0,sizeof(flag16)); //每次使用之前要清空
    int i ;
    if(G.vexnum == 0)
    {
        printf("为初始化或者为空\n");
        return 0;
    }
    int count = 0;
    for(i = 0; i<G.vexnum ;i++)
    {
        if(flag16[i] == 0)
        {
            count++;
            dfs(G,i);
        }
    }
}
```

```
    return count;
}

status VerticesSetLessThanK(ALGraph G,KeyType v,int 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;
{
    int a=1,flag[500],i,j,q[500],z=0,temp[500],s,m[200]={0};
    int pos=LocateVex(G,v);
    if(pos==-1||k<=1) return ERROR;
    for(i=0;i<500;i++) {flag[i]=-1;q[i]=-1;temp[i]=-1;}
    flag[0]=pos;
    while(a<k)//先从距离为 1 的找起, 距离逐渐增大
    {
        i=0;j=0;
        while(flag[i]!=-1){
            ArcNode *p=G.vertices[flag[i++]].firstarc;
            while(p){
                q[z++]=p->adjvex;
                temp[j++]=p->adjvex;
                p=p->nextarc;
            }
        }
        for(s=0;s<j;s++) flag[s]=temp[s];//把顶点序号存到 flag 里, 从 0 位置
        flag[s]=-1;
        a++;
    }
    if(z==0) printf("该顶点无相连顶点\n");
    for(a=0;a<z;a++){
        if(q[a]!=pos&&temp[a]==0){//原顶点和已经输出过的顶点不输出
            m[q[a]]=1;
            printf("%d %s ",G.vertices[q[a]].data.key,G.vertices[q[a]]
```

```
        }  
    }  
    return OK;  
}
```

```
status AddList(LISTS *Lists, char ListName[])
```

```
// 添加图
```

```
{    if(first==1) num+=1;  
    strcpy((*Lists).elem[(*Lists).length].name, ListName);  
    printf("请输入顶点序列, 以-1 nil 结束: ");  
    do{  
        scanf("%d%s", &V[num][i].key, V[num][i].others); //存入顶点关键字  
    }while (V[num][i++].key != -1);  
    i = 0;  
    printf("请输入关系对序列, 以-1 -1 结束: ");  
    do{  
        scanf("%d%d", &VR[num][i][0], &VR[num][i][1]); //存入弧  
    }while (VR[num][i++][0] != -1);  
    flag=CreateCraph(&(*Lists).elem[(*Lists).length].G, V[num], VR[num]);  
    (*Lists).length+=1;  
    if(flag==OK)  
        return OK;  
    if(flag==ERROR)  
        return ERROR;  
}
```

```
status change(ALGraph **G, int i, LISTS *Lists) //选择需要控制的图 18
```

```
{  
    first=1;  
    if(i<=(*Lists).length)
```

```
{(*G)=&((*Lists).elem[i-1].G);
num=i-1;
return OK;}
else
return ERROR;
}
int main(void)
{
    LISTS Lists;
    Lists.length=0;
    ALGraph *G;

    VertexType value;

    int op = 1, length, e, pre, next, flag1 = 0;
    char FileName[30] = { '\0' };
    char ListName[30] = { '\0' };
    printf("\n\n");
    printf("      Menu for Linear Table On Sequence Structure \n");
    printf("-----\n");
    printf("      1. CreateGraph 创建图          2. DestroyGraph 销毁图\n");
    printf("      3. LocateVex 查找顶点          4. PutVex 顶点赋值\n");
    printf("      5. FirstAdjVex 第一邻接点      6. NextAdjVex 下一邻接点\n");
    printf("      7. InsertVex 插入顶点          8. DeleteVex 删除顶点\n");
    printf("      9. InsertArc 插入弧            10.DeleteArc 删除弧\n");
    printf("      11.DFSTraverse 深度优先        12.BFSTraverse 广度优先\n");
    printf("      13.SaveGraph 存入文件          14.LoadGraph 写出文件\n");
    printf("      15.ShortestPathLength 最短路径  16.ConnectedComponentsNums 连通分支\n");
    printf("      17.VerticesSetLessThanK 距离小于 k 18.AddGraph 添加图\n");
    printf("      19.Change 切换\n");
```

```
printf("    0. Exit\n");
printf("-----\n");
printf("\n 请选择你的操作 [0~19]:\n");
while (op) {
    scanf("%d", &op);
    switch (op) {
        case 1:
            {i = 0;
            printf("请输入顶点序列, 以-1 nil 结束: ");
            do{
                scanf("%d%s", &V[num][i].key, V[num][i].others); //存入顶点关键字
            }while (V[num][i++].key != -1);
            i = 0;
            printf("请输入关系对序列, 以-1 -1 结束: ");
            do{
                scanf("%d%d", &VR[num][i][0], &VR[num][i][1]); //存入弧
            }while (VR[num][i++][0] != -1);
            if (CreateCraph(G, V[num], VR[num]) == OK)
                printf("图创建成功! \n");
            else
                printf("图创建失败! \n");
            break;}
        case 2:
            {if (DestroyGraph(G) == OK)
                printf("成功销毁图并释放数据元素的空间。 \n");
            else
                printf("不能对不存在的图进行销毁操作! \n");
            break;}
        case 3:
            {printf("请输入想要在图中查找的顶点的关键字: ");
            scanf("%d", &e);
```

```
i = LocateVex(*G, e);
if (i != -1) printf("图中关键字为%d 的顶点的位序为%d\n", e, i);
else
    printf("图中不存在关键字为%d 的顶点! \n", e);
break;}

case 4:
{printf("请输入想要修改的顶点的关键字: ");
scanf("%d", &e);
printf("将其顶点值修改为: ");
scanf("%d %s", &value.key, value.others);
flag1 = PutVex(G, e, value);
if (flag1 == ERROR)
    printf("赋值操作失败! \n");
else if (flag1 == OK)
    printf("已将关键字为%d 的顶点值修改为%d,%s\n", e, value.key, value.others);
break;}

case 5:
{printf("请输入想要查找其第一邻接点的顶点: ");
scanf("%d", &e);
i = FirstAdjVex(*G, e);
if (i != -1)
    printf("顶点%d 的第一邻接点的位序为%d\n", e, i);
else
    printf("顶点%d 没有第一邻接点! \n", e);
break;}

case 6:
{printf("请输入两个顶点的关键字: ");
scanf("%d %d", &e, &j);
i = NextAdjVex(*G, e, j);
if (i != -1)
    printf("顶点%d 相对于顶点%d 的下一个邻接顶点为%d %s\n", e, j, (*G).v[i].key, (*G).v[i].others);
else
    printf("顶点%d 没有下一个邻接顶点! \n", e);
break;}
```

```
        else printf("无下一邻接顶点! \n");
        break;}
case 7:
    {printf("请输入想要插入的顶点值: ");
    scanf("%d %s", &value.key, value.others);
    flag1 = InsertVex(G, value);
    if (flag1 == OK)
        printf("顶点 %d %s 已成功插入图中\n", value.key, value.others);
    else if (flag == ERROR)
        printf("插入失败! \n");
    break;}
case 8:
    {printf("请输入想要删除的顶点的关键字: ");
    scanf("%d", &e);
    flag1 = DeleteVex(G, e);
    if (flag1 == OK)
        printf("关键字为%d的顶点已从图中删除\n", e);
    else if (flag1 == ERROR)
        printf("删除失败! \n");
    break;}
case 9:
    {printf("请输入想要插入的弧: ");
    scanf("%d %d", &e, &j);
    flag1 = InsertArc(G, e, j);
    if (flag1 == OK)
        printf("插入成功! \n");
    else if (flag1 == ERROR)
        printf("插入失败! \n");
    break;}
case 10:
    printf("请输入想要删除的弧: ");
```



```
scanf("%d %d", &e, &j);
flag1 = DeleteArc(G, e, j);
if (flag1 == OK)
    printf("删除成功! \n");
else if (flag1 == ERROR)
    printf("删除失败! \n");
break;
case 11:
    for (int m = 0; m < 20; m++)
        f[m] = 0;
    printf("深度优先搜索遍历: \n");
    DFSTraverse(G, visit);
    printf("\n");
    break;
case 12:
    for (int m = 0; m < 20; m++)
        a[m] = 0;
    printf("广度优先搜索遍历: \n");
    BFSTraverse(G, visit);
    printf("\n");
    break;
case 13:
    {printf("请输入文件名称: ");
    scanf("%s", FileName);
    flag1 = SaveGraph(*G, FileName);
    if (flag1 == ERROR)
        printf("文件打开失败! \n");
    else if (flag1 == OK)
        printf("图的内容已经复制到名称为 %s 的文件中。 \n", FileName);
    break;}
case 14:
```

```
{printf("请输入文件名称: ");
scanf("%s", FileName);
flag1 = LoadGraph(G, FileName);
if (flag1 == ERROR)
    printf("文件打开失败! \n");
else if (flag1 == OK)
    printf("名称为 %s 的文件中的内容已复制到图中。 \n", FileName);
else if (flag1 == INFEASIBLE)
    printf("不能对已存在的图进行进行读文件操作。 \n");
else if (flag1 == OVERFLOW)
    printf("溢出! \n");
break;}
case 15:
    printf("请输入顶点 v 和顶点 w 的关键字\n");
    int v15,w15;
    scanf("%d %d",&v15,&w15);
    int feedback = ShortestPathLength(*G,v15,w15);
    if(feedback==-1)
    {
        printf("该图或顶点不存在或未初始化\n");
        break;
    }
    if(feedback != -1)
    {
        printf("最短路径为 %d \n",feedback);
    } else{
        printf("操作失败\n");
    }
    break;
case 16:
```

```
{
    flag1=ConnectedComponentsNums( *G);
    if(flag1!=0)
        printf("图的连通分量为: %d\n",flag1);
        break;
}
case 17:
{
    printf("请输入想要查找的顶点和长度: \n");
    int s,k;
    scanf("%d %d",&s,&k);
    printf("与顶点%d 距离小于%d 的顶点有: ",s,k);
    VerticesSetLessThanK( *G, s, k);
    printf("\n");
    break;
}
case 19:{
    printf("请输入需要操作的图的序号\n");
    scanf("%d",&i);
    flag1=change(&G,i,&Lists);
    if(flag1==OK)
        printf("已成功切换\n");
    else
        printf("未成功切换.\n");
    break;
}
case 18:{
    char name[30];
    printf("请输入图的名字: ");
    scanf("%s",name);
```

```
j=AddList(&Lists,name);
if (j==OK)
{
    printf("成功添加图\n");
}
else printf("失败\n");
break;
}
case 0:
    break;
} //end of switch
printf("\n");
} //end of while
printf("欢迎下次再使用本系统! \n");
} //end of main()
```