華中科技大學

课 程 实 验 报 告

课程名称： 数据结构实验

专业班级 CS2407

学　　号 U202414729

姓　　名 唐淼

指导教师 郝义学

报告日期 2025 年 6 月 10 日

计算机科学与技术学院

目　　录

1 基于链式存储结构的线性表实现

1.1 问题描述

本实验旨在通过链式存储结构实现线性表的基本操作，包括线性表的创建、销毁、插入、删除、查找、遍历等。通过实验，深入理解链式存储结构的特点及其在动态内存管理中的优势，掌握链表的基本操作及其应用场景。

1.2 系统设计

1) 整体系统结构设计

本系统采用模块化设计，主要分为两大功能板块：

(a) **单链表操作模块**

i. 提供线性表的基本操作功能，包括创建、销毁、插入、删除、查找等

ii. 通过菜单驱动方式与用户交互

iii. 每个功能对应独立的函数实现

(b) **多链表管理模块**

i. 使用带头结点的链表结构管理多个线性表

ii. 支持线性表的添加、删除、查找和选择操作

iii. 维护线性表名称与对应链表的关系

2) 数据结构设计

(a) 单链表结点结构

```
typedef struct LNode {
    ElemType data;        // 数据域，存储int类型元素
    struct LNode *next;   // 指针域，指向下一个结点
} LNode, *LinkList;
```

图 1-1 单链表节点结构

(b) 多链表管理结构

```c
typedef struct ListNode {
    struct {
        char name[20];    // 线性表名称
        LinkList L;       // 指向对应单链表
    } List;
    struct ListNode *next; // 指向下一个管理结点
} ListsNode;

typedef struct {
    ListNode *head;       // 管理链表的头指针
    ListNode *tail;       // 管理链表的尾指针
    int length;           // 管理的线性表数量
} LISTS;
```

图 1-2　多链表管理结构

(c) 状态定义

```c
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
typedef int status;       // 用于返回函数执行状态
```

图 1-3　状态定义

3) 关键设计特点

(a) **动态内存管理**:

i. 所有链表结点均动态分配内存

ii. 配套完善的销毁机制防止内存泄漏

(b) **错误处理机制**:

i. 统一的状态返回值规范

ii. 对非法操作进行严格检查

(c) **扩展性设计**:

i. 管理链表采用链式结构，可动态扩展

ii. 数据域使用 typedef 定义，便于修改元素类型

(d) **用户友好性**：

i. 分级菜单导航

ii. 操作结果即时反馈

1.3 系统实现

1.3.1 函数总览

函数名	时间复杂度	空间复杂度	功能简述
InitList	O(1)	O(1)	初始化空链表
DestroyList	O(n)	O(1)	销毁链表，释放内存
ClearList	O(n)	O(1)	清空链表元素，保留头节点
ListEmpty	O(1)	O(1)	判断链表是否为空
ListLength	O(n)	O(1)	获取链表长度
GetElem	O(n)	O(1)	获取第 i 个元素
LocateElem	O(n)	O(1)	查找元素值为 e 的位置
PriorElem	O(n)	O(1)	获取元素 e 的前驱
NextElem	O(n)	O(1)	获取元素 e 的后继
ListInsert	O(n)	O(1)	插入元素到第 i 个位置前
ListDelete	O(n)	O(1)	删除第 i 个元素
ListTraverse	O(n)	O(1)	遍历并打印链表元素
SaveList	O(n)	O(1)	保存链表到文件
LoadList	O(n)	O(n)	从文件读取链表数据
ReverseList	O(n)	O(1)	反转链表
RemoveNthFromEnd	O(n)	O(1)	删除倒数第 i 个元素
SortList	$O(n^2)$	O(1)	使用冒泡排序对链表排序
AddList	O(m)	O(1)	添加一个新的空链表
LocateList	O(m)	O(1)	查找链表在多链表结构中的位置
RemoveList	O(m + n)	O(1)	删除多链表结构中的一个链表
displayAllLists	O(m)	O(1)	打印所有链表名称

表 1-1　链表相关函数总览表

1.3.2 函数简介

1) 展示菜单：

(a) show1() 思想：显示主菜单，提供单线性表操作、多线性表管理和退出系统的选项。

关键点：

- 使用格式化输出清晰展示功能分区
- 提示用户输入范围 [0 2]
- 特别说明交互特性（需输入字符后重新显示菜单）

(b) show2()

思想：显示单线性表操作菜单，包含初始化、销毁、插入、删除、遍历等 17 种操作选项。

关键点：

- 将功能分类排列（初始化类、查询类、修改类等）
- 采用两列布局提高可读性
- 明确输入范围 [0 17]

(c) show3()

思想：显示多线性表管理菜单，提供添加、删除、查找和操作单个线性表的功能。

关键点：

- 简化菜单项
- 保持与主菜单一致的交互提示
- 突出”操作单个线性表”的二级菜单特性

2) 单线性表操作函数

(a) InitList(LinkList L)

思想：动态分配头结点，next 指针置空，表示空表。

关键点：

- 通过头结点存在性判断避免重复初始化
- 内存分配失败时应返回 ERROR（当前未处理）
- 头结点的 data 域不需要初始化（仅作为哨兵节点）

(b) DestroyList(LinkList L)

思想：从头结点开始遍历，逐个释放所有结点内存（包括头结点）。

关键点：

- 使用临时指针保存下一节点地址，防止内存访问冲突
- 最后将 L 置 NULL 避免野指针
- 与 ClearList 的区别在于是否保留头结点

(c) ClearList(LinkList L)

思想：从头结点开始遍历，逐个释放所有结点内存，保留头结点。

关键点：

- 保留头结点保持链表结构有效性
- 需要将头结点的 next 置 NULL
- 时间复杂度 O(n) 不可避免

(d) ListEmpty(LinkList L)

思想：检查线性表是否为空，即判断头结点指针域是否为空。

关键点：

- 仅检查头结点 next 指针，不移动指针
- 比 ListLength 效率更高（不需要遍历）
- 与 NULL 的比较必须严格

(e) ListLength(LinkList L)

思想：从头结点开始遍历，直至指针域为空。

关键点：

- 计数器从 0 开始（忽略头结点）
- 每次循环严格检查 temp!=NULL
- 时间复杂度 O(n) 不可避免

(f) GetElem(LinkList L, int i, ElemType e)

思想：通过遍历链表，找到第 i 个位置的结点，并将其数据域的值赋给 e。

关键点：

- 位置 i 的合法性检查（i<1 直接返回错误）
- 使用临时指针遍历，不修改原链表指针
- 循环终止条件需同时检查 j 和 temp

(g) LocateElem(LinkList L, ElemType e)

思想：遍历链表，查找数据域等于 e 的第一个结点，返回其位置序号。

关键点：

- 位置计数从 1 开始（符合直观）
- 仅返回第一个匹配项
- 比较使用 == 运算符，要求 ElemType 可比较

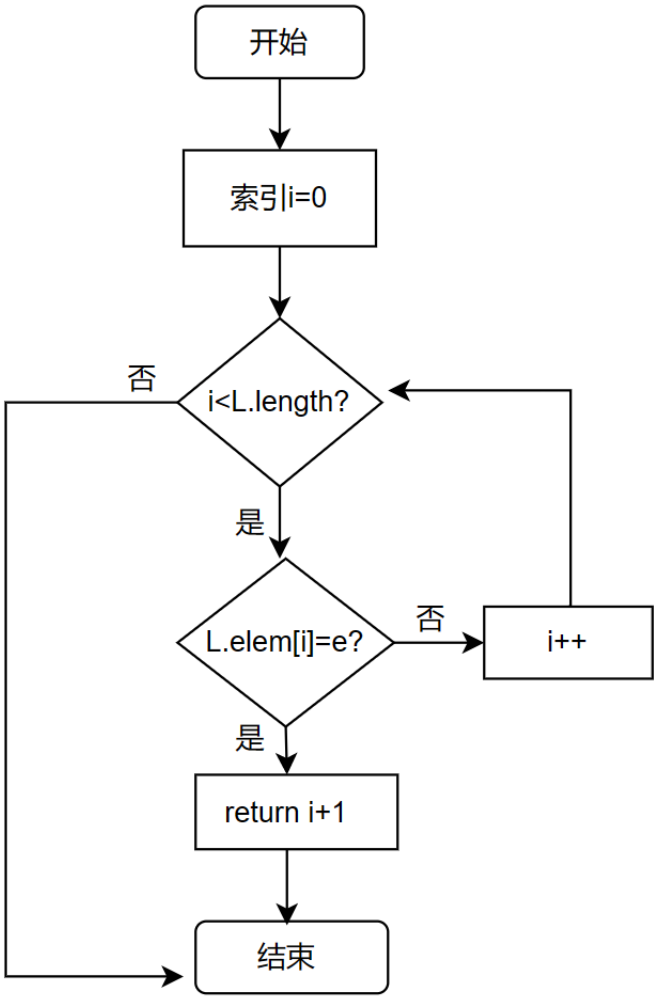


图 1-4　查找元素流程图

(h) PriorElem

思想：双指针法，前驱指针 Pre 始终跟随当前指针 temp，直到找到目标值。

关键点：

- 需要单独处理首元素无前驱的情况
- 双指针需保持同步移动

- 边界条件：空表/单元素表/目标为首元素

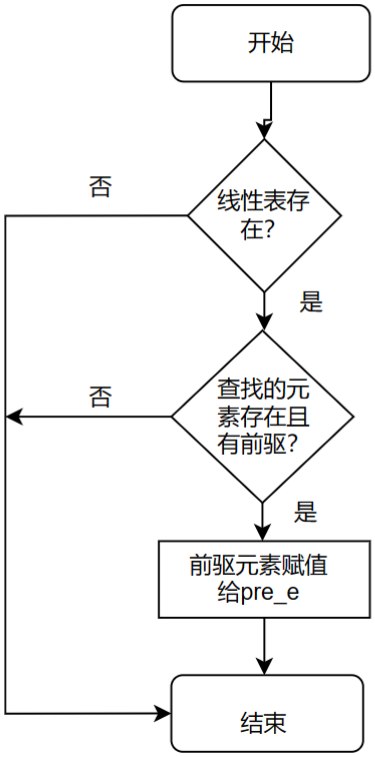


图 1-5　获得元素前驱流程图

(i) NextElem

思想：单指针遍历，找到目标值后直接访问其 next 指针。

关键点：

- 需要检查 next 是否为 NULL
- 仅返回直接后继
- 边界条件：末元素无后继

(j) ListInsert(LinkList L, int i, ElemType e)

思想：找到第 i-1 个结点，将新结点插入其后。

关键点：

- 需要处理插入位置为 1 的特殊情况
- 内存分配失败处理
- 插入后保持链表连续性

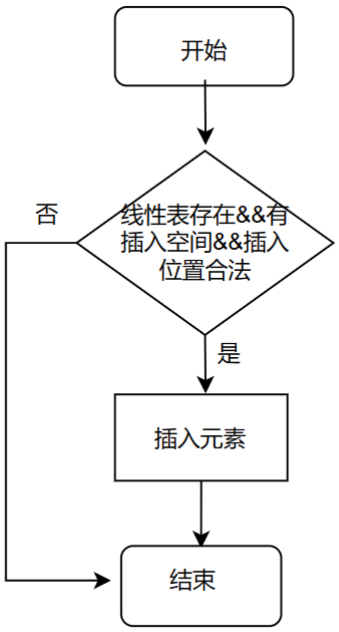


图 1-6　插入元素流程图

(k) ListDelete(LinkList L, int i, ElemType e)

思想：定位到第 i-1 个结点，修改其 next 指针跳过待删结点，并释放内存。

关键点：

- 需要保存被删结点的数据
- 严格检查位置 i 的合法性
- 防止内存泄漏（必须 free）

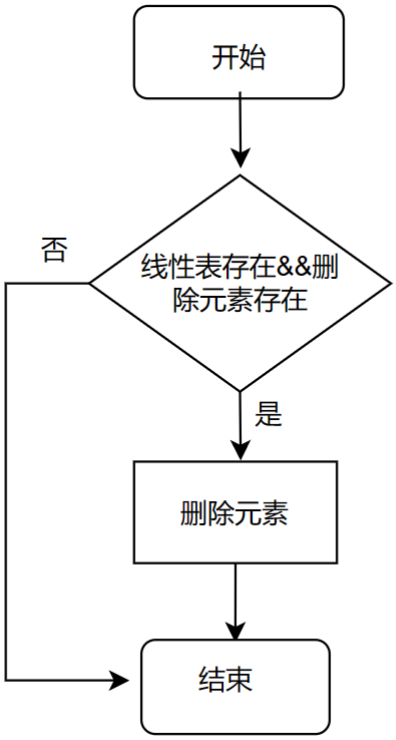


图 1-7　删除元素流程图

(l) ListTraverse(LinkList L)

思想：从头结点后的第一个结点开始，依次访问并输出数据，直到 next 为 NULL。

关键点：

- 空格控制：最后一个元素后不加空格
- 不修改链表结构
- 输出格式统一（当前为%d 需与 ElemType 匹配）

(m) ReverseList(LinkList L)

思想：三指针法（pre, cur, next），逐个反转结点指向，最后更新头结点指针。

关键点：

- 需要保存 next 指针以防断链
- 最后将头结点指向新的首元素
- 时间复杂度 O(n) 最优解

(n) RemoveNthFromEnd

思想：先计算链表长度 length，转化为正向位置 length-i+1，再调用 ListDelete。

关键点：

- 两次遍历（可优化为单次快慢指针）
- 位置转换公式的正确性
- 复用 ListDelete 保证删除逻辑一致

(o) SortList(LinkList L)

思想：冒泡排序，通过双重循环比较相邻结点数据并交换，利用 flag 优化无交换时的提前退出。

关键点：

- 仅交换数据域降低复杂度
- flag 优化最好情况时间复杂度
- 每次外循环减少内循环次数

(p) SaveList

思想：遍历链表，将每个结点的数据按格式写入文件。

关键点：

- 文件打开模式”w” 会覆盖原有内容
- 数据间隔使用空格分隔
- 必须检查 fopen 返回值

(q) LoadList

思想：从文件读取数据，动态创建新结点并链接成单链表。

关键点：

- 仅允许操作未初始化的链表（L==NULL）
- 文件格式需与 SaveList 匹配
- 内存分配失败时的回滚处理

3) 多链表管理

(a) AddList

思想：在 LISTS 链表中新增一个结点，存储新链表的名称和头指针。

关键点：

- 名称查重（strcmp 严格匹配）

- 维护尾指针提高追加效率
- 同时管理 length 计数器

(b) RemoveList

思想：查找目标链表结点，释放其内部线性表内存，再从 LISTS 中摘除该结点。

关键点：

- 需要处理头结点/中间结点不同情况
- 先销毁内部链表再释放节点
- 更新 tail 指针当删除尾节点时

(c) LocateList

思想：遍历 LISTS 的结点链表，通过 strcmp 比较名称，返回逻辑位置。

关键点：

- 位置计数从 1 开始（符合用户习惯）
- 字符串比较区分大小写
- 线性查找时间复杂度 O(n)

(d) displayAllLists

思想：遍历显示所有链表的名称及其序号。

关键点：

- 简单的顺序编号输出
- 不显示内部链表细节
- 空列表时的友好提示

1.4 系统测试

1.4.1 单线性表

构造了一个具有菜单的功能演示系统。先在总菜单中选择操作单线性表还是多线性表管理，此处我们输入 1，选择操作单线性表，如图 1-8所示：

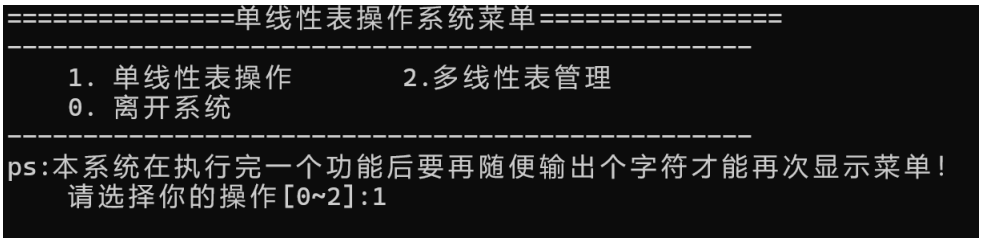


图 1-8　选择操作单表

进入单线性表操作系统后，会显示单线性表操作菜单，如图 1-9所示，我们可以在功能 0 至 17 中选择我们想要的功能。

图 1-9　单表菜单

1) 初始化表

初始情况	测试用例	理论结果	运行结果
线性表不存在	1	成功创建线性表	请选择你的操作[0~17]:1 线性表创建成功!
线性表存在	1	创建线性表失败	请选择你的操作[0~17]:1 线性表创建失败!

表 1-2　初始化表测试

2) 销毁表

3) 清空表、求表长、遍历表

初始情况	测试用例	理论结果	运行结果
线性表不存在	2	销毁线性表失败	请选择你的操作[0~17]:2 线性表销毁失败！
线性表存在	2	销毁线性表成功	请选择你的操作[0~17]:2 线性表销毁成功！

表 1-3　销毁线性表测试

功能	初始情况	测试用例	理论结果	运行结果
清空表	线性表存在	3	清空线性表	请选择你的操作[0~17]:3 线性表清空成功！
求表长	线性表：2 6 4 15 26 5	5	线性表长度为 6	请选择你的操作[0~17]:5 线性表长度为6!
遍历	线性表：2 6 4 15 26 5	12	输出当前线性表元素	请选择你的操作[0~17]:12 线性表内容如下: 2 6 4 15 26 5

表 1-4　部分简单功能测试

4) 判断空表

初始情况	测试用例	理论结果	运行结果
线性表为空	4	线性表为空	请选择你的操作[0~17]:4 线性表为空！
线性表不为空	4	线性表不为空	请选择你的操作[0~17]:4 线性表不为空！

表 1-5　判定空表测试

5) 获得元素

初始情况	测试用例	理论结果	运行结果
线性表：2 6 4 15 26 5	6 2	获得第二个元素 6	请选择你的操作[0~17]:6 请输入你所要获得的元素位置：2 第2个数据元素为6
线性表：2 6 4 15 26 5	6 7	越界，获得失败	请选择你的操作[0~17]:6 请输入你所要获得的元素位置：7 请输入合法的位置 请重新输入你要获得的元素位置：

表 1-6　获得元素测试

6) 查找元素

初始情况	测试用例	理论结果	运行结果
线性表：2 6 4 15 26 5	7 4	查找到元素 4 的位置 3	请选择你的操作[0~17]:7 请输入你要查找的元素！ 4 元素4在第3个位置
线性表：2 6 4 15 26 5	6 7	越界，查找失败	请选择你的操作[0~17]:7 请输入你要查找的元素！ 25 未查找到该元素25

表 1-7　查找元素测试

7) 获得前驱

初始情况	测试用例	理论结果	运行结果
线性表：2 6 4 15 26 5	8 4	获得元素 4 的前驱 6	请选择你的操作[0~17]:8 请输入你要查找前驱的元素： 4 4的前驱为6
线性表：2 6 4 15 26 5	8 2	无前驱，获得失败	请选择你的操作[0~17]:8 请输入你要查找前驱的元素： 2 未找到前驱！

表 1-8　获得前驱测试

8) 获得后继

初始情况	测试用例	理论结果	运行结果
线性表：2 6 4 15 26 5	9 15	获取到元素 15 的后继 26	请选择你的操作[0~17]:9 请输入你要查找后继的元素： 15 15的后继为26
线性表：2 6 4 15 26 5	9 5	无后继，获得失败	请选择你的操作[0~17]:9 请输入你要查找后继的元素： 5 未找到后继！

表 1-9　获得后继测试

9) 插入元素

初始情况	测试用例	理论结果	运行结果
线性表：2 6 4 15 26 5	10 2 5	将元素 2 插入到第 5 个位置	请选择你的操作[0~17]:10 请输入你要插入的元素以及要插入的位置！ 2 5 ListInsert功能实现成功！
线性表：2 6 4 15 2 26 5	10 2 9	越界，插入失败	请选择你的操作[0~17]:10 请输入你要插入的元素以及要插入的位置！ 2 9 ListInsert功能实现失败！

表 1-10　插入元素测试

10) 删除元素

初始情况	测试用例	理论结果	运行结果
线性表：2 6 4 15 2 26 5	11 5	删除第 5 个元素 2	请选择你的操作[0~17]:11 请输入你要删除的元素位置！ 5 ListDelete功能实现成功！ 删除的元素为2!
线性表：2 6 4 15 26 5	11 8	越界，删除失败	请选择你的操作[0~17]:11 请输入你要删除的元素位置！ 8 ListDelete功能实现失败！

表 1-11　删除元素测试

11) 附加功能

功能	初始情况	测试用例	理论结果	运行结果
链表反转	线性表：2 6 4 15 26 5	13	线性表：5 26 15 4 6 2	请选择你的操作[0~17]:13 线性表反转成功！ 线性表内容如下： 5 26 15 4 6 2
删除倒数第 i 个结点	线性表：2 6 4 15 26 5	14 2	删除倒数第 2 个元素 26	请选择你的操作[0~17]:14 请输入你要删除的倒数第i个结点！ 2 RemoveNthFromEnd功能实现成功！ 删除的元素为26!
链表排序	线性表: 2 6 4 15 5	15	线性表元素从小到大排序	请选择你的操作[0~17]:15 SortList功能已实现！ 线性表内容如下 2 4 5 6 15
文件保存	线性表: 2 4 5 6 15	16 list.txt	线性表元素存储在文件 list.txt 中	请选择你的操作[0~17]:16 请输入文件名：list.txt 写入完毕！ 文件 编辑 查看 2 4 5 6 15
读入	线性表存在	17 list.txt	读入失败	请选择你的操作[0~17]:17 请输入文件名：list.txt 不能对已存在的线性表进行读文件操作！
读入	线性表不存在	17 list.txt	将文件中的内容存储在线性表中	请选择你的操作[0~17]:17 请输入文件名：list.txt 读入完毕！

表 1-12　附加功能测试

1.4.2　多线性表

若在主菜单选择功能 2：多线性表管理，则可以进入多线性表管理系统，多线性表管理菜单如图 1-10所示：

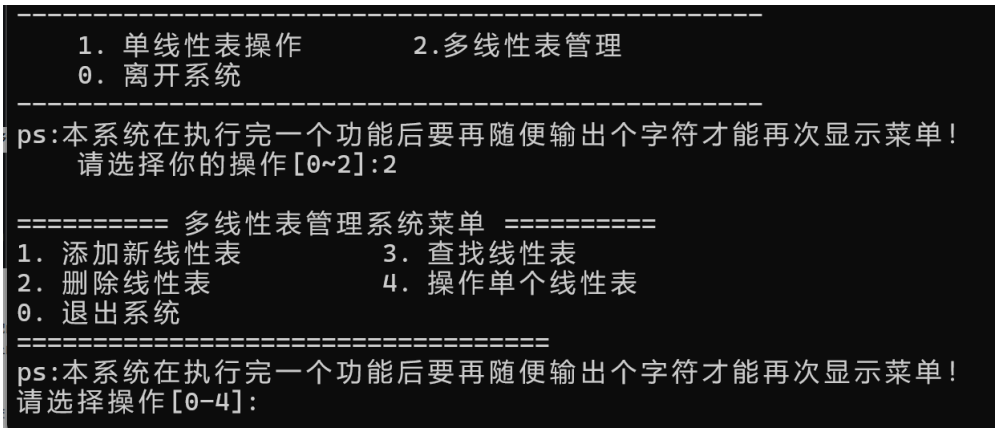


图 1-10 多线性表菜单

1) 添加线性表

初始情况	测试用例	理论结果	运行结果
无线性表	1 list1	创建一个名为 list1 的线性表	请选择操作[0-4]: 1 请输入线性表名称: list1 线性表 list1 添加成功!
线性表名单: 1.list1	1 list1	list1 已存在，添加失败	请选择操作[0-4]: 1 请输入线性表名称: list1 添加失败!

表 1-13 添加线性表测试

2) 删除线性表

初始情况	测试用例	理论结果	运行结果
线性表名单: 1.list1 2.list2	2 list1	删除名为 list1 的线性表	请选择操作[0-4]: 2 请输入要删除的线性表名称: list1 线性表 list1 删除成功!
线性表名单: 1.list2	2 list	list 不存在，删除失败	请选择操作[0-4]: 2 请输入要删除的线性表名称: list 删除失败，未找到该线性表!

表 1-14 删除线性表测试

3) 查找线性表

初始情况	测试用例	理论结果	运行结果
线性表名单：1.list2	3 list2	返回线性表 list2 的位置 1	请选择操作[0-4]: 3 请输入要查找的线性表名称：list2 成功找到线性表list2，线性表在第1个位置！
线性表名单：1.list2	3 list1	list1 不存在，查找失败	请选择操作[0-4]: 3 请输入要查找的线性表名称：list1 未找到线性表 list1

表 1-15　查找线性表测试

4) 操作单个线性表

初始情况	测试用例	理论结果	运行结果
线性表名单：1.list2	4 1	跳转单表操作系统，对 list2 进行一系列操作	请选择操作[0-4]: 4 当前线性表有： 1:list2 请输入想要单独操作的线性表逻辑序号：1

表 1-16　操作单个线性表测试

注：此时在单表菜单处在操作单线性表选择 0：Exit 后会返回操作多线性表菜单。如图 1-11所示

```
------------------------------------------------
    1. 单线性表操作        2.多线性表管理
    0. 离开系统
------------------------------------------------
ps:本系统在执行完一个功能后要再随便输出个字符才能再次显示菜单!
    请选择你的操作[0~2]:2

========== 多线性表管理系统菜单 ==========
1. 添加新线性表          3. 查找线性表
2. 删除线性表            4. 操作单个线性表
0. 退出系统
==================================
ps:本系统在执行完一个功能后要再随便输出个字符才能再次显示菜单!
请选择操作[0-4]:
```

图 1-11　返回多线性表菜单

1.5　实验小结

1) 经验总结

(a) 深入理解链式结构

通过手写链表操作，直观理解了指针操作和内存管理的关系，对比课堂

学习的理论复杂度与实际运行性能差异，在课堂上，我们学习了线性表的定义、基本操作及其逻辑结构。但真正用 C 语言去实现时，才体会到“纸上得来终觉浅”。比如，实现插入和删除操作时，指针的操作、内存的管理（尤其是动态内存分配 malloc 和 free）变得至关重要，稍有不慎就会导致内存泄漏或指针悬垂。这次实验让我对链表节点的动态创建、连接、断开和释放有了更直观和深刻的认识

(b) 用户交互与程序健壮性

为了让程序更实用，我们加入了一个交互式菜单系统。这让我意识到，一个程序不仅要实现核心功能，还要考虑用户体验。如何处理用户输入错误（比如输入非法选项或非数字），如何给出清晰的提示信息，都是需要仔细考虑的方面。这提升了我的软件工程意识。

(c) 模块化设计的价值：

我们将线性表的各种操作（如创建、销毁、查找、插入等）封装成独立的函数，这不仅使代码结构更清晰，也大大提高了代码的可重用性和可维护性。当需要修改某个功能时，只需关注对应的函数，而不必改动整个程序。这种模块化的思维方式在大型项目开发中尤为重要

2) 过失与不足

(a) 边界条件的处理：

在实现某些功能时，比如在空表上操作、在表尾之后插入、查找不存在的元素等边界条件，有时考虑不够全面，导致程序在某些极端情况下可能出错或行为未定义。这提醒我在未来的编程中，必须更加细致地考虑所有可能的输入和状态。

如内存泄漏：未在 DestroyList 中释放所有结点。

(b) 代码效率的优化：

虽然实现了所有基本功能，但在效率方面可能还有优化空间。例如，对于排序操作，使用了简单的冒泡排序或选择排序，对于大数据量来说效率不高，可以采用更高效的排序算法。

3) 改进方向

提高算法效率，增加更复杂的功能，如链表排序优化（归并排序）；完善代码逻辑，使代码更严谨。

2 基于二叉链表的二叉树实现

2.1 问题描述

1) 单二叉树操作部分：

(a) 创建二叉树：通过给定的带空枝的先根遍历序列（包含关键字和字符串）构建一棵二叉树，要求关键字不能重复。

(b) 销毁与清空二叉树：提供销毁（释放所有内存）和清空（保留内存空间但清空内容）功能。

(c) 判空与深度计算：判定二叉树是否为空，以及计算二叉树的深度。

(d) 节点查找与赋值：支持查找指定关键字的节点，赋予节点新的数据。

(e) 兄弟节点与父节点查找：提供获取指定节点的兄弟节点及父节点的功能。

(f) 插入与删除节点：在指定位置插入或删除节点，要求确保树结构的合法性（如无重复节点等）。

(g) 遍历操作：支持先序、中序、后序及层次遍历，便于查看树的结构。

(h) 路径和计算与公共祖先：提供计算最大路径和（根到叶的路径和）以及查找两个节点的最近公共祖先的功能。

(i) 树翻转与文件操作：可以翻转树结构，并将树数据保存到文件中，或者从文件中读取数据创建树。

2) 多线性表管理部分：

(a) 多二叉树管理：支持同时管理多个二叉树，能够进行创建、删除、查找以及查看已存在的二叉树。

(b) 内存管理：在内存不足时进行扩容，以支持更多的二叉树数据。

2.2 系统设计

1) 整体系统结构设计：

(a) **单二叉树操作模块**

i. 提供二叉树的基本操作功能，包括创建、销毁、插入、删除、查找、赋值等。

ii. 通过菜单驱动方式与用户交互，用户可以选择所需操作。

iii. 每个功能对应独立的函数实现，例如二叉树的遍历、路径计算、最大路径和计算等。

(b) **多二叉树管理模块**

i. 使用动态数组结构管理多个二叉树，每个二叉树通过树名与指针进行管理。

ii. 支持二叉树的添加、删除、查找、选择等操作。

iii. 维护二叉树名称与对应二叉树的关系，实现多树操作。

2) 数据结构设计：

(a) **二叉树节点结构**：每个二叉树节点包含关键字（整型）和一个字符串数据域。

```
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 {
    int pos;
    TElemType data;
} DEF;
```

图 2-1　二叉树节点结构

(b) **多二叉树管理结构**：使用动态数组存储多个二叉树，每个数组元素包含树名和指向树的指针。

```c
typedef struct
{
    char name[30];
    BiTNode *T;
} TreeEntry;

typedef struct
{
    TreeEntry *elem; // 改为动态数组指针
    int length;      // 当前实际存储了多少个二叉树
    int listsize;    // 当前已分配的容量
} LISTS;
```

图 2-2　多二叉树管理结构

(c) **状态定义**：定义了多种状态返回值（如 OK、ERROR 等）来处理不同的操作结果和错误。

```c
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define ERROR2 -3
#define INFEASIBLE -1
#define overflow -2
#define init_size 50

typedef int status;
typedef int KeyType;
```

图 2-3　状态定义

3) 关键设计特点：

(a) **动态内存管理**：

i. 所有二叉树的节点均使用动态内存分配。

ii. 完备的销毁机制，确保在删除二叉树时释放所有节点内存，防止内存泄漏。

(b) **错误处理机制**：

i. 统一的状态返回值规范（如 OK、ERROR 等）。

ii. 对非法操作进行严格的检查，如空节点插入、重复关键字等。

(c) **扩展性设计**：

i. 多二叉树管理采用动态扩展的数组结构，能够根据需要扩展管理更多的二叉树。

ii. 数据域（如关键字和字符串）使用 typedef 定义，便于修改元素类型，具备较好的扩展性。

(d) **用户友好性**：

i. 采用分级菜单导航，清晰地引导用户完成操作。

ii. 操作结果即时反馈，帮助用户理解每次操作的结果。

2.3 系统实现

2.3.1 函数总览

函数名	时间复杂度	空间复杂度	功能概述
CreateBiTree	$O(n)$	$O(n)$	根据带空枝的先根遍历序列创建二叉树，检测重复关键字。
DestroyBiTree	$O(n)$	$O(h)$	递归释放所有节点内存，销毁整棵树。
ClearBiTree	$O(n)$	$O(h)$	清空所有节点数据但不释放内存。
BiTreeEmpty	$O(n)$	$O(h)$	判断所有节点是否为“空”状态。
BiTreeDepth	$O(n)$	$O(h)$	递归计算树的最大深度。
LocateNode	$O(n)$	$O(h)$	查找指定关键字的节点并返回指针。
Assign	$O(n)$	$O(h)$	给指定关键字的节点赋新值，检查冲突。
GetSibling	$O(n)$	$O(h)$	获取指定节点的兄弟节点。
InsertNode	$O(n)$	$O(h)$	插入指定节点的左右子节点或作为新根插入。
DeleteNode	$O(n)$	$O(h)$	删除指定节点并重接子树。
PreOrderTraverse	$O(n)$	$O(n)$	使用栈实现非递归先序遍历。
InOrderTraverse	$O(n)$	$O(h)$	递归实现中序遍历。
PostOrderTraverse	$O(n)$	$O(h)$	递归实现后序遍历。
LevelOrderTraverse	$O(n)$	$O(n)$	层次遍历，使用队列。
MaxPathSum	$O(n)$	$O(h)$	计算根到叶路径最大权值和。
LowestCommonAncestor	$O(n)$	$O(h)$	查找两个节点的最近公共祖先。
InvertTree	$O(n)$	$O(h)$	递归交换所有节点左右子树，实现整树翻转。
SaveBiTree	$O(n)$	$O(h)$	按先根遍历顺序将树写入文件，支持空子树标记。
LoadBiTree	$O(n)$	$O(h)$	从文件中重建二叉树结构。

表 2-1 二叉树核心函数功能总览

2.3.2 函数简介

1) 菜单展示

(a) **show1()**

- **思想**: 显示系统主菜单，提供单二叉树操作、多二叉树管理和退出系统的选项，作为系统功能的入口界面。
- **关键点**: 菜单项清晰简洁，数字选择逻辑明确，便于后续函数根据输入跳转。

(b) **show2()**

- **思想**: 显示单棵二叉树的功能菜单，包含创建、销毁、遍历、插入、删除、查找等 19 种操作，便于用户在功能间切换。
- **关键点**: 实时预览二叉树结构（按层次遍历），帮助用户感知操作效果；提供错误提示如“树不存在”。

(c) **show3()**

- **思想**: 显示多棵二叉树管理菜单，支持添加、删除、查找、选择并操作各棵树。
- **关键点**: 强调“单棵树操作”入口，让多树结构与单树逻辑无缝结合。

2) 单二叉树操作

1. **CreateBiTree()**

   - **思想**: 通过递归方式构建二叉树，依据带空枝的先序序列输入；每个结点包含整数关键字和字符串信息。采用全局变量 ‘tot‘ 控制读入位置，遇 ‘0‘ 表示空子树，‘-1‘ 表示结束。
   - **关键点**:
     - 使用 ‘unordered$_{m}ap$‘□□

– 递归的终止条件为输入为 ‘0‘ 或 ‘-1‘，此时不再创建子节点。

2. **DestroyBiTree()**

   - **思想**: 采用后序遍历方式递归释放所有节点内存，即“左右根”顺序，确保子节点在父节点之前被释放。
   - **关键点**:
     - 判断节点非空后递归销毁左右子树，再释放当前节点。
     - 释放后将指针置为空，防止悬挂指针导致未定义行为。

3. **ClearBiTree()**

- **思想**: 清空节点数据但保留树结构，递归将节点关键字和字符串重置为空。
- **关键点**: 仅修改数据内容，不释放节点内存。

4. **BiTreeEmpty()**
   - **思想**: 递归检查树是否为空，包括树不存在或所有节点关键字为 0 的情况。
   - **关键点**: 需递归遍历整棵树确认无有效数据。
5. **BiTreeDepth()**
   - **思想**: 递归计算左右子树深度，取最大值加 1 作为当前子树深度。
   - **关键点**: 递归终止条件为节点为空，深度为 0。
6. **LocateNode()**
   - **思想**: 递归搜索关键字匹配的节点，优先左子树后右子树。
   - **关键点**: 递归搜索顺序影响效率，但确保全覆盖。
7. **Assign()**
   - **思想**: 修改指定节点数据，需检查目标关键字唯一性。
   - **关键点**: 需验证新关键字是否与已有节点冲突（除自身外）。
8. **GetSibling()**
   - **思想**: 递归查找兄弟节点，通过父节点判断左右子树关系。
   - **关键点**: 需处理根节点无兄弟的情况。

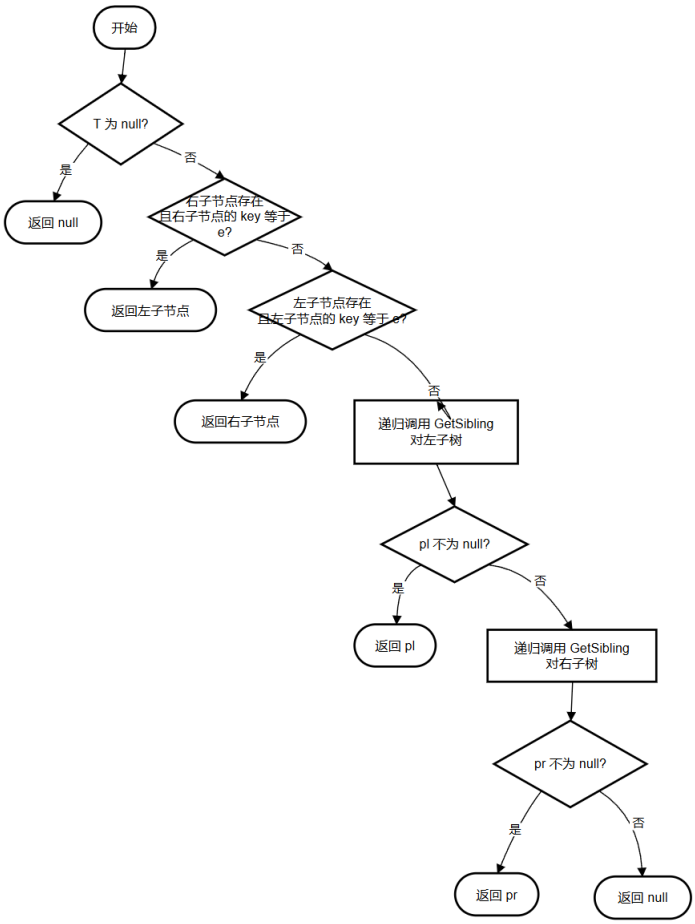


图 2-4　获得兄弟结点流程图

9. **InsertNode()**
   - **思想**: 插入新节点 ‘c‘ 到结点 ‘e‘ 的左/右子树位置（由 ‘LR‘ 参数指定），若 ‘LR = -1‘ 则构造新根节点。原有子树将作为新插入节点的右子树。
   - **关键点**:
     – 插入前需确认插入位置存在（通过 LocateNode 查找）且插入值唯一。
     – 动态申请新节点空间，灵活处理插入左右子树逻辑。
     – 若 ‘LR = -1‘，将整个旧树作为新节点右子树，构建新根节点。
10. **DeleteNode()**
    - **思想**: 删除指定节点 ‘e‘，按不同子树组合分为四种情况处理（无子树、单左/右子树、双子树），保持结构合理性。
    - **关键点**:
      – 找到父节点 ‘p‘ 与目标节点 ‘cur‘ 后，根据左右子树情况分类处理。

- 若为双子树，需将右子树接到左子树最右节点后，避免节点丢失。
- 特殊处理根节点删除的情况（无父节点）。

11. **PreOrderTraverse()**

- **思想**: 使用栈实现非递归先序遍历，模拟系统调用栈，按“根左右”顺序访问各节点。
- **关键点**:
  - 栈先进后出：访问根节点后，先压右子节点再压左子节点，确保左子树先访问。
  - 避免使用系统栈，提高稳定性与可控性。

12. **InOrderTraverse()**

- **思想**: 递归实现“左根右”顺序，直接递归调用左右子树。
- **关键点**: 递归终止条件为节点为空。

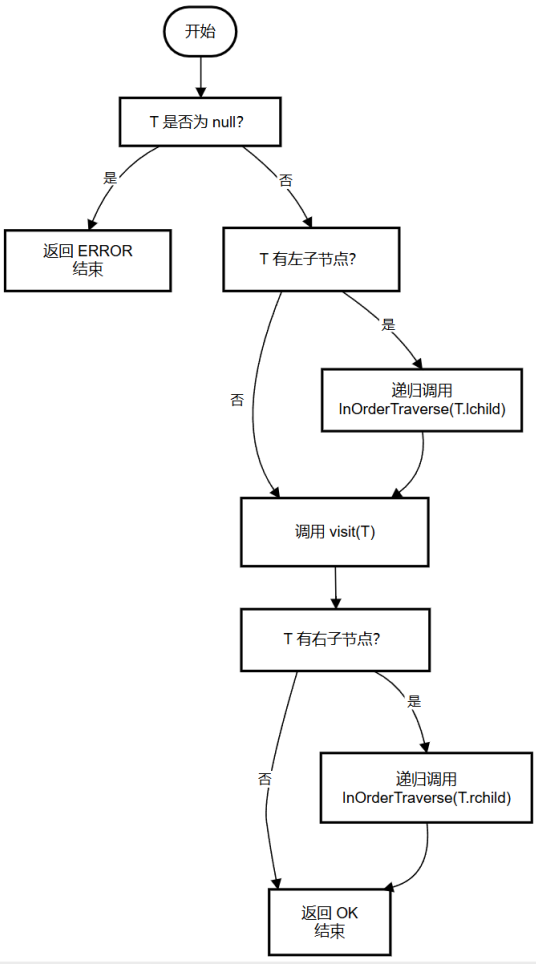


图 2-5　中序遍历流程图

13. **PostOrderTraverse()**
    - **思想**: 递归实现“左右根”顺序，确保子节点访问完毕再处理根。
    - **关键点**: 必须左右子树均递归完成后才能访问根。
14. **LevelOrderTraverse()**
    - **思想**: 使用队列实现层次遍历，按层依次处理节点。
    - **关键点**: 队列先进先出特性保证层级顺序。
15. **SaveBiTree()**
    - **思想**: 使用先序遍历递归将二叉树结构写入文件，节点数据按“关键字字符串”格式写入，空指针写“”作为标记。
    - **关键点**:
      - 文件格式应严格一致以兼容 LoadBiTree 的读取方式；
      - 文件写入前需确保文件成功打开；
      - 遍历使用辅助函数 ‘save()‘，递归保存左右子树。
16. **LoadBiTree()**
    - **思想**: 通过读取文件内容并以先序方式重建整棵树，遇“”表示空子树，字符串输入需分隔处理。
    - **关键点**:
      - 读取格式应与保存一致：每行一个节点或“”；
      - 使用 fgets + sscanf 实现灵活的输入读取；
      - 动态分配节点内存，并递归重建左右子树。
17. **MaxPathSum()**
    - **思想**: 自顶向下递归，分别求左右子树的最大路径和，然后返回当前节点 key 加上较大值，得到从根到叶最大路径和。
    - **关键点**:
      - 空树返回 0，递归终止条件清晰；
      - 路径为“单向路径”：不回溯，不包含多个分支；
      - 每个子树递归只取最大，不计算“全局最大子路径”。
18. **LowestCommonAncestor()**
    - **思想**: 通过递归在左右子树分别查找两个节点，若分别存在于两侧，则当前节点即为最近公共祖先；若某一侧为空，则说明两个节点在同一子树中。

- **关键点**:
  - 如果当前节点即为待查节点，直接返回；
  - 递归结果若同时非空，当前即为祖先；
  - 注意终止条件与空指针处理。

19. **InvertTree()**

- **思想**: 递归交换每个节点的左右子树，自上而下进行，最终达到镜像翻转整个树的目的。
- **关键点**:
  - 当前层交换后再递归处理子树；
  - 不可自下而上，否则已交换的子树会被再次修改；
  - 空树需跳过避免访问非法指针。

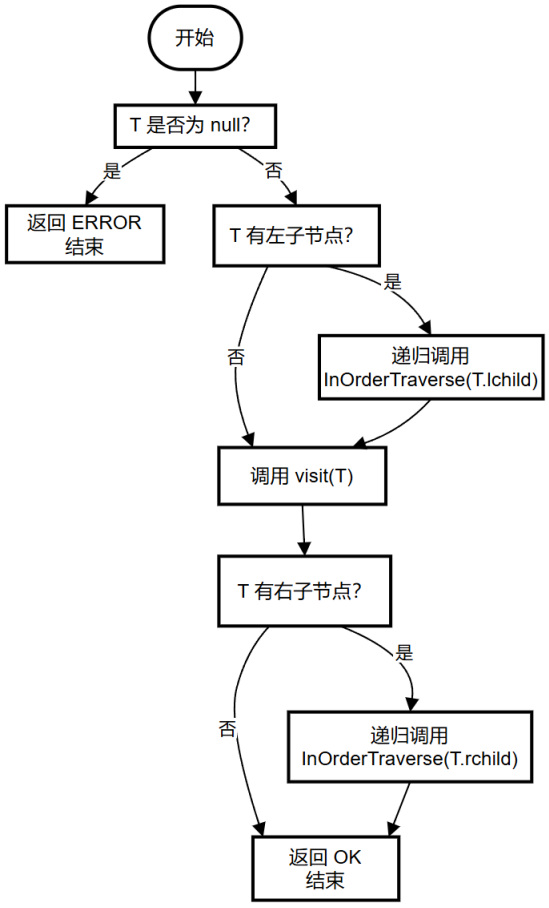


图 2-6　翻转流程图

3) 多二叉树管理

1. **Addtree()**

- **思想**: 动态扩容数组并添加新树，名称唯一性由调用者保证。
- **关键点**: 扩容时需迁移数据并释放旧内存。

2. **Removetree()**
   - **思想**: 查找并删除指定树，移动后续元素填补空缺。
   - **关键点**: 需先释放树内存再调整数组。
3. **Locatetree()**
   - **思想**: 线性搜索匹配名称的树，返回逻辑序号（从 1 开始）。
   - **关键点**: 返回 0 表示未找到，符合常规逻辑。
4. **TraversLists()**
   - **思想**: 遍历列表打印所有树名称，提供用户友好显示。
   - **关键点**: 输出格式简洁，序号从 1 开始。

2.4 系统测试

2.4.1 单二叉树操作

构造了一个具有菜单的功能演示系统。先在总菜单中选择操作单二叉树还是多二叉树管理，此处我们输入 1，选择操作单二叉树，如图 2-7所示：

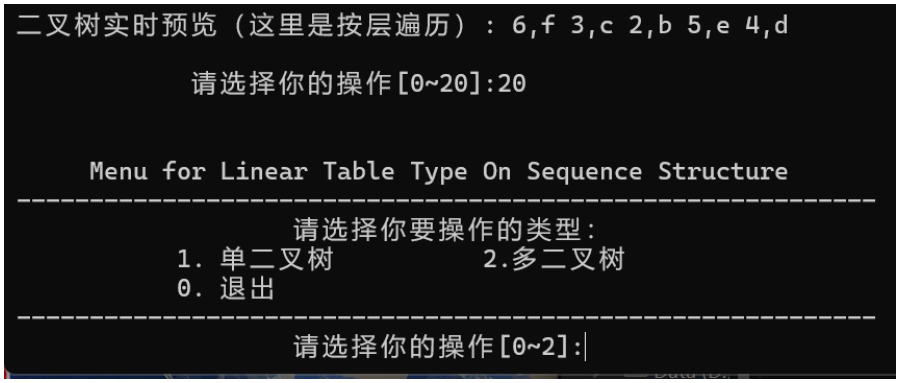


图 2-7 总菜单

进入单二叉树操作系统后，会显示单二叉树操作菜单，如图 2-8所示，我们可以在功能 0 至 20 中选择我们想要的功能。

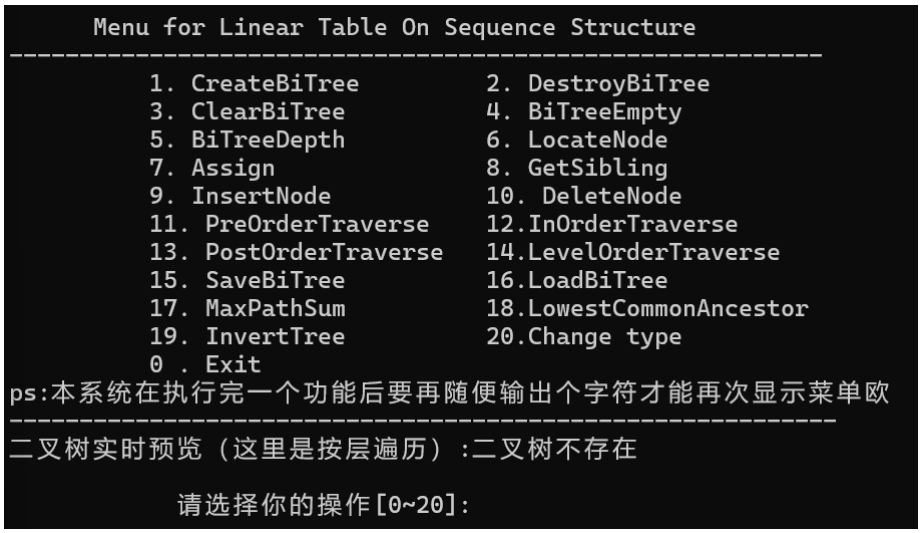


图 2-8　单二叉树菜单

1) 创建二叉树

初始情况	测试用例	理论结果	运行结果
二叉树不存在	1 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null	成功创建二叉树	二叉树实时预览（这里是按层遍历）:二叉树不存在 请选择你的操作[0~20]:1 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null 创建成功！
二叉树存在	1	创建二叉树失败	二叉树实时预览（这里是按层遍历）：1,a 2,b 3,c 4,d 5,e 请选择你的操作[0~20]:1 二叉树已存在，请勿重复创建

表 2-2　创建二叉树测试

2) 销毁二叉树

初始情况	测试用例	理论结果	运行结果
二叉树不存在	2	销毁二叉树失败	二叉树实时预览（这里是按层遍历）:二叉树不存在 请选择你的操作[0~20]:2 不能对不存在的二叉树进行操作！
二叉树存在	2	销毁二叉树成功	二叉树实时预览（这里是按层遍历）：0, 0, 0, 0, 0, 请选择你的操作[0~20]:2 销毁成功！

表 2-3　销毁二叉树测试

3) 部分功能测试

功能	初始情况	测试用例	理论结果	运行结果
清空二叉树	二叉树存在	3	清空二叉树	二叉树实时预览（这里是按层遍历）：6,f 2,b 3,c 4,d 5,e 请选择你的操作[0~20]:3 清空成功！
求深度	二叉树：1 a 2 b 3 c 4 d 5 e	5	二叉树深度为 3	二叉树实时预览（这里是按层遍历）：1,a 2,b 3,c 4,d 5,e 请选择你的操作[0~20]:5 二叉树的深度是3
赋值	二叉树：1 a 2 b 3 c 4 d 5 e	7 1 6 f	将关键字为 1 的结点赋值为（6，f）	二叉树实时预览（这里是按层遍历）：1,a 2,b 3,c 4,d 5,e 请选择你的操作[0~20]:7 输入想要赋值的节点关键字以及赋予的值：1 6 f 赋值成功！

表 2-4　部分简单功能测试

4) 二叉树判空

初始情况	测试用例	理论结果	运行结果
二叉树：1 a 2 b 3 c 4 d 5 e	4	二叉树不为空	二叉树实时预览（这里是按层遍历）：1,a 2,b 3,c 4,d 5,e 请选择你的操作[0~20]:4 二叉树不是空的！
二叉树为空	4	二叉树为空	二叉树实时预览（这里是按层遍历）：0, 0, 0, 0, 0, 请选择你的操作[0~20]:4 二叉树是空哒！

表 2-5　判空二叉树测试

5) 二叉树查找结点

初始情况	测试用例	理论结果	运行结果
二叉树：1 a 2 b 3 c 4 d 5 e	6 2	找到关键字为 2 的结点	二叉树实时预览（这里是按层遍历）：1,a 2,b 3,c 4,d 5,e 请选择你的操作[0~20]:6 想要获得哪个关键字对应的结点：2 这个节点是：2 b
二叉树：1 a 2 b 3 c 4 d 5 e	6 6	找不到关键字为 6 的结点	二叉树实时预览（这里是按层遍历）：1,a 2,b 3,c 4,d 5,e 请选择你的操作[0~20]:6 想要获得哪个关键字对应的结点：6 找不到该节点

表 2-6　二叉树查找结点测试

6) 二叉树获得兄弟结点

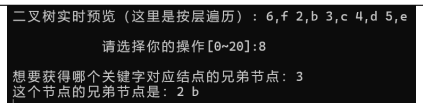
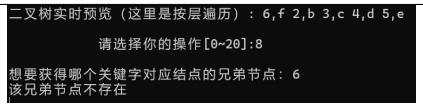
初始情况	测试用例	理论结果	运行结果
二叉树：6 f 2 b 3 c 4 d 5 e	8 3	找到关键字为 3 的兄弟结点	
二叉树：6 f 2 b 3 c 4 d 5 e	8 6	找不到关键字为 6 的兄弟结点	

表 2-7　二叉树获得兄弟结点测试

7) 二叉树插入结点

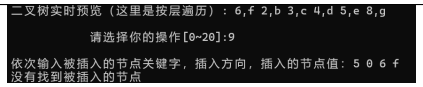
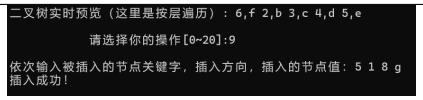
初始情况	测试用例	理论结果	运行结果
二叉树：6 f 2 b 3 c 4 d 5 e	9 5 0 6 f	插入失败	
二叉树：6 f 2 b 3 c 4 d 5 e	9 5 1 8 g	插入成功	

表 2-8　二叉树获得兄弟结点测试

8) 二叉树遍历

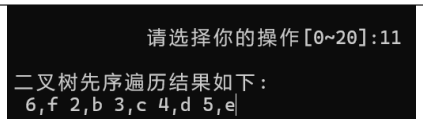
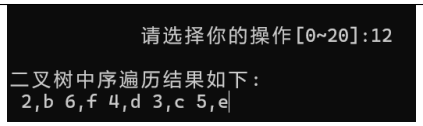
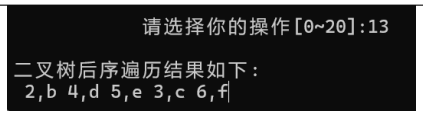
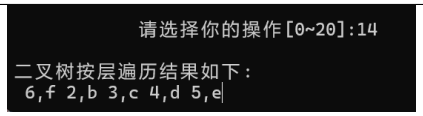
功能	初始情况	测试用例	运行结果
先序	二叉树：6 f 2 b 3 c 4 d 5 e	9 5 0 6 f	
中序	二叉树：6 f 2 b 3 c 4 d 5 e	9 5 1 8 g	
后序	二叉树：6 f 2 b 3 c 4 d 5 e	9 5 1 8 g	
层序	二叉树：6 f 2 b 3 c 4 d 5 e	9 5 1 8 g	

表 2-9　二叉树遍历测试

9) 附加功能测试

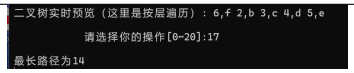
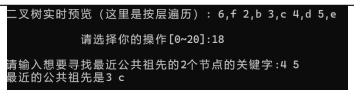
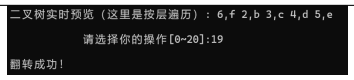
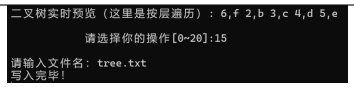
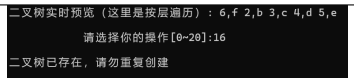
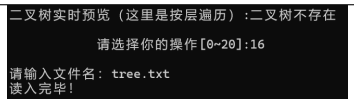
功能	初始情况	测试用例	理论结果	运行结果
最大路径和	二叉树：6 f 2 b 3 c 4 d 5 e	17	返回根节点到叶子节点的最大路径和	
最近公共祖先	二叉树：6 f 2 b 3 c 4 d 5 e	18 4 5	返回 4 和 5 的最近公共祖先 3（3，c)	
翻转二叉树	二叉树：6 f 2 b 3 c 4 d 5 e	19	翻转成功	
文件保存	二叉树：6 f 2 b 3 c 4 d 5 e	15 tree.txt	二叉树结点保存在文件 tree.txt 中	
文件读入	二叉树：6 f 2 b 3 c 4 d 5 e	16	二叉树存在，读入失败	
文件读入	二叉树不存在	16	文件读入成功，文件中的结点信息存储在二叉树中	

表 2-10　附加功能测试

10) 转换功能

选择功能 20，可跳转到多线性表管理系统。

2.4.2　多二叉树管理

若在主菜单选择功能 2：多线性表管理，或者在但线性表操作菜单选择 20：change Type, 则可以进入多线性表管理系统，多线性表管理菜单如图 2-9所示：

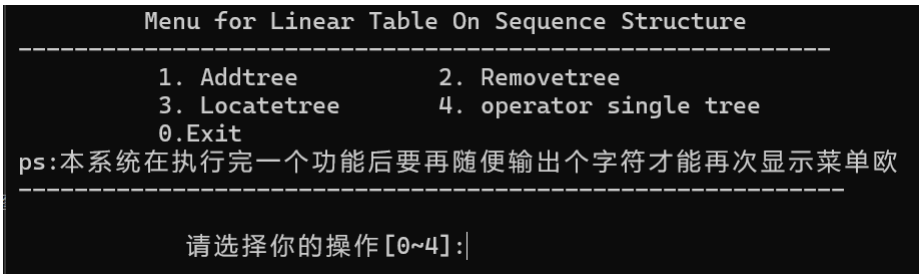


图 2-9　多二叉树管理菜单

1) 添加二叉树

初始情况	测试用例	理论结果	运行结果
无二叉树	1 tree1	创建一个名为 tree1 的二叉树	请选择你的操作[0~4]:1 请输入新建二叉树树名：tree1 创建成功！
二叉树名单：1.tree1	1 tree1	tree1 已存在，添加失败	请选择你的操作[0~4]:1 请输入新建二叉树树名：tree1 名称已存在，创建失败！

表 2-11　添加二叉树测试

2) 删除二叉树

初始情况	测试用例	理论结果	运行结果
二叉树名单：1.tree1 2.tree2	2 tree1	删除名为 tree1 的二叉树	请选择你的操作[0~4]:2 请输入想要移除的二叉树树名：tree1 移除成功！
二叉树名单：1.list2	2 tree1	tree1 不存在，删除失败	请选择你的操作[0~4]:2 请输入想要移除的二叉树树名：tree1 找不到该名称的二叉树，移除失败！

表 2-12　删除二叉树测试

3) 查找二叉树

初始情况	测试用例	理论结果	运行结果
二叉树名单：1.tree2	3 tree2	返回二叉树 tree2 的位置 1	请选择你的操作[0~4]:3 请输入想要查询的二叉树树名：tree2 tree2的逻辑序号是1
二叉树名单：1.tree2	3 tree1	tree1 不存在，查找失败	请选择你的操作[0~4]:3 请输入想要查询的二叉树树名：tree1 二叉树树名不存在！

表 2-13　查找二叉树测试

4) 操作单个二叉树

初始情况	测试用例	理论结果	运行结果
二叉树名单：1.tree2	4 1	跳转单二叉树操作系统，对 tree2 进行一系列操作	请选择你的操作[0~4]:4 1: tree2 请输入想要单独操作的二叉树逻辑序号：1

表 2-14　操作单个二叉树测试

2.5　实验小结

1) 经验总结

1. **递归与分治思想的应用**
   - 在二叉树的创建、遍历、删除等操作中，递归方法简洁高效，如 CreateBiTree 和 DestroyBiTree。
   - 分治策略在 LowestCommonAncestor 和 MaxPathSum 中体现明显，通过分解问题降低复杂度。
2. **动态内存管理**
   - 使用 malloc 和 free 手动管理内存，确保无泄漏（如 DestroyBiTree 递归释放节点）。
   - 动态数组（LISTS）的扩容机制（realloc）提升了多二叉树管理的灵活性。
3. **非递归遍历的实现**
   - 使用栈实现非递归先序遍历（PreOrderTraverse），队列实现层次遍历（LevelOrderTraverse），优化了空间效率。

2) 过失与不足

1. **代码冗余与重复检查**
   - ClearBiTree 中存在冗余的 if(T==NULL) 判断，需优化逻辑。
   - LocateNode 和 FindFather 功能部分重叠，可合并或抽象为通用查找函数。
2. **错误处理不够完善**
   - 文件操作（如 fopen 失败）仅返回 ERROR，未提供具体错误原因（如文件不存在或权限不足）。

- `InsertNode` 和 `DeleteNode` 中对边界条件（如空树、根节点操作）的处理不够细致。

3. **内存与性能问题**
   - 多二叉树管理的动态数组扩容时，需整体迁移数据并释放旧内存，可能引发性能瓶颈。
   - `MaxPathSum` 和 `LowestCommonAncestor` 的递归实现可能存在重复计算，未优化为动态规划。
4. **扩展性不足**
   - 二叉树节点仅支持 `int` 类型关键字和固定长度字符串，无法灵活适应其他数据类型。
   - 多二叉树管理的 `Lists` 未实现按名称排序或哈希索引，查找效率较低（`O(n)`）。

3) 改进方向

1. **代码优化与重构**
   - 提取公共逻辑（如节点查找、子树链接）为独立函数，减少重复代码。
   - 使用更高效的数据结构（如红黑树）管理多二叉树名称，提升查找速度。
2. **性能优化**
   - 将 `MaxPathSum` 改为记忆化搜索或动态规划，避免重复计算子树路径和。
   - 动态数组扩容策略改为倍增(如 `listsize *= 2`),减少频繁 `realloc` 的开销。

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

1) **收获**

- 通过实现顺序表的基本操作（如插入、删除、查找等），深入理解了顺序存储结构的连续内存特性及其优缺点，例如随机访问的高效性与插入/删除操作的时间复杂度问题。
- 掌握了动态数组的管理方法（如扩容机制），增强了内存管理和边界条件处理的实践能力。
- 附加功能（如最大子数组和、文件保存）提升了问题抽象能力和实际应用技能。

2) **建议**

- 实验可增加更多实际场景的案例（如学生成绩管理系统），帮助理解顺序表的应用价值。
- 对文件保存功能的设计可以引入更复杂的格式（如 JSON），以增强数据交互的通用性。

3.2 基于链式存储结构的线性表实现

1) **收获**

- 通过实现单链表的基本操作，理解了链式存储的非连续特性，以及指针操作在动态数据结构中的核心作用。
- 学会了处理链表的边界条件（如头结点、尾结点），并实践了递归与迭代两种编程思维（如链表翻转）。
- 附加功能（如删除倒数第 n 个结点）锻炼了双指针等算法技巧。

2) **建议**

- 可引入双向链表或循环链表的实现任务，扩展对链式结构的全面理解。
- 文件保存功能可要求支持多链表格式，以体现链表的灵活性优势。

3.3 基于二叉链表的二叉树实现

1) **收获**

- 实现了二叉树的递归与非递归遍历（如栈模拟中序遍历），加深了对树形结构和递归思想的理解。
- 通过操作结点（插入、删除、查找祖先等），掌握了二叉树的结构特性和指针操作技巧。
- 附加功能（如最大路径和）提升了动态规划在树结构中的应用能力。

2) **建议**

- 可增加平衡二叉树（AVL 树）的简单实现，帮助学生理解树的优化结构。
- 文件保存功能可支持图形化展示（如生成 DOT 语言文件），便于直观验证二叉树结构。

3.4 基于邻接表的图实现

1) **收获**

- 通过邻接表的实现，理解了图的链式存储方式及其与矩阵存储的差异，掌握了边和顶点的动态管理方法。
- 实现了 BFS/DFS 遍历，学会了用队列和栈处理图的层级与回溯问题。
- 附加功能（如最短路径、连通分量）引入了经典算法（如 Dijkstra 或并查集）的初步实践。

2) **建议**

- 可对比邻接矩阵的实现，分析两种存储结构的适用场景。
- 增加可视化工具（如 Graphviz）的支持，帮助学生直观理解图的构造与遍历过程。

4 附录 A 基于顺序存储结构线性表实现的源程序

```
//def.h

#include <bits/stdc++.h>
using namespace std;

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

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

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

```
    } elem[10];
    int length;
    int listsize;
} LISTS;

int main1(SqList &L);
int main2(LISTS &Lists);
status InitList(SqList& L);
status DestroyList(SqList& L);
status ClearList(SqList&L);
status ListEmpty(SqList L);
status ListLength(SqList L);
status GetElem(SqList L,int i,ElemType& e);
status LocateElem(SqList L,ElemType e);
status PriorElem(SqList L,ElemType e,ElemType& pre);
status NextElem(SqList L,ElemType e,ElemType& next);
status ListInsert(SqList&L,int i,ElemType e);
status ListDelete(SqList&L,int i,ElemType& e);
status ListTraverse(SqList L);
status MaxSubArray(SqList L);
status SubArrayNum(SqList L,int k);
status SortList(SqList &L);
status SaveList(SqList L,char FileName[]);
status LoadList(SqList &L, char FileName[]);
status AddList(LISTS &Lists, char ListName[]);
status RemoveList(LISTS &Lists, char ListName[]);
int LocateList(LISTS Lists, char ListName[]);
void show3(void);
void displayAllLists(LISTS Lists);
void show1(void);
void show2(void);
```

```
//function.cpp

#include"def.h"
#include"string.h"
status InitList(SqList& L)
{
    if(L.elem)     //判断线性表是否存在
    return INFEASIBLE;
    else{
        L.elem=(ElemType *)malloc(LIST_INIT_SIZE*sizeof(ElemType));
        //为创建的线性表分配存储空间
        L.listsize=LIST_INIT_SIZE;
        L.length=0;
        //更新listsize,length
        return OK;
    }
}

status DestroyList(SqList& L)
{
    if(L.elem){     //若线性表存在则销毁线性表
        free(L.elem);   //释放线性表空间
        L.elem=NULL;    //使线性表未初始化
        L.listsize=0;
        L.length=0;
        return OK;
    }
    return INFEASIBLE;
}
```

```
status ClearList(SqList& L)
{
    if(L.elem==NULL)
    return INFEASIBLE;  //若线性表不存在，则返回INFEASIBLE
    else{
    L.length=0;
    L.listsize=LIST_INIT_SIZE;      //清空线性表
    return OK;
    }
}

status ListEmpty(SqList L)
{
    if(L.elem==NULL)      //判断表是否存在
    return INFEASIBLE;
    else if(L.length==0)  //判断表是否为空
    return TRUE;
    return FALSE;
}

status ListLength(SqList L)
{
    if(L.elem==NULL)     //若表不存在则返回INFEASIBLE
    return INFEASIBLE;
    return L.length;     //返回表的长度
}

status GetElem(SqList L,int i,ElemType &e)
{
    if(L.elem==NULL)     //判断表是否存在
    return INFEASIBLE;
```

```
    if(i<1||i>L.length)  //判断i是否合法
    return ERROR;
    e=L.elem[i-1];       //获得第i个元素
    return OK;
}

int LocateElem(SqList L,ElemType e)
{
    if(L.elem==NULL)     //判断线性表是否存在
    return INFEASIBLE;
    int i;
    for(i=0;i<L.length;i++){   //遍历线性表查找e
        if(L.elem[i]==e)
        return i+1;
    }
    return ERROR;    //找到指定元素e
}

status PriorElem(SqList L,ElemType e,ElemType &pre)
{
    if(L.elem==NULL)     //判断线性表是否存在
    return INFEASIBLE;
    int i;
    for(i=0;i<L.length;i++){   //遍历查找指定元素e
        if(L.elem[i]==e)
        break;
    }
    if(i<L.length&&i>0){    //若找到e，则将e的前驱赋值给pre
    pre=L.elem[i-1];
    return OK;
    }
```

```
    return ERROR;
}

status NextElem(SqList L,ElemType e,ElemType &next)
{
    if(L.elem==NULL)    //判断线性表是否存在
    return INFEASIBLE;
    int i;
    for(i=0;i<L.length;i++){  //遍历查找指定元素e
        if(e==L.elem[i])
        break;
    }
    if(i<L.length-1){     ////若找到e，则将e的后继赋值给next
        next=L.elem[i+1];
        return OK;
    }
    return ERROR;
}

status ListInsert(SqList &L,int i,ElemType e)
{
    if(L.elem==NULL)    //判断线性表是否存在
    return INFEASIBLE;
    if(i>L.length+1||i<1)    //判断i是否合法
    return ERROR;
    if(L.length>=L.listsize){      //若线性表已满，则为线性表扩容
        ElemType *newelem;
        newelem=(ElemType *)malloc(sizeof(ElemType)*(L.listsize+LISTINCREMENT));
        //分配新的存储空间
        if (!newelem) {
            return OVERFLOW;  // 内存不足
```

```
        }
        for(int j=0;j<L.length;j++){    //复制原来的线性表
            newelem[j]=L.elem[j];
        }
        free(L.elem);
        L.elem=newelem;    //替换线性表
        L.listsize+=LISTINCREMENT;    //更新线性表可用长度
    }
    for(int j=L.length;j>=i;j--){   //将线性表中指定位置及之后的元素向后挪一位
        L.elem[j]=L.elem[j-1];
    }
    L.elem[i-1]=e;    //插入指定元素到指定位置
    L.length+=1;      //更新线性表长度
    return OK;
}

status ListDelete(SqList &L,int i,ElemType &e)
{
    if (L.elem==NULL)      //判断线性表是否存在
    return INFEASIBLE;
    if(i<1||i>L.length)    //判断i是否合法
    return ERROR;
    e=L.elem[i-1];         //获得删除的第i个元素
    for(int j=i-1;j<L.length-1;j++){    //将第i个位置后面的元素向前移一位
        L.elem[j]=L.elem[j+1];
    }
    L.length-=1;
    return OK;
}

status ListTraverse(SqList L)
```

```
{
    if(L.elem==NULL)      //判断线性表是否存在
    return INFEASIBLE;
    for(int i=0;i<L.length;i++){      //遍历线性表
        printf("%d",L.elem[i]);
        if(i<L.length-1)    //输出空格
        printf(" ");
    }
    return OK;
}

status MaxSubArray(SqList L)
{
    if(L.elem==NULL)    //判断线性表是否为空
    return INFEASIBLE;
    ElemType psum = L.elem[0], msum = L.elem[0];
    for (int i = 1; i < L.length; ++i)    //寻找最大子数组和
    {
        psum = max(L.elem[i], psum + L.elem[i]);
        msum = max(msum, psum);
    }
    return msum;
}

status SubArrayNum(SqList L,int k){
    if(L.elem==NULL||L.length==0)   //判断线性表是否存在以及是否为空
    return INFEASIBLE;
    int count=0;
    for(int start=0;start<L.length;start++){  //寻找和为k的子数组
        int sum=0;
        for(int end=start;end<L.length;end++){
```

```
            sum+=L.elem[end];
            if(sum==k)
            count++;
        }
    }
    return count;
}

status SortList(SqList &L){
    if(L.elem==NULL)    //判断线性表是否为空
    return INFEASIBLE;
    for(int j=0;j<L.length-1;j++){     //冒泡排序
        for(int k=j+1;k<L.length;k++){
            if(L.elem[j]>L.elem[k]){
                ElemType mid=L.elem[j];
                L.elem[j]=L.elem[k];
                L.elem[k]=mid;
            }
        }
    }
    return OK;
}

status AddList(LISTS &Lists, char ListName[])
// 只需要在Lists中增加一个名称为ListName的空线性表，线性表数据又后台测试程序插入。
{
    Lists.elem[Lists.length].L.elem = NULL; // 头歌里面一开始不是NULL，要人为设为空
    InitList(Lists.elem[Lists.length].L);
    strcpy(Lists.elem[Lists.length].name, ListName);
    Lists.length++;
    return 1;
```

```
}

status RemoveList(LISTS &Lists, char ListName[])
// Lists中删除一个名称为ListName的线性表
{
    for (int i = 0; i < Lists.length; ++i)
        if (strcmp(Lists.elem[i].name, ListName) == 0)    //遍历寻找线性表
        {
            DestroyList(Lists.elem[i].L);    //销毁线性表
            for (int k = i; k < Lists.length - 1; ++k)
            //将销毁的线性表后面的线性表前移一位
            {
                SqList &pre = Lists.elem[k].L, &now = Lists.elem[k + 1].L;
                pre.elem = NULL; // 要认为定义为NULL才能init
                InitList(pre);

                strcpy(Lists.elem[k].name, Lists.elem[k + 1].name);
                for (int x = 0; x < now.length; ++x)
                    ListInsert(pre, x + 1, now.elem[x]);
                pre.length = now.length, pre.listsize = now.listsize;
            }
            Lists.length--;     //线性表数量减一
            return OK;
        }
    return ERROR;
}

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

```
        if (strcmp(Lists.elem[i].name, ListName) == 0)     //遍历查找线性表
            return i + 1;
    return 0;
}

void show1(void){   //总体菜单
    printf("===============线性表操作系统菜单===============\n");
    printf("------------------------------------------------\n");
    printf("    1. 单线性表操作        2.多线性表管理\n");
    printf("    0. 离开系统\n");
    printf("------------------------------------------------\n");
    printf("ps:本系统在执行完一个功能后要再随便输出个字符才能再次显示菜单! \n");
    printf("    请选择你的操作[0~2]:");
}

void show2(void){    //单线性表操作菜单
    printf("\n\n");
        printf("===============单线性表操作系统菜单===============\n");
        printf("------------------------------------------------\n");
        printf("       1. InitList       10. ListInsert\n");
        printf("       2. DestroyList    11. ListDelete\n");
        printf("       3. ClearList      12. ListTraverse\n");
        printf("       4. ListEmpty      13. MaxSubArray\n");
        printf("       5. ListLength     14. SubArrayNum\n");
        printf("       6. GetElem        15. SortList\n");
        printf("       7. LocateElem     16. SaveList\n");
        printf("       8. PriorElem      17. LoadList\n");
        printf("       9. NextElem        0. Exit\n");
        printf("------------------------------------------------\n");
        printf("ps:本系统在执行完一个功能后要再随便输出个字符才能再次显示菜单! \n"
        printf("    请选择你的操作[0~17]:");
```

```
}

void show3() {    //多线性表管理菜单
    printf("\n========== 多线性表管理系统菜单 ==========\n");
    printf("1. 添加新线性表          3. 查找线性表\n");
    printf("2. 删除线性表            4. 操作单个线性表\n");
    printf("0. 退出系统\n");
    printf("====================================\n");
    printf("ps:本系统在执行完一个功能后要再随便输出个字符才能再次显示菜单! \n");
    printf("请选择操作[0-4]: ");
}

void displayAllLists(LISTS Lists) {
    printf("\n当前所有线性表:\n");
    for (int i = 0; i < Lists.length; i++) {
        printf("%d. %s: ", i+1, Lists.elem[i].name);
        ListTraverse(Lists.elem[i].L);
        printf("\n");
    }
}

status LoadList(SqList &L, char FileName[])
{
    if (L.elem)    //判断线性表是否存在
        return INFEASIBLE;
    FILE *fq = fopen(FileName, "r");    //打开文件
    if (!fq)    //判断文件是否打开成功
        return INFEASIBLE;
    L.elem = (ElemType*)malloc(LIST_INIT_SIZE*sizeof(ElemType));
    L.length = 0, L.listsize = LIST_INIT_SIZE;
```

```
    while (fscanf(fq, "%d", &L.elem[L.length]) == 1)    // 逐个读取整数
    {
        L.length++;
        if (L.length >= L.listsize)
        { // 如果数组满了，扩展数组
            L.elem = (ElemType *)realloc(L.elem, (L.length + LISTINCREMENT) * size
            L.listsize += LISTINCREMENT;
        }
    }
    fclose(fq);
    return OK;
}

status SaveList(SqList L, char FileName[])
{
    if (!L.elem)    //判断线性表是否存在
        return INFEASIBLE;
    FILE *fq = fopen(FileName, "w");    //打开文件并判断是否打开成功
    if (!fq)
        return INFEASIBLE;
    for (int i = 0; i < L.length; ++i)    //逐个输入字符
        fprintf(fq, "%d ", L.elem[i]);
    fclose(fq);
    return OK;
}

//run.cpp

#include"def.h"

SqList L={NULL,0,0};;
```

```
LISTS Lists;

int main(void){
    int mode=0;
    show1();
    while(1){
        scanf("%d",&mode);
        switch (mode)
        {
        case 1:
            main1(L);    //单线性表操作
            break;
        case 2:
            main2(Lists);    //多线性表管理
            break;
        case 0:     // 退出系统
            printf("\n感谢使用系统，再见!\n");
            return 0;
        default:
            printf("\n无效的选择，请重新输入!\n");
        }
        show1();
    }
}

int main1(SqList &L){       //单个线性表操作系统
    ElemType e=0;
    int op=1,i=0;
    status flag=0;
    while(op){
        show2();
```

```
scanf("%d",&op);
switch(op){
    case 1:  //创建线性表
        if(InitList(L)==OK) printf("线性表创建成功! \n");
        else printf("线性表创建失败! \n");
        getchar();getchar();    //清楚输入列多余的换行符及其他字符
        break;
    case 2: //销毁线性表
        if(DestroyList(L)==OK) printf("线性表销毁成功! \n");
        else printf("线性表销毁失败! ");
        getchar();getchar();
        break;
    case 3: //清空线性表
        flag==ClearList(L);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
        else if(flag==OK) printf("线性表清空成功! \n");
        getchar();getchar();
        break;
    case 4: //判断线性表是否为空
        flag=ListEmpty(L);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
        else if(flag==TRUE) printf("线性表为空! \n");
        else printf("线性表不为空! \n") ;
        getchar();getchar();
        break;
    case 5: //获得线性表的长度
        flag=ListLength(L);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
        else printf("线性表长度为%d!\n",flag);
        getchar();getchar();
        break;
```

```
case 6: //获得第i个位置的元素
    printf("请输入你所要获得的元素位置! \n");
    scanf("%d",&i);
    flag=GetElem(L,i,e);
    if(flag==INFEASIBLE) printf("线性表不存在! \n");
    else {
        while(flag==ERROR){
            printf("请输入合法的位置 (从0到%d) \n",L.length);
            printf("请重新输入你要获得的元素位置: ");
            scanf("%d",&i);
        }
        printf("第%d个数据元素为%d\n",i,e);
    }
    getchar();getchar();
    break;
case 7: //获得元素e的位置
    printf("请输入你要查找的元素:\n");
    scanf("%d",&e);
    flag=LocateElem(L,e);
    if(flag==INFEASIBLE) printf("线性表不存在! \n");
    else if(flag!=ERROR)  printf("元素%d在第%d个位置\n",e,flag);
    else printf("未查找到元素%d\n",e);
    getchar();getchar();
    break;
case 8:{  //获得e的前驱
    ElemType pre;
    printf("请输入你要查找前驱的元素: ");
    scanf("%d",&e);
    pre=0;
    flag=PriorElem(L,e,pre);
    if(flag==INFEASIBLE) printf("线性表不存在! \n");
```

```
        else if(flag) printf("%d的前驱为%d",e,pre);
        else printf("未找到前驱! \n");
        getchar();getchar();
        break;
    }
    case 9:{ //获得e的后继
        ElemType next;
        next=0;
        printf("请输入你要查找后继的元素: ");
        scanf("%d",&e);
        flag=NextElem(L,e,next);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
        else if(flag) printf("%d的后继为%d",e,next);
        else printf("未找到后继! \n");
        getchar();getchar();
        break;
    }
    case 10: //插入元素
        printf("请输入你要插入的元素以及要插入的位置! ");
        scanf(" %d %d",&e,&i);
        flag=ListInsert(L,i,e);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
        else if(flag==OK) printf("ListInsert功能实现成功! ");
        else printf("ListInsert功能实现失败! ");
        getchar();getchar();
        break;
    case 11: //删除第i个位置的元素
        printf("请输入你要删除的元素位置! ");
        scanf(" %d",&i);
        flag=ListDelete(L,i,e);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
```

```
        else if(flag==OK)
        printf("ListDelete功能实现成功! \n删除的元素为%d!\n",e);
        else printf("ListDelete功能实现失败! ");
        getchar();getchar();
        break;
    case 12:  //遍历线性表
        if(!ListTraverse(L)) printf("线性表是空表! \n");
        getchar();getchar();
        break;
    case 13:  //获得连续最大数组和
        flag=MaxSubArray(L);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
        else printf("最大连续数组和为%d!\n",flag);
        getchar();getchar();
        break;
    case 14:  //查找和为指定大小的子数组个数
        printf("请输入你要查找的和! ");
        int k;
        k=0;
        scanf("%d",&k);
        flag=SubArrayNum(L,k);
        if(flag==INFEASIBLE) printf("线性表不存在或线性表为空! \n");
        else printf("线性表和为%d的连续子数组的个数为%d!\n",k,flag);
        getchar();getchar();
        break;
    case 15: //线性表排序
        if(SortList(L)==INFEASIBLE) printf("线性表不存在或线性表为空! \n")
        else printf("SortList功能已实现! ");
        getchar();getchar();
        break;
    case 16:{   //将线性表元素保存在文件中
```

```
                printf("请输入文件名: ");
                char filename[200];
                scanf("%s",filename);
                flag=SaveList(L,filename);
                if(flag==INFEASIBLE)
                printf("仅能对空表或者不存在的线性表进行写文件操作!  \n");
                else if(flag==OK)
                printf("写入完毕! \n");
                getchar();getchar();
                break;
            }
            case 17:{   //读取文件内容保存到线性表中
                printf("请输入文件名: ");
                char filename[200];
                scanf("%s",filename);
                flag=LoadList(L,filename);
                if(flag==INFEASIBLE)
                printf("不能对已存在的线性表进行读文件操作!  \n");
                else if(flag==OK)
                printf("读入完毕! \n");
                getchar();getchar();
                break;
            }
            case 0:  //离开系统
                break;
        }//end of switch
    }//end of while
    printf("\n感谢使用系统, 再见! \n");
    return 0;
}//end of main()
```

```
int main2(LISTS &Lists) {    //多线性表管理系统
    extern SqList L;
    Lists.length = 0;
    int choice=1;
    char name[30];
    int elem, pos, index;

    while (choice) {
        show3();
        scanf("%d", &choice);

        switch (choice) {
            case 1: // 添加新线性表
                printf("请输入线性表名称: ");
                scanf("%s", name);
                if (AddList(Lists, name))
                printf("线性表 %s 添加成功!\n", name);
                else
                printf("添加失败!\n");
                getchar();getchar();
                break;

            case 2: // 删除线性表
                printf("请输入要删除的线性表名称: ");
                scanf("%s", name);
                if (RemoveList(Lists, name)==OK){
                    printf("线性表 %s 删除成功!\n", name);
                } else {
                    printf("删除失败, 未找到该线性表!\n");
                }
                getchar();getchar();
```

```
            break;

        case 3: // 查找线性表
            printf("请输入要查找的线性表名称: ");
            scanf("%s", name);
            index = LocateList(Lists, name);
            if (index) {
                printf("找到线性表 %s, 位置为 %d\n", name, index);
                printf("线性表内容为: ");
                ListTraverse(Lists.elem[index-1].L);
                printf("\n");
            } else {
                printf("未找到线性表 %s\n", name);
            }
            getchar();getchar();
            break;

        case 4: // 操作单个线性表
            displayAllLists(Lists);  //展示所有线性表
            printf("请输入想要单独操作的线性表逻辑序号: ");
            int num;
            scanf("%d", &num);
            if (num < 0 || num > Lists.length)
                printf("序号不合法! \n");
            else
            {
                L.elem = Lists.elem[num - 1].L.elem;
                L.length = Lists.elem[num - 1].L.length;
                L.listsize = Lists.elem[num - 1].L.listsize;
                main1(L);
                Lists.elem[num - 1].L.elem = L.elem;
```

```
                    Lists.elem[num - 1].L.length = L.length;
                    Lists.elem[num - 1].L.listsize = L.listsize;
                    L.elem = nullptr;
                    L.length = L.listsize = 0;
                }
                getchar();getchar();
                break;

            case 0: // 退出系统
                break;

            default:
                printf("无效的选择, 请重新输入!\n");
                break;
        }
    }
    printf("感谢使用系统, 再见!\n");
    return 0;
}
```

5 附录 B 基于链式存储结构线性表实现的源程序

```
//def.h

#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

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

#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10

typedef int ElemType;

typedef struct LNode
{ // 单链表（链式结构）结点的定义
 ElemType data;
 struct LNode *next;
} LNode, *LinkList;

typedef struct ListNode
{ // 线性表的管理表定义
```

```
 struct
 {
  char name[20];
  LinkList L;
 }List;
 struct ListNode *next;
} ListsNode;

typedef struct {
    ListNode *head;              // 链表头指针
 ListNode *tail;              //链表尾指针
 int length;
} LISTS;


void show1(void);
void show2(void);
void show3(void);
int main1(LinkList &L);
int main2(LISTS &Lists);
status InitList(LinkList &L);
status DestroyList(LinkList &L);
status ClearList(LinkList &L);
status ListEmpty(LinkList L);
status ListLength(LinkList L);
status GetElem(LinkList L,int i,ElemType& e);
status LocateElem(LinkList L,ElemType e);
status PriorElem(LinkList L,ElemType e,ElemType& pre);
status NextElem(LinkList L,ElemType e,ElemType& next);
status ListInsert(LinkList &L,int i,ElemType e);
status ListDelete(LinkList &L,int i,ElemType &e);
```

```
status ListTraverse(LinkList L);
status ReverseList(LinkList L);
status RemoveNthFromEnd(LinkList L,int i,ElemType &e);
status SortList(LinkList &L);
status SaveList(LinkList L,char FileName[]);
status LoadList(LinkList &L, char FileName[]);
status AddList(LISTS &Lists, char ListName[]);
status RemoveList(LISTS &Lists, char ListName[]);
int LocateList(LISTS Lists, char ListName[]);
void displayAllLists(LISTS Lists);
int main1(LinkList &L);
int main2(void);

//function.cpp

#include "def.h"

void show1(void){   //总体菜单
    printf("===============单线性表操作系统菜单===============\n");
    printf("--------------------------------------------------\n");
    printf("    1. 单线性表操作        2.多线性表管理\n");
    printf("    0. 离开系统\n");
    printf("--------------------------------------------------\n");
    printf("ps:本系统在执行完一个功能后要再随便输出个字符才能再次显示菜单! \n");
    printf("    请选择你的操作[0~2]:");
}

void show2(void){    //单线性表操作菜单
    printf("\n\n");
        printf("===============单线性表操作系统菜单===============\n");
        printf("--------------------------------------------------\n");
```

```
        printf("       1. InitList       10. ListInsert\n");
        printf("       2. DestroyList    11. ListDelete\n");
        printf("       3. ClearList      12. ListTraverse\n");
        printf("       4. ListEmpty      13. ReverseList\n");
        printf("       5. ListLength     14. RemoveNthFromEnd\n");
        printf("       6. GetElem        15. SortList\n");
        printf("       7. LocateElem     16. SaveList\n");
        printf("       8. PriorElem      17. LoadList\n");
        printf("       9. NextElem        0. Exit\n");
        printf("-------------------------------------------------\n");
        printf("ps:本系统在执行完一个功能后要再随便输出个字符才能再次显示菜单! \n"
        printf("    请选择你的操作[0~17]:");
}

void show3() {    //多线性表管理菜单
    printf("\n========== 多线性表管理系统菜单 ==========\n");
    printf("1. 添加新线性表          3. 查找线性表\n");
    printf("2. 删除线性表            4. 操作单个线性表\n");
    printf("0. 退出系统\n");
    printf("==================================\n");
    printf("ps:本系统在执行完一个功能后要再随便输出个字符才能再次显示菜单! \n");
    printf("请选择操作[0-4]: ");
}

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)    //判断线性表是否存在
    return INFEASIBLE;
    L=(LinkList)malloc(sizeof(LNode));    //创造首结点
    L->next=NULL;      //初始化首结点的指针域
```

```
    return OK;
}

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)    //判断线性表是否存在
    return INFEASIBLE;
    LNode *temp;
    while(L){    //遍历释放所有结点
        temp = L->next;
        free(L);
        L=temp;
    }
    return OK;
}

status ClearList(LinkList &L)
// 如果线性表L存在，删除线性表L中的所有元素，返回OK，否则返回INFEASIBLE。
{
    if(!L)    //判断线性表是否存在
    return INFEASIBLE;
    LNode *p,*q=L->next;
    while(q){    //遍历释放除首结点以外的所有结点
        p=q;
        q=q->next;
        free(p);
    }
    L->next=NULL;  //更新首结点的指针域
    return OK;
}
```

```
status ListEmpty(LinkList L)
// 如果线性表L存在，判断线性表L是否为空，空就返回TRUE，否则返回FALSE；如果线性表L不
{
    if(!L)    //判断线性表是否存在
    return INFEASIBLE;
    if(L->next==NULL)    //判断线性表是否为空，及是否存在元素结点
    return TRUE;
    return FALSE;
}

int ListLength(LinkList L)
// 如果线性表L存在，返回线性表L的长度，否则返回INFEASIBLE。
{
    if(!L)    //判断线性表是否存在
    return INFEASIBLE;
    int length=0;
    LNode *temp=L->next;
    while(temp){    //遍历计算元素结点数量
        length++;
        temp=temp->next;
    }
    return length;
}

status GetElem(LinkList L,int i,ElemType &e)
// 如果线性表L存在，获取线性表L的第i个元素，保存在e中，返回OK；如果i不合法，返回ER
{
    if(!L)    //判断线性表是否存在
    return INFEASIBLE;
    if(i<1)
```

```
    return ERROR;
    LNode *temp=L;
    for(int j=1;j<=i;j++){    //沿线性表移动到第i个元素结点
        temp=temp->next;
        if(!temp)
        return ERROR;
    }
    e=temp->data;    //将第i个元素结点的值赋值给e
    return OK;
}

status LocateElem(LinkList L,ElemType e)
// 如果线性表L存在，查找元素e在线性表L中的位置序号；如果e不存在，返回ERROR；当线性
{
    int loc=0;
    if(!L)    //判断线性表是否存在
    return INFEASIBLE;
    LNode *temp=L->next;
    while(temp){    //遍历线性表
        loc++;
        if(temp->data==e)    //判断结点的值域的值是否为e
        return loc;
        temp=temp->next;  //若不是，继续遍历
    }
    return ERROR;
}

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)    //判断线性表是否存在
```

```
    return INFEASIBLE;
    LNode *Pre=L->next;
    if(!Pre||Pre->data==e)    //判断线性表是否为空或e是否为首元素
    return ERROR;
    LNode *temp=Pre->next;
    while(temp){     //遍历查找e
        if(temp->data==e){
            pre=Pre->data;    //将前驱赋值给pre
            return OK;
        }
        else{
            Pre=temp;
            temp=temp->next;
        }
    }
    return ERROR;
}

status NextElem(LinkList L,ElemType e,ElemType &next)
// 如果线性表L存在，获取线性表L元素e的后继，保存在next中，返回OK；如果没有后继，返
{
    if(!L)    //判断线性表是否存在
    return INFEASIBLE;
    LNode *cur=L->next;
    while(cur){     //遍历查找元素e
        if(cur->data==e){
            if(cur->next==NULL)
            break;
            next=cur->next->data;    //将后继的值赋给next
            return OK;
        }
```

```
        cur=cur->next;
    }
    return ERROR;
}

status ListInsert(LinkList &L,int i,ElemType e)
// 如果线性表L存在，将元素e插入到线性表L的第i个元素之前，返回OK；当插入位置不正确时
{
    if(!L)    //判断线性表是否存在
    return INFEASIBLE;
    if(i<1)    //判断i是否合法
    return ERROR;
    LNode *pre=L;
    LNode *temp=(LNode *)malloc(sizeof(LNode));
    if (!temp)
    return ERROR;
    for(int j=1;j<i;j++){    //移动到指定位置
        if(pre->next==NULL)    //判断位置是否合法
        return ERROR;
        pre=pre->next;
    }
    temp->data=e;    //更新插入结点的值域
    temp->next=pre->next;    //插入结点
    pre->next=temp;
    return OK;
}

status ListDelete(LinkList &L,int i,ElemType &e)
// 如果线性表L存在，删除线性表L的第i个元素，并保存在e中，返回OK；当删除位置不正确时
{
    if(!L)    //判断线性表是否存在
```

```
    return INFEASIBLE;
    LNode *temp=L;
    LNode *cur=temp->next;
    if(i<1)
    return ERROR;
    for(int j=1;j<i;j++){      //遍历移动到第i个元素结点
        if(temp->next==NULL||cur->next==NULL)    //判断i是否合法
        return ERROR;
        temp=cur;
        cur=cur->next;
    }
    e=cur->data;    //将第i个元素结点的值赋值给e
    temp->next=cur->next;    //删除第i个元素结点
    free(cur);    //释放原来第i个元素结点的空间
    return OK;
}

status ListTraverse(LinkList L)
// 如果线性表L存在，依次显示线性表中的元素，每个元素间空一格，返回OK；如果线性表L不
{
    if(!L)    //判断线性表是否存在
    return INFEASIBLE;
    LNode *temp=L->next;
    while(temp){    //遍历线性表
        printf("%d",temp->data);
        if(L->next)
        printf(" ");
        temp=temp->next;
    }
    return OK;
}
```

```
status SaveList(LinkList L,char FileName[])
// 如果线性表L存在，将线性表L的的元素写到FileName文件中，返回OK，否则返回INFEASIBL
{
    if(!L)    //判断线性表是否存在
    return INFEASIBLE;
    FILE *fq=fopen(FileName,"w");    //打开文件
    if(!fq)    //判断文件是否打开成功
    return ERROR;
    LNode *temp=L->next;
    while(temp){
        fprintf(fq,"%d ",temp->data);    //遍历线性表并将内容输出到文件中
        temp=temp->next;
    }
    fclose(fq);    //关闭文件
    return OK;
}

status LoadList(LinkList &L,char FileName[])
// 如果线性表L不存在，将FileName文件中的数据读入到线性表L中，返回OK，否则返回INFEA
{
    if(L)    //判断线性表是否存在
    return INFEASIBLE;
    FILE *fq=fopen(FileName,"r");    //打开文件
    if(!fq)    //判断文件是否打开成功
    return ERROR;
    L=(LinkList)malloc(sizeof(LNode));    //创建线性表
    L->next=NULL;
    LNode *temp=L;
    ElemType e=0;
    while(fscanf(fq,"%d",&e)==1){    //读取文件并赋值给线性表
```

```
        temp->next=(LNode *)malloc(sizeof(LNode));
        temp->next->next=NULL;
        temp->next->data=e;
        temp=temp->next;
    }
    fclose(fq);
    return OK;
}

status ReverseList(LinkList L){
//如果线性表不存在，返回INFEASIBLE；否则翻转线性表
    if(!L)    //判断线性表是否存在
    return INFEASIBLE;
    LNode *pre=NULL;    //保存前一个结点
    LNode *cur=L->next;    //保存当前结点
    LNode *next=NULL;    //保存下一个结点
    while(cur){
        next=cur->next;    //保存下一个结点
        cur->next=pre;    //翻转当前节点与前一个结点
        pre=cur;    //顺着线性表向后移动
        cur=next;
    }
    L->next=pre;    //更新首元素结点
    return OK;
}

status RemoveNthFromEnd(LinkList L,int i,ElemType &e){
    status length=ListLength(L);    //获得线性表长度
    if(!L||!length)    //判断线性表是否存在以及是否为空
    return INFEASIBLE;
    if(i>length||i<1)    //判断i是否合法
```

```
    return ERROR;
    i=length-i+1;    //获得要删除元素的正向位置
    ListDelete(L,i,e);
    return OK;
}

status SortList(LinkList &L){
    if(L==NULL||L->next==NULL)    //判断线性表是否存在以及是否为空
    return INFEASIBLE;
    int flag=FALSE;    //线性表是否排序完成的标志
    status length=ListLength(L);  //获得线性表长度
    while(!flag){    //冒泡排序
        LNode *p1=L->next;
        LNode *p2=p1->next;
        flag=TRUE;
        for(int i=0;i<length-1&&p1&&p2;i++){
            if(p1->data>p2->data){
                int temp=p1->data;
                p1->data=p2->data;
                p2->data=temp;
                flag=FALSE;
            }
            p1=p2;
            p2=p2->next;
        }
        length--;
    }
    return OK;
}

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

```
// 只需要在Lists中增加一个名称为ListName的空线性表，线性表数据又后台测试程序插入。
{
    ListNode *p=Lists.head;
    while (p!=NULL) {    //判断线性表是否已存在
        if (strcmp(p->List.name, ListName) == 0)
        return ERROR;
        p=p->next;
    }
    LinkList NewList=NULL;    //创建新线性表
    if(InitList(NewList)!=OK)
    return ERROR;
    ListNode *NewNode=(ListNode *)malloc(sizeof(ListNode));   //创建新线性表结点
    if(!NewNode)
    return ERROR;
    NewNode->List.L = NewList;
    strcpy(NewNode->List.name, ListName);
    if(!Lists.head){
        Lists.head=NewNode;
        Lists.tail=NewNode;
    }
    else{
        Lists.tail->next=NewNode;
        Lists.tail=NewNode;
    }
    Lists.length++;
    return OK;
}

int LocateList(LISTS Lists, char ListName[])
// 在Lists中查找一个名称为ListName的线性表，成功返回在多线性表中的位置，否则返回0
{
```

```
    ListNode *p=Lists.head;
    int pos=1;
    while (p!=NULL) {    //遍历查找线性表
        if (strcmp(p->List.name, ListName) == 0)
        return pos;
        p=p->next;
        pos++;
    }
    return ERROR;
}

status RemoveList(LISTS &Lists, char ListName[])
// Lists中删除一个名称为ListName的线性表
{
    ListNode *prev = NULL;
    ListNode *curr = Lists.head;

    while (curr != NULL) {
        if (strcmp(curr->List.name, ListName) == 0) {
            // 找到要删除的节点
            DestroyList(curr->List.L);  // 销毁线性表
            if (prev == NULL) {       // 从链表中删除节点
                // 删除的是头节点
                Lists.head = curr->next;
            } else {
                prev->next = curr->next;
            }
            free(curr);  // 释放节点内存
            Lists.length--;
            return OK;
        }
```

```
        prev = curr;
        curr = curr->next;
    }
    return ERROR;
}

void displayAllLists(LISTS Lists) {    //展示所有线性表
    printf("当前线性表有: \n");
    int i=1;
    ListNode *p=Lists.head;
    while (p!=NULL) {    //遍历线性表
        printf("%d:%s\n",i++,p->List.name);
        p=p->next;
    }
}

//run.cpp

#include"def.h"

LinkList L=NULL;;
LISTS Lists;

int main(void){
    int mode=0;
    show1();
    while(1){
        scanf("%d",&mode);
        switch (mode)
        {
        case 1:
```

```
            main1(L);    //单线性表操作
            break;
        case 2:
            main2(Lists);    //多线性表管理
            break;
        case 0:     // 退出系统
            printf("感谢使用系统, 再见!\n");
            return 0;
        default:
            printf("无效的选择, 请重新输入!\n");
        }
        show1();
    }
}

int main1(LinkList &L){       //单个线性表操作系统
    ElemType e=0;
    int op=1,i=0;
    status flag=0;
    while(op){
        show2();
        scanf("%d",&op);
        switch(op){
            case 1: //创建线性表
                if(InitList(L)==OK) printf("线性表创建成功! \n");
                else printf("线性表创建失败! \n");
                getchar();getchar();    //清除输入列多余的换行符及其他字符
                break;
            case 2: //销毁线性表
                if(DestroyList(L)==OK) printf("线性表销毁成功! \n");
                else printf("线性表销毁失败! ");
```

```
        getchar();getchar();
        break;
    case 3: //清空线性表
        flag==ClearList(L);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
        else if(flag==OK) printf("线性表清空成功! \n");
        getchar();getchar();
        break;
    case 4:  //判断线性表是否为空
        flag=ListEmpty(L);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
        else if(flag==TRUE) printf("线性表为空! \n");
        else printf("线性表不为空! \n") ;
        getchar();getchar();
        break;
    case 5: //获得线性表长度
        flag=ListLength(L);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
        else printf("线性表长度为%d!\n",flag);
        getchar();getchar();
        break;
    case 6:  //获得指定位置线性表元素
        printf("请输入你所要获得的元素位置: ");
        scanf("%d",&i);
        flag=GetElem(L,i,e);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
        else {
            while(flag==ERROR){
                printf("请输入合法的位置\n");
                printf("请重新输入你要获得的元素位置: ");
                scanf("%d",&i);
```

```
            }
            printf("第%d个数据元素为%d\n",i,e);
        }
        getchar();getchar();
        break;
    case 7: //定位指定元素
        printf("请输入你要查找的元素! \n");
        scanf("%d",&e);
        flag=LocateElem(L,e);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
        else if(flag)  printf("元素%d在第%d个位置\n",e,flag);
        else printf("未查找到该元素%d\n",e);
        getchar();getchar();
        break;
    case 8:{ //获得指定元素前驱
        ElemType pre;
        pre=0;
        printf("请输入你要查找前驱的元素: ");
        scanf("%d",&e);
        flag=PriorElem(L,e,pre);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
        else if(flag) printf("%d的前驱为%d",e,pre);
        else printf("未找到前驱! \n");
        getchar();getchar();
        break;
    }
    case 9:{ //获得指定元素后继
        ElemType next;
        next=0;
        printf("请输入你要查找后继的元素: ");
        scanf("%d",&e);
```

```
        flag=NextElem(L,e,next);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
        else if(flag) printf("%d的后继为%d",e,next);
        else printf("未找到后继! \n");
        getchar();getchar();
        break;
    }
    case 10:  //插入元素
        printf("请输入你要插入的元素以及要插入的位置! ");
        scanf(" %d %d",&e,&i);
        flag=ListInsert(L,i,e);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
        else if(flag==OK) printf("ListInsert功能实现成功! ");
        else printf("ListInsert功能实现失败! ");
        getchar();getchar();
        break;
    case 11:  //删除指定位置元素
        printf("请输入你要删除的元素位置! ");
        scanf(" %d",&i);
        flag=ListDelete(L,i,e);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
        else if(flag==OK)
        printf("ListDelete功能实现成功! \n删除的元素为%d!\n",e);
        else printf("ListDelete功能实现失败! ");
        getchar();getchar();
        break;
    case 12:  //遍历线性表
        if(!L) printf("线性表是空表! \n");
        else printf("线性表内容如下: \n");
        ListTraverse(L);
        getchar();getchar();
```

```
        break;
    case 13:  //翻转线性表
        flag=ReverseList(L);
        if(flag==INFEASIBLE) printf("线性表不存在! \n");
        else printf("线性表反转成功! \n");
        getchar();getchar();
        break;
    case 14: //删除倒数第i个位置的元素
        printf("请输入你要删除的倒数第i个结点! \n");
        scanf("%d",&i);
        flag=RemoveNthFromEnd(L,i,e);
        if(flag==INFEASIBLE) printf("线性表不存在或线性表为空! \n");
        else printf("RemoveNthFromEnd功能实现成功! \n删除的元素为%d!\n",e)
        getchar();getchar();
        break;
    case 15:  //线性表排序
        if(SortList(L)==INFEASIBLE) printf("线性表不存在或线性表为空! \n")
        else printf("SortList功能已实现! ");
        getchar();getchar();
        break;
    case 16:{  //将线性表保存到文件
        printf("请输入文件名: ");
        char filename[200];
        scanf("%s",filename);
        flag=SaveList(L,filename);
        if(flag==INFEASIBLE)
        printf("仅能对空表或者不存在的线性表进行写文件操作!  \n");
        else if(flag==OK)
        printf("写入完毕! \n");
        getchar();getchar();
        break;
```

```
            }
            case 17:{   //读取文件保存到线性表
                printf("请输入文件名: ");
                char filename[200];
                scanf("%s",filename);
                flag=LoadList(L,filename);
                if(flag==INFEASIBLE)
                printf("不能对已存在的线性表进行读文件操作!  \n");
                else if(flag==OK)
                printf("读入完毕! \n");
                getchar();getchar();
                break;
            }
            case 0: //结束系统
                break;
        }//end of switch
    }//end of while
    printf("感谢使用本系统, 再见! \n");
    return 0;
}//end of main()

int main2(LISTS &Lists) {    //多线性表管理系统
    extern LinkList L;
    Lists.head=NULL;
    Lists.tail=Lists.head;
    Lists.length=0;
    int choice=1;
    char name[30];
    int elem, pos, index;

    while (choice) {
```

```
show3();
scanf("%d", &choice);

switch (choice) {
    case 1: // 添加新线性表
        printf("请输入线性表名称: ");
        scanf("%s", name);
        if (AddList(Lists, name))
        printf("线性表 %s 添加成功!\n", name);
        else
        printf("添加失败!\n");
        getchar();getchar();
        break;

    case 2: // 删除线性表
        printf("请输入要删除的线性表名称: ");
        scanf("%s", name);
        if (RemoveList(Lists, name)==OK){
            printf("线性表 %s 删除成功!\n", name);
        } else {
            printf("删除失败, 未找到该线性表!\n");
        }
        getchar();getchar();
        break;

    case 3: // 查找线性表
        printf("请输入要查找的线性表名称: ");
        scanf("%s", name);
        index = LocateList(Lists, name);
        if (index) {
            printf("成功找到线性表%s, 线性表在第%d个位置! \n",name,index);
```

```
        } else {
            printf("未找到线性表 %s\n", name);
        }
        getchar();getchar();
        break;

    case 4: // 操作单个线性表
        displayAllLists(Lists);
        printf("请输入想要单独操作的线性表逻辑序号: ");
        int num;
        scanf("%d", &num);
        if (num < 0 || num > Lists.length)
            printf("序号不合法! \n");
        else
        {
            ListNode *p=Lists.head;
            for(int pos=1;pos<num&&p!=NULL;pos++)
            p=p->next;
            L=p->List.L;
            main1(L);
        }
        L=NULL;
        getchar();getchar();
        break;

    case 0: // 退出系统
        break;

    default:
        printf("无效的选择, 请重新输入!\n");
```

```
                break;
            }
        }
        printf("感谢使用系统，再见!\n");
        return 0;
}
```

6 附录 C 基于二叉链表二叉树实现的源程序

```
//def.h

#include "stdio.h"
#include "stdlib.h"
#include <bits/stdc++.h>
using namespace std;

#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define ERROR2 -3
#define INFEASIBLE -1
#define overflow -2
#define init_size 50

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 {
    int pos;
    TElemType data;
} DEF;

typedef struct
{
    char name[30];
    BiTNode *T;
} TreeEntry;

typedef struct
{
    TreeEntry *elem; // 改为动态数组指针
    int length;      // 当前实际存储了多少个二叉树
    int listsize;    // 当前已分配的容量
} LISTS;

void visit(BiTree T);
void createinit(void);
status CreateBiTree(BiTree &T,TElemType definition[]);
status DestroyBiTree(BiTree &T);
status BiTreeEmpty(BiTree T);
status ClearBiTree(BiTree &T);
int BiTreeDepth(BiTree T);
BiTNode* LocateNode(BiTree T,KeyType e);
status Assign(BiTree &T,KeyType e,TElemType value);
BiTNode* GetSibling(BiTree T,KeyType e);
status InsertNode(BiTree &T,KeyType e,int LR,TElemType c);
BiTree FindFather(BiTree T,KeyType e);
```

```
status DeleteNode(BiTree &T,KeyType e);
status PreOrderTraverse(BiTree T,void (*visit)(BiTree));
status InOrderTraverse(BiTree T,void (*visit)(BiTree));
status PostOrderTraverse(BiTree T,void (*visit)(BiTree));
status LevelOrderTraverse(BiTree T,void (*visit)(BiTree));
int MaxPathSum(BiTree T);
BiTree LowestCommonAncestor(BiTree T, KeyType e1, KeyType e2);
void InvertTree(BiTree T);
void save(BiTree T,FILE *fp);
status SaveBiTree(BiTree T, char FileName[]);
void load(BiTree &T,FILE *fp);
status LoadBiTree(BiTree &T, char FileName[]);
void InitLists(LISTS &Lists);
status Addtree(LISTS &Lists, char *name);
status Removetree(LISTS &Lists, char *name);
int Locatetree(LISTS &Lists, char *name);
void TraversLists(LISTS Lists);
void show1();
void show2(BiTree T);
void show3();

//function.cpp

#include"def.h"

// ------------------------------------------------------------------
//                          单二叉树操作部分
// ------------------------------------------------------------------
int tot=0;
unordered_map<int, int> hashb;
```

```
void createinit()
{
    tot = 0;
    hashb.clear();
}

status CreateBiTree(BiTree &T,TElemType definition[])
{
// 创建二叉树，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
// 每个结点对应一个整型的关键字和一个字符串，当关键字为0时，表示空子树，为-1表示输
    if (definition[tot].key == 0 || definition[tot].key == -1)
    {
        ++tot;
        return OK;
    }

    if (hashb[definition[tot].key] != 0)
    return ERROR;
    hashb[definition[tot].key] = 1;

    T = (BiTree)malloc(sizeof(BiTNode));
    if (!T)
    return overflow;
    T->data = definition[tot];
    T->lchild = T->rchild = NULL;
    ++tot;

    if (CreateBiTree(T->lchild, definition) == ERROR)
    return ERROR;

    if (CreateBiTree(T->rchild, definition) == ERROR)
```

```
    return ERROR;

    return OK;
}

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

status DestroyBiTree(BiTree &T)
{
// 销毁二叉树，并删除所有结点，释放结点空间
    if (T == NULL)
        return OK;

    DestroyBiTree(T->lchild);
    DestroyBiTree(T->rchild);

    free(T);
    T = NULL;
    return OK;
}

status ClearBiTree(BiTree &T)
{
// 清空二叉树，只是将二叉树清空了，是空树，但是在内存中还存在空间，只是值不确定。
    if(T==NULL)
    return OK;

    if (T == NULL)
```

```
        return OK;

    ClearBiTree(T->lchild);
    ClearBiTree(T->rchild);

    T->data.key = 0;
    T->data.others[0] = '\0';
    return OK;
}

status BiTreeEmpty(BiTree T)
{
// 判定空二叉树：函数名称是；初始条件是二叉树T存在；操作结果是若T为空二叉树则返回T
    if (T == NULL)
        return TRUE;

    if (BiTreeEmpty(T->lchild) == FALSE)
        return FALSE;
    if (BiTreeEmpty(T->rchild) == FALSE)
        return FALSE;

    if (T->data.key == 0)
        return TRUE;
    else
        return FALSE;
}

int BiTreeDepth(BiTree T)
{
// 求二叉树深度
    if(T==NULL)
```

```
    return 0;
    int ldep=BiTreeDepth(T->lchild);
    int rdep=BiTreeDepth(T->rchild);
    return (ldep>rdep?ldep:rdep)+1;
}

BiTNode *LocateNode(BiTree T, KeyType e)
// 查找结点
{
    if (T == NULL)
        return NULL;

    if (T->data.key == e)
        return T;

    BiTree p1 = LocateNode(T->lchild, e);
    BiTree p2 = LocateNode(T->rchild, e);

    if (p1 != NULL)
        return p1;
    if (p2 != NULL)
        return p2;
    return NULL;
}

status Assign(BiTree &T,KeyType e,TElemType value)
{
// 结点赋值
    BiTree p1=LocateNode(T,e);
    BiTree p2=LocateNode(T,value.key);
    if(p1==NULL||(p2!=NULL&&e!=value.key)){
```

```
        return ERROR;
    }
    p1->data=value;
    return OK;
}

BiTNode* GetSibling(BiTree T,KeyType e)
//实现获得兄弟结点
{
    if(T==NULL)
    return NULL;

    if(T->rchild&&T->rchild->data.key==e)
    return T->lchild;
    if(T->lchild&&T->lchild->data.key==e)
    return T->rchild;

    BiTNode* pl=GetSibling(T->lchild,e);
    if(pl!=NULL)
    return pl;
    BiTNode* pr=GetSibling(T->rchild,e);
    if(pr!=NULL)
    return pr;

    return NULL;
}

status InsertNode(BiTree &T,KeyType e,int LR,TElemType c)
//插入结点
// 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。
```

```
{
    if(LR==-1){
        BiTree root=(BiTree)malloc(sizeof(BiTNode));
        if(!root)
        return overflow;
        root->rchild=T;
        root->lchild==NULL;
        root->data=c;
        T=root;
        return OK;
    }
    // 检验有没有可替换节点
    BiTree pos=LocateNode(T,e);
    if(pos==NULL)
    return ERROR;
    // 保证唯一性
    BiTree pos2=LocateNode(T,c.key);
    if(pos2!=NULL)
    return ERROR;
    BiTree newnode=(BiTree)malloc(sizeof(BiTNode));
    if(!newnode)
    return overflow;
    newnode->data=c;
    newnode->lchild=NULL;
    if(LR==1){
        // 此时插入到右子树
        newnode->rchild=pos->rchild;
        pos->rchild=newnode;
    }
    else if(LR==0){
        // 此时插入到左子树
```

```
        newnode->rchild=pos->lchild;
        pos->lchild=newnode;
    }
    return OK;
}

BiTree FindFather(BiTree T,KeyType e){
//查找父亲结点
    if(T==NULL)
    return NULL;
    if(T->data.key==e)
    return T;
    if(T->lchild && T->lchild->data.key == e || T->rchild && T->rchild->data.key =
    return T;
BiTree p1=FindFather(T->lchild,e);
BiTree p2=FindFather(T->rchild,e);
if(p1!=NULL)
return p1;
if(p2!=NULL)
return p2;
return NULL;
}

status DeleteNode(BiTree &T,KeyType e)
//删除结点
{
    BiTree p = FindFather(T, e);
    if (p == NULL)
    return ERROR;
    BiTree cur=T;
    int LR=-1;
```

```
// 找到要删除的节点
if(p->data.key!=e){
    if(p->lchild->data.key==e){
        cur=p->lchild;
        LR=0;
    }
    else{
        cur=p->rchild;
        LR=1;
    }
}
// 开始删除节点
if(cur->lchild==NULL&&cur->rchild==NULL){
    if(LR==-1)
    T=NULL;
    else if(LR==0)
    p->lchild=NULL;
    else if(LR==1)
    p->rchild=NULL;
    free(cur);
}
else if(cur->lchild!=NULL&&cur->rchild==NULL){
    // 结点只有一个左子树
    if(LR==-1)
    T=cur->lchild;
    else if(LR==0)
    p->lchild=cur->lchild;
    else if(LR==1)
    p->rchild=cur->lchild;
    free(cur);
}
```

```
    else if(cur->lchild==NULL&&cur->rchild!=NULL){
        // 结点只有一个右子树
        if(LR==-1)
        T=cur->rchild;
        else if(LR==0)
        p->lchild=cur->rchild;
        else if(LR==1)
        p->rchild=cur->rchild;
        free(cur);
    }
    else if(cur->lchild!=NULL&&cur->rchild!=NULL){
        // 结点有两个子树
        BiTree l=cur->lchild;
        BiTree r=cur->rchild;
       if(LR==-1)
        T=cur->lchild;
        else if(LR==0)
        p->lchild=cur->lchild;
        else if(LR==1)
        p->rchild=cur->lchild;
        while(l->rchild!=NULL)
        l=l->rchild;
        l->rchild=r;
        free(cur);
    }
    return OK;
}

status PreOrderTraverse(BiTree T,void (*visit)(BiTree))
//先序遍历二叉树T
{
```

```
    if (T == NULL)
        return ERROR;
    BiTree stack[1000] = {NULL};
    int top = -1;
    stack[++top] = T;
    while (top != -1)
    {
        BiTree cur = stack[top--];
        visit(cur);
        if (cur->rchild)
            stack[++top] = cur->rchild;
        if (cur->lchild)
            stack[++top] = cur->lchild;
    }
    return OK;
}

status InOrderTraverse(BiTree T,void (*visit)(BiTree))
//中序遍历二叉树T
{
    if (T == NULL)
        return ERROR;
    if (T->lchild)
        InOrderTraverse(T->lchild, visit);
    visit(T);
    if (T->rchild)
        InOrderTraverse(T->rchild, visit);
    return OK;
}

status PostOrderTraverse(BiTree T,void (*visit)(BiTree))
```

```
//后序遍历二叉树T
{
    if (T == NULL)
        return ERROR;
    if (T->lchild)
        PostOrderTraverse(T->lchild, visit);
    if (T->rchild)
        PostOrderTraverse(T->rchild, visit);
    visit(T);
    return OK;
}

status LevelOrderTraverse(BiTree T,void (*visit)(BiTree))
//按层遍历二叉树T
{
    if (T == NULL)
        return ERROR;
    BiTree queue[1000] = {NULL};
    int down = 0, top = -1;
    queue[++top] = T;

    while (down <= top)
    {
        BiTree cur = queue[down++];
        visit(cur);
        if (cur->lchild)
            queue[++top] = cur->lchild;
        if (cur->rchild)
            queue[++top] = cur->rchild;
    }
    return OK;
```

```
}

int MaxPathSum(BiTree T)
{
// 最大路径和，操作结果是返回根节点到叶子结点的最大路径和;
    if (T == NULL)
        return 0;

    // 递归计算左右子树的最大路径和
    int left = MaxPathSum(T->lchild);
    int right = MaxPathSum(T->rchild);

    // 返回当前节点的 key + 左右子树中较大的路径和
    return T->data.key + (left > right ? left : right);
}


BiTree LowestCommonAncestor(BiTree T, KeyType e1, KeyType e2)
{
// 最近公共祖先，操作结果是该二叉树中e1节点和e2节点的最近公共祖先
    if (T == NULL)
        return NULL;

    // 如果当前节点就是 e1 或 e2，直接返回它
    if (T->data.key == e1 || T->data.key == e2)
        return T;

    // 在左右子树中递归查找
    BiTree left = LowestCommonAncestor(T->lchild, e1, e2);
    BiTree right = LowestCommonAncestor(T->rchild, e1, e2);
```

```
    if (left != NULL && right != NULL)
        return T; // 若分别出现在左右子树，当前节点即为最近公共祖先

    return left != NULL ? left : right; // 否则返回非空的那一侧
}


void InvertTree(BiTree T)
{
// 翻转二叉树，操作结果是将T翻转，使其所有节点的左右节点互换;
    if (T == NULL)
        return;

    // 交换当前节点的左右孩子
    BiTree temp = T->lchild;
    T->lchild = T->rchild;
    T->rchild = temp;

    // 递归处理左右子树
    InvertTree(T->lchild);
    InvertTree(T->rchild);
}

void save(BiTree T,FILE *fp)
{
//文件存储
    if (T==NULL)
    {
        fprintf(fp, "#\n");
        return;
    }
```

```
    fprintf(fp, "%d %s\n",T->data.key, T->data.others);
    save(T->lchild, fp);
    save(T->rchild, fp);
}

status SaveBiTree(BiTree T, char FileName[])
//将二叉树的结点数据写入到文件FileName中
{
    FILE *fp = fopen(FileName, "w");
    if (!fp)
    return ERROR;
    save(T, fp);
    fclose(fp);
    return OK;
}

void load(BiTree &T,FILE *fp)
{
    char list[100];
    if(!fgets(list,sizeof(list),fp))
    return;
    if(list[0]=='#'){
        T=NULL;
        return;
    }
    T=(BiTree)malloc(sizeof(BiTNode));
    sscanf(list,"%d %s",&T->data.key,T->data.others);
    load(T->lchild,fp);
    load(T->rchild,fp);
}
```

```
status LoadBiTree(BiTree &T, char FileName[])
//读入文件FileName的结点数据，创建二叉树
{
    FILE *fp = fopen(FileName, "r");
    if (!fp)
        return ERROR;
    load(T, fp);
    fclose(fp);
    return OK;
}

// ------------------------------------------------------------------
//                          多线性表管理部分
// ------------------------------------------------------------------

// 创建线性表
void InitLists(LISTS &Lists)
{
    Lists.elem = (TreeEntry *)malloc(init_size * sizeof(TreeEntry));
    Lists.length = 0;
    Lists.listsize = init_size;
}


status Addtree(LISTS &Lists, char *name)
{
// 添加二叉树
    if (Lists.length == Lists.listsize)
    {
        // 扩容
        TreeEntry *newelem = (TreeEntry *)realloc(Lists.elem, (Lists.listsize + in
```

```
        if (!newelem)
            return overflow;
        // 扩容后释放原内存
        for (int i = 0; i < Lists.length; ++i)
            DestroyBiTree(Lists.elem[i].T); // 释放每棵二叉树
        free(Lists.elem);                   // 释放动态数组本身
        Lists.elem = newelem;
        Lists.listsize += init_size;
    }

    // 初始化空树
    Lists.elem[Lists.length].T = NULL;
    strcpy(Lists.elem[Lists.length].name, name);
    ++Lists.length;
    return OK;
}


status Removetree(LISTS &Lists, char *name)
{
// 移除二叉树
    for (int i = 0; i < Lists.length; ++i)
    {
        if (strcmp(Lists.elem[i].name, name) == 0)
        {
            // 释放二叉树内存
            DestroyBiTree(Lists.elem[i].T);

            // 将后面的元素全部向前移动
            for (int j = i; j < Lists.length - 1; ++j)
                Lists.elem[j] = Lists.elem[j + 1];
```

```
            --Lists.length;
            return OK;
        }
    }
    // 删除失败
    return ERROR;
}


int Locatetree(LISTS &Lists, char *name)
{
// 在Lists中查找一个名称为name的二叉树，成功返回逻辑序号，否则返回0
    for (int i = 0; i < Lists.length; ++i)
        if (strcmp(Lists.elem[i].name, name) == 0)
            return i + 1;
    return 0;
}

void TraversLists(LISTS Lists){
    for (int i = 0; i <Lists.length; ++i)
    printf("%d: %s\n",i+1,Lists.elem[i].name);
}

void show1()
{
    printf("\n");
    printf("     Menu for Linear Table Type On Sequence Structure \n");
    printf("------------------------------------------------------------\n");
    printf("                    请选择你要操作的类型: \n");
    printf("           1. 单二叉树          2.多二叉树 \n");
```

```
    printf("           0. 退出\n");
    printf("-------------------------------------------------------------\n");
    printf("                    请选择你的操作[0~2]:");
}

void show2(BiTree T)
{
    printf("\n");
    printf("      Menu for Linear Table On Sequence Structure         \n");
    printf("-------------------------------------------------------------\n");
    printf("       1. CreateBiTree         2. DestroyBiTree         \n");
    printf("       3. ClearBiTree          4. BiTreeEmpty           \n");
    printf("       5. BiTreeDepth          6. LocateNode                \n");
    printf("       7. Assign               8. GetSibling            \n");
    printf("       9. InsertNode           10. DeleteNode            \n");
    printf("       11. PreOrderTraverse    12.InOrderTraverse       \n");
    printf("       13. PostOrderTraverse   14.LevelOrderTraverse    \n");
    printf("       15. SaveBiTree          16.LoadBiTree            \n");
    printf("       17. MaxPathSum          18.LowestCommonAncestor  \n");
    printf("       19. InvertTree          20.Change type          \n");
    printf("       0 . Exit                 \n");
    printf("ps:本系统在执行完一个功能后要再随便输出个字符才能再次显示菜单欧\n");
    printf("-------------------------------------------------------------\n");
    printf("二叉树实时预览（这里是按层遍历）:");
    status j = LevelOrderTraverse(T, visit);
    if (j == ERROR)
        printf("二叉树不存在 \n");
    else
        printf("\n");
    printf("\n            请选择你的操作[0~20]:");
}
```

```
void show3()
{
    printf("\n");
    printf("         Menu for Linear Table On Sequence Structure        \n");
    printf("------------------------------------------------------------ \n");
    printf("       1. Addtree          2. Removetree                 \n");
    printf("       3. Locatetree       4. operator single tree     \n");
    printf("       0.Exit             \n");
    printf("ps:本系统在执行完一个功能后要再随便输出个字符才能再次显示菜单欧\n");
    printf("------------------------------------------------------------\n");
    printf("\n             请选择你的操作[0~4]:");
}

//run.cpp

#include"def.h"
BiTree T;
LISTS Lists;
unordered_set<string> namehash; // 用于防止多二叉树管理时出现两个相同的二叉树树名

void operator1(BiTree &T)
{
    int op2;
    show2(T);
    scanf("%d", &op2);
    printf("\n");

    while (op2 != 20)
    {
        if (op2 == 0)
```

```
{
    printf("%s", "感谢使用本系统 ");
    exit(0);
}
else if (op2 == 1)
{
    if (T != NULL)
        printf("二叉树已存在, 请勿重复创建\n");
    else
    {
        // 创建二叉树
        TElemType definition[100];
        int i = 0;
        do
        {
            scanf("%d%s", &definition[i].key, definition[i].others);
        } while (definition[i++].key != -1);
        createinit();
        status j = CreateBiTree(T, definition);
        if (j == ERROR)
            printf("关键字重复, 创建失败\n");
        else if (j == overflow)
            printf("内存溢出, 创建失败\n");
        else
            printf("创建成功! \n");
    }
}
else if (op2 == 16)
{
    if (T != NULL)
        printf("二叉树已存在, 请勿重复创建\n");
```

```
        else
        {
            printf("请输入文件名: ");
            char filename[200];
            scanf("%s", filename);
            status j = LoadBiTree(T, filename);
            if (j == ERROR)
                printf("文件打开失败!  \n");
            else if (j == OK)
                printf("读入完毕!  \n");
        }
    }
    else
    {
        // 现在对于所有操作来说二叉树都应该存在, 所以一次性判断二叉树是否不存在
        if (T == NULL)
        {
            printf("不能对不存在的二叉树进行操作!  \n");
        }
        else if (op2 == 2)
        {

            DestroyBiTree(T);
            printf("销毁成功!  \n");
        }
        else if (op2 == 3)
        {
            ClearBiTree(T);
            printf("清空成功!  \n");
        }
        else if (op2 == 4)
```

```
{
    status j = BiTreeEmpty(T);
    if (j == TRUE)
        printf("二叉树是空哒!  \n");
    else if (j == FALSE)
        printf("二叉树不是空的! \n");
}
else if (op2 == 5)
{
    int j = BiTreeDepth(T);
    printf("二叉树的深度是%d\n", j);
}
else if (op2 == 6)
{
    printf("想要获得哪个关键字对应的结点: ");
    KeyType e;
    scanf("%d", &e);
    BiTNode *j = LocateNode(T, e);
    if (j == NULL)
        printf("找不到该节点 \n");
    else
        printf("这个节点是: %d %s\n", j->data.key, j->data.others);
}
else if (op2 == 7)
{
    printf("输入想要赋值的节点关键字以及赋予的值: ");
    KeyType e;
    TElemType value;
    scanf("%d %d %s", &e, &value.key, &value.others);
    status j = Assign(T, e, value);
    if (j == ERROR)
```

```
            printf("没有找到想要赋值的节点 \n");
        else if (j == ERROR2)
            printf("赋值的关键字不唯一，赋值失败  \n");
        else
            printf("赋值成功！\n");
    }
    else if (op2 == 8)
    {
        printf("想要获得哪个关键字对应结点的兄弟节点：");
        KeyType e;
        scanf("%d", &e);

        // 先判断这个点自己在不在
        BiTNode *i = LocateNode(T, e);
        if (i == NULL)
            printf("找不到对应关键字的节点 \n");
        else
        {
            // 找兄弟节点
            BiTNode *j = GetSibling(T, e);
            if (j == NULL)
                printf("该兄弟节点不存在 \n");
            else
                printf("这个节点的兄弟节点是：%d %s\n", j->data.key, j->da
        }
    }
    else if (op2 == 9)
    {
        printf("依次输入被插入的节点关键字，插入方向，插入的节点值：");
        KeyType e;
        int LR;
```

```
        TElemType c;
        scanf("%d %d %d %s", &e, &LR, &c.key, &c.others);
        status j = InsertNode(T, e, LR, c);
        if (j == overflow)
            printf("内存溢出, 插入失败 \n");
        else if (j == ERROR)
            printf("没有找到被插入的节点 \n");
        else if (j == ERROR2)
            printf("插入节点的关键字不唯一, 插入失败  \n");
        else
            printf("插入成功! \n");
    }
    else if (op2 == 10)
    {
        printf("输入想要删除的节点的关键字: ");
        KeyType e;
        scanf("%d", &e);
        status j = DeleteNode(T, e);
        if (j == ERROR)
            printf("找不到要删除的节点 \n");
        else
            printf("删除成功 ");
    }
    else if (op2 == 11)
    {
        printf("二叉树先序遍历结果如下:\n");
        PreOrderTraverse(T, visit);
    }
    else if (op2 == 12)
    {
        printf("二叉树中序遍历结果如下:\n");
```

```
        InOrderTraverse(T, visit);
    }
    else if (op2 == 13)
    {
        printf("二叉树后序遍历结果如下:\n");
        PostOrderTraverse(T, visit);
    }
    else if (op2 == 14)
    {
        printf("二叉树按层遍历结果如下:\n");
        LevelOrderTraverse(T, visit);
    }
    else if (op2 == 15)
    {
        printf("请输入文件名: ");
        char filename[200];
        scanf("%s", filename);
        status j = SaveBiTree(T, filename);
        if (j == ERROR)
            printf("文件打开失败!  \n");
        else if (j == OK)
            printf("写入完毕!  \n");
    }
    else if (op2 == 17)
    {
        printf("最长路径为%d\n", MaxPathSum(T));
    }
    else if (op2 == 18)
    {
        printf("请输入想要寻找最近公共祖先的2个节点的关键字:");
        KeyType e1, e2;
```

```
                scanf("%d %d", &e1, &e2);
                BiTNode *p = LowestCommonAncestor(T, e1, e2);
                if (p == NULL)
                    printf("没有最近的公共祖先\n");
                else
                    printf("最近的公共祖先是%d %s\n", p->data.key, p->data.others);
            }
            else if (op2 == 19)
            {
                InvertTree(T);
                printf("翻转成功! \n");
            }
        }
        getchar(); // 读取掉输入去残留的换行符
        getchar(); // 使结果在屏幕上保持一段时间
        show2(T);
        scanf("%d", &op2);
        printf("\n");
    }
    return;
}

void operator2(LISTS &Lists)
{
    InitLists(Lists);
    show3();
    int op3;
    scanf("%d", &op3);
    printf("\n");

    while (1)
```

```
{
    if (op3 == 0)
    {
        printf("%s", "感谢使用本系统 ");
        exit(0);
    }
    else if (op3 == 1)
    {
        printf("请输入新建二叉树树名: ");
        char name[200];
        scanf("%s", name);
        string sname = name;
        if (namehash.count(sname) != 0)
            printf("名称已存在, 创建失败! \n");
        else
        {
            status j = Addtree(Lists, name);
            if (j == OK)
                printf("创建成功! \n");
            else
                printf("内存溢出, 创建失败! \n");
            namehash.insert(sname);
        }
    }
    else if (op3 == 2)
    {
        printf("请输入想要移除的二叉树树名: ");
        char name[200];
        scanf("%s", name);
        string sname = name;
        if (namehash.count(sname) == 0)
```

```
            printf("找不到该名称的二叉树, 移除失败! \n");
        else
        {
            status j = Removetree(Lists, name);
            if (j == OK)
                printf("移除成功! \n");
            else
                printf("移除失败\n");
            namehash.erase(sname);
        }
    }
    else if (op3 == 3)
    {
        printf("请输入想要查询的二叉树树名: ");
        char name[200];
        scanf("%s", name);
        status j = Locatetree(Lists, name);
        if (j == 0)
            printf("二叉树树名不存在! \n");
        else
            printf("%s的逻辑序号是%d\n", name, j);
    }
    else if (op3 == 4)
    {
        TraversLists(Lists);
        printf("请输入想要单独操作的二叉树逻辑序号: ");
        int num;
        scanf("%d", &num);
        if (num < 0 || num > Lists.length)
            printf("序号不合法! \n");
        else
```

```
            {
                T = Lists.elem[num - 1].T;
                operator1(T);
                Lists.elem[num - 1].T = T;
            }
        }
        getchar();
        getchar();
        show3();
        scanf("%d", &op3);
        printf("\n");
    }
}

int main()
{
    int op1;
    show1();
    scanf("%d", &op1);
    T = NULL;
    while (op1)
    {
        switch (op1)
        {
        case 1:
            operator1(T);
            break;

        case 2:
            operator2(Lists);
            break;
```

```
        case 0:
            printf("%s", "感谢使用本系统 ");
            return 0;
        }
        show1();
        scanf("%d", &op1);
    }
}
```

7 附录 D 基于邻接表图实现的源程序

```
//def.h

#include <bits/stdc++.h>

using namespace std;

#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define overflow -2
#define ERROR2 -3
#define MAX_VERTEX_NUM 20
#define init_size 10

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
{
    char name[30]; // 图的名称
    ALGraph *G;    // 指向图的指针
} GraphEntry;

typedef struct
{
    GraphEntry *elem; // 动态数组，存储多个图
    int length;       // 当前存储的图个数
```

```
    int listsize;      // 当前分配的总容量
} GRAPHLISTS;

status check(VertexType V[], KeyType VR[][2]);
status CreateCraph(ALGraph &G, VertexType V[], KeyType VR[][2]);
status DestroyGraph(ALGraph &G);
int LocateVex(ALGraph G, KeyType u);
status PutVex(ALGraph &G, KeyType u, VertexType value);
int FirstAdjVex(ALGraph G, KeyType u);
int NextAdjVex(ALGraph G, KeyType v, KeyType w);
status check2(ALGraph &G, KeyType u);
status InsertVex(ALGraph &G, VertexType v);
status DeleteVex(ALGraph &G, KeyType v);
status InsertArc(ALGraph &G, KeyType v, KeyType w);
status DeleteArc(ALGraph &G, KeyType v, KeyType w);
void visit(VertexType v);
status DFSTraverse(ALGraph &G, void (*visit)(VertexType));
status BFSTraverse(ALGraph &G, void (*visit)(VertexType));
status SaveGraph(ALGraph G, char FileName[]);
status LoadGraph(ALGraph &G, char FileName[]);
vector<int> dijkstra(ALGraph &G, KeyType v);
vector<VertexType> VerticesSetLessThanK(ALGraph &G, KeyType v, int k);
int ShortestPathLength(ALGraph &G, KeyType v, KeyType w);
int ConnectedComponentsNums(ALGraph &G);
void PrintGraph(ALGraph &G);
void InitGraphLists(GRAPHLISTS &Lists);
status AddGraph(GRAPHLISTS &Lists, char *name);
status RemoveGraph(GRAPHLISTS &Lists, char *name);
int LocateGraph(GRAPHLISTS &Lists, char *name);
void show1();
void show2(ALGraph G);
```

```
void show3();
void Traverse(GRAPHLISTS Lists);

//function.cpp

#include "def.h"

status check(VertexType V[], KeyType VR[][2])
{
    int k = 0;
    int flag[100] = {};
    // 先检查V是否正确
    while (V[k].key != -1)
    {
        if (flag[V[k].key] != 0)
            return ERROR; // 关键字重复
        else
            flag[V[k].key] = 1;
        ++k;
    }

    // 检查V数组
    if (k == 0 || k > MAX_VERTEX_NUM)
        return 0;

    // 在检查VR数组是否正确 (不能存在环, 顶点必须存在)
    k = 0;
    while (VR[k][0] != -1)
    {
        if (VR[k][0] == VR[k][1])
            return ERROR;
```

```
        if (flag[VR[k][0]] == 0 || flag[VR[k][1]] == 0)
            return ERROR;
        ++k;
    }
    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。此题允许通过增加其它函数辅助实现本关任务*/
{
    if (check(V, VR) == ERROR)
        return ERROR;
    int k = 0;
    int flag[100] = {};

    // 先对V数组进行处理
    G.arcnum = 0, G.vexnum = 0;
    while (V[k].key != -1)
    {
        flag[V[k].key] = k;
        ++G.vexnum;
        G.vertices[k].data.key = V[k].key;
        strcpy(G.vertices[k].data.others, V[k].others);
        G.vertices[k].firstarc = NULL;
        k++;
    }

    // 对边进行处理
    k = 0;
    while (VR[k][0] != -1)
```

```
    {
        ++G.arcnum;
        ArcNode *p1 = (ArcNode *)malloc(sizeof(ArcNode));
        ArcNode *p2 = (ArcNode *)malloc(sizeof(ArcNode));
        p1->adjvex = flag[VR[k][1]];
        p2->adjvex = flag[VR[k][0]];
        // 头插法
        p1->nextarc = G.vertices[flag[VR[k][0]]].firstarc;
        p2->nextarc = G.vertices[flag[VR[k][1]]].firstarc;
        G.vertices[flag[VR[k][0]]].firstarc = p1;
        G.vertices[flag[VR[k][1]]].firstarc = p2;
        ++k;
    }
    G.kind = UDG;
    return OK;
}

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

```
    G.arcnum = 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;
{
    for (int i = 0; i < G.vexnum; ++i)
        if (G.vertices[i].data.key == u)
            return 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 index = -1;
    for (int i = 0; i < G.vexnum; ++i)
    {
        if (G.vertices[i].data.key == value.key)
            return ERROR2;
        if (G.vertices[i].data.key == u)
            index = i;
    }
    if (index == -1)
        return ERROR;
    G.vertices[index].data = value;
    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 = -1;
    for (i = 0; i < G.vexnum; ++i)
    {
        if (G.vertices[i].data.key == u)
            break;
    }
    if (i == G.vexnum)
        return -1;
    return G.vertices[i].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）下一个邻接顶点的
{
    int i, j;
    for (i = 0; i < G.vexnum; ++i)
        if (G.vertices[i].data.key == v)
            break;
    for (j = 0; j < G.vexnum; ++j)
        if (G.vertices[j].data.key == w)
            break;
    if (i == G.vexnum || j == G.vexnum)
        return -1; // v||w 不存在
    ArcNode *p = G.vertices[i].firstarc;
    while (p)
    {
        if (p->adjvex == j)
            break;
```

```
        p = p->nextarc;
    }
    return p->nextarc == NULL ? -1 : p->nextarc->adjvex;
}

status check2(ALGraph &G, KeyType u)
{
    for (int i = 0; i < G.vexnum; ++i)
        if (G.vertices[i].data.key == u)
            return ERROR;
    return OK;
}

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
{
    if (check2(G, v.key) == ERROR)
        return ERROR;
    if (G.vexnum == MAX_VERTEX_NUM)
        return ERROR2;
    G.vertices[G.vexnum].data = v;
    G.vertices[G.vexnum].firstarc = NULL;
    ++G.vexnum;
    return OK;
}

status DeleteVex(ALGraph &G, KeyType v)
// 在图G中删除关键字v对应的顶点以及相关的弧, 成功返回OK,否则返回ERROR
{
    if (G.vexnum == 1)
        return ERROR;
```

```
int pos = LocateVex(G, v);
if (pos == -1)
    return ERROR2;

// 删除与该顶点相连的边，同时删除邻接点中的反向边
ArcNode *p = G.vertices[pos].firstarc;
while (p)
{
    int adj = p->adjvex;

    // 删除 adj 顶点中的 pos 节点
    ArcNode **q = &G.vertices[adj].firstarc;
    while (*q)
    {
        if ((*q)->adjvex == pos)
        {
            ArcNode *toDelete = *q;
            *q = (*q)->nextarc; // 这里可以直接修改q的值，让它指向下一个边
            free(toDelete);
            break;
        }
        q = &(*q)->nextarc; // 这里修改q的赋值，但是不会改变G中边数组
    }

    ArcNode *temp = p;
    p = p->nextarc;
    free(temp);
    --G.arcnum;
}
G.vertices[pos].firstarc = NULL;
```

```
    // 将顶点数组中 pos 之后的元素前移，顶点信息和邻接表指针一起移动
    for (int i = pos; i < G.vexnum - 1; ++i)
        G.vertices[i] = G.vertices[i + 1];
    G.vertices[G.vexnum - 1].firstarc = NULL;
    --G.vexnum;

    // 更新所有边中 adjvex 的索引（由于顶点数组下标变化）
    for (int i = 0; i < G.vexnum; ++i)
    {
        for (ArcNode *p = G.vertices[i].firstarc; p; p = p->nextarc)
        {
            if (p->adjvex > pos)
                p->adjvex--;
        }
    }

    return OK;
}

status InsertArc(ALGraph &G, KeyType v, KeyType w)
// 在图G中增加弧<v,w>，成功返回OK,否则返回ERROR
{
    int posv = LocateVex(G, v), posw = LocateVex(G, w);
    if (posv == -1 || posw == -1)
        return ERROR;

    // 检查v-w是否已经存在
    ArcNode *p = G.vertices[posv].firstarc;
    while (p)
    {
        if (p->adjvex == posw)
```

```
            return ERROR2;
        p = p->nextarc;
    }

    // 加边
    ArcNode *p1 = (ArcNode *)malloc(sizeof(ArcNode)), *p2 = (ArcNode *)malloc(size
    p1->adjvex = posw, p2->adjvex = posv;
    p1->nextarc = G.vertices[posv].firstarc, p2->nextarc = G.vertices[posw].firsta
    G.vertices[posv].firstarc = p1, G.vertices[posw].firstarc = p2;
    ++G.arcnum;
    return OK;
}

status DeleteArc(ALGraph &G, KeyType v, KeyType w)
// 在图G中删除弧<v,w>, 若是无向图则同时删除<w,v>, 成功返回OK, 否则返回ERROR
{
    int posv = LocateVex(G, v), posw = LocateVex(G, w);
    if (posv == -1 || posw == -1)
        return ERROR;

    // 删除<v,w>
    int flag = 0;
    ArcNode **p = &G.vertices[posv].firstarc;
    while (*p)
    {
        if ((*p)->adjvex == posw)
        {
            ArcNode *temp = (*p);
            (*p) = (*p)->nextarc;
            free(temp);
            flag = 1;
```

```
            break;
        }
        p = &(*p)->nextarc;
    }
    if (!flag)
        return ERROR2; // 没有这条边

    // 如果是无向图，还要删除<w,v>
    if (G.kind == UDG || G.kind == UDN)
    {
        p = &G.vertices[posw].firstarc;
        while (*p)
        {
            if ((*p)->adjvex == posv)
            {
                ArcNode *temp = (*p);
                (*p) = (*p)->nextarc;
                free(temp);
                flag = 1;
                break;
            }
            p = &(*p)->nextarc;
        }
    }

    --G.arcnum;
    return OK;
}

void visit(VertexType v)
{
```

```
    printf(" %d %s", v.key, v.others);
}

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 vis[30] = {}, stk[100], top = -1;

    for (int i = 0; i < G.vexnum; ++i)
    {
        if (!vis[i])
            stk[++top] = i;

        while (top != -1)
        {
            int v = stk[top--];
            if (!vis[v])
            {
                visit(G.vertices[v].data);
                vis[v] = 1;

                ArcNode *p = G.vertices[v].firstarc;
                while (p)
                {
                    if (!vis[p->adjvex])
                    {
                        stk[++top] = p->adjvex;
                        break;
                    }
                    p = p->nextarc;
                }
```

```
            }
        }
    }
    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 que[1000], top = -1, down = -1, vis[30] = {};

    for (int i = 0; i < G.vexnum; ++i)
    {
        if (!vis[i])
            que[++top] = i;
        while (down != top)
        {
            int v = que[++down];
            if (!vis[v])
            {
                visit(G.vertices[v].data);
                vis[v] = 1;

                ArcNode *p = G.vertices[v].firstarc;
                while (p)
                {
                    if (!vis[p->adjvex])
                        que[++top] = p->adjvex;
                    p = p->nextarc;
                }
            }
```

```
        }
    }
    return OK;
}

status SaveGraph(ALGraph G, char FileName[])
// 将图的数据写入到文件FileName中
{
    FILE *fp = fopen(FileName, "w");
    if (!fp)
        return ERROR;

    // 保存图的基本信息
    fprintf(fp, "%d %d %d\n", G.vexnum, G.arcnum, G.kind);

    // 保存所有顶点
    for (int i = 0; i < G.vexnum; ++i)
        fprintf(fp, "%d %s\n", G.vertices[i].data.key, G.vertices[i].data.others);

    // 写边
    for (int i = 0; i < G.vexnum; ++i)
    {
        ArcNode *stack[100]; // 假设某个顶点最多100条边
        int top = 0;
        ArcNode *p = G.vertices[i].firstarc;

        // 用数组模拟栈存储边
        while (p)
        {
            stack[top++] = p;
            p = p->nextarc;
```

```
        }

        // 逆序写出
        while (top > 0)
        {
            ArcNode *q = stack[--top];
            fprintf(fp, "%d %d\n", i, q->adjvex);
        }
    }

    fclose(fp);
    return OK;
}

status LoadGraph(ALGraph &G, char FileName[])
// 读入文件FileName的图数据，创建图的邻接表
{
    FILE *fp = fopen(FileName, "r");
    if (!fp)
        return ERROR;

    fscanf(fp, "%d %d %d\n", &G.vexnum, &G.arcnum, &G.kind);

    // 读取顶点
    for (int i = 0; i < G.vexnum; ++i)
    {
        fscanf(fp, "%d %s\n", &G.vertices[i].data.key, &G.vertices[i].data.others)
        G.vertices[i].firstarc = NULL;
    }

    // 读入边
```

```
    int u, v;
    while (fscanf(fp, "%d %d\n", &u, &v) == 2)
    {
        ArcNode *p = (ArcNode *)malloc(sizeof(ArcNode));
        p->adjvex = v;
        p->nextarc = G.vertices[u].firstarc;
        G.vertices[u].firstarc = p;
    }

    fclose(fp);
    return OK;
}

vector<int> dijkstra(ALGraph &G, KeyType v)
{
    int start = LocateVex(G, v);
    vector<int> vis(G.vexnum, 0); // 记录这个点访问过没有
    vector<int> d(G.vexnum, 1e9); // 记录到v点的距离
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    pq.push({d[start] = 0, start});

    while (!pq.empty())
    {
        int w = pq.top().first, x = pq.top().second;
        pq.pop();
        if (vis[x])
            continue;
        vis[x] = true;

        ArcNode *p = G.vertices[x].firstarc;
        while (p)
```

```
        {
            int y = p->adjvex;
            if (d[x] + 1 < d[y])
            {
                pq.push({d[y] = d[x] + 1, y});
            }
            p = p->nextarc;
        }
    }

    return d;
}

vector<VertexType> VerticesSetLessThanK(ALGraph &G, KeyType v, int k)
// 操作结果是返回与顶点v距离小于k的顶点集合;
{
    vector<VertexType> ans;

    vector<int> d = dijkstra(G, v);
    for (int i = 0; i < d.size(); ++i)
        if (d[i] < k)
            ans.push_back(G.vertices[i].data);

    return ans;
}

int ShortestPathLength(ALGraph &G, KeyType v, KeyType w)
// 初始条件是图G存在; 操作结果是返回顶点v与顶点w的最短路径的长度
{
    int pos1 = LocateVex(G, v), pos2 = LocateVex(G, w);
    if (pos1 == -1 || pos2 == -1)
```

```
        return -1;
    vector<int> d = dijkstra(G, v);
    return d[pos2] == (int)1e9 ? -2 : d[pos2];
}

void DFS(ALGraph &G, int u, vector<int> &vis)
// 从顶点 u 开始深度优先搜索，标记访问的顶点
{
    vis[u] = 1;
    ArcNode *p = G.vertices[u].firstarc;
    while (p)
    {
        int v = p->adjvex;
        if (!vis[v])
            DFS(G, v, vis);
        p = p->nextarc;
    }
}

int ConnectedComponentsNums(ALGraph &G)
// 操作结果：返回图G的所有连通分量的个数（仅适用于无向图）
{
    if (G.kind != UDG && G.kind != UDN)
    {
        printf( "该函数仅适用于无向图。" );
        return -1;
    }

    vector<int> vis(G.vexnum, 0); // 访问标记
    int cnt = 0;
```

```
    for (int i = 0; i < G.vexnum; ++i)
    {
        if (!vis[i])
        {
            ++cnt;
            DFS(G, i, vis);
        }
    }

    return cnt;
}

void PrintGraph(ALGraph &G)
{
    printf("-----------------------------------------------\n");
    printf("顶点数: %d, 边数: %d\n", G.vexnum, G.arcnum);
    printf("图类型: ");
    switch (G.kind)
    {
    case DG:
        printf("有向图\n");
        break;
    case DN:
        printf("有向网\n");
        break;
    case UDG:
        printf("无向图\n");
        break;
    case UDN:
        printf("无向网\n");
        break;
```

```
    default:
        printf("未知类型\n");
        break;
    }

    for (int i = 0; i < G.vexnum; ++i)
    {
        printf("[%d]%d, \"%s\"", i, G.vertices[i].data.key, G.vertices[i].data.oth
        ArcNode *p = G.vertices[i].firstarc;
        while (p)
        {
            printf(" --> %d", p->adjvex);
            p = p->nextarc;
        }
        printf("\n");
    }
    printf("------------------------------------------------\n");
    printf("\n");
}

void InitGraphLists(GRAPHLISTS &Lists)
{
    Lists.elem = (GraphEntry *)malloc(init_size * sizeof(GraphEntry));
    Lists.length = 0;
    Lists.listsize = init_size;
}

// 添加图
status AddGraph(GRAPHLISTS &Lists, char *name)
{
    if (Lists.length == Lists.listsize)
```

```
    {
        // 扩容
        GraphEntry *newelem = (GraphEntry *)realloc(Lists.elem, (Lists.listsize +
        if (!newelem)
            return overflow;

        Lists.elem = newelem;
        Lists.listsize += init_size;
    }

    // 初始化空图指针
    Lists.elem[Lists.length].G = NULL;
    strcpy(Lists.elem[Lists.length].name, name);
    ++Lists.length;
    return OK;
}

// 删除图
status RemoveGraph(GRAPHLISTS &Lists, char *name)
{
    for (int i = 0; i < Lists.length; ++i)
    {
        if (strcmp(Lists.elem[i].name, name) == 0)
        {
            // 释放图结构内存（需要你自己实现 DestroyGraph）
            if (Lists.elem[i].G){
                DestroyGraph(*Lists.elem[i].G);
                free(Lists.elem[i].G);
            }
            // 元素前移
            for (int j = i; j < Lists.length - 1; ++j)
```

```
                Lists.elem[j] = Lists.elem[j + 1];

            --Lists.length;
            return OK;
        }
    }
    return ERROR; // 未找到
}

// 查找图，成功返回逻辑序号（从1开始），失败返回0
int LocateGraph(GRAPHLISTS &Lists, char *name)
{
    for (int i = 0; i < Lists.length; ++i)
        if (strcmp(Lists.elem[i].name, name) == 0)
            return i + 1;
    return 0;
}

void show1()
{
    printf("\n");
    printf("     Menu for Linear Table Type On Sequence Structure \n");
    printf("------------------------------------------------------------\n");
    printf("                    请选择你要操作的类型: \n");
    printf("           1. 单图              2.多图 \n");
    printf("           0. 退出 \n");
    printf("------------------------------------------------------------\n");
    printf("                    请选择你的操作[0~2]:");
}

void show2(ALGraph G)
```

```
{
    // system("cls");
    printf("\n");
    printf("      Menu for Linear Table On Sequence Structure         \n");
    printf("-------------------------------------------------------\n");
    printf("       1. CreateCraph         2. DestroyGraph         \n");
    printf("       3. LocateVex           4. PutVex           \n");
    printf("       5. FirstAdjVex         6. NextAdjVex                \n");
    printf("       7. InsertVex           8. DeleteVex            \n");
    printf("       9. InsertArc           10.DeleteArc            \n");
    printf("       11. DFSTraverse        12.BFSTraverse       \n");
    printf("       13. VerticesSetLessThanK                \n");
    printf("       14. ShortestPathLength                  \n");
    printf("       15. ConnectedComponentsNums                \n");
    printf("       16. SaveGraph          17.LoadGraph          \n");
    printf("       0.  Exit                20.change type \n");
    printf("ps:本系统在执行完一个功能后要再随便输出个字符才能再次显示菜单欧\n");
    printf("-------------------------------------------------------------\n");
    printf("图实时预览:\n");
    if (G.vexnum == 0)
        printf("图不存在 \n");
    else
        PrintGraph(G);
    printf("\n            请选择你的操作[0~20]:");
}

void show3()
{
    printf("\n");
    printf("         Menu for Linear Table On Sequence Structure        \n");
    printf("-------------------------------------------------------- \n");
```

```
    printf("       1. AddGraph          2. RemoveGraph                \n");
    printf("       3. LocateGraph       4. operator single Graph      \n");
    printf("       0. Exit              \n");
    printf("ps:本系统在执行完一个功能后要再随便输出个字符才能再次显示菜单欧\n");
    printf("-------------------------------------------------------------\n");
    printf("\n             请选择你的操作[0~4]:");
}

void Traverse(GRAPHLISTS Lists){
    for(int i=0;i<Lists.length;i++){
        printf("%d.%s\n",i+1,Lists.elem[i].name);
    }
}

//run.cpp

#include "def.h"

ALGraph G;
GRAPHLISTS Lists;
unordered_set<string> namehash; // 用于防止多图管理时出现两个相同的图树名

void operator1(ALGraph &G)
{
    int op2;
    show2(G);
    scanf("%d", &op2);
    printf("\n");

    while (op2 != 20)
    {
```

```
if (op2 == 0)
{
    printf("%s", "感谢使用本系统 ");
    exit(0);
}
else if (op2 == 1)
{
    if (G.vexnum != 0)
        printf("图已存在, 请勿重复创建\n");
    else
    {
        // 创建图
        printf("开始创建!\n");
        VertexType V[30];
        KeyType VR[100][2];
        printf("依次输入顶点的关键字以及值: \n");
        int i = 0, ii;
        do
        {
            scanf("%d%s", &V[i].key, V[i].others);
        } while (V[i++].key != -1);
        i = 0;
        printf("依次输入边: \n");
        do
        {
            scanf("%d%d", &VR[i][0], &VR[i][1]);
        } while (VR[i++][0] != -1);
        status j = CreateCraph(G, V, VR);
        if (j == ERROR)
            printf("关键字重复, 创建失败\n");
        else if (j == overflow)
```

```
                printf("内存溢出，创建失败 \n");
            else
                printf("创建成功！ \n");
        }
    }
    else if (op2 == 17)
    {
        if (G.vexnum != 0)
            printf("图已存在，请勿重复创建\n");
        else
        {
            printf("请输入文件名：");
            char filename[200];
            scanf("%s", filename);
            status j = LoadGraph(G, filename);
            if (j == ERROR)
                printf("文件打开失败！  \n");
            else if (j == OK)
                printf("读入完毕！  \n");
        }
    }
    else
    {
        // 现在对于所有操作来说图都应该存在，所以一次性判断图是否不存在
        if (G.vexnum == 0)
        {
            printf("不能对不存在的图进行操作！  \n");
        }
        else if (op2 == 2)
        {
```

```
        DestroyGraph(G);
        printf("销毁成功! \n");
    }
    else if (op2 == 3)
    {
        printf("想要获得哪个关键字对应的顶点位序: ");
        KeyType e;
        scanf("%d", &e);
        int j = LocateVex(G, e);
        if (j == -1)
            printf("找不到该顶点 \n");
        else
            printf("这个顶点的位序是: %d\n", j);
    }
    else if (op2 == 4)
    {
        printf("输入想要赋值的顶点关键字以及赋予的值: ");
        KeyType e;
        VertexType value;
        scanf("%d %d %s", &e, &value.key, &value.others);
        status j = PutVex(G, e, value);
        if (j == ERROR)
            printf("没有找到想要赋值的顶点 \n");
        else if (j == ERROR2)
            printf("赋值的关键字不唯一, 赋值失败  \n");
        else
            printf("赋值成功! \n");
    }
    else if (op2 == 5)
    {
        printf("想要获得哪个位序的顶点对应的第一邻接点: ");
```

```
        int e;
        scanf("%d", &e);

        int j = FirstAdjVex(G, G.vertices[e].data.key);
        if (j == -1)
            printf("不存在第一邻接点 \n");
        else
            printf("第一邻接点是: %d\n", j);
    }

    else if (op2 == 6)
    {
        printf("想要获得哪个位序的顶点对应位序顶点的下一邻接点: ");
        int v, w;
        scanf("%d %d", &v, &w);

        int j = NextAdjVex(G, G.vertices[v].data.key, G.vertices[w].data.k
        if (j == -1)
            printf("不存在下一邻接点 \n");
        else
            printf("下一邻接点是: %d\n", j);
    }
    else if (op2 == 7)
    {
        printf("输入插入顶点的值: ");
        VertexType e;
        scanf("%d %s", &e.key, &e.others);
        status j = InsertVex(G, e);
        if (j == ERROR2)
            printf("内存溢出, 插入失败 \n");
        else if (j == ERROR)
```

```
            printf("插入节点的关键字不唯一, 插入失败   \n");
        else
            printf("插入成功! \n");
    }
    else if (op2 == 8)
    {
        printf("输入想要删除的顶点的关键字: ");
        KeyType e;
        scanf("%d", &e);
        status j = DeleteVex(G, e);
        if (j == ERROR)
            printf("图只有一个顶点, 不能删除! \n");
        else if (j == ERROR2)
            printf("找不到要删除的顶点 \n");
        else
            printf("删除成功 ");
    }
    else if (op2 == 9)
    {
        printf("输入想要插入的弧 (输入2个顶点的关键字) : ");
        KeyType a, b;
        scanf("%d %d", &a, &b);
        status j = InsertArc(G, a, b);
        if (j == ERROR)
            printf("找不到对应关键字的顶点 \n");
        else if (j == ERROR2)
            printf("插入的边已存在, 插入失败  \n");
        else
            printf("插入成功! \n");
    }
    else if (op2 == 10)
```

```
{
    printf("输入想要删除的弧（输入2个顶点的关键字）：");
    KeyType a, b;
    scanf("%d %d", &a, &b);
    status j = DeleteArc(G, a, b);
    if (j == ERROR)
        printf("找不到对应关键字的顶点 \n");
    else if (j == ERROR2)
        printf("不存在要删除的边, \n");
    else
        printf("删除成功! \n");
}
else if (op2 == 11)
{
    printf("图深度优先搜索遍历结果如下:\n");
    DFSTraverse(G, visit);
}
else if (op2 == 12)
{
    printf("图广度优先搜索遍历结果如下:\n");
    BFSTraverse(G, visit);
}
else if (op2 == 13)
{
    printf("请输入起始顶点v的关键字和最大距离k:");
    KeyType e;
    int k;
    scanf("%d %d", &e, &k);
    if (LocateVex(G, e) == -1)
        printf("这个起始顶点不存在! \n");
    else
```

```
        {
            vector<VertexType> ans = VerticesSetLessThanK(G, e, k);
            printf("顶点v距离小于k的顶点集合如下: ");
            for (VertexType i : ans)
                printf("%d %s ", i.key, i.others);
            printf("\n");
        }
    }
    else if (op2 == 14)
    {
        printf("请输入起始顶点v和终止顶点的关键字:");
        KeyType a, b;
        int k;
        scanf("%d %d", &a, &b);
        status j = ShortestPathLength(G, a, b);
        if (j == -1)
            printf("顶点不存在! \n");
        else if (j == -2)
            printf("路径不存在! \n");
        else
            printf("顶点%d与顶点%d的最短路径的长度是%d\n", a, b, j);
    }
    else if (op2 == 15)
    {
        printf("图G的所有连通分量的个数是%d\n", ConnectedComponentsNums(G)
    }
    else if (op2 == 16)
    {
        printf("请输入文件名: ");
        char filename[200];
        scanf("%s", filename);
```

```
                status j = SaveGraph(G, filename);
                if (j == ERROR)
                    printf("文件打开失败!  \n");
                else if (j == OK)
                    printf("写入完毕!  \n");
            }
        }
        getchar(); // 读取掉输入去残留的换行符
        getchar(); // 使结果在屏幕上保持一段时间
        show2(G);
        scanf("%d", &op2);
        printf("\n");
    }
    return;
}

void operator2(GRAPHLISTS &Lists)
{
    InitGraphLists(Lists);
    show3();
    int op3;
    scanf("%d", &op3);
    printf("\n");

    while (1)
    {
        if (op3 == 0)
        {
            printf("%s", "感谢使用本系统 ");
            exit(0);
        }
```

```
else if (op3 == 1)
{
    printf("请输入新建图名: ");
    char name[200];
    scanf("%s", name);
    string sname = name;
    if (namehash.count(sname) != 0)
        printf("名称已存在, 创建失败! \n");
    else
    {
        status j = AddGraph(Lists, name);
        if (j == OK)
            printf("创建成功! \n");
        else
            printf("内存溢出, 创建失败! \n");
        namehash.insert(sname);
    }
}
else if (op3 == 2)
{
    printf("请输入想要移除的图名: ");
    char name[200];
    scanf("%s", name);
    string sname = name;
    if (namehash.count(sname) == 0)
        printf("找不到该名称的图, 移除失败! \n");
    else
    {
        status j = RemoveGraph(Lists, name);
        if (j == OK)
            printf("移除成功! \n");
```

```
            else
                printf("移除失败\n");
            namehash.erase(sname);
        }
    }
    else if (op3 == 3)
    {
        printf("请输入想要查询的图名: ");
        char name[200];
        scanf("%s", name);
        status j = LocateGraph(Lists, name);
        if (j == 0)
            printf("图名不存在! \n");
        else
            printf("%s的逻辑序号是%d\n", name, j);
    }
    else if (op3 == 4)
    {
        Traverse(Lists);
        printf("请输入想要单独操作的图逻辑序号: ");
        int num;
        scanf("%d", &num);
        if (num < 0 || num > Lists.length)
            printf("序号不合法! \n");
        else
{
    // 如果图不存在则创建
    if (Lists.elem[num - 1].G == NULL)
    {
        Lists.elem[num - 1].G = (ALGraph*)malloc(sizeof(ALGraph));
        Lists.elem[num - 1].G->vexnum = Lists.elem[num - 1].G->arcnum = 0;
```

```
        }

        // 操作图
        operator1(*Lists.elem[num - 1].G);
    }
        }
        getchar();
        getchar();
        show3();
        scanf("%d", &op3);
        printf("\n");
    }
}

int main()
{
    int op1;
    show1();
    scanf("%d", &op1);
    G.vexnum = G.arcnum = 0;
    while (op1)
    {
        switch (op1)
        {
        case 1:
            operator1(G);
            break;

        case 2:
            operator2(Lists);
            break;
```

```
        case 0:
            printf("%s", "感谢使用本系统 ");
            return 0;
        }
        show1();
        scanf("%d", &op1);
    }
}
```