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 U202414739

姓　　名 赵冠杰

指导教师 郝义学

报告日期 2025 年 5 月 31 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

在数据结构的学习过程中，线性表作为最基本的数据组织形式，其核心操作和算法实现是各类高级数据结构与算法的基础。通常，线性表可以采用顺序存储或链式存储两种方式实现，其中链式存储结构由于其动态内存管理和灵活的插入删除优势，在实际应用中十分常见。

本实验旨在利用单链表来实现线性表的各项基本运算，包括初始化、销毁、插入、删除、查找、遍历等操作，并进一步扩展到链表翻转、删除倒数第 n 个结点、链表排序、文件保存加载以及多链表管理等附加功能。通过实验，希望能够加深对线性表概念和链式存储结构实际操作的理解，培养问题分析和模块化设计的能力，同时为以后的数据结构学习打下坚实基础。

1.1 问题描述

依据最小完备性和常用性相结合的原则，以函数形式定义了线性表的初始化表、销毁表、清空表、判定空表、求表长和获得元素等 12 种基本运算[1, 2]，具体基本运算功能定义如下：

1) 初始化表：函数名称是 InitList(L)；初始条件是线性表 L 不存在；操作结果是构造一个空的线性表；
2) 销毁表：函数名称是 DestroyList(L)；初始条件是线性表 L 已存在；操作结果是销毁线性表 L；
3) 清空表：函数名称是 ClearList(L)；初始条件是线性表 L 已存在；操作结果是将 L 重置为空表；
4) 判定空表：函数名称是 ListEmpty(L)；初始条件是线性表 L 已存在；操作结果是若 L 为空表则返回 TRUE, 否则返回 FALSE；
5) 求表长：函数名称是 ListLength(L)；初始条件是线性表已存在；操作结果是返回 L 中数据元素的个数；
6) 获得元素：函数名称是 GetElem(L,i,e)；初始条件是已知线性表存在，且 i 满足 1≤i≤ListLength(L)；操作结果是用 e 返回 L 中第 i 个数据元素的值；
7) 查找元素：函数名称是 LocateElem(L,e,compare())；初始条件是线性表已存在；操作结果是返回 L 中第 1 个与 e 满足关系 compare() 关系的数据元素的

位序，若这样的数据元素不存在，则返回值为 0；

8) 获得前驱：函数名称是 PriorElem(L,cur_e,pre_e)；初始条件是线性表 L 已存在；操作结果是若 cur_e 是 L 的数据元素，且不是第一个，则用 pre_e 返回它的前驱，否则操作失败，pre_e 无定义；
9) 获得后继：函数名称是 NextElem(L,cur_e,next_e)；初始条件是线性表 L 已存在；操作结果是若 cur_e 是 L 的数据元素，且不是最后一个，则用 next_e 返回它的后继，否则操作失败，next_e 无定义；
10) 插入元素：函数名称是 ListInsert(L,i,e)；初始条件是线性表 L 已存在，i 满足 1≤i≤ListLength(L)+1；操作结果是在 L 的第 i 个位置之前插入新的数据元素 e；
11) 删除元素：函数名称是 ListDelete(L,i,e)；初始条件是线性表 L 已存在且非空，1≤i≤ListLength(L)；操作结果：删除 L 的第 i 个数据元素，用 e 返回其值；
12) 遍历表：函数名称是 ListTraverse(L,visit())，初始条件是线性表 L 已存在；操作结果是依次对 L 的每个数据元素调用函数 visit( )。

附加功能定义如下:

1) 链表翻转：函数名称是 reverseList(L)，初始条件是线性表 L 已存在；操作结果是将 L 翻转；
2) 删除链表的倒数第 n 个结点：函数名称是 RemoveNthFromEnd(L,n); 初始条件是线性表 L 已存在且非空, 操作结果是该链表中倒数第 n 个节点；
3) 链表排序：函数名称是 sortList(L)，初始条件是线性表 L 已存在；操作结果是将 L 由小到大排序；
4) 实现线性表的文件形式保存：需要设计文件数据记录格式，以高效保存线性表数据逻辑结构 (D,R) 的完整信息；同时，还需要设计线性表文件保存和加载操作合理模式，使线性表保存到文件后，可以由一个空表加载文件生成单链表；
5) 实现多个线性表管理：设计相应的数据结构管理多个线性表的查找、添加、移除等功能。

实验要求构造一个具有菜单的功能演示系统来实现上述全部功能，其中，在主程序中完成函数调用所需实参值的准备和函数执行结果的显示，并给出适当的操作提示显示[3] 。

1.2 系统设计

为满足实验要求，本系统采用分文件编写的方式，分为 function.h、function.c 和 main.c 三个文件进行编写。

三个文件中内容的规划是系统设计的最重要部分。function.h 文件中的主要内容为文件包含、宏定义、变量声明和函数声明。其中变量声明包括单链表结点结构体的定义和管理多链表系统的结构体的定义，函数声明则包括所有 function.c 中的函数。function.c 文件中的主要内容为所有函数的具体实现，包括实现问题描述中的功能的函数以及菜单函数，其中，菜单函数的设计要格外注重界面直观性和易用性。main.c 文件中为主函数，在主函数中要完成函数调用所需实参值的准备和函数执行结果的显示，并给出适当的操作提示显示。

在数据结构设计方面，本系统以单链表作为线性表的物理存储结构。单链表采用带头结点的设计，即每个链表包含一个专门的头结点，保证了插入和删除操作的统一性和简洁性。链表结点的定义包含两个主要字段：存储数据的元素和指向下一个结点的指针。为满足实验中实现多个线性表管理的要求，系统额外设计了一个多链表管理结构 MultiList，其主要内容为：一个链表指针数组，用于保存系统中所有的单链表；一个整型变量记录当前已创建的链表总数，以及另一个变量标记当前活动链表的索引[4] 。这种设计不仅方便了对单个链表操作的实现，同时也使得系统能够灵活管理和调用多个线性表，满足复杂功能需求。

1.3 系统实现

functions.c 文件中主要函数的实现思想如下：

1) 初始化表 InitList(L)：首先检查传入的链表指针是否为空，如果链表已经被初始化则返回错误；否则分配一个头结点，并根据用户选择决定是否在初始化过程中进一步输入数据，构成一个完整的链表。
2) 销毁表 DestroyList(L)：先遍历链表，使用临时指针保存每一个节点后调用 free 函数来释放内存。最后通过将指向头结点的指针 *L 置为 NULL，将链表彻底销毁。
3) 清空表 ClearList(L)：遍历链表，使用临时指针保存每一个节点后调用 free 函数来释放内存。最后令链表为初始化的状态，将头结点的 next 指针设置为 NULL。

4) 获得前驱 PriorElem(L,e,pre)：使用两个指针 prev 和 curr 同时遍历，当 curr 指向的结点数据等于目标元素 e 时，prev 即为前驱。
5) 获得后继 NextElem(L,e,next)：利用 currnode 和 nextnode 两个指针同时向后移动，一个指向当前节点，另一个指向其后继节点，在遍历过程中，若 currnode 指向的节点数据等于目标元素 e，nextnode 即为后继。
6) 插入元素 ListInsert(L,i,e)：从头结点开始顺序查找，定位到第 i−1 个节点，然后分配内存创建一个新节点，将传入的元素 e 赋值给新节点的数据域，并将新节点的 next 指针指向原本的第 i 个节点，最后通过修改第 i−1 个节点的 next 指针将新节点链接到链表中。
7) 删除元素 ListDelete(L,i,e)；循环使指针停留在待删除节点的前一个节点处，修改该节点的 next 指针，将待删除节点跳过，然后将待删除节点中存储的数据赋值给 e，最后释放该节点的内存并返回 OK。
8) 链表翻转 reverseList(L)：利用三个指针依次将链表节点反转，初始时 prev 置为 NULL，curr 指向第一个有效节点；在循环中，先保存当前节点的后继 next，再将当前节点的 next 指向 prev，然后更新 prev 与 curr，直至所有节点均完成反转；最后将头节点的 next 指向 prev，从而实现整个链表的翻转。
9) 删除链表的倒数第 n 个结点 RemoveNthFromEnd(L,n)：创建一个临时节点 dummy，其 next 指向链表的首个有效节点。采用快慢指针的思想，将 fast 和 slow 两个指针均指向 dummy，然后通过一个 for 循环使 fast 指针先行走 n+1 步，确保 fast 与 slow 之间始终保持 n 个节点的间隔，随后同时移动 fast 和 slow，直到 fast 走到链表末尾，此时 slow->next 就是倒数第 n 个结点，跳过目标节点并释放其内存，最后将 dummy.next 赋值回 L->next，恢复链表的结构。
10) 从文件加载线性表 LoadList(L,filename)：以只读方式打开指定文件，从文件中读取一个整数 count，用以确定文件中存储的数据个数，然后为链表分配一个头结点，并以一个 tail 指针作为链表当前的尾部；接下来进入循环，循环次数取决于 count 的大小，读取文件中的下一个元素并建立结点，设置新建结点的 data 域和 next 域，循环结束即完成单链表的创建。
11) 删除管理器中某个单链表 RemoveList(ml,listNames,targetName)：先在名称数组 ListNames 中查找目标链表名称对应的索引，然后根据索引销毁单链表，再将后面的链表指针和名称整体前移，最后清空最后一个名称。

在多链表管理的实现上采用了模块化设计，将多个单链表统一管理于一个MultiList结构体中，该结构体包含链表指针数组、名称数组、链表计数器和当前活动链表索引，从而方便对多个线性表的整体管理。在main函数中先初始化管理器，将各链表指针均置空，计数器归零；创建新链表时，调用基本链表的初始化函数，并将新链表的指针及用户输入的名称记录在数组中，同时更新当前活动链表序号；在删除操作中，系统通过遍历名称数组确定目标链表的索引，调用销毁函数释放对应链表的内存，再将后续链表及名称前移保证数据一致性；同时，程序还提供了显示所有链表状态的功能，使用户能够直观地查看各链表信息；在多链表管理器中，通过切换当前活动链表这一关键操作，用户可以自由选择操作其中某个链表，实现问题描述中的全部功能。

1.4 系统测试

为了验证系统各项功能是否达到设计预期，本部分对系统进行了全面的测试，由于我完成了附加功能中的多链表管理任务，所以测试部分全部针对多链表管理系统展开，不再单独对单个链表操作进行测试。

如图1-1中所示，系统能够正确地在多链表管理子系统中创建新的单链表，并且我们可以给单链表命名，这里不妨命名为“湖北”；在创建时还可以选择是否输入数据，若不输入数据，则仅将链表初始化。

若创建时不输入数据，后续我们可以用向指定位置插入元素这一功能逐个输入数据，如图1-2所示，依次插入6、3、9、7、12、1、7、14、4、2，遍历结果如图1-3所示，可见是符合要求的。

链表创建完成后我们对菜单中的功能依次展开测试，图1-4为获取链表长度，当前链表共10个元素，故长度为10，输出正确。

图1-5为获取指定位置的元素，输入5，链表中第5个元素是12，故输出12，输出正确。

图1-6为获取指定元素的位置，输入14，元素14在链表中的位置是8，故输出8，输出正确。

图1-7和1-8为获取指定元素的前驱，其中图1-7输入的9的前驱元素是3，图1-8输入的是链表中的第1个元素，其前驱不存在，故输出“操作失败!”。

图1-9为删除指定位置元素，输入4，删除第4个元素7，此时再遍历链表输出应为6 3 9 12 1 7 14 4 2，输出正确。

```
 14.附加功能2: 删除当前链表的倒数第 n 个结点
 15.附加功能3: 对当前链表从小到大排序
 16.附加功能4: 保存当前链表到文件
 17.附加功能5: 从文件加载当前链表
 18.附加功能6: 多线性表管理
**************** ********************
请输入你的选择:
18

--- 多链表管理子系统 ---
  1.创建新的单链表
  2.删除某个单链表
  3.显示所有单链表状态
  4.操作其中某个单链表
  0.返回主菜单
请输入你的选择: 1
请输入新链表的名字: 湖北
是否在初始化时输入数据? (1-是, 0-否): 0
创建新链表成功, 链表名字为 "湖北".

--- 多链表管理子系统 ---
  1.创建新的单链表
  2.删除某个单链表
  3.显示所有单链表状态
  4.操作其中某个单链表
  0.返回主菜单
请输入你的选择: 4
请输入要操作的链表名字: 湖北

-----操作单链表 "湖北" -----
```

图 1-1　创建单链表

```
请输入您的选择:
9
请输入插入位置: 1
请输入要插入的元素: 6
插入成功!
```

图 1-2　插入元素

附加功能的测试如下：图1-10为翻转链表，原链表为 6 3 9 12 1 7 14 4 2，翻转后应为 2 4 14 7 1 12 9 3 6，输出正确。

图1-11为删除倒数第 n 个结点，输入 5，倒数第 5 个结点为 1，删除 1 后遍历链表得到 2 4 14 7 12 9 3 6，输出正确。

图1-12为从小到大排序，排序后结果为 2 3 4 6 7 9 12 14，输出正确。

图1-13为将链表“湖北”保存到文件中的内容，文件命名为“测试”，在多表管理器中添加新的单链表“湖南”，图1-14为加载文件“测试”到链表“湖南”，从加载结果看，链表的文件存取正确。

图1-15为删除掉链表“湖北”后显示的多链表管理系统中所有链表的状态。由于“湖北”已经被删除，故管理器中只有链表“湖南”。

```
请输入您的选择:
11
链表 "湖北" 的元素: 6 3 9 7 12 1 7 14 4 2
```

图 1-3　遍历链表

```
请输入您的选择：
4
链表 "湖北" 的长度为： 10
```

图 1-4　获取链表长度

```
请输入您的选择：
5
请输入要获取的元素位置： 5
链表 "湖北" 的第 5 个元素为： 12
```

图 1-5　获取指定位置的元素

```
请输入您的选择：
6
请输入要查找的元素： 14
元素 14 在链表 "湖北" 中的位置为： 8
```

图 1-6　获取指定元素的位置

```
请输入您的选择：
7
请输入要查询前驱的元素： 9
元素 9 的前驱为： 3
```

图 1-7　获取指定元素的前驱

```
请输入您的选择：
7
请输入要查询前驱的元素： 6
操作失败！
```

图 1-8　获取首元素的前驱

```
请输入您的选择：
11
链表 "湖北" 的元素： 6 3 9 12 1 7 14 4 2
```

图 1-9　删除指定位置元素

```
请输入您的选择：
11
链表 "湖北" 的元素： 2 4 14 7 1 12 9 3 6
```

图 1-10　翻转链表

```
请输入您的选择：
11
链表 "湖北" 的元素： 2 4 14 7 12 9 3 6
```

图 1-11　删除倒数第 n 个结点

```
请输入您的选择：
11
链表 "湖北" 的元素：2 3 4 6 7 9 12 14
```

图 1-12　从小到大排序

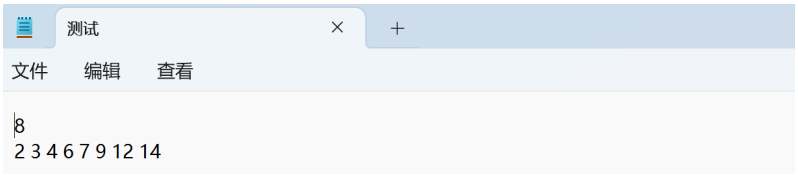

图 1-13　文件保存

```
请输入您的选择：
11
链表 "湖南" 的元素：2 3 4 6 7 9 12 14
```

图 1-14　文件加载

```
--- 多链表管理子系统 ---
  1.创建新的单链表
  2.删除某个单链表
  3.显示所有单链表状态
  4.操作其中某个单链表
  0.返回主菜单
请输入你的选择：2
请输入要删除的链表的名字：湖北
链表 "湖北" 删除成功！

所有链表状态：
链表名字："湖南"，长度：8，元素：2 3 4 6 7 9 12 14
```

图 1-15　显示链表状态

综上所述，本系统实现了问题描述中的全部功能，是一个具有完备菜单功能的单链表操作演示系统。不过系统在操作提示上还有一定不足，比如操作失败时没有给出具体的失败原因，这是后续仍需改进之处。

1.5 实验小结

本次实验围绕链式存储结构的线性表实现展开，我成功完成了初始化、插入、删除、遍历等单链表基本运算及翻转、删除倒数第 n 个结点、排序、文件操作、多链表管理等附加功能。在编写程序时，我通过分文件模块化设计，构建了清晰的系统架构，提升了代码可维护性。在存储结构上，我选择带头结点单链表的简化了操作逻辑，附加功能中的细节实现，也深化了我对链式存储特性的理解。

实验中，指针错误与内存管理是核心挑战，通过逐行调试和逻辑梳理，不仅强化了我动态内存分配的能力，也提升了我的调试能力，同时，在设计菜单和操作提示的过程中我更加深刻地理解了优化交互界面到意义。

本次实验不仅巩固了我的线性表链式存储的理论，更让我在代码实现、问题分析和系统设计层面积累了实践经验。通过解决实际问题，培养了我的数据结构与算法综合应用能力，为后续树、图等复杂结构的学习奠定了坚实基础。通过单链表操作的实验，我深刻体会到“理论 → 设计 → 实现 → 调试”的完整开发流程能大幅提高我的编程能力。

2 基于二叉链表的二叉树实现

本实验通过对二叉树各种基本运算和附加功能的设计与实现，全面探讨了二叉链表作为二叉树存储结构的优越性与适应性。从二叉树的创建、销毁、清空，到遍历、结点查找与赋值，每个函数模块都遵循最小完备性和常用性相结合的原则，既保证了功能的完整实现，又兼顾了工程实践中的高效调用。整个系统的设计过程，充分体现了模块化思想，每一项基本操作都相对独立，这样即简介有便于调试和后期维护。

除了常规的二叉树操作，本实验还补充了求最大路径和、最近公共祖先以及翻转二叉树等附加功能，进一步加深了对二叉树这种数据结构的理解。特别是对二叉树文件的保存和加载、多个二叉树的管理等附加要求的研究，使得实验从单一结构操作延伸到对多个数据的系统管理的综合设计。这样的设计思路不仅提高了程序的实用性，同时也为后续复杂应用的开发打下坚实的基础。

2.1 问题描述

依据最小完备性和常用性相结合的原则，以函数形式定义了二叉树的创建二叉树、销毁二叉树、清空二叉树、判定空二叉树和求二叉树深度等 14 种基本运算[1, 2] 。具体基本运算功能定义和说明如下：

1) 创建二叉树：函数名称是 CreateBiTree(T,definition)；初始条件是 definition 给出二叉树 T 的定义，如带空子树的二叉树前序遍历序列、或前序 + 中序、或后序 + 中序；操作结果是按 definition 构造二叉树 T，需要注意的是，要求 T 中各结点关键字具有唯一性，并且 CreateBiTree 函数必须根据已有的 definition 生成 T，而不应该在 CreateBiTree 函数中输入二叉树的定义；
2) 销毁二叉树：函数名称是 DestroyBiTree(T)；初始条件是二叉树 T 已存在；操作结果是销毁二叉树 T；
3) 清空二叉树：函数名称是 ClearBiTree (T)；初始条件是二叉树 T 存在；操作结果是将二叉树 T 清空；
4) 判定空二叉树：函数名称是 BiTreeEmpty(T)；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 TRUE，否则返回 FALSE；
5) 求二叉树深度：函数名称是 BiTreeDepth(T)；初始条件是二叉树 T 存在；操

作结果是返回 T 的深度；

6) 查找结点：函数名称是 LocateNode(T,e)；初始条件是二叉树 T 已存在，e 是和 T 中结点关键字类型相同的给定值；操作结果是返回查找到的结点指针，如无关键字为 e 的结点，返回 NULL；
7) 结点赋值：函数名称是 Assign(T,e,value)；初始条件是二叉树 T 已存在，e 是和 T 中结点关键字类型相同的给定值；操作结果是关键字为 e 的结点赋值为 value；
8) 获得兄弟结点：函数名称是 GetSibling(T,e)；初始条件是二叉树 T 存在，e 是和 T 中结点关键字类型相同的给定值；操作结果是返回关键字为 e 的结点的（左或右）兄弟结点指针。若关键字为 e 的结点无兄弟，则返回 NULL；
9) 插入结点：函数名称是 InsertNode(T,e,LR,c)；初始条件是二叉树 T 存在，e 是和 T 中结点关键字类型相同的给定值，LR 为 0 或 1，c 是待插入结点；操作结果是根据 LR 为 0 或者 1，插入结点 c 到 T 中，作为关键字为 e 的结点的左或右孩子结点，结点 e 的原有左子树或右子树则为结点 c 的右子树，特别地，当 c 作为根结点插入时，LR=-1，原根结点作为 c 的右子树；
10) 删除结点：函数名称是 DeleteNode(T,e)；初始条件是二叉树 T 存在，e 是和 T 中结点关键字类型相同的给定值。操作结果是删除 T 中关键字为 e 的结点；同时，如果关键字为 e 的结点度为 0，删除即可；如关键字为 e 的结点度为 1，用关键字为 e 的结点孩子代替被删除的 e 位置；如关键字为 e 的结点度为 2，用 e 的左孩子代替被删除的 e 位置，e 的右子树作为 e 的左子树中最右结点的右子树；
11) 前序遍历：函数名称是 PreOrderTraverse(T,Visit())；初始条件是二叉树 T 存在，Visit 是一个函数指针的形参（可使用该函数对结点操作）；操作结果：先序遍历，对每个结点调用函数 Visit 一次且一次，一旦调用失败，则操作失败。
12) 中序遍历：函数名称是 InOrderTraverse(T,Visit))；初始条件是二叉树 T 存在，Visit 是一个函数指针的形参（可使用该函数对结点操作）；操作结果是中序遍历 t，对每个结点调用函数 Visit 一次且一次，一旦调用失败，则操作失败；
13) 后序遍历：函数名称是 PostOrderTraverse(T,Visit))；初始条件是二叉树 T 存在，Visit 是一个函数指针的形参（可使用该函数对结点操作）；操作结果是后序遍历 t，对每个结点调用函数 Visit 一次且一次，一旦调用失败，则操作

失败。

14) 按层遍历：函数名称是 LevelOrderTraverse(T,Visit))；初始条件是二叉树 T 存在，Visit 是对结点操作的应用函数；操作结果是层序遍历 t，对每个结点调用函数 Visit 一次且一次，一旦调用失败，则操作失败。

附加功能定义和说明如下：

1) 求最大路径和：函数名称是 MaxPathSum(T)，初始条件是二叉树 T 存在；操作结果是返回根节点到叶子结点的最大路径和；
2) 求最近公共祖先：函数名称是 LowestCommonAncestor(T,e1,e2)；初始条件是二叉树 T 存在；操作结果是该二叉树中 e1 节点和 e2 节点的最近公共祖先；
3) 翻转二叉树：函数名称是 InvertTree(T)，初始条件是线性表 L 已存在；操作结果是将 T 翻转，使其所有节点的左右节点互换；
4) 实现线性表的文件形式保存：其中，需要设计文件数据记录格式，以高效保存二叉树数据逻辑结构 (D,R) 的完整信息，同时还要设计二叉树文件保存和加载操作合理模式，使二叉树保存到文件后，可以由一个空二叉树加载文件生成二叉树。
5) 实现多个二叉树管理：可采用线性表的方式管理多个二叉树，线性表中的每个数据元素为一个二叉树的基本属性，至少应包含有二叉树的名称。实现多个二叉树管理应能创建、添加、移除多个二叉树，并且可自由切换到管理的每个二叉树，单独对某二叉树进行单二叉树的所有操作。

实验要求构造一个具有菜单的功能演示系统来实现上述全部功能，其中，在主程序中完成函数调用所需实参值的准备和函数执行结果的显示，并给出适当的操作提示显示[3] 。

2.2 系统设计

本部分采用和基于链式存储结构的线性表实现中相同的分文件编写方式，故不再赘述。

系统整体采用模块化设计思想，将整个实验报告的功能分解为多个相对独立且互相协作的模块。系统主要包含用户交互模块、功能处理模块。用户交互模块通过菜单驱动的界面向用户展示各项操作，并负责接收和解析用户输入；功能处理模块则实现二叉树的创建、销毁、遍历、查找、插入、删除等基本功能以及

求最大路径和、寻找最近公共祖先、二叉树翻转等附加功能，模块内函数各司其职、相互调用。

系统中二叉树的数据结构采用二叉链表的形式进行实现，每个节点定义了数据域、左子树指针和右子树指针，并在必要时包含节点的唯一关键字信息，从而支持高效的查找、插入和删除操作。此外，为了便于管理多棵二叉树，系统还设计了一个线性表结构，每个线性表中的元素记录了单个二叉树的基本属性，从而可以在一个系统中创建、添加、删除和切换不同的二叉树[4]。这样的数据结构设计充分考虑了内存空间的利用率和操作的灵活性，满足了实验对基本功能以及附加功能的全面要求。

2.3 系统实现

本实验中的主要函数实现思想如下：

1) 创建二叉树 CreateBiTree(T,definition)：函数采用递归方法构造二叉树，通过遍历数组中各元素确定节点顺序。首先判断当前关键字是否为结束标志负一或为空子树标志零以实现递归终止；若关键字有效则申请动态内存创建新节点，并将当前数据赋于此节点，随后依次递归构造左子树和右子树直至数组遍历完毕，从而构建出完整的二叉树结构。
2) 销毁二叉树 DestroyBiTree(T)：函数采用递归的方式利用后序遍历销毁二叉树。首先判断传入指针是否为空以检验其有效性，如果当前结点存在则递归销毁左子树与右子树，再释放该结点内存并将其置空避免悬挂指针，确保二叉树整体销毁成功。
3) 求二叉树深度 BiTreeDepth(T)：函数采用递归方法求解二叉树深度。首先判断当前节点是否为空，若为空则返回深度零；若不为空则分别递归计算左子树和右子树深度，并取其中较大者加一作为当前树深度。
4) 查找结点 LocateNode(T,e)：函数采用递归方法在二叉树中查找与给定关键字匹配的结点。首先判断当前结点是否为空，若为空则返回空指针；若结点数据与给定关键字相等，则直接返回该结点；若不匹配，则先递归搜索左子树，若左子树中找到则返回，否则继续在右子树中查找。
5) 获得兄弟结点 GetSibling(T,e)：函数采用递归策略在二叉树中查找与给定关键字匹配结点的兄弟结点。首先检测当前节点的左右子树，若左子树存在且其结点关键字与给定值相同则返回右子树，反之亦然；若二者均不满足，

则对左右子树递归查找，直至找到满足条件的兄弟结点或遍历完整棵树，最终未找到时返回空指针。

6) 插入结点 InsertNode(T,e,LR,c)：函数首先判断树指针是否合法以及待插入结点关键字是否重复，避免出现重复关键字。当插入标志为负一时，新结点置为根结点，原根作为新结点的右子树；若插入标志为零或一，则先查找关键字对应的结点，再根据左右标志将新结点插入相应位置，并将原有子树挂接为新结点的右子树。
7) 删除结点 DeleteNode(T,e)：本函数通过辅助函数查找待删除结点及其父结点，根据该结点度数采用不同策略进行删除。若为叶结点则直接释放；若仅有一个子，则用该子结点替代删除结点；若左右子均存在，则在左子树中找到最右节点，将右子树挂接到其右侧，再以左子树取代原结点位置，最终释放目标结点，确保整体结构完整且内存得到正确回收。
8) 先序遍历 PreOrderTraverse(T,Visit())：采用递归方式实现，对非空二叉树首先访问根结点，再依次递归遍历左子树和右子树。
9) 中序遍历 InOrderTraverse(T,Visit()))：采用递归方式实现，首先递归访问左子树，再访问当前结点，最后依次递归遍历右子树。
10) 后序遍历 PostOrderTraverse(T,Visit()))：采用递归方式实现，先递归遍历左子树，再遍历右子树，最终访问根结点。
11) 层序遍历 PostOrderTraverse(T,Visit()))：层序遍历基于队列实现，首先将根结点入队，随后在队列非空时依次出队访问当前结点，并将其左右子结点依次入队。
12) 求最大路径和 MaxPathSum(T)：函数利用递归方法实现。首先判断二叉树是否为空，若为空则返回错误；若为叶结点则直接返回其关键字值。若仅存在一边子树，则必经该子树递归求解路径和；若左右子树均存在，则分别递归求解左右分支的最大路径和，并选取较大者加上当前结点的关键字值作为最终结果。
13) 最近公共祖先 LowestCommonAncestor(T,e1,e2)：函数采用递归分治思想。程序首先判断二叉树是否为空以及两个目标节点是否均在树中存在，当当前结点等于任一目标节点时直接返回该结点；否则分别在左右子树中递归搜索，若左右子树均返回非空则说明目标节点分处两侧，此时当前结点即为最近公共祖先；反之则返回非空子树的结果作为局部公共祖先，最终通过

递归回溯得到整体解。

14) 翻转二叉树 InvertTree(T)：函数采用递归方法实现二叉树的翻转。首先判断当前结点是否为空，若为空则递归终止，否则交换当前结点左右子树的位置，再递归调用自身对左右子树分别进行翻转。
15) 文件加载创建二叉树 LoadBiTree(T,FileName)：函数首先以只读方式打开指定文件，然后依照预设格式连续读入每个结点的关键字和对应字符串信息，将数据按先序遍历序列存入定义数组，其中正数表示有效结点、零代表空子树、-1 标记输入结束。读入完成后关闭文件，再调用辅助递归函数根据数组内容逐层构建二叉树。

在多二叉树管理的实现上，我用顺序表实现多个二叉树的集中管理，每个管理项包含唯一名称和对应二叉树指针。初始化时将数量置零；添加时先检验名称重复和数组容量，再将新的二叉树及其名称存入管理器；移除时需查找目标名称，调用销毁函数释放树的内存，然后依次前移后续项以保持连续性；同时模块还提供按名称查找和列表输出功能，以便快速访问和管理各二叉树。

2.4 系统测试

为了验证系统各项功能是否达到设计预期，本部分对系统进行了全面的测试，由于我完成了附加功能中的多二叉树管理任务，所以测试部分全部针对多二叉树管理系统展开，不再单独对单个二叉树操作进行测试。

如图2-1中所示，系统能够正确地在多二叉树管理子系统中创建新的二叉树并命名为“安徽”，在创建时我们有带空子树标记的先序遍历序列、前序和中序遍历序列、后序和中序遍历序列、仅初始化这四种创建选项，用户可自由选择，而系统将会根据用户的选择引导用户输入 definition 数组。在测试中，我们选择带空子树标记的先序遍历序列的创建方式。图2-2为测试二叉树的实际结构。

```
请输入你的选择：1
请选择创建二叉树的方式：
 1．带空子树标记的先序遍历序列
 2．前序和中序遍历序列
 3．后序和中序遍历序列
 4．仅初始化
请输入您的选择（1/2/3/4）：1
请输入带空子树标记的先序遍历序列的数据（格式：key others），
输入key为0表示空结点，输入key为-1表示序列结束。
请输入结点 1 的数据：1
b
请输入结点 2 的数据：16 u
请输入结点 3 的数据：19 g
请输入结点 4 的数据：0 null
请输入结点 5 的数据：0 null
请输入结点 6 的数据：0 null
请输入结点 7 的数据：
6 c
请输入结点 8 的数据：0 null
请输入结点 9 的数据：20 o
请输入结点 10 的数据：0 null
请输入结点 11 的数据：0 null
请输入结点 12 的数据：-1 null
二叉树创建成功。
请输入新二叉树的名称：安徽
二叉树添加到管理器成功。
```

图 2-1　创建二叉树

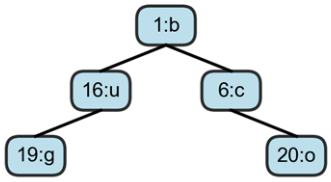

图 2-2　二叉树实际结构 1

在创建好二叉树后，我们还可以插入结点，如图2-3所示，我们插入关键字为 10 的结点作为根节点，插入结点后前序遍历结果如图2-4所示，符合如图2-5所示的二叉树的实际结构。

```
请输入您的选择：
8
请输入要插入位置的父结点关键字 e：0
请输入要插入结点的关键字key（整数）：10
请输入要插入结点的其他信息others（字符串）：x
请输入插入方向（0表示左子树；1表示右子树；-1表示根节点）：-1
插入成功。
```

图 2-3　插入结点

```
前序遍历结果:
 10,x 1,b 16,u 19,g 6,c 20,o
```

图 2-4　前序遍历结果

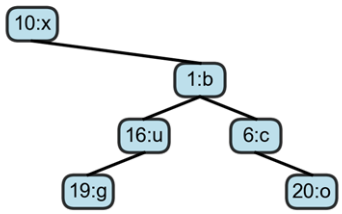

图 2-5　二叉树实际结构 2

图2-6为获取二叉树深度，二叉树深度为 4，输出正确；图2-7为查找关键字为 19 的结点，图2-8为查找关键字为 9 的结点，由于不存在关键字为 9 的结点，故输出 “未找到该结点!”；图2-9为修改结点的值，我们选择关键字为 10 的顶点，修改其关键字为 13，其他信息未 f，这时我们得到的树的实际结构如图2-10所示。

```
请输入您的选择:
4
二叉树深度为: 4
```

图 2-6　二叉树深度

```
请输入您的选择:
5
请输入查找结点的关键字: 19
找到关键字为 19 的结点
```

图 2-7　查找关键字为 19 的结点

```
请输入您的选择:
5
请输入查找结点的关键字: 9
未找到该结点!
```

图 2-8　查找关键字为 9 的结点

```
请输入您的选择:
6
请输入结点关键字 e: 10
请输入新关键字key（整数）: 13
请输入其他信息others（字符串）: f
赋值成功。
```

图 2-9　修改结点

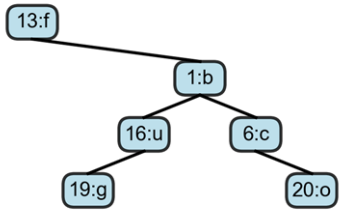

图 2-10　二叉树实际结构 3

图2-11为查找关键字为 6 的结点的兄弟结点，根据图2-10可知，其兄弟结点关键字为 16，图2-11中的输出结果符合预期。图2-12为查找关键字为 13 的兄弟结点，由于关键字为 13 的结点是根节点，无兄弟结点，故输出“查找兄弟结点出错!”。

```
请输入您的选择:
7
请输入查找兄弟结点的关键字 e: 6
兄弟结点关键字为: 16
```

图 2-11　查找关键字为 6 的兄弟结点

```
请输入您的选择:
7
请输入查找兄弟结点的关键字 e: 13
查找兄弟结点出错!
```

图 2-12　查找关键字为 13 的兄弟结点

删除关键字为 16 的结点，图2-132-142-15依次为删除结点后二叉树的中序、后序、层序遍历结果，都符合图2-16中的二叉树的实际结构。

```
中序遍历结果:
 13,f 19,g 1,b 6,c 20,o
```

图 2-13　中序遍历结果

```
后序遍历结果:
 19,g 20,o 6,c 1,b 13,f
```

图 2-14　后序遍历结果

```
层序遍历结果:
 13,f 1,b 19,g 6,c 20,o
```

图 2-15　层序遍历结果

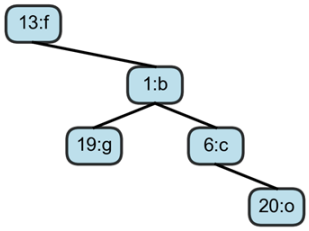

图 2-16　二叉树实际结构 4

将二叉树保存至名字为“测试”的文件中，结果如图2-17所示，在多链表管理系统中再创建一个名为“陕西”的二叉树，加载文件“测试”，中序遍历得到图2-18，结果正确。在多链表管理系统中删除二叉树“安徽”，再显示所有管理的二叉树，结果如图2-19所示，可知系统中只有二叉树“陕西”。

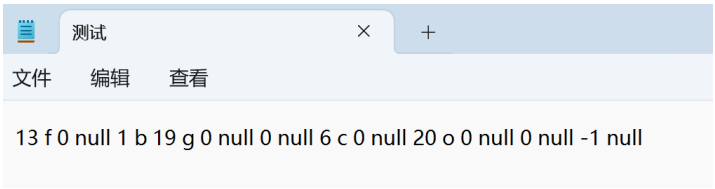

图 2-17　二叉树文件保存

图2-20为求二叉树中最大路径和，图2-21为求关键字为 20 和 6 的都最近公共祖先，图2-22为翻转二叉树，输出皆正确。

```
请输入您的选择：
15
请输入两个结点的关键字：20 6
最近公共祖先为：6
```

图 2-21　最近公共祖先

```
请输入您的选择：
16
二叉树已翻转。
```

图 2-22　翻转二叉树

```
请输入您的选择:
18
请输入加载文件名: 测试
加载成功。

-----操作二叉树 "陕西" -----
  0.返回子菜单
  1.销毁二叉树
  2.清空二叉树
  3.二叉树判空
  4.获取二叉树深度
  5.查找关键字为e的结点
  6.给关键字为e的结点赋值
  7.获得关键字为e的结点的兄弟结点
  8.插入结点
  9.删除结点
 10.前序遍历
 11.中序遍历
 12.后序遍历
 13.层序遍历
 14.附加功能1: 求最大路径和
 15.附加功能2: 求最近公共祖先
 16.附加功能3: 翻转二叉树
 17.附加功能4: 实现线性表的文件形式保存
 18.附加功能5: 从文件加载创建二叉树
请输入您的选择:
10
前序遍历结果:
 13,f 1,b 19,g 6,c 20,o
```

图 2-18　二叉树文件加载

```
请输入你的选择: 3
请输入要删除的二叉树名称: 安徽
删除成功!

--- 多二叉树管理子系统 ---
  1.创建新的二叉树
  2.显示所有二叉树状态
  3.删除某个二叉树
  4.操作其中某个二叉树
  0.返回主菜单
请输入你的选择: 2
当前管理的二叉树列表:
序号: 1, 名称: 陕西
```

图 2-19　多二叉树管理状态

```
请输入您的选择:
14
最大路径和为: 40
```

图 2-20　最大路径和

综上所述，本系统实现了问题描述中的全部功能，是一个具有完备菜单功能的二叉树操作演示系统。但同时，系统在直观性上做的并不完美，比如只能通过先序遍历和中序遍历或者后续遍历和中序遍历来确定一棵树，而非直接打印树的实际结构，这些会在一定程度上影响用户的体验感，这是需要改进的地方。

2.5 实验小结

本次实验采用二叉链表实现二叉树的各项基本运算与附加功能。在实验过程中，我实现了二叉树的创建、销毁、清空、遍历、查找、插入与删除等基本操作，同时还设计并实现了求最大路径和、寻找最近公共祖先和翻转二叉树等附加功能。为了满足文件保存与加载以及多二叉树管理的要求，我将系统的各个功能模块均采用模块化设计思路实现，使得每个功能既独立又相互协作，保证了系统整体的健壮性与灵活性。

实验过程中我不仅巩固了二叉链表数据结构和递归算法的基本原理，还进一步体会到模块化设计在程序编写中的重要意义。通过调试与测试，我发现部分边界条件处理仍存在不足，算法效率在极端情况中有待进一步优化。同时，在实验中我也发现我对递归的理解还不够深刻，无论是树，还是后面的图，都离不开递归的思想，因此我仍需更进一步理解递归。

在实验过程中，我深入研究了递归算法和树形结构的实现机制，对二叉树的内存管理和边界条件有了更全面的把握。实验成果不仅加深了我对二叉树数据结构及其链表实现方式的理解，更为今后开发更复杂的数据处理系统奠定了坚实基础。或许本系统的功能是微不足道的，但在开发系统中积累的经验一定会对我未来的工程实践有重大意义。

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

在这一部分实验中，我采用数组作为基本数据结构，实现了线性表的创建、插入、删除、查找等核心运算。通过对数组边界、内存分配以及数据连续性的管理，使我深刻认识到顺序存储在直接访问效率上的优势，同时也暴露出在数据量动态变化时扩容和内存浪费的问题。实验过程中，我格外关注了各种操作的时间复杂度，总体而言，顺序存储结构较为直观，便于我这样的初学者理解数据布局。

3.2 基于链式存储结构的线性表实现

这部分实验采用链表实现线性表，充分发挥了动态内存分配和节点链接的优势。在实现过程中，我重点关注了节点的插入、删除以及遍历操作，通过指针的灵活操作避免了数组扩容带来的局限性，同时有效管理了内存回收问题。实验让我体会到链式存储结构在处理不确定数据量时的优越性，但同时也发现其在随机访问上相较于顺序存储结构的劣势。

3.3 基于二叉链表的二叉树实现

本实验利用二叉链表形式实现了二叉树的各项基本运算，包括树的构建、遍历、查找、插入、删除以及辅助功能如最大路径和、最近公共祖先和树的翻转。通过递归和模块化设计方法，每个功能均在独立函数中实现，保证了整体结构的清晰和维护的便利。实践过程中，我不仅熟练掌握了树的递归遍历技术，也认识到在进行边界条件判断和内存释放方面需要更加谨慎。不过我认为，此部分可以增加一些内容，例如线索二叉树、二叉树与树的转换等难度更高的算法，以及哈夫曼树等二叉树的应用。

3.4 基于邻接表的图实现

在图的实现部分，我选用了邻接表作为数据存储方式，通过构造顶点和边的链表对来实现图的基本功能，如图的创建、遍历以及最短路径搜索等。该方法有

效解决了矩阵表示方法在处理稀疏图时存在的空间浪费问题，同时也提升了图运算的灵活性和效率。在实验过程中，我深入了解了图理论的基本思想和实际应用，积累了丰富的编程调试经验。由于这部分函数针对的全部是无向图，图这章中很多重要的算法都没有涉及，例如最小生成树、最短路径、拓扑排序等，希望以后可以增加有向图、无向网等部分的内容。

参考文献

[1] 严蔚敏, 吴伟民. 数据结构题集（C 语言版）[M]. [S.l.]: 清华大学出版社, 2007.

[2] 殷立峰. Qt C++ 跨平台图形界面程序设计基础 [M]. [S.l.]: 清华大学出版社, 2014: 192 – 197.

[3] NYHOFF L. ADTs, Data Structures, and Problem Solving with C++[M]. Second Edition. [S.l.]: Calvin College, 2005.

[4] 严蔚敏, 吴伟民. 数据结构（C 语言版）[M]. [S.l.]: 清华大学出版社, 2007.

附录 A 基于顺序存储结构线性表实现的源程序

```
#define _CRT_SECURE_NO_WARNINGS

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

#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10

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

/* 顺序表（顺序结构）的定义 */
typedef struct {
    ElemType* elem;
    int length;
    int listsize;
} SqList;

/* 多线性表管理的定义 */
typedef struct {
    struct {
        char name[30];
        SqList L;
    } elem[10];
    int length;
```

```
} LISTS;

LISTS Lists;  // 多线性表全局管理变量

SqList L;
int choice;          // 主菜单选择
int pre, next;       // 存储前驱、后继
ElemType e, k;       // 用于存储数据

/* 函数 1: 创建顺序表 */
status InitList(SqList* L) {
    if (L->elem == NULL) {  // 不存在则创建
        L->elem = (ElemType*)malloc(sizeof(ElemType) * LIST_INIT_SIZE);
        if (!L->elem)
            return ERROR;
        L->length = 0;
        L->listsize = LIST_INIT_SIZE;
        printf(" 成功创建, 输入 1 写入数据, 输入 0 跳过数据录入: \n");
        int numIn;
        scanf("%d", &numIn);
        if (numIn == 1) {
            int count;
            printf(" 请输入写入元素的个数 (不超过%d): \n", LIST_INIT_SIZE);
            scanf("%d", &count);
            if (count > LIST_INIT_SIZE) {
                ElemType* newbase = (ElemType*)realloc(L->elem, count *
                ↪  sizeof(ElemType));
                if (!newbase) {
                    printf(" 重新分配空间失败\n");
                    return ERROR;
                }
                L->elem = newbase;
                L->listsize = count;
            }
            L->length = count;
```

```
            printf(" 请写入%d 个数据: \n", count);
            for (int i = 0; i < count; i++) {
                scanf("%d", &L->elem[i]);
            }
        }
        return OK;
    }
    return INFEASIBLE;
}

/* 函数 2: 销毁顺序表 */
status DestroyList(SqList* L) {
    if (L->elem == NULL)
        return INFEASIBLE;
    free(L->elem);
    L->elem = NULL;
    L->length = 0;
    L->listsize = 0;
    return OK;
}

/* 函数 3: 清空顺序表 */
status ClearList(SqList* L) {
    if (L->elem == NULL)
        return INFEASIBLE;
    for (int i = 0; i < L->length; i++)
        L->elem[i] = 0;
    L->length = 0;
    return OK;
}

/* 函数 4: 判断线性表是否为空 */
status ListEmpty(SqList L) {
    if (L.elem == NULL)
        return INFEASIBLE;
```

```
    return (L.length == 0) ? TRUE : FALSE;
}

/* 函数 5: 返回线性表长度 */
status ListLength(SqList L) {
    if (L.elem == NULL)
        return INFEASIBLE;
    return L.length;
}

/* 函数 6: 获取第 i 个元素 */
status GetElem(SqList L, int i, ElemType* e) {
    if (L.elem == NULL)
        return INFEASIBLE;
    if (i >= 1 && i <= L.length) {
        *e = L.elem[i - 1];
        return OK;
    }
    else {
        return ERROR;
    }
}

/* 函数 7: 查找指定元素 (返回 1-based 位置) */
int LocateElem(SqList L, ElemType e) {
    if (L.elem == NULL)
        return -1;
    for (int i = 0; i < L.length; i++) {
        if (L.elem[i] == e)
            return i + 1;
    }
    return 0;
}

/* 函数 8: 获取指定元素的前驱 */
```

```
status PriorElem(SqList L, ElemType e, int* pre) {
    if (L.elem == NULL)
        return INFEASIBLE;
    for (int i = 0; i < L.length; i++) {
        if (L.elem[i] == e) {
            if (i == 0)
                return ERROR;
            *pre = L.elem[i - 1];
            return OK;
        }
    }
    return ERROR;
}

/* 函数 9: 获取指定元素的后继 */
status NextElem(SqList L, ElemType e, int* next) {
    if (L.elem == NULL)
        return INFEASIBLE;
    for (int i = 0; i < L.length; i++) {
        if (L.elem[i] == e) {
            if (i == L.length - 1)
                return ERROR;
            *next = L.elem[i + 1];
            return OK;
        }
    }
    return ERROR;
}

/* 函数 10: 在指定位置插入元素 */
status ListInsert(SqList* L, int i, ElemType e) {
    if (L->elem == NULL)
        return INFEASIBLE;
    if (i < 1 || i > L->length + 1)
        return ERROR;
```

```
    if (L->length >= L->listsize) {
        ElemType* newbase = (ElemType*)realloc(L->elem, (L->listsize +
        ↪  LISTINCREMENT) * sizeof(ElemType));
        if (newbase == NULL)
            return ERROR;
        L->elem = newbase;
        L->listsize += LISTINCREMENT;
    }
    for (int j = L->length; j >= i; j--) {
        L->elem[j] = L->elem[j - 1];
    }
    L->elem[i - 1] = e;
    L->length++;
    return OK;
}

/* 函数 11: 删除指定位置元素 */
status ListDelete(SqList* L, int i, ElemType* e) {
    if (L->elem == NULL)
        return INFEASIBLE;
    if (i < 1 || i > L->length)
        return ERROR;
    *e = L->elem[i - 1];
    for (int j = i - 1; j < L->length - 1; j++)
        L->elem[j] = L->elem[j + 1];
    L->length--;
    return OK;
}

/* 函数 12: 遍历顺序表 */
status ListTraverse(SqList L) {
    if (L.elem == NULL)
        return INFEASIBLE;
    for (int i = 0; i < L.length; i++) {
        printf("%d", L.elem[i]);
```

```
        if (i != L.length - 1)
            printf(" ");
    }
    return OK;
}

/* 函数 13: 求最大连续子数组和 */
status MaxSubArray(SqList L) {
    int res = 0, now = 0;
    for (int i = 0; i < L.length; i++) {
        now = 0;
        for (int j = i; j < L.length; j++) {
            now += L.elem[j];
            if (now > res)
                res = now;
        }
    }
    return res;
}

/* 函数 14: 求和为 k 的子数组个数 (暴力法) */
status SubArrayNum(SqList L, ElemType k) {
    int num = 0;
    for (int i = 0; i < L.length; i++) {
        int now = 0;
        for (int j = i; j < L.length; j++) {
            now += L.elem[j];
            if (now == k)
                num++;
        }
    }
    return num;
}

/* 函数 15: 从小到大排序 (冒泡排序) */
```

```
void SortList(SqList* L) {
    for (int i = 0; i < L->length; i++) {
        for (int j = 0; j < L->length - 1 - i; j++) {
            if (L->elem[j] > L->elem[j + 1]) {
                int temp = L->elem[j];
                L->elem[j] = L->elem[j + 1];
                L->elem[j + 1] = temp;
            }
        }
    }
}

/* 函数 16: 以二进制方式保存线性表到文件 */
status SaveInFile(SqList L, char* filename) {
    FILE* fp = fopen(filename, "wb");
    if (fp == NULL) {
        perror(" 文件打开失败");
        return ERROR;
    }
    fwrite(&L.length, sizeof(int), 1, fp);
    for (int i = 0; i < L.length; i++)
        fwrite(&L.elem[i], sizeof(int), 1, fp);
    fclose(fp);
    return OK;
}

/* 使用文本方式保存线性表到文件 */
status saveListToFile(SqList L, char* filename) {
    FILE* fp = fopen(filename, "w");
    if (fp == NULL)
        return INFEASIBLE;
    for (int i = 0; i < L.length; i++)
        fprintf(fp, "%d ", L.elem[i]);
    fclose(fp);
    return OK;
```

```
}

/* 函数：从文件加载线性表（以文本方式读取） */
status loadListFromFile(SqList* L, char* filename) {
    FILE* fp = fopen(filename, "r");
    if (fp == NULL)
        return INFEASIBLE;
    ClearList(L);
    ElemType data;
    while (fscanf(fp, "%d", &data) != EOF) {
        ListInsert(L, ListLength(*L) + 1, data);
    }
    ListTraverse(*L);
    fclose(fp);
    return OK;
}

/* 多线性表管理函数 */
/* 增加一个线性表：将一个名称为 ListName 的空顺序表加入 Lists */
status AddList(LISTS* Lists, char ListName[]) {
    if (Lists->length >= 10)
        return ERROR;  // 超出容量限制
    strcpy(Lists->elem[Lists->length].name, ListName);
    Lists->elem[Lists->length].L.elem = (ElemType*)malloc(LIST_INIT_SIZE *
    ↪  sizeof(ElemType));
    if (Lists->elem[Lists->length].L.elem == NULL)
        return ERROR;
    Lists->elem[Lists->length].L.length = 0;
    Lists->elem[Lists->length].L.listsize = LIST_INIT_SIZE;
    Lists->length++;
    return OK;
}

/* 删除一个线性表 */
status RemoveList(LISTS* Lists, char ListName[]) {
```

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

/* 查找线性表，成功返回 1-based 位置，失败返回 0 */
int LocateList(LISTS Lists, char ListName[]) {
    for (int i = 0; i < Lists.length; i++) {
        if (strcmp(Lists.elem[i].name, ListName) == 0)
            return i + 1;
    }
    return 0;
}

/* 主函数 */
int main() {
    /* 初始化全局变量 */
    L.elem = NULL;
    L.length = 0;
    L.listsize = 0;
    Lists.length = 0;

    while (1) {
        printf("\n-----顺序表演示系统-----\n");
        printf("0. 退出演示系统\n");
        printf("1. 创建单个顺序表\n");
        printf("2. 销毁单个顺序表\n");
```

```
printf("3. 清空单个顺序表\n");
printf("4. 判断单个顺序表是否为空\n");
printf("5. 输出单个顺序表长度\n");
printf("6. 获取单个顺序表指定位置元素\n");
printf("7. 查找元素在单个顺序表中的位置\n");
printf("8. 获取单个顺序表指定元素的前驱\n");
printf("9. 获取单个顺序表指定元素的后继\n");
printf("10. 在单个顺序表中指定位置插入元素\n");
printf("11. 删除单个顺序表指定位置元素\n");
printf("12. 遍历单个顺序表\n");
printf("13. 附加功能 1: 求最大连续子数组和\n");
printf("14. 附加功能 2: 求和为 k 的子数组个数\n");
printf("15. 附加功能 3: 从小到大排序\n");
printf("16. 附加功能 4: 保存单个顺序表到文件\n");
printf("17. 附加功能 5: 多线性表管理\n");
printf("18. 附加功能 6: 从文件读取单个顺序表\n");
printf(" 请输入您的选择: ");
scanf("%d", &choice);

int res, pos, tempPos, element;
char filename[30];

switch (choice) {
case 0:
    printf(" 您已结束使用\n");
    return 0;
case 1:
    if (InitList(&L) == OK)
        printf(" 创建顺序表成功, 数据已录入\n");
    else
        printf(" 顺序表创建失败或已存在\n");
    break;
case 2:
    if (DestroyList(&L) == OK)
        printf(" 销毁顺序表成功\n");
```

```
        else
            printf(" 顺序表不存在, 销毁失败\n");
        break;
    case 3:
        if (ClearList(&L) == OK)
            printf(" 清空顺序表成功\n");
        else
            printf(" 顺序表不存在, 无法清空\n");
        break;
    case 4: {
        int emptyResult = ListEmpty(L);
        if (emptyResult == INFEASIBLE)
            printf(" 顺序表不存在\n");
        else if (emptyResult == TRUE)
            printf(" 顺序表为空\n");
        else
            printf(" 顺序表不为空\n");
        break;
    }
    case 5:
    {
        int length = ListLength(L);
        if (length == -1) printf(" 顺序表不存在! \n");
        else
            printf(" 顺序表长度为: %d\n", length);
        break;
    }
    case 6:
        printf(" 请输入要获取元素的位置: \n");
        scanf("%d", &tempPos);
        switch (GetElem(L, tempPos, &element)) {
        case OK:
            printf(" 第%d 个元素为: %d\n", tempPos, element);
            break;
        case ERROR:
```

```
            printf(" 位置不合法\n");
            break;
        case INFEASIBLE:
            printf(" 顺序表不存在\n");
            break;
        }
        break;
    case 7:
        printf(" 请输入要查找的元素: \n");
        scanf("%d", &element);
        pos = LocateElem(L, element);
        if (pos == -1)
            printf(" 顺序表不存在\n");
        else if (pos == 0)
            printf(" 找不到该元素\n");
        else
            printf(" 该元素位于第%d 个位置\n", pos);
        break;
    case 8:
        printf(" 请输入查找前驱的目标元素: \n");
        scanf("%d", &element);
        switch (PriorElem(L, element, &pre)) {
        case INFEASIBLE:
            printf(" 顺序表不存在\n");
            break;
        case ERROR:
            printf(" 该元素没有前驱\n");
            break;
        case OK:
            printf(" 该元素的前驱为: %d\n", pre);
            break;
        }
        break;
    case 9:
        printf(" 请输入查找后继的目标元素: \n");
```

```
        scanf("%d", &element);
        switch (NextElem(L, element, &next)) {
        case INFEASIBLE:
            printf(" 顺序表不存在\n");
            break;
        case ERROR:
            printf(" 该元素没有后继\n");
            break;
        case OK:
            printf(" 该元素的后继为: %d\n", next);
            break;
        }
        break;
    case 10:
        printf(" 请输入插入的位置和元素: \n");
        scanf("%d %d", &tempPos, &element);
        switch (ListInsert(&L, tempPos, element)) {
        case INFEASIBLE:
            printf(" 顺序表不存在\n");
            break;
        case ERROR:
            printf(" 插入位置不合法\n");
            break;
        case OK:
            printf(" 插入成功\n");
            break;
        }
        break;
    case 11:
        printf(" 请输入删除元素的位置: \n");
        scanf("%d", &tempPos);
        switch (ListDelete(&L, tempPos, &element)) {
        case INFEASIBLE:
            printf(" 顺序表不存在\n");
            break;
```

```
        case ERROR:
            printf(" 删除位置错误或无元素\n");
            break;
        case OK:
            printf(" 删除成功, 被删元素为: %d\n", element);
            break;
        }
        break;
    case 12:
        ListTraverse(L);
        printf("\n");
        break;
    case 13:
        printf(" 最大连续子数组的和为: %d\n", MaxSubArray(L));
        break;
    case 14:
        printf(" 请输入目标值 k: \n");
        scanf("%d", &k);
        printf(" 满足条件的子数组个数为: %d\n", SubArrayNum(L, k));
        break;
    case 15:
        SortList(&L);
        printf(" 排序完成\n");
        break;
    case 16:
        printf(" 请输入保存文件名: \n");
        scanf("%s", filename);
        res = SaveInFile(L, filename);
        if (res == OK)
            printf(" 保存成功\n");
        else
            printf(" 保存失败\n");
        break;
    case 17: {
        /* 多线性表管理子菜单 */
```

```
int multiChoice;
printf(" 请选择操作: \n");
printf("1. 增加一个新线性表\n");
printf("2. 删除一个线性表\n");
printf("3. 查找线性表\n");
printf("4. 操作其中某个线性表\n");
scanf("%d", &multiChoice);
char listName[30];
printf(" 请输入线性表名: \n");
scanf("%s", listName);
if (multiChoice == 1) {
    if (AddList(&Lists, listName) == OK)
        printf(" 增加成功! \n");
    else
        printf(" 增加失败, 可能已满或内存申请失败\n");
}
else if (multiChoice == 2) {
    if (RemoveList(&Lists, listName) == OK)
        printf(" 删除成功! \n");
    else
        printf(" 删除失败, 未找到该线性表\n");
}
else if (multiChoice == 3) {
    int posFound = LocateList(Lists, listName);
    if (posFound == 0)
        printf(" 线性表不存在\n");
    else
        printf(" 线性表存在, 位置为: %d\n", posFound);
}
else if (multiChoice == 4) {
    int posFound = LocateList(Lists, listName);
    if (posFound == 0) {
        printf(" 线性表不存在\n");
        break;
    }
```

```
int listIndex = posFound - 1;
printf(" 找到线性表 \"%s\" ，进入操作子菜单...\n",
↪  listName);
int op = 0;
do {
    printf("\n-----操作线性表 \"%s\" -----\n", listName);
    printf("0. 返回主菜单\n");
    printf("1. 创建线性表\n");
    printf("2. 销毁线性表\n");
    printf("3. 清空线性表\n");
    printf("4. 线性表判空\n");
    printf("5. 线性表长度\n");
    printf("6. 获取指定位置元素\n");
    printf("7. 查找指定元素的位置\n");
    printf("8. 获取指定元素的前驱\n");
    printf("9. 获取指定元素的后继\n");
    printf("10. 插入元素至指定位置\n");
    printf("11. 删除指定位置元素\n");
    printf("12. 遍历线性表\n");
    printf("13. 附加功能 1: 求最大连续子数组和\n");
    printf("14. 附加功能 2: 求和为 k 的子数组个数\n");
    printf("15. 附加功能 3: 从小到大排序\n");
    printf("16. 附加功能 4: 保存线性表到文件\n");
    printf(" 请输入您的选择: \n");
    scanf("%d", &op);
    int posInput, opElem;
    switch (op) {
    case 0:
        printf(" 退出当前操作子菜单\n");
        break;
    case 1:
        if (InitList(&Lists.elem[listIndex].L) == OK)
            printf(" 创建成功，数据已录入\n");
        else
            printf(" 顺序表已存在或创建失败\n");
```

```
        break;
    case 2:
        if (DestroyList(&Lists.elem[listIndex].L) == OK)
            printf(" 销毁成功\n");
        else
            printf(" 顺序表不存在, 销毁失败\n");
        break;
    case 3:
        if (ClearList(&Lists.elem[listIndex].L) == OK)
            printf(" 清空成功\n");
        else
            printf(" 清空失败, 顺序表不存在\n");
        break;
    case 4: {
        int emptyRes = ListEmpty(Lists.elem[listIndex].L);
        if (emptyRes == INFEASIBLE)
            printf(" 顺序表不存在\n");
        else if (emptyRes == TRUE)
            printf(" 顺序表为空\n");
        else
            printf(" 顺序表不为空\n");
    }
          break;
    case 5:
    {
        int length = ListLength(Lists.elem[listIndex].L);
        if (length == -1) printf(" 顺序表不存在! \n");
        else
            printf(" 顺序表长度为: %d\n", length);
        break;
    }
    case 6:
        printf(" 请输入要获取元素的位置: \n");
        scanf("%d", &posInput);
```

```
            switch (GetElem(Lists.elem[listIndex].L, posInput,
            ↪  &opElem)) {
            case OK:
                printf(" 第%d 个元素为: %d\n", posInput,
                ↪  opElem);
                break;
            case ERROR:
                printf(" 位置不合法\n");
                break;
            case INFEASIBLE:
                printf(" 顺序表不存在\n");
                break;
            }
            break;
        case 7:
            printf(" 请输入要查找的元素: \n");
            scanf("%d", &opElem);
            pos = LocateElem(Lists.elem[listIndex].L, opElem);
            if (pos == -1)
                printf(" 顺序表不存在\n");
            else if (pos == 0)
                printf(" 找不到该元素\n");
            else
                printf(" 该元素位于第%d 个位置\n", pos);
            break;
        case 8:
            printf(" 请输入查找前驱的目标元素: \n");
            scanf("%d", &opElem);
            switch (PriorElem(Lists.elem[listIndex].L, opElem,
            ↪  &pre)) {
            case INFEASIBLE:
                printf(" 顺序表不存在\n");
                break;
            case ERROR:
                printf(" 该元素没有前驱\n");
```

```
            break;
        case OK:
            printf(" 该元素的前驱为: %d\n", pre);
            break;
        }
        break;
    case 9:
        printf(" 请输入查找后继的目标元素: \n");
        scanf("%d", &opElem);
        switch (NextElem(Lists.elem[listIndex].L, opElem,
        ↪  &next)) {
        case INFEASIBLE:
            printf(" 顺序表不存在\n");
            break;
        case ERROR:
            printf(" 该元素没有后继\n");
            break;
        case OK:
            printf(" 该元素的后继为: %d\n", next);
            break;
        }
        break;
    case 10:
        printf(" 请输入插入的位置和元素: \n");
        scanf("%d %d", &posInput, &opElem);
        switch (ListInsert(&Lists.elem[listIndex].L,
        ↪  posInput, opElem)) {
        case INFEASIBLE:
            printf(" 顺序表不存在\n");
            break;
        case ERROR:
            printf(" 插入位置不合法\n");
            break;
        case OK:
            printf(" 插入成功\n");
```

```
            break;
        }
        break;
    case 11:
        printf(" 请输入删除元素的位置: \n");
        scanf("%d", &posInput);
        switch (ListDelete(&Lists.elem[listIndex].L,
        ↪  posInput, &opElem)) {
        case INFEASIBLE:
            printf(" 顺序表不存在\n");
            break;
        case ERROR:
            printf(" 删除位置错误或无元素\n");
            break;
        case OK:
            printf(" 删除成功, 被删元素为: %d\n", opElem);
            break;
        }
        break;
    case 12:
        ListTraverse(Lists.elem[listIndex].L);
        printf("\n");
        break;
    case 13:
        printf(" 最大连续子数组的和为: %d\n",
        ↪  MaxSubArray(Lists.elem[listIndex].L));
        break;
    case 14:
        printf(" 请输入目标值 k: \n");
        scanf("%d", &k);
        printf(" 满足条件的子数组个数为: %d\n",
        ↪  SubArrayNum(Lists.elem[listIndex].L, k));
        break;
    case 15:
        SortList(&Lists.elem[listIndex].L);
```

```
                printf(" 排序完成\n");
                break;
            case 16:
                printf(" 请输入保存文件名: \n");
                scanf("%s", filename);
                res = saveListToFile(Lists.elem[listIndex].L,
                ↪  filename);
                if (res == OK)
                    printf(" 保存成功\n");
                else
                    printf(" 保存失败\n");
                break;
            default:
                printf(" 输入错误, 请重新选择\n");
                break;
            }
        } while (op != 0);
    }
    break;
}
case 18:
    printf(" 请输入加载文件的文件名: \n");
    scanf("%s", filename);
    res = loadListFromFile(&L, filename);
    printf("\n");
    if (res == OK)
        printf(" 加载成功\n");
    else
        printf(" 加载失败\n");
    break;
default:
    printf(" 输入选项错误, 请重新选择\n");
    break;
}
}
```

```
    return 0;
}
```

附录 B 基于链式存储结构线性表实现的源程序

文件 1：function.h

```
#pragma once
#define _CRT_SECURE_NO_WARNINGS

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

#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
#define MAX_LISTS 10    // 多链表管理中最多保存 MAX_LISTS 个链表

typedef int status;
typedef int ElemType;

// 单链表结点的定义（带头结点）
typedef struct LNode {
    ElemType data;
    struct LNode* next;
} LNode, * LinkList;

// 定义一个用于管理多个单链表的结构体
typedef struct {
    LinkList lists[MAX_LISTS]; // 保存多个链表
    int count;                 // 当前拥有的链表个数
    int current;               // 当前活动链表的索引
} MultiList;
```

```
//函数 1: 初始化单链表
status InitList(LinkList* L);

 //函数 2: 销毁单链表
status DestroyList(LinkList* L);

//函数 3: 清空单链表
status ClearList(LinkList* L);

//函数 4: 判断单链表是否为空
status ListEmpty(LinkList L);

//函数 5: 返回单链表的长度
int ListLength(LinkList L);

//函数 6: 获取单链表的第 i 个元素
status GetElem(LinkList L, int i, ElemType* e);

//函数 7: 查找元素 e 在单链表中的位置序号
int LocateElem(LinkList L, ElemType e);

//函数 8: 获取单链表中元素 e 的前驱
status PriorElem(LinkList L, ElemType e, ElemType* pre);

//函数 9: 获取单链表中元素 e 的后继
status NextElem(LinkList L, ElemType e, ElemType* next);

//函数 10: 单链表插入元素
status ListInsert(LinkList* L, int i, ElemType e);

//函数 11: 单链表删除元素
status ListDelete(LinkList* L, int i, ElemType* e);

//函数 12: 遍历打印单链表
status ListTraverse(LinkList L);
```

```
//附加功能 1: 链表翻转
status reverseList(LinkList L);

//附加功能 2: 删除链表的倒数第 n 个结点
status RemoveNthFromEnd(LinkList L, int n);

//附加功能 3: 链表排序
status sortList(LinkList L);

//附加功能 4: 将链表保存到文件
status SaveList(LinkList L, char filename[]);

//附加功能 4: 从文件加载链表
status LoadList(LinkList* L, const char* filename);

//附加功能 5: 多链表管理

/* 初始化多链表管理器 */
void InitMultiList(MultiList* ml);

/* 创建新的单链表并加入管理器 */
status CreateNewList(MultiList* ml);

/* 删除管理器中指定索引的单链表 */
status RemoveList(MultiList* ml, char listNames[][30], const char*
↪  targetName);

/* 切换当前活动链表 */
status SwitchCurrentList(MultiList* ml, int index);

/* 显示管理器中所有链表的情况（显示序号及各链表长度），当前链表做标记 */
void ListAllLists(MultiList* ml);

//菜单打印函数
```

```
void PrintMenu();
void PrintSubMenu();
void PrintSubSubMenu(char* listname);
```

文件 2：function.c

```
#define _CRT_SECURE_NO_WARNINGS

#include"functions.h"

//规定所有函数处理都是带头结点的链表

status InitList(LinkList* L)
{
    if (*L != NULL) return INFEASIBLE;  // 已经初始化过
    *L = (LinkList)malloc(sizeof(LNode));
    if (*L == NULL) {
        return OVERFLOW;
    }
    (*L)->next = NULL;

    // 询问是否进行数据初始化
    int choice;
    printf(" 是否在初始化时输入数据？(1-是, 0-否): ");
    scanf("%d", &choice);
    if (choice == 1) {
        int n, data;
        printf(" 请输入要初始化的元素个数: ");
        scanf("%d", &n);
        LinkList tail = *L;
        for (int i = 0; i < n; i++) {
            LinkList newNode = (LinkList)malloc(sizeof(LNode));
            if (newNode == NULL) return OVERFLOW;
            printf(" 请输入第 %d 个元素的值: ", i + 1);
            scanf("%d", &data);
```

```
            newNode->data = data;
            newNode->next = NULL;
            tail->next = newNode;
            tail = newNode;
        }
    }
    return OK;
}
status DestroyList(LinkList* L)
/* 完全销毁线性表，不仅要释放线性表中所有数据元素节点的内存，还要释放头节点的内
↪  存，
最终让线性表回到未初始化的状态，也就是让指向线性表的指针变为 NULL*/
{
    if (*L == NULL) return INFEASIBLE;
    LinkList current = *L;
    LinkList temp;
    while (current != NULL) {
        temp = current;
        current = current->next;
        free(temp);
    }
    *L = NULL;
    return OK;
}
status ClearList(LinkList* L)
/* 清空线性表中的数据元素，也就是释放除头节点以外的所有数据元素节点的内存，
但会保留头节点，使得线性表仍然处于已初始化的状态，只是其中没有数据元素。*/
{
    if (*L == NULL) return INFEASIBLE;
    LinkList current = (*L)->next;
    LinkList temp;
    while (current != NULL) {
        temp = current;
        current = current->next;
        free(temp);
```

```
    }
    (*L)->next = NULL;
    return OK;
}
status ListEmpty(LinkList L)
{
    if(L==NULL) return INFEASIBLE;
    if(L->next == NULL) 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 == NULL) return INFEASIBLE;
    int length = 0;
    LinkList temp = L->next;
    while (temp != NULL) {
        temp = temp->next;
        length++;
    }
    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 ERROR；如果线性表 L 不存在，返回 INFEASIBLE。
{
    if (L == NULL) return INFEASIBLE;
    if (i<1 || i>ListLength(L)) return ERROR;
    LinkList temp = L;
    for (int j = 1; j <= i; j++) {
        temp = temp->next;
    }
    *e = temp->data;
    return OK;
}
```

```
int LocateElem(LinkList L, ElemType e)
// 如果线性表 L 存在，查找元素 e 在线性表 L 中的位置序号；如果 e 不存在，返回
↪  ERROR；当线性表 L 不存在时，返回 INFEASIBLE。
{
    if (L == NULL) return INFEASIBLE;
    int position = 0, flag = 0;
    LinkList temp = L->next;
    while (temp != NULL) {
        position++;
        if (temp->data == e) {
            flag = 1;
            break;
        }
        temp = temp->next;
    }
    if (flag == 0) return ERROR;
    else return position;
}
status PriorElem(LinkList L, ElemType e, ElemType* pre)
// 如果线性表 L 存在，获取线性表 L 中元素 e 的前驱，保存在 pre 中，返回 OK；如
↪  果没有前驱，返回 ERROR；如果线性表 L 不存在，返回 INFEASIBLE。
{
    if (L == NULL) return INFEASIBLE;
    if (L->next == NULL) return INFEASIBLE;
    if (L->next->data == e) return ERROR; // 如果第一个元素就是目标 e，则 e
    ↪  没有前驱

    // 定义两个指针：prev 指向前驱，curr 指向当前节点
    LinkList prev = L->next;      // 第一个有效节点
    LinkList curr = prev->next;     // 从第二个有效节点开始

    while (curr != NULL) {
        // 如果找到了目标元素，prev 就是它的前驱
        if (curr->data == e) {
            *pre = prev->data;
```

```
            return OK;
        }
        prev = curr;
        curr = curr->next;
    }
    return ERROR; // 遍历完链表仍未找到元素 e
}
status NextElem(LinkList L, ElemType e, ElemType* next)
// 如果线性表 L 存在，获取线性表 L 元素 e 的后继，保存在 next 中，返回 OK；如
↪  果没有后继，返回 ERROR；如果线性表 L 不存在，返回 INFEASIBLE。
{
    if (L == NULL) return INFEASIBLE;
    if (L->next == NULL) return INFEASIBLE;
    LinkList currnode = L->next;
    LinkList nextnode = currnode->next;

    while (nextnode != NULL) {
        if (currnode->data == e) {
            *next = nextnode->data;
            return OK;
        }
        currnode = nextnode;
        nextnode = nextnode->next;
    }
    return ERROR;
}
status ListInsert(LinkList* L, int i, ElemType e)
// 如果线性表 L 存在，将元素 e 插入到线性表 L 的第 i 个元素之前，返回 OK；当插
↪  入位置不正确时，返回 ERROR；如果线性表 L 不存在，返回 INFEASIBLE。
{
    if (L == NULL) return INFEASIBLE;
    LinkList head = *L;
    if (head == NULL) return INFEASIBLE;
    int length = 0;
    LinkList temp = head->next;
```

```
    while (temp != NULL) {
        temp = temp->next;
        length++;
    }
    if (i<1 || i>length + 1) return ERROR;
    LinkList current, p;
    current = head;
    p = (LinkList)malloc(sizeof(LinkList));
    p->data = e;
    for (int j = 1; j < i; j++) {
        current = current->next;
    }
    p->next = current->next;
    current->next = p;
    return OK;
}
status ListDelete(LinkList* L, int i, ElemType* e)
// 如果线性表 L 存在，删除线性表 L 的第 i 个元素，并保存在 e 中，返回 OK；当删
↪  除位置不正确时，返回 ERROR；如果线性表 L 不存在，返回 INFEASIBLE。
{
    if (L == NULL || *L == NULL) return INFEASIBLE;
    LinkList head = *L;
    int length = 0;
    LinkList temp = head->next;
    while (temp != NULL) {
        temp = temp->next;
        length++;
    }
    if (i<1 || i>length) return ERROR;

    LinkList current = head;
    for (int j = 1; j < i; j++) {
        current = current->next;
    }
    *e = current->next->data;
```

```
    LinkList p = current->next;
    current->next = current->next->next;
    free(p);
    return OK;
}
status ListTraverse(LinkList L)
// 如果线性表 L 存在，依次显示线性表中的元素，每个元素间空一格，返回 OK；如果线
↪  性表 L 不存在，返回 INFEASIBLE。
{
    if (L == NULL) return INFEASIBLE;
    LinkList current = L->next;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
    return OK;
}
status reverseList(LinkList L)
//附加功能 1：链表翻转，若链表不存在或为空则返回 INFEASIBLE
{
    if (L == NULL || L->next == NULL) return INFEASIBLE;
    LNode* prev = NULL;
    LNode* curr = L->next;
    while (curr) {
        LNode* next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    L->next = prev;
    return OK;
}
status RemoveNthFromEnd(LinkList L, int n)
//附加功能 2：删除链表的倒数第 n 个结点（双指针法，使用虚拟头结点）
```

```
{
    if (L == NULL || L->next == NULL) return INFEASIBLE;
    LNode dummy;
    dummy.next = L->next;
    LNode* fast = &dummy;
    LNode* slow = &dummy;
    int i;
    for (i = 0; i <= n; i++) {
        if (fast == NULL)
            return ERROR;
        fast = fast->next;
    }
    while (fast) {
        fast = fast->next;
        slow = slow->next;
    }
    LNode* toDelete = slow->next;
    if (toDelete == NULL)
        return ERROR;
    slow->next = toDelete->next;
    free(toDelete);
    L->next = dummy.next;
    return OK;
}
status sortList(LinkList L)
//附加功能 3: 链表排序 (暴力法)
{
    if (L == NULL || L->next == NULL) return INFEASIBLE;
    for (LNode* p = L->next; p != NULL; p = p->next) {
        for (LNode* q = p->next; q != NULL; q = q->next) {
            if (p->data > q->data) {
                int temp = p->data;
                p->data = q->data;
                q->data = temp;
            }
```

```
        }
    }
    return OK;
}
status SaveList(LinkList L, char filename[])
// 附加功能 4: 以二进制方式将链表保存到文件
{
    if (L == NULL) return INFEASIBLE;
    FILE* fp = fopen(filename, "w");
    if (fp == NULL) {
        printf(" 无法打开文件进行保存! \n");
        return ERROR;
    }
    int len = ListLength(L);
    fprintf(fp, "%d\n", len);
    LNode* p = L->next;
    while (p) {
        fprintf(fp, "%d ", p->data);
        p = p->next;
    }
    fclose(fp);
    return OK;
}
status LoadList(LinkList* L, char filename[])
// 如果线性表 L 不存在, 将 FileName 文件中的数据读入到线性表 L 中, 返回 OK, 否
↪ 则返回 INFEASIBLE。
{
    // 如果链表已存在, 则返回 INFEASIBLE
    if (*L != NULL) return INFEASIBLE;

    FILE* fp = fopen(filename, "r");
    if (fp == NULL) {
        printf(" 无法打开文件进行加载! \n");
        return ERROR;
    }
```

```
int count;
if (fscanf(fp, "%d", &count) != 1) {
    fclose(fp);
    return ERROR;
}

LinkList head = (LinkList)malloc(sizeof(LNode));
if (head == NULL) {
    fclose(fp);
    return OVERFLOW;
}
head->next = NULL;
LinkList tail = head;

for (int i = 0; i < count; i++) {
    int value;
    if (fscanf(fp, "%d", &value) != 1) {
        // 出现读取错误，先释放已分配的结点
        LinkList temp;
        while (head != NULL) {
            temp = head;
            head = head->next;
            free(temp);
        }
        fclose(fp);
        return ERROR;
    }

    // 分配新结点
    LinkList newNode = (LinkList)malloc(sizeof(LNode));
    if (newNode == NULL) {
        // 内存分配失败，释放所有已分配的结点
        LinkList temp;
        while (head != NULL) {
```

```
                temp = head;
                head = head->next;
                free(temp);
            }
            fclose(fp);
            return OVERFLOW;
        }

        newNode->data = value;
        newNode->next = NULL;

        tail->next = newNode;
        tail = newNode;
    }

    fclose(fp);
    *L = head;
    return OK;
}

/* ==================== 多线性表管理 (附加功能 5) ==================== */

/* 初始化多链表管理器 */
void InitMultiList(MultiList* ml)
{
    ml->count = 0;
    ml->current = -1;
    for (int i = 0; i < MAX_LISTS; i++) {
        ml->lists[i] = NULL;
    }
}

/* 创建新的单链表并加入管理器 */
status CreateNewList(MultiList* ml)
{
```

```
    if (ml->count >= MAX_LISTS) {
        printf(" 已达到管理的最大链表个数! \n");
        return ERROR;
    }
    LinkList p = NULL;
    if (InitList(&p) != OK)
        return ERROR;
    ml->lists[ml->count] = p;
    ml->current = ml->count;  // 新创建的链表自动成为活动链表
    ml->count++;
    //printf(" 创建新链表成功! 当前活动链表序号: %d\n", ml->current);
    return OK;
}

/* 删除管理器中指定索引的单链表 */
status RemoveList(MultiList* ml, char listNames[][30], const char*
↪  targetName)
{
    int index = -1;
    // 在名称数组中查找目标链表名称对应的索引
    for (int i = 0; i < ml->count; i++) {
        if (strcmp(listNames[i], targetName) == 0) {
            index = i;
            break;
        }
    }
    if (index == -1) {
        printf(" 链表不存在! \n");
        return ERROR;
    }
    // 销毁链表
    if (DestroyList(&ml->lists[index]) != OK) {
        printf(" 销毁链表失败! \n");
        return ERROR;
    }
```

```
    // 将后面的链表指针和名称整体前移
    for (int i = index; i < ml->count - 1; i++) {
        ml->lists[i] = ml->lists[i + 1];
        strcpy(listNames[i], listNames[i + 1]);
    }
    ml->lists[ml->count - 1] = NULL;
    listNames[ml->count - 1][0] = '\0';  // 清空最后一个名称
    ml->count--;
    if (ml->count == 0)
        ml->current = -1;
    else if (ml->current >= ml->count)
        ml->current = ml->count - 1;
    printf(" 链表 \"%s\" 删除成功! \n", targetName);
    return OK;
}

/* 切换当前活动链表 */
status SwitchCurrentList(MultiList* ml, int index)
{
    if (index < 0 || index >= ml->count) {
        printf(" 不存在该序号的链表! \n");
        return ERROR;
    }
    ml->current = index;
    printf(" 当前活动链表切换到序号: %d\n", ml->current);
    return OK;
}

/* 显示管理器中所有链表的情况 (显示序号及各链表长度), 当前链表做标记 */
void ListAllLists(MultiList* ml)
{
    if (ml->count == 0) {
        printf(" 当前没有管理任何链表! \n");
        return;
    }
```

```
    printf(" 多链表管理信息: \n");
    for (int i = 0; i < ml->count; i++) {
        int len = ListLength(ml->lists[i]);
        printf(" 序号 %d: 链表长度 = %d", i, len);
        if (i == ml->current)
            printf("  [当前活动链表]");
        printf("\n");
    }
}

void PrintMenu()
{
    printf("\n  -------单链表演示系统-------  \n");
    printf("**************************************\n");
    printf("  0. 退出演示系统\n");
    printf("  1. 初始化单个单链表\n");
    printf("  2. 销毁当前单链表\n");
    printf("  3. 清空当前链表\n");
    printf("  4. 判断当前链表是否为空\n");
    printf("  5. 获取当前链表长度\n");
    printf("  6. 获得当前链表第 i 个元素\n");
    printf("  7. 查找元素 e 的位置\n");
    printf("  8. 获得元素 e 的前驱\n");
    printf("  9. 获得元素 e 的后继\n");
    printf(" 10. 在当前链表中插入元素\n");
    printf(" 11. 删除当前链表中的元素\n");
    printf(" 12. 遍历当前链表\n");
    printf(" 13. 附加功能 1: 翻转当前链表\n");
    printf(" 14. 附加功能 2: 删除当前链表的倒数第 n 个结点\n");
    printf(" 15. 附加功能 3: 对当前链表从小到大排序\n");
    printf(" 16. 附加功能 4: 保存当前链表到文件\n");
    printf(" 17. 附加功能 5: 从文件加载当前链表\n");
    printf(" 18. 附加功能 6: 多线性表管理\n");
    printf("**************************************\n");
    printf(" 请输入你的选择: ");
```

```
}

void PrintSubMenu()
{
    printf("\n--- 多链表管理子系统 ---\n");
    printf("  1. 创建新的单链表\n");
    printf("  2. 删除某个单链表\n");
    printf("  3. 显示所有单链表状态\n");
    printf("  4. 操作其中某个单链表\n");
    printf("  0. 返回主菜单\n");
    printf(" 请输入你的选择: ");
}

void PrintSubSubMenu(char* listname)
{
    printf("\n-----操作单链表 \"%s\" -----\n", listname);
    printf("  0. 返回子菜单\n");
    printf("  1. 销毁单链表\n");
    printf("  2. 清空单链表\n");
    printf("  3. 单链表判空\n");
    printf("  4. 获取线性表长度\n");
    printf("  5. 获取指定位置元素\n");
    printf("  6. 查找指定元素的位置\n");
    printf("  7. 获取指定元素的前驱\n");
    printf("  8. 获取指定元素的后继\n");
    printf("  9. 插入元素至指定位置\n");
    printf(" 10. 删除指定位置元素\n");
    printf(" 11. 遍历单链表\n");
    printf(" 12. 附加功能 1: 翻转当前链表\n");
    printf(" 13. 附加功能 2: 删除当前链表的倒数第 n 个结点\n");
    printf(" 14. 附加功能 3: 对单链表从小到大排序\n");
    printf(" 15. 附加功能 4: 保存单链表到文件\n");
    printf(" 请输入您的选择: \n");
}
```

文件 3：main.c

```
#define _CRT_SECURE_NO_WARNINGS

#include"functions.h"

int main()
{
    system("color F0");
    LinkList singleL = NULL;  // 单链表操作用的变量
    MultiList ml;             // 多链表管理子系统
    InitMultiList(&ml);       // 初始化多链表管理器

    int choice;
    char listNames[MAX_LISTS][30] = { 0 };

    do {
        PrintMenu();

        if (scanf("%d", &choice) != 1) {
            while (getchar() != '\n');  // 清除输入缓冲区
            continue;
        }

        switch (choice)
        {
        case 0:
            printf(" 退出系统。\n");
            break;
        case 1: {  // 初始化单链表（带可选数据输入）
            if (singleL != NULL) {
                int confirm;
                printf(" 单链表已存在，重新初始化将销毁当前链表。是否继续？
                ↪  (1-是，0-否)：");
                scanf("%d", &confirm);
```

```
            if (confirm == 1) {
                if (DestroyList(&singleL) == OK)
                    printf(" 原有单链表已销毁。\n");
                else
                    printf(" 销毁失败! \n");
            }
            else {
                printf(" 操作取消。\n");
                break;
            }
        }
        if (InitList(&singleL) == OK)
            printf(" 单链表初始化成功! \n");
        else
            printf(" 单链表初始化失败! \n");
        break;
    }
    case 2: {  // 销毁当前单链表
        if (singleL == NULL)
            printf(" 当前无单链表存在! \n");
        else {
            if (DestroyList(&singleL) == OK)
                printf(" 单链表销毁成功! \n");
            else
                printf(" 销毁失败! \n");
        }
        break;
    }
    case 3: {  // 清空当前单链表 (保留头结点)
        if (singleL == NULL)
            printf(" 当前无单链表存在! \n");
        else {
            if (ClearList(&singleL) == OK)
                printf(" 单链表已清空! \n");
            else
```

```
            printf(" 清空失败! \n");
        }
        break;
    }
    case 4: {  // 判断是否为空
        if (singleL == NULL)
            printf(" 当前无单链表存在! \n");
        else {
            status ret = ListEmpty(singleL);
            if (ret == TRUE)
                printf(" 单链表为空! \n");
            else if (ret == FALSE)
                printf(" 单链表不为空! \n");
            else
                printf(" 操作失败! \n");
        }
        break;
    }
    case 5: {  // 获取长度
        if (singleL == NULL)
            printf(" 当前无单链表存在! \n");
        else {
            int len = ListLength(singleL);
            if (len == INFEASIBLE)
                printf(" 操作失败! \n");
            else
                printf(" 单链表长度为: %d\n", len);
        }
        break;
    }
    case 6: {  // 获得第 i 个元素
        if (singleL == NULL) {
            printf(" 当前无单链表存在! \n");
        }
        else {
```

```
            int i;
            ElemType e;
            printf(" 请输入要获取的元素序号 i: ");
            scanf("%d", &i);
            if (GetElem(singleL, i, &e) == OK)
                printf(" 单链表第 %d 个元素的值为: %d\n", i, e);
            if(GetElem(singleL, i, &e) == INFEASIBLE)
                printf(" 当前无单链表存在! \n");
            if (GetElem(singleL, i, &e) == ERROR)
                printf(" 位置不合法! \n");
        }
        break;
    }
    case 7: {  // 查找元素 e 的位置
        if (singleL == NULL) {
            printf(" 当前无单链表存在! \n");
        }
        else {
            ElemType e;
            printf(" 请输入要查找的元素 e: ");
            scanf("%d", &e);
            int pos = LocateElem(singleL, e);
            if (pos == INFEASIBLE)
                printf(" 操作失败! \n");
            else if (pos == ERROR)
                printf(" 未找到元素 %d! \n", e);
            else
                printf(" 元素 %d 在单链表中的位置为: %d\n", e, pos);
        }
        break;
    }
    case 8: {  // 获得元素 e 的前驱
        if (singleL == NULL) {
            printf(" 当前无单链表存在! \n");
        }
```

```
        else {
            ElemType e, pre;
            printf(" 请输入要查询其前驱的元素 e: ");
            scanf("%d", &e);
            if (PriorElem(singleL, e, &pre) == OK)
                printf(" 元素 %d 的前驱为: %d\n", e, pre);
            if(PriorElem(singleL, e, &pre) == ERROR)
                printf(" 没有找到该元素或该元素为第一个! \n");
            if (PriorElem(singleL, e, &pre) == INFEASIBLE)
                printf(" 单链表不存在或单链表为空! \n");
        }
        break;
    }
    case 9: {  // 获得元素 e 的后继
        if (singleL == NULL) {
            printf(" 当前无单链表存在! \n");
        }
        else {
            ElemType e, next;
            printf(" 请输入要查询其后继的元素 e: ");
            scanf("%d", &e);
            if (NextElem(singleL, e, &next) == OK)
                printf(" 元素 %d 的后继为: %d\n", e, next);
            if (NextElem(singleL, e, &next) == ERROR)
                printf(" 没有找到后继或该元素为最后一个! \n");
            if (NextElem(singleL, e, &next) == INFEASIBLE)
                printf(" 单链表不存在或单链表为空! \n");
        }
        break;
    }
    case 10: { // 在单链表中插入元素
        if (singleL == NULL) {
            printf(" 当前无单链表存在, 请先初始化! \n");
        }
        else {
```

```
            int pos;
            ElemType e;
            printf(" 请输入插入位置 i: ");
            scanf("%d", &pos);
            printf(" 请输入要插入的元素 e: ");
            scanf("%d", &e);
            if (ListInsert(&singleL, pos, e) == OK)
                printf(" 插入成功! \n");
            else
                printf(" 插入失败! \n");
        }
        break;
    }
    case 11: { // 删除单链表中的元素
        if (singleL == NULL) {
            printf(" 当前无单链表存在, 请先初始化! \n");
        }
        else {
            int pos;
            ElemType e;
            printf(" 请输入要删除的元素的位置 i: ");
            scanf("%d", &pos);
            if (ListDelete(&singleL, pos, &e) == OK)
                printf(" 删除成功, 删除的元素为: %d\n", e);
            if (ListDelete(&singleL, pos, &e) == ERROR)
                printf(" 位置不合法, 删除失败! \n");
            if (ListDelete(&singleL, pos, &e) == INFEASIBLE)
                printf(" 单链表不存在! \n");
        }
        break;
    }
    case 12: { // 遍历单链表
        if (singleL == NULL) {
            printf(" 当前无单链表存在! \n");
        }
```

```
        else {
            printf(" 单链表中的元素为: ");
            if (ListTraverse(singleL) != OK)
                printf(" 遍历失败! \n");
        }
        break;
    }
    case 13: { // 翻转单链表
        if (singleL == NULL) {
            printf(" 当前无单链表存在! \n");
        }
        else {
            if (reverseList(singleL) == OK)
                printf(" 链表翻转成功! \n");
            else
                printf(" 翻转失败! \n");
        }
        break;
    }
    case 14: { // 删除倒数第 n 个结点
        if (singleL == NULL) {
            printf(" 当前无单链表存在! \n");
        }
        else {
            int n;
            printf(" 请输入删除倒数第 n 个结点的 n 值: ");
            scanf("%d", &n);
            if (RemoveNthFromEnd(singleL, n) == OK)
                printf(" 删除成功! \n");
            else
                printf(" 删除失败! \n");
        }
        break;
    }
    case 15: { // 对单链表进行排序
```

```
        if (singleL == NULL) {
            printf(" 当前无单链表存在! \n");
        }
        else {
            if (sortList(singleL) == OK)
                printf(" 排序成功! \n");
            else
                printf(" 排序失败! \n");
        }
        break;
    }
    case 16: { // 保存单链表到文件
        if (singleL == NULL) {
            printf(" 当前无单链表存在! \n");
        }
        else {
            char filename[100];
            printf(" 请输入保存文件的文件名: ");
            scanf("%s", filename);
            if (SaveList(singleL, filename) == OK)
                printf(" 保存成功! \n");
            else
                printf(" 保存失败! \n");
        }
        break;
    }
    case 17: { // 从文件加载单链表
        if (singleL != NULL) {
            printf(" 当前已有单链表, 请先销毁后再加载文件。\n");
        }
        else {
            char filename[100];
            printf(" 请输入加载文件的文件名: ");
            scanf("%s", filename);
            if (LoadList(&singleL, filename) == OK)
```

```
            printf(" 加载成功! \n");
        else
            printf(" 加载失败! \n");
    }
    break;
}
case 18: {
    /* --- 多线性表管理子系统 --- */
    int subChoice;
    do {
        PrintSubMenu();
        int ret = scanf("%d", &subChoice);
        if (ret != 1) {
            // 清空缓冲
            while (getchar() != '\n');
            printf(" 无效选择, 请重新输入! \n");
            continue; // 重新进入循环
        }
        switch (subChoice)
        {
        case 0:
            printf(" 返回主菜单。\n");
            break;
        case 1: {
            // 创建新的单链表, 并允许用户为该链表起名字
            char tempName[30];
            printf(" 请输入新链表的名字: ");
            scanf("%s", tempName);
            if (CreateNewList(&ml) == OK) {
                // 新链表在数组中的下标为 ml.count - 1
                strcpy(listNames[ml.count - 1], tempName);
                printf(" 创建新链表成功, 链表名字为 \"%s\".\n",
                ↪  tempName);
            }
            else {
```

```
            printf(" 创建链表失败! \n");
        }
        break;
    }
    case 2: {
        //删除某个单链表
        char listname[30];
        printf(" 请输入要删除的链表的名字: ");
        scanf("%s", listname);
        RemoveList(&ml, listNames, listname);
    }
    case 3: {
        // 显示所有单链表状态: 包括名字、所有元素、表长
        if (ml.count == 0)
            printf(" 当前无链表存在! \n");
        else {
            printf("\n 所有链表状态: \n");
            for (int j = 0; j < ml.count; j++) {
                int len = ListLength(ml.lists[j]);
                printf(" 链表名字: \"%s\", 长度: %d, 元素: ",
                ↪  listNames[j], len);
                ListTraverse(ml.lists[j]);
            }
        }
        break;
    }
    case 4: {
        // 操作其中某个单链表: 根据输入的链表名字查找
        char searchName[30];
        printf(" 请输入要操作的链表名字: ");
        scanf("%s", searchName);
        int foundIndex = -1;
        for (int j = 0; j < ml.count; j++) {
            if (strcmp(listNames[j], searchName) == 0) {
                foundIndex = j;
```

```
            break;
        }
    }
    if (foundIndex == -1) {
        printf(" 未找到单链表! \n");
    }
    else {
        int op;
        do {
            PrintSubSubMenu(searchName);
            int ret = scanf("%d", &op);
            if (ret != 1) {
                // 清空缓冲
                while (getchar() != '\n');
                printf(" 无效选择, 请重新输入! \n");
                continue; // 重新进入循环
            }
            switch (op)
            {
            case 0:
                printf(" 返回多链表管理子系统。\n");
                break;
            case 1: {
                if (DestroyList(&ml.lists[foundIndex]) ==
                ↪  OK) {
                    printf(" 链表 \"%s\" 已销毁! \n",
                    ↪  searchName);
                    // 删除该链表: 将后续记录前移
                    for (int k = foundIndex; k < ml.count -
                    ↪  1; k++) {
                        ml.lists[k] = ml.lists[k + 1];
                        strcpy(listNames[k], listNames[k +
                        ↪  1]);
                    }
                    ml.lists[ml.count - 1] = NULL;
```

```
            ml.count--;
            op = 0;  // 返回至子菜单
        }
        else {
            printf(" 销毁失败! \n");
        }
    } break;
    case 2: {
        if (ClearList(&ml.lists[foundIndex]) == OK)
            printf(" 链表 \"%s\" 清空成功! \n",
            ↪  searchName);
        else
            printf(" 清空操作失败! \n");
    } break;
    case 3: {
        status ret =
        ↪  ListEmpty(ml.lists[foundIndex]);
        if (ret == TRUE)
            printf(" 链表 \"%s\" 为空! \n",
            ↪  searchName);
        else if (ret == FALSE)
            printf(" 链表 \"%s\" 不为空! \n",
            ↪  searchName);
        else
            printf(" 操作失败! \n");
    } break;
    case 4: {
        int len = ListLength(ml.lists[foundIndex]);
        if (len == INFEASIBLE)
            printf(" 操作失败! \n");
        else
            printf(" 链表 \"%s\" 的长度为: %d\n",
            ↪  searchName, len);
    } break;
    case 5: {
```

```
            int pos;
            ElemType e;
            printf(" 请输入要获取的元素位置: ");
            scanf("%d", &pos);
            if (GetElem(ml.lists[foundIndex], pos, &e)
            ↪  == OK)
                printf(" 链表 \"%s\" 的第 %d 个元素为:
                ↪  %d\n", searchName, pos, e);
            else
                printf(" 操作失败! \n");
        } break;
        case 6: {
            ElemType e;
            printf(" 请输入要查找的元素: ");
            scanf("%d", &e);
            int pos = LocateElem(ml.lists[foundIndex],
            ↪  e);
            if (pos == ERROR)
                printf(" 未找到元素 %d! \n", e);
            else if (pos == INFEASIBLE)
                printf(" 操作失败! \n");
            else
                printf(" 元素 %d 在链表 \"%s\" 中的位置
                ↪  为: %d\n", e, searchName, pos);
        } break;
        case 7: {
            ElemType e, pre;
            printf(" 请输入要查询前驱的元素: ");
            scanf("%d", &e);
            if (PriorElem(ml.lists[foundIndex], e,
            ↪  &pre) == OK)
                printf(" 元素 %d 的前驱为: %d\n", e,
                ↪  pre);
            else
                printf(" 操作失败! \n");
```

```
} break;
case 8: {
    ElemType e, next;
    printf(" 请输入要查询后继的元素: ");
    scanf("%d", &e);
    if (NextElem(ml.lists[foundIndex], e,
    ↪  &next) == OK)
        printf(" 元素 %d 的后继为: %d\n", e,
        ↪  next);
    else
        printf(" 操作失败! \n");
} break;
case 9: {
    int pos;
    ElemType e;
    printf(" 请输入插入位置: ");
    scanf("%d", &pos);
    printf(" 请输入要插入的元素: ");
    scanf("%d", &e);
    if (ListInsert(&ml.lists[foundIndex], pos,
    ↪  e) == OK)
        printf(" 插入成功! \n");
    else
        printf(" 插入失败! \n");
} break;
case 10: {
    int pos;
    ElemType e;
    printf(" 请输入删除位置: ");
    scanf("%d", &pos);
    if (ListDelete(&ml.lists[foundIndex], pos,
    ↪  &e) == OK)
        printf(" 删除成功, 删除的元素为: %d\n",
        ↪  e);
    else
```

```
            printf(" 删除失败! \n");
        } break;
        case 11: {
            printf(" 链表 \"%s\" 的元素: ",
            ↪ searchName);
            if (ListTraverse(ml.lists[foundIndex]) !=
            ↪ OK)
                printf(" 操作失败! \n");
        } break;
        case 12: {
            if (reverseList(ml.lists[foundIndex]) ==
            ↪ OK)
                printf(" 链表 \"%s\" 翻转成功! \n",
                ↪ searchName);
            else
                printf(" 翻转失败! \n");
        } break;
        case 13: {
            int n;
            printf(" 请输入删除倒数第 n 个结点的 n 值:
            ↪ ");
            scanf("%d", &n);
            if (RemoveNthFromEnd(ml.lists[foundIndex],
            ↪ n) == OK)
                printf(" 删除成功! \n");
            else
                printf(" 删除失败! \n");
        } break;
        case 14: {
            if (sortList(ml.lists[foundIndex]) == OK)
                printf(" 链表 \"%s\" 排序成功! \n",
                ↪ searchName);
            else
                printf(" 排序失败! \n");
        } break;
```

```
                        case 15: {
                            char filename[100];
                            printf(" 请输入保存文件的文件名: ");
                            scanf("%s", filename);
                            if (SaveList(ml.lists[foundIndex],
                            ↪  filename) == OK)
                                printf(" 保存成功! \n");
                            else
                                printf(" 保存失败! \n");
                        } break;
                        default:
                            printf(" 无效选择, 请重新输入! \n");
                        } // end switch (op)
                    } while (op != 0);
                }
            } break;
            default:
                printf(" 无效选择, 请重新输入! \n");
            }
        } while (subChoice != 0);
    } break;
    default:
        printf(" 无效选择, 请重新输入! \n");
    }
} while (choice != 0);

// 退出前释放单链表及多链表管理中所有链表所占的内存
if (singleL != NULL)
    DestroyList(&singleL);
for (int i = 0; i < ml.count; i++) {
    DestroyList(&ml.lists[i]);
}
printf(" 程序已退出。\n");
return 0;
}
```

附录 C 基于二叉链表二叉树实现的源程序

文件 1：function.h

```
#pragma once
#define _CRT_SECURE_NO_WARNINGS

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

#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
#define MAX_TREE_SIZE 100
#define MAX_BTREES 100

typedef int status;
typedef int KeyType;
typedef struct {
        KeyType  key;    //结点关键字
        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;

// 单个二叉树管理项，包含二叉树名称和二叉树指针
typedef struct {
        char name[50];  // 二叉树名称
```

```
        BiTree tree;    // 指向二叉树根节点的指针
} BiTreeItem;

// 多二叉树管理结构：顺序表
typedef struct {
        BiTreeItem items[MAX_BTREES]; // 存放多个二叉树管理项的数组
        int count;                    // 当前管理的二叉树个数
} BiTreeManager;

status CreateBiTree_Recursive(BiTree* T, TElemType definition[], int*
↪  index); //带空子树的二叉树前序遍历序列创建二叉树辅助函数
status CreateBiTree_PreIn(BiTree* T, TElemType pre[], TElemType in[], int
↪  n); //前序 + 中序创建二叉树辅助函数
status CreateBiTree_PostIn(BiTree* T, TElemType post[], TElemType in[], int
↪  n); //后序 + 中序创建二叉树辅助函数
status CreateBiTree(BiTree* T, TElemType definition[], int choice); //函数
↪  1：创建二叉树

status InitBiTree(BiTree* T); //初始化二叉树
status DestroyBiTree(BiTree* T); //函数 2：销毁二叉树
status ClearBiTree(BiTree* T); //函数 3：清空二叉树
status BiTreeEmpty(BiTree T); //函数 4：二叉树判空
int BiTreeDepth(BiTree T); //函数 5：求二叉树的深度
BiTNode* LocateNode(BiTree T, KeyType e); //函数 6：查找结点
status Assign(BiTree* T, KeyType e, TElemType value); //函数 7：结点赋值
BiTNode* GetSibling(BiTree T, KeyType e); //函数 8：获得兄弟结点
status InsertNode(BiTree* T, KeyType e, int LR, TElemType c); //函数 9：插入
↪  结点
BiTNode* FindNodeWithParent(BiTree T, KeyType e, BiTNode* parent, BiTNode**
↪  outParent); //删除结点辅助函数，返回待删除结点的双亲结点
status DeleteNode(BiTree* T, KeyType e); //函数 10：删除结点
void visit(BiTree);
status PreOrderTraverse(BiTree T, void (*visit)(BiTree)); //函数 11：先序遍
↪  历
```

```
status InOrderTraverse(BiTree T, void (*visit)(BiTree)); //函数 12 (1): 中序
↪ 遍历
status InOderTraverse_nonrecursive(BiTree T, void (*visit)(BiTree)); //函数
↪ 12 (2): 中序遍历的非递归实现
status PostOrderTraverse(BiTree T, void (*visit)(BiTree)); //函数 13: 后序遍
↪ 历
status LevelOrderTraverse(BiTree T, void (*visit)(BiTree)); //函数 14: 层序
↪ 遍历

int MaxPathSum(BiTree T); //附加功能 1: 最大路径和
BiTNode* LowestCommonAncestor(BiTree T, KeyType e1, KeyType e2); //附加功能
↪ 2: 最近公共祖先求解
void InvertTree(BiTree T); //附加功能 3: 翻转二叉树
void SaveBiTree_PreOrder(BiTree T, FILE* fp); //二叉树保存至文件辅助函数
status SaveBiTree(BiTree T, char FileName[]); //附加功能 4: 二叉树的文件形式
↪ 保存
status LoadBiTree(BiTree* T, char FileName[]); //附加功能 4: 文件加载创建二
↪ 叉树
/* 附加功能 5: 多二叉树管理 */
status InitBiTreeManager(BiTreeManager* manager);
status AddBiTree(BiTreeManager* manager, char name[], BiTree tree);
status RemoveBiTree(BiTreeManager* manager, char name[]);
void ListBiTrees(BiTreeManager* manager);
BiTree GetBiTree(BiTreeManager* manager, char name[]);

/* 菜单函数 */
void PrintMenu();
void PrintSubMenu();
void PrintSubSubMenu(char* listname);
```

文件 2：function.c

```
#define _CRT_SECURE_NO_WARNINGS
#include"functions.h"
```

```
status InitBiTree(BiTree* T)
{
        *T = (BiTree)malloc(sizeof(BiTNode));
        if (*T == NULL) {
                return OVERFLOW;  // 内存分配失败
        }
        // 初始化头结点数据
        (*T)->data.key = 0;
        strcpy((*T)->data.others, "head");
        (*T)->lchild = NULL;
        (*T)->rchild = NULL;
        return OK;
}
void visit(BiTree T)
{
        printf(" %d,%s", T->data.key, T->data.others);
}
status PreOrderTraverse(BiTree T, void (*visit)(BiTree))
{
        if (T != NULL) {
                visit(T);
                PreOrderTraverse(T->lchild, visit);
                PreOrderTraverse(T->rchild, visit);
        }
        return OK;
}
status InOrderTraverse(BiTree T, void (*visit)(BiTree))
{
        if (T != NULL) {
                InOrderTraverse(T->lchild, visit);
                visit(T);
                InOrderTraverse(T->rchild, visit);
        }
        return OK;
}
```

```
status PostOrderTraverse(BiTree T, void (*visit)(BiTree))
{
        if (T != NULL) {
                PostOrderTraverse(T->lchild, visit);
                PostOrderTraverse(T->rchild, visit);
                visit(T);
        }
        return OK;
}
status InOderTraverse_nonrecursive(BiTree T, void (*visit)(BiTree))
//借鉴中序遍历递归算法执行过程中递归工作栈的原理，非递归算法可以设置一个指针栈
↪  暂存搜索路径中的结点指针。
{
        BiTree stack[MAX_TREE_SIZE];  //定义指针栈
        int top = 0; //初始化顺序栈
        stack[top++] = T; //根指针进栈
        BiTree temp;
        while (top) {
                temp = stack[top - 1]; //读取栈顶指针
                while (temp) {
                        temp = temp->lchild; //向左走到尽头
                        stack[top++] = temp; //最左侧指针依次进栈
                }
                top--; //空指针 NULL 退栈，现在 top 为栈顶元素的上一个
                if (top != 0) {
                        temp = stack[--top]; //路径上最左侧结点退栈并访问
                        visit(T);
                        stack[top++] = temp->rchild; //右孩子指针进栈
                }
        }
        return OK;

        //改进算法:
        /*
        BiTree stack[MAX_TREE_SIZE];
```

```
        int top = 0;
        BiTree cur = T;
        while (cur != NULL || top > 0) {
                while (cur != NULL) {
                        stack[top++] = cur;
                        cur = cur->lchild;
                }
                cur = stack[--top];
                visit(T);
                cur = cur->rchild;
        }
        return OK;
        */
}
status LevelOrderTraverse(BiTree T, void (*visit)(BiTree))
//层序遍历中先访问的结点其左右孩子也要先访问，队列先进先出的特性能满足这一要求。
↪
{
        BiTree Queue[MAX_TREE_SIZE];
        int front = 0, rear = 0;
        BiTree cur = T;
        if (cur) Queue[rear++] = cur; //根指针非空，则进队
        while (front != rear) {
                cur = Queue[front++]; //队首结点出队
                visit(cur);
                if (cur->lchild) Queue[rear++] = cur->lchild;
                if (cur->rchild) Queue[rear++] = cur->rchild;
        }
        return OK;
}

/* 辅助函数：带空子树标记的先序构造二叉树（choice==1）
   说明：
   - 当 definition[*index].key == -1 时，表示输入结束（该标记不构造成结点）。
   - 当 definition[*index].key == 0 时，表示空子树，将 *T 置为 NULL。 */
```

```
status CreateBiTree_Recursive(BiTree* T, TElemType definition[], int*
↪  index) {
        if (definition[*index].key == -1) {  // 终止标记
                (*index)++;
                return OK;
        }
        if (definition[*index].key == 0) {   // 空子树标记
                *T = NULL;
                (*index)++;
                return OK;
        }
        *T = (BiTree)malloc(sizeof(BiTNode));
        if (*T == NULL) {
                return OVERFLOW;
        }
        (*T)->data = definition[*index];
        (*index)++;
        CreateBiTree_Recursive(&((*T)->lchild), definition, index);
        CreateBiTree_Recursive(&((*T)->rchild), definition, index);
        return OK;
}

/* 辅助函数: 根据前序和中序序列构造二叉树 (choice==2)
   参数:
           pre   : 前序序列 (长度为 n)
           in    : 中序序列 (长度为 n)
           n     : 序列中结点数量
 */
status CreateBiTree_PreIn(BiTree* T, TElemType pre[], TElemType in[], int
↪  n) {
        if (n <= 0) {
                *T = NULL;
                return OK;
        }
        int i;
```

```
        *T = (BiTree)malloc(sizeof(BiTNode));
        if (*T == NULL)
                return OVERFLOW;
        (*T)->data = pre[0];
        // 在中序序列中查找根结点位置
        for (i = 0; i < n; i++) {
                if (in[i].key == pre[0].key) break;
        }
        if (i == n) return ERROR;  // 没有找到，数据错误
        // i 为左子树结点数，n-i-1 为右子树结点数
        CreateBiTree_PreIn(&((*T)->lchild), pre + 1, in, i);
        CreateBiTree_PreIn(&((*T)->rchild), pre + 1 + i, in + i + 1, n - i
        ↪  - 1);
        return OK;
}

/* 辅助函数：根据后序和中序序列构造二叉树（choice==3）
   参数：
           post  ：后序序列（长度为 n）
           in    ：中序序列（长度为 n）
           n     ：序列中结点数量
 */
status CreateBiTree_PostIn(BiTree* T, TElemType post[], TElemType in[], int
↪  n) {
        if (n <= 0) {
                *T = NULL;
                return OK;
        }
        int i;
        *T = (BiTree)malloc(sizeof(BiTNode));
        if (*T == NULL)
                return OVERFLOW;
        // 后序序列最后一个结点为根
        (*T)->data = post[n - 1];
        // 在中序序列中查找根的位置
```

```
        for (i = 0; i < n; i++) {
                if (in[i].key == post[n - 1].key)
                        break;
        }
        if (i == n) return ERROR;  // 数据错误
        // 左子树大小为 i，右子树大小为 n - i - 1
        CreateBiTree_PostIn(&((*T)->lchild), post, in, i);
        CreateBiTree_PostIn(&((*T)->rchild), post + i, in + i + 1, n - i -
        ↪  1);
        return OK;
}

/* 主创建二叉树函数，根据 choice 的值调用相应创建方法
   choice==1: 根据带空子树标记的先序遍历序列创建二叉树，
                        此时 definition 数组的格式中包含空子树标记 (key==0)
                        ↪  和终止标记 (key==-1)。
   choice==2: 根据前序 + 中序序列建立二叉树，
                        定义约定：definition[0].key 保存结点数 n，
                        接下来连续 n 个元素为前序序列，
                        紧跟着 n 个元素为中序序列。
   choice==3: 根据后序 + 中序序列建立二叉树，
                        定义约定同上，只不过前一部分为后序序列。
*/
status CreateBiTree(BiTree* T, TElemType definition[], int choice) {
        if (choice == 1) {
                int index = 0;
                return CreateBiTree_Recursive(T, definition, &index);
        }
        else if (choice == 2) {
                // 按约定，definition[0].key 为结点数 n，
                // 后续前 n 个为前序序列，紧跟着 n 个为中序序列
                int n = definition[0].key;
                return CreateBiTree_PreIn(T, &definition[1], &definition[1
                ↪  + n], n);
        }
```

```
        else if (choice == 3) {
                // 同样, definition[0].key 为结点数 n,
                // 后续前 n 个为后序序列, 紧跟着 n 个为中序序列
                int n = definition[0].key;
                return CreateBiTree_PostIn(T, &definition[1], &definition[1
                ↪  + n], n);
        }
        else {
                return ERROR;  // 不支持的输入方式
        }
}

status ClearBiTree(BiTree* T)
//将二叉树 T 置空, 并释放所有结点的空间
{
        // 如果 T 本身为 NULL, 则返回错误
        if (T == NULL)
                return ERROR;

        // 如果二叉树不为空, 递归清空左、右子树
        if (*T != NULL) {
                ClearBiTree(&((*T)->lchild));  // 清空左子树
                ClearBiTree(&((*T)->rchild));  // 清空右子树

                free(*T); // 释放当前节点的内存, 并置空指针, 避免野指针风险
                *T = NULL;
        }
        return OK;
}
int BiTreeDepth(BiTree T)
//求二叉树 T 深度并返回深度值
{
        if (T == NULL) return 0;
        else {
                int left_depth = BiTreeDepth(T->lchild);
```

```
                int right_depth = BiTreeDepth(T->rchild);
                // 当前树的深度为左右子树中较大者 +1
                return (left_depth > right_depth ? left_depth :
                ↪  right_depth) + 1;
        }
}
BiTNode* LocateNode(BiTree T, KeyType e)
//e 是和 T 中结点关键字类型相同的给定值；根据 e 查找符合条件的结点，返回该结点
↪  指针，如无关键字为 e 的结点，返回 NULL。
{
        if (T == NULL) return NULL;
        if (T->data.key == e) return T;
        BiTNode* p = LocateNode(T->lchild, e);// 先在左子树中查找
        if (p != NULL)
                return p;  // 找到则返回

        return LocateNode(T->rchild, e); // 若左子树中未找到，则继续在右子树
        ↪  中查找
}
status Assign(BiTree* T, KeyType e, TElemType value)
/*e 是和 T 中结点关键字类型相同的给定值；查找结点关键字等于 e 的结点，将该结点
↪  值修改成 value，
返回 OK（要求结点关键字保持唯一性）。如果查找失败，返回 ERROR。*/
{
        BiTNode* p = LocateNode(*T, e);
        if (p == NULL) return ERROR;
        // 如果欲更改的关键字不等于原来的关键字，则需检查树中是否已存在该新关
        ↪  键字
        if (value.key != p->data.key) {
                BiTNode* q = LocateNode(*T, value.key);
                // 如果找到的结点不是 p 本身，说明新关键字已存在
                if (q != NULL)
                        return ERROR;
        }
        p->data = value;
```

```
        return OK;
}
BiTNode* GetSibling(BiTree T, KeyType e)
/*e 是和 T 中结点关键字类型相同的给定值；查找结点关键字等于 e 的结点的兄弟结点，
↪  返回其兄弟结点指针。如果查找失败，返回 NULL。*/
{
        if (T == NULL) return NULL;

        // 如果左子树中的结点满足条件，则返回右子树指针
        if (T->lchild != NULL && T->lchild->data.key == e)
                return T->rchild;  // 若无右子树则返回 NULL

        // 如果右子树中的结点满足条件，则返回左子树指针
        if (T->rchild != NULL && T->rchild->data.key == e)
                return T->lchild;  // 若无左子树则返回 NULL

        // 先在左子树中递归查找
        BiTNode* sibling = GetSibling(T->lchild, e);
        if (sibling != NULL)
                return sibling;

        // 若左子树没有，再在右子树中递归查找
        return GetSibling(T->rchild, e);
}
status InsertNode(BiTree* T, KeyType e, int LR, TElemType c)
/*e 是和 T 中结点关键字类型相同的给定值，LR 为 0 或 1，c 是待插入结点；根据 LR
↪  为 0 或者 1，插入结点 c 到 T 中，
作为关键字为 e 的结点的左或右孩子结点，结点 e 的原有左子树或右子树则为结点 c
↪  的右子树，返回 OK。如果插
入失败，返回 ERROR。特别地，当 LR 为-1 时，作为根结点插入，原根结点作为 c 的右
↪  子树。*/
{
        if (T == NULL) return ERROR;

        /* 在整个树中查找是否已存在待插入结点 c 的关键字，关键字必须唯一 */
```

```
if (LocateNode(*T, c.key) != NULL) {
        return ERROR;  // 重复关键字，插入失败
}

// 特别地，LR 为 -1 时作为根结点插入，新结点的右子树挂原根
if (LR == -1) {
        BiTNode* newNode = (BiTNode*)malloc(sizeof(BiTNode));
        if (newNode == NULL)
                return ERROR;
        newNode->data = c;
        newNode->lchild = NULL;
        newNode->rchild = *T;  // 原根结点作为新结点的右子树
        *T = newNode;          // 修改根指针
        return OK;
}

// LR 为 0 或 1 的情况：先查找关键字为 e 的结点
BiTNode* parent = LocateNode(*T, e);
if (parent == NULL)
        return ERROR;

BiTNode* newNode = (BiTNode*)malloc(sizeof(BiTNode));
if (newNode == NULL)
        return ERROR;
newNode->data = c;
newNode->lchild = NULL;   // 默认新结点的左子树为空

if (LR == 0) {
        // 将新结点插入为 parent 的左孩子，新结点的右子树挂上原来的左
        ↪ 子树
        newNode->rchild = parent->lchild;
        parent->lchild = newNode;
}
else if (LR == 1) {
```

```
                // 将新结点插入为 parent 的右孩子，新结点的右子树挂上原来的右
                ↪ 子树
                newNode->rchild = parent->rchild;
                parent->rchild = newNode;
        }
        else {
                // 如果 LR 不是 -1、0 或 1，视为非法参数
                free(newNode);
                return ERROR;
        }
        return OK;
}
BiTNode* FindNodeWithParent(BiTree T, KeyType e, BiTNode* parent, BiTNode**
↪ outParent)
/* 辅助函数：查找关键字为 e 的结点，同时返回该结点的父结点。
 * 参数说明：
 *   T         : 当前子树的根结点
 *   e         : 要查找的关键字
 *   parent    : 当前结点的父结点（在递归调用中传入）
 *   outParent: 输出参数，用来返回找到结点的父结点（若结点为根则置为 NULL）
 */
{
        if (T == NULL) return NULL;
        if (T->data.key == e) {
                *outParent = parent;
                return T;
        }
        BiTNode* found = FindNodeWithParent(T->lchild, e, T, outParent);
        ↪ //依旧是先在左子树里面找，找不到再到右子树里面找
        if (found != NULL) return found;
        return FindNodeWithParent(T->rchild, e, T, outParent);
}
status DeleteNode(BiTree* T, KeyType e)
/*e 是和 T 中结点关键字类型相同的给定值。删除 T 中关键字为 e 的结点；同时，根据
↪ 该被删结点的度进行讨论：
```

```
        1. 如关键字为 e 的结点度为 0，删除即可；
        2. 如关键字为 e 的结点度为 1，用关键字为 e 的结点孩子代替被删除的 e 位
        ↪ 置；
        3. 如关键字为 e 的结点度为 2，用 e 的左子树（简称为 LC）代替被删除的 e
        ↪ 位置，将 e 的右子树（简称为 RC）作为 LC 中最右节点的右子树。
成功删除结点后返回 OK，否则返回 ERROR。*/
{
        if (T == NULL || *T == NULL) return ERROR;

        BiTNode* parent = NULL;
        BiTNode* p = FindNodeWithParent(*T, e, NULL, &parent);
        if (p == NULL) return ERROR; //未找到关键字为 e 的结点

        if (p->lchild == NULL && p->rchild == NULL) {  //情况 1：被删结点为
        ↪ 叶结点（度为 0）
                if (parent == NULL) *T = NULL; //被删结点为根结点
                else {
                        if (parent->lchild == p) parent->lchild = NULL;
                        else parent->rchild = NULL;
                }
                free(p);
                return OK;
        }
        else if (p->lchild == NULL || p->rchild == NULL) {  //情况 2：被删结
        ↪ 点度为 1
                BiTNode* child = (p->lchild != NULL) ? p->lchild :
                ↪ p->rchild;
                if (parent == NULL) *T = child; //被删结点为根结点
                else {
                        if (parent->lchild == p) parent->lchild = child;
                        else parent->rchild = child;
                }
                free(p);
                return OK;
        }
```

```
        else {  //情况 3: 被删结点度为 2
                // 令 LC 为 p 的左子树, RC 为 p 的右子树
                BiTNode* LC = p->lchild;
                BiTNode* RC = p->rchild;
                // 在 LC 中找到最右侧的结点
                BiTNode* mostright = LC;
                while (mostright->rchild != NULL)
                        mostright = mostright->rchild;
                // 将 RC 接到 mostright 的右孩子上
                mostright->rchild = RC;
                // 用 LC 代替 p 在父结点处的位置
                if (parent == NULL) {  // p 为根结点
                        *T = LC;
                }
                else {
                        if (parent->lchild == p)
                                parent->lchild = LC;
                        else
                                parent->rchild = LC;
                }
                free(p);
                return OK;
        }
}
status DestroyBiTree(BiTree* T)
{
        if (T == NULL) return ERROR;  // 如果传入的是空指针, 则返回错误

        if (*T != NULL) {
                // 先销毁左子树和右子树, 采用后序遍历
                DestroyBiTree(&((*T)->lchild));
                DestroyBiTree(&((*T)->rchild));
                // 释放当前结点
                free(*T);
                *T = NULL;  // 防止悬挂指针
```

```
        }
        return OK;
}
status BiTreeEmpty(BiTree T)
//二叉树判空
{
        return (T == NULL) ? TRUE : FALSE;
}

int MaxPathSum(BiTree T)
//该函数返回从根节点出发到任一叶子结点的路径中，各结点 data.key 值之和的最大值。
↪
{
        if (T == NULL) return ERROR;

        // 如果是叶子节点，直接返回该结点的 key 值
        if (T->lchild == NULL && T->rchild == NULL) return T->data.key;

        // 如果只有一边子树存在，则必经该分支
        if (T->lchild == NULL)
                return T->data.key + MaxPathSum(T->rchild);
        if (T->rchild == NULL)
                return T->data.key + MaxPathSum(T->lchild);

        // 两边都存在，取最大者
        int leftSum = MaxPathSum(T->lchild);
        int rightSum = MaxPathSum(T->rchild);
        return T->data.key + (leftSum > rightSum ? leftSum : rightSum);
}
BiTNode* LowestCommonAncestor(BiTree T, KeyType e1, KeyType e2)
//最近公共祖先：求解二叉树中两个给定关键字 (e1 和 e2) 所在结点的最近公共祖先。
{
        if (T == NULL) return NULL;
```

```
if (LocateNode(T, e1) == NULL || LocateNode(T, e2) == NULL) return
↪  NULL;

// 若当前结点已是 e1 或 e2，则返回当前结点
if (T->data.key == e1 || T->data.key == e2) return T;

BiTree left = LowestCommonAncestor(T->lchild, e1, e2);
BiTree right = LowestCommonAncestor(T->rchild, e1, e2);
/*
场景 1：目标节点均不在对应的子树中
            如果 T->lchild 的子树中既没有 e1 也没有 e2：递归搜索
            ↪  会一路走到底，发现空子树后，如果 T==NULL 则立刻返
            ↪  回 NULL。因此 left 返回 NULL。
        类似地，如果 T->rchild 的子树中既没有 e1 也没有 e2：则 right
        ↪  也返回 NULL。
场景 2：仅一侧子树找到目标节点（或该侧已经计算出该子树内的最近公共祖先）
↪
            例如：在 T 的左子树中找到了 e1 或 e2，或即使在左子树中同时
            ↪  出现了两目标，但它们的最近公共祖先就在左子树中（例如后
            ↪  续返回的结点），
        那么 left 将返回该非空指针。而如果在右子树中未找到目标，则
        ↪  right == NULL。反之：若右子树中找到了目标（或其某一侧已经
        ↪  形成了局部公
        共祖先），而左子树找不到，则 right 返回非空，而 left == NULL。
场景 3：两侧子树均找到了目标
            当 T 的左子树中搜索到 e1 而右子树中搜索到 e2（或者相反），此
            ↪  时 left != NULL 且 right != NULL。说明 e1 和 e2 分别
            ↪  位于当前节点 T 的左右
        两个不同方向，那么当前节点 T 就是这两个目标的最近公共祖先。（此
        ↪  时递归调用分别返回的是目标节点本身，或已经在某个子树中由递
        ↪  归判断出
        的局部祖先。）
*/

// 如果左右子树均找到了目标，则当前结点即为最近公共祖先
```

```
        if (left != NULL && right != NULL) return T;

        // 否则将非空的一侧向上回传，对于叶子结点，左右都为 NULL，所有返回 NULL
        return left != NULL ? left : right;
}
void InvertTree(BiTree T)
//二叉树翻转：将树中所有结点的左右子树互换
{
        if (T == NULL) return;

        // 交换左右子树
        BiTree temp = T->lchild;
        T->lchild = T->rchild;
        T->rchild = temp;

        // 递归翻转左右子树
        InvertTree(T->lchild);
        InvertTree(T->rchild);
}
// 辅助函数：先序遍历写入（按照标准格式输出：有效节点输出 key 和 others，空子树
↪  输出 "0 null"）
void SaveBiTree_PreOrder(BiTree T, FILE* fp)
{
        if (T == NULL) {
                // 空子树标记
                fprintf(fp, "0 null ");
                return;
        }
        // 输出当前有效结点：关键字与其它信息
        fprintf(fp, "%d %s ", T->data.key, T->data.others);
        // 递归写入左子树和右子树
        SaveBiTree_PreOrder(T->lchild, fp);
        SaveBiTree_PreOrder(T->rchild, fp);
}
status SaveBiTree(BiTree T, char FileName[])
```

```
{
        FILE* fp = fopen(FileName, "w");
        if (fp == NULL)
                return ERROR;
        // 先序遍历写入二叉树到文件，空子树输出 "0 null"
        SaveBiTree_PreOrder(T, fp);
        // 写入结束标记：-1 null
        fprintf(fp, "-1 null ");
        fclose(fp);
        return OK;
}
status LoadBiTree(BiTree* T, char FileName[])
{
        FILE* fp = fopen(FileName, "r");
        if (fp == NULL)
                return ERROR;

        // 使用 definition 数组存放先序定义序列，定义的标准与保存时一致：
        // 有效结点：key > 0，对应的字符串；空子树：key 为 0；输入结束：key
        ↪  为-1
        TElemType definition[MAX_TREE_SIZE];
        int count = 0;
        char token[20];

        // 循环读入标记：每次先读 key（字符串形式），转换为整型
        while (fscanf(fp, "%s", token) == 1)
        {
                int k = atoi(token);
                if (k == -1) {  // 遇到结束标记则停止读取
                        break;
                }
                // 将当前结点信息保存到 definition 数组中
                definition[count].key = k;
                fscanf(fp, "%s", token); // 读取对应的 others 字符串
                strcpy(definition[count].others, token);
```

```
                count++;
        }
        fclose(fp);

        int index = 0;
        // 调用辅助函数，根据 definition 数组构建二叉树
        return CreateBiTree_Recursive(T, definition, &index);
}

/* 多二叉树管理 */
status InitBiTreeManager(BiTreeManager* manager)
{
        if (manager == NULL) return ERROR;
        manager->count = 0;
        return OK;
}
status AddBiTree(BiTreeManager* manager, char name[], BiTree tree)
{
        if (manager == NULL) return ERROR;
        if (manager->count >= MAX_BTREES) return OVERFLOW;  // 管理器空间满

        //检查是否已存在相同名称的二叉树
        for (int i = 0; i < manager->count; i++) {
                if (strcmp(manager->items[i].name, name) == 0) {
                        return ERROR;  // 重名，不能重复添加
                }
        }

        strcpy(manager->items[manager->count].name, name);
        manager->items[manager->count].tree = tree;
        manager->count++;
        return OK;
}
status RemoveBiTree(BiTreeManager* manager, char name[])
{
```

```
        if (manager == NULL) return ERROR;

        int pos = -1;
        for (int i = 0; i < manager->count; i++) {
                if (strcmp(manager->items[i].name, name) == 0) {
                        pos = i;
                        break;
                }
        }
        if (pos == -1) return ERROR; // 没有找到指定名称的二叉树

        // 销毁该二叉树
        if(DestroyBiTree(&(manager->items[pos].tree))==ERROR) return ERROR;

        // 将后面的元素依次前移，以填补空缺
        for (int i = pos; i < manager->count - 1; i++) {
                manager->items[i] = manager->items[i + 1];
        }
        manager->count--;
        return OK;
}
void ListBiTrees(BiTreeManager* manager)
//列出管理中的所有二叉树
{
        if (manager == NULL) return;
        printf(" 当前管理的二叉树列表: \n");
        for (int i = 0; i < manager->count; i++) {
                printf(" 序号: %d, 名称: %s\n", i + 1,
                ↪  manager->items[i].name);
        }
}
BiTree GetBiTree(BiTreeManager* manager, char name[])
//根据二叉树名称返回二叉树指针，如果没找到则返回 NULL
{
        if (manager == NULL) return NULL;
```

```
        for (int i = 0; i < manager->count; i++) {
                if (strcmp(manager->items[i].name, name) == 0) {
                        return manager->items[i].tree;
                }
        }
        return NULL;
}

/* 菜单函数 */
void PrintMenu()
{
        printf("\n  -------二叉树演示系统-------  \n");
        printf("***************************************\n");
        printf("  0. 退出演示系统\n");
        printf("  1. 创建单个二叉树\n");
        printf("  2. 销毁当前二叉树\n");
        printf("  3. 清空当前二叉树\n");
        printf("  4. 判断当前二叉树是否为空\n");
        printf("  5. 获取当前二叉树的深度\n");
        printf("  6. 查找关键字为 e 的结点\n");
        printf("  7. 给关键字为 e 的结点赋值\n");
        printf("  8. 获得关键字为 e 的结点的兄弟结点\n");
        printf("  9. 插入结点\n");
        printf(" 10. 删除结点\n");
        printf(" 11. 前序遍历\n");
        printf(" 12. 中序遍历\n");
        printf(" 13. 后序遍历\n");
        printf(" 14. 层序遍历\n");
        printf(" 15. 附加功能 1: 求最大路径和\n");
        printf(" 16. 附加功能 2: 求最近公共祖先\n");
        printf(" 17. 附加功能 3: 翻转二叉树\n");
        printf(" 18. 附加功能 4: 实现线性表的文件形式保存\n");
        printf(" 19. 附加功能 5: 从文件加载创建二叉树\n");
        printf(" 20. 附加功能 6: 多二叉树管理\n");
        printf("***************************************\n");
```

```
        printf(" 请输入你的选择: ");
}
void PrintSubMenu()
{
        printf("\n--- 多二叉树管理子系统 ---\n");
        printf("  1. 创建新的二叉树\n");
        printf("  2. 显示所有二叉树状态\n");
        printf("  3. 删除某个二叉树\n");
        printf("  4. 操作其中某个二叉树\n");
        printf("  0. 返回主菜单\n");
        printf(" 请输入你的选择: ");
}
void PrintSubSubMenu(char* listname)
{
        printf("\n-----操作二叉树 \"%s\" -----\n", listname);
        printf("  0. 返回子菜单\n");
        printf("  1. 销毁二叉树\n");
        printf("  2. 清空二叉树\n");
        printf("  3. 二叉树判空\n");
        printf("  4. 获取二叉树深度\n");
        printf("  5. 查找关键字为 e 的结点\n");
        printf("  6. 给关键字为 e 的结点赋值\n");
        printf("  7. 获得关键字为 e 的结点的兄弟结点\n");
        printf("  8. 插入结点\n");
        printf("  9. 删除结点\n");
        printf(" 10. 前序遍历\n");
        printf(" 11. 中序遍历\n");
        printf(" 12. 后序遍历\n");
        printf(" 13. 层序遍历\n");
        printf(" 14. 附加功能 1: 求最大路径和\n");
        printf(" 15. 附加功能 2: 求最近公共祖先\n");
        printf(" 16. 附加功能 3: 翻转二叉树\n");
        printf(" 17. 附加功能 4: 实现线性表的文件形式保存\n");
        printf(" 18. 附加功能 5: 从文件加载创建二叉树\n");
        printf(" 请输入您的选择: \n");
```

```
}
```

文件 3：main.c

```
#define _CRT_SECURE_NO_WARNINGS

#include"functions.h"

int main()
{
    system("color F0");
    int choice = 0;
    BiTree currentTree = NULL;      // 当前使用的单棵二叉树
    BiTreeManager manager;          // 多二叉树管理器
    InitBiTreeManager(&manager);    // 初始化多二叉树管理器

    while (1)
    {
        PrintMenu();
        scanf("%d", &choice);
        switch (choice)
        {
        case 0:
            printf(" 退出演示系统。\n");
            exit(0);
            break;
        case 1:
        {
            int choice_;
            printf(" 请选择创建二叉树的方式: \n");
            printf(" 1. 带空子树标记的先序遍历序列\n");
            printf(" 2. 前序和中序遍历序列\n");
            printf(" 3. 后序和中序遍历序列\n");
            printf(" 请输入您的选择 (1/2/3): ");
            scanf("%d", &choice_);
```

```
if (choice_ == 1)
{
    TElemType def[MAX_TREE_SIZE];
    int count = 0;
    printf(" 请输入带空子树标记的先序遍历序列的数据 (格式: key
    ↪  others): \n");
    printf(" 当输入 key 为-1 时表示结束, 输入 0 表示空结点。\n");
    while (1)
    {
        printf(" 请输入结点%d 的数据: ", count + 1);
        scanf("%d %s", &def[count].key, def[count].others);
        if (def[count].key == -1) // 终止标记
            break;
        count++;
        if (count >= MAX_TREE_SIZE)
        {
            printf(" 已达最大输入个数, 停止输入! \n");
            break;
        }
    }
    if (CreateBiTree(&currentTree, def, 1) == OK)
        printf(" 二叉树初始化成功。\n");
    else
        printf(" 二叉树初始化失败! \n");
}
else if (choice_ == 2)
{
    int n;
    printf(" 请输入非空结点总数 n: ");
    scanf("%d", &n);
    TElemType
    ↪  *def=(TElemType*)malloc((1+2*n)*sizeof(TElemType)); //
    ↪  总长度为 1+2*n
    def[0].key = n; // 保存结点数, others 字段可以忽略或设为空
```

```
        printf(" 请输入前序遍历序列 (%d 个结点): \n", n);
        for (int i = 0; i < n; i++)
        {
            printf(" 前序序列第 %d 个结点 (key others): ", i + 1);
            scanf("%d %s", &def[1 + i].key, def[1 + i].others);
        }
        printf(" 请输入中序遍历序列 (%d 个结点): \n", n);
        for (int i = 0; i < n; i++)
        {
            printf(" 中序序列第 %d 个结点 (key others): ", i + 1);
            scanf("%d %s", &def[1 + n + i].key, def[1 + n +
            ↪  i].others);
        }
        if (CreateBiTree(&currentTree, def, 2) == OK)
            printf(" 二叉树初始化成功。\n");
        else
            printf(" 二叉树初始化失败! \n");
        free(def);
    }
    else if (choice_ == 3)
    {
        int n;
        printf(" 请输入非空结点总数 n: ");
        scanf("%d", &n);
        TElemType* def = (TElemType*)malloc((1 + 2 * n) *
        ↪  sizeof(TElemType)); // 总长度为 1+2*n
        def[0].key = n;
        printf(" 请输入后序遍历序列 (%d 个结点): \n", n);
        for (int i = 0; i < n; i++)
        {
            printf(" 后序序列第 %d 个结点 (key others): ", i + 1);
            scanf("%d %s", &def[1 + i].key, def[1 + i].others);
        }
        printf(" 请输入中序遍历序列 (%d 个结点): \n", n);
        for (int i = 0; i < n; i++)
```

```
            {
                printf(" 中序序列第 %d 个结点 (key others): ", i + 1);
                scanf("%d %s", &def[1 + n + i].key, def[1 + n +
                ↪  i].others);
            }
            if (CreateBiTree(&currentTree, def, 3) == OK)
                printf(" 二叉树初始化成功。\n");
            else
                printf(" 二叉树初始化失败! \n");
            free(def);
        }
        else
        {
            printf(" 创建方式输入错误! \n");
        }
    }
    break;
    case 2:
        // 销毁当前二叉树
        if (DestroyBiTree(&currentTree) == OK)
            printf(" 二叉树销毁成功。\n");
        else
            printf(" 二叉树销毁失败! \n");
        break;
    case 3:
        // 清空当前二叉树
        if (ClearBiTree(&currentTree) == OK)
            printf(" 二叉树已清空。\n");
        else
            printf(" 清空二叉树失败! \n");
        break;
    case 4:
        // 判断当前二叉树是否为空
        if (BiTreeEmpty(currentTree))
            printf(" 当前二叉树为空。\n");
```

```
        else
            printf(" 当前二叉树不为空。\n");
        break;
    case 5:
        // 获取当前二叉树深度
        printf(" 当前二叉树深度为: %d\n", BiTreeDepth(currentTree));
        break;
    case 6:
    {
        // 查找关键字为 e 的结点
        KeyType key;
        printf(" 请输入要查找的关键字: ");
        scanf("%d", &key);
        BiTNode* node = LocateNode(currentTree, key);
        if (node != NULL)
            printf(" 找到关键字为 %d 的结点。\n", key);
        else
            printf(" 未找到关键字为 %d 的结点! \n", key);
    }
    break;
    case 7:
    {
        // 给关键字为 e 的结点赋值
        KeyType key;
        TElemType newVal;
        printf(" 请输入目标结点关键字 e: ");
        scanf("%d", &key);
        printf(" 请输入新关键字 key (整数) : ");
        scanf("%d", &newVal.key);
        printf(" 请输入其他信息 others (字符串) : ");
        scanf("%s", newVal.others);
        if (Assign(&currentTree, key, newVal) == OK)
            printf(" 结点赋值成功。\n");
        else
            printf(" 结点赋值失败! \n");
```

```
}
break;
case 8:
{
    // 获得关键字为 e 的结点的兄弟结点
    KeyType key;
    printf(" 请输入要查找兄弟结点的关键字 e: ");
    scanf("%d", &key);
    BiTNode* sibling = GetSibling(currentTree, key);
    if (sibling != NULL)
        printf(" 找到兄弟结点，关键字为: %d\n", sibling->data.key);
    else
        printf(" 查找兄弟结点出错! \n");
}
break;
case 9:
{
    // 插入结点
    KeyType key;
    TElemType c;
    int lr;
    printf(" 请输入要插入位置的父结点关键字 e: ");
    scanf("%d", &key);
    printf(" 请输入要插入结点的关键字 key (整数) : ");
    scanf("%d", &c.key);
    printf(" 请输入要插入结点的其他信息 others (字符串) : ");
    scanf("%s", c.others);
    printf(" 请输入插入方向 (0 表示左子树; 1 表示右子树; -1 表示根节
    ↪  点) : ");
    scanf("%d", &lr);
    if (InsertNode(&currentTree, key, lr, c) == OK)
        printf(" 插入结点成功。\n");
    else
        printf(" 插入结点失败! \n");
}
```

```
break;
case 10:
{
    // 删除结点
    KeyType key;
    printf(" 请输入要删除的结点关键字: ");
    scanf("%d", &key);
    if (DeleteNode(&currentTree, key) == OK)
        printf(" 删除结点成功。\n");
    else
        printf(" 删除结点失败! \n");
}
break;
case 11:
    // 前序遍历
    printf(" 前序遍历结果: \n");
    PreOrderTraverse(currentTree, visit);
    printf("\n");
    break;
case 12:
    // 中序遍历
    printf(" 中序遍历结果: \n");
    InOrderTraverse(currentTree, visit);
    printf("\n");
    break;
case 13:
    // 后序遍历
    printf(" 后序遍历结果: \n");
    PostOrderTraverse(currentTree, visit);
    printf("\n");
    break;
case 14:
    // 层序遍历
    printf(" 层序遍历结果: \n");
    LevelOrderTraverse(currentTree, visit);
```

```
        printf("\n");
        break;
    case 15:
        // 附加功能 1: 求最大路径和
        printf(" 最大路径和为: %d\n", MaxPathSum(currentTree));
        break;
    case 16:
    {
        // 附加功能 2: 求最近公共祖先
        KeyType e1, e2;
        printf(" 请输入两个结点的关键字: ");
        scanf("%d %d", &e1, &e2);
        BiTNode* lca = LowestCommonAncestor(currentTree, e1, e2);
        if (lca != NULL)
            printf(" 最近公共祖先的关键字为: %d\n", lca->data.key);
        else
            printf(" 未找到最近公共祖先! \n");
    }
    break;
    case 17:
        // 附加功能 3: 翻转二叉树
        InvertTree(currentTree);
        printf(" 二叉树已翻转。\n");
        break;
    case 18:
    {
        // 附加功能 4: 实现线性表的文件形式保存
        char filename[100];
        printf(" 请输入保存文件名: ");
        scanf("%s", filename);
        if (SaveBiTree(currentTree, filename) == OK)
            printf(" 保存成功。\n");
        else
            printf(" 保存失败! \n");
    }
```

```
break;
case 19:
{
    // 附加功能 5: 从文件加载创建二叉树
    char filename[100];
    printf(" 请输入加载文件名: ");
    scanf("%s", filename);
    if (LoadBiTree(&currentTree, filename) == OK)
        printf(" 加载成功。\n");
    else
        printf(" 加载失败! \n");
}
break;
case 20:
{
    // 附加功能 6: 多二叉树管理子系统
    int subChoice = 0;
    do
    {
        PrintSubMenu();
        scanf("%d", &subChoice);
        switch (subChoice)
        {
        case 0:
            printf(" 返回主菜单...\n");
            break;
        case 1:
        {
            // 创建新的二叉树并添加到管理器
            int createChoice;
            printf(" 请选择创建二叉树的方式: \n");
            printf(" 1. 带空子树标记的先序遍历序列\n");
            printf(" 2. 前序和中序遍历序列\n");
            printf(" 3. 后序和中序遍历序列\n");
            printf(" 4. 仅初始化\n");
```

```
printf(" 请输入您的选择 (1/2/3/4): ");
scanf("%d", &createChoice);

BiTree newTree = NULL;
if (createChoice == 1)
{
    TElemType def[MAX_TREE_SIZE];
    int count = 0;
    printf(" 请输入带空子树标记的先序遍历序列的数据 (格式:
    ↪ key others), \n");
    printf(" 输入 key 为 0 表示空结点, 输入 key 为-1 表
    ↪ 示序列结束。\n");
    while (1)
    {
        printf(" 请输入结点 %d 的数据: ", count + 1);
        scanf("%d %s", &def[count].key,
        ↪ def[count].others);
        if (def[count].key == -1)
            break;
        count++;
        if (count >= MAX_TREE_SIZE)
        {
            printf(" 已达最大输入个数, 停止输入! \n");
            break;
        }
    }
    if (CreateBiTree(&newTree, def, 1) == OK)
        printf(" 二叉树创建成功。\n");
    else
        printf(" 二叉树创建失败! \n");
}
else if (createChoice == 2)
{
    int n;
    printf(" 请输入非空结点的总数 n: ");
```

```
        scanf("%d", &n);
        TElemType* def = (TElemType*)malloc((1 + 2 * n) *
        ↪  sizeof(TElemType)); // 总长度为 1+2*n
        def[0].key = n;  // 保存结点数 (others 可忽略)
        printf(" 请输入前序遍历序列的 %d 个结点数据 (key
        ↪  others):\n", n);
        for (int i = 0; i < n; i++)
        {
            printf(" 前序序列第 %d 个结点: ", i + 1);
            scanf("%d %s", &def[1 + i].key, def[1 +
            ↪  i].others);
        }
        printf(" 请输入中序遍历序列的 %d 个结点数据 (key
        ↪  others):\n", n);
        for (int i = 0; i < n; i++)
        {
            printf(" 中序序列第 %d 个结点: ", i + 1);
            scanf("%d %s", &def[1 + n + i].key, def[1 + n +
            ↪  i].others);
        }
        if (CreateBiTree(&newTree, def, 2) == OK)
            printf(" 二叉树创建成功。\n");
        else
            printf(" 二叉树创建失败! \n");
        free(def);
    }
    else if (createChoice == 3)
    {
        int n;
        printf(" 请输入非空结点的总数 n: ");
        scanf("%d", &n);
        TElemType* def = (TElemType*)malloc((1 + 2 * n) *
        ↪  sizeof(TElemType)); // 总长度为 1+2*n
        def[0].key = n;
```

```
        printf(" 请输入后序遍历序列的 %d 个结点数据 (key
        ↪  others):\n", n);
        for (int i = 0; i < n; i++)
        {
            printf(" 后序序列第 %d 个结点: ", i + 1);
            scanf("%d %s", &def[1 + i].key, def[1 +
            ↪  i].others);
        }
        printf(" 请输入中序遍历序列的 %d 个结点数据 (key
        ↪  others):\n", n);
        for (int i = 0; i < n; i++)
        {
            printf(" 中序序列第 %d 个结点: ", i + 1);
            scanf("%d %s", &def[1 + n + i].key, def[1 + n +
            ↪  i].others);
        }
        if (CreateBiTree(&newTree, def, 3) == OK)
            printf(" 二叉树创建成功。\n");
        else
            printf(" 二叉树创建失败! \n");
        free(def);
    }
    else if (createChoice == 4)
    {
        if (InitBiTree(&newTree) == OK)
            printf(" 初始化二叉树成功\n");
        else
            printf(" 初始化二叉树失败! \n");
    }
    else
    {
        printf(" 创建方式输入错误! \n");
        break;
    }
```

```
        // 完成新二叉树的创建后，询问用户为该二叉树命名，然后添加
        ↪ 到管理器中
        char name[50];
        printf(" 请输入新二叉树的名称: ");
        scanf("%s", name);
        if (AddBiTree(&manager, name, newTree) == OK)
            printf(" 二叉树添加到管理器成功。\n");
        else
            printf(" 二叉树添加失败，可能存在重名或管理器已满。
            ↪ \n");
    }
    break;
    case 2:
        // 显示管理器中所有二叉树的状态
        ListBiTrees(&manager);
        break;
    case 3:
    {
        //删除某个二叉树
        char delName[50];
        printf(" 请输入要删除的二叉树名称: ");
        scanf("%s", delName);
        if (RemoveBiTree(&manager, delName) == OK)
            printf(" 删除成功! \n");
        else
            printf(" 删除失败! \n");
        break;
    }
    case 4:
    {
        // 选择某个二叉树进行操作
        char targetName[50];
        printf(" 请输入要操作的二叉树名称: ");
        scanf("%s", targetName);
        BiTree targetTree = GetBiTree(&manager, targetName);
```

```
if (targetTree == NULL)
{
    printf(" 二叉树不存在! \n");
}
else
{
    int subSubChoice = 0;
    do
    {
        PrintSubSubMenu(targetName);
        scanf("%d", &subSubChoice);
        switch (subSubChoice)
        {
        case 0:
            printf(" 返回多二叉树管理子菜单...\n");
            break;
        case 1:
            if (DestroyBiTree(&targetTree) == OK)
                printf(" 销毁二叉树成功。\n");
            else
                printf(" 销毁失败! \n");
            break;
        case 2:
            if (ClearBiTree(&targetTree) == OK)
                printf(" 清空二叉树成功。\n");
            else
                printf(" 清空失败! \n");
            break;
        case 3:
            if (BiTreeEmpty(targetTree))
                printf(" 二叉树为空。\n");
            else
                printf(" 二叉树不为空。\n");
            break;
        case 4:
```

```
        printf(" 二叉树深度为: %d\n",
        ↪  BiTreeDepth(targetTree));
        break;
    case 5:
    {
        KeyType key;
        printf(" 请输入查找结点的关键字: ");
        scanf("%d", &key);
        BiTNode* res = LocateNode(targetTree, key);
        if (res)
            printf(" 找到关键字为 %d 的结点\n",
            ↪  res->data.key);
        else
            printf(" 未找到该结点! \n");
    }
    break;
    case 6:
    {
        KeyType key;
        TElemType newVal;
        printf(" 请输入结点关键字 e: ");
        scanf("%d", &key);
        printf(" 请输入新关键字 key (整数) : ");
        scanf("%d", &newVal.key);
        printf(" 请输入其他信息 others (字符串) : ");
        scanf("%s", newVal.others);
        if (Assign(&targetTree, key, newVal) == OK)
            printf(" 赋值成功。\n");
        else
            printf(" 赋值失败! \n");
    }
    break;
    case 7:
    {
        KeyType key;
```

```
            printf(" 请输入查找兄弟结点的关键字 e: ");
            scanf("%d", &key);
            BiTNode* sib = GetSibling(targetTree, key);
            if (sib)
                printf(" 兄弟结点关键字为: %d\n",
                ↪ sib->data.key);
            else
                printf(" 查找兄弟结点出错! \n");
        }
        break;
        case 8:
        {
            KeyType key;
            TElemType nodeVal;
            int lr;
            printf(" 请输入要插入位置的父结点关键字 e:
            ↪ ");
            scanf("%d", &key);
            printf(" 请输入要插入结点的关键字 key (整数)
            ↪ : ");
            scanf("%d", &nodeVal.key);
            printf(" 请输入要插入结点的其他信息 others
            ↪ (字符串) : ");
            scanf("%s", nodeVal.others);
            printf(" 请输入插入方向 (0 表示左子树; 1 表示
            ↪ 右子树; -1 表示根节点) : ");
            scanf("%d", &lr);
            if (InsertNode(&targetTree, key, lr,
            ↪ nodeVal) == OK)
                printf(" 插入成功。\n");
            else
                printf(" 插入失败! \n");
        }
        break;
        case 9:
```

```
{
    KeyType key;
    printf(" 请输入要删除结点的关键字: ");
    scanf("%d", &key);
    if (DeleteNode(&targetTree, key) == OK)
        printf(" 删除成功。\n");
    else
        printf(" 删除失败! \n");
}
break;
case 10:
    printf(" 前序遍历结果: \n");
    PreOrderTraverse(targetTree, visit);
    printf("\n");
    break;
case 11:
    printf(" 中序遍历结果: \n");
    InOrderTraverse(targetTree, visit);
    printf("\n");
    break;
case 12:
    printf(" 后序遍历结果: \n");
    PostOrderTraverse(targetTree, visit);
    printf("\n");
    break;
case 13:
    printf(" 层序遍历结果: \n");
    LevelOrderTraverse(targetTree, visit);
    printf("\n");
    break;
case 14:
    printf(" 最大路径和为: %d\n",
    ↪ MaxPathSum(targetTree));
    break;
case 15:
```

```
{
    KeyType a, b;
    printf(" 请输入两个结点的关键字: ");
    scanf("%d %d", &a, &b);
    BiTNode* lca =
    ↪ LowestCommonAncestor(targetTree, a, b);
    if (lca)
        printf(" 最近公共祖先为: %d\n",
        ↪ lca->data.key);
    else
        printf(" 未找到最近公共祖先! \n");
}
break;
case 16:
    InvertTree(targetTree);
    printf(" 二叉树已翻转。\n");
    break;
case 17:
{
    char fname[100];
    printf(" 请输入保存文件名: ");
    scanf("%s", fname);
    if (SaveBiTree(targetTree, fname) == OK)
        printf(" 保存成功。\n");
    else
        printf(" 保存失败! \n");
}
break;
case 18:
{
    char fname[100];
    printf(" 请输入加载文件名: ");
    scanf("%s", fname);
    if (LoadBiTree(&targetTree, fname) == OK)
        printf(" 加载成功。\n");
```

```
                            else
                                printf(" 加载失败! \n");
                        }
                        break;
                        default:
                            printf(" 无效选择, 请输入正确的选项! \n");
                            break;
                        }
                    } while (subSubChoice != 0);
                }
            }
            break;
            default:
                printf(" 无效选择, 请输入正确的选项! \n");
                break;
            }
        } while (subChoice != 0);
    }
    break;
    default:
        printf(" 无效选择, 请输入正确的选项! \n");
        break;
    }
}
    return 0;
}
```

附录 D 基于邻接表图实现的源程序

文件 1：function.h

```
#pragma once
#define _CRT_SECURE_NO_WARNINGS

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

#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
#define MAX_VERTEX_NUM 20
#define INF 1000000    //定义的无穷大
#define MAX_GRAPHS 20
#define NAME_LEN 50    // 每个图的名称最多 50 个字符

typedef int status;
typedef int KeyType;
typedef enum { DG, DN, UDG, UDN } GraphKind; //有向图，有向网，无向图，无向网

typedef struct {                    //顶点类型定义
        KeyType key;                    //顶点关键字
        char others[20];
} VertexType;

typedef struct ArcNode {            //表结点类型定义
        int adjvex;                     //顶点位置编号，通过这个索引可以找到某点
        ↪  的关键字
        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 {
        ALGraph* graphs[MAX_GRAPHS];  //存放多个图的指针
        char names[MAX_GRAPHS][NAME_LEN];
        int count;                      //当前图的数量
} GraphManager;

status CreateGraph(ALGraph* G, VertexType V[], KeyType VR[][2]); //函数 1:
↪  创建无向图
status DestroyGraph(ALGraph* G); //函数 2: 销毁图
int LocateVex(ALGraph G, KeyType u); //函数 3: 查找顶点
status PutVex(ALGraph* G, KeyType u, VertexType value); //函数 4: 顶点赋值
int FirstAdjVex(ALGraph G, KeyType u); //函数 5: 获得第一邻接点
int NextAdjVex(ALGraph G, KeyType v, KeyType w); //函数 6: 获得下一邻接点
status InsertVex(ALGraph* G, VertexType v); //函数 7: 插入顶点
status DeleteVex(ALGraph* G, KeyType v); //函数 8: 删除顶点
status InsertArc(ALGraph* G, KeyType v, KeyType w); //函数 9: 插入弧
status DeleteArc(ALGraph* G, KeyType v, KeyType w); //函数 10: 删除弧
void visit(VertexType); //访问函数
void DFS(ALGraph* G, int i, int visited[], void (*visit)(VertexType)); //深
↪  度优先搜索遍历辅助函数
status DFSTraverse(ALGraph* G, void (*visit)(VertexType)); //函数 11: 深度优
↪  先搜索遍历
```

```
status BFSTraverse(ALGraph* G, void (*visit)(VertexType)); //函数 12: 广度优
↪ 先搜索遍历

int* VerticesSetLessThanK(ALGraph* G, int v, int k, int* pSize); //附加功能
↪ 1: 求距离小于 k 的顶点集合
int ShortestPathLength(ALGraph* G, int v, int w); //附加功能 2: 求顶点间最短
↪ 路径和长度
void dummyVisit(VertexType v);
int ConnectedComponentsNums(ALGraph* G); //附加功能 3: 求图的连通分量
status SaveGraph(ALGraph G, char FileName[]); //附加功能 4: 实现图的文件形式
↪ 保存 (?)
status LoadGraph(ALGraph* G, char FileName[]); //读入文件中的结点数据创建无
↪ 向图的邻接表

/* 附加功能 5: 多个图管理 */
void InitGraphManager(GraphManager* gm); //初始化多图管理器
status AddGraph(GraphManager* gm, ALGraph* G, const char name[]); //加入新
↪ 的图
status DeleteGraph(GraphManager* gm, const char name[]);  //删除图
void DisplayGraphManager(GraphManager* gm); //显示所有图的状态
ALGraph* GetGraphByName(GraphManager* gm, const char name[]); //选择某个图
↪ 操作

/* 菜单函数 */
void PrintMenu();
void PrintSubMenu();
void PrintSubSubMenu(char* listname);
```

文件 2：functions.c

```
#define _CRT_SECURE_NO_WARNINGS

#include"functions.h"

status CreateGraph(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*/
{
    int vexnum = 0;
    int flag = 0;

    for (; vexnum < MAX_VERTEX_NUM + 1; vexnum++) {   //循环确定顶点个数
    ↪  vexnum
        if (V[vexnum].key == -1) {
            flag = 1;
            break;
        }
    }
    //if (vexnum <= 0 || (vexnum < MAX_VERTEX_NUM && flag == 0)) return
    ↪  ERROR;
    if (vexnum <= 0 || flag == 0) {
        printf(" 顶点数不符合要求! \n");
        return ERROR;
    }

    for (int i = 0; i < vexnum; i++) {  //循环检查关键字是否重复
        for (int j = i + 1; j < vexnum; j++)
        {
            if (V[i].key == V[j].key)
            {
                printf(" 关键字重复! \n");
                return ERROR;
            }
        }
    }

    G->vexnum = vexnum;
    G->arcnum = 0;
    G->kind = UDG;
```

```
for (int i = 0; i < vexnum; i++) {  //初始化顶点数组：复制顶点信息，同时
↪  将邻接链表头置为 NULL
    G->vertices[i].data = V[i];
    G->vertices[i].firstarc = NULL;
}

int k = 0;
int edgeCount = 0; // 记录边数
while (1) {
    if (VR[k][0] == -1 && VR[k][1] == -1) break;

    int key_u = VR[k][0];
    int key_v = VR[k][1];
    int idx_u = -1, idx_v = -1; //分别为 key_u 和 key_v 在图中对应的下标

    /* 查找关键字 key_u 和 key_v 在图中对应的下标 */
    for (int i = 0; i < vexnum; i++){
        if (G->vertices[i].data.key == key_u) {
            idx_u = i;
            break;
        }
    }
    for (int i = 0; i < vexnum; i++){
        if (G->vertices[i].data.key == key_v){
            idx_v = i;
            break;
        }
    }
    if (idx_u == -1 || idx_v == -1) {
        printf(" 边的邻点关键字输入错误! \n");
        return ERROR;
    }

    //对 idx_u 顶点的邻接链表，检查是否已存在指向 idx_v 的边
```

```
ArcNode* p = G->vertices[idx_u].firstarc; //p 是指向下标为 idx_u 的
↪  点的第一条邻边
int duplicate = 0; //0 表示不存在指向 idx_v 的边, 1 表示存在指向
↪  idx_v 的边
while (p) {
    if (p->adjvex == idx_v) {
        duplicate = 1;
        break;
    }
    p = p->nextarc;
}
if (!duplicate) {
    //采用首插法在 idx_u 的邻接链表插入新的边结点
    ArcNode* newNode = (ArcNode*)malloc(sizeof(ArcNode));
    if (!newNode) exit(OVERFLOW);
    newNode->adjvex = idx_v;
    newNode->nextarc = G->vertices[idx_u].firstarc;     //注意采用
    ↪  的是首插法, 更高效!!!
    G->vertices[idx_u].firstarc = newNode;
}

//对 idx_v 顶点的邻接链表, 同样检查是否已存在指向 idx_u 的边
duplicate = 0;
p = G->vertices[idx_v].firstarc;
while (p){
    if (p->adjvex == idx_u){
        duplicate = 1;
        break;
    }
    p = p->nextarc;
}
if (!duplicate) {
    ArcNode* newNode = (ArcNode*)malloc(sizeof(ArcNode));
    if (!newNode) exit(OVERFLOW);
    newNode->adjvex = idx_u;
```

```
            newNode->nextarc = G->vertices[idx_v].firstarc;    //注意采用的
            ↪ 是首插法!!!
            G->vertices[idx_v].firstarc = newNode;
            edgeCount++;  //仅当两侧均未插入时计数一次这条边
        }
        k++;
    }
    G->arcnum = edgeCount;
    return OK;
}
status DestroyGraph(ALGraph* G)
/* 销毁无向图 G, 删除 G 的全部顶点和边 */
{
    if (G == NULL || G->vertices == NULL) return ERROR;

    ArcNode* p, * temp;
    for (int i = 0; i < G->vexnum; i++) {  //遍历每个顶点, 释放邻接链表中所有
    ↪ 动态分配的边结点
        p = G->vertices[i].firstarc;
        while (p != NULL) {
            temp = p;
            p = p->nextarc;
            free(temp);
        }
        //将当前顶点的邻接链表头置为空
        G->vertices[i].firstarc = NULL;
    }
    //销毁后, 将顶点数和弧数置为 0
    G->vexnum = 0;
    G->arcnum = 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;
```

```
{
    if (G.vertices == NULL) return INFEASIBLE;
    int location = -1;
    for (int i = 0; i < G.vexnum; i++) {
        if (G.vertices[i].data.key == u) {
            location = i;
            break;
        }
    }
    return location;
}
status PutVex(ALGraph* G, KeyType oldKey, VertexType value)
{
    if (G == NULL || G->vertices == NULL) return ERROR;

    int count_u = 0;
    int target_idx = -1;

    for (int i = 0; i < G->vexnum; i++) {
        if (G->vertices[i].data.key == oldKey) {
            count_u++;
            target_idx = i;
        }
    }
    if (count_u != 1) return ERROR;

    // 检查新关键字是否已被另一个顶点使用
    for (int i = 0; i < G->vexnum; i++) {
        if (i == target_idx) continue;  // 跳过目标顶点自身
        if (G->vertices[i].data.key == value.key)
            return ERROR;  // 新关键字已存在
    }

    G->vertices[target_idx].data = value;
    return OK;
```

```
}
int FirstAdjVex(ALGraph G, KeyType u)
//u 是和 G 中顶点关键字类型相同的给定值。返回关键字为 u 的顶点第一个邻接顶点位置序号（简称位序），否则返回-1
{
    if (G.vertices == NULL) return -1;
    for (int i = 0; i < G.vexnum; i++) {
        if (G.vertices[i].data.key == u) {
            if (G.vertices[i].firstarc != NULL) {
                return G.vertices[i].firstarc->adjvex;
            }
            else return -1;
        }
    }
    return -1;
}
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）下一个邻接顶点的位序；如果 w 是最后一个邻接顶点，或 v、w 对应顶点不存在，则返回-1。
{
    if (G.vertices == NULL) return -1;
    for (int i = 0; i < G.vexnum; i++) {
        if (G.vertices[i].data.key == v) {
            ArcNode* p = G.vertices[i].firstarc;
            while (p != NULL) {
                if (G.vertices[p->adjvex].data.key == w) {
                    if (p->nextarc != NULL) return p->nextarc->adjvex;
                    else return -1;
                }
                p = p->nextarc;
            }
        }
    }
    return -1;
```

```
}
status InsertVex(ALGraph* G, VertexType v)
//在图 G 中插入新关键字为 v 的顶点（要求关键字具有唯一性，注意判断顶点个数是否已满）。成功返回 OK，否则返回 ERROR。
{
    if (G == NULL || G->vexnum == MAX_VERTEX_NUM) return ERROR;
    for (int i = 0; i < G->vexnum; i++) {
        if (G->vertices[i].data.key == v.key) return ERROR;
    }
    G->vertices[G->vexnum].data = v;
    G->vertices[G->vexnum].firstarc = NULL;
    G->vexnum++;
    return OK;
}
status DeleteVex(ALGraph* G, KeyType v)
//v 是和 G 中顶点关键字类型相同的给定值；操作结果是在图 G 中删除关键字 v 对应的顶点以及相关的弧。成功返回 OK，否则返回 ERROR。
{
    if (G == NULL) return ERROR;

    int v_pos = -1; //查找关键字为 v 的顶点在顶点数组中的位置
    for (int i = 0; i < G->vexnum; i++) {
        if (G->vertices[i].data.key == v) {
            v_pos = i;
            break;
        }
    }
    if (v_pos == -1) return ERROR;

    //删除所有其他顶点邻接表中指向被删除顶点的弧，并调整弧中存储的位序
    for (int i = 0; i < G->vexnum; i++) {
        if (i == v_pos) continue;  // 待删除顶点自己的邻接表，后面单独处理

        ArcNode* p = G->vertices[i].firstarc;
        ArcNode* prev = NULL;
```

```
    while (p != NULL) {
        if (p->adjvex == v_pos) {
            // 找到指向待删除顶点的弧，则删除它
            ArcNode* temp = p;
            if (prev == NULL) {
                // p 为链表头结点
                G->vertices[i].firstarc = p->nextarc;
            }
            else {
                prev->nextarc = p->nextarc;
            }
            p = p->nextarc;
            free(temp);
            G->arcnum--;
        }
        else {
            // 如果该弧的 adjvex 大于 delIndex，由于顶点数组将左移 1 个位
            ↪ 置，因此需要减 1
            if (p->adjvex > v_pos) p->adjvex--;
            prev = p;
            p = p->nextarc;
        }
    }
}

//删除待删除顶点自身邻接表中的所有弧
//由于是无向图，一条边会被两个单链表所记录，故前面删除边的时候已经
↪ G.arcnum--了，所以这里不用再减了
ArcNode* p_del = G->vertices[v_pos].firstarc;
while (p_del != NULL) {
    ArcNode* temp = p_del;
    p_del = p_del->nextarc;
    free(temp);
}
```

```
    // 4. 将待删除顶点在顶点数组中之后的顶点整体向前移动
    for (int i = v_pos; i < G->vexnum - 1; i++) {
        G->vertices[i] = G->vertices[i + 1];
    }
    G->vexnum--;

    return OK;
}
status InsertArc(ALGraph* G, KeyType v, KeyType w)
//v、w 是和 G 中顶点关键字类型相同的给定值；操作结果是在图 G 中增加弧<v,w>。成
↪  功返回 OK，否则返回 ERROR。
{
    if (G == NULL) return ERROR;
    int v_pos = -1, w_pos = -1;
    // 查找顶点 v 和 w 在图中的下标
    for (int i = 0; i < G->vexnum; i++) {
        if (G->vertices[i].data.key == v) {
            v_pos = i;
        }
        if (G->vertices[i].data.key == w) {
            w_pos = i;
        }
    }
    if (v_pos == -1 || w_pos == -1) return ERROR;

    // 检查是否已经存在<v, w> 这条边，避免重复插入
    ArcNode* p = G->vertices[v_pos].firstarc;
    while (p != NULL) {
        if (p->adjvex == w_pos) return ERROR;
        p = p->nextarc;
    }

    // 使用头插法在 v 的邻接表中插入一节点代表边<v, w>
    ArcNode* newarc_vw = (ArcNode*)malloc(sizeof(ArcNode));
    if (newarc_vw == NULL) return OVERFLOW;
```

```
    newarc_vw->adjvex = w_pos;
    newarc_vw->nextarc = G->vertices[v_pos].firstarc;
    G->vertices[v_pos].firstarc = newarc_vw;

    // 同样在 w 的邻接表中插入一节点代表边<w, v>
    ArcNode* newarc_wv = (ArcNode*)malloc(sizeof(ArcNode));
    if (newarc_wv == NULL) {
        // 释放已经分配给 v 的新节点, 防止内存泄漏
        free(newarc_vw);
        return OVERFLOW;
    }
    newarc_wv->adjvex = v_pos;
    newarc_wv->nextarc = G->vertices[w_pos].firstarc;
    G->vertices[w_pos].firstarc = newarc_wv;

    G->arcnum++;
    return OK;
}
status DeleteArc(ALGraph* G, KeyType v, KeyType w)
//v、w 是和 G 中顶点关键字类型相同的给定值; 操作结果是在图 G 中删除弧<v,w>。成
↪  功返回 OK, 否则返回 ERROR。
{
    if (G == NULL) return ERROR;
    int v_pos = -1, w_pos = -1;
    int flag1 = 0, flag2 = 0;
    for (int i = 0; i < G->vexnum; i++) {
        if (G->vertices[i].data.key == v) v_pos = i;
        if (G->vertices[i].data.key == w) w_pos = i;
    }
    if (v_pos == -1 || w_pos == -1) return ERROR;

    ArcNode* p1 = G->vertices[v_pos].firstarc;
    ArcNode* prev1 = NULL;
    while (p1 != NULL) {
        if (p1->adjvex == w_pos) {
```

```
            flag1 = 1;
            if (prev1 == NULL) G->vertices[v_pos].firstarc = p1->nextarc;
            ↪ //删除头结点，要单独讨论
            else prev1->nextarc = p1->nextarc;
            free(p1);
            break;
        }
        prev1 = p1;
        p1 = p1->nextarc;
    }
    ArcNode* p2 = G->vertices[w_pos].firstarc;
    ArcNode* prev2 = NULL;
    while (p2 != NULL) {
        if (p2->adjvex == v_pos) {
            flag2 = 1;
            if (prev2 == NULL) G->vertices[w_pos].firstarc = p2->nextarc;
            ↪ //删除头结点，要单独讨论
            else prev2->nextarc = p2->nextarc;
            free(p2);
            break;
        }
        prev2 = p2;
        p2 = p2->nextarc;
    }
    if (flag1 == 0 || flag2 == 0) return ERROR;

    G->arcnum--;
    return OK;
}
/* 深度优先搜索遍历 */
void visit(VertexType v){
    printf(" %d %s ", v.key, v.others);
}
void DFS(ALGraph* G, int i, int visited[], void (*visit)(VertexType))
// 辅助递归函数：从顶点 i 出发进行深度优先搜索
```

```
{
    // 标记当前顶点已访问，并调用 visit 函数访问该顶点的数据
    visited[i] = TRUE;
    visit(G->vertices[i].data);

    // 遍历顶点 i 的邻接表，对所有尚未访问的临接顶点调用 DFS
    ArcNode* p = G->vertices[i].firstarc;
    while (p != NULL) {
        if (!visited[p->adjvex])
            DFS(G, p->adjvex, visited, visit);
        p = p->nextarc;
    }
}
status DFSTraverse(ALGraph* G, void (*visit)(VertexType))
// 主函数：对图 G 进行深度优先搜索遍历
{
    int visited[MAX_VERTEX_NUM];

    // 初始化所有顶点为未访问
    for (int i = 0; i < G->vexnum; i++) {
        visited[i] = FALSE;
    }

    // 依次对每个未被访问的顶点调用 DFS（处理非连通图）
    for (int i = 0; i < G->vexnum; i++) {
        if (!visited[i])
            DFS(G, i, visited, visit);
    }
    return OK;
}
/* 广度优先搜索遍历 */
status BFSTraverse(ALGraph* G, void (*visit)(VertexType))
{
    int visited[MAX_VERTEX_NUM];
    for (int i = 0; i < G->vexnum; i++) {
```

```
        visited[i] = FALSE;
    }

    int queue[MAX_VERTEX_NUM];
    int front = 0, rear = 0; // 队头、队尾指针

    for (int i = 0; i < G->vexnum; i++) {
        if (!visited[i]) {
            // 将未访问的顶点入队，并标记为已访问
            visited[i] = TRUE;
            visit(G->vertices[i].data);
            queue[rear++] = i;

            // 当队列不空时，持续将队头出队，访问其所有邻接顶点
            while (front < rear) {
                int cur = queue[front++];
                // 遍历当前顶点 cur 的所有邻接点
                ArcNode* p = G->vertices[cur].firstarc;
                while (p != NULL) {
                    if (!visited[p->adjvex]) {
                        // 访问并入队
                        visited[p->adjvex] = TRUE;
                        visit(G->vertices[p->adjvex].data);
                        queue[rear++] = p->adjvex;
                    }
                    p = p->nextarc;
                }
            }
        }
    }
    return OK;
}

//距离小于 k 的顶点集合，利用 BFS 记录从顶点 v 到其它各顶点的最短距离，只有距
↪  离小于 k 的顶点被保留。
```

```
int* VerticesSetLessThanK(ALGraph* G, int v, int k, int* pSize) {
    if (G == NULL || v < 0 || v >= G->vexnum || pSize == NULL)
        return NULL;

    //动态分配数组保存各顶点到 v 的最短距离
    int* dist = (int*)malloc(G->vexnum * sizeof(int));
    if (dist == NULL) return NULL;
    for (int i = 0; i < G->vexnum; i++)
        dist[i] = INF;
    dist[v] = 0;

    //使用队列实现 BFS
    int queue[MAX_VERTEX_NUM];
    int front = 0, rear = 0;
    queue[rear++] = v;

    while (front < rear) {
        int cur = queue[front++];
        //当当前顶点距离达到 k-1 时，若扩展后续顶点可能达到 k，不再推进队列
        if (dist[cur] >= k - 1)
            continue;

        ArcNode* p = G->vertices[cur].firstarc;
        while (p) {
            int adj = p->adjvex;
            if (dist[adj] == INF) {  //若尚未标记过
                dist[adj] = dist[cur] + 1;
                if (dist[adj] < k)
                    queue[rear++] = adj;
            }
            p = p->nextarc;
        }
    }

    //统计所有距离 < k 的顶点数
```

```
    int count = 0;
    for (int i = 0; i < G->vexnum; i++) {
        if (dist[i] < k)
            count++;
    }

    //分配数组保存结果，存储满足条件顶点的下标
    int* result = (int*)malloc(count * sizeof(int));
    if (result == NULL) {
        free(dist);
        return NULL;
    }
    int index = 0;
    for (int i = 0; i < G->vexnum; i++) {
        if (dist[i] < k)
            result[index++] = i;
    }
    *pSize = count;
    free(dist);
    return result;
}
//顶点间最短路径和长度
int ShortestPathLength(ALGraph* G, int v, int w)
{
    if (G == NULL || v < 0 || w < 0 || v >= G->vexnum || w >= G->vexnum)
        return -1;

    int distance[MAX_VERTEX_NUM];
    for (int i = 0; i < G->vexnum; i++)
        distance[i] = INF;
    distance[v] = 0;

    int queue[MAX_VERTEX_NUM];
    int front = 0, rear = 0;
    queue[rear++] = v;
```

```
    while (front < rear)
    {
        int cur = queue[front++];
        if (cur == w)
            return distance[w];
        ArcNode* p = G->vertices[cur].firstarc;
        while (p)
        {
            int adj = p->adjvex;
            if (distance[adj] == INF)
            {
                distance[adj] = distance[cur] + 1;
                queue[rear++] = adj;
            }
            p = p->nextarc;
        }
    }
    return -1;  //无法到达 w
}
//图的连通分量数
void dummyVisit(VertexType v) {} //空 visit 函数
int ConnectedComponentsNums(ALGraph* G)
{
    if (G == NULL)
        return -1;
    int visited[MAX_VERTEX_NUM] = { 0 };
    int count = 0;
    for (int i = 0; i < G->vexnum; i++) {
        if (!visited[i]) {
            /* 对连通分量中所有顶点调用 DFS, 使用 dummyVisit 不打印或处理数据
            ↪  */
            DFS(G, i, visited, dummyVisit);
            count++;
        }
```

```
    }
    return count;
}
/* 将无向图的数据写到文件中 */
status SaveGraph(ALGraph G, char FileName[])
{
    FILE* fp = fopen(FileName, "w");
    if (fp == NULL)
        return ERROR;

    // 对每个顶点写一行: 先写顶点数据, 然后逆序写出它邻接表中各弧结点的 adjvex
    ↪ 值
    for (int i = 0; i < G.vexnum; i++) {
        // 先输出顶点关键字和其他数据
        fprintf(fp, "%d %s", G.vertices[i].data.key,
        ↪ G.vertices[i].data.others);

        // 将该顶点邻接链表中的结点 adjvex 值先存入临时数组中
        int tempArr[MAX_VERTEX_NUM];
        int count = 0;
        ArcNode* p = G.vertices[i].firstarc;
        while (p != NULL) {
            tempArr[count++] = p->adjvex;
            p = p->nextarc;
        }

        // 由于采用首插法, 链表中的顺序是逆序的, 为保证输出时和预期一致, 逆序输
        ↪ 出临时数组
        for (int j = count - 1; j >= 0; j--) {
            fprintf(fp, " %d", tempArr[j]);
        }
        fprintf(fp, "\n");
    }
    fclose(fp);
    return OK;
```

```
}
/* 读入文件中的结点数据创建无向图的邻接表 */
status LoadGraph(ALGraph* G, char FileName[]) {
    FILE* fp = fopen(FileName, "r");
    if (fp == NULL)
        return ERROR;

    char line[256];
    int vertexCount = 0;
    int totalAdjCount = 0;
    // 逐行读取文件，依据行数确定顶点个数
    while (fgets(line, sizeof(line), fp) != NULL) {
        if (strlen(line) <= 1) continue; // 如果行内容为空，则跳过

        char* token = strtok(line, " \n");
        if (token == NULL) continue;
        int key = atoi(token);// 第一个标记：顶点关键字

        token = strtok(NULL, " \n");
        if (token == NULL) continue;
        char others[20];
        strcpy(others, token);

        // 将读入数据存入顶点数组
        G->vertices[vertexCount].data.key = key;
        strcpy(G->vertices[vertexCount].data.others, others);
        G->vertices[vertexCount].firstarc = NULL;

        // 后续标记每个为邻接结点的下标（以整数读取）
        token = strtok(NULL, " \n");
        while (token != NULL) {
            int adjIndex = atoi(token);
            // 采用首插法建立链表
            ArcNode* p = (ArcNode*)malloc(sizeof(ArcNode));
            if (p == NULL) {
```

```
                fclose(fp);
                return OVERFLOW;
            }
            p->adjvex = adjIndex;
            p->nextarc = G->vertices[vertexCount].firstarc;
            G->vertices[vertexCount].firstarc = p;
            totalAdjCount++;
            token = strtok(NULL, " \n");
        }
        vertexCount++;
    }
    G->vexnum = vertexCount;
    G->arcnum = totalAdjCount / 2; // 对无向图，每条边存储两次
    G->kind = UDG;
    fclose(fp);
    return OK;
}

/* 多个图管理 */
void InitGraphManager(GraphManager* gm)
{
    if (gm == NULL) return;
    gm->count = 0;
    // 初始化名字字符串为空
    for (int i = 0; i < MAX_GRAPHS; i++) {
        gm->names[i][0] = '\0';
        gm->graphs[i] = NULL;
    }
}
status AddGraph(GraphManager* gm, ALGraph* G, const char name[])
{
    if (gm == NULL || G == NULL || name == NULL)
        return ERROR;
    if (gm->count >= MAX_GRAPHS)
        return ERROR;
```

```
    gm->graphs[gm->count] = G;
    strncpy(gm->names[gm->count], name, NAME_LEN - 1);
    gm->names[gm->count][NAME_LEN - 1] = '\0';  // 确保字符串终止
    gm->count++;
    return OK;
}
status DeleteGraph(GraphManager* gm, const char name[])
{
    if (gm == NULL || name == NULL)
        return ERROR;
    int index = -1;
    for (int i = 0; i < gm->count; i++) {
        if (strcmp(gm->names[i], name) == 0) {
            index = i;
            break;
        }
    }
    if (index == -1) {
        printf(" 未找到图: %s\n", name);
        return ERROR;
    }
    if (DestroyGraph(gm->graphs[index]) != OK) {
        return ERROR;
    }
    free(gm->graphs[index]);  // 释放图结构体内存
    // 将后续图和名称前移覆盖删除的图
    for (int i = index; i < gm->count - 1; i++) {
        gm->graphs[i] = gm->graphs[i + 1];
        strcpy(gm->names[i], gm->names[i + 1]);
    }
    gm->graphs[gm->count - 1] = NULL;
    gm->names[gm->count - 1][0] = '\0';
    gm->count--;
    return OK;
}
```

```
void DisplayGraphManager(GraphManager* gm)
{
    if (gm == NULL) return;
    printf(" 当前管理的图数量: %d\n", gm->count);
    for (int i = 0; i < gm->count; i++) {
        printf(" 序号: %d, 名称: %s\n", i + 1, gm->names[i]);
    }
}
ALGraph* GetGraphByName(GraphManager* gm, const char name[])
{
    if (gm == NULL || name == NULL)
        return NULL;
    for (int i = 0; i < gm->count; i++) {
        if (strcmp(gm->names[i], name) == 0)
            return gm->graphs[i];
    }
    return NULL;
}

/* 菜单函数 */
void PrintMenu()
{
    printf("\n  -------图 (邻接表) 演示系统-------  \n");
    printf("***************************************\n");
    printf("  0. 退出演示系统\n");
    printf("  1. 创建单个图\n");
    printf("  2. 销毁当前图\n");
    printf("  3. 查找顶点\n");
    printf("  4. 顶点赋值\n");
    printf("  5. 获得第一邻接点\n");
    printf("  6. 获得下一邻接点\n");
    printf("  7. 插入顶点\n");
    printf("  8. 删除顶点\n");
    printf("  9. 插入弧\n");
    printf(" 10. 删除弧\n");
```

```
    printf(" 11. 深度优先搜索遍历\n");
    printf(" 12. 广度优先搜索遍历\n");
    printf(" 13. 附加功能 1: 求距离小于 k 的顶点集合\n");
    printf(" 14. 附加功能 2: 求顶点间最短路径和长度\n");
    printf(" 15. 附加功能 3: 求图的连通分量数\n");
    printf(" 16. 附加功能 4: 实现图的文件形式保存\n");
    printf(" 17. 附加功能 5: 从文件加载创建图\n");
    printf(" 18. 附加功能 6: 多图管理\n");
    printf("************************************\n");
    printf(" 请输入你的选择: ");
}
void PrintSubMenu()
{
    printf("\n--- 多图管理子系统 ---\n");
    printf("  1. 创建新的图\n");
    printf("  2. 显示所有图状态\n");
    printf("  3. 删除某个图\n");
    printf("  4. 操作其中某个图\n");
    printf("  0. 返回主菜单\n");
    printf(" 请输入你的选择: ");
}
void PrintSubSubMenu(char* listname)
{
    printf("\n-----操作图 \"%s\" -----\n", listname);
    printf("  0. 返回子菜单\n");
    printf("  1. 销毁当前图\n");
    printf("  2. 查找顶点\n");
    printf("  3. 顶点赋值\n");
    printf("  4. 获得第一邻接点\n");
    printf("  5. 获得下一邻接点\n");
    printf("  6. 插入顶点\n");
    printf("  7. 删除顶点\n");
    printf("  8. 插入弧\n");
    printf("  9. 删除弧\n");
    printf(" 10. 深度优先搜索遍历\n");
```

```
    printf(" 11. 广度优先搜索遍历\n");
    printf(" 12. 附加功能 1: 求距离小于 k 的顶点集合\n");
    printf(" 13. 附加功能 2: 求顶点间最短路径和长度\n");
    printf(" 14. 附加功能 3: 求图的连通分量数\n");
    printf(" 15. 附加功能 4: 实现图的文件形式保存\n");
    printf(" 16. 附加功能 5: 从文件加载创建图\n");
    printf(" 请输入您的选择: \n");
}
```

文件 3：main.c

```
#define _CRT_SECURE_NO_WARNINGS

#include"functions.h"

int main(void)
{
    int choice;
    ALGraph* currentGraph = NULL;
    GraphManager gm;
    InitGraphManager(&gm);

    while (1)
    {
        PrintMenu();
        scanf("%d", &choice);
        switch (choice)
        {
        case 0:
            printf(" 退出系统! \n");
            exit(0);
            break;
        case 1:
        {
            // 创建单个图
```

```
if (currentGraph != NULL)
{
    printf(" 当前已有图，不能再创建图！!\n");
    break;
}
int n, m;
printf(" 请输入顶点个数 (<= %d): ", MAX_VERTEX_NUM);
scanf("%d", &n);
if (n <= 0 || n > MAX_VERTEX_NUM) {
    printf(" 顶点个数非法！\n");
    break;
}
VertexType V[MAX_VERTEX_NUM + 1];
for (int i = 0; i < n; i++) {
    printf(" 请输入第%d 个顶点的关键字（整数）和其他信息（字符
    ↪  串): ", i + 1);
    scanf("%d %s", &V[i].key, V[i].others);
}
// 设置哨兵，标识顶点输入结束
V[n].key = -1;

printf(" 请输入边数: ");
scanf("%d", &m);
KeyType VR[MAX_VERTEX_NUM + 1][2];
for (int i = 0; i < m; i++) {
    printf(" 请输入第%d 条边两个邻点的关键字: ", i + 1);
    scanf("%d %d", &VR[i][0], &VR[i][1]);
}
VR[m][0] = -1; VR[m][1] = -1;

currentGraph = (ALGraph*)malloc(sizeof(ALGraph));
if (currentGraph == NULL) {
    printf(" 内存分配失败！\n");
    break;
}
```

```
        if (CreateGraph(currentGraph, V, VR) == OK)
            printf(" 图创建成功! \n");
        else {
            printf(" 图创建失败! \n");
            free(currentGraph);
            currentGraph = NULL;
        }
    }
    break;
    case 2:
    {
        // 销毁当前单图
        if (currentGraph == NULL) {
            printf(" 当前没有图! \n");
            break;
        }
        if (DestroyGraph(currentGraph) == OK) {
            free(currentGraph);
            currentGraph = NULL;
            printf(" 图销毁成功! \n");
        }
        else {
            printf(" 图销毁失败! \n");
        }
    }
    break;
    case 3:
    {
        // 查找顶点
        if (currentGraph == NULL) {
            printf(" 当前没有图! \n");
            break;
        }
        int key;
        printf(" 请输入待查找的顶点关键字: ");
```

```
        scanf("%d", &key);
        int pos = LocateVex(*currentGraph, key);
        if (pos != -1)
            printf(" 找到关键字为%d 的顶点, 其下标为%d\n", key, pos);
        else
            printf(" 未找到该顶点! \n");
    }
    break;
    case 4:
    {
        // 顶点赋值
        if (currentGraph == NULL) {
            printf(" 当前没有图! \n");
            break;
        }
        int key;
        VertexType value;
        printf(" 请输入要赋值的顶点关键字 (整数) 和新的其他信息 (字符串) :
        ↪  ");
        scanf("%d %s", &key, value.others);
        value.key = key;
        if (PutVex(currentGraph, key, value) == OK)
            printf(" 顶点赋值成功! \n");
        else
            printf(" 顶点赋值失败! \n");
    }
    break;
    case 5:
    {
        // 获得第一邻接点
        if (currentGraph == NULL) {
            printf(" 当前没有图! \n");
            break;
        }
        int key;
```

```
        printf(" 请输入待求顶点的关键字: ");
        scanf("%d", &key);
        int first = FirstAdjVex(*currentGraph, key);
        if (first != -1) {
            printf(" 顶点%d 的第一邻接点下标为%d, ", key, first);
            printf(" 顶点信息为: %d %s\n",
            ↪  currentGraph->vertices[first].data.key,
            ↪  currentGraph->vertices[first].data.others);
        }
        else
            printf(" 未找到顶点%d 的邻接点! \n", key);
    }
    break;
    case 6:
    {
        // 获得下一邻接点
        if (currentGraph == NULL) {
            printf(" 当前没有图! \n");
            break;
        }
        int v, w;
        printf(" 请输入顶点 v 和已知邻接顶点 w 的关键字: ");
        scanf("%d %d", &v, &w);
        int next = NextAdjVex(*currentGraph, v, w);
        if (next != -1) {
            printf(" 顶点%d 在顶点%d 之后的邻接点下标为%d, ", v, w,
            ↪  next);
            printf(" 顶点信息为: %d %s\n",
            ↪  currentGraph->vertices[next].data.key,
            ↪  currentGraph->vertices[next].data.others);
        }
        else
            printf(" 未找到满足条件的邻接点! \n");
    }
    break;
```

```
case 7:
{
    // 插入顶点
    if (currentGraph == NULL) {
        printf(" 当前没有图! \n");
        break;
    }
    VertexType newV;
    printf(" 请输入要插入顶点的关键字 (整数) 和其它信息 (字符串) :");
    scanf("%d %s", &newV.key, newV.others);
    if (InsertVex(currentGraph, newV) == OK)
        printf(" 顶点插入成功! \n");
    else
        printf(" 顶点插入失败! \n");
}
break;
case 8:
{
    // 删除顶点
    if (currentGraph == NULL) {
        printf(" 当前没有图! \n");
        break;
    }
    int key;
    printf(" 请输入要删除的顶点的关键字: ");
    scanf("%d", &key);
    if (DeleteVex(currentGraph, key) == OK)
        printf(" 顶点删除成功! \n");
    else
        printf(" 顶点删除失败! \n");
}
break;
case 9:
{
    // 插入弧
```

```
        if (currentGraph == NULL) {
            printf(" 当前没有图! \n");
            break;
        }
        int v, w;
        printf(" 请输入要插入弧的两个邻点的关键字: ");
        scanf("%d %d", &v, &w);
        if (InsertArc(currentGraph, v, w) == OK)
            printf(" 插入弧成功! \n");
        else
            printf(" 插入弧失败! \n");
    }
    break;
    case 10:
    {
        // 删除弧
        if (currentGraph == NULL) {
            printf(" 当前没有图! \n");
            break;
        }
        int v, w;
        printf(" 请输入要删除弧的两个邻点的关键字: ");
        scanf("%d %d", &v, &w);
        if (DeleteArc(currentGraph, v, w) == OK)
            printf(" 删除弧成功! \n");
        else
            printf(" 删除弧失败! \n");
    }
    break;
    case 11:
    {
        // 深度优先搜索遍历
        if (currentGraph == NULL) {
            printf(" 当前没有图! \n");
            break;
```

```
        }
        printf(" 深度优先搜索遍历结果:\n");
        if (DFSTraverse(currentGraph, visit) == OK)
            printf("\n 深度优先搜索遍历完成! \n");
        else
            printf("\n 深度优先搜索遍历失败! \n");
    }
    break;
    case 12:
    {
        // 广度优先搜索遍历
        if (currentGraph == NULL) {
            printf(" 当前没有图! \n");
            break;
        }
        printf(" 广度优先搜索遍历结果:\n");
        if (BFSTraverse(currentGraph, visit) == OK)
            printf("\n 广度优先搜索遍历遍历完成! \n");
        else
            printf("\n 广度优先搜索遍历遍历失败! \n");
    }
    break;
    case 13:
    {
        // 附加功能 1: 求距离小于 k 的顶点集合
        if (currentGraph == NULL) {
            printf(" 当前没有图! \n");
            break;
        }
        int v, k, pSize;
        printf(" 请输入起始顶点下标 v 和距离 k: ");
        scanf("%d %d", &v, &k);
        int* set = VerticesSetLessThanK(currentGraph, v, k, &pSize);
        if (set != NULL) {
            printf(" 距离小于%d 的顶点下标集合为: ", k);
```

```
            for (int i = 0; i < pSize; i++) {
                printf("%d ", set[i]);
            }
            printf("\n");
            free(set);
        }
        else {
            printf(" 操作失败或无满足条件的顶点! \n");
        }
    }
    break;
    case 14:
    {
        // 附加功能 2: 求顶点间最短路径和长度
        if (currentGraph == NULL) {
            printf(" 当前没有图! \n");
            break;
        }
        int v, w;
        printf(" 请输入起始顶点下标和终止顶点下标: ");
        scanf("%d %d", &v, &w);
        int length = ShortestPathLength(currentGraph, v, w);
        if (length != -1)
            printf(" 顶点%d 到顶点%d 的最短路径长度为%d\n", v, w,
            ↪  length);
        else
            printf(" 两个顶点之间无路径! \n");
    }
    break;
    case 15:
    {
        // 附加功能 3: 求图的连通分量数
        if (currentGraph == NULL) {
            printf(" 当前没有图! \n");
            break;
```

```
        }
        int comps = ConnectedComponentsNums(currentGraph);
        printf(" 该图的连通分量数为 %d\n", comps);
    }
    break;
    case 16:
    {
        // 附加功能 4: 图的文件形式保存
        if (currentGraph == NULL) {
            printf(" 当前没有图! \n");
            break;
        }
        char fileName[100];
        printf(" 请输入保存图的文件名: ");
        scanf("%s", fileName);
        if (SaveGraph(*currentGraph, fileName) == OK)
            printf(" 图保存成功! \n");
        else
            printf(" 图保存失败! \n");
    }
    break;
    case 17:
    {
        // 附加功能 5: 从文件加载创建图
        if (currentGraph == NULL) {
            printf(" 当前没有图! \n");
            break;
        }
        char fileName[100];
        printf(" 请输入加载图的文件名: ");
        scanf("%s", fileName);
        if (LoadGraph(currentGraph, fileName) == OK)
            printf(" 图加载成功! \n");
        else
            printf(" 图加载失败! \n");
```

```
}
break;
case 18:
{
    // 进入多图管理子系统
    int subChoice;
    while (1)
    {
        PrintSubMenu();
        scanf("%d", &subChoice);
        if (subChoice == 0)
            break;
        switch (subChoice)
        {
        case 1:
        {
            // 创建新的图（多图管理）
            char name[NAME_LEN];
            int createOption;
            printf(" 请输入新图名称: ");
            scanf("%s", name);
            printf(" 请选择创建方式: 1. 仅初始化  2. 初始化 + 输入数
            ↪  据: ");
            scanf("%d", &createOption);
            ALGraph* newGraph = (ALGraph*)malloc(sizeof(ALGraph));
            if (newGraph == NULL) {
                printf(" 内存分配失败! \n");
                break;
            }
            if (createOption == 1) {
                // 仅初始化: 简单设置顶点数为 0, 弧数为 0
                newGraph->vexnum = 0;
                newGraph->arcnum = 0;
                newGraph->kind = UDG;
                printf(" 图已初始化。\n");
```

```
}
else if (createOption == 2) {
    // 初始化并输入数据
    int n, m;
    printf(" 请输入顶点个数 (<= %d): ", MAX_VERTEX_NUM);
    scanf("%d", &n);
    if (n <= 0 || n > MAX_VERTEX_NUM) {
        printf(" 顶点个数非法! \n");
        free(newGraph);
        break;
    }
    VertexType V[MAX_VERTEX_NUM + 1]; //为结束哨兵流出空
    ↪ 间
    for (int i = 0; i < n; i++) {
        printf(" 请输入第%d 个顶点的关键字 (整数) 和其他
        ↪ 信息 (字符串): ", i + 1);
        scanf("%d %s", &V[i].key, V[i].others);
    }
    // 设置哨兵, 标识顶点输入结束
    V[n].key = -1;

    printf(" 请输入边数: ");
    scanf("%d", &m);
    KeyType VR[MAX_VERTEX_NUM + 1][2];
    for (int i = 0; i < m; i++) {
        printf(" 请输入第%d 条边的两个顶点的关键字: ", i
        ↪ + 1);
        scanf("%d %d", &VR[i][0], &VR[i][1]);
    }
    VR[m][0] = -1; VR[m][1] = -1;

    if (CreateGraph(newGraph, V, VR) == OK)
        printf(" 图创建成功! \n");
    else {
        printf(" 图创建失败! \n");
```

```
                free(newGraph);
                break;
            }
        }
        else {
            printf(" 无效的选项! \n");
            free(newGraph);
            break;
        }
        if (AddGraph(&gm, newGraph, name) == OK)
            printf(" 图 '%s' 已添加到多图管理器中! \n", name);
        else {
            printf(" 添加失败! \n");
            free(newGraph);
        }
    }
    break;
    case 2:
    {
        // 显示所有图状态
        DisplayGraphManager(&gm);
    }
    break;
    case 3:
    {
        // 删除图
        char delName[NAME_LEN];
        printf(" 请输入要删除的图名称: ");
        scanf("%s", delName);
        if (DeleteGraph(&gm, delName) == OK)
            printf(" 图 '%s' 删除成功! \n", delName);
        else
            printf(" 删除失败! \n");
    }
    break;
```

```
case 4:
{
    char opName[NAME_LEN];
    printf(" 请输入要操作的图名称: ");
    scanf("%s", opName);
    ALGraph* opGraph = GetGraphByName(&gm, opName);
    if (opGraph == NULL) {
        printf(" 未找到图 '%s'! \n", opName);
        break;
    }
    int subSubChoice;
    while (1)
    {
        PrintSubSubMenu(opName);
        scanf("%d", &subSubChoice);
        switch (subSubChoice)
        {
        case 0:
            // 返回多图管理子系统
            printf(" 返回多图管理子系统。\n");
            break;
        case 1:
            // 销毁当前图
            if (DeleteGraph(&gm, opName) == OK)
                printf(" 图 '%s' 已销毁并从管理系统中删除!
                ↪ \n", opName);
            else
                printf(" 图 '%s' 销毁失败! \n", opName);
            // 退出子子菜单, 因为所操作图已不存在
            subSubChoice = 0;
            break;
        case 2:
        {
            int key;
            printf(" 请输入待查找的顶点关键字: ");
```

```
            scanf("%d", &key);
            int pos = LocateVex(*opGraph, key);
            if (pos != -1)
                printf(" 找到关键字为%d 的顶点, 下标为%d\n",
                ↪ key, pos);
            else
                printf(" 未找到该顶点! \n");
        }
        break;
        case 3: {
            int oldKey, newKey;
            VertexType value;
            printf(" 请输入原顶点关键字: ");
            scanf("%d", &oldKey);
            printf("\n 请输入新的顶点关键字 (整数) 和新的其他
            ↪ 信息 (字符串): ");
            scanf("%d %s", &newKey, value.others);
            value.key = newKey;
            printf("\n");
            if (PutVex(opGraph, oldKey, value) == OK)
                printf(" 顶点赋值成功! \n");
            else
                printf(" 顶点赋值失败! \n");
            break;
        }
        case 4:
        {
            int key;
            printf(" 请输入待求顶点的关键字: ");
            scanf("%d", &key);
            int first = FirstAdjVex(*opGraph, key);
            if (first != -1) {
                    printf(" 顶点%d 的第一邻接点下标为%d, ",
                    ↪ key, first);
```

```
                printf(" 顶点信息为: %d %s\n",
                ↪ opGraph->vertices[first].data.key,
                ↪ opGraph->vertices[first].data.others);
        }
        else
            printf(" 未找到顶点%d 的第一邻接点! \n",
            ↪ key);
    }
    break;
    case 5:
    {
        int v, w;
        printf(" 请输入顶点 v 和已知邻接顶点 w 的关键字:
        ↪ ");
        scanf("%d %d", &v, &w);
        int next = NextAdjVex(*opGraph, v, w);
        if (next != -1) {
            printf(" 顶点%d 在顶点%d 之后的邻接点下标
            ↪ 为%d, ", v, w, next);
            printf(" 顶点信息为: %d %s\n",
            ↪ opGraph->vertices[next].data.key,
            ↪ opGraph->vertices[next].data.others);
        }
        else
            printf(" 未找到满足条件的邻接点! \n");
    }
    break;
    case 6:
    {
        VertexType newV;
        printf(" 请输入新顶点的关键字 (整数) 和其他信息
        ↪ (字符串) : ");
        scanf("%d %s", &newV.key, newV.others);
        if (InsertVex(opGraph, newV) == OK)
            printf(" 顶点插入成功! \n");
```

```
        else
            printf(" 顶点插入失败! \n");
    }
    break;
    case 7:
    {
        int key;
        printf(" 请输入要删除的顶点的关键字: ");
        scanf("%d", &key);
        if (DeleteVex(opGraph, key) == OK)
            printf(" 顶点删除成功! \n");
        else
            printf(" 顶点删除失败! \n");
    }
    break;
    case 8:
    {
        int v, w;
        printf(" 请输入要插入弧的两个邻点的关键字: ");
        scanf("%d %d", &v, &w);
        if (InsertArc(opGraph, v, w) == OK)
            printf(" 插入弧成功! \n");
        else
            printf(" 插入弧失败! \n");
    }
    break;
    case 9:
    {
        int v, w;
        printf(" 请输入要删除弧的两个邻点关键字: ");
        scanf("%d %d", &v, &w);
        if (DeleteArc(opGraph, v, w) == OK)
            printf(" 删除弧成功! \n");
        else
            printf(" 删除弧失败! \n");
```

```
}
break;
case 10:
{
    printf(" 深度优先搜索遍历结果:\n");
    if (DFSTraverse(opGraph, visit) == OK)
        printf("\n 深度优先搜索遍历完成! \n");
    else
        printf("\n 深度优先搜索遍历失败! \n");
}
break;
case 11:
{
    printf(" 广度优先搜索遍历结果:\n");
    if (BFSTraverse(opGraph, visit) == OK)
        printf("\n 广度优先搜索遍历完成! \n");
    else
        printf("\n 广度优先搜索遍历失败! \n");
}
break;
case 12:
{
    int v, k, pSize;
    printf(" 请输入起始顶点下标 v 和距离 k: ");
    scanf("%d %d", &v, &k);
    int* set = VerticesSetLessThanK(opGraph, v, k,
    ↪  &pSize);
    if (set != NULL) {
        printf(" 距离小于%d 的顶点下标集合为: ", k);
        for (int i = 0; i < pSize; i++) {
            printf("%d ", set[i]);
        }
        printf("\n");
        free(set);
    }
```

```
        else {
            printf(" 操作失败或无顶点满足条件。\n");
        }
    }
    break;
    case 13:
    {
        int v, w;
        printf(" 请输入起始顶点下标和终止顶点下标: ");
        scanf("%d %d", &v, &w);
        int length = ShortestPathLength(opGraph, v, w);
        if (length != -1)
            printf(" 顶点%d 到顶点%d 的最短路径长度
            ↪  为%d\n", v, w, length);
        else
            printf(" 两个顶点之间无路径! \n");
    }
    break;
    case 14:
    {
        int comps = ConnectedComponentsNums(opGraph);
        printf(" 该图的连通分量数为%d\n", comps);
    }
    break;
    case 15:
    {
        char fileName[100];
        printf(" 请输入保存图的文件名: ");
        scanf("%s", fileName);
        if (SaveGraph(*opGraph, fileName) == OK)
            printf(" 图保存成功! \n");
        else
            printf(" 图保存失败! \n");
    }
    break;
```

```
                    case 16:
                    {
                        char fileName[100];
                        printf(" 请输入加载图的文件名: ");
                        scanf("%s", fileName);
                        if (LoadGraph(opGraph, fileName) == OK)
                            printf(" 图数据加载成功! \n");
                        else
                            printf(" 图数据加载失败! \n");
                    }
                    break;
                    default:
                        printf(" 请输入正确的选项! \n");
                    } // end switch subSubChoice

                    if (subSubChoice == 0)
                        break;
                } // end while sub-sub 菜单
                break;
            }
            default:
                printf(" 请输入正确的选项! \n");
            }
        }
        break;
        default:
            printf(" 请输入正确的选项! \n");
        }
    }
    return 0;
}
```