

华中科技大学

课程实验报告

课程名称： 数据结构实验

专业班级 CS2111

学 号 U202112313

姓 名 陈东平

指导教师 郑渤龙

报告日期 2022 年 5 月 30 日

计算机科学与技术学院

目 录

1 基于顺序存储结构的线性表实现.....	1
1.1 问题描述	1
1.2 系统设计	2
1.3 系统实现	4
1.4 系统测试	12
1.5 实验小结	31
2 基于二叉链表的二叉树实现	32
2.1 问题描述	32
2.2 系统设计	34
2.3 系统实现	37
2.4 系统测试	45
2.5 实验小结	64
3 课程的收获和建议	65
3.1 基于顺序存储结构的线性表实现	65
3.2 基于二叉链表的二叉树实现	65
参考文献	67
附录 A 基于顺序存储结构线性表实现的源程序	67
附录 B 基于二叉链表二叉树实现的源程序	86

1 基于顺序存储结构的线性表实现

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，具体运算功能定义如下：

(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)`；初始条件是线性表已存在， $1 \leq i \leq \text{ListLength}(L)$ ；操作结果是用 `e` 返回 `L` 中第 `i` 个数据元素的值；

(7) 查找元素：函数名称是 `LocateElem(L,e)`；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` 满足相同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， $1 \leq i \leq \text{ListLength}(L)+1$ ；操作结果是在 `L` 的第 `i` 个位置之前插入新的数据元素 `e`。

(11) 删除元素：函数名称是 `ListDelete(L,i,e)`；初始条件是线性表 `L` 已存在

且非空, $1 \leq i \leq \text{ListLength}(L)$; 操作结果: 删除 L 的第 i 个数据元素, 用 e 返回其值;

(12) 遍历表: 函数名称是 $\text{ListTraverse}(L)$, 初始条件是线性表 L 已存在; 操作结果是依次对 L 的每个数据元素进行访问。

附加功能:

(1) 最大连续子数组和: 函数名称是 $\text{MaxSubArray}(L)$; 初始条件是线性表 L 已存在且非空, 请找出一个具有最大和的连续子数组 (子数组最少包含一个元素), 操作结果是其最大和;

(2) 和为 K 的子数组: 函数名称是 $\text{SubArrayNum}(L, k)$; 初始条件是线性表 L 已存在且非空, 操作结果是该数组中和为 k 的连续子数组的个数;

(3) 顺序表排序: 函数名称是 $\text{sortList}(L)$; 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: i) 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; ii) 需要设计线性表文件保存和加载操作合理模式。附录 B 提供了文件存取的参考方法。

(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。

1.2 系统设计

1.2.1 整体系统结构设计

本实现方案首先进入一个循环, 循环中, 由用户输入待操作的线性表的序号, 然后对线性表进行操作, 进入不同的分支, 直到用户输入 0 循环结束, 同时算法结束。如果需要切换多线性表进行操作, 则要先创建多线性表中的线性表, 之后由操作分支切换过去即可。该算法的流程图如图 1.1 所示, 实际选择菜单图如图 1.2 所示。

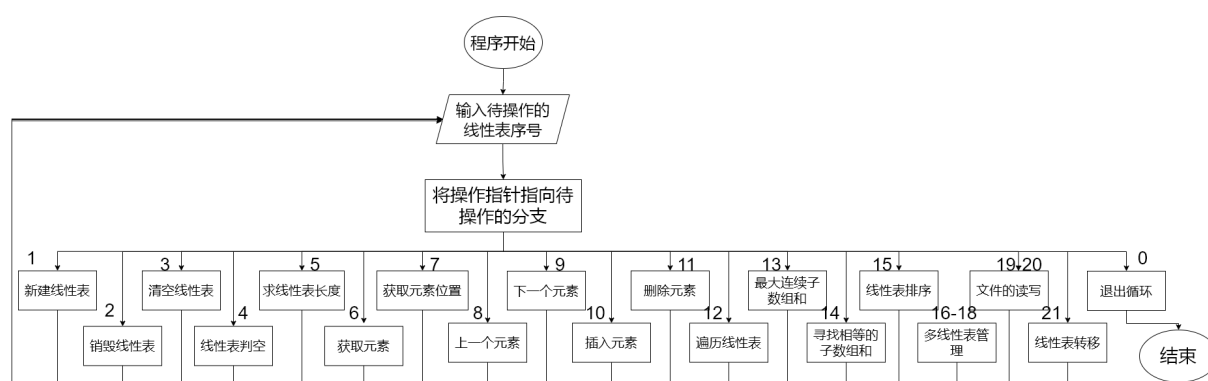


图 1-1 整体系统结构设计

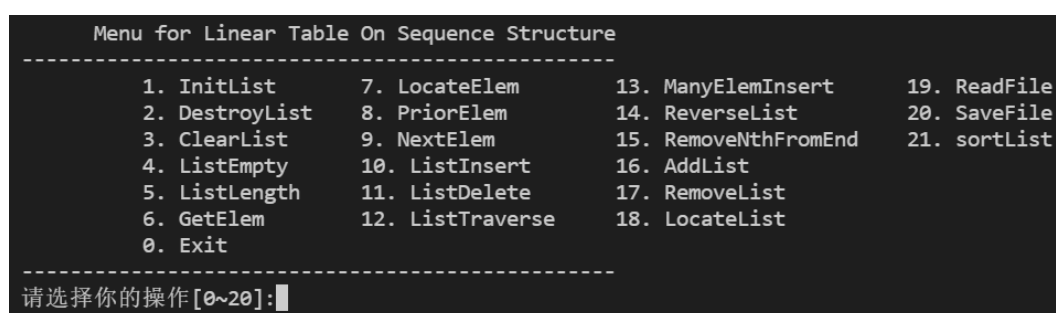


图 1-2 实际选择菜单图

1.2.2 数据结构设计

(1) 定义头文件。该程序使用了 `<stdio.h>`、`<malloc.h>`、`<stdlib.h>`、`<string.h>` 库, 使用这些 C 语言库可以很方便的进行动态分配结点或者字符串, 以及对字符串进行操作。

(2) 定义常量。一部分常量用来指定函数的返回值, 用于判断函数执行的情况。定义的常量有: `TRUE(1)`, `FALSE(0)`, `OK(1)`, `ERROR(0)`, `INFEASTABLE(-1)`, `OVERFLOW(-2)`。还有常量用于指定线性表存储空间的分配量, `LIST_INIT_SIZE(100)`, 表示线性表存储空间的分配量)。

(3) 定义数据元素的类型。此处定义了部分函数返回值的类型 `status` 以及线性表中单个数据的类型 `ElemType`。本实验中将二者都定义为 `int` 类型。

(4) 定义线性表的数据结构。线性表结构体中包含三个元素, 分别为元素存储空间的首指针 `elem`、当前已存储元素的数量 `length` 以及当前分配的存储空间的总长度 `listsize`。并将其类型定义为 `SqList`。

线性表结构体在内存中的存储状态如图 1.3 所示，图中每一行三个元素分别表示一个顺序表，线性表的长度和线性表的最大长度。

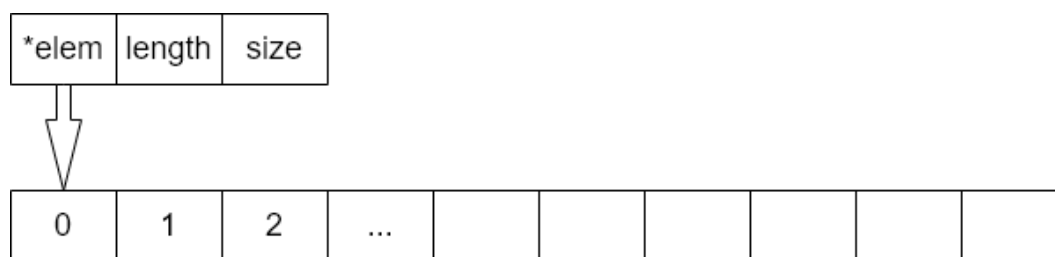


图 1-3 线性表结构体存储状态图

多线性表结构体在内存中的存储状态如图 1.4 所示，图中第一个元素表示多线性表的大小，第二个元素指向一个结构体数组，结构体数组中的每个元素包括一个线性表和一个关键字 (线性表的名字)。

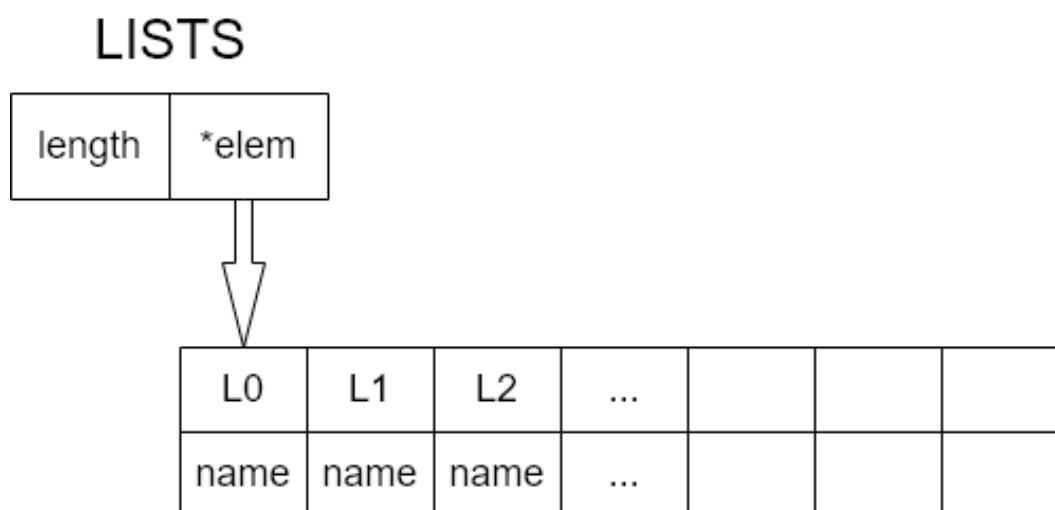


图 1-4 多线性表结构体存储状态图

1.3 系统实现

1.3.1 新建线性表 (InitList)

该函数接受一个 SqList 类型的指针，函数为该指针指向的线性表分配一段长度为 LIST_INIT_SIZE(由宏定义给出值，其值为 100) 的存储空间，并将该线性表的当前长度置为 0，总长度置为 LIST_INIT_SIZE。若存储空间分配成功，则线性表初始化成功，函数返回 OK, 若存储空间分配失败，则线性表初始化失败，

函数返回 OVERFLOW。

该函数只有顺序和选择结构，因此时间复杂度为 $O(1)$ 。

1.3.2 销毁线性表 (DestroyList)

该函数接受一个 SqList 类型的指针，函数将指针指向的线性表的数据存储空间释放，并将数据指针 elem 置为 NULL。并返回 OK。

该函数只有顺序结构，因此其时间复杂度为 $O(1)$ 。

1.3.3 清空线性表 (ClearList)

该函数接受一个 SqList 类型的指针，函数将指针指向的线性表的当前长度 length 置为 0 并返回 OK。若 SqList 指向 NULL，即线性表为空，则返回 INFEASIBLE，表明线性表是空表。

该函数只有顺序结构，因此其时间复杂度为 $O(1)$ 。

1.3.4 线性表判空 (ListEmpty)

该函数接受一个 SqList 类型的值，函数判断 SqList 线性表的当前长度 length 是否为 0，若为 0，说明线性表为空，函数返回 TRUE，否则，说明线性表不为空，函数返回 FALSE。若 SqList 指向 NULL，即线性表为空，则返回 INFEASIBLE，表明线性表是空表。

该函数只有顺序和选择结构，因此其时间复杂度为 $O(1)$ 。

1.3.5 求线性表长度 (ListLength)

该函数接受一个 SqList 类型的值，函数返回 SqList 线性表的当前长度 length。若 SqList 指向 NULL，即线性表为空，则返回 INFEASIBLE，表明线性表是空表。

该函数只有顺序结构，因此其时间复杂度为 $O(1)$ 。

1.3.6 获取元素 (GetElem())

该函数接受三个参数，第一个为 SqList 类型，为待查找的线性表；第二个为 int 类型，为线性表中待查找的元素的位置；第三个为 ElemType 类型的指针，函数将查找到的元素存储到其指向的位置空间。若待查找位置不在线性表元素位

置范围内，则返回 ERROR，否则，将该位置的元素赋值给指针 e 指向的位置并返回 OK。将线性表的线性存储结构视为数组，可通过数组的下标直接方位到线性表的任意一个元素。若 Sqlist 指向 NULL，即线性表为空，则返回 INFEASIBLE，表明线性表是空表。

该函数只有顺序和选择结构，因此其时间复杂度为 $O(1)$ 。

1.3.7 获取元素位置 (LocateElem)

该函数接受两个参数，第一个为 SqList 类型，为待查找的线性表；第二个为 Elemtype 类型，为待查找的元素的值。函数遍历整个线性表，并将其中的元素逐个的与待查找元素的值进行比较，若找到一个相等的元素，则返回该元素的位置，若遍历完整个线性表仍未找到该元素，则返回 ERROR，表明线性表中不存在该元素的情况。若 Sqlist 指向 NULL，即线性表为空，则返回 INFEASIBLE，表明线性表是空表。

该函数包含一个循环结构，在最好的情况下，即线性表中第一个元素即为待查找的元素，则循环执行 1 次，在最坏的情况下，即线性表中最后一个元素为待查找的元素或待查找的元素不在线性表中，则循环执行 n 次 (n 为线性表当前的表长)，因此平均执行次数为 $(n+1)/2$ 次，则函数的时间复杂度为 $O(n)$ 。

1.3.8 查找前驱 (PriorElem)

该函数接受三个参数，第一个为 SqList 类型，为待操作的线性表；第二个为 Elemtype 类型，为待查找其前驱的元素的值；第三个为 Elemtype 类型的指针，其指向的空间用于保存查找到的前驱的值。首先函数判断待查找前驱的元素是否为线性表的第一个元素，若是，则返回 ERROR，不能够查找前驱；否则，函数遍历线性表查找该元素的位置，若找到，则将其前一个位置的值赋给指针 e 所指向的空间，并返回 OK。若未找到该元素，则返回 ERROR，表明线性表中不存在该元素。若 Sqlist 指向 NULL，即线性表为空，则返回 INFEASIBLE，表明线性表是空表。

该函数包含一个循环结构，在待查找前驱的元素为第一个元素时，循环不执行，在待查找元素为第二个元素时，循环执行一次，这两种情况为最好的情况，当待查找前驱的元素为最后一个元素或者不在线性表中时，循环执行 n 次 (n 为线性表当前的表长)，因此平均执行为 $n/2$ 次，则函数的时间复杂度为 $O(n)$ 。

1.3.9 查找后继 (NextElem)

该函数接受三个参数，第一个为 `SqList` 类型，为待操作的线性表；第二个为 `ElemType` 类型，为待查找后继的元素的值；第三个为 `Elemtype` 类型的指针，其指向的空间用于保存查找到的后继的值。首先函数判断待查找后继的元素是否为线性表的最后一个元素，若是，则返回 `ERROR`，不能够查找后继；否则，函数遍历线性表查找该元素的位置，若找到，则将其后一个位置的值赋给指针 `e` 所指向的空间，并返回 `OK`，若未找到该元素，则返回 `ERROR`，表明线性表中不存在该元素。若 `SqList` 指向 `NULL`，即线性表为空，则返回 `INFEASIBLE`，表明线性表是空表。

该函数包含一个循环结构，在待查找后继的元素为线性表中最后一个元素时，循环不执行，在待查找后继的元素为线性表的第一个元素时，循环执行一次，这两种情况为最好的情况，当待查找后继的元素为线性表的倒数第二个元素或者不在线性表中时，平均执行次数为 $n - 1$ 次 (n 为线性表当前的表长)，因此循环平均执行 $(n - 1)/2$ 次，则函数的时间复杂度为 $O(n)$ 。

1.3.10 插入元素 (ListInsert)

该函数接受三个参数，第一个为 `SqList` 类型的指针，为待操作的线性表；第二个为 `int` 类型，为插入元素的位置；第三个为 `Elemtype` 类型，为待插入元素的值。函数首先判断插入位置是否合法，若插入位置小于 1 或比线性表的当前长度大于 1 以上或者线性表当前长度等于线性表的总长度，则返回 `ERROR`，表示不能够插入这个元素；否则，函数从最后一个元素开始，将待插入位置及其后的所有元素向后移动一个单位，然后将待插入元素存储到待插入的位置，将线性表的当前长度 `length` 加 1，并返回 `OK`。若 `SqList` 指向 `NULL`，即线性表为空，则返回 `INFEASIBLE`，表明线性表是空表。

该函数包含一个循环结构，当线性表已满或者输入的插入元素位置不合法时，循环不执行，当插入元素位置为线性表的最后一个位置时，循环执行 1 次，此为最好的情况，当插入元素位置为线性表的第一个位置时，函数执行 n 次，因此平均执行次数为 $n/2$ 次，则函数的时间复杂度为 $O(n)$ 。

1.3.11 删除元素 (ListDelete)

该函数接受三个参数，第一个为 `SqList` 类型的指针，为待操作的线性表；第二个为 `int` 类型，为待删除元素的位置；第三个为 `ElemType` 类型的指针，其指向的位置用于存储删除掉的元素的值。函数首先判断待删除元素的位置是否合法，若该位置小于 1 或大于线性表当前长度，则返回 `ERROR`，表明删除位置超出了线性表的可删除范围，无法删除这个元素；否则，将待删除位置的元素赋给指针 `e` 所指向的空间，从该位置开始将其后所有元素向前移动一个单位，并将线性表的当前长度 `length` 减 1，返回 `OK`。若 `SqList` 指向 `NULL`，即线性表为空，则返回 `INFEASIBLE`，表明线性表是空表。

该函数包含一个循环结构，当用户输入的待删除的位置不存在元素或待删除元素为最后一个元素时，循环执行 0 次，当待删除元素为线性表的倒数第二个元素时，循环执行 1 次，此为最好的情况，当待删除的元素为第一个元素时，循环执行 $n - 1$ 次，因此平均执行次数为 $(n - 1)/2$ 次，则函数的时间复杂度为 $O(n)$ 。

1.3.12 遍历线性表 (ListTraverse)

该函数接受一个参数，为 `SqList` 类型，为待操作的线性表。函数遍历线性表的每一个元素，并将其逐个输出。若 `SqList` 指向 `NULL`，即线性表为空，则返回 `INFEASIBLE`，表明线性表是空表。

该函数包含一个循环结构，该循环必定执行 n 次，故其时间复杂度为 $O(n)$ 。

1.3.13 最大连续子数组和 (MaxSubArray)

该函数接受一个参数，为 `SqList` 类型。该函数从头到尾开始遍历，设置了一个 `ElemType` 的数组，数组元素都初始化为 0。这个数组中的元素的意义是：从开头到这个元素最大的子数组，如果前一个元素小于零，那么这个元素的值就直接等于 `SqList` 中的这个元素的值，否则等于这个数组上一个元素的值加上 `SqList` 这一位的值。最后对这个数组进行遍历，查找最大的元素，返回这个元素。若 `SqList` 指向 `NULL`，即线性表为空，则返回 `INFEASIBLE`，表明线性表是空表。

该函数包含三个循环结构，没有嵌套，因此每个循环各执行了 n 次，因此平均执行次数为 $3n$ ，故时间复杂度为 $O(n)$ ；

1.3.14 寻找相等的子数组和 (SubArrayNum)

该函数接受一个参数，为 SqList 类型。该函数从头到尾开始遍历，设置了一个 ElemType 的数组，数组元素都是 0。这个数组中的元素的意义是：从第 0 位到这一位的元素的总和，这里运用前缀和的方法来求子序列的大小。之后用双重循环遍历这个，两个指针相减便是这两个指针指向元素之间的子序列的总和，设置一个 max 来记录最大的元素，最后返回这个最大元素即可。若 SqList 指向 NULL，即线性表为空，则返回 INFEASIBLE，表明线性表是空表。

该函数包含三个循环结构，有一个双重嵌套结构，因此平均执行次数为 n^2+n 次，因此平均执行次数为 $3n$ ，故时间复杂度为 $O(n)$;

1.3.15 线性表排序 (SortList)

该函数接受一个参数，为 SqList 类型。这里排序采用冒泡排序，每次循环遍历数组，找到需要排列的数组中最大的一个元素，移动到数组的最后，之后循环遍历的次数减少一次，即不用再遍历到这个已经排序好的元素。因此每次遍历的次数都减少 1，从 $n-1$ 一直到 1。若 SqList 指向 NULL，即线性表为空，则返回 INFEASIBLE，表明线性表是空表。

该函数采用了冒泡排序，因此最坏的情况是执行了 $n*(n+1)/2$ 次，因此时间复杂度为 $O(n^2)$;

1.3.16 多线性表管理

本实现方案是由用户指定需要操作的线性表，然后用户可以选择多线性表中的线性表以及一开始独立存在的线性表进行操作。

在使用多线性表中的线性表之前，用户需要新建 (AddList) 线性表，输入新建线性表的关键字 (名字)，程序从用户获取到待操作的线性表的名字 Listname 后，分配一段长度为 SqList 长度的空间，用于存储这个线性表，并保存其首指针，把名字 Listname 赋值给这个线性表的 name，多线性表的长度增加 1，初始化线性表的过程就完成了，用户便可以对这个线性表进行操作了。

除此之外，用户还可以选择删除 (RemoveList) 多线性表中的线性表。用户输入需要删除的线性表的关键字 (名字)，程序从用户获取到待删除线性表的名字 Listname 后，在多线性表中查找 name 与 Listname 相等的线性表，将其删除，

并将后面的线性表都往前移动一位，多线性表的长度减 1。如果未能找到，则返回 ERROR，表示没能找到这个线性表。

同时，还可以在多线性表中定位（LocateList）某个线性表。用户输入待定位线性表的关键字（名字），程序从用户获取到待定位的线性表的名称 Listname 后，在多线性表中查找 name 与 Listname 相等的线性表，如果找到了，就返回当前这个线性表所在的位置，否则返回 ERROR，表示没能找到这个线性表。

在对线性表进行操作的过程中，用户输入想要操作的线性表名称，系统查找到这个线性表并返回这个线性表的位置，之后便使用指针 L 指向待操作的线性表，这样就可以使用与单个线性表的操作时相同的操作来对 L 进行操作，从而实现多线性表操作的功能，其功能与单线性表的功能一致。

多线性表在内存中的存储方式如图 1.5 所示，图中每列表示一个线性表，用户选择一个线性表后，将操作指针指向该线性表所在的地址，从而对该线性表进行操作。

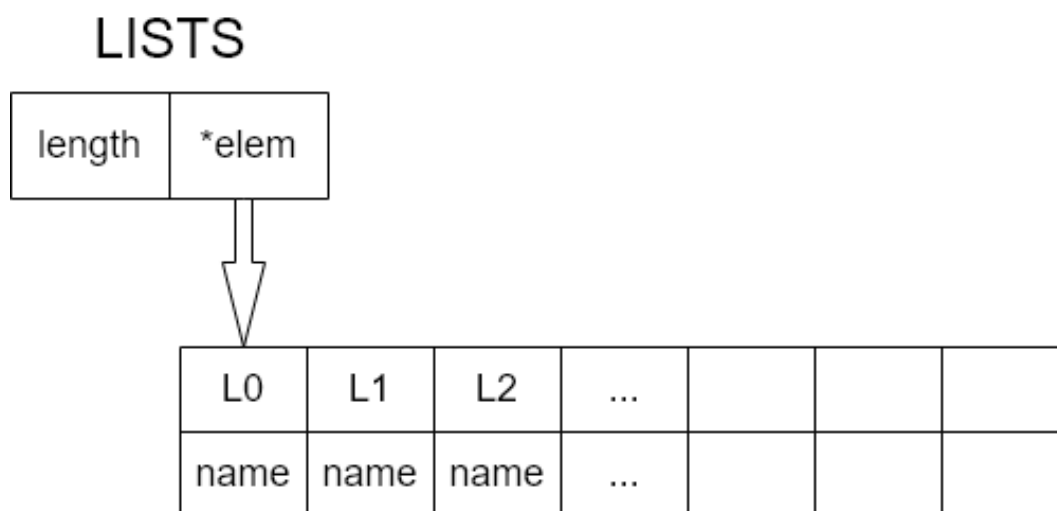


图 1-5 多线性表结构体存储状态图

该函数中需要一个循环遍历多线性表中的关键词，平均执行次数为 $n/2$ ，因此时间复杂度为 $O(n)$ 。

1.3.17 文件的读写

本实现方案对文件输入输出的实现通过两个分支 LoadFile 和 SaveFile 实现，其中 SaveFile 将当前操作的线性表存储到文件，LoadFile 将文件的内容读取并存储到当前操作的线性表中。两个函数的设计如下：

LoadFile 需要文件已存在并且线性表为空或者不存在，否则它会创建一个新的线性表，覆盖掉之前的数据，可能会造成数据丢失。SaveFile 则不需要文件已经存在，如果找不到用户输入的文件名，程序会创建一个同名的文件。

1) 将线性表存储到文件 (SaveFile) 该分支需要由用户输入一个字符串，程序会打开字符串对应的用来存储线性表内容的文件。函数首先打开待使用的文件，若文件已存在，则将文件指针 `fp` 指向该文件，若文件不存在，则创建该文件并将文件指针 `fp` 指向该文件，文件打开后，函数将覆盖文件中原有的内容，并写入其他的内容。若文件打开失败，则没有提示“写入完成”，表明文件未能正常打开。若文件正常打开，则将线性表中的数据逐个的写入到该文件中。文件写入完成后，关闭文件并输出“写入完成”，表明文件写入成功。

该分支需要遍历顺序表中的各个元素，执行了 n 次，因此时间复杂度为 $O(n)$ 。

2) 从文件读取数据并载入线性表 (LoadFile) 该分支需要由用户输入一个字符串，程序会打开字符串对应的用来加载线性表内容的文件。函数首先打开待使用的文件，若文件已存在，则将文件指针 `fp` 指向该文件，若文件不存在，则没有提示“写入完成”，表明文件未能正常打开。文件打开后，从文件中依次读取数据并加载到线性表上，执行 n 次 ListInsert 的操作。文件写入完成后，关闭文件并输出“写入完成”，表明文件写入成功。

该分支需要遍历文件中的各个元素，执行了 n 次，因此时间复杂度为 $O(n)$ 。

1.3.18 线性表转移

本实现方案通过改变指向待操作的线性表的指针 `L` 来实现线性表的转移。程序开始时，这个指针是指向多线性表以外的一个独立存在的线性表，并且用另一个固定的指针指向它，以免以后转移线性表操作指针后丢失。该分支需要由用户输入一个字符串，系统查找到这个线性表并返回这个线性表的位置，之后便使用指针 `L` 指向待操作的线性表，这样就可以使用与单个线性表的操作时相同的操作来对 `L` 进行操作，从而实现多线性表操作的功能，其功能与单线性表的功能一致。如果用户输入 0，则切换回程序开始时创建的那个独立的线性表。

该函数中需要一个循环遍历多线性表中的关键词，平均执行次数为 $n/2$ ，因此时间复杂度为 $O(n)$ 。

1.4 系统测试

完整的程序见附录 1，部分程序在测试中会给出部分代码。

1.4.1 新建线性表 (InitList)

初始化表的正常情况只有一种情况，即待操作的线性表为未初始化的线性表，此时函数返回 OK，程序应输出“线性表创建成功”。

异常情况有两种情况，一种为待操作的线性表未初始化，此时函数不调用，程序输出“线性表已存在”；另一种异常情况为系统内存不足，无法分配足够的空间给线性表使用，此时函数返回 OVERFLOW，程序输出“线性表创建失败”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-1。

	输入	理论结果	运行情况（截图）
正确输入	1（线性表未存在）	输出“线性表创建成功！”	
异常输入	1（线性表已存在）	输出“线性表创建失败！”	

表 1-1 新建线性表程序测试

1.4.2 销毁线性表 (DestroyList)

销毁线性表的正常情况为待操作的线性表已初始化，此时函数返回 OK，程序应输出“线性表销毁成功”。

异常情况为待操作的线性表未初始化，此时函数不调用，程序应输出“线性表不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-2。

	输入	理论结果	运行情况（截图）
--	----	------	----------

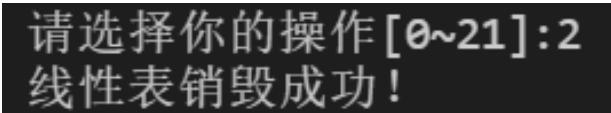
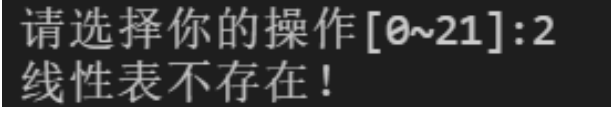
正确输入	2（线性表已存在）	输出“线性表销毁成功！”	
异常输入	2（线性表不存在）	输出“线性表不存在！”	

表 1-2 销毁线性表程序测试

1.4.3 清空线性表（ClearList）

清空线性表的正常情况为待操作的线性表已初始化，此时函数返回 OK，程序应输出“线性表清理成功”。

异常情况为待操作的线性表未初始化，此时函数不调用，程序应输出“线性表不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-3。

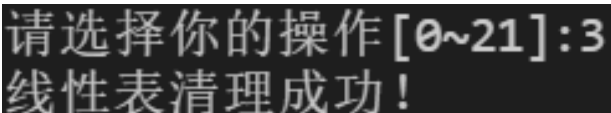
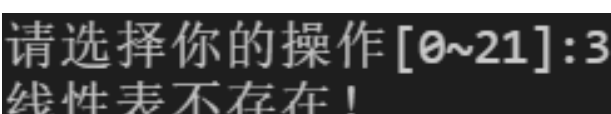
	输入	理论结果	运行情况（截图）
正确输入	3（线性表已存在）	输出“线性表清理成功！”	
异常输入	3（线性表不存在）	输出“线性表不存在！”	

表 1-3 清空线性表程序测试

1.4.4 求线性表长度（ListEmpty）

判断空表的正常情况为待操作的线性表已初始化，此时函数判断线性表是否为空，若为空则函数返回 OK，程序输出“线性表为空”，若不为空则返回 ERROR，

程序输出“线性表不为空”，因此测试时应分为两种正常输入分别测试。

异常情况为待操作的线性表未初始化，此时函数不调用，程序应输出“线性表不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-4。

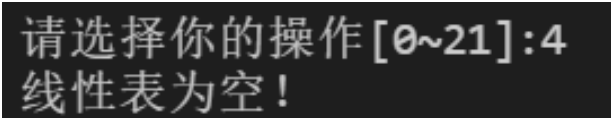
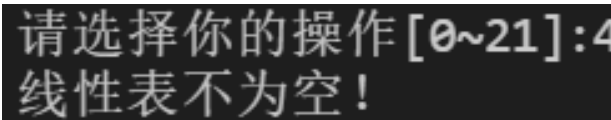
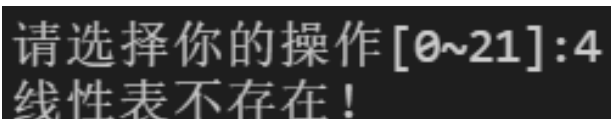
	输入	理论结果	运行情况（截图）
正确输入 1	4（线性表为空）	输出“线性表为空！”	
正确输入 2	4 线性表不为空）	输出“线性表不为空！”	
异常输入	4（线性表不存在）	输出“线性表不存在！”	

表 1-4 求线性表长度程序测试

1.4.5 求线性表长度（ListLength）

获取线性表长度的正常情况为待操作的线性表已初始化，此时函数获取并返回线性表的长度，此处测试时使用空和非空两种线性表进行测试。

异常情况为待操作的线性表未初始化，此时函数不调用，程序输出“线性表不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-5。

	输入	理论结果	运行情况（截图）
--	----	------	----------

正确 输入 1	5（长度为 0）	输出“线 性表长度 为0!”	
正确 输入 2	5（长度为 2）	输出“线 性表长度 为2!”	
异常 输入	5（线性表 不存在）	输出“线 性表不存 在!”	

表 1-5 求线性表长度程序测试

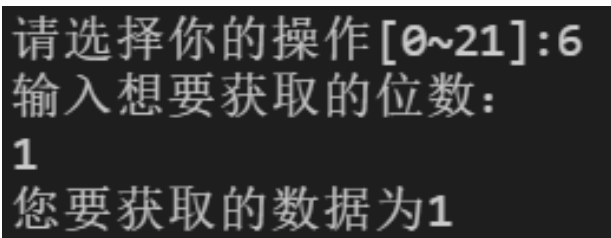
1.4.6 获取元素（GetElem()）

获取线性表元素的正常情况为线性表已存在且待查找位置存在元素，此时函数返回 OK，程序输出待查找元素的值。

异常情况有两种情况，第一种为线性表已存在但待查找位置不存在元素，此时函数返回 ERROR，程序输出“输入错误!”；

第二种为线性表未初始化，此时函数不调用，程序输出“线性表不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-6。本测试中，当线性表存在时，线性表的元素为 1。

	输入	理论结果	运行情况（截图）
正确 输入 1	6 1（查找 第 1 位的）	输出“您 要获取的 数据为 1!”	

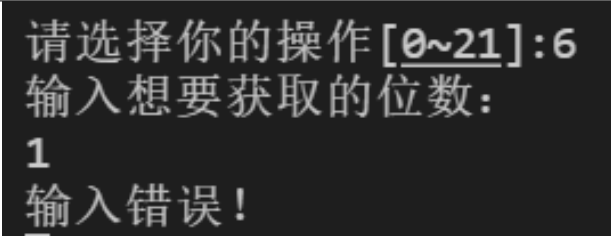
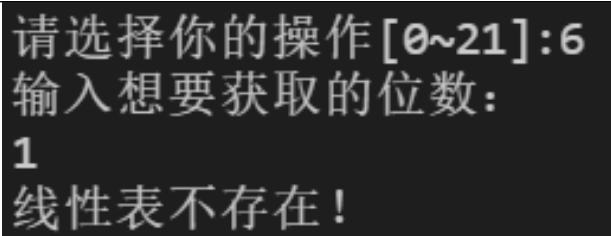
异常 输入 1	6 2（长度 为1）	输出“输入 错误！”	
异常 输入 2	6 1（线性 表不存在）	输出“线性 表不存在 在！”	

表 1-6 获取元素程序测试

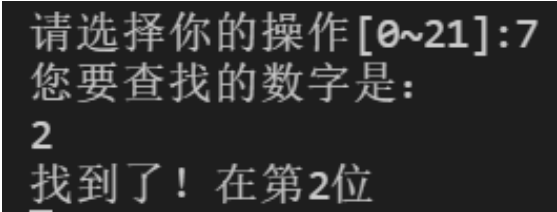
1.4.7 获取元素位置（LocateElem）

查找元素的正常情况为待操作的线性表已存在且待查找的元素在线性表中，此时函数返回该元素在线性表中第一次出现的位置，程序输出该元素在线性表中第一次出现的位置。

异常情况有两种情况，第一种为线性表存在但待查找元素不在线性表中，此时函数返回 ERROR，程序输出“没有找到！”；

第二种为待操作的线性表未初始化，此时函数返回 INFEASIBLE，程序输出“线性表不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-7。本测试中，当线性表存在时，线性表的元素为 1 2 3。

	输入	理论结果	运行情况（截图）
正确 输入 1	7 2（查找 2 的位置）	输出“找 到了！在 第 2 位！”	

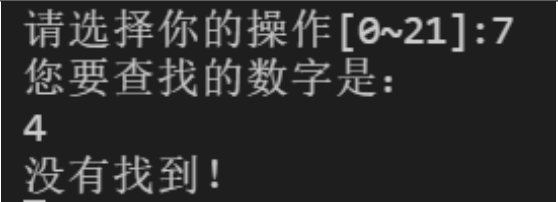
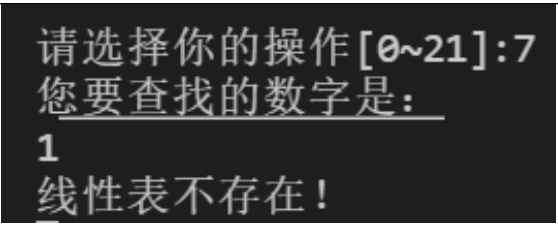
异常 输入 1	7 4 (查找 4 的位置)	输出“没 有找到!”	
异常 输入 2	7 1 (线性 表不存在)	输出“线 性表不存 在!”	

表 1-7 获取元素位置程序测试

1.4.8 查找前驱 (PriorElem)

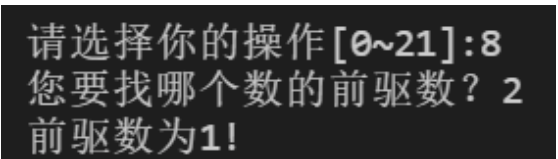
查找前驱的正常情况为待操作的线性表已初始化且待查找前驱的元素在线性表中且该元素不是线性表的第一个元素，此时函数返回 OK，将前驱存到指定位置，程序输出待查找元素的前驱。

异常情况有三种情况，第一种为待查找元素不在线性表中，此时函数返回 ERROR，程序输出未找到您要查找的数的前驱！；

第二种情况为该元素为线性表的第一个元素，此时函数返回 ERROR，程序输出未找到您要查找的数的前驱！；

第三种情况为线性表未初始化，此时函数返回 INFEASIBLE，程序输出“线性表不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-8。本测试中，当线性表存在时，线性表的元素为 1 2。

	输入	理论结果	运行情况（截图）
正确 输入 1	8 2 (查找 2 的位置)	输出“前 驱数为 1!”	

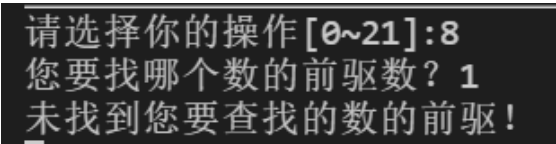
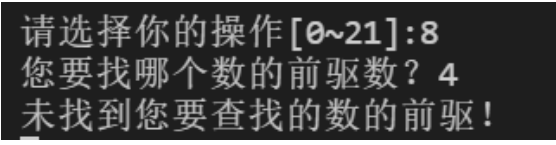
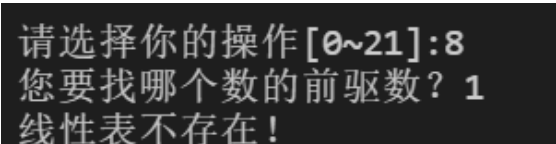
异常 输入 1	8 1（查找 1 的位置）	输出“未 找到您要 查找的数 的前驱！”	
异常 输入 2	8 4（查找 4 的位置）	输出“未 找到您要 查找的数 的前驱！”	
异常 输入 3	8 1（线性 表不存在）	输出“线 性表不存 在！”	

表 1-8 查找前驱程序测试

1.4.9 查找后继（NextElem）

查找后继的正常情况为待操作的线性表已存在且待查找后继的元素在线性表中且该元素不为线性表的最后一个元素，此时函数返回 OK，并将待查找元素的后继存到指定位置，程序输出该元素的后继。

异常情况有三种情况，第一种为线性表存在但待查找后继的元素为线性表的最后一个元素，此时函数返回 ERROR，程序输出未找到您要查找的数的后驱！；

第二种情况为该元素不在线性表中，此时函数返回 INFESTABLE，程序输出未找到您要查找的数的后驱！；

第三种情况为待操作的线性表未初始化，此时函数不调用，程序输出“线性表不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-9。本测试中，当线性表存在时，线性表的元素为 1 2。

	输入	理论结果	运行情况（截图）
--	----	------	----------

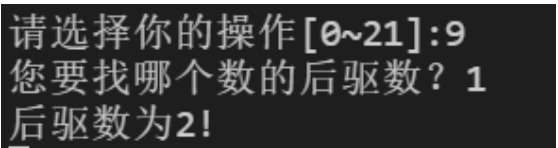
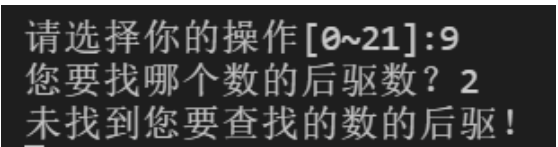
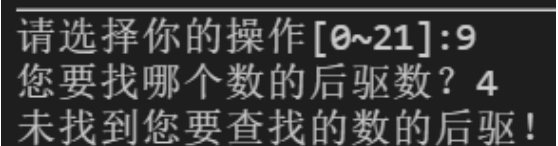
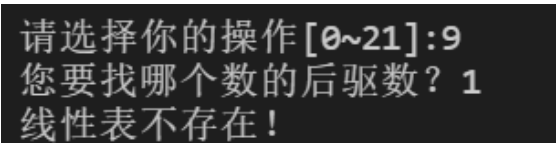
正确 输入 1	9 1（查找 1 的位置）	输出“后 驱数为 1!”	
异常 输入 1	9 2（查找 2 的位置）	输出“未 找到您要 查找的数 的后驱!”	
异常 输入 2	9 4（查找 4 的位置）	输出“未 找到您要 查找的数 的后驱!”	
异常 输入 3	9 1（线性 表不存在）	输出“线 性表不存 在!”	

表 1-9 查找后继程序测试

1.4.10 插入元素（ListInsert）

插入元素的正常情况为线性表已存在且待插入位置可以插入元素，此时函数返回 OK, 程序输出“插入成功”。插入成功可分为两种情况进行测试，分别在线性表的尾部以及中间进行插入操作。

异常情况有三种情况。第一种为线性表存在但待插入位置不能够插入元素，此时函数返回 ERROR，程序输出该位置超出线性表可插入范围；

第二种情况为待操作的线性表已满，无法插入元素，此时函数返回 ERROR，程序输出“线性表已满，不能够插入元素”。

第三种情况为待操作的线性表未初始化，此时函数返回 INFEASIBLE，程序

输出“线性表不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-10。本测试中，在正常输入中，线性表中的元素均为 1 2。

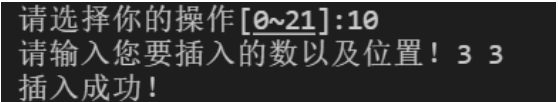
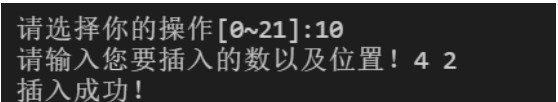
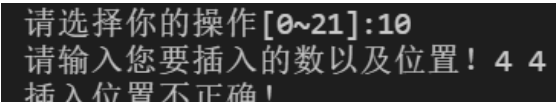
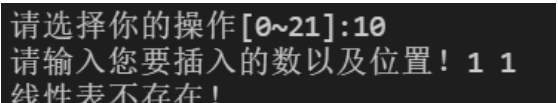
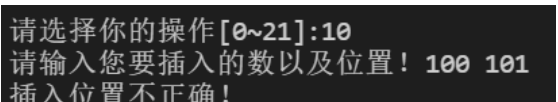
	输入	理论结果	运行情况（截图）
正确输入 1	10 3 3（在位置 3 插入 3）	输出“插入成功！”	
正确输入 2	10 4 2（在位置 2 插入 4）	输出“插入成功！”	
异常输入 1	10 4 4（在位置 4 插入 4）	输出“插入位置不正确！”	
异常输入 2	10 100 101（在位置 101 插入 100）	输出“插入位置不正确！”	
异常输入 3	10 1 1（线性表不存在）	输出“线性表不存在！”	

表 1-10 插入元素程序测试

1.4.11 删除元素（ListDelete）

删除元素的正常情况为待操作的线性表已初始化且待删除位置存在元素，此时函数将删除掉的元素存入指定位置，并返回 OK，程序输出“删除成功”并输出删除的元素的值。

成功删除元素可分为两种情况进行测试，分别为删除首元素和删除中间的

元素。

异常情况有两种情况，第一种为线性表已初始化但待删除位置没有元素，此时函数返回 ERROR，程序输出“删除成功!”;

第二种情况为待操作的线性表未初始化，此时函数返回 INFEASIBLE，程序输出“线性表不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-11。本测试中，在正常输入中，线性表的元素均为 1 2 3 4。

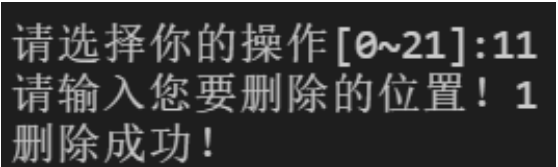
	输入	理论结果	运行情况（截图）
正确 输入 1	11 1（在位置 1 删除数据）	输出“删除成功!”	
正确 输入 2	11 3（在位置 3 删除数据）	输出“删除成功!”	
异常 输入 1	11 10（在位置 10 删除数据）	输出“删除位置不正确!”	
异常 输入 2	11 1（线性表不存在）	输出“线性表不存在!”	

表 1-11 删除元素程序测试

1.4.12 遍历线性表（ListTraverse）

遍历线性表的正常情况为线性表已初始化，此时函数依次打印线性表的元素并返回线性表的长度，若线性表长度为 0，则程序输出“线性表为空表”。

异常情况为线性表未初始化，此时函数返回 INFEASIBLE，程序输出“线性表是空表”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-12。本测试中，线性表存在且不为空时，线性表的元素为 1 2 3 4。

	输入	理论结果	运行情况（截图）
正确输入	12（线性表已存在）	输出“1 2 3 4”	
异常输入	12（线性表不存在）	输出“线性表是空表！”	

表 1-12 新建线性表程序测试

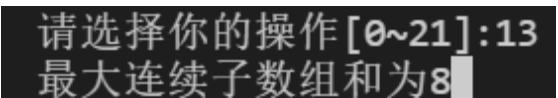
1.4.13 最大子数组和（MaxSubArray）

寻找最大子数组和的正常情况为线性表已初始化，此时函数会对数组中连续的若干个进行求和，返回子数组中和最大的那一个，程序输出最大连续子数组的和。

异常情况有两种情况，第一种为线性表的长度为 0，即表中没有元素，函数返回 ERROR，程序输出“线性表是空表！”；

第二种为线性表未初始化，此时函数返回 INFEASIBLE，程序输出“线性表不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-13。本测试中，线性表存在且不为空时，线性表的元素为 1 -2 3 4 -2 3。

	输入	理论结果	运行情况（截图）
正确输入 1	13（线性表存在）	输出“最大连续子数组和为 8！”	

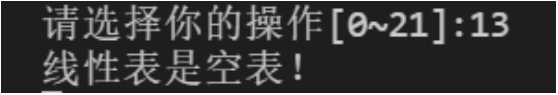
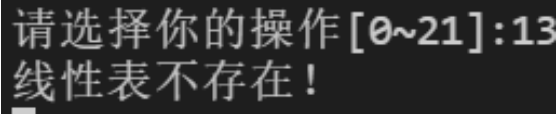
异常输入 1	13（线性表为空）	输出“线性表是空表！”	
异常输入 2	11 1（线性表不存在）	输出“线性表不存在！”	

表 1-13 最大子数组和程序测试

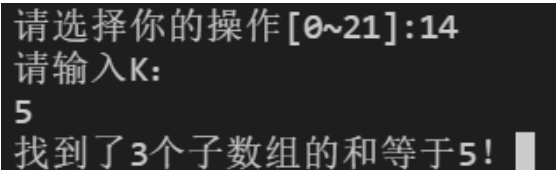
1.4.14 寻找相等的子数组和（SubArrayNum）

寻找相等的子数组和的正常情况为线性表已初始化，用户向程序输入一个数 k ，此时函数会对数组中连续的若干个进行求和，判断这些和中是否有与用户输入的数相等的，如果有，则返回 OK，程序输出“找到了 n 个子数组的和等于 k !”。

异常情况有两种，第一种为没有找到相应的子数组和，函数返回 ERROR，程序输出“找到了 0 个子数组的和等于 k !”；

第二种为线性表未初始化，此时函数返回 INFEASIBLE，程序输出“线性表不存在!”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-14。本测试中，线性表存在且不为空时，线性表的元素为 1 -2 3 4 -2 3。

	输入	理论结果	运行情况（截图）
正确输入 1	14 5（线性表存在）	输出“找到了 3 个子数组的和等于 5!”	

异常 输入 1	14 10（线性表存在）	输出“找到了0个子数组的和等于10!”	
异常 输入 2	14 1（线性表不存在）	输出“线性表不存在!”	

表 1-14 寻找相等的子数组和程序测试

1.4.15 线性表排序（SortList）

线性表排序的正常情况为线性表已初始化，函数会对顺序表中的数据进行从小到大的排序，如果排序成功，函数返回 OK，程序输出“排序完成!”。

异常情况为为线性表未初始化，此时函数返回 INFEASIBLE，程序输出“线性表不存在!”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-15 和表 1-16。本测试中，线性表存在且不为空时，线性表的元素为 4 3 2 1。


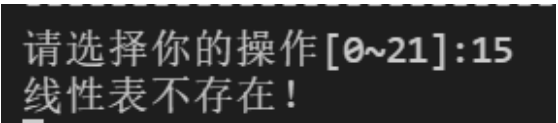
	输入	理论结果	运行情况（截图）
正确 输入	15（线性表已存在）	输出“排序完成!”	
异常 输入	15（线性表不存在）	输出“线性表不存在!”	

表 1-15 线性表排序程序测试

序号	运行前	运行后
1	<pre> -----all elements ----- 4 3 2 1 ----- end ----- </pre>	<pre> -----all elements ----- 1 2 3 4 ----- end ----- </pre>
2	<pre> -----all elements ----- 4 -2 -5 2 4 1 ----- end ----- </pre>	<pre> -----all elements ----- -5 -2 1 2 4 4 ----- end ----- </pre>

表 1-16 线性表排序程序运行前后对比

1.4.16 多线性表管理

多线性表管理包括新增线性表，删除线性表，定位线性表和切换线性表，使用函数和分支实现。

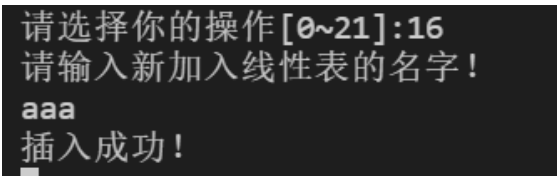
(1) 新增线性表（AddList）

新增线性表的正常情况为，由用户输入新增线性表的关键字（名字）后，程序将从用户获取到待操作的线性表的名字 `ch` 传给函数，函数分配一段长度为 `SqList` 长度的空间，用于存储这个线性表，并保存其首指针，把名字 `ch` 赋值给这个线性表的 `name`，多线性表的长度增加 1，初始化线性表的过程就完成了，函数返回 `OK`，程序输出“插入成功”。

异常情况有两种，第一种为多线性表的长度超过了上限，因此不能再插入线性表，函数返回 `ERROR`，程序输出“插入失败！”。

第二种为用户输入的关键字与已存在的线性表关键字一致，因此不能插入，函数返回 `ERROR`，程序输出“插入失败！”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-17。本测试中，测试 1 时多线性表为空，测试 2 时多线性表已满，测试 3 时多线性表的关键字为 `aaa bbb ccc`。

	输入	理论结果	运行情况（截图）
正确输入 1	16 aaa（多线性表未 满）	输出“插入成功！”	

异常 输入 1	16 kkk（多 线性表已 满）	输出“插 入失败！”	
异常 输入 2	16 aaa（关 键字一致）	输出“插 入失败！”	

表 1-17 新增线性表程序测试

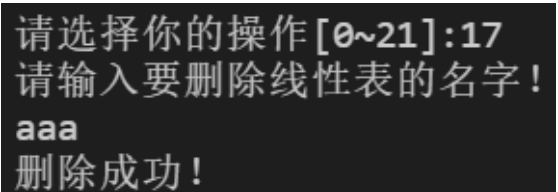
1.4.17 线性表排序（SortList）

(2) 删除线性表（RemoveList）

删除线性表的正常情况为，用户输入新建线性表的关键字（名字）后，程序将名字传给函数，函数在多线性表中查找 name 与 ch 相等的线性表，将其删除，并将后面的线性表都往前移动一位，多线性表的长度减 1，程序输出“删除成功！”。

异常情况为函数未能找到对应名称的线性表，函数返回 ERROR，表示没能找到这个线性表，程序输出“删除失败！”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-18。本测试中，多线性表的关键字均为 aaa aa a。

	输入	理论结果	运行情况（截图）
正确 输入 1	17 aaa（表 中有 aaa）	输出“删 除成功！”	

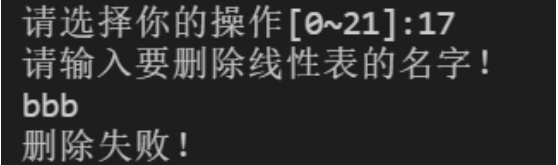
异常输入 1	17 bbb（表中无 bbb）	输出“删除失败！”	
-----------	-----------------	-----------	--

表 1-18 删除线性表程序测试

(3) 定位线性表（LocateList）

定位线性表的正常情况为，用户输入线性表的关键字（名字）后，程序将名字传给函数，函数在多线性表中查找 name 与 ch 相等的线性表，找到后函数返回其位置，程序输出“找到了，在第 n 个！”。

异常情况为函数未能找到对应名称的线性表，函数返回 ERROR，表示没能找到这个线性表，程序输出“没有找到！”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-19。本测试中，多线性表的关键字均为 aaa aa a。

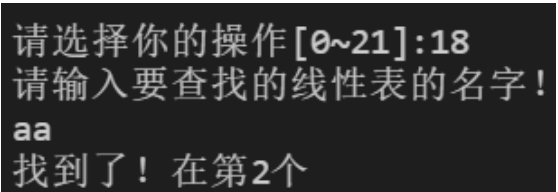
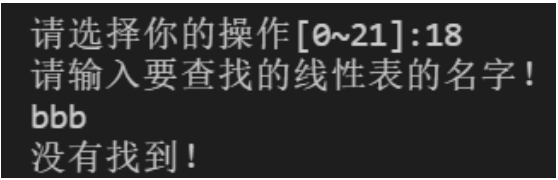
	输入	理论结果	运行情况（截图）
正确输入 1	18 aa（aa 在第二个）	输出“找到了，在第 2 个！”	
异常输入 1	18 bbb（表中无 bbb）	输出“没有找到！”	

表 1-19 定位线性表程序测试

(4) 切换线性表

切换线性表的正常情况有两种，第一种为用户输入想要操作的线性表名称，，系统查找到这个线性表并返回这个线性表的位置，程序输出“操作成功！”。

第二种为用户需要切换回一开始独立存在的单线性表，则输入 0，程序就会

切换回单线性表进行操作，程序输出“操作成功!”。

异常情况为未能找到对应名称的线性表，则程序输出“没有找到!”，表示未能切换成功。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-20。本测试中，多线性表的关键字均为 aaa aa a。

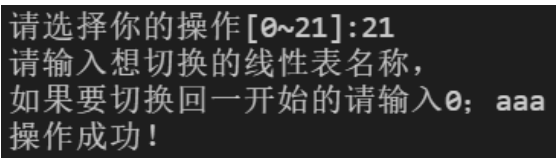
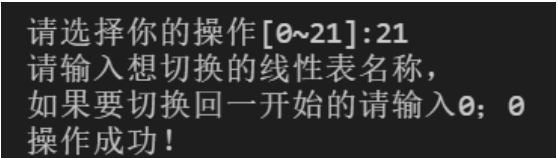
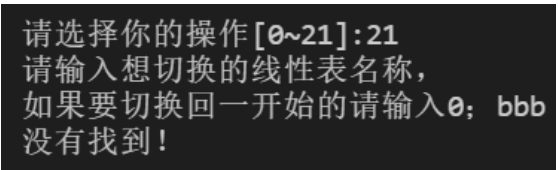
	输入	理论结果	运行情况（截图）
正确 输入 1	21 aaa（多 线性表中 有 aaa）	输出“操 作成功!”	
正常 输入 2	21 0（切换 回初始线 性表）	输出“操 作成功!”	
异常 输入 1	21 bbb（多 线性表中 无 bbb）	输出“没 有找到!”	

表 1-20 切换线性表程序测试

1.4.18 文件的读写

(1) 加载线性表（LoadFile）加载线性表的正常情况为由用户输入存储的文件名，待操作线性表已初始化且文件正常打开，之后程序将文件中的数据读入到线性表中，程序输出“读取成功!”。

异常情况为文件打开失败或者文件不存在，此时不执行读入到线性表的过程，程序输出“文件不存在!”。

第二种为待操作线性表已存在，此时不执行输出到文件的过程，程序输出“线性表已存在!”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-21 和表 1-22。本测试中，存在文件 111.txt，其中数据为 1 2 3 4。不存在文件 222.txt。

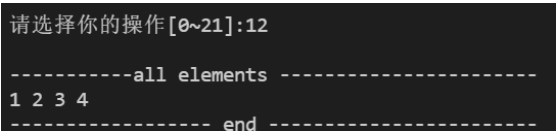
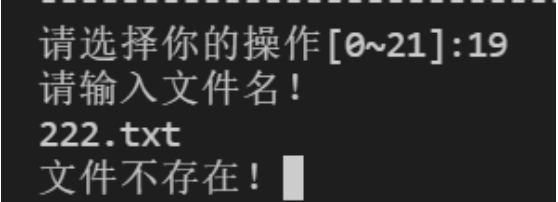
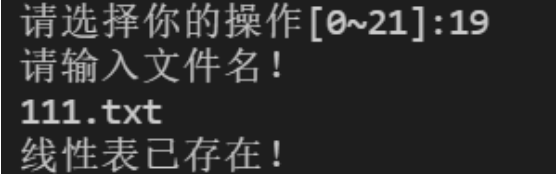
	输入	理论结果	运行情况（截图）
正确 输入 1	19 111.txt (111.txt 存在)	输出“读取成功!”	
异常 输入 1	19 222.txt (222.txt 不存在)	输出“文件不存在!”	
异常 输入 2	19 111.txt (线性表已存在)	输出“线性表已存在!”	

表 1-21 加载线性表程序测试

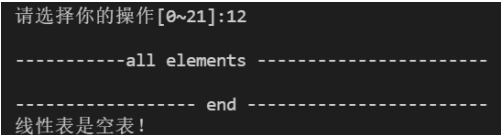
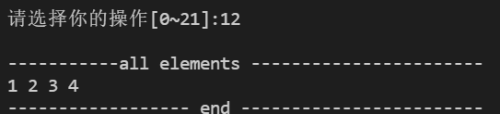
运行前	运行后
	

表 1-22 线性表运行前后对比

(2) 储存线性表 (SaveFile) 储存线性表的正常情况为由用户输入存储的文件名，待操作线性表已初始化且文件正常打开，如果文件不存在程序会创建一个文件，之后程序将数据存入文件中，程序输出“写入成功!”。

异常情况为待操作线性表未初始化，此时不执行输出到文件的过程，程序输出“线性表不存在!”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-23 和表 1-24。本测试中不存在文件 222.txt，线性表中的元素为 1 2 3 4。

	输入	理论结果	运行情况（截图）
--	----	------	----------

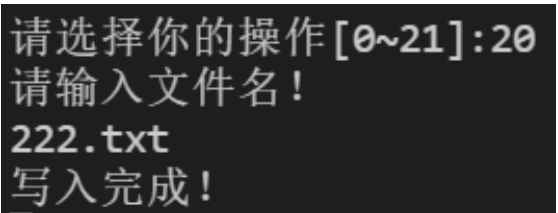
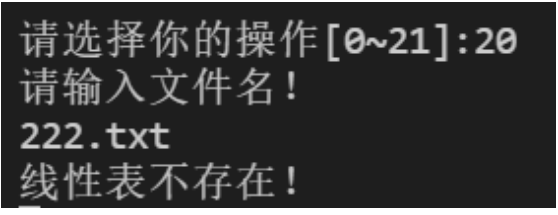
正确 输入 1	20 222.txt (222.txt 不 存在)	输出“写 入成功!”	
异常 输入 1	20 222.txt (线性表不 存在)	输出“线 性表不存 在!”	

表 1-23 储存线性表程序测试

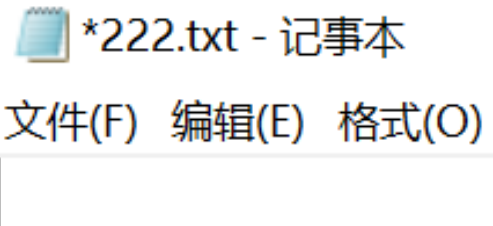
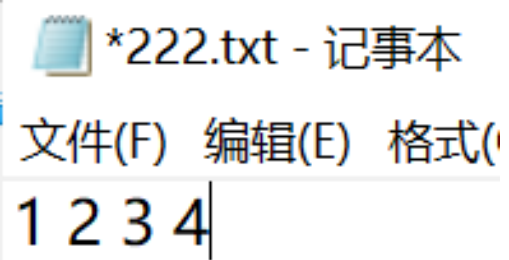
运行前	运行后
	

表 1-24 222.txt 运行前后对比

1.4.19 退出程序

退出程序的功能与线性表的状态无关，只要在根菜单输出 0 便可退出程序，程序输出“欢迎下次再使用本系统!”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-25。

	输入	理论结果	运行情况（截图）
--	----	------	----------

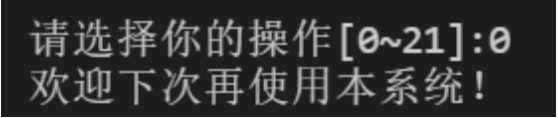
正确 输入	0（退出程 序）	输出“欢 迎下次再 使用本系 统！”	
----------	-------------	-----------------------------	--

表 1-25 储存线性表程序测试

1.5 实验小结

在本次实验中，我遇到了很多的困难和问题，不仅在于程序的编写，还有实现报告的写作，这些都让我有了很多的收获。我从中学到了许多以前不知道的知识。

在编写函数时，需要传递线性表、元素等信息，有些函数还需要用到指针。最开始写的时候有一些参数传递错误，不知道应该传值还是传址，指针方面的知识还不是很牢固，导致函数无法达到预期功能。在调试时也遇到了许多的困难，例如函数无返回值，指针悬挂等，导致一直没有结果输出。还学到了许多排版方面相关的知识，例如写 switch 分支，分支的处理，如何退出分支等，这些都为以后更大的项目打下了基础。

2 基于二叉链表的二叉树实现

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。具体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`；

注：i) 要求 `T` 中各结点关键字具有唯一性。后面各操作的实现，也都要满足一棵二叉树中关键字的唯一性，不再赘述；ii) `CreateBiTree` 中根据 `definition` 生成 `T`，不应在 `CreateBiTree` 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;

特殊情况, c 插入作为根结点, 考虑 LR 为 -1 时, 作为根结点插入, 原根结点作为 c 的右子树。

(10) 删除结点: 函数名称是 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。

注: 前序、中序和后序三种遍历算法, 要求至少一个用非递归算法实现。

(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:

(1) 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节点的最大路径和;

(2) 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；□ 需要设计二叉树文件保存和加载操作合理模式。附录 B 提供了文件存取的方法；

(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。基于顺序表实现的二叉树管理，其物理结构的参考设计如图 2-1 所示。

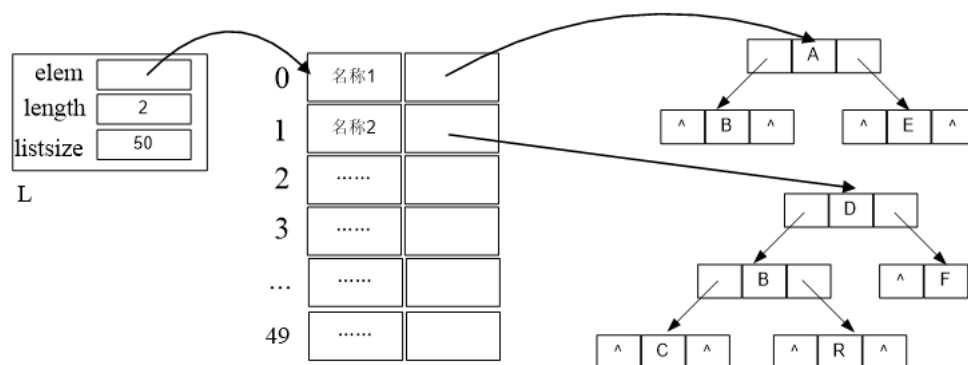


图 2-1 多二叉树的物理结构设计

2.2 系统设计

2.2.1 整体系统结构设计

本实现方案首先进入一个循环，循环中，由用户输入待操作的线性表的序号，然后对二叉树进行操作，进入不同的分支，直到用户输入 0 循环结束，同时算法结束。如果需要切换多二叉树进行操作，则要先创建多二叉树中的二叉树，之后由操作分支切换过去即可。该算法的流程图如图 2.1 所示，整体选择菜单如图 2.2 所示。

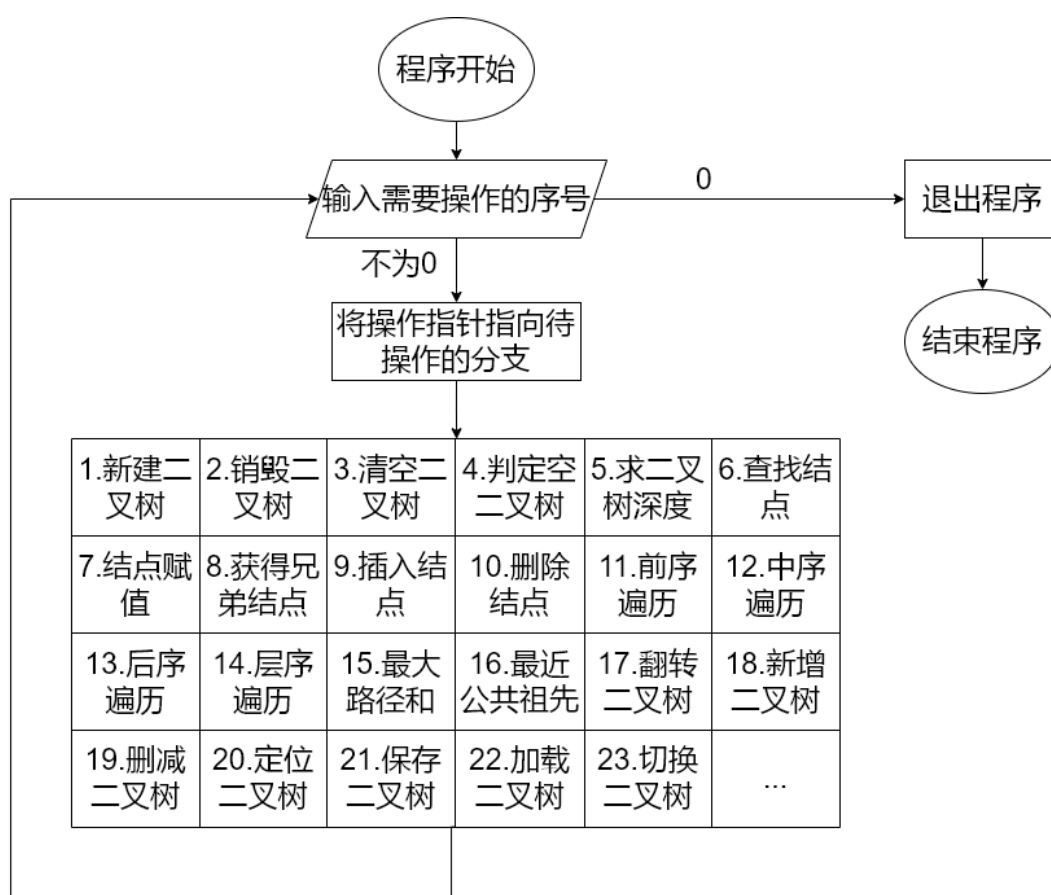


图 2-2 整体系统结构设计

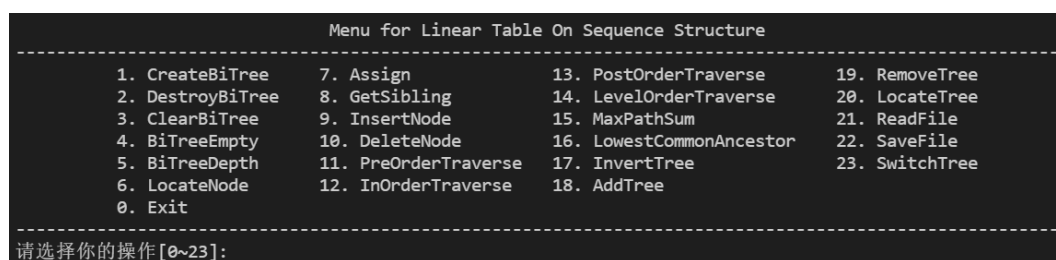


图 2-3 整体选择菜单设计

2.2.2 数据结构设计

(1) 定义头文件。该程序使用了 `<stdio.h>`、`<malloc.h>`、`<stdlib.h>`、`<string.h>` 库, 使用这些 C 语言库可以很方便的进行动态分配结点或者字符串, 以及对字符串进行操作。

(2) 定义常量。一部分常量用来指定函数的返回值, 用于判断函数执行的情

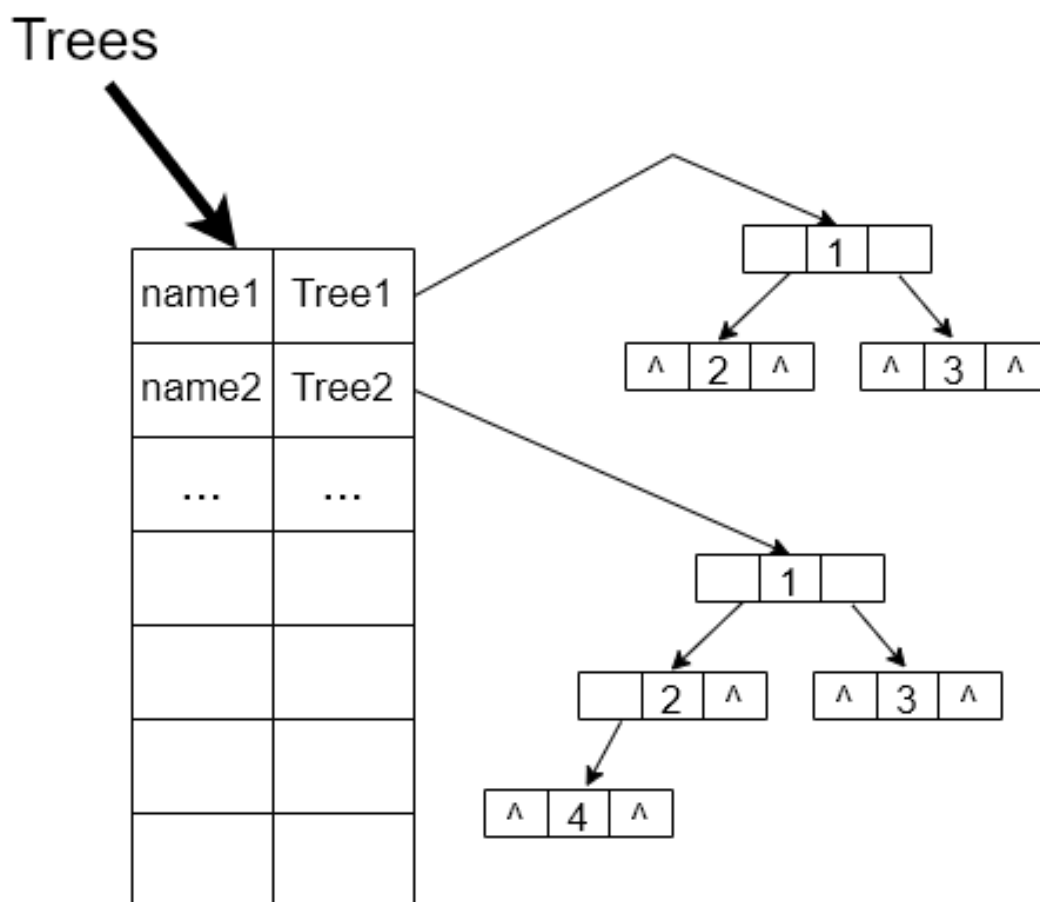


图 2-5 多二叉树结构体存储状态图

2.3 系统实现

关于函数调用方法的说明：创建二叉树（CreateBiTree）和加载二叉树（Load-File）接受的二叉树参数应为未初始化的二叉树。其余 21 个基本操作的函数所接受的二叉树参数应为已初始化的二叉树，否则不调用该函数并提醒用户“二叉树不存在！”。

2.3.1 创建二叉树（CreateBiTree）

该函数接受一个 BiTree 类型的指针 T 以及一个 TElemType 的数组 definition，数组数据由用户输入，接收到数据-1 后停止读入数据。该指针指向待初始化的二叉树的头结点，使用递归方式实现。如果树已存在，则返回 INFEASIBLE，表明该二叉树已存在，否则动态分配 n 个 BiTNode 的空间，分别将 definition 里的元素存入这些结点，之后将根结点赋值给 T，并返回 OK。

该函数需要遍历 definition 数组, 平均执行次数为 n , 因此时间复杂度为 $O(n)$ 。

2.3.2 销毁二叉树 (DestroyBiTree)

该函数接受一个 BiTree 类型的指针 T, 该指针指向待销毁的二叉树的头结点。函数首先遍历树中的所有结点, 将结点全部都用 free 函数释放, 然后将 T 指向 NULL, 并返回 OK。

该函数遍历树的所有结点, 执行次数为 n 次, 故本函数的时间复杂度为 $O(n)$ 。

2.3.3 清空二叉树 (ClearBiTree)

该函数接受一个 BiTree 类型的指针, 该指针指向待清空二叉树的根结点。该函数使用递归方式实现, 若根结点为空, 则函数返回 OK, 否则函数先清空二叉树的左子树, 再清空二叉树的右子树, 然后将根结点的数据 TElemType 的数字更改为 0, 字符串更改为 “”。最后函数返回 OK, 表明二叉树所有结点全部被清空。

该函数将对二叉树的每个结点进行释放, 执行情况为 n 次, 因此函数的时间复杂度为 $O(n)$ 。

2.3.4 判定空二叉树 (BiTreeEmpty)

该函数接受一个 BiTNode 类型的指针 T, 该指针指向待判断的二叉树的头结点, 若 T 指向 NULL, 则说明树不存在, 返回 INFEASIBLE, 否则判断 T 的元素 lchild 以及 rchild 是否为 NULL, 如果是, 则说明二叉树为空, 函数返回 OK, 否则说明二叉树不为空, 函数返回 ERROR。

该函数只包含顺序和选择结构, 因此函数的时间复杂度为 $O(1)$ 。

2.3.5 求二叉树深度 (BiTreeDepth)

该函数接受一个为 BiTree 类型 T, 该指针指向待求深度的二叉树的根结点。该函数使用递归实现。若参数 T 为空, 说明该二叉树不存在, 函数返回 INFEASIBLE, 否则返回二叉树的左子树的深度和右子树的深度的最大值加 1。

该函数会遍历到二叉树的每一个结点, 执行次数为 n 次, 因此函数的时间复杂度为 $O(n)$ 。

2.3.6 查找结点 (LocateNode)

该函数接受两个参数，第一个为 BiTree 类型 T，该指针指向待操作的二叉树的根结点，第二个参数为整型的 e，为待获取结点的 key 值。函数使用递归方式来查找关键字为 e 的结点直到找到其中指向的结点的数据域的关键字为 e 时，终止递归，并返回该指针。若遍历整个数组都未找到带查找结点，则函数返回 NULL。

该函数通过递归会遍历到二叉树的每一个结点，由于查找到结点后就返回，因此平均执行次数为 $n/2$ 次，因此函数的时间复杂度为 $O(n)$ 。

2.3.7 结点赋值 (Assign)

该函数接受三个参数，BiTree 类型的指针 T，该指针指向待操作的二叉树的根结点，第二个参数为整型的 e，为待更改结点的 key 值。第三个参数为 TElemType value，为待赋给结点 e 的值。函数使用递归方式来查找关键字为 e 的结点，直到找到结点的数据域的关键字为 e，则终止递归函数，对这个结点的 value 进行更改，并返回 OK。如果遍历完都没有找到关键字为 e 的结点，则返回 ERROR，表示没有找到关键字为 e 的结点。

该函数通过递归会遍历到二叉树的每一个结点，由于查找到结点后就返回，因此平均执行次数为 $n/2$ 次，因此函数的时间复杂度为 $O(n)$ 。

2.3.8 获得兄弟结点 (GetSibling)

该函数接受两个参数，第一个为 BiTree 类型 T，该指针指向待操作的二叉树的根结点，第二个参数为整型的 e，为待获取结点的 key 值。函数通过递归方式来先查找这个关键字为 e 的结点，每到一个新的结点，则检查它的左孩子和右孩子是否关键字为 e，如果是，则返回它的兄弟结点，否则对它的左右孩子分别调用这个函数。如果都没有找到，就返回 NULL，否则返回找到的结点的指针。

该函数通过递归会遍历到二叉树的每一个结点，由于查找到结点后就返回，因此平均执行次数为 $n/2$ 次，因此函数的时间复杂度为 $O(n)$ 。

2.3.9 插入结点 (InsertNode)

该函数接受四个参数，第一个为 BiTree 类型的指针 T，该指针指向待操作的二叉树的根结点，第二个参数为整型的 e，为待获取结点的 key 值，第三个参数为整型的 LR，为待插入子树的插入位置，第三个参数为 TElemType value，为赋给待创建的新结点的值。

这里插入分为两种情况，第一种情况是 LR 为 -1，则函数需要把新结点作为树的根节点，因此直接创建一个结点，把原先的根节点作为它的右子树，并将左子树置空，返回 OK 即可。

第二种情况是 LR 为 0 或者 1，函数使用递归方式来查找关键字为 e 的结点，直到找到结点的数据域的关键字为 e。如果 LR 为 0，则需要将新的结点插入到这个结点的左孩子处，把这个结点原来的左孩子插入到新的结点的右孩子处；如果 LR 为 1，则需要将新的结点插入到这个结点的右孩子处，把这个结点原来的右孩子插入到新的结点的右孩子处，返回 OK 即可。如果没有查找到这个结点，则返回 ERROR。

该函数通过递归会遍历到二叉树的每一个结点，由于查找到结点后就进行操作，因此平均执行次数为 $n/2$ 次，因此函数的时间复杂度为 $O(n)$ 。

2.3.10 删除结点 (DeleteNode)

该函数接受两个参数，第一个为 BiTree 类型的指针 T，该指针指向待操作的二叉树的根结点，第二个参数为整型的 e，为待删除结点的 key 值。函数使用递归方式来查找关键字为 e 的结点，直到找到结点的数据域的关键字为 e。之后便查找这个结点的双亲结点，如果没找到，则说明这个结点是根节点，删除之后还需要把根节点的指针修改到新的结点，如果找到了，则正常删除即可。删除的过程为：如果关键字为 e 的结点度为 0，直接删除即可；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，这里需要用到一个寻找一个结点子树中最右结点的函数 (FindRightestDnode)，找到之后就把这个 e 的右子树接到 e 的左子树中最右结点，返回 OK；

该函数通过递归会遍历到二叉树的每一个结点，由于查找到结点后就进行操作，可能还要查找一个结点的最右结点，因此平均执行次数为 n 次，因此函数

的时间复杂度为 $O(n)$ 。

2.3.11 前序遍历 (PreOrderTraverse)

该函数接受两个参数，第一个为 BiTree 类型 T，指向待操作二叉树的根结点，第二个为没有返回值、参数为 BiTNode 的指针类型的函数指针 Visit。函数使用递归实现，若根结点指针为空，则函数返回 OK，否则函数先对根结点 T 执行 Visit 函数，然后对 T 的左子树进行前序遍历，然后对 T 的右子树进行前序遍历。最后函数返回 OK。

该函数对二叉树的每个结点调用 Visit 函数，函数 Visit 的时间复杂度为 $O(1)$ ，因此函数的时间复杂度为 $O(n)$ 。

2.3.12 中序遍历 (InOrderTraverse)

该函数接受两个参数，第一个为 BiTree 类型 T，指向待操作二叉树的根结点，第二个为没有返回值、参数为 BiTNode 的指针类型的函数指针 Visit。函数使用递归实现，若根结点指针为空，则函数返回 OK，否则函数先对 T 的左子树进行中序遍历，然后对根结点 T 执行 Visit 函数，然后对 T 的右子树进行中序遍历。最后函数返回 OK。

该函数对二叉树的每个结点调用 Visit 函数，函数 Visit 的时间复杂度为 $O(1)$ ，因此函数的时间复杂度为 $O(n)$ 。

2.3.13 后序遍历 (PostOrderTraverse)

该函数接受两个参数，第一个为 BiTree 类型 T，指向待操作二叉树的根结点，第二个为没有返回值、参数为 BiTNode 的指针类型的函数指针 Visit。函数使用递归实现，若根结点指针为空，则函数返回 OK，否则函数先对 T 的左子树进行后序遍历，然后对 T 的右子树进行后序遍历，然后对根结点 T 执行 Visit 函数。最后函数返回 OK。

该函数对二叉树的每个结点调用 Visit 函数，函数 Visit 的时间复杂度为 $O(1)$ ，因此函数的时间复杂度为 $O(n)$ 。

2.3.14 按层遍历 (LevelOrderTraverse)

该函数接受两个参数，第一个为 `BiTNode` 类型的指针 `T`，指向待操作二叉树的根结点，第二个为没有返回值、参数为 `BiTNode` 的指针类型的函数指针 `Visit`。函数使用栈来实现遍历，首先先把根结点 `T` 放到队列中，之后对队列开始循环直到栈空。循环过程为：先将队列第一个结点弹出，当一个结点为中间结点时，先用 `Visit` 访问这个结点，会将它不为空的孩子推入队列中；当一个结点为叶子结点时，则直接用 `Visit` 访问后结束这个结点的操作。

该函数对二叉树的每个结点调用 `Visit` 函数，函数 `Visit` 的时间复杂度为 $O(1)$ ，因此函数的时间复杂度为 $O(n)$ 。

2.3.15 最大路径和 (MaxPathSum)

该函数接受一个参数，为 `BiTree` 类型 `T`，指向待操作二叉树的根结点，之后通过递归返回左结点的最大路径和和右结点的最大路径和的较大者加上这个结点的 `key` 值。如果根节点为空，则说明树还没有初始化，因此函数返回 `ERROR`。

该函数通过递归会遍历到二叉树的每一个结点，执行次数为 n 次，因此函数的时间复杂度为 $O(n)$ 。

2.3.16 最近公共祖先 (LowestCommonAncestor)

该函数接受三个参数，第一个为 `BiTNode` 类型的指针 `T`，指向待操作二叉树的根结点，第二个和第三个都是结点的关键字，分别为 `u` 和 `v`。函数运用递归和辅助函数 (`FindKey`) 实现，辅助函数接受两个参数，第一个为 `BiTNode` 类型的指针 `T`，指向待操作二叉树的结点，第二个为待查找的关键字，如果找到了则返回 1，否则返回 0。先对这个结点的左子树查找 `u` 和右子树查找 `v`，如果都找到了，那么这个结点就是这两个结点的最近公共祖先，返回这个结点的指针；否则就对左子树查找 `v` 和右子树查找 `u`，如果都找到了，那么这个结点就是这两个结点的最近公共祖先，返回这个结点的指针。如果这两个都失败了，则对这个结点的左孩子和右孩子分别调用这个函数，如果他们返回的不是 `NULL`，则说明他们中有找到最近公共祖先，那么就返回他们返回的值即可。

该函数通过递归会遍历到二叉树的每一个结点，还会遍历它的左子树和右子树，平均执行次数为 $n^2/2$ 次，因此函数的时间复杂度为 $O(n^2)$ 。

2.3.17 翻转二叉树 (InvertTree)

该函数接受一个参数，为 BiTree 类型 T，指向待操作二叉树的根结点，之后通过递归对每一个结点进行翻转操作，操作如下：设置一个暂存结点来存放左孩子结点，把右孩子结点赋给左孩子结点，再把这个暂存结点赋给右孩子结点，再对这个结点的左孩子和右孩子分别调用这个函数，返回 OK。

该函数通过递归会遍历到二叉树的每一个结点，执行次数为 $2n$ 次，因此函数的时间复杂度为 $O(n)$ 。

2.3.18 实现多个二叉树管理

本实现方案是由用户指定需要操作的二叉树，然后用户可以选择多二叉树中的二叉树以及一开始独立存在的二叉树进行操作。

在使用多线性表中的线性表之前，用户需要新建 (AddTree) 二叉树，输入新建二叉树的关键字 TreeName (名字) 程序从用户获取到待创建的二叉树的名字 TreeName 后，分配一段长度为 BiTree 长度的空间，用于存储这个二叉树，并保存其首指针，把名字 TreeName 赋值给这个二叉树的 name，多二叉树的长度增加 1，初始化线性表的过程就完成了，用户便可以对这个线性表进行操作了。

除此之外，用户还可以选择删除 (RemoveList) 多线性表中的线性表。用户输入需要删除的二叉树的关键字 TreeName (名字) 程序从用户获取到待删除二叉树的名字 TreeName 后，在多二叉树中查找 name 与 TreeName 对应的二叉树将其删除，并将后面的二叉树都往前移动一位，多二叉树的长度减 1。如果未能找到，则返回 ERROR，表示没能找到这个二叉树。

同时，还可以在多线性表中定位 (LocateList) 某个二叉树。用户输入新建二叉树的关键字 TreeName (名字) 程序从用户获取到要查找二叉树的名字 TreeName 后，在多二叉树中查找 name 与 TreeName 相等的二叉树，如果找到了，就返回当前这个二叉树所在的位置，否则返回 ERROR，表示没能找到这个二叉树。

在对二叉树进行操作的过程中，用户输入想要操作的二叉树名称，系统查找到这个二叉树并返回这个二叉树的位置，之后便使用指针 L 指向待操作的二叉树，这样就可以使用与单个二叉树的操作时相同的操作来对 L 进行操作，从而实现多二叉树操作的功能，其功能与单二叉树的功能一致。

多二叉树在内存中的存储方式如图 2.5 所示，图中每列表示一个线性表，用

户选择一个线性表后，将操作指针指向该线性表所在的地址，从而对该线性表进行操作。

该函数中需要一个循环遍历多线性表中的关键词，平均执行次数为 $n/2$ ，因此时间复杂度为 $O(n)$ 。

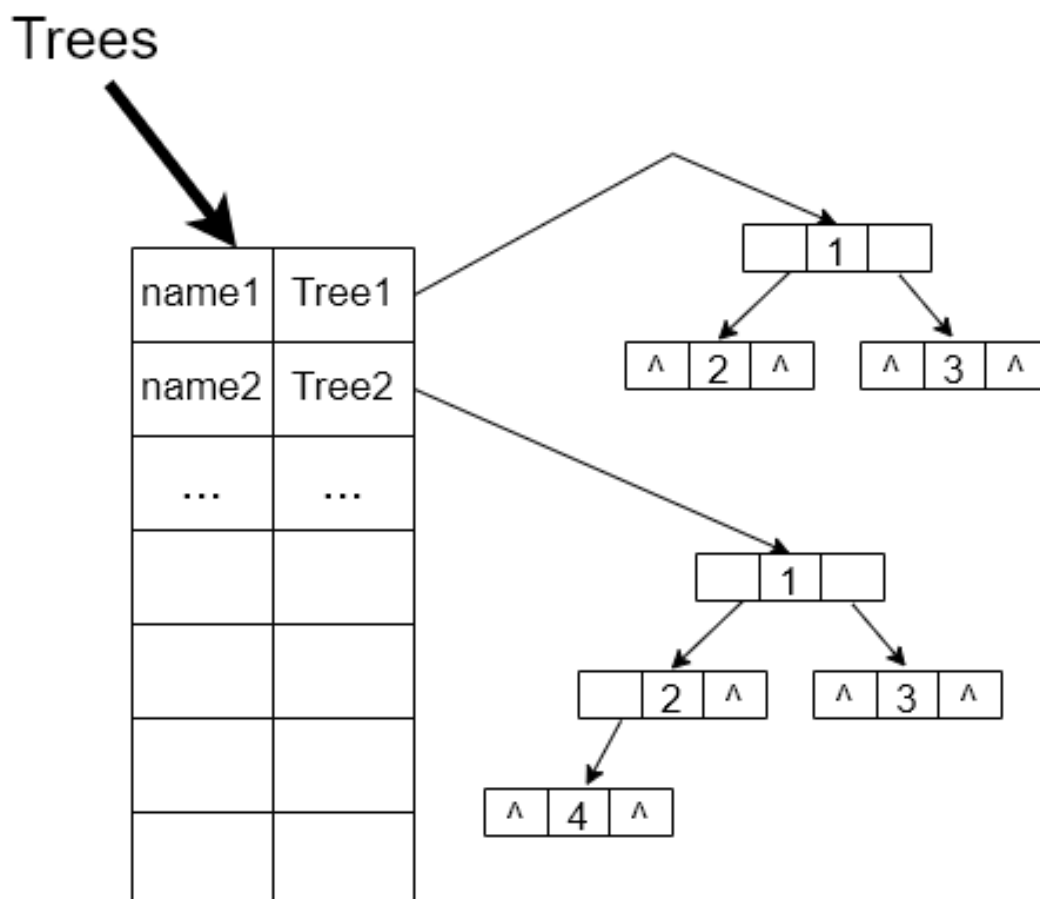


图 2-6 多二叉树结构体存储状态图

2.3.19 二叉树的文件读写

本实现方案对文件输入输出的实现通过两个分支 LoadFile 和 SaveFile 实现，其中 SaveFile 将当前操作的线性表存储到文件，LoadFile 将文件的内容读取并存储到当前操作的线性表中。两个函数的设计如下：

LoadFile 需要文件已存在并且二叉树为空或者不存在，否则它会创建一个新的二叉树，覆盖掉之前的数据，可能会造成数据丢失。SaveFile 则不需要文件已经存在，如果找不到用户输入的文件名，程序会创建一个同名的文件。

1) 将线性表存储到文件 (SaveFile) 该分支需要由用户输入一个字符串，程序

会打开字符串对应的用来存储二叉树内容的文件。函数首先打开待使用的文件，若文件已存在，则将文件指针 `fp` 指向该文件，若文件不存在，则创建该文件并将文件指针 `fp` 指向该文件，文件打开后，函数将覆盖文件中原有的内容，并写入其他的内容。若文件打开失败，则没有提示“写入完成”，表明文件未能正常打开。若文件正常打开，则将线性表中的数据逐个的写入到该文件中。文件写入完成后，关闭文件并输出“写入完成”，表明文件写入成功。

该分支需要遍历二叉树中的各个结点，执行了 n 次，因此时间复杂度为 $O(n)$ 。

2) 从文件读取数据并载入二叉树 (`LoadFile`) 该分支需要由用户输入一个字符串，程序会打开字符串对应的用来加载二叉树内容的文件。函数首先打开待使用的文件，若文件已存在，则将文件指针 `fp` 指向该文件，若文件不存在，则没有提示“写入完成”，表明文件未能正常打开。文件打开后，从文件中依次读取数据并加载到二叉树上。文件写入完成后，关闭文件并输出“写入完成”，表明文件写入成功。

该分支需要遍历文件中的各个元素，执行了 n 次，因此时间复杂度为 $O(n)$ 。

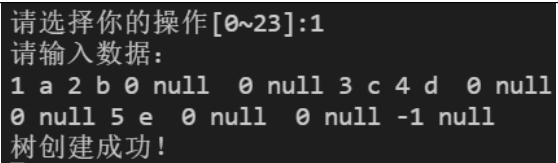
2.4 系统测试

完整的程序见附录 1，部分程序在测试中会给出部分代码。

2.4.1 创建二叉树 (`CreateBiTree`)

创建二叉树的正常情况只有一种情况，即二叉树还不存在，此时函数根据用户输入的数据创建二叉树，并返回 `OK`，程序输出“树创建成功!”。

异常情况有一种，即带创建的二叉树已经存在，此时函数返回 `INFEASIBLE`，程序输出“树已存在”。具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-1。

	输入	理论结果	运行情况（截图）
正确输入	1（二叉树未存在）	输出“树创建成功!”	

异常输入	1（二叉树已存在）	输出“树已存在！”	<pre> 请选择你的操作[0~23]:1 请输入数据: 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null 树已存在! </pre>
------	-----------	-----------	---

表 2-1 新建线性表程序测试

2.4.2 销毁二叉树（DestroyBiTree）

销毁二叉树的正常情况只有一种情况，即待操作的二叉树为已初始化的二叉树，此时函数将二叉树销毁，并返回 OK，程序应输出“二叉树销毁成功”。

异常情况也有一种情况，即待操作的二叉树为未初始化的二叉树，此时函数不调用，程序输出“二叉树不存在”。

具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-2。

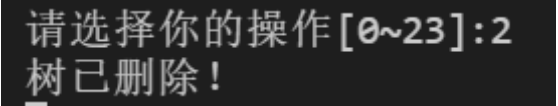
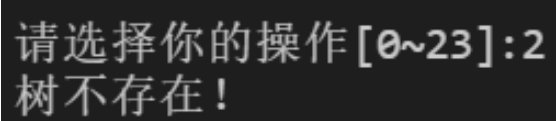
	输入	理论结果	运行情况（截图）
正确输入	2（二叉树未存在）	输出“树创建成功！”	
异常输入	2（二叉树已存在）	输出“树已存在！”	

表 2-2 销毁二叉树程序测试

2.4.3 清空二叉树（ClearBiTree）

该函数的正常情况只有一种情况，即待清空的二叉树已初始化，此时函数清空二叉树并返回 OK。程序应输出“树清空成功”。

异常情况也有一种情况，即待清空的二叉树未初始化，此时函数不调用，程序输出“树不存在”。

具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-3。

	输入	理论结果	运行情况（截图）
--	----	------	----------

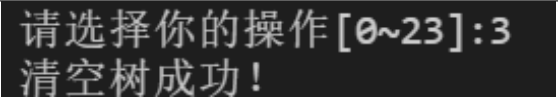
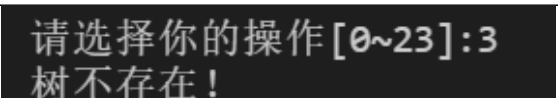
正确输入	3（二叉树未存在）	输出“清空树成功！”	
异常输入	3（二叉树已存在）	输出“树不存在！”	

表 2-3 清空二叉树程序测试

2.4.4 判定空二叉树（BiTreeEmpty）

该函数的正常情况有两种情况，第一种为二叉树为空树，此时函数返回 TRUE，程序输出“二叉树为空树”；第二种情况为二叉树不为空树，此时函数返回 FALSE，程序输出“二叉树不是空树”。

异常情况有一种情况，即二叉树为初始化，此时函数不调用，程序输出“二叉树不存在”。

具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-4。




	输入	理论结果	运行情况（截图）
正确输入 1	4（二叉树为空）	输出“树为空！”	
正确输入 2	4（二叉树不为空）	输出“树不为空！”	
异常输入	4（二叉树已存在）	输出“树不存在！”	

表 2-4 判定空二叉树程序测试

2.4.5 求二叉树深度 (BiTreeDepth)

该函数的正常情况有一种情况，即二叉树已初始化，此时函数返回二叉树的深度，程序输出二叉树的深度的值。本测试中分为两种情况进行测试，分别为二叉树为空树和二叉树不为空树的情况。

异常情况有一种情况，即二叉树未初始化，此时函数不调用，程序输出“二叉树不存在”。

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1 a), (2 b), (0 null), (0 null), (3 c), (4 d), (0 null), (0 null), (5 e), (0 null), (0 null), (-1 null)。具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-5。

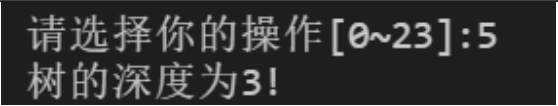
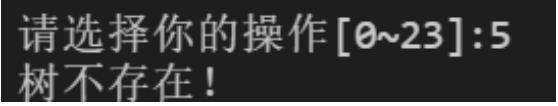
	输入	理论结果	运行情况（截图）
正确输入	5（二叉树已存在）	输出“树的深度为3！”	
异常输入	5（二叉树未存在）	输出“树不存在！”	

表 2-5 求二叉树深度程序测试

2.4.6 查找结点 (LocateNode)

该函数的正常情况有一种情况，即二叉树已初始化且待获得的结点在二叉树中，此时函数返回该结点的指针，程序输出该结点的 key 和 name。

异常情况有两种情况，第一种为二叉树已初始化，但是待获得的结点不在二叉树中，此时函数返回 NULL，程序输出“未找到！”；

第二种情况为二叉树未初始化，此时函数不调用，程序输出“未找到！”。

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1 a), (2 b), (0 null), (0 null), (3 c), (4 d), (0 null), (0 null), (5 e), (0 null), (0 null), (-1 null)。具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-6。

	输入	理论结果	运行情况（截图）
--	----	------	----------

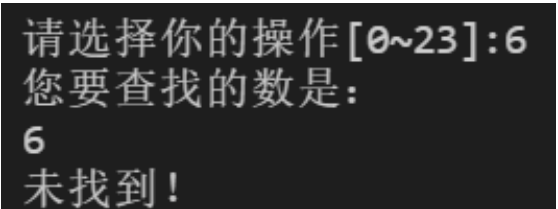
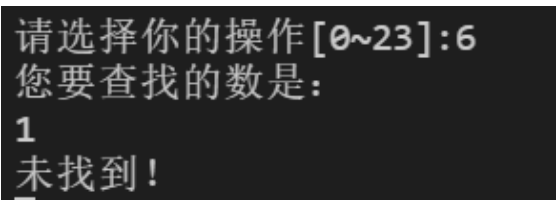
正确 输入 1	6（二叉树 为空）	输出“树 为空！”	
异常 输入 1	6（树中没 有6）	输出“未 找到！”	
异常 输入 2	6（二叉树 不存在）	输出“未 找到！”	

表 2-6 查找结点程序测试

2.4.7 结点赋值（Assign）

该函数的正常情况有一种情况，即二叉树已初始化且待赋值的结点在二叉树中，此时函数返回 OK，程序输出“给该结点赋值成功”。

异常情况有两种情况，第一种为二叉树已初始化但是待赋值的结点不在二叉树中，此时函数返回 ERROR，程序输出“二叉树中不存在该结点”；

第二种情况为二叉树未初始化，此时函数不调用，程序输出“二叉树不存在”。

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1 a), (2 b), (0 null), (0 null), (3 c), (4 d), (0 null), (0 null), (5 e), (0 null), (0 null), (-1 null)。具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-7。

	输入	理论结果	运行情况（截图）
--	----	------	----------

正确 输入 1	7 1 6 aaa (二叉树中有 1)	输出“赋值成功!”	<pre> 请选择你的操作[0~23]:7 要赋值的节点是: 1 赋值的点和名称是: 6 aaa 赋值成功! </pre>
异常 输入 1	7 6 7 bbb (树中没有 6)	输出“赋值失败!”	<pre> 请选择你的操作[0~23]:7 要赋值的节点是: 6 赋值的点和名称是: 7 bbb 赋值失败! </pre>
异常 输入 2	7 1 2 7 (二 叉树不存在)	输出“树不存在!”	<pre> 请选择你的操作[0~23]:7 要赋值的节点是: 1 赋值的点和名称是: 2 7 树不存在! </pre>

表 2-7 结点赋值程序测试

2.4.8 获得兄弟结点 (GetSibling)

该函数的正常情况有一种情况，即二叉树已初始化且要查找的结点在二叉树中且有兄弟结点，此时函数返回 OK，程序输出“兄弟找到了！是”。

异常情况有三种情况，第一种为二叉树已初始化但是要查找的结点不在二叉树中，此时函数返回 ERROR，程序输出“输入错误!”；

第二种情况为这个结点没有兄弟，此时函数返回 ERROR，程序输出“独生子女!”；

第三种情况为二叉树还未初始化，此时函数不调用，程序输出“树不存在!”。

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1

a), (2 b), (0 null), (0 null), (3 c), (4 d), (0 null), (0 null), (5 e), (0 null), (0 null), (-1 null)。具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-8。

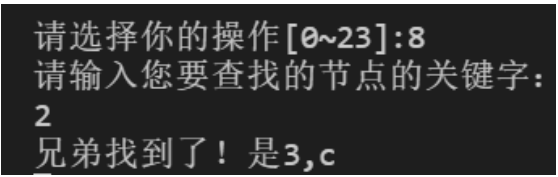
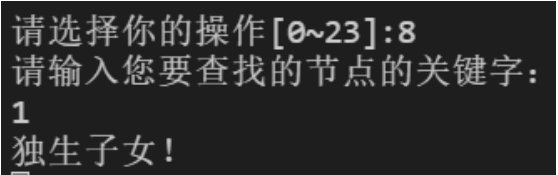

	输入	理论结果	运行情况（截图）
正确输入 1	8 2（二叉树中有 1）	输出“兄弟找到了！是 3,c”	
异常输入 1	8 1（树中 1 无兄弟结点）	输出“独生子女！”	
异常	8 6（树中	输出“输	

表 2-8 获得兄弟程序测试

2.4.9 插入结点 (InsertNode)

该函数的异常情况有两种情况，第一种即二叉树已初始化且待插入子树的结点不是根节点，此时，函数返回 OK，程序输出“”。

第二种为二叉树已初始化且待插入子树的结点是根节点，此时，函数返回 OK，程序输出“”。

异常情况有两种情况，第一种为二叉树已初始化但是待插入子树的结点不在二叉树中，此时函数返回 ERROR，程序输出“”；

第二种情况为二叉树未初始化，此时函数不调用，程序输出“二叉树不存在”。

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1 a), (2 b), (0 null), (0 null), (3 c), (4 d), (0 null), (0 null), (5 e), (0 null), (0 null), (-1 null)。

具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-9。

	输入	理论结果	运行情况（截图）
正确 输入 1	8 2（二叉 树中有 1）	输出“插 入完成 了！”	
异常 输入 1	8 1（树中 1 无兄弟 结点）	输出“插 入完成 了！”	
异常 输入 2	8 6（树中 没有 6）	输出“输 入错误！”	
异常 输入 3	8 1（二叉 树不存在）	输出“树 不存在！”	

表 2-9 插入结点程序测试

2.4.10 删除结点 (DeleteNode)

该函数的正常输出有一种，即二叉树已初始化且待删除子树的结点在二叉树中，此时函数返回 OK，程序输出“删除成功”。

异常输出有两种情况，第一种为二叉树已初始化但是待删除子树的的结点不在二叉树中，此时函数返回 ERROR，程序输出”删除失败”；

第二种情况为二叉树未初始化，此时函数不调用，程序输出“树不存在”。

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1

a), (2 b), (0 null), (0 null), (3 c), (4 d), (0 null), (0 null), (5 e), (0 null), (0 null), (-1 null)。
具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-10。

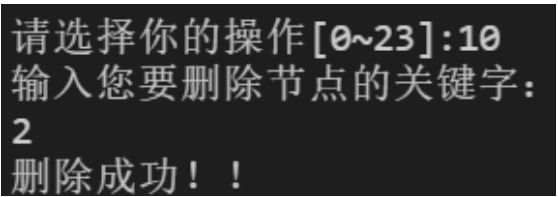
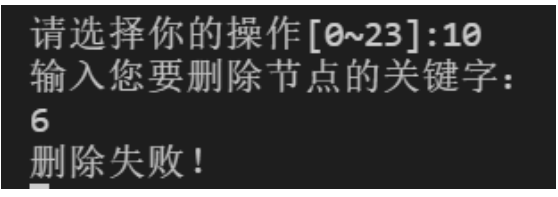

	输入	理论结果	运行情况（截图）
正确 输入 1	10 2（二 叉树中有 2）	输出“删 除成功！”	
异常 输入 1	10 1（树 中没有 6）	输出“删 除失败！”	
异常 输入 2	10（二叉 树不存在）	输出“树 不存在！”	


表 2-10 删除结点程序测试

2.4.11 前序遍历（PreOrderTraverse）

该函数的正常情况有一种，即二叉树已初始化，此时函数返回 OK，程序输出二叉树的前序遍历序列。

异常情况有一种，即二叉树未初始化，此时函数不调用，程序输出“树不存在”。

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1 a), (2 b), (0 null), (0 null), (3 c), (4 d), (0 null), (0 null), (5 e), (0 null), (0 null), (-1 null)。
具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-11。

	输入	理论结果	运行情况（截图）
正确 输入 1	11（二叉 树存在）	如图中输 出结果	

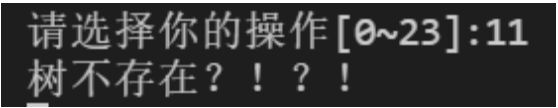
异常 输入 1	11（二叉 树不存在）	输出“树 不存在！”	
---------------	----------------	---------------	--

表 2-11 前序遍历程序测试

2.4.12 中序遍历 (InOrderTraverse)

该函数的正常情况有一种，即二叉树已初始化，此时函数返回 OK，程序输出二叉树的中序遍历序列。

异常情况有一种，即二叉树未初始化，此时函数不调用，程序输出“树不存在”。

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1 a), (2 b), (0 null), (0 null), (3 c), (4 d), (0 null), (0 null), (5 e), (0 null), (0 null), (-1 null)。具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-12。


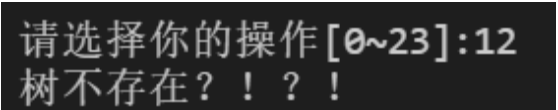
	输入	理论结果	运行情况（截图）
正确 输入 1	12（二叉 树存在）	如图中输出结果	
异常 输入 1	12（二叉 树不存在）	输出“树 不存在！”	

表 2-12 中序遍历程序测试

2.4.13 后序遍历 (PostOrderTraverse)

该函数的正常情况有一种，即二叉树已初始化，此时函数返回 OK，程序输出二叉树的后序遍历序列。

异常情况有一种，即二叉树未初始化，此时函数不调用，程序输出“树不存在”。

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1 a), (2 b), (0 null), (0 null), (3 c), (4 d), (0 null), (0 null), (5 e), (0 null), (0 null), (-1 null)。具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-13。

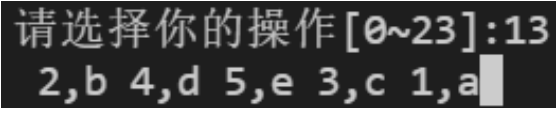
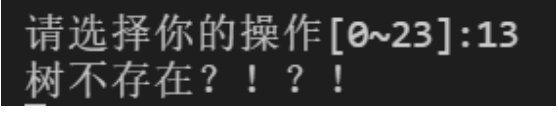
	输入	理论结果	运行情况（截图）
正确 输入 1	13（二叉 树存在）	如图中输出 结果	
异常 输入 1	13（二叉 树不存在）	输出“树 不存在！”	


表 2-13 后序遍历程序测试

2.4.14 按层遍历（LevelOrderTraverse）

该函数的正常情况有一种，即二叉树已初始化，此时函数返回 OK，程序输出二叉树的层序遍历序列。

异常情况有一种，即二叉树未初始化，此时函数不调用，程序输出“树不存在”。

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1 a), (2 b), (0 null), (0 null), (3 c), (4 d), (0 null), (0 null), (5 e), (0 null), (0 null), (-1 null)。具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-14。

	输入	理论结果	运行情况（截图）
正确 输入 1	14（二叉 树存在）	如图中输出 结果	

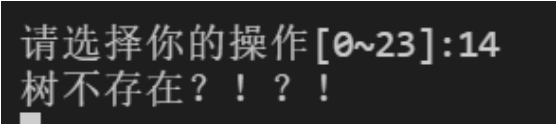
异常输入 1	14（二叉树不存在）	输出“树不存在！”	
-----------	------------	-----------	--

表 2-14 后序遍历程序测试

2.4.15 最大路径和 (MaxPathSum)

该函数的正常情况有一种，即二叉树已初始化，此时函数返回树的最大路径，程序输出最大路径和。

异常情况有一种，即二叉树未初始化，此时函数不调用，程序输出“树不存在”。

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1 a), (2 b), (0 null), (0 null), (3 c), (4 d), (0 null), (0 null), (5 e), (0 null), (0 null), (-1 null)。具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-15。

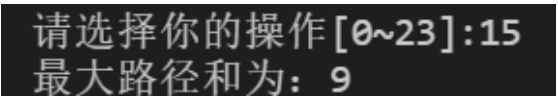

	输入	理论结果	运行情况（截图）
正确输入 1	15（二叉树存在）	输出“最大路径和为：9”	
异常输入 1	15（二叉树不存在）	输出“树不存在！”	

表 2-15 最大路径和程序测试

2.4.16 最近公共祖先 (LowestCommonAncestor)

该函数的正常情况有一种，即二叉树已初始化，此时函数返回树的最大路径，程序输出“找到了！它的关键字是”。

异常情况有两种，第一种为用户输入的结点无法在树中找到，此时函数不调用，程序输出“没找到哦！”；

第二种情况即二叉树未初始化，此时函数不调用，程序输出“树不存在”。

本测试中，当二叉树存在且不为空时，二叉树的含空的后序遍历结果为 (1 a), (2 b), (0 null), (0 null), (3 c), (4 d), (0 null), (0 null), (5 e), (0 null), (0 null), (-1 null)。具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-16。

	输入	理论结果	运行情况（截图）
正确输入 1	16 2 3（二叉树存在）	输出“找到了！它的关键字是 1”	
异常输入 1	16 1 2（二叉树不存在）	输出“没找到哦！”	
异常输入 2	16（二叉树不存在）	输出“树不存在！”	

表 2-16 最近公共祖先程序测试

2.4.17 翻转二叉树（InvertTree）

该函数的正常情况有一种，即二叉树已初始化，此时函数将二叉树所有结点的左右孩子交换位置，程序输出“翻转成功”。

异常情况有一种，即二叉树未初始化，此时函数不调用，程序输出“树不存在”。

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1 a), (2 b), (0 null), (0 null), (3 c), (4 d), (0 null), (0 null), (5 e), (0 null), (0 null), (-1 null)。具体的测试样例、理论输出结果以及程序的实际输出结果见表 2-17 和 2-18。

	输入	理论结果	运行情况（截图）
--	----	------	----------

正确 输入 1	17（二叉 树存在）	输出“翻 转成功”	
异常 输入 1	17（二叉 树不存在）	输出“树 不存在!”	

表 2-17 翻转二叉树程序测试

翻转前	翻转后

表 2-18 翻转二叉树前后结点位置展示

2.4.18 多个二叉树管理

多线性表管理包括新增线性表，删除线性表，定位线性表和切换线性表，使用函数和分支实现。

(1) 新增二叉树（AddTree）

新增二叉树的正常情况为，由用户输入新增二叉树的关键字（名字）后，程序将从用户获取到待操作的二叉树的名字 `ch` 传给函数，函数分配一段长度为 `BitTree` 长度的空间，用于存储这个二叉树，并保存其首指针，把名字 `ch` 赋值给这个二叉树的 `name`，多二叉树的长度增加 1，初始化二叉树的过程就完成了，函数返回 `OK`，程序输出“插入成功”。

异常情况有两种，第一种为多二叉树的长度超过了上限，因此不能再插入二叉树，函数返回 `ERROR`，程序输出“插入失败!”。

第二种为用户输入的关键字与已存在的线性表关键字一致，因此不能插入，函数返回 `ERROR`，程序输出“插入失败!”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-17。本测试中，测试 1 时多线性表为空，测试 2 时多线性表已满，测试 3 时多线性表的关键字为 `aaa aa a`。

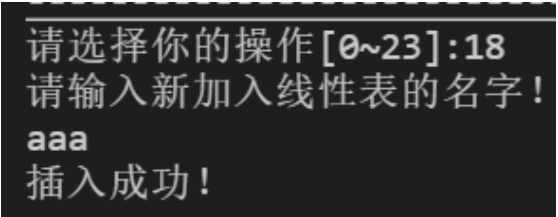
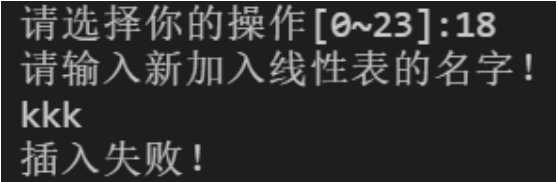
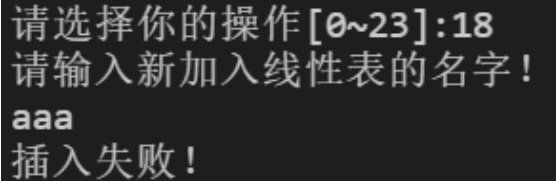
	输入	理论结果	运行情况（截图）
正确 输入 1	18 aaa（多 线性表未 满）	输出“插 入成功！”	
异常 输入 1	18 kkk（多 线性表已 满）	输出“插 入失败！”	
异常 输入 2	18 aaa（关 键字一致）	输出“插 入失败！”	

表 2-19 新增线性表程序测试

(2) 删除二叉树（RemoveTree）

删除二叉树的正常情况为，用户输入新建二叉树的关键字（名字）后，程序将名字传给函数，函数在多二叉树中查找 name 与 ch 相等的二叉树，将其删除，并将后面的二叉树都往前移动一位，多二叉树的长度减 1，程序输出“删除成功！”。

异常情况为函数未能找到对应名称的二叉树，函数返回 ERROR，表示没能找到这个二叉树，程序输出“删除失败！”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 2-20。本测试中，多线性表的关键字均为 aaa aa a。

	输入	理论结果	运行情况（截图）
--	----	------	----------

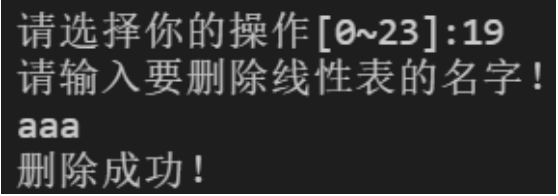
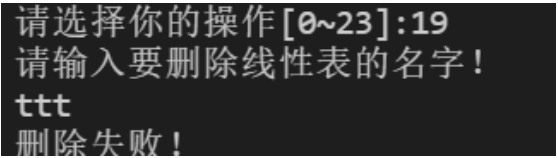
正确 输入 1	19 aaa（表 中有 aaa）	输出“删 除成功！”	
异常 输入 1	19 kkk（表 中无 kkk）	输出“删 除失败！”	

表 2-20 删除线性表程序测试

(3) 定位二叉树（LocateList）

定位二叉树的正常情况为，用户输入二叉树的关键字（名字）后，程序将名字传给函数，函数在多二叉树中查找 name 与 ch 相等的二叉树，找到后函数返回其位置，程序输出“找到了，在第 n 个！”。

异常情况为函数未能找到对应名称的二叉树，函数返回 ERROR，表示没能找到这个二叉树，程序输出“没有找到！”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 2-21。本测试中，多二叉树的关键字均为 aaa aa a。

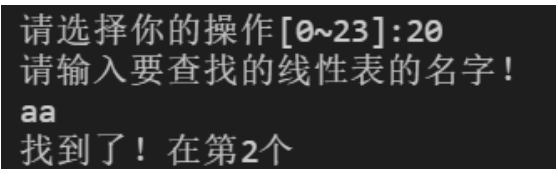
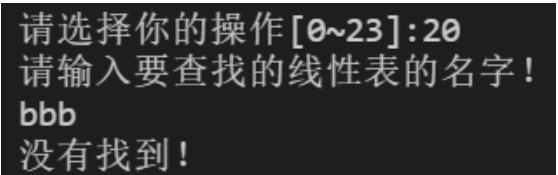
	输入	理论结果	运行情况（截图）
正确 输入 1	20 aa（aa 在第二个）	输出“找 到了，在 第 2 个！”	
异常 输入 1	20 bbb（表 中无 bbb）	输出“没 有找到！”	

表 2-21 定位二叉树程序测试

(4) 切换二叉树

切换二叉树的正常情况有两种，第一种为用户输入想要操作的二叉树名称，，系统查找到这个二叉树并返回这个二叉树的位置，程序输出“Success!”。

第二种为用户需要切换回一开始独立存在的单二叉树，则输入 0，程序就会切换回单二叉树进行操作，程序输出“Success!”。

异常情况为未能找到对应名称的二叉树，则程序输出“Failed!”，表示未能切换成功。

具体的测试样例，理论输出结果以及程序实际输出结果见表 2-22。本测试中，多二叉树的关键字均为 aaa aa a。

	输入	理论结果	运行情况（截图）
正确 输入 1	21 aaa（多 线性表中 有 aaa）	输出 “Success!”	
正常 输入 2	21 0（切换 回初始线 性表）	输出 “Success!”	
异常 输入 1	21 bbb（多 线性表中 无 bbb）	输出 “Failed!”	

表 2-22 切换二叉树程序测试

2.4.19 二叉树的文件读写

(1) 加载线性表（LoadTree）加载线性表的正常情况为由用户输入存储的文件名，待操作线性表已初始化且文件正常打开，之后程序将文件中的数据读入到线性表中，程序输出“读取成功!”。

异常情况为文件打开失败或者文件不存在，此时不执行读入到线性表的过

程，程序输出“文件不存在!”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 2-23 和表 2-24。本测试中，存在文件 111.txt 的含空的前序遍历结果为 (1 a), (2 b), (0 null), (0 null), (3 c), (4 d), (0 null), (0 null), (5 e), (0 null), (0 null), (-1 null)。具体的测试样例、理论输出结果以及程序的输出结果见表 2-23 和 2-24。。不存在文件 222.txt。

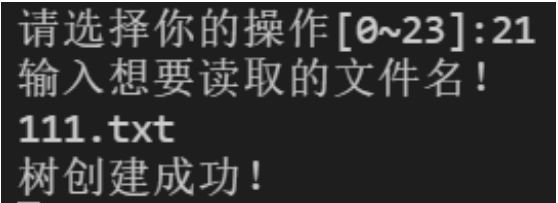
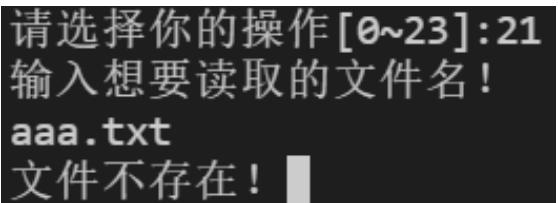
	输入	理论结果	运行情况（截图）
正确 输入 1	21 111.txt (111.txt 存在)	输出“读取成功!”	
异常 输入 1	21 aaa.txt (aaa.txt 不存在)	输出“文件不存在!”	

表 2-23 加载线性表程序测试

运行前	运行后
	

表 2-24 线性表运行前后对比

(2) 储存线性表（SaveTree）存储线性表的正常情况为由用户输入存储的文件名，待操作线性表已初始化且文件正常打开，如果文件不存在程序会创建一个文件，之后程序将数据存入文件中，程序输出“写入成功!”。

异常情况为待操作线性表未初始化，此时不执行输出到文件的过程，程序输出“线性表不存在!”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 2-25 和表 2-26。本测试中不存在文件 222.txt，线性表中的元素为 1 2 3 4。

	输入	理论结果	运行情况（截图）
--	----	------	----------

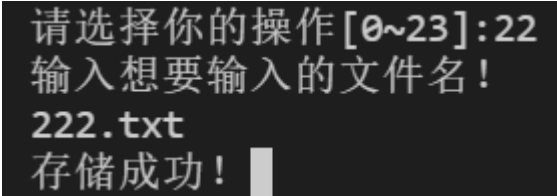
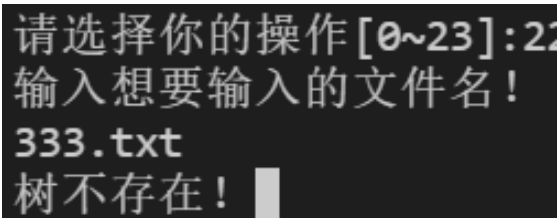
正确 输入 1	20 222.txt (222.txt 不 存在)	输出“存 储成功!”	
异常 输入 1	20 333.txt (线性表不 存在)	输出“树 不存在!”	

表 2-25 储存线性表程序测试

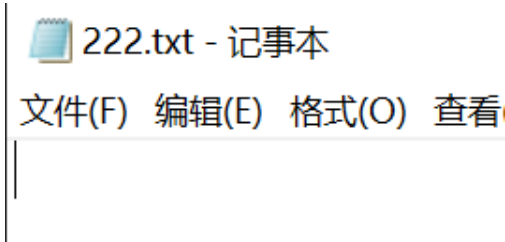
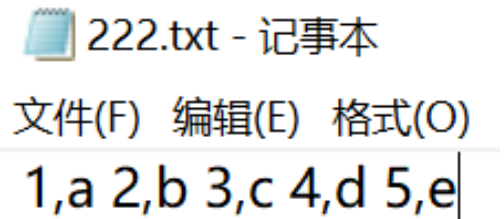
运行前	运行后
	

表 2-26 222.txt 运行前后对比

2.4.20 退出程序

退出程序的功能与线性表的状态无关，只要在根菜单输出 0 便可退出程序，程序输出“欢迎下次再使用本系统!”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 2-27。

	输入	理论结果	运行情况（截图）
--	----	------	----------

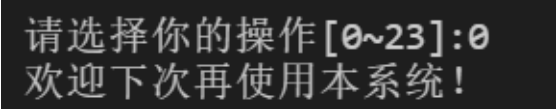
正确 输入	0（退出程序）	输出“欢迎下次再使用本系统！”	
----------	---------	-----------------	--

表 2-27 储存线性表程序测试

2.5 实验小结

本次实验相较前两次线性表和顺序表的实验，难度提升了一大截。这次的 ADT 采用了全新架构，函数也是全部重新写的，没有照搬头歌上的代码，因此花费的时间相较前两次也多得多。这次实验最难的部分我认为在于插入结点和删除结点这两个部分，不仅需要很长的代码，而且需要构造辅助函数来帮助操作，花费了很多时间和心思。

这次实验中我也对 c 语言中结构和动态分配方面产生了更加深刻的理解，使用了很多动态分配和释放的操作来节省内存，学到了许多关于运用结构的知识，例如 . 和 -> 的相关知识，这些让我对指针有了更加深刻的理解，不仅在于学会了节省内存，更在于学会了在编写函数之前就应该提前设想这些。

本次实验中使用了大量的递归算法，通过本次实验，我也对递归的使用、递归中可能存在的问题以及递归和循环结构的优劣有了更加直观的认识。

3 课程的收获和建议

数据结构是计算机课程中非常重要的一课，仅仅靠上课学习是不够的，因此一定要勤奋练习，通过数据结构实验来学习操作和修改程序。通过这门课程，我从中学到了许多以前不知道的知识，例如各种数据结构，例如链表，二叉树，图，也学到了许多非常重要的算法，这会为我以后的计算机学习打好基础。

在这里我要感谢我的数据结构老师郑渤龙老师，他上课风趣幽默，非常有意思，也是他这种上课风格让我对数据结构和大数据方面产生了浓郁的兴趣，让我受益匪浅。

课程建议是降低实验报告的难度，这份实验报告花了我 22 小时左右，总字符数大约在 13w 字，除去程序源码大约 9w5 的字符，如果可以减少一些报告量就最好了。

3.1 基于顺序存储结构的线性表实现

在基于顺序存储结构的线性表实现的实验中，我遇到了很多的困难和问题，不仅在于程序的编写，还有实现报告的写作，这些都让我有了很多的收获。我从中学到了许多以前不知道的知识。这也是我第一次编写 ADT，因此从中也收获了许多经验。

在编写函数时，需要传递线性表、元素等信息，有些函数还需要用到指针。最开始写的时候有一些参数传递错误，不知道应该传值还是传址，指针方面的知识还不是很牢固，导致函数无法达到预期功能。在调试时也遇到了许多的困难，例如函数无返回值，指针悬挂等，导致一直没有结果输出。还学到了许多排版方面相关的知识，例如写 switch 分支，分支的处理，如何退出分支等，这些都为以后更大的项目打下了基础。

3.2 基于二叉链表的二叉树实现

本次基于二叉链表的二叉树实现的实验相较前两次线性表和顺序表的实验，难度提升了一大截。这次的 ADT 采用了全新架构，函数也是全部重新写的，没有照搬头歌上的代码，因此花费的时间相较前两次也多得多。这次实验最难的部分我认为在于插入结点和删除结点这两个部分，不仅需要很长的代码，而且需要

构造辅助函数来帮助操作，花费了很多时间和心思。

这次实验中我也对 c 语言中结构和动态分配方面产生了更加深刻的理解，使用了很多动态分配和释放的操作来节省内存，学到了许多关于运用结构的知识，例如 . 和 -> 的相关知识，这些让我对指针有了更加深刻的理解，不仅在于学会了节省内存，更在于学会了在编写函数之前就应该提前设想这些。

本次实验中使用了大量的递归算法，通过本次实验，我也对递归的使用、递归中可能存在的问题以及递归和循环结构的优劣有了更加直观的认识。

附录 A 基于顺序存储结构线性表实现的源程序

```
/*-----头文件引用-----*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>

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

typedef int status;
typedef int ElemType;

#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
typedef struct
{
    ElemType *elem;
    int length;
    int listsize;
} SqList;

typedef struct
{ // 线性表的集合类型定义
    struct
    {
        char name[30];
        SqList L;
    } elem[10];
    int length;
} LISTS;
int state = 0;

/*-----函数-----*/
ElemType max(ElemType a, ElemType b)
```

```
{
    if (a > b)
        return a;
    else
        return b;
}

/*-----函数声明-----*/
status InitList(SqList *L);
status DestroyList(SqList *L);
status ClearList(SqList *L);
status ListEmpty(SqList L);
status ListLength(SqList L);
status GetElem(SqList L, int i, ElemType *e);
status LocateElem(SqList L, ElemType e);
status PriorElem(SqList L, ElemType cur, ElemType *pre);
status NextElem(SqList L, ElemType cur, ElemType *next);
status ListInsert(SqList *L, int i, ElemType e);
status ListDelete(SqList *L, int i, ElemType *e);
status ListTraverse(SqList L);
status MaxSubArray(SqList L);
status SubArrayNum(SqList L, ElemType k);
status sortList(SqList L);
status AddList(LISTS *Lists, char ListName[]);
status RemoveList(LISTS *Lists, char ListName[]);
status LocateList(LISTS Lists, char ListName[]);

/*-----*/
void main(void)
{
    int op = 1;
    /*-----*/
    SqList LL, *L = &LL;
    L->elem = NULL;
    L->length = 0;
    L->listsize = 0;
    /*-----*/
    LISTS Lists;
    Lists.length = 0;
    /*-----*/
    while (op)
    {
```

```
printf("\n\n");
printf("      Menu for Linear Table On Sequence Structure \n");
printf("-----\n");
printf("1. InitList      7. LocateElem      13.MaxSubArray   19.
      ReadFile\n");
printf("2. DestroyList  8. PriorElem        14.SubArrayNum   20.
      SaveFile\n");
printf("3. ClearList    9. NextElem          15.sortList      21.
      ChangeList\n");
printf("4. ListEmpty    10. ListInsert       16.AddList\n");
printf("5. ListLength   11. ListDelete       17.RemoveList\n");
printf("6. GetElem      12. ListTraverse    18. LocateList\n");
printf("0. Exit\n");
printf("-----\n");
printf("请选择你的操作[0~21]:");
scanf("%d", &op);

switch (op)
{
case 1:
    if (InitList(L) == OK)
        printf("线性表创建成功! \n");
    else
        printf("线性表创建失败! \n");
    getchar();
    getchar();
    break;
case 2:
    if (DestroyList(L) == OK)
        printf("线性表销毁成功! \n");
    else
        printf("线性表不存在! \n");
    getchar();
    getchar();
    break;
case 3:
    if (ClearList(L) == OK)
        printf("线性表清理成功! \n");
    else
        printf("线性表不存在! \n");
    getchar();
```

```
    getchar();
    break;
case 4:
    if (ListEmpty(*L) == TRUE)
        printf("线性表为空! \n");
    else if (ListEmpty(*L) == FALSE)
        printf("线性表不为空! \n");
    else
        printf("线性表不存在! \n");
    getchar();
    getchar();
    break;
case 5:
    if (ListLength(*L) != INFEASIBLE)
        printf("线性表长度为%d! \n", ListLength(*L));
    else
        printf("线性表不存在! \n");
    getchar();
    getchar();
    break;
case 6:
{
    ElemType e = 0;
    int i;
    printf("输入想要获取的位数: \n");
    scanf("%d", &i);
    if (GetElem(*L, i, &e) == INFEASIBLE)
        printf("线性表不存在! \n");
    else if (GetElem(*L, i, &e) == ERROR)
        printf("输入错误! \n");
    else
        printf("您要获取的数据为%d\n", e);
    getchar();
    getchar();
    break;
}
case 7:
{
    ElemType e;
    printf("您要查找的数字是: \n");
    scanf("%d", &e);
```

```
    if (LocateElem(*L, e) == INFEASIBLE)
        printf("线性表不存在! \n");
    else if (LocateElem(*L, e) == ERROR)
        printf("没有找到! \n");
    else
        printf("找到了! 在第%d位\n", LocateElem(*L, e));
    getchar();
    getchar();
    break;
}
case 8:
{
    ElemType pre, e;
    printf("您要找哪个数的前驱数? ");
    scanf("%d", &e);
    if (PriorElem(*L, e, &pre) == INFEASIBLE)
        printf("线性表不存在! \n");
    else if (PriorElem(*L, e, &pre) == ERROR)
        printf("未找到您要查找的数的前驱! \n");
    else
        printf("前驱数为%d!\n", pre);
    getchar();
    getchar();
    break;
}
case 9:
{
    ElemType next, e;
    printf("您要找哪个数的后驱数? ");
    scanf("%d", &e);
    if (NextElem(*L, e, &next) == INFEASIBLE)
        printf("线性表不存在! \n");
    else if (NextElem(*L, e, &next) == ERROR)
        printf("未找到您要查找的数的后驱! \n");
    else
        printf("后驱数为%d!\n", next);
    getchar();
    getchar();
    break;
}
case 10:
```

```
{
    ElemType e, i, k;
    printf("请输入您要插入的数以及位置!", &e, &i);
    scanf("%d %d", &e, &i);
    if ((k = ListInsert(L, i, e)) == INFEASIBLE)
        printf("线性表不存在! \n");
    else if (k == ERROR)
        printf("插入位置不正确! \n");
    else
        printf("插入成功! \n");
    getchar();
    getchar();
    break;
}

case 11:
{
    ElemType e, i, k;
    printf("请输入您要删除的位置!", &i);
    scanf("%d", &i);
    if ((k = ListDelete(L, i, &e)) == INFEASIBLE)
        printf("线性表不存在! \n");
    else if (k == ERROR)
        printf("删除位置不正确! \n");
    else
        printf("删除成功! \n");
    getchar();
    getchar();
    break;
}

case 12:
{
    if (!ListTraverse(*L))
        printf("线性表是空表! \n");
    getchar();
    getchar();
    break;
}

case 13:
{
    if (MaxSubArray(*L) == INFEASIBLE)
        printf("线性表不存在! \n");
}
```

```
else if (MaxSubArray(*L) == ERROR)
    printf("线性表是空表! \n");
else
    printf("最大连续子数组和为%d", MaxSubArray(*L));
getchar();
getchar();
break;
}
case 14:
{
    ElemType k;
    printf("请输入K: \n");
    scanf("%d", &k);
    if (SubArrayNum(*L, k) == INFEASIBLE)
        printf("线性表不存在! \n");
    else
        printf("找到了%d个子数组的和等于%d!", SubArrayNum(*L, k), k);
    getchar();
    getchar();
    break;
}
case 15:
{
    if (sortList(*L) == INFEASIBLE)
        printf("线性表不存在! \n");
    else
        printf("排序完成!");
    getchar();
    getchar();
    break;
}
case 16:
{
    char ListName[30];
    printf("请输入新加入线性表的名字! \n");
    scanf("%s", ListName);
    if (AddList(&Lists, ListName) == ERROR)
        printf("插入失败! \n");
    else
        printf("插入成功! \n");
}
```



```
        getchar();
        getchar();
        break;
    }
    case 17:
    {
        char ListName[30];
        printf("请输入要删除线性表的名字! \n");
        scanf("%s", ListName);
        if (RemoveList(&Lists, ListName) == ERROR)
            printf("删除失败! \n");
        else
            printf("删除成功! \n");
        getchar();
        getchar();
        break;
    }
    case 18:
    {
        char ListName[30];
        int k;
        printf("请输入要查找的线性表的名字! \n");
        scanf("%s", ListName);
        if ((k = LocateList(Lists, ListName)) == ERROR)
            printf("没有找到! \n");
        else
            printf("找到了! 在第%d个\n", k);
        getchar();
        getchar();
        break;
    }
    case 19:
    {
        FILE *fp;
        printf("请输入文件名! \n");
        char FileName[30];
        scanf("%s", FileName);
        if ((fp = fopen(FileName, "r+")) == NULL)
        {
            printf("文件不存在! ");
        }
    }
```

```
else
{
    int e, i = 0;
    if (InitList(L) == ERROR)
        printf("线性表已存在! \n");
    else
    {
        while (fscanf(fp, "%d ", &e) != EOF)
        {
            int result = ListInsert(L, ++i, e);
            if (result == OK)
                continue;
            else if (result == ERROR)
            {
                printf("读入失败! \n");
                break;
            }
            else if (result == INFEASIBLE)
            {
                printf("线性表不存在! \n");
                break;
            }
        }
        printf("读取成功! \n");
    }
}

getchar();
getchar();
break;
}

case 20:
{
    FILE *fp;
    printf("请输入文件名! \n");
    char FileName[30];
    scanf("%s", FileName);
    fp = fopen(FileName, "w+");
    if (L->elem == NULL)
        printf("线性表不存在! \n");
    else
    {
```

```
        for (int i = 0; i < L->length - 1; i++)
        {
            fprintf(fp, "%d ", L->elem[i]);
        }
        if (L->length != 0)
            fprintf(fp, "%d", L->elem[L->length - 1]);
        printf("写入完成! \n");
    }
    getchar();
    getchar();
    break;
}
case 21:
{
    char ch[20];
    printf("请输入想切换的线性表名称, \n如果要切换回一开始的请输入0
        : ");
    scanf("%s", ch);
    if (strcmp(ch, "0") == 0)
    {
        L = &LL;
        printf("操作成功! ");
        state = 0;
    }
    else
    {
        int k;
        if ((k = LocateList(Lists, ch)) == ERROR)
            printf("没有找到! \n");
        else
        {
            if (state == 0)
            {
                LL = *L;
            }
            L = &Lists.elem[k - 1].L;
            printf("操作成功! ");
            state = 1;
        }
    }
    getchar();
}
```

```
        getchar();
        break;
    }
    case 0:
        break;
    } // end of switch
} // end of while
printf("欢迎下次再使用本系统! \n");
} // end of main()

status InitList(SqList *L)
{
    if ((*L).elem != NULL)
        return ERROR;
    (*L).elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if ((*L).elem == NULL)
        exit(OVERFLOW);
    (*L).length = 0;
    (*L).listsize = LIST_INIT_SIZE;
    return OK;
}

status ListTraverse(SqList L)
{
    int i;
    printf("\n-----all elements -----\\n");
    for (i = 0; i < L.length; i++)
        printf("%d ", L.elem[i]);
    printf("\n----- end -----\\n");
    return L.length;
}

status DestroyList(SqList *L)
// 如果线性表L存在, 销毁线性表L, 释放数据元素的空间, 返回OK, 否则返回
// INFEASIBLE。
{
    if ((*L).elem == NULL)
        return INFEASIBLE;
    free((*L).elem);
    (*L).elem = NULL;
    (*L).listsize = 0;
}
```

```
(*L).length = 0;
return OK;
}

status ClearList(SqList *L)
// 如果线性表L存在，删除线性表L中的所有元素，返回OK，否则返回INFEASIBLE。
{
    if ((*L).elem == NULL)
        return INFEASIBLE;
    for (int i = 0; i < (*L).length; i++)
    {
        (*L).elem[i] = 0;
    }
    (*L).length = 0;
    return OK;
}

status ListEmpty(SqList L)
// 如果线性表L存在，判断线性表L是否为空，空就返回TRUE，否则返回FALSE；如果线性表L不存在，返回INFEASIBLE。
{
    if (L.elem == NULL)
        return INFEASIBLE;
    if (L.length == 0)
        return TRUE;
    else
        return FALSE;
}

status ListLength(SqList L)
// 如果线性表L存在，返回线性表L的长度，否则返回INFEASIBLE。
{
    if (L.elem == NULL)
        return INFEASIBLE;
    else
        return L.length;
}

status GetElem(SqList L, int i, ElemType *e)
// 如果线性表L存在，获取线性表L的第i个元素，保存在e中，返回OK；如果i不合法，返回ERROR；如果线性表L不存在，返回INFEASIBLE。
```

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

int LocateElem(SqList L, ElemType e)

// 如果线性表L存在，查找元素e在线性表L中的位置序号并返回该序号；如果e不存在，返回0；当线性表L不存在时，返回INFEASIBLE（即-1）。

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

status PriorElem(SqList L, ElemType e, ElemType *pre)

// 如果线性表L存在，获取线性表L中元素e的前驱，保存在pre中，返回OK；如果没有前驱，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

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

```
{
    *pre = L.elem[i - 1];
    return OK;
}
else
    return ERROR;
}
```

status NextElem(SqList L, ElemType e, ElemType *next)

// 如果线性表L存在，获取线性表L元素e的后继，保存在next中，返回OK；如果没有后继，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

```
{
    if (L.elem == NULL)
        return INFEASIBLE;
    int i;
    for (i = 0; i < L.length; i++)
    {
        if (L.elem[i] == e)
            break;
    }
    if (i >= L.length - 1)
        return ERROR;
    else
    {
        *next = L.elem[i + 1];
        return OK;
    }
}
```

status ListInsert(SqList *L, int i, ElemType e)

// 如果线性表L存在，将元素e插入到线性表L的第i个元素之前，返回OK；当插入位置不正确时，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

```
{
    if ((*L).elem == NULL)
        return INFEASIBLE;
    if (i - 1 > (*L).length || i <= 0)
        return ERROR;
    if ((*L).length >= (*L).listsize)
    {
        (*L).elem = (ElemType *)realloc((*L).elem, ((*L).listsize +
            LISTINCREMENT) * sizeof(ElemType));
    }
}
```



```
        (*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;
}
```

status ListDelete(SqList *L, int i, ElemType *e)

// 如果线性表L存在，删除线性表L的第i个元素，并保存在e中，返回OK；当删除位置不正确时，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

```
{
    if ((*L).elem == NULL)
        return INFEASIBLE;
    if (i > (*L).length || i <= 0)
        return ERROR;
    *e = (*L).elem[i - 1];
    for (int j = i - 1; j < (*L).length; j++)
    {
        (*L).elem[j] = (*L).elem[j + 1];
    }
    (*L).length--;
    return OK;
}
```

status ListTraverse(SqList L)

// 如果线性表L存在，依次显示线性表中的元素，每个元素间空一格，返回OK；如果线性表L不存在，返回INFEASIBLE。

```
{
    if (L.elem == NULL)
        return INFEASIBLE;
    for (int i = 0; i < L.length - 1; i++)
    {
        printf("%d ", L.elem[i]);
    }
    if (L.length != 0)
        printf("%d", L.elem[L.length - 1]);
    return OK;
}
```

```
}

status MaxSubArray(SqList L)
{
    if (L.elem == NULL)
        return INFEASIBLE;
    if (L.length == 0)
        return ERROR;
    ElemType a[101];
    for (int i = 0; i < L.length; i++)
    {
        a[i] = 0;
    }
    a[0] = L.elem[0];
    for (int i = 1; i < L.length; i++)
    {
        a[i] = max(a[i - 1] + L.elem[i], L.elem[i]);
    }
    ElemType ans = a[0];
    for (int i = 0; i < L.length; i++)
    {
        if (a[i] > ans)
            ans = a[i];
    }
    return ans;
}

status SubArrayNum(SqList L, ElemType k)
{
    if (L.elem == NULL)
        return INFEASIBLE;
    int cnt = 0, sum = 0;
    for (int i = 0; i < L.length; i++)
    {
        for (int j = i; j < L.length; j++)
        {
            for (int k = i; k <= j; k++)
            {
                sum += L.elem[k];
            }
            if (sum == k)
                cnt++;
        }
    }
    return cnt;
}
```

```
        cnt++;
        sum = 0;
    }
}
return cnt;
}

status sortList(SqList L)
{
    if (L.elem == NULL)
        return INFEASIBLE;
    for (int i = L.length - 1; i >= 1; i--)
    {
        for (int j = 0; j < i; j++)
        {
            if (L.elem[j] > L.elem[j + 1])
            {
                ElemType t;
                t = L.elem[j];
                L.elem[j] = L.elem[j + 1];
                L.elem[j + 1] = t;
            }
        }
    }
    return OK;
}

status AddList(LISTS *Lists, char ListName[])
// 只需要在 Lists 中增加一个名称为 ListName 的空线性表，线性表数据又后台测试程序
// 插入。
{
    if ((*Lists).length >= 10)
        return ERROR;
    for (int i = 0; i < (*Lists).length; i++)
    {
        if (strcmp((*Lists).elem[i].name, ListName) == 0)
            return ERROR;
    }
    ElemType *newbase;
    newbase = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (!newbase)
```

```
        return ERROR;
    (*Lists).elem[(*Lists).length].L.elem = newbase;
    (*Lists).elem[(*Lists).length].L.length = 0;
    (*Lists).elem[(*Lists).length].L.listsize = LIST_INIT_SIZE;
    for (int i = 0; i < 30; i++)
    {
        (*Lists).elem[(*Lists).length].name[i] = ListName[i];
    }
    (*Lists).length++;
    return OK;
}

status RemoveList(LISTS *Lists, char ListName[])
// Lists中删除一个名称为ListName的线性表
{
    int i, j;
    if ((*Lists).length == 0)
        return ERROR;
    for (i = 0; i < (*Lists).length; i++)
    {
        if (strcmp((*Lists).elem[i].name, ListName) == 0)
        {
            for (j = i; j < (*Lists).length - 1; j++)
            {
                (*Lists).elem[j] = (*Lists).elem[j + 1];
            }
            (*Lists).length--;
            return OK;
        }
    }
    return ERROR;
}

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

```
}  
return 0;  
}
```

附录 B 基于二叉链表二叉树实现的源程序

```
/*-----头文件引用-----*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>

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

typedef int status;
typedef int ElemType;

#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
typedef int KeyType;
typedef struct
{ // 二叉数结点数据类型定义
    KeyType key;
    char others[20];
} TElemType;
int state1 = 0, state2 = 0;
typedef struct BiTNode
{ // 二叉链表结点的定义
    TElemType data;
    struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;

typedef struct
{ // 线性表的集合类型定义
    struct
    {
        char name[30];
        BiTree T;
    }
}
```

```
    } elem[10];
    int length;
} TREES;
TElemType definition[LIST_INIT_SIZE];
int num, state = 0;
/*-----函数-----*/
ElemType max(ElemType a, ElemType b)
{
    if (a > b)
        return a;
    else
        return b;
}
void visit(BiTree T)
{
    if (T == NULL)
        return;
    printf(" %d,%s", T->data.key, T->data.others);
}
void fvisit(BiTree T, FILE *fp)
{
    if (T == NULL)
        return;
    fprintf(fp, " %d,%s", T->data.key, T->data.others);
}
/*-----函数声明-----*/
status CreateBiTree(BiTree *T, TElemType definition[], int n);
status DestroyBiTree(BiTree *T);
status ClearBiTree(BiTree *T);
status BiTreeEmpty(BiTree T);
int BiTreeDepth(BiTree T);
BiTNode *LocateNode(BiTree T, KeyType e);
status Assign(BiTree *T, KeyType e, TElemType value);
BiTNode *GetSibling(BiTree T, KeyType e);
status InsertNode(BiTree *T, KeyType e, int LR, TElemType c);
status DeleteNode(BiTree *T, KeyType e);
status PreOrderTraverse(BiTree T, void (*visit)(BiTree));
status InOrderTraverse(BiTree T, void (*visit)(BiTree));
status PostOrderTraverse(BiTree T, void (*visit)(BiTree));
status LevelOrderTraverse(BiTree T, void (*visit)(BiTree));
int MaxPathSum(BiTree T);
```


华中科技大学课程实验报告

```
BiTNode *LowestCommonAncestor(BiTree T, KeyType e1, KeyType e2);
status InvertTree(BiTree *T);
status AddTree(TREES *Trees, char TreeName[]);
status RemoveTree(TREES *Trees, char TreeName[]);
status LocateTree(TREES Trees, char TreeName[]);
status fPreOrderTraverse(BiTree T, FILE *fp, void (*fvisit)(BiTree, FILE *))
;
/*-----*/

void main(void)
{
    int op = 1;
    /*-----*/
    BiTree TT = NULL;
    BiTree *T = &TT;
    /*-----*/
    TREES Trees;
    Trees.length = 0;
    /*-----*/
    while (op)
    {
        printf("\n\n");
        printf("Menu for Linear Table On\n\n");
        printf("Sequence Structure\n");
        printf("-----\n");
        printf("1. CreateBiTree 7. Assign 13. PostOrderTraverse\n\n");
        printf("19. RemoveTree\n");
        printf("2. DestroyBiTree 8. GetSibling 14. LevelOrderTraverse\n\n");
        printf("20. LocateTree\n");
        printf("3. ClearBiTree 9. InsertNode 15. MaxPathSum\n\n");
        printf("21. ReadFile\n");
        printf("4. BiTreeEmpty 10. DeleteNode 16.\n\n");
        printf("LowestCommonAncestor 22. SaveFile\n");
        printf("5. BiTreeDepth 11. PreOrderTraverse 17. InvertTree\n\n");
        printf("23. SwitchTree\n");
        printf("6. LocateNode 12. InOrderTraverse 18. AddTree\n");
        printf("0. Exit\n");
        printf("-----\n");
        printf("请选择你的操作[0~23]:");
        scanf("%d", &op);
        switch (op)
```

```
{
case 1:
{
    printf("请输入数据: \n");
    for (num = 0; definition[num - 1].key != -1; num++)
    {
        int a;
        char c[20];
        scanf("%d %s", &a, c);
        definition[num].key = a;
        strcpy(definition[num].others, c);
    }
    int k = CreateBiTree(T, definition, num);
    if (k == OK)
        printf("树创建成功! \n");
    else if (k == INFEASIBLE)
        printf("树已存在! \n");
    else
        printf("数据错误\n");
    getchar();
    getchar();
    break;
}
case 2:
{
    if (DestroyBiTree(T) == OK)
        printf("树已删除! \n");
    else
        printf("树不存在! \n");
    getchar();
    getchar();
    break;
}
case 3:
{
    if (ClearBiTree(T) == OK)
        printf("清空树成功! \n");
    else
        printf("树不存在! \n");
    getchar();
    getchar();
}
```

```
        break;
    }
    case 4:
    {
        if (BiTreeEmpty(*T) == OK)
            printf("树为空! \n");
        else if (BiTreeEmpty(*T) == INFEASIBLE)
            printf("树不存在! \n");
        else
            printf("树不为空! ");
        getchar();
        getchar();
        break;
    }
    case 5:
    {
        int k;
        if (k = BiTreeDepth(*T))
            printf("树的深度为%d!\n", k);
        else
            printf("树不存在! \n");
        getchar();
        getchar();
        break;
    }
    case 6:
    {
        KeyType e;
        printf("您要查找的数是: \n");
        scanf("%d", &e);
        BiTree p = LocateNode(*T, e);
        if (p == NULL)
            printf("未找到! \n");
        else
        {
            printf("找到了! 是%d,%s! \n", p->data.key, p->data.others);
        }
        getchar();
        getchar();
        break;
    }
}
```

```
case 7:
{
    int e, k;
    TElemType p;
    state1 = 0, state2 = 0;
    printf("要赋值的节点是: \n");
    scanf("%d", &e);
    printf("赋值的点和名称是:\n");
    scanf("%d %s", &p.key, &p.others);
    if ((k = Assign(T, e, p)) == OK)
        printf("赋值成功! \n");
    else if (k == INFEASIBLE)
        printf("树不存在! \n");
    else
        printf("赋值失败! \n");
    getchar();
    getchar();
    break;
}

case 8:
{
    ElemType e;
    state1 = 0, state2 = 0;
    printf("请输入您要查找的节点的关键字: \n");
    scanf("%d", &e);
    if (T == NULL)
        printf("树不存在! ");
    else
    {
        if (LocateNode(*T, e) != NULL)
        {
            BiTree p = GetSibling(*T, e);
            if (p)
                printf("兄弟找到了! 是%d,%s\n", p->data.key, p->data.others);
            else
                printf("独生子女! \n");
        }
        else
            printf("输入错误! ");
    }
}
```

```
    getchar();
    getchar();
    break;
}
case 9:
{
    int e, LR, state = 0, state3 = 0;
    TElemType c;
    printf("输入想要插入节点的关键字和位置: \n");
    scanf("%d %d", &e, &LR);
    printf("输入插入节点的关键字和名字: \n");
    scanf("%d %s", &c.key, c.others);
    if (T != NULL)
    {
        for (int i = 0; i < num; i++)
        {
            if (definition[i].key == c.key)
                state = 1;
            if (definition[i].key == e)
                state3 = 1;
        }
        if (state || !state3)
            printf("插入失败~~~~~\n");
        if (!state && state3 && LR != -1)
        {
            InsertNode(T, e, LR, c);
            printf("插入完成了! \n");
            definition[num].key = c.key;
            strcpy(definition[num].others, c.others);
            num++;
        }
        else if (LR == -1)
        {
            definition[num].key = c.key;
            strcpy(definition[num].others, c.others);
            num++;
            InsertNode(T, e, LR, c);
            printf("插入完成了! \n");
        }
    }
    else
```

```
        printf("树不存在！");
        getchar();
        getchar();
        break;
    }
    case 10:
    {
        int e, state = 0;
        if (T != NULL)
        {
            printf("输入您要删除节点的关键字：\n");
            scanf("%d", &e);
            for (int i = 0; i < num; i++)
            {
                if (definition[i].key == e)
                    state = 1;
            }
            DeleteNode(T, e);
            if (state == 1)
                printf("删除成功！！\n");
            else
                printf("删除失败！\n");
        }
        else
            printf("树不存在！");
        getchar();
        getchar();
        break;
    }
    case 11:
    {
        if (PreOrderTraverse(*T, (*visit)) == OK)
            ;
        else
            printf("树不存在？！！\n");
        getchar();
        getchar();
        break;
    }
    case 12:
    {
```

```
    if (InOrderTraverse(*T, (*visit)) == OK)
        ;
    else
        printf("树不存在? ! ? ! \n");
        getchar();
        getchar();
        break;
}
case 13:
{
    if (PostOrderTraverse(*T, (*visit)) == OK)
        ;
    else
        printf("树不存在? ! ? ! \n");
        getchar();
        getchar();
        break;
}
case 14:
{
    if (LevelOrderTraverse(*T, (*visit)) == OK)
        ;
    else
        printf("树不存在? ! ? ! \n");
        getchar();
        getchar();
        break;
}
case 15:
{
    if (MaxPathSum(*T) == 0)
        printf("树不存在! \n");
    else
        printf("最大路径和为: %d\n", MaxPathSum(*T));
        getchar();
        getchar();
        break;
}
case 16:
{
    KeyType e1, e2;
```

```
    if (T != NULL)
    {
        printf("输入想要查找的两个节点的关键字：\n");
        scanf("%d %d", &e1, &e2);
        if (LowestCommonAncestor(*T, e1, e2))
            printf("找到了,它的关键字是%d!\n", LowestCommonAncestor
                (*T, e1, e2)->data.key);
        else
            printf("没找到哦!! \n");
    }
    else
        printf("树不存在! ");
    getchar();
    getchar();
    break;
}
case 17:
{
    if (*T != NULL)
    {
        if (InvertTree(T))
            printf("翻转成功! \n");
    }
    else
        printf("树不存在! \n");
    getchar();
    getchar();
    break;
}
case 18:
{
    char TreeName[30];
    printf("请输入新加入线性表的名字! \n");
    scanf("%s", TreeName);
    if (AddTree(&Trees, TreeName) == ERROR)
        printf("插入失败! \n");
    else
        printf("插入成功! \n");
    getchar();
    getchar();
    break;
}
```



```
}
case 19:
{
    char TreeName[30];
    printf("请输入要删除线性表的名字! \n");
    scanf("%s", TreeName);
    if (RemoveTree(&Trees, TreeName) == ERROR)
        printf("删除失败! \n");
    else
        printf("删除成功! \n");
    getchar();
    getchar();
    break;
}
case 20:
{
    char TreeName[30];
    int k;
    printf("请输入要查找的线性表的名字! \n");
    scanf("%s", TreeName);
    if ((k = LocateTree(Trees, TreeName)) == ERROR)
        printf("没有找到! \n");
    else
        printf("找到了! 在第%d个\n", k);
    getchar();
    getchar();
    break;
}
case 21:
{
    FILE *fp;
    printf("输入想要读取的文件名! \n");
    char FILENAME[30];
    scanf("%s", FILENAME);
    if ((fp = fopen(FILENAME, "r+")) != NULL)
    {
        DestroyBiTree(T);
        for (num = 0; definition[num - 1].key != -1; num++)
        {
            int a;
            char c[20];
```

```
fscanf(fp, "%d %s", &a, c);
definition[num].key = a;
strcpy(definition[num].others, c);
}
int k = CreateBiTree(T, definition, num);
if (k == OK)
    printf("树创建成功! \n");
else if (k == INFEASIBLE)
    printf("树已存在! \n");
else
    printf("数据错误\n");
}
else
    printf("文件不存在! ");
getchar();
getchar();
break;
}
case 22:
{
    FILE *fp;
    printf("输入想要输入的文件名! \n");
    char FILENAME[30];
    scanf("%s", FILENAME);
    if (*T != NULL)
    {
        if ((fp = fopen(FILENAME, "w+")) != NULL)
        {
            fPreOrderTraverse(*T, fp, fvisit);
            printf("存储成功! ");
        }
        else
            printf("文件不存在! ");
    }
    else
        printf("树不存在! ");
    getchar();
    getchar();
    break;
}
case 23:
```

```
{
    char Treename[10];
    printf("输入你想要切换的树：\n输入0来切换到最初的树：");
    scanf("%s", Treename);
    if (strcmp(Treename, "0") == 0)
    {
        *T = TT;
        printf("Success!");
        state = 0;
    }
    else
    {
        if (state == 0)
        {
            TT = *T;
        }
        state = 1;
        if (LocateTree(Trees, Treename) != ERROR)
        {
            T = &Trees.elem[LocateTree(Trees, Treename) - 1].T;
            printf("Success!");
        }
        else
            printf("Failed!Can't find Tree called %s\n", Treename);
    }
    getchar();
    getchar();
    break;
}

case 0:
    break;

default:
    printf("输入格式错误，请重新输入！\n");
    break;
} // end of switch
} // end of while
printf("欢迎下次再使用本系统！\n");
} // end of main()

int cnt = -1;

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

```
if ((*T) != NULL)
    return INFEASIBLE;
if (definition[++cnt].key)
{
    (*T) = (BiTree)malloc(sizeof(BiTreeNode));
    (*T)->data.key = definition[cnt].key;
    (*T)->lchild = NULL;
    (*T)->rchild = NULL;
    strcpy((*T)->data.others, definition[cnt].others);
    CreateBiTree(&(*T)->lchild, definition, n);
    CreateBiTree(&(*T)->rchild, definition, n);
}
else
    (*T) = NULL;
return OK;
}
```

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

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

```
(*T)->data.key = 0;
strcpy((*T)->data.others, " ");
}
return OK;
}

status BiTreeEmpty(BiTree T)
{
    if (T == NULL)
        return INFEASIBLE;
    else
    {
        if (T->lchild == NULL && T->rchild == NULL)
            return OK;
    }
    return ERROR;
}

int BiTreeDepth(BiTree T)
{
    if (T == NULL)
        return 0;
    int n = 1;
    n += max(BiTreeDepth(T->lchild), BiTreeDepth(T->rchild));
    return n;
}

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

```
status Assign(BiTree *T, KeyType e, TElemType value)
{
    if ((*T) == NULL)
        return INFEASIBLE;
    if ((*T))
    {
        if ((*T)->data.key == value.key)
            state1 = 1;
        if ((*T)->data.key == e && (*T)->data.key == value.key)
        {
            (*T)->data.key = value.key;
            strcpy((*T)->data.others, value.others);
            state2 = 1;
            state1 = 0;
        }
        if ((*T)->data.key == e)
        {
            (*T)->data.key = value.key;
            strcpy((*T)->data.others, value.others);
            state2 = 1;
        }
        Assign(&(*T)->lchild, e, value);
        Assign(&(*T)->rchild, e, value);
    }
    if (state1)
        return ERROR;
    else if (state2)
        return OK;
    else
        return ERROR;
}
```

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

```
        return T->lchild;
    if (GetSibling(T->lchild, e))
        return GetSibling(T->lchild, e);
    if (GetSibling(T->rchild, e))
        return GetSibling(T->rchild, e);
}
return NULL;
}

status InsertNode(BiTree *T, KeyType e, int LR, TElemType c)
{
    if (LR == -1)
    {
        BiTree p = NULL;
        p = (BiTree)malloc(sizeof(BiTreeNode));
        p->lchild = NULL;
        p->data.key = c.key;
        strcpy(p->data.others, c.others);
        p->rchild = (*T);
        (*T) = p;
        return OK;
    }
    else
    {
        if ((*T))
        {
            if ((*T)->data.key == e)
            {

                BiTreeNode *p = NULL;
                p = (BiTree)malloc(sizeof(BiTree));
                p->lchild = NULL;
                p->data.key = c.key;
                strcpy(p->data.others, c.others);
                if (LR == 0)
                {
                    p->rchild = (*T)->lchild;
                    (*T)->lchild = p;
                }
                else if (LR == 1)
                {
```

```
        p->rchild = (*T)->rchild;
        (*T)->rchild = p;
    }
}
else
{
    InsertNode(&(*T)->lchild, e, LR, c);
    InsertNode(&(*T)->rchild, e, LR, c);
}
}
}
return OK;
}
```

```
BiTree FindRightestDnode(BiTree T)
{
    if (T->rchild == NULL)
        return T;
    if (T->rchild)
        return FindRightestDnode(T->rchild);
}
```

```
status DeleteNode(BiTree *T, KeyType e)
{
    if ((*T))
    {
        if ((*T)->data.key == e)
        {
            BiTree *p = T;
            if ((*T)->lchild && !(*T)->rchild)
            {
                (*T) = (*T)->lchild;
                free(*p);
                return OK;
            }
            else if (!(*T)->lchild && (*T)->rchild)
            {
                (*T) = (*T)->rchild;
                free(*p);
                return OK;
            }
        }
    }
}
```



```
        }
        else if ((*T)->lchild && (*T)->rchild)
        {
            BiTree q = FindRightestDnode((*T)->lchild);
            q->rchild = (*T)->rchild;
            (*T) = (*T)->lchild;
            free(p);
            return OK;
        }
    }
    else
    {
        DeleteNode(&(*T)->lchild, e);
        DeleteNode(&(*T)->rchild, e);
    }
}
return 0;
}

status PreOrderTraverse(BiTree T, void (*visit)(BiTree))
{
    if (T)
    {
        visit(T);
        if (T->lchild)
            PreOrderTraverse(T->lchild, visit);
        if (T->rchild)
            PreOrderTraverse(T->rchild, visit);
        return OK;
    }
    return INFEASIBLE;
}

status fPreOrderTraverse(BiTree T, FILE *fp, void (*fvisit)(BiTree, FILE *))
// 前序遍历二叉树T
{
    if (T)
    {
        fvisit(T, fp);
        if (T->lchild)
            fPreOrderTraverse(T->lchild, fp, fvisit);
    }
}
```

```
        if (T->rchild)
            fPreOrderTraverse(T->rchild, fp, fvisit);
        return OK;
    }
    else
    {
        fprintf(fp, " 0 0 NULL ");
    }
    return INFEASIBLE;
}

status InOrderTraverse(BiTree T, void (*visit)(BiTree))
// 中序遍历二叉树T
{
    if (T)
    {
        if (T->lchild)
            InOrderTraverse(T->lchild, visit);
        visit(T);
        if (T->rchild)
            InOrderTraverse(T->rchild, visit);
        return 1;
    }
    return 0;
}

status PostOrderTraverse(BiTree T, void (*visit)(BiTree))
// 后序遍历二叉树T
{
    if (T)
    {
        if (T->lchild)
            PostOrderTraverse(T->lchild, visit);
        if (T->rchild)
            PostOrderTraverse(T->rchild, visit);
        visit(T);
        return 1;
    }
    return 0;
}
```

```
BiTree a[100];
int front = 0, tail = 0;
status LevelOrderTraverse(BiTree T, void (*visit)(BiTree))
// 按层遍历二叉树T
{
    if (T == NULL)
        return INFEASIBLE;
    a[front++] = T;
    while (front > tail)
    {
        if (a[tail])
        {
            visit(a[tail]);
            a[front++] = a[tail]->lchild;
            a[front++] = a[tail]->rchild;
        }
        tail++;
    }
    return OK;
}

status find(BiTree T, KeyType e)
{
    if (T)
    {
        if (T->data.key == e)
            return OK;
        else
        {
            if (T->lchild)
                if (find(T->lchild, e))
                    return OK;
            if (T->rchild)
                if (find(T->rchild, e))
                    return OK;
        }
    }
    return 0;
}

BiTNode *LowestCommonAncestor(BiTree T, KeyType e1, KeyType e2)
```

```
{
    if (T)
    {
        if (find(T->lchild , e1) && find(T->rchild , e2))
            return T;
        else if (find(T->lchild , e2) && find(T->rchild , e1))
            return T;
        else
        {
            if (T->lchild)
            {
                if (LowestCommonAncestor(T->lchild , e1 , e2))
                    return LowestCommonAncestor(T->lchild , e1 , e2);
            }
            if (T->rchild)
            {
                if (LowestCommonAncestor(T->rchild , e1 , e2))
                    return LowestCommonAncestor(T->rchild , e1 , e2);
            }
        }
    }
    return NULL;
}

status InvertTree(BiTree *T)
{
    if (*T == NULL)
        return INFEASIBLE;
    if ((*T))
    {
        BiTree p = (*T)->lchild;
        (*T)->lchild = (*T)->rchild;
        (*T)->rchild = p;
        InvertTree(&(*T)->lchild);
        InvertTree(&(*T)->rchild);
    }
}

int MaxPathSum(BiTree T)
{
    if (T)
```

```
{
    return max(MaxPathSum(T->lchild), MaxPathSum(T->rchild)) + T->data.
        key;
}
return 0;
}
```

```
status AddTree(TREES *Trees, char TreeName[])
{
    if ((*Trees).length >= 10)
        return ERROR;
    (*Trees).elem[(*Trees).length].T = (BiTree)malloc(sizeof(BiTree));
    (*Trees).elem[(*Trees).length].T = NULL;
    strcpy((*Trees).elem[(*Trees).length].name, TreeName);
    (*Trees).length++;
    return OK;
}
```

```
status RemoveTree(TREES *Trees, char TreeName[])
{
    int i, j;
    if ((*Trees).length == 0)
        return ERROR;
    for (i = 0; i < (*Trees).length; i++)
    {
        if (strcmp((*Trees).elem[i].name, TreeName) == 0)
        {
            for (j = i; j < (*Trees).length - 1; j++)
            {
                (*Trees).elem[j] = (*Trees).elem[j + 1];
            }
            (*Trees).length--;
            return OK;
        }
    }
    return ERROR;
}
```

```
status LocateTree(TREES Trees, char TreeName[])
{
    int state = 0, i, j;
```

```
for (i = 0; i < Trees.length; i++)
{
    if (strcmp(Trees.elem[i].name, TreeName) == 0)
        return i + 1;
}
return 0;
}
```