華中科技大學

课 程 实 验 报 告

课程名称： 数据结构实验

专业班级 CS2109

学　　号 U202115612

姓　　名 俞若鹏

指导教师 郑渤龙

报告日期 2022 年 5 月 28 日

计算机科学与技术学院

目　　录

1 基于链式存储结构的线性表实现

1.1 问题描述

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

1.2 系统设计

”链表作为线性结构”，是数据结构中最常用、最基本的结构类型，本实验采用链表的线性存储方式, 并实现了链表各类基本功能如：增添、删除、遍历等，并在此基础之上附加翻转、倒序删除、排序。多线性表操作等功能，更好地方便使用者对链表进行操作，且附带了语言文字提示，降低了程序的使用难度。

基本结构设计代码

```
typedef int ElemType;
typedef struct LNode{
        ElemType data;
        struct LNode *next;
} LNode , *LinkList;
typedef struct {
        struct {
        char name[30];
                LinkList L;
        } elem[10];
        int length ;
        int listsize ;
} LISTS;
```

typedef 了 int 基本数据类型为 ElemType，便于后续程序维护以及功能增加等，减少代码的修改量且增加代码的可读性，而且 typedef 结点的结构体变量和指针，减少代码的书写量，也增大了代码的可读性，是程序整体美观。

多线性表在结构体内部套用结构体并采取结构体数组的方式实现多链表的储存以及命名，同时设计 length 变量实时监控表的长度，listsize 变量设置最大容量，防止溢出。

在程序实际运行的可视化简易菜单上，采用 printf 的简易方式打印出菜单并设计 while 循环实现程序的多次操作，并根据输入数字对用 switch 函数中的不同具体功能选项。

```
system("cls");  printf("\n\n");
printf("      Menu for Linear Table On Sequence Structure \n");
printf("-------------------------------------------------\n" );
printf("          1. InitList       12.ListTraverse\n"       );
printf("          2. DestroyList    13.SaveList\n"           );
printf("          3. ClearList      14.LoadList\n"           );
printf("          4. ListEmpty      15.reverseList\n"        );
printf("          5. ListLength     16.RemoveNthFromEnd\n"   );
printf("          6. GetElem        17.sortList\n"           );
printf("          7. LocateElem     18.AddList\n"            );
printf("          8. PriorElem      19.RemoveList\n"         );
printf("          9. NextElem       20.LocateList\n"         );
printf("          10.ListInsert     21.ListPrint\n"          );
printf("          11.ListDelete     22.MutlListInsert\n"     );
printf("          0. Exit           23.Delete\n"             );
printf("-------------------------------------------------\n" );
printf("    请选择你的操作[0~23]: "                          );
```

图 1-1　菜单设计代码

1.3　系统实现

下列函数操作的前提是链表存在，故在每个函数的开始都增添了判空的条件语句，当链表的头结点为空时，返回 INFEASIBLE，直接终止函数。

1.3.1　基础功能

- 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；

  因为链表结构体在构建设计时在其中添加了名为 next 的结构体指针实现结构体单元的关联，但该指针是并未被赋值的，因此链表的初始化的一个重要步骤就在于使用 malloc 函数给结构体指针赋值指向固定空间大小的地址，初始化表即是创建一个头结点，但值得注意的是为保证程序的稳定，在创建完头结点后有必要给头结点的 next 指针赋空值。

- 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；

  在对链表的销毁中，我们必须知道，不能直接把头结点赋为空值而作为销毁链表的程序运行结果，在链表创建的过程中，每个结点的创建都需要使用 malloc 函数分配空间，因此倘若程序终止运行，在程序运行的过程中，这些分配的空间是会一直占用的，如若只是单纯地清空头结点，那么这些表面上失去索引而消失的链表结点实际上仍会占用系统空间，如果又在此之上有不断创建链表，程序的运行空间会越来越大，最终导致的结果可能会是内存溢出而导致程序崩溃。

  因此，为了避免这种错误，我们在销毁链表时需要对链表中的每一个结点进行清空，我们的清空过程会使用 free 函数，该函数可以释放由 malloc()、calloc()、realloc() 等函数申请的内存空间，可以及时释放占用的空间，使程序占用的空间于一个平稳的状态，有效提高程序的稳定性。

  清除的过程采用 while 循环直到出现空指针，在清空所有结点后，我们需要把头指针 L 赋值为空，至此才算真正完成对链表的销毁。

- 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；

  与上一个函数销毁表唯一的区别仅在于是否对头结点进行处理，销毁的意思即为是整个表彻底消失，而清空是在保证表存在的前提之下实现对表全部内容的清空，而表存在的前提就是头结点的存在与否，所以该函数对销毁表函数进行改进，清空从头结点的 next 结点开始清空，且循环结束后的赋值改为对头结点的 next 指针赋空值。

- 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；

  空表的定义即为头结点存在而表中没有任何内容即没有任何子结点，因此除开判断头结点是否为空之后，仅需要判断头结点的 next 指针是否为空即可。

- 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；

  判断链表的长度实际上就是对链表进行一次全遍历，没历经一个子结点就对创建的一个 int 类型初值为零的 length 变量进行加运算，当 while 循环跳

出即遇到指针时，则链表全遍历完毕，这时再把 length 作为返回值返回即可。

- 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≤i≤ListLength(L);操作结果是用 e 返回 L 中第 i 个数据元素的值;

  因为函数参数 i 已经给定范围，所以我们可以直接对数据进行处理，引入一个 int 型变量 count 记录当前链表遍历的位置，当 count==i 时即可跳出循环。为了判断要查找的该元素是否存在设置了如下判断条件，在程序末尾判断结构体指针是否为 null 值，如果为空值，则说明 while 循环一直遍历到表尾而仍未跳出循环，即没有找到所需元素，这时返回 ERROR。若不为空，则说明找到目标元素，把目标元素的值赋给 e，返回 OK，即可完成对该功能的需求。

- 查找元素：函数名称是 LocateElem(L,e,compare())；初始条件是线性表已存在；操作结果是返回 L 中第 1 个与 e 满足关系 compare() 关系的数据元素的位序，若这样的数据元素不存在，则返回值为 0。

  LocateElem 函数和 GetElem 函数可是说是具备着几乎一模一样的程序架构，不同之处在于函数返回值的不同，GetElem 最终会返回 e 的值，而 LocteElem 返回的是目标元素在链表的物理位置，LocateElem 函数在 while 循环里加了一个 if 判断语句，当遇到第一个满足 compare() 关系的元素就直接 return position 给主函数。在本次实验中，compare () 关系即为 == 等价关系。

- 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。

  因为链表时顺序遍历的，为了保证能获取到元素的前驱，所以采取使用两个结构体指针，一个为快指针，初值为 L->next->next, 另一个为慢指针，初值为 L->next，当快指针遇到目标元素是即可给 pre 赋上慢指针的值。

  在此需要特殊处理的是链表的首节点的情况，因为首节点不存在前驱，为此在函数开头判断如果慢指针的指针域为空，则直接返回 ERROR。

- 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。

  与前面的获取前驱的函数思路相同，但是由于链表的顺序遍历特性，因此

相较于获取前驱，获取后驱不需要设置多个指针，在处理特殊情况是也不需要加以特殊的判断语句，只需对 while 语句的判断条件略加改进，由 head!=NULL 改为 head->next!=NULL，以此完成对核心代码的设计。

- 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≤i≤ListLength(L)+1；操作结果是在 L 的第 i 个位置之前插入新的数据元素 e。

  结合之前的获取前驱函数的思想，设置快慢两个指针，当快指针找寻到目标元素，那么就新创建一个结点作为慢指针的后驱，而新结点的后驱也赋值为快指针指向的目标元素，以此完成对元素的插入。

  这样的做法需要注意的地方是：基于循环结束的条件 fast != NULL，当链表为空表且插入位置为开头时，我们需要进行单独的判断即处理，因为此时 fast 的值为空值，不会进行 while 循环；同理的是，当插入位置在表尾时，fast 的值同样为空值，这时，程序会跳出 while 循环，也不会执行插入操作，也需要在 while 里面加上单独的特殊判断语句，当插入位置为队尾是就直接执行操作。

```
    if(L->next == NULL && i == 1) {
        L->next = (LinkList)malloc(sizeof(LNode));
        L->next->data = e;
        L->next->next = NULL;
        return OK;
    }
    else if(L->next == NULL && i != 1)
    return ERROR;
```

```
if(fast == NULL && position == i-1) {
    LNode *insert = (LinkList)malloc(sizeof(LNode));
    insert->data = e;
    insert->next = slow->next;
    slow->next = insert;
    return OK;
}
```

图 1-2　fast 指针为空的特殊处理

- 删除元素：函数名称是 ListDelete(L,i,e)；初始条件是链表 L 已存在且非空，1≤i≤ListLength(L)；操作结果：删除 L 的第 i 个数据元素，用 e 返回其值。

  运用类似的思维方式，初始化一个遍历指针，然后再定义一个前置指针用于保存的遍历指针指向结点的上一个结点，在遍历指针遍历到目标函数时，采取 free 函数清空目标结点同时将前置指针指向结点的 next 指针指向目标结点的下一结点，以此完成对结点的删除。

- 遍历表：函数名称是 ListTraverse(L,visit())，初始条件是链表 L 已存在；操

作结果是依次对 L 的每个数据元素调用函数 visit()。

遍历的表的思想在上述几乎绝大多数函数中得以呈现，遍历均体现出链表的线性结构，在此就不加以过多叙述。进一步健全函数，在开头即可判断链表是否为空，如果为空，直接输出信息告诉使用者并终止函数即可。

1.3.2 附加功能

- 链表翻转：函数名称是 reverseList(L)，初始条件是线性表 L 已存在；操作结果是将 L 翻转；

  整体上的思路在于是原链表的第一个结点以此后移，每次后移一位就把原位的结点放置到 L->next 处即作为头结点，具体难点在于如何把后一结点移动到头结点位置。算法采取首插法，即每次在首部插入元素以此实现倒序，

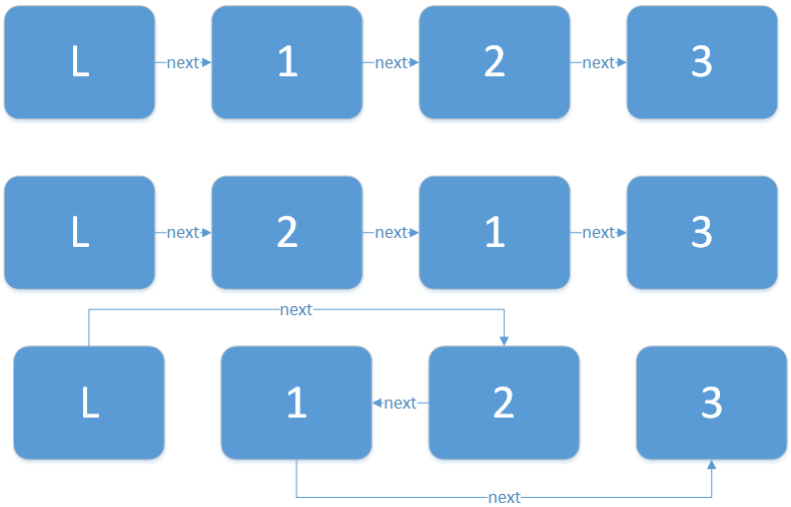


图 1-3 翻转链表的大致思路

如图以一个长度为 3 的链表为例，辅助我们理解结点的移动，本质上就是改变结点的 next 指针的指向，为了实现这样的设想，我们现需要定义一个 temp 结构体指针，该指针指向原始链表的头结点 head 的下一结点，即 temp = head->next，head 的指向在之后的函数运行过程中一直没有发生改变，即一直指向图中值为 1 的结点，temp 当前指向值为 2 的结点然后 head 的 next 指针指向 temp 的 next 指向的结点，即图中值为 3 的结点，再让 temp 的 next 指针指向 L 的 next 指向的结点即 head 结点，最后让 L 的 next 指针指向 temp 即可完成结点的移位，在后续结点的插入过程采取同样的方法，当初始链表的第一个结点移动到链表尾部即 head->next 为空值时，即可说明链表已经

翻转完毕。

在程序运行过程中，head 的指向保持不变，而 temp 也始终指向 head 的 next 指向的结点，但是 head 的 next 指针的指向是在不断变化的，每次的操作实际上就是把当前 head 的 next 指针指向的结点移位至表首, 让 L->next 指向 head->next，以此循环直至跳出循环。

- 删除链表的倒数第 n 个结点：函数名称是 RemoveNthFromEnd(L,n); 初始条件是线性表 L 已存在且非空, 操作结果是该链表中倒数第 n 个节点；

  对于这个功能的实现，我采取了一个投机取巧的办法，运用上一个函数——链表的翻转，在链表翻转过后，对于原链表倒数的结点不就变成正数的结点序号了吗？于是，在函数的最开始先翻转函数，再调用之前写过的删除结点函数删除对应位置的结点即可，在函数结束时再把链表翻转回来即可。

  但这样在链表体量比较大的时候显然有些欠缺，于是想出另一个做法是，设置两个指针，慢指针比快指针慢 n 个结点，当快指针遇到链表尾时，慢指针指向的位置即为要删除的位置。

- 链表排序：函数名称是 sortList(L)，初始条件是线性表 L 已存在；操作结果是将 L 由小到大排序；

  对于数组的排序我们都不陌生，我们对于数组排序常用的冒泡排序，对于链表来说同样适用，但需要作出略微的改动，链表交换有两种方式，一种是交换指针域，另一种是交换数据域，本函数采取的是交换数据域，具体思路如下：

  数据的交换与数组的交换过程类似，都需要一个中间变量 temp 作为媒介进行交换，难点在于如何让冒泡排序在数组排序中的两个 for 循环在此得以体现，因为链表是不存在下标访问的，在此采用的方法是，定义头尾两个指针，头指针每遍历一次链表，尾指针就往前移动一个结点，然后头指针又赋值回头结点的位置，while 的结束条件改为头指针的 next 不等于尾指针，以此实现类似于数组交换第二个 for 循环的判断条件 j < len - i - 1 这样的判断效果。

- 实现链表的文件形式保存：其中，需要设计文件数据记录格式，以高效保存线性表数据逻辑结构 (D,R) 的完整信息；需要设计链表文件保存和加载操作合理模式。

  文件的存储与读取是相互配套的，文件存储的形式决定文件读取的方式，对

于链表这样的简单的线性数据结构，实现文件的存储较为容易，只需满足把每个结点的数据以空格分开的形式用 fprintf 函数存储在文件中，文件的打开形式是'w'，使得每次写入文件都是从头开始写入；在读取文件时，使用 fscanf 函数，利用它不读取空的特性，结合链表的创建初始化，实现读取每个结点的信息并加以创建，文件的打开形式是'r'，保证原文件不会发生改变。

在使用文件指针 fp 时，我们的一个好习惯是在使用 fopen 函数后应加一个空值判断，以确保文件能正确打开，避免程序错误，同时在程序结尾文件使用完毕后，也应使用 fclose 函数将文件关闭，避免溢出。

- 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。

  针对多线性表的管理，实验设计一个新的结构体，其中定义了本实验链表的一个结构体数组，本质上是通过数组存储多个链表以此实现多线性表的管理，针对数组中每个线性表的管理和上述基础功能对单个链表的操作基本一致，不同在于，我们需要实现不同链表的切换，即找寻并切换至目标链表进行管理。

  在设计过程中每一个链表都有一个 name 数组对链表进行唯一命名，因此，我们可以采取 strcmp 函数通过在输入流输入链表的名字实现对不同链表的切换。

  针对多线性表操作的函数主体架构均和单链表操作的主体架构一直，不同仅在于在函数的开头有判断找寻目的链表的 strcmp 函数操作，找寻到目标函数则进行后续操作，没有找到目标链表直接返回 ERROR。

1.4 系统测试

以下测试数据以及结果均来自与 educoder 平台以及本机终端，保证数据的精确性以及可靠性、真实性。

- InitList 测试：

  正常测试：0 即创建一个空的头结点指针并传入函数；

  异常测试：1 即在主函数中先创建一个非空链表然后把该链表传入函数；

图 1-4 InitList 函数测试

- DestroyList 测试：

  正常测试：输入数据并创建好一个非空链表后，传入函数，并在主函数中判断链表是否结点均清空，成功返回输出 OK；

  异常测试：传入一个不存在链表，返回并输出 INFEASIBLE；

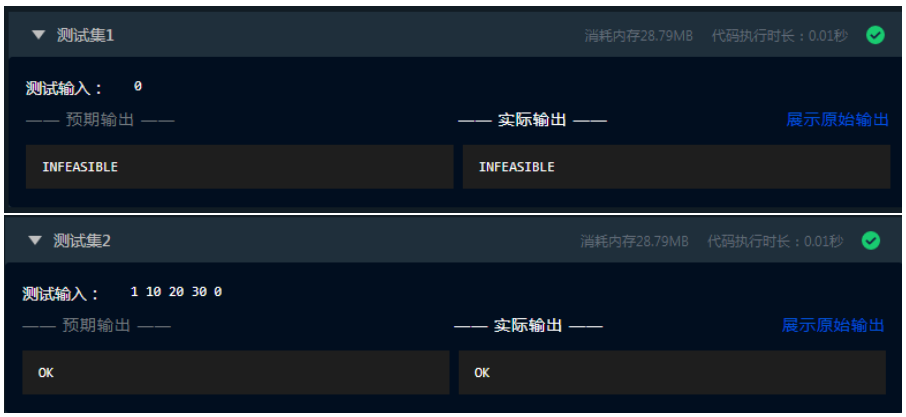


图 1-5 DestroyList 函数测试

- ClearList 测试：

  正常测试：输入数据并创建好一个非空链表后，传入函数，并在主函数中判断除头结点外链表是否结点均清空，即是否为空链表，成功则输出 OK；

异常测试：传入一个不存在链表，返回并输出 INFEASIBLE；

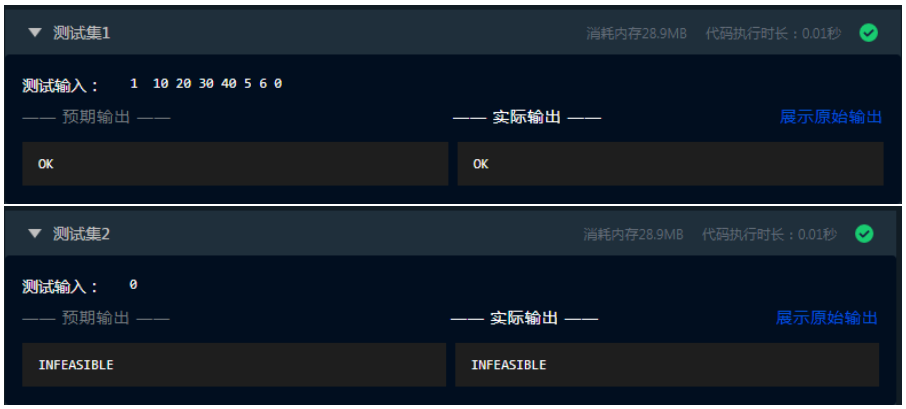


图 1-6　ClearList 函数测试

- ListEmpty 测试：

  正常测试：创建一个非空链表后传入函数，返回 FALSE；创建空链表后传入函数，返回 TRUE；

  异常测试：传入一个不存在链表，返回 INFEASIBLE；

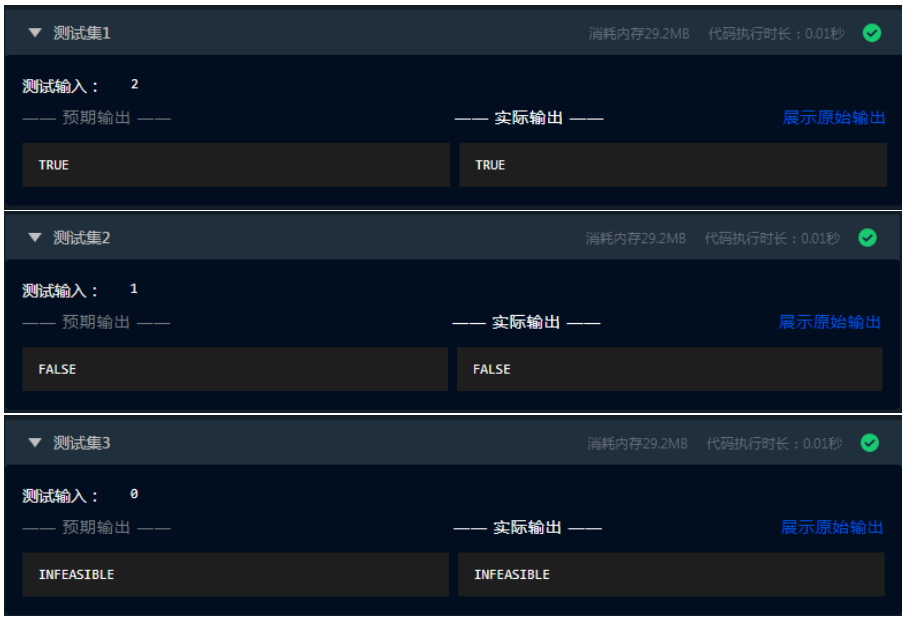


图 1-7　ListEmpty 函数测试

- ListLength 测试：

  正常测试：创建一个长度为 5 的链表，传入函数，函数返回值为 5；

  异常测试：传入一个不存在链表，返回 INFEASIBLE；

图 1-8　ListLength 函数测试

- GetElem 测试：

  正常测试：创建一个长度为 5 的非空链表后传入函数，在函数长度范围内查询元素，返回 OK；

  异常测试：创建一个长度为 5 的非空链表后传入函数，在函数长度范围之外查询元素如输入的查询位置为 0 或者 8，返回 ERROR；传入一个不存在链表，返回 INFEASIBLE；

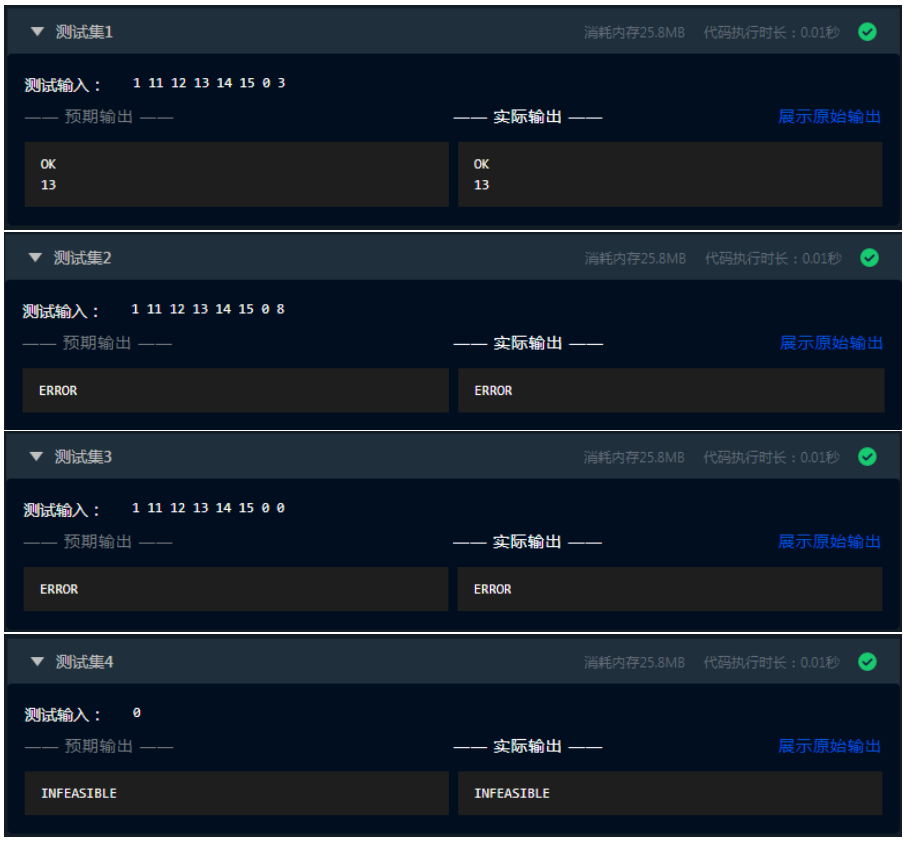


图 1-9　GetElem 函数测试

- LocateElem 测试：

  正常测试：创建一个数据为 11 12 13 14 15 长度为 5 的链表，查询 14 的位置，返回 4；

  异常测试：传入同样链表查询不存在的元素 18 的位置，返回 ERROR；传入一个不存在链表，返回 INFEASIBLE；

图 1-10　LocateElem 函数测试

- PriorElem 测试：

  正常测试：创建一个数据为 11 12 13 14 15 长度为 5 的链表，查询 14 的前驱，返回 13，输出 OK；

  异常测试：查询 11 的前驱，返回 ERROR；查询不存在元素 18 的前驱，返回 ERROR；传入一个不存在链表，返回 INFEASIBLE；

图 1-11　PriorElem 函数测试

- NextElem 测试：

  正常测试：创建一个数据为 11 12 13 14 15 长度为 5 的链表，查询 14 的后驱，返回 15，输出 OK；

  异常测试：查询 15 的后驱，返回 ERROR；查询不存在元素 18 的后驱，返回 ERROR；传入一个不存在链表，返回 INFEASIBLE；

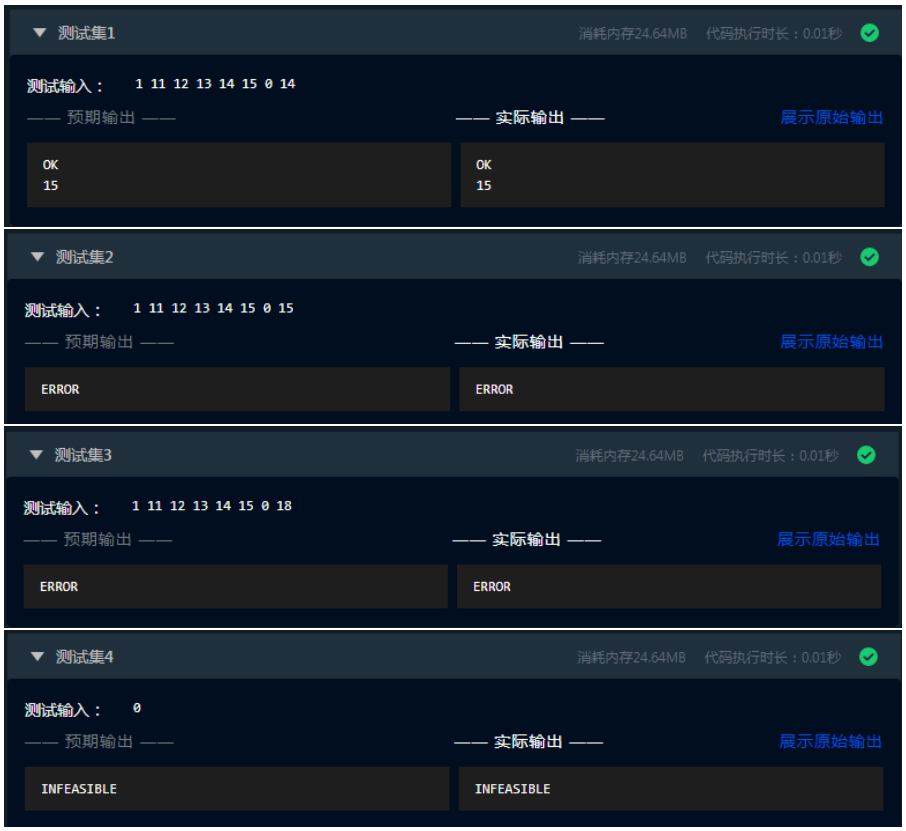


图 1-12　NextElem 函数测试

- ListInsert 测试：

  正常测试：创建一个数据为 1 3 5 7 9 2 4 6 8 10 的链表，在第五个元素前面

插入 80，返回 OK；

异常测试：创建一个数据为 10 20 30 的链表，在第 0 个和第 5 个位置插入元素，返回 ERROR；传入一个不存在链表，返回 INFEASIBLE；

特殊数据测试：传入正常测试的链表数据，在最大容量元素的后一元素位置前插入，本实验中链表最大容量为 10，但可以在第十一个元素的前面插入元素，返回 OK；传入一个空链表，在第一个位置前插入，返回 OK；

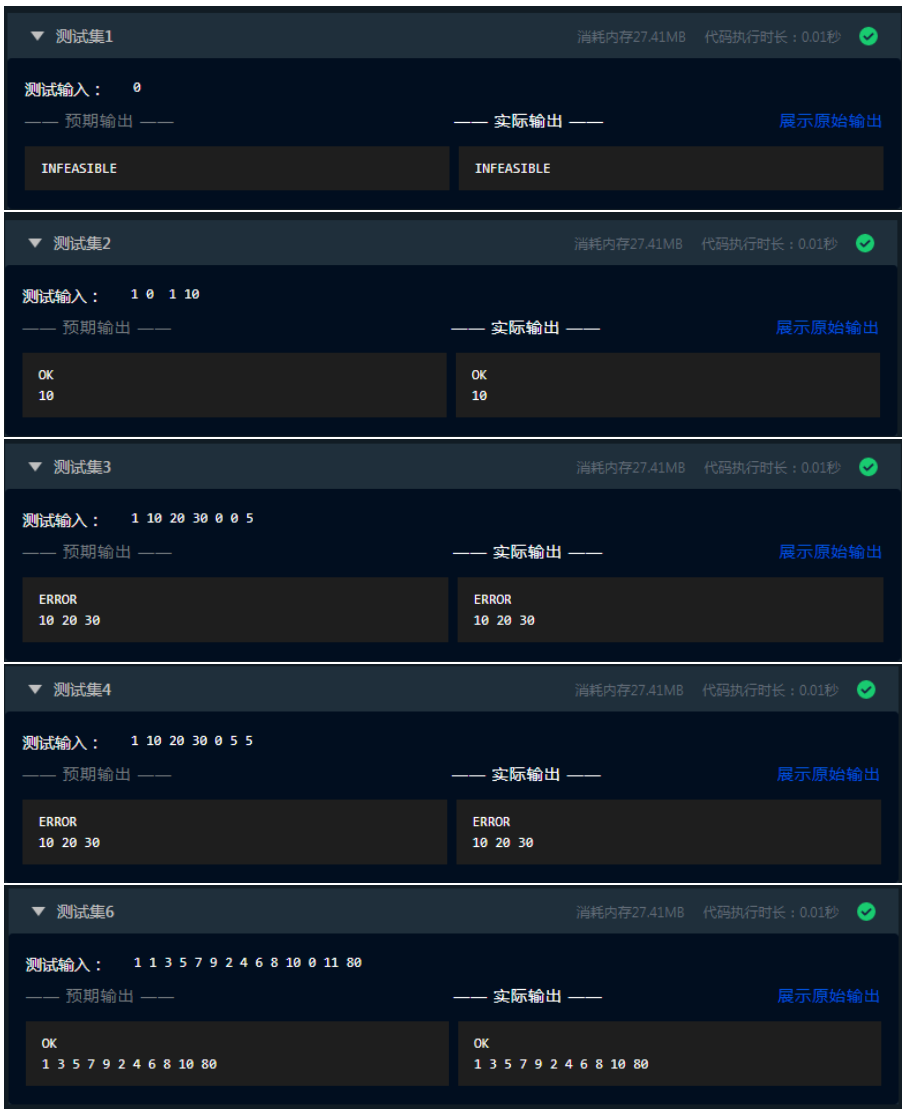


图 1-13　ListInsert 函数测试

- ListDelete 测试：

正常测试：创建一个数据为 11 12 13 14 15 的链表，删除 3 位置的元素返回 OK；

异常测试：同样的链表，删除 0 位置和 6 位置的元素，即超出链表长度范围删除，返回 ERROR；传入一个不存在链表，返回 INFEASIBLE；

特殊数据测试：表首和表尾数据删除测试，返回 OK

图 1-14　ListDelete 函数测试

- ListTraverse 测试：

  正常测试：遍历输出数据为 11 12 13 14 15 的链表；

  异常测试：传入一个不存在链表，返回 INFEASIBLE；

  特殊数据测试：传入一个空链表，输出该链表为空；

图 1-15　ListTraverse 函数测试

- reverseList 测试：

  正常测试: 翻转数据为 11 12 13 14 15 的链表，并遍历输出;

  异常测试：传入一个不存在链表，返回 INFEASIBLE；

  特殊数据测试：传入一个空链表；

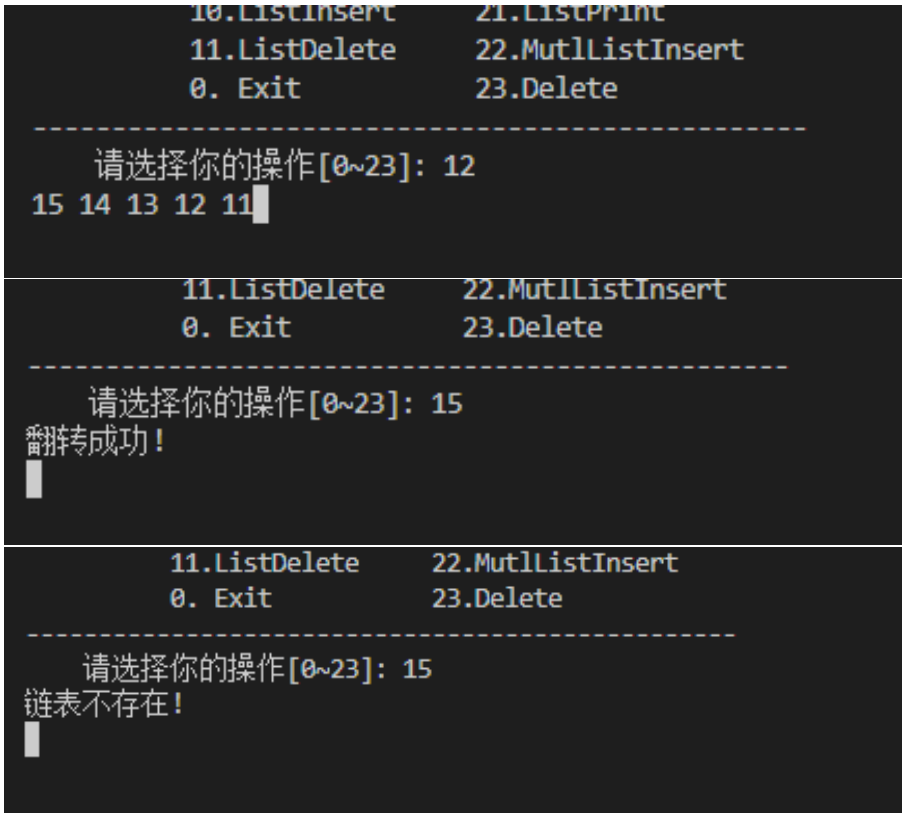
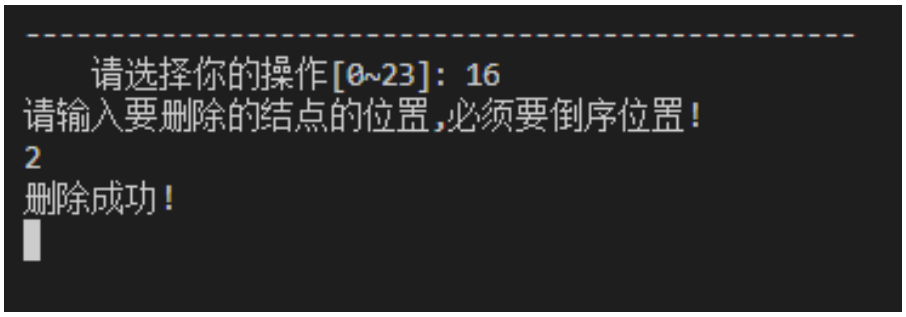


图 1-16　reverseList 函数测试

- RemoveNthFromEnd 测试：

  正常测试：创建一个数据为 11 12 13 14 15 的链表，删除倒数第二个结点；

  异常测试：传入一个不存在链表，返回 INFEASIBLE；



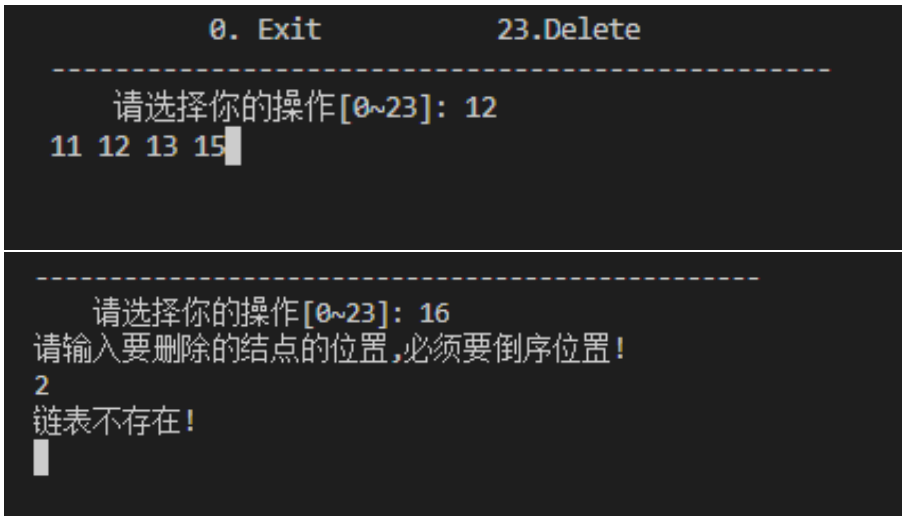


图 1-17　RemoveNthFromEnd 函数测试

- sortList 测试：

  正常测试：创建一个数据为 1 3 5 2 4 的链表，排序并输出;

  异常测试：传入一个不存在链表，返回 INFEASIBLE;

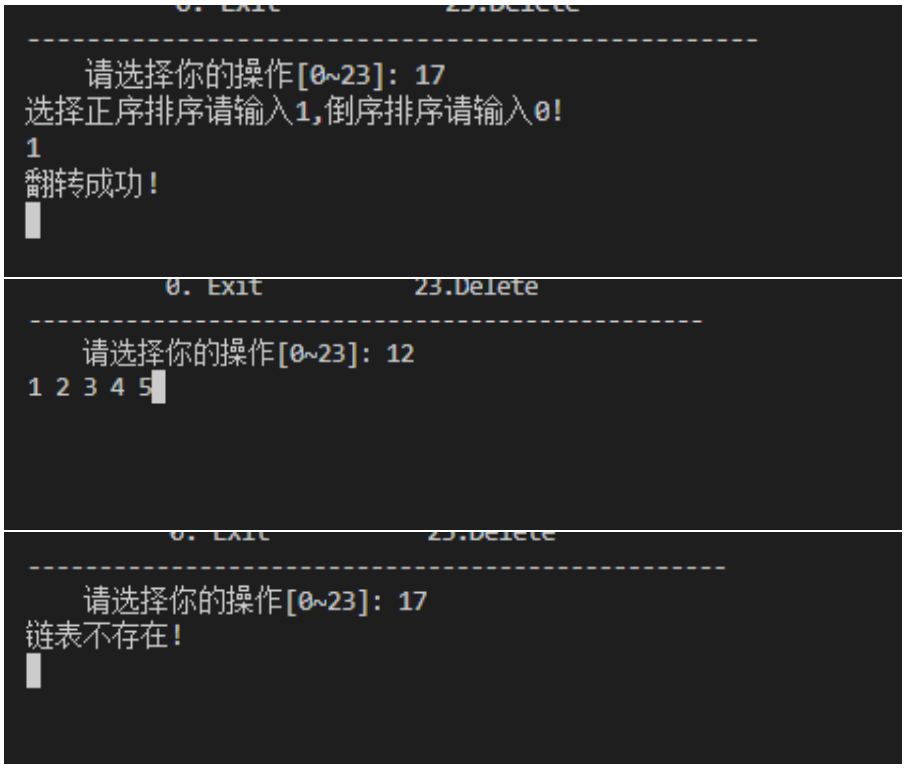


图 1-18　RemoveNthFromEnd 函数测试

- SaveList、LoadList 测试：

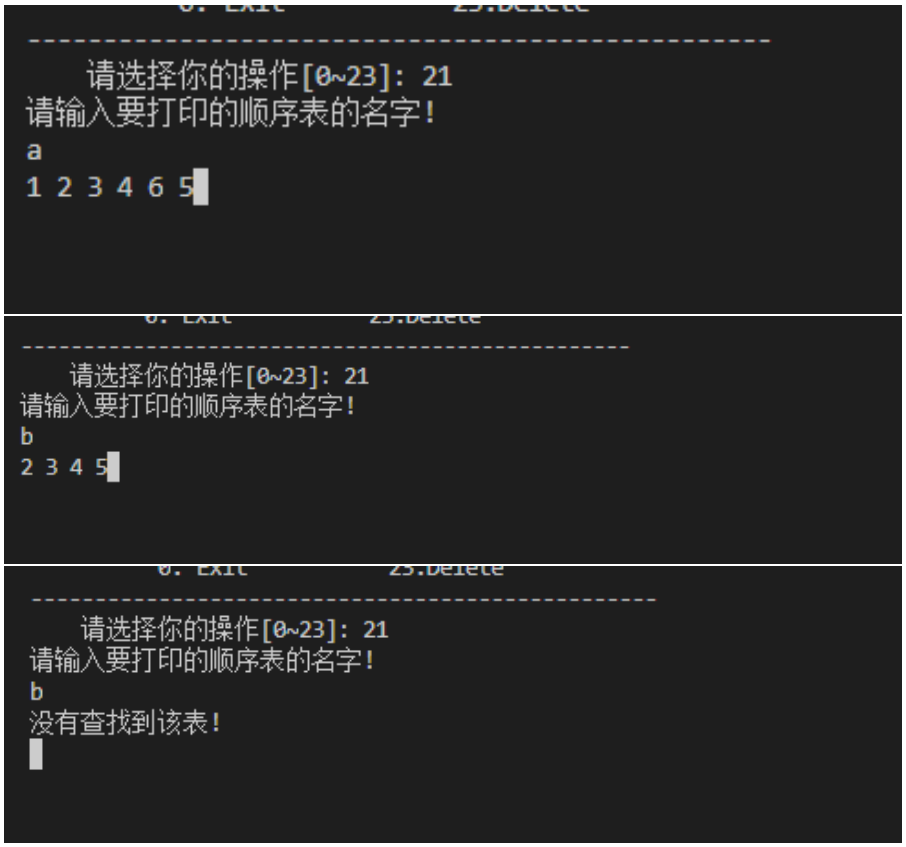
  正常测试：创建一个非空链表后传入函数，存入文件后销毁链表，在重新读取文件创建链表并输出;

异常测试：传入一个不存在链表，返回 INFEASIBLE；

图 1-19　SaveList、LoadList 函数测试

- 多线性表测试：

  创建名为 a，b 的两个链表, 两个链表的数据均为 1 2 3 4 5，对 a 增添数据 6 在 4 和 5 中间，对 b 删除数据 1; 随后销毁表 b，再创建一个空表 c



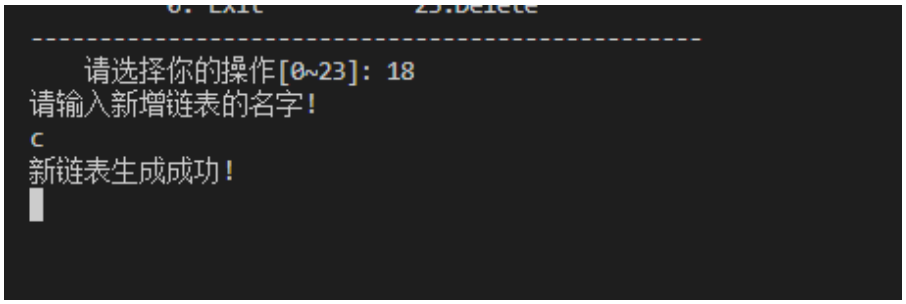


图 1-20　SaveList、LoadList 函数测试

1.5　实验小结

重点说明在实验中取得的实际经验，例如调试中碰到的典型错误等，不要写套话。

在编写 ClearList 的过程中一开始并不能通过头歌平台的测试，对自己的函数一番审视过后发现，后来加入了 L->next = NULL; 这个语句后就顺利地通过了测试，在我百思不得其解，后通过上网查资料才得知，free 函数不能使 free 掉指针指向空，即指针所指向的地址没有发生改变，只是指针指向地址存储的内容均已被情况，因为我们是为了清空链表，L->next 依照定义应该指向空，也避免程序报错，因此，将 L->next 赋值为空是必要且必须的。

GetElem 函数在实验要求中，虽然对 i 的值做了规范，但难免会出现使用者输错值的情况，为了是程序更加的完善，有必要对不合法的 i 进行报错，并输出相关信息提示。

PriorElem 在编写设计的过程中设置了快慢指针，但是在最初的调试过程中发现，函数对空表进行运行时会导致程序报错退出，使用单步调试后发现，快指针 fast 指向的是 L->next->next，当 L->next 为空，即链表为空时，是不存在 L->next->next 的，因而程序会退出崩溃，这样提示我，在未来程序的编写中应当注意跨度大的指针能否存在的问题。

ListInsert 在头歌平台评测时，最初没有通过特殊情况的数据测试，分别是在空表时进行的插入，以及非空表情况下的尾部插入，空表时插入失败的原因和 ClearList 的问题非常类似，因为 while 循环的判断条件为 fast != NULL，而表为空表时，fast 的值为空，那么这时就不会进入 while 循环进行插入操作；同理，尾部插入失败也是一样的道理，while 循环的末尾，fast = fast->next，当 fast 为表尾时，判断就直接跳出，假若此时插入位置也刚好在表尾，因此跳出循环而不会进行插入操作，因此，对于这两种情况，在函数里都做了特殊判断。

在编写获得前驱这些需要获取目标结点的上一结点的函数时，我都运用了两个指针一快一慢来确保能够获取前一个指针，但其实完全没有必要这么麻烦，要求获取第 i 个元素的前驱，那么我直接定位获取第 i-1 位的元素不就行了吗？这样的做法就不需要定义两个指针，而且在表首没有前驱的问题上也不需要特殊判断，只要 i-1 大于等于 0 那么就即可插入, 这是我的程序可以有所改进的地方。

2 基于二叉链表的二叉树实现

本章将实现新的数据类型结构——二叉树，并实现和探讨二叉链表结构之下的非线性结构的基础功能的实现及其设计原理。

2.1 问题描述

- 加深对二叉树的概念、基本运算的理解；
- 熟练掌握二叉树的逻辑结构与物理结构的关系；
- 以二叉链表作为物理结构，熟练掌握二叉树基本运算的实现。

2.2 系统设计

基于二叉树的定义：二叉树是一种每个结点至多只有两个子树（即二叉树的每个结点的度不大于 2），并且二叉树的子树有左右之分，其次序不能任意颠倒。二叉树同时又需具备存储数据的能力，于是设计如图所示结构体：

基本结构设计代码

```
typedef  int   status ;
typedef  int  KeyType;
typedef  struct  {
        KeyType key;
        char  others [20];
} TElemType; //二叉树结点类型定义

typedef  struct  BiTNode{  //二叉链表结点的定义
        TElemType data ;
         struct  BiTNode *lchild,*rchild ;
} BiTNode, *BiTree;
```

二叉树的结构体中包含名为 data 的结点类型结构体变量，同时还包含左右子树的结构体指针，后续树的创建过程中分别指向本结点的左右子结点，完成树结点间的连接。

结点类型结构体定义为 int 类型的关键词，这是值唯一的数据，保证结点数据不会出现重复。char 类型的 others 数组则就是用来存储数据的。为了实现后续的多二叉树管理，设计如下多二叉树管理的结构体：

多二叉树结构设计代码

```
typedef  struct {   //多二叉树的管理表定义
 struct  {
          char  name[30];
          BiTree  T;
} elem[10];
int  length ;
int   listsize  ;
}LISTS;
```

结构原理同链表操作中的多链表管理类似，在此不再加以解释。

在程序实际运行的可视化简易菜单上，采用 printf 的简易方式打印出菜单并设计 while 循环实现程序的多次操作，并根据输入数字对用 switch 函数中的不同具体功能选项。

```
system("cls");  printf("\n\n");
printf("      Menu for Linear Table On Sequence Structure \n"    );
printf("-------------------------------------------\n"    );
printf("          1. CreatrBiTree      2. ClearBiTree\n"    );
printf("          3. BiTreeDepth       4. LocateNode\n"    );
printf("          5. Assign            6. GetSibling\n"    );
printf("          7. InsertNode        8. DeleteNode\n"    );
printf("          9. PreOrderTraverse  10.InOrderTraverse\n"    );
printf("          11.PostOrderTraverse 12.LevelOrderTraverse\n"  );
printf("          13.SaveBiTree        14.LoadBiTree\n"    );
printf("          15.MaxPathSum        16.LowestCommonAncestor\n");
printf("          17.InvertTree        18.AddList\n"    );
printf("          19.TreeChange        20.BiTreeEmpty\n"    );
printf("           0.exit\n"    );
printf("-------------------------------------------\n"    );
printf("    请选择你的操作[0~20]: "    );
```

图 2-1　二叉树菜单代码设计

2.3　系统实现

下列函数操作的前提是链表存在，故在每个函数的开始都增添了判空的条件语句，当链表的头结点为空时, 直接终止函数。

2.3.1 基础功能

- 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；

  definition 在当前函数以及后续几乎所有函数中都是遵循这样的输入原则，如：1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null

  definition 会按照先序遍历的顺序给出所有包括空结点在内的结点，在此基础之上的树的创建的函数思路如下：

  因为 definition 采用先序存储方式，所以函数也采取先序构建树，由先序的特性我们可以知道，任何一个结点的右子树的创建一定是在其左子树已经构建完的基础之上，因此一棵树一定会先创建完左子树再开始创建右子树，创建过程如图：基于这种创建方式，避免不了回溯的问题，在非递归的算法

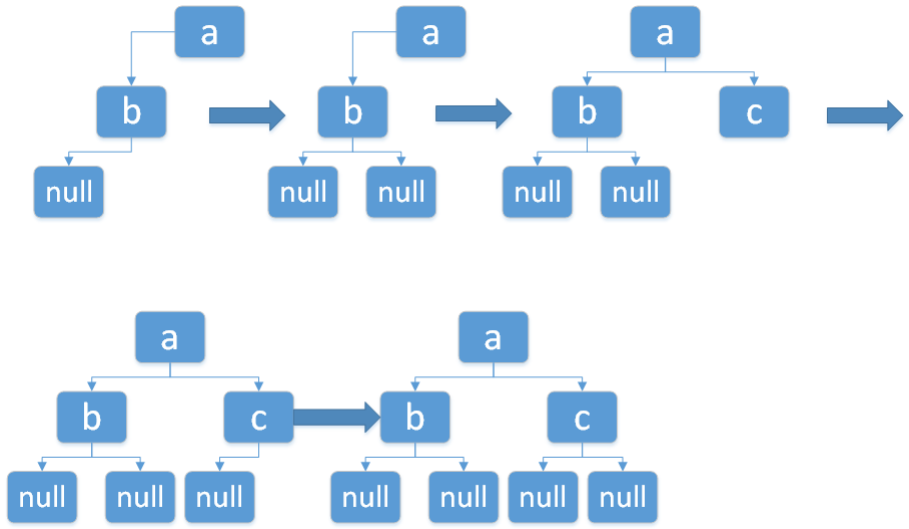


图 2-2 二叉树创建流程示意图

之下，本函数采用使用栈来进行回退，当一条分支不能再创建左子树时，即图中第一步时，那么就开始给最末端的树叶创建右子树，树叶创建好后，如果右子树也为空，那么就会开始回退到上一个根节点即图中的 a 结点，此时有一个问题显而易见，我们如何判断栈中回溯的结点是否已经创建了左子树呢？这个时候通过引入一个与栈同步的标记数组，初值为 0，把栈中已经创建左子树的结点的位置在数组中标记为一，回退时通过判断数组值是否为 1 来决定回溯结点是创建左子树还是创建右子树。

在函数的 while 循环中把创建左子树放在前面，把创建右子树放在后面，而且创建右子树的函数功能增添一个判断语句来判断是否应该启动，在创建完一个结点的左右结点后，我们要记得进行退栈操作，同时把标记数组对应位置初始化为 0。

针对特殊情况的数据处理，例如树为空的情况，本函数在函数开头就加以判断，若为空，则直接返回 OK，表示空树创建完毕。

- 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；

  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；

  这里将清空与销毁一同叙述，与链表的销毁清空类似的是，这个过程都需要全部遍历一遍树的所有结点，并使用 free 函数清空遍历过的结点。清空与销毁的不同又在于是否对头结点进行 free 函数操作，清空不对头结点操作，而销毁是会对头结点进行清除操作。

  本函数采用非递归后序遍历的方式实现树的遍历的。

- 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；

  仅需要判断头结点指针是否为空即可，T 指向 NULL 则返回 TRUE，不为空则返回 FALSE。

- 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；

  该函数实现的核心思想是，遍历到一个层数最大的一个分支，本函数的思想是，在非递归的算法中，遍历为了实现回退需要用到栈，而栈所能达到的最大高度是和树的层数有关的，在本函数采取的后序非递归遍历中，栈的栈顶位置就等于当前所处的层数，所以新定义一个 max 变量用于保存在遍历过程中栈顶位置的最大值，在遍历完所有结点后，返回的 max 值就即为二叉树的深度。

- 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；

  该函数的返回值为指针类型，在查找过程中基于关键字的唯一性，所以通

过比较语句发现关键字相同的结点即可把当前遍历的结点作为返回值返回，在循环外返回空指针，表示遍历过程如果没有正常 return，就说明树中没有对应的结点，返回空指针表示查找失败。

- 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；

  该函数的核心算法以及思想与上面的查找结点函数相同，都是在遍历过程中找到对应的结点，但是我们值得注意的地方是，我们重新赋完值后，仍需保证树中不会存在重复的关键字，即须保证关键字的唯一性，所以我们不能在遍历到目标结点就直接完成赋值操作并返回，而是先把目标结点的信息先保存下来，在遍历完所有结点后，再进行赋值操作。函数定义一个标记数组，初值为 0，当关键字出现过一次时就把标记值改为 1，在遍历完成后，我们需要判断标记数组中重新赋值的关键字的标记值是否为 0，为 0 说明该关键字没出现过，为 1 则说明该关键字已经存在，不能进行重新赋值操作，返回 ERROR。

- 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；函数的基本思路是：运用后序遍历找到目标子结点，然后回溯到该结点的父节点，然后返回该父节点的另一子节点。

  思路很简单，难点在于如何回溯到父节点，由后序遍历的特性我们知道，后序遍历是先遍历两个子节点再遍历父节点的，因此在非递归算法中，一个结点在栈中位置的下一个结点必然为它的父节点，后序每次都是遍历栈顶元素，因此，目标结点的父节点就即为 Stack[top-2] 保存的结点，获得父节点后再通过 if 语句判断目标结点是父节点的左子树或者右子树，依据结果返回父节点的右子树或者左子树。

- 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？可以考虑 LR 为-1 时，作为根结点插入，原根

结点作为 c 的右子树。

该函数要实现的功能较多，首先考虑当 LR 为-1 的时候，操作相较于其他插入有所不同，因而单独拎出来进行操作，依据功能要求将插入根节点作为插入结点的右子树即可。

插入结点的算法是结合了结点查找和结点赋值两个函数的算法思想，首先需要查找到插入的位置，即查找到目标结点，但是同样的，为了保证插入结点之后仍能保证关键字的唯一性，我们需要保存目标结点的位置，然后让程序继续遍历完整个树，在此过程中用标记数组记录遇到的每一个关键字，并修改标记值为 1。

在遍历完成后，判断关键字是否重复，若重复则返回 ERROR，若唯一，那么再依据 LR 的值完成相应操作即可，赋值的过程较为简单，在此就不加以过多叙述。

- 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；

  该函数的主体思路与上面的函数大致相同，无非就是遍历到目标位置再进行操作，本函数的复杂之处在于对于删除结点不同度的要进行不同操作。

  度为 0 时的操作是最简单的，无需考虑删除过后子节点的后续处理工作，直接 free 掉该结点即可。

  度为 1 时，首先先判断该结点是其父节点的左子树还是右子树，这里运用到获取兄弟结点的思想获取父节点，判断完左右子树后，再让父节点的左右指针指向删除结点的子节点即可。

  度为 2 时，与度为 1 时的操作思路大致相同，只不过需要用一个 while 循环获取 e 左子树的最右子树的位置，具体操作过程和度为 1 的情况大致相同，在此略过。

  需要特殊处理的是对头结点的删除操作，因为对头结点的操作会改变 T 的指向，而非头结点的删除操作并不会改变 T 的指向，因此需要单独对这种情况进行特殊处理，处理的过程和非头结点的操作流程是一致的，只是操作

完后需要对 T 重新赋值。

- 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。
  实现思想：借助栈，对于根节点，先将当前节点压入栈中，然后遍历的时候弹出栈中的一个元素，输出，当该节点的右节点不为空时，将节点压入栈，当左节点不为空时，将左节点压入栈，前序遍历是根左右但是栈的数据结构时先入后出，先访问到左节点，需要将右节点先压入栈中。继续循环，弹出栈顶元素，输出，将右节点和左节点压入栈中。
- 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；
  实现思想：在第二次经过结点的时候才去访问结点数据，要一直去寻找结点的左子树，访问完左子树在返回结点并获取结点数据，然后访问右子树，重复这个过程，也就是说如果当前结点有左子树就要转去左子树，访问完左子树才访问当前结点，这个场景刚好可以使用栈来实现，有左子树则把当前结点入栈，访问完左子树，再出栈并访问结点数据。
- 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。
  实现思想：对于任一结点 P，将其入栈，然后沿其左子树一直往下搜索，直到搜索到没有左孩子的结点，此时该结点出现在栈顶，但是此时不能将其出栈并访问，因此其右孩子还为被访问。所以接下来按照相同的规则对其右子树进行相同的处理，当访问完其右孩子时，该结点又出现在栈顶，此时可以将其出栈并访问。这样就保证了正确的访问顺序。可以看出，在这个过程中，每个结点都两次出现在栈顶，只有在第二次出现在栈顶时，才能访问它。因此需要多设置一个变量 flag 标识该结点是否是第一次出现在栈顶。
- 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。

实现思想：该遍历采用队列的思想，没遍历到一个结点就该结点的左右孩子的结点存放在队列里，先将根节点入队，只要队列不为空，进入循环，输出根节点，并遍历根节点的左右孩子，若左右孩子不为空，则入队并输出；

2.3.2 附加功能

- 最大路径和：函数名称是 MaxPathSum(T)，初始条件是二叉树 T 存在；操作结果是返回根节点到叶子结点的最大路径和；

  该函数采用递归的思想，先设置 max 变量为全局变量，具体思路是，采用深度优先搜索，先获取一条分支的路径之和 sum，如果比 max 大，那么就把 max 赋值为还 sum 值。递归完一条路径后会回跳到上一个结点继续搜索，让 sum = sum - T->data.key，在每次探索完一个结点的所有左右孩子后回溯上一结点是，sum 值也应回溯到没有加上当前结点的状态，通过该计算即可实现对 sum 值的一个回溯，通俗来讲就是在每个结点都会保存到达这个结点之前的 sum 值，sum 值会依据保存的值进行加操作并与 max 比较以获取最大值。

```
int max = 0 ,sum = 0;
void PreView(BiTree T)
{
    if(T == NULL) return;
    sum = sum + T->data.key;
    PreView(T->lchild);
    if(sum > max)
    max = sum;
    PreView(T->rchild);
    if(sum > max)
    max = sum;
    sum = sum - T->data.key;
}
status MaxPathSum(BiTree T)
//初始条件是二叉树T存在；操作结果是返回根节点到叶子结点的最大路径和；
{
    PreView(T);
    return max;
}
```

图 2-3 求最大和递归算法

- 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；

  该函数的核心算法思想是运用完全二叉树结点的编号性质，结点在完全二叉树的父节点的序号与该结点序号的关系是向下取整子节点的序号除以 2 然后向下取整即为父节点的序号。

基于这种特性，我们首先给树通过增添空指针达到完全二叉树的效果，并根据它作为完全二叉树的排序赋值在结构体数组的对应位置中，然后根据两个结点的完全二叉树序号依次除 2 且向下取整直至两个结点序号相同，相同时即可说明该序号就是他们的公共祖先，在同该序号去数组中调取相应位置的结点即可。

- 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；

  该函数功能在此处采取层序遍历最为便捷方便，而且利于理解，因为左右互换并不会改变层序关系，因此在层序遍历的算法中，对结点的输出改为对该结点的两个孩子结点进行互换，互换需采用 temp 变量作为中间值。
- 实现二叉树的文件形式保存：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。

  针对树的文件存储形式，本函数采用 CreateBiTree 函数中 definition 的数据存储形式存储，因此，需要对树进行先序遍历，每遍历一个结点便把该结点的数据用 fprintf 函数写入文件，并用空格分开，遇到空指针就写入”0 null ”表示空指针，最后在读取完所有结点后写入”-1 null”，表示输入流结束，读取时，则就可以采取与 CreateBiTree 函数一模一样的创建思路创建一颗新树。
- 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。

  有了管理多线性表的经验，我们不难发现，多线性表管理的函数结构与单表的管理函数结构几乎一模一样，不同的地方仅仅在于操作的对象是可以变换的表，因此，在二叉树的多个管理中加入函数 TreeChange 使主函数中的二叉树指针 T 能依据 TreeChange 的返回值指向不同的二叉树，然后直接依靠现有的基础功能完成对二叉树的全部操作，无需再重复编写内容功能相同的函数，具体的实现办法是，在主函数中定义一个 record 变量，每次使用完 Treechange 函数后，record 值都会更改为多线性表中目标二叉树的位置。在主函数中成功调用 CreateBiTree 函数就会让 L.elem[record].T = T 以此完成多线性表与单树的联系。

2.4 系统测试

主要说明针对各个函数正常和异常的测试用例及测试结果画图说明网页的整体框架，进行简要的文字描述等

- CreateBiTree 测试：

  正常测试：1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null 的测试数据；

  异常测试：1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 3 e 0 null 0 null -1 null 关键字不唯一的测试数据；

  特殊数据测试：0 null -1 null 树为空树的情况。

图 2-4 CreateBiTree 函数测试

- ClearBiTree 测试：

  正常测试：非空树；

  特殊数据测试：空树

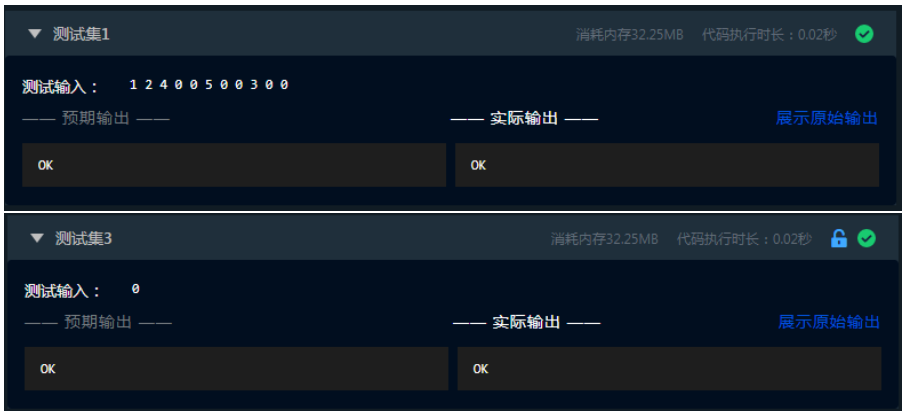


图 2-5　ClearBiTree 函数测试

- BiTreeEmpty 测试：

  正常测试：1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null 非空树，0 null -1 null 空树。

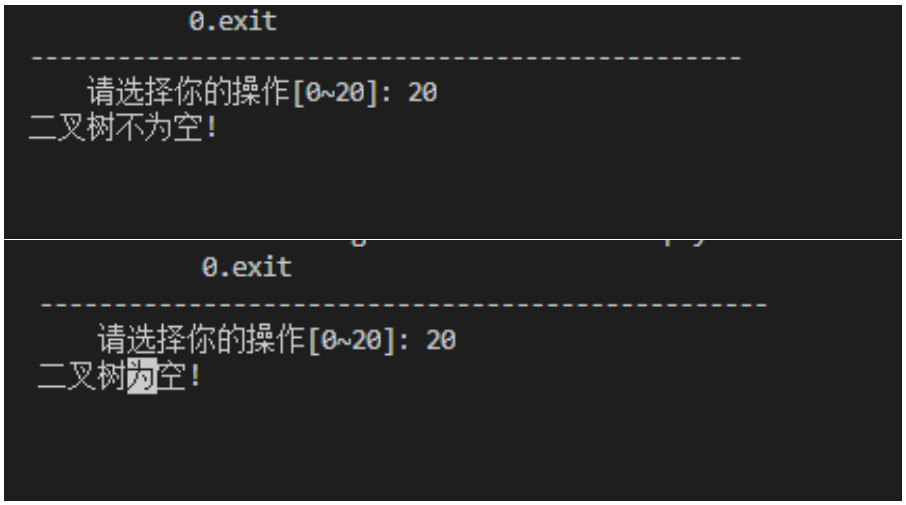


图 2-6　BiTreeEmpty 函数测试

- BiTreeDepth 测试：

  正常测试：1 1 a 2 2 b 3 3 c 6 4 d 7 5 e 0 0 null 创建深度为 3 的树，返回 3；

  特殊数据测试：空树，返回 0；

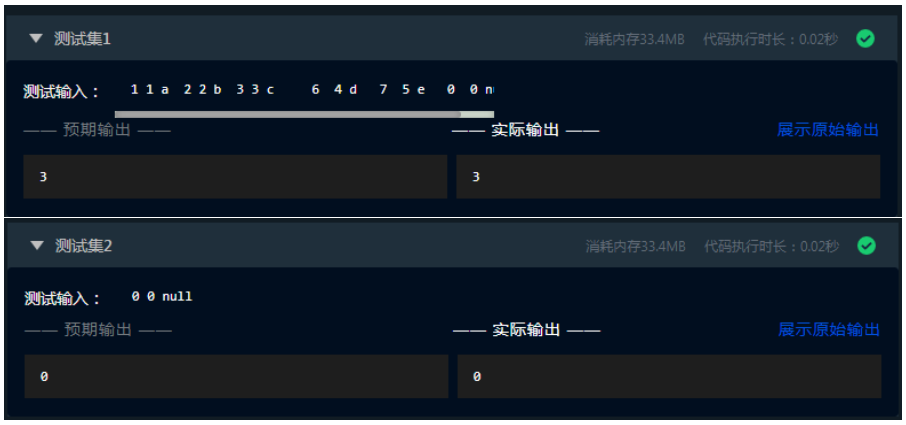


图 2-7　BiTreeDepth 函数测试

- LocateNode 测试：

  正常测试：创建 1 1 a 2 2 b 3 3 c 6 4 d 7 5 e 0 0 null 的树，获得关键字为 3 的结点，返回 3,c，获得关键字为 10 的结点，返回空指针输出查找失败；

图 2-8　LocateNode 函数测试

- Assign 测试：

  正常测试：树结点范围内正常赋值，返回 Ok 并输出修改后的树；测试数据 1 1 a 2 2 b 3 3 c 6 4 d 7 5 e 0 0 null 3 10 new

  异常测试：重复关键字赋值，测试数据 1 1 a 2 2 b 3 3 c 6 4 d 7 5 e 0 0 null 3 2 new、超出树结点范围的赋值，1 1 a 2 2 b 3 3 c 6 4 d 7 5 e 0 0 null 10 20 new；

图 2-9　Assign 函数测试

- GetSibling 测试：

  正常测试：兄弟结点存在的情况，兄弟结点为空的情况；测试数据 1 1 a 2 2 b 3 3 c 6 4 d 7 5 e 0 0 null，查询 3 的兄弟结点，1 11 a 2 12 b 5 13 c 3 20 f 6 14 d 7 15 e 0 0 null 查询 13 的兄弟结点

  异常测试：查询结点不在树的结点范围内，测试数据 1 1 a 2 2 b 3 3 c 6 4 d 7 5 e 0 0 null，查询 22 的兄弟结点;

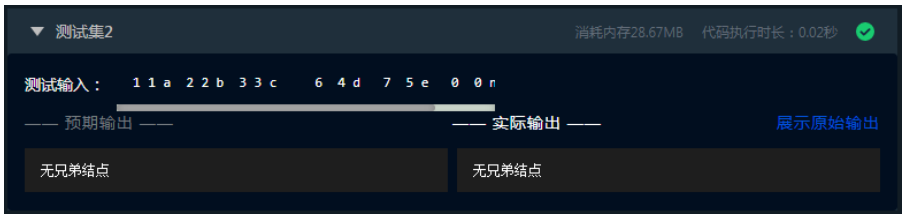


图 2-10 GetSibling 函数测试

- InsertNode 测试：

  正常测试：中间结点插入，测试数据 1 1 a 2 2 b 3 3 c 6 4 d 7 5 e 0 0 null 3 0 6 f；根节点插入，测试数据 1 1 a 2 2 b 3 3 c 6 4 d 7 5 e 0 0 null 3 -1 6 f；

  异常测试：关键字重复插入，测试数据 1 1 a 2 2 b 3 3 c 6 4 d 7 5 e 0 0 null 3 0 3 f；超出树结点范围测试，测试数据 1 1 a 2 2 b 3 3 c 6 4 d 7 5 e 0 0 null 10 0 6 f；

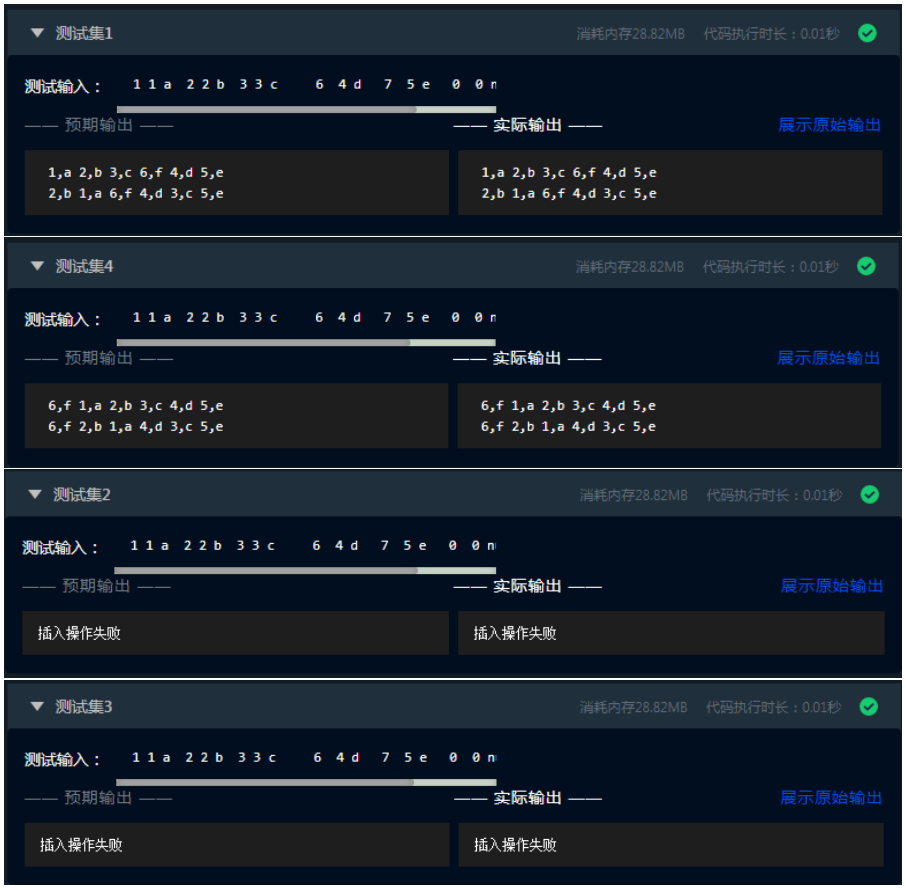


图 2-11 InsertNode 函数测试

- DeleteNode 测试：

  正常测试：依据测试数据 1 1 a 2 2 b 5 6 f 3 3 c 6 4 d 7 5 e 0 0 null 删除度为 2 的结点 a，度为 1 的结点 b，度为 0 的结点 d；

异常测试：依托正常测试的数据，删除树结点范围之外的结点；

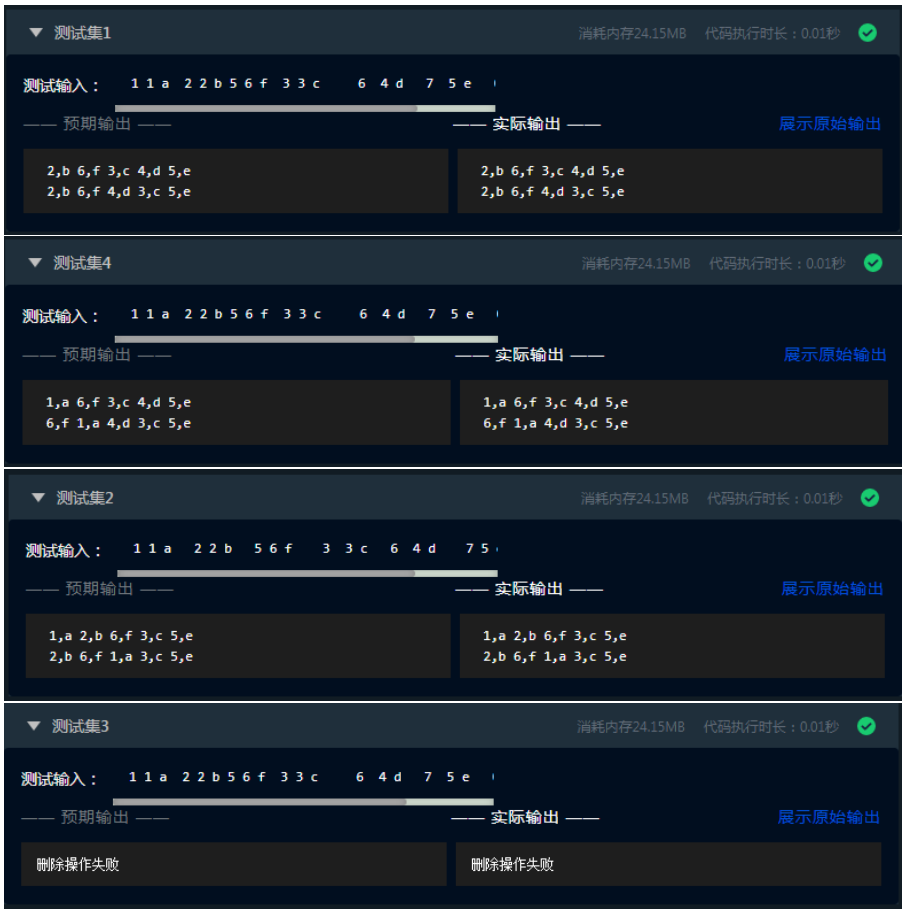
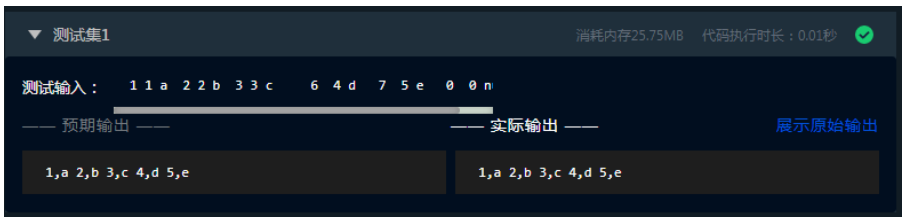


图 2-12 DeleteNode 函数测试

- PreOrderTraverse 测试：

  测试数据：1 1 a 2 2 b 3 3 c 6 4 d 7 5 e 0 0 null；

  1 1 a 2 2 b 5 3 c 10 4 d 21 5 e 0 0 null



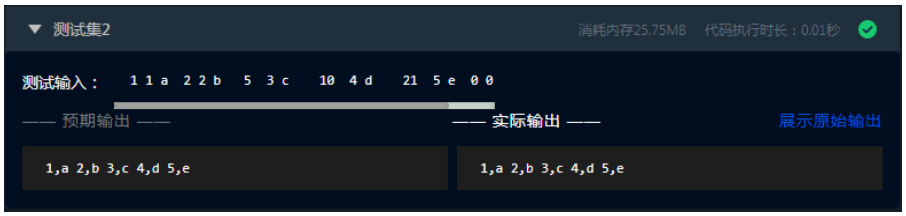


图 2-13 PreOrderTraverse 函数测试

- InOrderTraverse 测试：

  测试数据：1 1 a 2 2 b 3 3 c 6 4 d 7 5 e 0 0 null；

  1 1 a 2 2 b 5 3 c 10 4 d 21 5 e 0 0 null

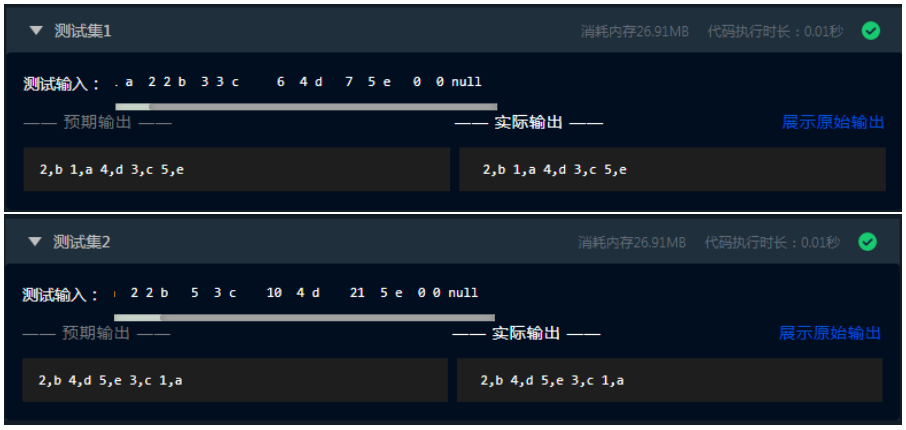


图 2-14 InOrderTraverse 函数测试

- PostOrderTraverse 测试：

  测试数据：1 1 a 2 2 b 3 3 c 6 4 d 7 5 e 0 0 null；

  1 1 a 2 2 b 5 3 c 10 4 d 21 5 e 0 0 null

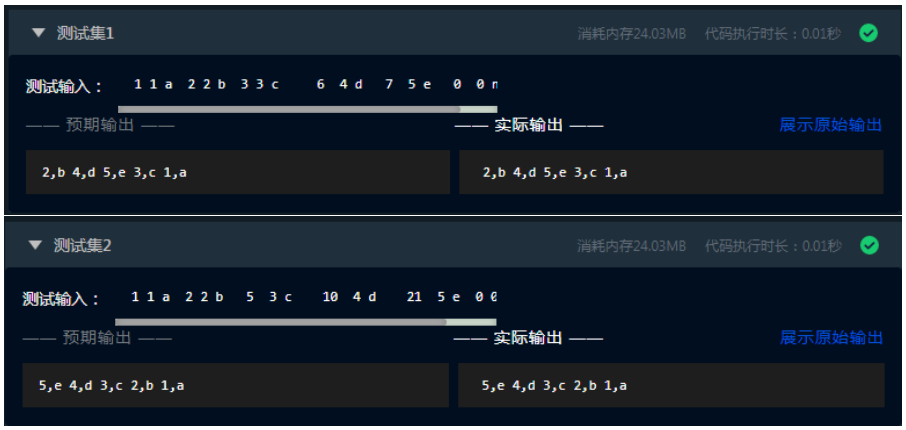


图 2-15 PostOrderTraverse 函数测试

- LevelOrderTraverse 测试：

  测试数据：1 1 a 2 2 b 3 3 c 6 4 d 7 5 e 0 0 null；

1 1 a 2 2 b 5 6 f 3 3 c 6 4 d 0 0 nul;

1 1 a 2 2 b 5 3 c 10 4 d 21 5 e 0 0 null

图 2-16　LevelOrderTraverse 函数测试

- MaxPathSum 测试：

  测试数据：1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null；

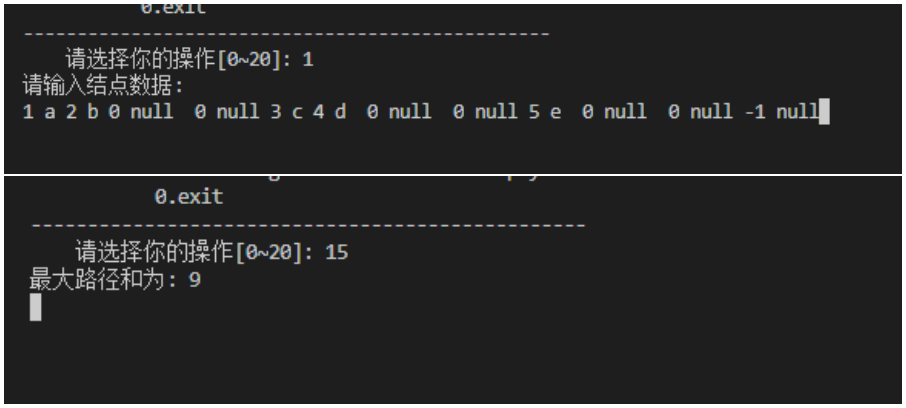


图 2-17　MaxPathSum 函数测试

- LowestCommonAncestor 测试：

  测试数据：1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null；求 b 和 d 的公共祖先，求 d 和 e 的公共祖先，求 a 和 b 的公共祖先

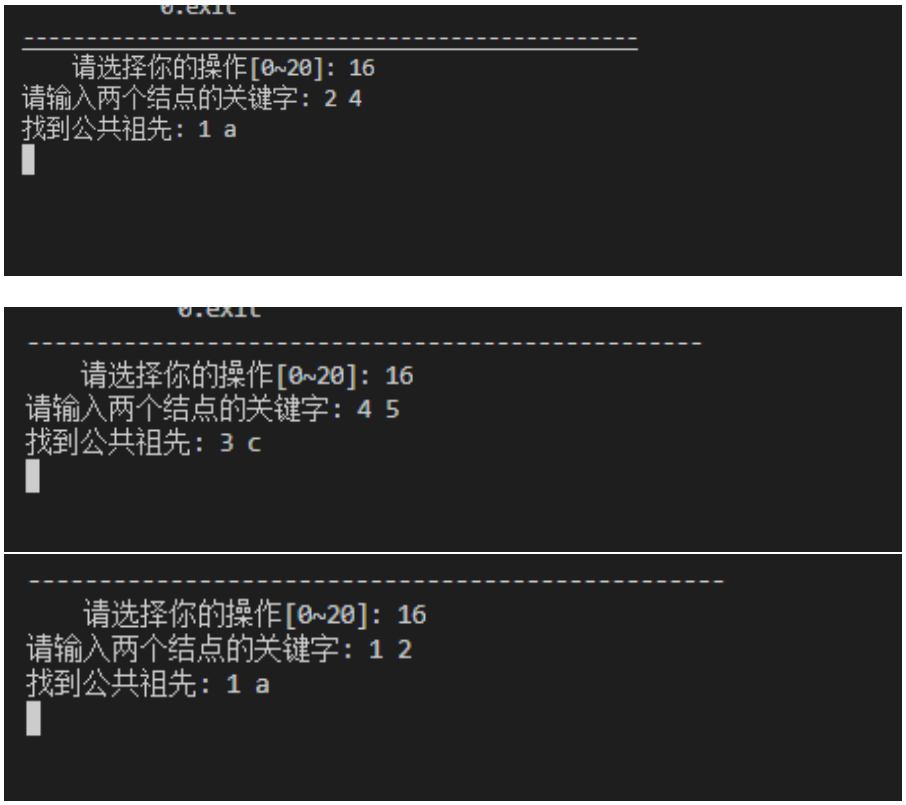
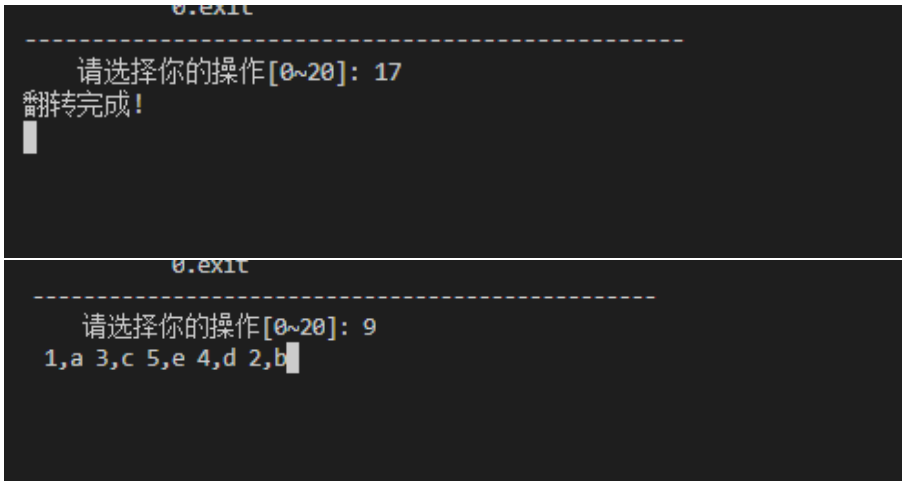


图 2-18　LowestCommonAncestor 函数测试

- InvertTree 测试：

  测试数据：1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null；分别输出前序遍历和中序遍历；



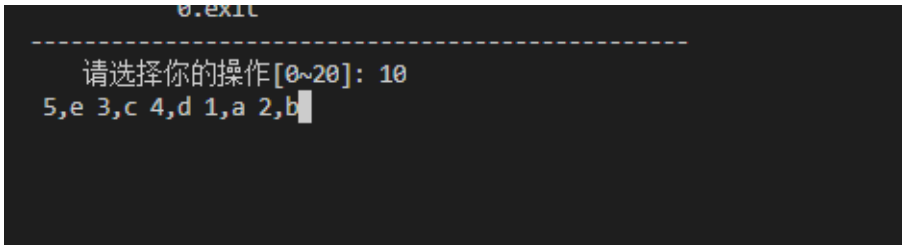
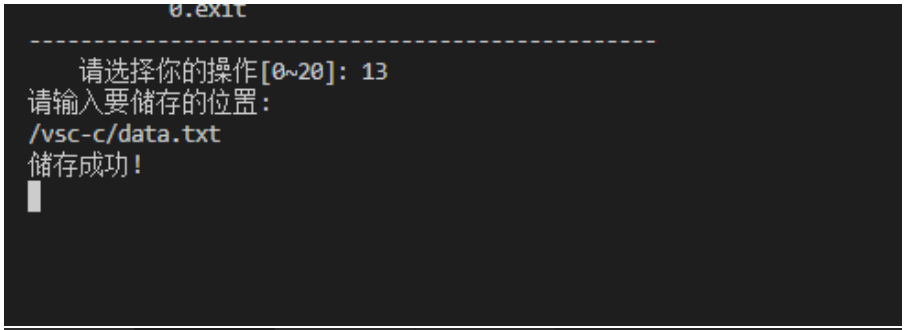
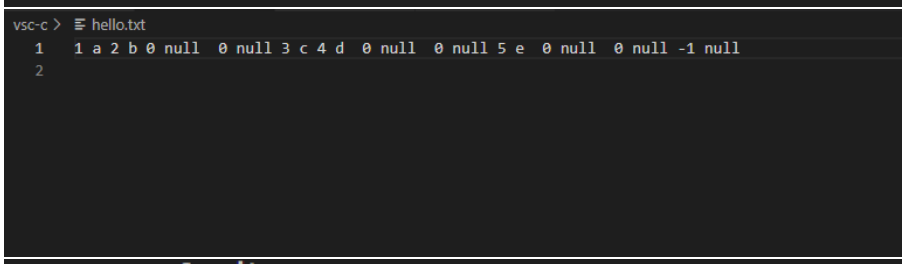


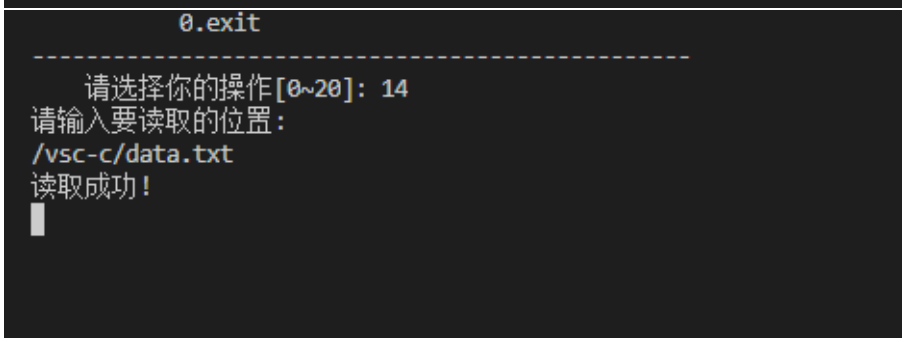
图 2-19　InvertTree 函数测试

- SaveBiTree、LoadBiTree 测试：

  测试数据：1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null；







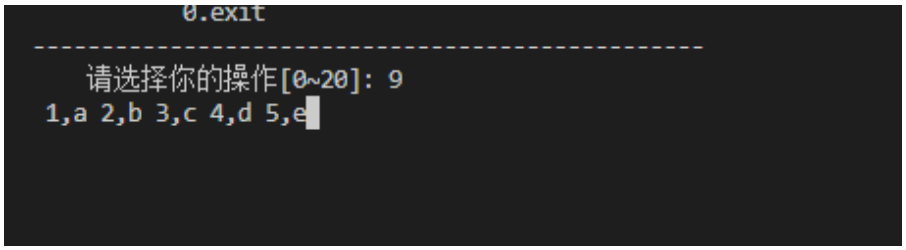
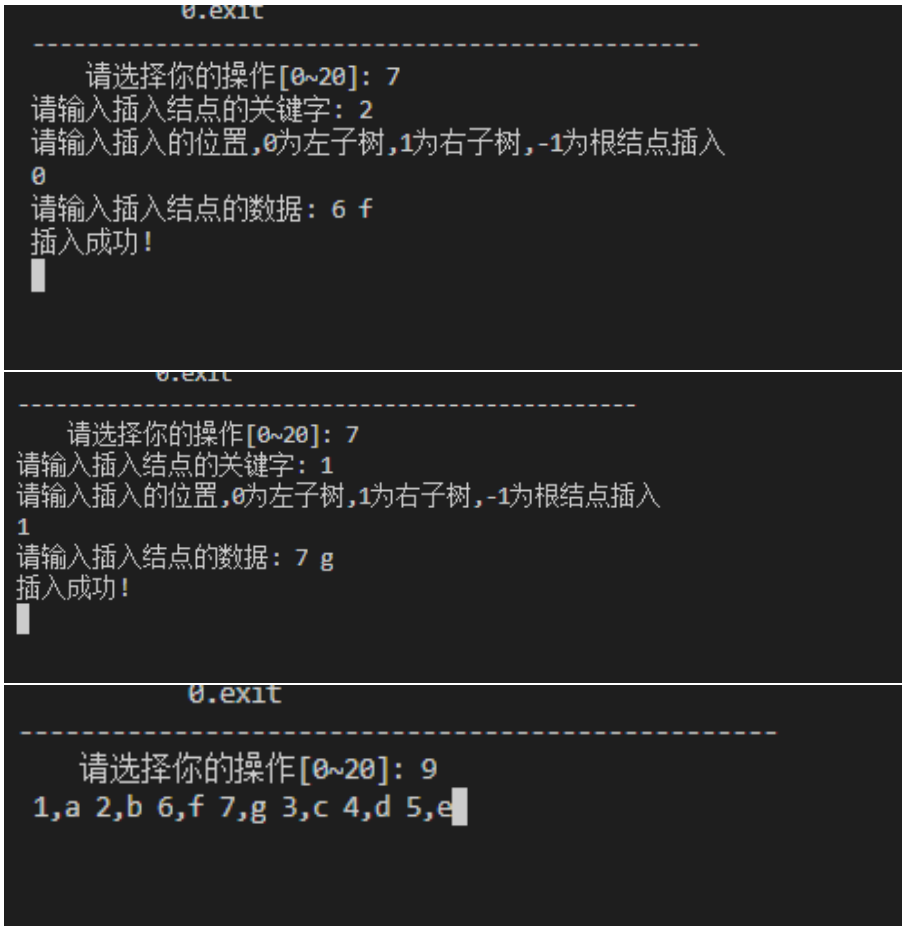


图 2-20　SaveBiTree、LoadBiTree 函数测试

- 多二叉树管理测试：

  创建两个名为 A,B，数据均为 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null 的树；A 在 b 结点的左子树插入 6 f，在 a 的右子树插入 7 g，前序，中序输出；B 删除 c 结点，前中序输出一遍；



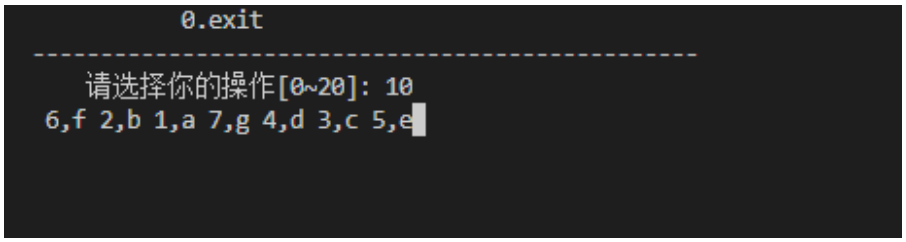


图 2-21　A 二叉树操作及输出

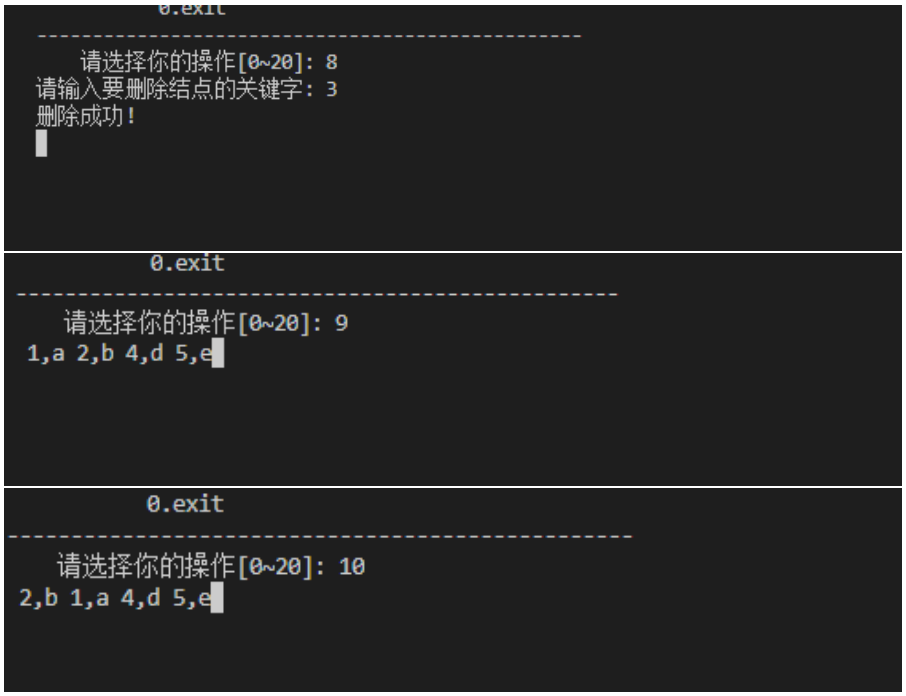


图 2-22　B 二叉树操作及输出

2.5　实验小结

在编写 CreateBitree 函数的时候想了很久，不知道到从何入手，尝试了很多方法，试过前中序确立唯一二叉树的思想，但最终还是采用有空结点下的前序构造法，核心思想其实很简单，就是充分了解前序遍历是怎么遍历的，创建的过程其实就是一个前序遍历的过程，先遍历完左子树在遍历右子树，创建的过程也遵循这样的方式，在回溯的过程中又遇到了是否该创建右子树的问题，基于此引入一个标记数组就非常好的解决了创建何个子结点的问题。

在测试 InsertNode 函数的时候，头歌平台测试错误的案例让我才猛然醒悟要保证关键词的唯一性，但在测试 Assign 的时候却没有出现类似的问题，回去看了看 Assign 函数的测试集才发现出现相同关键字的结点在目标结点之前已经遍历过了，因此标记数组中把该关键值修改为了 1，但是在 InsertNode 函数的测试

集中，出现相同关键字的结点在目标结点之后，即还没有遍历到该结点，而我的程序在找到目标结点执行完操作后就直接返回了，因而没有发现重复关键值，因此，修改了程序，让关键字的修改发生在全部遍历之后，这是可以确保已有的关键字均已被标记数组记录，这样可以解决关键字重复的问题。

在 DeleteNode 函数的测试过程中，发现头结点的删除不能有效进行，进行单步调试跟踪后发现，T 的值并没有发生改变，由此把头结点的删除单独拎出来进行操作，基于此我也发现了一个我们常忽略的一个问题，就是指针的指向问题，假设我们让 t = L->next; 随后我们对 t 进行了一系列操作，比如改变 t 的指向，但是当我们再调用 L->next 时，我们会发现 L->next 的指向并没有随 t 的改变而改变，这一点在删除结点这个函数的测试过程中尤为凸显，当我删除完这个结点后，并有子节点替换了删除结点，但我调用父节点去访问子结点是我们会发现，我们访问不到修改过的子节点，这就在于我们没有改变 lchild 或者 rchild 的指向，它们的指向依然是原来的指向。因此，我们以后在进行有前后关系的指针操作时，要切记一个结点发生变动，也要记得修改前一结点对下一结点的指向关系，避免造成无效的修改。

在最初尝试 TreeChange 函数时，发现主函数中可以切换不同树，但是对树进行操作完，切换到下一个树后又切回原来的树，会发现数据并没有保存到，这里的原因和上面说的一样，当我们在主函数中让 T = TreeChange(Lists,listname,record); 时，对应的 L.elem[aim].T 的指向并没有随 T 的改变而改变，依然指向 NULL，所以当 T 等于其他表又切换回来时就会发现数据没有得到存储，因此我们仍需在 CreateBitree 函数执行后增添 L.elem[record].T = T; 的语句让指针改变指向。

3 课程的收获和建议

一学期的课程过去，通过对数据结构知识的系统学习，了解如下这些数据结构，并有了一些自己的认识与体悟。

3.1 基于线性存储结构的线性表实现

通过对该章的学习，我学会了如何使用线性表以及多个线性表的管理及使用，同时明白表的清空与销毁是需要对所有元素进行的，以及对指针的必要初始化赋值，如 NULL；

同时，从第一次的实验中也了解到数据结构类型基本的函数操作功能，如增添修改删除等，了解如何创建并维护一个数据结构，也注意到一些需要注意的边缘问题，如：对于特定位置的函数操作，如对表首、表尾，以及空表的函数操作。明白了多线性表操作结构体的基本构造方式，对于代码的规范性有了进一步的理解，例如需要设置最大容量，常量一般使用 define，多加运用 typedef 等等增强代码的可读性等等

3.2 基于链式存储结构的线性表实现

链表的理论知识在上学期的 C 语言课上已有所涉及，再通过本学期的课程学习，我对链表有了更深的理解。

在链表的种类上，知道了除了单向链表之外的许多链表类型，例如双向链表，循环链表，十字交叉链表等等，基于不同类型链表可以有不同的更加灵活的数据存储及运用。

链表相交于以数组为基础的线性表在操作上更加灵活，例如元素的增添与删减，线性表需要对增添删减位置往后的元素均需要进行移位变动，而链表的插入与删除就显得非常的快捷高效，同时链表存储的数据类型更加的多样，依托结构体存在的链表，可以在结构体中定义不同类型的数据类型，并依据需求选择变量进行存储，但链表的难点在于不好把控结点与结点之间的关联，在进行换位操作时容易出现逻辑混乱，不易于理解，例如链表的冒泡排序，有指针域的交换和数据域的交换，指针域的交换的逻辑要比数组的直接交换复杂许多，这更加要求具有良好的逻辑思维能力，我们也可以在链表的日常使用中增强自己的逻辑思

维能力。

3.3 基于二叉链表的二叉树实现

二叉树是本学期新接触学习的数据结构类型，我对二叉树的理解仍然较浅，但也能感受到二叉树对于我们解决问题提供的强大支撑。

基于二叉树产生了平衡二叉树，二叉搜索树，二叉排序树，完全二叉树，哈夫曼编码等等，他们能将结点与结点之间的关系转化为数学上的数字关系，例如在编写查找公共祖先的函数问题时，就运用了完全二叉树的数学特性，将结点的物理联系转化为数学的数字关系，简化了问题，也更加直观地将问题展现并加以解决。

二叉树的四种遍历方式也使我对递归函数有了进一步的了解，非递归的遍历算法又使我加深了对栈和队列的了解以及实际应用，同时，不同的遍历方式在处理不同的问题上也有独特的优势，并基于这些遍历方式，认识和了解波兰式和逆波兰式等的使用。

3.4 基于邻接表的图实现

图也是新接触的一个数据类型，是一种比链表和树更加灵活更加复杂的一种数据类型，体现的是不同结点之间的复杂联系，并依据这些联系找出最优的路径解等等。

图的应用场景非常广泛，比如互联网中的人际关系，不同终端之间的关联，一个工程的分支进度，一个学校学生的课表安排，这些都离不开图，也离不开基于图之上的一系列图操作。

图采用邻接表的形式存储，邻接表的形式也多种多样，以此应对不同的应用场景，本次实验也是采用邻接表的方式存储顶点集与边集，并实现顶点与边增添删减的一系列基础操作，加深了我对简单图的理解，也了解了深度优先搜索和广度优先搜索这两种搜索方式，学会如何计算最短路径，如何计算连通度等等问题。