華中科技大学

课 程 实 验 报 告

课程名称： 数据结构实验

专业班级： 计算机科学与技术 1607

学　　号： U201613570

姓　　名： 洪志远

指导教师：

报告日期： 2018 年 11 月 31 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 问题描述

线性表是由n（n≥0）个数据元素（结点）a[0]，a[1]，a[2],…，a[n-1]组成的有限序列。本实验的目的是封装一个基于顺序存储结构的线性表ADT，提供线性表基本的、常用的12中运算和操作。要求中还有关于文件I/O的细节需要实现，使得线性表可以实现内外存交换，便于数据读写。同时，需要实现一个演示系统实现简单的演示，以此作为可用性检查的工具。

1.2 系统设计

1.2.1 总体系统

该线性表的实现包括一个.CPP文件。文件中定义了线性表的结构体、相关运算的函数定义及其实现、测试系统、多表管理、文件操作等函数。由于整个线性表比较简单，而且没有“#include”的需要，因此没有使用头文件。

1.2.2 数据结构

数据结构的实现根据实验要求的ADT定义，定义了基于C语言结构体的ADT定义。由于实验要求给出的代码的Codebase是经典的C语言面向过程的写法，没有使用C++的类。结构体如下：

```
struct SqList {
  ElemType* elem;
  size_t length;
  size_t listSize;
};
```

1.2.3 ADT操作的设计

根据实验的要求以及线性表ADT的定义，该线性表ADT应包括如下的操作：

1. InitList（初始化）
2. DestroyList（销毁）

3. ClearList（清空）

4. ListEmpty（判断表是否为空）

5. ListLength（求表长）

6. GetElem（按下标取得元素）

7. LocateElem（按满足关系取得元素）

8. PriorElem（返回满足关系的元素的前驱）

9. NextElem（返回满足关系元素的后继）

10.ListInsert（插入元素）

11.ListDelete（删除元素）

12.ListTraverse（遍历表）

为了便于错误处理，定义以下常量作为错误标识：

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

为了简化编码和操作，定义以下常量：

```
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
```

为了方便更改线性表中元素的类型，使用以下的宏定义：

```
typedef int status;
typedef int ElemType;
```

其中，status 是某个方法的返回类型，用于标志是否正确执行了相应的运算。Status 的取值由上面的宏定义定义。当不为 1 时，认为发生了错误。然后可以

根据 status 的值确定发生了何种错误。

ADT 的运算有下面的函数定义，这些函数均返回 status（如果不返回其他有用的值）、bool（判断是否为空）和 size_t（求出长度或者位置）：

```
status InitList(SqList&);
status DestroyList(SqList&);
status ClearList(SqList&);
bool ListEmpty(SqList);
size_t ListLength(SqList);
status GetElem(SqList, size_t i, ElemType& e);
size_t LocateElem(SqList, ElemType e);
status PriorElem(SqList, ElemType cur, ElemType& pre_e);
status NextElem(SqList, ElemType cur, ElemType& next_e);
status ListInsert(SqList&, size_t i, ElemType e);
status ListDelete(SqList&, size_t i, ElemType& e);
status ListTraverse(SqList);
```

上面的 ListTraverse 经过了简化，使得其遍历行为变为固定的打印。这一点使线性表的遍历失去了灵活性。这是一个可以优化和改善的点。

1.2.4 多表系统的设计

使用 C++ STL 的 Vector 作为多表的容器。使用容器可以简化相关的操作，同时具有很高的健壮性，也可以减少问题。

具体的思路为：在程序开始的时候创建一个 vector，将第一个默认的 SqList 添加到里面，同时设置一个索引变量，用于指示线性表在 vector 中的位置。在运行的时候，每当用户选择新建一个线性表的时候，就将一个线性表添加到里面。同时将当前的线性表索引指向新的线性表。

容器的定义如：vector<SqList> Lists = {SqList()};

1.2.5 文件存储系统设计

文件操作可以使线性表的元素存放到硬盘上永久保存，更加贴近真实的使用

场景。通过 C 语言标准库提供的文件读写 API，可以很方便的操作文件。

通过 fopen 可以返回一个指针，该指针可以用来操作先前打开的文件。同时可以制定打开文件的方式，例如“r”可以用来打开一个只读文件。然后使用此文件进行读写操作，最后使用 fclose 销毁该指针。在写文件的时候，可以将整个 elems 数组以 2 进制的方式存入文件。在读取的时候，连续读取每次读取 sizeof(ElemType)个字节，然后将这些字节拷贝到 elems 里面，直到读取到文件尾为止。同时，每次读取使线性表的长度增加 1 个长度。

1.3 系统实现

1.3.1 开发环境

开发环境选用 Windows 10 上的 VSCode 作为编辑器，g++作为编译器，g++的版本为 8.1.0。

1.3.2 ADT 操作的实现

status InitList(SqList&);

该操作接受一个 SqList 的引用，将其作为初始化对象进行初始化。首先，先要检查该线性表是否存在，如果存在，则将其长度设为 0，最大长度设为 LIST_INIT_SIZE，将 elems 指向一块长度为 LIST_INIT_SIZE 的内存，从而完成初始化。如果 elems 已经指向了一块内存，则将其 delete。如果成功，则返回 status 中的 OK，否则返回相应的错误类型。

该操作的时间复杂度 O(1)，空间复杂度 O(1)。

status DestroyList(SqList&);

该操作用于销毁一个线性表。该函数接受一个 SqList 的引用，将其 elems 所指向的内存 delete，并将该指针置为 0；同时，将该线性表设置为已销毁。如果成功，则返回 status 中的 OK，否则返回相应的错误类型。

该操作的时间复杂度为 O(1),空间复杂度为 O(1)。

status ClearList(SqList&);

该操作用于将一个线性表 SqList 清空。该函数接受一个 SqList 的引用，如果 SqList 没有被初始化，则直接返回。否则将 size 设为 0。如果成功，则返回 status 中的 OK，否则返回相应的错误类型。

该操作的时间复杂度是 O(1)，空间复杂度 O(1)。

```
bool ListEmpty(SqList);
```

该操作用于判断一个线性表是否为空，返回值为 bool 类型，当为真时说明线性表为空，否则说明线性表存在元素。如果某线性表未经初始化，那么他一定为空。

该操作的时间复杂度是 O(1)，空间复杂度 O(1)。

```
size_t ListLength(SqList);
```

该操作用于求取一个线性表的长度（实际包含元素的长度）。对于一个已经初始化的线性表，直接返回 size 的值。其他情况下则返回 -1（会被隐式类型转换为 MAX_LONG_LONG_INT）。

该操作的时间复杂度是 O(1)，空间复杂度 O(1)。

```
status GetElem(SqList, size_t i, ElemType& e);
```

该操作用于获取线性表中的某一个元素。接受 3 个参数，第一个为线性表的拷贝，第二个为要获取的线性表所在的位置，第三个为变量的引用，获取成功的元素的值将被存放到这个变量中。如果该操作正常执行，则返回 status 中的 OK，否则返回相应的错误。

该操作的时间复杂度是 O(1)，空间复杂度 O(1)。

```
size_t LocateElem(SqList, ElemType e);
```

该函数接受两个参数，一个是线性表的复制。一个是要定位的元素的值。该操作用于求取给定的 e 在线性表中的位置，可以实现为返回从左到右第一次出现的位置，位置从零开始，-1 表示未找到。该函数的实现为遍历该线性表，当第一次遇到和 e 相等的元素时，则返回当前的循环变量。

该操作的时间复杂度是O(n)，空间复杂度O(1)。

status PriorElem(SqList, ElemType cur, ElemType& pre_e);

该操作接收三个参数：一个SqList的拷贝、一个ElemType的变量cur、一个ElemType的引用pre_e，找到和cur变量相等的元素的前驱（如果存在），并且将pre_e赋值为它，否则操作失败，pre_e无定义。

该操作首先判断表是否存在，如果存在则依次遍历整个线性表，判断线性表中的每个元素是否与传入的cur_e变量相等，如果满足则直接结束遍历，并且将pre_e赋值为该元素的前驱。如果成功，则返回status中的OK，否则返回相应的错误类型。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

status NextElem(SqList, ElemType cur, ElemType& next_e);

该操作接收三个参数：一个SqList的拷贝、一个ElemType的变量cur、一个ElemType的引用next_e，找到和cur变量相等的元素的前驱（如果存在），并且将next_e赋值为它，否则操作失败，next_e无定义。

该操作首先判断表是否存在，如果存在则依次遍历整个线性表，判断线性表中的每个元素是否与传入的cur_e变量相等，如果满足则直接结束遍历，并且将pre_e赋值为该元素的前驱。如果成功，则返回status中的OK，否则返回相应的错误类型。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

status ListInsert(SqList&, size_t i, ElemType e);

该操作接收三个参数：一个SqList的拷贝、一个size_t的变量i、一个ElemType的变量e。用于将e元素插入到位于i位置的元素之前。

该操作首先判断表是否存在以及表示下标的i是否越界，如果表存在且未越界，则进一步判断线性表是否已满，如果已满，则重新分配空间，确认分配成功后修正线性表的大小。然后从线性表的最后一个元素开始到给定下标的元素为止，将所有的元素向后移动一位。最后将e存入已经空出来的线性表特定下标所对应

的位置并更新线性表长度。

该操作的时间复杂度是 O(n)，空间复杂度是 O(1)。

status ListDelete(SqList&, size_t i, ElemType& e);

该操作接收三个参数：一个 SqList 的拷贝、一个 size_t 的变量 i、一个 ElemType 的引用 e。用于将 i 位置的元素删除

从线性表的最后一个元素开始到给定下标的元素为止，将所有的元素向前移动一位。最后将该元素存入 e 并更新线性表长度。

该操作的时间复杂度是 O(n)，空间复杂度是 O(1)。

status ListTraverse(SqList);

该操作接收一个 SqList 拷贝、依次对线性表中的每个元素执行打印操作。

该操作首先检查表是否存在、是否为空，存在且非空的话，则循环遍历整个线性表，并且对每个元素执行打印操作。

该操作的时间复杂度是 O(n)，空间复杂度是 O(1)。

1.3.3 多表系统的实现

多表系统实现了添加一张表、更改当前表两个操作（没有实现删除表）。

添加一张表：

添加一张表就等价于在 vector<SqList>中添加了一个 SqList,该 SqList 的所有字段都是按照默认初始化进行初始化的。因此，该线性表虽然有了内存空间但是却是一个空表，其 size 以及 elems 等关键字段都是无效的值。

更改当前表：

更改当前表就等价于修改了指向 vector<SqList>的下标。该下标的取值为 0 至 vector<SqList>的长度。修改下标即可实现修改当前表所指向的值。从而达到修改当前表的目的。

容易可得，这两个操作的时间复杂度都为 O(1)。

1.3.4 文件存储系统实现

通过 fopen 可以返回一个指针，该指针可以用来操作先前打开的文件。同时

可以制定打开文件的方式，例如“r”可以用来打开一个只读文件。然后使用此文件进行读写操作，最后使用 fclose 销毁该指针。在写文件的时候，可以将整个 elems 数组以 2 进制的方式存入文件。在读取的时候，连续读取每次读取 sizeof(ElemType)个字节，然后将这些字节拷贝到 elems 里面，直到读取到文件尾为止。同时，每次读取使线性表的长度增加 1 个长度。

1.4 系统测试

1.4.1 演示系统测试

演示系统测试如下图所示：

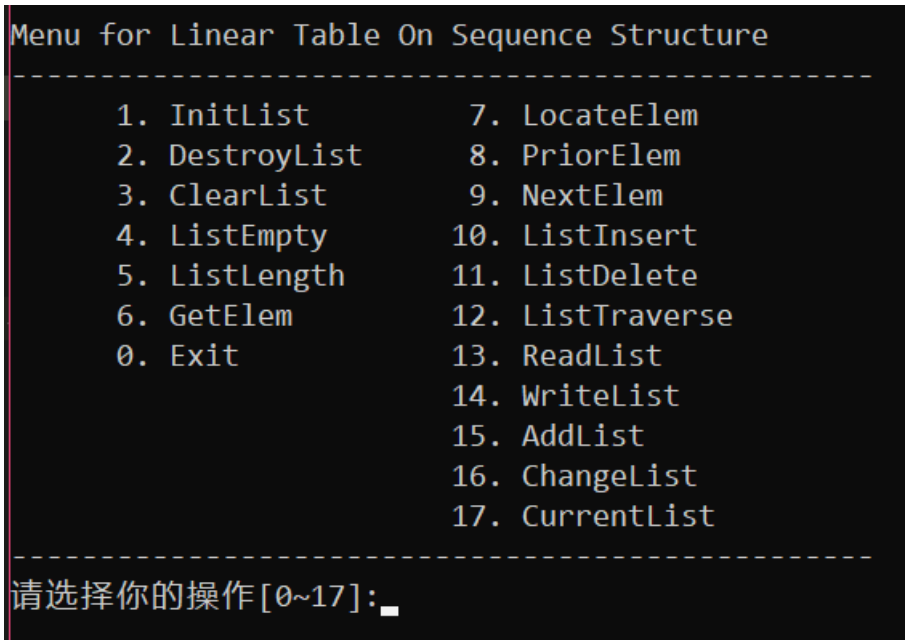


图 1-1 演示系统

该演示系统操作简单，输入对应的数字并按下回车即可。输入 1-17 表明运行对应的功能，且相应操作的结果会被输出到屏幕上，输入 0 退出系统。

1.4.2 测试样例

使用有效的典型的测试样例可以有效的寻找系统的错误，同时可以提高测试的覆盖率,使测试更加全面.正确的测试样例可以有效地测试系统是否正确,而错误的测试样例则可以用来测试系统的健壮性.按照如下的操作流程即可完整的测试该系统:

1. 初始化一个线性表。
2. 测试线性表的长度。
3. 依次将 1-5 插入线性表下标为 0 的位置。

4. 测试线性表的长度。
5. 遍历线性表。
6. 测试线性表的长度。
7. 得到下标为 3 的元素。
8. 得到下标为 1000 的元素
9. 获取 3 元素的下标。
10. 获取 1000 元素的下标,错误样例
11. 获取 3 元素的前驱
12. 获取 1 元素的前驱,错误样例
13. 获取 3 元素的后继
14. 获取 5 元素的后继,错误样例
15. 删除下标为 1 和 1000 的元素。
16. 遍历线性表。
17. 将线性表存入文件。
18. 添加一个线性表。
19. 切换到新添加的线性表 1，并初始化。
20. 从刚存储的文件中加载线性表。
21. 遍历线性表。
22. 删除下标为 1 的元素。
23. 遍历线性表。
24. 切换到旧的线性表 0。
25. 遍历线性表。

1.4.2 测试结果

初始化一个线性表：

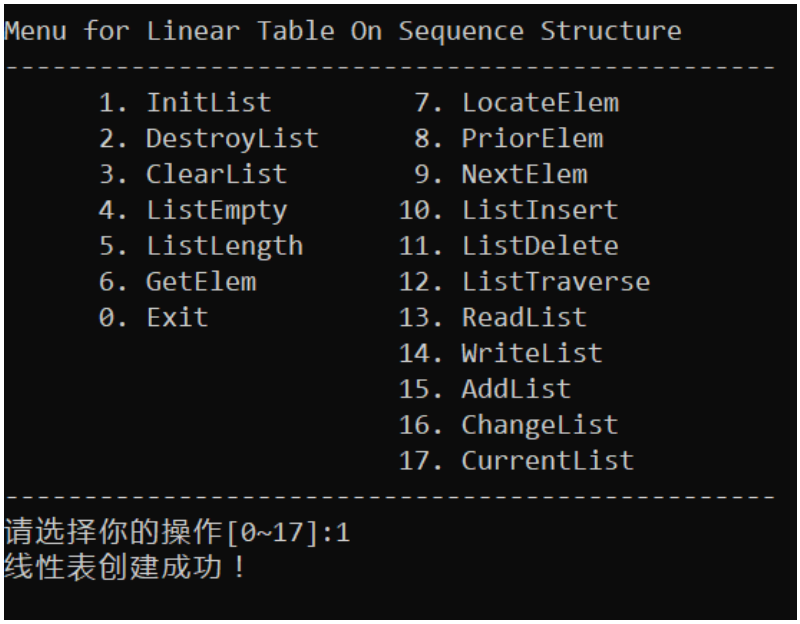


图 1-2 初始化成功

由图可见，初始化线性表成功。

测试线性表的长度：

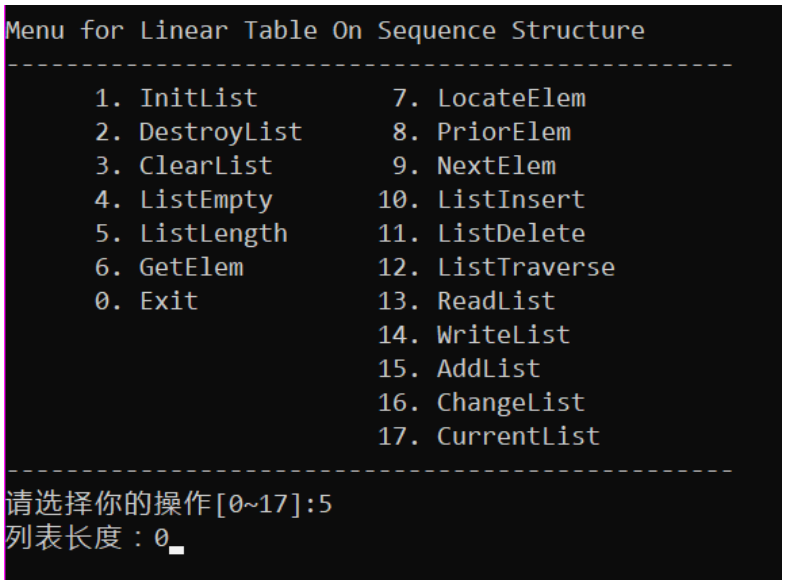


图 1-3 长度测试正确

线性表的长度为 0，符合预期结果。

然后插入 1~5 的元素，如下图所示，依次在 0 位置插入 5、4、3、2、1，即可实现插入 1~5 元素。

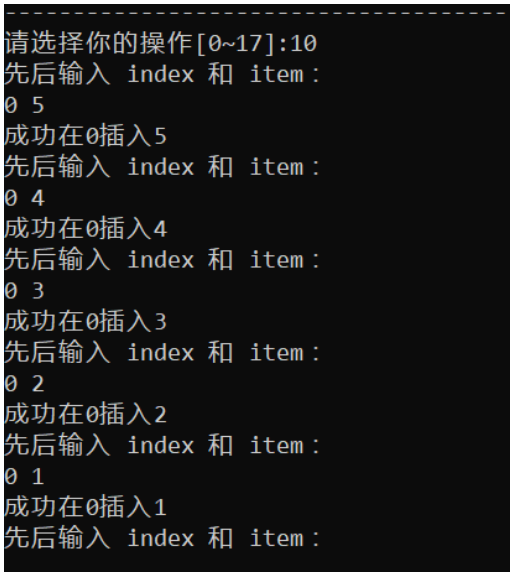


图 1-4 插入元素

为了验证我们确实插入了 1~5 的元素，我们使用 ListTraverse 来验证：如果 ListTraverse 输出 1 2 3 4 5，则说明该操作正确。完成操作之后的截图如下，可见，该操作是正确的。

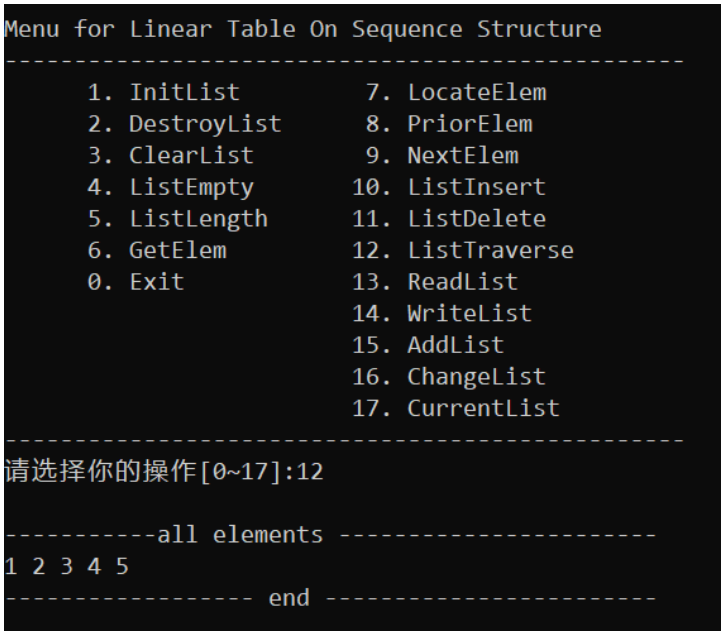


图 1-5 遍历线性表

此时，测试改线性表的长度，即使用 ListLength 操作，得到的输出如下，列表长度为 5，该结果正确且符合预期。由此可得 ListInsert、ListTraverse、ListLength 可以正常执行操作。

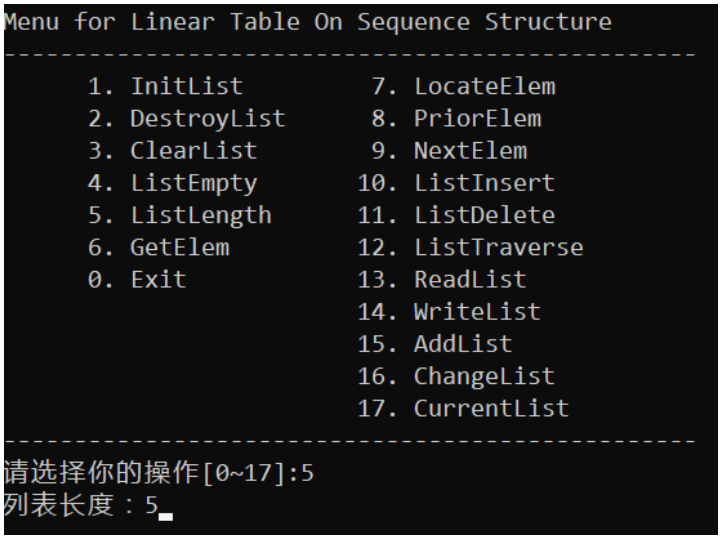


图 1-6 获取线性表长度

接下来获取下标为 3 的元素，使用 GetElem 来实现该操作，输入了 index（下标）3 之后，按下回车即可获取改下标对应的值。

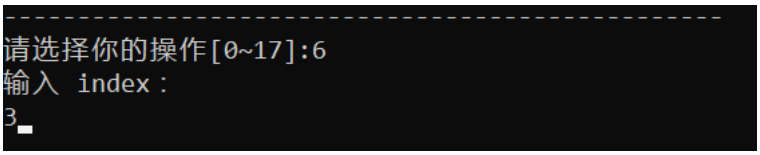


图 1-7 获取相应下标的元素

如图所示，使用 GetElem 获得下标为 3 的元素，输出的值为 4，因为我们在线性表中存储了 1、2、3、4、5，所以下标为三的元素为 4，结果正确，符合预期。如下图所示：

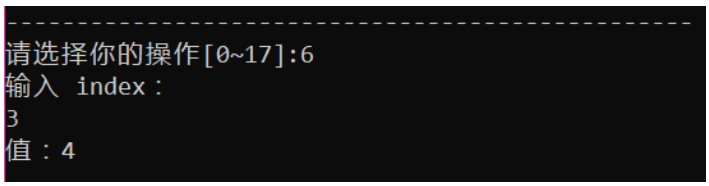


图 1-8 获取相应下标的元素

为了测试该系统的鲁棒性，测试获取下标为 1000 的元素时，该操作的行为。由于线性表中只有 5 个元素，因此获取下标为 1000 的元素直接报错。通过该测试可知该操作具有很好的容错机制。

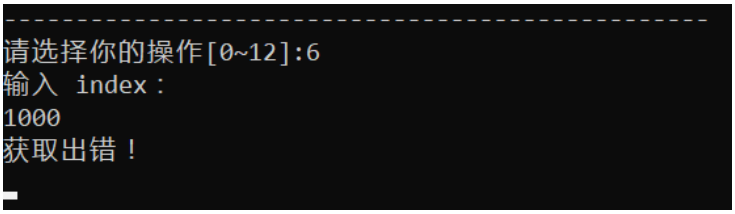


图 1-9 获取相应下标的元素（错误案例）

为了测试 3 元素的下标，我们使用 LocateElem 操作，对 3 元素进行定位，根

据前面的描述，该操作接受某元素的拷贝为参数，返回从左到右第一次出现该元素的下标，该下标从 0 开始。测试过程如下：

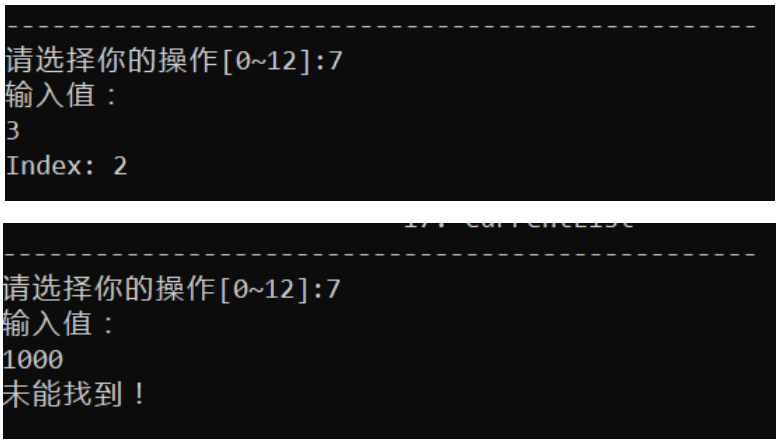


图 1-10 定位元素的位置

如上图所示，先后测试元素 3 和 1000 为参数的 LocateElem，由于 3 的下标是 2，因此程序输出 2，结果正确且符合预期。1000 不在线性表中，因此该值的下标输出为“未能找到!”，以此提醒使用者该操作无效。由截图可见，LocateElem 的行为完全正确。

为了测试 PriorElem 的正确性，使用 1 和 3 为参数进行测试，由于元素 1 实际上没有前驱，因此该操作应该输出错误信息。而 3 的前驱为 2，我们应该期望对于 3 进行获取前驱操作应当输出 2。

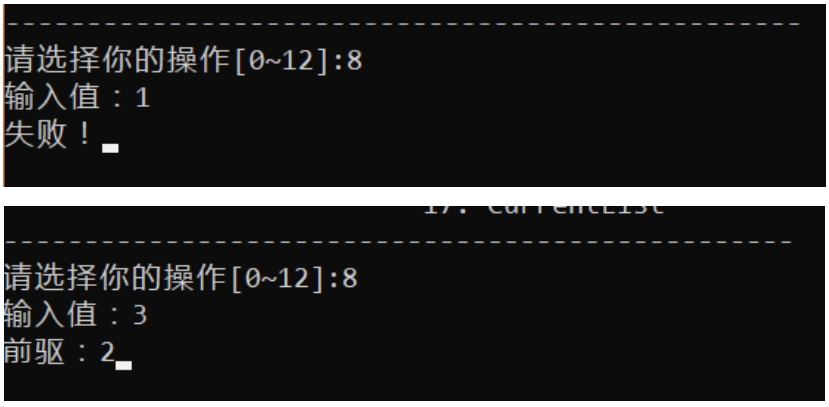


图 1-11 获取元素的前驱

如上图所示，该操作输出了正确的信息，因此可以认为 PriorElem 的行为是正确的。

为了测试 NextElem 的正确性，使用 3 和 5 为参数进行测试，由于元素 5 实际上没有后继，因此该操作应该输出错误信息。而 3 的后继为 4，我们应该期望对于 3 进行获取前驱操作应当输出 4。

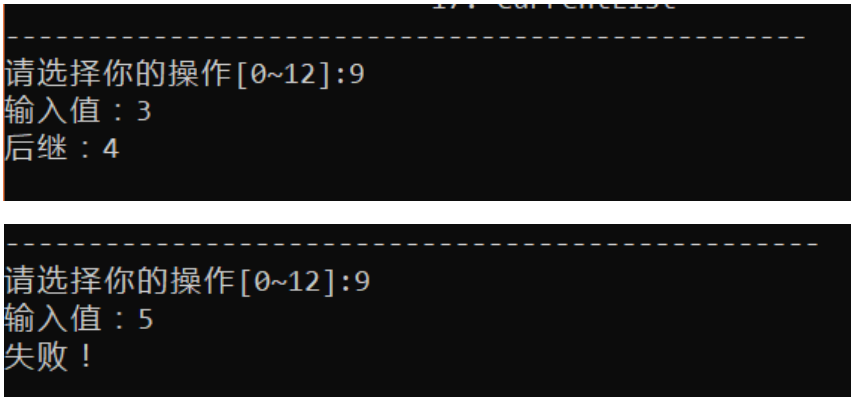


图 1-12 获取元素的后继

如上图所示，该操作输出了正确的信息，因此可以认为 NextElem 的行为是正确的。

为了测试删除元素操作 ListDelete 的正确性，使用 1 和 1000 作为参数进行测试，将 1 和 1000 作为参数传递给 ListDelete，由于下标为 1 的元素为 2，因此执行完 ListDelete(1)之后，线性表中的元素应为 1、3、4、5；由于 1000 下标的元素不存在与线性表中，因此该操作将输出错误。

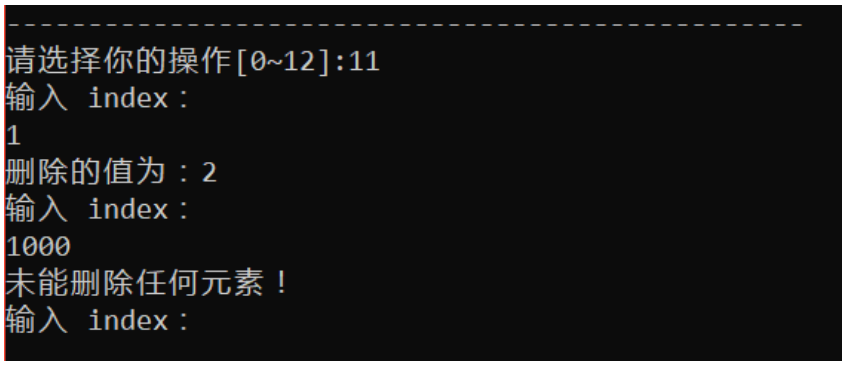


图 1-13 删除元素

如上图可知，该操作的输出符合预期，因此可以认为该操作对应的程序无错误。为了进一步确认 ListDelete 对线性表的修改，使用 ListTraverse 来进行进一步测试，通过判断 ListTraverse 的输出值，查看执行 ListDelete 之后线性表的内容。

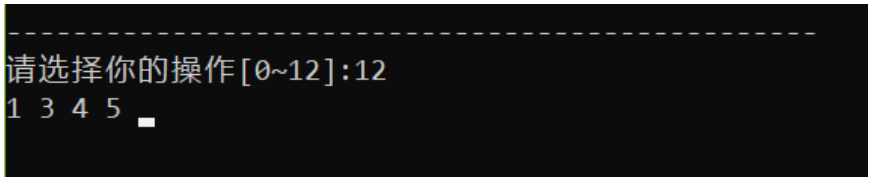


图 1-14 删除后遍历

为了测试文件操作的可行性，使用 WriteList 操作。输入一个随意的文件名，例如 2 进行测试，文件将会被打开然后，程序将会将数据写入到打开的文件中，如果文件打开失败，会输出“File open error”。

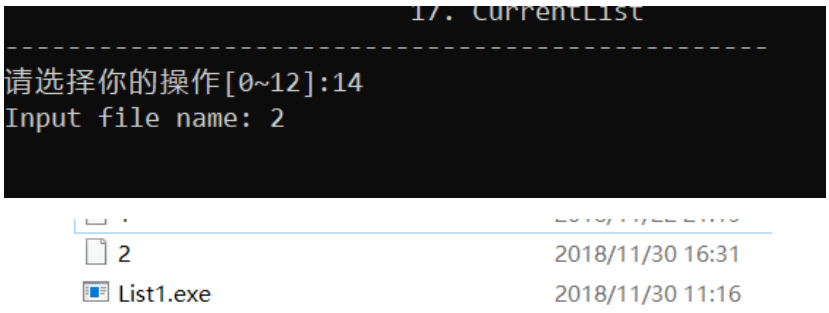


图 1-15 写出到文件

如图所示，文件成功创建。为了验证文件内容的正确性，新建一个 SqList 并将该文件导入，即可检测该功能的正确性。依次进行 AddList、ChangeList、InitList 等操作，即可创建、切换并初始化一个新的线性表。然后运行 ListTraverse 操作，即可得到改线性表的所有元素。可见，这些元素和之前的线性表完全一致。

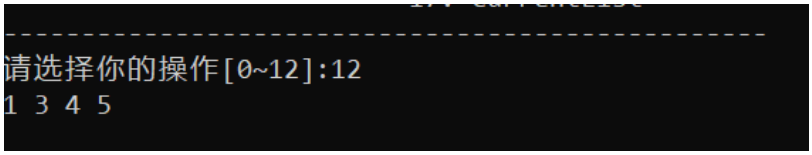


图 1-16 读入文件

为了证明该线性表和之前的线性表不是同一个线性表，修改新的线性表并删除一个元素，为了方便演示，删除下标为 1 的元素。可见，下标为 1 的元素被成功删除，由于删除之前该线性表存储的元素是 1、3、4、5，删除下标为 1 的元素导致线性表变为 1、4、5。

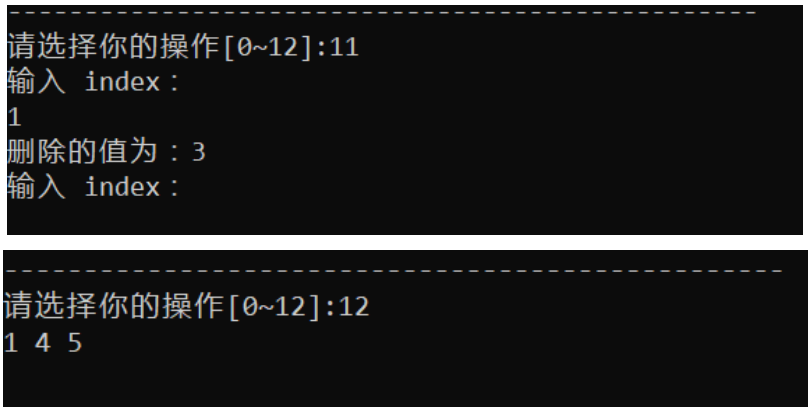


图 1-17 删除元素（新线性表）

由上图可知，遍历的结果和预期吻合，所以改线性表的文件操作无错误。而我们的目的是验证新的线性表不会影响旧的线性表，因此我们切换到之前的线性

表，并执行遍历操作。

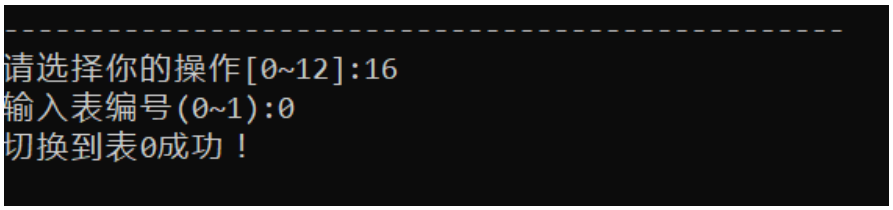


图 1-18 切换到原表

可见，当前的线性表是线性表 0，也就是之前的旧的线性表。在该线性表上执行 ListTraverse 操作，即可获取所有的元素。由下图可见，所有的元素均列出，同时，没有出现和新线性表保持同步的现象，这说明在多表的情况下，线性表操作是独立的，不会互相影响。

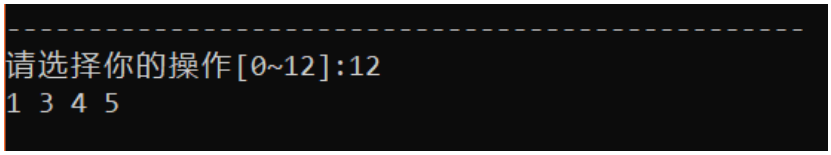


图 1-19 遍历原表

1.5 实验小结

通过本次实验，我加深了对于基于顺序结构的线性表的认识和了解。

在这次实验中，我使用 C 语言的风格封装了一个可以使用的线性表。这次实验我知道了如何实现一个线性表，将课本的知识转化为可以使用的程序，同时我对 C 语言的错误处理、面向过程的编程有了更深的理解。通过对测试系统的实现，我学会了如何全面的测试我的程序，也知道了如何提高系统的鲁棒性。

2 基于链式存储结构的线性表实现

2.1 问题描述

线性表是由 n（n≥0）个数据元素（结点）a[0]，a[1]，a[2],…，a[n-1]组成的有限序列。本实验的目的是封装一个基于顺序存储结构的线性表 ADT，提供线性表基本的、常用的 12 中运算和操作。要求中还有关于文件 I/O 的细节需要实现，使得线性表可以实现内外存交换，便于数据读写。同时，需要实现一个演示系统实现简单的演示，以此作为可用性检查的工具。

2.2 系统设计

2.2.1 总体系统

该线性表的实现包括一个.CPP 文件。文件中定义了线性表的结构体、相关运算的函数定义及其实现、测试系统、多表管理、文件操作等函数。由于整个线性表比较简单，而且没有“#include”的需要，因此没有使用头文件。

2.2.2 数据结构

数据结构的实现根据实验要求的 ADT 定义，定义了基于 C 语言结构体的 ADT 定义。由于实验要求给出的代码的 Codebase 是经典的 C 语言面向过程的写法，没有使用 C++的类。为了提高可复用性和聚合度，使用 C++的模板。结构体如下：

```
template <typename T>
    struct Node {
    T data;
    Node* next;
};

template <typename T>
    struct LinkedList {
    int length;
    bool init;
    Node<T>* head;
};
```

2.2.3 ADT 操作的设计

根据实验的要求以及线性表 ADT 的定义，该线性表 ADT 应包括如下的操作：

1. InitList（初始化）
2. DestroyList（销毁）
3. ClearList（清空）
4. ListEmpty（判断表是否为空）
5. ListLength（求表长）
6. GetElem（按下标取得元素）
7. LocateElem（按满足关系取得元素）
8. PriorElem（返回满足关系的元素的前驱）
9. NextElem（返回满足关系元素的后继）
10. ListInsert（插入元素）
11. ListDelete（删除元素）
12. ListTraverse（遍历表）

为了便于错误处理，定义以下常量作为错误标识：

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

为了方便更改线性表中元素的类型，使用以下的宏定义：

```
typedef int status;
typedef int ElemType;
```

其中，status 是某个方法的返回类型，用于标志是否正确执行了相应的运算。Status 的取值由上面的宏定义定义。当不为 1 时，认为发生了错误。然后可以

根据 status 的值确定发生了何种错误。

ADT 的运算有下面的函数定义，这些函数均返回 status（如果不返回其他有用的值）、bool（判断是否为空）和 size_t（求出长度或者位置）：

```
status InitList(LinkedList<T>&);
status DestroyList(LinkedList<T>&);
status ClearList(LinkedList<T>&);
bool ListEmpty(LinkedList<T>);
size_t ListLength(LinkedList<T>);
status GetElem(LinkedList<T>, size_t i, ElemType& e);
size_t LocateElem(LinkedList<T>, ElemType e);
status PriorElem(LinkedList<T>, ElemType cur, ElemType& pre_e);
status NextElem(LinkedList<T>, ElemType cur, ElemType& next_e);
status ListInsert(LinkedList<T>&, size_t i, ElemType e);
status ListDelete(LinkedList<T>&, size_t i, ElemType& e);
status ListTraverse(LinkedList<T>);
```

上面的 ListTraverse 经过了简化，使得其遍历行为变为固定的打印。这一点使线性表的遍历失去了灵活性。这是一个可以优化和改善的点。

2.2.4 多表系统的设计

使用 C++ STL 的 Vector 作为多表的容器。使用容器可以简化相关的操作，同时具有很高的健壮性，也可以减少问题。

具体的思路为：在程序开始的时候创建一个 vector，将第一个默认的 LinkedList 添加到里面，同时设置一个索引变量，用于指示线性表在 vector 中的位置。在运行的时候，每当用户选择新建一个线性表的时候，就将一个线性表添加到里面。同时将当前的线性表索引指向新的线性表。

容器的定义如：vector<LinkedList<T>> Lists = {LinkedList<T>()};

2.2.5 文件存储系统设计

文件操作可以使线性表的元素存放到硬盘上永久保存，更加贴近真实的使用

场景。通过 C 语言标准库提供的文件读写 API，可以很方便的操作文件。

通过 fopen 可以返回一个指针，该指针可以用来操作先前打开的文件。同时可以制定打开文件的方式，例如“r”可以用来打开一个只读文件。然后使用此文件进行读写操作，最后使用 fclose 销毁该指针。在写文件的时候，可以将整个 elems 数组以 2 进制的方式存入文件。在读取的时候，连续读取每次读取 sizeof(T)个字节，然后将这些字节强制类型转换为 T，然后使用 ListInsert 将其插入到线性表的尾部，直到读取到文件尾为止。

2.3 系统实现

2.3.1 开发环境

开发环境选用 Windows 10 上的 VSCode 作为编辑器，g++作为编译器，g++的版本为 8.1.0。

2.3.2 ADT 操作的实现

status InitList(LinkedList<T>&);

该操作接受一个 LinkedList<T>的引用，将其作为初始化对象进行初始化。首先，先要检查该线性表是否存在，如果存在，则将其长度设为 0，将 init 设置为 true，将 head 指向一个 Node<T>的结构体，该结构体作为头节点。完成初始化，则返回 status 中的 OK，否则返回相应的错误类型。

该操作的时间复杂度 O(1)，空间复杂度 O(1)。

status DestroyList(LinkedList<T>&);

该操作用于销毁一个线性表。该函数接受一个 LinkedList<T>的引用，将其 head 所指向的内存 delete，然后沿着链表的 next，delete 所有的内存，由于 C++中有构造函数和析构函数的概念，因此这里只能使用 delete。将头节点的 head 指针置为 0；同时，将该线性表设置为已销毁。如果成功，则返回 status 中的 OK，否则返回相应的错误类型。

该操作的时间复杂度为 O(n),空间复杂度为 O(1)。

status ClearList(LinkedList<T>&);

该操作用于将一个线性表 LinkedList<T>清空。该函数接受一个LinkedList<T>的引用，如果LinkedList<T>没有被初始化，则直接返回。否则将size 设为0，然后沿着链表的next，将所有的Node<T>都delete 掉。如果成功，则返回status 中的OK，否则返回相应的错误类型。

该操作的时间复杂度是O(n)，空间复杂度O(1)。

bool ListEmpty(LinkedList<T>);

该操作用于判断一个线性表是否为空，返回值为 bool 类型，当为真时说明线性表为空，否则说明线性表存在元素。如果某线性表未经初始化，那么他一定为空。

该操作的时间复杂度是O(1)，空间复杂度O(1)。

size_t ListLength(LinkedList<T>);

该操作用于求取一个线性表的长度（实际包含元素的长度）。对于一个已经初始化的线性表，直接返回size 的值。其他情况下则返回 -1（会被隐式类型转换为MAX_LONG_LONG_INT）。

该操作的时间复杂度是O(1)，空间复杂度O(1)。

status GetElem(LinkedList<T>, size_t i, ElemType& e);

该操作用于获取线性表中的某一个元素。接受3个参数，第一个为线性表的拷贝，第二个为要获取的线性表所在的位置，第三个为变量的引用。获取元素需要从链表的一端向后遍历，知道找到相应下标的元素。获取成功的元素的值将被存放到这个变量中。如果该操作正常执行，则返回status 中的OK，否则返回相应的错误。

该操作的时间复杂度是O(n)，空间复杂度O(1)。

size_t LocateElem(LinkedList<T>, ElemType e);

该函数接受两个参数，一个是线性表的复制。一个是要定位的元素的值。该

操作用于求取给定的 e 在线性表中的位置，可以实现为返回从左到右第一次出现的位置，位置从零开始，-1 表示未找到。该函数的实现为遍历该线性表，当第一次遇到和 e 相等的元素时，则返回当前的循环变量。

该操作的时间复杂度是 O(n)，空间复杂度 O(1)。

status PriorElem(LinkedList<T>, T cur, T& pre_e);

该操作接收三个参数：一个 LinkedList<T>的拷贝、一个 T 的变量 cur、一个 T 的引用 pre_e，找到和 cur 变量相等的元素的前驱（如果存在），并且将 pre_e 赋值为它，否则操作失败，pre_e 无定义。

该操作首先判断表是否存在，如果存在则依次遍历整个线性表，判断线性表中的每个元素是否与传入的 cur_e 变量相等，如果满足则直接结束遍历，并且将 pre_e 赋值为该元素的前驱。如果成功，则返回 status 中的 OK，否则返回相应的错误类型。

该操作的时间复杂度是 O(n)，空间复杂度是 O(1)。

status NextElem(LinkedList<T>, T cur, T& next_e);

该操作接收三个参数：一个 LinkedList<T>的拷贝、一个 T 的变量 cur、一个 T 的引用 next_e，找到和 cur 变量相等的元素的前驱（如果存在），并且将 next_e 赋值为它，否则操作失败，next_e 无定义。

该操作首先判断表是否存在，如果存在则依次遍历整个线性表，判断线性表中的每个元素是否与传入的 cur_e 变量相等，如果满足则直接结束遍历，并且将 pre_e 赋值为该元素的前驱。如果成功，则返回 status 中的 OK，否则返回相应的错误类型。

该操作的时间复杂度是 O(n)，空间复杂度是 O(1)。

status ListInsert(LinkedList<T>&, size_t i, T e);

该操作接收三个参数：一个 LinkedList<T>的拷贝、一个 size_t 的变量 i、一个 T 的变量 e。用于将 e 元素插入到位于 i 位置的元素之前。

该操作利用链表的特点，直接使用 PriorElem 获取到相应的元素的前驱结点，

然后修改它前驱节点的 next，以完成插入。

该操作的时间复杂度是 O(n)，空间复杂度是 O(1)。

status ListDelete(LinkedList<T>&, size_t i, ElemType& e);

该操作接收三个参数：一个 LinkedList<T>的拷贝、一个 size_t 的变量 i、一个 ElemType 的引用 e。

该操作利用链表的特点,直接使用 PriorElem 获取到相应的元素的前驱结点，然后修改它前驱节点的 next，修改为该前驱节点的下一节点的下一节点，以完成删除。

该操作的时间复杂度是 O(n)，空间复杂度是 O(1)。

status ListTraverse(LinkedList<T>, function<void(Node<T>)> cb);

该操作接收一个 LinkedList<T>拷贝和一个 cb 回调函数。依次对线性表中的每个元素执行 cb 函数指定的操作。

该操作首先检查表是否存在、是否为空，存在且非空的话，则循环遍历整个线性表，并且对每个元素执行 cb 函数指定操作。

该操作的时间复杂度是 O(n)，空间复杂度是 O(1)。

2.3.3 多表系统的实现

多表系统实现了添加一张表、更改当前表两个操作（没有实现删除表）。

添加一张表：

添加一张表就等价于在 vector<LinkedList<T>>中添加了一个 LinkedList<T>,该LinkedList<T>的所有字段都是按照默认初始化进行初始化的。因此，该线性表虽然有了内存空间但是却是一个空表，其 size 以及 elems 等关键字段都是无效的值。

更改当前表：

更改当前表就等价于修改了指向 vector<LinkedList<T>>的下标。该下标的取值为 0 至 vector<LinkedList<T>>的长度。修改下标即可实现修改当前表所指向的值。从而达到修改当前表的目的。

容易可得，这两个操作的时间复杂度都为 O(1)。

2.3.4 文件存储系统实现

通过 fopen 可以返回一个指针，该指针可以用来操作先前打开的文件。同时可以制定打开文件的方式，例如“r”可以用来打开一个只读文件。然后使用此文件进行读写操作，最后使用 fclose 销毁该指针。在写文件的时候，可以将整个 elems 数组以 2 进制的方式存入文件。在读取的时候，连续读取每次读取 sizeof(T)个字节，然后将这些字节拷贝到 elems 里面，直到读取到文件尾为止。同时，每次读取使线性表的长度增加 1 个长度。

2.4 系统测试

2.4.1 演示系统测试

演示系统测试如下图所示：

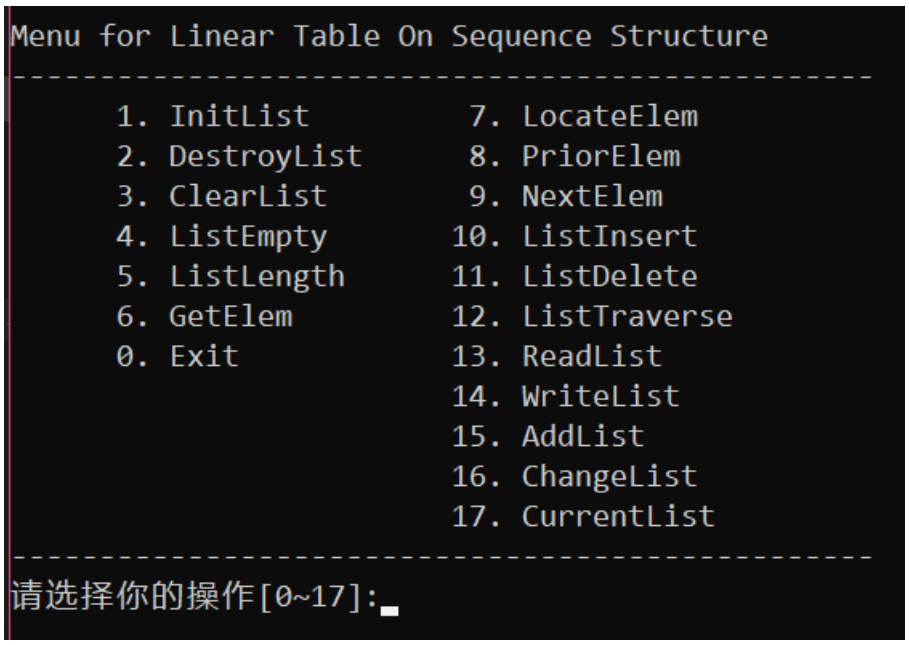


图 2-0-1 演示系统

该演示系统操作简单，输入对应的数字并按下回车即可。输入 1-17 表明运行对应的功能，且相应操作的结果会被输出到屏幕上，输入 0 退出系统。

2.4.2 测试样例

使用有效的典型的测试样例可以有效的寻找系统的错误，同时可以提高测试的覆盖率,使测试更加全面.正确的测试样例可以有效地测试系统是否正确,而错误的测试样例则可以用来测试系统的健壮性.按照如下的操作流程即可完整的测

试该系统:

1. 获取 3 元素的下标。
2. 获取 1000 元素的下标,错误样例
3. 获取 3 元素的前驱
4. 获取 1 元素的前驱,错误样例
5. 获取 3 元素的后继
6. 获取 5 元素的后继,错误样例
7. 删除下标为 1 和 1000 的元素。
8. 遍历线性表。
9. 将线性表存入文件。
10. 添加一个线性表。
11. 切换到新添加的线性表 1，并初始化。
12. 从刚存储的文件中加载线性表。
13. 遍历线性表。
14. 删除下标为 1 的元素。
15. 遍历线性表。
16. 切换到旧的线性表 0。
17. 遍历线性表。

2.4.2 测试结果

初始化一个线性表：

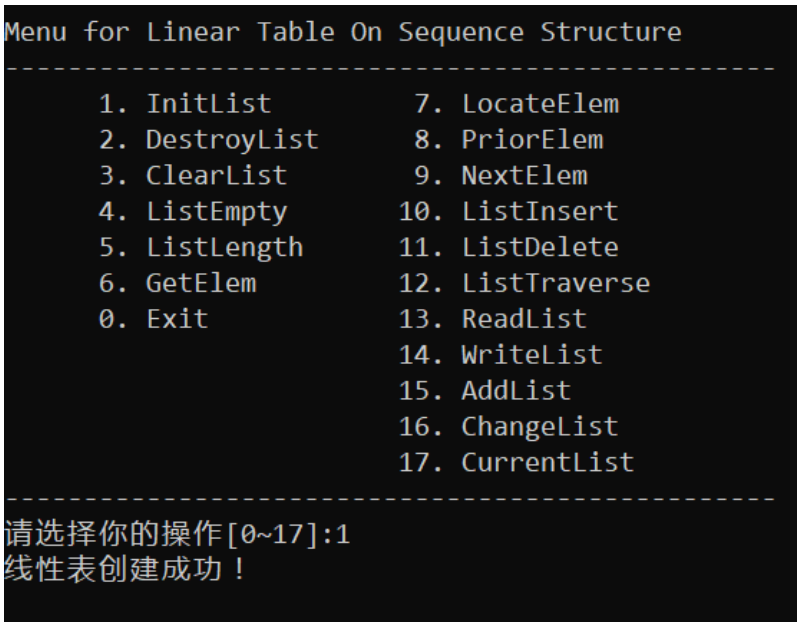


图 2-0-2 初始化成功

由图可见，初始化线性表成功。

测试线性表的长度：

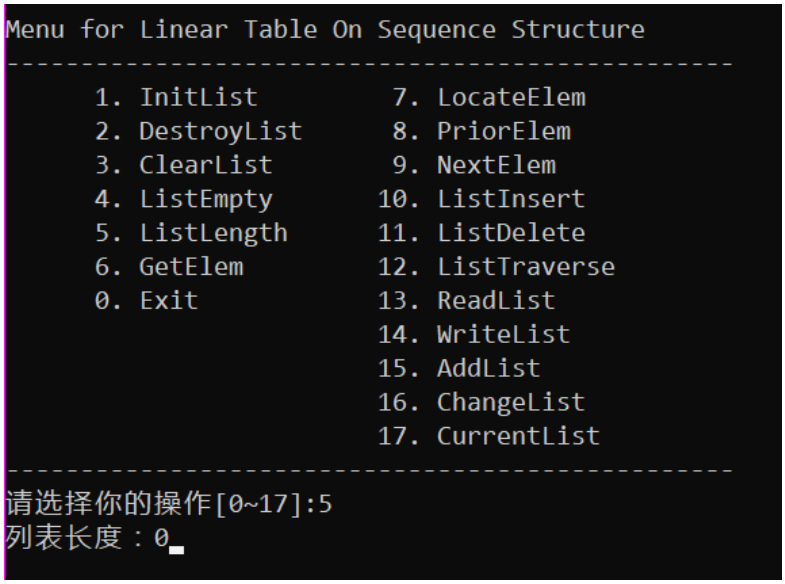


图 2-0-3 长度测试正确

线性表的长度为 0，符合预期结果。

然后插入 1~5 的元素，如下图所示，依次在 0 位置插入 5、4、3、2、1，即可实现插入 1~5 元素。

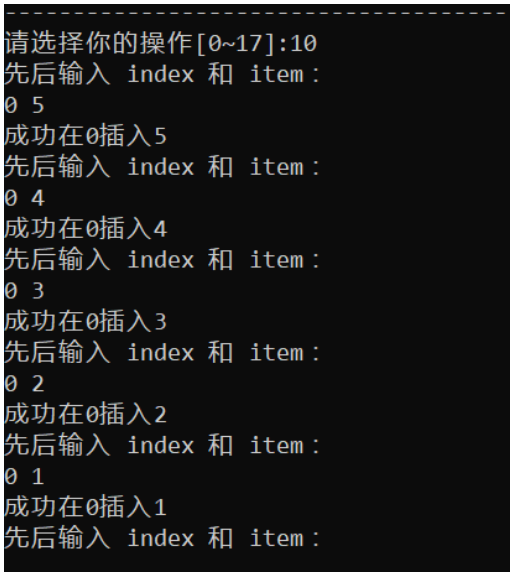


图 2-0-4 插入元素

为了验证我们确实插入了 1~5 的元素，我们使用 ListTraverse 来验证：如果 ListTraverse 输出 1 2 3 4 5，则说明该操作正确。完成操作之后的截图如下，可见，该操作是正确的。

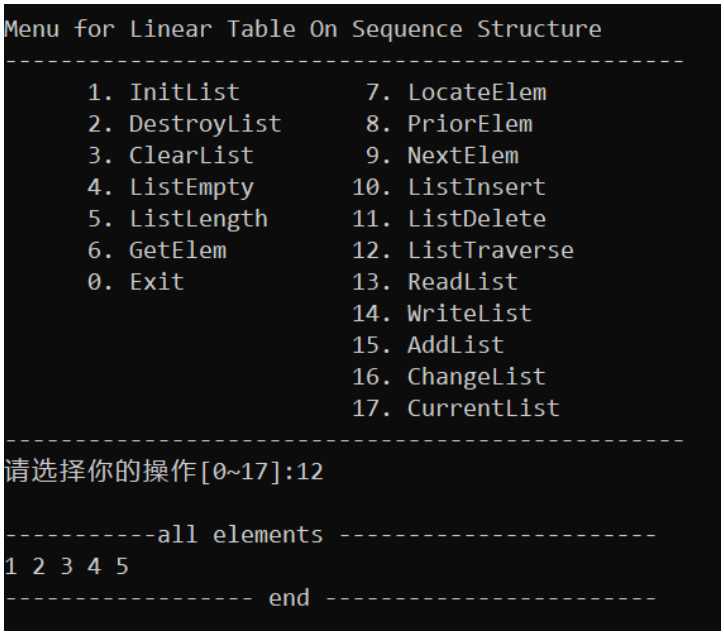


图 2-0-5 遍历线性表

此时，测试改线性表的长度，即使用 ListLength 操作，得到的输出如下，列表长度为 5，该结果正确且符合预期。由此可得 ListInsert、ListTraverse、ListLength 可以正常执行操作。

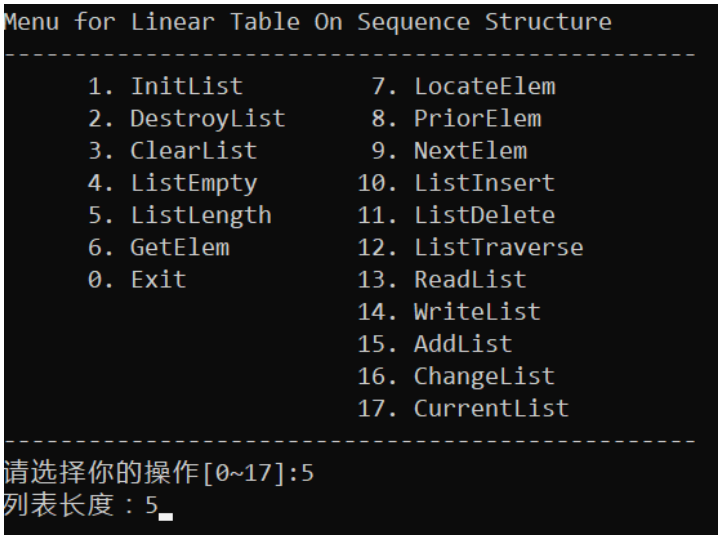


图 2-0-6 获取线性表长度

接下来获取下标为 3 的元素，使用 GetElem 来实现该操作，输入了 index（下标）3 之后，按下回车即可获取改下标对应的值。

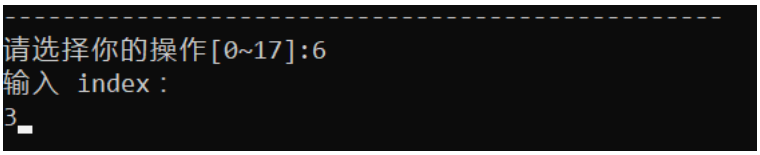


图 2-0-7 获取相应下标的元素

如图所示，使用 GetElem 获得下标为 3 的元素，输出的值为 4，因为我们在线性表中存储了 1、2、3、4、5，所以下标为三的元素为 4，结果正确，符合预期。如下图所示：

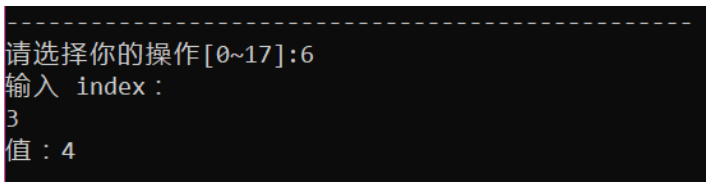


图 2-0-8 获取相应下标的元素

为了测试该系统的鲁棒性，测试获取下标为 1000 的元素时，该操作的行为。由于线性表中只有 5 个元素，因此获取下标为 1000 的元素直接报错。通过该测试可知该操作具有很好的容错机制。

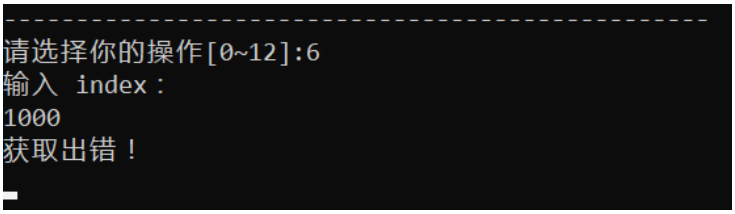


图 0-9 获取相应下标的元素（错误案例）

为了测试 3 元素的下标，我们使用 LocateElem 操作，对 3 元素进行定位，根

据前面的描述，该操作接受某元素的拷贝为参数，返回从左到右第一次出现该元素的下标，该下标从 0 开始。测试过程如下：

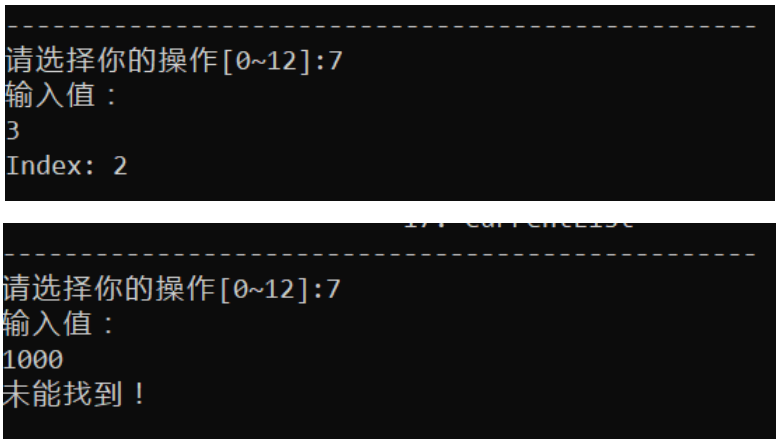


图 2-0-10 定位元素的位置

如上图所示，先后测试元素 3 和 1000 为参数的 LocateElem，由于 3 的下标是 2，因此程序输出 2，结果正确且符合预期。1000 不在线性表中，因此该值的下标输出为“未能找到!”，以此提醒使用者该操作无效。由截图可见，LocateElem 的行为完全正确。

为了测试 PriorElem 的正确性，使用 1 和 3 为参数进行测试，由于元素 1 实际上没有前驱，因此该操作应该输出错误信息。而 3 的前驱为 2，我们应该期望对于 3 进行获取前驱操作应当输出 2。

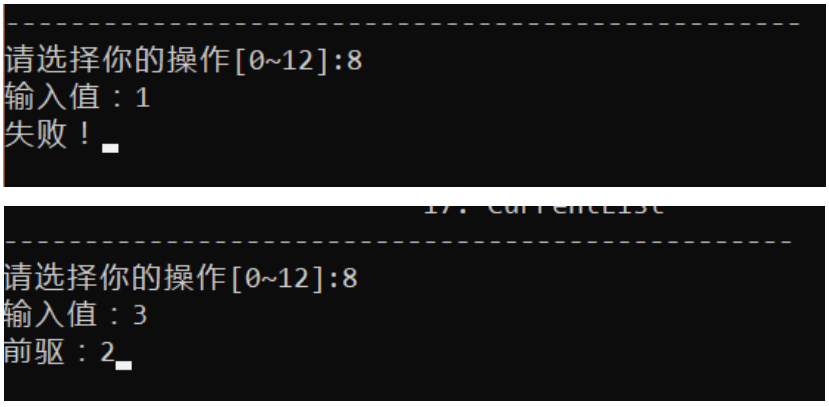


图 2-0-11 获取元素的前驱

如上图所示，该操作输出了正确的信息，因此可以认为 PriorElem 的行为是正确的。

为了测试 NextElem 的正确性，使用 3 和 5 为参数进行测试，由于元素 5 实际上没有后继，因此该操作应该输出错误信息。而 3 的后继为 4，我们应该期望对于 3 进行获取前驱操作应当输出 4。

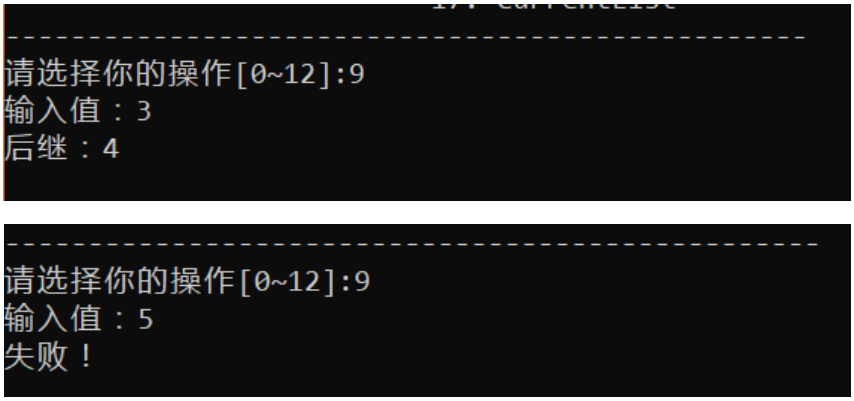


图 2-0-12 获取元素的后继

如上图所示，该操作输出了正确的信息，因此可以认为 NextElem 的行为是正确的。

为了测试删除元素操作 ListDelete 的正确性，使用 1 和 1000 作为参数进行测试，将 1 和 1000 作为参数传递给 ListDelete，由于下标为 1 的元素为 2，因此执行完 ListDelete(1)之后，线性表中的元素应为 1、3、4、5；由于 1000 下标的元素不存在与线性表中，因此该操作将输出错误。

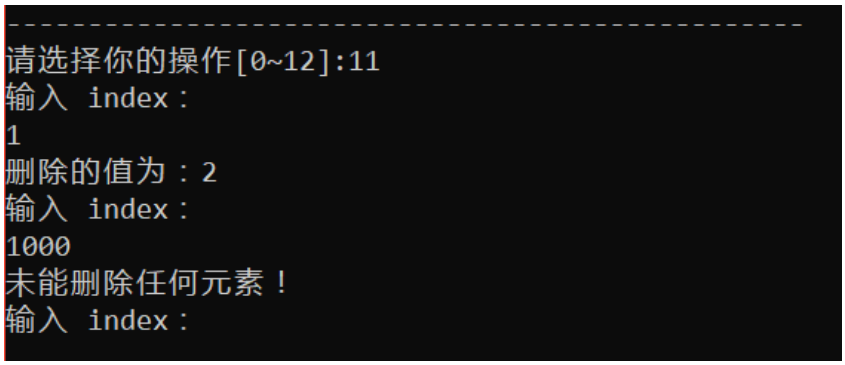


图 2-0-13 删除元素

如上图可知，该操作的输出符合预期，因此可以认为该操作对应的程序无错误。为了进一步确认 ListDelete 对线性表的修改，使用 ListTraverse 来进行进一步测试，通过判断 ListTraverse 的输出值，查看执行 ListDelete 之后线性表的内容。

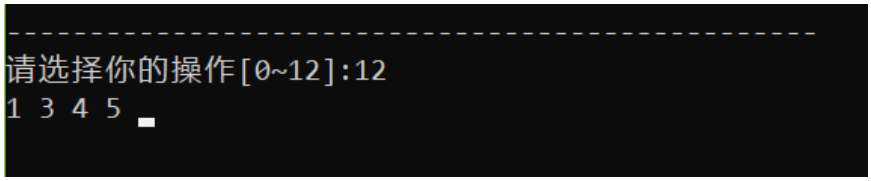


图 2-0-14 删除后遍历

为了测试文件操作的可行性，使用 WriteList 操作。输入一个随意的文件名，例如 2 进行测试，文件将会被打开然后，程序将会将数据写入到打开的文件中，如果文件打开失败，会输出“File open error”。

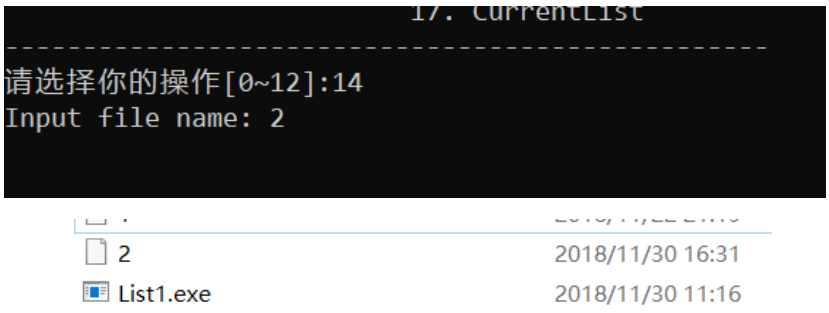


图 2-0-15 写出到文件

如图所示，文件成功创建。为了验证文件内容的正确性，新建一个 SqList 并将该文件导入，即可检测该功能的正确性。依次进行 AddList、ChangeList、InitList 等操作，即可创建、切换并初始化一个新的线性表。然后运行 ListTraverse 操作，即可得到改线性表的所有元素。可见，这些元素和之前的线性表完全一致。

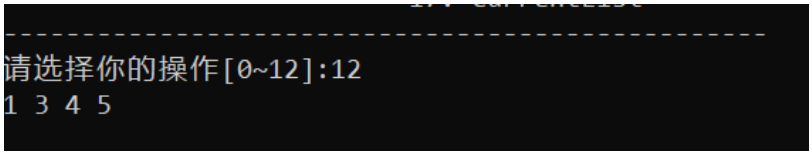


图 2-0-16 读入文件

为了证明该线性表和之前的线性表不是同一个线性表，修改新的线性表并删除一个元素，为了方便演示，删除下标为 1 的元素。可见，下标为 1 的元素被成功删除，由于删除之前该线性表存储的元素是 1、3、4、5，删除下标为 1 的元素导致线性表变为 1、4、5。

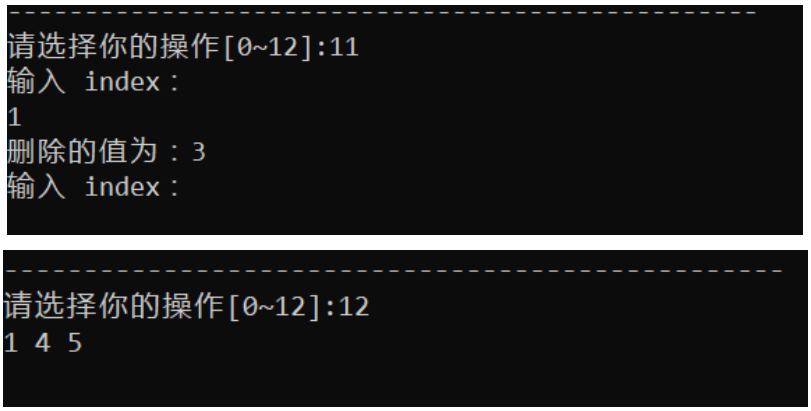


图 2-0-17 删除元素（新线性表）

由上图可知，遍历的结果和预期吻合，所以改线性表的文件操作无错误。而我们的目的是验证新的线性表不会影响旧的线性表，因此我们切换到之前的线性

表，并执行遍历操作。

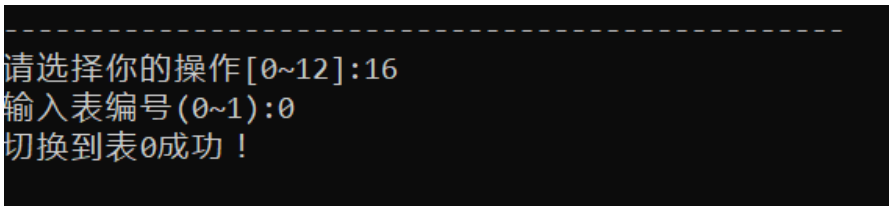


图 2-0-18 切换到原表

可见，当前的线性表是线性表 0，也就是之前的旧的线性表。在该线性表上执行 ListTraverse 操作，即可获取所有的元素。由下图可见，所有的元素均列出，同时，没有出现和新线性表保持同步的现象，这说明在多表的情况下，线性表操作是独立的，不会互相影响。

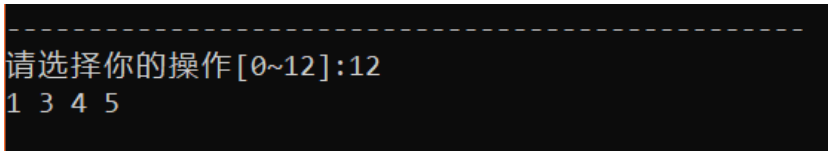


图 2-0-19 遍历原表

2.5 实验小结

通过本次实验，我加深了对于基于链表数据结构的线性表的认识和了解。

在这次实验中，我使用 C 语言的风格封装了一个可以使用的线性表，同时，为了使该系统的可复用性提高，使用了 C++的模板来实现了泛型。这次实验我知道了如何实现一个线性表，将课本的知识转化为可以使用的程序，同时我对 C 语言的错误处理、面向过程的编程有了更深的理解。通过对测试系统的实现，我学会了如何全面的测试我的程序，也知道了如何提高系统的鲁棒性。

3 基于二叉链表的二叉树实现

3.1 问题描述

采用二叉链表作为树的物理结构，通过 C 语言程序实现课本§3.2 的基本运算。要求具有易于操作易于理解的简易菜单，可选择实现树的文件形式存储。源代码须有适当的注释，便于检查和后期的修改与理解。

3.2 系统设计

该线性表的实现包括一个.CPP 文件。文件中定义了线性表的结构体、相关运算的函数定义及其实现、测试系统、多表管理、文件操作等函数。由于整个线性表比较简单，而且没有“#include”的需要，因此没有使用头文件。

3.2.1 总体系统

通过实验达到：⑴加深对二叉树的概念、基本运算的理解；⑵熟练掌握二叉树的逻辑结构与物理结构的关系；⑶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。

3.2.1 数据结构

数据结构的实现根据实验要求的 ADT 定义，定义了基于 C 语言结构体的 ADT 定义。由于之前的实验（线性表）要求给出的代码的 Codebase 是经典的 C 语言面向过程的写法，没有使用 C++的类，对于此次实验按照该模式模仿。结构体如下：

```
struct ElemType {
    ValueType value;
    size_t index;
    bool null = true;
};
struct BiTreeNode {
    ElemType data;
    BiTreeNode* parent = NULL;
    BiTreeNode* left = NULL;
    BiTreeNode* right = NULL;
};
```

```
struct BiTree {
  bool init = false;
  BiTreeNode* root = NULL;
};
```

3.2.3 ADT 操作的设计

根据实验的要求以及二叉树 ADT 的定义，该二叉树 ADT 应包括如下的操作：

1. InitBiTree（初始化）
2. DestroyBiTree（销毁）
3. CreateBiTree（根据 definition 创建）
4. ClearBiTree（清空）
5. BiTreeEmpty（判断二叉树是否为空）
6. BiTreeDepth（求二叉树的深度）
7. Root（获取二叉树的根节点）
8. Value（根据 index 获取二叉树的节点值）
9. Assign（根据 index 获取某个节点并赋新值）
10. Parent（返回某节点的双亲节点）
11. LeftChild（返回某节点的左孩子）
12. RightChild（返回某节点的右孩子）
13. LeftSibling（返回某节点的左兄弟）
14. RightSibling（返回某节点的右兄弟）
15. InsertChild（将某非空无右子树的二叉树插入到某节点的左或右孩子处）
16. DeleteChild（删除某节点的左或右孩子）
17. PreOrderTraverse（先序遍历二叉树）
18. InOrderTraverse（中序遍历二叉树）
19. PostOrderTraverse（后序遍历二叉树）
20. LevelOrderTraverse（层级遍历二叉树）

为了便于错误处理，定义以下常量作为错误标识：

```
enum Error {
  INIT,
  NOT_INIT,
  WRONG_DEF,
```

```
    NO_SUCH_NODE,
};
```

其中，status 是某个方法的返回类型，用于标志是否正确执行了相应的运算。Status 的取值由上面的宏定义定义。当不为 1 时，认为发生了错误。然后可以根据 status 的值确定发生了何种错误。

ADT 的运算有下面的函数定义，这些函数均返回 status（如果不返回其他有用的值）、bool（判断是否为空）、size_t（求出长度或者位置）、BiTreeNode*（获取某个节点）。

```
status InitBiTree(BiTree& T);
status DestroyBiTree(BiTree& T);
status CreateBiTree(BiTree& T, vector<ElemType> def);
status ClearBiTree(BiTree& T);
bool BiTreeEmpty(const BiTree& T);
int BiTreeDepth(const BiTree& T);
BiTreeNode* Root(const BiTree& T);
status Value(const BiTree& T, size_t index, ElemType& value);
status Assign(BiTree& T, size_t index, ElemType& value);
BiTreeNode* Parent(const BiTree& T, size_t index);
BiTreeNode* LeftChild(const BiTree& T, size_t index);
BiTreeNode* RightChild(const BiTree& T, size_t index);
BiTreeNode* LeftSibling(const BiTree& T, size_t index);
BiTreeNode* RightSibling(const BiTree& T, size_t index);
status InsertChild(BiTree& T, size_t index, bool LR, BiTree& c);
status DeleteChild(BiTree& T, size_t index, bool LR);
status PreOrderTraverse(const BiTree& T, function<void(BiTreeNode*)>);
status InOrderTraverse(const BiTree& T, function<void(BiTreeNode*)>);
status PostOrderTraverse(const BiTree& T, function<void(BiTreeNode*)>);
status LevelOrderTraverse(const BiTree& T);
```

3.2.4 多表系统的设计

使用 C++ STL 的 Vector 作为多二叉树的容器。使用容器可以简化相关的操作，同时具有很高的健壮性，也可以减少问题。

具体的思路为：在程序开始的时候创建一个空 vector，同时设置一个索引变量，用于指示当前二叉树在 vector 中的位置。在运行的时候，每当用户选择新建一个线性表的时候，就将一个线性表添加到里面。

容器的定义如：vector<BiTree> trees = {};

2.2.5 文件存储系统设计

文件操作可以使二叉树的节点存放到硬盘上永久保存，更加贴近真实的使用场景。通过 C 语言标准库提供的文件读写 API，可以很方便的操作文件。
通过 fopen 可以返回一个指针，该指针可以用来操作先前打开的文件。同时可以制定打开文件的方式，例如“r”可以用来打开一个只读文件。然后使用此文件进行读写操作，最后使用 fclose 销毁该指针。

具体的实现为，写出到文件时，先将二叉树容器 trees 的 size 写出到文件，然后遍历多二叉树的容器 trees，对于每一个 BiTree，先将其节点数目写出，然后先序遍历该二叉树（由于需要可以复原该二叉树，该先序遍历允许空节点），将所有的节点依次写出到文件中，每次写出 sizeof(ElemType)个字节。

读文件时将上述操作逆着执行一遍即可。

3.3 系统实现

3.3.1 开发环境

开发环境选用 Windows 10 上的 VSCode 作为编辑器，g++作为编译器，g++的版本为 8.1.0。

3.3.2 ADT 操作的实现

status InitBiTree(BiTree& T);

该操作接受一个已经开辟好内存的 BiTree 结构体的引用为参数，返回该操作执行的状态。该操作用于初始化相关的二叉树。首先，该操作检查二叉树是否已经被初始化了，如果已经被初始化了，则返回一个 Error::INIT 的错误。否则将该二叉树设为已初始化。

该操作的时间复杂度为 O(1)，空间复杂度为 O(1)。

status DestroyBiTree(BiTree& T);

该操作接受一个已经开辟好内存的 BiTree 结构体的引用为参数，返回该操作执行的状态。该操作用于销毁相关的二叉树。首先，该操作检查二叉树是否已经被初始化了，如果没有被初始化，则返回一个 Error::NOT_INIT 的错误。否则将该二叉树设为未初始化，由于二叉树占用的内存需要被释放，因此会调用

ClearBiTree 方法。ClearBiTree 方法会在后面说明。

该操作的时间复杂度为 O(N)，空间复杂度为 O(logN)。

status CreateBiTree(BiTree& T, vector<ElemType> def);

该操作接受一个已经开辟好内存的 BiTree 结构体的引用为参数，返回该操作执行的状态。该操作用于初始化相关的二叉树。首先，该操作检查二叉树是否已经被初始化了，如果没有被初始化，则返回一个 Error::NOT_INIT 的错误。否则根据 def 来创建该二叉树。创建二叉树的方法为：def 是一个 ElemType 的线性表，其中的每一个元素代表二叉树的一个节点。节点允许是空值，通过结构体中的 null 来指定。创建时，递归调用 Create 函数。Create 函数根据当前指向 ElemType 线性表中元素位置的指针，进行如下操作：若该元素对应的是空节点，则直接返回，将指针后移一位；若不是空节点，将指针后移一位，并递归调用 Create 分别创建左右子树。直到指针指向列表尾停止。

该操作的时间复杂度为 O(N)，空间复杂度为 O(logN)。

status ClearBiTree(BiTree& T);

该操作接受一个已经开辟好内存的 BiTree 结构体的引用为参数，返回该操作执行的状态。该操作用于清空相关的二叉树。首先，该操作检查二叉树是否已经被初始化了，如果没有被初始化，则返回一个 Error::NOT_INIT 的错误。否则清空该二叉树。清空二叉树的方法是：后序遍历该二叉树，将 Visit 函数指定为销毁相应的节点。由于后序遍历是按照左、右、根的顺序，该二叉树可以被正确销毁。

该操作的时间复杂度为 O(N)，空间复杂度为 O(logN)。

bool BiTreeEmpty(const BiTree& T);

该操作接受一个已经开辟好内存的 BiTree 结构体的引用为参数，返回该二叉树是否为空。该操作用于判断相关的二叉树是否为空。首先，该操作检查二叉树是否已经被初始化了，如果没有被初始化，则返回一个 Error::NOT_INIT 的错误。判断二叉树是否为空的方法是：判断该二叉树的根节点是否为空节点。

该操作的时间复杂度为 O(1)，空间复杂度为 O(1)。

int BiTreeDepth(const BiTree& T);

该操作接受一个已经开辟好内存的 BiTree 结构体的引用为参数，返回该二叉

树的深度。该操作用于求取相关的二叉树的深度。首先，该操作检查二叉树是否已经被初始化了，如果没有被初始化，则返回一个 Error::NOT_INIT 的错误。求取二叉树的深度的方法是：递归执行 Depth 函数。该函数的作用是求取二叉树的深度。当该函数作用的子二叉树是叶子节点时，返回 1，否则递归的调用 Depth 函数求取左右子树的深度，并取最大值。

该操作的时间复杂度为 O(N)，空间复杂度为 O(logN)。

BiTreeNode* Root(const BiTree& T);

该操作接受一个已经开辟好内存的 BiTree 结构体的引用为参数，返回该操作执行的结果。该操作用于获取二叉树的根节点。首先，该操作检查二叉树是否已经被初始化了，如果没有被初始化，则返回一个 Error::NOT_INIT 的错误。获取二叉树根节点的方法是直接返回 BiTree::root 域。对于空的二叉树，返回一个 NULL 指针。

该操作的时间复杂度为 O(1)，空间复杂度为 O(1)。

status Value(const BiTree& T, size_t index, ElemType& value);

该操作接受一个已经开辟好内存的 BiTree 结构体的引用为第一个参数、要获取 Value 的节点 index 为第二个参数、内容将要存放的位置为第三个引用参数，返回该操作执行的状态。该操作用于获取二叉树某个节点的值（ElemType）。首先，该操作检查二叉树是否已经被初始化了，如果没有被初始化，则返回一个 Error::NOT_INIT 的错误。获取二叉树某个节点值的方法是：遍历该二叉树，依次判断各个节点的 index 是否和参数提供的相等，如果相等则将 value 引用赋值为该节点。

该操作的时间复杂度为 O(N)，空间复杂度为 O(logN)。

status Assign(BiTree& T, size_t index, ElemType& value);

该操作接受一个已经开辟好内存的 BiTree 结构体的引用为第一个参数、被赋值的节点 index 为第二个参数、要赋的值为第三个引用参数，返回该操作执行的状态。该操作用于为二叉树某个节点赋值。首先，该操作检查二叉树是否已经被初始化了，如果没有被初始化，则返回一个 Error::NOT_INIT 的错误。为二叉树某个节点赋值的方法是：遍历该二叉树，依次判断各个节点的 index 是否和参数提供的相等，如果相等则将 value 赋值给该节点。

该操作的时间复杂度为 O(N)，空间复杂度为 O(logN)。

BiTreeNode* Parent(const BiTree& T, size_t index);

BiTreeNode* LeftChild(const BiTree& T, size_t index);

BiTreeNode* RightChild(const BiTree& T, size_t index);

BiTreeNode* LeftSibling(const BiTree& T, size_t index);

BiTreeNode* RightSibling(const BiTree& T, size_t index);

这五个操作接受一个已经开辟好内存的 BiTree 结构体的引用为第一个参数、要获取 Parent、左孩子、右孩子、左兄弟、右兄弟的节点的 index 为第二个参数，返回与该节点有相应关系的节点。首先，该操作检查二叉树是否已经被初始化了，如果没有被初始化，则返回一个 Error::NOT_INIT 的错误。获取二叉树某个节点的相应关系的节点的方法是：

双亲结点：遍历该二叉树，依次判断各个节点的左右 Child 节点的 index 是否和参数提供的相等，如果相等则将该节点返回。

左孩子结点：遍历该二叉树，依次判断各个节点的 index 是否和参数提供的相等，如果相等则将该节点的 BiTreeNode::left 域返回。

右孩子结点：遍历该二叉树，依次判断各个节点的 index 是否和参数提供的相等，如果相等则将该节点的 BiTreeNode::right 域返回。

左兄弟结点：遍历该二叉树，依次判断各个节点的 index 是否和参数提供的相等，如果相等则将该节点的父节点的左孩子节点返回，如果等于该节点本身，则返回 NULL。

右兄弟结点：遍历该二叉树，依次判断各个节点的 index 是否和参数提供的相等，如果相等则将该节点的父节点的右孩子节点返回，如果等于该节点本身，则返回 NULL。

如果要寻找的节点不存在，这五个操作均会抛出（throw）一个 Error::NO_SUCH_NODE 错误。

这五个操作的时间复杂度为 O(N)，空间复杂度为 O(logN)。

status InsertChild(BiTree& T, size_t index, bool LR, BiTree& c);

该操作接受一个已经开辟好内存的 BiTree 结构体的引用为第一个参数、被插入的节点的 index 为第二个参数、插入位置左或右作为第三个参数、要插入的树

作为第四个引用参数，返回该操作执行的状态。该操作用于将某个非空无右子树的树插入到某个节点中。首先，该操作检查二叉树是否已经被初始化了，如果没有被初始化，则返回一个 Error::NOT_INIT 的错误。插入的方法是：首先找到要删除左或右子树的节点 T，根据 LR 为 0 或者 1，插入 c 为 T 中 p 所指结点的左或右子树，p 所指结点的原有左子树或右子树则为 c 的右子树。

该操作的时间复杂度为 O(N)，空间复杂度为 O(logN)。

status DeleteChild(BiTree& T, size_t index, bool LR);

该操作接受一个已经开辟好内存的 BiTree 结构体的引用为第一个参数、被插入的节点的 index 为第二个参数、插入位置左或右作为第三个参数，返回该操作执行的状态。该操作用于将树的某节点的左或右子树删除。首先，该操作检查二叉树是否已经被初始化了，如果没有被初始化，则返回一个 Error::NOT_INIT 的错误。插入的方法是：首先找到要删除左或右子树的节点 T，根据 LR 为 0 或者 1，删除 c 为 T 中 p 所指结点的左或右子树。

该操作的时间复杂度为 O(N)，空间复杂度为 O(logN)。

status PreOrderTraverse(const BiTree& T, function<void(BiTreeNode*)>);

status InOrderTraverse(const BiTree& T, function<void(BiTreeNode*)>);

status PostOrderTraverse(const BiTree& T, function<void(BiTreeNode*)>);

这三个操作接受一个已经开辟好内存的 BiTree 结构体的引用为第一个参数、一个回调函数指针作为第二个参数，返回该操作执行的状态。该操作用于遍历该二叉树。首先，该操作检查二叉树是否已经被初始化了，如果没有被初始化，则返回一个 Error::NOT_INIT 的错误。遍历的方法是递归调用遍历函数。遍历函数根据先序、中序或者是后序，按照对应的顺序递归执行遍历函数：

先序遍历：对当前节点执行 Visit 函数、对左孩子递归调用遍历函数、对右孩子递归调用遍历函数。

中序遍历：对左孩子递归调用遍历函数、对当前节点执行 Visit 函数、对右孩子递归调用遍历函数。

后序遍历：对左孩子递归调用遍历函数、对右孩子递归调用遍历函数、对当前节点执行 Visit 函数。

该操作的时间复杂度为 O(N)，空间复杂度为 O(logN)。

status LevelOrderTraverse(const BiTree& T, function<void(BiTreeNode*)>);

这三个操作接受一个已经开辟好内存的 BiTree 结构体的引用为第一个参数、一个回调函数指针作为第二个参数，返回该操作执行的状态。该操作用于遍历该二叉树。首先，该操作检查二叉树是否已经被初始化了，如果没有被初始化，则返回一个 Error::NOT_INIT 的错误。层级遍历的实现方法如下：将根节点放入队列中，然后对它执行 Visit 函数并将其出队。然后将其左右子树放入队列（如果非空），循环执行直到队列为空。

该操作的时间复杂度为 O(N)，空间复杂度为 O(logN)。

3.3.3 多表系统的实现

多表系统实现了添加一张表、更改当前表两个操作（没有实现删除表）。

添加一张二叉树：

添加一张二叉树就等价于在 vector<BiTree>中添加了一个 BiTree 的实例，该 BiTree 的所有字段都是按照默认初始化进行初始化的。因此，该二叉树虽然有了内存空间但是却是一个空二叉树，其 root 等关键字段都是无效的值。

更改当前二叉树：

更改当前表就等价于修改了指向 vector<BiTree>的下标。该下标的取值为 0 至 vector<BiTree>的长度。修改下标即可实现修改当前表所指向的值。从而达到修改当前表的目的。

容易可得，这两个操作的时间复杂度都为 O(1)。

3.2.4 文件存储系统实现

通过 fopen 可以返回一个指针，该指针可以用来操作先前打开的文件。同时可以制定打开文件的方式，例如“r”可以用来打开一个只读文件。然后使用此文件进行读写操作，最后使用 fclose 销毁该指针。在写文件的时候，可以将整个 elems 数组以 2 进制的方式存入文件。在读取的时候，连续读取每次读取 sizeof(T)个字节，然后将这些字节拷贝到 elems 里面，直到读取到文件尾为止。同时，每次读取使线性表的长度增加 1 个长度。

文件操作可以使二叉树的节点存放到硬盘上永久保存，更加贴近真实的使用场景。通过 C 语言标准库提供的文件读写 API，可以很方便的操作文件。

通过 fopen 可以返回一个指针，该指针可以用来操作先前打开的文件。同时可以

制定打开文件的方式，例如“r”可以用来打开一个只读文件。然后使用此文件进行读写操作，最后使用 fclose 销毁该指针。

具体的实现为，写出到文件时，先将二叉树容器 trees 的 size 写出到文件，然后遍历多二叉树的容器 trees，对于每一个 BiTree，先将其节点数目写出，然后先序遍历该二叉树（由于需要可以复原该二叉树，该先序遍历允许空节点），将所有的节点依次写出到文件中，每次写出 sizeof(ElemType)个字节。

读文件时将上述操作逆着执行一遍即可。

3.4 系统测试

3.4.1 演示系统测试

演示系统测试如下图所示：

图 0-1 测试系统的界面

该演示系统操作简单，输入对应的数字并按下回车即可。输入 0-22 表明运行对应的功能，且相应操作的结果会被输出到屏幕上，输入 0 退出系统。输入-1 进入 debug 界面。

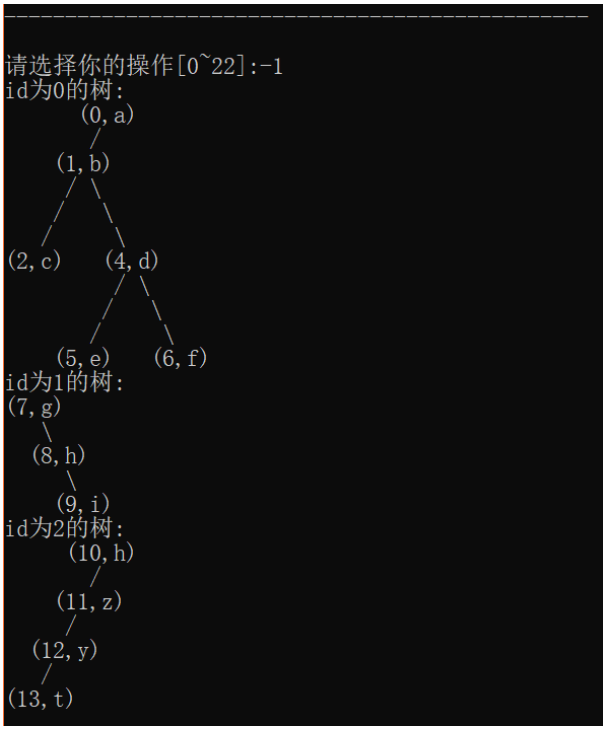


图 0-2 debug 界面

3.4.2 测试样例

使用有效的典型的测试样例可以有效的寻找系统的错误，同时可以提高测试的覆盖率,使测试更加全面. 正确的测试样例可以有效地测试系统是否正确,而错误的测试样例则可以用来测试系统的健壮性. 按照如下的操作流程即可完整的测试该系统:

1. 创建一棵树
2. 使用定义{(0,a),(1,b),(2,c),(-1,NULL)*2,(4,d),(5,e),(-1,NULL)*2,(6,f),(-1,NULL)}创建这棵树
3. 使用-1 Debug 检查这棵树
4. 再创建一棵树
5. 使用定义{(7,g), (-1,NULL),(8,h), (-1,NULL),(9,i),(-1,NULL)*2}创建这棵树
6. 再创建一棵树
7. 使用定义{(10,h),(11,z),(12,y),(13,t),(-1,NULL)*5}创建这棵树
8. 使用-1 Debug 检查这三棵树
9. 如果所有树都创建正确，则可继续依据下面的操作测试
10. 使用 Write 命令写出所有树

11. 销毁 id 为树 1 的树
12. 重新加载文件，清空 id 为 1 的树
13. 判断 id 为 0、1 的树是否为空
14. 求取 id 为 0、1 的树的高度
15. 获取 id 为 0、1 的树的 Root
16. 获取 id 为 0 的树中 index 为 4、1000 的节点的 Value
17. 获取 id 为 1 的树中 index 为 10 的节点的 Value
18. 将 id 为 0 的树中 index 为 4、1000 的节点赋值为(4,q)、(1000,p)
19. 尝试为 id 为 1 的树中任意节点赋值
20. 获取 id 为 0、1 的树中某节点的 Parent
21. 获取 id 为 0、1 的树中某节点的 LeftChild
22. 获取 id 为 0、1 的树中某节点的 RightChild
23. 获取 id 为 0、1 的树中某节点的 LeftSibling
24. 获取 id 为 0、1 的树中某节点的 RightSibling
25. 将 id 为 2 的树插入到 id 为 0 的树的 index 为 1 的节点的右孩子上
26. 将 id 为 0 的树的 index 为 1 的节点的右孩子删除
27. 使用 Read 命令读取刚才写出的树
28. 使用四种方式遍历 id 为 0 的树

3.4.2 测试结果

创建一棵树：

图 0-3 InitBiTree 成功

由图可见，初始化线性表成功。

使用定义创建一棵二叉树：

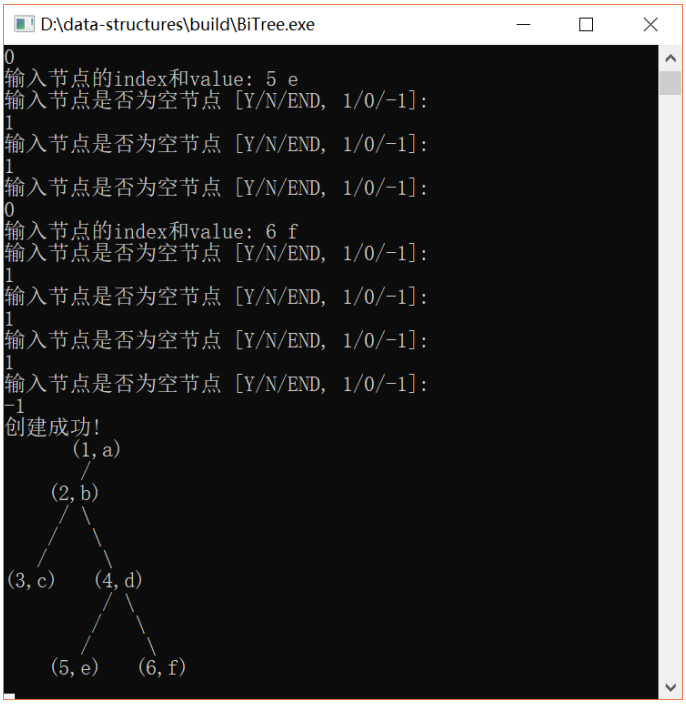


图 0-4 根据定义创建二叉树

使用-1 Debug 选项查看这可已经创建的二叉树：

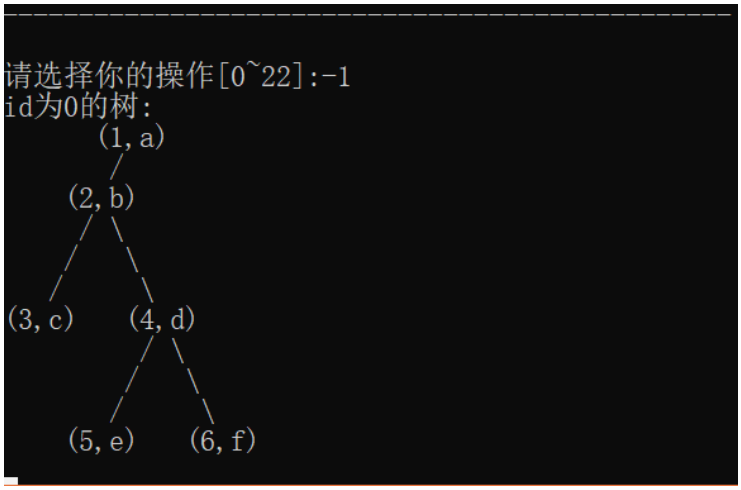


图 0-5 使用 Debug 选项查看这颗二叉树

类似的，创建三颗二叉树：

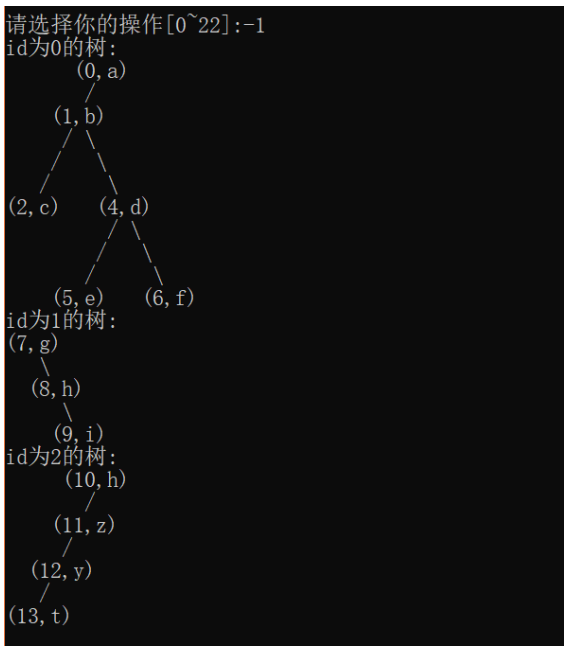


图 0-6 三棵二叉树

使用 Write 将三棵二叉树写出到文件，方便以后反复利用：

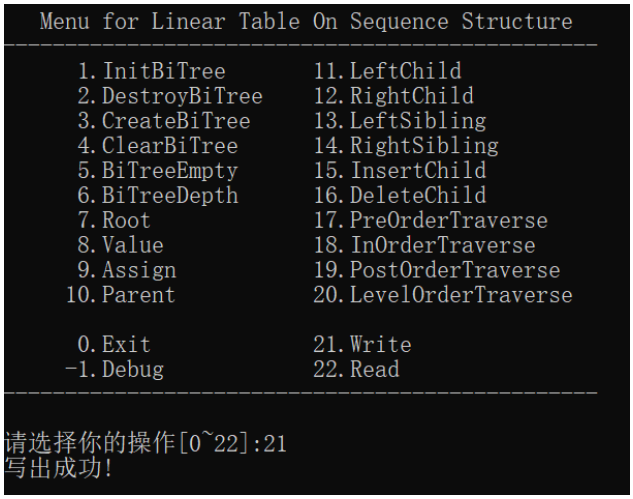


图 0-7 写出成功

使用 DestoryBiTree 销毁 id 为 1 的树：

图 0-8

销毁成功后，使用 Debug 选项查看所有的二叉树：

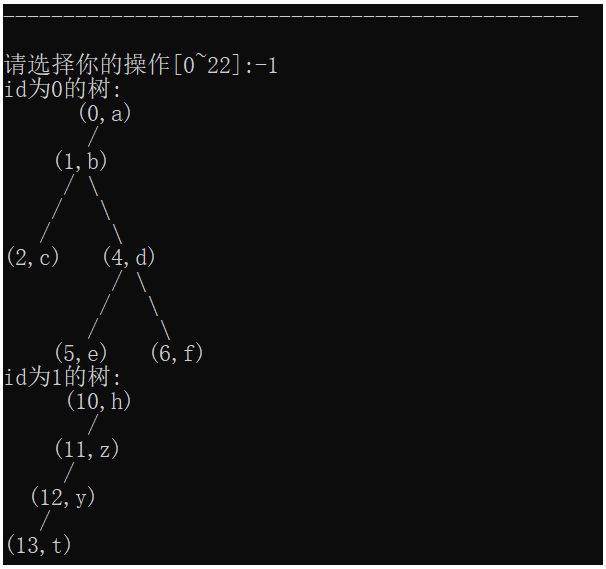


图 0-9 id 为 1 的树被删除

为了进行进一步测试，重新加载文件，将 id 为 1 的树清空。进行进一步测试。如下图所示：

图 0-10 清空 id 为 1 的树

为了验证清空是正确的，使用 Debug 选项再次输出所有的树：

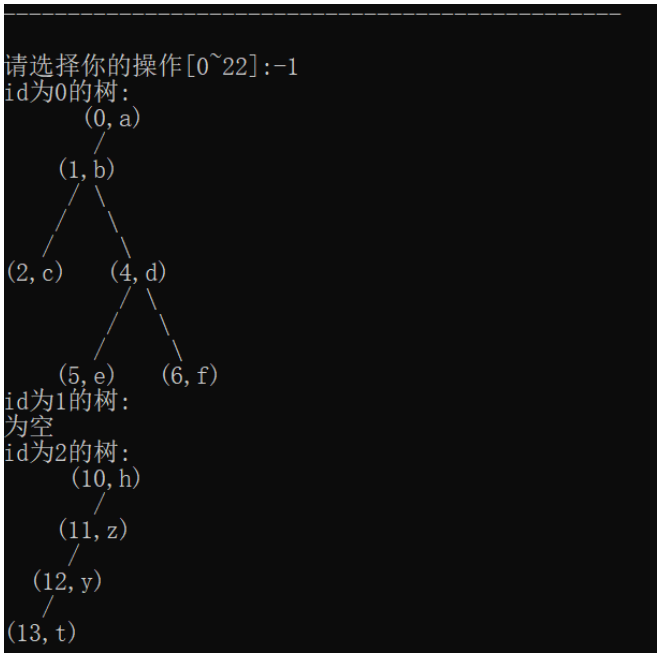
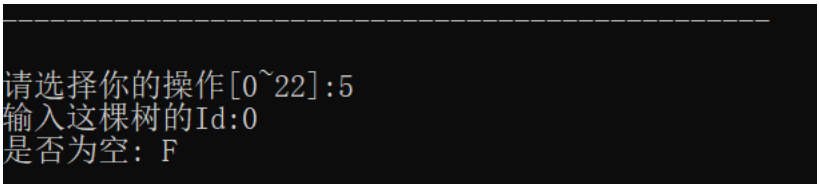


图 0-11 id 为 1 的树变为空

可见，id 为 1 的树被清空，变为了空树。为了进一步验证该系统的正确性，现在对该空树进行一系列的测试。



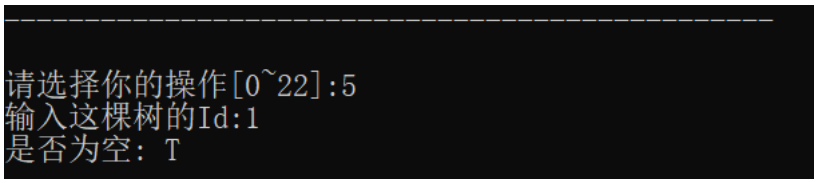


图 0-12 是否为空测试

由上图可知，是否为空的测试符合预期。

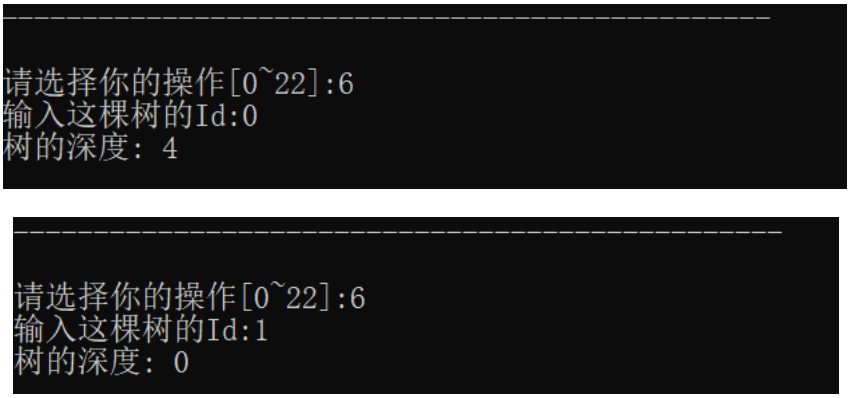


图 0-13 树的深度测试

由上图可知，求取树的深度的测试是成功的，符合预期。由于空树的深度显然为 0，该程序的行为是正常的。

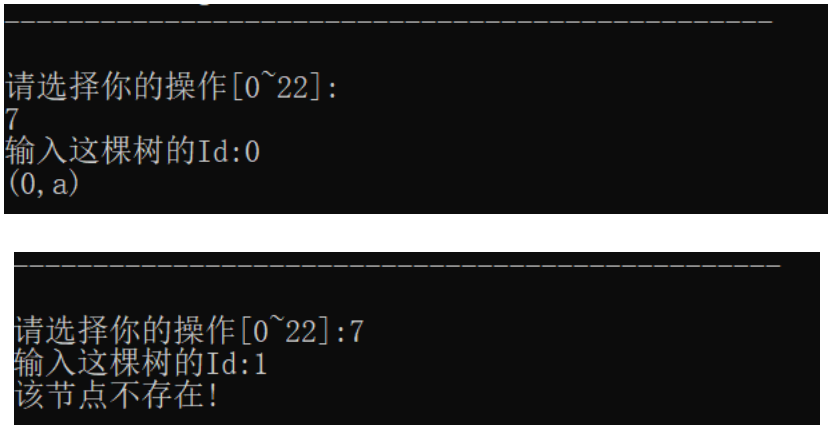


图 0-14 测试获取根节点的功能

由于 id 为 0 的树的根节点确实为(0,a)，而 id 为 1 的树由于是空树，其根节点必然为不存在。该程序的行为符合预期。

为了充分测试根据 index 获取 Value，使用 4 和 1000 对树 0 进行测试。

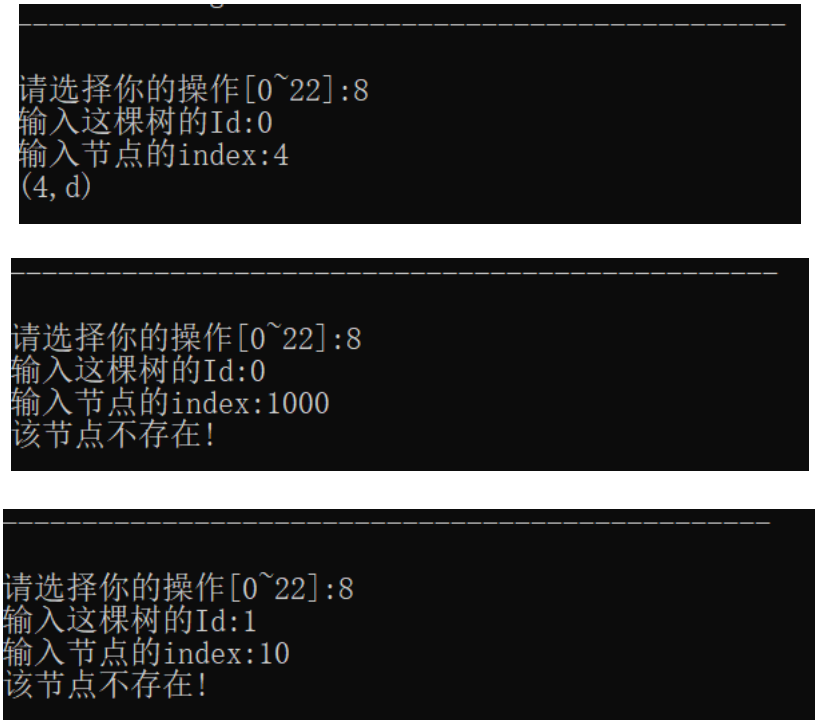


图 0-15 根据 index 获取 Value

对于存在的节点(4,d)，获取节点内容的结果符合预期。对于不存在的 index 1000，该程序输出节点不存在，同样符合预期。而对于空树 1 来说，由于任何节点都不存在，因此获取该 index 对应的节点输出该节点不存在，是合理的。

为了充分测试根据 index 赋值，使用 4 和 1000 对树 0 进行赋值测试。

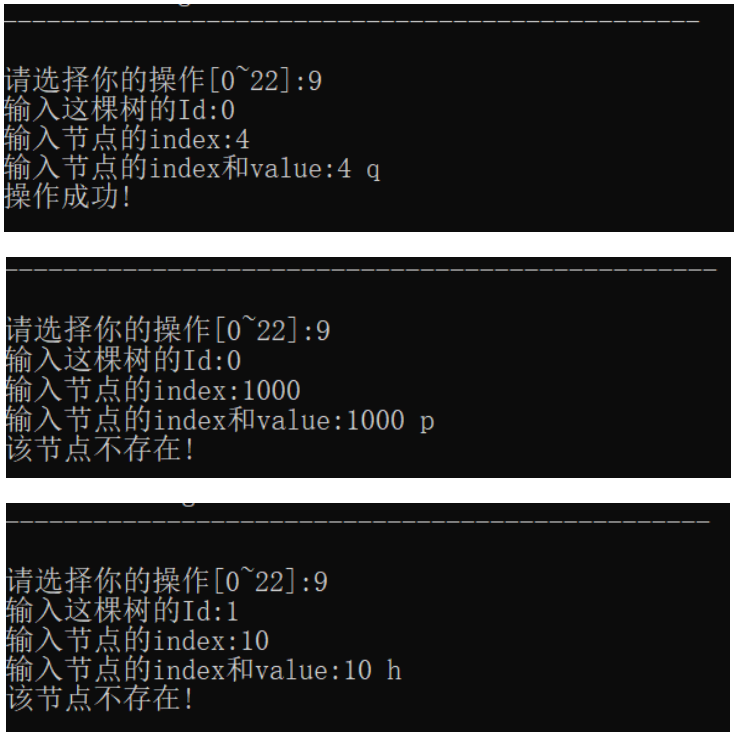


图 0-16 测试赋值

由于 index 为 4 的节点在树 0 中存在，对该节点的赋值导致操作成功。但是 index 为 1000 的节点和 index 为 10 的节点在树中不存在，因此赋值导致节点不存在的错误。该行为符合预期，而且是正确的。

为了进一步确认该赋值操作的正确性，使用-1 Debug 选项进行检查。

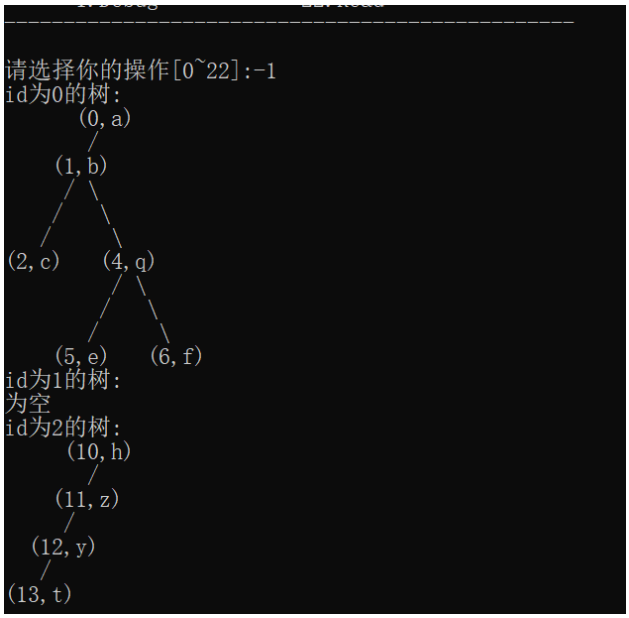


图 0-17 检查是否真的赋值

可见，index 为 4 的节点被赋值为了(4,q)，该行为正确而且符合预期。

为了测试获取双亲、左右儿子、左右兄弟节点等功能的正确性，使用 id 为 0 和 1 的树分别测试各功能：

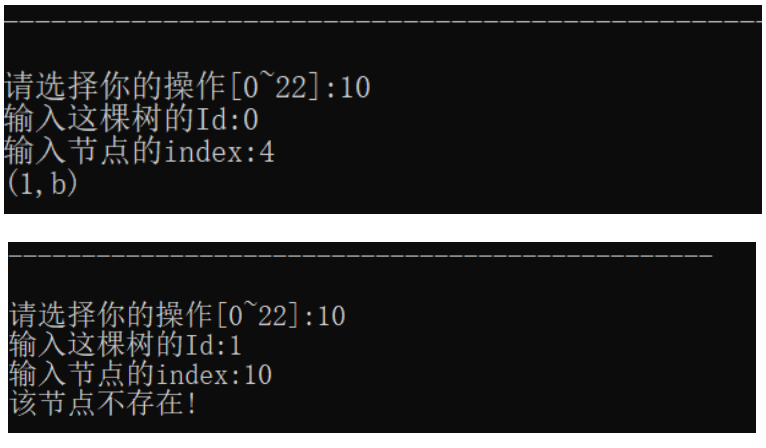
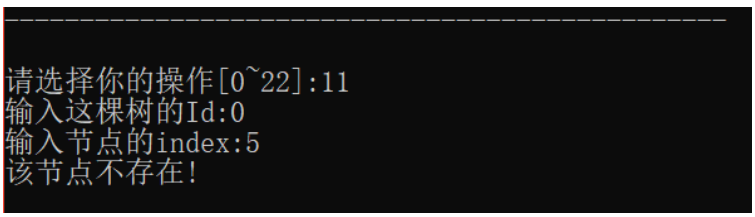
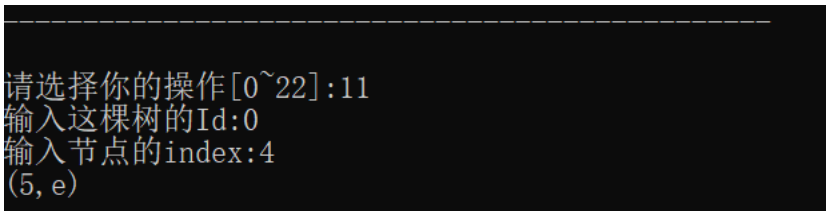


图 0-18 双亲结点测试





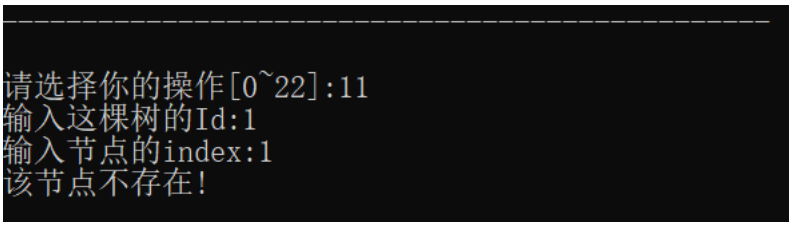
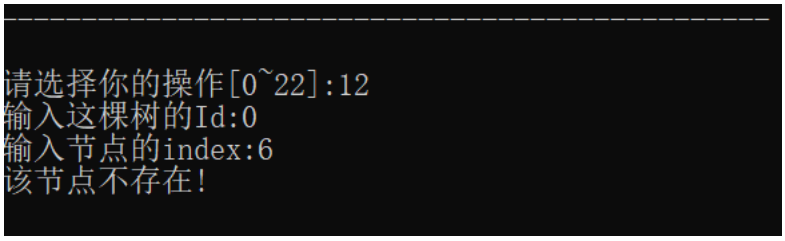


图 0-19 测试左儿子



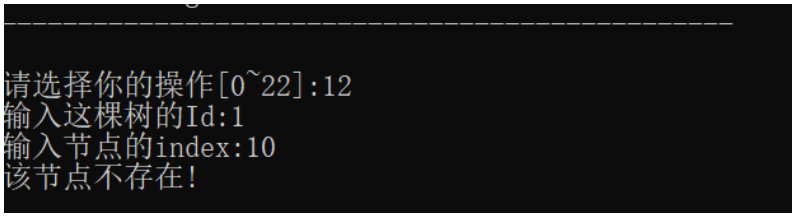
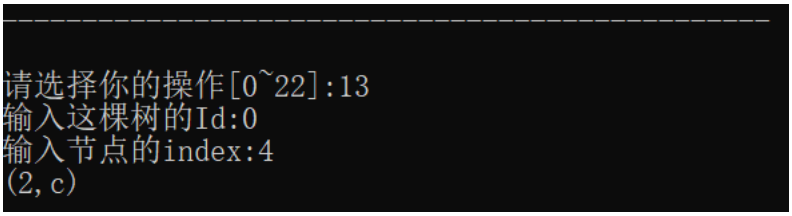


图 0-20 测试右儿子



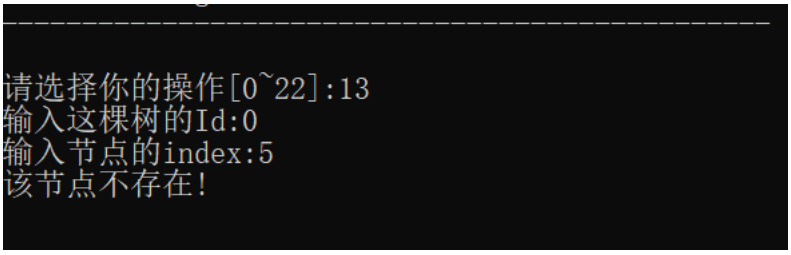


图 0-21 左兄弟

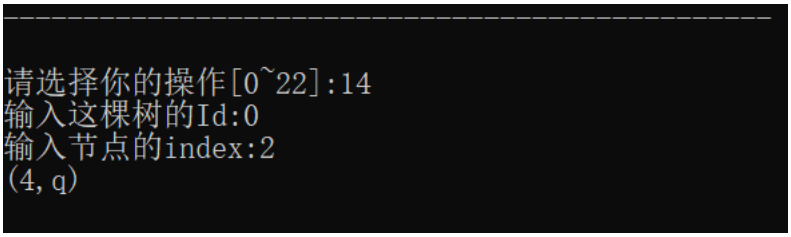
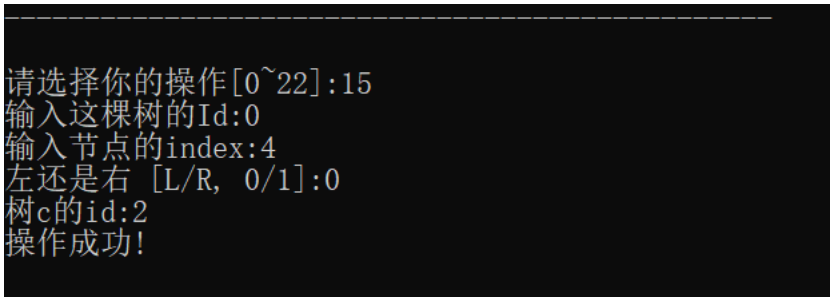


图 0-22 右兄弟

上面对获取双亲结点，左儿子，右儿子，左兄弟，右兄弟进行了测试，对于可以正确获得的节点的操作，输出了该节点的值。对于不存在的节点，输出了该节点不存在的错误提示。对于空树，由于节点不存在，因此直接提示节点不存在。所有行为都符合预期且正确。

为了测试插入和删除，我们使用 id 为 2 的树进行测试。



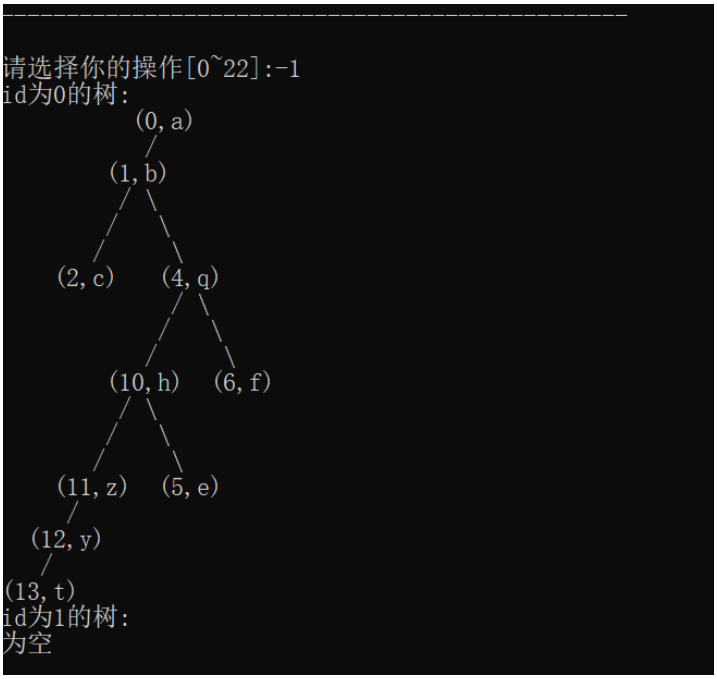
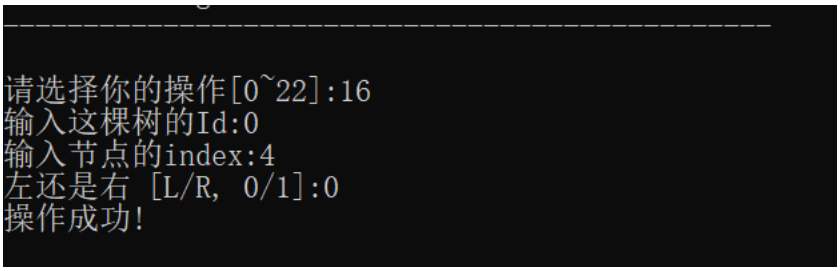


图 0-23 插入 InsertChild

如图所示，依次输入参数，即可完成该操作，如果任何一个参数不合法，则该插入行为不会被执行，不会进行任何修改动作。提示插入成功后，说明该操作已经完成，该操作的结果导致如上图的树形结构。该结构符合预期。

为了避免插入之后的树和被插入的树相交，树 c 被移除。

为了测试删除节点，我们将 index 为 4 的节点的左子树删除。



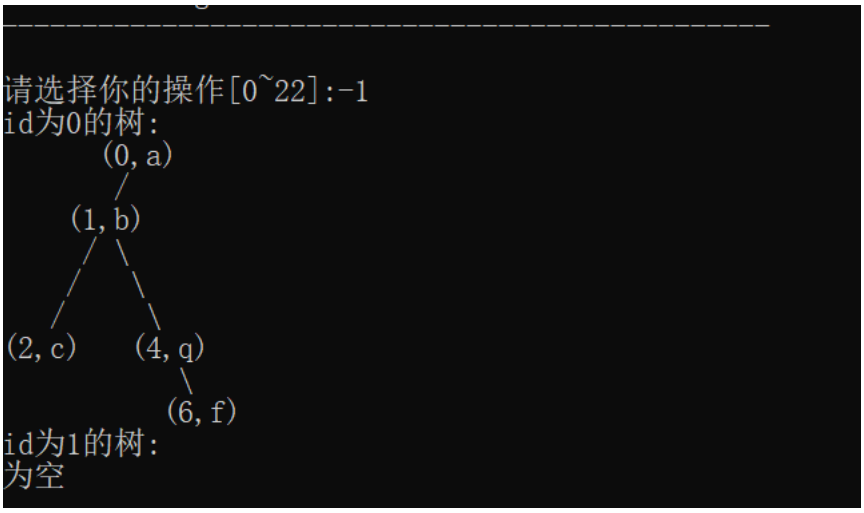
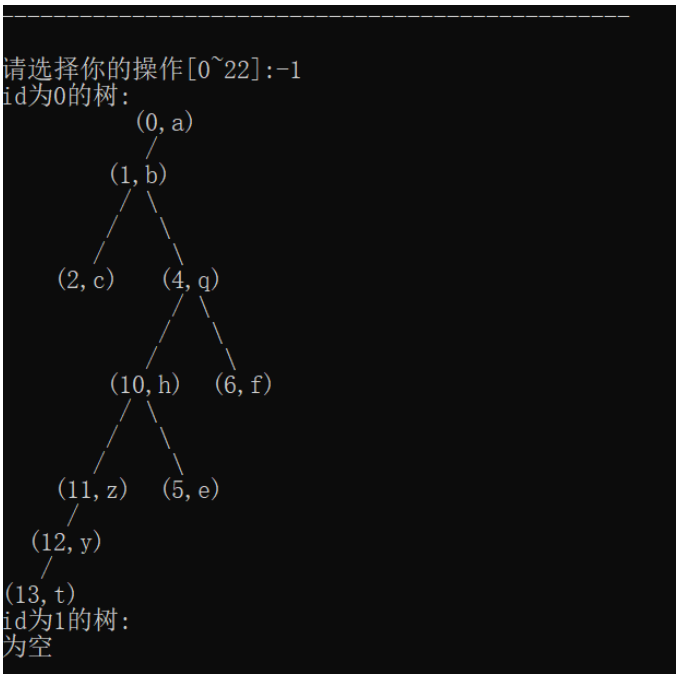


图 0-24 删除节点

删除之后的树的结构如上图，该行为正确。

为了测试四种遍历方式，我们重新加载插入节点之后的树。



如上图，已经从文件中读出了该树。现在使用四种遍历方式分别遍历该树。

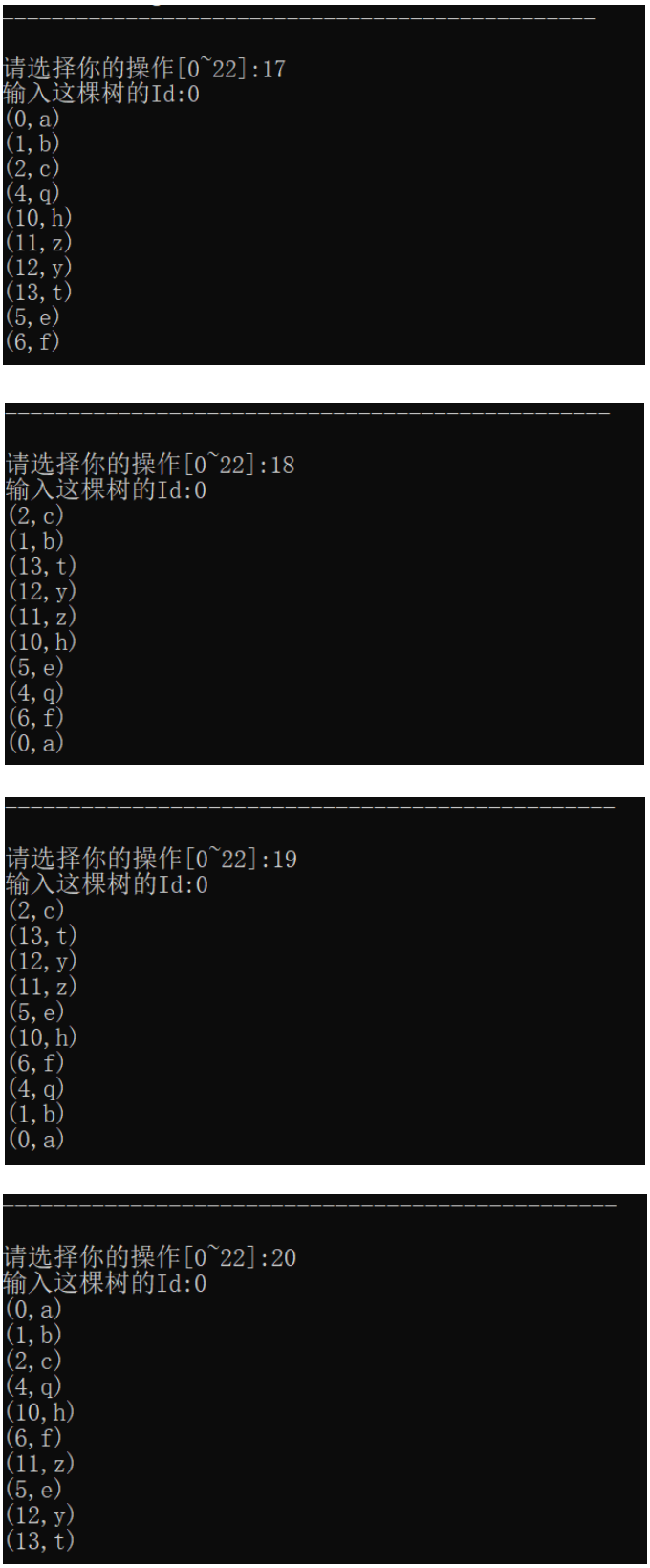


图 0-25 四种遍历方式的结果

经过认真检查，四种遍历方式的结果均正确。

3.5 实验小结

通过此次实验，我深刻的体会到了树这个数据结构的特点。在这次实验中，我使用 C 语言的风格封装了一个可以使用的二叉树。这次实验我知道了如何实现一个二叉树、如何将二叉树存到文件、如何从文件复原二叉树等。将课本的知识转化为可以使用的程序，同时我对 C 语言的错误处理、面向过程的编程有了更深的理解。通过对测试系统的实现，我学会了如何全面的测试我的程序，也知道了如何提高系统的鲁棒性。

4 基于邻接表的图实现

4.1 问题描述

4.2 系统设计

4.3 系统实现

4.4 系统测试

4.5 实验小结

参考文献

[1] 严蔚敏等. 数据结构(C 语言版). 清华大学出版社

[2] Larry Nyhoff. ADTs, Data Structures, and Problem Solving with C++. Second Edition, Calvin College, 2005

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

[4] 严蔚敏等.数据结构题集(C 语言版). 清华大学出版社

附录 A 基于顺序存储结构线性表实现的源程序

```
/* Linear Table On Sequence Structure */
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <vector>
using namespace std;

/*---------page 10 on textbook ---------*/
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASTABLE -1
#define OVERFLOW -2

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

/*-------page 22 on textbook -------*/
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10

struct SqList {  //顺序表（顺序结构）的定义
   ElemType* elem;
   size_t length;
   size_t listSize;
};

/*-----page 19 on textbook ---------*/
status InitList(SqList&);
status DestroyList(SqList&);
status ClearList(SqList&);
bool ListEmpty(SqList);
size_t ListLength(SqList);
status GetElem(SqList, size_t i, ElemType& e);
size_t LocateElem(SqList, ElemType e);
status PriorElem(SqList, ElemType cur, ElemType& pre_e);
status NextElem(SqList, ElemType cur, ElemType& next_e);
status ListInsert(SqList&, size_t i, ElemType e);
status ListDelete(SqList&, size_t i, ElemType& e);
status ListTraverse(SqList);  //简化过
/*-----------------------------------------*/
int main(void) {
   int i = 0;
   vector<SqList> Lists = {SqList()};
   int op = 1;
   while (op) {
```

```
system("cls");
printf("\n\n");
printf("Menu for Linear Table On Sequence Structure \n");
printf("-------------------------------------------------\n");
printf("          1. InitList              7. LocateElem\n");
printf("          2. DestroyList           8. PriorElem\n");
printf("          3. ClearList             9. NextElem \n");
printf("          4. ListEmpty             10. ListInsert\n");
printf("          5. ListLength            11. ListDelete\n");
printf("          6. GetElem               12. ListTraverse\n");
printf("          0. Exit                  13. ReadList\n");
printf("                                   14. WriteList\n");
printf("                                   15. AddList\n");
printf("                                   16. ChangeList\n");
printf("                                   17. CurrentList\n");
printf("-------------------------------------------------\n");
printf("请选择你的操作[0~17]:");
scanf("%d", &op);
switch (op) {
   case 1:
      //printf("\n----InitList 功能待实现！ \n");
      if (InitList(Lists[i]) == 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:
      DestroyList(Lists[i]);
      getchar();
      getchar();
      break;
   case 3:
      ClearList(Lists[i]);
      getchar();
      getchar();
      break;
   case 4:
      printf("列表是否为空：%s", ListEmpty(Lists[i]) ? "是" : "否");
      getchar();
      getchar();
      break;
   case 5:
      printf("列表长度：%d", ListLength(Lists[i]));
      getchar();
      getchar();
      break;
   case 6: {
```

```
        if (ListEmpty(Lists[i])) {
            printf("列表为空！");
            getchar();
            getchar();
            break;
        }
        int __i;
        printf("输入 index：\n");
        scanf("%u", &__i);
        ElemType e;
        GetElem(Lists[i], __i, e);
        printf("值：%d", e);
        getchar();
        getchar();
        break;
    }
    case 7: {
        ElemType e;
        scanf("%d", &e);
        printf("值：%d", LocateElem(Lists[i], e));
        getchar();
        getchar();
        break;
    }
    case 8: {
        ElemType e;
        ElemType p_e;
        printf("输入值：");
        scanf("%d", &e);
        PriorElem(Lists[i], e, p_e);
        printf("前驱：%d", p_e);
        getchar();
        getchar();
        break;
    }
    case 9: {
        ElemType e;
        ElemType n_e;
        printf("输入值：");
        scanf("%d", &e);
        NextElem(Lists[i], e, n_e);
        printf("后继：%d", n_e);
        getchar();
        getchar();
        break;
    }
    case 10: {
        ElemType e;
        size_t __i;
```

```
            printf("先后输入 index 和 item：\n");
            while (scanf("%u", &__i) && scanf("%d", &e) && __i <= Lists[i].length -
1) {
                if (ListInsert(Lists[i], __i, e))
                    printf("成功在%d 插入%d\n", __i, e);
                printf("先后输入 index 和 item：\n");
            }
            getchar();
            getchar();
            break;
        }
        case 11: {
            size_t __i;
            printf("输入 index：\n");
            while (scanf("%u", &__i)) {
                ElemType e;
                if (ListDelete(Lists[i], __i, e))
                    printf("删除的值为：%d\n", e);
                printf("输入 index：\n");
            }
            getchar();
            getchar();
            break;
        }
        case 12:
            if (!ListTraverse(Lists[i]))
                printf("线性表是空表！\n");
            getchar();
            getchar();
            break;
        case 13: {
            FILE* fp;
            char filename[30];
            printf("Input file name: ");
            scanf("%s", filename);

            Lists[i].length = 0;
            if ((fp = fopen(filename, "r")) == NULL) {
                printf("File open error\n");
                return 1;
            }
            while (fread(&Lists[i].elem[Lists[i].length], sizeof(ElemType), 1, fp))
                Lists[i].length++;
            break;
        }
        case 14: {
            FILE* fp;
            char filename[30];
            printf("Input file name: ");
```

```
            scanf("%s", filename);

            if ((fp = fopen(filename, "w")) == NULL) {
                printf("File open error\n");
                getchar();
                getchar();
                break;
            }
            fwrite(Lists[i].elem, sizeof(ElemType), Lists[i].length, fp);
            fclose(fp);
            getchar();
            getchar();
            break;
        }

        case 15: {
            SqList l;
            InitList(l);
            // l.elem = new int();
            Lists.push_back(l);
            printf("添加一张空表成功！\n");
            getchar();
            getchar();
            break;
        }
        case 16: {
            int __i;
            printf("输入表编号(0~%d):", Lists.size() - 1);
            while (!(scanf("%u", &__i) && (__i <= Lists.size() - 1))) {
                printf("输入表编号(0~%d):", Lists.size() - 1);
            }
            i = __i;
            printf("切换到表%d 成功！\n", __i);
            getchar();
            getchar();
            break;
        }
        case 17: {
            printf("当前是表%d\n", i);

            getchar();
            getchar();
            break;
        }

        case 0:
            break;
    }
}
```

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

/*--------page 23 on textbook --------------------*/
status InitList(SqList& L) {
    L.elem = (ElemType*)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (!L.elem)
        exit(OVERFLOW);
    L.length = 0;
    L.listSize = LIST_INIT_SIZE;
    return OK;
}
status ListTraverse(SqList L) {
    size_t 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) {
    free(L.elem);
    L.length = 0;
    return OK;
}
status ClearList(SqList& L) {
    free(L.elem);
    L.elem = (ElemType*)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (!L.elem)
        exit(OVERFLOW);
    L.length = 0;
    return OK;
}
bool ListEmpty(SqList L) {
    if (L.elem != nullptr)
        return L.length == 0;
    else
        return false;
}
size_t ListLength(SqList L) {
    if (L.elem != nullptr)
        return L.length;
    return -1;
}
status GetElem(SqList L, size_t i, ElemType& e) {
    if (L.elem != nullptr)
        if (i >= 0 && i < ListLength(L)) {
            e = L.elem[i];
            return OK;
```

```
        }
    return ERROR;
}
size_t LocateElem(
        SqList L,
        ElemType e)
// std::function<bool(ElemType, ElemType)> compare =
//     [](ElemType e1, ElemType e2) -> bool { return e1 == e2; })
{
    for (size_t i = 0; i < ListLength(L); i++) {
        ElemType _e;
        GetElem(L, i, _e);
        if (e == _e) {
            return i;
        }
    }
    return -1;
}
status PriorElem(SqList L, ElemType cur, ElemType& pre_e) {
    size_t l = LocateElem(L, cur);
    if (l != 0)
        return GetElem(L, l - 1, pre_e);
    return ERROR;
}
status NextElem(SqList L, ElemType cur, ElemType& next_e) {
    size_t l = LocateElem(L, cur);
    if (l != ListLength(L) - 1)
        return GetElem(L, l + 1, next_e);
    return ERROR;
}
status ListInsert(SqList& L, size_t i, ElemType e) {
    if (i < 0 || i >= L.length + 1) {
        return ERROR;
    }
    ElemType *p, *q;
    q = &(L.elem[i]);
    for (p = &(L.elem[L.length - 1]); p >= q; --p)
        *(p + 1) = *p;
    *q = e;
    ++L.length;
    return OK;
}

status ListDelete(SqList& L, size_t i, ElemType& e) {
    if (i < 0 || i >= L.length)
        return ERROR;

    int *p, *q;

    p = &(L.elem[i]);
```

```
    e = *p;
    q = L.elem + L.length - 1;
    for (++p; p <= q; ++p) {
        *(p - 1) = *p;
    }
    --L.length;
    return OK;
}
```

附录 B 基于链式存储结构线性表实现的源程序

```
/* Linear Table On Sequence Structure */
#include <stdio.h>
#include <stdlib.h>
#include <functional>
#include <string>
#include <vector>

/*---------page 10 on textbook ---------*/
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASTABLE -1
#define OVERFLOW -2

/*-------page 22 on textbook -------*/
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10

using namespace std;

typedef int status;

template <typename T>
struct Node {
   T data;
   Node* next;
};

template <typename T>
struct LinkedList {
   int length;
   bool init;
   Node<T>* head;
};

// using IntLinkedList = LinkedList<int>;

/*-----page 19 on textbook ---------*/
// template <typename T>
// status InitList(LinkedList<T>&);
// template <typename T>
// status DestroyList(LinkedList<T>&);
// template <typename T>
// status ClearList(LinkedList<T>&);
// template <typename T>
// bool ListEmpty(const LinkedList<T>);
// template <typename T>
```

```
// size_t ListLength(const LinkedList<T>);
// template <typename T>
// status GetElem(const LinkedList<T>, size_t, T&);
// template <typename T>
// size_t LocateElem(const LinkedList<T>, const T);
// template <typename T>
// status PriorElem(LinkedList<T>, T cur, T& pre_e);
// template <typename T>
// status NextElem(LinkedList<T>, T cur, T& next_e);
// template <typename T>
// status ListInsert(LinkedList<T>&, size_t i, T e);
// template <typename T>
// status ListDelete(LinkedList<T>&, size_t i, T& e);
// template <typename T>
// status ListTraverse(LinkedList<T>, function<void(Node<T>)>);

template <typename T>
status InitList(LinkedList<T>& L) {
   L.head = (Node<T>*)malloc(sizeof(Node<T>));
   if (L.head == NULL)
      return ERROR;
   L.head->next = NULL;
   L.length = 0;
   L.init = true;
   return OK;
}

template <typename T>
status DestroyList(LinkedList<T>& L) {
   Node<T>* tmp = L.head->next;
   Node<T>* tmp2 = tmp;
   while (tmp != NULL) {
      tmp2 = tmp->next;
      free(tmp);
      tmp = tmp2;
   }
   free(L.head);
   L.head = NULL;
   L.length = 0;
   L.init = false;
   return OK;
}

template <typename T>
status ClearList(LinkedList<T>& L) {
   Node<T>* tmp = L.head->next;
   Node<T>* tmp2 = tmp;
   while (tmp != NULL) {
      tmp2 = tmp->next;
      free(tmp);
```

```
        tmp = tmp2;
    }
    L.head->next = NULL;
    L.length = 0;
    return OK;
}

template <typename T>
bool ListEmpty(const LinkedList<T> L) {
    if (L.length == 0)
        return true;
    else
        return false;
}

template <typename T>
size_t ListLength(const LinkedList<T> L) {
    return L.length;
}

template <typename T>
status GetElem(const LinkedList<T> L, size_t i, T& e) {
    if (i < 0 || i > L.length - 1) {
        return ERROR;
    }
    Node<T>* ele = L.head->next;
    for (int j = 0; j < i; j++) {
        ele = ele->next;
    }
    e = ele->data;
    return OK;
}

template <typename T>
int LocateElem(const LinkedList<T> L, const T e) {
    Node<T>* ele = L.head;
    for (int i = 0; i < L.length; i++) {
        ele = ele->next;
        if (ele->data == e)
            return i;
    }
    return -1;
}

template <typename T>
status PriorElem(const LinkedList<T> L, const T cur_e, T& pre_e) {
    int loc = LocateElem(L, cur_e);
    if (loc == 0 || loc == -1)
        return ERROR;
    else {
```

```
        loc--;
        GetElem(L, loc, pre_e);
        return OK;
    }
}

template <typename T>
status NextElem(const LinkedList<T> L, const T cur_e, T& next_e) {
    int loc = LocateElem(L, cur_e);
    if (loc == L.length - 1 || loc == -1)
        return ERROR;
    else {
        loc++;
        GetElem(L, loc, next_e);
        return OK;
    }
}

template <typename T>
status ListInsert(LinkedList<T>& L, size_t i, T e) {
    if (i == ListLength(L) && i != 0) {
        Node<T>* ele = L.head;
        while (ele->next != NULL)
            ele = ele->next;

        ele->next = new Node<T>();
        ele->next->data = e;
        L.length++;
        return OK;
    }
    if (i < 0 || i > L.length)
        return ERROR;
    Node<T>* ele = L.head;
    for (int j = 0; j < i; j++) {
        ele = ele->next;
    }
    Node<T>* tmp = (Node<T>*)malloc(sizeof(Node<T>));
    tmp->data = e;
    tmp->next = ele->next;
    ele->next = tmp;
    L.length++;
    return OK;
}

template <typename T>
status ListDelete(LinkedList<T>& L, size_t i, T& e) {
    if (i < 0 || L.length == 0 || i > L.length - 1)
        return ERROR;
    L.length--;
    GetElem<T>(L, i, e);
```

```
    Node<T>* ele = L.head;
    for (int j = 0; j < i; j++) {
        ele = ele->next;
    }
    Node<T>* tmp = ele->next;
    ele->next = tmp->next;
    free(tmp);
    return OK;
}

template <typename T>
status ListTraverse(const LinkedList<T> L, function<void(Node<T>)> cb) {
    Node<T>* ele = L.head->next;
    while (ele != NULL) {
        cb(*ele);
        ele = ele->next;
    }
    return OK;
}

int main(void) {
    using ElemType = int;
    int i = 0;
    auto Lists = vector<LinkedList<ElemType>>{LinkedList<ElemType>()};
    int op = 1;
    while (op) {
        system("cls");
        printf("\n\n");
        printf("       Menu for Linear Table On Linked Structure       \n");
        printf("-------------------------------------------------\n");
        printf("            1. InitList              7. LocateElem            \n");
        printf("            2. DestroyList           8. PriorElem             \n");
        printf("            3. ClearList             9. NextElem              \n");
        printf("            4. ListEmpty             10. ListInsert           \n");
        printf("            5. ListLength            11. ListDelete           \n");
        printf("            6. GetElem               12. ListTraverse         \n");
        printf("            0. Exit                  13. ReadList             \n");
        printf("                                     14. WriteList            \n");
        printf("                                     15. AddList              \n");
        printf("                                     16. ChangeList           \n");
        printf("                                     17. CurrentList          \n");
        printf("-------------------------------------------------\n");
        printf("请选择你的操作[0~12]:");
        scanf("%d", &op);
        if (op != 1 && Lists[i].init == false) {
            printf("线性表未创建！\n");
            getchar();
            getchar();
            continue;
        }
```

```
switch (op) {
    case 1:
        //printf("\n----InitList 功能待实现！\n");
        if (InitList(Lists[i]) == 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:
        DestroyList(Lists[i]);
        getchar();
        getchar();
        break;
    case 3:
        ClearList(Lists[i]);
        getchar();
        getchar();
        break;
    case 4:
        printf("列表是否为空：%s", ListEmpty(Lists[i]) ? "是" : "否");
        getchar();
        getchar();
        break;
    case 5:
        printf("列表长度：%d", ListLength(Lists[i]));
        getchar();
        getchar();
        break;
    case 6: {
        if (ListEmpty(Lists[i])) {
            printf("列表为空！");
            getchar();
            getchar();
            break;
        }
        int __i;
        printf("输入 index：\n");
        scanf("%u", &__i);
        ElemType e;
        if (GetElem(Lists[i], __i, e) != OK) {
            printf("获取出错！\n");
        } else {
            printf("值：%d", e);
        }
        getchar();
        getchar();
        break;
```

```
}
case 7: {
    ElemType e;
    printf("输入值：\n");
    scanf("%d", &e);
    int index;
    if ((index = LocateElem(Lists[i], e)) == -1) {
        printf("未能找到！\n");
    } else {
        printf("Index: %d \n", index);
    }
    getchar();
    getchar();
    break;
}
case 8: {
    ElemType e;
    ElemType p_e;
    printf("输入值：");
    scanf("%d", &e);
    if (PriorElem(Lists[i], e, p_e) == OK) {
        printf("前驱：%d", p_e);
    } else {
        printf("失败！");
    }
    getchar();
    getchar();
    break;
}
case 9: {
    ElemType e;
    ElemType n_e;
    printf("输入值：");
    scanf("%d", &e);
    if (NextElem(Lists[i], e, n_e) == OK) {
        printf("后继：%d", n_e);
    } else {
        printf("失败！");
    }
    getchar();
    getchar();
    break;
}
case 10: {
    ElemType e;
    size_t __i;
    printf("先后输入 index 和 item：\n");
    while (scanf("%u %d", &__i, &e) && __i <= ListLength(Lists[i]) - 1) {
        if (ListInsert(Lists[i], __i, e) == OK)
```

```
            printf("成功在%d 插入%d\n", __i, e);
        else
            printf("未能插入任何元素！\n");
        printf("先后输入 index 和 item：\n");
    }
    getchar();
    getchar();
    break;
}
case 11: {
    int __i;
    printf("输入 index：\n");
    while (scanf("%d", &__i)) {
        ElemType e;
        if (ListDelete(Lists[i], __i, e) == OK)
            printf("删除的值为：%d\n", e);
        else
            printf("未能删除任何元素！\n");
        printf("输入 index：\n");
    }
    getchar();
    getchar();
    break;
}
case 12:
    if (!ListTraverse<ElemType>(Lists[i],
                                [](auto ele) -> void {
                                    printf("%d ", ele.data);
                                    return;
                                }))

        printf("线性表是空表！\n");
    getchar();
    getchar();
    break;
case 13: {
    FILE* fp;
    char filename[30];
    printf("Input file name: ");
    scanf("%s", filename);

    Lists[i].length = 0;
    if ((fp = fopen(filename, "r")) == NULL) {
        printf("File open error\n");
        getchar();
        getchar();
        break;
    }
    ElemType e;
```

```
    ClearList(Lists[i]);
    while (fread(&e, sizeof(ElemType), 1, fp)) {
        ListInsert(Lists[i], ListLength(Lists[i]), e);
    }

    fclose(fp);
    getchar();
    getchar();
    break;
}
case 14: {
    FILE* fp;
    char filename[30];
    printf("Input file name: ");
    scanf("%s", filename);

    if ((fp = fopen(filename, "w")) == NULL) {
        printf("File open error\n");
        getchar();
        getchar();
        break;
    }

    ListTraverse<ElemType>(
            Lists[i],
            [&](auto ele) -> void {
                fwrite(&ele.data, sizeof(ElemType), 1, fp);
            });

    fclose(fp);
    getchar();
    getchar();
    break;
}

case 15: {
    LinkedList<ElemType> l;
    InitList(l);
    // l.elem = new int();
    Lists.push_back(l);
    printf("添加一张空表成功！\n");
    getchar();
    getchar();
    break;
}
case 16: {
    int __i;
    printf("输入表编号(0~%d):", Lists.size() - 1);
    while (!(scanf("%u", &__i) && (__i <= Lists.size() - 1))) {
        printf("输入表编号(0~%d):", Lists.size() - 1);
```

```
            }
            i = __i;
            printf("切换到表%d 成功！\n", __i);
            getchar();
            getchar();
            break;
        }
        case 17: {
            printf("当前是表%d\n", i);

            getchar();
            getchar();
            break;
        }

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

附录 C 基于二叉链表二叉树实现的源程序

```
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <functional>
#include <queue>
#include <vector>
#define MAX_HEIGHT 1000
#define INFINITY (1 << 20)

using namespace std;
using status = void;
using ValueType = char;

struct ElemType {
    ValueType value;
    size_t index;
    bool null = true;
};

struct BiTreeNode {
    ElemType data;
    BiTreeNode* parent = NULL;
    BiTreeNode* left = NULL;
    BiTreeNode* right = NULL;
};

struct BiTree {
    bool init = false;
    BiTreeNode* root = NULL;
};

enum TraverseMethod {
    PRE,
    IN,
    POST,
    LEVEL
};

enum Error {
    INIT,
    NOT_INIT,
    WRONG_DEF,
```

```
    NO_SUCH_NODE,
};

status InitBiTree(BiTree& T);
status DestroyBiTree(BiTree& T);
status CreateBiTree(BiTree& T, vector<ElemType> def);
status ClearBiTree(BiTree& T);
bool BiTreeEmpty(const BiTree& T);
int BiTreeDepth(const BiTree& T);
BiTreeNode* Root(const BiTree& T);
status Value(const BiTree& T, size_t index, ElemType& value);
status Assign(BiTree& T, size_t index, ElemType& value);
BiTreeNode* Parent(const BiTree& T, size_t index);
BiTreeNode* LeftChild(const BiTree& T, size_t index);
BiTreeNode* RightChild(const BiTree& T, size_t index);
BiTreeNode* LeftSibling(const BiTree& T, size_t index);
BiTreeNode* RightSibling(const BiTree& T, size_t index);
status InsertChild(BiTree& T, size_t index, bool LR, BiTree& c);
status DeleteChild(BiTree& T, size_t index, bool LR);
status PreOrderTraverse(const BiTree& T, function<void(BiTreeNode*)>);
status InOrderTraverse(const BiTree& T, function<void(BiTreeNode*)>);
status PostOrderTraverse(const BiTree& T, function<void(BiTreeNode*)>);
status LevelOrderTraverse(const BiTree& T);

BiTreeNode* _Find(const BiTree& T, size_t index);
void _PrintNode(BiTreeNode* n);

status InitBiTree(BiTree& T) {
    if (T.init == false) {
        T.init = true;

    } else {
        throw Error::INIT;
    }
}

status DestroyBiTree(BiTree& T) {
    if (T.init != false) {
        ClearBiTree(T);
        T.init = false;
    } else {
        throw Error::NOT_INIT;
    }
}
```

```
status CreateBiTree(BiTree& T, vector<ElemType> def) {
    if (!T.init)
        throw Error::NOT_INIT;
    if (def.size() == 0)
        throw Error::WRONG_DEF;

    size_t i = 0;
    size_t l = def.size();

    function<void(BiTreeNode*&)> Create = [&](BiTreeNode*& n) -> void {
        if (i > l)
            return;
        ElemType data = def[i++];
        if (data.null == true)
            return;
        (n = new BiTreeNode())->data = data;
        Create(n->left);
        Create(n->right);
    };

    Create(T.root);
}

status ClearBiTree(BiTree& T) {
    if (!T.init)
        throw Error::NOT_INIT;

    PostOrderTraverse(T, [](BiTreeNode* pn) -> void {
        delete pn;
    });
    T.root = NULL;
}

status Value(const BiTree& T, size_t index, ElemType& value) {
    if (!T.init)
        throw Error::NOT_INIT;
    value = _Find(T, index)->data;
}

status Assign(BiTree& T, size_t index, ElemType& value) {
    if (!T.init)
        throw Error::NOT_INIT;
    _Find(T, index)->data = value;
```

```
}

BiTreeNode* _Find(const BiTree& T, size_t index) {
    if (!T.init)
        throw Error::NOT_INIT;
    bool stop = false;
    BiTreeNode* pRtn = NULL;
    PreOrderTraverse(T, [&](BiTreeNode* pn) -> void {
        if (stop == true)
            return;

        if (pn->data.index == index) {
            pRtn = pn;
            stop = true;
        }
    });

    if (pRtn == NULL) {
        throw Error::NO_SUCH_NODE;
    }
    return pRtn;
}

void _PrintNode(BiTreeNode* n) {
    if (!n)
        throw Error::NO_SUCH_NODE;
    printf("(%llu,%c)\n", n->data.index, n->data.value);
}

BiTreeNode* Parent(const BiTree& T, size_t index) {
    if (!T.init)
        throw Error::NOT_INIT;
    bool stop = false;
    BiTreeNode* pRtn = NULL;
    PreOrderTraverse(T, [&](BiTreeNode* pn) -> void {
        if (stop == true)
            return;

        if (
            (pn->left && pn->left->data.index == index) ||
            (pn->right && pn->right->data.index == index)) {
            pRtn = pn;
            stop = true;
        }
```

```
    });

    if (pRtn == NULL)
        throw Error::NO_SUCH_NODE;

    return pRtn;
}

BiTreeNode* LeftChild(const BiTree& T, size_t index) {
    return _Find(T, index)->left;
}

BiTreeNode* RightChild(const BiTree& T, size_t index) {
    return _Find(T, index)->right;
}

BiTreeNode* LeftSibling(const BiTree& T, size_t index) {
    BiTreeNode* parent = Parent(T, index);
    BiTreeNode* self = _Find(T, index);
    return parent->left == self ? NULL : parent->left;
}

BiTreeNode* RightSibling(const BiTree& T, size_t index) {
    BiTreeNode* parent = Parent(T, index);
    BiTreeNode* self = _Find(T, index);
    return parent->right == self ? NULL : parent->right;
}

BiTreeNode* Root(const BiTree& T) {
    if (!T.init)
        throw Error::NOT_INIT;
    return T.root;
}

bool BiTreeEmpty(const BiTree& T) {
    return Root(T) == NULL;
}

int BiTreeDepth(const BiTree& T) {
    if (!T.init)
        throw Error::NOT_INIT;
    function<int(BiTreeNode*)> Depth = [&](BiTreeNode* root) -> int {
        if (root == NULL)
            return 0;
```

```
        int depthLeft = Depth(root->left);
        int depthRight = Depth(root->right);
        int depth = depthLeft > depthRight ? depthLeft : depthRight;
        return depth + 1;
    };

    return Depth(T.root);
}

status InsertChild(BiTree& T, size_t index, bool LR, BiTree& c) {
    BiTreeNode* self = _Find(T, index);
    BiTreeNode* original = NULL;
    if (LR == true) {
        original = self->left;
        self->left = c.root;
        c.root->right = original;
    } else {
        original = self->right;
        self->right = c.root;
        c.root->left = original;
    }
}

status DeleteChild(BiTree& T, size_t index, bool LR) {
    if (!T.init)
        throw Error::NOT_INIT;
    function<void(BiTreeNode*&)> Traverse = [&](BiTreeNode*& root) -> void {
        if (root == NULL || root->data.null == true)
            return;
        Traverse(root->left);
        Traverse(root->right);
        delete root;
        root = NULL;
    };
    if (LR == true) {
        Traverse(_Find(T, index)->left);
    } else {
        Traverse(_Find(T, index)->right);
    }
}

status PreOrderTraverse(const BiTree& T, function<void(BiTreeNode*)> Visit) {
    if (!T.init)
        throw Error::NOT_INIT;
```

```
    function<void(BiTreeNode*)> Traverse = [&](BiTreeNode* root) -> void {
        if (root == NULL)
            return;
        Visit(root);
        Traverse(root->left);
        Traverse(root->right);
    };

    Traverse(T.root);
}

status InOrderTraverse(const BiTree& T, function<void(BiTreeNode*)> Visit) {
    if (!T.init)
        throw Error::NOT_INIT;
    function<void(BiTreeNode*)> Traverse = [&](BiTreeNode* root) -> void {
        if (root == NULL)
            return;
        Traverse(root->left);
        Visit(root);
        Traverse(root->right);
    };

    Traverse(T.root);
}

status PostOrderTraverse(const BiTree& T, function<void(BiTreeNode*)> Visit) {
    if (!T.init)
        throw Error::NOT_INIT;
    function<void(BiTreeNode*)> Traverse = [&](BiTreeNode* root) -> void {
        if (root == NULL)
            return;
        Traverse(root->left);
        Traverse(root->right);
        Visit(root);
    };

    Traverse(T.root);
}

status LevelOrderTraverse(const BiTree& T, function<void(BiTreeNode*)> Visit) {
    if (!T.init)
        throw Error::NOT_INIT;
    queue<BiTreeNode*> q;
    q.push(T.root);
```

```
    while (q.size() != 0) {
        BiTreeNode* n = q.front();
        Visit(n);
        q.pop();
        if (n->left != NULL)
            q.push(n->left);
        if (n->right != NULL)
            q.push(n->right);
    }
}

struct asciinode {
    asciinode *left, *right;
    int edge_length;
    int height;
    int lablen;
    int parent_dir;
    char label[11];
};

int lprofile[MAX_HEIGHT];
int rprofile[MAX_HEIGHT];
int gap = 3;
int print_next;

int MIN(int X, int Y) {
    return ((X) < (Y)) ? (X) : (Y);
}

int MAX(int X, int Y) {
    return ((X) > (Y)) ? (X) : (Y);
}

asciinode* build_ascii_tree_recursive(BiTreeNode* t) {
    asciinode* node;

    if (t == NULL || t->data.null == true)
        return NULL;

    node = (asciinode*)malloc(sizeof(asciinode));
    node->left = build_ascii_tree_recursive(t->left);
    node->right = build_ascii_tree_recursive(t->right);

    if (node->left != NULL) {
```

```
        node->left->parent_dir = -1;
    }

    if (node->right != NULL) {
        node->right->parent_dir = 1;
    }

    sprintf(node->label, "(%llu,%c)", t->data.index, t->data.value);
    node->lablen = strlen(node->label);

    return node;
}

asciinode* build_ascii_tree(BiTreeNode* t) {
    asciinode* node;
    if (t == NULL)
        return NULL;
    node = build_ascii_tree_recursive(t);
    node->parent_dir = 0;
    return node;
}

void free_ascii_tree(asciinode* node) {
    if (node == NULL)
        return;
    free_ascii_tree(node->left);
    free_ascii_tree(node->right);
    free(node);
}

void compute_lprofile(asciinode* node, int x, int y) {
    int i, isleft;
    if (node == NULL)
        return;
    isleft = (node->parent_dir == -1);
    lprofile[y] = MIN(lprofile[y], x - ((node->lablen - isleft) / 2));
    if (node->left != NULL) {
        for (i = 1; i <= node->edge_length && y + i < MAX_HEIGHT; i++) {
            lprofile[y + i] = MIN(lprofile[y + i], x - i);
        }
    }
    compute_lprofile(node->left, x - node->edge_length - 1, y + node->edge_length +
1);
    compute_lprofile(node->right, x + node->edge_length + 1, y + node->edge_length +
```

```
1);
}

void compute_rprofile(asciinode* node, int x, int y) {
   int i, notleft;
   if (node == NULL)
      return;
   notleft = (node->parent_dir != -1);
   rprofile[y] = MAX(rprofile[y], x + ((node->lablen - notleft) / 2));
   if (node->right != NULL) {
      for (i = 1; i <= node->edge_length && y + i < MAX_HEIGHT; i++) {
         rprofile[y + i] = MAX(rprofile[y + i], x + i);
      }
   }
   compute_rprofile(node->left, x - node->edge_length - 1, y + node->edge_length +
1);
   compute_rprofile(node->right, x + node->edge_length + 1, y + node->edge_length
+ 1);
}

void compute_edge_lengths(asciinode* node) {
   int h, hmin, i, delta;
   if (node == NULL)
      return;
   compute_edge_lengths(node->left);
   compute_edge_lengths(node->right);

   if (node->right == NULL && node->left == NULL) {
      node->edge_length = 0;
   } else {
      if (node->left != NULL) {
         for (i = 0; i < node->left->height && i < MAX_HEIGHT; i++) {
            rprofile[i] = -INFINITY;
         }
         compute_rprofile(node->left, 0, 0);
         hmin = node->left->height;
      } else {
         hmin = 0;
      }
      if (node->right != NULL) {
         for (i = 0; i < node->right->height && i < MAX_HEIGHT; i++) {
            lprofile[i] = INFINITY;
         }
         compute_lprofile(node->right, 0, 0);
```

```
            hmin = MIN(node->right->height, hmin);
        } else {
            hmin = 0;
        }
        delta = 4;
        for (i = 0; i < hmin; i++) {
            delta = MAX(delta, gap + 1 + rprofile[i] - lprofile[i]);
        }
        if (((node->left != NULL && node->left->height == 1) ||
                (node->right != NULL && node->right->height == 1)) &&
               delta > 4) {
            delta--;
        }
        node->edge_length = ((delta + 1) / 2) - 1;
    }
    h = 1;
    if (node->left != NULL) {
        h = MAX(node->left->height + node->edge_length + 1, h);
    }
    if (node->right != NULL) {
        h = MAX(node->right->height + node->edge_length + 1, h);
    }
    node->height = h;
}

void print_level(asciinode* node, int x, int level) {
    int i, isleft;
    if (node == NULL)
        return;
    isleft = (node->parent_dir == -1);
    if (level == 0) {
        for (i = 0; i < (x - print_next - ((node->lablen - isleft) / 2)); i++) {
            printf(" ");
        }
        print_next += i;
        printf("%s", node->label);
        print_next += node->lablen;
    } else if (node->edge_length >= level) {
        if (node->left != NULL) {
            for (i = 0; i < (x - print_next - (level)); i++) {
                printf(" ");
            }
            print_next += i;
            printf("/");
```

```
            print_next++;
        }
        if (node->right != NULL) {
            for (i = 0; i < (x - print_next + (level)); i++) {
                printf(" ");
            }
            print_next += i;
            printf("\\");
            print_next++;
        }
    } else {
        print_level(node->left,
                    x - node->edge_length - 1,
                    level - node->edge_length - 1);
        print_level(node->right,
                    x + node->edge_length + 1,
                    level - node->edge_length - 1);
    }
}

void print_ascii_tree(BiTreeNode* t) {
    asciinode* proot;
    int xmin, i;
    if (t == NULL)
        return;
    proot = build_ascii_tree(t);
    compute_edge_lengths(proot);
    for (i = 0; i < proot->height && i < MAX_HEIGHT; i++) {
        lprofile[i] = INFINITY;
    }
    compute_lprofile(proot, 0, 0);
    xmin = 0;
    for (i = 0; i < proot->height && i < MAX_HEIGHT; i++) {
        xmin = MIN(xmin, lprofile[i]);
    }
    for (i = 0; i < proot->height; i++) {
        print_next = 0;
        print_level(proot, -xmin, i);
        printf("\n");
    }
    if (proot->height >= MAX_HEIGHT) {
        printf("(This tree is taller than %d, and may be drawn incorrectly.)\n",
MAX_HEIGHT);
    }
```

```
    free_ascii_tree(proot);
}

int main() {
    int selection = -1;
    size_t I = -1;
    size_t index = -1;
    vector<BiTree> trees = {};
    while (selection != 0) {
        system("cls");
        printf("\n");
        printf("      Menu for Linear Table On Sequence Structure    \n");
        printf("-------------------------------------------------\n");
        printf("          1.InitBiTree            11.LeftChild              \n");
        printf("          2.DestroyBiTree         12.RightChild              \n");
        printf("          3.CreateBiTree          13.LeftSibling            \n");
        printf("          4.ClearBiTree           14.RightSibling           \n");
        printf("          5.BiTreeEmpty            15.InsertChild            \n");
        printf("          6.BiTreeDepth            16.DeleteChild            \n");
        printf("          7.Root                   17.PreOrderTraverse       \n");
        printf("          8.Value                  18.InOrderTraverse        \n");
        printf("          9.Assign                 19.PostOrderTraverse      \n");
        printf("         10.Parent                 20.LevelOrderTraverse     \n");
        printf("                                                               \n");
        printf("          0.Exit                   21.Write                  \n");
        printf("         -1.Debug                   22.Read                  \n");
        printf("-------------------------------------------------\n");
        printf("\n");
        printf("请选择你的操作[0~22]:");
        scanf("%d", &selection);
        try {
            switch (selection) {
                case -1:
                    I = 0;
                    for (auto tree : trees) {
                        printf("id 为%llu 的树:\n", I++);
                        if (!BiTreeEmpty(tree))
                            print_ascii_tree(tree.root);
                        else
                            printf("为空\n");
                    }
                    break;
                case 1:
                    trees.push_back(BiTree());
```

```
            InitBiTree(trees.back());
            printf("创建成功!当前 id 范围:[0, %llu]\n", trees.size() - 1);
            break;
        case 21: {
            size_t tree_s = trees.size();
            if (tree_s <= 0) {
                printf("没有树!\n");
                break;
            }
            FILE* fp = fopen("trees", "w");
            fwrite(&tree_s, sizeof(size_t), 1, fp);
            for (auto tree : trees) {
                vector<ElemType> elems = {};
                function<void(BiTreeNode*)> Traverse = [&](BiTreeNode* root) ->
void {
                    if (root == NULL) {
                        elems.push_back(ElemType());
                        return;
                    }

                    elems.push_back(root->data);
                    Traverse(root->left);
                    Traverse(root->right);
                };

                Traverse(tree.root);
                size_t s = elems.size();
                fwrite(&s, sizeof(size_t), 1, fp);
                for (auto elem : elems) {
                    fwrite(&elem, sizeof(ElemType), 1, fp);
                }
            }
            fclose(fp);
            printf("写出成功!\n");
            break;
        }
        case 22: {
            trees = {};
            FILE* fp = fopen("trees", "r");
            size_t tree_s;
            fread(&tree_s, sizeof(size_t), 1, fp);
            for (size_t i = 0; i < tree_s; i++) {
                size_t s;
                fread(&s, sizeof(size_t), 1, fp);
```

```
            vector<ElemType> elems = {};
            for (size_t i = 0; i < s; i++) {
                ElemType* elem = new ElemType();
                fread(elem, sizeof(ElemType), 1, fp);
                elems.push_back(*elem);
            }
            BiTree T;
            InitBiTree(T);
            CreateBiTree(T, elems);
            trees.push_back(T);
        }
        fclose(fp);
        printf("读取成功!\n");
        break;
    }
    case 0:
        printf("欢迎下次再使用本系统！\n");
        break;
    default:
        if (selection >= 2 && selection <= 20) {
            printf("输入这棵树的 Id:");
            if (scanf("%llu", &I) != 0) {
                if (I <= trees.size() - 1) {
                    switch (selection) {
                        case 2:
                            DestroyBiTree(trees[I]);
                            trees.erase(trees.begin() + I);
                            printf("删除成功!\n");
                            break;
                        case 3: {
                            vector<ElemType> elems = {};
                            while (true) {
                                int null;
                                char value;
                                size_t index;
                                printf("输入节点是否为空节点 [Y/N/END, 1/0/-1]:\n");
                                if (scanf("%d", &null) != 0 && (null == 0 || null == 1)) {
                                    if (null == 0) {
                                        printf("输入节点的 index 和 value: ");
                                        if (scanf("%llu %c", &index, &value) != 0) {
                                            elems.push_back({value, index, null == 1});
                                        }
                                    } else {
                                        elems.push_back(ElemType());
```

```
                }
            } else
                break;
        }
        CreateBiTree(trees[I], elems);
        printf("创建成功!\n");
        print_ascii_tree(trees[I].root);
        break;
    }
    case 4:
        ClearBiTree(trees[I]);
        printf("清空成功!\n");
        break;
    case 5:
        printf("是否为空: %c\n", BiTreeEmpty(trees[I]) == true ? 'T' :
'F');
        break;
    case 6:
        printf("树的深度: %d\n", BiTreeDepth(trees[I]));
        break;
    case 7:
        _PrintNode(Root(trees[I]));
        break;
    case 9:
        printf("输入节点的 index:");
        if (scanf("%llu", &index) != 0) {
            printf("输入节点的 index 和 value:");
            size_t newIndex;
            char value;
            if (scanf("%llu %c", &newIndex, &value) != 0) {
                ElemType elem = {value, newIndex, false};
                Assign(trees[I], index, elem);
                printf("操作成功!\n");
                break;
            }
        }
    case 8:
        printf("输入节点的 index:");
        if (scanf("%llu", &index) != 0) {
            _PrintNode(_Find(trees[I], index));
            break;
        }
    case 10:
    case 11:
```

```
case 12:
case 13:
case 14:
    printf("输入节点的 index:");
    if (scanf("%llu", &index) != 0) {
        switch (selection) {
            case 10:
                _PrintNode(Parent(trees[I], index));
                break;
            case 11:
                _PrintNode(LeftChild(trees[I], index));
                break;
            case 12:
                _PrintNode(RightChild(trees[I], index));
                break;
            case 13:
                _PrintNode(LeftSibling(trees[I], index));
                break;
            case 14:
                _PrintNode(RightSibling(trees[I], index));
                break;
        }
        break;
    }
case 15:
case 16:
    printf("输入节点的 index:");
    if (scanf("%llu", &index) != 0) {
        printf("左还是右  [L/R, 0/1]:");
        int LR;
        if (scanf("%d", &LR) != 0) {
            if (selection == 15) {
                printf("树 c 的 id:");
                size_t c_tree;
                if (scanf("%llu", &c_tree) != 0) {
                    InsertChild(trees[I], index, LR == 0, trees[c_tree]);
                    trees.erase(trees.begin() + c_tree);
                    printf("操作成功!\n");
                    break;
                }
            } else {
                DeleteChild(trees[I], index, LR == 0);
                printf("操作成功!\n");
                break;
```

```
                        }
                    }
                }
                case 17:
                    PreOrderTraverse(trees[I], _PrintNode);
                    break;
                case 18:
                    InOrderTraverse(trees[I], _PrintNode);
                    break;
                case 19:
                    PostOrderTraverse(trees[I], _PrintNode);
                    break;
                case 20:
                    LevelOrderTraverse(trees[I], _PrintNode);
                    break;
                }
                break;
            }
            }
            printf("无效输入\n");
        }
        break;
    }
  } catch (Error e) {
    switch (e) {
      case Error::NO_SUCH_NODE:
        printf("该节点不存在!\n");
        break;
      case Error::WRONG_DEF:
        printf("definition 错误!\n");
        break;
      case Error::NOT_INIT:
        printf("未初始化!\n");
        break;
      default:
        printf("未知错误!\n");
        break;
    }
  }
  getchar();
  getchar();
 }
}
```

附录 D 基于邻接表图实现的源程序