

# 华中科技大学

## 课程实验报告

课程名称： 数据结构实验

专业班级： 计算机科学与技术 201706 班

学 号： U201714761

姓 名： 胡澳

指导教师： 周时阳

报告日期： 2018 年 12 月 30 日

计算机科学与技术学院

## 目 录

<b>1 基于顺序存储结构的线性表实现</b>	<b>1</b>
1.1 问题描述	1
1.2 系统设计	2
1.3 系统实现	10
1.4 实验小结	24
<b>2 基于链式存储结构的线性表实现</b>	<b>25</b>
2.1 问题描述	25
2.2 系统设计	26
2.3 系统实现	33
2.4 实验小结	47
<b>3 基于二叉链表的二叉树实现</b>	<b>48</b>
3.1 问题描述	48
3.2 系统设计	50
3.3 系统实现	63
3.4 实验小结	87
<b>4 基于邻接表的有向网实现</b>	<b>89</b>
4.1 问题描述	89
4.2 系统设计	91
4.3 系统实现	101
4.4 实验小结	118
<b>参考文献</b>	<b>119</b>

# 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  $n$  个数据元素的有限序列。线性表的顺序表示指的是用一组地址连续的存储单元依次存储线性表的数据元素。

本实验要求利用线性表的顺序存储结构以函数方式实现线性表 12 种基本操作。这 12 中基本操作如下。

1) 初始化表：函数名称是 `InitList(L)`；初始条件是线性表  $L$  不存在已存在；操作结果是构造一个空的线性表。

2) 销毁表：函数名称是 `DestroyList(L)`；初始条件是线性表  $L$  已存在；操作结果是销毁线性表  $L$ 。

3) 清空表：函数名称是 `ClearList(L)`；初始条件是线性表  $L$  已存在；操作结果是将  $L$  重置为空表。

4) 判定空表：函数名称是 `ListEmpty(L)`；初始条件是线性表  $L$  已存在；操作结果是若  $L$  为空表则返回 `TRUE`, 否则返回 `FALSE`。

5) 求表长：函数名称是 `ListLength(L)`；初始条件是线性表已存在；操作结果是返回  $L$  中数据元素的个数。

6) 获得元素：函数名称是 `GetElem(L, i, e)`；初始条件是线性表已存在,  $1 \leq i \leq \text{ListLength}(L)$ ；操作结果是用  $e$  返回  $L$  中第  $i$  个数据元素的值。

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

8) 获得前驱：函数名称是 `PriorElem(L, cur_e, pre_e)`；初始条件是线性表  $L$  已存在；操作结果是若  $cur\_e$  是  $L$  的数据元素，且不是第一个，则用  $pre\_e$  返回它的前驱，否则操作失败， $pre\_e$  无定义。

9) 获得后继：函数名称是 `NextElem(L, cur_e, next_e)`；初始条件是线性表  $L$  已存在；操作结果是若  $cur\_e$  是  $L$  的数据元素，且不是最后一个，则用  $next\_e$

返回它的后继，否则操作失败，next\_e 无定义。

10) 插入元素：函数名称是 ListInsert(L, i, e)；初始条件是线性表 L 已存在且非空， $1 \leq i \leq \text{ListLength}(L)+1$ ；操作结果是在 L 的第 i 个位置之前插入新的数据元素 e。

11) 删除元素：函数名称是 ListDelete(L, i, e)；初始条件是线性表 L 已存在且非空， $1 \leq i \leq \text{ListLength}(L)$ ；操作结果：删除 L 的第 i 个数据元素，用 e 返回其值。

12) 遍历表：函数名称是 ListTraverse(L)，初始条件是线性表 L 已存在；操作结果是依次输出 L 的每个数据元素。

本系统还实现了对多个线性表操作的支持。

本系统提供对线性表的文件输入输出功能。本功能通过两个函数实现，两个函数的基本情况如下。

1) 保存到文件：函数名称为 LoadToFile(L, filename)；初始条件是线性表 L 已存在，操作结果是将线性表 L 中的元素逐个存入文件 filename 中。

2) 从文件载入数据到线性表：函数名称为 LoadFromFile(L, filename)；初始条件为线性表 L 已存在且不包含元素；操作结果是从文件 filename 中逐个读取元素并载入到线性表 L 中。

## 1.2 系统设计

### 1.2.1 总控流程框架

本实现方案首先由用户输入待操作的线性表的数量 n，然后创建 n 个线性表，然后进入一个循环，循环中，首先由用户输入待操作的线性表的序号，然后将线性表的操作指针指向待操作的线性表，然后由用户输入对该线性表的操作的序号，然后根据用户输入的操作序号进入不同的分支，直到用户输入 0 循环结束，同时算法结束。该算法的流程图如图 1.1 所示。

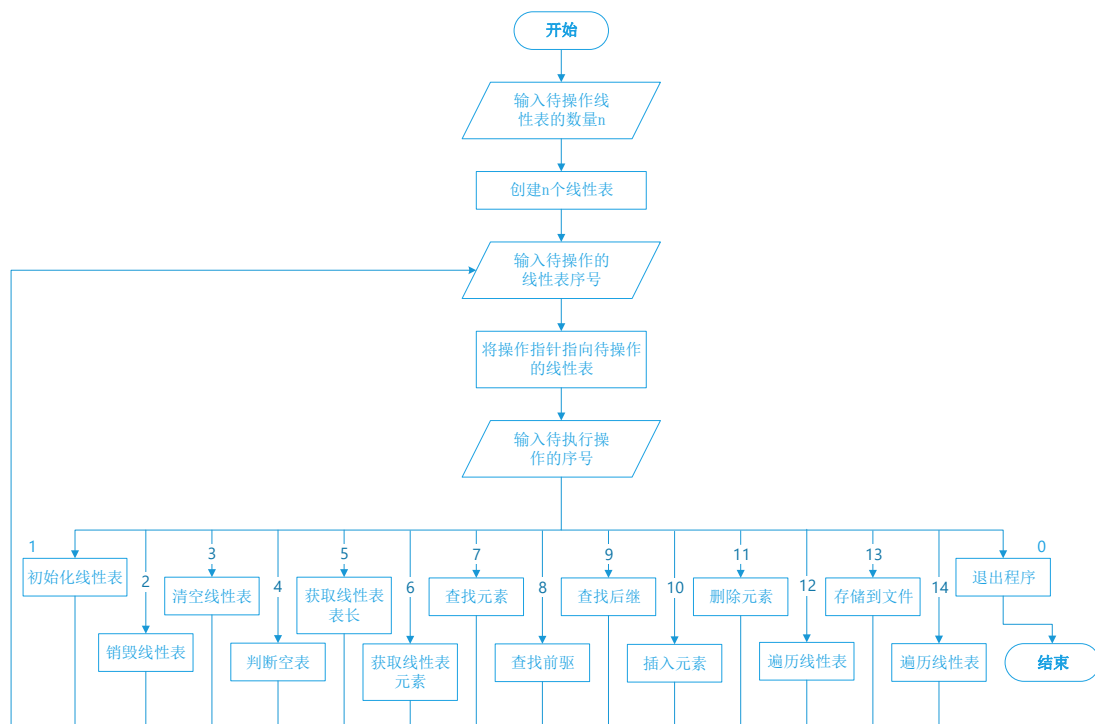


图 1.1 算法的简略流程图

### 1.2.2 定义常量、数据类型及数据结构

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

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

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

线性表结构体在内存中的存储状态如图 1.2 所示，图中每一行三个元素表示一个线性表，线性表的第一个元素 elem 指向线性表的元素存储空间。

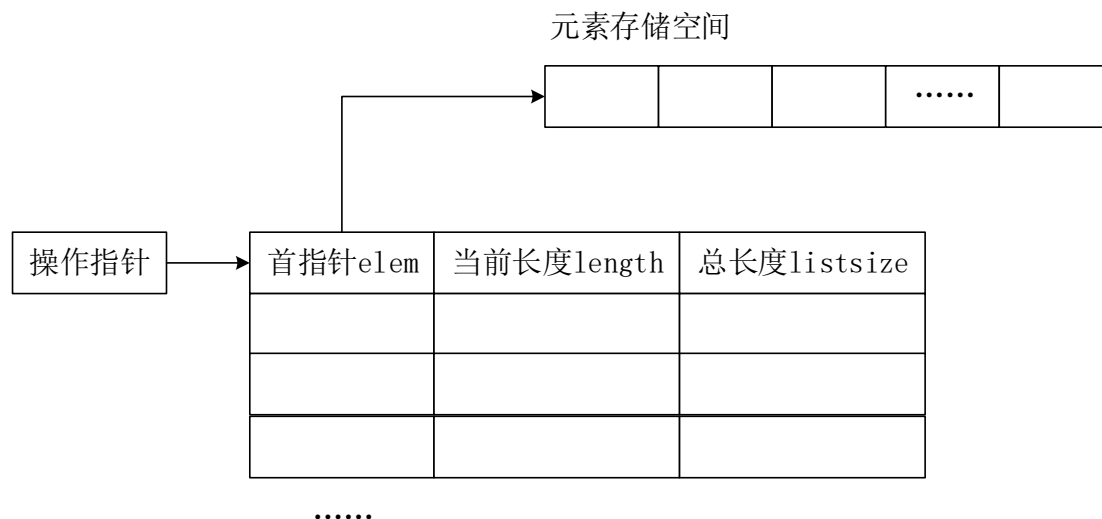


图 1.2 线性表数据结构的定义及多表表示

### 1.2.3 12 个基本操作的设计

关于函数调用方法的说明：初始化表函数（InitaList）接受的线性表指针指向的线性表应为未初始化表，否则不调用该函数并提醒用户“线性表已存在”。其余 11 个基本操作的函数所接受的线性表指针指向的线性表应为已初始化表，否则不调用该函数并提醒用户“线性表不存在”。

#### 1) 初始化表（InitaList）

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

该函数只有顺序和选择结构，因此时间复杂度为  $O(1)$ 。

#### 2) 销毁线性表(DestroyList)

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

该函数只有顺序结构，因此其时间复杂度为  $O(1)$ 。

#### 3) 清空线性表(ClearList)

该函数接受一个 SqList 类型的指针，函数将指针指向的线性表的当前长度 length 置为 0 并返回 OK。

该函数只有顺序结构，因此其时间复杂度为  $O(1)$ 。

#### 4) 判断空表(ListEmpty)

该函数接受一个 SqList 类型的值，函数判断 SqList 线性表的当前长度 length 是否为 0，若为 0，说明线性表为空，函数返回 TRUE，否则，说明线性表不为空，函数返回 FALSE。

该函数只有顺序和选择结构，因此其时间复杂度为  $O(1)$ 。

#### 5) 获取线性表表长(ListLength)

该函数接受一个 SqList 类型的值，函数返回 SqList 线性表的当前长度 length。

该函数只有顺序结构，因此其时间复杂度为  $O(1)$ 。

#### 6) 获取线性表元素(GetElem)

该函数接受三个参数，第一个为 SqList 类型，为待查找的线性表；第二个为 int 类型，为线性表中待查找的元素的位置；第三个为 ElemType 类型的指针，函数将查找到的元素存储到其指向的位置空间。若待查找位置不在线性表元素位置范围内，则返回 OVERFLOW，否则，将该位置的元素赋值给参数 3 指向的位置并返回 OK。将线性表的线性存储结构视为数组，可通过数组的下标直接方位到线性表的任意一个元素。

该函数只有顺序和选择结构，因此其时间复杂度为  $O(1)$ 。

#### 7) 查找元素(LocateElem)

该函数接受两个参数，第一个为 SqList 类型，为待查找的线性表；第二个为 Elemtype 类型，为待查找的元素的值。函数遍历整个线性表，并将其中的元素逐个的与待查找元素的值进行比较，若找到一个相等的元素，则返回该元素的位置，若遍历完整个线性表仍未找到该元素，则返回-1，表明线性表中不存在该元素的情况。

该函数包含一个循环结构，在最好的情况下，即线性表中第一个元素即为待查找的元素，则循环执行 1 次，在最坏的情况下，即线性表中最后一个元素为待

查找的元素或待查找的元素不在线性表中，则循环执行  $n$  次( $n$  为线性表当前的表长)，因此循环平均执行  $(n+1)/2$  次，则函数的时间复杂度为  $O(n)$ 。

#### 8) 查找前驱(PriorElem)

该函数接受三个参数，第一个为 SqList 类型，为待操作的线性表；第二个为 Elemtype 类型，为待查找其前驱的元素的值；第三个为 Elemtype 类型的指针，其指向的空间用于保存查找到的前驱的值。首先函数判断待查找前驱的元素是否为线性表的第一个元素，若是，则返回-3，表明其为第一个元素，不能够查找前驱；否则，函数遍历线性表查找该元素的位置，若找到，则将其前一个位置的值赋给参数 3 所指向的空间，并返回 OK，若未找到该元素，则返回 INFEASTABLE，表明线性表中不存在该元素。

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

#### 9) 查找后继(NextElem)

该函数接受三个参数，第一个为 SqList 类型，为待操作的线性表；第二个为 ElemType 类型，为待查找后继的元素的值；第三个为 Elemtype 类型的指针，其指向的空间用于保存查找到的后继的值。首先函数判断待查找后继的元素是否为线性表的最后一个元素，若是，则返回-3，表明其为最后一个元素，不能够查找后继；否则，函数遍历线性表查找该元素的位置，若找到，则将其后一个位置的值赋给参数 3 所指向的空间，并返回 OK，若未找到该元素，则返回 INFEASTABLE，表明线性表中不存在该元素。

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

#### 10) 插入元素(ListInsert)



该函数接受三个参数，第一个为 `SqList` 类型的指针，为待操作的线性表；第二个为 `int` 类型，为插入元素的位置；第三个为 `Elemtype` 类型，为待插入元素的值。函数首先判断插入位置是否合法，若插入位置小于 1 或比线性表的当前长度大 1 以上，则返回 `OVERFLOW`，表明插入位置超出了线性表的可插入范围，若线性表当前长度等于线性表的总长度，则返回 -2，表明线性表已满，不能够继续插入元素；否则，函数从最后一个元素开始，将待插入位置及其后的所有元素向后移动一个单位，然后将待插入元素存储到待插入的位置，将线性表的当前长度 `length` 加 1，并返回 `OK`。

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

#### 11) 删除元素 (`ListDelete`)

该函数接受三个参数，第一个为 `SqList` 类型的指针，为待操作的线性表；第二个为 `int` 类型，为待删除元素的位置；第三个为 `ElemType` 类型的指针，其指向的位置用于存储删除掉的元素的值。函数首先判断待删除元素的位置是否合法，若该位置小于 1 或大于线性表当前长度，则返回 `OVERFLOW`，表明删除位置超出了线性表的可删除范围；否则，将待删除位置的元素赋给参数 3 所指向的空间，从该位置开始将其后所有元素向前移动一个单位，并将线性表的当前长度 `length` 减 1，返回 `OK`。

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

#### 12) 遍历线性表 (`ListTraverse`)

该函数接受一个参数，为 `SqList` 类型，为待操作的线性表。函数遍历线性表的每一个元素，并将其逐个输出。函数返回线性表的当前长度。

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

#### 1.2.4 多线性表操作的设计

本实现方案在最初由用户指定需要操作的线性表的数量,然后每次由用户选择待操作的线性表并对其进行操作。

程序从用户获取到待操作的线性表的数量  $n$  以后,分配一段长度为  $SqList$  长度的  $n$  倍的空间,用于存储  $n$  个线性表,并保存其首指针。

在对线性表进行操作的过程中,用户选择要操作的线性表的序号(范围为  $0 \sim n-1$ ),用户选定第  $i$  个线性表时,使用指针  $L$  指向待操作的线性表,这样就可以使用与单个线性表的操作时相同的操作来对  $L$  进行操作,从而实现多线性表操作的功能。

多线性表在内存中的存储方式如图 1.2 所示,图中每行表示一个线性表,用户选择一个线性表后,将操作指针指向该线性表所在的地址,从而对该线性表进行操作。

#### 1.2.5 文件输入输出的设计

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

两个函数调用前均要求待操作的线性表已存在,即已初始化,否则不调用函数,并提示用户“线性表不存在”。

##### 1) 将线性表存储到文件(`LoadToFile`)

该函数接受两个参数,第一个为 `SqList` 类型,为待操作的线性表;第二个为指向 `char` 类型的指针,为用来存储线性表内容的文件的文件名。函数首先打开待使用的文件,若文件已存在,则将文件指针 `fp` 指向该文件,若文件不存在,则创建该文件并将文件指针 `fp` 指向该文件,文件打开后,函数将覆盖文件中原有的内容,并写入其他的内容。若文件打开失败,则返回 1,向主函数表明文件未能正常打开。若文件正常打开,则将线性表中的数据逐个的写入到该文件中。文件写入完成后,关闭文件并返回 0,向主函数表明文件写入成功。

该函数中只有顺序结构和选择结构，因此时间复杂度为  $O(1)$ 。

通过该函数将线性表内容写入文件后文件中的线性表数据的存储结构如下。函数以二进制方式将线性表中的数据写入到文件中，这样虽然文件本身不便于阅读，但是便于快速将文件中的数据加载到其他线性表中。文件中数据的格式如下：由于本实现中线性表的每个元素的类型均为 `int` 类型，占用 4 个字节，在 Intel 微处理器下，数据在内存中采用小端存储的方式存储，即数据以字节为单位逆序的在内存中存储。该函数输出到文件的结果为线性表中元素在内存中的样式，因此在文件中，线性表中的每个元素按顺序相应地对应 8 个十六进制数字，这 8 个十六进制数字按顺序每两个一组分为四组，与元素的十六进制表示按字节的顺序按逆序输出。

例如：图 1.3 和图 1.4 为线性表的文件输出及线性表的可视化内容，其中，元素 1423 的十六进制表示为 00 00 05 8F，对应文件中的 8F 05 00 00，即按字节序反向输出。

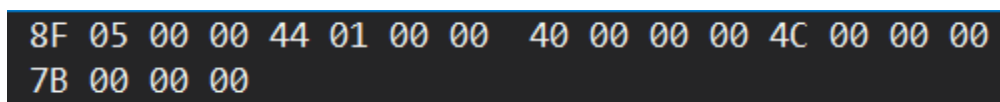


图 1.3 文件内容的二进制表示(用 visual studio 打开得到)

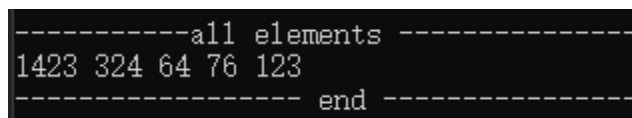


图 1.4 线性表内容(调用函数 ListTraverse 得到)

## 2) 从文件读取数据并载入线性表(LoadFromFile)

该函数接受两个参数，第一个为指向 `SqList` 类型的指针，为待操作的线性表，第二个为指向 `char` 类型的指针，为数据源文件的文件名。函数首先打开待读取数据的文件，若文件打开失败，则返回 -1，告知主函数文件打开失败；若传入函数的线性表的长度不为 0，则返回 1，告知主函数线性表中已存在元素，不允许从文件载入数据；若文件成功打开且线性表为空，则从文件中每次读取 `Elemtype` 类型长度的数据并将其写入到线性表中，每写入一个元素都将线性表的长度自增 1，直到将文件中的数据读取完毕。然后关闭文件，并返回 0，向主

函数表明函数执行成功。

该函数包含一个循环结构，若函数正常的完成了将文件中的数据载入线性表的功能，则循环必定执行了  $n$  次 ( $n$  为文件中包含的数据元素数量)，故函数的时间复杂度为  $O(n)$ 。

### 1.2.6 异常输入的处理

本解决方案中包含大量用户输入，而用户的输入内容具有不可预见性，因此每次读取用户输入的时候都应该对其进行相应的处理，若输入符合输入要求，则进行正常的操作，如用户输入不合法，则提示用户输入不合法并要求用户重新输入，若用户输入合法数据后多输入了一些其他数据，则应该清空缓冲区中的多余数据，以防下次获取输入时出现未知的错误。

对每一个用户输入都应该执行上述操作以保证程序得到正确的用户输入，从而保证程序能够正确稳定的运行。

## 1.3 系统实现

### 1.3.1 实现方案

本系统使用 C 语言实现，实现方案包括一个头文件 (`linear_list.h`) 和一个源文件 (`main.c`)。

头文件 `linear_list.h` 中定义了常量、数据类型，构建了线性表的结构并完成了对线性表操作的相关函数。

源文件 `main.c` 中通过调用各函数对线性表进行各种操作，并实现了多线性表的操作选项和目录选项。

完成本实验的操作系统为 Windows 10 64 位，使用 visual studio 2017 编写代码、进行调试及进行功能测试。

### 1.3.2 程序设计

完整的程序见附录 1，部分程序的说明如下。

- 1) 常量、数据类型及数据结构的定义

在头文件 linear\_list.h 中首先实现了 1.2.2 中对常量、数据类型以及数据结构的定义。使用#define 宏定义了常量，使用 typedef 定义了数据类型，并定义了线性表结构体类型。具体的实现代码如下。

```
//定义常量
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASTABLE -1
#define OVERFLOW -2

/*-----线性表的动态分配顺序存储结构-----*/
#define LIST_INIT_SIZE 100//线性表存储空间的初始分配量

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

#ifndef __SqList__
#define __SqList__
//定义线性表的结构
typedef struct { //顺序表(顺序结构)的定义 动态分配
    ElemType * elem; //存储空间基址
    int length; //当前长度
    int listsize; //当前分配的存储容量
}SqList;
#endif // __SqList__
```

## 2) 对异常输入的处理的代码

在 1.2.6 中设计了对用户异常的处理方法，下面以解决方案中的第一次输入（获取用户想要处理的线性表的数量）为例说明对异常输入的处理方法。代码如下。

```
1  printf("请输入想要处理的线性表的数量: ");
2  while (scanf("%d", &numList) != 1)
3  {
4      printf("输入错误, 请重新输入想要处理的线性表数量: ");
5      while (getchar() != '\n');
6  }
```

7 while (getchar() != '\n');

首先打印提示信息，要求用户输入想要处理的线性表的数量。然后使用 while 循环，若 scanf("%d", &numList) 的返回值为 1，即 scanf 成功地从键盘缓冲区读取到一个合法的值（此处为 int 类型的值），则终止循环，然后使用第 7 行的 while 循环，不断地使用 getchar() 从缓冲区读取元素，直到读取到 '\n' 为止，即将用户在输入换行符以前输入的所有元素以及换行符本身全部读取掉；若 scanf("%d", &numList) 的返回值不为 1，则说明用户没有输入合法的数据，此时执行循环内容，提示用户输入错误，要求用户重新输入，并使用第 5 行的 while 循环读取掉用户输入的所有非法的数据。第 2 行的 while 循环不断执行，直到用户输入合法的数据。

### 1.3.3 系统测试

本测试中，对 12 个基本操作以及文件输入输出功能的测试均采用创建 3 个线性表并选用其中的第 1 个线性表作为待操作的线性表。

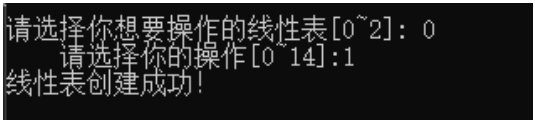
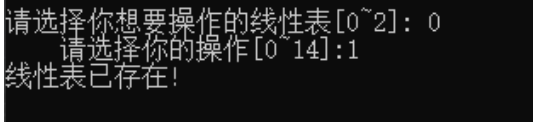
#### 1) 初始化表(InitaList)

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

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

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-1。

表 1-1 初始化表函数测试

	输入	理论结果	运行结果（截图）
正常 输入	0 1 选择第 0 个线性表 （该线性表未初始 化），选择功能 1	输出“线 性表创建 成功”	
异常 输入 1	0 1 选择第 0 个线性表 （该线性表已初始 化），选择功能 1	输出“线 性表已存 在”	

异常输入 2	0 1 选择第 0 个线性表 (该线性表未初始化), 选择功能 1	输出“线性表创建失败”	本系统对内存要求较小, 基本不会出现内存不足的情况。
-----------	---	-------------	----------------------------

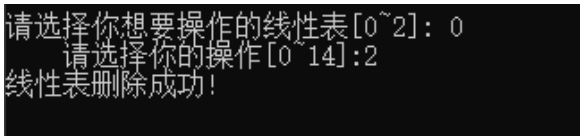
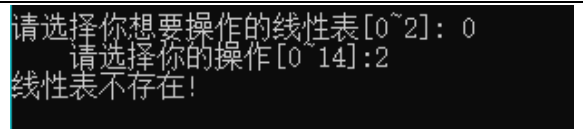
## 2) 销毁线性表(DestroyList)

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

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

具体的测试样例, 理论输出结果以及程序实际输出结果见表 1-2。

表 1-2 销毁线性表功能测试

	输入	理论结果	运行结果(截图)
正常输入	0 2 选择第 0 个线性表 (该线性表已初始化), 选择功能 2	输出“线性表删除成功”	
异常输入	0 2 选择第 0 个线性表 (该线性表未初始化), 选择功能 2	输出“线性表不存在”	

## 3) 清空线性表(ClearList)

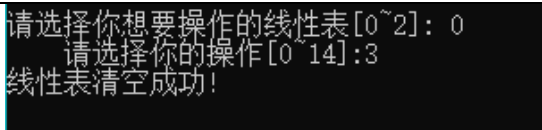
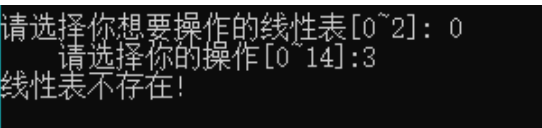
清空线性表的正常情况为待操作的线性表已初始化, 此时函数返回 OK, 程序应输出“线性表清空成功”。

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

具体的测试样例, 理论输出结果以及程序实际输出结果见表 1-3。

表 1-3 清空线性表功能测试

	输入	理论结果	运行结果(截图)
--	----	------	----------

正常输入	0 3 选择第 0 个线性表（该线性表已初始化），选择功能 3	输出“线性表清空成功”	
异常输入	0 3 选择第 0 个线性表（该线性表未初始化），选择功能 3	输出“线性表不存在”	

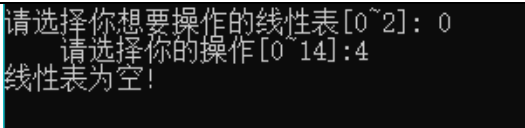
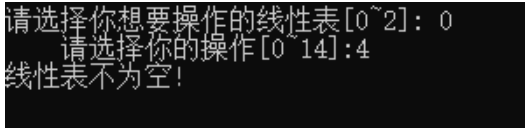
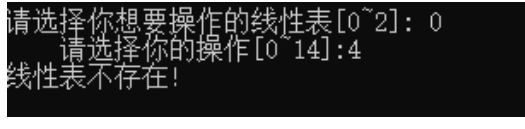
### 3) 判断空表(ListEmpty)

判断空表的正常情况为待操作的线性表已初始化，此时函数判断线性表是否为空，若为空则函数返回 TRUE，程序输出“线性表为空”，若不为空则返回 FALSE，程序输出“线性表不为空”，因此测试时应分为两种正常输入分别测试。

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

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-4。

表 1-4 判断空表功能测试

	输入	理论结果	运行结果（截图）
正常输入 1	0 4 选择第 0 个线性表（该线性表已初始化但没有元素），选择功能 4	输出“线性表为空”	
正常输入 2	0 4 选择第 0 个线性表（该线性表已初始化且有元素），选择功能 4	输出“线性表不为空”	
异常输入	0 4 选择第 0 个线性表（该线性表未初始化），选择功能 4	输出“线性表不存在”	

### 5) 获取线性表表长(ListLength)

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

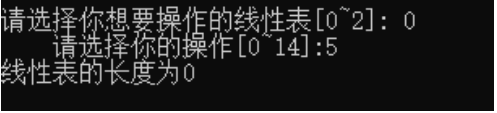
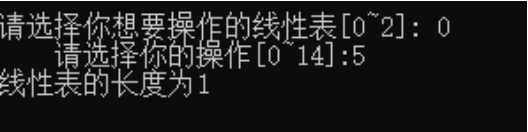
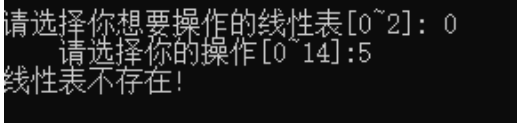
异常情况为待操作的线性表未初始化，此时函数不调用，程序输出“线性表



不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-5。

表 1-5 获取线性表表长功能测试

	输入	理论结果	运行结果（截图）
正常 输入 1	0 5 选择第 0 个线性表 （已初始化且没有元 素），选择功能 5	输出“线 性表长度 为 0”	
正常 输入 2	0 5 选择第 0 个线性表 （已初始化且含有 1 个元素），选择功能 5	输出“线 性表长度 为 1”	
异常 输入	0 5 选择第 0 个线性表 （未初始化），选择功 能 5	输出“线 性表不存 在”	

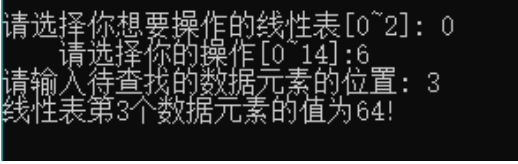
#### 6) 获取线性表元素(GetElem)

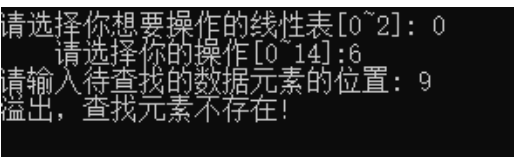
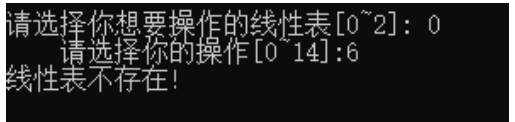
获取线性表元素的正常情况为线性表已存在且待查找位置存在元素，此时函数返回 OK，程序输出待查找元素的值。

异常情况有两种情况，第一种为线性表已存在但待查找位置不存在元素，此时函数返回 OVERFLOW，程序输出“查找元素不存在”；第二种为线性表未初始化，此时函数不调用，程序输出“线性表不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-6。本测试中，当线性表存在时，线性表的元素为 1423 324 64 76 123。

表 1-6 获取线性表元素功能测试

	输入	理论结果	运行结果（截图）
正常 输入	0 6 3 选择第 0 个线性表 （已初始化），选择 功能 6，查找第 3 个 元素	输出“线性 表第 3 个元 素为 64”	

异常输入 1	0 6 9 选择第 0 个线性表（已初始化），选择功能 6，查找第 9 个元素	输出“溢出，查找元素不存在”	
异常输入 2	0 6 选择第 0 个线性表（未初始化），选择功能 6	输出“线性表不存在”	

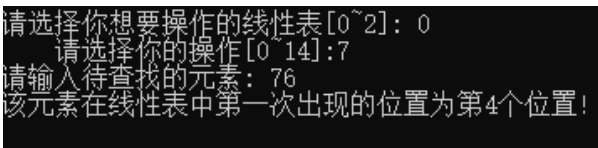
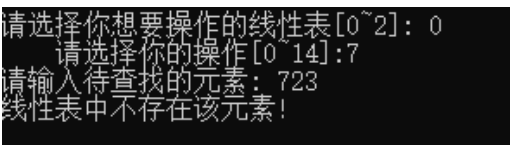
### 7) 查找元素(LocateElem)

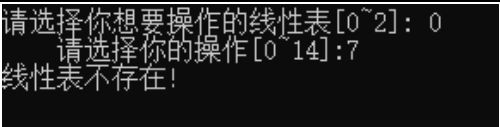
查找元素的正常情况为待操作的线性表已存在且待查找的元素在线性表中，此时函数返回该元素在线性表中第一次出现的位置，程序输出该元素在线性表中第一次出现的位置。

异常情况有两种情况，第一种为线性表存在但待查找元素不在线性表中，此时函数返回-1，程序输出“线性表中不存在该元素”；第二种为待操作的线性表未初始化，此时函数不调用，程序输出“线性表不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-7。本测试中，当线性表存在时，线性表的元素为 1423 324 64 76 123。

表 1-7 查找元素功能测试

	输入	理论结果	运行结果（截图）
正常输入	0 7 76 选择第 0 个线性表（已初始化），选择功能 7，查找元素 76	输出“该元素在线性表中第一次出现的位置为第 4 个位置”	
异常输入 1	0 7 723 选择第 0 个线性表（已初始化），选择功能 7，查找元素 723	输出“线性表中不存在该元素”	

异常输入 2	0 7 选择第 0 个线性表（未初始化），选择功能 7	输出“线性表不存在”	
--------	--------------------------------	------------	--

#### 8) 查找前驱(PriorElem)

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

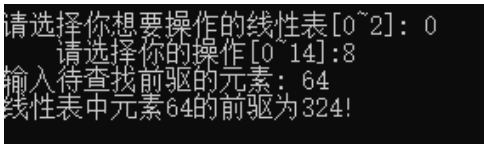
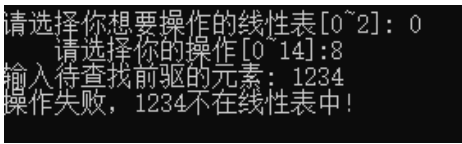
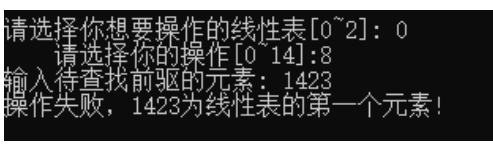
异常情况有三种情况，第一种为待查找元素不在线性表中，此时函数返回 INFEASTABLE，程序输出该元素不在线性表中；

第二种情况为该元素为线性表的第一个元素，此时函数返回-3，程序输出该元素为线性表的第一个元素；

第三种情况为线性表未初始化，此时函数不调用，程序输出“线性表不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-8。本测试中，当线性表存在时，线性表的元素为 1423 324 64 76 123。

表 1-8 查找前驱功能测试

	输入	理论结果	运行结果（截图）
正常输入	0 8 64 选择第 0 个线性表（已初始化），选择功能 8，查找元素 64 的前驱	输出“线性表中元素 64 的前驱为 324”	
异常输入 1	0 8 1234 选择第 0 个线性表（已初始化），选择功能 8，查找元素 1234 的前驱	输出“操作失败，1234 不在线性表中”	
异常输入 2	0 8 1423 选择第 0 个线性表（已初始化），选择功能 8，查找元素 1423 的前驱	输出“操作失败，1423 为线性表的第一个元素”	

异常输入 3	0 8 选择第 0 个线性表 (未初始化), 选择 功能 8	输出“线性表 不存在”	
--------	---	----------------	--

#### 9) 查找后继 (NextElem)

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

异常情况有三种情况, 第一种为线性表存在但待查找后继的元素为线性表的最后一个元素, 此时函数返回-3, 程序输出该元素为线性表的最后一个元素;

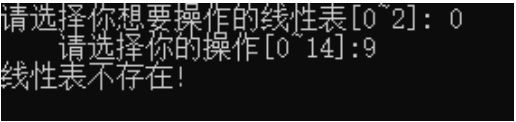
第二种情况为该元素不在线性表中, 此时函数返回 INFEASTABLE, 程序输出该元素不在线性表中;

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

具体的测试样例, 理论输出结果以及程序实际输出结果见表 1-9。本测试中, 当线性表存在时, 线性表的元素为 1423 324 64 76 123

表 1-9 查找后继功能测试

	输入	理论结果	运行结果 (截图)
正常输入	0 9 1423 选择第 0 个线性表 (已初始化), 选 择功能 9, 查找元 素 1423 的后继	输出“线性表 中元素 1423 的后继为 423”	
异常输入 1	0 9 123 选择第 0 个线性表 (已初始化), 选 择功能 9, 查找元 素 123 的后继	输出“操作失 败, 123 为线 性表的最后 一个元素”	
异常输入 2	0 9 1881 选择第 0 个线性表 (已初始化), 选 择功能 9, 查找元 素 1881 的后继	输出“操作失 败, 1881 不 在线性表中”	

异常输入 3	0 9 选择第 0 个线性表（已初始化），选择功能 9	输出“线性表不存在”	
--------	--------------------------------	------------	--

#### 10) 插入元素(ListInsert)

插入元素的正常情况为线性表已存在且待插入位置可以插入元素，此时函数返回 OK, 程序输出“插入成功”。插入成功可分为三种情况进行测试，分别在线性表的首部、尾部以及中间进行插入操作。

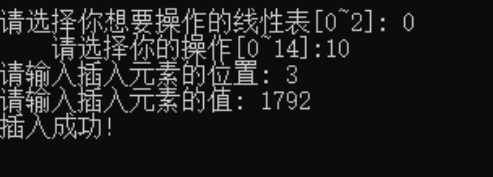
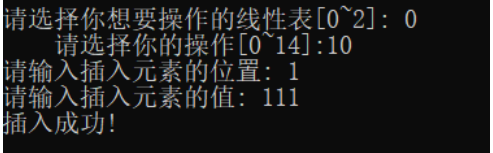
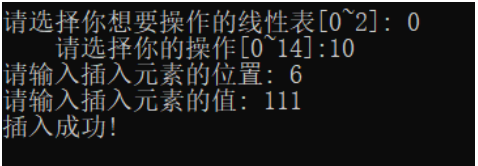
异常情况有三种情况。第一种为线性表存在但待插入位置不能够插入元素，此时函数返回 OVERFLOW，程序输出该位置超出线性表可插入范围；

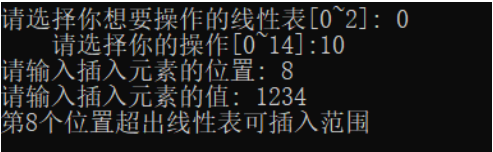
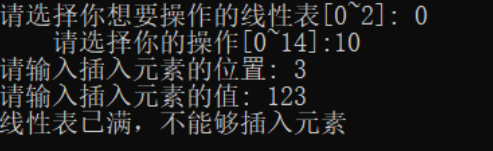
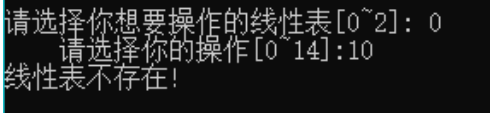
第二种情况为待操作的线性表未初始化，此时函数不调用，程序输出“线性表不存在”。

第三种情况为待操作的线性表已满，无法插入元素，此时函数返回-3，程序输出“线性表已满，不能够插入元素”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-10。本测试中，在正常输入中，线性表中的元素均为 1423 324 64 76 123。

表 1-10 插入元素功能测试

	输入	理论结果	运行结果（截图）
正常输入 1	0 10 3 1792 选择第 0 个线性表（已初始化且含有元素），选择功能 10，在第 3 个位置插入元素 1792	输出“插入成功”	
正常输入 2	0 10 1 111 选择第 0 个线性表(已初始化)，选择功能 10，在第 1 个位置插入元素 111	输出“插入成功”	
正常输入 3	0 10 6 111 选择第 0 个线性表(已初始化)，选择功能 10，在第 6 个位置插入元素 111	输出“插入成功”	

异常输入 1	0 10 8 1234 选择第 0 个线性表（已初始化且含有元素），选择功能 10，在第 8 个位置插入元素 1234	输出“第 8 个位置超出线性表可插入范围”	
异常输入 2	0 10 3 123 选择第 0 个线性表（已初始化且已满），选择功能 10，在第 3 个位置插入元素 123	输出“线性表已满，不能够插入元素”	
异常输入 3	0 10 选择第 0 个线性表（未初始化），选择功能 10	输出“线性表不存在”	

#### 11) 删除元素(ListDelete)

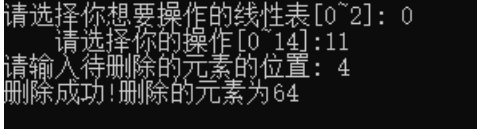
删除元素的正常情况为待操作的线性表已初始化且待删除位置存在元素，此时函数将删除掉的元素存入指定位置，并返回 OK，程序输出“删除成功”并输出删除的元素的值。成功删除元素可分为三种情况进行测试，分别为删除首元素、删除尾元素和删除中间的元素。

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

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

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-11。本测试中，在正常输入中，线性表的元素均为 1423 324 1792 64 76 123。

表 1-11 删除元素功能测试

	输入	理论结果	运行结果（截图）
正常输入 1	0 11 4 选择第 0 个线性表（已初始化且含有元素），选择功能 11，删除第 4 个位置的元素	输出“删除成功，删除的元素为 64”	

正常输入 2	0 11 1 选择第 0 个线性表(已初始化), 选择功能 11, 删除第 1 个元素	输出“删除成功, 删除的元素为 1423”	请选择你想要操作的线性表[0~2]: 0 请选择你的操作[0~14]:11 请输入待删除的元素的位置: 1 删除成功!删除的元素为1423
正常输入 3	0 11 6 选择第 0 个线性表(已初始化), 选择功能 11, 删除第 6 个元素	输出“删除成功, 删除的元素为 123”	请选择你想要操作的线性表[0~2]: 0 请选择你的操作[0~14]:11 请输入待删除的元素的位置: 6 删除成功!删除的元素为123
异常输入 1	0 11 10 选择第 0 个线性表 (已初始化且含有元素), 选择功能 11, 删除第 10 个位置的元素	输出“溢出”	请选择你想要操作的线性表[0~2]: 0 请选择你的操作[0~14]:11 请输入待删除的元素的位置: 10 溢出!
异常输入 2	0 11 选择第 0 个线性表 (未初始化), 选择功能 11	输出“线性表不存在”	请选择你想要操作的线性表[0~2]: 0 请选择你的操作[0~14]:11 线性表不存在!

## 12) 遍历线性表(ListTraverse)

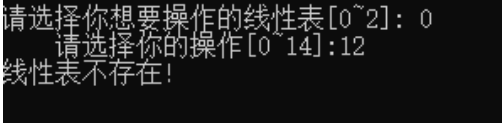
遍历线性表的正常情况为线性表已初始化, 此时函数依次打印线性表的元素并返回线性表的长度, 若线性表长度为 0, 则程序输出“线性表为空表”。

异常情况为线性表未初始化, 此时函数不调用, 程序输出“线性表不存在”。

具体的测试样例, 理论输出结果以及程序实际输出结果见表 1-12。本测试中, 线性表存在且不为空时, 线性表的元素为 1423 324 64 76 123。

表 1-12 遍历线性表功能测试

	输入	理论结果	运行结果 (截图)
正常输入 1	0 12 选择第 0 个线性表 (已初始化, 含有元素), 选择功能 12	输出“1423 324 64 76 123”	请选择你想要操作的线性表[0~2]: 0 请选择你的操作[0~14]:12  -----all elements ----- 1423 324 64 76 123 ----- end -----
正常输入 2	0 12 选择第 0 个线性表 (已初始化, 没有元素), 选择功能 12	输出“线性表为空”	请选择你想要操作的线性表[0~2]: 0 请选择你的操作[0~14]:12  -----all elements ----- ----- end ----- 线性表是空表!

异常输入	0 12 选择第 0 个线性表（未初始化），选择功能 12	输出“线性表不存在”	
------	----------------------------------	------------	--

### 13) 将线性表存储到文件 (LoadToFile)

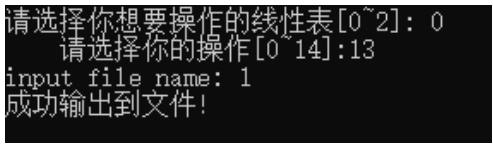
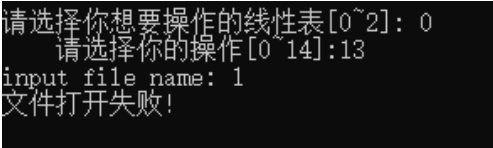
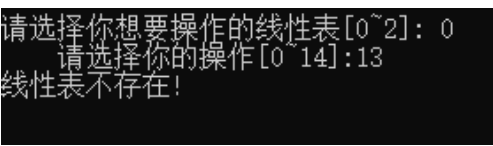
将线性表存储到文件的正常情况为待操作线性表已初始化且文件正常打开，此时函数返回 0，程序输出“成功输出到文件”。

异常情况有两种情况，第一种为线性表已初始化但文件打开失败，此时函数返回-1，程序输出“文件打开失败”；

第二种情况为待操作线性表未初始化，此时函数不调用，程序输出“线性表不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-13。本测试中线性表存在时，线性表的元素为 1423 324 64 76 123。

表 1-13 存储到文件功能测试

	输入	理论结果	运行结果（截图）
正常输入	0 13 1 选择第 0 个线性表（已初始化），选择功能 13，将数据存储到文件 1	输出“成功输出到文件”	
异常输入 1	0 13 1 选择第 0 个线性表（已初始化），选择功能 13，将数据存储到文件 1	输出“文件打开失败”	
异常输入 2	0 13 选择第 0 个线性表（未初始化），选择功能 13	输出“线性表不存在”	

### 14) 从文件读取数据并载入线性表 (LoadFromFile)

从文件读取数据并载入线性表的正常情况为线性表已初始化且为空且文件正常打开，此时函数返回 OK，程序输出“载入成功”。

异常情况有三种情况，第一种为线性表存在，但文件打开失败，此时函数返回-1，程序输出“文件打开失败”；

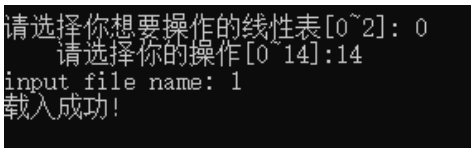
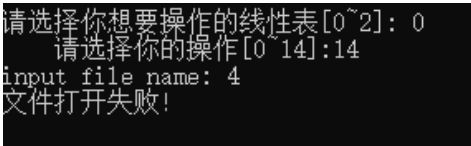
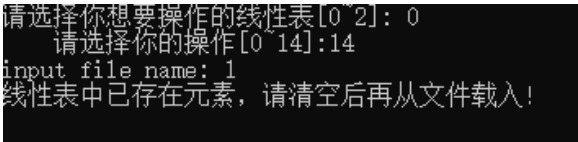
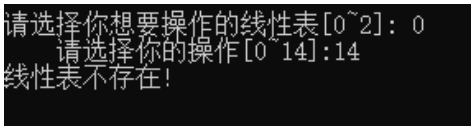


第二种情况为线性表存在且文件成功打开但线性表不为空，此时函数返回 1，程序输出“线性表中已存在元素，请清空后再从文件载入”；

第三种情况为线性表未初始化，此时函数不调用，程序输出“线性表不存在”。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-14。

表 1-14 从文件读取数据功能测试

	输入	理论结果	运行结果（截图）
正常输入	0 14 1 选择第 0 个线性表（已初始化，没有元素），选择功能 14，从文件 1 中读取数据	输出“载入成功”	
异常输入 1	0 14 3 选择第 0 个线性表（已初始化，没有元素），选择功能 14，从文件 4（文件不存在）读取数据	输出“文件打开失败”	
异常输入 2	0 14 1 选择第 0 个线性表（已初始化，有元素），选择功能 14，从文件 1 中读取数据	输出“线性表中已存在元素，请清空后再从文件载入”	
异常输入 3	0 14 选择第 0 个线性表（未初始化），选择功能 14	输出“线性表不存在”	

#### 15) 退出程序

退出程序的功能与线性表的状态无关，均为正常情况。

具体的测试样例，理论输出结果以及程序实际输出结果见表 1-15。

表 1-15 退出程序功能测试

	输入	理论结果	运行结果（截图）
--	----	------	----------

正常 输入	0 0 选择第 0 个线性 表, 选择功能 0	输出“欢迎下次 再使用本系统”	请选择你想要操作的线性表[0~2]: 0 请选择你的操作[0~14]: 0 欢迎下次再使用本系统!
----------	-------------------------------	--------------------	---

### 1.3.4 其他问题的说明

在测试环境(Windows 10 64 位)中, 若可执行文件在 C 盘根目录下, 则无法使用 LoadToFile 函数将线性表内容写入到不存在的文件中, 原因为: 程序没有权限在 C 盘根目录下创建文件。若可执行文件位于 C 盘根目录下, 则需使用管理员权限运行该程序, 才能够正常使用文件 I/O 功能。

## 1.4 实验小结

在本次实验中, 我遇到了很多的困难和问题, 在解决问题和不断完善解决方案的功能的过程中, 我也有了很多的收获。

在编写函数时, 需要传递线性表、元素等信息, 在这里需要考虑传递变量本身还是传递变量指针。最开始写的时候有一些参数传递错误导致函数无法达到预期功能, 在调试时, 在函数内部的操作均为正确操作, 但函数调用结束后, 一些数据并没有按照预期的方式变化, 这个问题困扰了我一段时间, 这也使我对函数传递参数时使用指针还是使用变量自身有了更深刻的理解。

另外, 本次试验中的对通过文件输入输出来操纵线性表的实现使我对 C 语言程序与系统文件之间的读写有了更加深入的理解, 仔细观察程序输出的文件的二进制内容, 也使我对数据在内存中的存储方式(小端存储)有了更加直观的印象和更加深刻的认识。

## 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  $n$  个数据元素的有限序列。线性表的链式存储表示指的是用一组任意的存储单元存储线性表的数据元素，这组内存可以是连续的，也可以是不连续的。本实验要求利用线性表的链式存储结构以函数方式实现线性表 12 种基本操作。这 12 中基本操作如下。

1) 初始化表：函数名称是 `InitaList(L)`；初始条件是线性表  $L$  不存在已存在；操作结果是构造一个空的线性表。

2) 销毁表：函数名称是 `DestroyList(L)`；初始条件是线性表  $L$  已存在；操作结果是销毁线性表  $L$ 。

3) 清空表：函数名称是 `ClearList(L)`；初始条件是线性表  $L$  已存在；操作结果是将  $L$  重置为空表。

4) 判定空表：函数名称是 `ListEmpty(L)`；初始条件是线性表  $L$  已存在；操作结果是若  $L$  为空表则返回 `TRUE`, 否则返回 `FALSE`。

5) 求表长：函数名称是 `ListLength(L)`；初始条件是线性表已存在；操作结果是返回  $L$  中数据元素的个数。

6) 获得元素：函数名称是 `GetElem(L, i, e)`；初始条件是线性表已存在,  $1 \leq i \leq \text{ListLength}(L)$ ；操作结果是用  $e$  返回  $L$  中第  $i$  个数据元素的值。

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

8) 获得前驱：函数名称是 `PriorElem(L, cur_e, pre_e)`；初始条件是线性表  $L$  已存在；操作结果是若  $cur\_e$  是  $L$  的数据元素，且不是第一个，则用  $pre\_e$  返回它的前驱，否则操作失败， $pre\_e$  无定义。

9) 获得后继：函数名称是 `NextElem(L, cur_e, next_e)`；初始条件是线性表  $L$  已存在；操作结果是若  $cur\_e$  是  $L$  的数据元素，且不是最后一个，则用  $next\_e$

返回它的后继，否则操作失败，next\_e 无定义。

10) 插入元素：函数名称是 ListInsert(L, i, e)；初始条件是线性表 L 已存在且非空， $1 \leq i \leq \text{ListLength}(L)+1$ ；操作结果是在 L 的第 i 个位置之前插入新的数据元素 e。

11) 删除元素：函数名称是 ListDelete(L, i, e)；初始条件是线性表 L 已存在且非空， $1 \leq i \leq \text{ListLength}(L)$ ；操作结果：删除 L 的第 i 个数据元素，用 e 返回其值。

12) 遍历表：函数名称是 ListTraverse(L)，初始条件是线性表 L 已存在；操作结果是依次输出 L 的每个数据元素。

本系统还实现了对多个线性表操作的支持。

本系统提供对线性表的文件输入输出功能。本功能通过两个函数实现，两个函数的基本情况如下。

1) 保存到文件：函数名称为 LoadToFile(L, filename)；初始条件是线性表 L 已存在，操作结果是将线性表 L 中的元素逐个存入文件 filename 中。

2) 从文件载入数据到线性表：函数名称为 LoadFromFile(L, filename)；初始条件为线性表 L 已存在且不包含元素；操作结果是从文件 filename 中逐个读取元素并载入到线性表 L 中。

## 2.2 系统设计

### 2.2.1 总控流程框架

本实现方案首先由用户输入待操作的线性表的数量 n，然后创建 n 个线性表，然后进入一个循环，循环中，首先由用户输入待操作的线性表的序号，然后将线性表的操作指针指向待操作的线性表，然后由用户输入对该线性表的操作的序号，然后根据用户输入的操作序号进入不同的分支，直到用户输入 0 循环结束，同时算法结束。该算法的流程图如图 2.1 所示。

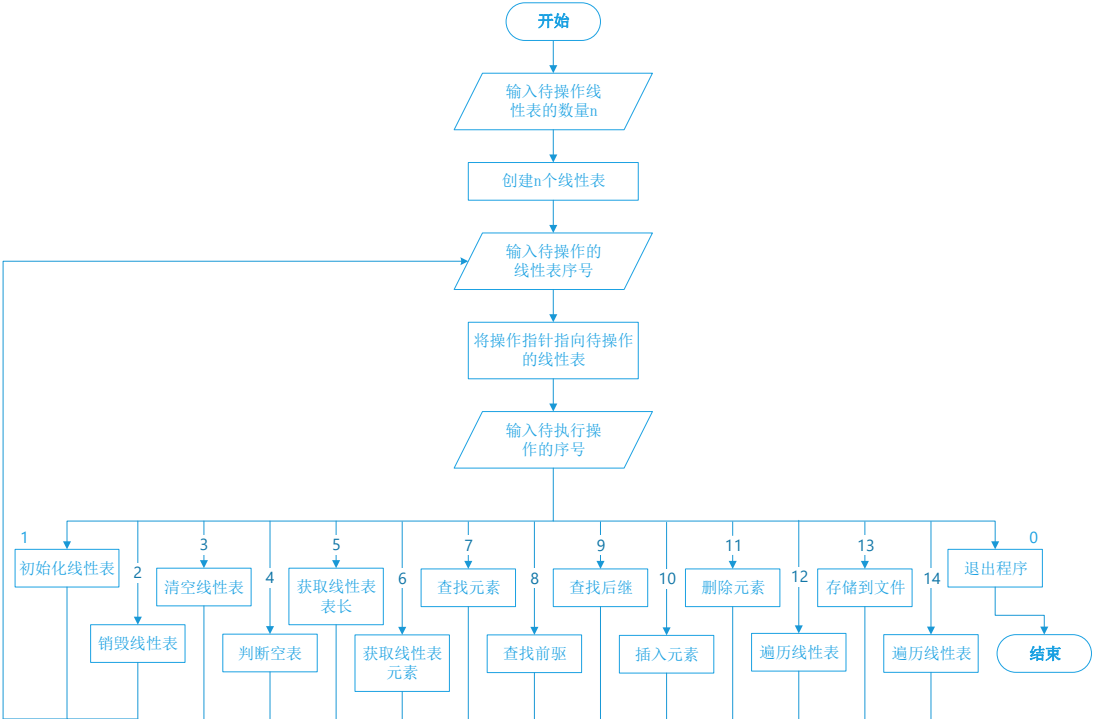


图 2.1 算法总控流程图

2.2.2 定义常量、数据类型及数据结构

1) 定义常量。定义常量用于指定函数的返回值,用来判断函数执行的情况。定义的常量有: TRUE(1), FALSE(0), OK(1), ERROR(0), INFEASTABLE(-1), OVERFLOW(-2)。

2) 定义数据元素的类型。此处定义了部分函数返回值的类型 status 以及单链表中耽搁元素的类型 ElemType。本实现中将两者均定义为 int 类型。

3) 定义单链表的数据结构。单链表结构体中包含两个元素,第一个元素为单链表的数据域 data,为 ElemType 类型,第二个元素为单链表的指针域 next,为单链表结构体类型的指针。将该结构体类型定义为 Lnode。

单链表的数据结构在内存中的存储状态如图 2.2 所示,图中操作指针指向单链表的第一个节点,单链表中的节点之间通过指针域的指针相互关联。

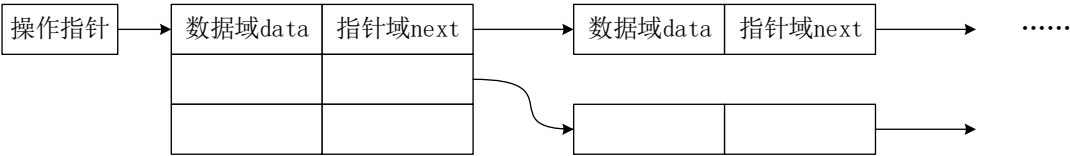


图 2.2 单链表数据结构的定义及多表表示

### 2.2.3 12 个基本操作的设计

关于函数调用方法的说明：初始化表函数 (InitaList) 接受的单链表指针指向的单链表应为未初始化的单链表，否则不调用函数并输出“单链表已存在”。其余 11 个基本操作的函数所接受的单链表指针指向的单链表应为已初始化表，否则不调用该函数并输出“线性表不存在”。

#### 1) 初始化表 (InitaList)

该函数接受一个 LNode 类型的指针 L，函数将 L 的数据域 data 修改为 0，表明该单链表已初始化，将 L 的指针域 next 置为 NULL，并返回 OK。

该函数只包含顺序结构，因此其时间复杂度为  $O(1)$ 。

#### 2) 销毁单链表 (DestroyList)

该函数接受一个 LNode 类型的指针 L，函数首先将 L 的数据域 data 置为 INT\_MIN (该值为 int 类型的最小值，由系统定义)，用于标志该单链表不存在，接下来，若 L 的指针域为 NULL，说明单链表中没有元素，则返回 OK，若 L 的指针域不为 NULL，则令 node1 为 L 的指针域，令 node2 为 node1 的指针域，将 node1 和 node2 逐个后移，同时将 node1 节点的内存逐个释放，直到 node2 为空，将 node1 释放，此时单链表中的全部元素均被释放，将 L 的指针域 next 置为 NULL，返回 OK。

该函数包含一个循环结构，由于函数需要逐个的将单链表中的每个元素的内存空间释放，因此需要循环 n 次 (n 为单链表的表长)，故函数的时间复杂度为  $O(n)$ 。

#### 3) 清空单链表 (ClearList)

该函数接受一个 LNode 类型的指针 L，若 L 的指针域为 NULL，则单链表为空表，返回 OK，若 L 的指针域不为 NULL，则令 node1 为 L 的指针域，令 node2 为 node1 的指针域，将 node1 和 node2 逐个后移，同时将 node1 节点的内存逐个释放，直到 node2 为空，将 node1 释放，此时单链表中的全部元素均被释放，将 L 的指针域 next 置为 NULL，返回 OK。

该函数包含一个循环结构，由于函数需要逐个的将单链表中的每个元素的内存空间释放，因此需要循环 n 次 (n 为单链表的表长)，故函数的时间复杂度为

$O(n)$ 。

#### 4) 判断空表(ListEmpty)

该函数接受一个 LNode 类型的指针 L，若 L 的指针域为 NULL，说明单链表为空，返回 TRUE，若 L 的指针域不为 NULL，说明单链表不为空，返回 FALSE。

该函数只有顺序和选择结构，因此时间复杂度为  $O(1)$ 。

#### 5) 求单链表表长(ListLength)

该函数接受一个 LNode 类型的指针 L，令 length 为 0，将 L 每次置为 L 的指针域 next，并将 length 自增 1，直到 L 的指针域为 NULL。此时 length 的值即为单链表的长度。返回 length。

该函数包含一个循环结构，函数需要遍历单链表的每一个节点来计算元素的数量，因此循环需要执行  $n$  次，故函数的时间复杂度为  $O(n)$ 。

#### 6) 获取线性表元素(GetElem)

该函数接受一个 LNode 类型的指针 L，待获取元素的位置  $i$ ，以及该元素的值的存放地址  $e$ 。若位置  $i$  超出了单链表的范围，则函数返回 OVERFLOW，否则，函数从单链表头指针 L 开始向后移动  $i$  次，将此时 L 指向的节点数据域的值存储到  $e$  所指向的位置，并返回 OK。

该函数包含一个循环结构，当  $i$  小于 0 时，循环不执行，当获取第一个元素的位置时，循环执行 1 次，此时函数耗时最短，当获取单链表最后一个元素或  $i$  大于单链表的长度时，循环执行  $n$  次 ( $n$  为单链表表长)，此时耗时最长，因此循环平均执行  $n/2$  次，故函数的时间复杂度为  $O(n)$ 。

#### 7) 查找元素(LocateElem)

该函数接受一个 LNode 类型的指针 L，以及待查找的元素  $e$ 。函数从 L 开始不断将 L 向后移动并设置计数器  $i$  统计移动次数，直到 L 的数据域与  $e$  的值相等，此时找到待查找元素，其位置为计数器的值  $i$ ，若 L 的值最终变为 NULL，则表明单链表中不存在待查找元素  $e$ ，此时返回 -1。

该函数包含一个循环结构，若待查找元素为第一个元素，则循环执行 1 次，此时函数执行时间最短，若待查找元素为单链表的最后一个元素或待查找元素不在该单链表中，则循环执行  $n$  次，此时函数执行时间最长。因此循环平均执行  $n/2$

次，故函数的时间复杂度为  $O(n)$ 。

#### 8) 查找前驱(PriorElem)

该函数接受一个 LNode 类型的指针 L，待查找前驱的元素 cur，以及存储前驱值的内存地址 pre\_e。若带查找前驱的元素 cur 不在单链表 L 中，则函数返回 INFEASTABLE，若单链表的第一个元素为 cur，则函数返回-3，否则，设置另一个指针 node 指向 L 的下一个元素，两个指针一起向后移动，直到 node 指向待查找前驱的元素，此时 L 指向该元素的前驱，将 L 指向的节点的数据域的值赋值给 pre\_e 指向的内存空间，并返回 OK。若 node 最终指向 NULL，即未在单链表中找到待查找前驱的元素，则返回 INFEASTABLE。

该函数包含一个循环结构，若单链表为空或待查找元素为单链表的第一个元素，则循环不执行，若待查找前驱的元素为单链表的第二个元素，则循环执行 1 次，此时函数耗时最短；若待查找前驱的元素为单链表的最后一个元素或该元素不在单链表中，则循环执行 n 次，此时函数耗时最长。因此循环平均执行  $n/2$  次，故函数的时间复杂度为  $O(n)$ 。

#### 9) 查找后继(NextElem)

该函数接受一个 LNode 类型的指针 L，待查找后继的元素 cur，以及存储后继值的内存地址 next\_e。若单链表为空，则函数返回 INFEASTABLE；将 L 沿着单链表向后移动，直到找到待查找后继的元素，若该元素为单链表的最后一个元素，则返回-3，若该元素不为单链表的最后一个元素，则将其的下一个元素赋值给 next\_e 指向的内存空间，并返回 OK，若未在单链表中找到该元素，则返回 INFEASTABLE。

该函数包含一个循环结构。若单链表为空，则循环不执行，若待查找后继的元素为单链表的第一个元素，则循环执行一次，此时函数耗时最短；若待查找元素为单链表的最后一个元素或者该元素不在单链表中，则循环执行 n 次，此时函数耗时最长。因此循环平均执行  $n/2$  次，故函数的时间复杂度为  $O(n)$ 。

#### 10) 插入元素(ListInsert)

该函数接受一个 LNode 类型的指针 L，待插入元素的位置 i，以及待插入的元素 e。首先将 L 逐个后移到第 i-1 个元素的位置，若该位置没有元素，则表明



$i$  超出了可插入的范围, 函数返回-3, 否则函数新建一个节点存储待插入的元素, 若创建节点失败, 则说明内存不足, 无法分配足够的空间, 此时返回 OVERFLOW, 若成功创建节点, 则将该节点指向第  $i$  个节点并将第  $i-1$  个节点指向该节点, 这样函数成功插入一个元素到单链表中, 返回 OK。

该函数包含一个循环结构, 循环用于将  $L$  移动到第  $i-1$  个元素, 故循环执行  $i-1$  次( $i$  为待插入元素的位置), 故函数的时间复杂度为  $O(n)$ 。

#### 11) 删除元素(ListDelete)

该函数接受一个 LNode 类型的指针  $L$ , 待删除元素的位置  $i$ , 以及保存待删除元素的位置地址  $e$ 。首先将  $L$  逐个后移到第  $i-1$  个元素的位置, 若第  $i$  个位置没有元素, 则表明待删除元素不存在, 函数返回 OVERFLOW, 否则设置指针  $node$  指向第  $i$  个元素即待删除节点, 将第  $i-1$  个节点指向第  $i+1$  个节点, 并将  $node$  指向的内存空间释放, 这样就将第  $i$  个元素从单链表中删除, 函数返回 OK。

该函数包含一个循环结构, 该循环用于将  $L$  移动到第  $i-1$  个位置, 故循环执行  $i-1$  次( $i$  为待删除元素的位置), 故函数的时间复杂度为  $O(n)$ 。

#### 12) 遍历单链表(ListTraverse)

该函数接受一个 LNode 类型的指针  $L$ 。函数将  $L$  逐个后移并依次打印每一个节点的数据域, 直到  $L$  指向 NULL, 表明单链表全部读取完毕, 函数返回 OK。

该函数包含一个循环结构, 该循环将  $L$  从单链表的首部逐个的移动到尾部, 故循环执行  $n$  次, 因此函数的时间复杂度为  $O(n)$ 。

### 2.2.4 多单链表操作的设计

本系统首先由用户选择需要操作的单链表数量  $n$ , 用户指定数量后, 程序创建一个元素类型为 LNode 的、长度为  $n$  的数组 LList。每次用户需要选择待操作的单链表的编号, 当用户输入一个编号后, 程序将会将一个操作指针指向数组中的该元素, 并在接下来的对单链表的操作中使用该指针作为操作对象。从而使对不同单链表的相同操作使用相同的方法完成函数的调用, 从而实现对多个单链表的管理和操作。

多单链表的存储方式如图 2.2 所示, 图中每行代表一个单链表, 用户选择一

个单链表后，将操作指针指向该单链表从而对该单链表进行操作。

### 2.2.5 文件输入输出的设计

本实现方案对文件输入输出的实现通过两个函数 LoadToFile 和 LoadFromFile 实现，其中 LoadToFile 将当前操作的单链表存储到文件，LoadFromFile 将文件的内容读取并存储到当前操作的单链表中。两个函数的设计如下。

两个函数调用前均要求待操作的单链表已存在，即已初始化，否则不调用函数，并提示用户“线性表不存在”。

#### 1) 将单链表存储到文件(LoadToFile)

该函数接受一个 LNode 类型的指针 L，以及一个字符串 filename。函数首先打开文件 filename，若文件打开失败，则函数返回 1，否则，将文件内容清空，将 L 沿着单链表的顺序逐个的后移，并将每一个元素的数据域的内容写入到 filename 中，直到 L 读完单链表的每一个元素。这样函数将单链表的元素全部写入到文件 filename 中，然后函数关闭文件并返回 0。

该函数包含一个循环结构，该循环将 L 逐个的从单链表的首部移动到尾部，故循环执行 n 次，因此函数的时间复杂度为  $O(n)$ 。

数据在文件中的存储方式见 1.2.5 节的说明。

#### 2) 从文件读取数据并载入单链表(LoadFromFile)

该函数接受一个 LNode 类型的指针 L，以及一个字符串 filename。首先函数打开文件 filename，若打开失败，则返回-1，若单链表不为空，则返回 1，若单链表为空，则从文件 filename 中每次读取一个元素长度的数据并将其插入到单链表的尾部，直到将文件中的数据读取完，并返回 0。

该函数包含一个循环结构，该循环将文件中的数据逐个的读取到单链表中，故循环执行 n 次，因此函数的时间复杂度为  $O(n)$ 。

### 2.2.6 异常输入的处理

本解决方案中包含大量用户输入，而用户的输入内容具有不可预见性，因此每次读取用户输入的时候都应该对其进行相应的处理，若输入符合输入要求，则

进行正常的操作，如用户输入不合法，则提示用户输入不合法并要求用户重新输入，若用户输入合法数据后多输入了一些其他数据，则应该清空缓冲区中的多余数据，以防下次获取输入时出现未知的错误。

对每一个用户输入都应该执行上述操作以保证程序得到正确的用户输入，从而保证程序能够正确稳定的运行。

## 2.3 系统实现

### 2.3.1 实现方案

本系统使用 C 语言实现，实现方案包括一个头文件（linear\_list.h）和一个源文件（main.c）。

头文件 linear\_list.h 中定义了常量、数据类型，构建了线性表的结构并完成了对线性表操作的相关函数。

源文件 main.c 中通过调用各函数对线性表进行各种操作，并实现了多线性表的操作选项和目录选项。

完成本实验的操作系统为 Windows 10 64 位，使用 visual studio 2017 编写代码、进行调试及进行功能测试。

### 2.3.2 程序设计

完整程序见附录 2，部分代码的说明如下。

#### 1) 常量、数据类型以及数据结构的定义

在头文件 linear\_list.h 中首先实现了 2.2.2 中对常量、数据类型以及数据结构的定义。使用#define 宏定义了常量，使用 typedef 定义了数据类型，并定义了单链表结构体类型。具体的实现代码如下。

```
//定义常量
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASTABLE -1
#define OVERFLOW -2
```

```
//定义数据类型
typedef int status;
typedef int ElemType;
```

```
//定义数据结构
typedef struct LNode {
    ElemType data;
    struct LNode * next;
} LNode;
```

## 2) 对异常输入的处理的代码

在 2.2.6 中设计了对用户异常的处理方法，下面以解决方案中的第一次输入（获取用户想要处理的线性表的数量）为例说明对异常输入的处理方法。代码如下。

```
1  printf("请输入想要处理的线性表的数量: ");
2  while (scanf("%d", &numList) != 1)
3  {
4      printf("输入错误, 请重新输入想要处理的线性表数量: ");
5      while (getchar() != '\n');
6  }
7  while (getchar() != '\n');
```

首先打印提示信息，要求用户输入想要处理的线性表的数量。然后使用 while 循环，若 scanf("%d", &numList) 的返回值为 1，即 scanf 成功地从键盘缓冲区读取到一个合法的值（此处为 int 类型的值），则终止循环，然后使用第 7 行的 while 循环，不断地使用 getchar() 从缓冲区读取元素，直到读取到 ‘\n’ 为止，即将用户在输入换行符以前输入的所有元素以及换行符本身全部读取掉；若 scanf("%d", &numList) 的返回值不为 1，则说明用户没有输入合法的数据，此时执行循环内容，提示用户输入错误，要求用户重新输入，并使用第 5 行的 while 循环读取掉用户输入的所有非法的数据。第 2 行的 while 循环不断执行，直到用户输入合法的数据。

### 2.3.3 系统测试

本测试中，对 12 个基本操作以及文件输入输出功能进行测试。测试时创建 3 个单链表，并均选择第 1 个单链表作为待操作的单链表。

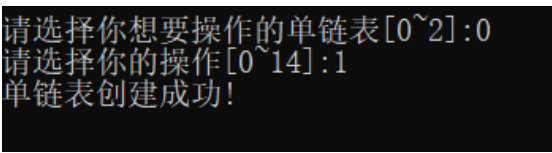
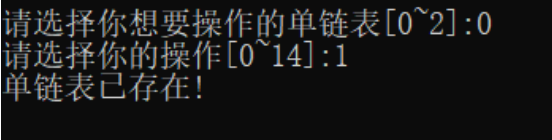
### 1) 初始化表(InitaList)

初始化单链表的正常情况只有一种情况，即待操作的单链表为未初始化的单链表，此时函数将单链表初始化并返回 OK，程序应输出“单链表创建成功”。

异常情况也有一种情况，即待操作的单链表已初始化，此时函数不调用，程序输出“单链表已存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 2-1。

表 2-1 初始化表函数测试

	输入	理论输出	运行结果(截图)
正常输入	0 1 选择第 0 个单链表 (该单链表未初始化)，选择功能 1	输出“单链表创建成功”	
异常输入	0 1 选择第 0 个单链表 (该单链表已初始化)，选择功能 1	输出“单链表已存在”	

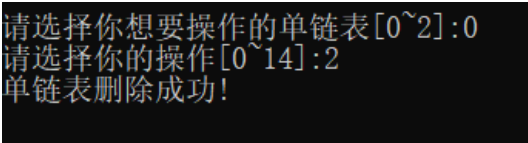
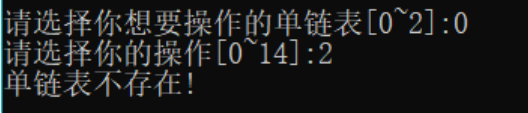
### 2) 销毁单链表(DestroyList)

销毁单链表的正常情况只有一种情况，即待操作的单链表已初始化，此时函数销毁单链表并返回 OK，程序应输出“单链表删除成功”。

异常情况也有一种情况，即待操作的单链表未初始化，此时函数不调用，程序输出“单链表已存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 2-2。

表 2-2 销毁单链表函数测试

	输入	理论输出	运行结果(截图)
正常输入	0 2 选择第 0 个单链表 (已初始化)，选择功能 2	输出“单链表删除成功”	
异常输入	0 2 选择第 0 个单链表 (未初始化)，选择功能 2	输出“单链表不存在”	

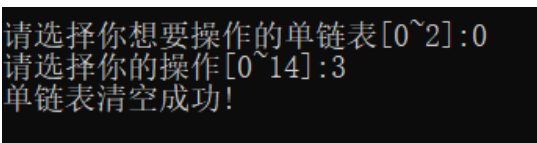
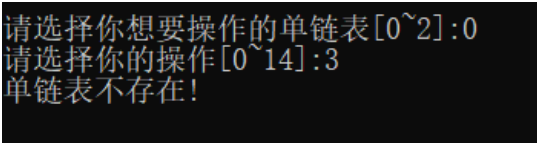
### 3) 清空单链表(ClearList)

清空单链表的正常情况有一种情况，即待操作的单链表已初始化，此时函数清空单链表并返回 OK，程序输出“单链表清空成功”。

异常情况有一种情况，即待操作的单链表未初始化，此时函数不调用，程序输出“单链表已存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 2-3。

表 2-3 清空单链表函数测试

	输入	理论输出	运行结果(截图)
正常输入	0 3 选择第 0 个单链表 (已初始化)，选择 功能 3	输出“单链表清空成功”	
异常输入	0 3 选择第 0 个单链表 (未初始化)，选择 功能 3	输出“单链表不存在”	

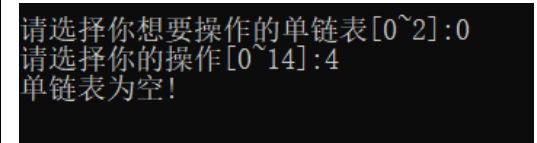
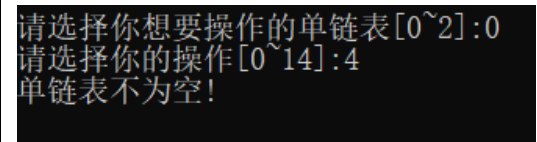
### 4) 判断空表(ListEmpty)

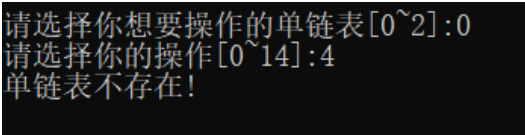
判断空表的正常情况为待操作的单链表已初始化，此时函数判断单链表是否为空，若为空则函数返回 TRUE，程序输出“单链表为空”，否则函数返回 FALSE，程序输出“单链表不为空”，因此测试时分为两种正常情况输入分别测试。

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

具体的测试样例、理论输出结果以及程序实际输出结果见表 2-4。

表 2-4 判断空表功能测试

	输入	理论输出	运行结果(截图)
正常输入 1	0 4 选择第 0 个单链表(已初始化但没有元素)， 选择功能 4	输出“单链表为空”	
正常输入 2	0 4 选择第 0 个单链表(已初始化且含有元素)， 选择功能 4	输出“单链表不为空”	

异常输入	0 4 选择第 0 个单链表(未初始化), 选择功能 4	输出“单链表不存在”	
------	---------------------------------	------------	--

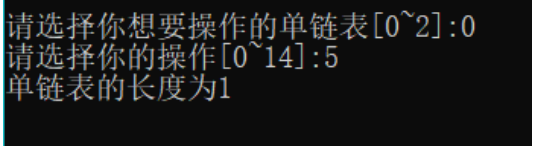
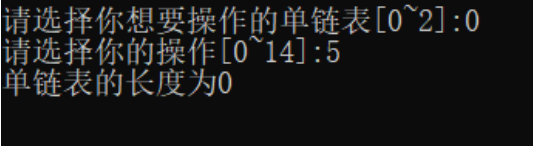
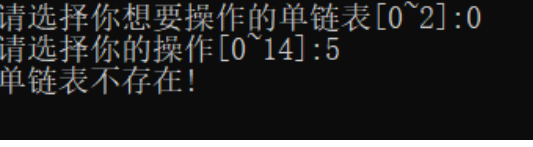
#### 5) 获取单链表表长(ListLength)

获取单链表表长的正常情况为待操作的单链表已初始化, 此时函数获取并返回单链表的表长, 此时在测试时使用空和非空两种单链表进行测试。

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

具体的测试样例、理论输出结果以及程序实际输出结果见表 2-5。

表 2-5 获取单链表表长功能测试

	输入	理论结果	运行结果(截图)
正常输入 1	0 5 选择第 0 个单链表(已初始化且还有 1 个元素), 选择功能 5	输出“单链表长度为 1”	
正常输入 2	0 5 选择第 0 个单链表(已初始化且不含有元素), 选择功能 5	输出“单链表长度为 0”	
异常输入	0 5 选择第 0 个单链表(未初始化), 选择功能 5	输出“单链表不存在”	

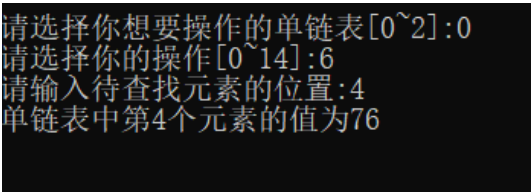
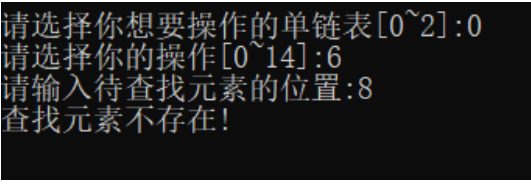
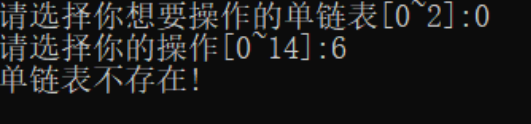
#### 6) 获取单链表元素(GetElem)

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

异常情况有两种情况, 第一种为单链表已存在但待查找位置不存在元素, 此时函数返回 OVERFLOW, 程序输出“查找元素不存在”; 第二种情况为单链表未初始化, 此时函数不调用, 程序输出“单链表不存在”。

具体测试样例、理论输出结果以及程序实际输出结果见表 2-6。本测试中, 当单链表存在时, 单链表的元素为 1423 423 64 76 123。

表 2-6 获取单链表元素功能测试

	输入	理论结果	运行结果(截图)
正常情况	0 6 4 选择第 0 个单链表(已初始化), 选择功能 6, 查找第 4 个元素	输出“单链表中第 4 个元素的值为 76”	
异常情况 1	0 6 8 选择第 0 个单链表(已初始化), 选择功能 6, 查找第 8 个元素	输出“查找元素不存在”	
异常情况 2	0 6 选择第 0 个单链表(未初始化), 选择功能 6	输出“单链表不存在”	

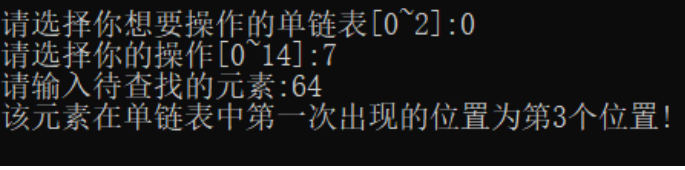
#### 7) 查找元素(LocateElem)

查找元素的正常情况为待操作的单链表已存在且待查找的元素在单链表中, 此时函数返回该元素在单链表中第一次出现的位置, 出现输出该元素在单链表中第一次出现的位置。

因此情况有两种情况, 第一种为单链表存在但待查找元素不在单链表中, 此时函数返回-1, 程序输出“单链表中不存在该元素”; 第二种为待操作的单链表未初始化, 此时函数不调用, 程序输出“单链表不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 2-7。本测试中, 当单链表存在时, 单链表的元素为 1423 324 64 76 123。

表 2-7 查找元素功能测试

	输入	理论结果	运行结果(截图)
正常输入	0 7 64 选择第 0 个单链表(已初始化), 选择功能 7, 查找元素 64	输出“该元素在单链表中第一次出现的位置为第 3 个位置”	



异常输入 1	0 7 12345 选择第 0 个单链表(已初始化), 选择功能 7, 查找元素 12345	输出“单链表中不存在该元素”	
异常输入 2	0 7 选择第 0 个单链表(未初始化), 选择功能 7	输出“单链表不存在”	

#### 8) 查找前驱(PriorElem)

查找前驱的正常情况为待操作的单链表已初始化且待查找前驱的元素在单链表中却该元素不是单链表的第一个元素, 此时函数返回 OK, 并将该元素的前驱存储到指定位置, 程序输出待查找元素的前驱。

异常情况有三种情况, 第一种为待查找元素不在单链表中, 此时函数返回 INFEASTABLE, 程序输出该元素不在单链表中;

第二种情况为待查找前驱的元素为单链表的第一个元素, 此时函数返回-3, 程序输出该元素为单链表的第一个元素;

第三种情况为单链表未初始化, 此时函数不调用, 程序输出“单链表不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 2-8。本测试中, 当单链表存在时, 单链表的元素为 1423 324 64 76 123。

表 2-8 查找前驱功能测试

	输入	理论结果	运行结果(截图)
正常输入	0 8 123 选择第 0 个单链表(已初始化), 选择功能 8, 查找元素 123 的前驱	输出“单链表中元素 123 的前驱为 76”	

异常输入 1	0 8 1234 选择第 0 个单链表 (已初始化), 选择功能 8, 查找元素 1234 的前驱	输出“操作失败, 1234 不在单链表中”	请选择你想要操作的单链表[0~2]:0 请选择你的操作[0~14]:8 请输入待查找前驱的元素:1234 操作失败, 1234不在单链表中!
异常输入 2	0 8 1423 选择第 0 个单链表 (已初始化), 选择功能 8, 查找元素 1423 的前驱	输出“操作失败, 1423 为单链表的第一个元素”	请选择你想要操作的单链表[0~2]:0 请选择你的操作[0~14]:8 请输入待查找前驱的元素:1423 操作失败, 1423为单链表的第一个元素!
异常输入 3	0 8 选择第 0 个单链表 (未初始化), 选择功能 8	输出“单链表不存在”	请选择你想要操作的单链表[0~2]:0 请选择你的操作[0~14]:8 单链表不存在!

#### 9) 查找后继 (NextElem)

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

异常情况有三种情况, 第一种为单链表存在但待查找后继的元素为单链表的最后一个元素, 此时函数返回-3, 程序输出该元素为单链表的最后一个元素;

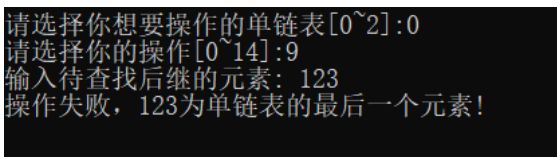
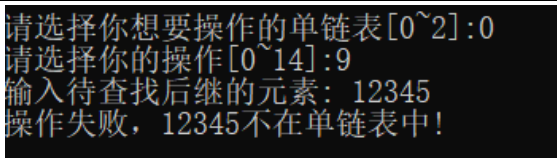
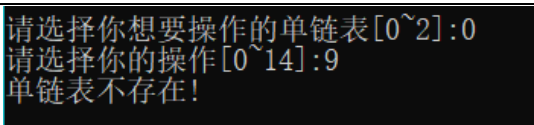
第二种情况为该元素不在单链表中, 此时函数返回 INFEASTABLE, 程序输出该元素不在单链表中;

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

具体的测试样例、理论输出结果以及程序实际输出结果见表 2-9。本测试中, 当单链表存在时, 单链表的元素为 1423 324 64 76 123。

表 2-9 查找后继功能测试

	输入	理论结果	运行结果(截图)
正常输入	0 9 423 选择第 0 个单链表 (已初始化), 选择功能 9, 查找元素 423 的后继	输出“单链表中元素 423 的后继为 64”	请选择你想要操作的单链表[0~2]:0 请选择你的操作[0~14]:9 输入待查找后继的元素: 423 单链表中元素423的后继为64

异常输入 1	0 9 123 选择第 0 个单链表(已初始化)，选择功能 9，查找元素 123 的后继	输出“操作失败，123 位单链表的最后一个元素”	
异常输入 2	0 9 12345 选择第 0 个单链表(已初始化)，选择功能 9，查找元素 12345 的后继	输出“操作失败，12345 不在单链表中”	
异常输入 3	0 9 选择第 0 个单链表(未初始化)，选择功能 9	输出“单链表不存在”	

#### 10) 插入元素(ListInsert)

插入元素的正常情况为单链表已存在且待插入位置可以插入元素，此时函数返回 OK，程序输出“插入成功”。插入成功可分为三种情况进行测试，分别在单链表的首部、尾部以及中间进行插入操作。

异常情况有三种情况。第一种为单链表存在但带插入位置不能够插入元素，此时函数返回-3，程序输出该位置超出单链表范围；

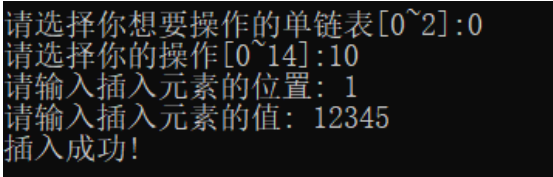
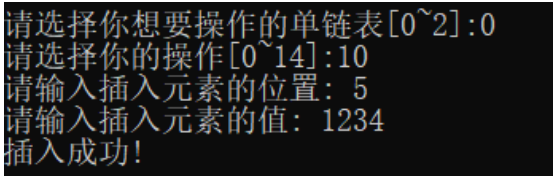
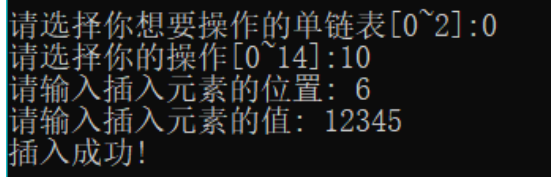
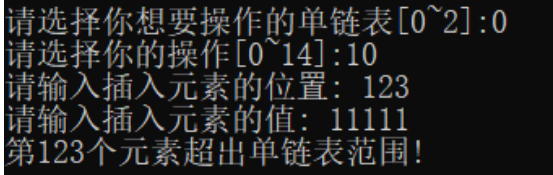
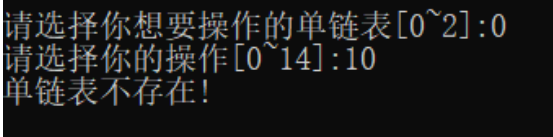
第二种情况为给节点分配空间失败，此时函数返回 OVERFLOW，程序输出“插入失败，空间不足”。

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

具体的测试样例、理论输出结果以及程序实际输出结果见表 2-10。本测试中，在单链表存在时，单链表的元素均为 1423 324 64 76 123。

表 2-10 插入元素功能测试

	输入	理论结果	运行结果(截图)
--	----	------	----------

正常输入 1	0 10 1 12345 选择第 0 个单链表 (已初始化), 选择功能 10, 在第 1 个位置插入元素 12345	输出“插入成功”	 <pre> 请选择你想要操作的单链表[0~2]:0 请选择你的操作[0~14]:10 请输入插入元素的位置: 1 请输入插入元素的值: 12345 插入成功!                     </pre>
正常输入 2	0 10 5 1234 选择第 0 个单链表 (已初始化), 选择功能 10, 在第 5 个位置插入元素 1234	输出“插入成功”	 <pre> 请选择你想要操作的单链表[0~2]:0 请选择你的操作[0~14]:10 请输入插入元素的位置: 5 请输入插入元素的值: 1234 插入成功!                     </pre>
正常输入 3	0 10 6 12345 选择第 0 个单链表 (已初始化), 选择功能 10, 在第 6 个位置插入元素 12345	输出“插入成功”	 <pre> 请选择你想要操作的单链表[0~2]:0 请选择你的操作[0~14]:10 请输入插入元素的位置: 6 请输入插入元素的值: 12345 插入成功!                     </pre>
异常输入 1	0 10 123 11111 选择第 0 个单链表 (已初始化), 选择功能 10, 在第 123 个位置插入元素 11111	输出“第 123 个元素超出单链表范围”	 <pre> 请选择你想要操作的单链表[0~2]:0 请选择你的操作[0~14]:10 请输入插入元素的位置: 123 请输入插入元素的值: 11111 第123个元素超出单链表范围!                     </pre>
异常输入 2	0 10 3 123 选择第 0 个单链表 (已初始化), 选择功能 10, 在第 3 个位置插入元素 123(内存不足的情况)	输出“插入失败, 空间不足”	本系统中单个节点占用内存空间较少, 一般不会出现内存不足的情况
异常输入 3	0 10 选择第 0 个单链表 (未初始化), 选择功能 10	输出“单链表不存在”	 <pre> 请选择你想要操作的单链表[0~2]:0 请选择你的操作[0~14]:10 单链表不存在!                     </pre>

### 11) 删除元素(ListDelete)

删除元素的正常情况为待操作的单链表已初始化且待删除位置存在元素, 此时函数将删除掉的元素存储到指定位置, 并返回 OK, 程序输出“删除成功”并输

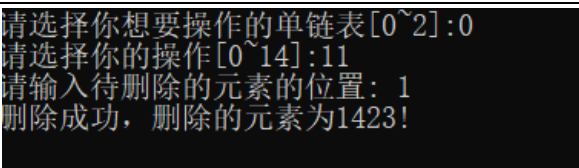
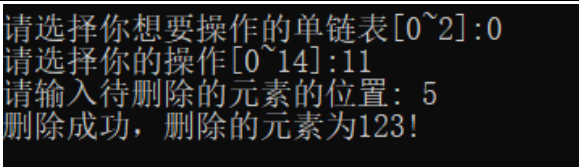
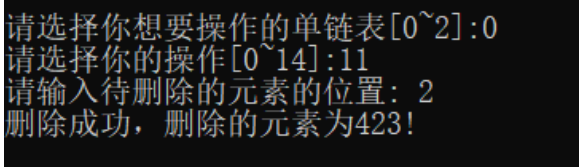
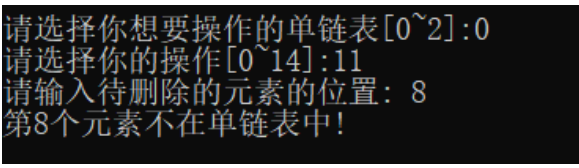
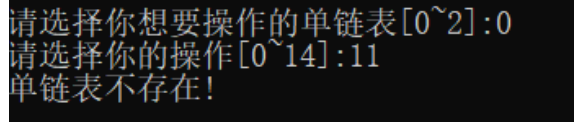
出删除元素的值。成功删除元素可分为三种情况进行测试，分别为删除首元素、尾元素和删除中间的元素。

异常情况有两种情况，第一种为单链表已初始化但是待删除位置没有元素，此时函数返回 OVERFLOW，程序输出该位置不在单链表中；

第二种情况为单链表未初始化，此时函数不调用，程序输出“单链表不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 2-11。本测试中，在单链表存在时，单链表的元素均为 1423 324 64 76 123。

表 2-11 删除元素功能测试

	输入	理论结果	运行结果(截图)
正常输入 1	0 11 1 选择第 0 个单链表(已初始化)，选择功能 11，删除第 1 个元素	输出“删除成功，删除的元素为 1423”	
正常输入 2	0 11 5 选择第 0 个单链表(已初始化)，选择功能 11，删除第 5 个元素	输出“删除成功，删除的元素为 123”	
正常输入 3	0 11 2 选择第 0 个单链表(已初始化)，选择功能 11，删除第 2 个元素	输出“删除成功，删除的元素为 423”	
异常输入 1	0 11 8 选择第 0 个单链表(已初始化)，选择功能 11，删除第 8 个元素	输出“第 8 个元素不在单链表中”	
异常输入 2	0 11 选择第 0 个单链表(未初始化)，选择功能 11	输出“单链表不存在”	

## 12) 遍历单链表(ListTraverse)

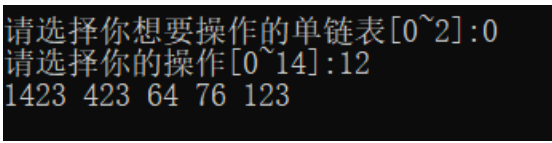
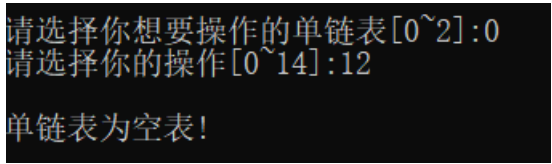
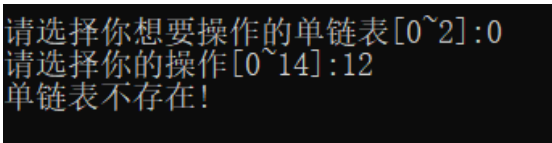
遍历单链表的正常情况为单链表已初始化，此时函数依次答应单链表的元素

并返回单链表的长度，若单链表长度为 0，则程序输出“单链表为空表”。

异常情况为单链表未初始化，此时函数不调用，程序输出“单链表不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 2-12。本测试中，单链表存在且不为空时，单链表的元素为 1423 423 64 76 123。

表 2-12 遍历单链表功能测试

	输入	理论结果	运行结果(截图)
正常 输入 1	0 12 选择第 0 个单链表 (已初始化且不为 空)，选择功能 12	输出“1423 423 64 76 123”	
正常 输入 2	0 12 选择第 0 个单链表 (已初始化但为 空)，选择功能 12	输出“单链 表为空表”	
异常 输入	0 12 选择第 0 个单链表 (未初始化)，选择 功能 12	输出“单链 表不存在”	

### 13) 将单链表存储到文件 (LoadToFile)

将单链表存储到文件的正常情况为待操作单链表已初始化且文件正常打开，此时函数返回 0，程序输出“成功输出到文件”。

异常情况有两种情况，第一种为单链表已初始化但是文件打开失败，此时函数返回-1，程序输出“文件打开失败”；

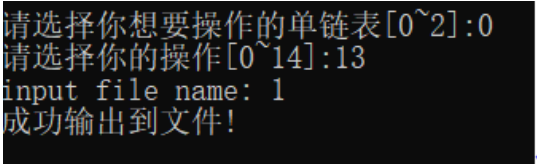
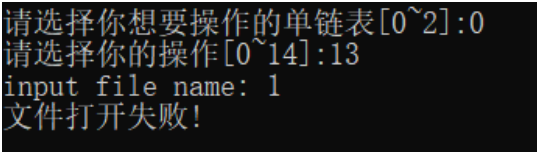
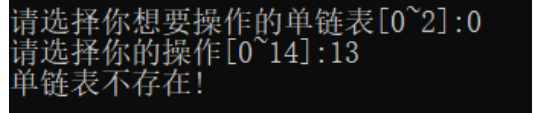
第二种情况为待操作单链表未初始化，此时函数不调用，程序输出“单链表不存在”。

具体测试样例、理论输出结果以及程序实际输出结果见表 2-13。本测试中单链表存在时，单链表的元素为 1423 423 64 76 123。

表 2-13 存储到文件功能测试

	输入	理论结果	运行结果(截图)
--	----	------	----------



正常输入	0 13 1 选择第 0 个单链表(已初始化), 选择功能 13, 将数据存储到文件 1	输出“成功输出到文件”	
异常输入 1	0 13 1 选择第 0 个单链表(已初始化), 选择功能 13, 将数据存储到文件 1	输出“文件打开失败”	
异常输入 2	0 13 选择第 0 个单链表(未初始化), 选择功能 13	输出“单链表不存在”	

#### 14) 从文件读取数据并载入单链表(LoadFromFile)

从文件读取数据并载入单链表的正常情况为单链表已初始化且为空且文件正常打开, 此时函数返回 OK, 此时输出“载入成功”。

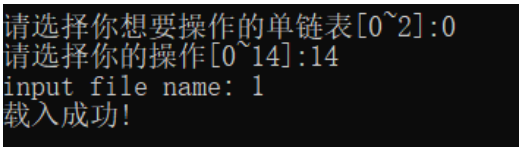
异常情况有三种情况, 第一种为单链表存在但是文件打开失败, 此时函数返回-1, 程序输出“文件打开失败”;

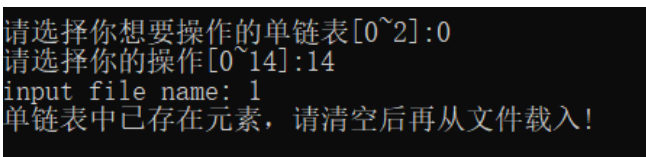
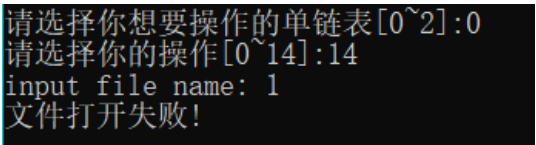
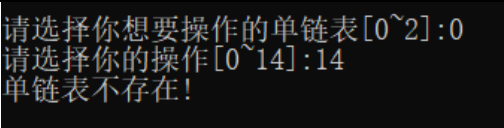
第二种情况为单链表存在且文件成功打开但单链表不为空, 此时函数返回 1, 程序输出“单链表中已存在元素, 请清空后再从文件载入”;

第三种情况为单链表未初始化, 此时函数不调用, 程序输出“单链表不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 2-14。

表 2-14 从文件读取数据功能测试

	输入	理论结果	运行结果(截图)
正常输入	0 14 1 选择第 0 个单链表(已初始化且没有元素), 选择功能 14, 从文件 1(存在)中读取数据	输出“载入成功”	

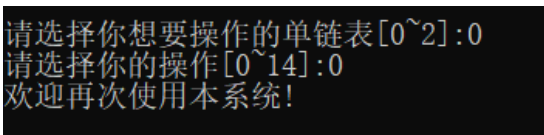
异常输入 1	0 14 1 选择第 0 个单链表(已初始化且含有元素), 选择功能 14, 从文件 1 读取数据	输出“单链表中已存在元素, 请清空后再从文件载入”	
异常输入 2	0 14 1 选择第 0 个单链表(已初始化且没有元素), 选择功能 14, 从文件 1(不存在)读取数据	输出“文件打开失败”	
异常输入 3	0 14 选择第 0 个单链表(未初始化), 选择功能 14	输出“单链表不存在”	

#### 15) 退出程序

退出程序的功能与单链表的状态无关, 均为正常情况。

具体的测试样例、理论输出结果以及程序实际输出结果见表 2-15。

表 2-15 退出程序功能测试

	输入	理论结果	运行结果(截图)
正常输入	0 0 选择第 0 个单链表, 选择功能 0	输出“欢迎再次使用本系统”	

### 2.3.4 其他问题的说明

在测试环境(Windows 10 64 位)中, 若可执行文件在 C 盘根目录下, 则无法使用 LoadToFile 函数将线性表内容写入到不存在的文件中, 原因为: 程序没有权限在 C 盘根目录下创建文件。若可执行文件位于 C 盘根目录下, 则因使用管理员权限运行该程序, 才能够正常使用文件 I/O 功能。



## 2.4 实验小结

本实验与实验一功能相同，但是使用的数据结构不同导致两次实验完成同样的工作时使用的算法有较大区别。从这两次实验中，我更加深刻的理解了线性表的顺序存储结构和链式存储结构，对两者各自的优缺点有了更加直观的认识。相比之下，顺序表比链式表要节省空间，且随机存取更加方便，而链式表在插入删除元素等操作上要更加方便。

在本次实验中，我还对单链表的头结点的作用有了直观的认识。

另外，本次试验中，编写各个函数以及计算时间复杂度时，让我了解到其他类型的链表，例如双向链表以及循环链表，在使用对操作的简化以及对时间复杂度的优化。

## 3 基于二叉链表的二叉树实现

### 3.1 问题描述

二叉树是一种树形结构，二叉树的每个节点至多有两棵子树。二叉树的二叉链表表示指数据元素在物理地址上不相邻的二叉树，而是通过附加指针域的方式提供逻辑上的关系。本实验要求利用二叉链表结构以函数形式实现二叉树的 21 个基本操作。这 21 个基本操作如下。

(1) 初始化二叉树：函数名称是 `InitBiTree(T)`；初始条件是二叉树 `T` 不存在；操作结果是构造空二叉树 `T`。

(2) 销毁二叉树：函数名称是 `DestroyBiTree(T)`；初始条件是二叉树 `T` 已存在；操作结果是销毁二叉树 `T`。

(3) 创建二叉树：函数名称是 `CreateBiTree(T, definition, length)`；初始条件是 `definition` 给出二叉树 `T` 的定义；操作结果是按 `definition` 构造二叉树 `T`。

(4) 清空二叉树：函数名称是 `ClearBiTree (T)`；初始条件是二叉树 `T` 存在；操作结果是将二叉树 `T` 清空。

(5) 判定空二叉树：函数名称是 `BiTreeEmpty(T)`；初始条件是二叉树 `T` 存在；操作结果是若 `T` 为空二叉树则返回 `TRUE`，否则返回 `FALSE`。

(6) 求二叉树深度：函数名称是 `BiTreeDepth(T)`；初始条件是二叉树 `T` 存在；操作结果是返回 `T` 的深度。

(7) 获得根结点：函数名称是 `Root(T)`；初始条件是二叉树 `T` 已存在；操作结果是返回 `T` 的根。

(8) 获得结点：函数名称是 `Value(T, e)`；初始条件是二叉树 `T` 已存在，`e` 是 `T` 中的某个结点；操作结果是返回 `e` 的值。

(9) 结点赋值：函数名称是 `Assign(T, e, value)`；初始条件是二叉树 `T` 已存在，`e` 是 `T` 中的某个结点；操作结果是结点 `e` 赋值为 `value`。

(10) 获得双亲结点：函数名称是 `Parent(T, e)`；初始条件是二叉树 `T` 已存

在,  $e$  是  $T$  中的某个结点; 操作结果是若  $e$  是  $T$  的非根结点, 则返回它的双亲结点指针, 否则返回 NULL。

(11) 获得左孩子结点: 函数名称是  $\text{LeftChild}(T, e)$ ; 初始条件是二叉树  $T$  存在,  $e$  是  $T$  中某个节点; 操作结果是返回  $e$  的左孩子结点指针。若  $e$  无左孩子, 则返回 NULL。

(12) 获得右孩子结点: 函数名称是  $\text{RightChild}(T, e)$ ; 初始条件是二叉树  $T$  已存在,  $e$  是  $T$  中某个结点; 操作结果是返回  $e$  的右孩子结点指针。若  $e$  无右孩子, 则返回 NULL。

(13) 获得左兄弟结点: 函数名称是  $\text{LeftSibling}(T, e)$ ; 初始条件是二叉树  $T$  存在,  $e$  是  $T$  中某个结点; 操作结果是返回  $e$  的左兄弟结点指针。若  $e$  是  $T$  的左孩子或者无左兄弟, 则返回 NULL。

(14) 获得右兄弟结点: 函数名称是  $\text{RightSibling}(T, e)$ ; 初始条件是二叉树  $T$  已存在,  $e$  是  $T$  中某个结点; 操作结果是返回  $e$  的右兄弟结点指针。若  $e$  是  $T$  的右孩子或者无右兄弟, 则返回 NULL。

(15) 插入子树: 函数名称是  $\text{InsertChild}(T, p, LR, c)$ ; 初始条件是二叉树  $T$  存在,  $p$  指向  $T$  中的某个结点,  $LR$  为 0 或 1, 非空二叉树  $c$  与  $T$  不相交且右子树为空; 操作结果是根据  $LR$  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。

(16) 删除子树: 函数名称是  $\text{DeleteChild}(T, p, LR)$ ; 初始条件是二叉树  $T$  存在,  $p$  指向  $T$  中的某个结点,  $LR$  为 0 或 1。操作结果是根据  $LR$  为 0 或者 1, 删除  $T$  中  $p$  所指结点的左或右子树。

(17) 前序遍历: 函数名称是  $\text{PreOrderTraverse}(T, \text{Visit})$ ; 初始条件是二叉树  $T$  存在,  $\text{Visit}$  是对结点操作的应用函数; 操作结果: 先序遍历  $T$ , 对每个结点调用函数  $\text{Visit}$  一次且一次, 一旦调用失败, 则操作失败。

(18) 中序遍历: 函数名称是  $\text{InOrderTraverse}(T, \text{Visit})$ ; 初始条件是二叉树  $T$  存在,  $\text{Visit}$  是对结点操作的应用函数; 操作结果是中序遍历  $T$ , 对每个结点调用函数  $\text{Visit}$  一次, 一旦调用失败, 则操作失败。

(19) 后序遍历: 函数名称是  $\text{PostOrderTraverse}(T, \text{Visit})$ ; 初始条件是二

叉树 T 存在, Visit 是对结点操作的应用函数; 操作结果是后序遍历 T, 对每个结点调用函数 Visit 一次, 一旦调用失败, 则操作失败。

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

(21) 按树形遍历: 函数名称为 TreeOrderTraverse(T, Visit, n); 初始条件是二叉树 T 存在, Visit 是对结点操作的应用函数; 操作结果是将二叉树 T 参照 Windows 文件资源管理器的样式输出到控制台。

本系统还实现了对多个二叉树操作的支持。

本系统还提供对二叉树的文件输入输出功能。本功能通过两个函数实现, 两个函数的基本情况如下。

1) 保存到文件: 函数名称为 LoadToFile(T, filename); 初始条件为二叉树 T 已存在, 操作结果是将二叉树 T 的含空的前序序列逐个存储到文件 filename 中。

2) 从文件载入数据到二叉树: 函数名称为 LoadFromFile(T, filename); 初始条件为二叉树 T 已存在且不包含元素, 操作结果为从文件中 filename 中读取数据并将这些数据创建为二叉树 T。

## 3.2 系统设计

### 3.2.1 总控流程框架

本系统首先由用户输入待操作的二叉树的数量 n, 然后创建 n 个二叉树, 然后进入一个循环, 循环中, 首先由用户输入待操作的二叉树的序号, 然后将二叉树的操作指针指向用户选择的二叉树, 然后由用户输入对该二叉树进行的操作的序号, 然后根据用户输入的操作序号进入不同的分支、进行相应的操作, 直到用户输入 0, 循环结束, 算法结束。该算法的流程图如图 3.1 所示。

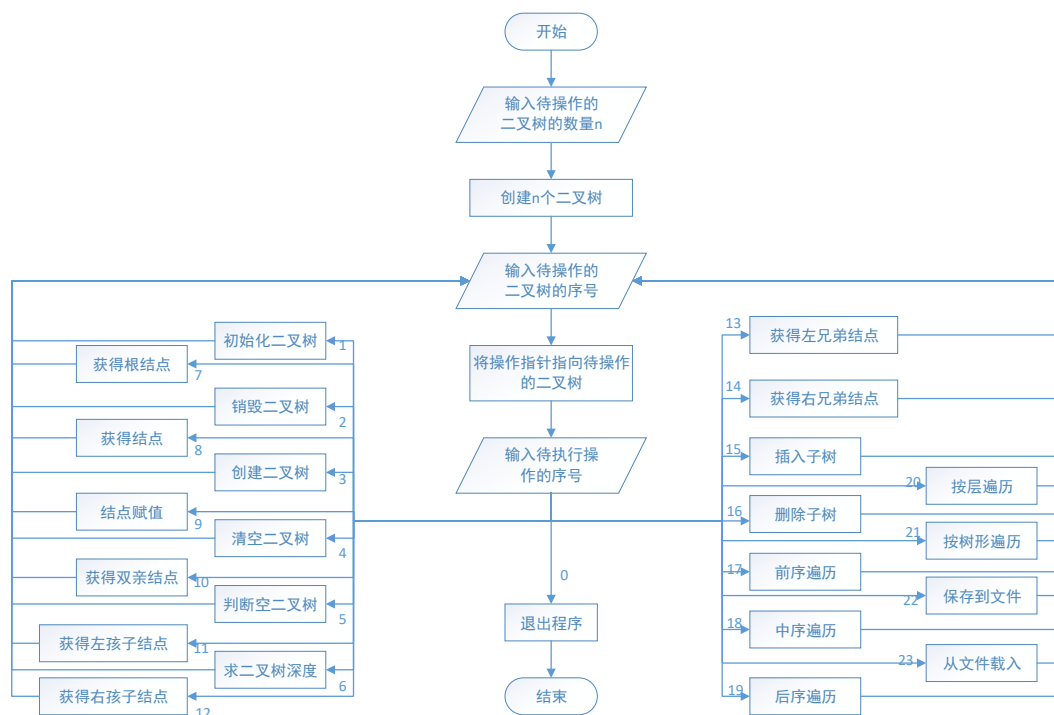


图 3.1 总控流程框架流程图

### 3.2.2 定义常量、数据类型及数据结构

1) 定义常量。定义常量作为函数返回值用于说明函数执行状态。定义的常量有 TRUE(1), FALSE(0), OK(1), ERROR(0), INFEASTABLE(-1), OVERFLOW(-2)。

2) 定义数据元素的类型。此处定义了数据类型 status 用来说明函数的执行状态, ElemType 用来作为二叉树的结点数据域, TreeHead 用来作为二叉树的头结点, statusNode 用来作为需要返回结点指针的函数返回值类型。本系统中将 status 定义为 int 类型, ElemType 为由一个 int 类型的 ID 和一个字符串 name 组成的结构体, TreeHead 为由一个 int 类型的 exist 和一个指向二叉树的指针 child 组成的结构体, statusNode 为由一个 status 类型的 s 和一个指向二叉树节点的指针 node 组成的结构体。

3) 定义二叉树的数据结构。二叉树结构体包含三个元素, 分别为 ElemType 类型的数据域、指向该结点左孩子的指针 lchild 和指向该结点右孩子的指针 rchild。并将其定义为 TreeNode。

二叉树在内存中的存储结构如图 3.2 所示。图中操作指针指向第一个二叉树的头结点, 每个 TreeHead 及其指向的所有 TreeNode 构成一个二叉树, 该二叉树

的各节点之间的关系在图中通过箭头指向表明。

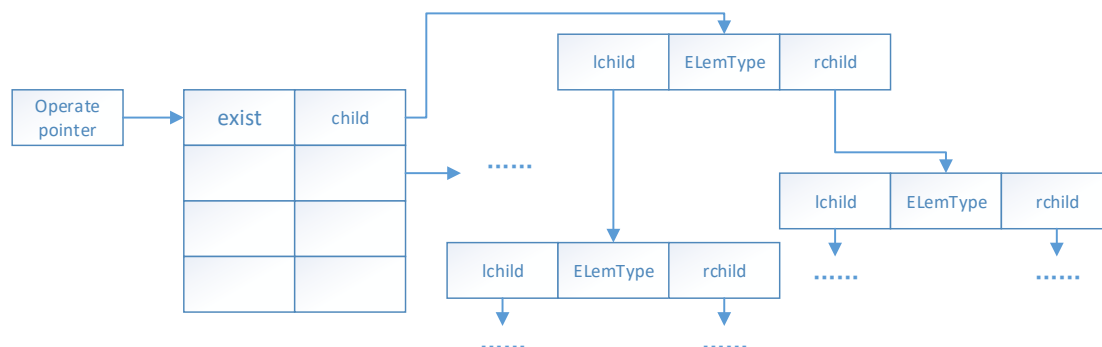


图 3.2 二叉链表在内存中的存储方式及其多树表示

### 3.2.3 21 个基本操作的设计

关于函数调用方法的说明：初始化二叉树函数(InitBiTree)接受的二叉树参数应为未初始化的二叉树，否则不调用该函数并提醒用户“二叉树已存在”。其余 20 个基本操作的函数所接受的二叉树参数应为已初始化的二叉树，否则不调用该函数并提醒用户“二叉树不存在”。

#### 1) 初始化二叉树(InitBiTree)

该函数接受一个 TreeHead 类型的指针 t，该指针指向待初始化的二叉树的头结点。函数将 t 的元素 exist 赋值为 1，表明该二叉树已初始化，将 t 的元素 child 赋值为 NULL，并返回 OK。

该函数只有顺序结构，因此时间复杂度为  $O(1)$ 。

#### 2) 销毁二叉树(DestroyBiTree)

该函数接受一个 TreeHead 类型的指针 t，该指针指向待销毁的二叉树的头结点。函数首先清空二叉树(使用 ClearBiTree 函数，其算法设计见函数 4)，然后将 t 的元素 exist 赋值为 0、元素 child 赋值为 NULL，并返回 OK。

该函数调用函数 ClearBiTree，余下部分均为顺序结构，由于函数 ClearBiTree 的时间复杂度为  $O(n)$  (该函数的时间复杂度分析见函数 4)，故本函数的时间复杂度为  $O(n)$ 。

#### 3) 创建二叉树(CreateBiTree)

该函数接受一个 TreeNode 类型的指针的指针 rootNode，一个 ElemType 类型的指针 definition 和一个整型的 length。参数 rootNode 指向一个指向待创

建的二叉树的根结点的指针，参数 definition 指向一个存储有二叉树含空前序序列的数组，参数 length 为 definition 数组的长度。

该函数使用递归实现。递归中使用一个全局变量 CreateBiTreeLevel 来控制当前读取 definition 的位置，在调用函数前将 CreateBiTreeLevel 置为 0。递归过程中，当当前读取 definition 中的元素为-1 时，将 CreateBiTreeLevel 自增 1，并返回 OK，否则，给 rootNode 指向的指针分配一段长度为 TreeNode 的内存空间，将当前读取到的 definition 的元素赋值给 rootNode 所指向的指针指向的结点的数据域，并将该结点的两个孩子指针置为 NULL。然后调用函数 CreateBiTree 创建 rootNode 的左子树，然后调用函数 CreateBiTree 创建 rootNode 的右子树。最后返回 OK，表明函数执行完成，创建二叉树成功。

该函数会对二叉树的每个结点做逐个的创建，因此函数的时间复杂度  $O(n)$ 。

#### 4) 清空二叉树(ClearBiTree)

该函数接受一个 TreeNode 类型的指针，该指针指向待清空二叉树的根结点。该函数使用递归方式实现，若根结点为空，则函数返回 OK，否则函数先清空二叉树的左子树，再清空二叉树的右子树，然后将根结点所占用的内存释放。最后函数返回 OK，表明二叉树所有结点全部被释放。

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

#### 5) 判断空二叉树(BiTreeEmpty)

该函数接受一个 TreeHead 类型的指针 t，该指针指向待判断的二叉树的头结点，若 t 的元素 child 为 NULL，则说明二叉树为空，函数返回 TRUE，否则说明二叉树不为空，函数返回 FALSE。

该函数只包含顺序和选择结构，因此函数的时间复杂度为  $O(1)$ 。

#### 6) 求二叉树的深度(BiTreeDepth)

该函数接受一个 TreeNode 类型的指针 rootNode，该指针指向待求深度的二叉树的根结点。该函数使用递归实现。若参数 rootNode 为空，说明该二叉树的深度为 0，函数返回 0，否则返回二叉树的左子树的深度和右子树的深度的最大值加 1。

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

#### 7) 获得根结点(Root)

该函数接受一个 TreeHead 类型的指针 t，该指针指向二叉树的头结点。函数返回 t 的元素 child。

该函数只包含顺序结构，因此函数的时间复杂度为  $O(1)$ 。

#### 8) 获得结点(Value)

该函数接受两个参数，第一个为 TreeNode 类型的指针 node，该指针指向待操作的二叉树的根结点，第二个参数为整型的 e，为待获取结点的 ID 值。函数调用函数 TreeArrayLength 获得二叉树的含空长度 TreeLength，然后函数调用函数 LoadToArray 将二叉树的每个结点指针存储到数组 treeArray 中。然后函数遍历 treeArray 数组，直到找到其中指向的结点的数据域的 ID 为 e 时，终止循环，并返回该指针。若遍历整个数组都未找到带查找结点，则函数返回 NULL。

该函数调用了函数 TreeArrayLength 和函数 LoadToArray，这两个函数的时间复杂度均为  $O(n)$ ，函数的余下部分包含一个循环结构，在最优的情况下，即待查找结点为二叉树的根结点时，循环执行 1 次，在最坏的情况下，即待查找结点为二叉树的最后一个结点或待查找结点不存在时，循环执行 n 次，故循环平均执行  $(n+1)/2$  次，故函数的时间复杂度为  $O(n)$ 。

#### 9) 结点赋值(Assign)

该函数接受三个参数，第一个为 TreeNode 类型的指针 node，该指针指向待操作的二叉树的根结点，第二个参数为整型的 e，为待赋值的结点的 ID，第三个参数为字符串 value，为待赋给结点 e 的值。函数首先调用函数 Value 获得二叉树中 ID 为 e 的结点的指针 assignNode，若 assignNode 为 NULL，说明二叉树中不存在该结点，函数返回 ERROR，否则将 value 赋值给 assignNode 的数据域的名称元素，并返回 OK。

该函数调用了函数 Value，函数 Value 的时间复杂度为  $O(n)$ ，余下部分只有顺序结构和选择结构，因此函数的时间复杂度为  $O(n)$ 。

#### 10) 获得双亲结点(Parent)

该函数接受两个参数，第一个为 TreeNode 类型的指针 node，该指针指向待操作二叉树的根结点，第二个参数为整型的 e，为待查找双亲的结点的 ID。首先



声明一个 `statusNode` 类型的 `result` 用来作为函数返回值。若二叉树的根结点的 ID 为 `e`，则将 `result` 的元素 `s` 置为 -1，元素 `node` 置为 `NULL`，说明结点 `e` 为二叉树的根结点，不存在双亲结点，并返回 `result`。否则使用 `TreeArrayLength` 函数获得二叉树的含空长度 `TreeLength`，调用函数 `LoadToArray` 将二叉树的所有结点指针存储到数组 `treeArray` 中。然后进入一个循环，直到找到某个结点的左孩子或者右孩子的 ID 为 `e` 时，将 `result` 的元素 `s` 置为 `OK`，元素 `node` 置为该结点的指针，并返回 `result`。若遍历完整个数组都未找到该结点，则将 `result` 的元素 `s` 置为 `ERROR`，元素 `node` 置为 `NULL`，并返回 `result`。

该函数在时间最优的情况下，即待获得双亲结点的结点为二叉树的根结点时，函数不调用函数 `TreeArrayLength` 和 `LoadToArray`，也不执行循环结构，此时时间最短，与二叉树的规模无关，否则，函数调用函数 `TreeArrayLength` 和函数 `LoadToArray`，这两个函数的时间复杂度均为  $O(n)$ ，余下部分包含一个循环结构，在当前情况的最优情况下，即结点 `e` 为根结点的孩子结点时，此时循环只执行一次，在当前情况的最坏情况下，即结点 `e` 为二叉树最靠右的孩子结点或二叉树中不存在 ID 为 `e` 的结点时，此时循环执行  $n$  次，因此循环平均执行  $(n+1)/2$  次，故函数的时间复杂度为  $O(n)$ 。

#### 11) 获得左孩子结点 (LeftChild)

该函数接受两个参数，第一个为 `TreeNode` 类型的指针 `node`，该指针指向待操作的二叉树的根结点，第二个参数为整型的 `e`，为待获得左孩子结点的结点 ID。函数首先声明一个 `statusNode` 类型的变量 `result`。函数调用 `Value` 函数获得待获得左孩子结点的结点的指针 `findChildNode`。若 `findChildNode` 为空，说明该结点不存在，将 `result` 的元素 `s` 置为 `ERROR`，元素 `node` 置为 `NULL`，并返回 `result`；否则将 `result` 的元素 `s` 置为 `OK`，元素 `node` 置为 `findChildNode` 的左孩子指针，并返回 `result`。

该函数调用了函数 `Value`，该函数的时间复杂度为  $O(n)$ ，余下部分仅包含顺序和选择结构，因此函数的时间复杂度为  $O(n)$ 。

#### 12) 获得右孩子结点 (RightChild)

该函数接受两个参数，第一个为 `TreeNode` 类型的指针 `node`，该指针指向待

操作的二叉树的根结点,第二个参数为整型的  $e$ ,为带获得右孩子结点的结点 ID。函数首先声明一个 `statusNode` 类型的变量 `result`。函数调用 `Value` 函数获得待获得右孩子结点的结点的指针 `findChildNode`。若 `findChildNode` 为空,说明该结点不存在,将 `result` 的元素 `s` 置为 `ERROR`,元素 `node` 置为 `NULL`,并返回 `result`;否则将 `result` 的元素 `s` 置为 `OK`,元素 `node` 置为 `findChildNode` 的右孩子指针,并返回 `result`。

该函数调用了函数 `Value`,该函数的时间复杂度为  $O(n)$ ,余下部分仅包含顺序和选择结构,因此函数的时间复杂度为  $O(n)$ 。

### 13) 获得左兄弟结点(LeftSibling)

该函数接受两个参数,第一个为 `TreeNode` 类型的指针 `node`,该指针指向待操作的二叉树的根结点,第二个参数为整型的  $e$ ,为待获得左兄弟结点的结点 ID。函数首先声明一个 `statusNode` 类型的变量 `result`。函数调用 `Parent` 函数获得待获得左兄弟结点的双亲结点的存在状态及其指针 `parentNode`。若 `parentNode` 的元素 `s` 为 `ERROR`,则双亲结点不存在,若元素 `s` 为 `-1`,则结点  $e$  为二叉树的根结点,这两种情况下,函数直接返回 `parentNode`,否则,若 `parentNode` 的元素 `node` 的左孩子指针为空,说明该结点的左兄弟结点不存在,将 `result` 的元素 `s` 置为 `-3`,元素 `node` 置为 `NULL`,若 `parentNode` 的元素 `node` 的左孩子指针指向的结点的 ID 为  $e$ ,说明该结点为其双亲结点的左孩子,则该结点没有左兄弟,将 `result` 的元素 `s` 置为 `-2`,元素 `node` 置为 `NULL`,否则,该结点的左兄弟存在,将 `result` 的元素 `s` 置为 `OK`,元素 `node` 置为 `parentNode` 的元素 `node` 的左孩子指针。最后,函数返回 `result`。

该函数调用了函数 `Parent`,该函数的时间复杂度为  $O(n)$ ,余下部分仅包含顺序和选择结构,因此函数的时间复杂度为  $O(n)$ 。

### 14) 获得右兄弟结点(RightSibling)

该函数接受两个参数,第一个为 `TreeNode` 类型的指针 `node`,该指针指向待操作的二叉树的根结点,第二个参数为整型的  $e$ ,为待获得右孩子结点的结点 ID。函数首先声明一个 `statusNode` 类型的变量 `result`,函数调用 `Parent` 函数获得待获得右兄弟结点的双亲结点的存在状态及其指针 `parentNode`。若 `parentNode`

的元素  $s$  为 ERROR, 则双亲结点不存在, 若元素  $s$  为 -1, 则结点  $e$  为二叉树的根结点, 这两种情况下, 函数直接返回  $parentNode$ , 否则, 若  $parentNode$  的元素  $node$  的右孩子指针为空, 说明该结点的右兄弟结点不存在, 将  $result$  的元素  $s$  置为 -3, 元素  $node$  置为 NULL, 若  $parentNode$  的元素  $node$  的右孩子指针指向的结点的 ID 为  $e$ , 说明该结点为其双亲结点的右孩子, 则该结点没有右兄弟, 将  $result$  的元素  $s$  置为 -2, 元素  $node$  置为 NULL, 否则, 该结点的右兄弟存在, 将  $result$  的元素  $s$  置为 OK, 元素  $node$  置为  $parentNode$  的元素  $node$  的右孩子指针。最后, 函数返回  $result$ 。

该函数调用了函数 Parent, 该函数的时间复杂度为  $O(n)$ , 余下部分仅包含顺序和选择结构, 因此函数的时间复杂度为  $O(n)$ 。

#### 15) 插入子树 (InsertChild)

该函数接受四个参数, 第一个为 TreeHead 类型的指针  $t$ , 该指针指向待操作的二叉树的头结点, 第二个参数为整型的  $e$ , 为待插入子树的结点的 ID, 第三个参数为整型的 LR, 为待插入子树的插入位置, 第四个参数为 TreeHead 类型的指针  $c$ , 该指针指向待插入的子树的头结点。

函数首先调用函数 Value 获得二叉树  $t$  中结点 ID 为  $e$  的结点的指针  $node$ , 若  $node$  为空, 则返回 -1, 说明二叉树中不存在结点  $e$ ; 若待插入的子树  $c$  的根结点的右子树不为空, 则返回 ERROR, 说明不允许插入该二叉树; 否则, 二叉树可以执行插入操作, 若 LR 为 0, 则将结点  $node$  的左子树指针赋值给二叉树  $c$  的根结点的右孩子指针, 并将结点  $node$  的左孩子指针赋值为二叉树  $c$  的根结点指针; 若 LR 为 1, 则将结点  $node$  的右子树指针赋值给二叉树  $c$  的根结点的右孩子指针, 并将结点  $node$  的右孩子指针赋值为二叉树  $c$  的根结点指针。最后函数返回 OK。

该函数调用函数 Value, 该函数的时间复杂度为  $O(n)$ 。余下部分仅包含顺序和选择结构, 因此函数的时间复杂度为  $O(n)$ 。

#### 16) 删除子树 (DeleteChild)

该函数接受三个参数, 第一个为 TreeHead 类型的指针  $t$ , 该指针指向待操作的二叉树的头结点, 第二个参数为整型的  $e$ , 为待删除子树的结点 ID, 第三个

参数为整型的 LR，为待删除子树的位置。

函数首先调用函数 Value 获得二叉树 t 中 ID 为 e 的结点指针 node，若 node 为空，说明二叉树 t 中不存在 ID 为 e 的结点，函数返回 ERROR；否则，二叉树可以执行删除子树的操作，若 LR 为 0，则调用函数 ClearBiTree 将 node 的左子树清空，并将 node 的左孩子指针置为 NULL，若 LR 为 1，则调用函数 ClearBiTree 将 node 的右子树清空，并将 node 的右孩子指针置为 NULL。最后函数返回 OK。

该函数调用了函数 Value，该函数的时间复杂度为  $O(n)$ ，还调用了函数 ClearBiTree，该函数执行的时间与待删除子树的结点位置有关，若该结点为二叉树 t 的叶子结点，则删除了 0 个结点，函数执行时间最短，若该结点为二叉树 t 的根结点，则函数最多清空二叉树 t 中除根节点外的所有结点，即  $n-1$  个结点，此时函数执行时间最长，因此函数的时间复杂度为  $O(n)$ 。

#### 17) 前序遍历 (PreOrderTraverse)

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

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

#### 18) 中序遍历 (InOrderTraverse)

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

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

#### 19) 后序遍历 (PostOrderTraverse)

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

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

#### 20) 按层遍历 (LevelOrderTraverse)

该函数接受两个参数，第一个为 `TreeNode` 类型的指针 `rootNode`，指向待操作二叉树的根结点，第二个为返回值为整型、参数为 `TreeNode` 的指针类型的函数指针 `Visit`。该函数首先调用函数 `TreeArrayLength` 和函数 `LoadToArray` 得到二叉树的含空长度 `LoadToArrayCount` 和将二叉树存储到数组的结果 `treeArray`。然后函数调用函数 `BiTreeDepth` 得到二叉树的深度 `depth`，然后函数遍历数组 `treeArray` 并将每个节点所在的层数存储到 `count` 中。然后函数将层数从  $1 \sim \text{depth}$  依次输出即得到二叉树的按层遍历的结果。

该函数开始调用了函数 `TreeArrayLength` 和函数 `LoadToArray`，这两个函数的时间复杂度均为  $O(n)$ ，函数在计算每个元素所处的层数时，使用了一个循环结构，循环次数为  $n$  ( $n$  为二叉树的含空结点数目)，循环体中包含一个 `Parent` 函数，和一个选择结构中的循环结构，该循环同样执行  $n$  次，故获得该结点的层数的部分的时间复杂度为  $O(n^2)$ ，按层输出的部分包含一个循环结构，该循环次数为 `depth`，由二叉树的性质可知， $O(\text{depth}) = O(\log n)$ ，该循环中包含另一个循环结构，该循环结构执行  $n$  次，故输出部分的时间复杂度为  $O(n \log n)$ 。综上，按层遍历函数的时间复杂度为  $O(n^2)$ 。

#### 21) 树形输出 (TreeOrderTraverse)

该函数接受三个参数，第一个为 `TreeNode` 类型的指针 `rootNode`，指向待操作二叉树的根结点，第二个为返回值为整型、参数为 `TreeNode` 的指针类型的函数指针 `Visit`，第三个参数为辅助计数的参数 `n`。该函数使用递归实现，若根结点 `rootNode` 为空，则函数返回 `OK`，否则，函数首先输出  $n$  个 `\t` 字符，然后对

rootNode 调用 Visit 函数，然后对 rootNode 的左子树进行按树形遍历的操作，然后对 rootNode 的右子树进行按树形遍历的操作，每次对下一层进行递归时，传递的参数 n 均比上一层大 1，用于控制不同层数中输出不同数量的\t，从而形成树状的输出结果。

该函数对二叉树的每个结点调用 Visit 函数，该函数的时间复杂度为  $O(1)$ ，而函数对第 i 层会循环输出 i 个\t 字符，因此函数的时间复杂度为  $O(n \log n)$ 。

### 3.2.4 多二叉树操作的实现

本系统首先由用户输入待操作的二叉树的数量，然后每次由用户选择待操作的二叉树并对其进行操作。

程序从用户获取到待操作的二叉树的数量 numTree 以后，分配一段长度为 TreeHead 的长度的 numTree 倍的空间，用于存储 numTree 个二叉树的头指针，并保存该内存空间的首地址。

在对二叉树进行操作的过程中，用户选择待操作的二叉树的序号，用户选定一个二叉树后，将操作指针移动相应的长度后即可指向待操作的二叉树的头指针，这样使用与对单个二叉树的操作相同时的操作来对操作指针指向的二叉树进行操作，从而实现多二叉树操作的功能。

多二叉树的存储方式如图 3.2 所示，图中每个 TreeHead 代表一个二叉树，用户选择一个二叉树为待操作的二叉树后，将操作指针指向该二叉树从而对该二叉树进行操作。

### 3.2.5 文件输入输出的设计

本系统对文件输入输出功能的实现通过两个函数，LoadToFile 和 LoadFromFile 实现，其中 LoadToFile 将当前操作的二叉树存储到文件中，LoadFromFile 将文件的内容读取并存储到当前操作的二叉树中。两个函数的设计如下。

#### 1) 将二叉树存储到文件(LoadToFile)

该函数接受两个参数，第一个为 TreeNode 类型的指针 rootNode，该指针指向待存储到文件的二叉树的根结点，第二个参数为指向字符类型的指针

filename, 该指针指向待存储二叉树的文件的文件名字符串的首地址。函数首先尝试打开文件 filename, 若文件打开失败, 则函数返回 1, 否则, 调用函数 TreeArrayLength 和函数 LoadToArray, 得到二叉树的含空长度 LoadToArrayCount 和将二叉树存储到数组得到的数组的首地址 treeArray。然后函数进入一个循环结构, 循环执行 LoadToArrayCount 次, 依次将 treeArray 中的每一个元素的 ID 和 name 写入到文件中, 全部写入完成后, 关闭文件并返回 0。

该函数调用了函数 TreeArrayLength 和 LoadToArray, 这两个函数的时间复杂度均为  $O(n)$ , 余下部分包含一个循环结构, 该循环体中仅包含顺序结构, 循环执行  $n$  次, 因此函数的时间复杂度为  $O(n)$ 。

## 2) 从文件读取数据并载入二叉树 (LoadFromFile)

该函数接受两个参数, 第一个为 TreeNode 类型的指针的指针 rootNode, 该指针指向一个指向待操作二叉树的根结点的指针, 第二个参数为指向字符类型的指针 filename, 该指针指向待读取文件的文件名字符串的首地址。函数首先尝试打开文件 filename, 若文件打开失败, 则函数返回 1。否则, 函数将文件中的数据逐个读取并存储到 ElemType 类型的数组 definition 中, 然后调用函数 CreateBiTree, 使用 definition 中的数据对二叉树 rootNode 创建二叉树, 然后函数释放 definition 占用的内存并返回 0。

该函数从文件中逐个读取二叉树的每个结点, 该循环需执行  $n$  次, 函数还调用了 CreateBiTree 函数, 该函数的时间复杂度为  $O(n)$ , 因此函数 LoadFromFile 的时间复杂度为  $O(n)$ 。

### 3.2.6 辅助函数的设计

本系统中部分功能进行大量的相同的操作, 对其进行重构, 定义下列函数。

#### 1) 访问二叉树节点 (Visit)

该函数接受一个 TreeNode 类型的指针 node, 该指针指向待访问的二叉树节点。函数依次打印该结点数据域的元素 ID 和 name。

该函数在遍历二叉树的时候以函数指针的方式作为参数传递给遍历函数, 用

于对二叉树的结点进行访问操作。

该函数仅包含顺序结构，因此函数的时间复杂度为  $O(1)$ 。

#### 2) 判断二叉树是否存在(TreeExist)

该函数接受一个 `TreeNode` 类型的指针 `t`，该指针指向带判断的二叉树的头结点。若 `t` 的元素 `exist` 为 0 则答应“二叉树不存在”。最后函数返回该元素。

该函数在调用各函数对二叉树进行操作前调用，若返回值为 0，则二叉树不存在，函数中输出错误提示，操作函数不调用，否则，调用相应的函数对二叉树进行操作。

该函数仅包含顺序和选择结构，因此函数的时间复杂度为  $O(1)$ 。

#### 3) 获得二叉树的含空结点数(TreeArrayLength)

该函数接受一个 `TreeNode` 类型的指针 `rootNode`，该指针指向待操作的二叉树的根结点。函数中使用全局变量 `TreeLength` 来记录二叉树的长度。函数使用递归实现，若 `rootNode` 为空，则将 `TreeLength` 自增 1 并返回 `TreeLength`。否则，将 `TreeLength` 自增 1，并分别计算 `rootNode` 左子树和右子树的长度，最后函数返回 `TreeLength`。

该函数用于在将二叉树存储到数组前计算数组的长度。

该函数会遍历二叉树中的每个结点，因此函数的时间复杂度为  $O(n)$ 。

#### 4) 将二叉树存储到数组(LoadToArray)

该函数接受两个参数，第一个参数为 `TreeNode` 类型的指针 `rootNode`，该指针指向待存储的二叉树的根结点，第二个参数为 `TreeNode` 类型的指针的指针 `result`。该函数使用递归实现，若 `rootNode` 为空，则创建一个 ID 为 -1 的结点，并将该结点的指针赋值给 `result` 的相应位置，否则，将 `rootNode` 赋值给 `result` 的相应位置，并分别对 `rootNode` 的左子树和右子树调用函数 `LoadToArray`。最后函数返回 `result`。

该函数会遍历二叉树的每一个结点，并存储每个结点的指针，因此函数的时间复杂度为  $O(n)$ 。



### 3.2.7 异常输入的处理

本系统中包含大量用户输入，而用户的输入内容具有不可预见性，因此每次读取用户输入的时候都应该对其进行相应的处理，若输入符合输入要求，则进行正常的操作，如用户输入不合法，则提示用户输入不合法并要求用户重新输入，若用户输入合法数据后多输入了一些其他数据，则应该清空缓冲区中的多余数据，以防下次获取输入时出现未知的错误。

对每一次用户输入都应该执行上述操作以保证程序得到正确的用户输入，从而保证程序能够正确稳定的运行。

## 3.3 系统实现

### 3.3.1 实现方案

本系统使用 C 语言实现，实现方案包括一个头文件 (biTree.h) 和两个源文件 (biTree.c 和 main.c)。

头文件 biTree.h 中定义了常量、数据类型，构建了二叉树的数据结构，声明了函数原型，并创建了全局变量。

源文件 biTree.c 中对头文件中声明的函数进行了逐个的定义。

源文件 main.c 中通过调用各函数对二叉树进行各种操作。

完成本实验的操作系统为 Windows 10 64 位，使用 visual studio 2017 编写代码、进行调试以及进行功能测试。

### 3.3.2 程序设计

完整程序见附录 3，部分代码的说明如下。

#### 1) 常量、数据类型以及数据结构的定义

在头文件 biTree.h 中，实现了 3.2.2 节中对常量、数据类型以及数据结构的定义。使用#define 定义了相关常量，使用 typedef 定义了需要的数据类型，并定义了二叉树结构体类型。具体的实现代码如下。

```
//定义常量
#define TRUE 1
#define FALSE 0
#define OK 1
```

```
#define ERROR 0
#define INFEASTABLE -1
#define OVERFLOW -2

//定义数据类型
typedef int status;      //函数执行状态
typedef struct ElemType {
    int studentID;
    char name[20];
} ElemType;      //结点数据域

typedef struct TreeNode {
    ElemType data;
    struct TreeNode * lchild;
    struct TreeNode * rchild;
} TreeNode;      //二叉树结点

typedef struct TreeHead {
    int exist; //0 表示 不存在, 1 表示 存在
    struct TreeNode * child;
} TreeHead;      //二叉树头结点

typedef struct statusNode {
    status s;
    TreeNode * node;
} statusNode; //需要返回结点指针的函数返回值类型
```

## 2) 创建二叉树函数的实现

创建二叉树函数将存储在顺序结构中的数据按照相应的逻辑转化为二叉树的逻辑结构,该过程需要将用户输入的所有数据全部存储后整体传入函数中,并在函数中实现二叉树的创建。该过程的具体实现代码如下。

该创建过程中存在一个可能出现的错误情况,即用户输入的序列不能够完全创建一个二叉树,此时函数应当对其进行相应的操作。考虑到对任意的序列,仅需要在其后添加足够多的空结点,即可正确地创建出二叉树,因此函数添加了一个 length 参数用来标记 definition 的长度,若函数将顺序表读取完以后,二叉树为创建完成,则说明出现该错误情况,此时,将待创建结点始终指向空结点,直到二叉树创建完成。

```
//创建二叉树
```

```

int CreateBiTree(TreeNode ** rootNode, ElemType * definition, int length)
{
    if (definition[CreateBiTreeLevel].studentID == -1)
    {
        if (CreateBiTreeLevel < length - 1)
            CreateBiTreeLevel++;
        else
            definition[CreateBiTreeLevel].studentID = -1;
        return OK;
    }
    *rootNode = (TreeNode *)malloc(sizeof(TreeNode));
    (*rootNode)->data = definition[CreateBiTreeLevel];
    (*rootNode)->lchild = NULL;
    (*rootNode)->rchild = NULL;
    if (CreateBiTreeLevel < length - 1)
        CreateBiTreeLevel++;
    else
        definition[CreateBiTreeLevel].studentID = -1;
    CreateBiTree(&((*rootNode)->lchild), definition, length);
    CreateBiTree(&((*rootNode)->rchild), definition, length);
    return OK;
}

```

### 3.3.3 系统测试

本测试中对二叉树的 21 个基本操作以及文件输入输出功能进行测试。测试时创建 3 个二叉树，并均选择第 1 个二叉树(编号为 1)作为待操作的二叉树。

#### 1) 初始化二叉树(InitBiTree)

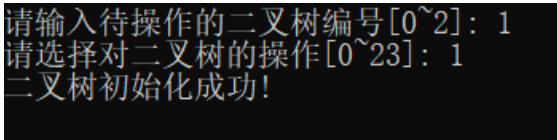
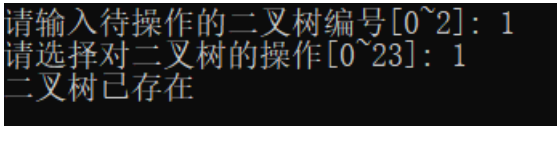
初始化二叉树的正常情况只有一种情况，即待操作的二叉树为未初始化的二叉树，此时函数将二叉树初始化并返回 OK，程序应输出“二叉树初始化成功”。

异常情况也有一种情况，即待操作的二叉树已初始化，此时函数不调用，程序输出“二叉树已存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 3-1。

表 3-1 初始化二叉树函数测试

	输入	理论输出	运行结果(截图)
--	----	------	----------

正常输入	1 1 选择第 1 个二叉树 (该二叉树未初始化), 选择功能 1	输出“二叉树初始化成功”	
异常输入	1 1 选择第 1 个二叉树 (该二叉树已初始化), 选择功能 1	输出“二叉树已存在”	

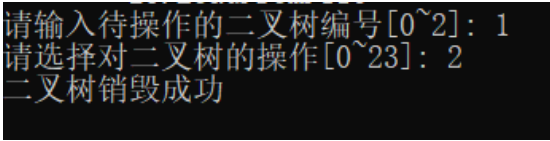
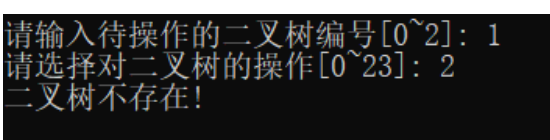
## 2) 销毁二叉树(DestroyBiTree)

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

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

具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-2。

表 3-2 销毁二叉树函数测试

	输入	理论输出	运行结果(截图)
正常输入	1 2 选择第 1 个二叉树 (该二叉树已初始化), 选择功能 2	输出“二叉树销毁成功”	
异常输入	1 2 选择第 1 个二叉树 (该二叉树未初始化), 选择功能 2	输出“二叉树不存在”	

## 3) 创建二叉树(CreateBiTree)

创建二叉树的正常情况只有一种情况,即待创建的二叉树已初始化,此时函数根据用户输入的数据创建二叉树,并返回 OK,程序应输出“二叉树创建成功”。

异常情况也有一种情况,即带创建的二叉树未初始化,此时函数不调用,程序输出“二叉树不存在”。

具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-3。

表 3-3 创建二叉树函数测试

	输入	理论输出	运行结果(截图)
--	----	------	----------

正常输入	1 3 15 1 EEE 2 CCC 3 AAA -1 4 BBB -1 -1 5 DDD -1 -1 6 GGG 7 FFF -1 -1 -1 选择第 1 个二叉树(该二叉树已初始化), 选择功能 3, 并根据提示输入二叉树的相应数据	输出“二叉树创建成功”	<pre> 请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 3 请输入待创建的二叉树的结点数量(包含空): 15 请输入第1个结点的ID(用-1表示空): 1 请输入第1个结点的name: EEE 请输入第2个结点的ID(用-1表示空): 2 请输入第2个结点的name: CCC 请输入第3个结点的ID(用-1表示空): 3 请输入第3个结点的name: AAA 请输入第4个结点的ID(用-1表示空): -1 请输入第5个结点的ID(用-1表示空): 4 请输入第5个结点的name: BBB 请输入第6个结点的ID(用-1表示空): -1 请输入第7个结点的ID(用-1表示空): -1 请输入第8个结点的ID(用-1表示空): 5 请输入第8个结点的name: DDD 请输入第9个结点的ID(用-1表示空): -1 请输入第10个结点的ID(用-1表示空): -1 请输入第11个结点的ID(用-1表示空): 6 请输入第11个结点的name: GGG 请输入第12个结点的ID(用-1表示空): 7 请输入第12个结点的name: FFF 请输入第13个结点的ID(用-1表示空): -1 请输入第14个结点的ID(用-1表示空): -1 请输入第15个结点的ID(用-1表示空): -1 二叉树创建成功! </pre>
异常输入	1 3 选择第 1 个二叉树(该二叉树未初始化), 选择功能 3	输出“二叉树不存在”	<pre> 请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 3 二叉树不存在! </pre>

#### 4) 清空二叉树(ClearBiTree)

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

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

具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-4。

表 3-4 清空二叉树函数测试

	输入	理论输出	运行结果(截图)
正常输入	1 4 选择第 1 个二叉树(已初始化), 选择功能 4	输出“二叉树清空成功”	<pre> 请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 4 二叉树清空成功! </pre>
异常输入	1 4 选择第 1 个二叉树(未初始化), 选择功能 4	输出“二叉树不存在”	<pre> 请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 4 二叉树不存在! </pre>

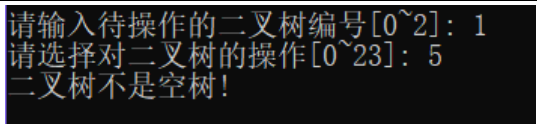
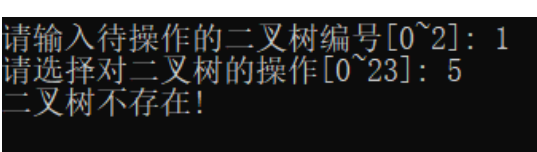
### 5) 判断空二叉树 (BiTreeEmpty)

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

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

具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-5。

表 3-5 判断空二叉树功能测试

	输入	理论输出	运行结果(截图)
正常 输入 1	1 5 选择第 1 个二叉树 (已初始化, 为空), 选择功能 5	输出“二 叉树为空 树”	
正常 输入 2	1 5 选择第 1 个二叉树 (已初始化, 不为 空), 选择功能 5	输出“二 叉树不是 空树”	
异常 输入	1 5 选择第 1 个二叉树 (未初始化), 选择功 能 5	输出“二 叉树不存 在”	

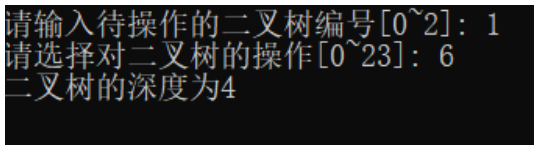
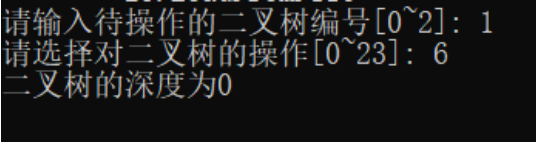
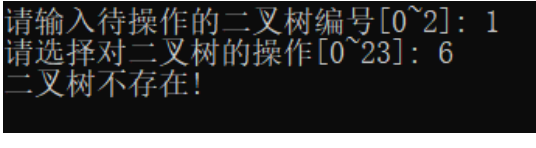
### 6) 求二叉树的深度 (BiTreeDepth)

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

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

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-6。

表 3-6 求二叉树深度功能测试

	输入	理论输出	运行结果(截图)
正常 输入 1	1 6 选择第 1 个二叉树 (已初始化, 非 空), 选择功能 6	输出“二叉 树的深度为 4”	
正常 输入 2	1 6 选择第 1 个二叉树 (已初始化, 为 空), 选择功能 6	输出“二叉 树的深度为 0”	
异常 输入	1 6 选择第 1 个二叉树 (未初始化), 选择 功能 6	输出“二叉 树不存在”	

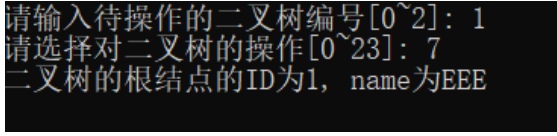
#### 7) 获得根结点(Root)

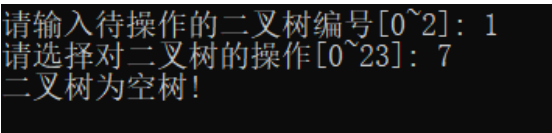
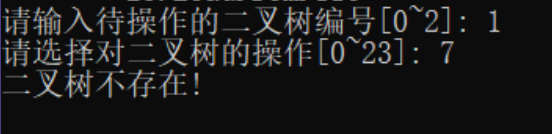
该函数的正常情况有两种情况, 第一种情况为二叉树已初始化且含有元素, 此时函数返回根结点指针, 程序输出二叉树的根结点 ID 和 name; 第二种情况为二叉树已初始化但为空, 此时函数返回 NULL, 程序输出“二叉树为空树”。

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

本测试中, 当二叉树存在且不为空时, 二叉树的含空的前序遍历结果为 (1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-7。

表 3-7 获取根结点功能测试

	输入	理论输出	运行结果(截图)
正 常 输 入 1	1 7 选择第 1 个二 叉树(已初始 化, 非空), 选 择功能 7	输出“二叉树 的根结点的 ID 为 1, name 为 EEE”	

正常输入 2	1 7 选择第 1 个二叉树(已初始化, 为空), 选择功能 7	输出“二叉树为空树”	
异常输入	1 7 选择第 1 个二叉树(未初始化), 选择功能 7	输出“二叉树不存在”	

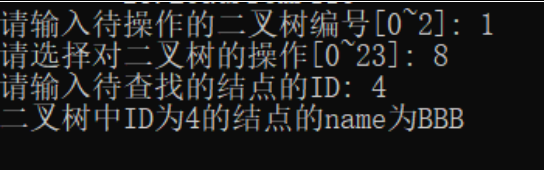
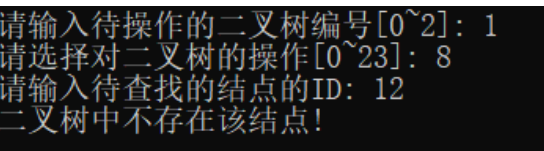
### 8) 获得结点(Value)

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

异常情况有两种情况, 第一种为二叉树已初始化, 但是待获得的结点不在二叉树中, 此时函数返回 NULL, 程序输出“二叉树中不存在该结点”; 第二种情况为二叉树未初始化, 此时函数不调用, 程序输出“二叉树不存在”。

本测试中, 当二叉树存在且不为空时, 二叉树的含空的前序遍历结果为 (1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-8。

表 3-8 获得结点功能测试

	输入	理论输出	运行结果(截图)
正常输入	1 8 4 选择第 1 个二叉树(已初始化), 选择功能 8, 获得 ID 为 4 的结点	输出“二叉树中 ID 为 4 的结点的 name 为 BBB”	
异常输入 1	1 8 4 选择第 1 个二叉树(已初始化), 选择功能 8, 获得 ID 为 12 的结点	输出“二叉树中不存在该结点”	



异常输入 2	1 8 选择第 1 个二叉树(未初始化), 选择功能 8	输出“二叉树不存在”	
--------	------------------------------------	------------	--

### 9) 结点赋值 (Assign)

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

异常情况有两种情况,第一种为二叉树已初始化但是待赋值的结点不在二叉树中,此时函数返回 ERROR,程序输出“二叉树中不存在该结点”;第二种情况为二叉树未初始化,此时函数不调用,程序输出“二叉树不存在”。

本测试中,当二叉树存在且不为空时,二叉树的含空的前序遍历结果为 (1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-9。

表 3-9 结点赋值功能测试

	输入	理论输出	运行结果(截图)
正常情况	1 9 1 EEEEEEE 选择第 1 个二叉树(已初始化),选择功能 9,将结点 ID 为 1 的结点的 name 赋值为 EEEEEEE	输出“给该结点赋值成功”	
异常情况 1	1 9 12 121212 选择第 1 个二叉树(已初始化),选择功能 9,将结点 ID 为 1 的结点的 name 赋值为 121212	输出“二叉树中不存在该结点”	
异常情况 2	1 9 选择第 1 个二叉树(未初始化),选择功能 9	输出“二叉树不存在”	

### 10) 获得双亲结点 (Parent)

该函数的正常情况有一种情况,即二叉树已初始化且待查找双亲结点的结点

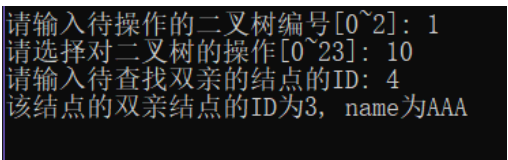
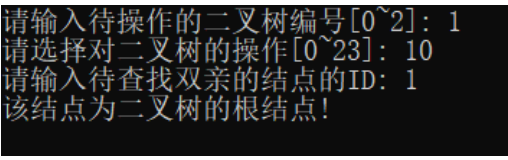
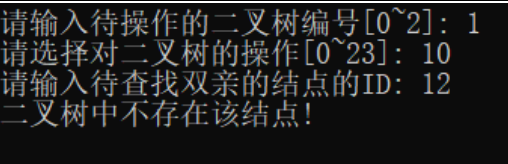
在二叉树中，且该结点的双亲结点存在，此时函数返回 OK 和双亲节点的指针。

程序输出双亲结点的 ID 和 name。

异常情况有三种情况，第一种情况为二叉树已初始化，但是待查找双亲结点的双亲结点不存在，即该结点为二叉树的根结点，此时函数返回 (-1, NULL)，程序输出“该结点为二叉树的根结点”；第二种情况为二叉树已初始化，但是待查找双亲结点的结点不在二叉树中，此时函数返回 (ERROR, NULL)。程序输出“二叉树中不存在该结点”；第三种情况为二叉树未初始化，此时函数不调用，程序输出“二叉树不存在”。

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-10。

表 3-10 获得双亲结点功能测试

	输入	理论输出	运行结果(截图)
正常输入	1 10 4 选择第 1 个二叉树(已初始化)，选择功能 10，查找结点 ID 为 4 的结点的双亲结点	输出“该结点的双亲结点的 ID 为 3，name 为 AAA”	
异常输入 1	1 10 1 选择第 1 个二叉树(已初始化)，选择功能 10，查找结点 ID 为 1 的结点的双亲结点	输出“该结点为二叉树根结点”	
异常输入 2	1 10 12 选择第 1 个二叉树(已初始化)，选择功能 10，查找结点 ID 为 12 的结点的双亲结点	输出“二叉树中不存在该结点”	

异常输入 3	1 10 选择第 1 个二叉树 (未初始化), 选择 功能 10	输出“二叉树 不存在”	
-----------	---	----------------	--

#### 11) 获得左孩子结点(LeftChild)

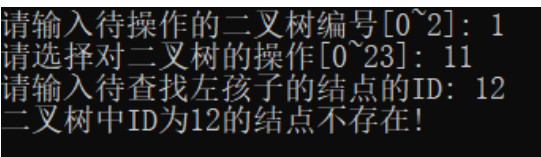
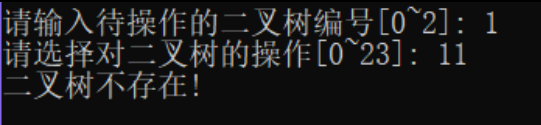
该函数的正常情况只有一种情况,即二叉树已初始化且待查找左孩子结点的结点在二叉树中且该结点的左孩子存在,此时函数返回 OK 和左孩子结点指针。程序输出该结点的左孩子的 ID 和 name。

异常情况有三种情况,第一种情况为二叉树已初始化且待查找左孩子结点的结点在二叉树中,但是该结点的左孩子不存在,此时函数返回 (OK, NULL), 程序输出“该结点没有左孩子”;第二种情况为二叉树已初始化但是待查找左孩子结点的结点不在二叉树中,此时函数返回 (ERROR, NULL), 程序输出二叉树中不存在该结点;第三种情况为二叉树未初始化,此时函数不调用,程序输出“二叉树不存在”。

本测试中,当二叉树存在且不为空时,二叉树的含空的前序遍历结果为 (1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-11。

表 3-11 获得左孩子结点功能测试

	输入	理论输出	运行结果(截图)
正常输入	1 11 1 选择第 1 个二叉树 (已初始化), 选择 功能 11, 获得结点 ID 为 1 的结点的左 孩子	输出“该结 点的左孩子 的 ID 为 2, name 为 CCC”	
异常输入 1	1 11 5 选择第 1 个二叉树 (已初始化), 选择 功能 11, 获得结点 ID 为 5 的结点的左 孩子	输出“该结 点没有左孩 子”	

异常输入 2	1 11 12 选择第 1 个二叉树 (已初始化), 选择功能 11, 获得结点 ID 为 12 的结点的左孩子	输出“二叉树中 ID 为 12 的结点不存在”	
异常输入 3	1 11 选择第 1 个二叉树 (未初始化), 选择功能 11	输出“二叉树不存在”	

## 12) 获得右孩子结点(RightChild)

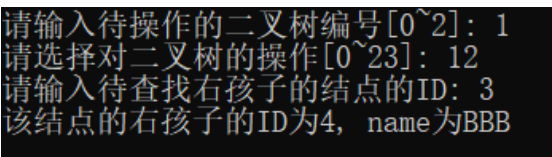
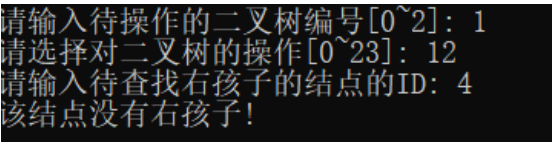
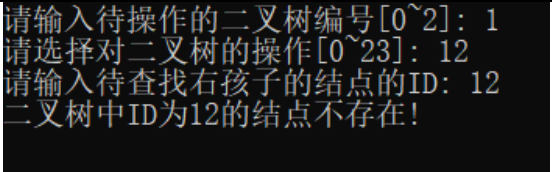
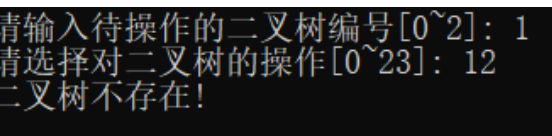
该函数的正常情况只有一种情况,即二叉树已初始化且待查找右孩子结点的结点在二叉树中且该结点的右孩子存在,此时函数返回 OK 和右孩子结点指针。程序输出该结点的右孩子的 ID 和 name。

异常情况有三种情况,第一种情况为二叉树已初始化且待查找右孩子结点的结点在二叉树中,但是该结点的右孩子不存在,此时函数返回(OK, NULL),程序输出“该结点没有右孩子”;第二种情况为二叉树已初始化但是待查找右孩子结点的结点不在二叉树中,此时函数返回(ERROR, NULL),程序输出二叉树中不存在该结点;第三种情况为二叉树未初始化,此时函数不调用,程序输出“二叉树不存在”。

本测试中,当二叉树存在且不为空时,二叉树的含空的前序遍历结果为(1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-12。

表 3-12 获得右孩子节点功能测试

	输入	理论输出	运行结果(截图)
--	----	------	----------

正常输入	1 12 3 选择第 1 个二叉树(已初始化), 选择第 12 个功能, 查找 ID 为 3 的结点的右孩子结点	输出“该结点的右孩子的 ID 为 4, name 为 BBB”	 <pre> 请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 12 请输入待查找右孩子的结点的ID: 3 该结点的右孩子的ID为4, name为BBB                     </pre>
异常输入 1	1 12 4 选择第 1 个二叉树(已初始化), 选择第 12 个功能, 查找 ID 为 4 的结点的右孩子结点	输出“该结点没有右孩子”	 <pre> 请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 12 请输入待查找右孩子的结点的ID: 4 该结点没有右孩子!                     </pre>
异常输入 2	1 12 12 选择第 1 个二叉树(已初始化), 选择第 12 个功能, 查找 ID 为 12 的结点的右孩子结点	输出“二叉树中 ID 为 12 的结点不存在”	 <pre> 请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 12 请输入待查找右孩子的结点的ID: 12 二叉树中ID为12的结点不存在!                     </pre>
异常输入 3	1 12 选择第 1 个二叉树(未初始化), 选择第 12 个功能	输出“二叉树不存在”	 <pre> 请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 12 二叉树不存在!                     </pre>

### 13) 获得左兄弟结点(LeftSibling)

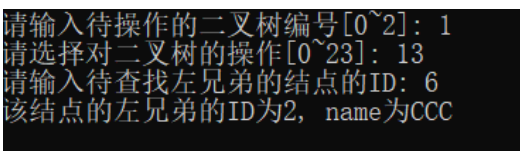
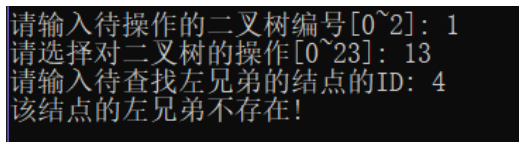
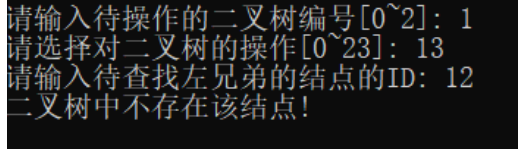
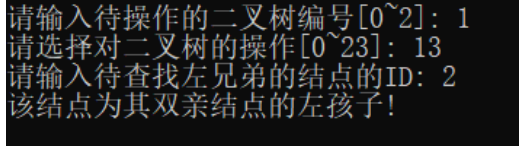
该函数的正常情况有一种情况, 即二叉树已初始化且待查找左兄弟的结点在二叉树中且该结点的左兄弟结点存在, 此时函数返回 OK 和左兄弟结点指针, 程序输出左兄弟结点的 ID 和 name。

异常情况有 5 种情况, 第一种情况为二叉树已初始化但是待查找左兄弟的结点不在二叉树中, 此时函数返回 (ERROR, NULL), 程序输出“二叉树中不存在该结点”; 第二种情况为二叉树已初始化且待查找左兄弟的结点在二叉树中但是该结点为左孩子, 此时函数返回 (-2, NULL), 程序输出“该结点为其双亲结点的左孩子”; 第三种情况为二叉树已初始化但是待查找左兄弟的结点为二叉树的根结点,

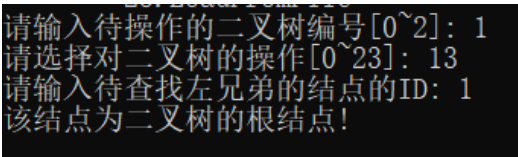
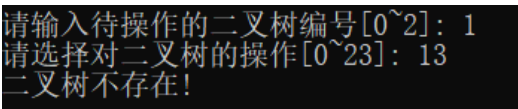
此时函数返回(-1, NULL)，程序输出“该结点为二叉树的根结点”；第四种情况为二叉树已初始化但是待查找左兄弟的结点的左兄弟不存在，此时函数返回(-3, NULL)，程序输出“该结点的左兄弟不存在”；第五种情况为二叉树未初始化，测试函数不调用，程序输出“二叉树不存在”。

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为(1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表3-13。

表 3-13 获得左孩子结点功能测试

	输入	理论输出	运行结果(截图)
正常输入	1 13 6 选择第 1 个二叉树(已初始化)，选择功能 13，查找 ID 为 6 的结点的左兄弟结点	输出“该结点的左兄弟的 ID 为 2，name 为 CCC”	
异常输入 1	1 13 4 选择第 1 个二叉树(已初始化)，选择功能 13，查找 ID 为 4 的结点的左兄弟结点	输出“该结点的左兄弟不存在”	
异常输入 2	1 13 12 选择第 1 个二叉树(已初始化)，选择功能 13，查找 ID 为 12 的结点的左兄弟结点	输出“二叉树中不存在该结点”	
异常输入 3	1 13 2 选择第 1 个二叉树(已初始化)，选择功能 13，查找 ID 为 2 的结点的左兄弟结点	输出“该结点为其双亲结点的左孩子”	



异常输入 4	1 13 1 选择第 1 个二叉树 (已初始化), 选择功能 13, 查找 ID 为 1 的结点的左兄弟结点	输出“该结点为二叉树的根结点”	
异常输入 5	1 13 选择第 1 个二叉树 (未初始化), 选择功能 13	输出“二叉树不存在”	

#### 14) 获得右兄弟结点(RightSibling)

该函数的正常情况有一种情况,即二叉树已初始化且待查找右兄弟的结点在二叉树中且该结点的右兄弟结点存在,此时函数返回 OK 和右兄弟结点指针,程序输出右兄弟结点的 ID 和 name。

异常情况有 5 种情况,第一种情况为二叉树已初始化但是待查找右兄弟的结点不在二叉树中,此时函数返回 (ERROR, NULL), 程序输出“二叉树中不存在该结点”;第二种情况为二叉树已初始化且待查找右兄弟的结点在二叉树中但是该结点为右孩子,此时函数返回 (-2, NULL), 程序输出“该结点为其双亲结点的右孩子”;第三种情况为二叉树已初始化但是待查找右兄弟的结点为二叉树的根结点,此时函数返回 (-1, NULL), 程序输出“该结点为二叉树的根结点”;第四种情况为二叉树已初始化但是待查找右兄弟的结点的右兄弟不存在,此时函数返回 (-3, NULL), 程序输出“该结点的右兄弟不存在”;第五种情况为二叉树未初始化,测试函数不调用,程序输出“二叉树不存在”。

本测试中,当二叉树存在且不为空时,二叉树的含空的前序遍历结果为 (1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-14。

表 3-14 获得右兄弟结点功能测试

	输入	理论输出	运行结果(截图)
--	----	------	----------

正常输入	1 14 3 选择第 1 个二叉树 (已初始化), 选择功能 14, 查找 ID 为 3 的结点的右兄弟	输出 “该结点的右兄弟的 ID 为 5, name 为 DDD”	请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 14 请输入待查找左兄弟的结点的ID: 3 该结点的右兄弟的ID为5, name为DDD
异常输入 1	1 14 1 选择第 1 个二叉树 (已初始化), 选择功能 14, 查找 ID 为 1 的结点的右兄弟	输出 “该结点为二叉树的根结点”	请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 14 请输入待查找左兄弟的结点的ID: 1 该结点为二叉树的根结点!
异常输入 2	1 14 5 选择第 1 个二叉树 (已初始化), 选择功能 14, 查找 ID 为 5 的结点的右兄弟	输出 “该结点为其双亲结点的右孩子”	请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 14 请输入待查找左兄弟的结点的ID: 5 该结点为其双亲结点的右孩子!
异常输入 3	1 14 7 选择第 1 个二叉树 (已初始化), 选择功能 14, 查找 ID 为 7 的结点的右兄弟	输出 “该结点的右兄弟不存在”	请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 14 请输入待查找左兄弟的结点的ID: 7 该结点的右兄弟不存在!
异常输入 4	1 14 12 选择第 1 个二叉树 (已初始化), 选择功能 14, 查找 ID 为 12 的结点的右兄弟	输出 “二叉树中不存在该结点”	请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 14 请输入待查找左兄弟的结点的ID: 12 二叉树中不存在该结点!
异常输入 5	1 14 选择第 1 个二叉树 (未初始化), 选择功能 14	输出 “二叉树不存在”	请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 14 二叉树不存在!

#### 15) 插入子树 (InsertChild)

该函数的正常情况只有一种情况, 即二叉树已初始化且待插入子树的结点在二叉树中且待插入的子树的右子树为空, 此时, 函数返回 OK, 程序输出 “插入子

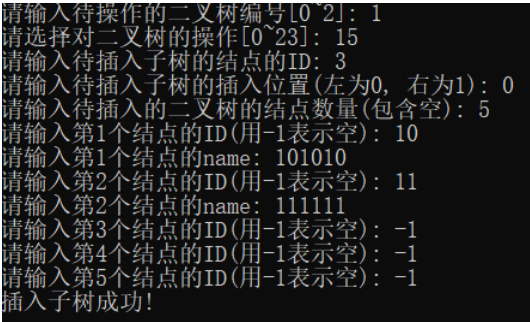
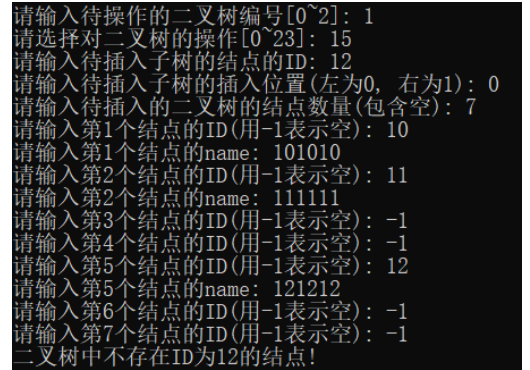


树成功”。

异常情况有三种情况，第一种为二叉树已初始化但是待插入子树的结点不在二叉树中，此时函数返回-1，程序输出二叉树中不存在该结点；第二种情况为待插入子树的右子树不为空，此时函数返回 ERROR，程序输出“待插入子树的右子树存在”；第三种情况为二叉树未初始化，此时函数不调用，程序输出“二叉树不存在”。

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-15。

表 3-15 插入子树功能测试

	输入	理论输出	运行结果(截图)
正常输入	1 15 3 0 5 10 101010 11 111111 -1 -1 -1 选择第 1 二叉树(已初始化)，选择功能 15，按提示建立子树并将其插入到 ID 为 3 的结点的左子树上	输出“插入子树成功”	
异常输入 1	1 15 12 0 7 10 101010 11 111111 -1 -1 12 121212 -1 -1 选择第 1 个二叉树(已初始化)，选择功能 15，按提示建立子树并将其插入到 ID 为 12 的结点的左子树上	输出“二叉树中不存在 ID 为 12 的结点”	

异常输入 2	1 15 3 0 7 10 101010 11 111111 -1 -1 12 121212 -1 -1 选择第 1 个二叉树(已初始化), 选择功能 15, 按提示建立子树并将其插入到 ID 为 3 的结点的左子树上	输出“待插入子树的右子树存在”	<pre> 请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 15 请输入待插入子树的结点的ID: 3 请输入待插入子树的插入位置(左为0, 右为1): 0 请输入待插入的二叉树的结点数量(包含空): 7 请输入第1个结点的ID(用-1表示空): 10 请输入第1个结点的name: 101010 请输入第2个结点的ID(用-1表示空): 11 请输入第2个结点的name: 111111 请输入第3个结点的ID(用-1表示空): -1 请输入第4个结点的ID(用-1表示空): -1 请输入第5个结点的ID(用-1表示空): 12 请输入第5个结点的name: 121212 请输入第6个结点的ID(用-1表示空): -1 请输入第7个结点的ID(用-1表示空): -1 待插入子树的右子树存在! </pre>
异常输入 3	1 15 选择第 1 个二叉树(未初始化), 选择功能 15	输出“二叉树不存在”	<pre> 请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 15 二叉树不存在! </pre>

#### 16) 删除子树(DeleteChild)

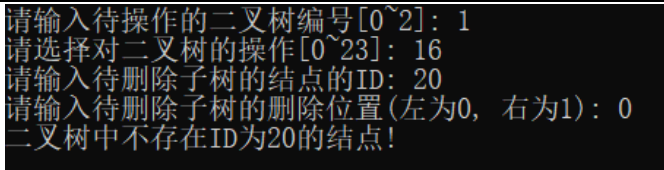
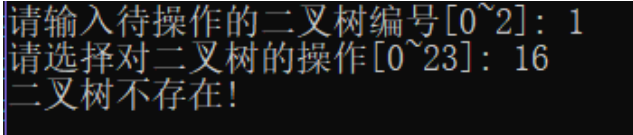
该函数的正常输入有一种, 即二叉树已初始化且待删除子树的结点在二叉树中, 此时函数返回 OK, 程序输出“删除子树成功”。

异常输入有两种情况, 第一种为二叉树已初始化但是待删除子树的的结点不在二叉树中, 此时函数返回 ERROR, 程序输出二叉树中不存在该结点; 第二种情况为二叉树未初始化, 此时函数不调用, 程序输出“二叉树不存在”。

本测试中, 当二叉树存在且不为空时, 二叉树的含空的前序遍历结果为 (1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-16。

表 3-16 删除子树功能测试

	输入	理论输出	运行结果(截图)
正常输入	1 16 5 0 选择第 1 个二叉树(已初始化), 选择功能 16, 删除 ID 为 5 的结点的左子树	输出“删除子树成功”	<pre> 请输入待操作的二叉树编号[0~2]: 1 请选择对二叉树的操作[0~23]: 16 请输入待删除子树的结点的ID: 5 请输入待删除子树的删除位置(左为0, 右为1): 0 删除子树成功! </pre>

异常输入 1	1 16 20 0 选择第 1 个二叉树(已初始化), 选择功能 16, 删除 ID 为 20 的结点的左子树	输出“二叉树中不存在 ID 为 20 的结点”	
异常输入 2	1 16 选择第 1 个二叉树(未初始化), 选择功能 16	输出“二叉树不存在”	

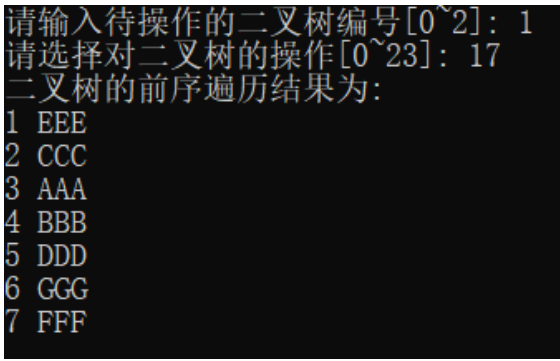
### 17) 前序遍历(PreOrderTraverse)

该函数的正常情况有一种, 即二叉树已初始化, 此时函数返回 OK, 程序输出二叉树的前序遍历序列。本测试中, 使用两种输入进行测试, 分别测试二叉树是否为空时的输出结果。

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

本测试中, 当二叉树存在且不为空时, 二叉树的含空的前序遍历结果为 (1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-17。

表 3-17 前序遍历功能测试

	输入	理论输出	运行结果(截图)
正常输入 1	1 17 选择第 1 个二叉树(已初始化, 非空), 选择功能 17	输出二叉树的前序遍历序列	

正常 输入 2	1 17 选择第 1 个二叉树 (已初始化, 为 空), 选择功能 17	输出“二 叉树为 空”	
异常 输入	1 17 选择第 1 个二叉树 (未初始化), 选择 功能 17	输出“二 叉树不存 在”	

### 18) 中序遍历(InOrderTraverse)

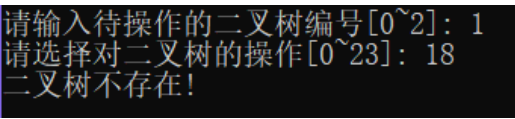
该函数的正常情况有一种, 即二叉树已初始化, 此时函数返回 OK, 程序输出二叉树的中序遍历序列。本测试中, 使用两种输入进行测试, 分别测试二叉树是否为空时的输出结果。

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

本测试中, 当二叉树存在且不为空时, 二叉树的含空的前序遍历结果为 (1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-18。

表 3-18 中序遍历功能测试

	输入	理论输出	运行结果(截图)
正常 输入 1	1 18 选择第 1 个二叉树 (已初始化, 不为 空), 选择功能 18	输出二叉树 的中序遍历 序列	
正常 输入 2	1 18 选择第 1 个二叉树 (已初始化, 为空), 选择功能 18	输出“二 叉树为 空”	

异常输入	1 18 选择第 1 个二叉树 (未初始化), 选择功能 18	输出“二叉树不存在”	
------	---------------------------------------	------------	--

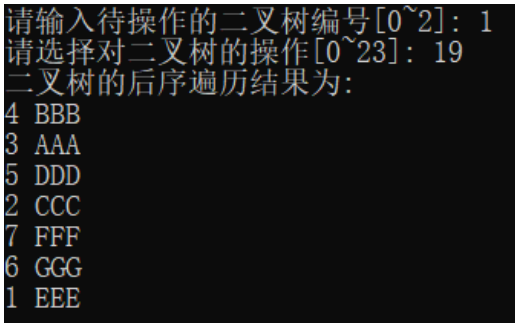
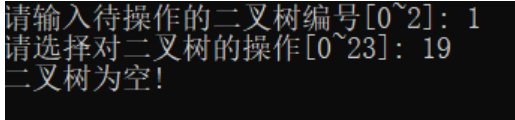
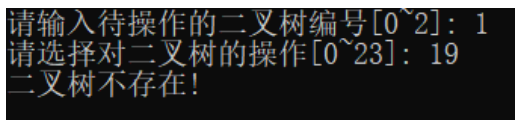
### 19) 后序遍历(PostOrderTraverse)

该函数的正常情况有一种, 即二叉树已初始化, 此时函数返回 OK, 程序输出二叉树的后序遍历序列。本测试中, 使用两种输入进行测试, 分别测试二叉树是否为空时的输出结果。

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

本测试中, 当二叉树存在且不为空时, 二叉树的含空的前序遍历结果为 (1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-19。

表 3-19 后序遍历功能测试

	输入	理论输出	运行结果(截图)
正常输入 1	1 19 选择第 1 个二叉树 (已初始化, 非空), 选择功能 19	输出二叉树的后序遍历序列	
正常输入 2	1 19 选择第 1 个二叉树 (已初始化, 为空), 选择功能 19	输出“二叉树为空”	
异常输入	1 19 选择第 1 个二叉树 (未初始化), 选择功能 19	输出“二叉树不存在”	

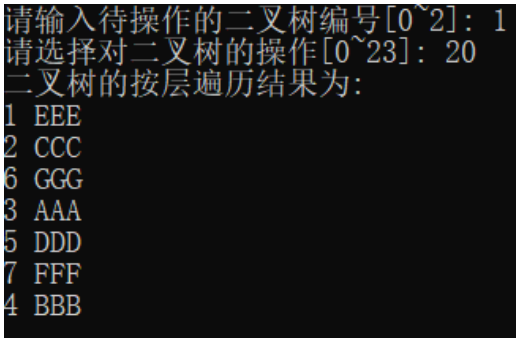
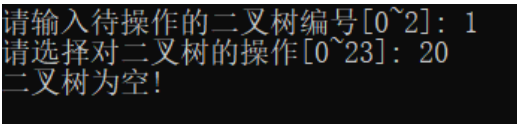
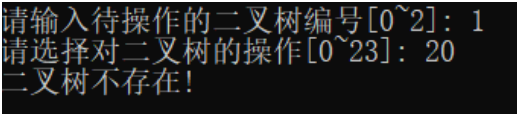
### 20) 按层遍历(LevelOrderTraverse)

该函数的正常情况有一种，即二叉树已初始化，此时函数返回 OK，程序输出二叉树的按层遍历序列。本测试中，使用两种输入进行测试，分别测试二叉树是否为空时的输出结果。

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

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-20。

表 3-20 按层遍历功能测试

	输入	理论输出	运行结果(截图)
正常 输入 1	1 20 选择第 1 个二叉树 (已初始化, 非空), 选择功能 20	输出二叉树的 按层遍历 序列	
正常 输入 2	1 20 选择第 1 个二叉树 (已初始化, 为空), 选择功能 20	输出“二叉 树为空”	
异常 输入	1 20 选择第 1 个二叉树 (未初始化), 选择功 能 20	输出“二叉 树不存在”	

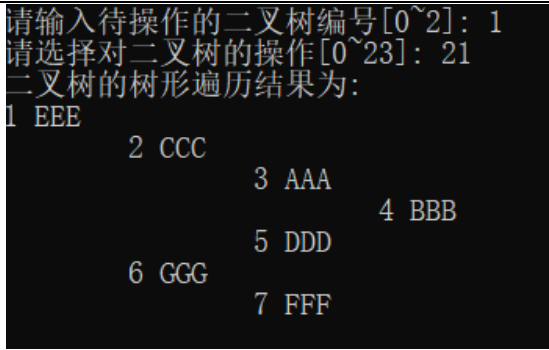
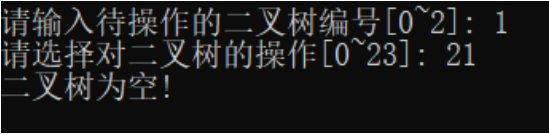
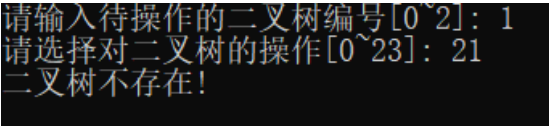
#### 21) 按树形遍历(TreeOrderTraverse)

该函数的正常情况有一种，即二叉树已初始化，此时函数返回 OK，程序输出二叉树的按树形遍历序列。本测试中，使用两种输入进行测试，分别测试二叉树是否为空时的输出结果。

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

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-21。

表 3-21 按树形遍历功能测试

	输入	理论输出	运行结果(截图)
正常 输入 1	1 21 选择第 1 个二叉树 (已初始化, 非 空), 选择功能 21	输出二叉树 的树形遍历 序列	
正常 输入 2	1 21 选择第 1 个二叉树 (已初始化, 为 空), 选择功能 21	输出“二义 树为空”	
异常 输入	1 21 选择第 1 个二叉树 (未初始化), 选择 功能 21	输出“二义 树不存在”	

## 22) 存储到文件(LoadToFile)

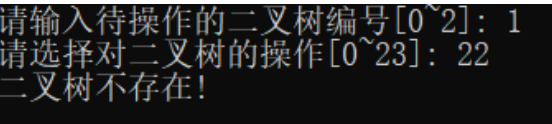
该函数的正常情况有一种情况，即二叉树存在且文件打开成功，此时函数返回 0，程序输出“输出到文件成功”。

异常情况有两种情况，第一种为文件打开失败，此时函数返回 1，程序输出“文件打开失败”；第二种情况为二叉树不存在，此时函数不调用，程序输出“二叉树不存在”。

本测试中，当二叉树存在且不为空时，二叉树的含空的前序遍历结果为 (1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-22。



表 3-22 存储到文件功能测试

	输入	理论输出	运行结果(截图)
正常输入	1 22 tree 选择第 1 个二叉树(已初始化), 选择功能 22, 将二叉树存储到文件 tree 中	输出“输出到文件成功”	
异常输入 1	1 22 tree 选择第 1 个二叉树, 选择功能 22, 将二叉树存储到文件 tree 中	输出“文件打开失败”	
异常输入 2	1 22 tree 选择第 1 个二叉树(未初始化), 选择功能 22	输出“二叉树不存在”	

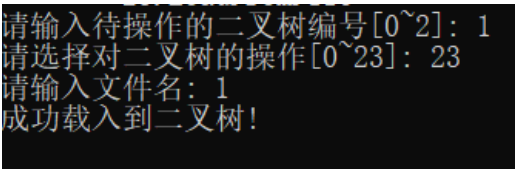
### 23) 从文件加载二叉树(LoadFromFile)

该函数正常情况只有一种情况, 即二叉树已初始化且为空且文件成功打开, 此时函数返回 0, 程序输出“成功载入到二叉树”。

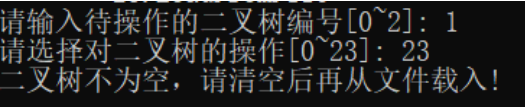
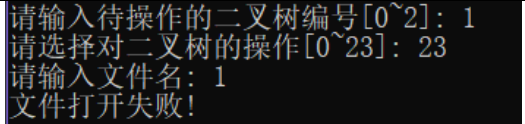
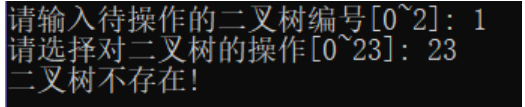
异常情况有三种情况, 第一种情况为二叉树已初始化但不为空, 此时函数不调用, 程序输出“二叉树不为空, 请清空后再从文件载入”; 第二种情况为二叉树已初始化, 但是文件打开失败, 此时函数返回 1, 程序输出“文件打开失败”; 第三种情况为二叉树未初始化, 此时函数不调用, 程序输出“二叉树不存在”。

本测试中, 当二叉树存在且不为空时, 二叉树的含空的前序遍历结果为 (1, EEE), (2, CCC), (3, AAA), -1, (4, BBB), -1, -1, (5, DDD), -1, -1, (6, GGG), (7, FFF), -1, -1, -1。具体的测试样例、理论输出结果以及程序的实际输出结果见表 3-23。

表 3-23 从文件加载二叉树功能测试

	输入	理论输出	运行结果(截图)
正常输入	1 23 1 选择第 1 个二叉树(已初始化, 为空), 选择功能 23, 从文件 1 加载二叉树	输出“成功载入到二叉树”	



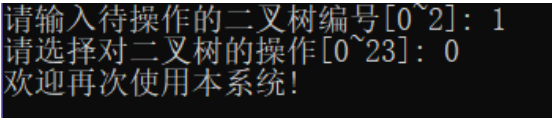
异常输入 1	1 23 1 选择第 1 个二叉树 (已初始化, 非空), 选择功能 23, 从文件 1 加载二叉树	输出“二叉树不为空, 请清空后再从文件载入”	
异常输入 2	1 23 1 选择第 1 个二叉树 (已初始化), 选择功能 23, 从文件 1 加载二叉树	输出“文件打开失败”	
异常输入 3	1 23 1 选择第 1 个二叉树 (未初始化), 选择功能 23	输出“二叉树不存在”	

#### 24) 退出程序

退出程序的功能与二叉树的状态无关, 均为正常情况。

具体的测试样例、理论输出结果以及程序实际输出结果见表 3-24。

表 3-24 退出程序功能测试

	输入	理论输出	运行结果(截图)
正常输入	1 0 选择第 1 个二叉树, 选择功能 0	输出“欢迎再次使用本系统”	

### 3.3.4 其他问题的说明

在测试环境(Windows 10 64 位)中, 若可执行文件在 C 盘根目录下, 则无法使用 LoadToFile 函数将二叉树写入到不存在的文件中, 原因为: 程序没有权限在 C 盘根目录下创建文件。若可执行文件位于 C 盘根目录下, 则因使用管理员权限运行该程序, 才能够正常使用文件 I/O 功能。

## 3.4 实验小结

本次实验内容较多, 相比于前两次实验, 实验内容也更加有难度, 通过本次

实验，我对二叉树这种特殊的树形结构有了更加深刻的认识，对树以及二叉树的应用也有了一定的了解。

本次实验中，我对 C 语言中内存的管理有了更加深刻的了解，我在实验中使用了较多的动态内存分配，而对这些内存的使用以及使用完后的释放问题在程序中是一个较为重要的问题。通过解决这些问题，我更为直观的看到了程序中出现内存泄漏的问题以及释放内存后指针的指向问题。这些让我对程序中的内存管理有了更多的了解。

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

## 4 基于邻接表的有向网实现

### 4.1 问题描述

图是有顶点集  $V$  和边集  $E$  组成的数据结构，每条边是一个点对  $(v, w)$ ，其中  $v, w$  属于  $V$ ，如果点对是有序的，则图称为有向图，否则称为无向图。网是边带有权值的图。邻接表是对于每一个顶点，都有一个表保存所有邻接的顶点，而将边的权值保存在表示邻接顶点的元素中的一种图的表示方法。本实验使用邻接表作为存储结构实现有向网的逻辑结构，本实验实现了有向网的 13 个基本操作。这 13 个基本操作的具体内容如下。

(1) 创建图：函数名称是 `CreateGraph(graph, setOfVertex, numOfVertex, infoVertexAndArc, numOfArc)`；初始条件是图 `graph` 存在，`setOfVertex` 是图的顶点集，`infoVertexAndArc` 是图的关系集，`numOfVertex` 是图中顶点的数量，`numOfArc` 是图中弧的数量；操作结果是按照上述四个参数的定义构造图 `graph`。

(2) 销毁图：函数名称是 `DestroyGraph(graph)`；初始条件图 `graph` 已存在；操作结果是销毁图 `graph`。

(3) 查找顶点：函数名称是 `LocateVex(graph, vertexID)`；初始条件是图 `graph` 存在，ID 为 `vertexID` 的顶点和 `graph` 中的顶点具有相同特征；操作结果是，若 ID 为 `vertexID` 的顶点在图 `G` 中存在，返回该顶点的位置信息，否则返回其它信息。

(4) 获得顶点值：函数名称是 `GetVex(graph, vertexID)`；初始条件是图 `graph` 存在，`G` 中的某个顶点的 ID 为 `vertexID`；操作结果是返回该顶点的值。

(5) 顶点赋值：函数名称是 `PutVex(graph, vertexID, vertexNewValue)`；初始条件是图 `graph` 存在，`G` 中的某个顶点的 ID 为 `vertexID`；操作结果是给该顶点赋值 `vertexNewValue`。

(6) 获得第一邻接点：函数名称是 `FirstAdjVex(graph, vertexID)`；初始条件是图 `graph` 存在，`G` 的一个顶点的 ID 为 `vertexID`；操作结果是返回该顶点的第一个邻接顶点，如果没有邻接顶点，返回“空”。

(7) 获得下一邻接点：函数名称是 `NextAdjVex(graph, vertexID, vertexFirstAdjID)`；初始条件是图 `graph` 存在, `G` 的一个顶点的 ID 为 `vertexID`, 该顶点的一个邻接顶点的 ID 为 `vertexFirstAdjID`；操作结果是返回该顶点(相对于该邻接顶点)的下一个邻接顶点，如果该邻接顶点是最后一个邻接顶点，返回“空”。

(8) 插入顶点：函数名称是 `InsertVex(graph, newVertex)`；初始条件是图 `graph` 存在，信息为 `newVertex` 的顶点和 `graph` 中的顶点具有相同特征；操作结果是在图 `graph` 中增加该顶点。

(9) 删除顶点：函数名称是 `DeleteVex(graph, vertexID)`；初始条件是图 `graph` 存在, `G` 的一个顶点的 ID 为 `vertexID`；操作结果是在图 `G` 中删除该顶点以及和该顶点相关的弧。

(10) 插入弧：函数名称是 `InsertArc(graph, newArc)`；初始条件是图 `graph` 存在, `newArc` 是一条弧；操作结果是在图 `graph` 中增加弧 `newArc`。

(11) 删除弧：函数名称是 `DeleteArc(graph, startID, endID)`；初始条件是图 `graph` 存在, `G` 的两个顶点的 ID 分别为 `startID` 和 `endID`；操作结果是在图 `G` 中删除由 ID 为 `startID` 的顶点指向 ID 为 `endID` 的顶点的弧。

(12) 深度优先搜索遍历：函数名称是 `DFS Traverse(graph, vertex, Visit, outputFlag)`；初始条件是图 `graph` 存在；操作结果是对图 `graph` 从顶点 `vertex` 开始进行深度优先搜索遍历，依次对图中的每一个顶点使用函数 `visit` 访问一次，且仅访问一次。

(13) 广度优先搜索遍历：函数名称是 `BFSTraverse(graph, vertex, Visit, outputFlag)`；初始条件是图 `graph` 存在；操作结果是图 `graph` 从顶点 `vertex` 开始进行广度优先搜索遍历，依次对图中的每一个顶点使用函数 `visit` 访问一次，且仅访问一次。

本系统还实现了对多个有向网操作的支持。

本系统还提供了对有向网的文件输入输出功能。本功能通过两个函数实现，两个函数的定义如下。

(1) 保存到文件：函数名称是 `LoadToFile(graph, filename)`；初始条件为有

向网 graph 存在，操作结果是将有向网 graph 存储到文件 filename 中。

(2) 从文件读取有向网：函数名称是 LoadFromFile(graph, filename); 初始条件为有向网 graph 不存在，操作结果为从文件 filename 中读取数据并构建有向网 graph。

## 4.2 系统设计

### 4.2.1 总控流程框架

本系统首先由用户输入待操作的有向网的数量 n，然后建立 n 个不包含顶点的有向网，然后进入一个循环，循环中，首先由用户输入待操作的有向网的编号，然后将有向网的操作指针指向用户选择的有向网，然后由用户输入待执行操作的编号，然后根据用户输入的操作编号进入不同的分支，从而对有向网进行相应的操作，直到用户输入 0，循环结束，算法结束。该算法的流程图如图 4.1 所示。

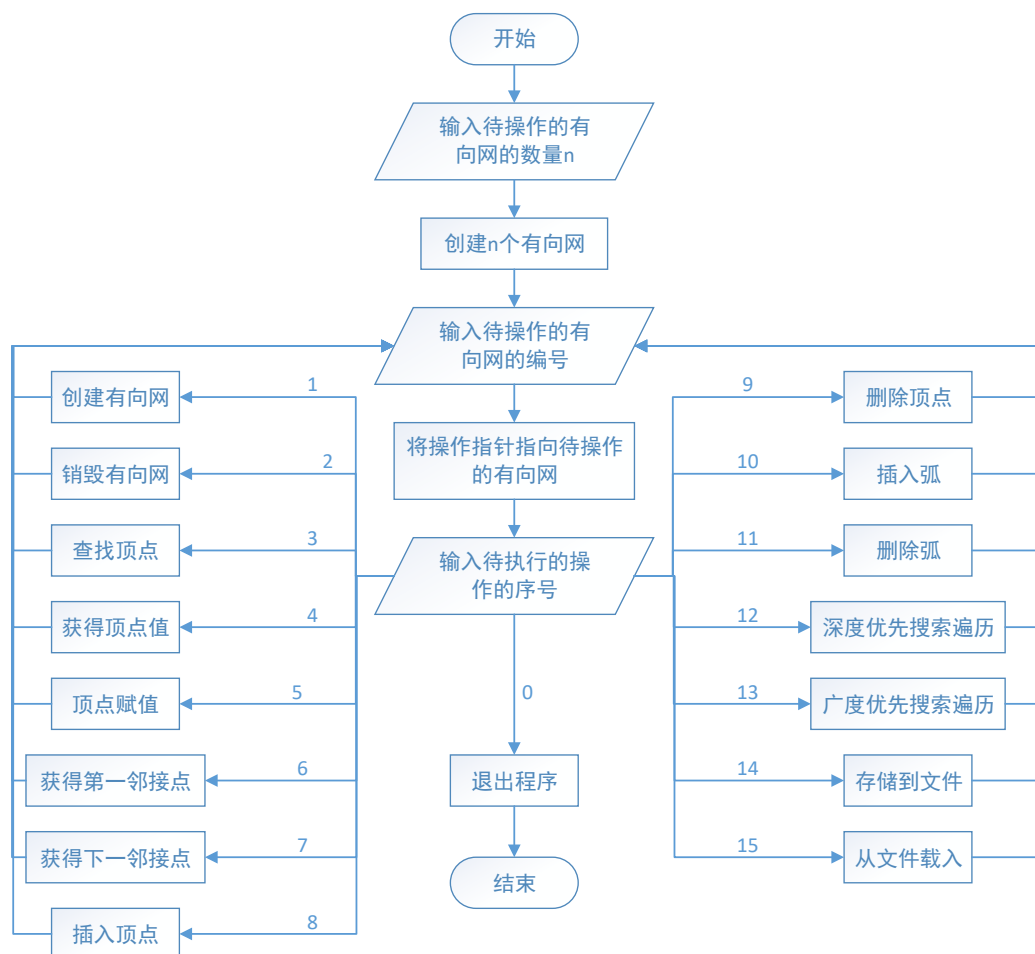


图 4.1 算法流程图

### 4.2.2 定义常量、数据类型及数据结构

1) 定义常量。定义常量作为函数返回值用于说明函数的执行状态。定义的常量有 TRUE(1), FALSE(0), OK(1), ERROR(0), INFEASTABLE(-1), OVERFLOW(-2)。

2) 定义数据元素的类型。此处定义了数据类型 status 用来说明函数的执行状态, VertexType 用来作为有向网的顶点, weight 用来存储弧的权值, vexWeightArc 用来存储有向网的弧的信息, GraphNode 用来表示有向网的邻接表中的邻接点, HeadNode 用来表示有向网的邻接表中的顶点, resultNode 用来表示需要返回顶点指针的函数的返回值。在本系统中, 将 status 定义为整型, VertexType 为两个整型值组成的结构体, weight 为整型, vexWeightArc 为两个整型和一个 weight 类型组成的结构体, GraphNode 为一个整型, 一个 weight 类型和一个 GraphNode 类型的指针组成的结构体, HeadNode 为一个 VertexType 类型和一个 GraphNode 类型的指针组成的结构体, resultNode 为一个 status 类型和一个 HeadNode 类型的指针组成的结构体。

3) 定义有向网的数据结构。有向网邻接表结构体包含三个元素, 分别为整型的有向网顶点数、整型的有向网弧数以及 HeadNode 类型的指针, 该指针为有向网顶点数组的首指针。

有向网邻接表在内存中存储的逻辑结构如图 4.2 所示。

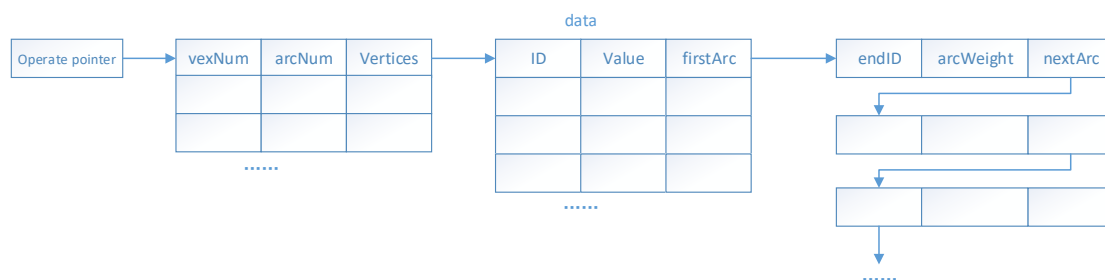


图 4.2 有向网邻接表的存储结构

### 4.2.3 13 个基本操作的设计

关于函数调用的说明, 13 个基本操作函数中, 创建有向网的函数调用时应保证该有向网不存在, 若存在, 则在不调用该函数并告知用户该有向网已存在, 其余 12 个函数在调用时应保证该有向网存在, 若不存在, 则不调用该函数并告知用户该有向网不存在。

### 1) 创建有向网(CreateGraph)

该函数接受 5 个参数，第一个参数为 Graph 类型的指针 graph，指向待创建的有向网的位置，第二个参数为 VertexType 类型的指针 setOfVertex，指向存储有向网顶点信息的数组，第三个参数为整型的 numOfVertex，存储有向网的顶点的数量，第四个参数为 vexWeightArc 类型的指针 infoVertexAndArc，指向有向网的弧信息的数组，第五个参数为整型的 numOfArc，存储有向网的弧的数量。函数首先给 graph 的 vexNum 和 arcNum 元素分别赋值为 numOfVertex 和 numOfArc，然后分配一段长度为 HeadNode 类型长度的 numVertex 倍的内存空间，并将 graph 的 Vertices 元素指向该内存空间，若内存分配失败，则函数返回 OVERFLOW，否则，将 setOfVertex 中的元素逐个的赋值给 graph 的 Vertices 指向的空间。然后函数逐个的读取 numOfArc 的内容，对其中的任意一个元素，函数找到其 startID 对应的顶点，并创建一个新的弧结点插入到该顶点的后面，直到将所有的弧读取完成，函数返回 OK。有向网创建成功。

该函数需要逐个的读取和写入有向网的每一个顶点和弧，因此函数的时间复杂度为  $O(v+e)$  (其中， $v$  为有向网的顶点数量， $e$  为有向网的弧数量)。

### 2) 销毁有向网(DestroyGraph)

该函数接受一个 Graph 类型的指针 graph，该指针指向待销毁的有向网，函数逐个的访问有向网的每一个顶点，对每个顶点的 firstArc 元素调用函数 DestroyArc (该函数的功能为销毁其指向的单链表，其具体定义见辅助函数的设计中第 1 个函数)，并将该元素赋值为空，然后函数释放 graph 的元素 Vertices 指向的内存空间，将 graph 的 arcNum 和 vexNum 元素均赋值为 0，Vertices 元素赋值为空，此时有向网销毁成功，函数返回 OK。

该函数需要逐个的释放有向网的弧的存储空间，但由于有向网的顶点的存储是连续的，因此不需要逐个释放，因此函数的时间复杂度为  $O(e)$  ( $e$  为有向网的弧的数量)。

### 3) 查找顶点(LocateVex)

该函数接受两个参数，第一个为 Graph 类型的指针 graph，该指针指向待操作的有向网，第二个参数为整型的 vertexID，存储待查找顶点的 ID。该函数逐

个的读取有向网的顶点，直到找到 ID 与 vertexID 相等的顶点，函数返回该顶点的指针，若遍历完所有的顶点仍未找到该顶点，则有向网中不存在该顶点，函数返回空。

该函数需要遍历有向网的每个顶点直到找到待查找的顶点，因此时间最短的情况为第一个顶点即待查找的顶点，此时循环执行一次，时间最长的情况为待查找的顶点为有向网的最后一个顶点，或者该顶点不在有向网中，此时循环执行  $v$  次 ( $v$  为有向网的顶点数)，因此函数的时间复杂度为  $O(v)$ 。

#### 4) 获得顶点值 (GetVex)

该函数接受两个参数，第一个参数为 Graph 类型的指针 graph，该指针指向待操作的有向网，第二个参数为整型的 vertexID，存储待获得顶点值的顶点 ID。函数逐个的访问有向网的每个顶点，直到找到顶点 ID 与 vertexID 相等的顶点，函数返回该顶点的内容，若函数访问完每个顶点仍未找到待查找的顶点，则函数返回一个 ID 和 value 均为 -1 的顶点表示该顶点不存在。

该函数需要遍历有向网的每个顶点直到找到待查找的顶点，因此时间最短的情况为第一个顶点集待查找的顶点，此时循环执行一次，时间最长的情况为待查找的顶点为有向网的最后一个顶点，或者该顶点不在有向网中，此时循环执行  $v$  次 ( $v$  为有向网的顶点数)，因此函数的时间复杂度为  $O(v)$ 。

#### 5) 顶点赋值 (PutVex)

该函数接受三个参数，第一个参数为 Graph 类型的指针 graph，该指针指向待操作的有向网，第二个参数为整型的 vertexID，存储待赋值的顶点的 ID，第三个参数为整型的 vertexNewValue，存储待赋的值。首先函数调用 LocateVex 函数，获得 ID 为 vertexID 的顶点的指针，若该指针为空，则说明该顶点不存在，函数返回 ERROR，否则将 vertexNewValue 的值赋给该顶点 value 元素，并返回 OK。

该函数调用了时间复杂度为  $O(v)$  的函数 LocateVex，余下部分均为顺序结构，因此函数的时间复杂度为  $O(v)$  ( $v$  为有向网中的顶点数)。

#### 6) 获得第一邻接点 (FirstAdjVex)

该函数接受两个参数，第一个参数为 Graph 类型的指针 graph，该指针指向



待操作的有向网，第二个参数为整型的 `vertexID`，存储待获得第一邻接点的顶点的 ID。函数首先调用 `LocateVex` 函数获得 ID 为 `vertexID` 的顶点的指针，若该指针为空，则该顶点不存在，函数返回 `OVERFLOW` 和空组成的结构体，若该顶点的 `firstArc` 元素为空，说明该顶点没有邻接点，函数返回 `ERROR` 和空组成的结构体，否则函数调用 `LocateVex` 函数，找到有向网中 ID 为该顶点第一邻接点的 ID 的顶点，函数返回 `OK` 和该顶点指针组成的结构体。

当待获得第一邻接点的顶点不存在或该顶点的第一邻接点不存在时，函数调用一次 `LocateVex` 函数，否则函数调用两次 `LocateVex` 函数，余下部分均为顺序结构，`LocateVex` 函数的时间复杂度为  $O(v)$ ，因此函数的时间复杂度为  $O(v)$  ( $v$  为有向网中的顶点数)。

#### 7) 获得下一邻接点(`NextAdjVex`)

该函数接受三个参数，第一个参数为 `Graph` 类型的指针 `graph`，该指针指向待操作的有向网，第二个参数为整型的 `vertexID`，存储待获得下一邻接点的顶点 ID，第三个参数为整型的 `vertexFirstAdjID`，存储待获得下一邻接点的邻接点的 ID。函数首先调用 `LocateVex` 函数获得 ID 为 `vertexID` 的顶点指针，若该指针为空，则函数返回 `OVERFLOW` 和空组成的结构体，若该顶点的 `firstArc` 元素为空，则函数返回 `ERROR` 和空组成的结构体，否则，调用函数 `FindNode` (该函数的功能为(获得单链表中相应顶点的指针，其具体定义见辅助函数的设计中第 2 个函数)获得该顶点的 ID 为 `vertexFirstAdjID` 的邻接点，若该邻接点为空，说明该顶点不存在该邻接点，因此函数返回 -3 和空组成的结构体，若该邻接点的下一个邻接点为空，则说明该顶点相对于该邻接点不存在下一邻接点，函数返回 -4 和空组成的结构体，否则，调用 `LocateVex` 函数获得该邻接点的下一邻接点的 ID 对应的顶点的指针，函数返回 `OK` 和该指针组成的结构体。

若该顶点不存在或者该顶点不存在邻接点，则函数调用一次 `LocateVex` 函数，若该顶点存在但 `vertexFirstAdjID` 不是该顶点的邻接点或者该邻接点为该顶点的最后一个邻接点，则函数调用一次 `LocateVex` 函数和一次 `FindNode` 函数，否则函数调用两次 `LocateVex` 函数和一次 `FindNode` 函数，由于 `LocateVex` 函数的时间复杂度为  $O(v)$ ，`FindNode` 函数的时间复杂度为  $O(e)$ ，因此函数的时间复

杂度为  $O(v+e)$  ( $v$  为有向网的顶点数,  $e$  为有向网的弧数)。

#### 8) 插入顶点 (InsertVex)

该函数接受两个参数, 第一个参数为 Graph 类型的指针 graph, 第二个参数为 VertexType 类型的 newVertex。首先函数分配长度比原长度大 1 的用于存储有向网顶点的空间, 并将原有向网中的顶点信息全部赋值到现有空间中, 将原有空间释放, 并将 graph 的 Vertices 元素指向新的空间, 然后将 graph 的 vexNum 元素加一, 将 newVertex 的内容赋值给 graph 的 Vertices 的最后一个元素。此时, 顶点插入成功, 函数返回 OK。

该函数将有向网的每个顶点的信息移动到新的内存空间, 因此函数的时间复杂度为  $O(v)$  ( $v$  为有向网的顶点数)。

#### 9) 删除顶点 (DeleteVex)

该函数接受两个参数, 第一个参数为 Graph 类型的指针 graph, 该指针指向待操作的有向网, 第二个参数为整型的 vertexID, 存储待删除顶点的 ID。首先函数调用函数 existNode (该函数的功能为判断有向网中某个顶点是否存在, 其具体定义见辅助函数的设计中第 3 个函数) 判断该顶点是否存在, 若该顶点不存在, 函数返回 ERROR, 否则, 函数逐个的访问有向网的每个顶点, 每访问一个顶点, 若该顶点不为待删除的顶点, 则遍历其邻接顶点, 若存在 ID 为 vertexID 的邻接顶点, 将该邻接点删除, 否则, 将其邻接点全部删除, 并将其后的每个顶点向前移动一个单位, 将 graph 的 vexNum 减 1, 直到遍历完有向网的所有顶点。函数返回 OK。本函数中, 每删除一个邻接点, 都将 graph 的 arcNum 元素减 1。

该函数调用函数 existNode, 若该顶点存在, 则函数遍历有向网的每个顶点, 并遍历每个顶点的邻接顶点, 因此函数的时间复杂度为  $O(v+e)$  ( $v$  为有向网的顶点数,  $e$  为有向网的弧数)。

#### 10) 插入弧 (InsertArc)

该函数接受两个参数, 第一个参数为 Graph 类型的指针 graph, 该指针指向待操作的有向网, 第二个参数为 vexWeightArc 类型的 newArc, 存储待插入的弧的信息。函数首先调用 existNode 函数判断有向网中是否存在 ID 为 newArc 的 endID 的顶点, 若不存在, 则函数返回 OVERFLOW, 表明该弧的终止顶点不存在,

否则，函数逐个访问有向网的每个顶点，直到找到 ID 与 newArc 的 startID 相等的顶点，将该弧插入到该顶点的邻接点链表中，函数返回 OK。若遍历完所有顶点仍未找到该顶点，则说明该弧的起始顶点不存在，函数返回 ERROR。

该函数需遍历有向网的每一个顶点，由于插入邻接点时直接插入为第一个邻接点，因此函数的时间复杂度为  $O(v)$  ( $v$  为有向网的顶点数)。

#### 11) 删除弧 (DeleteArc)

该函数接受三个参数，第一个参数为 Graph 类型的指针 graph，该指针指向待操作的有向网，第二个和第三个参数均为整型，分别为待删除的弧的起始顶点 ID(startID) 和终止顶点 ID(endID)。函数首先调用 existNode 函数判断 ID 为 endID 的顶点是否存在，若不存在该顶点，则函数返回 1，否则，函数逐个访问有向网的顶点，直到找到 ID 为 startID 的顶点，若该顶点的第一邻接点的 ID 为 endID，则函数直接删除掉该邻接点，并返回 0，否则，使用两个相邻的指针逐个的遍历该顶点的每个邻接点，直到找到 ID 为 endID 的邻接点，将该邻接点删除，并返回 0，若访问完所有邻接点仍未找到该邻接点，则返回 3，说明不存在该弧，若访问完所有顶点仍未找到 ID 为 startID 的顶点，则返回 2，说明起始顶点不存在。函数删除一个邻接点后将 graph 的 arcNum 元素减 1。

该函数需要逐个访问有向网的每个顶点，找到 startID 的顶点后还需要逐个访问其邻接点找到 endID 的邻接点，因此函数的时间复杂度为  $O(v+e)$  ( $v$  为有向网的顶点数， $e$  为有向网的弧数)。

#### 12) 深度优先搜索遍历 (DFS Traverse)

该函数接受 4 个参数，第一个参数为 Graph 类型的指针 graph，该指针指向待操作的有向网，第二个参数为 HeadNode 类型的指针 vertex，该指针指向遍历开始的顶点，第三个参数为访问函数 Visit，第四个参数为整型的指针 outputFlag，该指针指向的数组用于标记结点的输出情况。该函数使用全局变量 sum 来记录已输出的数量，若 sum 以有向网的顶点数相等，则函数返回 OK，若当前待访问顶点在 outputFlag 中未标记，即该顶点未输出，则对该顶点调用函数 Visit，并将该顶点标记为已访问，将 sum 加 1，否则函数返回 OK。然后函数对该顶点的所有邻接点依次进行深度优先搜索遍历，全部完成后，函数返回 OK。

该函数会访问邻接表的每个顶点和邻接点，依次函数的时间复杂度为  $O(v+e)$  ( $v$  为有向网的顶点数， $e$  为有向网的弧数)。

#### 13) 广度优先搜索遍历 (BFSTraverse)

该函数接受 4 个参数，第一个参数为 Graph 类型的指针 graph，该指针指向待操作的有向网，第二个参数为 HeadNode 类型的指针 vertex，该指针指向遍历开始的顶点，第三个参数为访问函数 Visit，第四个参数为整型的指针 outputFlag，该指针指向的数组用于标记结点的输出情况。首先该函数创建一个长度为 graph 的顶点数，元素类型为 VertexType 的数组 BFSArray，使用 begin 和 end 来标记当前待进行广度遍历的结点的位置，将这些元素广度遍历的结果存储到 BFSArray 中 end 后面的位置，若 end 等于有向网的顶点数或者一次操作中未向 BFSArray 中添加新的元素，说明广度遍历结束，然后对 BFSArray 中的每一个元素调用 Visit 函数，访问完成后，函数返回 OK。

该函数访问每一个邻接点并查找该邻接点对应的顶点位置，因此函数的时间复杂度为  $O(v+e)$  ( $v$  为有向网的顶点数， $e$  为有向网的弧数)。

### 4.3.4 多有向网操作的设计

本系统首先由用户输入待操作的有向网的数量，然后每次由用户选择待操作的二叉树并对其进行操作。

程序从用户获得待操作的有向网的数量 numGraph 后，创建一个长度为 numGraph、元素类型为 Graph 的数组 graphList，并声明一个变量 graph 作为操作指针。程序将对 graphList 中的每一个元素进行初始化。

在有向网的操作过程中，首先由用户选择需要操作的有向网，然后程序将 graph 指向待操作的有向网，即可使用相同的方法对所有的有向网进行操作。从而实现对多个有向网的操作。

多有向网的存储方式如图 4.2 所示，通过移动操作指针 Operate pointer 指向用户选择的待操作的有向网，从而实现对多个有向网的操作。

### 4.3.5 文件输入输出的设计

本系统对文件输入输出功能的实现通过两个函数，LoadToFile 和

LoadFromFile 实现，其中 LoadToFile 将当前操作的有向网存储到文件中，LoadFromFile 将文件的内容读取并存储到当前操作的有向网中。两个函数的设计如下。

#### 1) 将有向网存储到文件(LoadToFile)

该函数接受两个参数，第一个参数为 Graph 类型的指针 graph，该指针指向待存储到文件的有向网，第二个参数为字符串 filename，为待写入的文件名。函数首先打开文件 filename，若文件打开失败，则函数返回 ERROR，否则，首先将有向网的顶点的数量和弧的数量写入到文件中，然后函数将每个顶点的内容逐个写入到文件中，然后函数将每个顶点的邻接点逐个的写入到文件中，每写完一个顶点的邻接点，都写入一个 ID 和 weight 均为-1 的结点，用来标记该顶点的邻接点写入完成，将所有顶点的邻接点全部写入文件后，关闭文件并返回 OK。

该函数将读取有向网的每个顶点以及每一个邻接点，因此函数的时间复杂度为  $O(v+e)$  ( $v$  为有向网的顶点数， $e$  为有向网的弧数)。

#### 2) 从文件读取数据并载入有向网(LoadFromFile)

该函数接受两个参数，第一个参数为 Graph 类型的指针 graph，该指针指向操作的有向网，第二个参数为字符串 filename，为待读取的文件名。函数首先打开文件 filename，若打开失败，函数返回 ERROR，否则，首先函数从文件读取两个整型的元素，分别为有向网的顶点数和弧数，然后函数声明长度为顶点数，元素类型为 HeadNode 类型的数组，并将 graph 的 Vertices 元素指向该存储空间，然后从文件中读取顶点数量的顶点信息，并将其逐个的写入到该空间中。然后函数从文件中读取邻接点的信息，每读取到一个有效的数据，都将其添加到相应顶点的最后一个邻接点的位置，直到读取到一个 ID 和 weight 均为-1 的邻接点，说明该顶点的邻接点已读取完毕，然后对下一个顶点进行相同的操作，直到将文件中的数据全部读取完毕，关闭文件，函数返回 OK。

该函数将从文件读取有向网中的每一个顶点以及每一个邻接点，并将它们写入到有向网中，因此函数的时间复杂度为  $O(v+e)$  ( $v$  为有向网的顶点数， $e$  为有向网的弧数)。

#### 4.3.6 辅助函数的设计

本系统中部分函数使用了一些有具体含义的非常用功能,将这些功能以函数的形式设计,定义了如下函数。

##### 1) 删除一个顶点的全部邻接点 (DestroyArc)

该函数接受一个 GraphNode 类型的指针 arc,该指针指向待删除的第一个邻接点。函数使用两个指针在该单链表中逐个后移,并删除前面的结点,直到所有结点全部删除。

该函数在销毁有向网函数中被调用。

该函数删除该顶点的所有邻接点,因此函数的时间复杂度为  $O(e)$  ( $e$  为该顶点的邻接点的数量)。

##### 2) 查找邻接点 (FindNode)

该函数接受两个参数,第一个参数为 GraphNode 类型的指针 arc,该指针指向待查找邻接点的顶点的一个邻接点,第二个参数为整型的 vertexID,为待查找的邻接点的 ID。函数将 arc 逐个后移,直到找到 ID 为 vertexID 的邻接点,函数返回 arc,若未找到该邻接点,则函数返回空。

该函数在查找下一邻接点函数中调用。

该函数逐个访问该顶点的邻接点,直到找到待查找的邻接点,因此函数的时间复杂度为  $O(e)$  ( $e$  为该顶点的邻接点的数量)。

##### 3) 判断顶点是否存在 (existNode)

该函数接受两个参数,第一个参数为 Graph 类型的指针 graph,该指针指向待操作的有向网,第二个参数为整型的 vertexID,为待判断是否存在的顶点的 ID。函数逐个访问有向网的顶点,直到找到 ID 为 vertexID 的顶点,若找到该顶点,函数返回 TRUE,否则函数返回 FALSE。

该函数在删除顶点、插入弧、删除弧函数中调用。

该函数逐个访问有向网的每个顶点,因此函数的时间复杂度为  $O(v)$  ( $v$  为有向网的顶点数)。

##### 4) 访问顶点 (visit)

该函数接受两个参数,第一个参数为 Graph 类型的指针 graph,该指针指向

待操作的有向网，第二个参数为整型的 vertexID，为待访问的顶点 ID。函数逐个访问有向网的每个顶点，直到找到 ID 为 vertexID 的顶点，将该顶点的 ID 和 value 打印，并返回 0。

该函数在深度优先搜索遍历和广度优先搜索遍历函数中调用。

该函数逐个访问有向网的每个顶点，因此函数的时间复杂度为  $O(v)$  ( $v$  为有向网的顶点数)。

#### 5) 查找最大 ID(maxID)

该函数接受一个 Graph 类型的指针 graph，该指针指向待操作的有向网，函数首先将 maxid 置为 0，然后逐个访问有向网中的每个顶点，每次都将在 maxid 和当前访问顶点 ID 的较大值，访问完所有顶点后，函数返回 maxid。

该函数在主函数中调用深度优先搜索遍历和广度优先搜索遍历函数前初始化 outputFlag 时调用。

该函数逐个访问有向网的每个顶点，因此函数的时间复杂度为  $O(v)$  ( $v$  为有向网的顶点数)。

### 4.3.7 异常输入的处理

本系统中包含大量用户输入，而用户的输入内容具有不可预见性，因此每次读取用户输入的时候都应该对其进行相应的操作，若输入符合要求，则进行正常操作，否则，应当提示用户输入不合法并要求重新输入，若用户输入合法数据后多输入了其他数据，应当清除缓冲区中的多余数据，防止下次获取输入时出现错误。

对用户的每一次输入都应该执行上述操作以保证程序得到正确的用户输入，从而保证程序能够正常稳定的运行。

## 4.3 系统实现

### 4.3.1 实现方案

本系统使用 C 语言实现，实现方案包括一个头文件(graph.h)和两个源文件(graph.c 和 main.c)。

头文件 graph.h 中定义了常量、数据类型，构建了有向网的数据结构，声明了函数原型并创建了全局变量。

源文件 graph.c 中对头文件中声明的函数进行了逐个的定义。

源文件 main.c 中通过调用各函数对有向网进行各种操作。

完成本实验的操作系统为 Windows 10 64 位，使用 visual studio 2017 编写代码、进行调试以及进行功能测试。

#### 4.3.2 程序设计

本系统的完整程序见附录 4。部分程序的说明如下。

##### 1) 常量、数据类型以数据结构的定义

在头文件 graph.h 中实现了 4.2.2 中对常量、数据类型以及数据结构的定义。使用#define 定义了常量、使用 typedef 定义了数据类型并定义了有向网邻接表的结构。具体的实现代码如下。

```
//定义常量
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASTABLE -1
#define OVERFLOW -2

//定义数据类型
typedef int status;           //函数返回值类型
typedef struct VertexType {
    int ID;
    int value;
} VertexType;               //顶点
typedef int weight;          //弧的权值
typedef struct vexWeightArc {
    int startID;
    int endID;
    weight arcWeight;
} vexWeightArc;
typedef struct GraphNode {
    int endID;               //该弧指向的顶点的 ID
    struct GraphNode * nextArc; //指向下一条弧的指针
```



```

        weight arcWeight;           //该弧的权值
    } GraphNode;
typedef struct HeadNode {
    VertexType data; //顶点信息
    GraphNode * firstArc; //指向第一条依附该顶点的弧的指针
} HeadNode;
typedef struct Graph {
    HeadNode * Vertices;
    int vexNum; //顶点数
    int arcNum; //弧数
} Graph;
typedef struct resultNode {
    status s;
    HeadNode * node;
} resultNode;

```

## 2) 深度优先搜索遍历函数的实现

深度优先搜索遍历函数的定义见 4.2.3 中第 12 个函数的定义，该函数使用递归算法实现，同时使用了标志数组和全局变量来控制对顶点的输出，其具体实现如下。

```

status DFSTraverse(Graph * graph, HeadNode * vertex, int(*Visit)(Graph * graph, int
vertexID), int * outputFlag)
{
    if (sum == graph->vexNum)
        return OK;
    if (outputFlag[vertex->data.ID] == 0)
    {
        Visit(graph, vertex->data.ID);
        outputFlag[vertex->data.ID] = 1;
        sum++;
    }
    else
        return OK;
    GraphNode * node = vertex->firstArc;
    while (1)
    {
        if (node == NULL)
            break;
        DFSTraverse(graph, LocateVex(graph, node->endID), Visit, outputFlag);
        node = node->nextArc;
    }
}

```

```
    return OK;
}
```

### 3) 广度优先搜索遍历函数的实现

广度优先搜索遍历函数的定义见 4.2.3 中第 13 个函数的定义，该函数运用了队列的思想，并使用数组实现，同时该函数还使用了标记数组，来控制顶点的输出状态。该函数的具体实现如下。

```
status BFSTraverse(Graph * graph, HeadNode * vertex, int(*Visit)(Graph *graph, int
vertexID), int * outputFlag)
{
    int i, loc = 0, change;
    VertexType * BFSArray = (VertexType *)malloc(sizeof(VertexType) *
graph->vexNum);
    GraphNode * node;
    int begin, end;
    begin = 0;
    BFSArray[loc].ID = vertex->data.ID;
    BFSArray[loc].value = vertex->data.value;
    outputFlag[vertex->data.ID] = 1;
    loc++;
    end = loc;
    while (end != graph->vexNum)
    {
        change = loc;
        for (i = begin; i < end; i++)
        {
            node = LocateVex(graph, BFSArray[i].ID)->firstArc;
            while (node != NULL)
            {
                if (outputFlag[node->endID] == 0)
                {
                    outputFlag[node->endID] = 1;
                    BFSArray[loc].ID = node->endID;
                    BFSArray[loc].value = LocateVex(graph,
node->endID)->data.value;
                    loc++;
                }
                node = node->nextArc;
            }
        }
        begin = end;
    }
}
```

```

        end = loc;
        if (change == loc)
            break;
    }
    for (i = 0; i < end; i++)
        Visit(graph, BFSArray[i].ID);
    return OK;
}

```

#### 4.3.3 系统测试

本测试中对有向网的 13 个基本操作以及文件输入输出功能进行测试。测试时创建 3 个有向网，局选择第 1 个有向网(编号为 1)作为待操作的有向网。

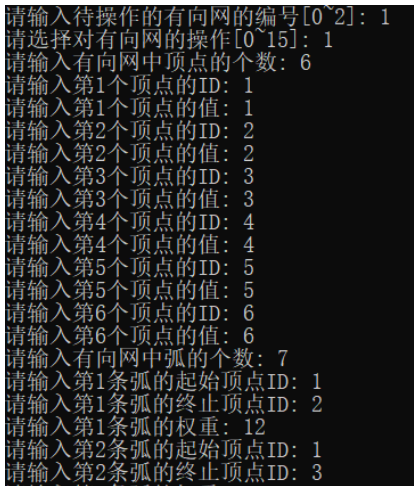
##### 1) 创建有向网(CreateGraph)

创建有向网的正常情况只有一种情况，即待操作的有向网不存在，此时函数根据用户输入创建有向网并返回 OK，程序输出“有向网创建成功”。

异常情况也有一种情况，即待操作的有向网已存在，此时函数不调用，程序输出“二叉树已存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 4-1。

表 4-1 创建有向网函数测试

	输入	理论输出	运行结果(截图)
正常输入	1 1 6 1 1 2 2 3 3 4 4 5 5 6 6 7 1 2 12 1 3 13 2 4 24 3 4 34 3 5 35 4 5 45 5 6 56 选择第 1 个有向网(该有向网不存在)，选择功能 1，建立一个 6 个顶点，7 条弧的有向网	输出“有向网创建成功”	 <pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 1 请输入有向网中顶点的个数: 6 请输入第1个顶点的ID: 1 请输入第1个顶点的值: 1 请输入第2个顶点的ID: 2 请输入第2个顶点的值: 2 请输入第3个顶点的ID: 3 请输入第3个顶点的值: 3 请输入第4个顶点的ID: 4 请输入第4个顶点的值: 4 请输入第5个顶点的ID: 5 请输入第5个顶点的值: 5 请输入第6个顶点的ID: 6 请输入第6个顶点的值: 6 请输入有向网中弧的个数: 7 请输入第1条弧的起始顶点ID: 1 请输入第1条弧的终止顶点ID: 2 请输入第1条弧的权重: 12 请输入第2条弧的起始顶点ID: 1 请输入第2条弧的终止顶点ID: 3 </pre>

			<pre> 请输入第3条弧的起始顶点ID: 2 请输入第3条弧的终止顶点ID: 4 请输入第3条弧的权重: 24 请输入第4条弧的起始顶点ID: 3 请输入第4条弧的终止顶点ID: 4 请输入第4条弧的权重: 34 请输入第5条弧的起始顶点ID: 3 请输入第5条弧的终止顶点ID: 5 请输入第5条弧的权重: 35 请输入第6条弧的起始顶点ID: 4 请输入第6条弧的终止顶点ID: 5 请输入第6条弧的权重: 45 请输入第7条弧的起始顶点ID: 5 请输入第7条弧的终止顶点ID: 6 请输入第7条弧的权重: 56 有向网创建成功! </pre>
异常输入	1 1 选择第 1 个有向网(该有向网存在), 选择功能 1	输出“有向网已存在”	<pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 1 有向网已存在! </pre>

### 2) 销毁有向网 (DestroyGraph)

销毁有向网的正常情况有一种情况, 即待销毁的有向网存在, 此时, 函数销毁有向网并返回 OK, 程序应输出“有向网销毁成功”。

异常情况也有一种情况, 即有向网不存在, 此时函数不调用, 程序输出“有向网不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 4-2。

表 4-2 销毁有向网函数测试

	输入	理论输出	运行结果(截图)
正常输入	1 2 选择第 1 个有向网(该有向网存在), 选择功能 2	输出“有向网销毁成功”	<pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 2 有向网销毁成功! </pre>
异常输入	1 2 选择第 1 个有向网(该有向网不存在), 选择功能 2	输出“有向网不存在”	<pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 2 有向网不存在! </pre>

### 3) 查找顶点 (LocateVex)

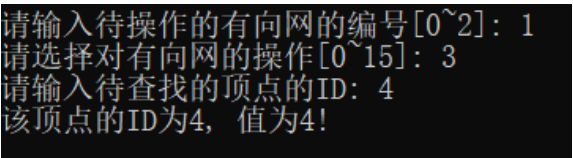
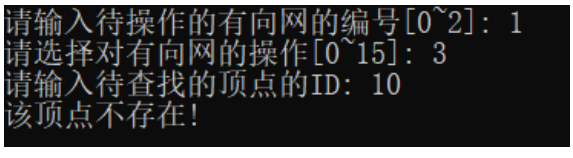
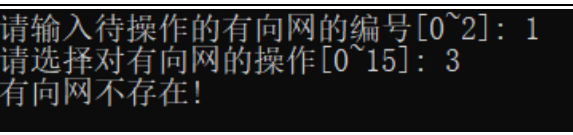
查找顶点的正常情况有一种情况, 即待操作的有向网存在且待查找的顶点在该有向网中, 此时函数返回待查找顶点的指针, 程序应输出该顶点的信息。

异常情况有两种情况, 第一种情况为有向网存在但是不存在待查找顶点, 此时函数返回空, 程序输出“该顶点不存在”; 第二种情况为有向网不存在, 此时

函数不调用，程序输出“有向网不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 4-3。本测试中使用的有向网为函数 1 测试时创建的有向网。

表 4-3 查找顶点函数测试

	输入	理论输出	运行结果(截图)
正常输入	1 3 4 选择第 1 个有向网(该有向网存在), 选择功能 3, 查找 ID 为 4 的顶点	输出“该顶点的 ID 为 4, 值为 4”	
异常输入 1	1 3 10 选择第 1 个有向网(该有向网存在), 选择功能 3, 查找 ID 为 10 的顶点	输出“该顶点不存在”	
异常输入 2	1 3 选择第 1 个有向网(该有向网不存在), 选择功能 3	输出“有向网不存在”	

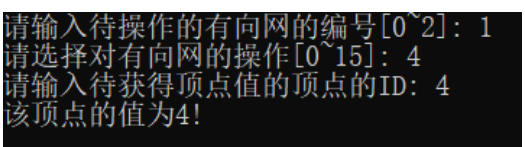
#### 4) 获得顶点值 (GetVex)

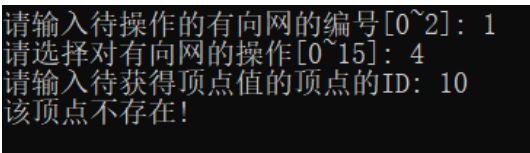
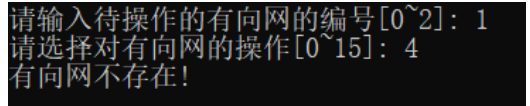
获得顶点值的正常情况有一种情况，即待操作的有向网存在且待获得顶点值的顶点在该有向网中，此时函数返回该顶点的信息，程序应输出该顶点的值。

异常情况有两种情况，第一种情况为有向网存在但是不存在待获得顶点值的顶点，此时函数返回错误顶点的信息 (ID 和 value 均为 -1)，程序输出“该顶点不存在”；第二种情况为有向网不存在，此时函数不调用，程序输出“有向网不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 4-4。本测试中使用的有向网为函数 1 测试时创建的有向网。

表 4-4 获得顶点值函数测试

	输入	理论输出	运行结果(截图)
正常输入	1 4 4 选择第 1 个有向网(该有向网存在), 选择功能 4, 获得 ID 为 4 的顶点的值	输出“该顶点的值为 4”	

异常输入 1	1 4 10 选择第 1 个有向网(该有向网存在), 选择功能 4, 获得 ID 为 10 的顶点的值	输出“该顶点不存在”	 <pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 4 请输入待获得顶点值的顶点的ID: 10 该顶点不存在!                     </pre>
异常输入 2	1 4 选择第 1 个有向网(该有向网不存在), 选择功能 4	输出“有向网不存在”	 <pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 4 有向网不存在!                     </pre>

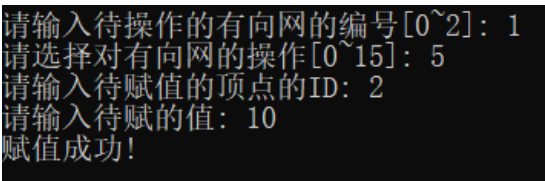
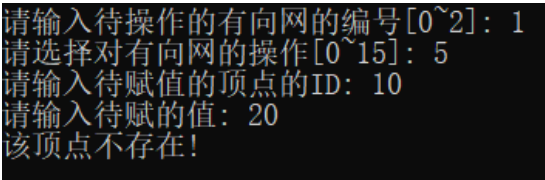
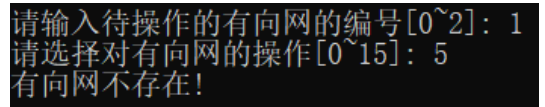
#### 5) 顶点赋值(PutVex)

顶点赋值函数的正常情况有一种情况, 即待操作的有向网存在且待赋值的顶点在该有向网中, 此时函数给该顶点赋值, 并返回 OK。

异常情况有两种情况, 第一种情况为待操作的有向网存在, 但待赋值的顶点不在有向网中, 此时, 函数返回 ERROR, 程序应输出“该顶点不存在”; 第二种情况为待操作的有向网不存在, 此时函数不调用, 程序输出“有向网不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 4-5。本测试中使用的有向网为函数 1 测试时创建的有向网。

表 4-5 顶点赋值函数测试

	输入	理论输出	运行结果(截图)
正常输入	1 5 2 10 选择第 1 个有向网(已存在), 选择功能 5, 给 ID 为 2 的顶点赋值为 10	输出“赋值成功”	 <pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 5 请输入待赋值的顶点的ID: 2 请输入待赋的值: 10 赋值成功!                     </pre>
异常输入 1	1 5 10 20 选择第 1 个有向网(已存在), 选择功能 5, 给 ID 为 10 的顶点赋值为 20	输出“该顶点不存在”	 <pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 5 请输入待赋值的顶点的ID: 10 请输入待赋的值: 20 该顶点不存在!                     </pre>
异常输入 2	1 5 选择第 1 个有向网(不存在), 选择功能 5	输出“有向网不存在”	 <pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 5 有向网不存在!                     </pre>

#### 6) 获得第一邻接点(FirstAdjVex)

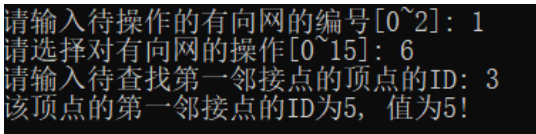
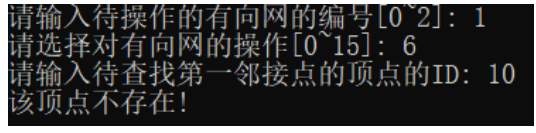
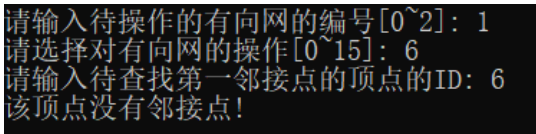
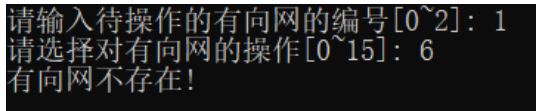
该函数的正常情况有一种情况, 即待操作的有向网存在却待查找邻接点的顶点存在且该顶点存在邻接点, 此时函数返回 OK 和该顶点的第一邻接点对应的顶

点的指针，程序输出该邻接点的信息。

异常情况有三种情况，第一种情况为有向网存在，但是待查找邻接点的顶点不在有向网中，此时函数返回 OVERFLOW，程序输出“该顶点不存在”；第二种情况为有向网存在且待查找邻接点的顶点在有向网中，但是该顶点没有邻接点，此时函数返回 ERROR，程序输出“该顶点没有邻接点”；第三种情况为有向网不存在，此时函数不调用，程序输出“有向网不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 4-6。本测试中使用的有向网为函数 1 测试时创建的有向网。

表 4-6 获得第一邻接点函数测试

	输入	理论输出	运行结果(截图)
正常输入	1 6 3 选择第 1 个有向网 (该有向网存在)，选择功能 6，查找 ID 为 3 的顶点的第一邻接点	输出“该顶点的第一邻接点的 ID 为 5，值为 5”	
异常输入 1	1 6 10 选择第 1 个有向网 (该有向网存在)，选择功能 6，查找 ID 为 10 的顶点的第一邻接点	输出“该顶点不存在”	
异常输入 2	1 6 6 选择第 1 个有向网 (该有向网存在)，选择功能 6，查找 ID 为 6 的顶点的第一邻接点	输出“该顶点没有邻接点”	
异常输入 3	1 6 选择第 1 个有向网 (该有向网不存在)，选择功能 6	输出“有向网不存在”	

#### 7) 获得下一邻接点 (NextAdjVex)

该函数的正常情况有一种情况，即待操作的有向网存在、待查找下一邻接点的顶点在该有向网中、待查找下一邻接点的邻接点为该顶点邻的接点且该邻接点

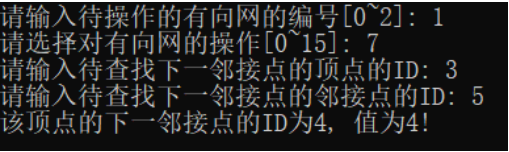
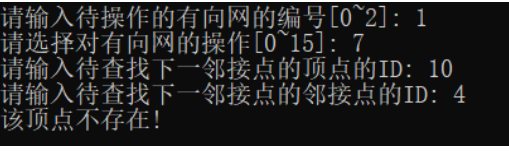
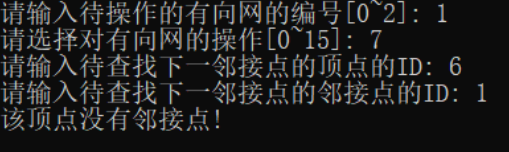


存在下一邻接点，此时，函数返回 OK 和下一邻接点对应的顶点的指针。程序输出下一邻接点的信息。

异常情况有 5 种情况，第一种情况为有向网存在但是待查找邻接点的顶点不在该有向网中，此时函数返回 OVERFLOW 和空指针，程序输出“该顶点不存在”；第二种情况为该顶点存在但是没有邻接点，此时函数返回 ERROR 和空指针，程序输出“该顶点没有邻接点”；第三种情况为该顶点存在但是该邻接点不是该顶点的邻接点，此时函数返回-3 和空指针，程序输出“该邻接点不是该顶点的邻接点”；第四种情况为该邻接点是该顶点的邻接点，但是该邻接点为最后一个邻接点，此时函数返回-4 和空指针，程序输出“该邻接点为该顶点的最后一个邻接点”，第五种情况为该有向网不存在，此时函数不调用，程序输出“有向网不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 4-7。本测试中使用的有向网为函数 1 测试时创建的有向网。

表 4-7 获得下一邻接点函数测试

	输入	理论输出	运行结果(截图)
正常输入	1 7 3 5 选择第 1 个有向网(该有向网存在)，选择功能 7，查找 ID 为 3 的顶点的 ID 为 5 的邻接点的下一个邻接点	输出“该顶点的下一邻接点的 ID 为 4，值为 4”	
异常输入 1	1 7 10 4 选择第 1 个有向网(该有向网存在)，选择功能 7，查找 ID 为 10 的顶点的 ID 为 4 的邻接点的下一个邻接点	输出“该顶点不存在”	
异常输入 2	1 7 6 1 选择第 1 个有向网(该有向网存在)，选择功能 7，查找 ID 为 6 的顶点的 ID 为 1 的邻接点的下一个邻接点	输出“该顶点没有邻接点”	



异常输入 3	1 7 1 5 选择第 1 个有向网(该有向网存在), 选择功能 7, 查找 ID 为 1 的顶点的 ID 为 5 的邻接点的下一个邻接点	输出“该邻接点不是该顶点的邻接点”	<pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 7 请输入待查找下一邻接点的顶点的ID: 1 请输入待查找下一邻接点的邻接点的ID: 5 该邻接点不是该顶点的邻接点!                     </pre>
异常输入 4	1 7 3 4 选择第 1 个有向网(该有向网存在), 选择功能 7, 查找 ID 为 3 的顶点的 ID 为 4 的邻接点的下一个邻接点	输出“该邻接点为该顶点的最后一个邻接点”	<pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 7 请输入待查找下一邻接点的顶点的ID: 3 请输入待查找下一邻接点的邻接点的ID: 4 该邻接点为该顶点的最后一个邻接点!                     </pre>
异常输入 5	1 7 选择第 1 个有向网(该有向网不存在), 选择功能 7	输出“有向网不存在”	<pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 7 有向网不存在!                     </pre>

#### 8) 插入顶点(InsertVex)

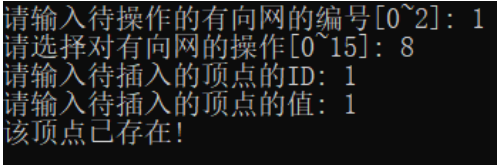
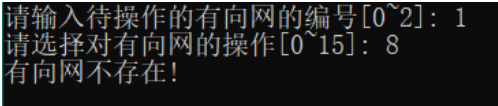
该函数的正常情况有一种情况, 即待操作的有向网存在且有向网中不存在该元素, 此时函数返回 OK, 程序输出“插入成功”。

异常情况有三种情况, 第一种情况为有向网存在但是待插入的顶点以及在有向网中, 此时函数返回 ERROR, 程序输出“该顶点已存在”; 第二种情况为空间不足, 再分配新的存储空间时出现问题, 此时函数返回 OVERFLOW, 程序输出“空间不足, 插入失败”; 第三种情况为待操作的有向网不存在, 此时函数不调用, 程序输出“插入成功”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 4-8。本测试中使用的有向网为函数 1 测试时创建的有向网。

表 4-8 插入顶点函数测试

	输入	理论输出	运行结果(截图)
正常输入	1 8 10 10 选择第 1 个有向网(该有向网已存在), 选择功能 8, 插入一个 ID 为 10, 值为 10 的顶点	输出“插入成功”	<pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 8 请输入待插入的顶点的ID: 10 请输入待插入的顶点的值: 10 插入成功!                     </pre>

异常输入 1	1 8 1 1 选择第 1 个有向网(该有向网已存在), 选择功能 8, 插入一个 ID 为 1, 值为 1 的顶点	输出“该顶点已存在”	 <pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 8 请输入待插入的顶点的ID: 1 请输入待插入的顶点的值: 1 该顶点已存在!                     </pre>
异常输入 2	1 8 10 10 选择第 1 个有向网(该有向网已存在), 选择功能 8, 插入一个 ID 为 10, 值为 10 的顶点	输出“空间不足, 插入失败”	本系统对内存要求较小, 一般不会出现此情况
异常输入 3	1 8 选择第 1 个有向网(该有向网不存在), 选择功能 8	输出“有向网不存在”	 <pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 8 有向网不存在!                     </pre>

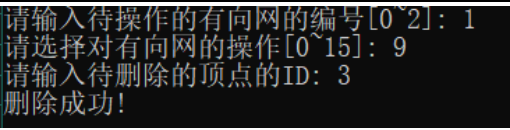
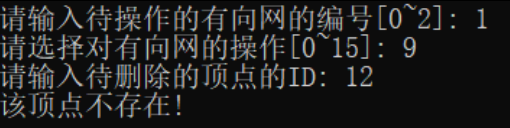
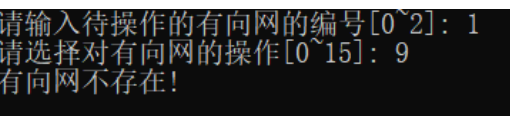
#### 9) 删除顶点>DeleteVex)

该函数的正常情况有一种情况, 即有向网存在且待删除的顶点在有向网中, 此时函数删除该顶点以及与之相关的弧, 返回 OK, 出现输出“删除成功”。

异常情况有两种情况, 第一种情况为有向网存在但是待删除的顶点不在该有向网中, 此时函数返回 ERROR, 程序输出“该顶点不存在”; 第二种情况为有向网不存在, 此时函数不调用, 程序输出“有向网不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 4-9。本测试中使用的有向网为函数 1 测试时创建的有向网。

表 4-9 删除顶点函数测试

	输入	理论输出	运行结果(截图)
正常输入	1 9 3 选择第 1 个有向网(该有向网存在), 选择功能 9, 删除 ID 为 3 的顶点	输出“删除成功”	 <pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 9 请输入待删除的顶点的ID: 3 删除成功!                     </pre>
异常输入 1	1 9 12 选择第 1 个有向网(该有向网存在), 选择功能 9, 删除 ID 为 12 的顶点	输出“该顶点不存在”	 <pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 9 请输入待删除的顶点的ID: 12 该顶点不存在!                     </pre>
异常输入 2	1 9 选择第 1 个有向网(该有向网不存在), 选择功能 9	输出“有向网不存在”	 <pre> 请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 9 有向网不存在!                     </pre>

#### 10) 插入弧(InsertArc)

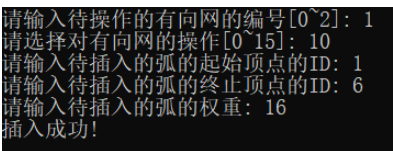
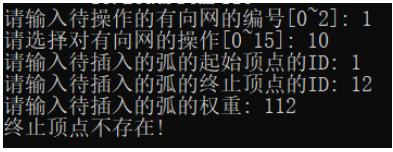
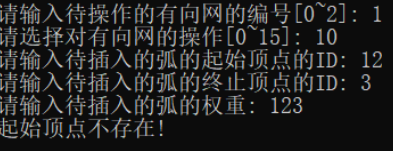
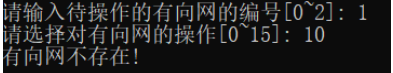
该函数的正常情况有一种情况，即有向网存在且待插入的弧的起始和终止顶点都在有向网中，此时函数插入该弧并返回 OK，程序输出“插入成功”。

异常情况有三种情况，第一种情况为有向网存在但是待插入的弧的终止顶点不存在，此时函数返回 OVERFLOW，程序输出“终止顶点不存在”；第二种情况为有向网存在但是待插入的弧的起始顶点不存在，此时函数返回 ERROR，程序输出“起始顶点不存在”；第三种情况为有向网不存在，此时函数不调用，程序输出“有向网不存在”。

由于本系统中先判断终止顶点是否存在，因此当起始顶点和终止顶点均不存在时，程序输出“终止顶点不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 4-10。本测试中使用的有向网为函数 1 测试时创建的有向网。

表 4-10 插入弧函数测试

	输入	理论输出	运行结果(截图)
正常输入	1 10 1 6 16 选择第 1 个有向网(该有向网存在)，选择功能 10，插入一条从 ID 为 1 的顶点指向 ID 为 6 的顶点的弧，该弧的权值为 16	输出“插入成功”	
异常输入 1	1 10 1 12 112 选择第 1 个有向网(该有向网存在)，选择功能 10，插入一条从 ID 为 1 的顶点指向 ID 为 12 的顶点的弧，该弧的权值为 112	输出“终止顶点不存在”	
异常输入 2	1 10 12 3 123 选择第 1 个有向网(该有向网存在)，选择功能 10，插入一条从 ID 为 12 的顶点指向 ID 为 3 的顶点的弧，该弧的权值为 123	输出“起始顶点不存在”	
异常输入 3	1 10 选择第 1 个有向网(该有向网不存在)，选择功能 10	输出“有向网不存在”	

#### 11) 删除弧(DeleteArc)

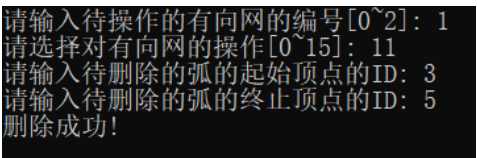
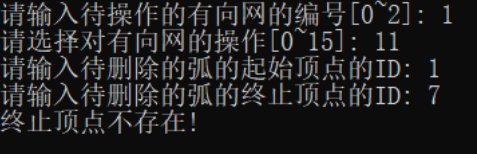
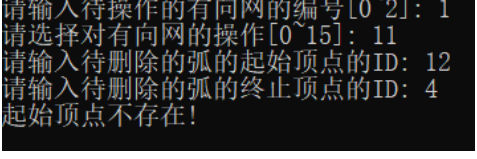
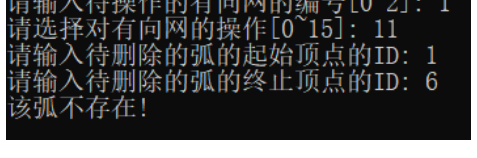
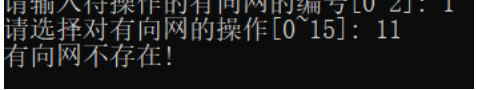
该函数的正常情况有一种情况，即有向网存在且待删除的弧存在，此时函数删除该弧并返回 0，程序输出“删除成功”。

异常情况有四种情况，第一种情况为有向网存在但是起始顶点不存在，此时函数返回 2，程序输出“起始顶点不存在”；第二种情况为有向网存在但是终止不存在，此时函数返回 1，程序输出“终止顶点不存在”；第三种情况为有向网存在、起始及终止顶点均存在但是不存在该弧，此时函数返回 3，程序输出“该弧不存在”；第四种情况为有向网不存在，此时函数不调用，程序输出“有向网不存在”。

由于本系统中先判断终止顶点是否存在，因此当起始顶点和终止顶点均不存在时，程序输出“终止顶点不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 4-11。本测试中使用的有向网为函数 1 测试时创建的有向网。

表 4-11 删除弧函数测试

	输入	理论输出	运行结果(截图)
正常输入	1 11 3 5 选择第 1 个有向网(该有向网存在)，选择功能 11，删除有 ID 为 3 的顶点指向 ID 为 5 的顶点的弧	输出“删除成功”	
异常输入 1	1 11 1 7 选择第 1 个有向网(该有向网存在)，选择功能 11，删除有 ID 为 1 的顶点指向 ID 为 7 的顶点的弧	输出“终止顶点不存在”	
异常输入 2	1 11 12 4 选择第 1 个有向网(该有向网存在)，选择功能 11，删除有 ID 为 12 的顶点指向 ID 为 4 的顶点的弧	输出“起始顶点不存在”	
异常输入 3	1 11 1 6 选择第 1 个有向网(该有向网存在)，选择功能 11，删除有 ID 为 1 的顶点指向 ID 为 6 的顶点的弧	输出“该弧不存在”	
异常输入 4	1 11 选择第 1 个有向网(该有向网不存在)，选择功能 11	输出“有向网不存在”	

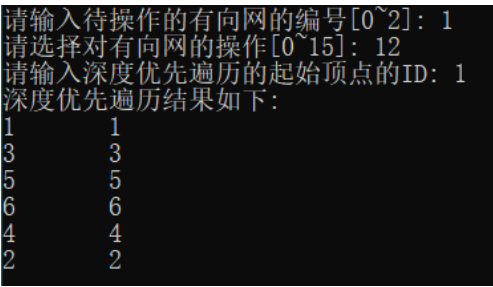
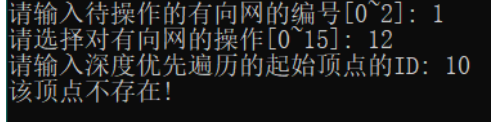
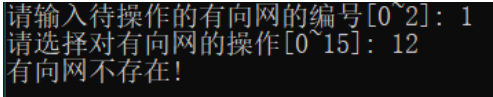
## 12) 深度优先搜索遍历 (DFS Traverse)

该函数的正常情况有一种情况，即待遍历的有向网存在且起始顶点在有向网中，此时函数对有向网指向深度优先搜索遍历，并返回 OK。

异常情况有两种情况，第一种情况为有向网存在但是起始顶点不在有向网中，此时函数不调用，程序输出“该顶点不存在”；第二种情况为有向网不存在，此时函数不调用，程序输出“有向网不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 4-12。本测试中使用的有向网为函数 1 测试时创建的有向网。

表 4-12 深度优先搜索遍历函数测试

	输入	理论输出	运行结果(截图)
正常输入	1 12 1 选择第 1 个有向网(该有向网存在)，选择功能 12，从 ID 为 1 的顶点开始遍历	输出有向网的深度优先遍历结果 1 3 4 5 6 2	
异常输入 1	1 12 10 选择第 1 个有向网(该有向网存在)，选择功能 12，从 ID 为 10 的顶点开始遍历	输出“该顶点不存在”	
异常输入 2	1 12 选择第 1 个有向网(该有向网不存在)，选择功能 12	输出“有向网不存在”	

## 13) 广度优先搜索遍历 (BFS Traverse)

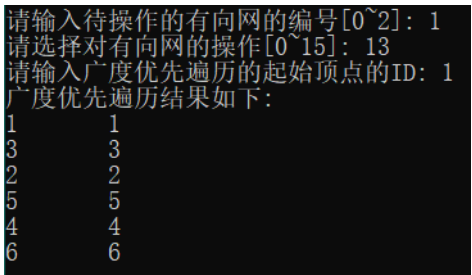
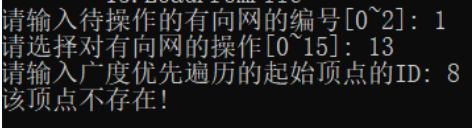
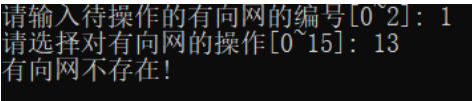
该函数的正常情况有一种情况，即待遍历的有向网存在且起始顶点在有向网中，此时函数对有向网进行广度优先搜索遍历，并返回 OK。

异常情况有两种情况，第一种情况为有向网存在但是起始顶点不在有向网中，此时函数不调用，程序输出“该顶点不存在”；第二种情况为有向网不存在，此时函数不调用，程序输出“有向网不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 4-13。本测试中使用的有向网为函数 1 测试时创建的有向网。



表 4-13 广度优先搜索遍历函数测试

	输入	理论输出	运行结果(截图)
正常输入	1 13 1 选择第 1 个有向网(该有向网存在), 选择功能 13, 从 ID 为 1 的顶点开始遍历	输出有向网的广度优先遍历结果, 该结果为 1 3 2 5 4 6	
异常输入 1	1 13 8 选择第 1 个有向网(该有向网存在), 选择功能 13, 从 ID 为 8 的顶点开始遍历	输出“该顶点不存在”	
异常输入 2	1 13 选择第 1 个有向网(该有向网不存在), 选择功能 13	输出“有向网不存在”	

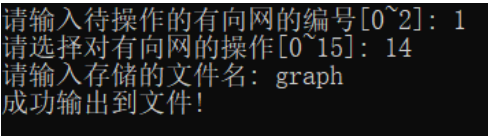
#### 14) 存储到文件(LoadToFile)

该函数的正常情况有一种情况, 即有向网存在且文件成功打开, 此时函数将有向网的内容写入到文件中, 并返回 OK。

异常情况有两种情况, 第一种情况为有向网存在但是文件无法打开, 此时函数返回 ERROR, 程序输出“文件打开失败”; 第二种情况为有向网不存在, 此时函数不调用, 程序输出“有向网不存在”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 4-14。本测试中使用的有向网为函数 1 测试时创建的有向网。

表 4-14 存储到文件函数测试

	输入	理论输出	运行结果(截图)
正常输入	1 14 graph 选择第 1 个有向网(该有向网存在), 选择功能 14, 将有向网存储到存储到文件 graph 中(有写文件权限)	输出“成功输出到文件”	

异常输入 1	1 14 graph 选择第 1 个有向网(该有向网存在), 选择功能 14, 将有向网存储到存储到文件 graph 中(无写文件权限)	输出“文件打开失败”	请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 14 请输入存储的文件名: graph 文件打开失败!
异常输入 2	1 14 选择第 1 个有向网(该有向网不存在), 选择功能 14	输出“有向网不存在”	请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 14 有向网不存在!

#### 15) 从文件载入有向网(LoadFromFile)

该函数的正常情况只有一种情况, 即有向网不存在且文件打开成功, 此时函数将文件中的内容载入到有向网中, 程序输出“成功载入有向网”。

异常情况有两种情况, 第一种情况为有向网不存在但是文件打开失败, 此时函数返回 ERROR, 程序输出“文件打开失败”; 第二种情况为有向网存在, 此时函数不调用, 程序输出“有向网中已存在内容, 请清空后再从文件载入”。

具体的测试样例、理论输出结果以及程序实际输出结果见表 4-15。本测试中使用的有向网文件为函数 14 测试时输出的文件。

表 4-15 从文件载入有向网函数测试

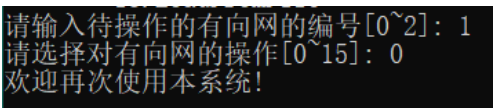
	输入	理论输出	运行结果(截图)
正常输入	1 15 graph 选择第 1 个有向网(该有向网不存在), 选择功能 15, 从文件 graph(文件存在)中读取有向网	输出“成功载入有向网”	请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 15 请输入读取的文件名: graph 成功载入有向网!
异常输入 1	1 15 graph2 选择第 1 个有向网(该有向网不存在), 选择功能 15, 从文件 graph(文件不存在)中读取有向网	输出“文件打开失败”	请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 15 请输入读取的文件名: graph2 文件打开失败!
异常输入 2	1 15 选择第 1 个有向网(该有向网存在), 选择功能 15	输出“有向网中已存在内容, 请清空后再从文件载入”	请输入待操作的有向网的编号[0~2]: 1 请选择对有向网的操作[0~15]: 15 有向网中已存在内容, 请清空后再从文件载入!

#### 16) 退出程序

退出程序的功能与有向网的状态无关, 不存在异常情况。

具体的测试样例、理论输出结果以及程序实际输出结果见表 4-16。

表 4-16 退出程序功能测试

	输入	理论输出	运行结果(截图)
正常输入	1 0 选择第 1 个有向网, 选择功能 0	输出“欢迎再次使用本系统”	

#### 4.3.4 其他问题的说明

在测试环境(Windows 10 64 位)中, 若可执行文件在 C 盘根目录下, 则无法使用 LoadToFile 函数将二叉树写入到不存在的文件中, 原因为: 程序没有权限在 C 盘根目录下创建文件。若可执行文件位于 C 盘根目录下, 则因使用管理员权限运行该程序, 才能够正常使用文件 I/O 功能。

#### 4.4 实验小结

通过本次实验, 我对基于邻接表的图结构有了更深刻的认识。

这次实验中, 我对图以及邻接表有了更加深刻直观的了解。图是一种比较复杂的数据结构, 图有多种类型, 如有向图、无向图、有向网、无向网等, 而邻接表这种存储结构可以用于存储各种不同的图, 相比于图的其他存储结构, 邻接表具有一定的通用性; 另外邻接表仅存储了图中存在的信息, 因此相比于邻接矩阵的存储方式, 邻接表的存储密度更大, 可以节省更多的内存空间。

不过, 邻接表的存储结构也有一些不足之处, 例如使用邻接表表示图在对弧的查找等操作的时间复杂度相比于邻接矩阵要高。



## 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] 严蔚敏等. 数据结构题集 (C 语言版). 清华大学出版社