

**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201804**

**学 号： U201814755**

**姓 名： 彭子晨**

**指导教师： 袁 凌**

**报告日期： 2019年 10月 28 日**

**计算机科学与技术学院**

目 录

[1 基于顺序存储结构的线性表实现 2](#_Toc27655560)

[1.1 问题描述 2](#_Toc27655561)

[1.2 系统设计 3](#_Toc27655562)

[1.3 系统实现 4](#_Toc27655563)

[1.4 系统测试 5](#_Toc27655564)

[1.5 实验小结 15](#_Toc27655565)

[2 基于链式存储结构的线性表实现 16](#_Toc27655566)

[2.1 问题描述 16](#_Toc27655567)

[2.2 系统设计 17](#_Toc27655568)

[2.3 系统实现 18](#_Toc27655569)

[2.4 系统测试 20](#_Toc27655570)

[2.5 实验小结 29](#_Toc27655571)

[3 基于二叉链表的二叉树实现 30](#_Toc27655572)

[3.1 问题描述 30](#_Toc27655573)

[3.2 系统设计 32](#_Toc27655574)

[3.3 系统实现 34](#_Toc27655575)

[3.4 系统测试 36](#_Toc27655576)

[3.5 实验小结 48](#_Toc27655577)

[4 基于邻接表的图实现 50](#_Toc27655578)

[4.1 问题描述 50](#_Toc27655579)

[4.2 系统设计 51](#_Toc27655580)

[4.3 系统实现 53](#_Toc27655581)

[4.4 系统测试 54](#_Toc27655582)

[4.5 实验小结 62](#_Toc27655583)

[附录A 基于顺序存储结构线性表实现的源程序 65](#_Toc27655584)

[附录B 基于链式存储结构线性表实现的源程序 74](#_Toc27655585)

[附录C 基于二叉链表二叉树实现的源程序 84](#_Toc27655586)

[附录D 基于邻接表图实现的源程序 103](#_Toc27655587)

# 1 基于顺序存储结构的线性表实现

## 问题描述

通过实验达到：

（1）加深对线性表的概念、基本运算的理解；

（2）熟练掌握线性表的逻辑结构与物理结构的关系；

（3）物理结构采用顺序表,熟练掌握线性表的基本运算的实现。

**1.1.1 具体问题**

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

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

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

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

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

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

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

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

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

（9）获得后继：函数名称是NextElem(L,cur\_e,next\_e)；初始条件是线性表L已存在；操作结果是若cur\_e是L的数据元素，且不是最后一个，则用next\_e返回它的后继，否则操作失败，next\_e无定义。

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

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

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

## 1.2 系统设计

**1.2.1 总体设计**

（1）使用一个大的while语句实现整体功能；

（2）使用switch语句实现菜单栏的操作选择；

（3）在每个case语句先进行对表是否存在的判断；

**1.2.2 算法设计**

（1）InitList(\*L);

设计思路：给L.elem分配存储空间；使表长的值为0；修改全局变量isNull为false（表存在）；

（2）DestroyList(\*L);

设计思路：使表长的值为0；修改isNull为true（表不存在）；释放L.elem的空间；

（3）ClearList(\*L);

设计思路：使表长的值为0；

（4）ListEmpty(L);

设计思路：判断表长是否为0，是则返回true，否则返回false；

（5）ListLength(L);

设计思路：返回表长L.length；

（6）­GetElem(L,i,\*e)；

设计思路：如果线性表为空，则显示获取失败；然后判断输入i是否在已有数据的位置范围内；将L.elem[i-1]赋值给e；

（7）LocateElem(L,e,Compare());

设计思路：遍历表中数据，返回满足Compare函数条件的元素位置；

（8）PriorElem(L,cur\_e,\*pre\_e);

设计思路：遍历表中数据，如果输入元素为第一个元素或未找到元素则返回错误；将L.elem[i-1]赋值给pre\_e；

（9）NextElem(L,cur\_e,\*next\_e);

设计思路：遍历表中数据，如果输入元素为最后一个元素或未找到元素则返回错误；将L.elem[i+1]赋值给next\_e；

（10）ListInsert(\*L,i,e);

设计思路：如果位置i不合理则报错；如果顺序表已满则分配新的存储空间；将插入位置后的元素整体右移，表长增加1，再插入e；

（11）ListDelete(\*L,i,&e);

设计思路：如果位置i不合理则报错；将要删除元素赋值给e，将删除位置后的元素整体左移；表长减少1，返回e的值；

（12）ListTraverse(L);

设计思路：利用for循环语句遍历顺序表并输出。

## 1.3 系统实现

**1.3.1 函数实现**

（1）InitList：为表分配存储空间，若存储分配成功，头地址L为存储空间基址,表长为0，表的存储容量为100，返回OK；否则返回ERROR；

（2）DestroyList：表长置零，表存在变量设置为否，释放存储空间，使头地址指空，成功则返回OK；

（3）ClearList：清空表，令表长为0；如果成功就返回OK；否则返回ERROR；

（4）ListEmpty：判断表是否为空，即判断表长是否为0；

（5）ListLength：返回当前表长值；

（6）GetElem：获取元素位置。输入一个位置数，如果该位置数不小于1且在表长范围内，则用e返回此位置数据元素的值；否则返回ERROR；

（7）LocateElem：定位元素。输入一个元素，遍历全表，如果表内有与之满足Compare()关系的元素则把元素的位置赋值给e，返回e的值；否则返回ERROR；

（8）PriorElem：获得前驱。输入一个元素，遍历全表，如果找到该元素且位置不在表头，将该元素的前驱元素赋值给pre\_e，并返回OK；否则返回ERROR；

（9）NextElem：获得后继。输入一个元素，遍历全表，如果找到该元素且位置不在表尾，将该元素的后继元素赋值给next\_e，并返回OK；否则返回ERROR；

（10）ListInsert：插入元素。输入插入位置、插入元素的值，如果表内空间不足则开辟新空间；如果该位置数不小于1且在表长范围内，则将该位置后的数逐个后移一位次，在空出的位置插入元素，表长加1并返回OK；否则返回ERROR；

（11）ListDelete：删除元素。输入删除位置，如果该位置数不小于1且在表长范围内，则删除该元素，并将该位置后的数逐个前移一位次；表长减1并将删除元素值赋给e，返回e的值；否则返回ERROR；

（12）ListTrabverse：遍历全表。使用For循环将表内元素顺序输出；

**1.3.2 源代码**

（源代码详细见附录A）

## 1.4 系统测试

编程环境：Win10

编译器：Visual Studio 2019

主模块运行如图1-1所示：

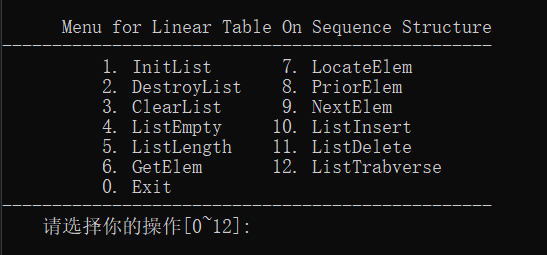


图1-1 主模块图

功能操作：

1. InitList：为表分配存储空间，若存储分配成功，头地址L为存储空间基址,表长为0，表的存储容量为100，返回OK；否则返回ERROR；
2. 线性表不存在，则创建新表成功；（如图1-2所示）
3. 线性表已存在，则清除之前的表并新建空表；（如图1-3所示）

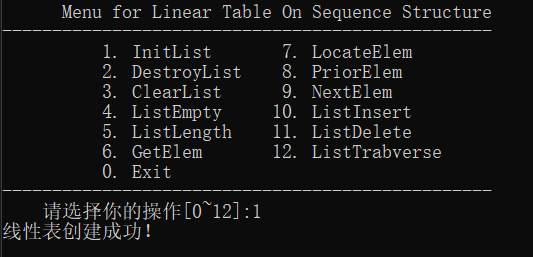


图1-2 顺序表创建

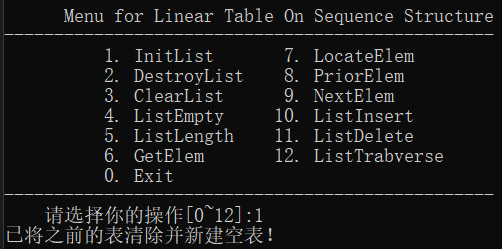


图1-3 清除之前的表并创建新表

1. DestroyList：表长置零，表存在变量设置为否，释放存储空间，使头地址指空，成功则返回OK；
2. 线性表已存在，则销毁该表；（如图1-4所示）
3. 线性表不存在，则返回操作失败；

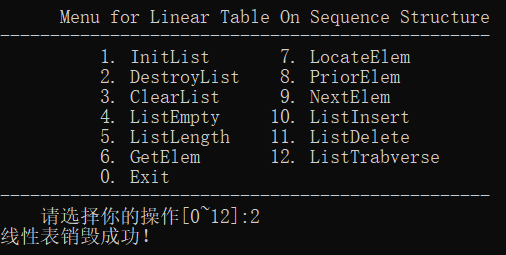


图1-4 线性表销毁成功

1. ClearList：清空表，令表长为0；如果成功就返回OK；
2. 线性表已存在，则清空该表；（如图1-5所示）
3. 线性表不存在，则返回操作失败；

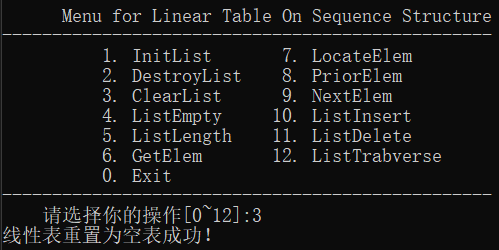


图1-5 线性表清空成功

1. ListEmpty：判断表是否为空，判断表长是否为0；为验证该功能，调用后面函数；
2. 新建一个表，插入元素，遍历；（如图1-6所示）
3. 此时表不为空，返回表不为空；（如图1-7所示）
4. 使用ClearList清空表后，返回表为空；（如图1-8所示）
5. 线性表不存在，则返回操作失败；

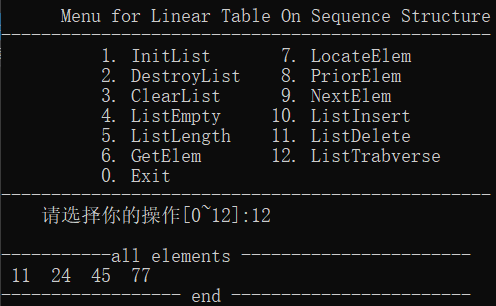


图1-6 线性表内含有元素

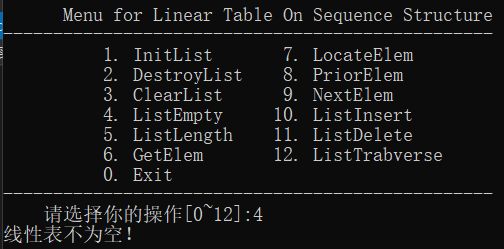


图1-7 线性表判定不为空

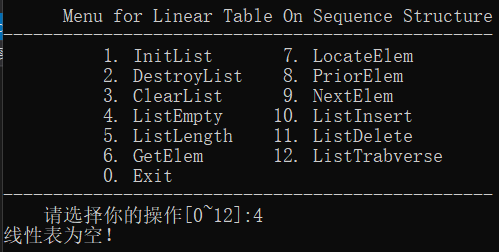


图1-8 线性表为空

1. ListLength：返回当前表长；
2. 使用（4）中第一步建立的表，返回表长为4；（如图1-9所示）
3. 线性表不存在，则返回操作失败；

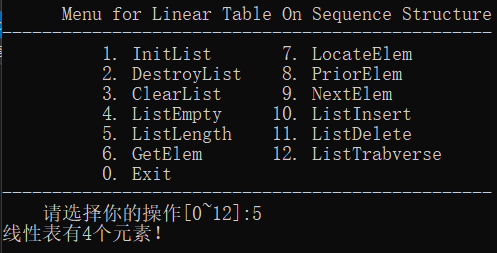


图1-9 返回线性表长

1. GetElem：获取元素位置。输入一个位置数，如果该位置数不小于1且在表长范围内，则用e返回此位置数据元素的值；否则返回ERROR；
2. 使用（4）中第一步建立的表，遍历表；（如图1-10所示）
3. 输入2，获取到第二个元素值，返回为24；（如图1-11所示）
4. 输入7，返回获取失败；（如图1-12所示）
5. 线性表不存在，则返回操作失败；

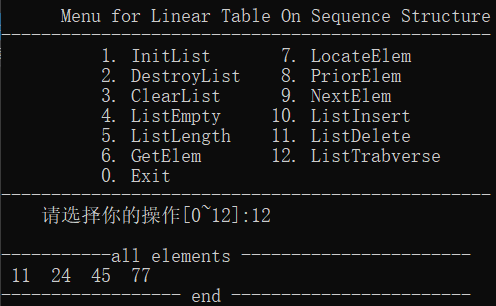


图1-10 遍历表

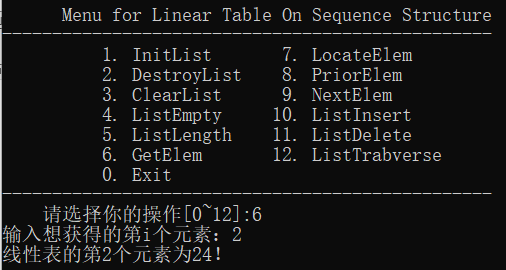


图1-11 返回线性表第2个元素

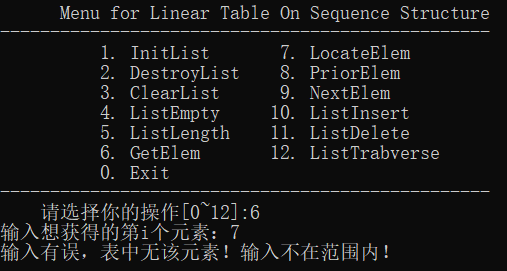


图1-12 输入超出范围

1. LocateElem：定位元素。输入一个元素，遍历全表，如果表内有与之满足Compare()关系的元素则把元素的位置赋值给e，返回e的值；否则返回ERROR；
2. 使用（4）中第一步建立的表，遍历表；（如图1-13所示）
3. 输入想查找的元素值，返回为3；（如图1-14所示）
4. 输入表内没有的元素值，返回定位失败；（如图1-15所示）
5. 线性表不存在，则返回操作失败；

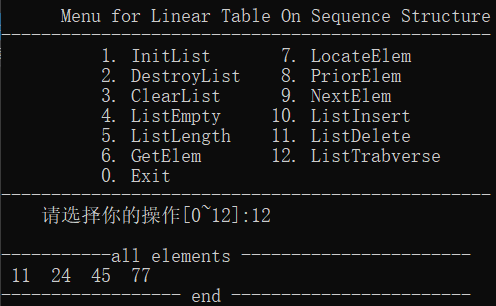


图1-13 遍历表

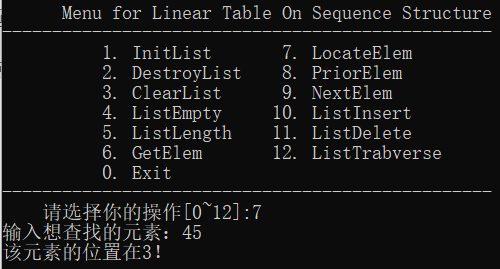


图1-14 查找位置成功

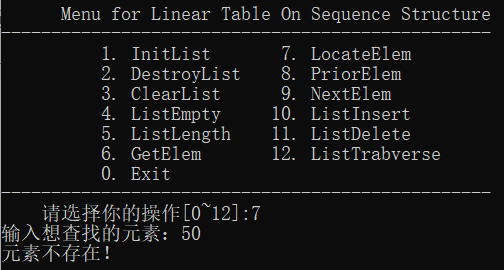


图1-15 查找位置失败

1. PriorElem：获得前驱。输入一个元素，遍历全表，如果找到该元素且位置不在表头，将该元素的前驱元素赋值给pre\_e，并返回OK；否则返回ERROR；
2. 使用（4）中第一步建立的表，遍历表；
3. 输入表头值，返回失败；（如图1-16所示）
4. 输入表内存在的元素值，返回前驱；（如图1-17所示）
5. 输入表内没有的元素值，返回失败；（如图1-18所示）
6. 线性表不存在，则返回操作失败；

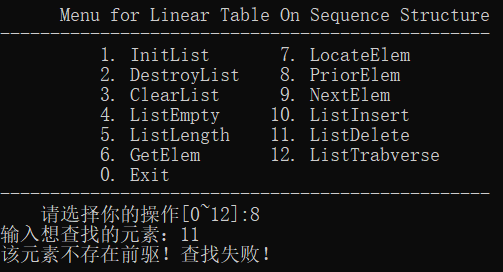


图1-16 表头无前驱

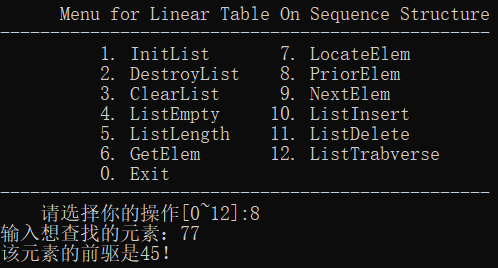


图1-17 返回前驱成功

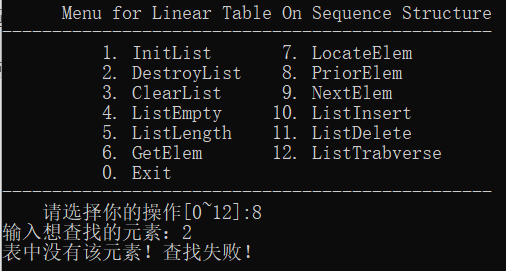


图1-18 返回前驱失败

1. NextElem：获得后继。输入一个元素，遍历全表，如果找到该元素且位置不在表尾，将该元素的后继元素赋值给next\_e，并返回OK；否则返回ERROR；

1) 使用（4）中第一步建立的表，遍历表；

2) 输入表尾值，返回失败；（如图1-19所示）

3) 输入表内存在的元素值，返回后继；（如图1-20所示）

4) 输入表内没有的元素值，返回失败；（如图1-21所示）

5) 线性表不存在，则返回操作失败；

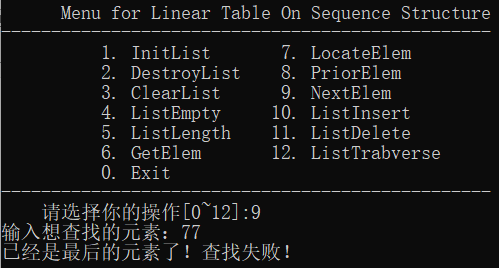


图1-19 最后的元素

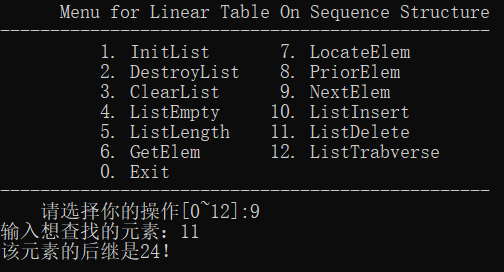


图1-20 返回后继成功

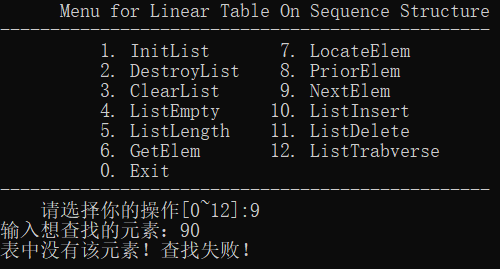


图1-21 返回后继失败

1. ListInsert：插入元素。输入插入位置、插入元素的值，如果表内空间不足则开辟新空间；如果该位置数不小于1且在表长范围内，则将该位置后的数逐个后移一位次，在空出的位置插入元素，表长加1并返回OK；否则返回ERROR；

1) 使用（4）中第一步建立的表，遍历表；

2) 在第2位插入元素，并遍历表进行检查；（如图1-22，1-23所示）

3) 在表长外插入元素，返回插入失败；（如图1-24所示）

4) 线性表不存在，则返回操作失败；

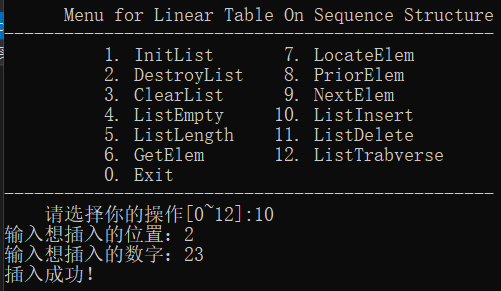


图1-22 插入元素

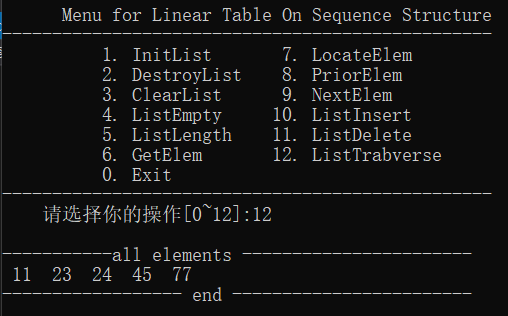


图1-23 插入后遍历检查

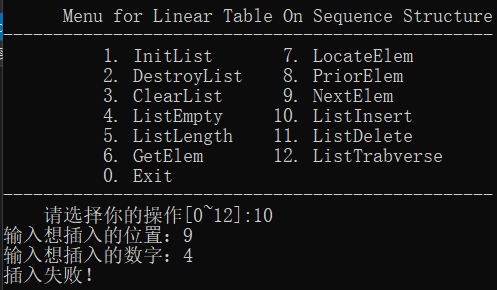


图1-24 插入元素失败

1. ListDelete：删除元素。输入删除位置，如果该位置数不小于1且在表长范围内，则删除该元素，并将该位置后的数逐个前移一位次；表长减1并将删除元素值赋给e，返回e的值；否则返回ERROR；
2. 使用（10）中第二步建立的表，遍历表；（如图1-25所示）
3. 删除位置3的元素，并遍历表；（如图1-26，1-27所示）
4. 删除位置9的元素，返回删除失败；（如图1-28所示）
5. 线性表不存在，则返回操作失误；

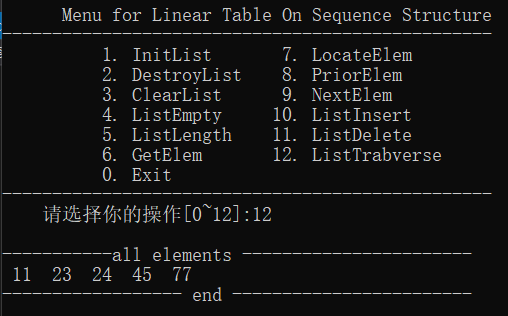


图1-25 遍历表

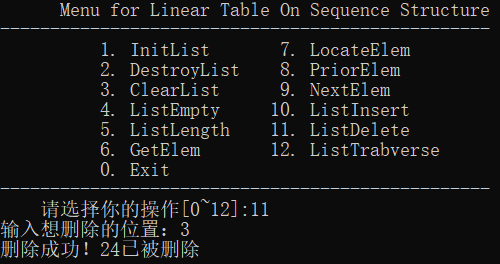


图1-26 删除元素成功

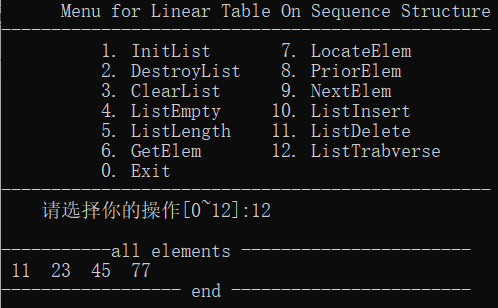


图1-27 删除后遍历表

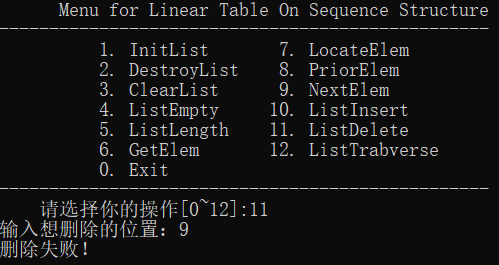


图1-28 删除元素失败

1. ListTrabverse：遍历全表。使用For循环将表内元素顺序输出；
2. 遍历（11）第二步中的表；（如图1-29所示）
3. 线性表不存在，则返回操作失误；（如图1-30所示）

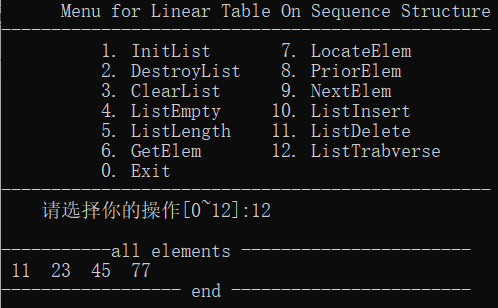


图1-29 遍历表

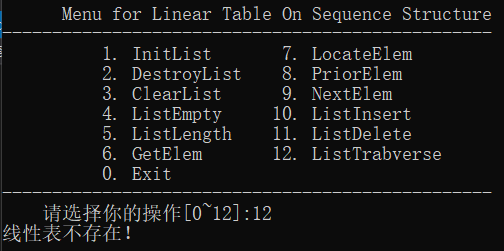


图1-30 线性表不存在

## 实验小结

本次实验为数据结构的第一次实验。本次实验的总体难度并不是很大，但是距离之前学习C语言间隔比较久，有些语法需要细心规范，下面是实验中遇到的问题：

1. 很多C语言的语法记得不太清晰，加上编译器的更新导致一些函数和引用有了改变，所以一开始在修改语法错误上花了很多时间；
2. 对异常情况的处理考虑的不够仔细和全面，一开始没有想好如何判断表是否存在，后来设置了一个全局变量isNull，在每个case内先进行表是否为空的判断方便了很多；
3. 进行异常处理的时候，刚开始并没有将错误反映给用户，比如说表空、删除超出范围、首元素得前驱等等，后面对输出的语言进行了调整，使得反馈更加清晰明了；

这次实验既对上学期的C语言知识进行了回顾，也巩固了这学期新学习的数据结构知识。在亲手实践的过程中也填补了理论知识上的一些盲区，收获了很多。

# 2 基于链式存储结构的线性表实现

## 2.1 问题描述

通过实验达到：

1. 加深对线性表的概念、基本运算的理解；
2. 熟练掌握线性表的逻辑结构与物理结构的关系；
3. 物理结构采用单链表,熟练掌握线性表的基本运算的实现。

**2.1.1 具体问题**

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

1. 初始化表：函数名称是InitList(L)；初始条件是线性表L不存在；操作结果是构造一个空的线性表。
2. 销毁表：函数名称是DestroyList(L)；初始条件是线性表L已存在；操作结果是销毁线性表L。
3. 清空表：函数名称是ClearList(L)；初始条件是线性表L已存在；操作结果是将L重置为空表。
4. 判定空表：函数名称是ListEmpty(L)；初始条件是线性表L已存在；操作结果是若L为空表则返回TRUE,否则返回FALSE。
5. 求表长：函数名称是ListLength(L)；初始条件是线性表已存在；操作结果是返回L中数据元素的个数。
6. 获得元素：函数名称是GetElem(L,i,e)；初始条件是线性表已存在，1≤i≤ListLength(L)；操作结果是用e返回L中第i个数据元素的值。
7. 查找元素：函数名称是LocateElem(L,e,compare())；初始条件是线性表已存在；操作结果是返回L中第1个与e满足关系compare（）关系的数据元素的位序，若这样的数据元素不存在，则返回值为0。
8. 获得前驱：函数名称是PriorElem(L,cur\_e,pre\_e)；初始条件是线性表L已存在；操作结果是若cur\_e是L的数据元素，且不是第一个，则用pre\_e返回它的前驱，否则操作失败，pre\_e无定义。
9. 获得后继：函数名称是NextElem(L,cur\_e,next\_e)；初始条件是线性表L已存在；操作结果是若cur\_e是L的数据元素，且不是最后一个，则用next\_e返回它的后继，否则操作失败，next\_e无定义。
10. 插入元素：函数名称是ListInsert(L,i,e)；初始条件是线性表L已存在，1≤i≤ListLength(L)+1；操作结果是在L的第i个位置之前插入新的数据元素e。
11. 删除元素：函数名称是ListDelete(L,i,e)；初始条件是线性表L已存在且非空，1≤i≤ListLength(L)；操作结果：删除L的第i个数据元素，用e返回其值。
12. 遍历表：函数名称是ListTraverse(L,visit())，初始条件是线性表L已存在；操作结果是依次对L的每个数据元素调用函数visit()。

## 2.2 系统设计

**2.2.1 总体设计**

（1）使用一个大的while语句实现整体功能；

（2）使用switch语句实现菜单栏的操作选择；

（3）在每个case语句先进行对链表是否存在的判断；

**2.2.2 算法设计**

（1）InitList(\*L);

设计思路：开辟一个表头结点的空间；使data为零，下一个指针指向NULL；修改全局变量isNull为false（表存在）；

（2）DestroyList(\*L);

设计思路：遍历表并释放每一个节点的空间；最后把表头节点L设置为NULL；修改全局变量isNull为true（表不存在）；

（3）ClearList(\*L);

设计思路：遍历表并释放每一个节点的空间；最后把表头节点L的下一个节点设置为NULL；

（4）ListEmpty(L);

设计思路：判断表头节点后是否为Null，是则返回true，否则返回false；

（5）ListLength(L);

设计思路：创建一个计数变量i，每遍历表的一个节点i+1，最后返回i的值；

（6）­GetElem(L,i,\*e)；

设计思路：如果链表为空，则显示获取失败；判断输入i是否在已有数据的位置范围内，如果超出范围则返回ERROR；遍历链表直到第i个节点，将当前节点的data值赋值给e并返回OK；

（7）LocateElem(L,e,Compare());

设计思路：遍历链表，返回满足Compare函数条件的元素位置；如果遍历完还未找到则返回ERROR；

（8）PriorElem(L,cur\_e,\*pre\_e);

设计思路：遍历链表，如果当前节点的next结点数据与所寻找的元素相等，则将当前节点的data值赋给pre\_e；如果遍历完还未找到则返回ERROR；

（9）NextElem(L,cur\_e,\*next\_e);

设计思路：遍历链表，如果当前节点的数据与寻找的元素相等，且next节点不为NULL，则将next节点的data值赋给next\_e；如果遍历完还未找到则返回ERROR；

（10）ListInsert(\*L,i,e);

设计思路：创建一个新节点q，把e的值赋给q->data，从表头结点开始遍历链表i个节点，如果i值超出范围则返回ERROR；否则执行插入操作；

（11）ListDelete(\*L,i,&e);

设计思路：创建一个节点q用于存储删除节点的值，从表头结点开始遍历链表i个节点，如果i值超出范围则返回ERROR；否则执行删除操作；

（12）ListTraverse(L);

设计思路：若链表长度为零，则反馈后退出；利用while循环语句遍历整个链表并输出每个节点的data值。

## 2.3 系统实现

**2.3.1 函数实现**

（1）InitList：为表头结点L分配存储空间，若存储分配成功，给L的data值赋值为0，设L的next节点为NULL，返回OK；否则返回ERROR；

（2）DestroyList：初始化两个节点指针p和q，p初始指向表头结点L，q为p的next节点，用p和q遍历表并释放每一个节点的空间；最后把表头节点L设置为NULL；修改全局变量isNull为true（表不存在）；

（3）ClearList：初始化两个节点指针p和q，p初始指向表头结点L，q为p的next节点，用p和q遍历表并释放每一个节点的空间；最后把表头节点L的next节点设置为NULL；

（4）ListEmpty：判断表头节点L的next节点是否为Null，是则返回true，否则返回false；

（5）ListLength：创建一个计数变量i和指向L->next的节点p，p每遍历表的一个节点则i++，最后返回i的值；

（6）GetElem：如果链表为空，则显示获取失败；判断输入i是否在已有数据的位置范围内，如果超出范围则返回ERROR；创建指向L->next的节点p，用p遍历链表直到第i个节点，将当前节点的data值赋值给e并返回OK；

（7）LocateElem：创建指向L->next的节点p，用p遍历整个链表，返回满足Compare函数条件的元素位置；如果遍历完还未找到则返回ERROR；

（8）PriorElem： 创建指向L->next的节点p，用p遍历整个链表，如果p->next结点数据与所寻找的元素相等，则将当前节点的data值赋给pre\_e；否则返回ERROR；

（9）NextElem： 创建指向L->next的节点p，用p遍历整个链表，如果p的结点数据与所寻找的元素相等且p->next不为NULL，则将p->next节点的data值赋给next\_e；否则返回ERROR；

（10）ListInsert： 创建一个新节点q，把e的值赋给q->data，创建指向L的节点p，用p遍历链表i个节点，如果i值超出范围则返回ERROR；否则执行插入操作；

（11）ListDelete：创建一个节点q用于存储删除节点的值，创建指向L的节点p，用p遍历链表i个节点，如果i值超出范围则返回ERROR；否则执行删除操作；

（12）ListTrabverse：若链表长度为零，则反馈后退出；利用while循环语句, 创建指向L->next的节点p，用p遍历整个链表并输出每个节点的data值。

**2.3.2 源代码**

（源代码详细见附录B）

## 2.4 系统测试

编程环境：Win10

编译器：Visual Studio 2019

主模块运行如图2-1所示：

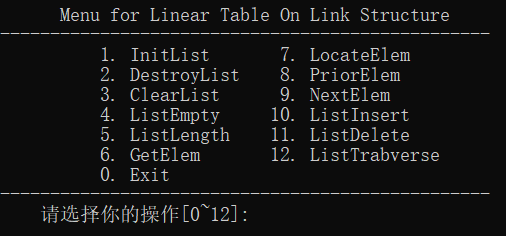


图2-1 主模块图

功能操作：

1. InitList：开辟一个表头结点的空间；使data为零，下一个指针指向NULL；修改全局变量isNull为false（表存在）；
2. 链表不存在，则创建新表成功；（如图2-2所示）
3. 链表不存在的情况下选择任一选项都会返回“链表不存在”；（如图2-3所示）

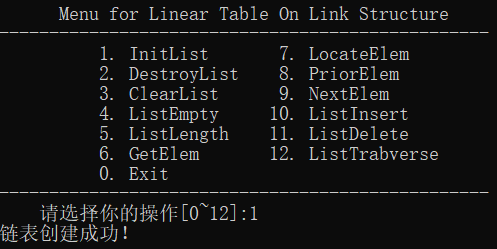


图2-2 链表创建

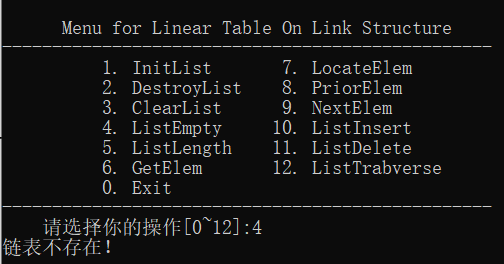


图2-3 链表不存在反馈

1. DestroyList：销毁链表；
2. 链表已存在，则销毁该表；（如图2-4所示）
3. 链表不存在，则返回操作失败；

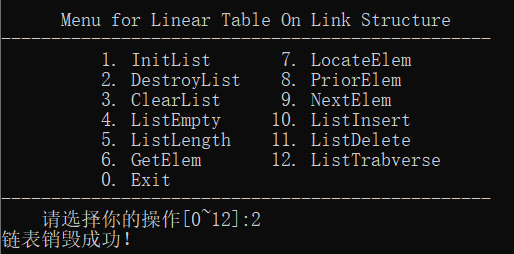


图2-4 链表销毁成功

1. ClearList：释放链表内所有节点空间；如果成功就返回OK；
2. 链表已存在，则清空该表；（如图2-5所示）
3. 链表不存在，则返回操作失败；

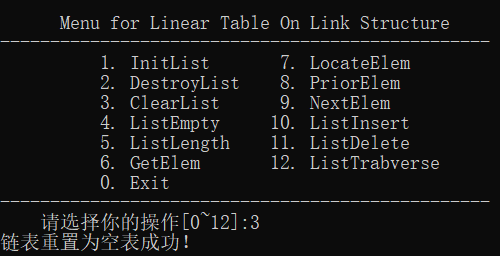


图2-5 链表清空成功

1. ListEmpty：判断链表是否为空；为验证该功能，需要调用后面的函数；
2. 新建一个表，插入元素，遍历；（如图2-6所示）
3. 此时表不为空，返回表不为空；（如图2-7所示）
4. 使用ClearList清空表后，返回表为空；（如图2-8所示）
5. 链表不存在，则返回操作失败；

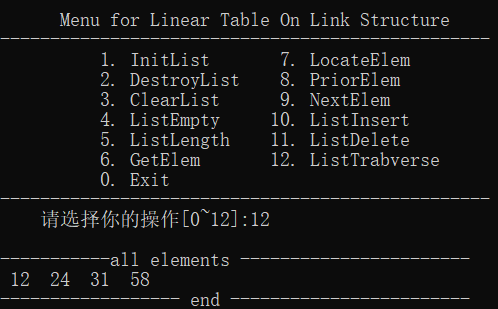


图2-6 链表内含有元素

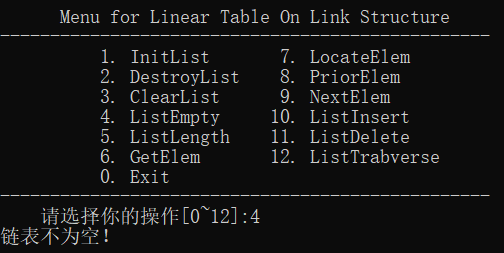


图2-7 链表判定不为空

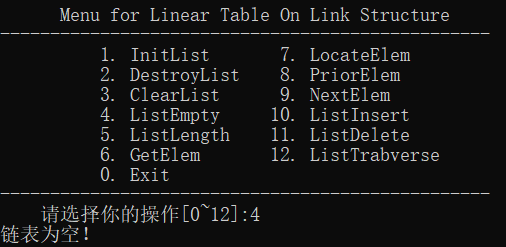


图2-8 链表为空

1. ListLength：返回当前链表长度；
2. 使用（4）中第一步建立的表，返回表长为4；（如图2-9所示）
3. 链表不存在，则返回操作失败；

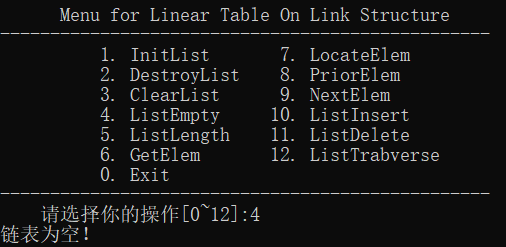


图2-9 返回链表长

1. GetElem：获取元素位置。输入一个位置数，如果该位置数不小于1且在表长范围内，则用e返回此位置数据元素的值；否则返回ERROR；
2. 使用（4）中第一步建立的表，遍历表；（如图2-10所示）
3. 输入3，获取到第三个元素值，返回为31；（如图2-11所示）
4. 输入7，返回获取失败；（如图2-12所示）
5. 链表不存在，则返回操作失败；

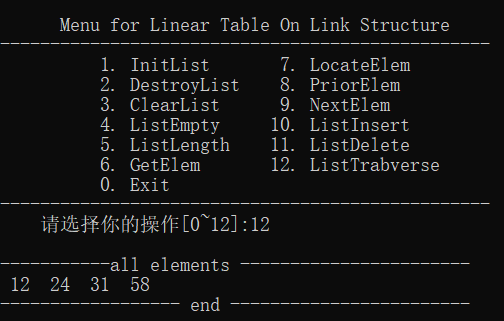


图2-10 遍历表

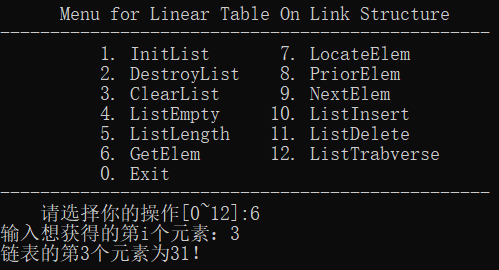


图2-11 返回链表第3个元素

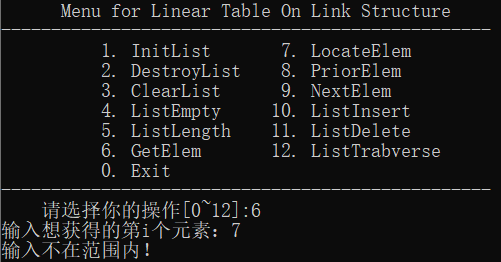


图2-12 输入超出范围

1. LocateElem：定位元素。输入一个元素，遍历链表表，如果表内有与之满足Compare()关系的元素则把元素的位置赋值给e，返回e的值；否则返回ERROR；
2. 使用（4）中第一步建立的表，遍历表；（如图2-13所示）
3. 输入想查找的元素值，返回为2；（如图2-14所示）
4. 输入表内没有的元素值，返回定位失败；（如图2-15所示）
5. 链表不存在，则返回操作失败；

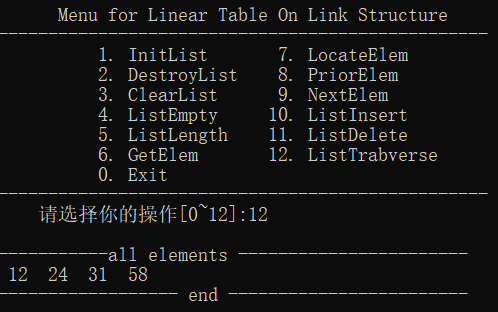


图2-13 遍历表

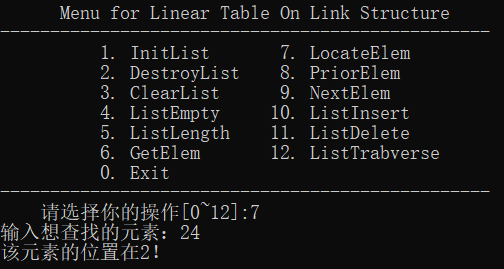


图2-14 查找位置成功

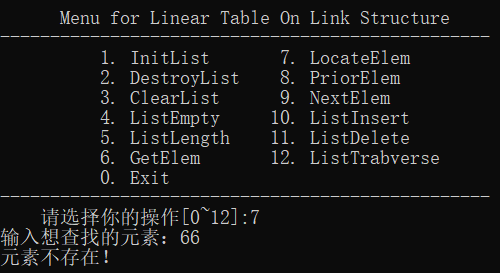


图2-15 查找位置失败

1. PriorElem：获得前驱。输入一个元素，遍历全表，如果找到该元素且位置不在表头，将该元素的前驱元素赋值给pre\_e，并返回OK；否则返回ERROR；
2. 使用（4）中第一步建立的表，遍历表；
3. 输入表头值，返回失败；（如图2-16所示）
4. 输入表内存在的元素值，返回前驱；（如图2-17所示）
5. 输入表内没有的元素值，返回失败；（如图2-18所示）
6. 链表不存在，则返回操作失败；

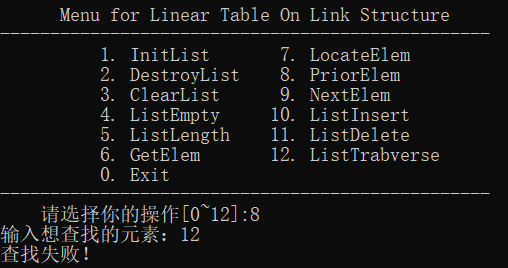


图2-16 表头无前驱

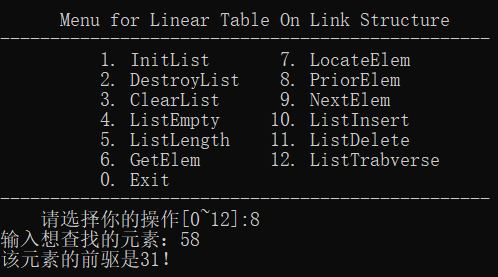


图2-17 返回前驱成功

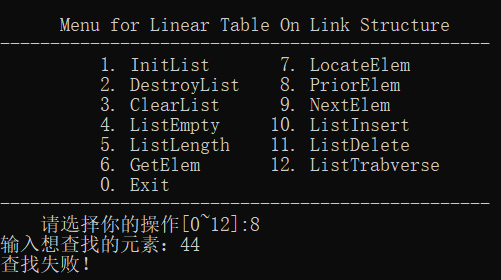


图2-18 返回前驱失败

1. NextElem：获得后继。输入一个元素，遍历全表，如果找到该元素且位置不在表尾，将该元素的后继元素赋值给next\_e，并返回OK；否则返回ERROR；

1) 使用（4）中第一步建立的表，遍历表；

2) 输入表尾值，返回失败；（如图2-19所示）

3) 输入表内存在的元素值，返回后继；（如图2-20所示）

4) 输入表内没有的元素值，返回失败；（如图2-21所示）

5) 链表不存在，则返回操作失败；

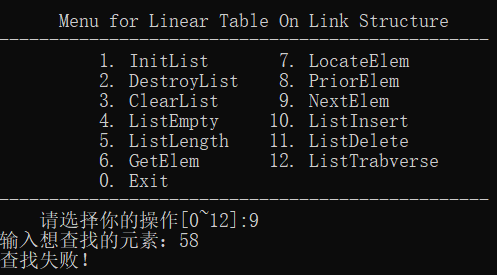


图2-19 最后的元素

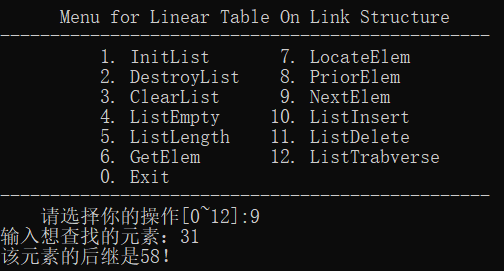


图2-20 返回后继成功

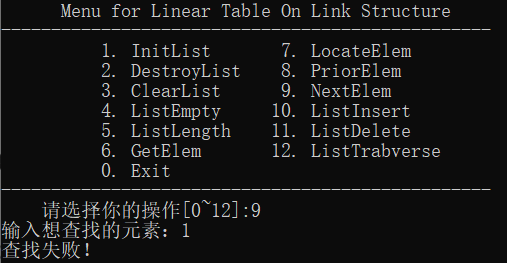


图2-21 返回后继失败

1. ListInsert：插入元素。输入位置、元素的值，如果该位置数不小于1且在表长范围内，则插入新节点并返回OK；否则返回ERROR；

1) 使用（4）中第一步建立的表，遍历表；

2) 在第2位插入元素，并遍历表进行检查；（如图2-22，2-23所示）

3) 在表长外插入元素，返回插入失败；（如图2-24所示）

4) 链表不存在，则返回操作失败；

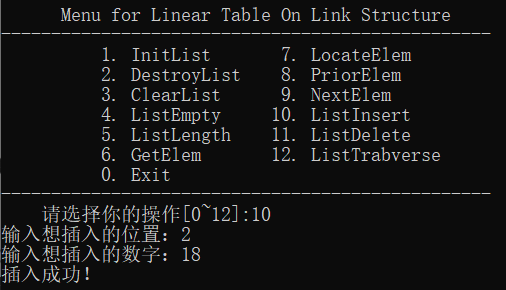


图2-22 插入元素

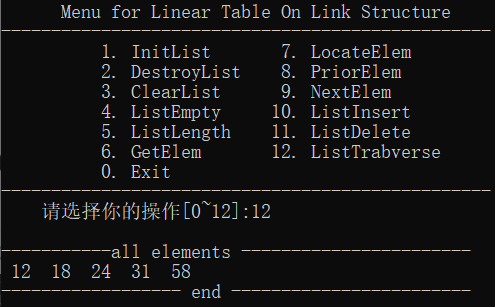


图2-23 插入后遍历检查

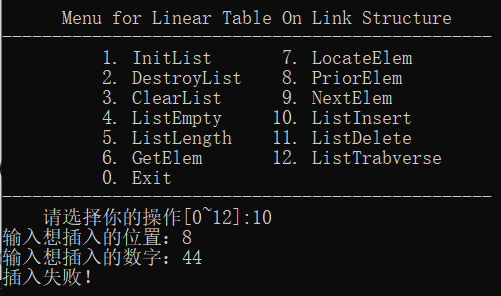


图2-24 插入元素失败

1. ListDelete：删除元素。输入删除位置，如果该位置数不小于1且在表长范围内，则删除该元素，并将删除元素值赋给e，返回e的值；否则返回ERROR；
2. 使用（10）中第二步建立的表，遍历表；（如图2-25所示）
3. 删除位置3的元素，并遍历表；（如图2-26，2-27所示）
4. 删除位置9的元素，返回删除失败；（如图2-28所示）
5. 链表不存在，则返回操作失误；

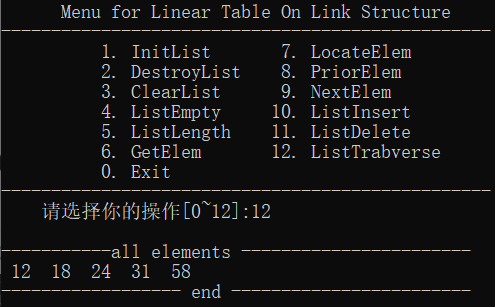


图2-25 遍历表

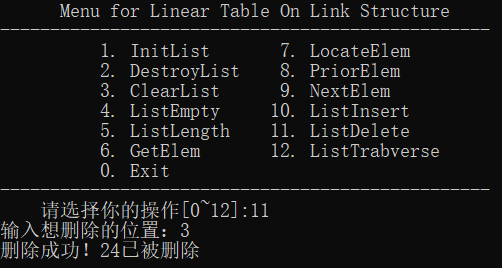


图2-26 删除元素成功

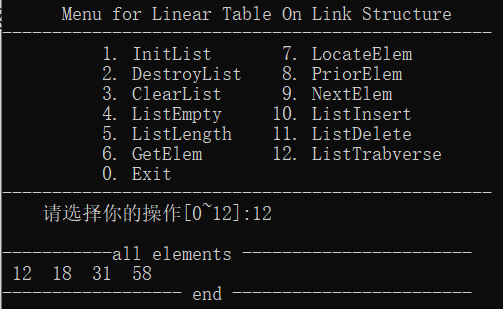


图2-27 删除后遍历表

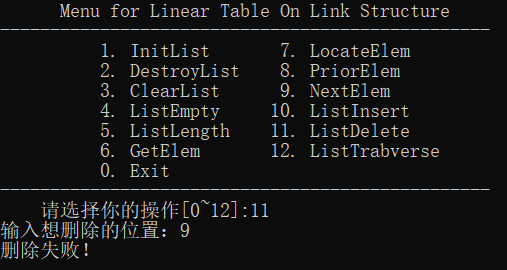


图2-28 删除元素失败

1. ListTrabverse：遍历全表。将表内结点的data值按顺序输出；
2. 遍历（11）第二步中的表；（如图2-29所示）
3. 链表不存在，则返回操作失误；

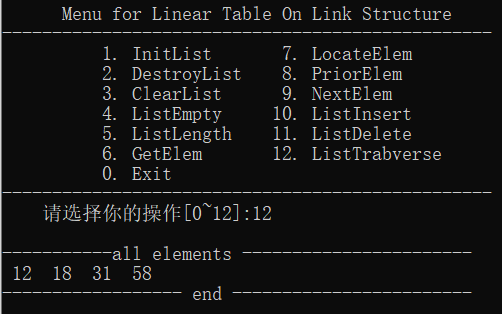


图2-29 遍历表

## 2.5 实验小结

本次实验为数据结构的第二次实验。通过第一次实验的确定了大体结构，再加上单链表在上学期有做过类似的实验，所以完成得不算太困难。但是在编程过程中仍然碰到了一些问题：

1. 在debug过程中遇到多次系统报错引用空指针，所以以后在使用指针的时候应当注意其初始化的值以及各种异常情况，避免指针指向不定造成严重后果；
2. 在进行插入、删除节点、求前驱后继的函数设计时，可以采用多种方式，尽量选择最简洁优雅的实现，方便之后的观看和完善。

在本次实验中我对链表的逻辑结构和存储结构有了更深的了解，在日后对指针的使用也会更加谨慎，受益良多。

# 3 基于二叉链表的二叉树实现

## 3.1 问题描述

通过实验达到：

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

**3.1.1 具体问题**

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

1. 创建二叉树：函数名称是CreateBiTree(T,definition)；初始条件是definition 给出二叉树T的定义，如带空子树的二叉树前序遍历序列、或前序+中序、或后序+中序；操作结果是按definition构造二叉树T。
2. 注：①要求T中各结点关键字具有唯一性。后面各操作的实现，也都要满足一棵二叉树中关键字的唯一性，不再赘述；②CreateBiTree中根据definition生成T，不应在CreateBiTree中输入二叉树的定义。
3. 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。
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. 查找结点：函数名称是LocateNode(T,e)；初始条件是二叉树T已存在，e是和T中结点关键字类型相同的给定值；操作结果是返回查找到的结点指针，如无关键字为e的结点，返回NULL。
8. 结点赋值：函数名称是Assign(T,e,value)；初始条件是二叉树T已存在，e是和T中结点关键字类型相同的给定值；操作结果是关键字为e的结点赋值为value。
9. 获得兄弟结点：函数名称是GetSibling(T,e)；初始条件是二叉树T存在，e是和T中结点关键字类型相同的给定值；操作结果是返回关键字为e的结点的（左或右）兄弟结点指针。若关键字为e的结点无兄弟，则返回NULL。
10. 插入结点：函数名称是InsertNode(T,e,LR,c)；初始条件是二叉树T存在，e是和T中结点关键字类型相同的给定值，LR为0或1，c是待插入结点；操作结果是根据LR为0或者1，插入结点c到T中，作为关键字为e的结点的左或右孩子结点，结点e的原有左子树或右子树则为结点c的右子树。
11. 删除结点：函数名称是DeleteNode(T,e)；初始条件是二叉树T存在，e是和T中结点关键字类型相同的给定值。操作结果是删除T中关键字为e的结点；同时，如果关键字为e的结点度为0，删除即可；如关键字为e的结点度为1，用关键字为e的结点孩子代替被删除的e位置；如关键字为e的结点度为2，用e的左孩子代替被删除的e位置，e的右子树作为e的左子树中最右结点的右子树。
12. 前序遍历：函数名称是PreOrderTraverse(T,Visit())；初始条件是二叉树T存在，Visit是一个函数指针的形参（可使用该函数对结点操作）；操作结果：先序遍历，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。
13. 中序遍历：函数名称是InOrderTraverse(T,Visit))；初始条件是二叉树T存在，Visit是一个函数指针的形参（可使用该函数对结点操作）；操作结果是中序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。
14. 后序遍历：函数名称是PostOrderTraverse(T,Visit))；初始条件是二叉树T存在，Visit是一个函数指针的形参（可使用该函数对结点操作）；操作结果是后序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。
15. 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。
16. 实现二叉树的文件形式保存：函数名称是SaveTree(T,path)和LoadTree(T,path)；初始条件是二叉树存在，操作结果是实现二叉树的文件形式保存。

## 3.2 系统设计

**3.2.1 总体设计**

（1）使用一个大的while语句实现整体功能；

（2）使用switch语句实现菜单栏的操作选择；

（3）在每个case语句先进行对二叉树是否存在的判断；

**3.2.2 算法设计**

（1）CreateBiTree(TNode\* T, char\* definition);

设计思路：按带空节点的前序遍历顺序输入每个key的值（输入包括叶节点的两个空子树），存在字符串中。用一个全局变量改变字符串指针的移动，用递归的思想按照前序创建二叉树的结点。

（2）DestroyBiTree(TNode\* T);

设计思路：使用递归释放每一个结点，最后将全局变量isNull（判定表是否存在）改为TRUE值；

（3）ClearBiTree(TNode& T);

设计思路：使用递归释放每一个结点，最后将根节点指针T的key值设置为“#”；

（4）BiTreeEmpty(TNode T);

设计思路：判断根节点T的key值是否为“#”，是则返回true，否则返回false；

（5）BiTreeDepth(TNode T);

设计思路：创建两个计数变量L和R，递归遍历二叉树的左右子树，返回值为L和R中较大的一个并加一的值；

（6）LocateNode(TNode T, KeyType e);

设计思路：设置全局变量Findnode默认为空结点，递归遍历二叉树，当找到对应结点时令Findnode等于该结点。函数返回值为Findnode；

（7）Assign(TNode\* T, KeyType e, TElemType value);

设计思路：调用LocateNode函数查找对应结点，如果找到结点则给该结点的value赋值，否则返回ERROR；

（8）GetSibling(TNode T, KeyType e);

设计思路：递归遍历二叉树，如果找到一结点的左孩子是查找的结点，且其右孩子存在，则返回其右孩子，另一方向同理；反之返回ERROR；

（9）InsertNode(TNode\* T, KeyType e, int LR, TNode c);

设计思路：新建一个结点，为它初始化赋值；调用LocateNode函数查找插入位置的结点；若找到，则判断LR的值分类情况完成插入；插入成功返回OK，否则返回ERROR；

（10）DeleteNode(TNode\* T, KeyType e);

设计思路：新建一个结点delnode代表被删除结点；先调用LocateNode函数查找需要被删除的结点，如果找到该结点则进行是否为根节点的分类讨论；调用查找父节点的函数GetParent找到delnode的父节点，根据结点度数为0,1,2再进行分类讨论进行删除；删除成功返回OK，否则返回ERROR；

（11）PreOrderTraverse(TNode T);

设计思路：按照根-左子树-右子树的顺序进行递归遍历；

（12）InOrderTraverse(TNode T);

设计思路：用栈实现的非递归中序遍历。先让指针遍历到最左结点，push经过的结点到栈中，在指针指向空的时候pop并将弹出的结点赋给当前指针位置，遍历右子树，直到整棵树遍历完成且栈内无元素，返回OK；

（13）PostOrderTraverse(TNode T);

设计思路：按照左子树-右子树-根的顺序进行递归遍历。

（14）LevelOrderTraverse(TNode T);

设计思路：利用队列完成二叉树的按层遍历。首先将二叉树的根节点push到队列中，判断队列不为NULL，就输出队头的元素，判断节点如果有孩子，就将孩子push到队列中，遍历过的节点出队列，循环以上操作，直到遍历完成，返回OK。

1. SaveTree(TNode T, char\* path);

设计思路：在函数体内调用递归函数ReSaveTree(T)来存储二叉树的每个结点到文件中。

（16）LoadTree(TNode\* T, char\* path)

设计思路：在用文件读取操作将文件内的数据输出到屏幕上。

## 3.3 系统实现

**3.3.1 函数实现**

（1）CreateBiTree：按带空节点的前序遍历顺序输入每个key的值（输入包括叶节点的两个空子树），存储在字符串中。用一个全局变量改变字符串指针的移动，逐个读取字符创建树结点，直到字符串被读取完毕；

（2）DestroyBiTree：使用递归释放每一个结点，将根节点指针T置空，最后将全局变量isNull（判定表是否存在）改为TRUE值；

（3）ClearBiTree：若T不存在，则返回ERROR；使用递归释放每一个结点，将根节点指针T的key值改为“#”；

（4）BiTreeEmpty：判断根节点T的key值是否为“#”，是则返回true，否则返回false；

（5）BiTreeDepth：若T不存在，则返回ERROR；否则创建两个计数变量L和R，递归遍历二叉树的左右子树，返回值为L和R中较大的一个加一的值；

（6）LocateNode：若T不存在，则返回ERROR；设置全局变量Findnode，默认为空结点指针，递归遍历二叉树，当找到对应结点时把该结点的地址赋给Findnode。函数返回值为Findnode；每次使用完LocateNode需要把Findnode再次置空。

（7）Assign：新建一个暂存结点cue，调用LocateNode函数查找需要赋值的对应结点并赋给cue，如果找到结点则给cue的value赋值；否则返回ERROR；

（8）GetSibling：递归遍历二叉树，对当前结点的左右孩子进行判断，若该结点的左孩子存在且其key值与输入的值相等，且其右孩子存在，则返回其右孩子结点指针，另一方向同理；反之返回ERROR；

（9）InsertNode：新建一个结点，开辟一个存储空间并进行初始化；调用LocateNode函数查找插入位置的结点；若找到，则判断LR的值，分类情况完成插入；插入成功返回OK，否则返回ERROR；

（10）DeleteNode：新建一个结点delnode代表被删除结点；先调用LocateNode函数查找需要被删除的结点，如果找到该结点则进行是否为根节点的分类讨论；调用查找父节点的函数GetParent()找到delnode的父节点，根据结点度数为0,1,2再进行分类讨论进行不同的删除操作；删除成功返回OK，否则返回ERROR；

（11）PreOrderTraverse：若T不存在，则返回ERROR；否则按照以下顺序：对当前结点使用Visit()函数访问，递归调用PreOrderTraverse函数访问当前结点的左孩子，再递归调用PreOrderTraverse函数访问当前结点的右孩子，返回OK；

（12） InOrderTraverse：用栈实现的非递归中序遍历。若T不存在，则返回ERROR；先让指针遍历到最左结点，push经过的结点到栈中，在指针指向空的时候pop并将弹出的结点赋给当前指针位置，遍历右子树，直到整棵树遍历完成且栈内无元素，返回OK；

（13）PostOrderTraverse：若T不存在，则返回ERROR；否则按照以下顺序：递归调用PostOrderTraverse函数访问当前结点的左孩子，再递归调用PostOrderTraverse函数访问当前结点的右孩子，对当前结点使用Visit()函数访问，返回OK；

（14）LevelOrderTraverse(TNode T)：若T不存在，则返回ERROR；利用队列完成二叉树的按层遍历。用一个结点指针p完成遍历，p的初始值为T，将p结点push到队列中；使用条件循环判断队列不为NULL，就输出队头的结点赋给p，判断p如果有孩子，就将孩子push到队列中，遍历过的节点出队并调用Visit()函数访问，循环以上操作，直到队列为空，返回OK。

（15）SaveTree(TNode T, char\* path)：若T不存在，则返回ERROR；打开指定的文件，调用递归函数ReSaveTree(T)按前序遍历顺序将二叉树的每个结点的key值和value值存储到文件中。

（16）LoadTree(TNode\* T, char\* path)：在用文件读取操作将文件内的数据输出到屏幕上。

**3.3.2 源代码**

（源代码详细见附录C）

## 3.4 系统测试

编程环境：Win10

编译器：Visual Studio 2019

主模块运行如图3-1所示：

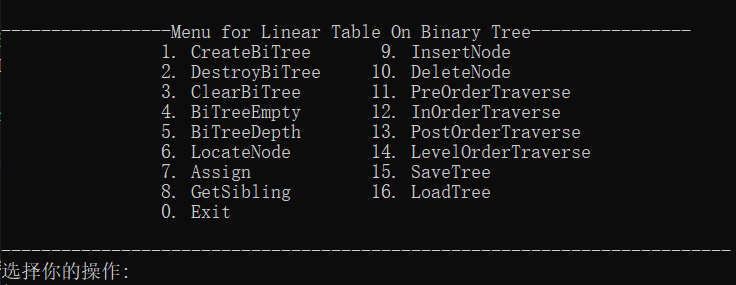


图3-1 主模块图

功能操作：

1. CreateBiTree：按带空结点的前序序列给definition字符串赋一串key值，创建一棵新二叉树；
2. 输入“ABD###C##”；（树的样子和输出结果如图3-2，图3-3所示）
3. 在树不存在的情况下选择任一选项(除了LoadTree)都会返回“二叉树不存在”；（如图3-4所示）

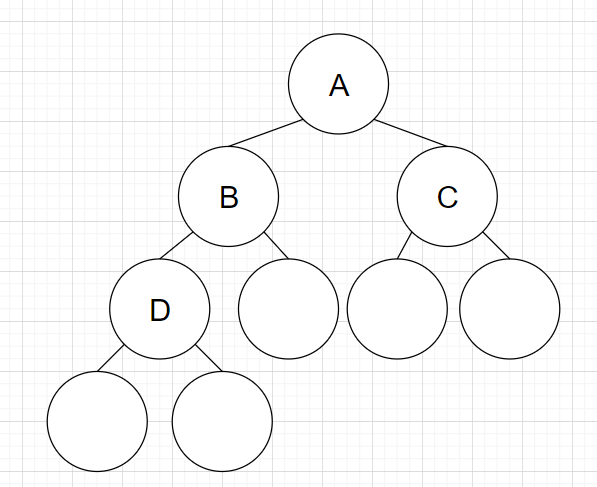


图3-2 创建树图示



图3-3 建立二叉树树成功

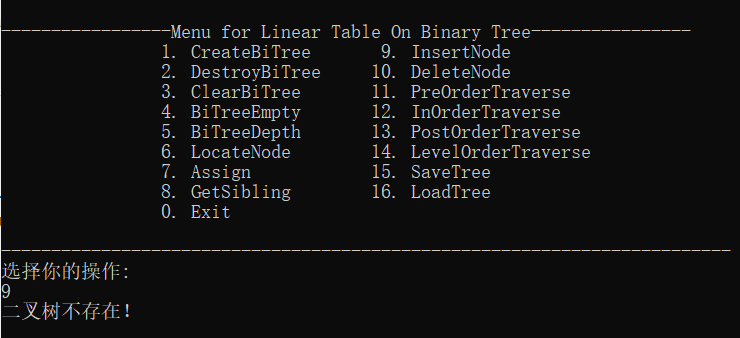


图3-4 二叉树不存在反馈

1. DestroyBiTree：销毁二叉树；成功则返回OK；若二叉树不存在则返回ERROR；
2. 二叉树已存在，则销毁该树；（如图3-5所示）
3. 二叉树不存在，则返回操作失败；（如图3-6所示）

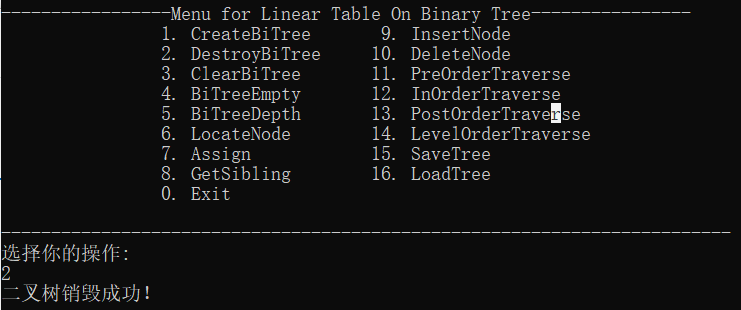


图3-5 二叉树销毁成功

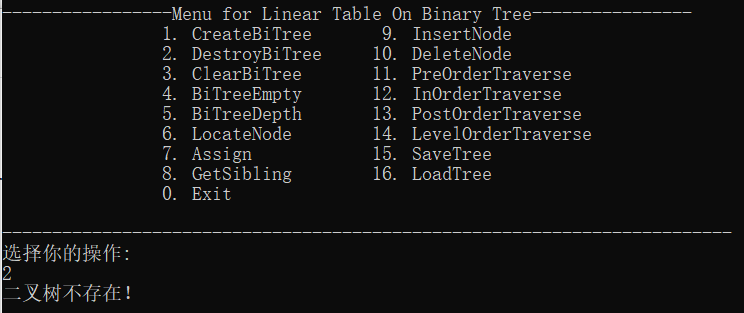


图3-6 二叉树销毁失败

1. ClearBiTree：清空二叉树；
2. 二叉树不为空，则清空成功；（如图3-7所示）
3. 二叉树已经为空，则返回“二叉树为空”；（如图3-8所示）

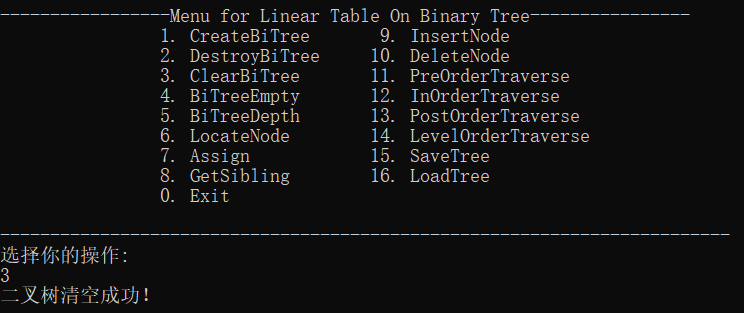


图3-7 二叉树清空成功

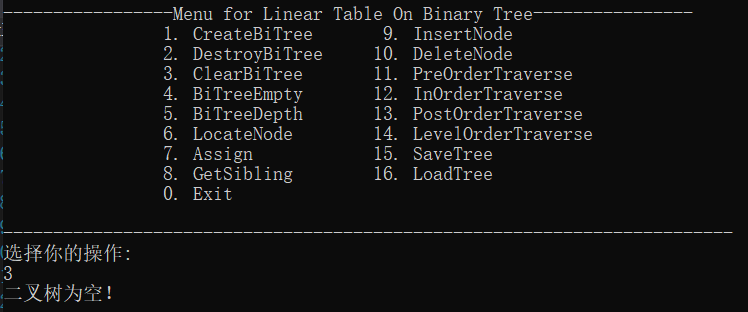


图3-8 二叉树已经为空

1. BiTreeEmpty：判断二叉树是否为空；为验证该功能，需要调用后面的函数；
2. 调用（1）中新建的二叉树，先进行前序遍历；（如图3-9所示）
3. 此时树不为空，返回树不为空；（如图3-10所示）
4. 使用ClearList清空树后，返回树为空；（如图3-11所示）

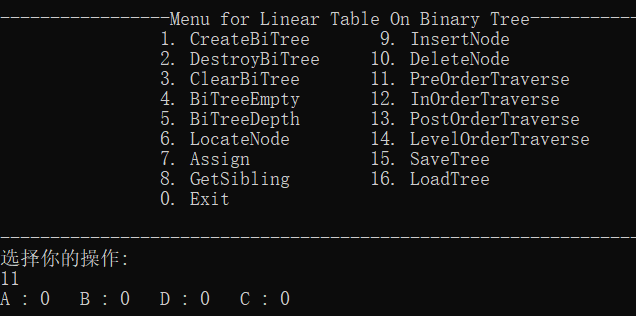


图3-9 前序遍历二叉树

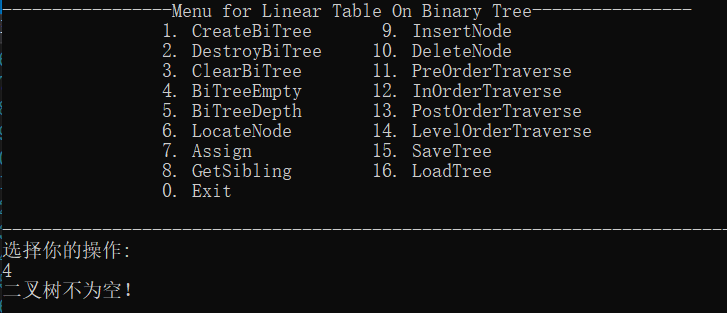


图3-10 二叉树不为空

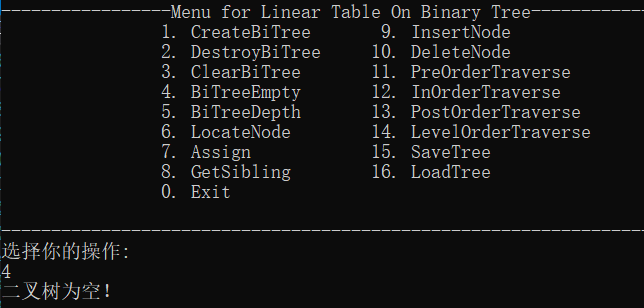


图3-11 二叉树为空

1. BiTreeEmpty：返回当前二叉树深度；
2. 使用（1）中第一步建立的二叉树，返回深度为3；（如图3-12所示）
3. 二叉树不存在，则返回操作失败；

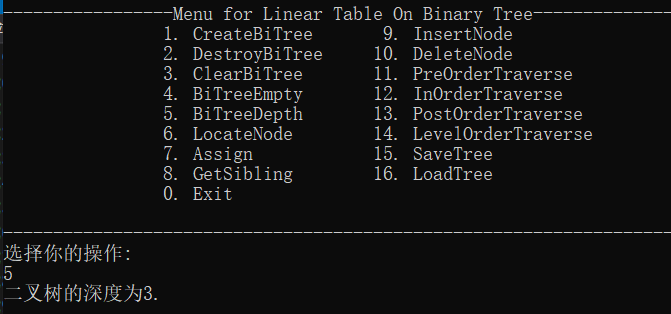


图3-12 返回二叉树深度

1. LocateNode：查找关键字结点。输入一个结点的key值，如果树内有该结点，则返回此结点的全部信息；否则返回ERROR；
2. 使用（1）中第一步建立的树，进行中序遍历；（如图3-13所示）
3. 输入D，查找成功，获取返回值；（如图3-14所示）
4. 输入E，查找失败；（如图3-15所示）
5. 二叉树不存在，则返回操作失败；

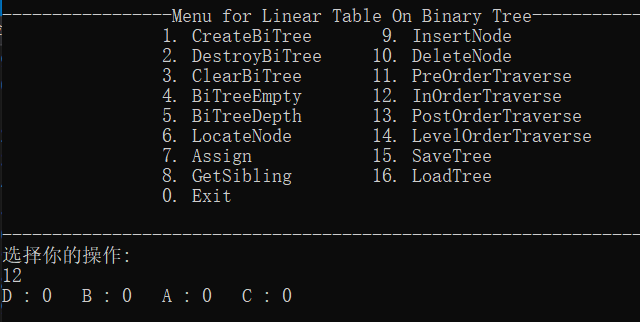


图3-13 中序遍历二叉树

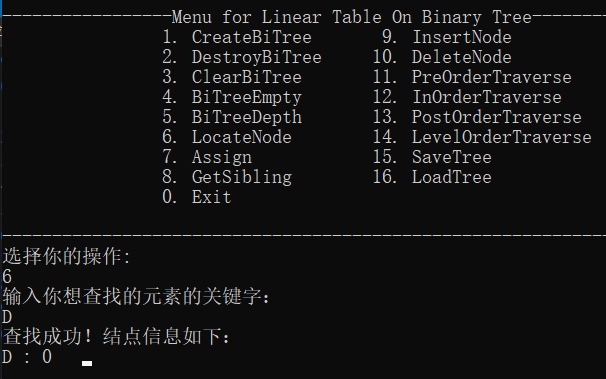


图3-14 查找成功

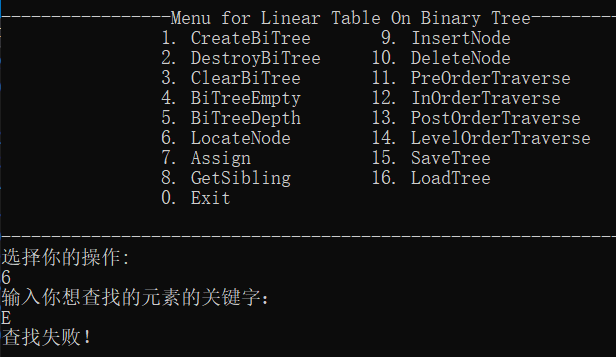


图3-15 查找失败

1. Assign：为结点元素赋值；
2. 使用（1）中第一步建立的树，前序遍历二叉树；（如图3-16所示）
3. 分别为A、B、C、D赋值1、2、3、4；（如图3-17所示）
4. 再次前序遍历二叉树；（如图3-18所示）
5. 输入为E赋值，返回赋值失败；（如图3-19所示）
6. 二叉树不存在，则返回操作失败；

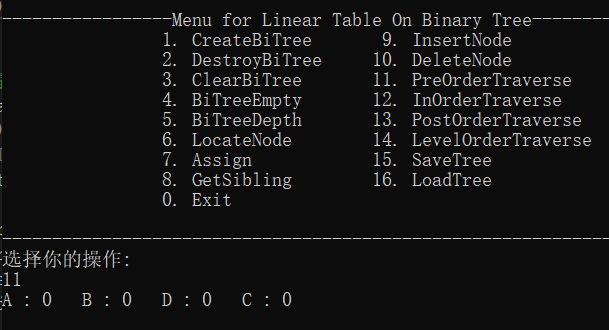


图3-16 前序遍历二叉树

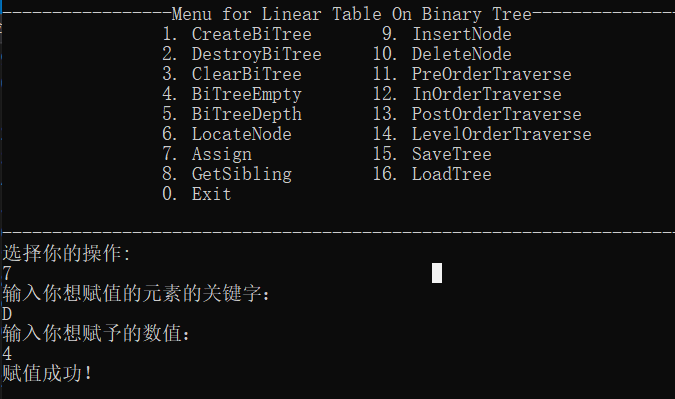


图3-17 赋值成功

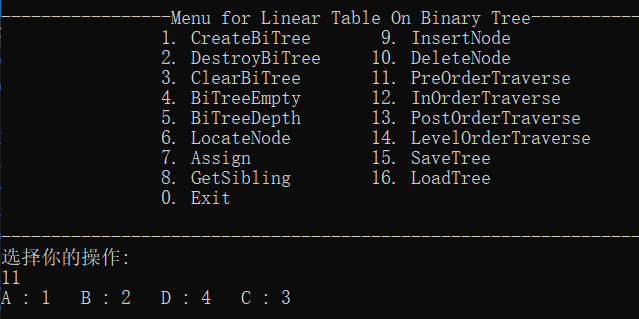


图3-18 再次前序遍历二叉树



图3-19 赋值失败

1. GetSibling：获得兄弟节点；
2. 使用（7）中已被赋值的树；
3. 输入根结点A，返回失败；（如图3-20所示）
4. 输入结点B，返回兄弟结点；（如图3-21所示）
5. 输入结点D，返回失败；（如图3-22所示）
6. 二叉树不存在，则返回操作失败；



图3-20 根结点

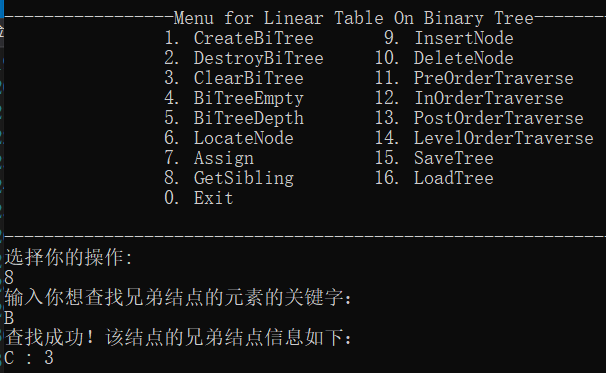


图3-21 返回兄弟节点

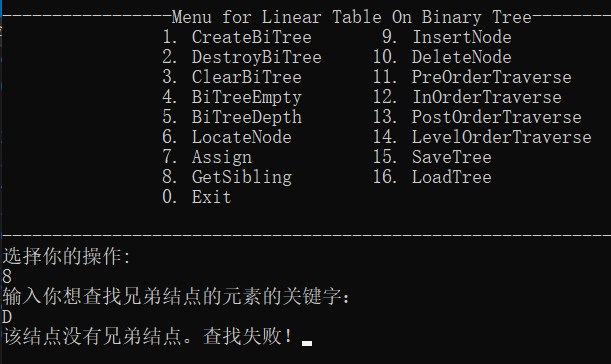


图3-22 查找兄弟结点失败

1. InsertNode：插入节点；

1) 使用（7）中已被赋值的树；

2) 插入新节点E到D的右子树，并赋值5；（如图3-23所示）

3) 插入新节点F到A的右子树，并赋值6；（如图3-24所示）

4) 前序、中序遍历二叉树；（如图3-25，3-26所示）

5) 插入结点为树内已有的值，插入失败；（如图3-27所示）

6) 二叉树不存在，则返回操作失败；



图3-23 插入元素E

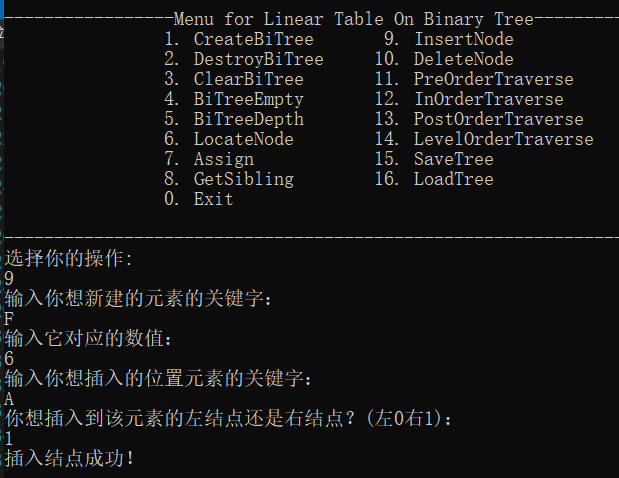


图3-24 插入元素F

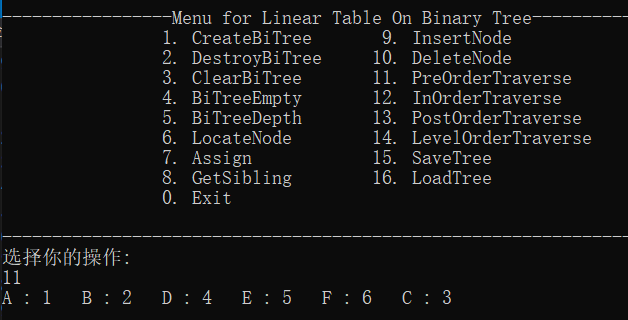


图3-25 前序遍历二叉树

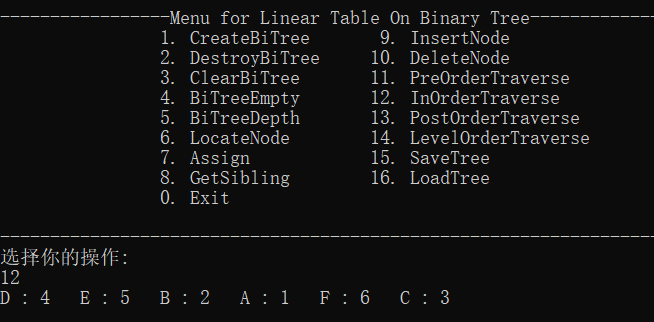


图3-26 中序遍历二叉树

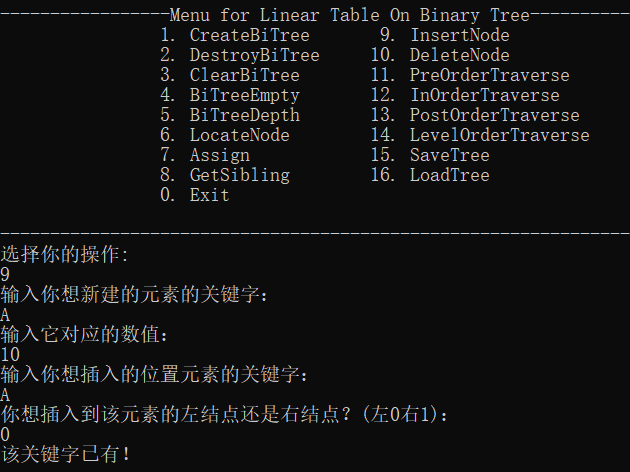


图3-27 插入失败

1. DeleteNode：删除结点；

1) 使用（4）中插入元素后的树；（树的样子如图3-28所示）

2) 删除度数为0的结点E，前序遍历；（如图3-29，3-30所示）

3) 删除度数为1的结点B，中序遍历；（如图3-31所示）

4) 删除度数为2的根结点A，按层遍历；（如图3-32所示）

6) 删除不存在的结点H，删除失败；（如图3-33所示）

5) 二叉树不存在，则返回操作失败；

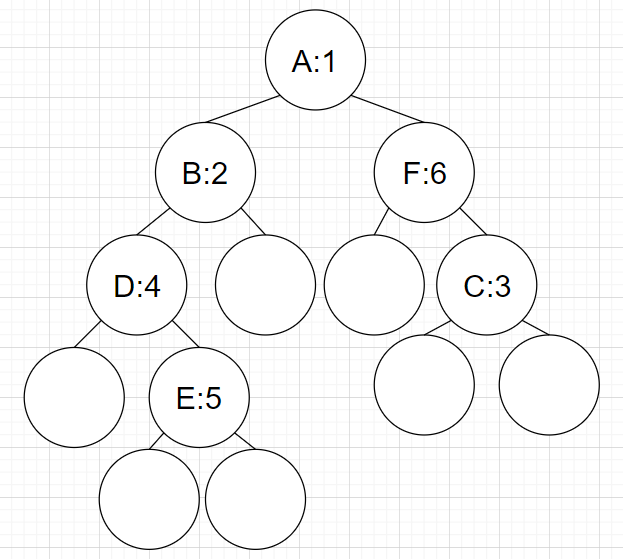


图3-28 树的样子



图3-29 删除结点E

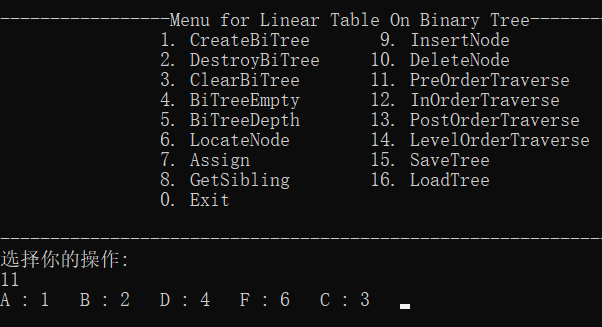


图3-30 前序遍历树

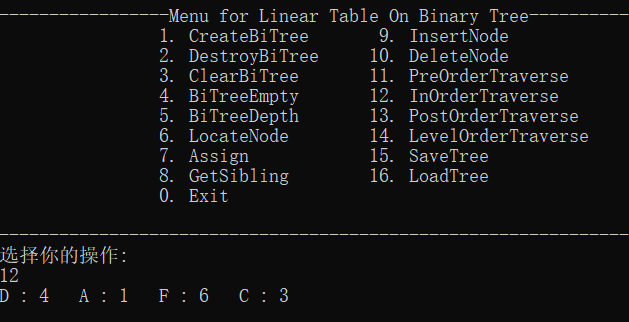


图3-31 删除结点B后中序遍历树

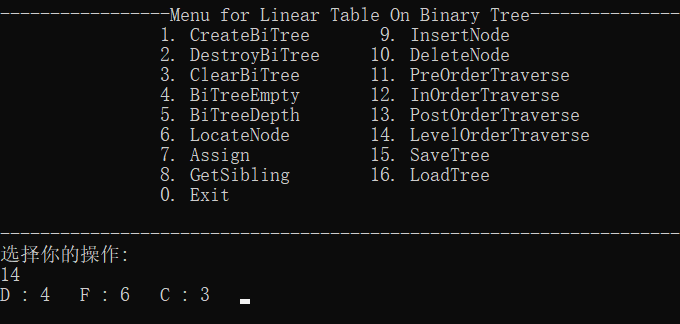


图3-32 删除结点A后按层遍历树



图3-33 删除失败

此时的树如图3-34所示

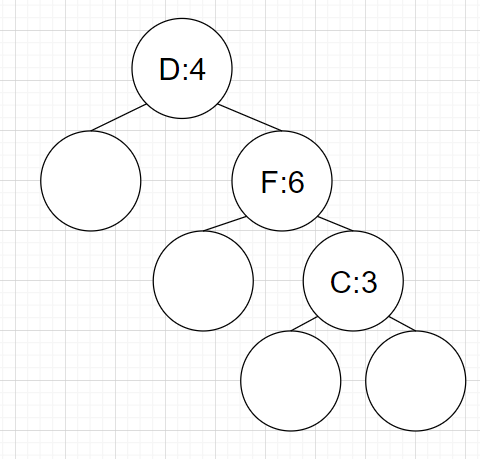


图3-34 树的样子

1. PreOrderTraverse：前序遍历二叉树；
2. 使用（10）中删除完成的树，前序遍历；（如图3-35所示）
3. 二叉树不存在，则返回操作失败；

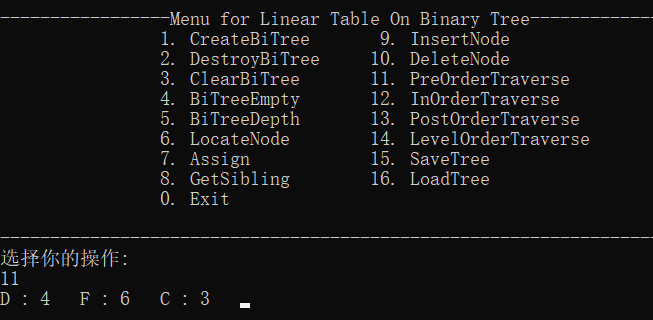


图3-35 前序遍历树

1. InOrderTraverse：中序遍历二叉树；
2. 使用（10）中删除完成的树，中序遍历；（如图3-36所示）
3. 二叉树不存在，则返回操作失误；

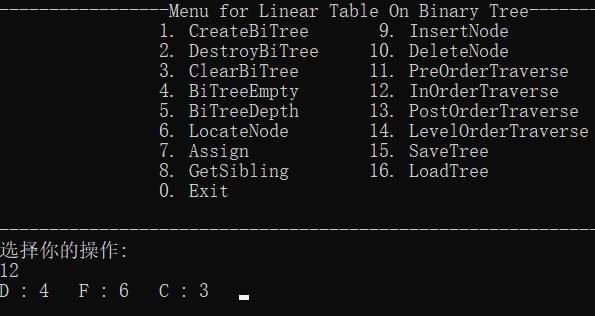


图3-36 中序遍历树

1. PostOrderTraverse：后序遍历二叉树；
2. 使用（10）中删除完成的树，后序遍历；（如图3-37所示）
3. 二叉树不存在，则返回操作失误；

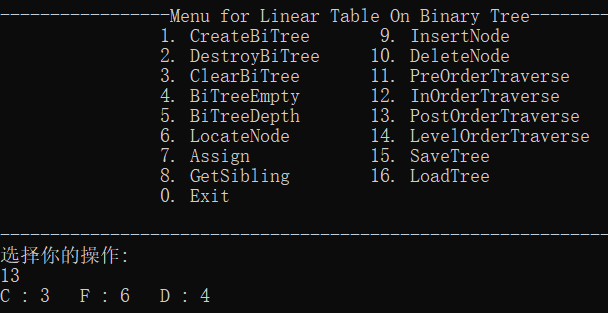


图3-37 后序遍历树

1. LevelOrderTraverse：按层遍历二叉树；
2. 使用（10）中删除完成的树，按层遍历；（如图3-38所示）
3. 二叉树不存在，则返回操作失误；

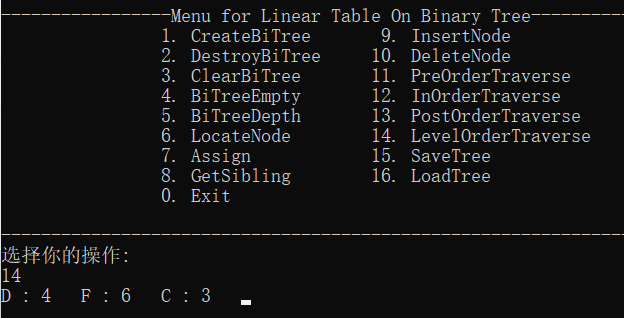


图3-38 按层遍历树

1. SaveTree：保存二叉树；
2. 使用（10）中删除完成的树，进行文件保存；（如图3-39所示）
3. 二叉树不存在，则返回操作失误；

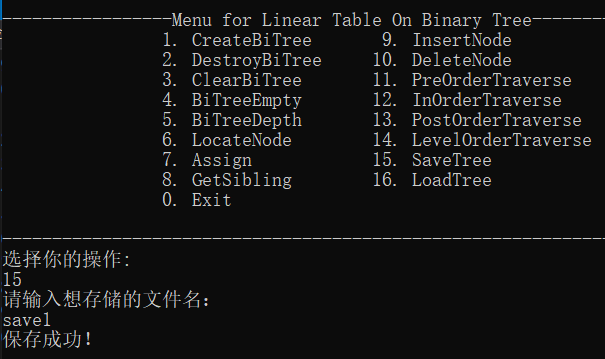


图3-39 保存树

1. LoadTree：读取二叉树；
2. 打开之前保存的文件，将树导入并输出；（如图3-40、3-41所示）
3. 二叉树不存在，则返回操作失误；

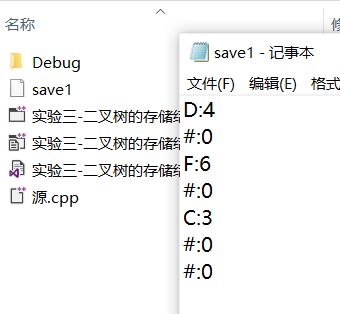


图3-40 文件

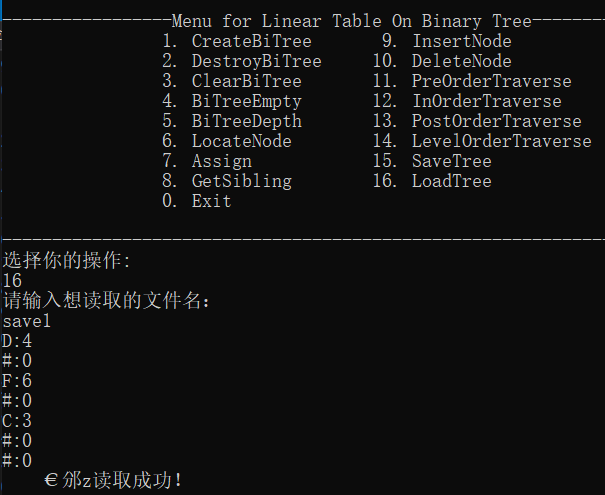


图3-41 读取树

## 3.5 实验小结

本次实验总体难度较大，前后写代码花费了将近八九个小时，改bug的时间又花了很久。而且由于自身疏忽，造成了一次代码遗失，多花了一些时间重新修补，确实是一次比较坎坷的实验经历。下面我大致说一下在本次实验中遇到的一些问题及收获：

1. 在本次实验中我使用了大量的递归思想，在结点的创建、搜索、遍历等各函数内都使用了递归。递归的优点是函数体简洁，可读性强；缺点是时间复杂度较高，而且当牵涉到一些变量变化时容易出错。因此我额外设定了四个全局变量来调控，导致整体实验代码虽然完成了功能，但是可维护性不算很好，也是本次实验的不足点之一，之后需要改善；
2. 在写插入和删除等一些函数的时候需要分类讨论，经常有考虑不周到的地方，比如根结点的插入和删除与其它情况都要分开来讨论。最后通过断点调试测出了问题解决了bug；
3. 在变量的命名规范上有所改善。自己在重新审视代码时发现了许多取名为类似p,q的临时变量，在没有注释的情况下理解起来很费劲，所以尽量都改成了类似Findnode，Brothernode这样容易理解的、更加规范的变量名。同时我也会在之后的实验中努力保持这一点。

通过本次实验我学习到了很多，同时也更加坚定了提升自己的决心，有的时候理论上感觉自己懂了其实是不够的，需要亲手实现出来，把每一个错误点理解并改正，在这个过程中自身对数据结构的理解才会更深、更确切。

# 4 基于邻接表的图实现

## 4.1 问题描述

通过实验达到：

1. 加深对图的概念、基本运算的理解；
2. 熟练掌握图的逻辑结构与物理结构的关系；
3. 以邻接表作为物理结构，熟练掌握图基本运算的实现。

**4.1.1 具体问题**

依据最小完备性和常用性相结合的原则，以函数形式定义了创建图、销毁图、查找顶点、获得顶点值和顶点赋值等12种基本运算。具体运算功能定义和说明如下。具体运算功能定义如下。

1. 创建图：函数名称是CreateCraph(G,V,VR)；初始条件是V是图的顶点集，VR是图的关系集；操作结果是按V和VR的定义构造图G。

注：①要求图G中顶点关键字具有唯一性。后面各操作的实现，也都要满足一个图中关键字的唯一性，不再赘述；② V和VR对应的是图的逻辑定义形式，比如V为顶点序列，VR为关键字对的序列。不能将邻接矩阵等物理结构来代替V和VR。

1. 销毁图：函数名称是DestroyGraph(G)；初始条件图G已存在；操作结果是销毁图G。
2. 查找顶点：函数名称是LocateVex(G,u)；初始条件是图G存在，u是和G中顶点关键字类型相同的给定值；操作结果是若u在图G中存在，返回关键字为u的顶点位置信息，否则返回其它信息。
3. 顶点赋值：函数名称是PutVex (G,u,value)；初始条件是图G存在，u是和G中顶点关键字类型相同的给定值；操作结果是对关键字为u的顶点赋值value。
4. 获得第一邻接点：函数名称是FirstAdjVex(G, u)；初始条件是图G存在，u是G中顶点的位序；操作结果是返回u对应顶点的第一个邻接顶点位置信息，如果u的顶点没有邻接顶点，否则返回其它表示“空”的信息。
5. 获得下一邻接点：函数名称是NextAdjVex(G, v, w)；初始条件是图G存在，v和w是G中两个顶点的位序，v对应G的一个顶点,w对应v的邻接顶点；操作结果是返回v的（相对于w）下一个邻接顶点的位置信息，如果w是最后一个邻接顶点，返回其它表示“空”的信息。
6. 插入顶点：函数名称是InsertVex(G,v)；初始条件是图G存在，v和G中的顶点具有相同特征；操作结果是在图G中增加新顶点v。（在这里也保持顶点关键字的唯一性）
7. 删除顶点：函数名称是DeleteVex(G,v)；初始条件是图G存在，v是G的一个顶点；操作结果是在图G中删除顶点v和与v相关的弧。
8. 插入弧：函数名称是InsertArc(G,v,w)；初始条件是图G存在，v、w是G的顶点；操作结果是在图G中增加弧<v,w>，如果图G是无向图，还需要增加<w,v>。
9. 删除弧：函数名称是DeleteArc(G,v,w)；初始条件是图G存在，v、w是G的顶点；操作结果是在图G中删除弧<v,w>，如果图G是无向图，还需要删除<w,v>。
10. 深度优先搜索遍历：函数名称是DFSTraverse(G,visit())；初始条件是图G存在；操作结果是图G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。
11. 广深度优先搜索遍历：函数名称是BFSTraverse(G,visit())；初始条件是图G存在；操作结果是图G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。

## 4.2 系统设计

**4.2.1 总体设计**

（1）使用一个大的while语句实现整体功能；

（2）使用switch语句实现菜单栏的操作选择；

（3）在每个case语句先进行对图是否存在的判断；

**4.2.2 算法设计**

（1）CreateGraph(ALGraph\* G);

设计思路：创建图。开辟一个图的存储空间，新建一个顶点并对其进行初始化。

（2）DestroyGraph(ALGraph\* G);

设计思路：销毁图。遍历邻接表的每个顶点和弧进行空间释放，最后将全局变量isNull（判定表是否存在）改为TRUE值；

（3）LocateVex(ALGraph G,KeyType u);

设计思路：查找顶点。遍历邻接表内的头结点，按顺序查找与输入值相对应的顶点；

（4）PutVex(ALGraph\* G,KeyType u,ValueType v);

设计思路：顶点赋值。遍历邻接表内的顶点，按顺序查找与输入值相对应的头结点，如找到则对该头结点进行赋值操作，否则返回查找失败；

（5）FirstAdjVex(ALGraph G, int index);

设计思路：获得第一邻接点。输入位序，判断输入是否在表长范围内，然后直接用if语句判断该位序的顶点的第一邻接点是否存在，若存在则返回第一邻接点的顶点信息；

（6）NextAdjVex(ALGraph G, int v, int w);

设计思路：获得下一邻接点。遍历邻接表内的头结点，按顺序查找与输入值v相对应的头结点，从该顶点的邻接弧中逐个遍历，若找到邻接顶点w且其下一邻接点存在，则返回下一邻接点的顶点信息，否则返回查找失败；

（7）InsertVex(ALGraph\* G, VertexType vnode)；

设计思路：插入顶点。若检测到输入的关键字已存在，则返回错误；否则新建一个头结点；

（8）DeleteVex(ALGraph\* G, KeyType vnode);

设计思路：删除顶点。首先查找对应关键字的顶点，若找到则在顺序表内删除头结点和以其为弧尾的表结点，然后遍历每个顶点和相关的弧，找到以该顶点为弧头的表结点并删除弧；

（9）InsertArc(ALGraph\* G, KeyType v, KeyType w);

设计思路：插入弧。首先定位v和w在头结点中都存在；然后找到弧尾v所在的头结点，遍历到表结点的最尾，若未找到已有的弧(v,w)，则在最尾插入弧头结点w，否则返回插入失败；

（10）DeleteArc(ALGraph\* G, KeyType v, KeyType w);

设计思路：删除弧。首先定位v和w在头结点中都存在，然后找到弧尾v所在的头结点，遍历到表结点的最尾，若找到已有的弧(v,w)，则删除该弧，否则返回删除失败；

（11）DFSTraverse(ALGraph\* G);

设计思路：深度优先搜索遍历。对每个顶点打上标记，对未遍历的顶点利用递归进行深度优先搜索遍历；

（12）BFSTraverse(ALGraph\* G);

设计思路：利用队列完成广度优先搜索遍历，将每个顶点及所对应的弧入队，然后根据出队的顶点顺序，对这些顶点的表结点进行入队，直到队列为空。

## 4.3 系统实现

**4.3.1 函数实现**

（1）CreateGraph：输入顶点的key值（整形，唯一确定一个顶点），和该结点所携带的信息，如果输入无误则新建一个头结点放入顺序表中，并设置它的第一邻接点为NULL；

（2）DestroyGraph：遍历邻接表的每个顶点和弧进行空间释放，最后将全局变量isNull（判定表是否存在）改为TRUE值；

（3）LocateVex： 遍历邻接表内的顺序表，当遍历到的头结点的关键字与输入值u相同时，则返回当前头结点的位序，否则返回ERROR；

（4）PutVex：遍历邻接表内的顺序表，当遍历到的头结点的关键字与输入值u相同时，则对该头结点的结点信息进行赋值操作并返回OK，否则返回ERROR；

（5）FirstAdjVex：输入位序，判断输入是否在表长范围内，然后直接用if语句判断该位序的顶点的第一邻接点是否存在，若存在则返回第一邻接点的顶点信息；否则返回ERROR；

（6）NextAdjVex：遍历邻接表内的顺序表，当遍历到的头结点的关键字与输入值u相同时，从该顶点的邻接弧中逐个遍历，若找到邻接顶点w且其下一邻接点存在，则返回下一邻接点的顶点信息，否则返回ERROR；

（7）InsertVex：若检测到输入的关键字已存在，则输入有误，返回菜单页面；否则在顺序表最后新建一个头结点，对其进行初始化赋值，同时表G的vexnum（顶点数）加一；

（8）DeleteVex：首先查找对应关键字的顶点，若找不到则返回查找失败；若找到顶点则在顺序表内删除此头结点和所有以其为弧尾的表结点；然后遍历每个顶点和相关的弧，找到以该顶点为弧头的表结点并删除这些弧；每删除一个顶点，全局变量DeleteNum加一；删除成功则同时表G的vexnum（顶点数）减一并返回OK；

（9）InsertArc：首先调用LocataVex函数判断v和w在头结点中是否存在，若有一个不存在就返回ERROR；然后找到弧尾v所在的头结点，遍历到表结点的最尾，若未找到已有的弧(v,w)，则在最尾插入表结点w，邻接表G的arcnum（弧的数量）加一返回OK；否则打印该弧已存在并返回ERROR；

（10）DeleteArc：首先调用LocataVex函数判断v和w在头结点中是否存在，若有一个不存在就返回ERROR；然后找到弧尾v所在的头结点，遍历到表结点的最尾，若找到已有的弧(v,w)，则通过链表操作删除该弧，邻接表G的arcnum（弧的数量）减一并返回OK；否则返回ERROR；

（11）DFSTraverse：先把邻接表G内每个头结点的book值置零（用于判断是否遍历过该顶点）；从位序为0的头结点的第一邻接点开始进行递归遍历，直到遇到遍历过的顶点，则返回上一层的下一个邻接点继续遍历，并打印顶点信息。所有顶点遍历完毕则返回OK；

（12）BFSTraverse：新建一个队列q；把邻接表G内每个头结点的book值置零（用于判断是否遍历过该顶点）；从位序为0的头结点开始，把它作为弧尾的弧所指向的表结点信息按顺序push进入队列；打印队头结点信息并对该结点执行同上的操作；反复执行直到队列为空；所有顶点遍历完毕则返回OK；

**4.3.2 源代码**

（源代码详细见附录D）

## 4.4 系统测试

编程环境：Win10

编译器：Visual Studio 2019

主模块运行如图4-1所示：

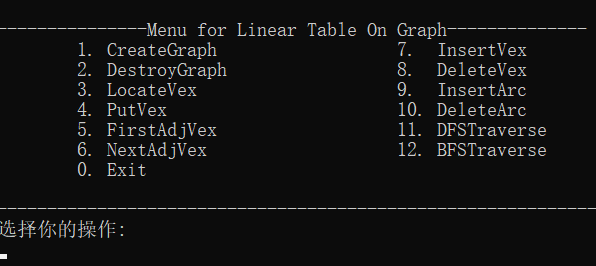


图4-1 主模块图

功能操作：

1. CreateGraph：创建图；
2. 输入“100”和“95”，创建一个位序为0，名称为100，携带信息为95的头结点；（如图4-2所示）
3. 在未创建图的情况下选择任一选项都会返回“图不存在！”；（如图4-3所示）

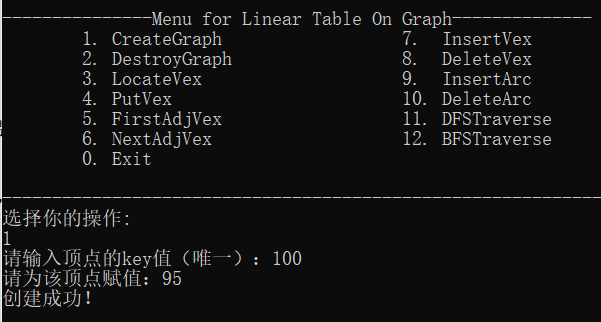


图4-2 创建头结点1

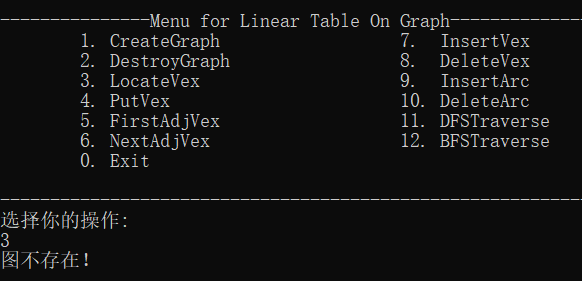


图4-3 图不存在反馈

1. DestroyGraph：销毁图；
2. 返回销毁成功；（如图4-4所示）
3. 图不存在，则返回操作失败；

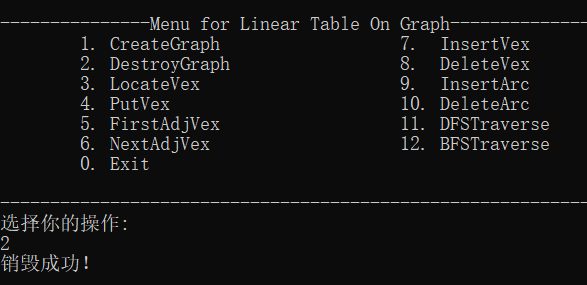


图4-4 二叉树销毁成功

1. LocateVex：查找顶点；
2. 调用后面的函数创建一个图；（图的形式如图4-5所示）
3. 对图进行深度遍历和广度遍历；（如图4-6，4-7所示）
4. 输入102，查找成功，获取返回值；（如图4-8所示）
5. 输入200，查找失败；（如图4-9所示）
6. 图不存在，则返回操作失败；

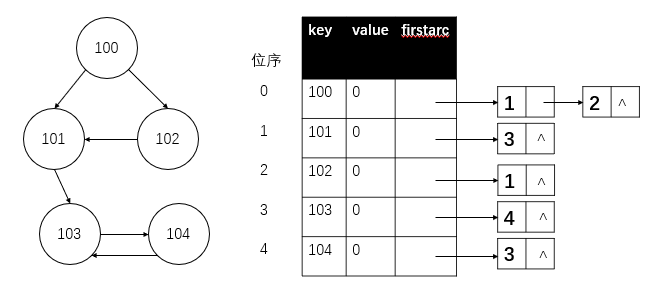


图4-5创建图

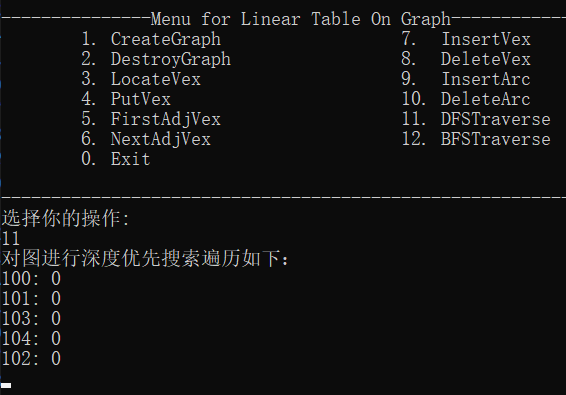


图4-6 深度遍历图



图4-7 广度遍历图

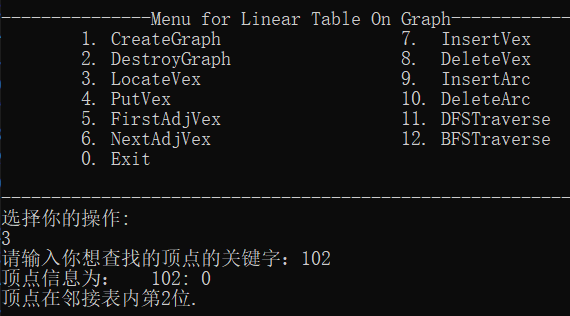


图4-8 查找成功

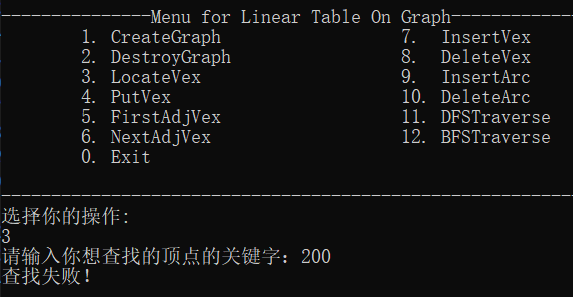


图4-9 查找失败

1. PutVex：为顶点赋值；
2. 使用（3）中第一步建立的图；
3. 分别为100,101,102,103,104赋值96,97,98,99,100；（如图4-10所示）
4. 对图进行深度遍历；（如图4-11所示）
5. 输入为201赋值，则返回找不到该顶点；
6. 图不存在，则返回操作失败；

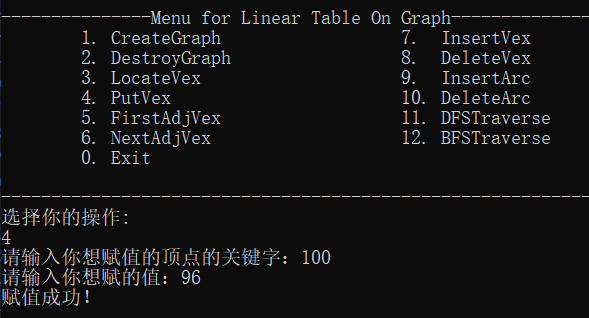


图4-10 赋值成功



图4-11 深度遍历图

1. FirstAdjVex：获得第一邻接点；
2. 使用（4）中已被赋值的图；
3. 输入0，返回位序为0的100的第一邻接点101的位序1；（如图4-12所示）
4. 输入6，返回输入不符合要求；（如图4-13所示）
5. 图不存在，则返回操作失败；

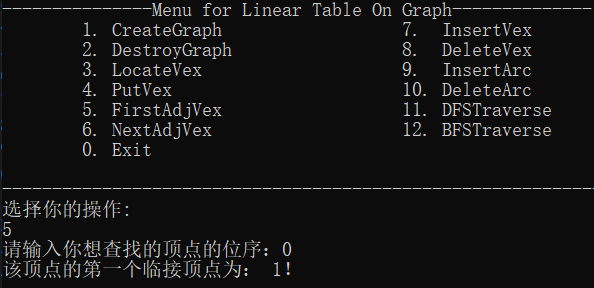


图4-12 返回第一邻接点

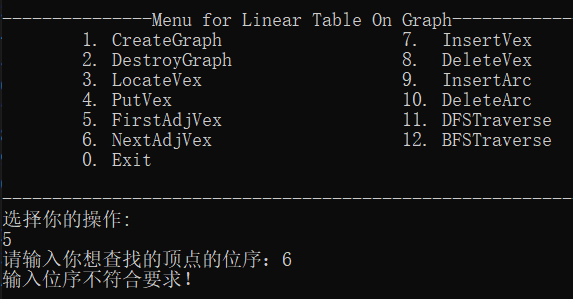


图4-13 输入不符合要求

1. NextAdjVex：获得下一邻接点；

1) 使用（4）中已被赋值的图；

2) 输入0和1，返回下一邻接点102的位序2；（如图4-14所示）

3) 输入6和7，返回输入有误，查找失败；

4) 输入3和4，不存在下一邻接点，查找失败；

5) 图不存在，则返回操作失败；

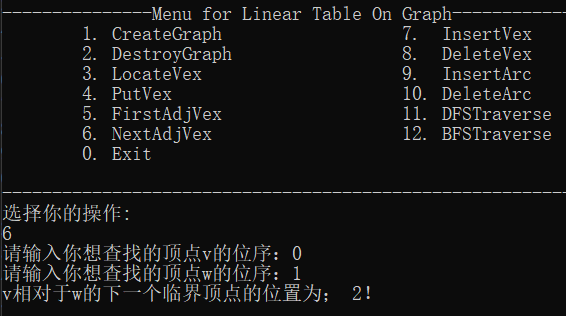


图4-14 查找成功

1. InsertVex：插入顶点；
2. 使用（4）中已被赋值的图；
3. 插入顶点105，60，插入成功；（如图4-15所示）
4. 插入顶点101，10，该关键字已存在，插入失败；（如图4-16所示）

4) 图不存在，则返回操作失败；

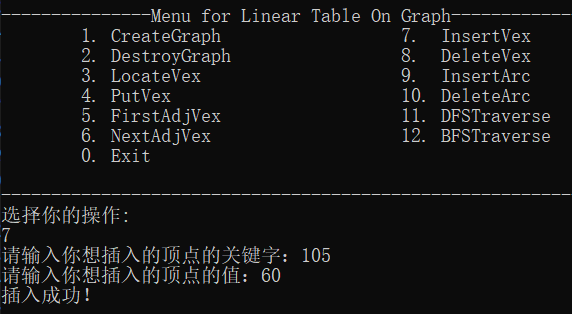


图4-15 插入成功

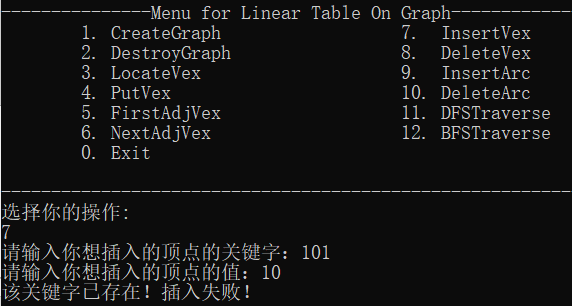


图4-16 插入失败

1. DeleteVex：删除顶点；
2. 使用（7）中插入新顶点后的图；
3. 删除104，返回删除成功（如图4-17所示）；
4. 此时图的形状；（如图4-18所示）
5. 删除200，顶点不存在，返回删除失败；
6. 图不存在，则返回操作失败；

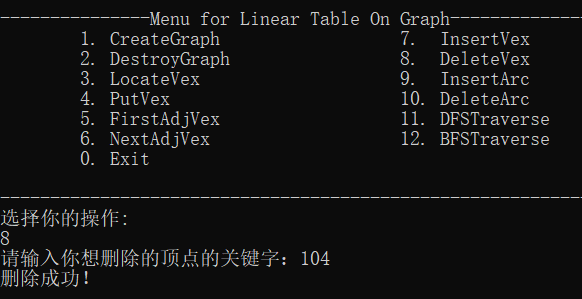


图4-17 删除成功

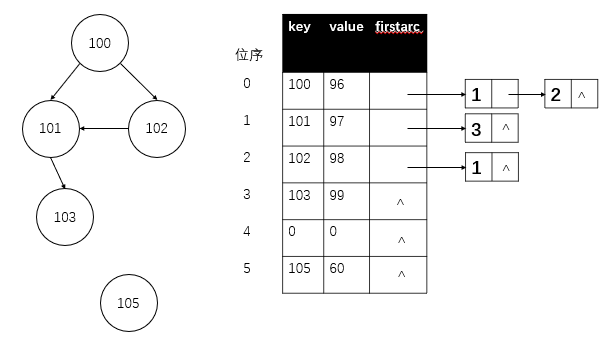


图4-18 此时的图

1. InsertArc：插入弧；
2. 使用（7）中删除完成的图；
3. 插入（103,105）弧，进行深度遍历；（如图4-19，4-20所示）
4. 插入（101,103）弧，返回该弧已存在；
5. 插入（100,200）弧，返回插入失败；
6. 图不存在，则返回操作失误；

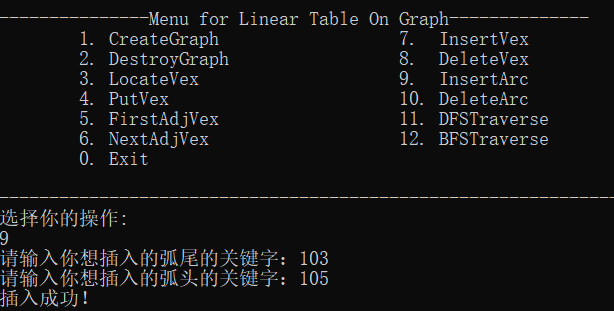


图4-19 插入弧成功

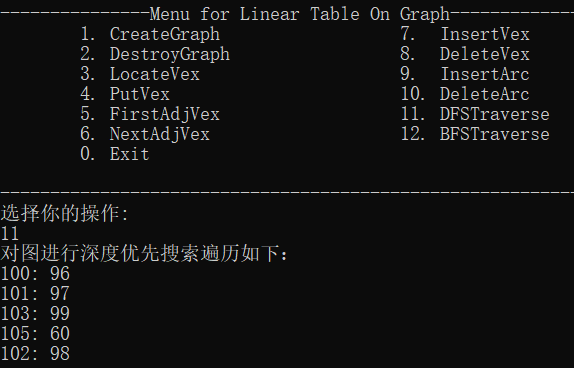


图4-20 深度遍历

1. DeleteArc：删除弧；
2. 使用（9）中插入弧完成的图；
3. 删除（102,101），删除成功；（如图4-21所示）
4. 删除（105,101），返回该弧不存在，删除失败；
5. 图不存在，则返回操作失误；

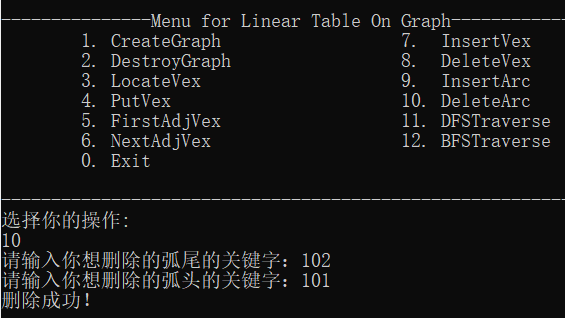


图4-21 删除弧成功

1. DFSTraverse：深度优先搜索遍历；
2. 使用（10）中删除完成的图，进行深度优先搜索遍历；（图的形态如图4-22所示，遍历如图4-23所示）
3. 图不存在，则返回操作失误；

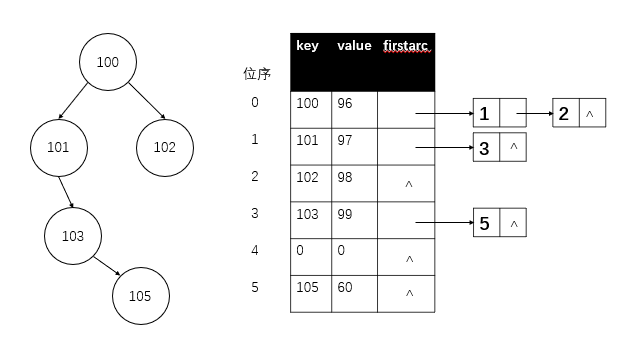


图4-22 图的形态

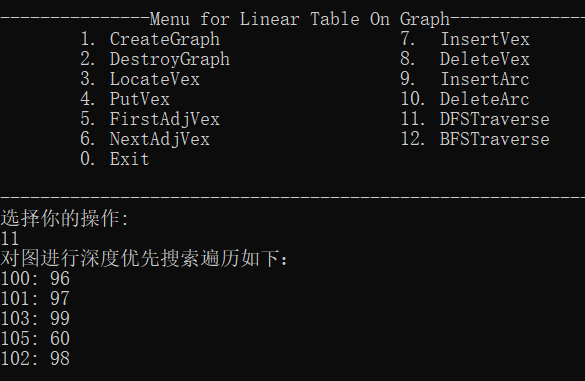


图4-23 深度遍历

1. BFSTraverse：广度优先搜索遍历；
2. 使用（10）中删除完成的图，进行广度优先搜索遍历；（如图4-24所示）
3. 图不存在，则返回操作失误；

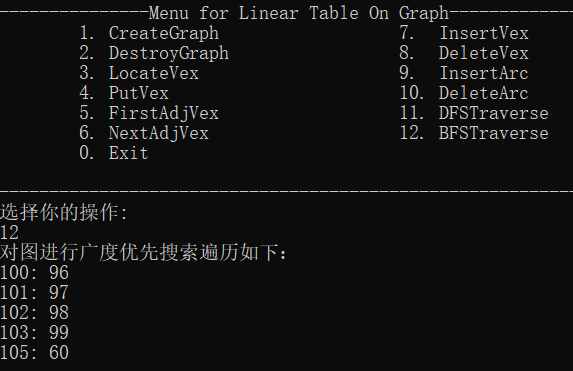


图4-24 广度遍历

## 实验小结

本次 “基于邻接表的图实现”实验也是数据结构的最后一个实验，也是一次综合性的实验，在邻接表的构建中，用到了第一次顺序表实验、第二次链表实验的思想，同时写完二叉树实验的经历也让我编写遍历函数时的能力有所提升。当然，在本次实验中也出现了一些问题，收获了一些新经验：

1. 在写添加新弧时，一开始没有加上检测（v,w）弧是否存在的判断语句，导致检查时候闹了乌龙。所以以后对边界条件的判断要更加小心谨慎，尽量考虑好每一种输入可能带来的结果；
2. 在写深度优先搜索遍历和广度优先搜索遍历的时候，一开始没有思路，于是去网站上搜寻别人的代码，参照之前写二叉树遍历的经验，一行行地理解下来，在看懂后再自己动手按照思路写了一遍，然后多次运行修正。虽然花费了一些时间，但我认为这是最有效且掌握最佳的学习方法，帮助我更好地理解了深度遍历和广度遍历函数的理论思想和构建方法。

通过本次实验我学习到了很多，接下来的数据结构课设会是前所未有的难关，但我从这四次实验中学到的不只是函数的构建方法，更多的是高效的学习方法、细心谨慎的态度，以及完成实验的成就感所激发的学习的主观能动性。之后我也会不停止努力，充分吸收这四次实验给我带来的经验和启发，更好地完成之后的学习和实验。

# 参考文献

[1] 严蔚敏等. 数据结构(C语言版). 清华大学出版社

[2] [Larry Nyhoff](http://www.calvin.edu/~nyhl/index.html). [ADTs, Data Structures, and Problem Solving with C++.](http://vig.prenhall.com/catalog/academic/product/0,1144,0131409093,00.html)Second Edition, [Calvin College](http://cs.calvin.edu/), 2005

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

[4] 严蔚敏等.数据结构题集(C语言版). 清华大学出版社

# 附录A 基于顺序存储结构线性表实现的源程序

|  |
| --- |
| /\* Linear Table On Sequence Structure \*/  #include <stdio.h>  #include <malloc.h>  #include <stdlib.h>  /\*---------DATA DEFINE---------\*/  #define TRUE 1  #define FALSE 0  #define OK 1  #define ERROR 0  #define INFEASTABLE -1  #define OVERFLOW -2  /\*-------TYPE DEFINE-------\*/  typedef int status;  typedef int ElemType;  #define LIST\_INIT\_SIZE 100  #define LISTINCREMENT 10  typedef struct { //顺序表（顺序结构）的定义  ElemType\* elem;  int length;  int listsize;  }SqList;  extern bool isNull = TRUE; //全局变量，判定表是否为空  /\*-----FUNCTION DEFINE---------\*/  status InitList(SqList& L); //创建表  status DestroyList(SqList& L); //删除表  status ClearList(SqList& L); //清空表  status ListEmpty(SqList L); //判断表是否为空  int ListLength(SqList L); //表长  status GetElem(SqList L, int i, ElemType& e); //获得表元素  status LocateElem(SqList L, ElemType e, status(\*Compare)(ElemType a, ElemType b)); //定位表元素  status PriorElem(SqList L, ElemType cur, ElemType\* pre\_e); //求前驱  status NextElem(SqList L, ElemType cur, ElemType\* next\_e); //求后继  status ListInsert(SqList& L, int i, ElemType e); //插入元素  status ListDelete(SqList& L, int i, ElemType& e); //删除元素  status ListTrabverse(SqList L); //删除表  status Compare(ElemType a, ElemType b); //比较  void main(void) {  SqList L;  int op = 1;  int index;  ElemType e, cue, pre, next;  while (op) {  system("cls");  printf("\n\n");  printf(" Menu for Linear Table On Sequence Structure \n");  printf("-------------------------------------------------\n");  printf(" 1. InitList 7. LocateElem\n");  printf(" 2. DestroyList 8. PriorElem\n");  printf(" 3. ClearList 9. NextElem \n");  printf(" 4. ListEmpty 10. ListInsert\n");  printf(" 5. ListLength 11. ListDelete\n");  printf(" 6. GetElem 12. ListTrabverse\n");  printf(" 0. Exit\n");  printf("-------------------------------------------------\n");  printf(" 请选择你的操作[0~12]:");  scanf\_s("%d", &op);  getchar();  switch (op) {  case 1:  if (InitList(L) == OK)  printf("线性表创建成功！\n");  else  printf("线性表创建失败！\n");  getchar();  break;  case 2:  if (isNull == TRUE) {  printf("线性表不存在！\n");  getchar();  break;  }    if (DestroyList(L) == OK)  printf("线性表销毁成功！\n");  else  printf("线性表销毁失败！\n");  getchar();  break;  case 3:  if (isNull == TRUE) {  printf("线性表不存在！\n");  getchar();  break;  }  if (ClearList(L) == OK)  printf("线性表重置为空表成功！\n");  else  printf("线性表重置为空表失败！\n");  getchar();  break;  case 4:  if (isNull == TRUE) {  printf("线性表不存在！\n");  getchar();  break;  }  if (ListEmpty(L) == TRUE)  printf("线性表为空！\n");  else  printf("线性表不为空！\n");  getchar();  break;  case 5:  if (isNull == TRUE) {  printf("线性表不存在！\n");  getchar();  break;  }  printf("线性表有%d个元素！\n", ListLength(L));  getchar();  break;  case 6:  if (isNull == TRUE) {  printf("线性表不存在！\n");  getchar();  break;  }  printf("输入想获得的第i个元素：");  scanf\_s("%d", &index);  getchar();  if (GetElem(L, index, e) != ERROR)  printf("线性表的第%d个元素为%d！\n", index, e);  else  printf("输入不在范围内！\n");  getchar();  break;  case 7:  if (isNull == TRUE) {  printf("线性表不存在！\n");  getchar();  break;  }  printf("输入想查找的元素：");  scanf\_s("%d", &e);  getchar();  if (LocateElem(L, e, Compare) > 0)  printf("该元素的位置在%d！\n", LocateElem(L, e, Compare));  else  printf("元素不存在！\n");  getchar();  break;  case 8:  if (isNull == TRUE) {  printf("线性表不存在！\n");  getchar();  break;  }  printf("输入想查找的元素：");  scanf\_s("%d", &cue);  getchar();  if (PriorElem(L, cue, &pre) > 0)  printf("该元素的前驱是%d！\n", pre);  else  printf("查找失败！\n");  getchar();  break;  case 9:  if (isNull == TRUE) {  printf("线性表不存在！\n");  getchar();  break;  }  printf("输入想查找的元素：");  scanf\_s("%d", &cue);  getchar();  if (NextElem(L, cue, &next) > 0)  printf("该元素的后继是%d！\n", next);  else  printf("查找失败！\n");  getchar();  break;  case 10:  if (isNull == TRUE) {  printf("线性表不存在！\n");  getchar();  break;  }  printf("输入想插入的位置：");  scanf\_s("%d", &index);  getchar();  printf("输入想插入的数字：");  scanf\_s("%d", &e);  getchar();  if (ListInsert(L, index, e) == OK)  printf("插入成功！\n");  else  printf("插入失败！\n");  getchar();  break;  case 11:  if (isNull == TRUE) {  printf("线性表不存在！\n");  getchar();  break;  }  printf("输入想删除的位置：");  scanf\_s("%d", &index);  getchar();  if (ListDelete(L, index, e) != ERROR)  printf("删除成功！%d已被删除\n", e);  else  printf("删除失败！\n");  getchar();  break;  case 12:  if (isNull == TRUE) {  printf("线性表不存在！\n");  getchar();  break;  }  ListTrabverse(L);  getchar();  break;  case 0:  break;  }//end of switch  }//end of while  printf("欢迎下次再使用本系统！\n");  getchar();  }//end of main()  /\*--------Funcions---------------\*/  status InitList(SqList& L) {  if (isNull == FALSE) {  printf("已将之前的表清除并新建空表！");  getchar();  }  L.elem = (ElemType\*)malloc(LIST\_INIT\_SIZE \* sizeof(ElemType));  if (!L.elem)  exit(OVERFLOW);  L.length = 0;  L.listsize = LIST\_INIT\_SIZE;  isNull = FALSE;  return OK;  }  status DestroyList(SqList& L) {  L.length = 0;  isNull = TRUE;  free(L.elem);  L.elem = NULL;  return OK;  }  status ClearList(SqList& L) {  if (!L.elem) return ERROR;  L.length = 0;  return OK;  }  status ListEmpty(SqList L) {  if (L.length == 0) return TRUE;  else return FALSE;  }  int ListLength(SqList L) {  if (L.elem) return L.length;  else return ERROR;  }  status GetElem(SqList L, int i, ElemType& e) {  if (ListEmpty(L) == TRUE) {  printf("线性表为空！");  return ERROR;  }  if (i >= 1 && i <= ListLength(L)) {  e = L.elem[i - 1];  return e;  }  else {  printf("输入有误，表中无该元素！");  return ERROR;  }  }  status Compare(ElemType p, ElemType q) {  if (p == q)  return TRUE;  else  return FALSE;  }  status LocateElem(SqList L, ElemType e, status Compare(ElemType, ElemType)) {  int index = 1;  for (index; index <= L.length; index++) {  if ((\*Compare)(L.elem[index - 1], e)) {  return index;  }  }  return ERROR;  }  status PriorElem(SqList L, ElemType cur\_e, ElemType\* pre\_e) {  int i;  for (i = 0; i < L.listsize; i++) {  if (L.elem[i] == cur\_e) {  if (i == 0) {  printf("该元素不存在前驱！");  return ERROR;  }  \*pre\_e = L.elem[i - 1];  return OK;  }  }  printf("表中没有该元素！");  return ERROR;  }  status NextElem(SqList L, ElemType cur\_e, ElemType\* next\_e) {  int i;  for (i = 0; i < L.listsize - 1; i++) {  if (L.elem[i] == cur\_e) {  if (i == L.length - 1) {  printf("已经是最后的元素了！");  return ERROR;  }  \*next\_e = L.elem[i + 1];  return OK;  }  }  printf("表中没有该元素！");  return ERROR;  }  status ListInsert(SqList& L, int i, ElemType e) {  if (i < 1 || i > L.length + 1) return ERROR;  if (L.length >= L.listsize) {  ElemType\* newbase = (ElemType\*)realloc(L.elem, (L.length + LISTINCREMENT) \* sizeof(ElemType));  if (!newbase) exit(OVERFLOW);  L.elem = newbase;  L.listsize += LISTINCREMENT;  }  ElemType\* q = &(L.elem[i - 1]);  for (ElemType\* p = &(L.elem[L.length - 1]); p >= q; p--)  {  \*(p + 1) = \*p;  }  \*q = e;  L.length += 1;  return OK;  }  status ListDelete(SqList& L, int i, ElemType& e) {  if (i < 1 || i > L.length)  return ERROR;  ElemType\* p = &(L.elem[i - 1]), \* q;  e = L.elem[i - 1];  q = L.elem + L.length - 1;  for (++p; p <= q; ++p)\* (p - 1) = \*p;  L.length -= 1;  return e;  }  status ListTrabverse(SqList L) {  int i;  printf("\n-----------all elements -----------------------\n");  for (i = 0; i < L.length; i++) printf(" %d ", L.elem[i]);  printf("\n------------------ end ------------------------\n");  return L.length;  } |

# 附录B 基于链式存储结构线性表实现的源程序

|  |
| --- |
| /\* Linear Table On Sequence Structure \*/  #include <stdio.h>  #include <malloc.h>  #include <stdlib.h>  #include <string.h>  /\*---------DATA DEFINE---------\*/  #define TRUE 1  #define FALSE 0  #define OK 1  #define ERROR 0  #define INFEASTABLE -1  #define OVERFLOW -2  /\*-------TYPE DEFINE-------\*/  typedef int status;  typedef int ElemType;  #define LIST\_INIT\_SIZE 10  #define LISTINCREMENT 10  typedef struct Lnode{ //链表的定义  ElemType data;  struct Lnode\* next;  }LNode , \*LinkList ;  extern bool isNull = TRUE;  /\*-----FUNCTION DEFINE---------\*/  status InitList(LinkList\* L); //创建表  status DestroyList(LinkList\* L); //删除表  status ClearList(LinkList\* L); //清空表  status ListEmpty(LinkList L); //判断表是否为空  int ListLength(LinkList L); //表长  status GetElem(LinkList L, int i, ElemType\* e); //获得表元素  status LocateElem(LinkList L, ElemType e, status(\*Compare)(ElemType a, ElemType b)); //定位表元素  status PriorElem(LinkList L, ElemType cur, ElemType\* pre\_e); //求前驱  status NextElem(LinkList L, ElemType cur, ElemType\* next\_e); //求后继  status ListInsert(LinkList\* L, int i, ElemType e); //插入元素  status ListDelete(LinkList\* L, int i, ElemType& e); //删除元素  status ListTrabverse(LinkList L); //删除表  status Compare(ElemType a, ElemType b); //比较  int main(void) {  LinkList L = NULL;  int op = 1;  int index;  ElemType e, cue, pre, next;  while (op) {  system("cls");  printf("\n\n");  printf(" Menu for Linear Table On Link Structure \n");  printf("-------------------------------------------------\n");  printf(" 1. InitList 7. LocateElem\n");  printf(" 2. DestroyList 8. PriorElem\n");  printf(" 3. ClearList 9. NextElem \n");  printf(" 4. ListEmpty 10. ListInsert\n");  printf(" 5. ListLength 11. ListDelete\n");  printf(" 6. GetElem 12. ListTrabverse\n");  printf(" 0. Exit\n");  printf("-------------------------------------------------\n");  printf(" 请选择你的操作[0~12]:");  scanf\_s("%d", &op);  getchar();  switch (op) {  case 1:  if (InitList(&L) == OK)  printf("链表创建成功！\n");  else  printf("链表创建失败！\n");  getchar();  break;  case 2:  if (isNull == TRUE) {  printf("链表不存在！\n");  getchar();  break;  }  if (DestroyList(&L) == OK)  printf("链表销毁成功！\n");  else  printf("链表销毁失败！\n");  getchar();  break;  case 3:  if (isNull == TRUE) {  printf("链表不存在！\n");  getchar();  break;  }  if (ClearList(&L) == OK)  printf("链表重置为空表成功！\n");  else  printf("链表重置为空表失败！\n");  getchar();  break;  case 4:  if (isNull == TRUE) {  printf("链表不存在！\n");  getchar();  break;  }  if (ListEmpty(L) == TRUE)  printf("链表为空！\n");  else  printf("链表不为空！\n");  getchar();  break;  case 5:  if (isNull == TRUE) {  printf("链表不存在！\n");  getchar();  break;  }  printf("链表有%d个元素！\n", ListLength(L));  getchar();  break;  case 6:  if (isNull == TRUE) {  printf("链表不存在！\n");  getchar();  break;  }  printf("输入想获得的第i个元素：");  scanf\_s("%d", &index);  getchar();  if (GetElem(L, index, &e) != ERROR)  printf("链表的第%d个元素为%d！\n", index, e);  else  printf("输入不在范围内！\n");  getchar();  break;  case 7:  if (isNull == TRUE) {  printf("链表不存在！\n");  getchar();  break;  }  printf("输入想查找的元素：");  scanf\_s("%d", &e);  getchar();  if (LocateElem(L, e, Compare) !=ERROR)  printf("该元素的位置在%d！\n", LocateElem(L, e, Compare));  else  printf("元素不存在！\n");  getchar();  break;  case 8:  if (isNull == TRUE) {  printf("线性表不存在！\n");  getchar();  break;  }  printf("输入想查找的元素：");  scanf\_s("%d", &cue);  getchar();  if (PriorElem(L, cue, &pre) > 0)  printf("该元素的前驱是%d！\n", pre);  else  printf("查找失败！\n");  getchar();  break;  case 9:  if (isNull == TRUE) {  printf("链表不存在！\n");  getchar();  break;  }  printf("输入想查找的元素：");  scanf\_s("%d", &cue);  getchar();  if (NextElem(L, cue, &next) > 0)  printf("该元素的后继是%d！\n", next);  else  printf("查找失败！\n");  getchar();  break;  case 10:  if (isNull == TRUE) {  printf("链表不存在！\n");  getchar();  break;  }  printf("输入想插入的位置：");  scanf\_s("%d", &index);  getchar();  printf("输入想插入的数字：");  scanf\_s("%d", &e);  getchar();  if (ListInsert(&L, index, e) == OK)  printf("插入成功！\n");  else  printf("插入失败！\n");  getchar();  break;  case 11:  if (isNull == TRUE) {  printf("线性表不存在！\n");  getchar();  break;  }  printf("输入想删除的位置：");  scanf\_s("%d", &index);  getchar();  if (ListDelete(&L, index, e) != ERROR)  printf("删除成功！%d已被删除\n", e);  else  printf("删除失败！\n");  getchar();  break;  case 12:  if (isNull == TRUE) {  printf("链表不存在！\n");  getchar();  break;  }  ListTrabverse(L);  getchar();  break;  case 0:  break;  }//end of switch  }//end of while  printf("欢迎下次再使用本系统！\n");  getchar();  }//end of main()  /\*--------Funcions---------------\*/  status InitList(LinkList\* L) {  \*L = (LinkList)malloc(sizeof(LNode));  if (\*L == NULL)  exit(OVERFLOW);  isNull = FALSE;  (\*L)->data = 0;  (\*L)->next = NULL;  return OK;  }  status DestroyList(LinkList\* L) {  LinkList p ,q ;  p = \*L;  while (p) {  q = p->next;  free(p);  p = q;  }  \*L = NULL;  isNull = TRUE;  return OK;  }  status ClearList(LinkList\* L) {  LinkList p , q ;  p = (\*L)->next;  while (p) {  q = p->next;  free(p);  p = q;  }  (\*L)->next = NULL;  return OK;  }  status ListEmpty(LinkList L) {  if (L->next == NULL) return TRUE;  else return FALSE;  }  int ListLength(LinkList L) {  int i = 0;  LinkList p ;  p = L->next;  while (p) {  p = p->next;  i++;  }  return i;  }  status GetElem(LinkList L, int i, ElemType\* e) {  int index = 1;  LinkList p = L->next;  if (i < 1 || i > ListLength(L)) {  return ERROR;  }  if (ListEmpty(L) == TRUE) {  printf("链表为空！");  return ERROR;  }  if (i == 1)  return p->data;  while (index < i && p) {  p = p->next;  index++;  }  \*e = p->data;  return OK;  }  status Compare(ElemType p, ElemType q) {  if (p == q)  return TRUE;  else  return FALSE;  }  status LocateElem(LinkList L, ElemType e, status Compare(ElemType, ElemType)) {  int index = 1;  LinkList p = L->next;  for (index; index <= ListLength(L) && p; index++) {  if (Compare(p->data,e)== TRUE) {  return index;  }  p = p->next;  }  return ERROR;  }  status PriorElem(LinkList L, ElemType cur\_e, ElemType\* pre\_e) {  int i;  LinkList p = L->next ;  while (p->next != NULL && p->next->data != cur\_e) {  p = p->next;  }  if (p->next == NULL) {  return ERROR;  }  \*pre\_e = p->data;  return OK;  }  status NextElem(LinkList L, ElemType cur\_e, ElemType\* next\_e) {  int i;  LinkList p = L->next ;  while (p->next != NULL && p->data != cur\_e) {  p = p->next;  }  if (p->next == NULL) {  return ERROR;  }  \*next\_e = p->next->data;  return OK;  }  status ListInsert(LinkList\* L, int i, ElemType e) {  LinkList p, q;  p = \*L;  int j = 1;  q = (LinkList)malloc(sizeof(LNode));  if (q == NULL) exit(OVERFLOW);  q->data = e;  if (i == 1) {  q->next = p->next;  (\*L)->next = q;  return OK;  }  for (p ; j < i && p; j++) {  p = p->next;  }  if (p == NULL || j > i) {  return ERROR;  }  if (p->next == NULL) {  q->next = NULL;  }  else{  q->next = p->next;  }  p->next = q;  return OK;  }  status ListDelete(LinkList\* L, int i, ElemType& e) {  if (i < 1 || i > ListLength(\*L))  return ERROR;  LinkList p ,q;  p = \*L;  int j = 1;  while (p!=NULL && j < i) {  p = p->next;  j++;  }  if (p == NULL || j > i) {  return ERROR;  }  q = p->next;  if (q->next == NULL) {  p->next = NULL;  e = q->data;  free(q);  return OK;  }  else {  p->next = q->next;  e = q->data;  free(q);  q = NULL;  return OK;  }  return ERROR;  }  status ListTrabverse(LinkList L) {  LinkList p ;  p = L->next;  if (ListLength(L) == 0) {  printf("链表中无元素！");  }  printf("\n-----------all elements -----------------------\n");  while (p != NULL) {  printf(" %d ", p->data);  p = p->next;  }  printf("\n------------------ end ------------------------\n");  return OK;  } |

# 附录C 基于二叉链表二叉树实现的源程序

|  |
| --- |
| #include <stdio.h>  #include <malloc.h>  #include <stdlib.h>  #include <string.h>  #include <math.h>  #define TRUE 1  #define FALSE 0  #define OK 1  #define ERROR 0  #define INFEASIBLE -1  #define OVERFLOW -2  #define LIST\_INIT\_SIZE 1024  #define NUM\_MAX 30  typedef int Status;  typedef int TElemType;  typedef char KeyType;  static bool isNull = TRUE;  static int i = -1;  typedef struct ElemType/\*树结点存放的数据的结构类型\*/  {  KeyType key; /\*唯一标识\*/  TElemType value;  }ElemType;  typedef struct BiTNode { /\*树的每个结点的结构\*/  ElemType data;  BiTNode\* lchild, \* rchild;  }BiTNode, \* TNode;  typedef struct QueueList {  TNode data[LIST\_INIT\_SIZE];  int front;  int rear;  }queue, \* Queue;  typedef struct Stack {  TNode data[LIST\_INIT\_SIZE];  int top;  }Stack, \* stacknode;  static TNode Findnode = NULL; /\*全局变量，查找结点\*/  static TNode Brothernode = NULL; /\*全局变量，兄弟结点\*/  static TNode Parentnode = NULL; /\*全局变量，双亲节点\*/  FILE\* tree = NULL; /\*全局变量，文件指针\*/  /\*Function defined\*/  Status CreateBiTree(TNode\* T, char\* definition); /\*生成二叉树\*/  Status DestroyBiTree(TNode\* T); /\*销毁二叉树\*/  Status ReDestroyBiTree(TNode temp); /\*销毁二叉树-递归用\*/  Status ClearBiTree(TNode\* T); /\*清空二叉树\*/  Status BiTreeEmpty(TNode T); /\*判定空二叉树\*/  Status BiTreeDepth(TNode T); /\*求二叉树深度\*/  TNode LocateNode(TNode T, KeyType e); /\*查找结点\*/  Status Assign(TNode\* T, KeyType e, TElemType value); /\*结点赋值\*/  TNode GetSibling(TNode T, KeyType e); /\*获得兄弟结点\*/  TNode GetParent(TNode T, KeyType e); /\*获得父节点\*/  Status InsertNode(TNode\* T, KeyType e, int LR, TNode c); /\*插入结点\*/  Status DeleteNode(TNode\* T, KeyType e); /\*删除结点\*/  Status PreOrderTraverse(TNode T); /\*前序遍历\*/  Status InOrderTraverse(TNode T); /\*中序遍历\*/  Status PostOrderTraverse(TNode T); /\*后序遍历\*/  Status LevelOrderTraverse(TNode T); /\*按层遍历\*/  Status Visit(TNode T); /\*访问结点\*/  void InitQueue(Queue\* q); /\*新建队列\*/  Status EnQueue(Queue\* q, TNode T); /\*入队\*/  TNode OutQueue(Queue\* q); /\*出队\*/  Status SaveTree(TNode T, char\* path); /\*保存树\*/  Status ReSaveTree(TNode T);  Status LoadTree(TNode\* T, char\* path); /\*读取树\*/  void InitStack(stacknode\* s); /\*建立栈\*/  Status push(stacknode\* s ,TNode T); /\*入栈\*/  TNode pop(stacknode\* s); /\*出栈\*/  int main(void) {  int op = 1, LR;  TNode T = NULL, FindNode, BrotherNode, SertNode;  KeyType find, name;  char definition[NUM\_MAX], filename[NUM\_MAX], loadname[NUM\_MAX];  TElemType value;  while (op) {  system("cls");  printf("\n\n-----------------Menu for Linear Table On Binary Tree----------------\n");  printf(" 1. CreateBiTree 9. InsertNode\n");  printf(" 2. DestroyBiTree 10. DeleteNode \n");  printf(" 3. ClearBiTree 11. PreOrderTraverse \n");  printf(" 4. BiTreeEmpty 12. InOrderTraverse \n");  printf(" 5. BiTreeDepth 13. PostOrderTraverse\n");  printf(" 6. LocateNode 14. LevelOrderTraverse\n");  printf(" 7. Assign 15. SaveTree\n");  printf(" 8. GetSibling 16. LoadTree\n");  printf(" 0. Exit\n\n");  printf("-------------------------------------------------------------------------\n");  printf("选择你的操作:\n");  scanf\_s("%d", &op);  getchar();  switch (op)  {  case 1:  T = NULL;  printf("请输入带空子树的前序列：(空结点输入‘#’)\n");  gets\_s(definition);  if (CreateBiTree(&T, definition) == OK) {  printf("二叉树创建成功！\n");  }  else {  printf("二叉树创建失败！\n");  }  getchar();  break;  case 2:  if (isNull == TRUE) {  printf("二叉树不存在！\n");  getchar();  break;  }  if (DestroyBiTree(&T) == OK)  printf("二叉树销毁成功！\n");  else  printf("二叉树销毁失败！\n");  getchar();  break;  case 3:  if (isNull == TRUE) {  printf("二叉树不存在！\n");  getchar();  break;  }  if (ClearBiTree(&T) == TRUE)  printf("二叉树清空成功！\n");  else  printf("二叉树为空！\n");  getchar();  break;  case 4:  if (isNull == TRUE) {  printf("二叉树不存在！\n");  getchar();  break;  }  if (BiTreeEmpty(T) == TRUE)  printf("二叉树为空！\n");  else  printf("二叉树不为空！\n");  getchar();  break;  case 5:  if (isNull == TRUE) {  printf("二叉树不存在！\n");  getchar();  break;  }  if (BiTreeEmpty(T) == TRUE)  printf("二叉树为空树，深度为0.\n");  else  printf("二叉树的深度为%d.\n", BiTreeDepth(T));  getchar();  break;  case 6:  if (isNull == TRUE) {  printf("二叉树不存在！\n");  getchar();  break;  }  printf("输入你想查找的元素的关键字：\n");  scanf\_s("%c", &find);  getchar();  FindNode = LocateNode(T, find);  Findnode = NULL;  if (!FindNode)  printf("查找失败！");  else {  printf("查找成功！结点信息如下：\n");  Visit(FindNode);  }  getchar();  break;  case 7:  if (isNull == TRUE) {  printf("二叉树不存在！\n");  getchar();  break;  }  printf("输入你想赋值的元素的关键字：\n");  scanf\_s("%c", &name);  getchar();  printf("输入你想赋予的数值：\n");  scanf\_s("%d", &value);  getchar();  if (Assign(&T, name, value) == OK)  printf("赋值成功！\n");  else  printf("赋值失败！\n");  getchar();  break;  case 8:  if (isNull == TRUE) {  printf("二叉树不存在！\n");  getchar();  break;  }  printf("输入你想查找兄弟结点的元素的关键字：\n");  scanf\_s("%c", &find);  getchar();  if (BiTreeEmpty(T) == TRUE) {  printf("二叉树为空！");  getchar();  break;  }  if (find == T->data.key) {  printf("查找失败！该结点为树的根结点。\n");  getchar();  break;  }  if (LocateNode(T, find) == NULL) {  Findnode = NULL;  printf("找不到该结点！\n");  getchar();  break;  }  BrotherNode = GetSibling(T, find);  Brothernode = NULL;  if (BrotherNode) {  printf("查找成功！该结点的兄弟结点信息如下：\n");  Visit(BrotherNode);  }  else  printf("查找失败！");  getchar();  break;  case 9:  if (isNull == TRUE) {  printf("二叉树不存在！\n");  getchar();  break;  }  printf("输入你想新建的元素的关键字：\n");  scanf\_s("%c", &name);  getchar();  printf("输入它对应的数值：\n");  scanf\_s("%d", &value);  getchar();  printf("输入你想插入的位置元素的关键字：\n");  scanf\_s("%c", &find);  getchar();  printf("你想插入到该元素的左结点还是右结点？(左0右1)：\n");  scanf\_s("%d", &LR);  getchar();  if (LocateNode(T, name) != NULL) {  Findnode = NULL;  printf("该关键字已有！\n");  getchar();  break;  }  SertNode = (TNode)malloc(sizeof(BiTNode));  SertNode->data = { name,value };  SertNode->lchild = NULL;  SertNode->rchild = NULL;  if (LocateNode(T, find) != NULL) {  Findnode = NULL;  if (InsertNode(&T, find, LR, SertNode) == OK) {  printf("插入结点成功！\n");  }  else  printf("插入失败！\n");  }  else  printf("插入失败！\n");  getchar();  break;  case 10:  if (isNull == TRUE) {  printf("二叉树不存在！\n");  getchar();  break;  }  printf("输入你想删除的元素的关键字：\n");  scanf\_s("%c", &name);  getchar();  if (LocateNode(T, name) != NULL) {  FindNode = NULL;  if (DeleteNode(&T, name) == OK)  printf("删除结点成功！\n");  else  printf("删除失败！\n");  }  else  printf("删除失败！\n");  getchar();  break;  case 11:  if (isNull == TRUE) {  printf("二叉树不存在！\n");  getchar();  break;  }  PreOrderTraverse(T);  getchar();  break;  case 12:  if (isNull == TRUE) {  printf("二叉树不存在！\n");  getchar();  break;  }  InOrderTraverse(T);  getchar();  break;  case 13:  if (isNull == TRUE) {  printf("二叉树不存在！\n");  getchar();  break;  }  PostOrderTraverse(T);  getchar();  break;  case 14:  if (isNull == TRUE) {  printf("二叉树不存在！\n");  getchar();  break;  }  LevelOrderTraverse(T);  getchar();  break;  case 15:  if (isNull == TRUE) {  printf("二叉树不存在！\n");  getchar();  break;  }  printf("请输入想存储的文件名：\n");  scanf\_s("%s", filename, NUM\_MAX);  getchar();  if (SaveTree(T, filename) == INFEASIBLE)  printf("保存失败！\n");  if (SaveTree(T, filename) == ERROR)  printf("保存失败！\n");  else  printf("保存成功！\n");  getchar();  break;  case 16:  printf("请输入想读取的文件名：\n");  scanf\_s("%s", loadname, NUM\_MAX);  getchar();  if (LoadTree(&T, loadname) == INFEASIBLE)  printf("读取失败！\n");  else  printf("读取成功！\n");  getchar();  break;  case 0:  break;  }  }  printf("感谢使用本系统！\n");  return 0;  }  /\*case1: 构建一个新的二叉树\*/  Status CreateBiTree(TNode\* T, char\* definition) {  isNull = FALSE;  i++;  if (definition[i] != '\0') {  if (definition[i] == '#') {  \*T = NULL;  return ERROR;  }  else {  \*T = (TNode)malloc(sizeof(BiTNode));  if ((\*T) == NULL) exit(OVERFLOW);  (\*T)->data.key = definition[i];  (\*T)->data.value = 0;  CreateBiTree(&((\*T)->lchild), definition);  CreateBiTree(&((\*T)->rchild), definition);  }  }  return OK;  }  /\*case2: 销毁二叉树\*/  Status DestroyBiTree(TNode\* T) {  TNode temp;  temp = \*T;  if (!temp) return ERROR;  ReDestroyBiTree(temp->lchild);  ReDestroyBiTree(temp->rchild);  free(temp);  T = NULL;  isNull = TRUE;  return OK;  }  Status ReDestroyBiTree(TNode t) {  if (!t) return ERROR;  ReDestroyBiTree(t->lchild);  ReDestroyBiTree(t->rchild);  free(t);  }  /\*case3: 清空二叉树\*/  Status ClearBiTree(TNode\* T) {  if (!T) return ERROR;  if ((\*T)->data.key == '#') return ERROR;  TNode temp;  temp = \*T;  ReDestroyBiTree(temp->lchild);  ReDestroyBiTree(temp->rchild);  free(temp);  (\*T)->data.key='#';  return OK;  }  /\*case4: 判断二叉树是否为空\*/  Status BiTreeEmpty(TNode T) {  if (T->data.key == '#') return TRUE;  else return FALSE;  }  /\*case5：判断二叉树的深度\*/  Status BiTreeDepth(TNode T) {  if (!T) return ERROR;  int L = 0, R = 0;  L = BiTreeDepth(T->lchild);  R = BiTreeDepth(T->rchild);  return R > L ? R + 1 : L + 1;  }  /\*case6: 找到元素所在结点\*/  TNode LocateNode(TNode T, KeyType e) {  if (!T) return Findnode;  if (T->data.key == e)  Findnode = T;  else {  LocateNode(T->lchild, e);  LocateNode(T->rchild, e);  }  return Findnode;  }  /\*case7: 给二叉树中的结点赋值\*/  Status Assign(TNode\* T, KeyType e, TElemType value) {  if (!T) return ERROR;  TNode\* cue;  cue = (TNode\*)malloc(sizeof(BiTNode));  if (!cue) exit(OVERFLOW);  \*cue = LocateNode(\*T, e);  Findnode = NULL;  if (\*cue == NULL) {  printf("未找到结点。\n");  return ERROR;  }  else {  (\*cue)->data.value = value;  return OK;  }  }  /\*case8: 获得兄弟结点\*/  TNode GetSibling(TNode T, KeyType e) {  if (!T) return Brothernode;  if (T->lchild && T->lchild->data.key == e) {  if (T->rchild)  Brothernode = T->rchild;  else {  printf("该结点没有兄弟结点。");  return Brothernode;  }  }  else if (T->rchild && T->rchild->data.key == e) {  if (T->lchild)  Brothernode = T->lchild;  else {  printf("该结点没有兄弟结点。");  return Brothernode;  }  }  GetSibling(T->lchild, e);  GetSibling(T->rchild, e);  return Brothernode;  }  /\*获得父结点\*/  TNode GetParent(TNode T, KeyType e) {  if (!T) return Parentnode;  if (T->lchild && T->lchild->data.key == e)  Parentnode = T;  if (T->rchild && T->rchild->data.key == e)  Parentnode = T;  GetParent(T->lchild, e);  GetParent(T->rchild, e);  return Parentnode;  }  /\*case 9: 插入结点\*/  Status InsertNode(TNode\* T, KeyType e, int LR, TNode c) {  if (!T) return ERROR;  TNode newnode;  newnode = (TNode)malloc(sizeof(BiTNode));  if (!newnode) exit(OVERFLOW);  newnode = LocateNode(\*T, e);  Findnode = NULL;  if (newnode == NULL) {  printf("未找到关键结点。");  return ERROR;  }  if (LR == 0) {  if (newnode->lchild) c->rchild = newnode->lchild;  else c->rchild = NULL;  newnode->lchild = c;  }  else if (LR == 1) {  if (newnode->rchild) c->rchild = newnode->rchild;  else c->rchild = NULL;  newnode->rchild = c;  }  else {  printf("LR的输入有误。请输入0或1。");  return ERROR;  }  return OK;  }  /\*case 10: 删除结点\*/  Status DeleteNode(TNode\* T, KeyType e) {  if (!(\*T)) return ERROR;  TNode delnode;  int LR = 0; //判断删除结点是父节点的左or右结点  delnode = LocateNode(\*T, e);  Findnode = NULL;  if (delnode == NULL) {  printf("未找到关键结点。");  return ERROR;  }  if (delnode == \*T) { /\*删除根节点\*/  if (delnode->lchild && delnode->rchild) {  TNode temprchild = (\*T)->rchild;  \*T = (\*T)->lchild;  TNode temp = \*T;  while (temp->rchild) {  temp = temp->rchild;  }  temp->rchild = temprchild;  return OK;  }  else if ((!delnode->lchild) && (!delnode->rchild)) {  isNull = TRUE;  \*T = NULL;  printf("该树只有一个根节点，二叉树已被删除！");  return OK;  }  else {  if ((\*T)->lchild)  (\*T) = (\*T)->lchild;  else  (\*T) = (\*T)->rchild;  }  return OK;  }  TNode parentnode = GetParent(\*T, e);  if (parentnode->rchild == delnode) LR = 1;  if (delnode->lchild && delnode->rchild) { /\*结点度数为2\*/  TNode tempnode = delnode->lchild;  if (LR == 0) {  parentnode->lchild = delnode->lchild;  while (tempnode->rchild) {  tempnode = tempnode->rchild;  }  tempnode->rchild = delnode->rchild;  free(delnode);  }  if (LR == 1) {  parentnode->rchild = delnode->lchild;  while (tempnode->rchild) {  tempnode = tempnode->rchild;  }  tempnode->rchild = delnode->rchild;  free(delnode);  }  return OK;  }  else if ((!delnode->lchild) && (!delnode->rchild)) { /\*结点度数为0\*/  if (LR == 0)  parentnode->lchild = NULL;  if (LR == 1)  parentnode->rchild = NULL;  free(delnode);  return OK;  }  else { /\*结点度数为1\*/  if (delnode->lchild) {  if (LR == 0)  parentnode->lchild = delnode->lchild;  else  parentnode->rchild = delnode->lchild;  }  if (delnode->rchild) {  if (LR == 0)  parentnode->lchild = delnode->rchild;  else  parentnode->rchild = delnode->rchild;  }  return OK;  }  }  /\*case 11: 前序遍历\*/  Status PreOrderTraverse(TNode T) {  if (!T) return ERROR;  Visit(T);  PreOrderTraverse(T->lchild);  PreOrderTraverse(T->rchild);  return OK;  }  /\*case 12: 中序遍历-非递归-使用栈实现\*/  Status InOrderTraverse(TNode T) {  if (!T) return ERROR;  stacknode s;  TNode p = T;  InitStack(&s);  while( p || s->top>=0) {  if (p)  {  push(&s, p);  p = p->lchild;  }  else  {  p = pop(&s);  Visit(p);  p = p->rchild;  }  }  return OK;  }  /\*case 13: 后序遍历\*/  Status PostOrderTraverse(TNode T) {  if (!T) return ERROR;  PostOrderTraverse(T->lchild);  PostOrderTraverse(T->rchild);  Visit(T);  return OK;  }  /\*case 14: \*按层遍历\*/  Status LevelOrderTraverse(TNode T) {  if (!T) return ERROR;  Queue q = NULL;  TNode p = T;  InitQueue(&q);  EnQueue(&q, p);  while (q->front != q->rear) {  p = q->data[q->front];  if (p->lchild) EnQueue(&q, p->lchild);  if (p->rchild) EnQueue(&q, p->rchild);  Visit(OutQueue(&q));  }  return OK;  }  void InitQueue(Queue\* q) {  \*q = (Queue)malloc(sizeof(queue));  if ((\*q) == NULL) exit(OVERFLOW);  (\*q)->front = 0;  (\*q)->rear = 0;  }  Status EnQueue(Queue\* lq, TNode ch) { /\*入队\*/  if ((\*lq)->rear >= LIST\_INIT\_SIZE)  return ERROR;  (\*lq)->data[(\*lq)->rear] = ch;  (\*lq)->rear++;  return OK;  }  TNode OutQueue(Queue\* q) { /\*出队\*/  if ((\*q)->front == (\*q)->rear)  return ERROR;  TNode tempnode = (\*q)->data[(\*q)->front];  (\*q)->front++;  return tempnode;  }  Status Visit(TNode T) {  if (T == NULL) return ERROR;  printf("%c : %d ", T->data.key, T->data.value);  return OK;  }  /\*case15：保存树\*/  Status SaveTree(TNode T, char\* path) {  if (!T) return ERROR;  errno\_t err = 0;  err = fopen\_s(&tree, path, "wb");  if (err) {  printf("路径错误。");  return INFEASIBLE;  }  ReSaveTree(T);  fclose(tree);  return OK;  }  Status ReSaveTree(TNode T) {  if (!T) {  fprintf(tree, "%c:%d\n",'#',0);  return ERROR;  }  fprintf(tree, "%c:%d\n", T->data.key, T->data.value);  ReSaveTree(T->lchild);  ReSaveTree(T->rchild);  return OK;  }  /\*case15：读取树\*/  Status LoadTree(TNode\* T, char\* path)  {  errno\_t err = 0;  err = fopen\_s(&tree, path, "rb");  if (err) {  printf("路径错误。");  return INFEASIBLE;  }  fseek(tree, 0, SEEK\_END); /\*求得文件大小\*/  int filesize = ftell(tree);  rewind(tree);  char\* space = (char\*)malloc(filesize);  if (filesize == 0) {  printf("文件为空。");  return ERROR;  }  while (fread(space, filesize, 1, tree))  printf("%s",space);  fclose(tree);  return OK;  }  void InitStack(stacknode\* s) {  \*s = (stacknode)malloc(sizeof(Stack));  if ((\*s) == NULL) exit(OVERFLOW);  (\*s)->top = -1;  }  Status push(stacknode\* s, TNode T) {  if ((\*s)->top > LIST\_INIT\_SIZE) return ERROR;  (\*s)->top++;  (\*s)->data[(\*s)->top] = T;  return OK;  }  TNode pop(stacknode\* s) {  if ((\*s)->top < 0) return ERROR;  TNode outnode = (\*s)->data[(\*s)->top];  (\*s)->top--;  return outnode;  } |

# 附录D 基于邻接表图实现的源程序

|  |
| --- |
| #include <stdio.h>  #include <malloc.h>  #include <stdlib.h>  #include <string.h>  #include <math.h>  #define TRUE 1  #define FALSE 0  #define OK 1  #define ERROR -1  #define INFEASIBLE -1  #define OVERFLOW -2  #define MAX\_VERTEX\_NUM 100  #define LIST\_INIT\_SIZE 100  #define NUM\_MAX 30  typedef int Status;  typedef int ValueType;  typedef int KeyType;  typedef int InfoType;  static bool isNull = TRUE;  static int DeleteNum = 0;  FILE\* graph;  typedef struct VertexType/\*顶点存放的数据的结构类型\*/  {  KeyType key; /\*唯一标识\*/  ValueType value;  }VertexType;  typedef struct ArcNode {  int adjvex; //该弧所指向的顶点的位置  struct ArcNode\* nextarc; //指向下一条弧的指针  InfoType info; //该弧相关信息  }ArcNode;  typedef struct VNode {  VertexType data; //顶点信息  ArcNode\* firstarc; //指向第一条依附该顶点的弧的指针  }VNode, AdjList[MAX\_VERTEX\_NUM];  typedef struct QueueList {  VNode data[LIST\_INIT\_SIZE];  int front;  int rear;  }queue, \* Queue;  typedef struct ALGRAPH{  AdjList vertices;  int vexnum,arcnum; //图的当前节点数和弧数  int book[MAX\_VERTEX\_NUM] = { 0 };  } ALG, \*ALGraph ;  /\*Function defined\*/  Status CreateGraph(ALGraph\* G); /\*创建图\*/  Status DestroyGraph(ALGraph\* G); /\*销毁图\*/  int LocateVex(ALGraph G,KeyType u); /\*查找顶点\*/  Status PutVex(ALGraph\* G,KeyType u,ValueType v); /\*顶点赋值\*/  Status FirstAdjVex(ALGraph G, int index); /\*获得第一邻接点\*/  Status NextAdjVex(ALGraph G, int v, int w); /\*获得下一邻接点\*/  Status InsertVex(ALGraph\* G, VertexType vnode); /\*插入顶点\*/  Status DeleteVex(ALGraph\* G, KeyType vnode); /\*删除顶点\*/  Status InsertArc(ALGraph\* G, KeyType v, KeyType w); /\*插入弧\*/  Status DeleteArc(ALGraph\* G, KeyType v, KeyType w); /\*删除弧\*/  Status DFSTraverse(ALGraph\* G); /\*深度优先搜索遍历\*/  void dfs(ALGraph\* G, int i);  Status BFSTraverse(ALGraph\* G); /\*广度优先搜索遍历\*/  Status Visit(VNode);  void InitQueue(Queue\* q); /\*新建队列\*/  Status EnQueue(Queue\* q, VNode T); /\*入队\*/  VNode OutQueue(Queue\* q); /\*出队\*/  int main(void) {  int op = 1, index, vp, wp;  KeyType find, insert;  ValueType value;  ALGraph G = NULL;  VertexType\* vnode;  char filename[NUM\_MAX], loadname[NUM\_MAX];  while (op) {  system("cls");  printf("\n\n-----------------Menu for Linear Table On Graph----------------\n");  printf(" 1. CreateGraph 7. InsertVex\n");  printf(" 2. DestroyGraph 8. DeleteVex\n");  printf(" 3. LocateVex 9. InsertArc\n");  printf(" 4. PutVex 10. DeleteArc \n");  printf(" 5. FirstAdjVex 11. DFSTraverse\n");  printf(" 6. NextAdjVex 12. BFSTraverse\n");  printf(" 0. Exit\n\n");  printf("---------------------------------------------------------------------\n");  printf("选择你的操作:\n");  scanf\_s("%d", &op);  getchar();  switch (op)  {  case 1:  if (CreateGraph(&G) == OK)  printf("创建成功！\n");  else  printf("创建失败！\n");  getchar();  break;  case 2:  if (isNull == TRUE) {  printf("图不存在！\n");  getchar();  break;  }  if (DestroyGraph(&G) == OK)  printf("销毁成功！\n");  else  printf("销毁失败！\n");  getchar();  break;  case 3:  if (isNull == TRUE) {  printf("图不存在！\n");  getchar();  break;  }  printf("请输入你想查找的顶点的关键字：");  scanf\_s("%d", &find);  getchar();  index = LocateVex(G, find);  if (index != ERROR) {  printf("顶点信息为： ");  Visit(G->vertices[index]);  printf("顶点在邻接表内第%d位.\n",index);  }  else {  printf("查找失败！");  }  getchar();  break;  case 4:  if (isNull == TRUE) {  printf("图不存在！\n");  getchar();  break;  }  printf("请输入你想赋值的顶点的关键字：");  scanf\_s("%d", &find);  getchar();  index = LocateVex(G, find);  if (index != ERROR) {  printf("请输入你想赋的值：");  scanf\_s("%d", &value);  getchar();  }  if (PutVex(&G, find, value) == OK)  printf("赋值成功！\n");  else  printf("赋值失败！\n");  getchar();  break;  case 5:  if (isNull == TRUE) {  printf("图不存在！\n");  getchar();  break;  }  printf("请输入你想查找的顶点的位序：");  scanf\_s("%d", &index);  getchar();  if (index > G->vexnum + DeleteNum || index < 0) {  printf("输入位序不符合要求！\n");  getchar();  break;  }  if (FirstAdjVex(G, index) != ERROR)  printf("该顶点的第一个临接顶点为： %d！\n", FirstAdjVex(G, index));  else  printf("查找失败！\n");  getchar();  break;  case 6:  if (isNull == TRUE) {  printf("图不存在！\n");  getchar();  break;  }  printf("请输入你想查找的顶点v的位序：");  scanf\_s("%d", &vp);  getchar();  printf("请输入你想查找的顶点w的位序：");  scanf\_s("%d", &wp);  getchar();  if (vp == wp || vp > G->vexnum + DeleteNum || wp > G->vexnum + DeleteNum) {  printf("输入有误，查找失败。\n");  }  else if (NextAdjVex(G,vp,wp) != ERROR) {  printf("v相对于w的下一个临界顶点的位置为； %d！\n", NextAdjVex(G, vp, wp));  }  else  printf("查找失败！\n");  getchar();  break;  case 7:  if (isNull == TRUE) {  printf("图不存在！\n");  getchar();  break;  }  printf("请输入你想插入的顶点的关键字：");  scanf\_s("%d", &insert);  getchar();  printf("请输入你想插入的顶点的值：");  scanf\_s("%d", &value);  getchar();  vnode = (VertexType\*)malloc(sizeof(VertexType));  if (!vnode) exit(OVERFLOW);  vnode->key = insert;  vnode->value = value;  if (InsertVex(&G, \*vnode) == OK)  printf("插入成功！\n");  else  printf("插入失败！\n");  getchar();  break;  case 8:  if (isNull == TRUE) {  printf("图不存在！\n");  getchar();  break;  }  printf("请输入你想删除的顶点的关键字：");  scanf\_s("%d", &find);  getchar();  if (DeleteVex(&G, find) == OK)  printf("删除成功！\n");  else  printf("删除失败！\n");  getchar();  break;  case 9:  if (isNull == TRUE) {  printf("图不存在！\n");  getchar();  break;  }  printf("请输入你想插入的弧尾的关键字：");  scanf\_s("%d", &vp);  getchar();  printf("请输入你想插入的弧头的关键字：");  scanf\_s("%d", &wp);  getchar();  if (InsertArc(&G, vp, wp) == OK)  printf("插入成功！\n");  else  printf("插入失败！\n");  getchar();  break;  case 10:  if (isNull == TRUE) {  printf("图不存在！\n");  getchar();  break;  }  printf("请输入你想删除的弧尾的关键字：");  scanf\_s("%d", &vp);  getchar();  printf("请输入你想删除的弧头的关键字：");  scanf\_s("%d", &wp);  getchar();  if (DeleteArc(&G,vp,wp) == OK)  printf("删除成功！\n");  else  printf("删除失败！\n");  getchar();  break;  case 11:  if (isNull == TRUE) {  printf("图不存在！\n");  getchar();  break;  }  printf("对图进行深度优先搜索遍历如下：\n");  DFSTraverse(&G);  getchar();  break;  case 12:  if (isNull == TRUE) {  printf("图不存在！\n");  getchar();  break;  }  printf("对图进行广度优先搜索遍历如下：\n");  BFSTraverse(&G);  getchar();  break;  case 0:  break;  }  }  printf("感谢使用本系统！\n");  return 0;  }  /\*case1: 创建图\*/  Status CreateGraph(ALGraph\* G) {  KeyType key;  ValueType value;  VNode\* vnode = (VNode\*)malloc(sizeof(VNode));  isNull = FALSE;  \*G = (ALGraph)malloc(sizeof(ALG));  if (!G) exit(OVERFLOW);  (\*G)->vexnum = 0;  (\*G)->arcnum = 0;  printf("请输入顶点的key值（唯一）：");  scanf\_s("%d", &key);  getchar();  printf("请为该顶点赋值：");  scanf\_s("%d", &value);  getchar();  if (key) {  vnode->data.key = key;  vnode->data.value = value;  vnode->firstarc = NULL;  }  (\*G)->vertices[(\*G)->vexnum] = \*vnode;  (\*G)->vexnum++;  return OK;  }  /\*case2: 销毁图\*/  Status DestroyGraph(ALGraph\* G) {  if (!G) return ERROR;  isNull = TRUE;  VNode\* vnode;  ArcNode\* del,\*temp=NULL;  for (int i = 0; i < (\*G)->vexnum + DeleteNum; i++) {  del = (\*G)->vertices[i].firstarc;  while (del != NULL) {  temp = del->nextarc;  free(del);  del = temp;  }  }  \*G = NULL;  return OK;  }  /\*case3: 查找顶点\*/  int LocateVex(ALGraph G, KeyType u) {  for (int i = 0; i < (\*G).vexnum + DeleteNum; i++) {  if ((\*G).vertices[i].data.key == u) {  return i;  }  }  return ERROR;  }  /\*case4: 顶点赋值\*/  Status PutVex(ALGraph\* G, KeyType u, ValueType v) {  if (!G) return ERROR;  for (int i = 0; i < (\*G)->vexnum + DeleteNum; i++) {  if ((\*G)->vertices[i].data.key == u) {  (\*G)->vertices[i].data.value = v;  return OK;  }  }  return ERROR;  }  /\*case5：获得第一邻接点\*/  Status FirstAdjVex(ALGraph G, int index) {  ArcNode\* arc = NULL;  if (index < (\*G).vexnum + DeleteNum) {  if ((\*G).vertices[index].firstarc) {  arc = (\*G).vertices[index].firstarc;  return arc->adjvex;  }  }  return ERROR;  }  /\*case6: 获得下一邻接点\*/  Status NextAdjVex(ALGraph G, int v, int w) {  ArcNode\* arc = NULL;  if (v == w) return ERROR;  if (v < (\*G).vexnum + DeleteNum && w < (\*G).vexnum + DeleteNum) {  arc = (\*G).vertices[v].firstarc;  while (arc) {  if (arc->adjvex == w && arc->nextarc)  return arc->nextarc->adjvex;  arc = arc->nextarc;  }  }  return ERROR;  }  /\*case7: 插入顶点\*/  Status InsertVex(ALGraph\* G, VertexType vnode) {  if (!G) return ERROR;  if (LocateVex(\*G, vnode.key) != ERROR) {  printf("该关键字已存在！");  return ERROR;  }  (\*G)->vertices[(\*G)->vexnum].data = vnode;  (\*G)->vertices[(\*G)->vexnum].firstarc = NULL;  (\*G)->vexnum++;  return OK;  }  /\*case8: 删除顶点\*/  Status DeleteVex(ALGraph\* G, KeyType vnode) {  if (!G) return ERROR;  ArcNode\* temp;  int num = 0;  if (LocateVex(\*G, vnode) == ERROR) {  printf("找不到该顶点！");  return ERROR;  }  for (int i = 0; i < (\*G)->vexnum + DeleteNum; i++) {  temp = (\*G)->vertices[i].firstarc;  if (!temp) continue;  if ((\*G)->vertices[i].data.key == vnode) {//删顶点  (\*G)->vertices[i].data.key = 0;  (\*G)->vexnum--;  while (temp) {  num++;  temp = temp->nextarc;  }  (\*G)->vertices[i].firstarc = NULL;  (\*G)->arcnum = (\*G)->arcnum - num;  num = 0;  }  else { //删除弧  if ( temp->adjvex == LocateVex(\*G,vnode)) {  (\*G)->vertices[i].firstarc = temp->nextarc;  (\*G)->arcnum--;  }  while (temp->nextarc) {  if (temp->nextarc->adjvex == LocateVex(\*G, vnode)) {  temp->nextarc = temp->nextarc->nextarc;  (\*G)->arcnum--;  }  temp = temp->nextarc;  }  }    }  DeleteNum++;  return OK;  }  /\*case 9: 插入弧\*/  Status InsertArc(ALGraph\* G, KeyType v, KeyType w) {  if (!G) return ERROR;  int insert;  ArcNode\* temp,\*p;  if (LocateVex(\*G, v)!=ERROR && LocateVex(\*G, w)!=ERROR) {  insert = LocateVex(\*G, v);  temp = (\*G)->vertices[insert].firstarc;  if (!temp) {  (\*G)->vertices[insert].firstarc = (ArcNode\*)malloc(sizeof(ArcNode));  (\*G)->vertices[insert].firstarc->adjvex = LocateVex(\*G, w);  (\*G)->vertices[insert].firstarc->info = 0;  (\*G)->vertices[insert].firstarc->nextarc = NULL;  (\*G)->arcnum++;  return OK;  }  while (temp->nextarc) {  if ((\*G)->vertices[temp->adjvex].data.key == w) {  printf("该弧已存在。 ");  return ERROR;  }  temp = temp->nextarc;  }  if ((\*G)->vertices[temp->adjvex].data.key == w) {  printf("该弧已存在。 ");  return ERROR;  }  p = temp;  temp = temp->nextarc;  temp = (ArcNode\*)malloc(sizeof(ArcNode));  temp->adjvex = LocateVex(\*G, w);  temp->info = 0;  temp->nextarc = NULL;  p->nextarc = temp;  (\*G)->arcnum++;  return OK;  }  return ERROR;  }  /\*case 10: 删除弧\*/  Status DeleteArc(ALGraph\* G, KeyType v, KeyType w) {  if (!G) return ERROR;  ArcNode\* temp;  int indexv,indexw;  if (LocateVex(\*G, v)!=ERROR && LocateVex(\*G, w)!=ERROR) {  indexv = LocateVex(\*G, v);  indexw = LocateVex(\*G, w);  temp = (\*G)->vertices[indexv].firstarc;  if (!temp) {  printf("该顶点不是弧尾！");  return ERROR;  }  if (temp && temp->adjvex == indexw) {  (\*G)->vertices[indexv].firstarc = temp->nextarc;  (\*G)->arcnum--;  return OK;  }  while (temp->nextarc) {  if (temp->nextarc->adjvex == indexw) {  temp->nextarc = temp->nextarc->nextarc;  (\*G)->arcnum--;  return OK;  }  temp = temp->nextarc;  }  printf("不存在从v指向w的弧！");  }  return ERROR;  }  /\*case 11: 深度优先搜索遍历\*/  Status DFSTraverse(ALGraph\* G) {  if (!G) return ERROR;  int i = 0;  for (int j = 0; j < MAX\_VERTEX\_NUM; j++) (\*G)->book[j] = 0 ;  for (i = 0; i < (\*G)->vexnum + DeleteNum; i++) {  if (!(\*G)->book[i])  dfs(G, i);  }  return OK;  }  void dfs(ALGraph\* G, int i) {  ArcNode\* a;  (\*G)->book[i] = 1;  Visit((\*G)->vertices[i]);  a = (\*G)->vertices[i].firstarc;  while (a) {  if ((\*G)->book[a->adjvex] == 0)  {  dfs(G, a->adjvex);  }  a = a->nextarc;  }  }  /\*case 12: 广度优先搜索遍历\*/  Status BFSTraverse(ALGraph\* G) {  if (!G) return ERROR;  Queue q;  ArcNode\* arc;  int i = 0;  VNode v = (\*G)->vertices[0];  for (int j = 0; j < MAX\_VERTEX\_NUM; j++) (\*G)->book[j] = 0;  InitQueue(&q);  EnQueue(&q, v);  while (q->front != q->rear) {  v = OutQueue(&q);  if ((\*G)->book[LocateVex(\*G, v.data.key)] == 0) {  Visit(v);  (\*G)->book[LocateVex(\*G, v.data.key)] = 1;  }  arc = v.firstarc;  while (arc) {  if ((\*G)->book[arc->adjvex] == 0) {  Visit((\*G)->vertices[arc->adjvex]);  (\*G)->book[arc->adjvex] = 1;  EnQueue(&q, (\*G)->vertices[arc->adjvex]);  }  arc = arc->nextarc;  }  }  for (i = 0; i < (\*G)->vexnum + DeleteNum; i++) {  if (!(\*G)->book[i])  Visit((\*G)->vertices[i]);  }  return OK;  }  void InitQueue(Queue\* q) {  \*q = (Queue)malloc(sizeof(queue));  if ((\*q) == NULL) exit(OVERFLOW);  (\*q)->front = 0;  (\*q)->rear = 0;  }  Status EnQueue(Queue\* lq, VNode ch) { /\*入队\*/  if ((\*lq)->rear >= LIST\_INIT\_SIZE)  return ERROR;  (\*lq)->data[(\*lq)->rear] = ch;  (\*lq)->rear++;  return OK;  }  VNode OutQueue(Queue\* q) { /\*出队\*/  VNode tempnode = (\*q)->data[(\*q)->front];  (\*q)->front++;  return tempnode;  }  Status Visit(VNode v) {  if ( v.data.key == 0) return ERROR;  printf("%d: %d\n", v.data.key, v.data.value);  return OK;  } |