

**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1703**

**学 号： U201714607**

**姓 名： 钟子琛**

**指导教师： 袁凌**

**报告日期： 2018年 11 月 12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](#_Toc532686743)

[1.1 问题描述 2](#_Toc532686744)

[1.2 系统设计 3](#_Toc532686745)

[1.3 系统实现 5](#_Toc532686746)

[1.4 实验小结 18](#_Toc532686747)

[2基于链式存储结构的线性表实现 19](#_Toc532686748)

[2.1 问题描述 19](#_Toc532686749)

[2.2 系统设计 20](#_Toc532686750)

[2.3 系统实现 22](#_Toc532686751)

[2.4 实验小结 35](#_Toc532686752)

[3基于二叉链表的二叉树实现 36](#_Toc532686753)

[3.1 问题描述 36](#_Toc532686754)

[3.2 系统设计 38](#_Toc532686755)

[3.3 系统实现 40](#_Toc532686756)

[3.4 实验小结 60](#_Toc532686757)

[附录A 基于顺序存储结构线性表实现的源程序 61](#_Toc532686758)

[附录B 基于链式存储结构线性表实现的源程序 70](#_Toc532686759)

[附录C 基于二叉链表的二叉树实现的源程序 80](#_Toc532686760)

# 1 基于顺序存储结构的线性表实现

## 1.1 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。

⑴初始化表：函数名称是InitaList(L)；初始条件是线性表L不存在已存在；操作结果是构造一个空的线性表。

⑵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。

⑶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。

⑷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。

⑸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。

⑹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。

⑺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。

⑻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。

⑼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。

⑽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。

⑾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。

⑿遍历表：函数名称是ListTrabverse(L)，初始条件是线性表L已存在；操作结果是利用循环依次输出表中的每一个元素。

## 1.2 系统设计

**1.2.1 系统总体设计**

（1）通过switch语句选择需要使用的功能，然后调用对应的方法实现相关功能。

（2）再通过while循环语句实现反复使用菜单中的功能。

（3）设置一个选项让用户能够退出该程序。

**1.2.2 算法设计**

（1）InitaList(&L)

设计：分配存储空间，并将length值设为0，listsize值设为预定义的初始存储容量。

复杂度：时间复杂度T(n) = O(1)，空间复杂度S(n) = O(1)。

（2）DestroyList(\*L)

设计：让ele指向NULL，表长及存储容量置为0，全局变量isNull设置为TRUE。

复杂度：时间复杂度T(n) = O(1)，空间复杂度S(n) = O(1)。

（3）ClearList (&L)

设计：将表长设为0。

复杂度：时间复杂度T(n) = O(1)，空间复杂度S(n) = O(1)。

（4）ListEmpty(L)

设计：读取表长length的值，为0则返回TRUE，否则FALSE。

复杂度：时间复杂度T(n) = O(1)，空间复杂度S(n) = O(1)。

（5）ListLength(L)

设计：返回L.length的值。

复杂度：时间复杂度T(n) = O(1)，空间复杂度S(n) = O(1)。

（6）GetElem(L, i, &e)

设计：将L.elem[i - 1]的值赋给e，并返回OK

复杂度：时间复杂度T(n) = O(1)，空间复杂度S(n) = O(1)。

（7）LocateElem(L, e，compare())

设计：遍历顺序表，将与给定元素e满足关系compare的元素的位序返回。 复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（8）PriorElem（L，cur\_e，\*pre\_e）

设计：从头遍历，若当前节点后继值为cur\_e，则将当前结点的元素赋给pre\_e，return OK。若未找到，则返回FALSE。

复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（9）NextElem（L，cur\_e，\*next\_e）

设计：从头遍历，若当前节点值为cur\_e且非表尾，则将后继结点的元素赋给pre\_e，return OK。若未找到，则返回FALSE。

复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（10）ListInsert(&L, i, e)

设计：先判断i值是否符合要求，不符返回ERROR。i值合法，则检查空间大小，若空间不够，增加存储容量。然后将插入位置及之后的元素右移，最后插入e，表长加一。

复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（11）ListDelete(&L, i, &e)

设计：先判断i值是否符合要求，不符返回ERROR。i值合法，则将被删除元素的值赋给e，然后将被删除元素之后的元素左移，最后表长减一。

复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（12）ListTrabverse(L)

设计：遍历顺序表，用循环输出表中的每一个元素。

复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

## 1.3 系统实现

编程环境：

操作系统：WIN 10

IDE：Dev—C++，C语言。

主模块：包含各变量申明、运行各个方法需要的输入与输出以及使用switch函数实现各函数的调用。程序运行主模块图如图1.1所示：

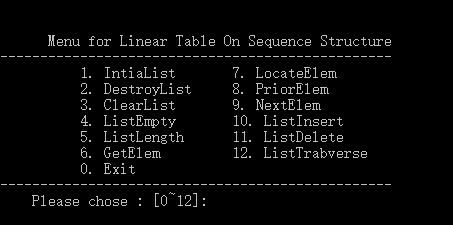


图1.1主模块图

功能模块：

1、初始化表：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,表的长度为0，表的存储容量为10，返回OK，输出Success；否则返回ERROR；如图1.2所示：

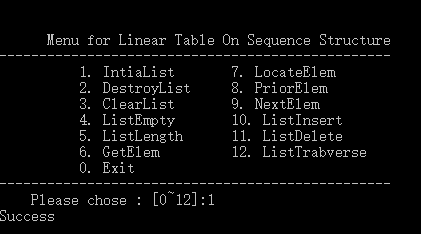


图1.2 初始化表

2、销毁表：释放存储空间，使头地址为空，表长与存储容量置为0，若成功，输出success；如图1.3所示：

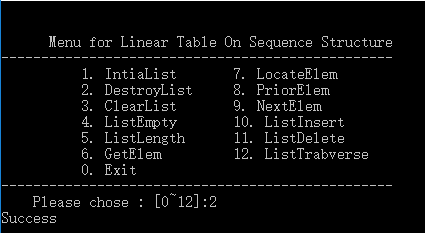


图1.3 销毁表

当表并不存在时，销毁表会失败，如图1.4所示：

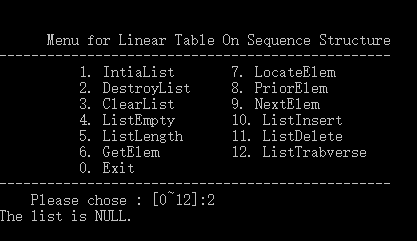


图1.4 销毁表失败

3、清空表，将表长置为0，即将表置为空表。调用插入元素方法、求表长和遍历表方法函数验证功能。如图1.5，1.6，1.7，1.8所示：

（1）新建一个表，插入元素，遍历；

（2）清空表；

（3）再次遍历表；

（4）求表长；



图1.5 新建表

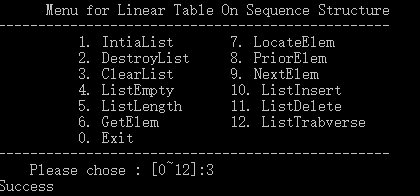


图1.6 清空表

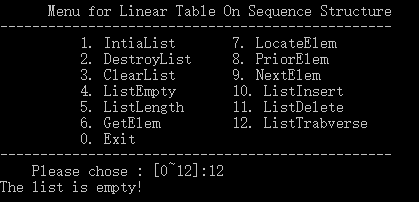


图1.7 遍历表

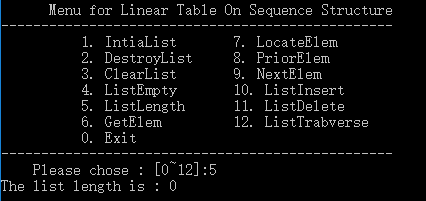


图1.8 求表长

4、判断表空，通过判断表长是否为0来判断表是否为空。使用上面已经初始化的表1，如图1.9，1.10，1.11，1.12所示：

（1）使用之前创建的表；

（2）判断是否为空；

（3）清空表，再次判断是否为空



图1.9 新建表

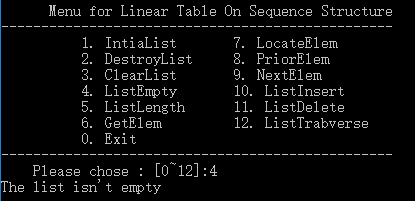


图1.10 判断是否为空

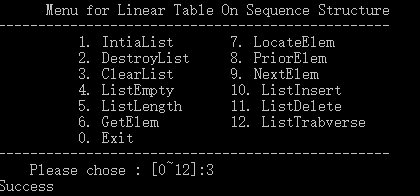


图1.11 清空表

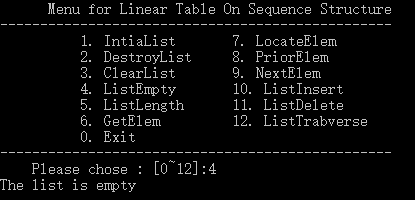


图1.12 判断是否为空

5、求表长，通过循环求当前表中元素的个数，即求表长，如图1.13，1.14所示：

（1）使用之前初始化的表

（2）求表长应该为：5



图1.13 新建表

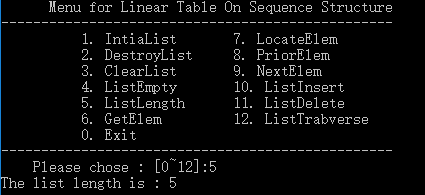


图1.14 求表长

6、获取元素，使用自定义数据类型，将该数据类型变量e作为实参与头地址、所求元素位置一起传入函数，将表内对应位置元素赋值给e，再返回e值，获得所求元素。如图1.15，1.16所示：

（1）使用之前初始化的表；

（2）使用该方法获取对应的元素；



图1.15 新建表

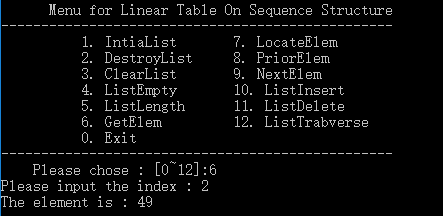


图1.16 获取元素

7、查找元素，将用户需要查找的元素赋值给变量e，传入函数，通过循环遍历查找表中与e相同的第一个元素，返回元素的位序，若找不到则输出” There is not the element.”，如图1.17，1.18，1.19所示：

（1）使用之前初始化的表，添加一个相同的元素

（2）使用该方法获取对应元素的位置；

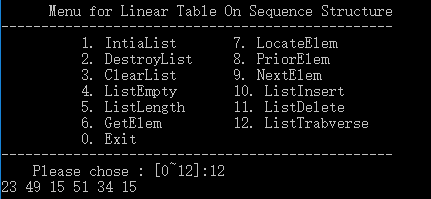


图1.17 新建表

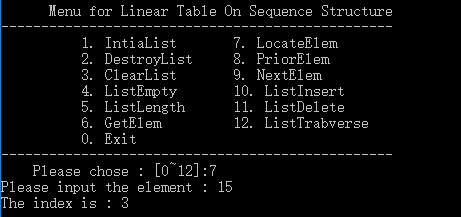


图1.18 搜索元素

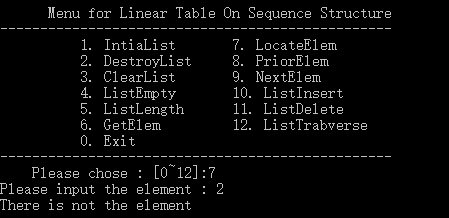


图1.19 搜索不存在元素

8、获得前驱，输入元素，遍历表，若找到该元素且序位不为1，返回该元素的前驱元素；若找不到或者元素位序为1则报错。如图1.20，1.21，1.22所示：

（1）使用之前初始化的表；

（2）寻找非第一个元素的前驱；

（3）寻找第一个元素的前驱；

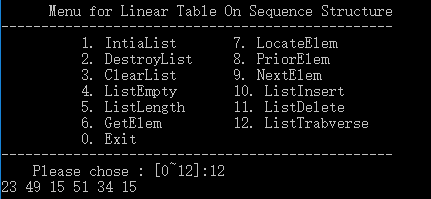


图1.20 新建表

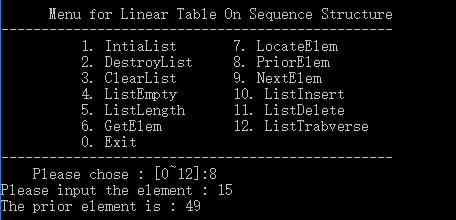


图1.21 找前驱

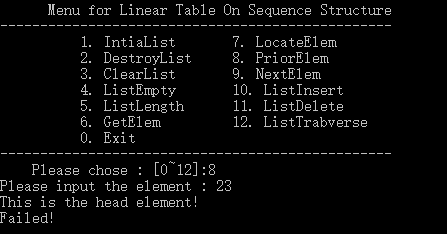


图1.22 找首元素前驱

9、获得后继，输入元素，遍历表，若找到该元素且不为尾元素，返回该元素的后继元素；若找不到或者元素为尾元素则报错。如图1.23，1.24，1.25所示：

（1）使用之前初始化的表；

（2）寻找非第一个元素的后继；

（3）寻找第一个元素的后继；



图1.23 新建表

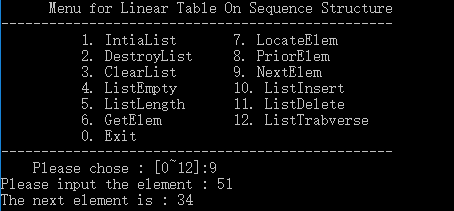


图1.24 获取后继

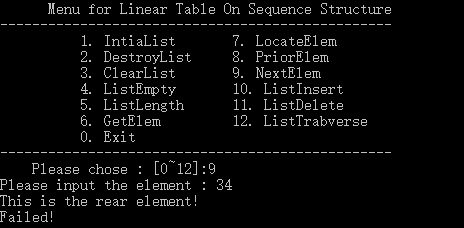


图1.25 获取尾元素后继

10、插入元素，输入插入位置、插入元素与表地址一起传递给函数，若插入位置超出范围则报错，通过循环，将插入点之后的元素全部后移一位，插入元素后，将表长增加1。如图1.26，1.27，1.28，1.29所示：

（1）使用之前初始化的表；

（2）插入一个元素；

（3）遍历表，看插入是否成功；

（4）在超出范围处插入元素；



图1.26 新建表

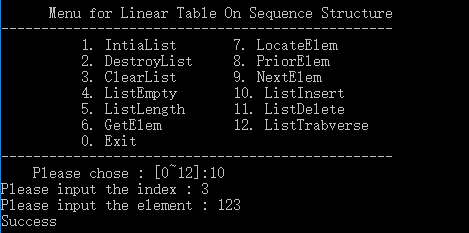


图1.27 插入元素

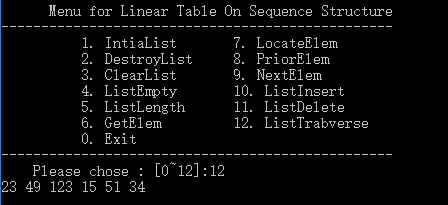


图1.28 遍历表

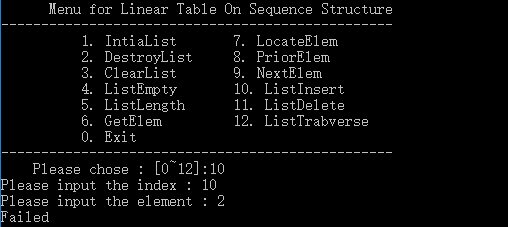


图1.29 插入失败

10、删除元素，输入删除元素的位序与表头地址一起传入函数，若删除位置超出范围则会报错，删除了该位序元素后，将之后的元素全向前移动一位，表长减1，返回被删除的元素。如图1.30，1.31，1.32，1.33所示：

（1）使用之前初始化的表；

（2）删除一个元素；

（3）遍历表，看删除是否成功；

（4）删除超出范围的元素；

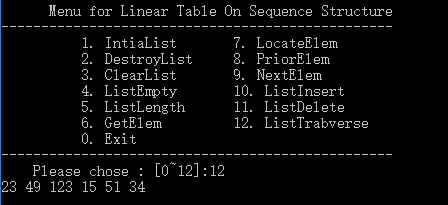


图1.30 初始表

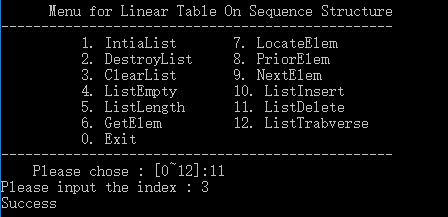


图1.31 删除元素



图1.32 遍历表

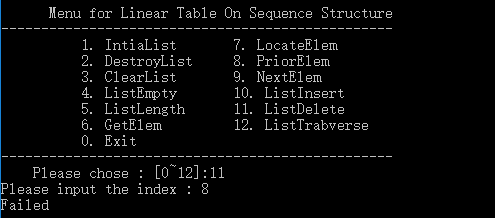


图1.33 删除失败

12、遍历表，通过循环输出表中的所有元素。如图1.34所示：



图1.34 遍历表

## 1.4 实验小结

因为有较好的C语言基础，本次实验总体难度并不是很大，只不过有许多细节需要处理得更好，下面总结一下遇到的问题：

1. 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，后面进行了修改，使得用户明白了自己得错误。
2. 在最初进行销毁表的操作时，并没有真正做到销毁表，对free函数不太理解，后面通过查询资料，理解了free函数的用法，解决了销毁表遇到的问题。

这次实验相当于自己做了一个小的项目，对自己的将来有了更加清楚的认识；回忆了上学期学习的C语言基础知识，巩固了这学期学习的数据结构知识，总的来说，有很大收获。

# 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。

⑴初始化表：函数名称是InitaList(L)；初始条件是线性表L不存在已存在；操作结果是构造一个空的线性表。

⑵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。

⑶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。

⑷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。

⑸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。

⑹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。

⑺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。

⑻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。

⑼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。

⑽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。

⑾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。

⑿遍历表：函数名称是ListTrabverse(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 系统总体设计**

（1）通过switch语句选择需要使用的功能，然后调用对应的方法实现相关功能。

（2）再通过while循环语句实现反复使用菜单中的功能。

（3）设置一个选项让用户能够退出该程序。

**2.2.2 算法设计**

（1）InitaList(&L)

设计：分配存储空间，设置链表头节点的数据域和指针域皆为空。

复杂度：时间复杂度T(n) = O(1)，空间复杂度S(n) = O(1)。

（2）DestroyList(\*L)

设计：将链表的所有节点依次free()，让L指向NULL，全局变量isNull设置为TRUE。

复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（3）ClearList (&L)

设计：将除了头节点以外的节点空间全部释放，头节点指针域指向NULL。

复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（4）ListEmpty(L)

设计：利用循环计算链表中的元素个数，为0则返回TRUE，否则FALSE。

复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（5）ListLength(L)

设计：利用循环计算链表中的元素个数，返回元素个数。

复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（6）GetElem(L, i, &e)

设计：检索链表，循环i - 1次，将对应的节点的数据域的值赋给e，并返回OK。

复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（7）LocateElem(L, e，compare())

设计：遍历链表，将与给定元素e满足关系compare的元素的位序返回。

复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（8）PriorElem（L，cur\_e，\*pre\_e）

设计：从头遍历，若当前节点后继值为cur\_e，则将当前结点的元素赋给pre\_e，return OK。若未找到，则返回FALSE。

复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（9）NextElem（L，cur\_e，\*next\_e）

设计：从头遍历，若当前节点值为cur\_e且非表尾，则将后继结点的元素赋给pre\_e，return OK。若未找到，则返回FALSE。

复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（10）ListInsert(&L, i, e)

设计：先判断i值是否符合要求，不符返回ERROR。i值合法，则检查空间大小，若空间不够，增加存储容量。然后将插入位置之前的节点的指针域指向新节点，新节点的指针域指向原位置的节点。

复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（11）ListDelete(&L, i, &e)

设计：先判断i值是否符合要求，不符返回ERROR。i值合法，则将被删除元素的值赋给e，将该位置的前驱节点的指针域指向该节点的后继节点，释放该节点的空间。

复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（12）ListTrabverse(L)

设计：遍历顺序表，用循环输出链表中的每一个元素。

复杂度：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

## 2.3 系统实现

编程环境：

操作系统：WIN 10

IDE：Dev—C++，C语言。

主模块：包含各变量申明、运行各个方法需要的输入与输出以及使用switch函数实现各函数的调用。程序运行主模块图如图2.1所示：

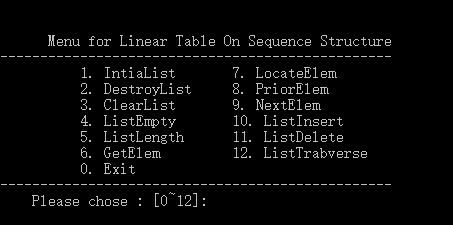


图2.1主模块图

功能模块：

1、初始化表：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，表的存储容量为10，返回OK，输出Success；否则返回ERROR；如图2.2所示：

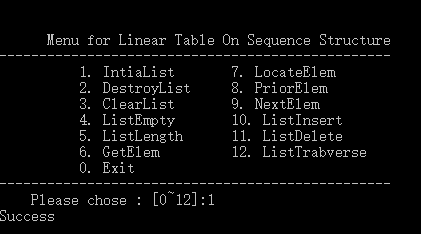


图2.2 初始化表

2、销毁表：释放存储空间，使头地址为空，若成功，输出Success；如图2.3所示：

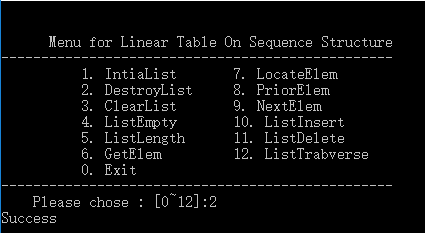


图2.3 销毁表

当表并不存在时，销毁表会失败，如图2.4所示：

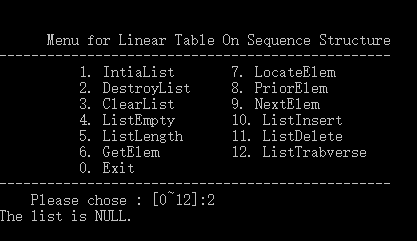


图2.4 销毁表失败

3、清空表，一次释放节点空间。调用插入元素方法、求表长和遍历表方法函数验证功能。如图2.5，2.6，2.7，2.8所示：

（1）新建一个表，插入元素，遍历；

（2）清空表；

（3）再次遍历表；

（4）求表长；



图2.5 新建表

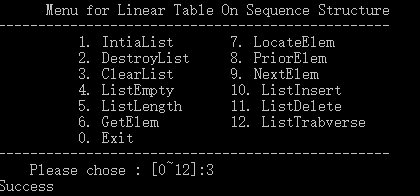


图2.6 清空表

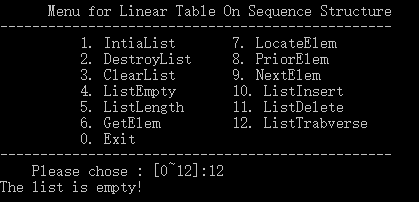


图2.7 遍历表

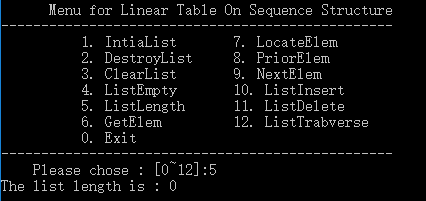


图2.8 求表长

4、判断表空，通过循环获取链表的长度来判断是否为空。使用上面已经初始化的链表，如图2.9，2.10，2.11，2.12所示：

（1）使用之前创建的表；

（2）判断是否为空；

（3）清空表，再次判断是否为空



图2.9 新建表

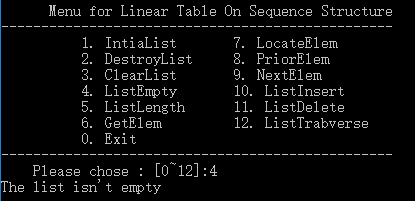


图2.10 判断是否为空

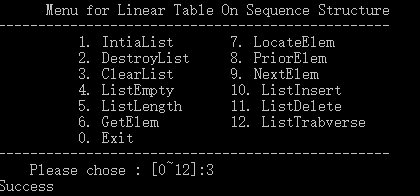


图2.11 清空表

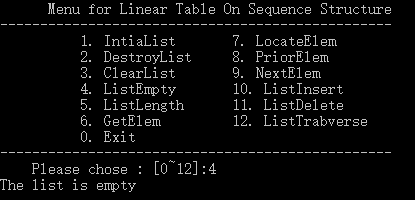


图2.12 判断是否为空

5、求表长，通过循环求当前表中元素的个数，即求表长，如图2.13，2.14所示：

（1）使用之前初始化的表

（2）求表长应该为：5



图2.13 新建表

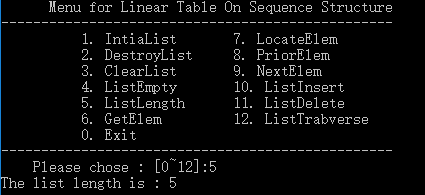


图2.14 求表长

6、获取元素，使用自定义数据类型，将该数据类型变量e作为实参与头地址、所求元素位置一起传入函数，将表内对应位置元素赋值给e，再返回e值，获得所求元素。如图2.15，2.16所示：

（1）使用之前初始化的表；

（2）使用该方法获取对应的元素；



图2.15 新建表

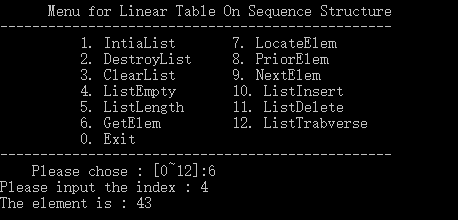


图2.16 获取元素

7、查找元素，将用户需要查找的元素赋值给变量e，传入函数，通过循环遍历查找表中与e相同的第一个元素，返回元素的位序，若找不到则输出” There is not the element.”，如图2.17，2.18，2.19所示：

（1）使用之前初始化的表，添加一个相同的元素

（2）使用该方法获取对应元素的位置；



图2.17 新建表

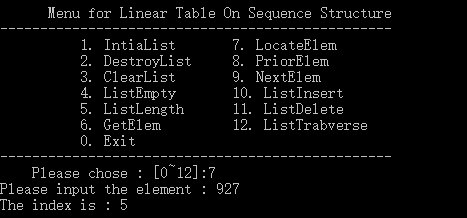


图2.18 搜索元素

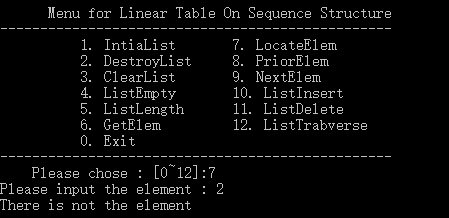


图2.19 搜索不存在元素

8、获得前驱，输入元素，遍历表，若找到该元素且序位不为1，返回该元素的前驱元素；若找不到或者元素位序为1则报错。如图2.20，2.21，2.22所示：

（1）使用之前初始化的表；

（2）寻找非第一个元素的前驱；

（3）寻找第一个元素的前驱；



图2.20 新建表

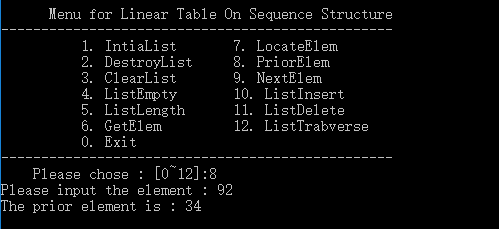


图2.21 找前驱

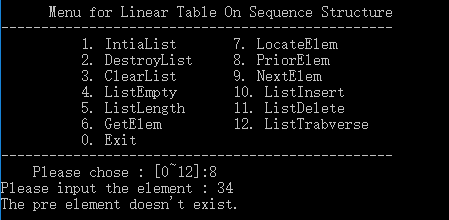


图2.22 找首元素前驱

9、获得后继，输入元素，遍历表，若找到该元素且不为尾元素，返回该元素的后继元素；若找不到或者元素为尾元素则报错。如图2.23，2.24，2.25所示：

（1）使用之前初始化的表；

（2）寻找非第一个元素的后继；

（3）寻找第一个元素的后继；



图2.23 新建表

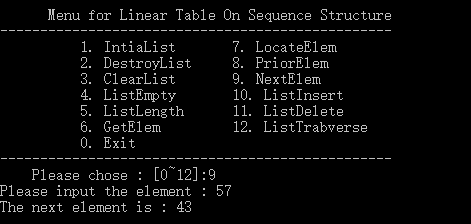


图2.24 获取后继

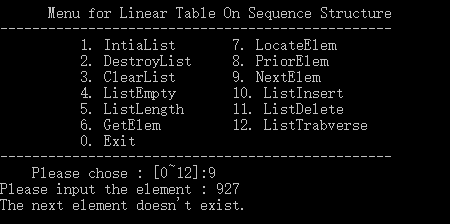


图2.25 获取尾元素后继

10、插入元素，输入插入位置、插入元素与表地址一起传递给函数，若插入位置超出范围则报错。如图2.26，2.27，2.28，2.29所示：

（1）使用之前初始化的表；

（2）插入一个元素；

（3）遍历表，看插入是否成功；

（4）在超出范围处插入元素；



图2.26 新建表

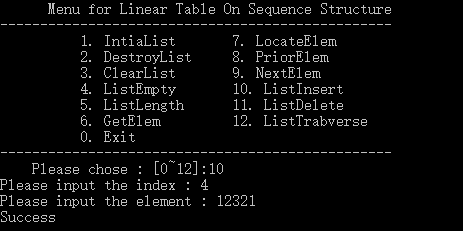


图2.27 插入元素

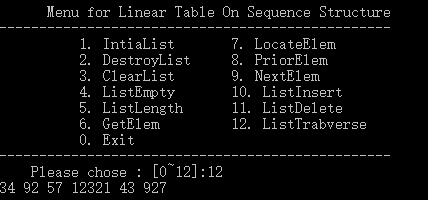


图2.28 遍历表

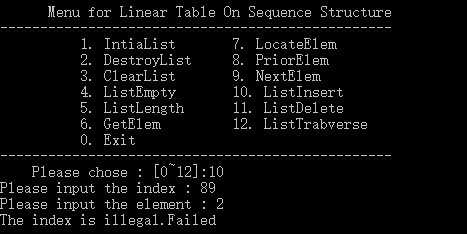


图2.29 插入失败

11、删除元素，输入删除元素的位序与表头地址一起传入函数，若删除位置超出范围则会报错。如图2.30，2.31，2.32，2.33所示：

（1）使用之前的表；

（2）删除一个元素；

（3）遍历表，看删除是否成功；

（4）删除超出范围的元素；

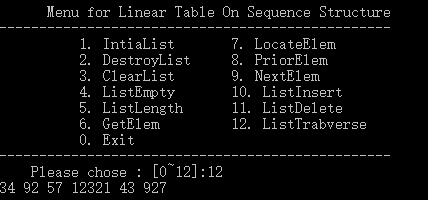


图2.30 初始表

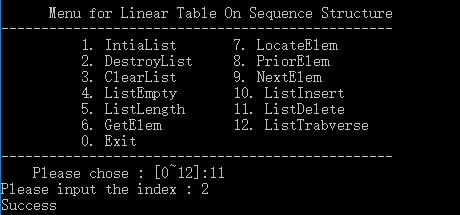


图2.31 删除元素

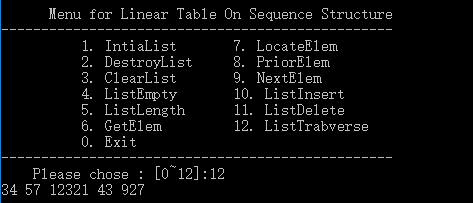


图2.32 遍历表

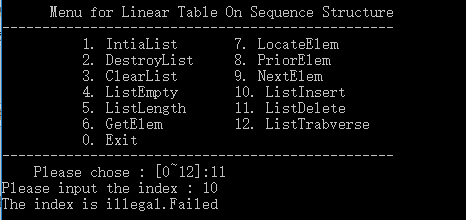


图2.33 删除失败

12、遍历表，通过循环输出表中的所有元素。如图2.34所示：

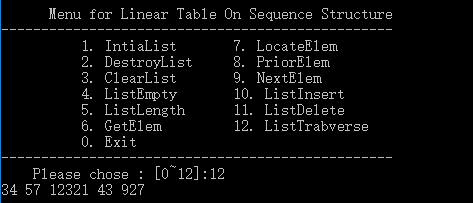


图2.34 遍历表

## 2.4 实验小结

在大一第一学期已经有接触过链表了，而且此次实验内容与第一次的实验答题相同，不同点主要就是数据存储方式的差别，但是只要区分了与顺序表的差别以及熟练掌握的链式表结构，本次实验就不会有太大的问题。

虽然在实验过程中遇到了很多小问题，但是都不是关于数据结构的问题，因此这些问题经过自己的反复调试，都得到了解决。

# 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初始化二叉树、销毁二叉树、创建二叉树、清空二叉树、判定空二叉树和求二叉树深度等20种基本运算，具体运算功能定义如下。

⑴初始化二叉树：函数名称是InitBiTree(T)；初始条件是二叉树T不存在；操作结果是构造空二叉树T。

⑵销毁二叉树：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。

⑶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；操作结果是按definition构造二叉树T。

⑷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。

⑸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。

⑹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。

⑺获得根结点：函数名称是Root(T)；初始条件是二叉树T已存在；操作结果是返回T的根。

⑻获得结点：函数名称是Value(T,e)；初始条件是二叉树T已存在，e是T中的某个结点；操作结果是返回e的值。

⑼结点赋值：函数名称是Assign(T,&e,value)；初始条件是二叉树T已存在，e是T中的某个结点；操作结果是结点e赋值为value。

⑽获得双亲结点：函数名称是Parent(T,e) ；初始条件是二叉树T已存在，e是T中的某个结点；操作结果是若e是T的非根结点，则返回它的双亲结点指针，否则返回NULL。

⑾获得左孩子结点：函数名称是LeftChild(T,e)；初始条件是二叉树T存在，e是T中某个节点；操作结果是返回e的左孩子结点指针。若e无左孩子，则返回NULL。

⑿获得右孩子结点：函数名称是RightChild(T,e)；初始条件是二叉树T已存在，e是T中某个结点；操作结果是返回e的右孩子结点指针。若e无右孩子，则返回NULL。

⒀获得左兄弟结点：函数名称是LeftSibling(T,e)；初始条件是二叉树T存在，e是T中某个结点；操作结果是返回e的左兄弟结点指针。若e是T的左孩子或者无左兄弟，则返回NULL。

⒁获得右兄弟结点：函数名称是RightSibling(T,e)；初始条件是二叉树T已存在，e是T中某个结点；操作结果是返回e的右兄弟结点指针。若e是T的右孩子或者无有兄弟，则返回NULL。

⒂插入子树：函数名称是InsertChild(T,p,LR,c)；初始条件是二叉树T存在，p指向T中的某个结点，LR为0或1，,非空二叉树c与T不相交且右子树为空；操作结果是根据LR为0或者1，插入c为T中p所指结点的左或右子树，p 所指结点的原有左子树或右子树则为c的右子树。

⒃删除子树：函数名称是DeleteChild(T.p.LR)；初始条件是二叉树T存在，p指向T中的某个结点，LR为0或1。 操作结果是根据LR为0或者1，删除c为T中p所指结点的左或右子树。

⒄前序遍历：函数名称是PreOrderTraverse(T)；初始条件是二叉树T存在；操作结果：先序遍历t。

⒅中序遍历：函数名称是InOrderTraverse(T)；初始条件是二叉树T存在；操作结果是中序遍历t。

⒆后序遍历：函数名称是PostOrderTraverse(T)；初始条件是二叉树T存在，操作结果是后序遍历t。

## 3.2 系统设计

**3.2.1 系统总体设计**

（1）通过switch语句选择需要使用的功能，然后调用对应的方法实现相关功能。

（2）再通过while循环语句实现反复使用菜单中的功能。

（3）设置一个选项让用户能够退出该程序。

**3.2.2 方法设计**

（1）InitBiTree(&T)

操作结果：构造空二叉树T。

（2）DestroyBiTree(&T)

初始条件：二叉树T已存在。

操作结果：销毁二叉树T。

（3）CreateBiTree(&T,definition)

初始条件：definition 给出二叉树T的定义。

操作结果：按definition构造二叉树T。

（4）ClearBiTree (&T)

初始条件：二叉树T存在。

操作结果：将二叉树T清空。

（5）BiTreeEmpty(T)

初始条件：二叉树T存在。

操作结果：若T为空二叉树，则返回TRUE,否则返回FALSE.

（6）BiTreeDepth(T)

初始条件：二叉树T存在。

操作结果：返回T的深度。

（7）Root(T)

初始条件：二叉树T已存在。

操作结果：返回T的根。

（8）Value(T，e)

初始条件：二叉树T已存在，e是T中的某个结点。

操作结果：返回e的值。

（9）Assign(T，&e，value)

初始条件：二叉树T已存在，e是T中的某个结点。

操作结果：结点e赋值为value。

（10）Parent(T，e)

初始条件：二叉树T已存在，e是T中的某个结点。

操作结果：若e是T的非根结点，则返回它的双亲结点指针，否则返回NULL。

（11）LeftChild(T，e)

初始条件：二叉树T存在，e是T中某个节点。

操作结果：返回e的左孩子结点指针。若e无左孩子，则返回NULL。

（12）RightChild(T，e)

初始条件：二叉树T已存在，e是T中某个结点。

操作结果：返回e的右孩子结点指针。若e无右孩子，则返回NULL。

（13）LeftSibling(T，e)

初始条件：二叉树T存在，e是T中某个结点。

操作结果：返回e的左兄弟结点指针。若e是T的左孩子或者无左兄弟，则返回NULL。

（14）RightSibling(T，e)

初始条件：二叉树T已存在，e是T中某个结点。

操作结果：返回e的右兄弟结点指针。若e是T的右孩子或者无有兄弟，则返回NULL。

（15）InsertChild(T,p,LR,c)

初始条件：二叉树T存在，p指向T中的某个结点，LR为0或1，,非空二叉树c与T不相交且右子树为空。

操作结果：根据LR为0或者1，插入c为T中p所指结点的左或右子树，p所指结点的原有左子树或右子树则为c的右子树。

（16）DeleteChild(T，p，LR)

初始条件：二叉树T存在，p指向T中的某个结点，LR为0或1。

操作结果：根据LR为0或者1，删除c为T中p所指结点的左或右子树。

（17）PreOrderTraverse（T）

初始条件：二叉树T存在，Visit是对结点操作的应用函数。

操作结果：先序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。

（18）InOrderTraverse（T）

初始条件：二叉树T存在，Visit是对结点操作的应用函数。

操作结果：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。

（19）PostOrderTraverse（T）

初始条件：二叉树T存在，Visit是对结点操作的应用函数。

操作结果：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。

## 3.3 系统实现

编程环境：

操作系统：WIN 10

IDE：Dev—C++，C语言。

主模块：包含各变量申明、运行各个方法需要的输入与输出以及使用switch函数实现各函数的调用。程序运行主模块图如图3.1所示：



图3.1主模块图

功能模块：

1、初始化二叉树：为一棵二叉树分配存储空间，若存储分配成功，二叉树头节点T为存储空间基址，返回OK；否则返回ERROR。如图3.2所示：



图3.2 初始化二叉树

2、销毁二叉树：释放存储空间，使头地址为空。如图3.3所示：



图3.3 销毁二叉树

当表并不存在时，销毁表会失败，如图3.4所示：

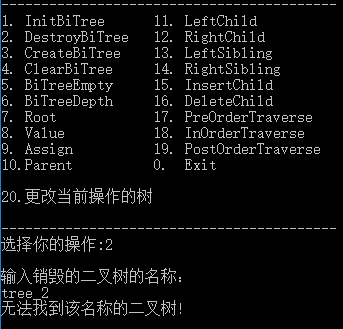


图3.4 销毁二叉树失败

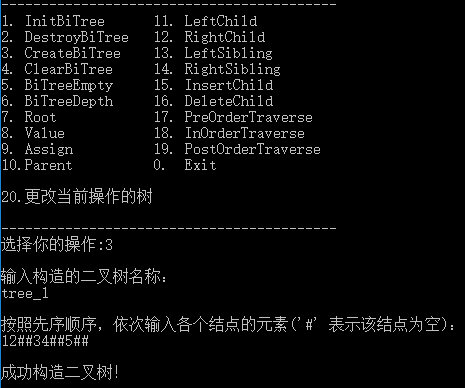
3、创建二叉树，通过用户输入前序遍历来构造二叉树。如图3.5，3.6所示：

图3.5 新建二叉树



图3.6 遍历二叉树

4、清空二叉树，初始条件是二叉树T存在，操作结果是利用循环将二叉树T的每个节点清空，T头节点再指向空。如图3.7，3.8，3.9，3.10所示：

（1）使用之前创建的表；

（2）判断是否为空；

（3）清空表，再次判断是否为空



图3.7 二叉树



图3.8 判断是否为空



图3.9 清空二叉树



图3.10 判断是否为空

5、判断二叉树是否为空，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，如图3.11，3.12，3.13所示：

（1）使用之前初始化的表

（2）清空二叉树，再次遍历



图3.11 二叉树



图3.12 清空二叉树



图3.13 遍历二叉树

6、计算二叉树深度，遍历二叉树，找到二叉树最深的字数节点，返回深度。如图3.14所示：

（1）使用之前的二叉树；

（2）计算深度；

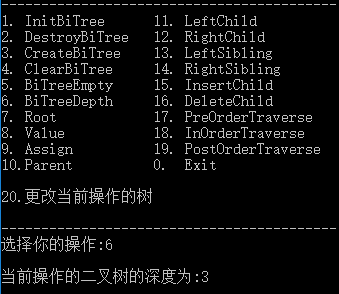


图3.14 计算二叉树深度

7、求二叉树的根，二叉树的根储存在T头结点的左子树上，返回头结点的左子树上的元素。如图3.15所示：

（1）使用之前二叉树；

（2）求二叉树的根；



图3.15 求二叉树的根

8、获得节点的值，初始条件是二叉树T已存在，e是T中的某个结点；操作结果是返回e的值。如图3.16，3.17所示：

（1）使用之前的二叉树，为其中的一个节点赋值；

（2）求这个节点的值；

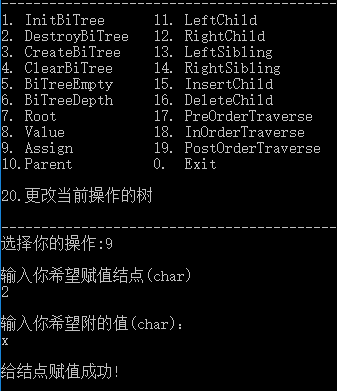


图3.16 为节点赋值

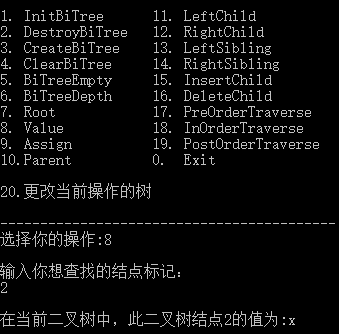


图3.17 找到对应节点的值

9、为节点赋值，e是T中的某个结点；操作结果是结点e赋值为value。如图3.18所示：

（1）使用之前的二叉树；

（2）为其中一个节点赋值；

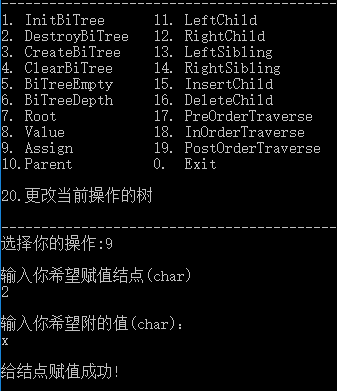


图3.18 为节点赋值

10、求二叉树节点的父节点，初始条件是二叉树T已存在，e是T中的某个结点；操作结果是若e是T的非根结点，则返回它的双亲结点指针，否则返回NULL。如图3.19，3.20所示：

（1）使用之前的二叉树；

（2）获取一个节点的父节点；

（3）获取失败；

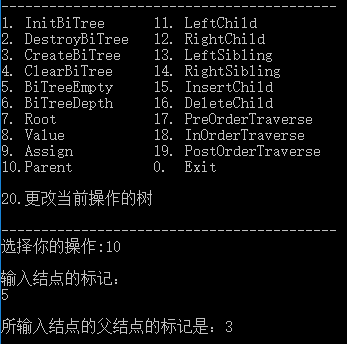


图3.19 新建表

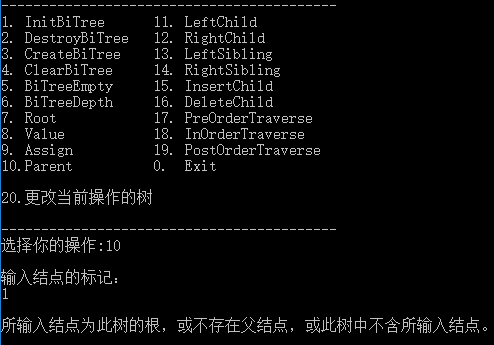


图3.20 获取失败

11、获得左孩子结点，初始条件是二叉树T存在，e是T中某个节点；操作结果是返回e的左孩子结点指针。若e无左孩子，则返回NULL。如图3.21，3.22，3.23所示：

（1）新建一棵树（前序、中序遍历）；

（2）获取某个节点的左孩子；

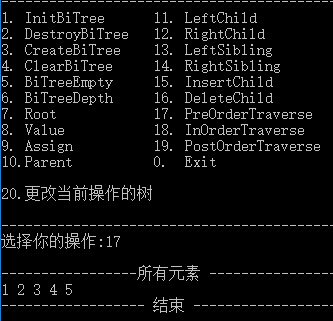


图3.21 前序遍历

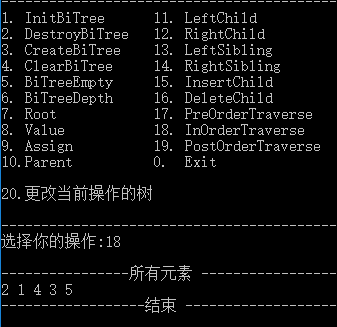


图3.22 中序遍历

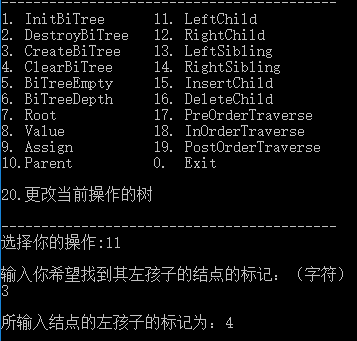


图3.23 左孩子

12、获得右孩子结点，初始条件是二叉树T存在，e是T中某个节点；操作结果是返回e的右孩子结点指针。若e无右孩子，则返回NULL。如图3.24所示：

（1）使用之前的树；

（2）获取某个节点的右孩子；

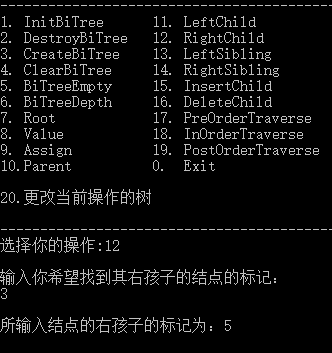


图3.24 右孩子

13、获得左兄弟结点，初始条件是二叉树T存在，e是T中某个结点；操作结果是返回e的左兄弟结点指针。若e是T的左孩子或者无左兄弟，则返回NULL。如图3.25所示：

（1）使用之前的树；

（2）获取某个节点的左兄弟；

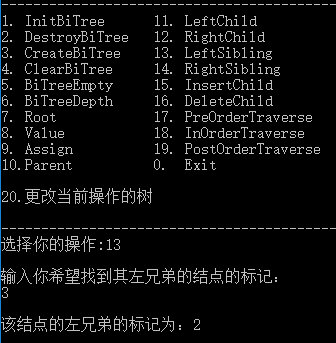


图3.25 左兄弟

14、获得右兄弟结点，初始条件是二叉树T存在，e是T中某个结点；操作结果是返回e的右兄弟结点指针。若e是T的右孩子或者无右兄弟，则返回NULL。如图3.26所示：

（1）使用之前的树；

（2）获取某个节点的右兄弟；

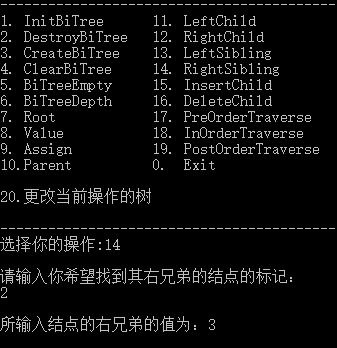


图3.26 右兄弟

15、插入子树，初始条件是二叉树T存在，p指向T中的某个结点，LR为0或1，,非空二叉树c与T不相交且右子树为空；操作结果是根据LR为0或者1，插入c为T中p所指结点的左或右子树，p 所指结点的原有左子树或右子树则为c的右子树。如图3.27，3.28所示：

（1）使用之前的树；

（2）插入一个子树；

（3）遍历；

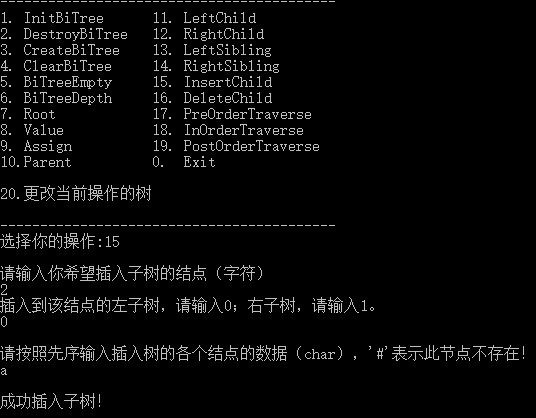


图3.27 插入子树

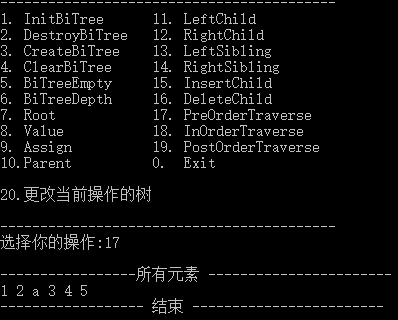


图3.28 遍历

16、删除子树，初始条件是二叉树T存在，p指向T中的某个结点，LR为0或1。操作结果是根据LR为0或者1，删除c为T中p所指结点的左或右子树。如图3.29，3.30所示：

（1）使用之前的树；

（2）删除一个子树；

（3）遍历；

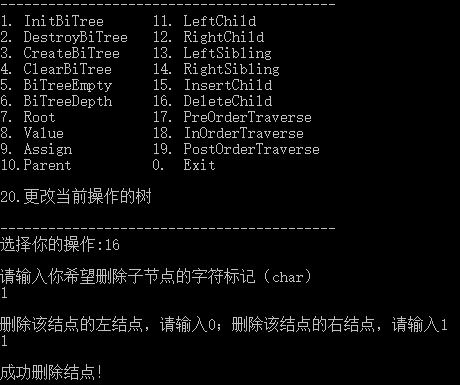


图3.29 删除子树

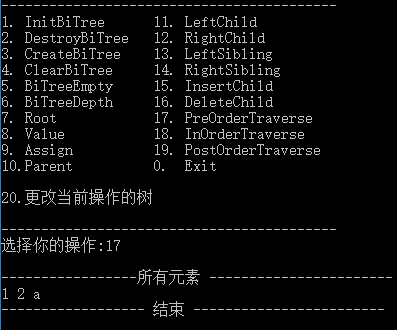


图3.30 遍历

17、前序遍历，输出二叉树的先序遍历。如图3.31所示：

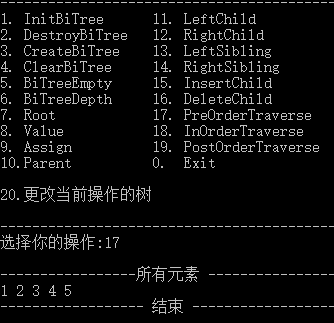


图3.31 先序遍历

18、中序遍历，输出二叉树的中序遍历。如图3.32所示：

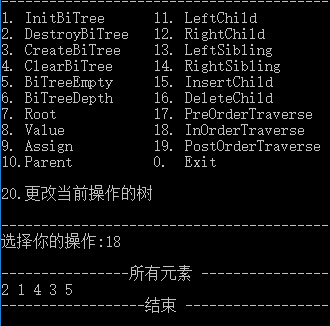


图3.31 中序遍历

19、后序遍历，输出二叉树的后序遍历。如图3.33所示：

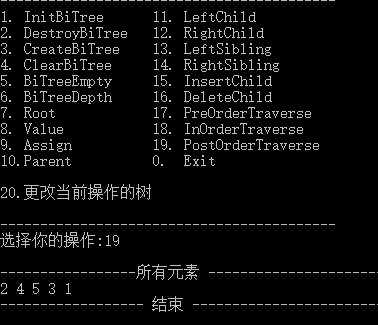


图3.31 后序遍历

## 3.4 实验小结

这是我目前在课程学习中接触到的规模最大的实验，需要实现的内容非常多，但是我通过自己自主学习、查找资料，慢慢的将整个程序写出来了。虽然花的时间很长，但是体会到了以后自己写代码不仅仅需要技术，还需要坚持和耐心。只要这样，再大的工程都能够完成。

附录A 基于顺序存储结构线性表实现的源程序

#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 1

#define LISTINCREMENT 10

typedef struct {

ElemType \* elem;

int length;

int listSize;

} SqList;

status IntiaList(SqList & L);

status DestroyList(SqList \* L);

status ClearList(SqList & L);

status ListEmpty(SqList L);

int ListLength(SqList L);

status GetElem(SqList L,int i,ElemType & e);

status LocateElem(SqList L,ElemType e, status (\*Compare)(ElemType a, ElemType b));

status PriorElem(SqList L,ElemType cur,ElemType \* pre\_e);

status NextElem(SqList L,ElemType cur,ElemType \* next\_e);

status ListInsert(SqList & L,int i,ElemType e);

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

status ListTrabverse(SqList L);

status Compare(ElemType a, ElemType b);

extern bool isNull = TRUE;

int main() {

SqList L;

int op=1;

while(op){

system("cls"); printf("\n\n");

printf(" Menu for Linear Table On Sequence Structure \n");

printf("-------------------------------------------------\n");

printf(" 1. IntiaList 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(" Please chose : [0~12]:");

scanf("%d",&op);

getchar();

switch(op) {

case 1:

if (IntiaList(L) == OK) {

printf("Success\n");

}

else

printf("failed\n");

getchar();

break;

case 2:

if (isNull) {

printf("The list is NULL.");

getchar();

break;

}

if (DestroyList(&L) == OK) {

printf("Success\n");

}

else

printf("Failed\n");

getchar();

break;

case 3:

if (ClearList(L) == OK)

printf("Success\n");

else

printf("Failed\n");

getchar();

break;

case 4:

if (ListEmpty(L) == TRUE)

printf("The list is empty\n");

else if (!isNull)

printf("The list isn't empty\n");

getchar();

break;

case 5:

if (!isNull)

printf("The list length is : %d\n", ListLength(L));

else

ListLength(L);

getchar();

break;

case 6:

int index;

ElemType e;

if (isNull) {

printf("The list is NULL.");

getchar();

break;

}

printf("Please input the index : ");

scanf("%d", &index);

getchar();

if (GetElem(L, index, e) != ERROR)

printf("The element is : %d\n", e);

else

printf("Failed\n");

getchar();

break;

case 7:

if (isNull) {

printf("The list is NULL.");

getchar();

break;

}

printf("Please input the element : ");

scanf("%d", &e);

getchar();

if (LocateElem(L, e, Compare) == ERROR)

printf("There is not the element\n");

else

printf("The index is : %d", LocateElem(L, e, Compare));

getchar();

break;

case 8:

if (isNull) {

printf("The list is NULL.");

getchar();

break;

}

ElemType cue, pre;

printf("Please input the element : ");

scanf("%d", &cue);

getchar();

if (PriorElem(L, cue, &pre) == OK)

printf("The prior element is : %d\n", pre);

else

printf("Failed!\n");

getchar();

break;

case 9:

if (isNull) {

printf("The list is NULL.");

getchar();

break;

}

ElemType next;

printf("Please input the element : ");

scanf("%d", &cue);

getchar();

if (NextElem(L, cue, &next) == OK)

printf("The next element is : %d\n", next);

else

printf("Failed!\n");

getchar();

break;

case 10:

if (isNull) {

printf("The list is NULL.");

getchar();

break;

}

printf("Please input the index : ");

scanf("%d", &index);

getchar();

printf("Please input the element : ");

scanf("%d", &e);

getchar();

if (ListInsert(L, index, e) == OK)

printf("Success\n");

else

printf("Failed\n");

getchar();

break;

case 11:

if (isNull) {

printf("The list is NULL.");

getchar();

break;

}

printf("Please input the index : ");

scanf("%d", &index);

getchar();

if (ListDelete(L, index, &e) == OK)

printf("Success\n");

else

printf("Failed\n");

getchar();

break;

case 12:

if (isNull) {

printf("The list is NULL.");

getchar();

break;

}

ListTrabverse(L);

getchar();

break;

case 0:

break;

}

}

printf("Thank you for using!\n");

getchar();

return 0;

}

status IntiaList(SqList & L) {

L.elem = (ElemType \*)malloc(LIST\_INIT\_SIZE \* sizeof(ElemType));

if (!L.elem)

exit(OVERFLOW);

isNull = FALSE;

L.length = 0;

L.listSize = LIST\_INIT\_SIZE;

return OK;

}

status DestroyList(SqList \* L) {

L->length = 0;

isNull = TRUE;

free(L->elem);

L = NULL;

return OK;

}

status ClearList(SqList & L) {

if (isNull) {

printf("The list is NULL.");

return ERROR;

}

L.length = 0;

return OK;

}

status ListEmpty(SqList L) {

if (isNull) {

printf("The list is NULL.");

return ERROR;

}

if (L.length == 0) {

return TRUE;

}

else {

return ERROR;

}

}

int ListLength(SqList L) {

if (isNull) {

printf("The list is NULL.");

return ERROR;

}

return L.length;

}

status GetElem(SqList L, int i, ElemType & e) {

if (i < 1 || i > ListLength(L))

return ERROR;

e = L.elem[i - 1];

return e;

}

int 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 cue, ElemType \* pre) {

int i;

for (i = 0; i < L.listSize; i++) {

if (L.elem[i] == cue) {

if (i == 0) {

printf("This is the head element!\n");

return ERROR;

}

\*pre = L.elem[i - 1];

return OK;

}

}

return FALSE;

}

status NextElem(SqList L, ElemType cue, ElemType \* next) {

int i;

for (i = 0; i < L.listSize - 1; i++) {

if (L.elem[i] == cue) {

if (i == L.length - 1) {

printf("This is the rear element!\n");

return ERROR;

}

\*next = L.elem[i + 1];

return OK;

}

}

printf("There is not the element!\n");

return FALSE;

}

status ListInsert(SqList & L, int i, ElemType e) {

ElemType \* newbase, \*p, \*q;

if (i < 1 || i > L.length + 1 || &L == NULL)

return ERROR;

if (L.length >= L.listSize) {

newbase = (ElemType \* )realloc(L.elem, (L.listSize + LISTINCREMENT) \* sizeof(ElemType));

if (!newbase)

exit(OVERFLOW);

L.elem = newbase;

L.listSize += LISTINCREMENT;

}

q = &(L.elem[i - 1]);

for (p = &(L.elem[L.length - 1]); p >= q; --p) {

\*(p + 1) = \*p;

}

\*q = e;

++L.length;

return OK;

}

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

ElemType \*p, \*q;

if (i < 1 || i > L.length)

return ERROR;

p = &(L.elem[i - 1]);

e = p;

q = &(L.elem[L.length - 1]);

for (p++; p <= q; p++) {

\*(p - 1) = \*p;

}

--L.length;

return OK;

}

status ListTrabverse(SqList L) {

int i;

if (ListEmpty(L)) {

printf("The list is empty!\n");

return ERROR;

}

for (i = 0; i < L.length; i++) {

printf("%d ", L.elem[i]);

}

return OK;

}

status Compare(ElemType a, ElemType b) {

if (a == b) {

return TRUE;

}

else {

return FALSE;

}

}

附录B 基于链式存储结构线性表实现的源程序

#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 -1

#define OVERFLOW -2

/\* TYPE DEFINE \*/

typedef int status;

typedef int ElemType;

#define LIST\_INIT\_SIZE 10

#define LISTINCREMENT 10

typedef struct Lnode {

ElemType data;

struct Lnode \* next;

}LNode, \*LinkList;

status IntiaList(LinkList \* L);

status DestroyList(LinkList \* L);

status ClearList(LinkList \* L);

status ListEmpty(LinkList L);

int ListLength(LinkList L);

status GetElem(LinkList L, int i, ElemType \* e);

status LocateElem(LinkList L, ElemType e, status (\*Compare)(ElemType a, ElemType b));

status PriorElem(LinkList L, ElemType cur, ElemType \* pre\_e);

status NextElem(LinkList L, ElemType cur, ElemType \* next\_e);

status ListInsert(LinkList \* L, int i, ElemType e);

status ListDelete(LinkList \* L, int i, ElemType \* e);

status ListTrabverse(LinkList L, status (\*Visit)(ElemType e));

status Compare(ElemType a, ElemType b);

status Visit(ElemType e);

extern bool isNULL = true;

int main() {

LinkList L = NULL;

int op=1;

while(op){

system("cls"); printf("\n\n");

printf(" Menu for Linear Table On Sequence Structure \n");

printf("-------------------------------------------------\n");

printf(" 1. IntiaList 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(" Please chose : [0~12]:");

scanf("%d",&op);

getchar();

switch(op) {

case 1:

if (IntiaList(&L) == OK) {

printf("Success\n");

}

else

printf("failed\n");

getchar();

break;

case 2:

if (DestroyList(&L) == OK) {

printf("Success\n");

}

else

printf("Failed\n");

getchar();

break;

case 3:

if (ClearList(&L) == OK)

printf("Success\n");

else

printf("Failed\n");

getchar();

break;

case 4:

ListEmpty(L);

getchar();

break;

case 5:

if (ListLength(L) != ERROR) {

printf ("The length is : %d", ListLength(L));

}

getchar();

break;

case 6:

int index;

ElemType e;

printf("Please input the index : ");

scanf("%d", &index);

getchar();

if (GetElem(L, index, &e) != ERROR)

printf("The element is : %d\n", e);

else

printf("Failed\n");

getchar();

break;

case 7:

printf("Please input the element : ");

scanf("%d", &e);

getchar();

if (LocateElem(L, e, Compare) != ERROR)

printf("The index is : %d", LocateElem(L, e, Compare));

getchar();

break;

case 8:

ElemType cue, pre;

printf("Please input the element : ");

scanf("%d", &cue);

getchar();

if (PriorElem(L, cue, &pre) == OK)

printf("The prior element is : %d\n", pre);

getchar();

break;

case 9:

ElemType next;

printf("Please input the element : ");

scanf("%d", &cue);

getchar();

if (NextElem(L, cue, &next) == OK)

printf("The next element is : %d\n", next);

getchar();

break;

case 10:

printf("Please input the index : ");

scanf("%d", &index);

getchar();

printf("Please input the element : ");

scanf("%d", &e);

getchar();

if (ListInsert(&L, index, e) == OK)

printf("Success\n");

else

printf("Failed\n");

getchar();

break;

case 11:

printf("Please input the index : ");

scanf("%d", &index);

getchar();

if (ListDelete(&L, index, &e) == OK)

printf("Success\n");

else

printf("Failed\n");

getchar();

break;

case 12:

ListTrabverse(L, Visit);

getchar();

break;

case 0:

break;

}

}

printf("Thank you for using!\n");

getchar();

return 0;

}

status IntiaList(LinkList \* L) {

\*L = (LinkList)malloc(sizeof(LNode));

if (\*L == NULL)

exit(OVERFLOW);

(\*L)->data = 0;

(\*L)->next = NULL;

isNULL = false;

return OK;

}

status DestroyList(LinkList \* L) {

if (isNULL) {

printf ("The list is NULL.");

return ERROR;

}

LinkList p, q;

p = \*L;

while (p) {

q = p->next;

free(p);

p = q;

}

\*L = NULL;

isNULL = true;

return OK;

}

status ClearList(LinkList \* L) {

if (isNULL) {

printf ("The list is NULL.");

return ERROR;

}

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 (isNULL) {

printf ("The list is NULL.");

return ERROR;

}

if (L->next == NULL) {

printf ("The list is empty.");

return TRUE;

}

else {

printf ("The list isn't empty.");

return FALSE;

}

}

int ListLength(LinkList L) {

if (isNULL) {

printf ("The list is NULL.");

return ERROR;

}

int i = 0;

LinkList p = L->next;

while (p) {

i++;

p = p->next;

}

return i;

}

status GetElem(LinkList L, int i, ElemType \* e) {

if (isNULL) {

printf ("The list is NULL.");

return ERROR;

}

int times = 1;

LinkList p = L->next;

if (i < 1 || i > ListLength(L)) {

printf ("The index is illegal.");

return ERROR;

}

while (p && times < i) {

p = p->next;

times++;

}

\*e = p->data;

return OK;

}

int LocateElem(LinkList L, ElemType e, status(\*Compare)(ElemType, ElemType)) {

if (isNULL) {

printf ("The list is NULL.");

return ERROR;

}

int index = 0;

LinkList p = L->next;

while (p) {

index++;

if (Compare(p->data, e)) {

return index;

}

p = p->next;

}

printf ("The element is not in the list.");

return ERROR;

}

status PriorElem(LinkList L, ElemType cue, ElemType \* pre) {

if (isNULL) {

printf ("The list is NULL.");

return ERROR;

}

LinkList p = L->next;

while (p->next != NULL && p->next->data != cue) {

p = p->next;

}

if (p->next == NULL) {

printf ("The pre element doesn't exist.");

return ERROR;

}

\*pre = p->data;

return OK;

}

status NextElem(LinkList L, ElemType cue, ElemType \* next) {

if (isNULL) {

printf ("The list is NULL.");

return ERROR;

}

LinkList p = L->next;

while (p->next != NULL && p->data != cue) {

p = p->next;

}

if (p->next == NULL) {

printf ("The next element doesn't exist.");

return ERROR;

}

\*next = p->next->data;

return OK;

}

status ListInsert(LinkList \* L, int i, ElemType e) {

if (isNULL) {

printf ("The list is NULL.");

return ERROR;

}

int j = 1;

LinkList p, newLode;

p = \*L;

while (p != NULL && j < i) {

p = p->next;

j++;

}

if (p == NULL || j > i) {

printf ("The index is illegal.");

return ERROR;

}

newLode = (LinkList)malloc(sizeof(LNode));

if (newLode == NULL) {

exit(OVERFLOW);

}

newLode->data = e;

if (p->next == NULL) {

newLode->next = NULL;

}

else {

newLode->next = p->next;

}

p->next = newLode;

return OK;

}

status ListDelete(LinkList \* L, int i, ElemType \* e) {

if (isNULL) {

printf ("The list is NULL.");

return ERROR;

}

int j = 1;

LinkList p, q;

p = \*L;

while (p != NULL && j < i) {

p = p->next;

j++;

}

if (i > ListLength(\*L)) {

printf ("The index is illegal.");

return ERROR;

}

q = p->next;

p->next = q->next;

\*e = q->data;

free(q);

return OK;

}

status ListTrabverse(LinkList L, status(\*visit)(ElemType e)) {

if (isNULL) {

printf ("The List is NULL.");

return ERROR;

}

if (ListLength(L) == 0) {

printf ("The List is empty.");

return ERROR;

}

LinkList p = L->next;

while (p != NULL) {

if (!(\*visit)(p->data)) {

printf ("Failed");

}

p = p->next;

}

return OK;

}

status Compare(ElemType a, ElemType b) {

if (a == b) {

return TRUE;

}

else {

return FALSE;

}

}

status Visit(ElemType e) {

printf ("%d ", e);

return OK;

}

附录C 基于二叉链表的二叉树实现的源程序

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <string.h>

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

typedef int Status;

typedef char TElemType;

typedef struct ElemType

{

/\*树结点存放的数据的结构类型\*/

char tag;

char number;

int isInit;

}ElemType;

typedef struct BiTNode {

ElemType data;

struct BiTNode \*lchild, \*rchild;

}BiTNode, \*TNode, \*BiTree;

typedef struct Tree {

char name[20];

struct BiTNode \*HeadNode;

}Tree;

typedef struct {

struct Tree \*elem;

int length;

int listsize;

}SqList;

Status InitBiTree(SqList &L, char \*treeName);

Status DestroyBiTree(BiTree &T);

Status CreateBiTree(SqList &L, char \*treeName, char \*treeString);

Status ClearBiTree(BiTree &T);

Status BiTreeEmpty(BiTree T);

int BiTreeDepth(BiTree T);

BiTree Root(BiTree T);

char Value(BiTree &T, TElemType e);

Status Assign(BiTree T, char e, char v);

BiTree Parent(BiTree T, char e);

BiTree LeftChild(BiTree T, char e);

BiTree RightChild(BiTree T, char e);

BiTree LeftSibling(BiTree T, char e);

BiTree RightSibling(BiTree T, char e);

Status InsertChild(SqList &L, int j, TElemType e, char LorR, TElemType \*y);

Status DeleteChild(SqList &L, int j, TElemType e, int LorR);

Status PreOrderTraverse(BiTree T);

Status InOrderTraverse(BiTree T);

Status PostOrderTraverse(BiTree T);

Status PreOrderTraverseSave(BiTree T);

int main(void) {

int operate = 1;

SqList L;

TElemType input[100], y[100], e, v;

int defaultTree = -1;

int i, j;

char t, LorR, treeName[20];

char filename[30];

/\*将线性表初始化\*/

L.elem = (Tree \*)malloc(LIST\_INIT\_SIZE \* sizeof(Tree));

L.length = 0;

L.listsize = LIST\_INIT\_SIZE;

while (operate) {

system("cls");

printf("------------------------------------------\n");

printf("1. InitBiTree 11. LeftChild\n");

printf("2. DestroyBiTree 12. RightChild\n");

printf("3. CreateBiTree 13. LeftSibling\n");

printf("4. ClearBiTree 14. RightSibling\n");

printf("5. BiTreeEmpty 15. InsertChild\n");

printf("6. BiTreeDepth 16. DeleteChild\n");

printf("7. Root 17. PreOrderTraverse\n");

printf("8. Value 18. InOrderTraverse\n");

printf("9. Assign 19. PostOrderTraverse\n");

printf("10.Parent 0. Exit\n\n");

printf("20.更改当前操作的树\n\n");

printf("------------------------------------------\n");

printf("选择你的操作:");

scanf("%d", &operate);

switch (operate)

{

case 1:

/\*构造空二叉树\*/

printf("\n输入构造的空二叉树名称：\n");

scanf("%s", treeName);

for (i = 1; i <= L.length; i++) {

if (strcmp(L.elem[i - 1].name, treeName) == 0) break;

}

if (i <= L.length) {

printf("此名称已经存在。\n\n");

getchar();

getchar();

break;

}

if (InitBiTree(L, treeName) == OK)

{

printf("成功构造空二叉树！\n\n");

}

else printf("空二叉树构造失败!\n\n");

getchar();

getchar();

break;

case 2:

/\*销毁二叉树\*/

printf("\n输入销毁的二叉树的名称：\n");

scanf("%s", treeName);

for (i = 1; i <= L.length; i++) {

if (strcmp(L.elem[i - 1].name, treeName) == 0) break;

}

if (i>L.length) {

printf("无法找到该名称的二叉树!\n");

getchar();

getchar();

break;

}

if (DestroyBiTree(L.elem[i - 1].HeadNode) == OK) {

for (; i<L.length; i++) { //移动顺序表之后的数据元素

strcpy(L.elem[i - 1].name, L.elem[i].name);

L.elem[i - 1].HeadNode = L.elem[i].HeadNode;

}

printf("成功销毁了二叉树！\n");

L.length--;

defaultTree = L.length - 1 ;

}

else printf("二叉树的销毁操作失败！\n");

getchar();

getchar();

break;

case 3:

//CreateBiTree

printf("\n输入构造的二叉树名称：\n");

scanf("%s", treeName);

for (i = 1; i <= L.length; i++)

{//在线性表中逐个查找，看二叉树是否存在

if (strcmp(L.elem[i - 1].name, treeName) == 0)break;

}

if (i <= L.length)

{

printf("\n此名称已经存在\n");

getchar();

getchar();

break;

}

printf("\n按照先序顺序，依次输入各个结点的元素('#' 表示该结点为空)：\n");

getchar();

i = 0;

char input[100];

while ((t = getchar()) != '\n') {

input[i] = t;

i++;

}

input[i] = '\n';

i = CreateBiTree(L, treeName, input);

if (i == OK)

{

defaultTree = L.length;

printf("\n成功构造二叉树！\n");

}

else if (i == INFEASIBLE) printf("\n成功构造二叉树，但是，所构造的为空二叉树!\n");

else if (i == FALSE) printf("\n构造二叉树失败\n");

getchar();

break;

case 4:

//清空二叉树

if (defaultTree == -1) {

printf("当前无二叉树存在，请先创建二叉树!\n");

printf("\n（按任意键返回……）\n");

getchar(); getchar();

break;

}

if (ClearBiTree(L.elem[defaultTree - 1].HeadNode) == OK) printf("\n成功清空二叉树!\n");

else printf("清空二叉树失败!\n");

getchar();

getchar();

break;

case 5:

//判定二叉树是否为空

if (defaultTree == -1) {

printf("当前无二叉树存在，请先创建二叉树!\n");

printf("\n（按任意键返回……）\n");

getchar(); getchar();

break;

}

if (BiTreeEmpty(L.elem[defaultTree - 1].HeadNode) == OK) printf("\n当前二叉树是空二叉树。\n");

else printf("\n当前二叉树不是空的。\n");

getchar();

getchar();

break;

case 6:

if (defaultTree == -1) {

printf("当前无二叉树存在，请先创建二叉树!\n");

printf("\n（按任意键返回……）\n");

getchar(); getchar();

break;

}

if (BiTreeDepth(L.elem[defaultTree - 1].HeadNode->lchild) == 0) printf("\n当前二叉树是空二叉树!\n");

else printf("\n当前操作的二叉树的深度为:%d\n", BiTreeDepth(L.elem[defaultTree - 1].HeadNode->lchild));

getchar();

getchar();

break;

case 7:

//找到树根

if (defaultTree == -1) {

printf("当前无二叉树存在，请先创建二叉树!\n");

printf("\n（按任意键返回……）\n");

getchar(); getchar();

break;

}

if (Root(L.elem[defaultTree - 1].HeadNode) == NULL) printf("\n当前二叉树是空的,没有树根!\n");

else printf("\n当前二叉树根结点标记为：%c \n", Root(L.elem[defaultTree - 1].HeadNode)->data.tag);

getchar();

getchar();

break;

case 8:

//找到树中某结点的值

if (defaultTree == -1) {

printf("当前无二叉树存在，请先创建二叉树!\n");

printf("\n（按任意键返回……）\n");

getchar(); getchar();

break;

}

printf("\n输入你想查找的结点标记：\n");

getchar(); e = getchar();

if (Value(L.elem[defaultTree - 1].HeadNode, e) == NULL)

{

printf("\n该二叉树中不存在所输入的结点！！\n");

}else if (Value(L.elem[defaultTree - 1].HeadNode, e) == INFEASIBLE) printf("\n该结点无值!请先用Assign函数对该结点赋值。\n");

else printf("\n在当前二叉树中，此二叉树结点%c的值为:%c\n",e, Value(L.elem[defaultTree - 1].HeadNode, e));

getchar();

getchar();

break;

case 9:

//给树中某结点附值

if (defaultTree == -1) {

printf("当前无二叉树存在，请先创建二叉树!\n");

printf("\n（按任意键返回……）\n");

getchar(); getchar();

break;

}

printf("\n输入你希望赋值结点(char)\n");

getchar(); e = getchar();

printf("\n输入你希望附的值(char)：\n");

getchar(); v = getchar();

if (Assign(L.elem[defaultTree - 1].HeadNode, e, v) == OK) printf("\n给结点赋值成功!\n");

else printf("\n二叉树中没有此结点，给结点赋值失败!\n");

getchar();

getchar();

break;

case 10:

//找到树的父结点

if (defaultTree == -1) {

printf("当前无二叉树存在，请先创建二叉树!\n");

printf("\n（按任意键返回……）\n");

getchar(); getchar();

break;

}

printf("\n输入结点的标记：\n");

getchar();

e = getchar();

if (Parent(L.elem[defaultTree - 1].HeadNode, e) == NULL)

printf("\n所输入结点为此树的根，或不存在父结点，或此树中不含所输入结点。\n");

else

printf("\n所输入结点的父结点的标记是：%c \n", Parent(L.elem[defaultTree - 1].HeadNode, e)->data.tag);

getchar();

getchar();

break;

case 11:

//找到树的左孩子

if (defaultTree == -1) {

printf("当前无二叉树存在，请先创建二叉树!\n");

printf("\n（按任意键返回……）\n");

getchar(); getchar();

break;

}

printf("\n输入你希望找到其左孩子的结点的标记：（字符）\n");

getchar(); e = getchar();

if (LeftChild(L.elem[defaultTree - 1].HeadNode, e) == NULL) printf("\n所输入结点没有左孩子，或该二叉树中不含此结点！\n");

else printf("\n所输入结点的左孩子的标记为：%c \n", LeftChild(L.elem[defaultTree - 1].HeadNode, e)->data.tag);

getchar();

getchar();

break;

case 12:

//找到树的右孩子

if (defaultTree == -1) {

printf("当前无二叉树存在，请先创建二叉树!\n");

printf("\n（按任意键返回……）\n");

getchar(); getchar();

break;

}

printf("\n输入你希望找到其右孩子的结点的标记：\n");

getchar(); e = getchar();

if (RightChild(L.elem[defaultTree - 1].HeadNode, e) == NULL) printf("\n所输入结点无右孩子，或该二叉树中不含此结点！\n");

else printf("\n所输入结点的右孩子的标记为：%c\n", RightChild(L.elem[defaultTree - 1].HeadNode, e)->data.tag);

getchar();

getchar();

break;

case 13:

//寻找左兄弟

if (defaultTree == -1) {

printf("当前无二叉树存在，请先创建二叉树!\n");

printf("\n（按任意键返回……）\n");

getchar(); getchar();

break;

}

printf("\n输入你希望找到其左兄弟的结点的标记：\n");

getchar(); e = getchar();

if (LeftSibling(L.elem[defaultTree - 1].HeadNode, e) == NULL) printf("\n此结点没有左兄弟，或该二叉树不含所输入结点！\n");

else printf("\n该结点的左兄弟的标记为：%c\n", LeftSibling(L.elem[defaultTree - 1].HeadNode, e)->data.tag);

getchar();

getchar();

break;

case 14:

//寻找右兄弟

if (defaultTree == -1) {

printf("当前无二叉树存在，请先创建二叉树!\n");

printf("\n（按任意键返回……）\n");

getchar(); getchar();

break;

}

printf("\n请输入你希望找到其右兄弟的结点的标记：\n");

getchar(); e = getchar();

if (RightSibling(L.elem[defaultTree - 1].HeadNode, e) == NULL) printf("\n此结点没有右兄弟，或二叉树中不含所输入结点！\n");

else printf("\n所输入结点的右兄弟的值为：%c\n", RightSibling(L.elem[defaultTree - 1].HeadNode, e)->data.tag);

getchar();

getchar();

break;

case 15:

//插入子树

if (defaultTree == -1) {

printf("当前无二叉树存在，请先创建二叉树!\n");

printf("\n（按任意键返回……）\n");

getchar(); getchar();

break;

}

printf("\n请输入你希望插入子树的结点（字符）\n");

getchar(); e = getchar();

printf("插入到该结点的左子树，请输入0；右子树，请输入1。\n");

getchar(); LorR = getchar();

printf("\n请按照先序输入插入树的各个结点的数据（char），'#'表示此节点不存在！\n");

getchar();

j = 0;

while ((t = getchar()) != '\n') {

input[j] = t;

j++;

}

input[j] = '\n';

j = InsertChild(L, defaultTree - 1, e, LorR, input);

if (j == INFEASIBLE) printf("\n当前二叉树是空的！\n");

else if (j == OVERFLOW) printf("\n所输入的子树不符合条件，插入无效！\n");

else if (j == OK) printf("\n成功插入子树！\n");

else printf("\n插入失败了,因为二叉树%s中不存在节点%c \n", L.elem[defaultTree - 1].name, e);

getchar();

break;

case 16:

//删除子树

if (defaultTree == -1) {

printf("当前无二叉树存在，请先创建二叉树!\n");

printf("\n（按任意键返回……）\n");

getchar(); getchar();

break;

}

printf("\n请输入你希望删除子节点的字符标记（char）\n");

getchar();

e = getchar();

printf("\n删除该结点的左结点，请输入0；删除该结点的右结点，请输入1\n");

getchar(); LorR = getchar();

j = DeleteChild(L, i - 1, e, LorR);

if (j == OK) printf("\n成功删除结点！\n");

else if (j == INFEASIBLE) printf("\n所删除的二叉树为空，不可执行！\n");

else printf("\n删除失败,当前二叉树%s中不含节点%c！\n", L.elem[defaultTree - 1].name, e);

getchar();

getchar();

break;

case 17:

//前序遍历

if (defaultTree == -1) {

printf("当前无二叉树存在，请先创建二叉树!\n");

printf("\n（按任意键返回……）\n");

getchar(); getchar();

break;

}

printf("\n-----------------所有元素 -----------------------\n");

if (PreOrderTraverse(L.elem[defaultTree - 1].HeadNode->lchild) == FALSE) printf("此二叉树是空树！");

printf("\n------------------ 结束 ------------------------\n\n");

getchar();

getchar();

break;

case 18:

//中序遍历

if (defaultTree == -1) {

printf("当前无二叉树存在，请先创建二叉树!\n");

printf("\n（按任意键返回……）\n");

getchar(); getchar();

break;

}

printf("\n----------------所有元素 -----------------------\n");

if (InOrderTraverse(L.elem[defaultTree - 1].HeadNode->lchild) == FALSE) printf("\n此二叉树是空树！");

printf("\n------------------结束 ------------------------\n\n");

getchar();

getchar();

break;

case 19:

//后序遍历

if (defaultTree == -1) {

printf("当前无二叉树存在，请先创建二叉树!\n");

printf("\n（按任意键返回……）\n");

getchar(); getchar();

break;

}

printf("\n-----------------所有元素 -----------------------\n");

if (PostOrderTraverse(L.elem[defaultTree - 1].HeadNode->lchild) == FALSE) printf("\n此二叉树是空树！");

printf("\n------------------ 结束 ------------------------\n\n");

printf("\n");

getchar();

getchar();

break;

case 20:

printf("\n输入更换的二叉树的名称：\n");

scanf("%s", treeName);

for (i = 1; i <= L.length; i++) {

if (strcmp(L.elem[i - 1].name, treeName) == 0) break;

}

if (i>L.length) {

printf("无法找到该名称的二叉树!\n");

getchar();

getchar();

break;

}

defaultTree = i;

printf("成功将当前操作树转为:%s\n", treeName);

getchar();

getchar();

break;

case 0:

break;

}

}

printf("感谢使用本系统！\n");

return 0;

}

/\*case 1: 构建一个新的二叉树\*/

Status InitBiTree(SqList &L, char \*treeName)

{

Tree \*newbase;

if (L.length >= LIST\_INIT\_SIZE)

{

newbase = (Tree \*)realloc(L.elem, (L.listsize + LISTINCREMENT) \* sizeof(Tree));

if (!newbase)exit(OVERFLOW);

L.listsize += LISTINCREMENT;

L.elem = newbase;

}

strcpy(L.elem[L.length].name, treeName);

L.elem[L.length].HeadNode = (BiTree)malloc(sizeof(BiTNode));

L.elem[L.length].HeadNode->data.tag = 0;

L.elem[L.length].HeadNode->lchild = L.elem[L.length].HeadNode->rchild = NULL;

L.length++;

return OK;

}

/\*case 2: 销毁二叉树\*/

Status DestroyBiTree(BiTree &T)

{

int peak = 0;

BiTree temp[100], p, q;

p = T;

if (p->lchild == NULL)

{

free(T);

return OK;

}

temp[peak] = p->lchild;

while (peak>-1)

{

p = temp[peak--];

q = p;

if (p->lchild)temp[++peak] = p->lchild;

if (p->rchild)temp[++peak] = p->rchild;

free(q);

}

free(T);

return OK;

}

/\*case 3: 创建一个二叉树\*/

Status CreateBiTree(SqList &L, char \*treeName, char \*treeString) {

BiTree temp[100], curp, q;

Tree \*newbase;

int i = 1, peak = 0, n = 1;

if (L.length >= L.listsize)

{

newbase = (Tree \*)realloc(L.elem, (L.listsize + LISTINCREMENT) \* sizeof(Tree));

if (!newbase)exit(OVERFLOW);

L.elem = newbase;

L.listsize += LISTINCREMENT;

}

strcpy(L.elem[L.length].name, treeName);

L.elem[L.length].HeadNode = (BiTree)malloc(sizeof(BiTNode));

L.elem[L.length].HeadNode->data.tag = 0;

L.elem[L.length].HeadNode->lchild = L.elem[L.length].HeadNode->rchild = NULL;

if (treeString[0] == '#')

{

//树根不存在，则返回不可执行

L.length++;

return INFEASIBLE;

}

curp = (BiTree)malloc(sizeof(BiTNode));

curp->data.tag = treeString[0];

curp->lchild = curp->rchild = NULL;

L.elem[L.length].HeadNode->lchild = curp; //将当前树的头指向根

temp[peak] = curp;

while (treeString[i] != '\n')

{

if (treeString[i] != '#')

{

curp = (BiTree)malloc(sizeof(BiTNode));

curp->data.tag = treeString[i];

curp->lchild = curp->rchild = NULL;

while (temp[peak]->rchild != NULL) peak--; //退回到子结点未满的结点上

if (peak <= -1) return FALSE;

q = temp[peak];

temp[++peak] = curp;

if (treeString[i - 1] != '#')

q->lchild = curp;

else {

q->rchild = curp;

}

n++;

}

else

{

while (temp[peak]->rchild != NULL) peak--;

q = temp[peak];

if (treeString[i - 1] != '#')

q->lchild = NULL;

else

{

q->rchild = NULL;

peak--;

}

}

i++; //下一个

}

L.elem[L.length].HeadNode->data.tag = n;

L.length++;

return OK;

}

/\*case4:清空二叉树\*/

Status ClearBiTree(BiTree &T) {

BiTree temp[100], p, q;

int peak = 0;

p = T->lchild;

if (T->lchild == NULL)return FALSE;

temp[peak] = T->lchild;

while (peak>-1)

{

p = temp[peak--];

q = p;

if (p->rchild) temp[++peak] = p->rchild;

if (p->lchild) temp[++peak] = p->lchild;

free(q);

}

T->data.tag = 0;

T->lchild = NULL;

return OK;

}

/\*case5:判断二叉树是否为空\*/

Status BiTreeEmpty(BiTree T) {

if (T->data.tag == 0) return OK;

return FALSE;

}

/\*case6：判断二叉树的深度\*/

int BiTreeDepth(BiTree T) {

int i, j;

if (T == NULL) return 0;

i = BiTreeDepth(T->lchild);

j = BiTreeDepth(T->rchild);

if (i>j) return (i + 1);

return (j + 1);

}

/\*case7: 找到树的根\*/

BiTree Root(BiTree T) {

return T->lchild;

}

/\*case8: 找到元素所在结点\*/

char Value(BiTree &T, TElemType e) {

int peak = 0;

BiTree stack[100], p;

stack[peak] = T->lchild;

while (peak>-1)

{

p = stack[peak--];

if (p->data.tag == e)

{

if (p->data.isInit == 1)

{

return p->data.number;

}

else

{

return INFEASIBLE;

}

}

if (p->rchild) stack[++peak] = p->rchild;

if (p->lchild) stack[++peak] = p->lchild;

}

return NULL;

}

/\*case9:给二叉树中的结点赋值\*/

Status Assign(BiTree T, char e, char v) {

int peak = 0;

BiTree temp[100], p;

temp[peak] = T->lchild;

while (peak>-1)

{

p = temp[peak--];

if (p->data.tag == e)break;

if (p->rchild)temp[++peak] = p->rchild;

if (p->lchild)temp[++peak] = p->lchild;

}

if (p->data.tag != e)return FALSE;

p->data.number = v;

p->data.isInit = 1;

return OK;

}

/\*case10:找到所输入值的父结点\*/

BiTree Parent(BiTree T, char e)

{

int peak = 0;

BiTree temp[100], p;

temp[peak] = T->lchild;

while (peak > -1)

{

p = temp[peak--];

if (p->rchild)

{

if (p->rchild->data.tag == e) {

peak++;

break;

}

temp[++peak] = p->rchild;

}

if (p->lchild)

{

if (p->lchild->data.tag == e) {

peak++;

break;

}

temp[++peak] = p->lchild;

}

}

if (peak <= -1)

return NULL;

return p;

}

/\*case11:找到左孩子\*/

BiTree LeftChild(BiTree T, char e)

{

int peak = 0;

BiTree temp[100], p;

temp[peak] = T->lchild;

while (peak > -1)

{

p = temp[peak--];

if (p->data.tag == e) return p->lchild;

if (p->rchild)temp[++peak] = p->rchild;

if (p->lchild)temp[++peak] = p->lchild;

}

if (peak <= -1)return NULL;

}

/\*case12:找到右孩子\*/

BiTree RightChild(BiTree T, char e)

{

int peak = 0;

BiTree temp[100], p;

temp[peak] = T->lchild;

while (peak > -1)

{

p = temp[peak--];

if (p->data.tag == e) return p->rchild;

if (p->rchild)temp[++peak] = p->rchild;

if (p->lchild)temp[++peak] = p->lchild;

}

if (peak <= -1)return NULL;

}

/\*case13:找到左兄弟\*/

BiTree LeftSibling(BiTree T, char e)

{

int peak = 0;

BiTree temp[100], p;

temp[peak] = T->lchild;

if (T->lchild->data.tag == e)return NULL;

while (peak > -1)

{

p = temp[peak]--;

if (p->rchild)

{

if (p->rchild->data.tag == e)

{

if (p->lchild)

{

return p->lchild;

}

else {

return NULL;

}

}

temp[++peak] = p->rchild;

}

if (p->lchild)

{

if (p->lchild->data.tag == e) return NULL;

temp[++peak] = p->lchild;

}

}

if (peak <= -1)return NULL;

}

/\*case14:找到右兄弟\*/

BiTree RightSibling(BiTree T, char e)

{

int peak = 0;

BiTree temp[100], p;

temp[peak] = T->lchild;

if (T->lchild->data.tag == e)return NULL;

while (peak > -1)

{

p = temp[peak]--;

if (p->rchild)

{

if (p->rchild->data.tag == e) return NULL;

temp[++peak] = p->rchild;

}

if (p->lchild)

{

if (p->lchild->data.tag == e)

{

if (p->rchild)

{

return p->rchild;

}

else

{

return NULL;

}

}

temp[++peak] = p->lchild;

}

}

if (peak <= -1)return NULL;

}

/\*case 15:插入子树\*/

Status InsertChild(SqList &L, int j, TElemType e, char LorR, char\* x)

{

int i, n, peak;

TNode stack[100], p, q, CT, T;

i = 1, n = 0, peak = -1;

stack[++peak] = L.elem[j].HeadNode->lchild;

if (L.elem[j].HeadNode->data.tag == 0) return INFEASIBLE;

while (peak>-1)

{

p = stack[peak--];

if (p->data.tag == e) break;

if (p->rchild) stack[++peak] = p->rchild;

if (p->lchild) stack[++peak] = p->lchild;

}

if (p->data.tag != e) return FALSE;

T = p;

if (x[0] == '#') return OVERFLOW;

p = (BiTree)malloc(sizeof(BiTNode));

n++;

p->data.tag = x[0];

p->lchild = p->rchild = NULL;

CT = p;

peak = -1;

stack[++peak] = p;

while (x[i] != '\n')

{

if (x[i] != '#')

{

p = (BiTree)malloc(sizeof(BiTNode));

p->data.tag = x[i];

p->lchild = p->rchild = NULL;

while (stack[peak]->rchild != NULL) peak--;

if (peak <= -1) return FALSE;

q = stack[peak];

stack[++peak] = p;

if (x[i - 1] != '#')

q->lchild = p;

else

{

q->rchild = p;

}

n++;

}

else

{

while (stack[peak]->rchild != NULL) peak--;

q = stack[peak];

if (x[i - 1] != '#')

q->lchild = NULL;

else

{

q->rchild = NULL;

peak--;

}

}

i++;

}

if (CT->rchild != NULL) return OVERFLOW; //构造的子树右子树不为空，报错

if (LorR == '0')

{

q = T->lchild;

CT->rchild = q;

T->lchild = CT;

}

else

{

q = T->rchild;

CT->rchild = q;

T->rchild = CT;

}

L.elem[j].HeadNode->data.tag = L.elem[j].HeadNode->data.tag + n;

return OK;

}

/\*case 16:删除子树\*/

Status DeleteChild(SqList &L, int j, TElemType e, int LorR)

{

int peak = -1, i = 0;

TNode stack[100], p, q;

stack[++peak] = L.elem[j].HeadNode->lchild;

if (L.elem[j].HeadNode->data.tag == 0) return INFEASIBLE;

while (peak>-1)

{

p = stack[peak--];

if (p->data.tag == e) break;

if (p->rchild) stack[++peak] = p->rchild;

if (p->lchild) stack[++peak] = p->lchild;

}

if (p->data.tag != e) return FALSE;

peak = -1;

if (LorR == '0')

{

stack[++peak] = p->lchild;

p->lchild = NULL;

}

else

{

stack[++peak] = p->rchild;

p->rchild = NULL;

}

while (peak>-1)

{

q = stack[peak--];

i++;

if (q->rchild) stack[++peak] = q->rchild;

if (q->lchild) stack[++peak] = q->lchild;

}

L.elem[j].HeadNode->data.tag = L.elem[j].HeadNode->data.tag - i;

return OK;

}

/\*case 17:前序遍历\*/

Status PreOrderTraverse(BiTree T)

{

if (T != NULL)

{

if (T->data.tag)printf("%c ", T->data.tag);

PreOrderTraverse(T->lchild);

PreOrderTraverse(T->rchild);

return OK;

}

return FALSE;

}

/\*case 18:中序遍历\*/

Status InOrderTraverse(BiTree T)

{

if (T != NULL) {

InOrderTraverse(T->lchild);

printf("%c ", T->data.tag);

InOrderTraverse(T->rchild);

return OK;

}

return FALSE;

}

/\*case 19:后序遍历\*/

Status PostOrderTraverse(BiTree T)

{

if (T != NULL)

{

PostOrderTraverse(T->lchild);

PostOrderTraverse(T->rchild);

printf("%c ", T->data.tag);

return OK;

}

return FALSE;

}