

**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201701**

**学 号：**

**姓 名：**

**指导教师：**

**报告日期： 2017年 11月 22 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](#_Toc501905202)

[1.1问题描述 2](#_Toc501905203)

[1.2 系统设计 2](#_Toc501905204)

[1.3系统实现 16](#_Toc501905205)

[1.4 实验小结 23](#_Toc501905206)

[2 基于链式存储结构的线性表实现 24](#_Toc501905207)

[2.1 问题描述 24](#_Toc501905208)

[2.2 系统设计 24](#_Toc501905209)

[2.3 系统实现 37](#_Toc501905210)

[2.4 实验小结 44](#_Toc501905211)

[3 基于二叉链表的二叉树实现 46](#_Toc501905212)

[3.1 问题描述 46](#_Toc501905213)

[3.2 系统设计 46](#_Toc501905214)

[3.3 系统实现 74](#_Toc501905215)

[3.4 实验小结 86](#_Toc501905216)

[4 基于邻接表的图实现 86](#_Toc501905217)

[4.1 问题描述 86](#_Toc501905218)

[4.2 系统设计 86](#_Toc501905219)

[4.3 系统实现 86](#_Toc501905220)

[4.4 实验小结 86](#_Toc501905221)

[参考文献 87](#_Toc501905222)

[附录A 基于顺序存储结构线性表实现的源程序 90](#_Toc501905223)

[附录B 基于链式存储结构线性表实现的源程序 105](#_Toc501905224)

[附录C 基于二叉链表二叉树实现的源程序 123](#_Toc501905225)

[附录D 基于邻接表图实现的源程序 156](#_Toc501905226)

# 1 基于顺序存储结构的线性表实现

## 1.1问题描述

在本次实验中，我以顺序表作为线性表的物理结构，使用C语言实现了动态分配顺序表的基本运算，演示过程中仅对一个顺序表进行操作，并将数据元素抽象成为一个整型变量,具体应用背景下可修改数据元素类型。程序源代码可在VS2015编译环境下编译。

具体到程序的实现，我的程序实现了线性表的12个基本运算：初始化表，销毁表，清空表，判定表空，求表长，获得元素，查找元素，获得前驱，获得后继，插入元素，删除元素，遍历表。2个附加功能：保存数据和加载数据。以及1个简单的菜单框架进行演示。在实现过程中，我为一些函数增加了特殊功能：（1）查找操作：运用了函数指针，通过compare()函数进行查找，这样在面对不同应用背景下的线性表查找操作只需更改compare()函数即可继续实现该背景下的查找。（2）遍历操作，使用visit()函数进行遍历，可以只修改visit()函数即可实现不同背景下的遍历内容。（3）存储操作，使用二进制的方式进行存储，存储效率高，速度快。（4）在调用功能函数前已对线性表的初始条件进行检验，确保程序不会异常退出。

## 1.2 系统设计

1.2.1 系统总体设计

用户打开程序后会看到如下界面：

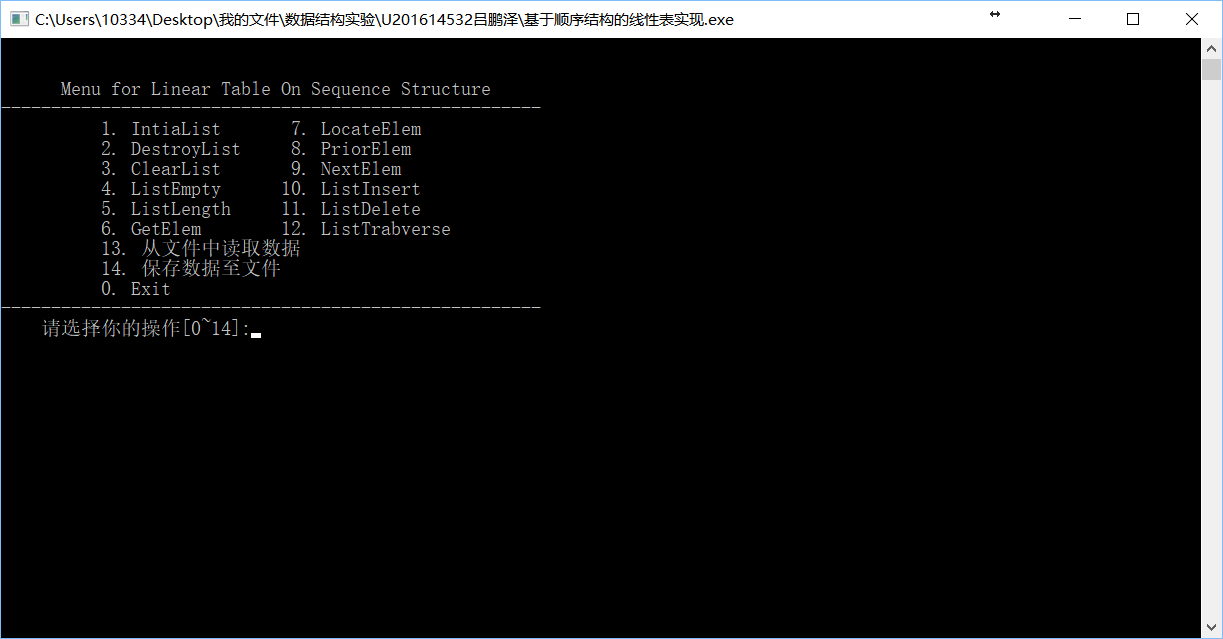


图1-1系统载入界面

用户通过输入在[1,14]区间的整数选择相应的功能，输入0退出程序。演示的过程可抽象为如图1-2所示的流程图。

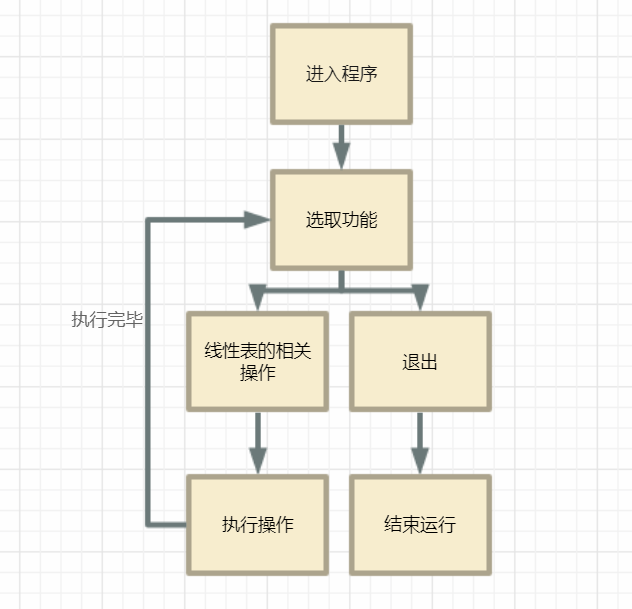


图1-2系统总体结构

1.2.2线性表物理结构

如图1-3，表示的是顺序表的物理结构。表的信息存储在顺序表结构L中，elem指示线性表的基地址，length指示线性表的当前长度，Listsize指示表所能存储的最大的数据元素的个数。

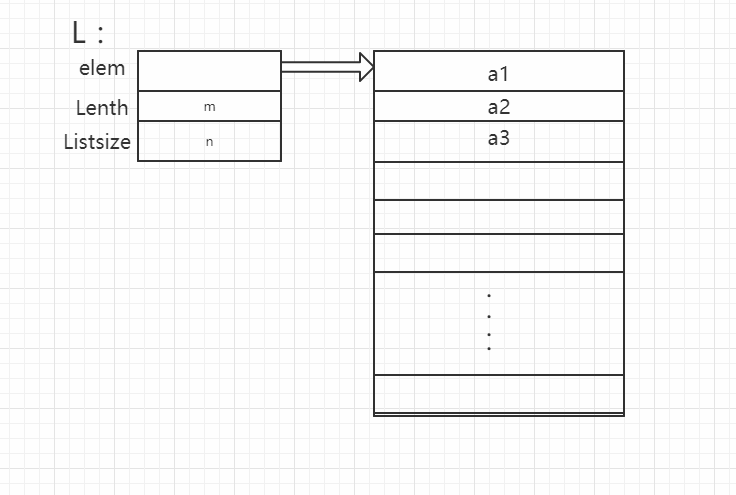


图1-3 顺序表物理结构

1.2.3相关常量的类型与定义

1.函数返回状态定义：

函数运行成功返回TRUE，失败返回FALSE，正常执行完毕返回OK，异常结束返回ERROR，动态分配空间不足返回OVERFLOW。在我的程序中用C语言描述如下所示：

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

2.相关常量

FILENAME表示保存顺序表信息的文件名称，status表示函数运行状态，Elemtype为数据元素类型，LIST\_INIT\_SIZE为初始表大小，LISTINCREMENT为表增容大小。C语言描述如下所示：

#define FILENAME "data"

typedef int status;

typedef int ElemType;

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

3.顺序表结构定义

表的信息存储在顺序表结构L中，elem指示线性表的基地址，length指示线性表的当前长度，Listsize指示表所能存储的最大的数据元素的个数。C语言描述如下所示：

typedef struct{

ElemType \* elem;

int length;

int listsize;

}SqList;

1.2.4算法设计

* 1. InitiaList(&L)

算法思想：1.申请存储数据的空间；2.置表长为0。

操作结果：构造一个空的线性表

时间复杂度：O(1) （说明：以下时间复杂度均指平均时间复杂度）

流程图：



图1-4 InitiaList()流程图

* 1. DestroyList(&L)

算法思想：1.释放线性表的存储空间；2.置表指针为空

操作结果：销毁线性表L

时间复杂度：O(1)

流程图：

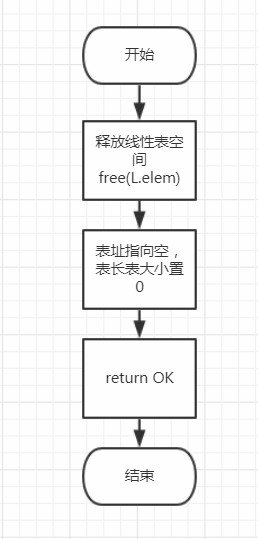


图1-5 DestroyList()流程图

* 1. ClearList(&L)

算法思想：置表长为0

操作结果：线性表L置空

时间复杂度：O(1)

流程图：

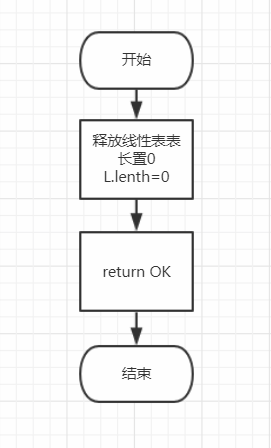


图1-6 ClearList()流程图

* 1. ListEmpty(L)

算法思想：若表长为0返回”TRUE”，否则返回”FALSE”。

操作结果：L为空返回TRUE,否则返回FALSE

时间复杂度：O(1)

流程图：

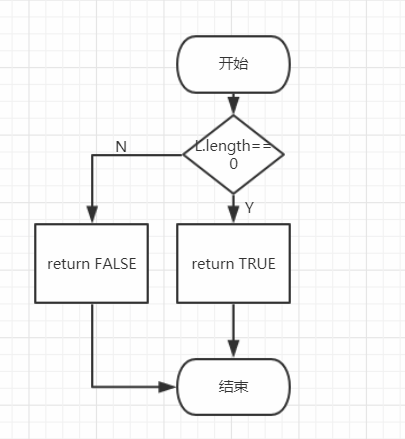


图1-7 ListEmpty()流程图

* 1. ListLength(L)

算法思想：返回表长

操作结果：返回线性表中元素的个数

时间复杂度：O(1)

流程图：

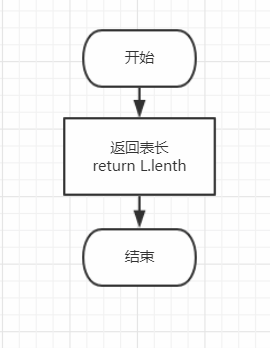


图1-8 ListLength(L)流程图

* 1. GetElem(L,i,&e)

算法思想：1.寻址公式定位第i个元素。2.将第i个元素赋值给e

操作结果：用e返回L中第i个数据元素的值

时间复杂度：O(1)

流程图：

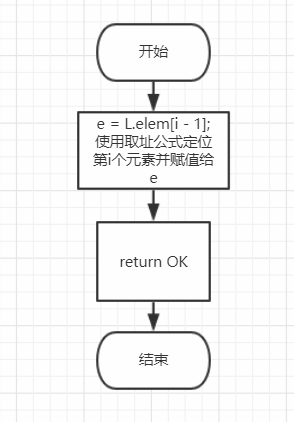


图1-9 GetElem()流程图

* 1. LocateElem(L,e,compare())

算法思想：1.用compare()函数查找e;2.找到返回该元素的位序,否则返回0。

操作结果：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。

时间复杂度：O(n)

流程图：

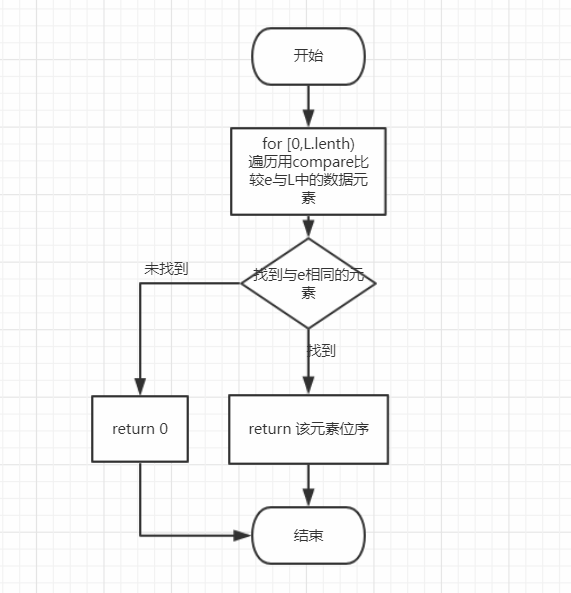


图1-10 LocateElem()流程图

* 1. PriorElem(L,cur\_e,&pre\_e)

算法思想：1.查找cur\_e获得其序号order; 2.若order>1,将order-1单元的元素值赋值给pre\_e,否则返回FALSE。

操作结果：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

时间复杂度：O(n)

流程图：



图1-11 PriorElem()流程图

* 1. NextElem(L,cur\_e,&next\_e)

算法思想：1.查找cur\_e获得其序号order; 2.若order<表长,将order+1单元的元素值赋值next\_e,否则返回FALSE。

操作结果：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

时间复杂度：O(n)

流程图：

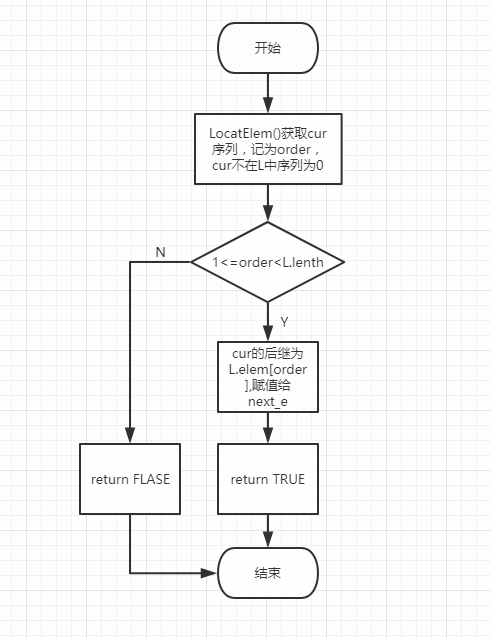


图1-12 NextElem()流程图

* 1. ListInsert(&L,i,e)

算法思想：1.判断空间是否已满，若满则增配空间，并修改listsize2.将序号为i-L.lenth的元素依次后移一位3.位置i插入e4.表长+1

操作结果：在L的第i个位置之前插入新的数据元素e。

时间复杂度：O(n)

流程图：

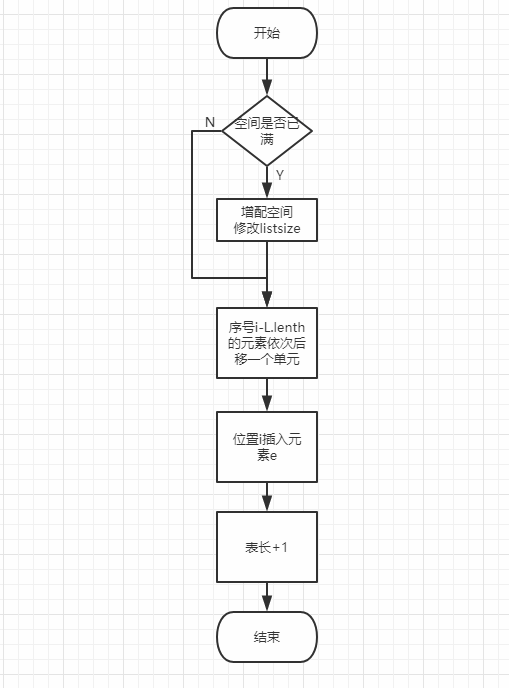


图1-13 ListInsert()流程图

* 1. ListDelete(&L,i,&e)

算法思想：1.i单元的值赋值给e；2.序号i+1-L.lenth的元素依次前移一个单元；3.表长-1

操作结果：删除L的第i个数据元素，用e返回其值。

时间复杂度：O(n)

流程图：

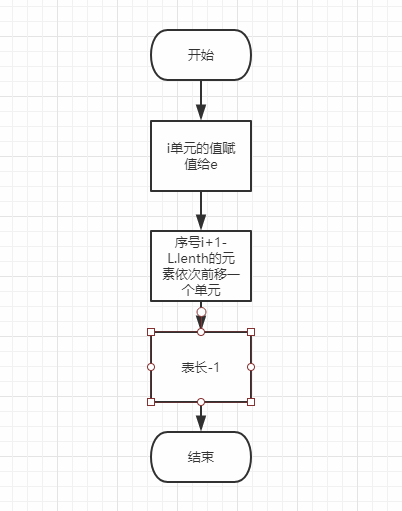


图1-14 ListDelete()流程图

* 1. ListTraverse(L,visit())

算法思想：使用visit()函数依次访问1-L.lenth数据元素

操作结果：对L的每个数据元素用函数visit()访问

时间复杂度：O(n)

流程图：

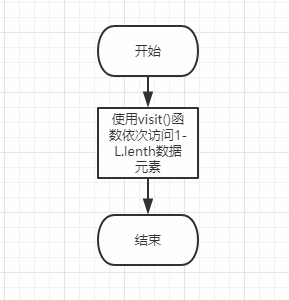


图1-15 ListTraverse()流程图

## 1.3系统实现

顺序表初始值为1，2，3，4，5，6，7，8共8个数据元素

1.初始界面

用户打开程序后可以看到如下界面，输入数字选择相应功能函数，输入0退出程序。

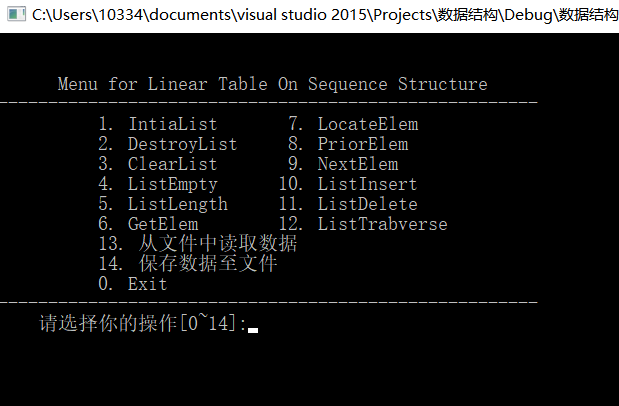
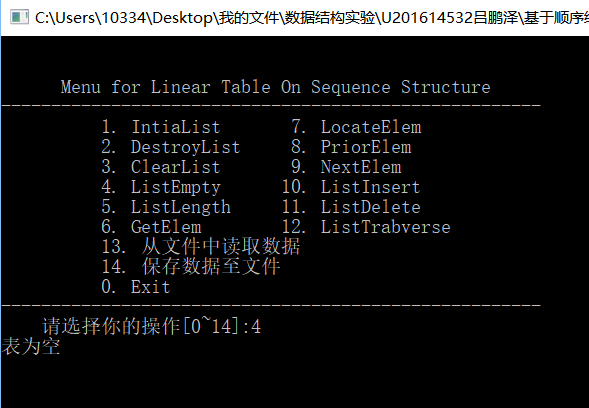


图1-16 系统载入界面

2.创建表

如图a，在表未创建前无法对表进行操作，如图b，c，在表创建后才可以对表进行操作。

|  |  |
| --- | --- |
| a | b |

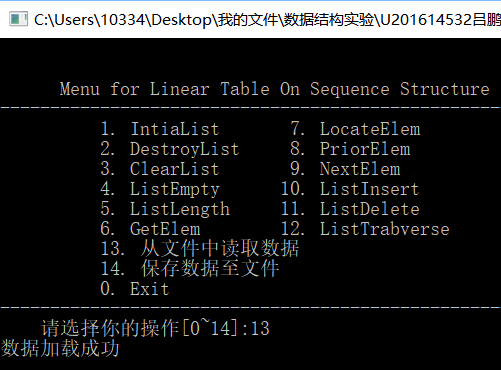


c

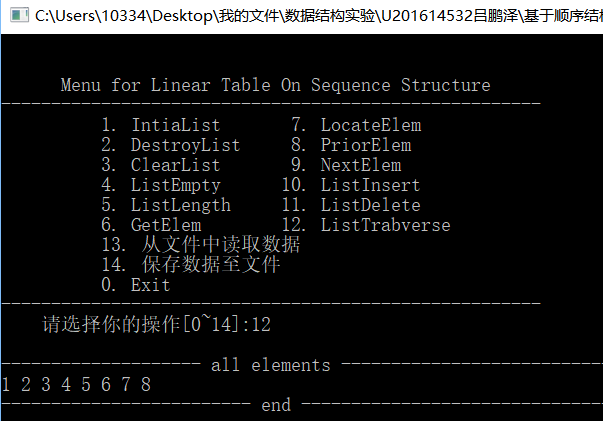
图1-16 创建表操作演示

3.从文件中加载数据及遍历操作的演示

创建表后表中不含数据元素，如图a，遍历表的结果为空。如图b，从文件中加载数据后再次遍历，结果如图c。



a b

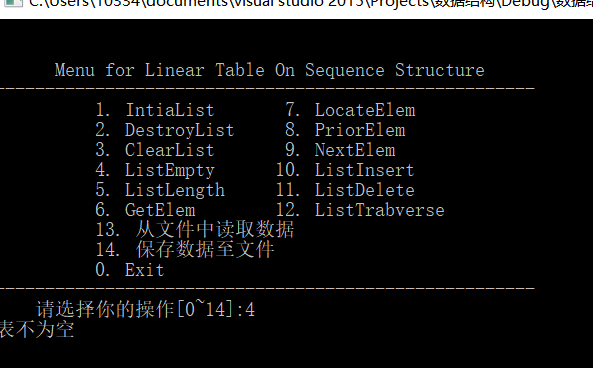


c

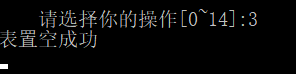
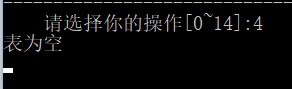
图1-17 加载数据及遍历操作的演示

4.判表空及置空表

如图a，初始化表后的操作结果，此时表为空。如图b，从文件中加载数据后再次判空表，此时表不为空。如图c、d，置空表后再次判表空。



a b

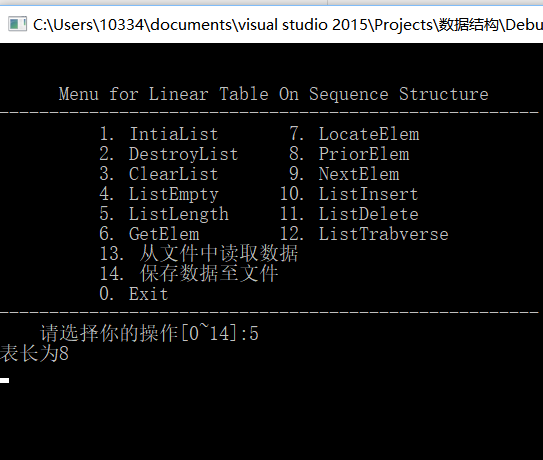
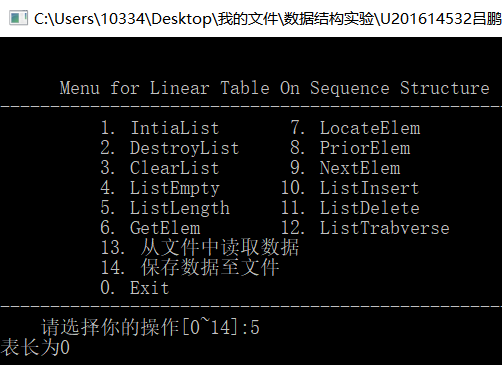
 

c d

图1-18 判空表

5.求表长

如图a，表为空是求表长的结果。如图b，从文件中读取数据后求表长的结果

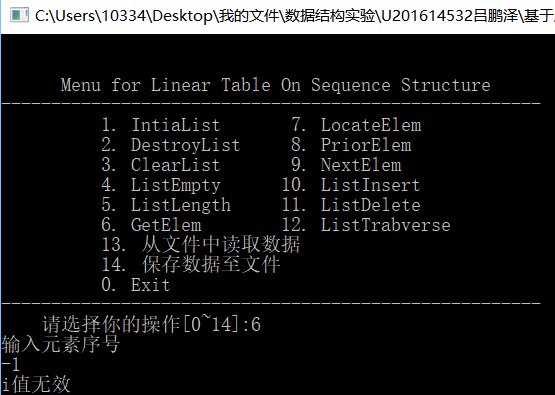


a b

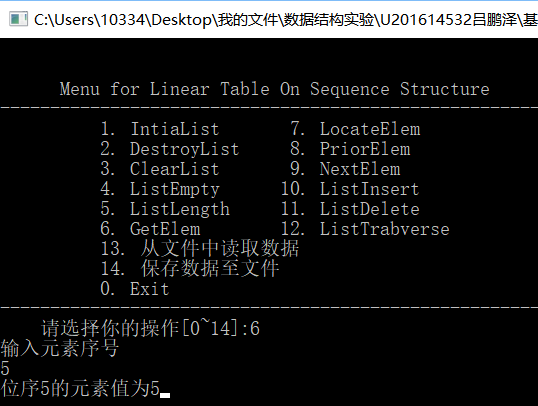
图1-19 求表长

6.获取元素

如图a，b，当元素序号的值小于1或大于表长时，程序会提示出错。如图c，只有元素序号大于等于1，小于等于表长时才能正确获取元素值。



a b

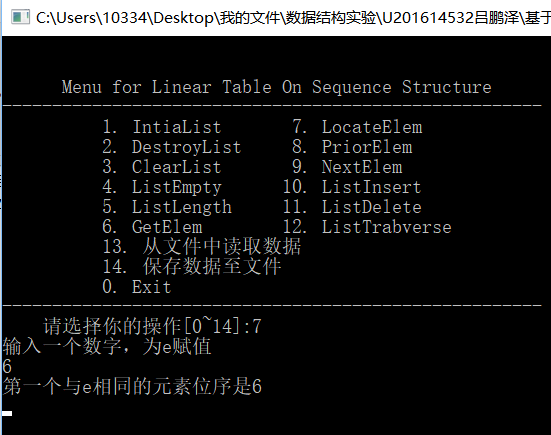
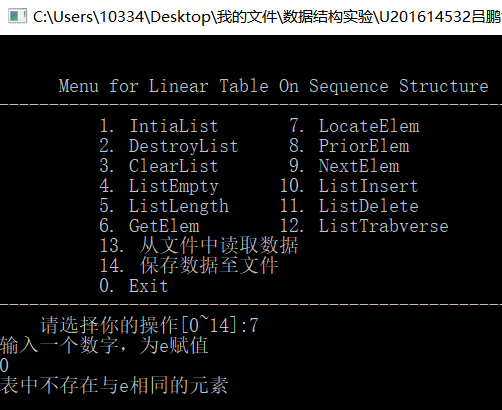


c

图1-20 获取元素

7.查找元素

如图a，为元素不在表中的查找结果。如图b，为元素在表中的查找结果。

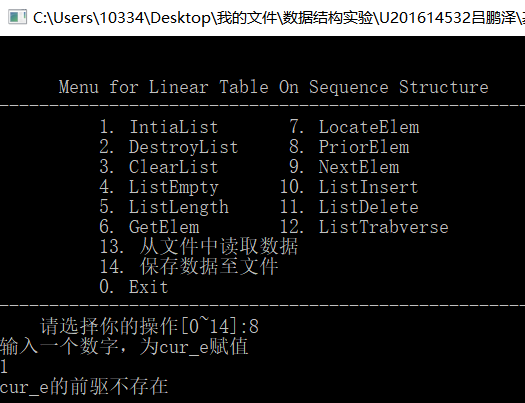
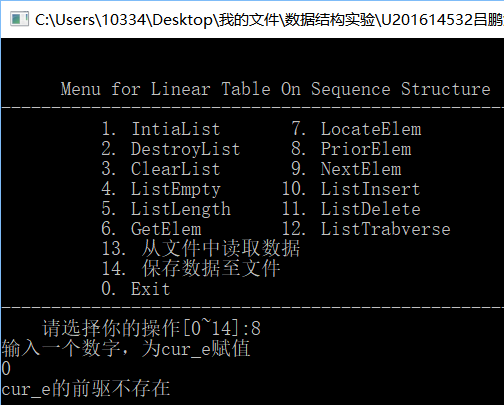


a b

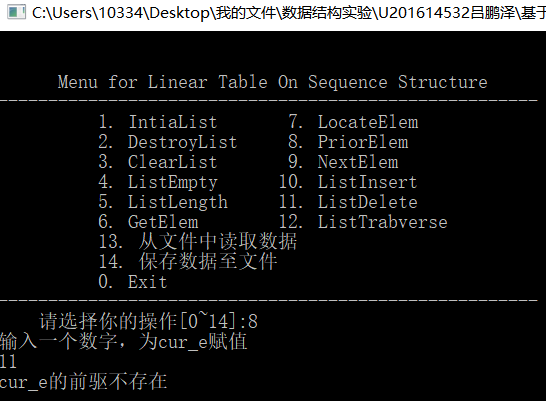
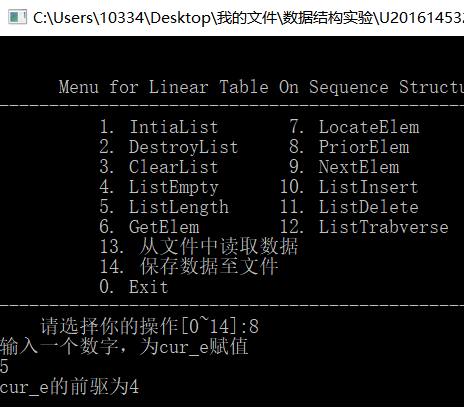
图1-21 查找元素

8.获取前驱

如图a、d，为元素cur不在表中的结果。如图b，为元素在表头的结果。如图c，为元素在表中的结果。



a b

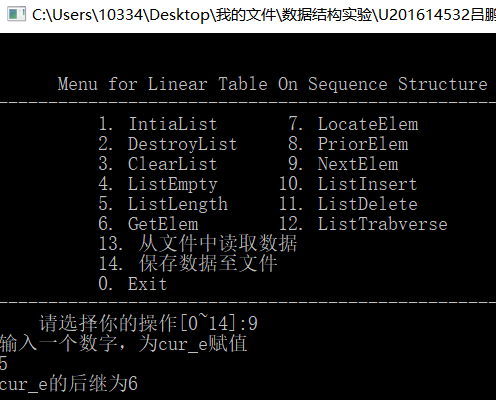
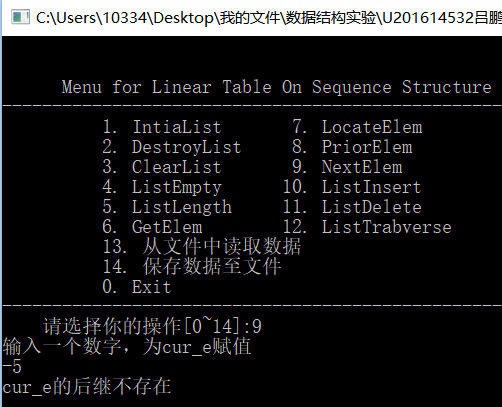


c d

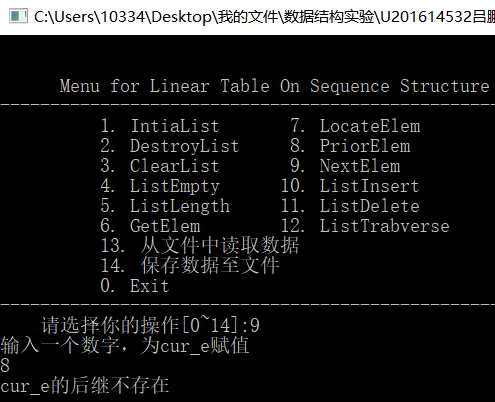
图1-22 获取前驱

9.获取后继：

如图a，为元素不在表中的结果。如图b，为元素在表中的结果。如图c，为元素在表尾的结果。



a b



c

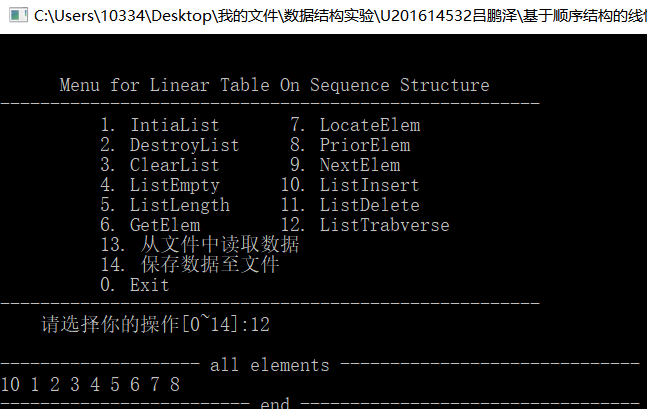
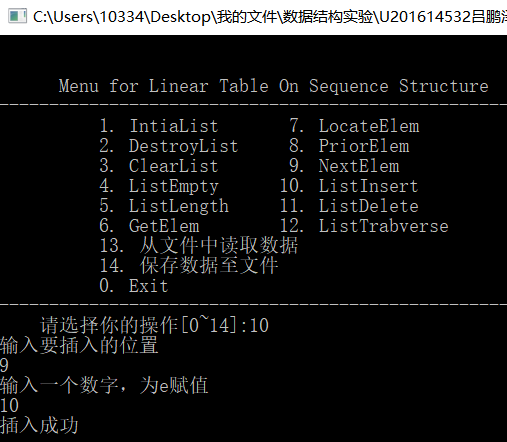
图1-23 获取后继

10.插入元素及插入后的结果

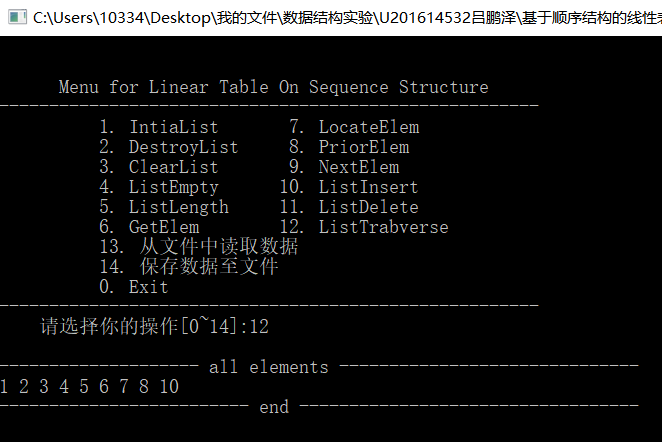
如图a，为在表外插入的结果。如图b、c，在表头插入结果及插入后的遍历。如d、e，在表尾插入结果及插入后的遍历结果。



a b

c d

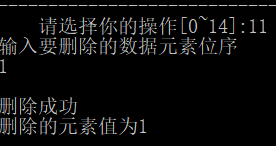
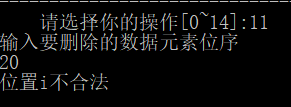


e

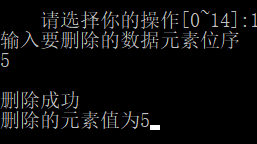
图1-24 插入元素

11.删除元素及删除后的结果：

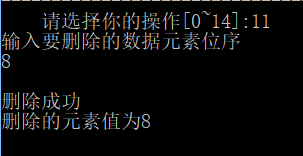
如图a，删除表外的元素会提示位置不合法。如图b、c，删除表头元素及删除后的遍历结果。如图d、e，删除表中元素及遍历后的结果。如图f、g，删除表尾元素及遍历后的结果。



a b



c d

e f



g

13.销毁表

销毁表后其余功能无法被调用。

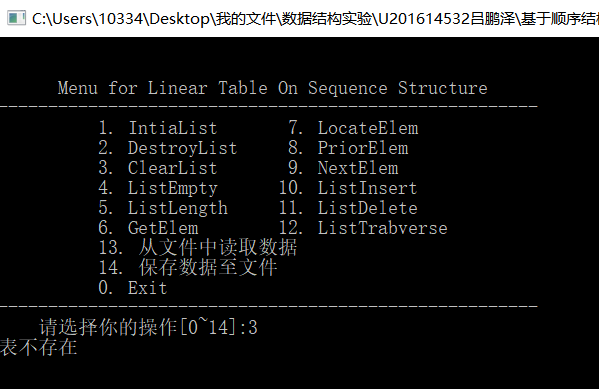


图1-26 销毁表

## 1.4 实验小结

本次数据结构实验让我加深了对线性表的理解，并让我复习了malloc()的用法，且重新掌握了realloc()的用法，以及形参&修饰符的运用。在课堂上的理论学习过程中，我一开始觉得线性表挺简单的，但是通过这次实际操作我发现从理论到时间还是有许多工作要做的，比如在调用函数时初始条件的判断、线性表空间的增配的都是十分容易出错的地方，在实验过程中，虽然没有遇到大的问题，但小问题遇到了不少，比如一开始在清空表时我把他同销毁表搞混了，后来仔细阅读书上的要求后才更正；另外，在写case是忘记加break了，也出现了一些小问题。在这次实验中，我强化了自己的编程能力。

# 2 基于链式存储结构的线性表实现

## 2.1 问题描述

在本次实验中，我将以链表作为线性表的物理结构，实验中我采用了带头结点的动态链表，使用C语言实现了该链表的线性表基本运算，演示过程中对两个动态链表进行操作，因此用户在调用功能前需要先确认对哪一个链表进行操作。此外，实验过程中我将数据元素抽象成为一个整型变量，在具体的应用背景下可修改数据元素类型来满足具体需求。程序源代码可在VS2015编译环境下编译通过。

具体到程序的实现，我的程序实现了线性表的12个基本运算：初始化表，销毁表，清空表，判定表空，求表长，获得元素，查找元素，获得前驱，获得后继，插入元素，删除元素，遍历表。3个附加功能：多链表操作、保存数据、加载数据。并使用1个简单的菜单框架进行演示。在实现过程中，我为一些函数增加了特殊功能：（1）查找操作：运用了函数指针，通过compare()函数进行比较，这样在面对不同应用背景下的线性表查找操作只需更改compare()函数即可继续实现该背景下的比较。（2）遍历操作，使用visit()函数进行遍历，可以只修改visit()函数即可实现不同应用背景下的遍历。（3）存储操作，使用二进制的方式进行存储，存储效率高，速度快。（4）在调用功能函数前已对线性表的初始条件进行检验，确保程序不会异常退出。（5）实现了对多链表进行操作。

## 2.2 系统设计

2.2.1 系统总体设计：

用户打开程序后会看到如下界面：

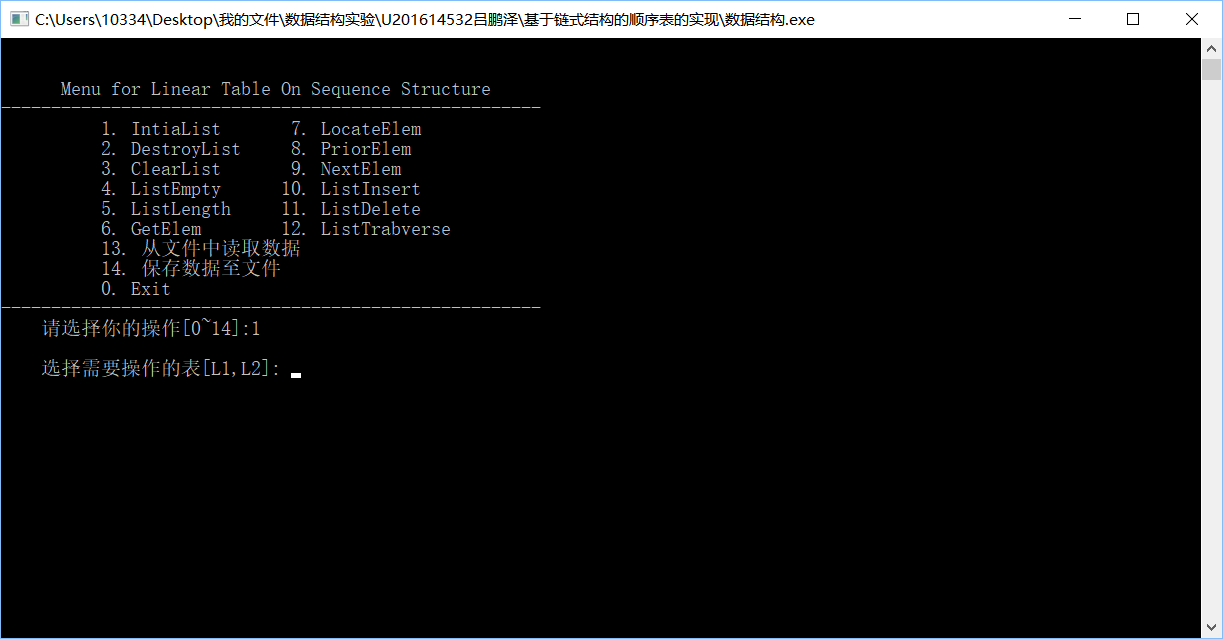


图2-1系统载入界面

用户通过输入在[1,14]区间的整数选择相应的功能，输入0退出程序，输入回车确认后需要选择操作的链表，输入L1或L2，。演示的过程可抽象为如图2-2所示的流程图。

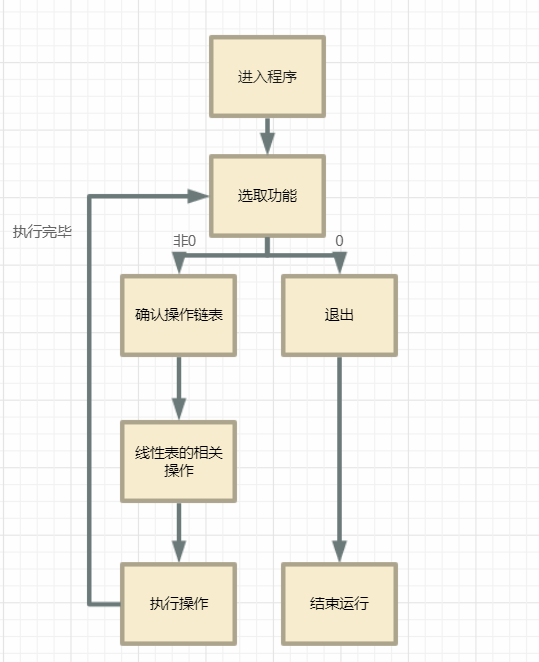
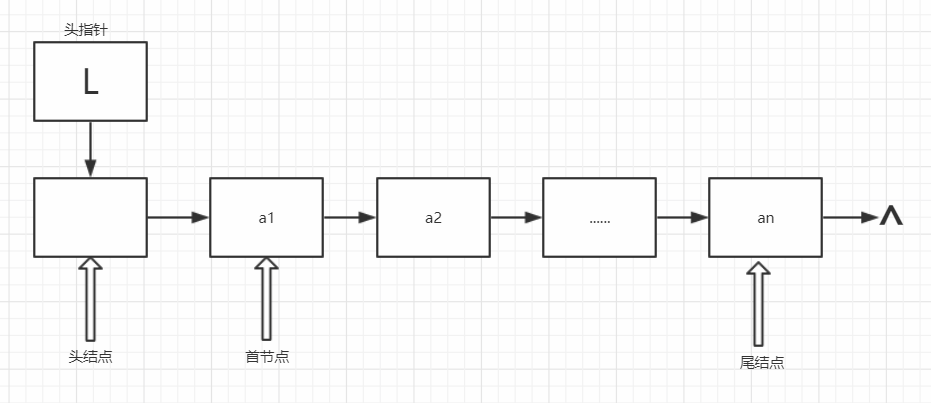


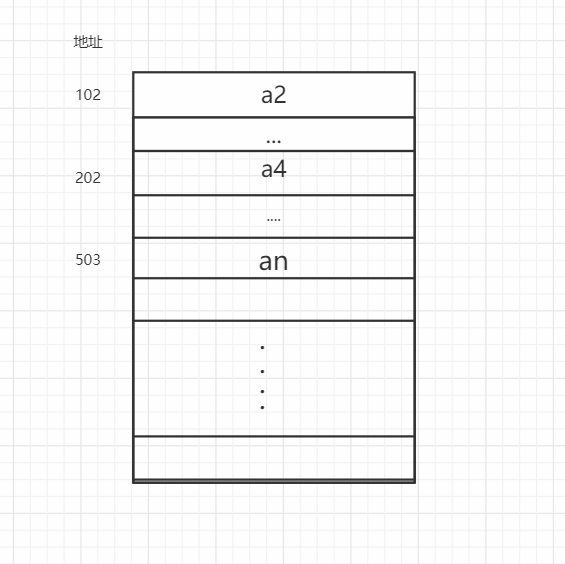
图2-2系统总体结构

2.2.2链表的物理结构

如图2-3-a，表示的是动态链表的逻辑结构，有一个头指针L指向头结点，每一个结点表示一个数据元素，每个结点包含数据域和指针域，尾节点指向空。如图2-3-b，表示的是动态链表在计算机中的存储结构，各个结点随机分布在内存中，并通过指针相连接。



a 动态链表逻辑结构



b 动态链表存储结构

图2-3 动态链表结构

2.2.3相关常量的类型与定义

1.函数返回状态定义：

函数运行成功返回TRUE，失败返回FALSE，正常执行完毕返回OK，异常结束返回ERROR，动态分配空间不足返回OVERFLOW。在我的程序中用C语言描述如下所示：

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

2.相关常量

FILENAME\_L表示保存动态链表信息的文件名称，status表示函数执行状态。C语言描述如下所示：

#define FILENAME\_L1 "dataL1"

#define FILENAME\_L2 "dataL2"

typedef int status;

3.顺序表结构定义

SqList表示头指针，指向头结点，Elemtype为数据元素类型，包含数据域和指针域，在本实验中数据域抽象为一个整数，指针域仅包含指向下一个节点的指针。C语言描述如下所示：

typedef ElemType\* SqList;//头指针

typedef struct elemtype

{

int date;

elemtype \* next;

}ElemType; //数据元素类型定义

2.2.4算法设计

* 1. InitiaList(&L)

算法思想：1.头指针声明一个存储结点；2.存储结点的指针域置空

操作结果：构造一个空的线性表

时间复杂度：O(1) （说明：以下时间复杂度均指平均时间复杂度）

流程图：

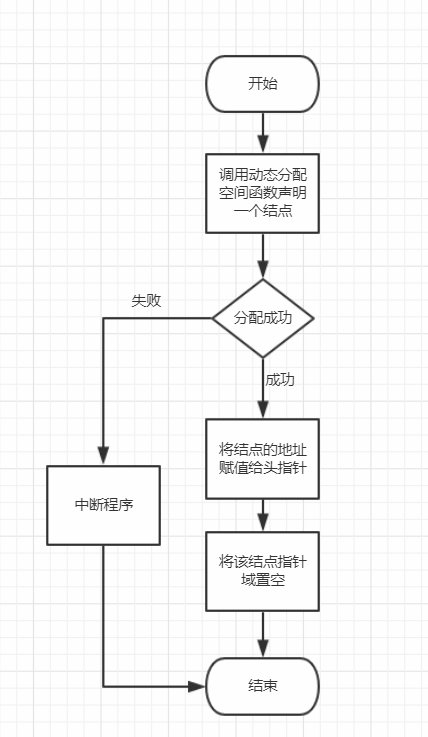


图2-4 InitiaList()流程图

* 1. DestroyList(&L)

算法思想：1.释放包括头结点在内的所有存储结点；2.置头指针为空

操作结果：销毁线性表L

时间复杂度：O(n)

流程图：

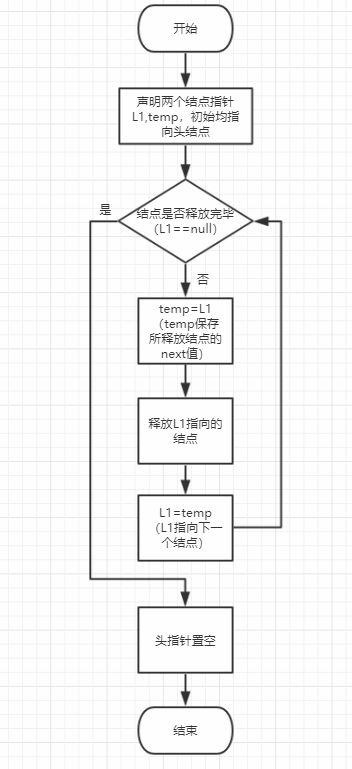


图2-5 DestroyList()流程图

* 1. ClearList(&L)

算法思想：1.释放除头结点之外的所有存储结点；2.头结点指针域置空

操作结果：线性表L置空

时间复杂度：O(n)

流程图：

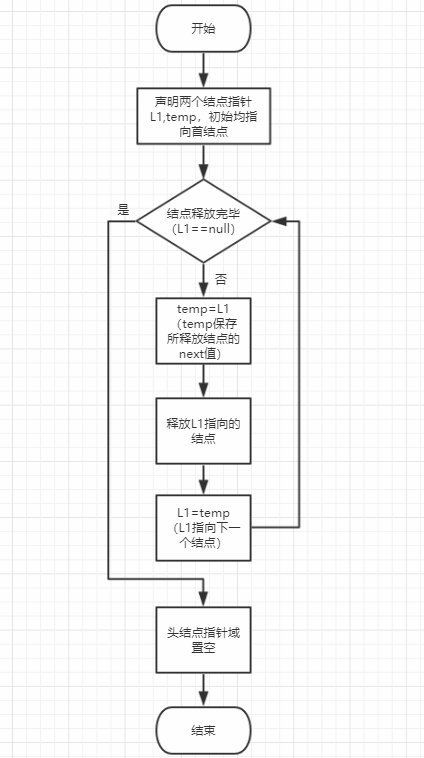


图2-6 ClearList()流程图

* 1. ListEmpty(L)

算法思想：若表长为0返回”TRUE”，否则返回”FALSE”。

操作结果：L为空返回TRUE,否则返回FALSE

时间复杂度：O(1)

流程图：

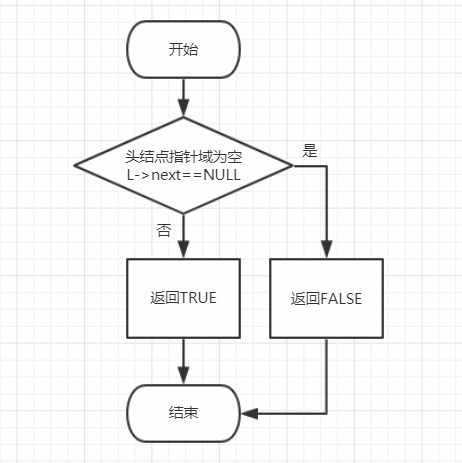


图2-7 ListEmpty()流程图

* 1. ListLength(L)

算法思想：1.指针p指向头结点；2.当p非空时移动p，统计移动次数count；3.返回count

操作结果：返回线性表中元素的个数

时间复杂度：O(n)

流程图：

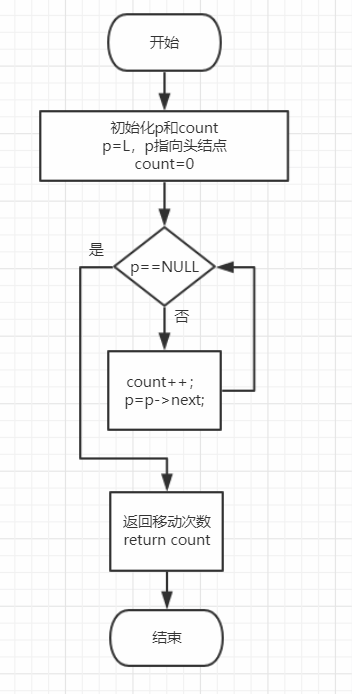


图2-8 ListLength(L)流程图

* 1. GetElem(L,i,&e)

算法思想：1.遍历链表定位第i个元素；2.将第i个元素赋值给e

操作结果：用e返回L中第i个数据元素的值

时间复杂度：O(n)

流程图：

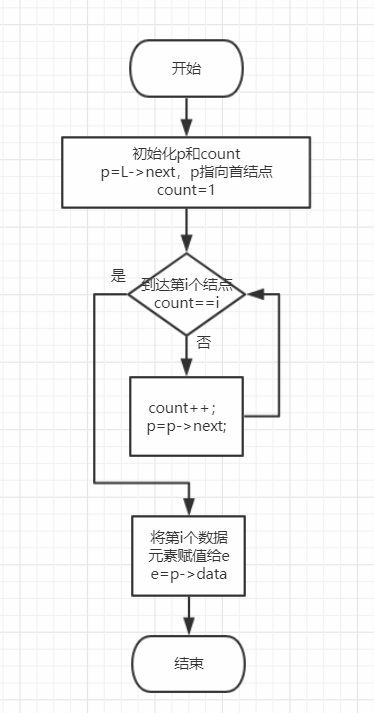


图2-9 GetElem()流程图

* 1. LocateElem(L,e,compare())

算法思想：1.遍历表中所有结点，分别与e进行比较，并记录结点位置；2.若找到与e满足compare关系的结点，返回该节点位置；3.未找到返回0；

操作结果：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。

时间复杂度：O(n)

流程图：

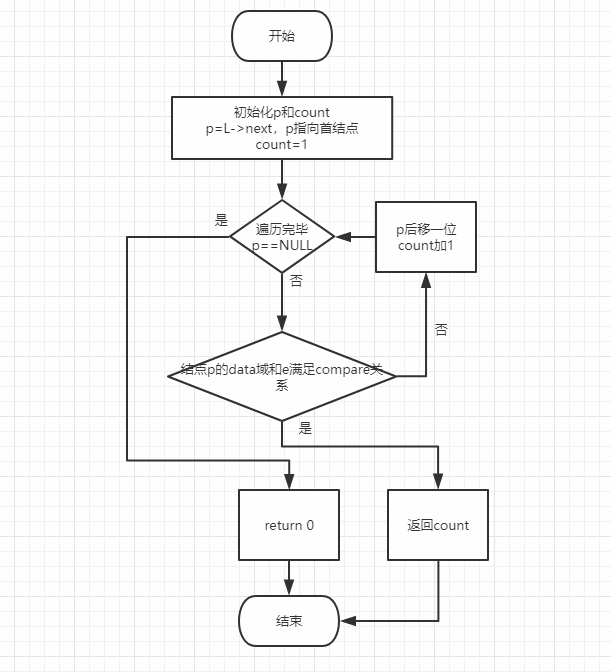


图2-10 LocateElem()流程图

* 1. PriorElem(L,cur\_e,&pre\_e)

算法思想：1.在表中查找cur\_e; 2.若找到并且cur\_e不是第一个元素，将cur\_e前驱的元素值赋值pre\_e,否则返回FALSE。

操作结果：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

时间复杂度：O(n)

流程图：

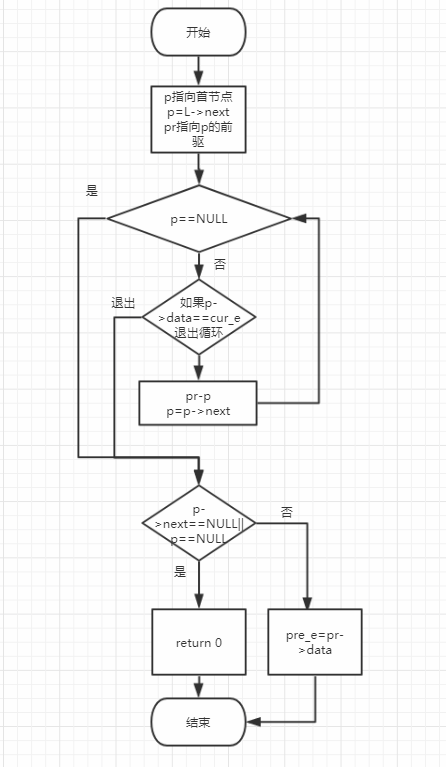


图2-11 PriorElem()流程图

* 1. NextElem(L,cur\_e,&next\_e)

算法思想：1.在表中查找cur\_e; 2.若找到并且cur\_e不是最后一个元素，将cur\_e下一个单元的元素值赋值next\_e,否则返回FALSE。

操作结果：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

时间复杂度：O(n)

流程图：

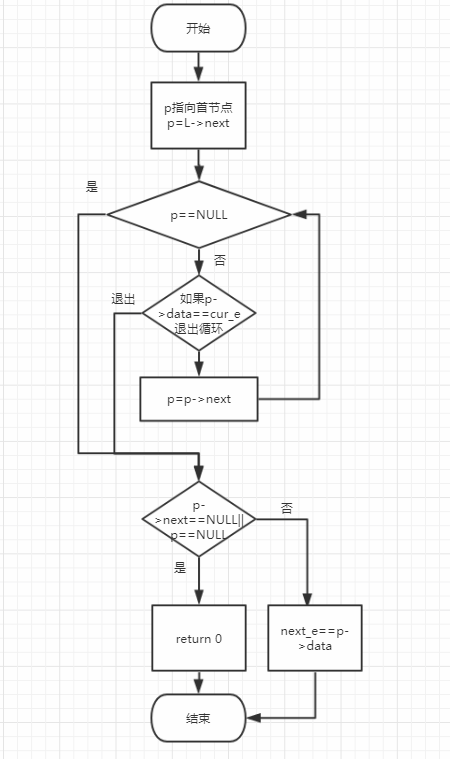


图2-12 NextElem()流程图

* 1. ListInsert(&L,i,e)

算法思想：1. 将指针p向后移动指向第i存储结点的前驱结点；2. 指针q指向申请的新结点，数据元素e赋值 到该结点的数据域；3.修改p，q指针。

操作结果：在L的第i个位置之前插入新的数据元素e。

时间复杂度：O(n)

流程图：

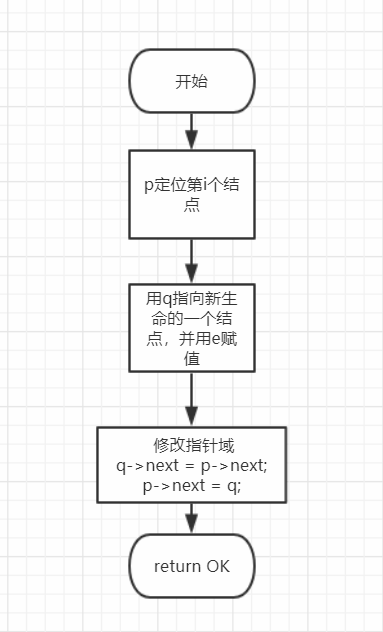


图2-13 ListInsert()流程图

* 1. ListDelete(&L,i,&e)

算法思想：1. 将指针p向后移动指向第i存储结点之直接前驱结点； 指针q指向第i存储结点，该结点的数据赋值到变量e ；2.修改指针；3.释放结点；

操作结果：删除L的第i个数据元素，用e返回其值。

时间复杂度：O(n)

流程图：

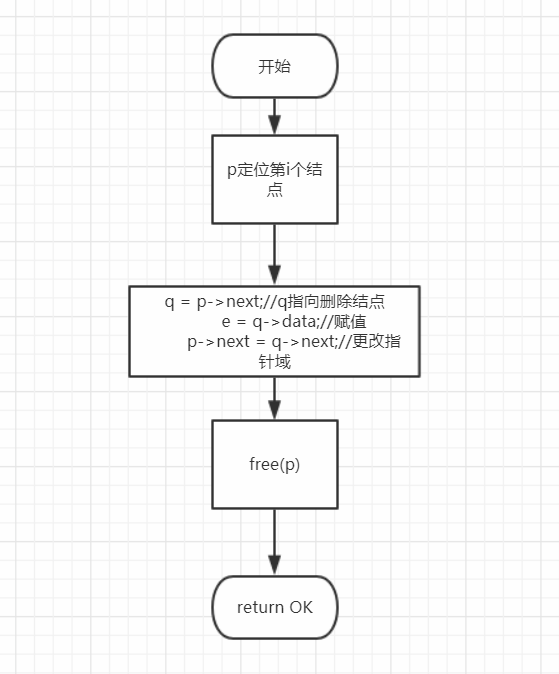


图2-14 ListDelete()流程图

* 1. ListTraverse(L,visit())

算法思想：使用visit()函数依次访问所有数据元素

操作结果：对L的每个数据元素用函数visit()访问

时间复杂度：O(n)

流程图：

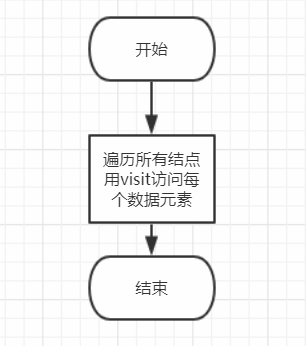


图2-15 ListTraverse()流程图

* 1. Savedata(L,filename)

算法思想：使用二进制存储方式保存所有结点的值

操作结果：将链表L的数据保存到filename文件中

时间复杂度：O(n)

流程图：

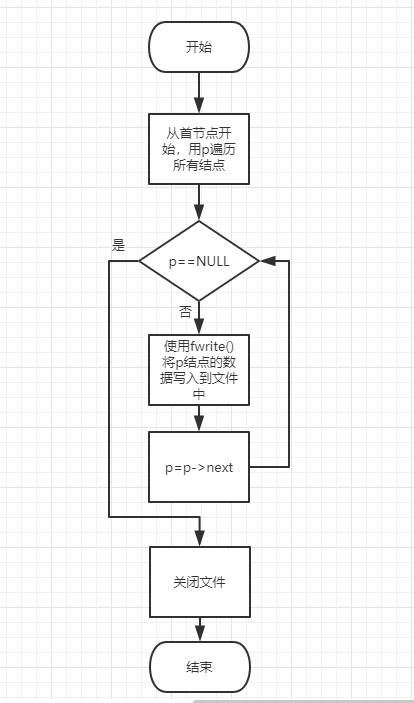


图2-16 Savedata()流程图

* 1. Loaddata(L,filename)

算法思想：使用二进制读取方式加载所有结点的值，动态创建链表

操作结果：将filename文件中的数据加载到动态链表中

时间复杂度：O(n)

流程图：

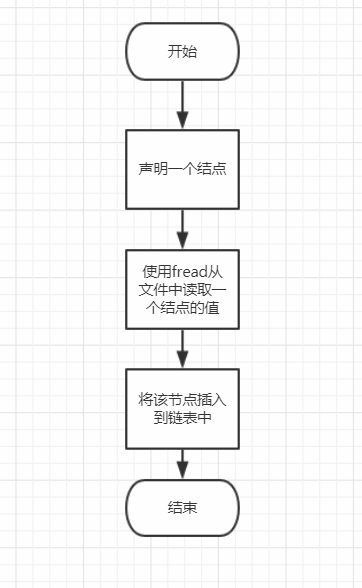


图2-16 Loaddata()流程图

* 1. visit(e)

算法思想：输出数据元素e的值

操作结果：输出数据元素e的值

时间复杂度：O(1)

流程图：

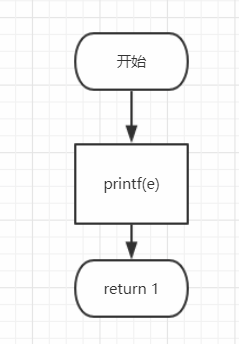


图2-17 visit()流程图

## 2.3 系统实现

L1的数据文件值为1-8，L2的数据文件值为1-5,为便于表现多链表操作，函数演示交替调用L1、L2。

1.初始界面

用户打开程序后可以看到如下界面，输入数字选择相应功能函数，输入0退出程序。

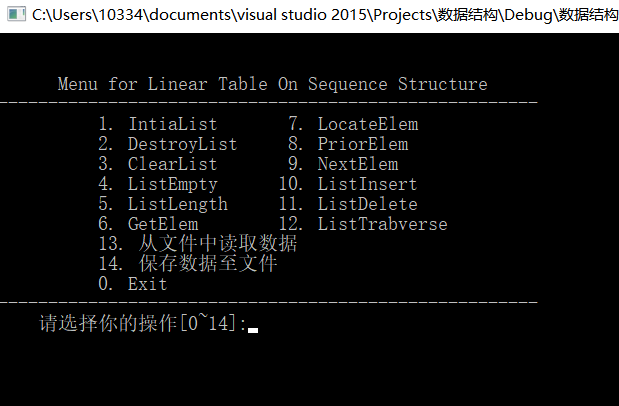


图2-18 系统载入界面

2.创建表

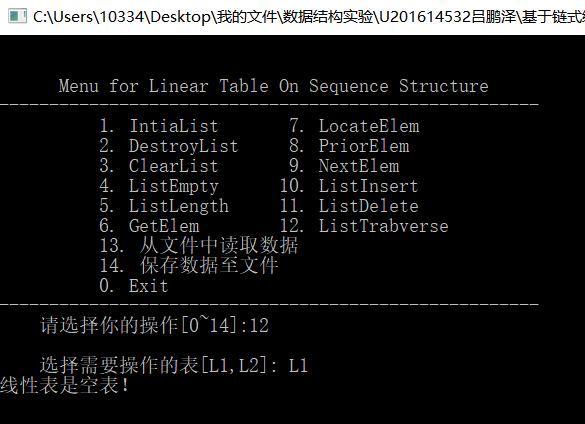
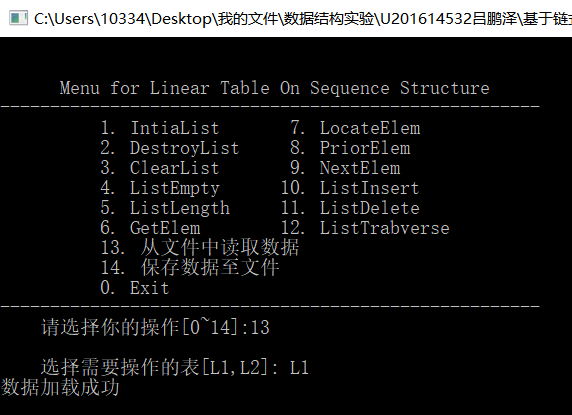
如图a，在表L1未创建前无法对表进行操作，如图b，表创建成功。

|  |  |
| --- | --- |
| a | b |

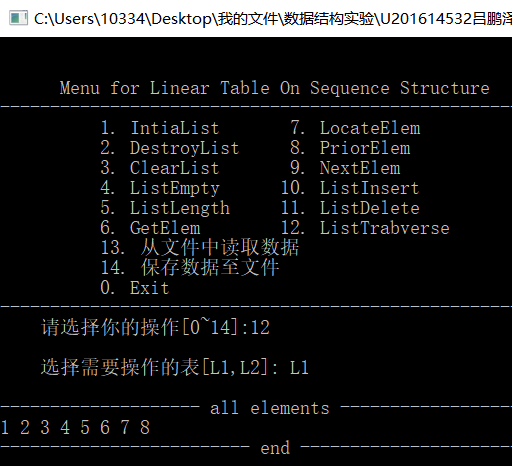
图2-19 创建表操作演示

3.从文件中加载数据及遍历操作的演示

创建表L1后表中不含数据元素，如图a，遍历表的结果为空。如图b，从文件中加载数据后再次遍历，结果如图c。

a b

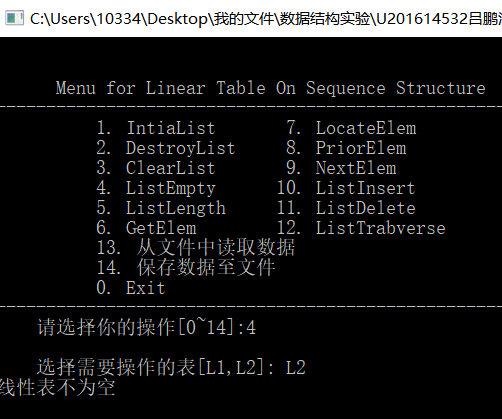
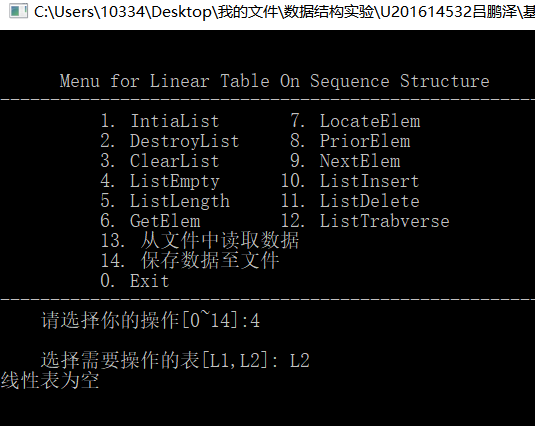


c

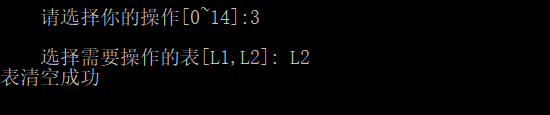
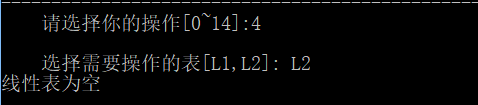
图2-20 加载数据及遍历操作的演示

4.判表空及置空表

如图a，初始化表L2后的操作结果，此时表为空。如图b，从文件中加载数据后再次判空表，此时表不为空。如图c、d，置空表后再次判表空。



a b

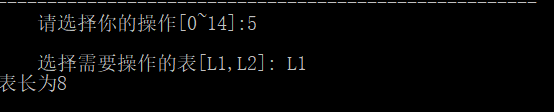
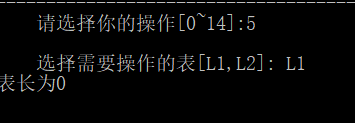
 

c d

图2-21 判空表

5.求表长

如图a，表为空是求表长的结果。如图b，从文件中读取数据后求L1表长的结果

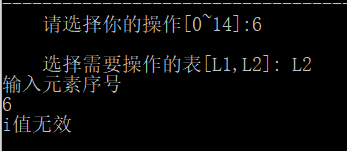
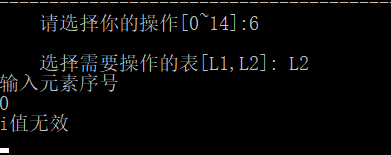


a b

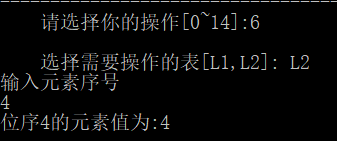
图2-22 求表长

6.获取元素

如图a，b，当元素序号的值小于1或大于L2表长时，程序会提示出错。如图c，只有元素序号大于等于1，小于等于表长时才能正确获取元素值。



a b

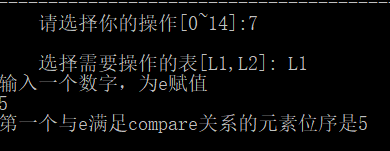
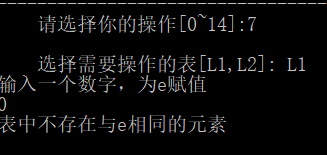


c

图2-23 获取元素

7.查找元素

如图a，为元素不在表L1中的查找结果。如图b，为元素在表中的查找结果。

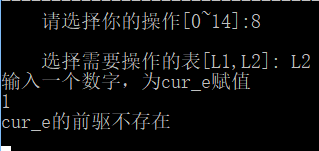
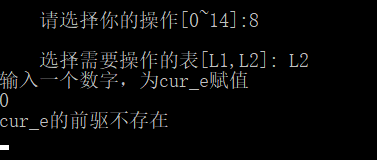


a b

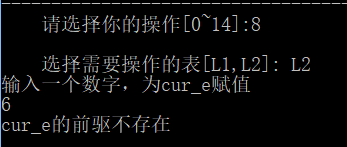
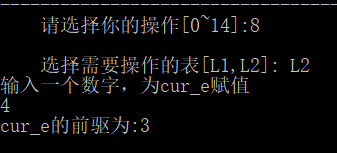
图2-24 查找元素

8.获取前驱

如图a、d，为元素cur不在表L2中的结果。如图b，为元素在表头的结果。如图c，为元素在表中的结果。



a b

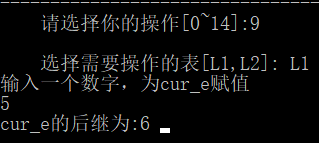
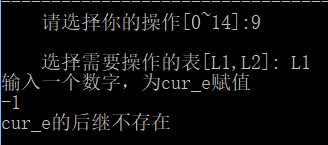


c d

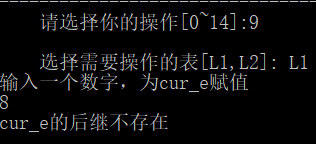
图2-25 获取前驱

9.获取后继：

如图a，为元素不在表L1中的结果。如图b，为元素在表中的结果。如图c，为元素在表尾的结果。



a b

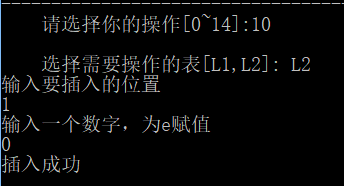


c

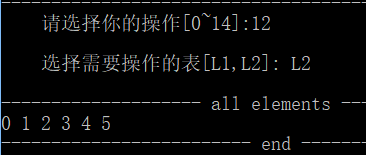
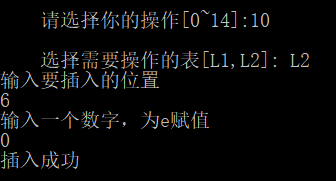
图2-26 获取后继

10.插入元素及插入后的结果

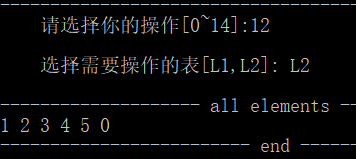
如图a，为在表L2外插入的结果。如图b、c，在表头插入结果及插入后的遍历。如d、e，在表尾插入结果及插入后的遍历结果。



a b

c d

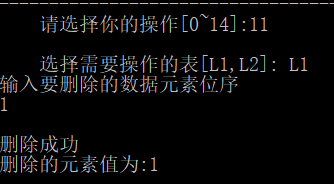
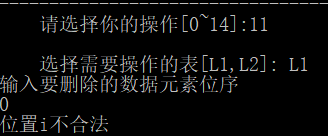


e

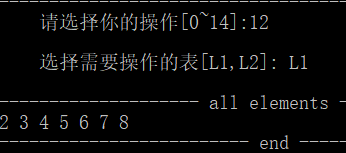
图2-27 插入元素

11.删除元素及删除后的结果：

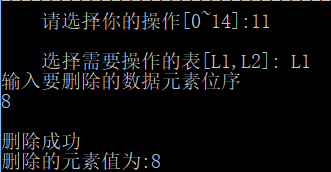
如图a，删除表L1外的元素会提示位置不合法。如图b、c，删除表头元素及删除后的遍历结果。如图d、e，删除表中元素及遍历后的结果。如图f、g，删除表尾元素及遍历后的结果。



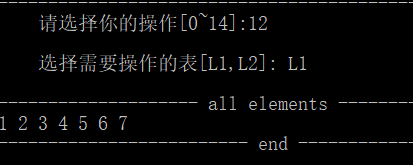
a b



c d

e f



g

图2-28 删除元素

13.销毁表

销毁表后其余功能无法被调用。

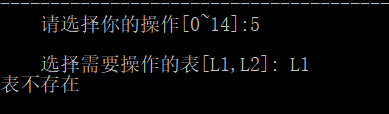
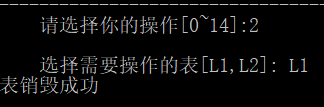


图2-29 销毁表

## 2.4 实验小结

在这次实验中，我实现了最基本的带头结点的动态链表。虽然C语言已经学过链表了，但通过理论课的学习我发现我对链表的掌握还很不够，尤其是通过这次的上机实验，我发现了自己动手能力不足，理论课听懂了不代表自己会写，这是我平时动手少的原因。另外，在实现的过程中，我看到了函数编程的好处，在编写链表的基本运算时，我出现了一些问题，由于是函数编程，我只修改一个函数就可以修正错误，而不用扫描整个程序代码。更重要的是，通过这两次实验，我感受到了自己程序调试水平的提高，之前好久都找不到的错误经过这两次的上机训练可以很快找出。

# 3 基于二叉链表的二叉树实现

## 3.1 问题描述

在本次实验中，我以二叉链表作为二叉树的物理结构，用C语言实现了二叉树的基本运算，实验中我采用了线性表控制多个二叉树，默认最多可控制50棵二叉树。演示程序中可以选择对多个不同的二叉树进行操作，可以通过功能选择改变操作的二叉树。此外，实验过程中我将树结点的数据域抽象成为一个整型变量key，表示树的关键字，字符变量TElemType，表示存储在树结点的数据元素。在具体的应用背景下可修改数据元素类型来满足具体需求。程序源代码已在VS2015编译环境下编译通过。

具体到程序的实现，我的程序实现了二叉树的20个基本运算：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，求二叉树深度，获得根节点，获得节点，节点赋值，获得双亲结点，获得左孩子结点，获得右孩子结点，获得左兄弟结点，获得右兄弟结点，插入子树，删除子树，前序遍历，中序遍历，后序遍历，按层遍历。3个附加功能：多二叉树操作、保存二叉树数据、加载二叉树数据，并使用1个简单的菜单框架进行演示。在实现过程中，我为一些函数增加了特殊功能：（1）遍历操作，使用visit()函数进行遍历，可以只修改visit()函数即可实现不同应用背景下的遍历。（2）存储操作，线性表使用二进制的方式进行存储，存储效率高，速度快。树存储时将包括空结点的树的数据域前序遍历结果输出到文件中，节省空间。（3）在调用功能函数前已对线性表的初始条件进行检验，确保程序不会异常退出。（4）使用线性表实现了对多二叉树进行操作。（5）运用了递归，栈，队列。

## 3.2 系统设计

3.2.1 系统总体设计：

在程序中实现消息处理与二叉树基本运算的演示，包括数据的输入和输出，程序的退出，并将数据以文件的形式存储。

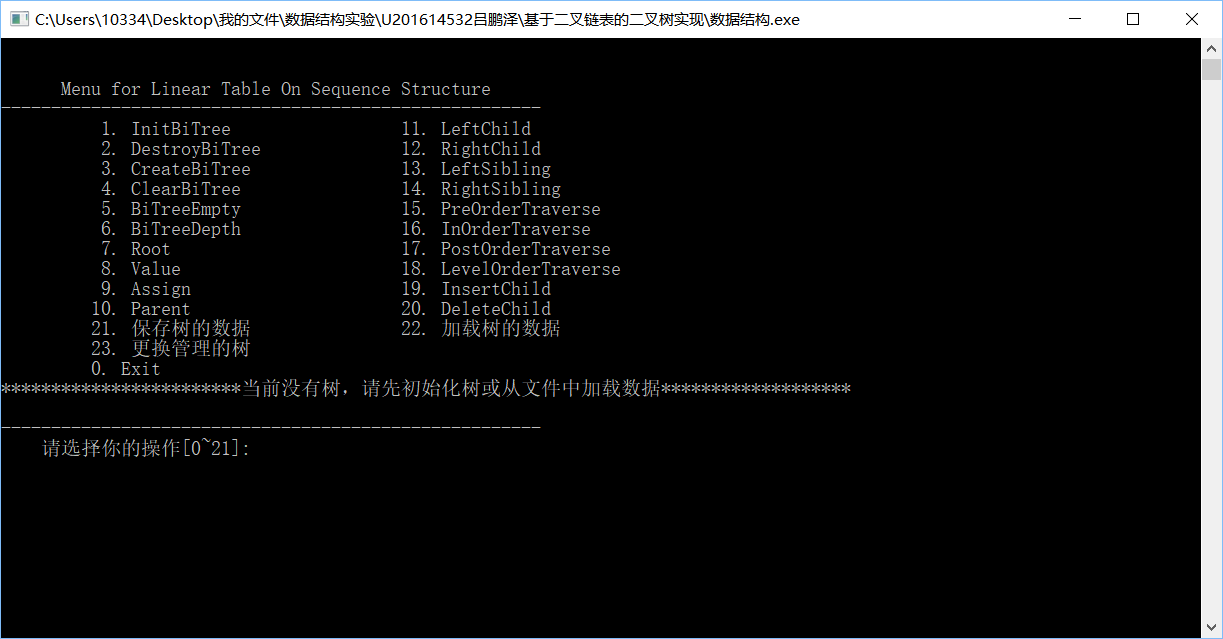


图3-1系统载入界面

用户通过输入在[0,23]区间的整数选择相应的功能，输入0退出程序，其他数字进行相应功能的演示。打开程序后是没有创建二叉树的，用户可以选择功能1创建一棵空二叉树或者选择功能22从数据文件中加载上次保存的数据。此外，只有当不存在二叉树时才能从文件中读取数据。当已存在二叉树时，再次调用功能1可以继续创建二叉树，即可实现多二叉树的管理，功能23可以选择管理的二叉树。

演示的过程可抽象为如图3-2所示的流程图。



图3-2系统总体结构

3.2.2二叉树的物理结构

如图3-3，表示的是多二叉树管理的物理结构，顺序表L记录着不同的二叉树的信息，即二叉树的名称和其根节点的指针，根节点指针指向二叉树的根节点。对于每一棵二叉树来说，其结点有两个指针，一个数据域，两个指针分别指向其左孩子和右孩子，数据域记录着树节点的关键字和具体数据。



图3-3 多二叉树管理的物理结构示意图

3.2.3相关常量的类型与定义

1.函数返回状态定义：

函数运行成功返回TRUE，失败返回FALSE，正常执行完毕返回OK，异常结束返回ERROR，动态分配空间不足返回OVERFLOW。在我的程序中用C语言描述如下所示：

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

2.相关常量

LIST\_INIT\_SIZE 为顺序表大小，最多可管理50棵二叉树。QUEUESIZE为循环队列大小，在实现按层遍历时用到。L\_DATA\_NAME为顺序表数据文件名称，T\_DATA\_NAME为二叉树数据文件名称。C语言描述如下所示：

#define LIST\_INIT\_SIZE 50

#define QUEUESIZE 100

#define L\_DATA\_NAME "ListData"

#define T\_DATA\_NAME "TreeData"

3.相关结构体定义

（1）顺序表结构

SqList记录顺序表信息，包括顺序表首地址，表长，表大小。每一个顺序表的结点记录一颗二叉树的信息，包括二叉树的名称和二叉树的根节点地址。C语言描述如下所示：

typedef struct {

struct treedata \* pT;

int listlenth;

int listsize;

}SqList;

typedef struct treedata {

struct bitnode \* pRoot;

char name[20];

}TreeData;

（2）树结点结构

树结点包括两个指针和一个数据域，两个指针分别指向其左右孩子，数据域抽象为ElemType结构体，包括树结点的关键字和数据，关键字为KEY类型，数据为TElemtype类型，我的程序中将关键字设为int型，数据设为char型。C语言描述如下所示：

typedef char TElemtype;//树结点的数据类型

typedef int KEY;//树结点的关键字类型

typedef struct elemType

{

KEY key;

TElemtype e;

}ElemType; //树结点中数据元素类型定义

typedef struct bitnode {

struct bitnode \* lchild;

ElemType data;

struct bitnode \* rchild;

}BiTNode, \*BiTree;//二叉链表结点定义

3.2.4算法设计

（1）InitBiTree(&T)

操作结果：构造空二叉树T。

算法思想：1.将根节点指针置空；2.管理二叉树的线性表表长加一。

操作结果：构造一个空的二叉树。

时间复杂度：O(1) （说明：以下时间复杂度均指平均时间复杂度,n为树结点的个数）。

流程图：



图3-4 InitBiTree(&T)流程图

（2）DestroyBiTree(&T)

初始条件：二叉树T已存在。

操作结果：销毁二叉树T。

算法思想：1.递归释放左右子树结点，最后释放双亲结点。2.管理二叉树的线性表同步删除该树。

时间复杂度：O(n)

流程图：



图3-5 DestroyBiTree(&T)流程图

（3）CreateBiTree(&T,definition)

初始条件：二叉树T已存在。

操作结果：按definition构造二叉树T。

算法思想：按照前序遍历构造二叉树，空结点用0 #表示。

时间复杂度：O(n)

流程图：



图3-6 CreateBiTree(&T)流程图

（4）ClearBiTree (&T)

初始条件：二叉树T已存在。

操作结果：将二叉树T清空。

算法思想：1.递归释放左右子树结点，最后释放双亲结点。2.根节点指针置空。

时间复杂度：O(n)

流程图：



图3-7 ClearBiTree(&T)流程图

（5）BiTreeEmpty(T)

初始条件：二叉树T已存在。

操作结果：若T为空二叉树，则返回TRUE,否则返回FALSE.

算法思想：判断二叉树T是否为空

时间复杂度：O(1)

流程图：



图3-8 BiTreeEmpty(T)流程图

（6）BiTreeDepth(T)

初始条件：二叉树T已存在。

操作结果：返回T的深度

算法思想：1.当T有子树时，T的深度为max（T的左子树深度，T的右子树深度）+1；2.对T的左右子树重复步骤1。

时间复杂度：O(n)

流程图：



图3-9 BiTreeDepth(T)流程图

（7）Root(T)

初始条件：二叉树T已存在。

操作结果：返回T的根。

算法思想：返回T的根

时间复杂度：O(1)

流程图：



图3-10 BiTreeDepth(T)流程图

（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的地址。

算法思想：利用栈迭代查找T中是否有关键字为e的结点

时间复杂度：O(n)

流程图：



图3-10 Value(T，e)流程图

（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。

算法思想：1.利用栈迭代查找T中关键字为e的结点。2.将关键字为e的结点的数据域赋值为value。

时间复杂度：O(n)

流程图：



图3-11 Assign(T，e，value)流程图

（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。

算法思想：1.判断根节点的关键字是否为e。2.利用栈迭代查找T中关键字为e的结点的双亲结点。

时间复杂度：O(n)

流程图：



图3-12 Parent(T，e)流程图

（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。

算法思想：1.利用栈迭代查找关键字为e的结点p。2.判断p是否有左孩子。

时间复杂度：O(n)

流程图：



图3-13 LeftChild(T，e)流程图

（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。

算法思想：1.利用栈迭代查找关键字为e的结点p。2.判断p是否有右孩子。

时间复杂度：O(n)

流程图：



图3-14 RightChild(T，e)流程图

（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，返回NULL。

算法思想：1.利用栈迭代查找e的双亲结点p。2.判断e是否有左兄弟

时间复杂度：O(n)

流程图：



图3-15 LeftSibling(T，e)流程图

（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。

算法思想：1.利用栈迭代查找e的双亲结点p。2.判断e是否有右兄弟

时间复杂度：O(n)

流程图：



图3-16 RightSibling(T，e)流程图

（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。

算法思想：当树C右子树为空时，根据LR，其值为0，将C插为左子树，当LR为1时，将C插为右子树，将断开部分连接为C的右子树。

时间复杂度：O(1)

流程图：



图3-17 InsertChild(T,p,LR,c)流程图

（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。

算法思想：对p执行一次Destroy()函数。

时间复杂度：O(n)（n为p的结点个数）

流程图：



图3-18 DeleteChild(T，p，LR)流程图

（17）PreOrderTraverse（T,Visit（））

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。

算法思想：1.先visit()T的根，其次visit()T的左子树，在visit()T的右子树2.对于T的左右子树，重复步骤1。

时间复杂度：O(n)

流程图：



图3-19 PreOrderTraverse（T,Visit（））流程图

（18）InOrderTraverse（T,Visit（））

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。

算法思想：1.先visit()T的左子树，其次visit()T的根，在visit()T的右子树2.对于T的左右子树，重复步骤1。

时间复杂度：O(n)

流程图：



图3-20 InOrderTraverse（T,Visit（））流程图

（19）PostOrderTraverse（T,Visit（））

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。

算法思想：1.先visit()T的左子树，其次visit()T的右子树，在visit()T的根2.对于T的左右子树，重复步骤1。

时间复杂度：O(n)

流程图：



图3-21 PostOrderTraverse（T,Visit（））流程图

（20）LevelOrderTraverse（T,Visit（））

操作结果：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。

算法思想：利用队列，先将根节点入队，然后队列顺序出队，若出队元素的左右子树存在，将它们入队，之后对该元素调用visit函数，直到队列为空

时间复杂度：O(n)

流程图：



图3-22 LevelOrderTraverse（T,Visit（））流程图

（21）SaveData(L)

操作结果：从文件中加载数据

算法思想：对顺序表使用fwrite()保存数据，对二叉树将数节点的值（包括空结点）按照前序遍历结果输出到文件中。

时间复杂度：O(m+n)（m为二叉树的个数）

流程图：



图3-23 SaveData(L)流程图

（22）LoadData(&L)

操作结果：将二叉树林的数据保存到文件中

算法思想：对顺序表使用fread()加载数据，对二叉树使用CreateBiTree()加载数据。

时间复杂度：O(m+n)（m为二叉树的个数）

流程图：



图3-24 LoadData(&L)流程图

## 3.3 系统实现

说明：①.在执行CreateBiTree()功能时，需要用户按照前序遍历结果给出二叉树的定义definition，包括二叉树每一个结点的信息：输入格式为关键字(int) 数据(char)，遇到空结点时用0 #表示。

②.数据文件中保存着3棵二叉树（tree1，tree2，tree3）的信息，如图a所示，tree1为一颗拥有6个节点的完全二叉树，如图b，tree2为一颗除叶结点外其余节点只有左孩子、深度为5的二叉树，如图c，tree3为一颗空二叉树。在之后的程序测试中基于这3棵树进行测试。



a.tree1



b.tree2



c.tree3

图3-25 数据文件中保存的树物理结构示意图

1.初始界面

用户打开程序后可以看到如下界面，输入数字选择相应功能函数，输入0退出程序。



图3-26 系统载入界面

2.创建表

如图a，在二叉树未创建前无法对表进行操作，需要初始化树或者从文件中读取数据。如图b，二叉树创建成功。



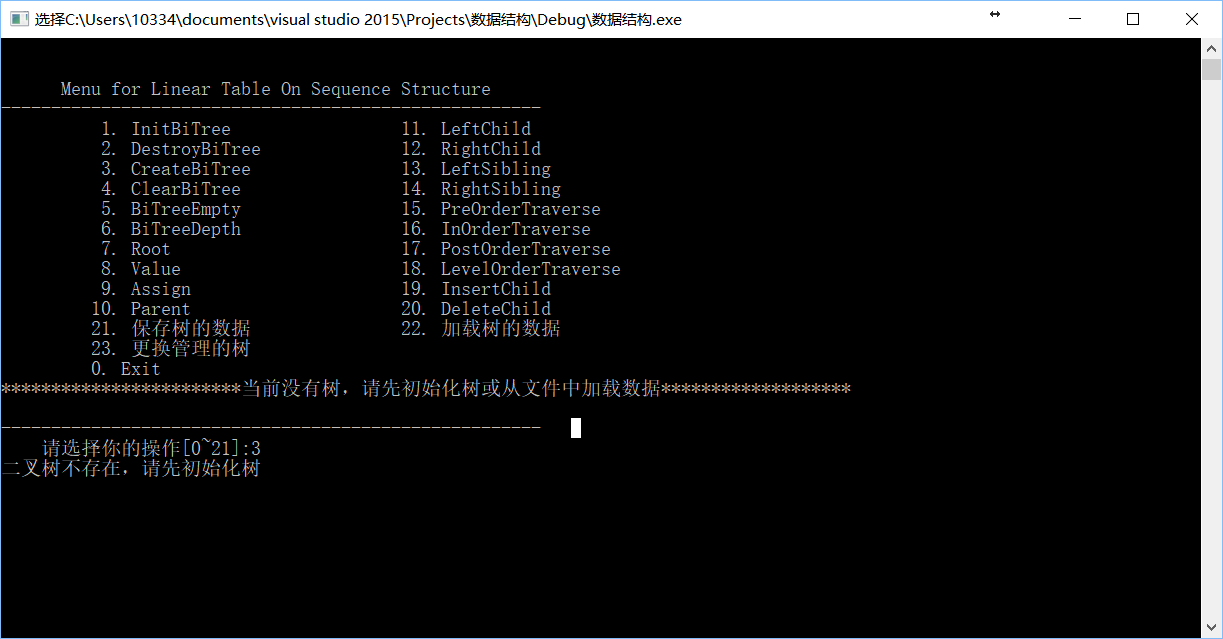
a



图3-27 创建树操作演示

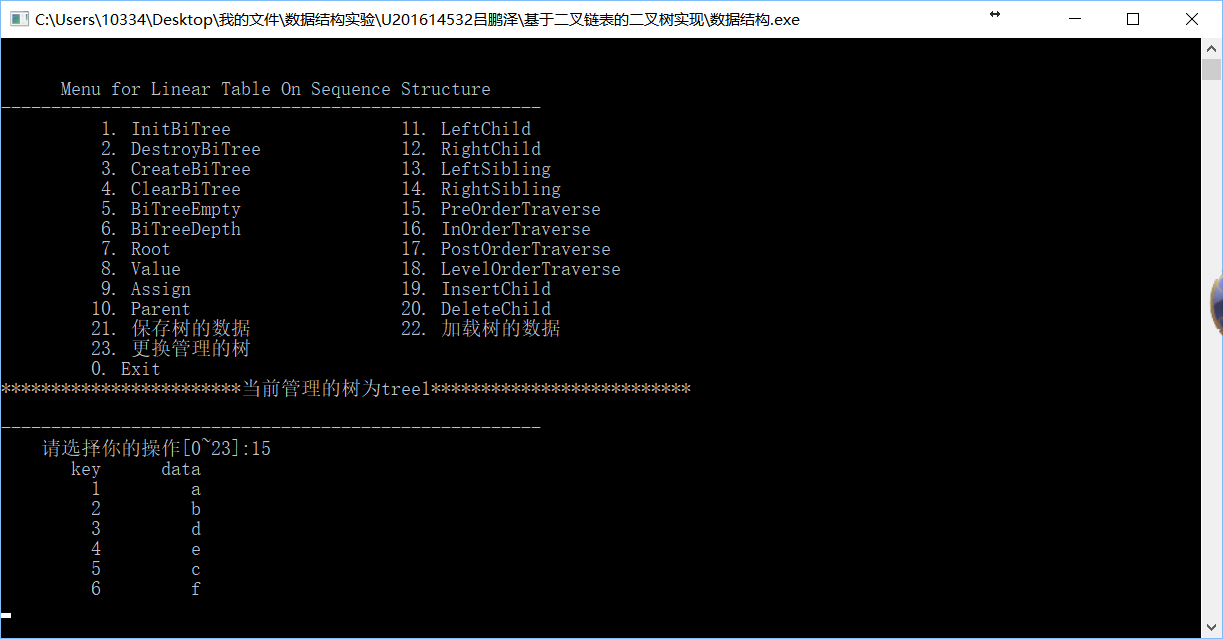
3.从文件中加载数据及遍历操作的演示

创建二叉树后表中不含数据，如图a，无法调用功能函数。如图b，从文件中加载数据后再次前序遍历tree1，结果如图c。



a

b

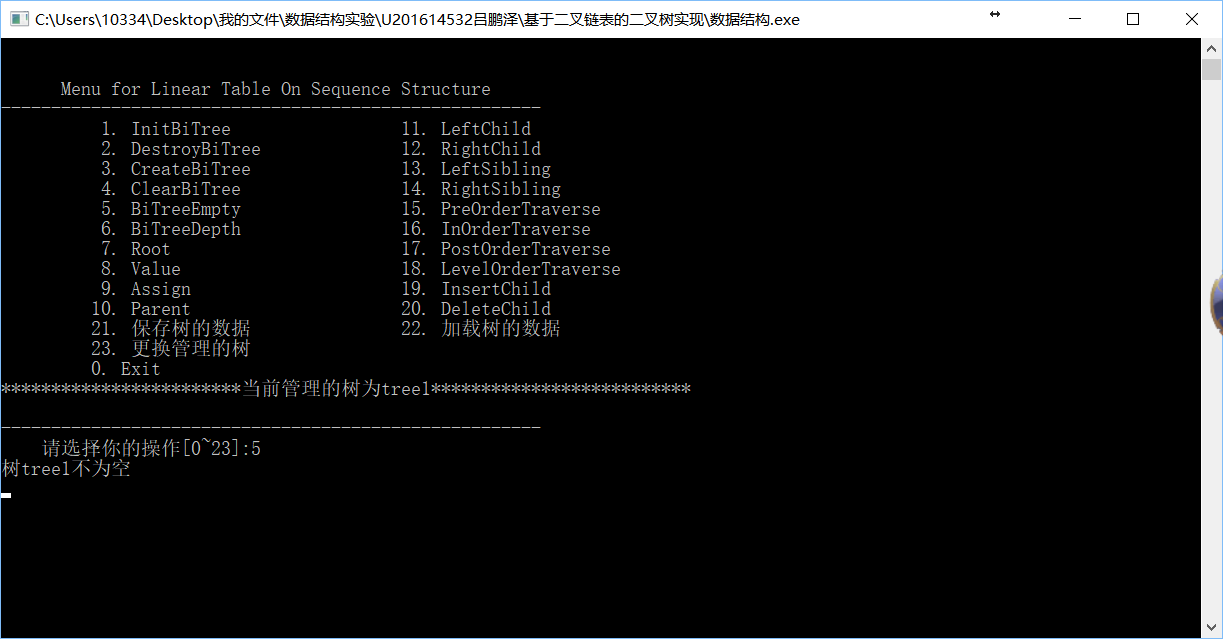


c

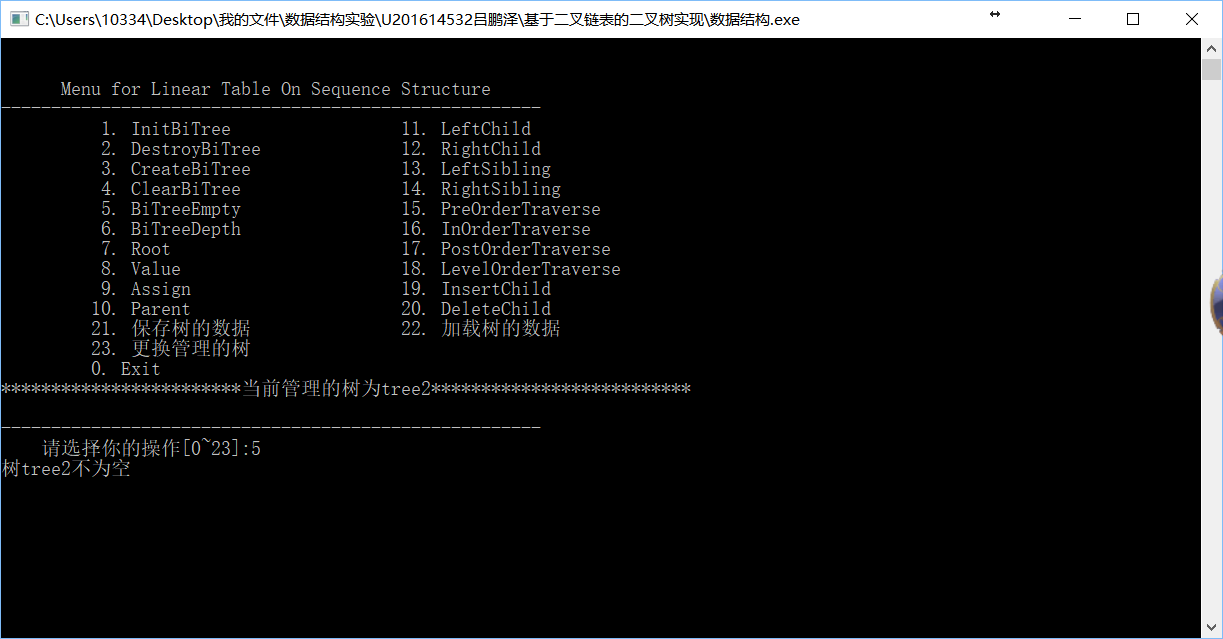
图3-28 加载数据的演示

4.判表空及置空表

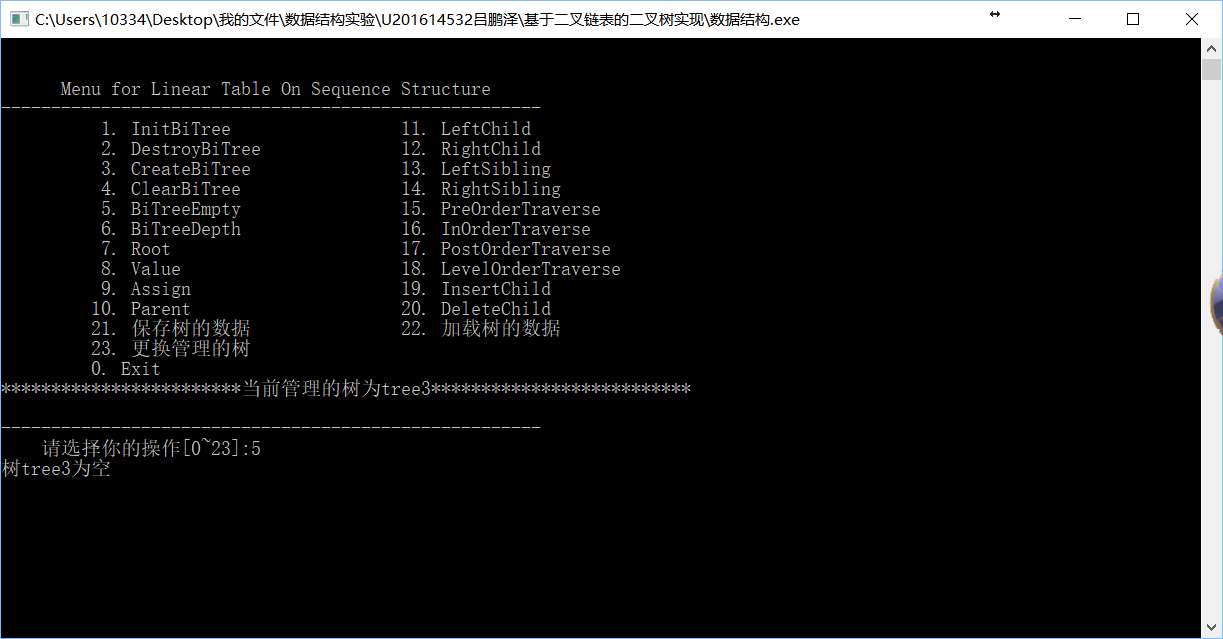
如图a，判空tree1。如图b，判空tree2。如图c，判空tree3。如图d，清空tree1后再次判空



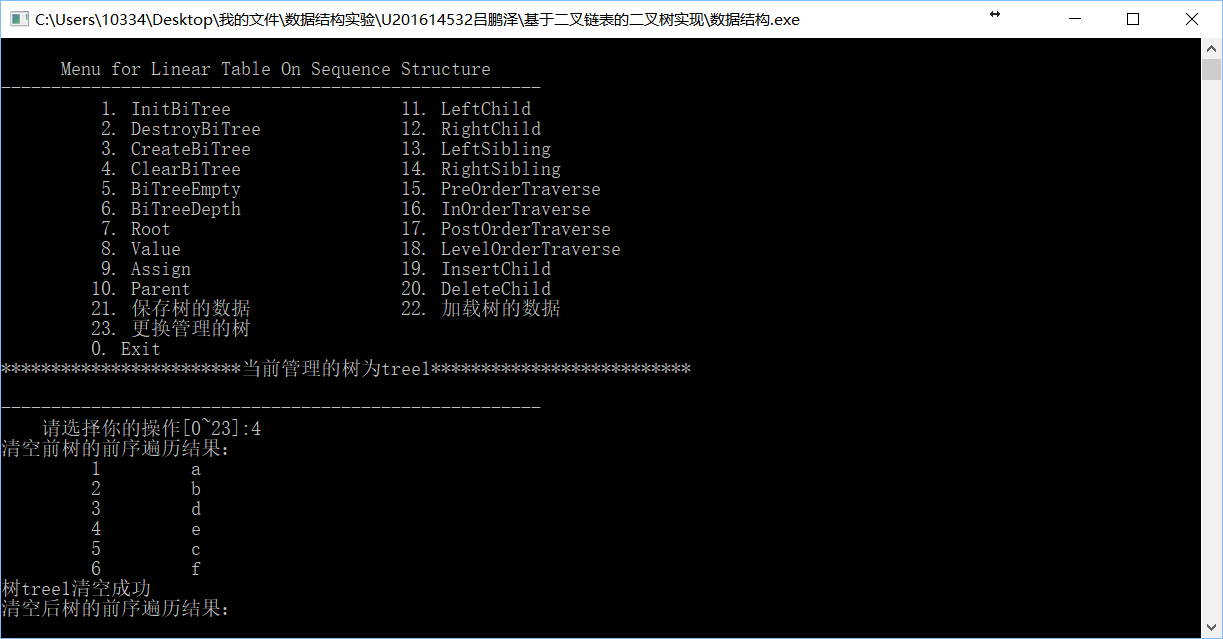
a

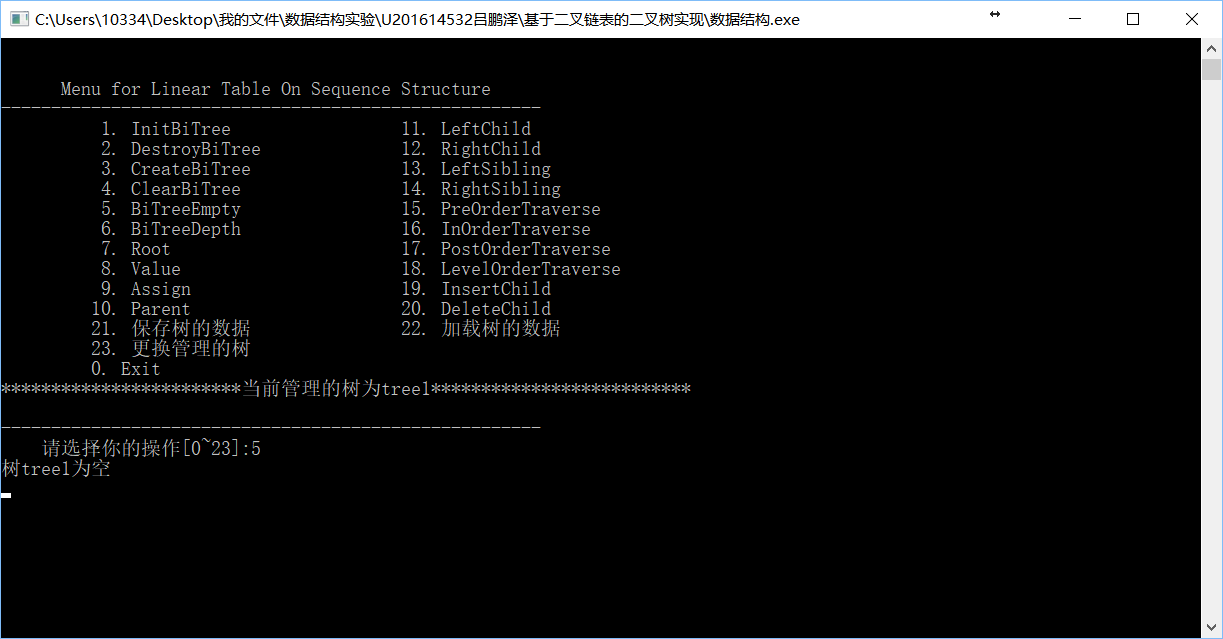


b



c



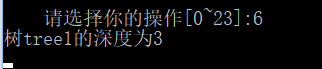


d

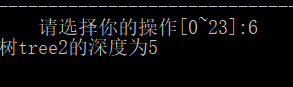
图3-28 判空二叉树及清空二叉树演示

5.求树深

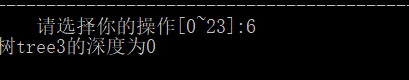
如图a，tree1的深度。如图b，tree2的深度。如图c，tree3的深度



a



b

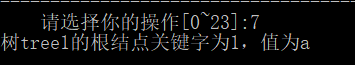
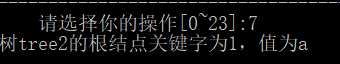


c

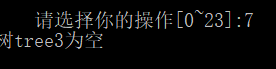
图3-29 求树深

6.获取树根

如图a，tree1的树根。如图b，tree2的树根。如图c，tree3的树根

a b

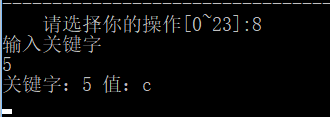
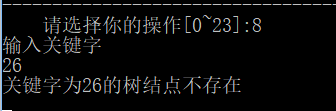


c

图3-30 获取元素

7.获取树结点

如图a，为元素tree1中的查找结果。如图b，为元素不在tree1中的查找结果。

a b

图3-31 获取树结点

8.为树结点赋值

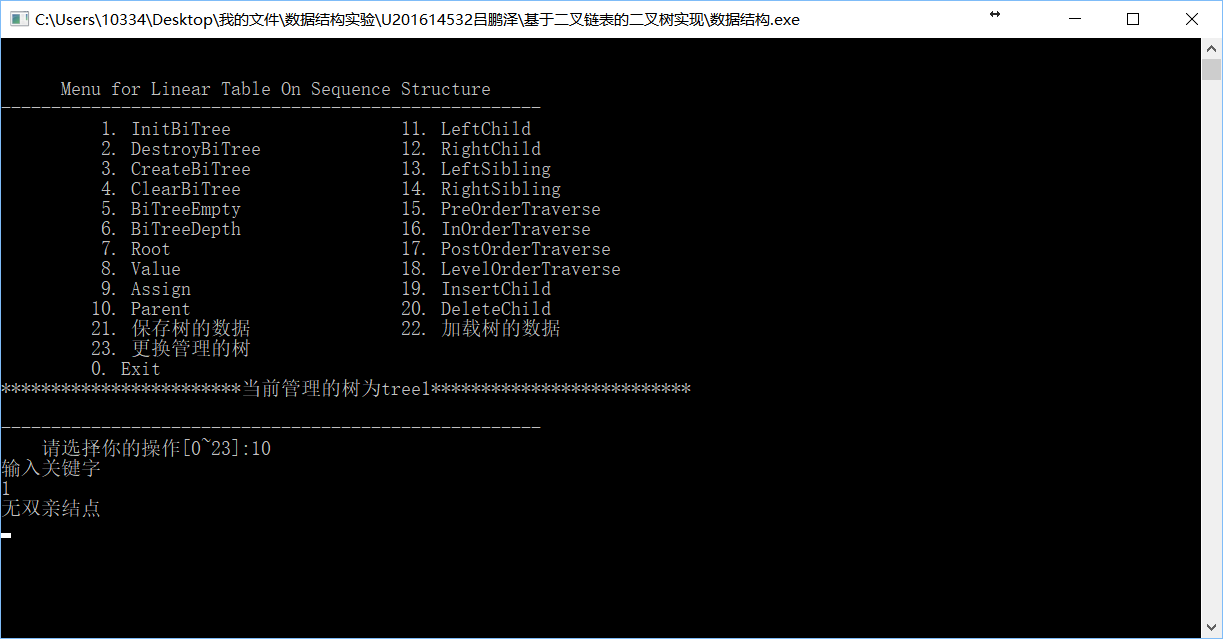
如图3-32，将tree1中关键字为2的结点的元素值改为x



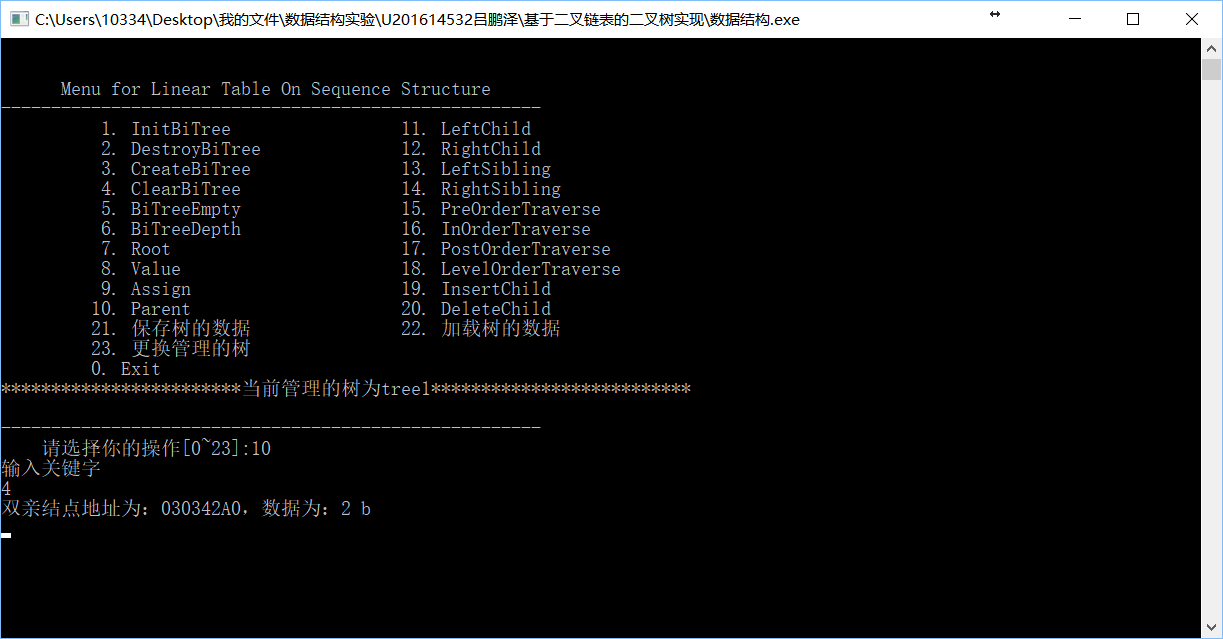
图3-32 赋值树结点

9.获取双亲结点

如图a，获取tree1根节点的双亲结点。如图b，获取tree1关键字为4的结点的双亲结点。



a

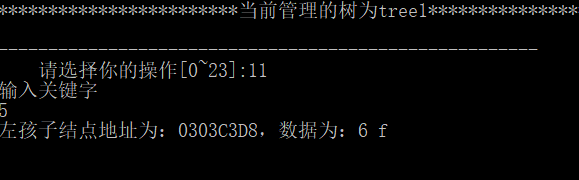


b

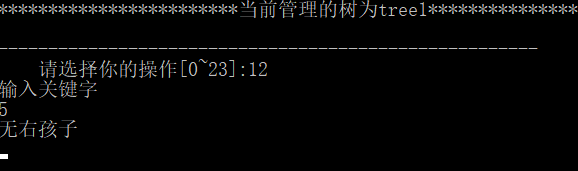
图3-33 获取双亲结点

10.获取左孩子和右孩子

如图a，获取tree1关键字为5结点的左孩子。如图b，获取tree1关键字为5结点的右孩子。



a

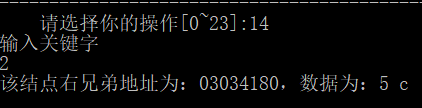
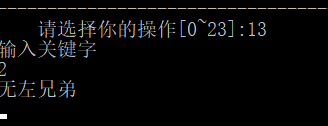


b

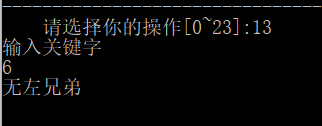
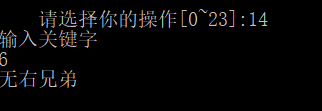
图3-34 获取孩子结点

11.获取兄弟结点

如图a，tree1关键字为2结点的左右兄弟结点。如图b获取关键字为6结点的左右兄弟结点。



a

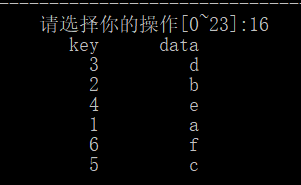
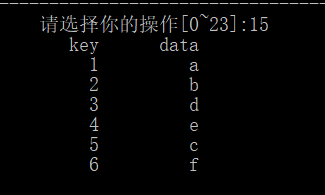
 

b

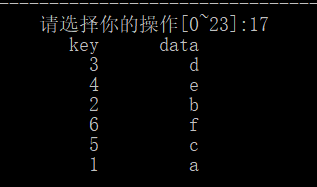
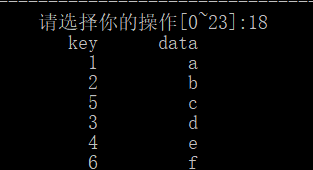
图3-35 获取兄弟结点

13.二叉树遍历

如图a、b、c、d，分别为二叉树的前、中、后、层序遍历结果

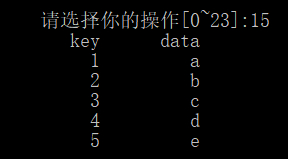
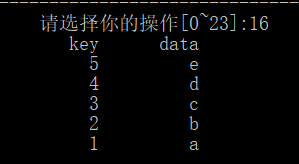


a b

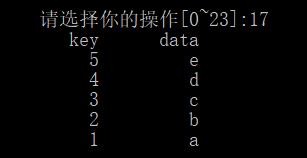
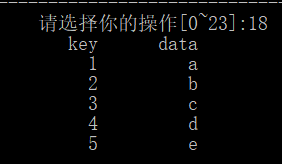
 

c d

图3-36 二叉树tree1遍历

a b

c d

图3-37 二叉树tree2遍历

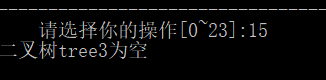


图3-38 二叉树tree3遍历

14.插入子树

将子树插入到tree1关键字为5结点的右子树中

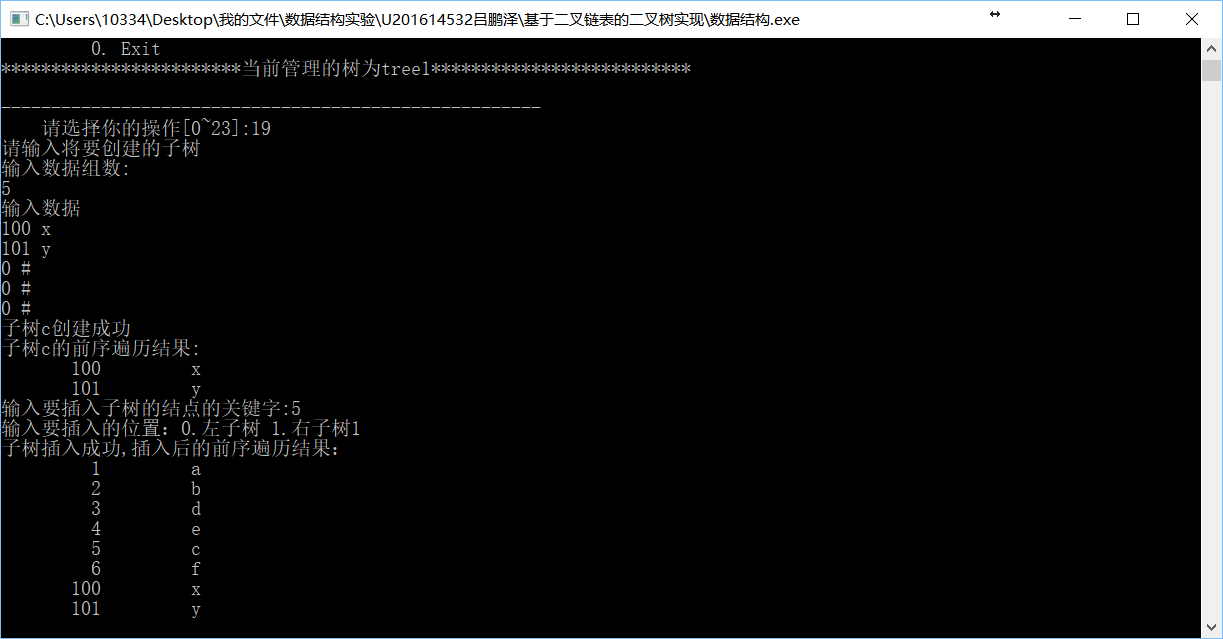


图3-39 插入子树

15.删除子树

删除tree1关键字为1的左子树

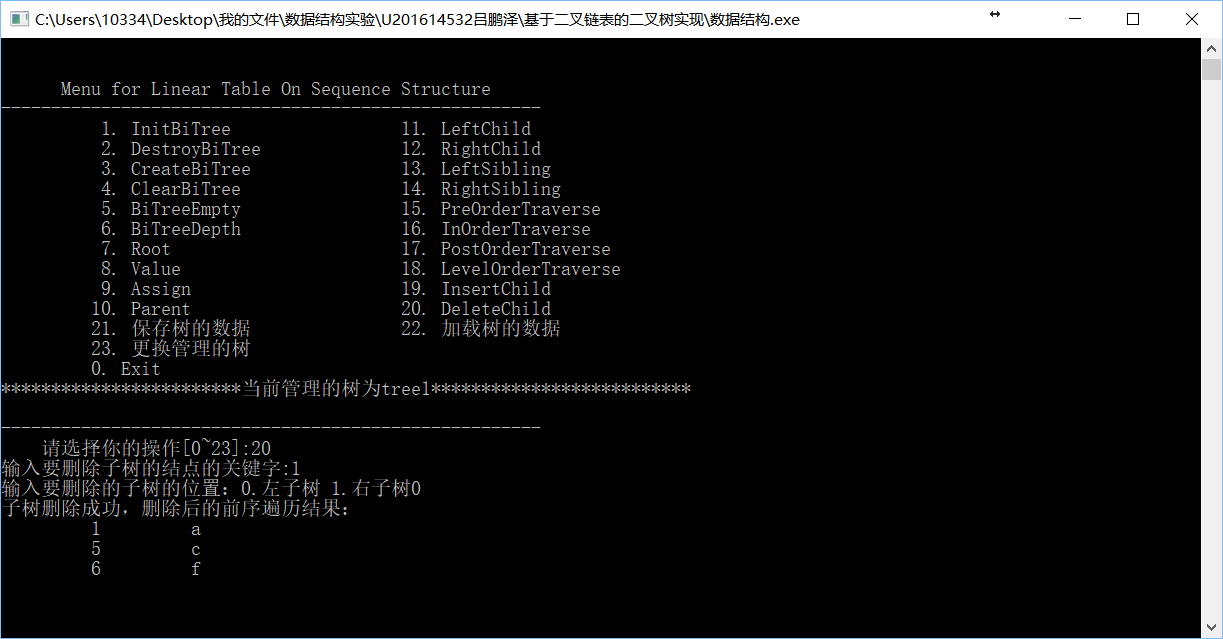


图3-40 删除子树

## 3.4 实验小结

本次实验写起来不是很容易，相比与上两次的实验难度有了很大的提高，主要体现在递归和栈的应用上，不过在百度和老师同学的帮助下我最终还是完成了。本次实验不仅加深了我对二叉树的理解，另外在写二叉树的功能函数还运用了递归、栈、队列、顺序表的思想，让我对数据结构整体有了更深的体会，可以联想，在解决具体的实际问题时也会是多种数据结构混合使用的。在实验过程中，我初步掌握了用栈实现递归的方法，将递归化为了迭代，这样不仅提高了程序的执行效率，同时也避免了栈溢出的风险。在写层遍历二叉树时，我运用了循环队列，加深了我对队列的理解。总之，通过这次实验我的编程能力得到了提高。

# 4 基于邻接表的图实现

## 4.1 问题描述

## 4.2 系统设计

## 4.3 系统实现

## 4.4 实验小结

# 参考文献

[1] 严蔚敏等. 数据结构(C语言版). 清华大学出版社

[2] [Larry Nyhoff](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

指导教师评定意见

一、对实验报告的评语

|  |
| --- |
|  |

二、对实验报告评分

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 评分项目  (分值) | 程序内容  (36.8分) | 程序规范  (9.2分) | 报告内容  (36.8分) | 报告规范  (9.2分) | 考勤  （8分） | 逾期扣分 | 合 计  (100分) |
| 得分 |  |  |  |  |  |  |  |

# 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>

#include <memory.h>

/\*--------------------page 10 on textbook ----------------------------------\*/

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

#define FILENAME "data"//数据保存的文件名

typedef int status;

typedef int ElemType; //数据元素类型定义

/\*--------------------page 22 on textbook ----------------------------------\*/

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

typedef struct{ //顺序表（顺序结构）的定义

ElemType \* elem;

int length;

int listsize;

}SqList;

/\*--------------------page 19 on textbook ----------------------------------\*/

status InitiaList(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 LocatElem(SqList L, ElemType e, status(\*compare)(ElemType x, ElemType y));

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(\*visit)(ElemType e));

status compare(ElemType x, ElemType y);

status Loaddata(SqList &L);//加载数据

status Savedata(SqList L);

status visit(ElemType e);

/\*------------------------------------------------------\*/

void main(void){

SqList L;

L.elem = NULL;

int i;//插入位置

int op = 1;

int flat = 1;//表不存在

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(" 13. 从文件中读取数据\n");

printf(" 14. 保存数据至文件\n");

printf(" 0. Exit\n");

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

printf(" 请选择你的操作[0~14]:");

scanf("%d", &op);

getchar();

switch (op) {

case 1:

if (!L.elem&&flat)//若线性表不存在，则创建

{

if (InitiaList(L) == OK) {

printf("线性表创建成功！\n");

flat = 0;//表已创建

}

else printf("线性表创建失败！\n");

}

else printf("线性表已存在\n");

getchar(); getchar();

break;

case 2:

if (L.elem)//若表存在

{

DestroyList(L);

printf("表销毁成功\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 3:

if (L.elem)//若表存在

{

ClearList(L);

printf("表置空成功\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 4:

if (L.elem)//若表存在

{

if (ListEmpty(L) == 1)

printf("表为空\n");

else printf("表不为空\n");

}

else printf("表不存在\n");

getchar();

break;

case 5:

if (L.elem)//若表存在

{

printf("表长为%d\n", ListLength(L));

}

else printf("表不存在\n");

getchar();

break;

case 6:

ElemType e;

if (L.elem)//表存在

{

printf("输入元素序号\n");

scanf(" %d", &i);

getchar();

if (i >= 1 && i <= ListLength(L))//若i值有效

{

GetElem(L, i, e);//获取位序为i的元素

printf("位序%d的元素值为%d", i, e);//输出获取的元素值

}

else printf("i值无效\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 7:

if (L.elem)

{

ElemType e;

printf("输入一个数字，为e赋值\n");

scanf("%d", &e);

getchar();

//输出第一个与e满足compare关系的元素位序，不存在位序为0

int order = LocatElem(L, e, compare);

if (order == 0)

printf("表中不存在与e相同的元素\n");

else printf("第一个与e相同的元素位序是%d\n", order);

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 8:

if (L.elem)//若表存在

{

ElemType cur\_e, pre\_e;

printf("输入一个数字，为cur\_e赋值\n");

scanf("%d", &cur\_e);

getchar();

if (PriorElem(L, cur\_e, pre\_e) == 1)//如果cur\_e存在前驱

printf("cur\_e的前驱为%d\n", pre\_e);

else printf("cur\_e的前驱不存在\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 9:

if (L.elem)//若表存在

{

ElemType cur\_e, next\_e;

printf("输入一个数字，为cur\_e赋值\n");

scanf("%d", &cur\_e);

getchar();

if (NextElem(L, cur\_e, next\_e) == 1)//如果cur\_e存在后继

printf("cur\_e的后继为%d\n", next\_e);

else printf("cur\_e的后继不存在\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 10:

if (L.elem)//若表存在

{

printf("输入要插入的位置\n");

scanf("%d", &i);

getchar();

if (i >= 1 && i <= ListLength(L) + 1)//i值合法

{

ElemType e;//插入的元素

printf("输入一个数字，为e赋值\n");

scanf("%d", &e);

getchar();

if (ListInsert(L, i, e))

printf("插入成功\n");

else printf("插入失败\n");

}

else printf("位置i不合法\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 11:

if (L.elem)//表存在

{

printf("输入要删除的数据元素位序\n");

scanf("%d", &i);

getchar();

if (i >= 1 && i <= ListLength(L))

{

ElemType e;//删除的元素存放在e中

getchar();

if (ListDelete(L, i, e))

{

printf("删除成功\n");

printf("删除的元素值为%d", e);

}

else printf("删除失败\n");

}

else printf("位置i不合法\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 12:

if (L.elem)//若表存在

{

if (!ListTrabverse(L, visit)) printf("线性表是空表！\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 13:

if (L.elem)//若表存在

{

if (Loaddata(L))

printf("数据加载成功\n");

else printf("数据加载失败\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 14:

if (L.elem)//若表存在

{

if (Savedata(L))

printf("数据保存成功\n");

else printf("数据保存失败\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 0:

break;

}//end of switch

}//end of while

printf("欢迎下次再使用本系统！\n");

}//end of main()

/\*--------------------page 23 on textbook ----------------------------------\*/

status InitiaList(SqList & L){

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

if(!L.elem) exit(OVERFLOW);//如果内存不足创建失败，则异常退出

L.length=0;

L.listsize=LIST\_INIT\_SIZE;

return OK;//创建成功

}

status DestroyList(SqList & L) {

free(L.elem);//释放空间

L.elem = NULL;//表址指向空

L.length = L.listsize = 0;//表厂及表大小置0

return TRUE;//销毁成功

}

status ClearList(SqList & L) {

L.length = 0;//置表长为0

return TRUE;

}

status ListEmpty(SqList L) {

if (L.length == 0)

return TRUE;//如果L为空表(即表长==0)，返回TRUE;

else return FALSE;//否则返回FALSE;

}

status ListLength(SqList L) {

return L.length;//返回元素个数

}

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

e = L.elem[i - 1];//使用取址公式定位第i个元素并赋值给e

return OK;

}

status LocatElem(SqList L, ElemType e, status(\*compare)(ElemType x, ElemType y))

{

int i;

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

{

if (compare(L.elem[i], e) == 0)//compare函数比较两个参数，相同返回0

return i + 1;//返回第一个与e满足关系compare的数据元素的位序

}

return FALSE;//不存在返回0

}

status PriorElem(SqList L, ElemType cur, ElemType & pre\_e) {

int order = LocatElem(L, cur, compare);//查找cur元素获取其序号（cur不在表中order为0）

if (order > 1)//如果cur在L中且不为第一个元素

{

pre\_e = L.elem[order - 2];//cur的前驱值赋给&pre\_e

return TRUE;

}

return FALSE;

}

status NextElem(SqList L, ElemType cur, ElemType & next\_e) {

int order = LocatElem(L, cur, compare);//查找cur元素获取其序号

if (order >= 1 && order<L.length)//如果cur在L中且不为最后一个元素

{

next\_e = L.elem[order];//cur的后继值赋给&next\_e

return TRUE;

}

return FALSE;

}

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

if (L.length + 1 >= L.listsize)//空间已满，增配空间

{

ElemType \* newbase;

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

if (newbase == NULL)

return FALSE;//分配失败

L.elem = newbase;//新基址

L.listsize += LISTINCREMENT;//增加容量

}

//i之后的元素依次后移

ElemType \*in\_i = &(L.elem[i - 1]);//in\_i指向将要插入的位置

ElemType \*cur;

for (cur = &(L.elem[L.length - 1]); cur >= in\_i; cur--)

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

\*in\_i = e;//i处插入e

L.length++;//表长加1

return TRUE;

}

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

{

e = L.elem[i - 1];//赋值e

ElemType \*tail = &L.elem[L.length - 1];//tail指向表尾的位置

ElemType \*cur;

for (cur = &(L.elem[i - 1]); cur < tail; cur++)

\*cur = \*(cur + 1);//i之后的元素依次前移

L.length--;//表长减1

return TRUE;

}

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

int i;

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

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

{

if (!visit(L.elem[i])) return ERROR;//如果visit失败，则退出

}

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

return L.length;//遍历完毕，返回遍历的元素个数

}

status compare(ElemType x, ElemType y)

{/\*通过内存比较判断x和y元素,相等返回0，x<y返回0-，x>y返回0+\*/

return memcmp(&x, &y, sizeof(ElemType));

}

status Loaddata(SqList &L)

{

FILE \*fp;

L.length = 0;

if ((fp = fopen(FILENAME, "r")) == NULL)//文件打开失败

{

printf("File open error\n ");

return FALSE;

}

while (fread(&L.elem[L.length], sizeof(ElemType), 1, fp))//每次读取一个数据元素，顺序写入打顺序表中

{

if (L.length + 1 >= L.listsize)//空间已满，增配空间

{

ElemType \* newbase;

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

if (newbase == NULL)

return FALSE;//分配失败

L.elem = newbase;//新基址

L.listsize += LISTINCREMENT;//增加容量

}

L.length++;//准备写入下一个元素

}

fclose(fp);

return TRUE;//写入完毕

}

status Savedata(SqList L)

{

FILE \*fp;

if ((fp = fopen(FILENAME, "w")) == NULL)//打开文件失败

{

printf("File open erroe\n ");

return FALSE;

}

fwrite(L.elem, sizeof(ElemType), L.length, fp);//从地址L.elem开始，往后sizeof(ElemType)\*L.length个字节的数据一次性写入到文件中

fclose(fp);//关闭文件

return TRUE;

}

status visit(ElemType e)

{

printf("%d ", e);

return TRUE;

}

# 附录B 基于链式存储结构线性表实现的源程序

/\*含头结点的链式存储结构的线性表实现\*/

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include<string.h>

/\*--------------------page 10 on textbook ----------------------------------\*/

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

#define FILENAME\_L1 "dataL1"

#define FILENAME\_L2 "dataL2"

typedef int status;

typedef int Elemtype;

typedef struct lnode

{

Elemtype data;

lnode \* next;

}Lnode; //数据元素类型定义

typedef Lnode\* SqList;//头指针

status IntiaList(SqList \*L);//L为指向头指针的指针

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 \*x, Elemtype \*y));

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(\*visit)(Elemtype e));

status Loaddata(SqList \*L, char \*filename);//加载数据

status Savedata(SqList L, char \*filename);//保存数据

status visit(Elemtype e);

status compare(Elemtype \*x, Elemtype \*y);

SqList \* list\_verify(SqList \*L1, SqList \*L2);

/\*------------------------------------------------------\*/

void main(void) {

SqList L1, L2;

SqList \*L;

L1 = L2 = NULL;

int op = 1;

int i;

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(" 13. 从文件中读取数据\n");

printf(" 14. 保存数据至文件\n");

printf(" 0. Exit\n");

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

printf(" 请选择你的操作[0~14]:");

scanf("%d", &op);

getchar();

L = list\_verify(&L1, &L2);//确认需要操作的表

switch (op) {

case 1:

if (\*L == NULL)//若线性表不存在，则创建

{

if (IntiaList(L) == OK) printf("线性表创建成功！\n");

else printf("线性表创建失败！\n");

//printf("%p", (\*L)->next);

}

else printf("线性表已存在\n");

getchar(); getchar();

break;

case 2:

if (\*L)//若表存在

{

DestroyList(L);

printf("表销毁成功\n");

//printf("%p", \*L);

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 3:

if (\*L)//若表存在

{

ClearList(L);

printf("表清空成功\n");

//printf("%p", \*L);

//printf("%p", (\*L)->next);

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 4:

if (\*L)//若表存在

{

if (ListEmpty(\*L))

printf("线性表为空\n");

else printf("线性表不为空\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 5:

if (\*L)//若表存在

{

int len;

len = ListLength(\*L);

printf("表长为%d\n", len);

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 6:

if (\*L)//若表存在

{

Elemtype e;

printf("输入元素序号\n");

scanf(" %d", &i);

getchar();

if (i >= 1 && i <= ListLength(\*L))//若i值有效

{

GetElem(\*L, i, e);//获取位序为i的元素

printf("位序%d的元素值为:", i); visit(e);//输出获取的元素值

}

else printf("i值无效\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 7:

if (\*L)//若表存在

{

Elemtype e;

printf("输入一个数字，为e赋值\n");

scanf("%d", &e);

getchar();

//输出第一个与e满足compare关系的元素位序，不存在位序为0

int order = LocateElem(\*L, e, compare);

if (order == 0)

printf("表中不存在与e相同的元素\n");

else printf("第一个与e满足compare关系的元素位序是%d\n", order);

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 8:

if (\*L)//若表存在

{

Elemtype cur\_e, pre\_e;

printf("输入一个数字，为cur\_e赋值\n");

scanf("%d", &cur\_e);

getchar();

if (PriorElem(\*L, cur\_e, pre\_e) == 1)//如果cur\_e存在前驱

{

printf("cur\_e的前驱为:"); visit(pre\_e);

}

else printf("cur\_e的前驱不存在\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 9:

if (\*L)//若表存在

{

Elemtype cur\_e, next\_e;

printf("输入一个数字，为cur\_e赋值\n");

scanf("%d", &cur\_e);

getchar();

if (NextElem(\*L, cur\_e, next\_e) == 1)//如果cur\_e存在后继

{

printf("cur\_e的后继为:"); visit(next\_e);

}

else printf("cur\_e的后继不存在\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 10:

if (\*L)//若表存在

{

printf("输入要插入的位置\n");

scanf("%d", &i);

getchar();

if (i >= 1 && i <= ListLength(\*L) + 1)//i值合法

{

Elemtype e;//插入的元素

printf("输入一个数字，为e赋值\n");

scanf("%d", &e);

getchar();

if (ListInsert(\*L, i, e))

printf("插入成功\n");

else printf("插入失败\n");

}

else printf("位置i不合法\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 11:

if (\*L)//若表存在

{

printf("输入要删除的数据元素位序\n");

scanf("%d", &i);

getchar();

if (i >= 1 && i <= ListLength(\*L))//若i值合法

{

Elemtype e;//保存删除的结点数据

if (ListDelete(\*L, i, e))

{

printf("删除成功\n");

printf("删除的元素值为:"); visit(e);

}

else printf("删除失败\n");

}

else printf("位置i不合法\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 12:

if (\*L)//若表存在

{

if (!ListTrabverse(\*L, visit)) printf("线性表是空表！\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 13:

if (\*L == NULL || (\*L)->next == NULL)//若表不存在或表为空，则加载数据

{

int status;

if (\*L == L1)//选择L1

status = Loaddata(L, FILENAME\_L1);

else if (\*L == L2)//选择L2

status = Loaddata(L, FILENAME\_L2);

if (status)

printf("数据加载成功\n");

else printf("数据加载失败\n");

}

else printf("表已存在，请销毁或清空后加载数据\n");

getchar(); getchar();

break;

case 14:

if (\*L)//若表存在

{

int status;

if (\*L == L1)//选择L1

status = Savedata(\*L, FILENAME\_L1);

else if (\*L == L2)//选择L2

status = Savedata(\*L, FILENAME\_L2);

if (status)

printf("数据保存成功\n");

else printf("数据保存失败\n");

}

else printf("表不存在\n");

getchar(); getchar();

break;

case 0:

break;

}//end of switch

}//end of while

printf("欢迎下次再使用本系统！\n");

}//end of main()

/\*--------------------page 23 on textbook ----------------------------------\*/

status IntiaList(SqList \* L) {

\*L = (Lnode \*)malloc(sizeof(Lnode));//头指针申请一个存储结点

if (\*L)//如果分配空间成功

(\*L)->next = NULL;//头结点指针域置空

else exit(OVERFLOW);//分配失败

return OK;

}

status DestroyList(SqList \* L)

{

Lnode \* L1;

Lnode \* temp;//存储上一个释放节点的next值

L1 = \*L;//L1用来遍历链表

for (L1 = \*L; L1 != NULL; L1 = temp)

{

temp = L1->next;//保存next值

free(L1);//依次释放线性表的内存空间

}

\*L = NULL;//头指针置空

return OK;

}

status ClearList(SqList \*L)

{

Lnode \* L1;

Lnode \* temp;//存储上一个释放节点的next值

L1 = \*L;//L1用来遍历链表

for (L1 = (\*L)->next; L1 != NULL; L1 = temp)//释放头结点之后的结点

{

temp = L1->next;//保存next值

free(L1);//依次释放线性表的内存空间

}

(\*L)->next = NULL;//头结点指针域置空

return OK;

}

status ListEmpty(SqList L)

{

if (L->next == NULL)//若头结点的指针域为空

return TRUE;

else return FALSE;

}

int ListLength(SqList L)

{

Lnode \*p;

int count;

for (p = L->next, count = 0; p != NULL; p = p->next)

count++;//p每后移一次，表长加1

return count;

}

status GetElem(SqList L, int i, Elemtype & e)

{

Lnode \*p;

int count;

for (p = L->next, count = 1; count != i; p = p->next, count++);//定位第i个数据元素

e = p->data;//赋值

return OK;

}

status LocateElem(SqList L, Elemtype e, status(\*compare)(Elemtype \*x, Elemtype \*y))

{

Lnode \*p;

int count;

for (p = L->next, count = 1; p != NULL; p = p->next, count++)//从首节点开始比较

if (compare(&(p->data), &e) == 0)

return count;

return FALSE;//遍历完毕，未找到

}

status PriorElem(SqList L, Elemtype cur, Elemtype & pre\_e)

{

Lnode \*p;

Lnode \*pr;//p的前驱

p = L->next;//p指向首节点

if (p == NULL || p->data == cur)//如果链表为空或cur是首节点，不存在前驱

return FALSE;

pr = p;//pr指向首节点

p = p->next;//p指向第二结点

while (p != NULL)

{

if (p->data == cur)//如果cur在表中

{

pre\_e = pr->data;//赋值

return TRUE;

}

pr = p;

p = p->next;

}

return FALSE;//表中未找到cur，cur无前驱

}

status NextElem(SqList L, Elemtype cur, Elemtype & next\_e)

{

Lnode \*p;//p

p = L->next;//p指向首节点

if (p == NULL)//如果链表为空

return FALSE;

while (p != NULL)

{

if (p->data == cur)//如果cur在表中

break;

p = p->next;

}

if (!p || !(p->next))//cur不在链表中或cur为最后一个结点

return FALSE;

else {

next\_e = (p->next)->data;//赋值

return TRUE;

}

}

status ListInsert(SqList L, int i, Elemtype e)

{

int count;//记录当前位置

Lnode \*p;//使用p遍历链表查找位置i

Lnode \*q;

for (p = L, count = 0; count < i - 1; p = p->next, count++);//定位，执行完毕后p指向位置i的前驱

q = (Lnode \*)malloc(sizeof(Lnode));//为插入的结点声明空间

q->data = e;//将e的数据域赋值给q的数据域

q->next = p->next;//插入结点

p->next = q;

return OK;

}

status ListDelete(SqList L, int i, Elemtype & e)

{

Lnode \*p, \*q;

int count;

for (p = L, count = 0; count < i - 1; p = p->next, count++);//定位，执行完毕后p指向位置i的前驱

q = p->next;//q指向删除结点

e = q->data;//赋值

p->next = q->next;//更改指针域

free(q);//删除结点

return OK;

}

status ListTrabverse(SqList L, status(\*visit)(Elemtype e)) {

if (L->next == NULL)

return 0;

Lnode \* p;

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

for (p = L->next; p != NULL; p = p->next)//使用visit访问首结点及其之后的结点

visit(p->data);

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

return 1;

}

status Loaddata(SqList \*L, char \*filename)

{

FILE \*fp;

if ((fp = fopen(filename, "r")) == NULL)//文件打开失败

{

printf("File open error\n ");

return FALSE;

}

if (\*L == NULL)//若表不存在，则创建表及头结点

\*L = (Lnode \*)malloc(sizeof(Lnode));

(\*L)->next = NULL;//头结点指针域置空

Lnode newEle;

Lnode \*p, \*q;

q = \*L;//q指向将要插入位置的前驱结点

while (fread(&newEle, sizeof(Lnode), 1, fp))//每次读取一个数据元素

{

p = (Lnode \*)malloc(sizeof(Lnode));//动态声明一个结点，作为新结点插入到链表中

\*p = newEle;//为新节点赋值

//插入结点

p->next = q->next;

q->next = p;

q = p;//q后移一个结点，进行下一次的插入

}

fclose(fp);

return TRUE;//写入完毕

}

status Savedata(SqList L, char \*filename)

{

FILE \*fp;

if ((fp = fopen(filename, "w")) == NULL)//打开文件失败

{

printf("File open error\n ");

return FALSE;

}

Lnode \*p;

p = L->next;//p指向首结点

while (p)

{

fwrite(p, sizeof(Lnode), 1, fp);//以二进制读写一次写入一个结点的数据

p = p->next;

}

fclose(fp);//关闭文件

return TRUE;

}

status visit(Elemtype e)

{

printf("%d ", e);

return TRUE;

}

status compare(Elemtype \*x, Elemtype \*y)

{

return (\*x) - (\*y);//相同返回0

}

SqList \* list\_verify(SqList \*L1, SqList \*L2)

{

char Listname[9];

do {

printf("\n 选择需要操作的表[L1,L2]: ");

scanf("%s", Listname);

} while (strcmp(Listname, "L1") && strcmp(Listname, "L2"));//只能在L1，L2中选择

SqList \*plist;

if (strcmp(Listname, "L1") == 0)

plist = L1;

else plist = L2;//plist为用户所选的表

return plist;

}

# 附录C 基于二叉链表二叉树实现的源程序

/\* Linear Table On Sequence Structure \*/

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

/\*--------------------page 10 on textbook ----------------------------------\*/

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

#define LIST\_INIT\_SIZE 50

#define QUEUESIZE 100 //队列的大小

#define L\_DATA\_NAME "ListData"

#define T\_DATA\_NAME "TreeData"

typedef int status;

typedef struct {

struct treedata \* pT;

int listlenth;

int listsize;

}SqList;//顺序表，用于管理二叉树

typedef struct treedata {

struct bitnode \* pRoot;

char name[20];

}TreeData;//二叉树信息(线性表的结点)

typedef char TElemtype;//树结点的数据类型

typedef int KEY;//树结点的关键字类型

typedef struct elemType

{

KEY key;

TElemtype e;

}ElemType; //树结点中数据元素类型定义

typedef struct bitnode {

struct bitnode \* lchild;

ElemType data;

struct bitnode \* rchild;

}BiTNode, \*BiTree;//二叉链表结点定义

status InitBiTree(BiTree \*T);

status DestroyBiTree(BiTree \*T);

status CreateBiTree(BiTree \*T, ElemType \* &definition);

//definition：以"关键字 字符"形式给出前序遍历结果，空树用"0 #"表示

status ClearBiTree(BiTree \*T);

status BiTreeEmpty(BiTree T);

status BiTreeDepth(BiTree T);

BiTNode \*Root(BiTree T);

BiTNode \*Value(BiTree T, KEY e);

status Assign(BiTree T, KEY e, ElemType value);

BiTNode \* Parent(BiTree T, KEY e);

BiTNode \* LeftChild(BiTree T, KEY e);

BiTNode \* RightChild(BiTree T, KEY e);

BiTNode \* LeftSibling(BiTree T, KEY e);

BiTNode \* RightSibling(BiTree T, KEY e);

status InsertChild(BiTree T, BiTNode \*p, int LR, BiTree c);

status DeleteChild(BiTree T, BiTNode \*p, int LR);

status PreOrderTraverse(BiTree T, void(\*visit)(ElemType e));

status InOrderTraverse(BiTree T, void(\*visit)(ElemType e));

status PostOrderTraverse(BiTree T, void(\*visit)(ElemType e));

status LevelOrderTraverse(BiTree T, void(\*visit)(ElemType e));

status SaveData(SqList L);

status SaveTree(BiTree T, FILE \*fp);

status LoadData(SqList &L);

status LoadTree(BiTree &T, FILE \*fp);

int SelectTree(SqList L);

void visit(ElemType e);

/\*------------------------------------------------------\*/

int main(void) {

SqList L;

L.pT = (TreeData \*)malloc(LIST\_INIT\_SIZE \* sizeof(TreeData));//线性表最多可管理LIST\_INIT\_SIZE棵树

L.listsize = LIST\_INIT\_SIZE;

L.listlenth = 0;

int mytree;

TreeData \*pMyTree = NULL;

int op = 1;

int i;

while (op) {

system("cls");

printf("\n\n");

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

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

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

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

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

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

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

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

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

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

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

printf(" 10. Parent 20. DeleteChild\n");

printf(" 21. 保存树的数据 22. 加载树的数据\n");

printf(" 23. 更换管理的树\n");

printf(" 0. Exit\n");

if (L.listlenth != 0) printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*当前管理的树为%s\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n", pMyTree->name);

else printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*当前没有树，请先初始化树或从文件中加载数据\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

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

printf(" 请选择你的操作[0~23]:");

scanf("%d", &op);

if (L.listlenth == 0 && (op != 1 && op != 22))

{

printf("二叉树不存在，请先初始化树\n");

getchar(); getchar();

continue;

}

switch (op) {

case 1:

{

if (L.listlenth >= L.listsize)

{

printf("空间不足，无法创建树\n");

break;

}

InitBiTree(&(L.pT[L.listlenth].pRoot));//在线性表表尾创建一棵树

printf("输入树的名称:\n");

scanf\_s("%s", L.pT[L.listlenth].name,20);

mytree = L.listlenth;

pMyTree = &L.pT[mytree];

printf("二叉树%s创建成功！\n", L.pT[L.listlenth].name);

L.listlenth++;//线性表同步创建一棵树

getchar(); getchar();

break;

}

case 2:

{

DestroyBiTree(&(pMyTree->pRoot));

printf("二叉树%s销毁成功！\n", pMyTree->name);

for (i = mytree; i < L.listlenth - 1; i++)

{

L.pT[i] = L.pT[i + 1];

}

L.listlenth--;//线性表同步删除一棵树

mytree = 0;

pMyTree = &L.pT[mytree];

getchar(); getchar();

break;

}

case 3:

{

ElemType def[500];//二叉树定义

ElemType \*definition = def;

int i;

i = 0;

for (i = 0; i < 500; i++)

{

def[i].e = '#';

def[i].key = 0;

}//初始树结点定义均为空

int N;

i = 0;

printf("输入数据组数:\n");

scanf("%d", &N);

if (N > 2000)

{

printf("数据量过大\n");

system("pause");

break;

}

printf("输入数据\n");

while (N--)

{

scanf("%d %c", &def[i].key, &def[i].e);//读取树定义

i++;

}

CreateBiTree(&pMyTree->pRoot, definition);

printf("树%s创建成功\n", pMyTree->name);

getchar(); getchar();

break;

}

case 4:

{

printf("清空前树的前序遍历结果：\n");

PreOrderTraverse(pMyTree->pRoot, visit);

ClearBiTree(&pMyTree->pRoot);

printf("树%s清空成功\n", pMyTree->name);

printf("清空后树的前序遍历结果：\n");

PreOrderTraverse(pMyTree->pRoot, visit);

getchar(); getchar();

break;

}

case 5:

{

if (BiTreeEmpty(pMyTree->pRoot))

printf("树%s为空\n", pMyTree->name);

else printf("树%s不为空\n", pMyTree->name);

getchar(); getchar();

break;

}

case 6:

{

printf("树%s的深度为%d\n", pMyTree->name, BiTreeDepth(pMyTree->pRoot));

getchar(); getchar();

break;

}

case 7:

{

if (pMyTree->pRoot) {//根节点非空

BiTNode \*p;

p = Root(pMyTree->pRoot);

printf("树%s的根结点关键字为%d，值为%c\n", pMyTree->name, p->data.key, p->data.e);

}

else printf("树%s为空\n", pMyTree->name);

getchar(); getchar();

break;

}

case 8:

{

if (pMyTree->pRoot)//根节点非空

{

BiTNode \*p;

KEY e;

printf("输入关键字\n");

scanf(" %d", &e);

p = Value(pMyTree->pRoot, e);

if (p)//结点在树中

printf("关键字：%d 值：%c\n", p->data.key, p->data.e);

else printf("关键字为%d的树结点不存在\n", e);

}

else printf("树%s为空\n", pMyTree->name);

getchar(); getchar();

break;

}

case 9:

{

if (pMyTree->pRoot) //根节点非空

{

KEY key;

ElemType value;

printf("更改前结果:\n");

PreOrderTraverse(pMyTree->pRoot, visit);//展示结果

printf("输入结点的关键字:\n");

scanf("%d", &key);

printf("输入更改后的结点的关键字和值(格式如：1 c):\n");

scanf("%d %c", &value.key, &value.e);

if (Assign(pMyTree->pRoot, key, value))//赋值成功

{

printf("更改后结果:\n");

char \*p1 = "key";

char \*p2 = "data";

printf("%10s%10s\n", p1, p2);

PreOrderTraverse(pMyTree->pRoot, visit);//展示结果

}

else printf("结点不存在\n");

}

else printf("树%s为空\n", pMyTree->name);

getchar(); getchar();

break;

}

case 10:

{

if (pMyTree->pRoot)//根节点非空

{

BiTNode \*p;

KEY e;

printf("输入关键字\n");

scanf(" %d", &e);

p = Parent(pMyTree->pRoot, e);

if (p)//存在双亲结点

{

printf("双亲结点地址为：%p，数据为：%d %c\n", p, p->data.key, p->data.e);

}

else printf("无双亲结点\n");

}

else printf("树%s为空\n", pMyTree->name);

getchar(); getchar();

break;

}

case 11:

{

if (pMyTree->pRoot == NULL)//根节点为空

printf("树%s为空\n", pMyTree->name);

else {

BiTNode \*p;

KEY e;

printf("输入关键字\n");

scanf(" %d", &e);

if (!Value(pMyTree->pRoot, e))

{

printf("该结点不在树中\n"); getchar(); getchar(); break;

}//判断关键字为e的结点是否在树中,若不在树中跳出case

p = LeftChild(pMyTree->pRoot, e);

if (p)//如果有左孩子

{

printf("左孩子结点地址为：%p，数据为：%d %c\n", p, p->data.key, p->data.e);

}

else printf("无左孩子\n");

}

getchar(); getchar();

break;

}

case 12:

{

if (pMyTree->pRoot == NULL)

printf("树%s为空\n", pMyTree->name);

else {

BiTNode \*p;

KEY e;

printf("输入关键字\n");

scanf(" %d", &e);

if (!Value(pMyTree->pRoot, e))

{

printf("该结点不在树中\n"); getchar(); getchar(); break;

}

p = RightChild(pMyTree->pRoot, e);

if (p)//如果有右孩子

{

printf("右孩子结点地址为：%p，数据为：%d %c\n", p, p->data.key, p->data.e);

}

else printf("无右孩子\n");

}

getchar(); getchar();

break;

}

case 13:

{

if (pMyTree->pRoot == NULL)

printf("树%s为空\n", pMyTree->name);

else {

BiTNode \*p;

KEY e;

printf("输入关键字\n");

scanf(" %d", &e);

if (!Value(pMyTree->pRoot, e))

{

printf("该结点不在树中\n"); getchar(); getchar(); break;

}

p = LeftSibling(pMyTree->pRoot, e);

if (p)//有左兄弟

{

printf("该结点左兄弟地址为：%p，数据为：%d %c\n", p, p->data.key, p->data.e);

}

else printf("无左兄弟\n");

}

getchar(); getchar();

break;

}

case 14:

{

if (pMyTree->pRoot == NULL)

printf("树%s为空\n", pMyTree->name);

else {

BiTNode \*p;

KEY e;

printf("输入关键字\n");

scanf(" %d", &e);

if (!Value(pMyTree->pRoot, e))

{

printf("该结点不在树中\n"); getchar(); getchar(); break;

}

p = RightSibling(pMyTree->pRoot, e);

if (p)

{

printf("该结点右兄弟地址为：%p，数据为：%d %c\n", p, p->data.key, p->data.e);

}

else printf("无右兄弟\n");

}

getchar(); getchar();

break;

}

case 15:

{

if (pMyTree->pRoot == NULL)

printf("二叉树%s为空\n", pMyTree->name);

else {

char \*p1 = "key";

char \*p2 = "data";

printf("%10s%10s\n", p1, p2);

PreOrderTraverse(pMyTree->pRoot, visit);

}

getchar(); getchar();

break;

}

case 16:

{

if (pMyTree->pRoot == NULL)

printf("二叉树%s为空\n", pMyTree->name);

else {

char \*p1 = "key";

char \*p2 = "data";

printf("%10s%10s\n", p1, p2);

InOrderTraverse(pMyTree->pRoot, visit);

}

getchar(); getchar();

break;

}

case 17:

{

if (pMyTree->pRoot == NULL)

printf("二叉树%s为空\n", pMyTree->name);

else {

char \*p1 = "key";

char \*p2 = "data";

printf("%10s%10s\n", p1, p2);

PostOrderTraverse(pMyTree->pRoot, visit);

}

getchar(); getchar();

break;

}

case 18:

{

if (pMyTree->pRoot == NULL)

printf("二叉树%s为空\n", pMyTree->name);

else {

char \*p1 = "key";

char \*p2 = "data";

printf("%10s%10s\n", p1, p2);

LevelOrderTraverse(pMyTree->pRoot, visit);

}

getchar(); getchar();

break;

}

case 19:

{

BiTNode \*p;

BiTree c;

printf("请输入将要创建的子树\n");

{

ElemType def[500];//二叉树定义

ElemType \*definition = def;

int i;

i = 0;

for (i = 0; i < 500; i++)

{

def[i].e = '#';

def[i].key = 0;

}//初始树结点定义均为空

int N;

i = 0;

printf("输入数据组数:\n");

scanf("%d", &N);

if (N > 2000)

{

printf("数据量过大\n");

system("pause");

break;

}

printf("输入数据\n");

while (N--)

{

scanf("%d %c", &def[i].key, &def[i].e);//读取树定义

i++;

}

CreateBiTree(&c, definition);

printf("子树c创建成功\n");

}

printf("子树c的前序遍历结果:\n");

PreOrderTraverse(c, visit);

int key, LR;

printf("输入要插入子树的结点的关键字:");

scanf("%d", &key);

p = Value(pMyTree->pRoot, key);

printf("输入要插入的位置：0.左子树 1.右子树");

scanf("%d", &LR);

if (LR != 0 && LR != 1)

{

printf("位置输入错误\n");

}

else if (c->rchild == NULL)

{

InsertChild(pMyTree->pRoot, p, LR, c);

printf("子树插入成功,插入后的前序遍历结果：\n");

PreOrderTraverse(pMyTree->pRoot, visit);

}

getchar(); getchar(); break;

}

case 20:

{

BiTNode \*p;

int key, LR;

printf("输入要删除子树的结点的关键字:");

scanf("%d", &key);

p = Value(pMyTree->pRoot, key);

printf("输入要删除的子树的位置：0.左子树 1.右子树");

scanf("%d", &LR);

if (LR != 0 && LR != 1)

{

printf("位置输入错误\n");

}

else

{

DeleteChild(pMyTree->pRoot, p, LR);

printf("子树删除成功，删除后的前序遍历结果：\n");

PreOrderTraverse(pMyTree->pRoot, visit);

}

getchar(); getchar(); break;

}

case 21:

{

SaveData(L);

printf("数据保存成功\n");

getchar(); getchar();

break;

}

case 22:

{

if (L.listlenth)

printf("线性表非空，无法初始化，请销毁树后再加载数据\n");

else

{

if (LoadData(L))

printf("数据加载成功\n");

mytree = 0;

pMyTree = &L.pT[mytree];

}

getchar(); getchar();

break;

}

case 23:

{

mytree = SelectTree(L);//从TreeNum棵树中选择一棵树

pMyTree = &L.pT[mytree];

getchar(); getchar();

break;

}

case 0:

break;

}//end of switch

}//end of while

printf("欢迎下次再使用本系统！\n");

}//end of main()

/\*--------------------page 23 on textbook ----------------------------------\*/

status InitBiTree(BiTree \*T)

{

\*T = NULL;

return OK;

}

status DestroyBiTree(BiTree \*T)

{

BiTNode \*pnode = \*T;//pnode指向树根

if (pnode)//如果树存在

{

if (pnode->lchild)//如果有左孩子

DestroyBiTree(&(pnode->lchild));

if (pnode->rchild)//如果有右孩子

DestroyBiTree(&(pnode->rchild));

free(pnode);//释放根节点

}

return OK;

}

status CreateBiTree(BiTree \*T, ElemType \* &definition)

{

if ((\*definition).e == '#' && (\*definition).key == 0)

{

\*T = NULL;//# 0表示该节点无子树

definition++;//definition指向下一个元素

}

else

{

if (!(\*T = (BiTNode \*)malloc(sizeof(BiTNode))))//声明一个树结点

exit(OVERFLOW);//空间分配失败

(\*T)->data = \*(definition++);//创建根节点,definition指向下一个元素

CreateBiTree(&(\*T)->lchild, definition);//创建左子树

CreateBiTree(&(\*T)->rchild, definition);//创建右子树

}

return OK;

}

status ClearBiTree(BiTree \*T)

{

BiTNode \*pnode = \*T;//pnode指向树根

if (pnode)//如果树存在

{

if (pnode->lchild)//如果有左孩子

ClearBiTree(&(pnode->lchild));

if (pnode->rchild)//如果有右孩子

ClearBiTree(&(pnode->rchild));

free(pnode);//清空根结点

\*T = NULL;//置空树指针

}

return OK;

}

status BiTreeEmpty(BiTree T)

{

if (T == NULL)

return TRUE;

else return FALSE;

}

BiTNode \* Root(BiTree T)

{

return T;

}

status BiTreeDepth(BiTree T)

{

int i, j;

if (!T)//如果树为空

return 0;

if (T->lchild)

i = BiTreeDepth(T->lchild); // 左子树深度

else

i = 0;

if (T->rchild)

j = BiTreeDepth(T->rchild); // 右子树深度

else

j = 0;

return i>j ? i + 1 : j + 1;

}

BiTNode \*Value(BiTree T, KEY e)

{

int top = -1;

BiTree stack[100], p;

stack[++top] = T;

while (top>-1) {

p = stack[top--];//出栈

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

if (p->rchild) //如果有右孩子

stack[++top] = p->rchild;//右孩子进栈

if (p->lchild)//如果有左孩子

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

}//栈空，未找到

return NULL;

}

status Assign(BiTree T, KEY e, ElemType value)

{

int top = -1;

BiTree stack[100], p;

stack[++top] = T;

while (top>-1)

{

p = stack[top--];

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

break;

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

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

}

if (p->data.key != e)//结点不存在

return FALSE;

else p->data = value;

return TRUE;

}

BiTNode \* Parent(BiTree T, KEY e)

{

int top = -1;

BiTree stack[100], p;

stack[++top] = T;//根节点进栈

if (T->data.key == e)

return NULL;//e为根节点

while (top>-1)

{

p = stack[top--];//出栈

if (p->lchild)

{

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

return p;

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

}

if (p->rchild)

{

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

return p;

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

}

}

return NULL;

}

BiTNode \* LeftChild(BiTree T, KEY e)

{

int top = -1;

BiTree stack[100], p;

stack[++top] = T;

while (top>-1)

{

p = stack[top--];

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

return p->lchild;

if (p->lchild)

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

if (p->rchild)

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

}

return NULL;

}

BiTNode \* RightChild(BiTree T, KEY e)

{

int top = -1;

BiTree stack[100], p;

stack[++top] = T;

while (top>-1)

{

p = stack[top--];

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

return p->rchild;

if (p->lchild)

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

if (p->rchild)

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

}

return NULL;

}

BiTNode \* LeftSibling(BiTree T, KEY e)

{

int top = -1;

BiTree stack[100], p;

stack[++top] = T;//根节点进栈

if (T->data.key == e)

return NULL;//e为根节点，无左兄弟

while (top>-1)

{

p = stack[top--];//出栈

if (p->lchild)

{

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

}

if (p->rchild)

{

if (p->rchild->data.key == e)//p的右孩子关键字为e

return p->lchild;//返回p的左孩子

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

}

}

return NULL;//树中无关键字为e的结点，返回空

}

BiTNode \* RightSibling(BiTree T, KEY e)

{

int top = -1;

BiTree stack[100], p;

stack[++top] = T;//根节点进栈

if (T->data.key == e)

return NULL;//e为根节点，无左兄弟

while (top>-1)

{

p = stack[top--];//出栈

if (p->lchild)

{

if (p->lchild->data.key == e) // p的左孩子关键字为e

return p->rchild;//返回p的右孩子

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

}

if (p->rchild)

{

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

}

}

return NULL;//树中无关键字为e的结点，返回空

}

status InsertChild(BiTree T, BiTNode \*p, int LR, BiTree c)

{

BiTNode \*temp;

if (LR == 0)//c插入到p的左子树

{

temp = p->lchild;//p的原左子树地址

p->lchild = c;//c插入到p的左子树

c->rchild = temp;//p的原左子树插入到c的右子树

}

else//c插入到p的右子树

{

temp = p->rchild;

p->rchild = c;

c->rchild = temp;

}

return OK;

}

status DeleteChild(BiTree T, BiTNode \*p, int LR)

{

if (LR == 0)//删除p的左子树

{

ClearBiTree(&p->lchild);

}

else//删除p的右子树

{

ClearBiTree(&p->rchild);

}

return OK;

}

status PreOrderTraverse(BiTree T, void(\*visit)(ElemType e))

{

if (T)

{

visit(T->data);

PreOrderTraverse(T->lchild, visit);

PreOrderTraverse(T->rchild, visit);

}

return OK;

}

status InOrderTraverse(BiTree T, void(\*visit)(ElemType e))

{

if (T)

{

InOrderTraverse(T->lchild, visit);

visit(T->data);

InOrderTraverse(T->rchild, visit);

}

return OK;

}

status PostOrderTraverse(BiTree T, void(\*visit)(ElemType e))

{

if (T)

{

PostOrderTraverse(T->lchild, visit);

PostOrderTraverse(T->rchild, visit);

visit(T->data);

}

return OK;

}

status LevelOrderTraverse(BiTree T, void(\*visit)(ElemType e))

{

int head, tail;

BiTree queue[QUEUESIZE], p;

head = 0; tail = 0;//队空：head==tail,队满：(tail+1)%QUEUESIZE==head

queue[tail++] = T;//根节点进队

while (tail != head && (tail + 1) % QUEUESIZE != head)//队非空且非满

{

p = queue[head];//出队

head = (head + 1) % QUEUESIZE;

visit(p->data);

if (p->lchild)

{//左孩子进队

queue[tail] = p->lchild;

tail = (tail + 1) % QUEUESIZE;

}

if (p->rchild)

{//右孩子进队

queue[tail] = p->rchild;

tail = (tail + 1) % QUEUESIZE;

}

}

return OK;

}

int SelectTree(SqList L)

{

int i, op;

printf("请重新选择管理的树(1-%d)\n", L.listlenth);

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

{

printf(" %d. %s\n", i, L.pT[i - 1].name);

}

scanf("%d", &op);

return op - 1;

}

void visit(ElemType e)

{

printf("%10d%10c\n", e.key, e.e);

}

status SaveData(SqList L)

{

FILE \*fp;

/\*保存线性表信息\*/

if ((fp = fopen(L\_DATA\_NAME, "w")) == NULL)//打开文件失败

{

printf("File open error\n ");

return FALSE;

}

fwrite(L.pT, sizeof(TreeData), L.listlenth, fp);//线性表数据一次性写入到文件中

fclose(fp);

/\*保存树结点信息\*/

fp = fopen(T\_DATA\_NAME, "w");

int i;

for (i = 0; i < L.listlenth; i++)

{

SaveTree(L.pT[i].pRoot, fp);

}

fclose(fp);

return TRUE;

}

status SaveTree(BiTree T, FILE \*fp)

{

if (T)

{

fprintf(fp, "%d %c\n", T->data.key, T->data.e);

SaveTree(T->lchild, fp);

SaveTree(T->rchild, fp);

}

else

{

fprintf(fp, "0 #\n");

}

return OK;

}

status LoadData(SqList &L)

{

/\*写入线性表数据\*/

FILE \*fp;

if ((fp = fopen(L\_DATA\_NAME, "r")) == NULL)//文件打开失败

{

printf("File open error\n ");

return FALSE;

}

while (fread(&L.pT[L.listlenth], sizeof(TreeData), 1, fp))

{

L.listlenth++;//准备写入下一个数据

}

fclose(fp);

/\*写入二叉树数据\*/

if ((fp = fopen(T\_DATA\_NAME, "r")) == NULL)//文件打开失败

{

printf("File open error\n ");

return FALSE;

}

int i;

for (i = 0; i < L.listlenth; i++)

{

LoadTree(L.pT[i].pRoot, fp);

}

return TRUE;//写入完毕

}

status LoadTree(BiTree &T, FILE \*fp)

{

int key; char c;

fscanf(fp, "%d %c", &key, &c);

if (key == 0 && c == '#')

{

T = NULL;//# 0表示该节点无子树

}

else

{

if (!(T = (BiTNode \*)malloc(sizeof(BiTNode))))//声明一个树结点

exit(OVERFLOW);//空间分配失败

T->data.key = key;

T->data.e = c;

LoadTree(T->lchild, fp);//加载左子树

LoadTree(T->rchild, fp);//加载右子树

}

return OK;

}

# 附录D 基于邻接表图实现的源程序