
華中科技大學

課程實驗報告

課程名稱： 數據結構實驗

專業班級： 計算機科學與技術 201601

學 號： U201614532

姓 名： 呂鵬澤

指導教師： 周時阻

報告日期： 2017 年 1 月 6 日

計算機科學與技術學院

目 录

1 基于顺序存储结构的线性表实现.....	2
1.1 问题描述.....	2
1.2 系统设计.....	2
1.3 系统实现.....	16
1.4 实验小结.....	23
2 基于链式存储结构的线性表实现.....	24
2.1 问题描述.....	24
2.2 系统设计.....	24
2.3 系统实现.....	37
2.4 实验小结.....	44
3 基于二叉链表的二叉树实现.....	46
3.1 问题描述.....	46
3.2 系统设计.....	46
3.3 系统实现.....	74
3.4 实验小结.....	86
4 基于邻接表的图实现.....	86
4.1 问题描述.....	86
4.2 系统设计.....	87
4.3 系统实现.....	107
4.4 实验小结.....	117
参考文献.....	118
附录 A 基于顺序存储结构线性表实现的源程序	121
附录 B 基于链式存储结构线性表实现的源程序	136
附录 C 基于二叉链表二叉树实现的源程序	154
附录 D 基于邻接表图实现的源程序	187

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：

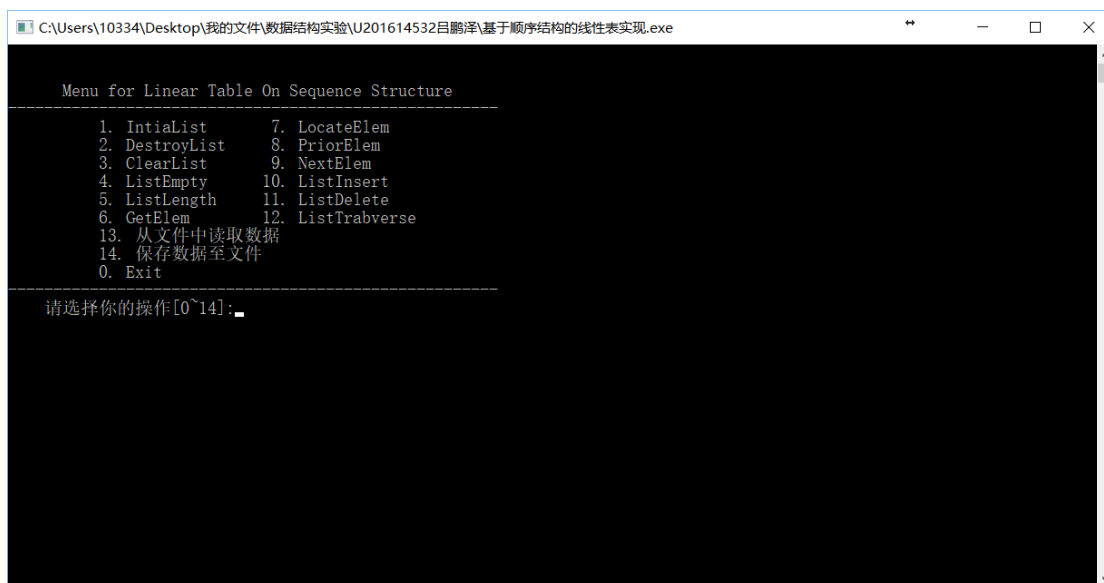


图 1-1 系统载入界面

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。

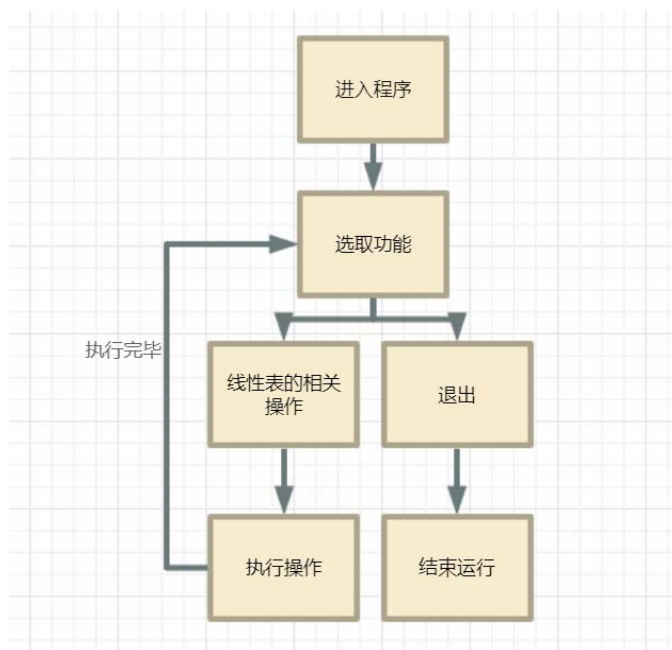


图 1-2 系统总体结构

1.2.2 顺序表物理结构

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。

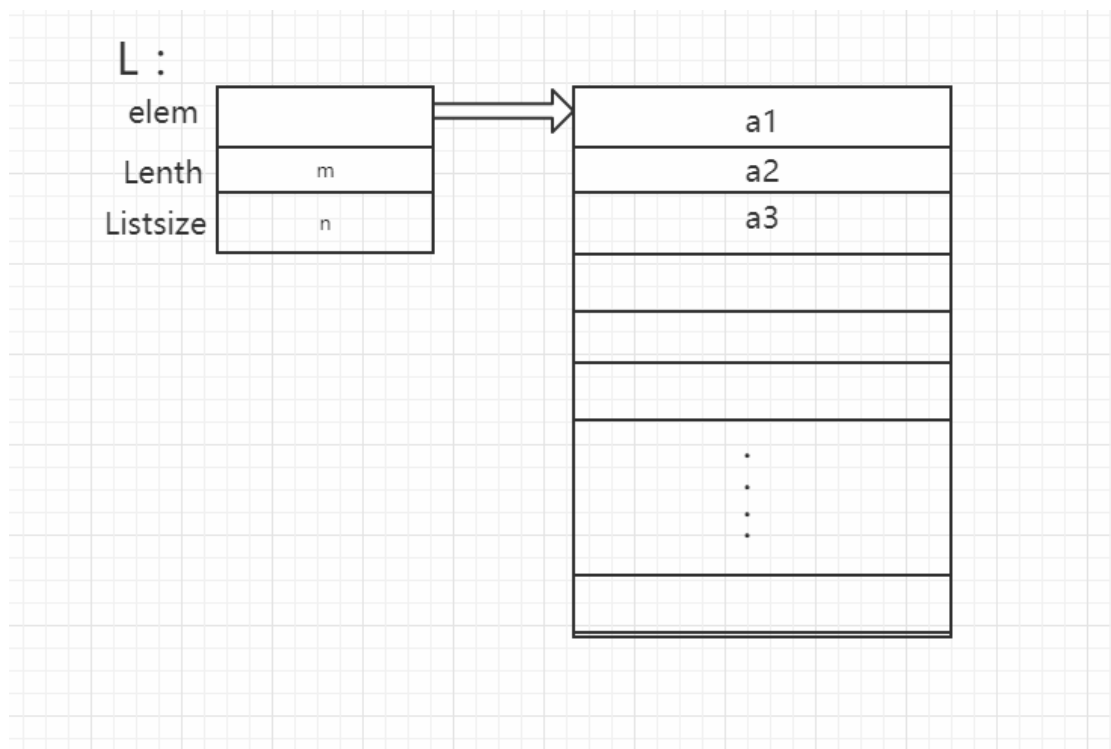


图 1-3 顺序表物理结构

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：

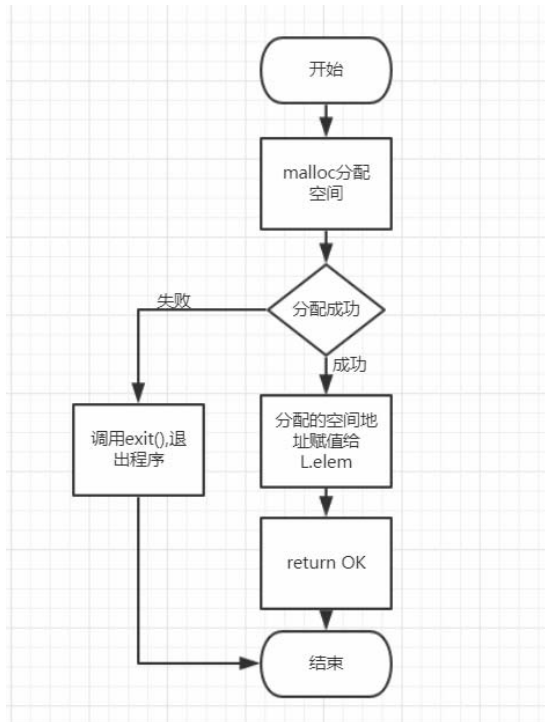


图 1-4 InitiaList()流程图

2) 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：

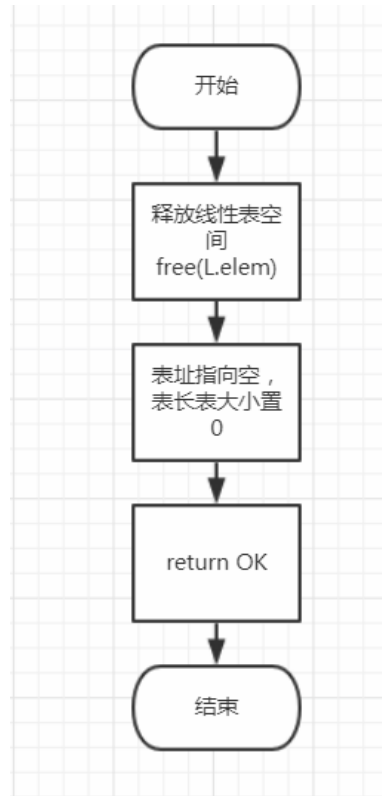


图 1-5 DestroyList()流程图

3) 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：

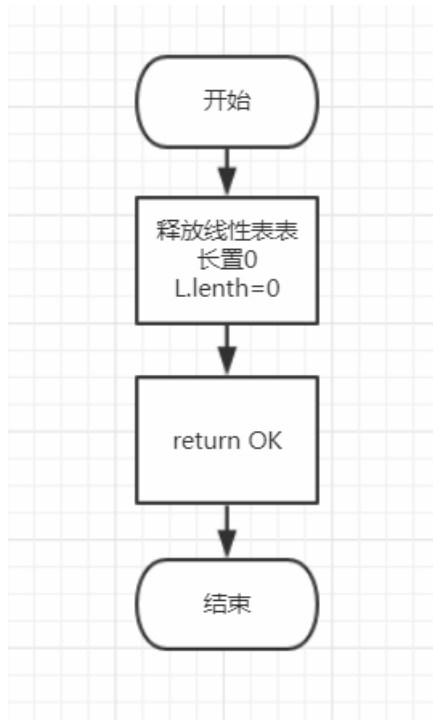


图 1-6 ClearList()流程图

4) 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：

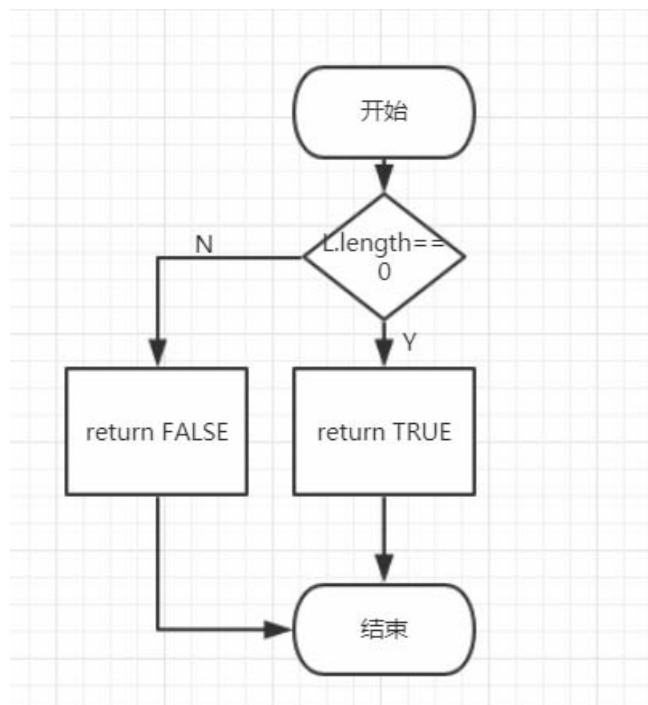


图 1-7 ListEmpty()流程图

5) 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：

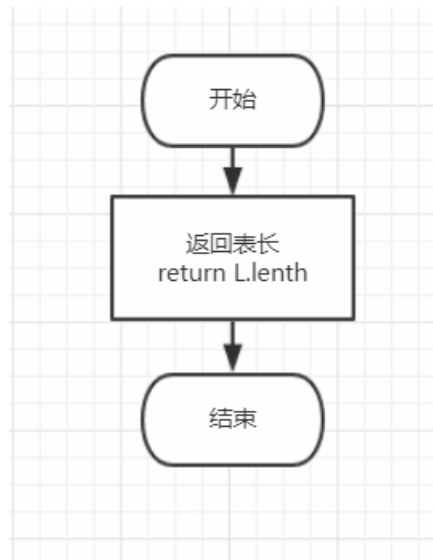


图 1-8 ListLength(L)流程图

6) 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：

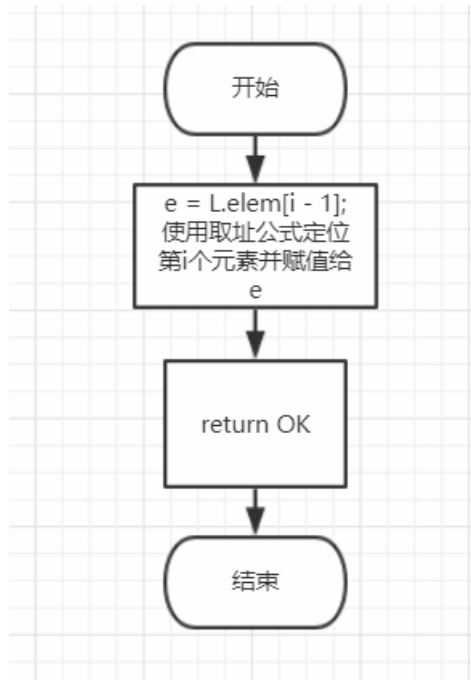


图 1-9 GetElem()流程图

7) 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：

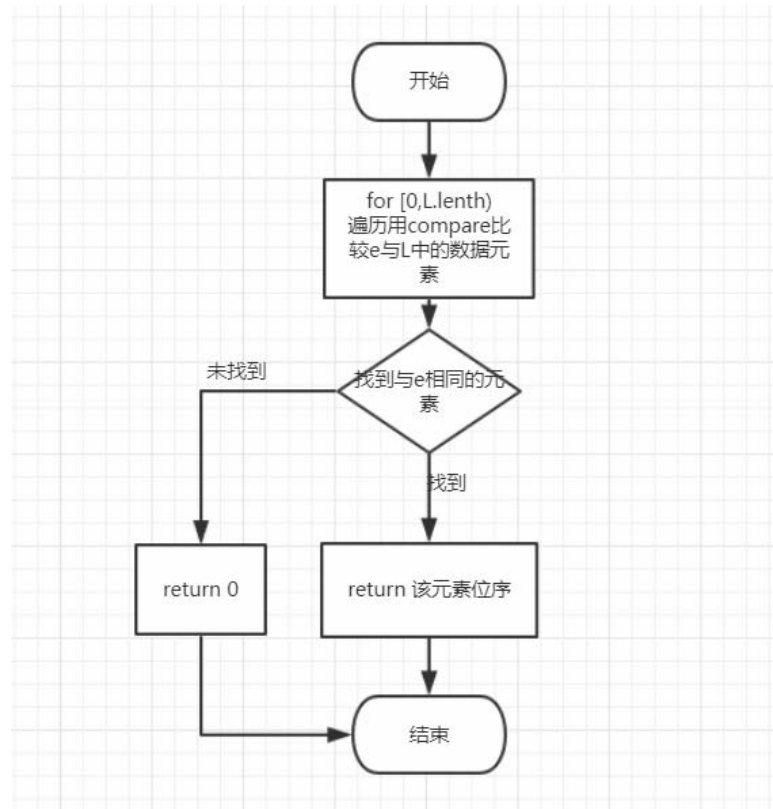


图 1-10 LocateElem()流程图

8) 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：

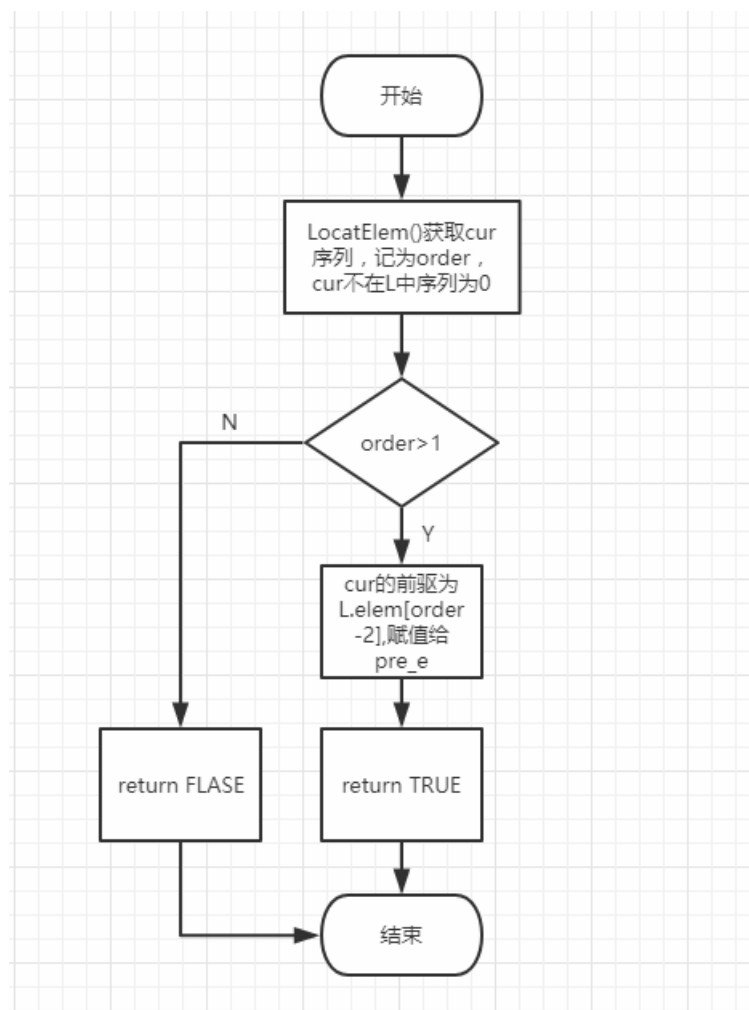


图 1-11 PriorElem()流程图

9) 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:

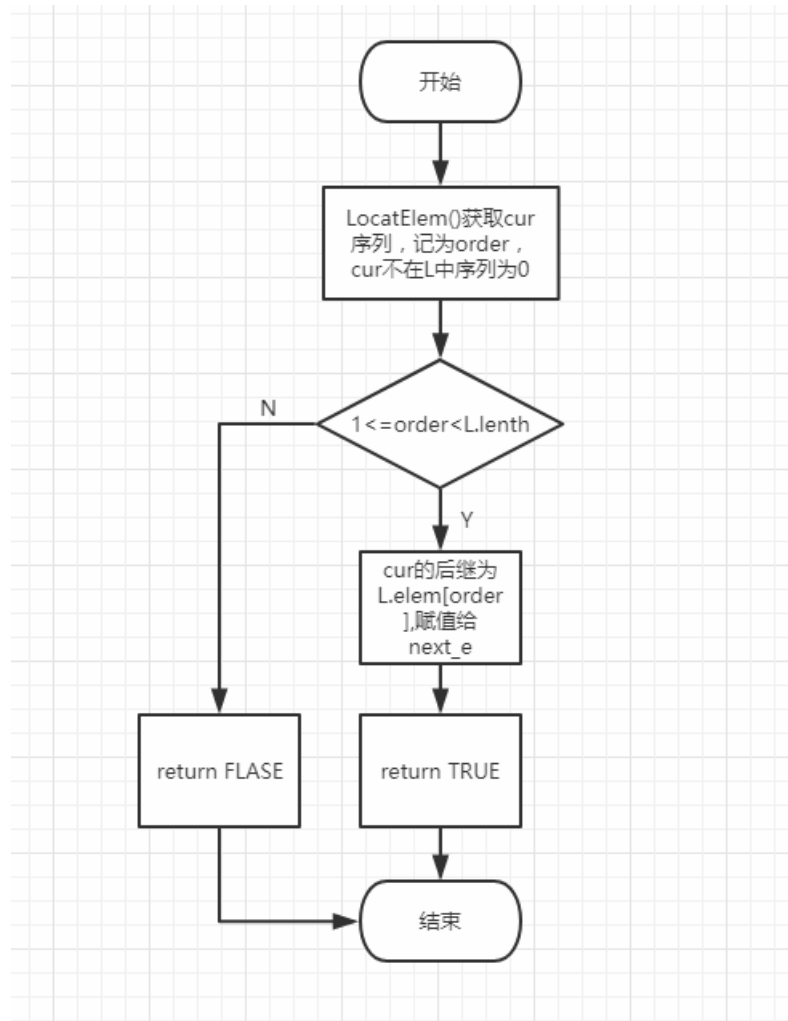


图 1-12 NextElem()流程图

10) 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 e 4.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：

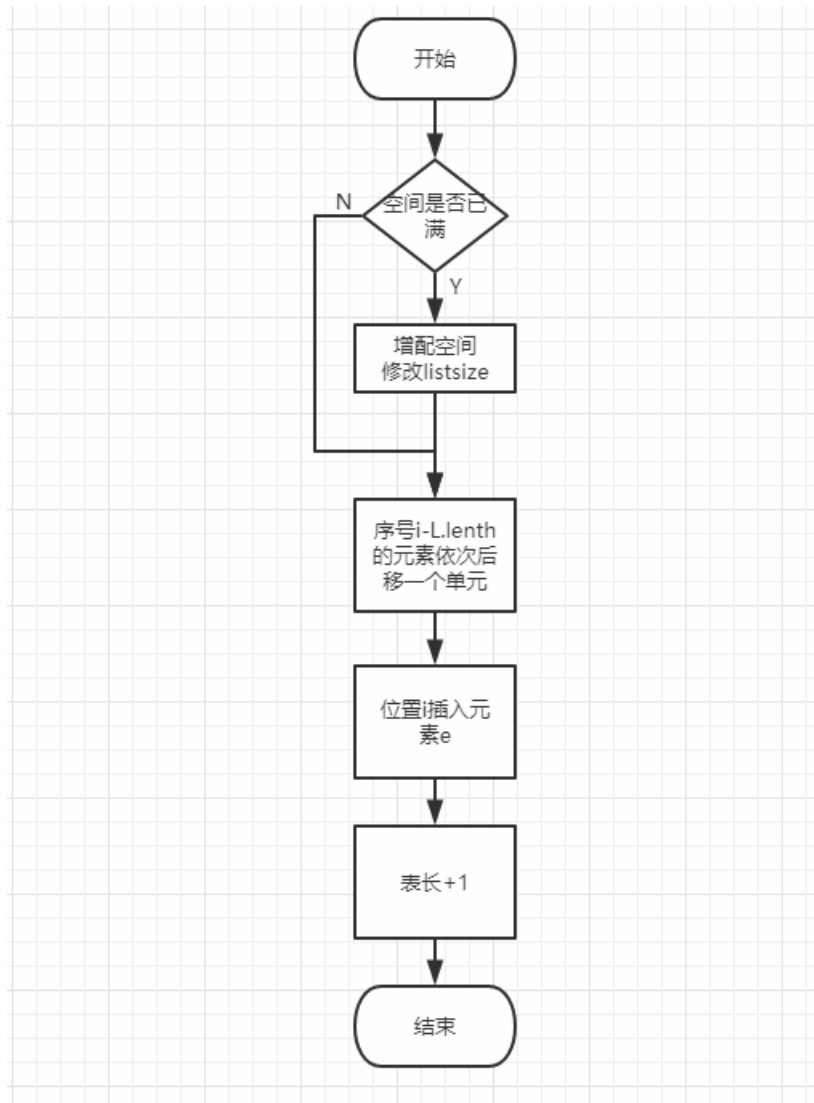


图 1-13 ListInsert()流程图

11) 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：

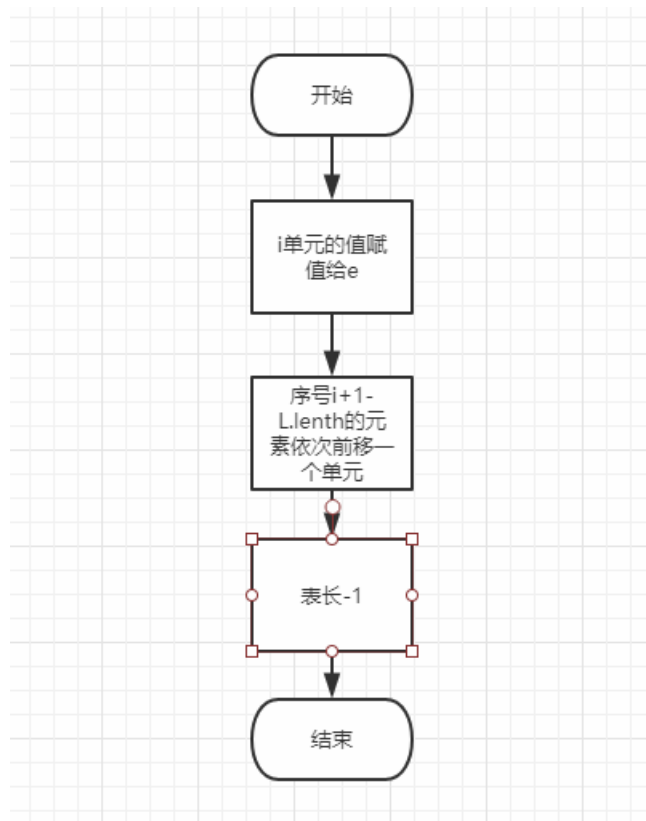


图 1-14 ListDelete()流程图

12) 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：

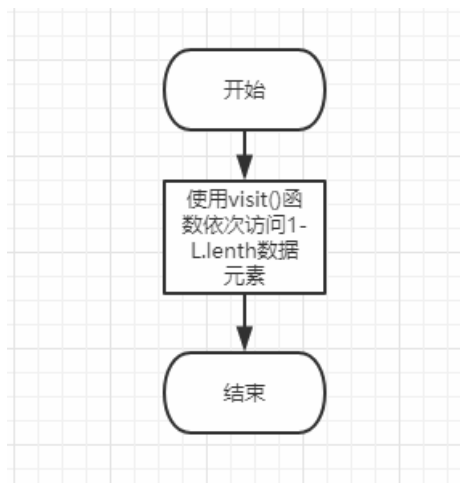


图 1-15 ListTraverse()流程图

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

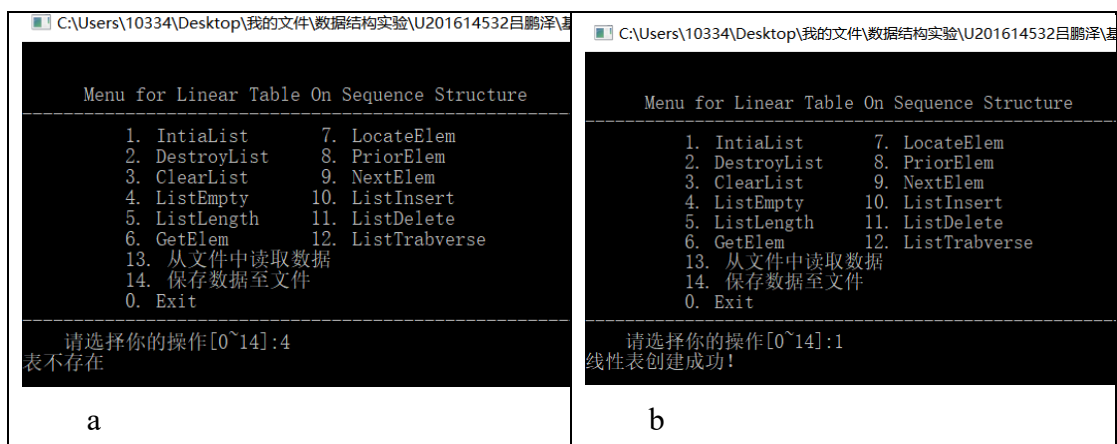
用户打开程序后可以看到如下界面，输入数字选择相应功能函数，输入 0 退出程序。

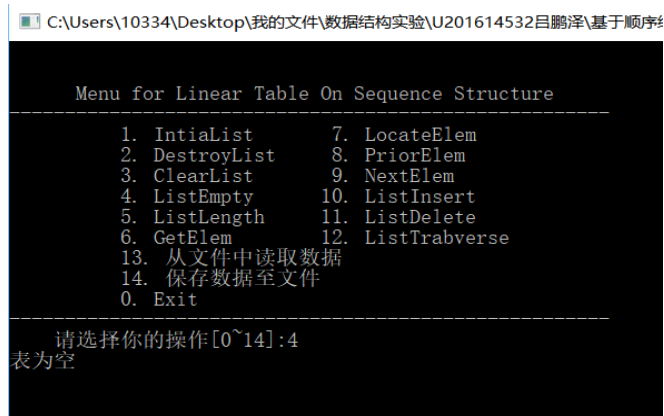


图 1-16 系统载入界面

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。



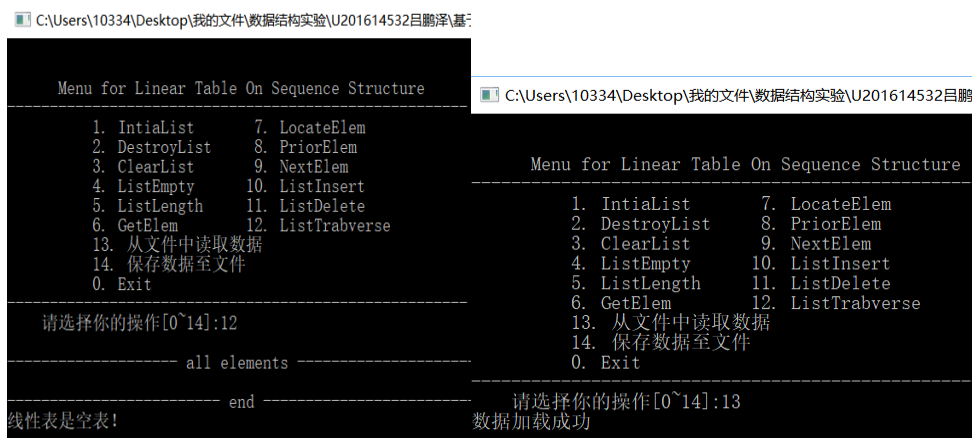


c

图 1-16 创建表操作演示

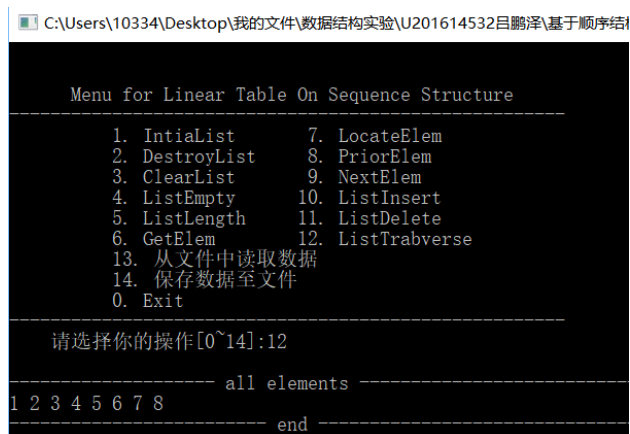
3.从文件中加载数据及遍历操作的演示

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

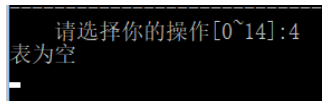
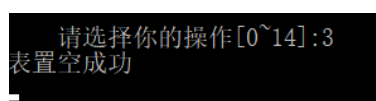
4.判表空及置空表

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



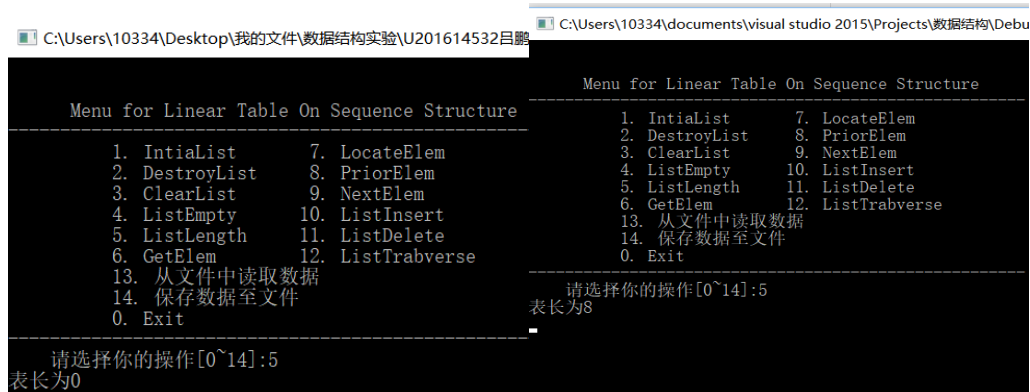
c

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



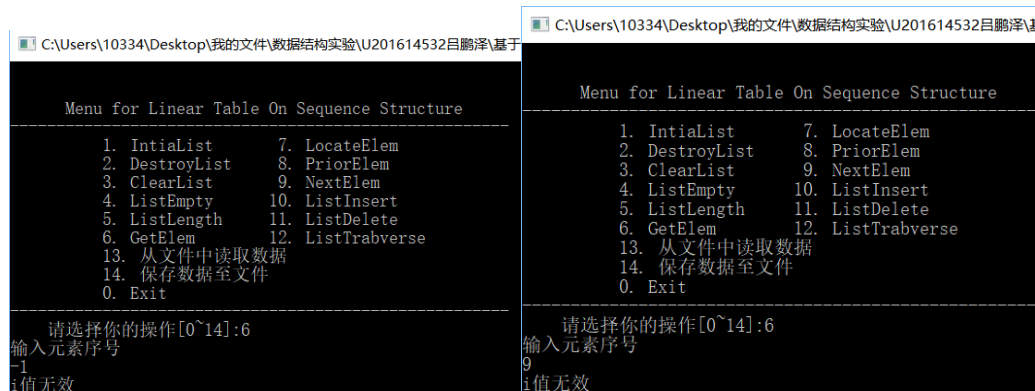
a

b

图 1-19 求表长

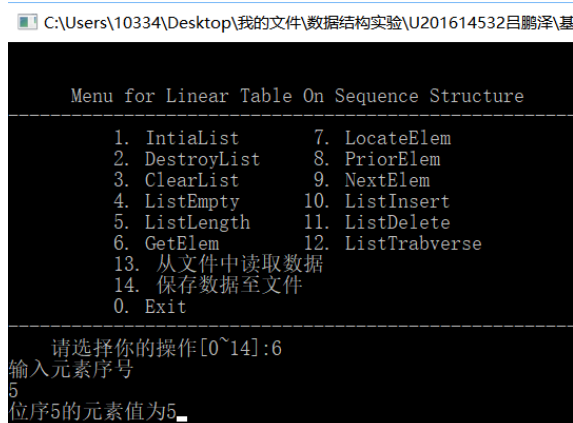
6. 获取元素

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。



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。

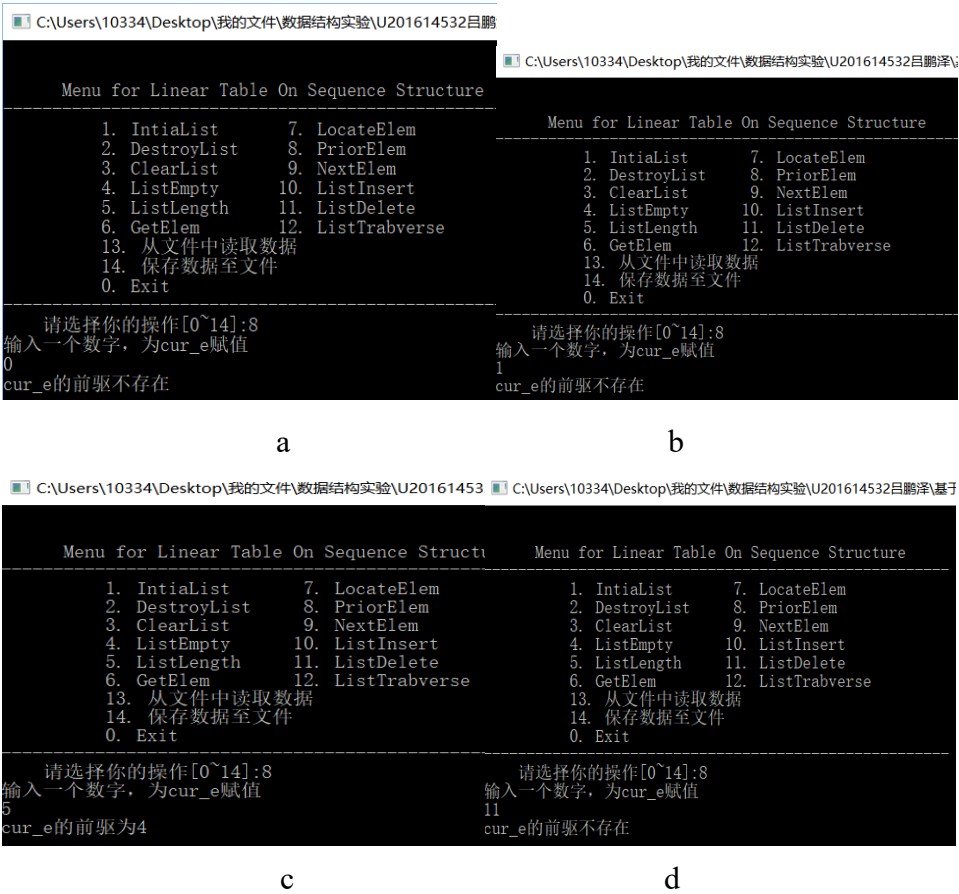


图 1-22 获取前驱

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。

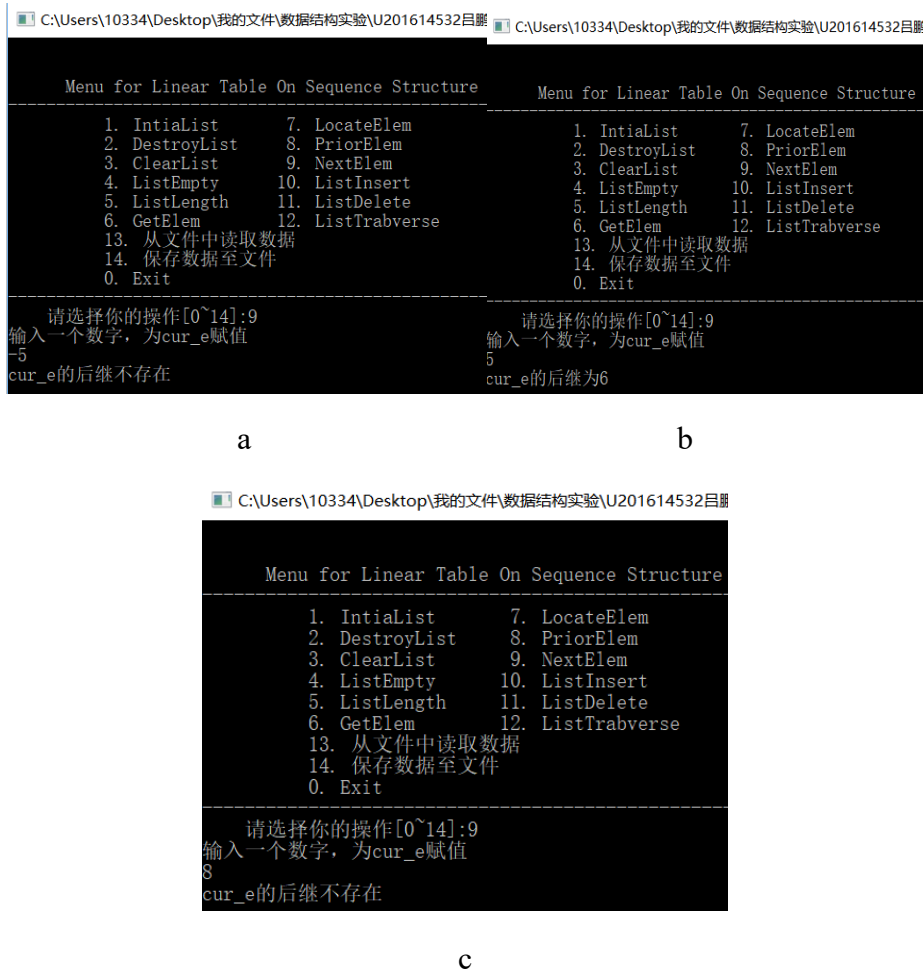


图 1-23 获取后继

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。



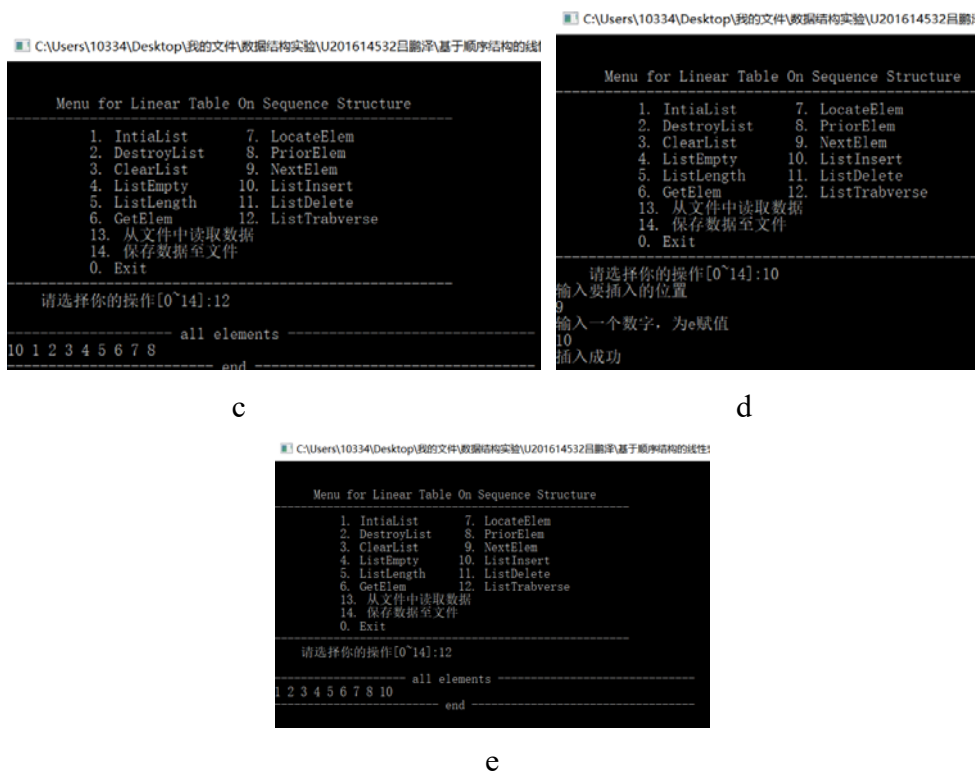
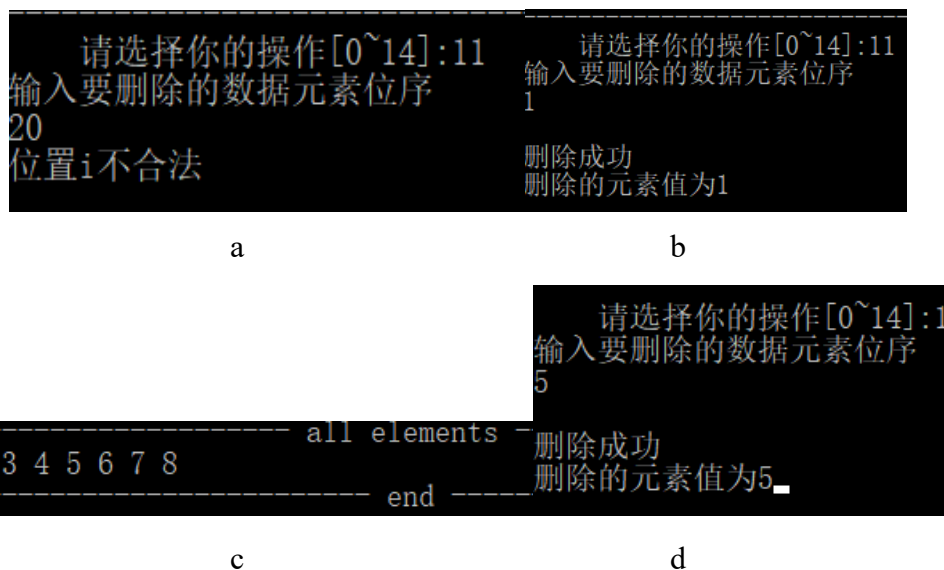
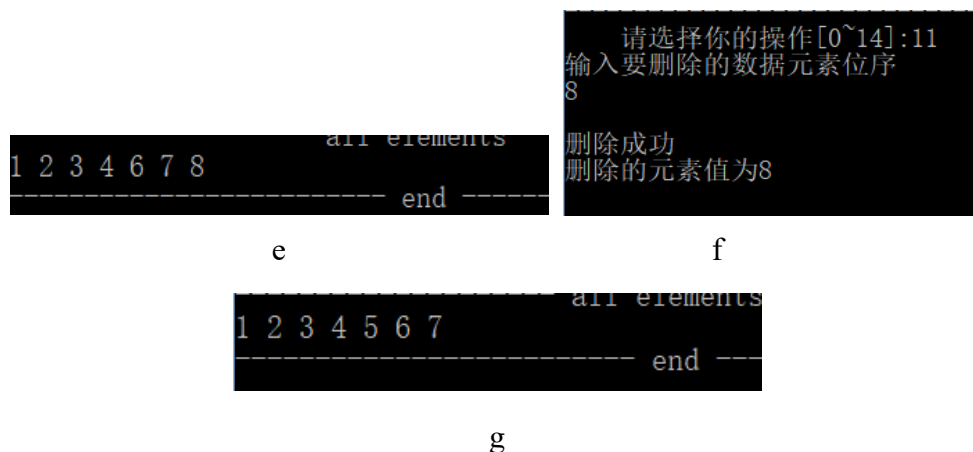


图 1-24 插入元素

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。





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：

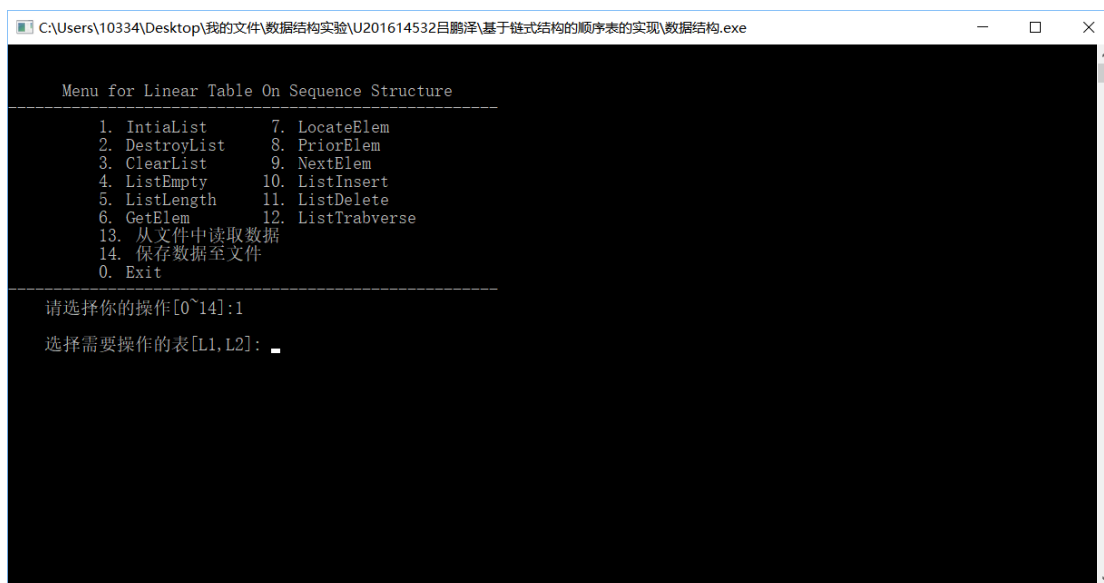


图 2-1 系统载入界面

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。

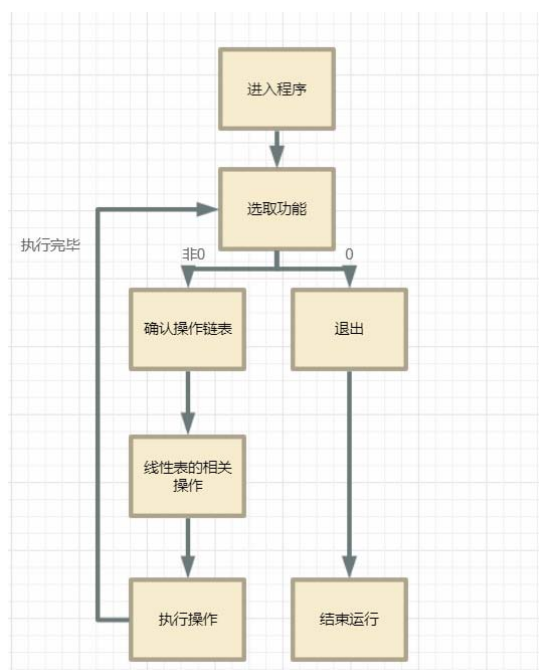
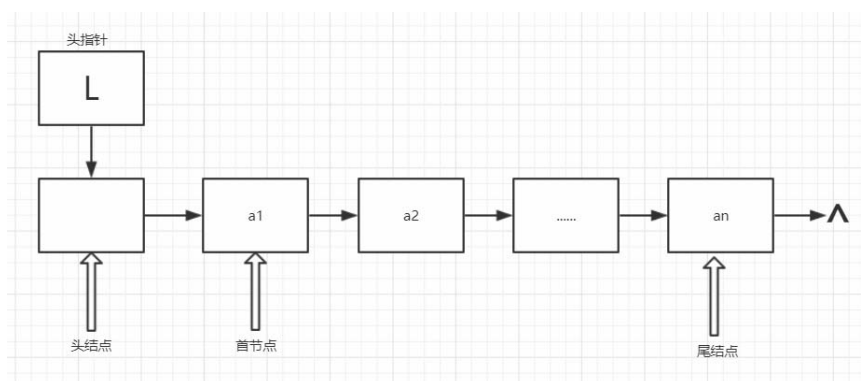


图 2-2 系统总体结构

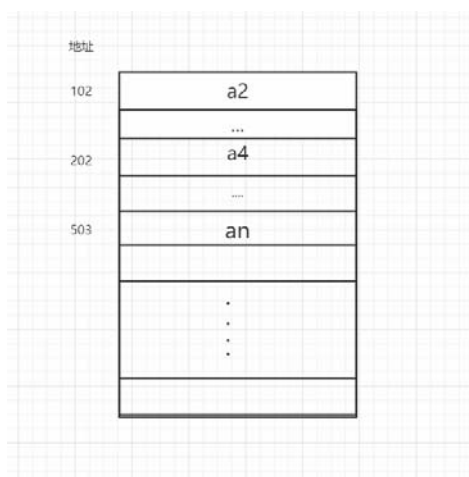
2.2.2 链表的物理结构

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：

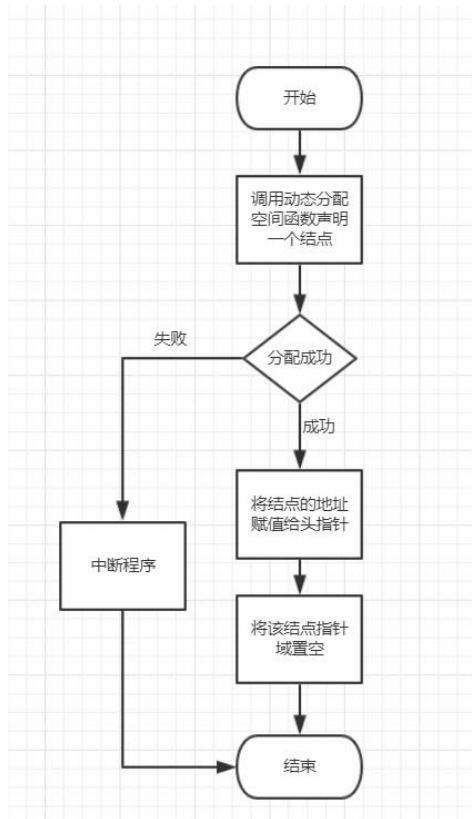


图 2-4 InitiaList()流程图

2) 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：

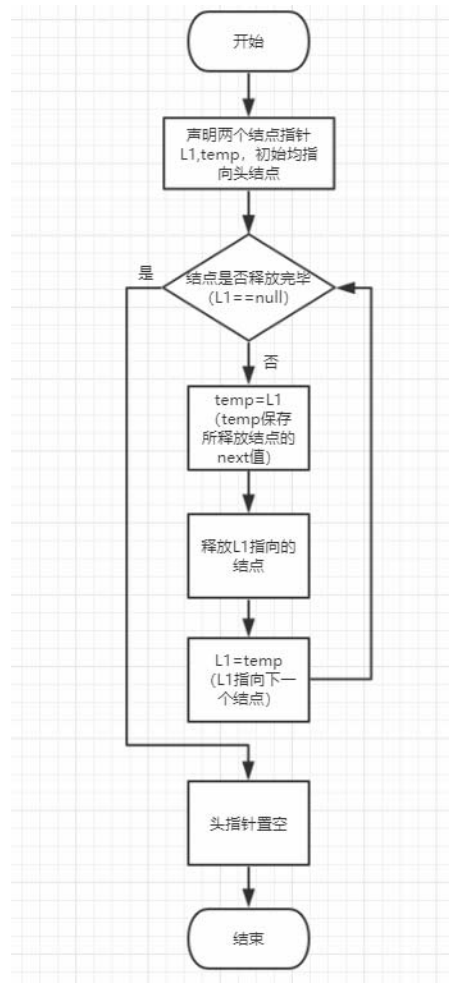


图 2-5 DestroyList()流程图

3) 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：

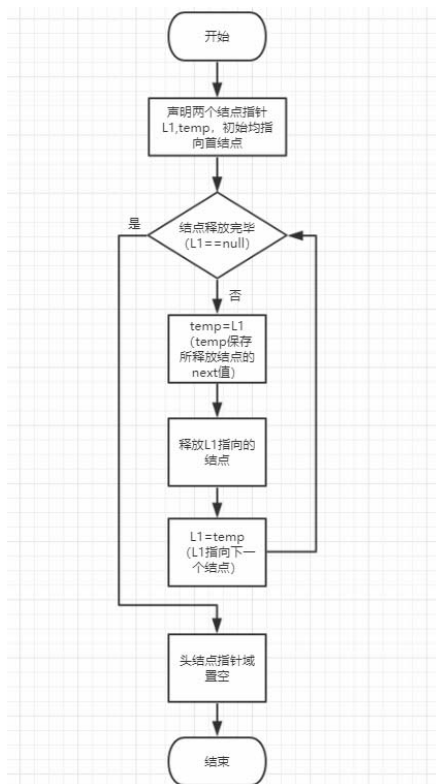


图 2-6 ClearList()流程图

4) 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：

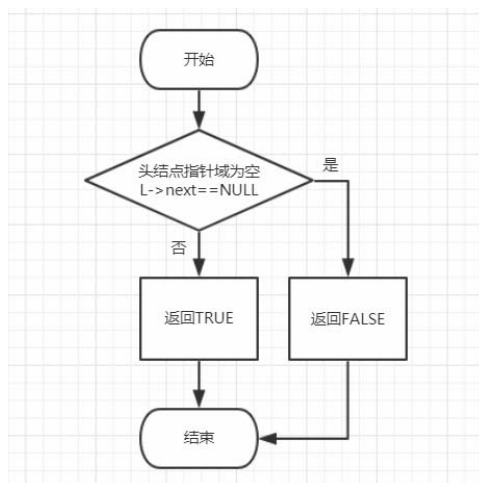


图 2-7 ListEmpty()流程图

5) 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：

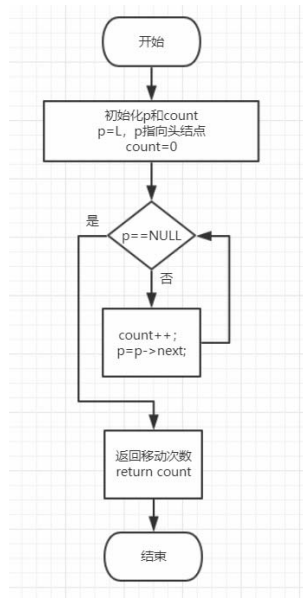


图 2-8 ListLength(L)流程图

6) 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：

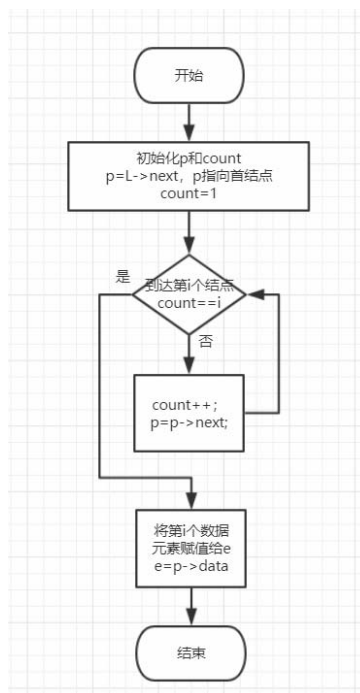


图 2-9 GetElem()流程图

7) 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：

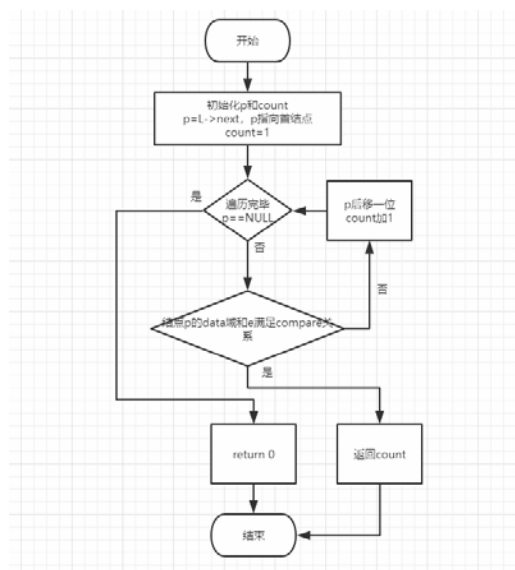


图 2-10 LocateElem()流程图

8) 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:

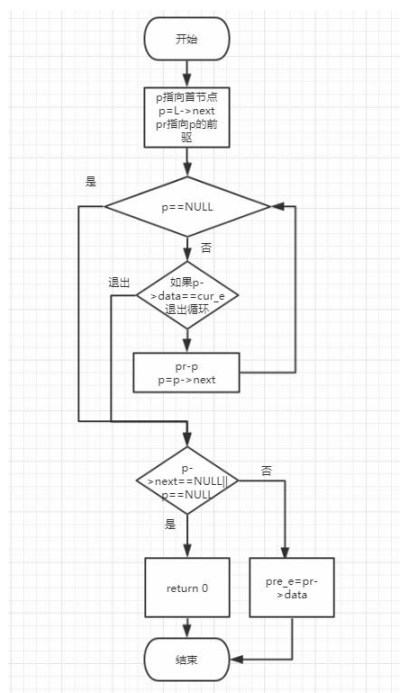


图 2-11 PriorElem()流程图

9) 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:

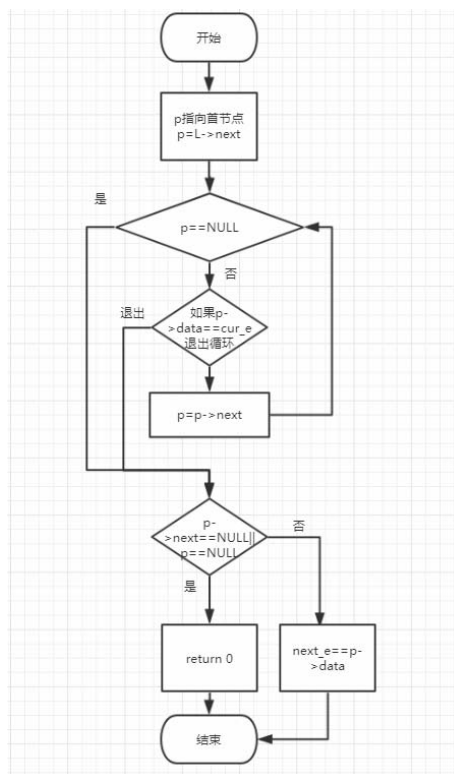


图 2-12 NextElem()流程图

10) 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:

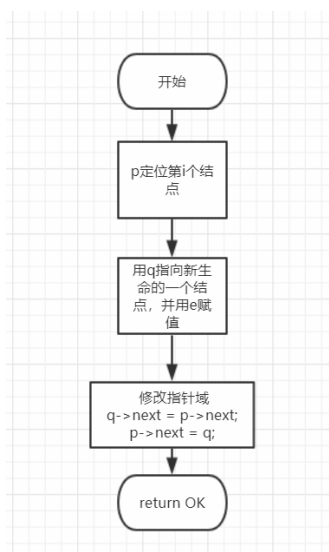


图 2-13 ListInsert()流程图

11) 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：

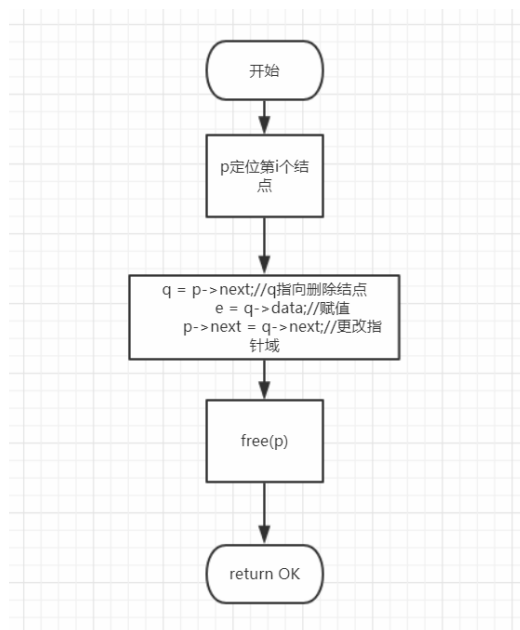


图 2-14 ListDelete()流程图

12) 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：

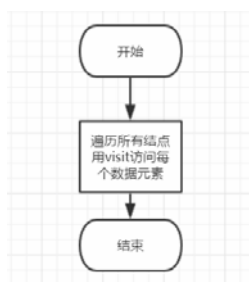


图 2-15 ListTraverse()流程图

13) 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：

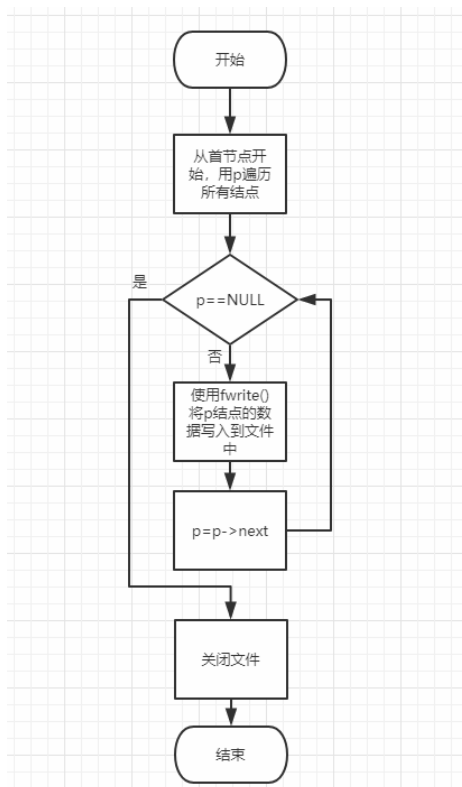


图 2-16 Savedata()流程图

14) 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：

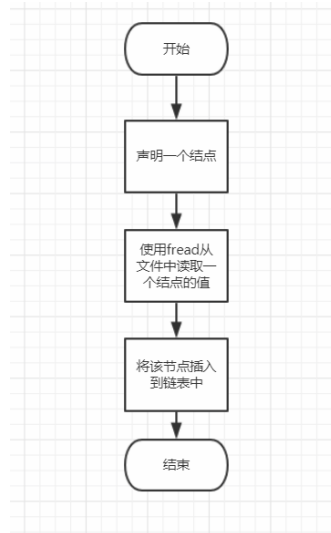


图 2-16 Loaddata()流程图

15) 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：

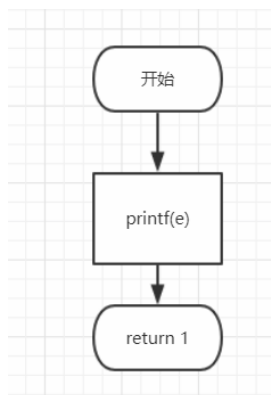


图 2-17 visit()流程图

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。

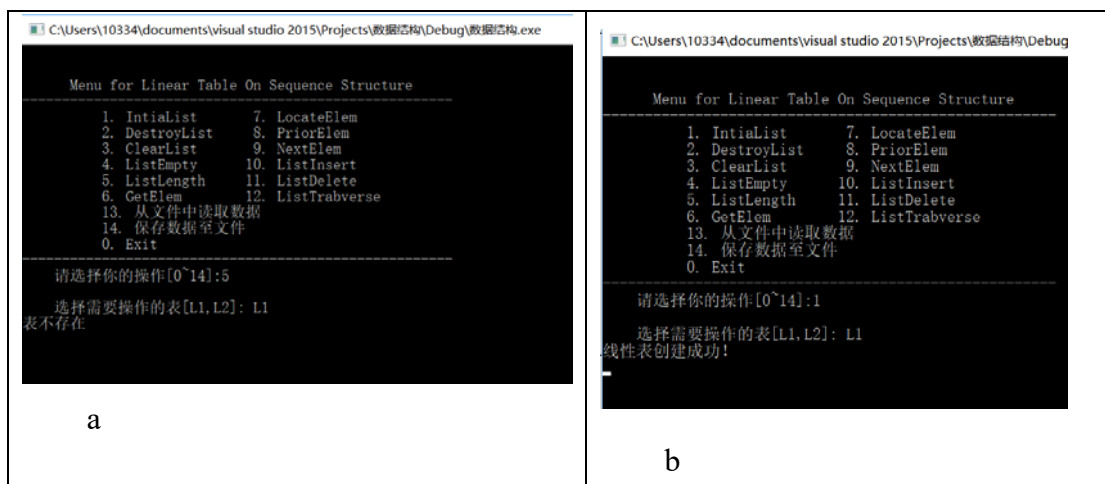
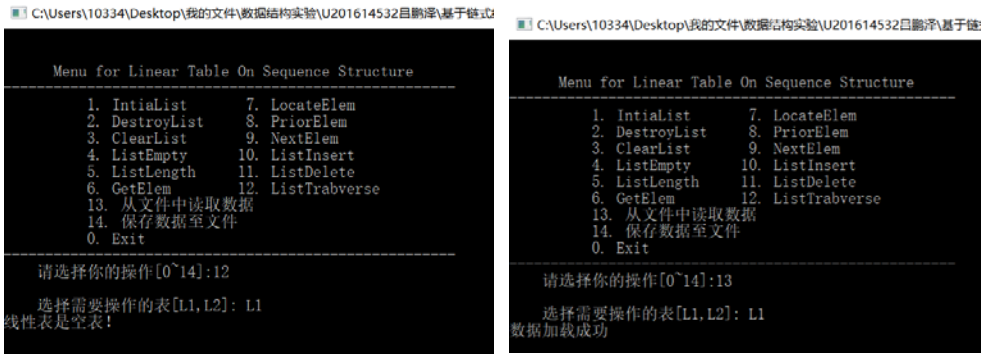


图 2-19 创建表操作演示

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



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

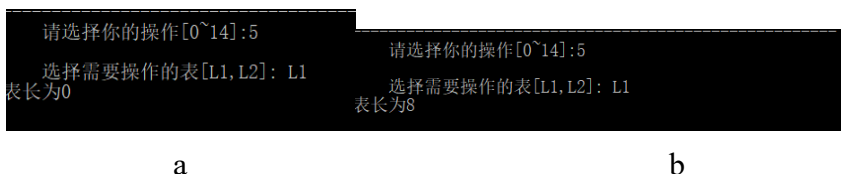


图 2-22 求表长

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。

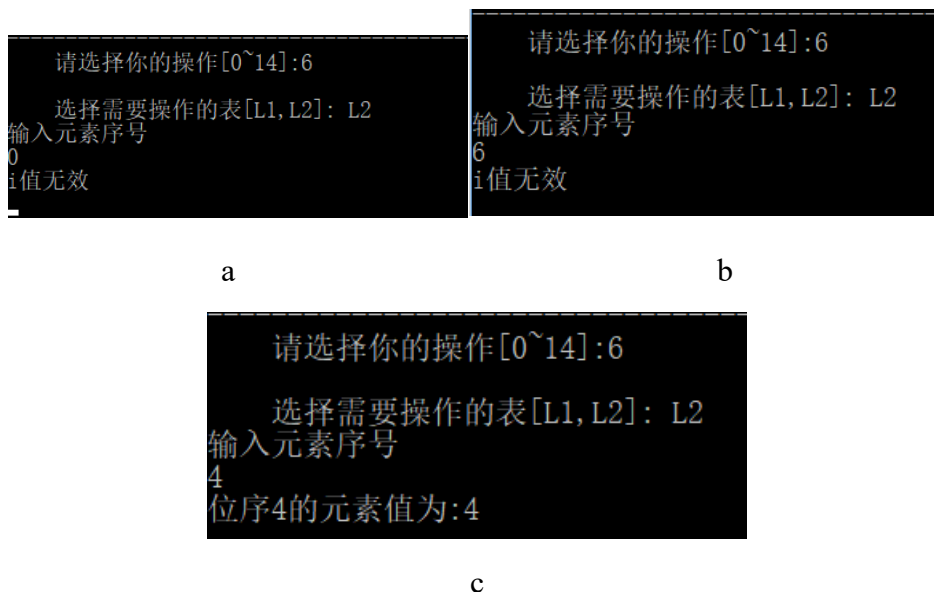


图 2-23 获取元素

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。

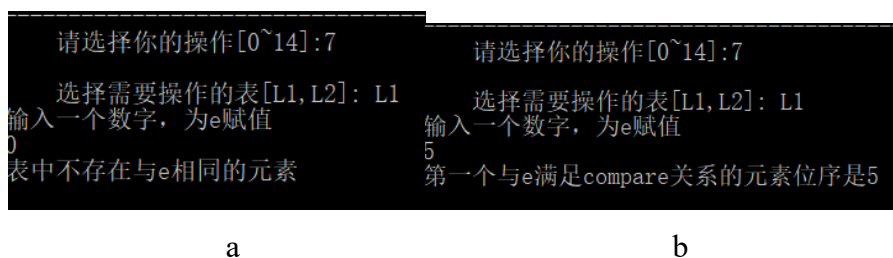


图 2-24 查找元素

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。

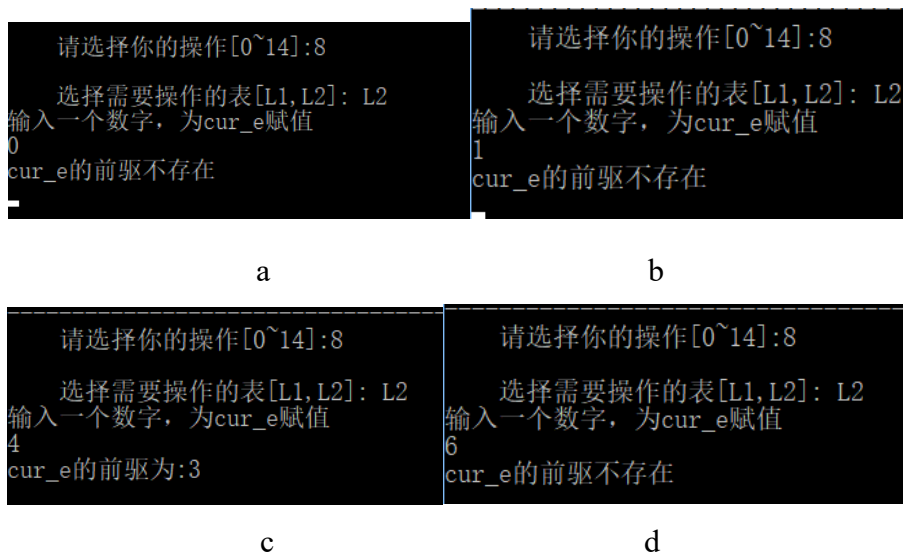
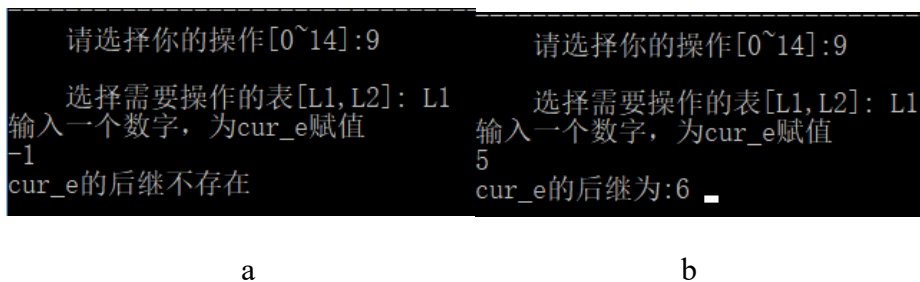


图 2-25 获取前驱

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。



```

请选择你的操作[0~14]:9

选择需要操作的表[L1,L2]: L1
输入一个数字，为cur_e赋值
8
cur_e的后继不存在
    
```

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。

<pre> 请选择你的操作[0~14]:10 选择需要操作的表[L1,L2]: L2 输入要插入的位置 0 位置i不合法 </pre>	<pre> 请选择你的操作[0~14]:10 选择需要操作的表[L1,L2]: L2 输入要插入的位置 1 输入一个数字，为e赋值 0 插入成功 </pre>
--	--

a

b

<pre> 请选择你的操作[0~14]:12 选择需要操作的表[L1,L2]: L2 ----- all elements ----- 0 1 2 3 4 5 ----- end ----- </pre>	<pre> 请选择你的操作[0~14]:10 选择需要操作的表[L1,L2]: L2 输入要插入的位置 6 输入一个数字，为e赋值 0 插入成功 </pre>
--	--

c

d

```

请选择你的操作[0~14]:12

选择需要操作的表[L1,L2]: L2

----- all elements -----
1 2 3 4 5 0
----- end -----
    
```

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。

```

请选择你的操作[0~14]:11
选择需要操作的表[L1,L2]: L1
输入要删除的数据元素位序
0
位置i不合法

```

```

请选择你的操作[0~14]:11
选择需要操作的表[L1,L2]: L1
输入要删除的数据元素位序
1
删除成功
删除的元素值为:1

```

a

b

```

请选择你的操作[0~14]:12
选择需要操作的表[L1,L2]: L1
----- all elements -----
2 3 4 5 6 7 8
----- end -----

```

```

请选择你的操作[0~14]:11
选择需要操作的表[L1,L2]: L1
输入要删除的数据元素位序
4
删除成功
删除的元素值为:4

```

c

d

```

----- all elements -----
1 2 3 5 6 7 8
----- end -----

```

```

请选择你的操作[0~14]:11
选择需要操作的表[L1,L2]: L1
输入要删除的数据元素位序
8
删除成功
删除的元素值为:8

```

e

f

```

请选择你的操作[0~14]:12
选择需要操作的表[L1,L2]: L1
----- all elements -----
1 2 3 4 5 6 7
----- end -----

```

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。

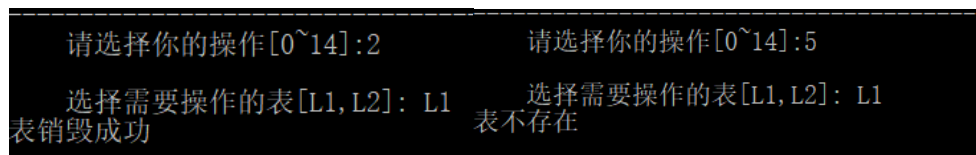


图 2-29 销毁表

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。

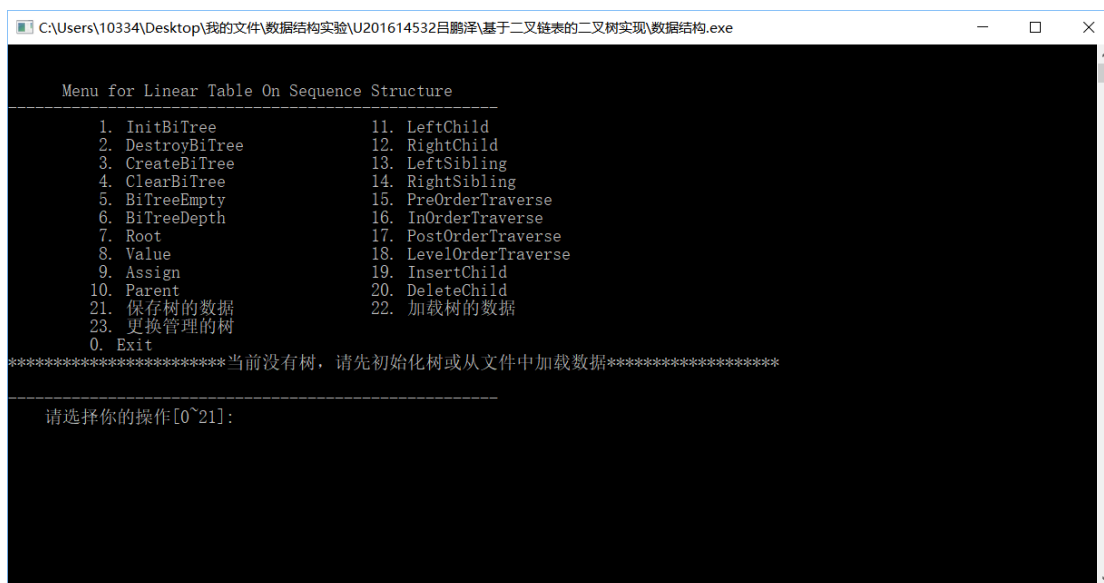


图 3-1 系统载入界面

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。

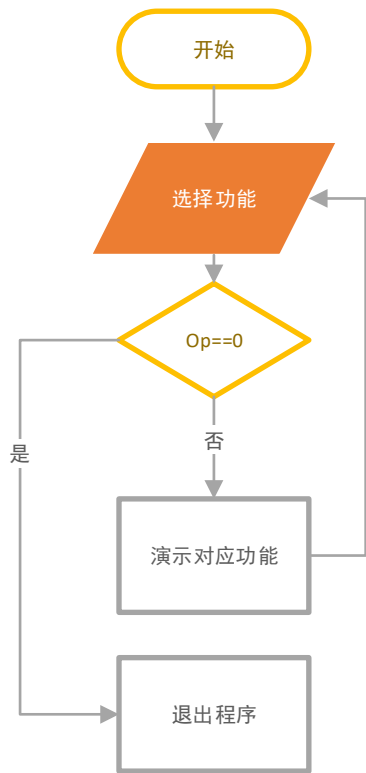


图 3-2 系统总体结构

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。

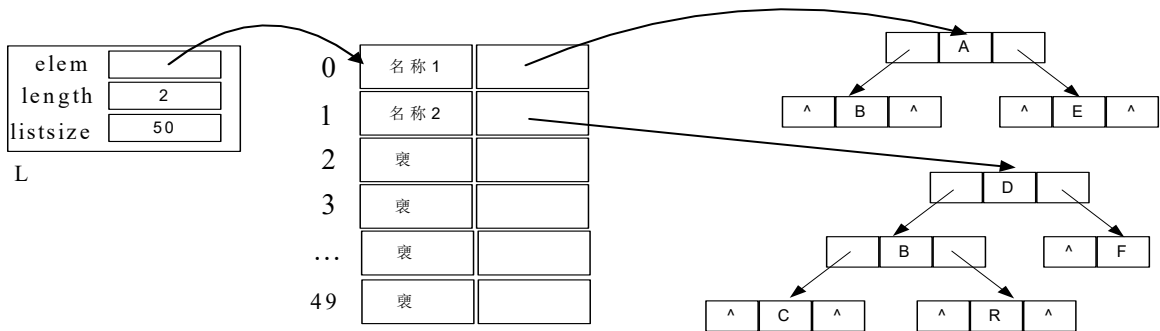


图 3-3 多二叉树管理的物理结构示意图

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)

初始条件：二叉树 T 不存在。

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：

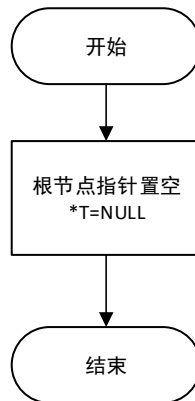


图 3-4 InitBiTree(&T)流程图

(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：

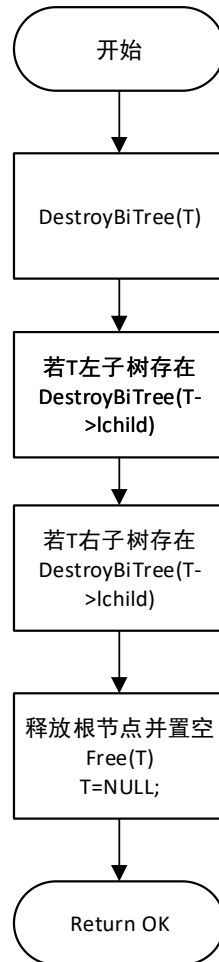


图 3-5 DestroyBiTree(&T)流程图

(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：

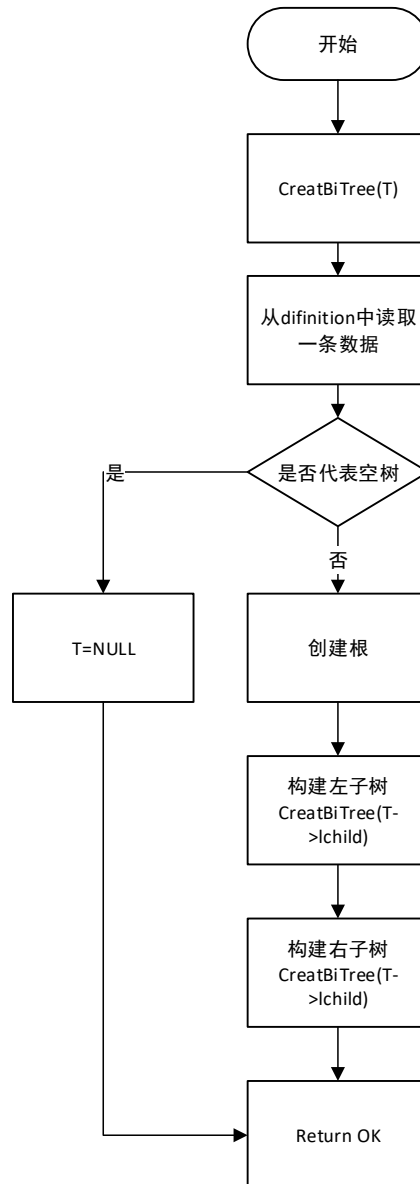


图 3-6 CreateBiTree(&T)流程图

(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：

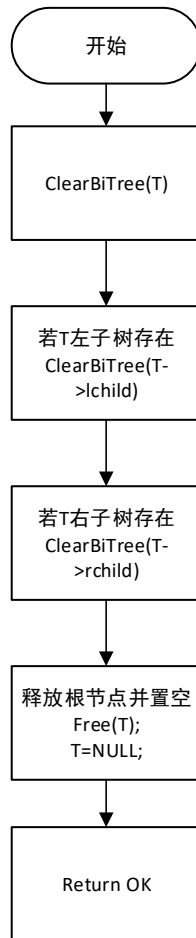


图 3-7 ClearBiTree(&T)流程图

(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：

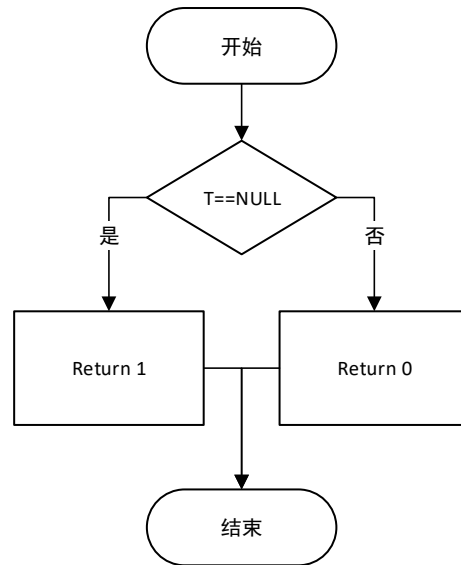


图 3-8 BiTreeEmpty(T)流程图

(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：

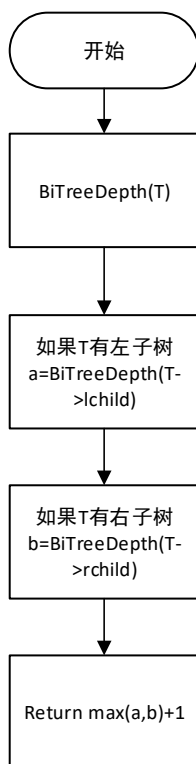


图 3-9 BiTreeDepth(T)流程图

(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：

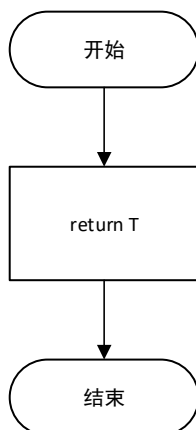


图 3-10 BiTreeDepth(T)流程图

(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：

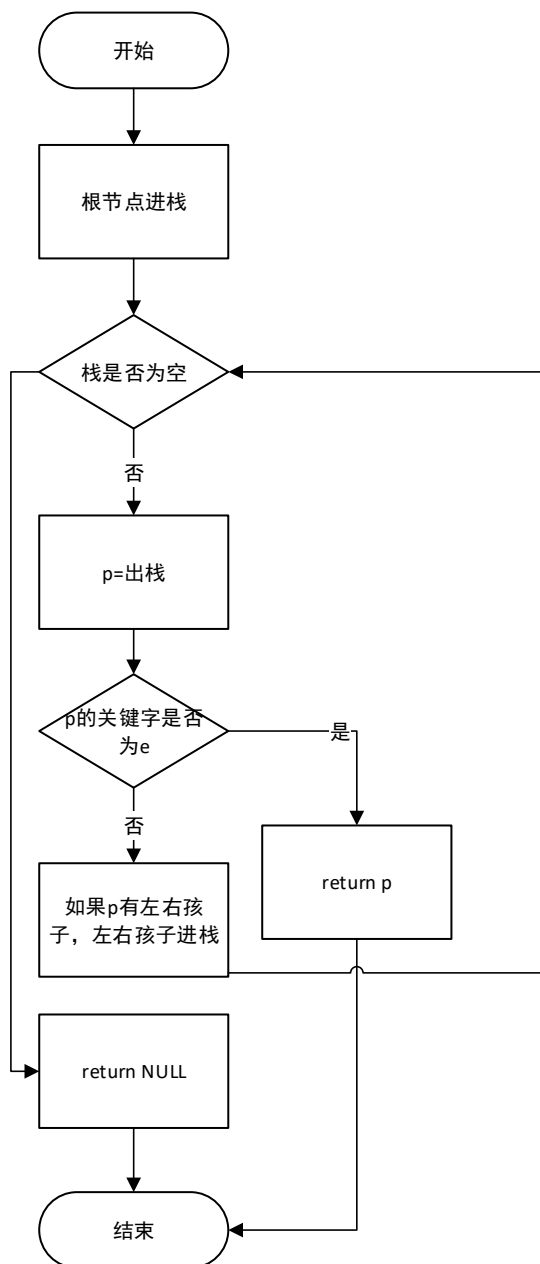


图 3-10 Value(T, e)流程图

(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：

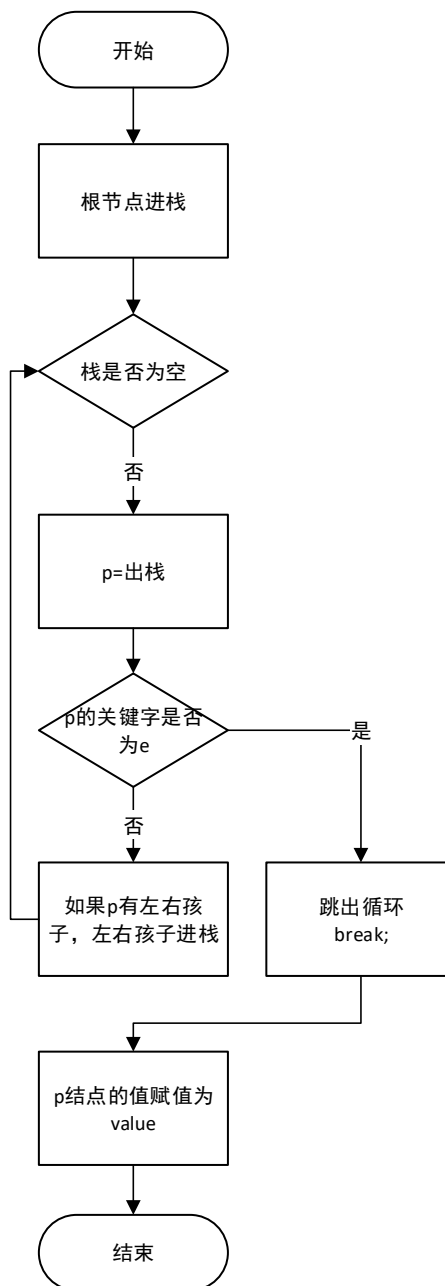


图 3-11 Assign(T, e, value)流程图

(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：

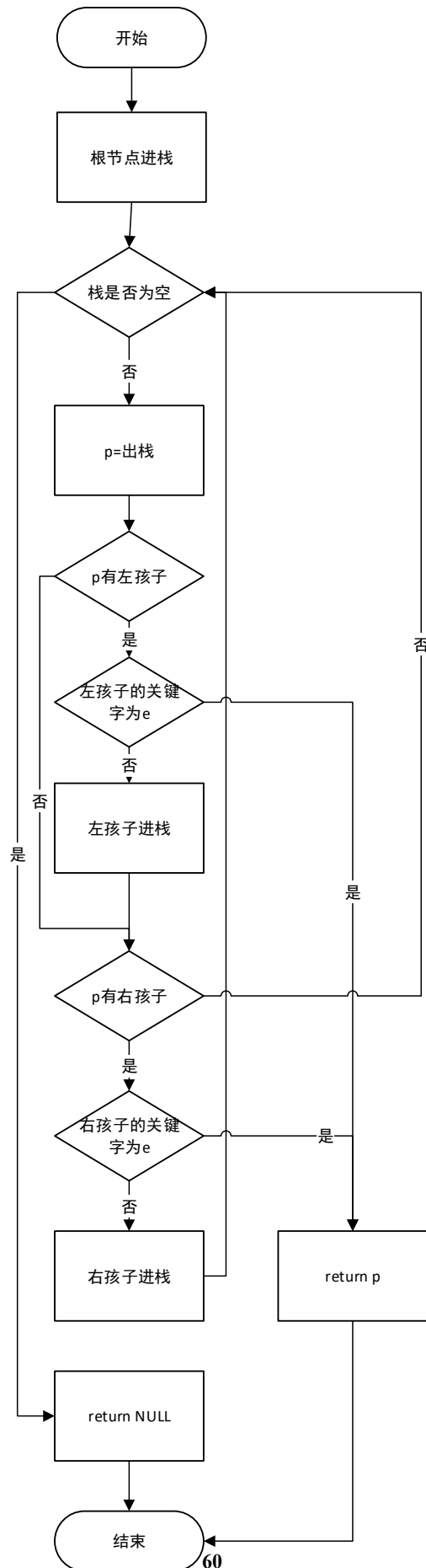


图 3-12 Parent(T, e)流程图

(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：

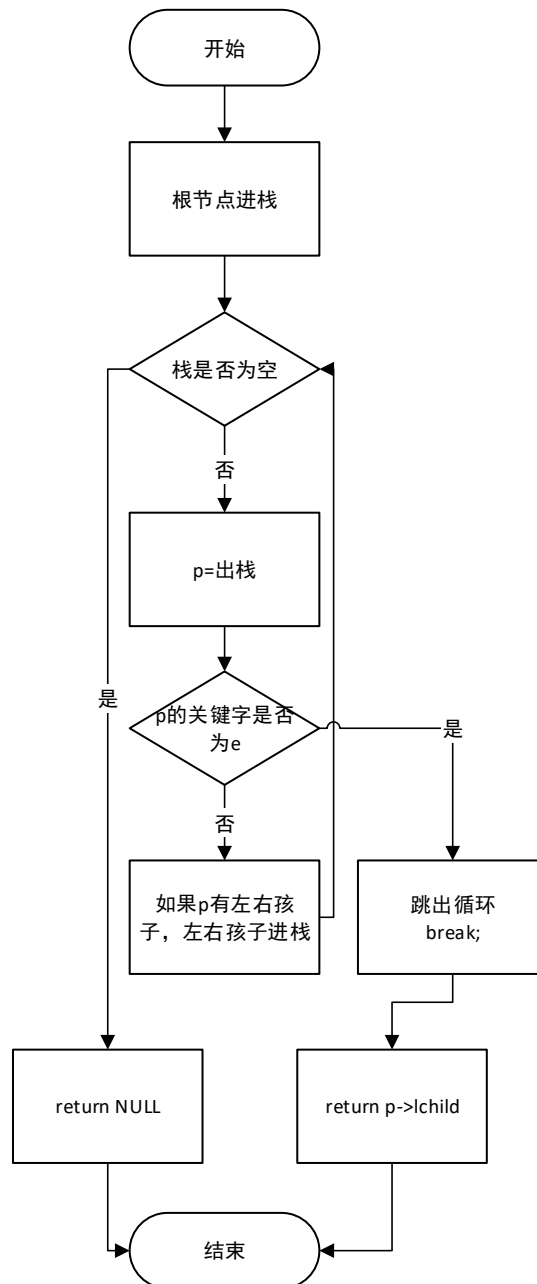


图 3-13 LeftChild(T, e)流程图

(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：

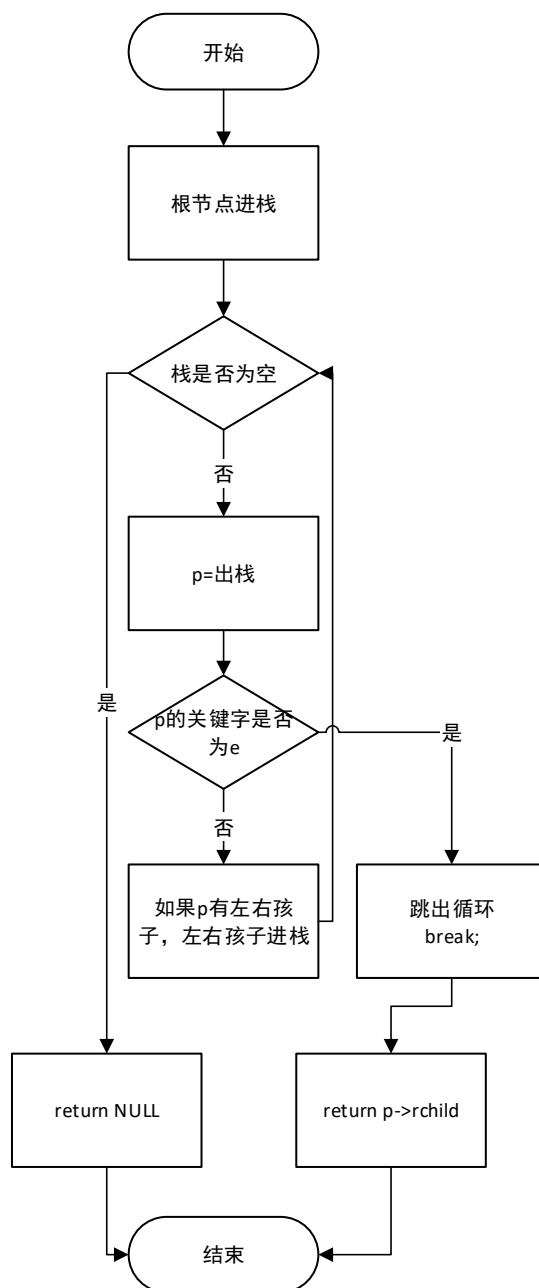


图 3-14 RightChild(T, e)流程图

(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：

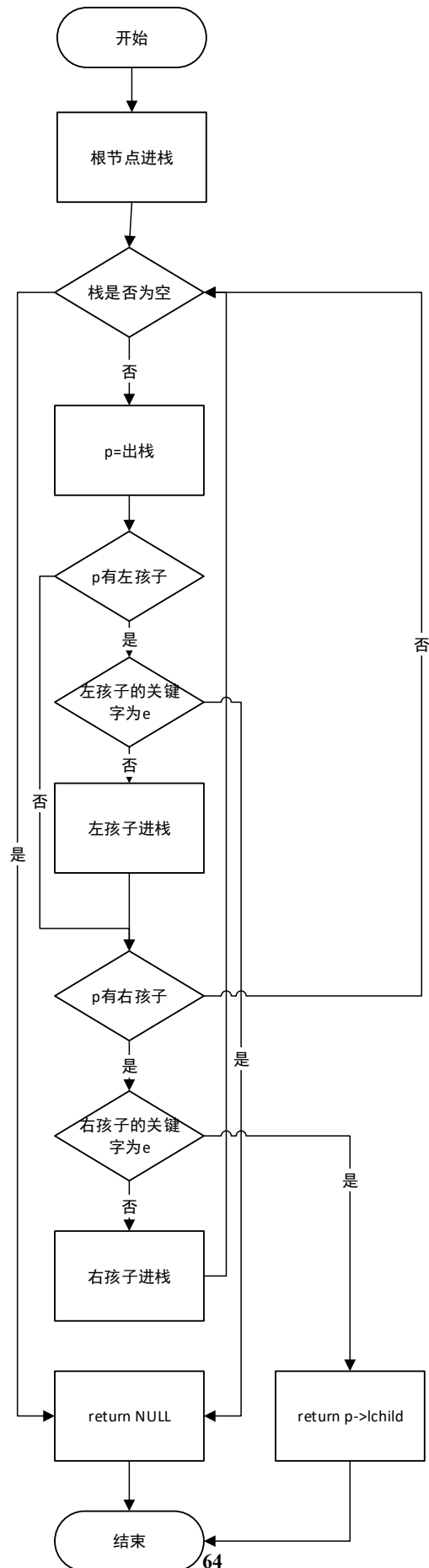


图 3-15 LeftSibling(T, e)流程图

(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：

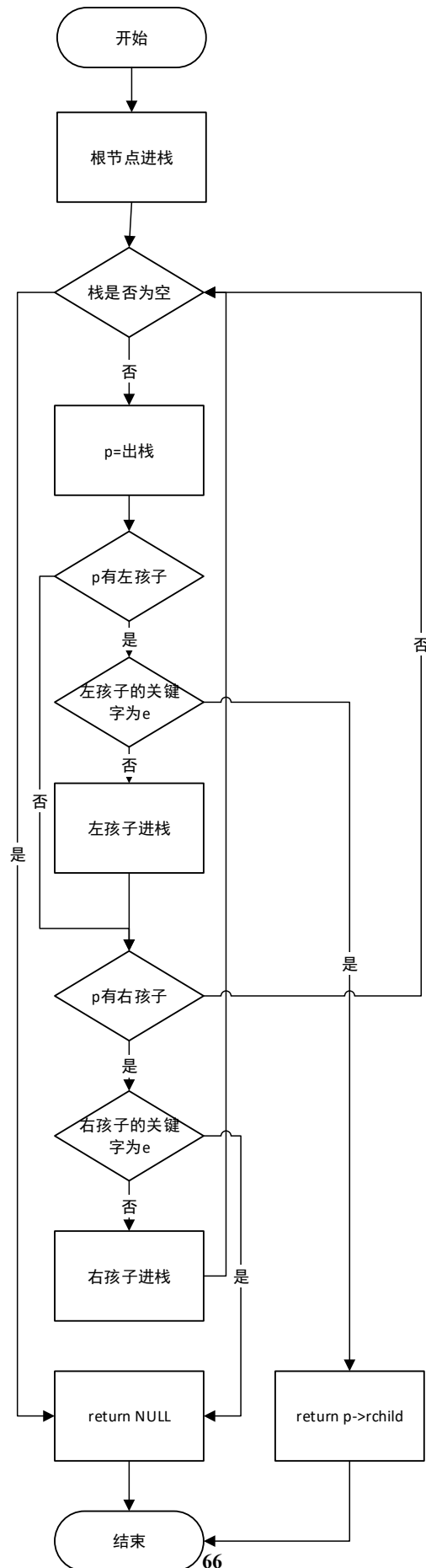


图 3-16 RightSibling(T, e)流程图

(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：

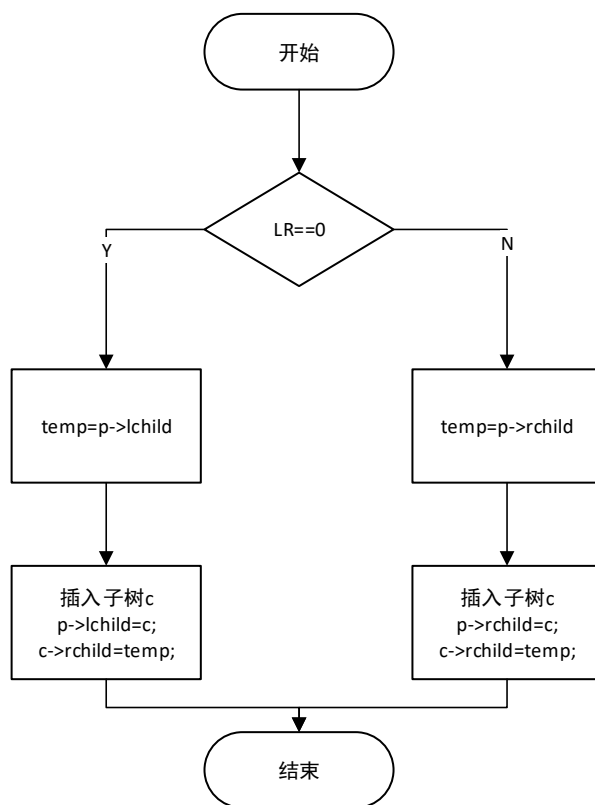


图 3-17 InsertChild(T,p,LR,c)流程图

(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：

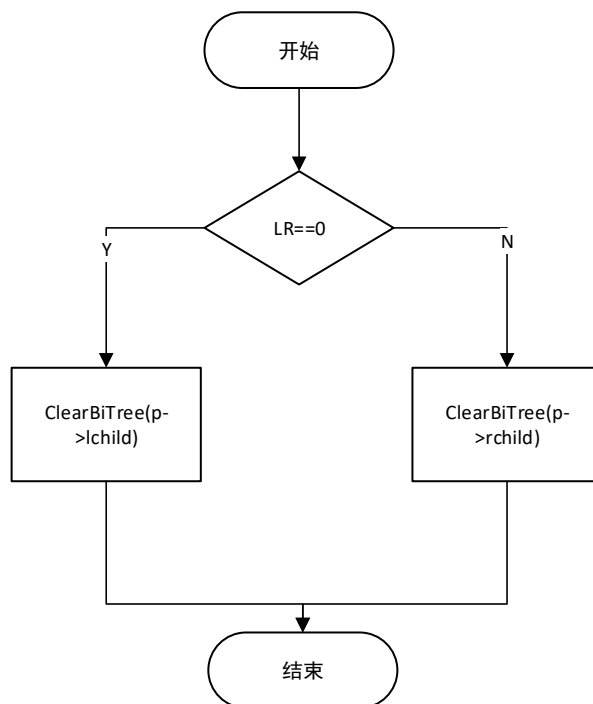


图 3-18 DeleteChild(T, p, LR)流程图

(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：

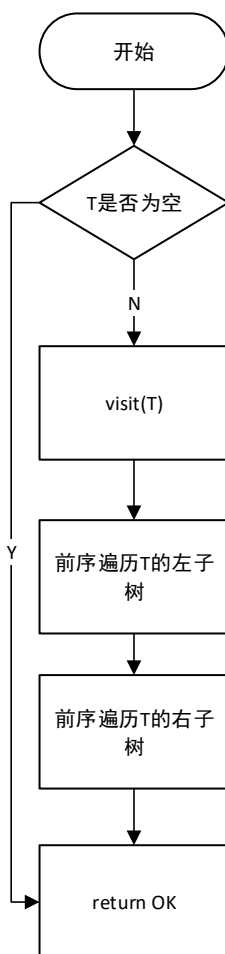


图 3-19 PreOrderTraverse (T,Visit ()) 流程图

(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：

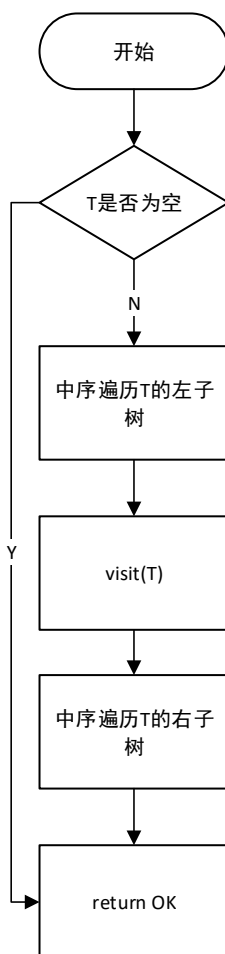


图 3-20 InOrderTraverse (T,Visit ()) 流程图

(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：

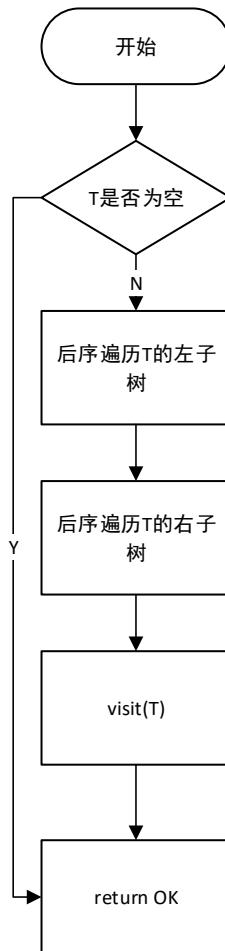


图 3-21 PostOrderTraverse (T,Visit ()) 流程图

(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：

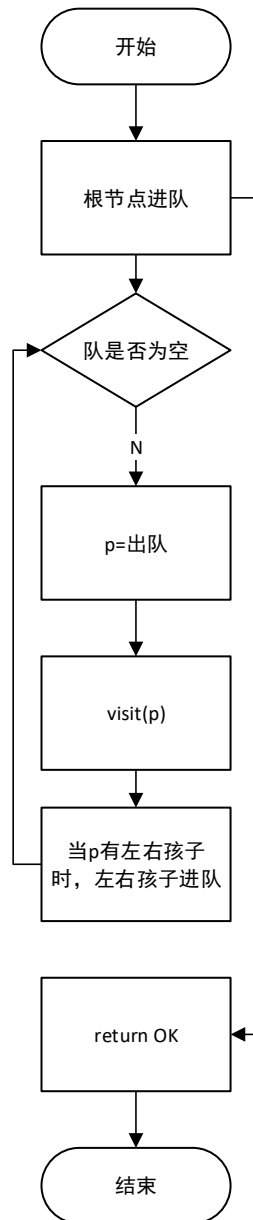


图 3-22 LevelOrderTraverse (T,Visit ()) 流程图

(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：

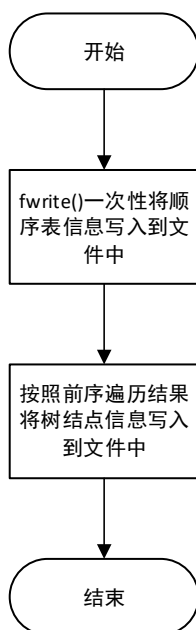


图 3-23 SaveData(L)流程图

(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：

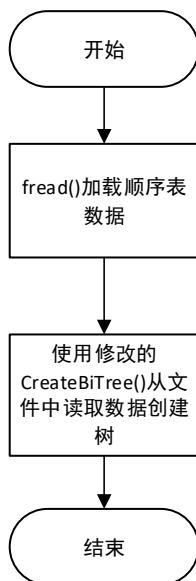
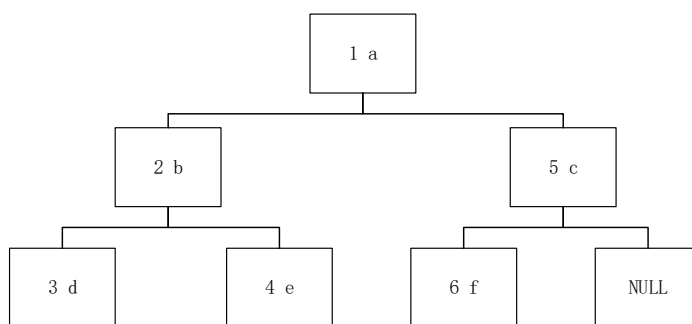


图 3-24 LoadData(&L)流程图

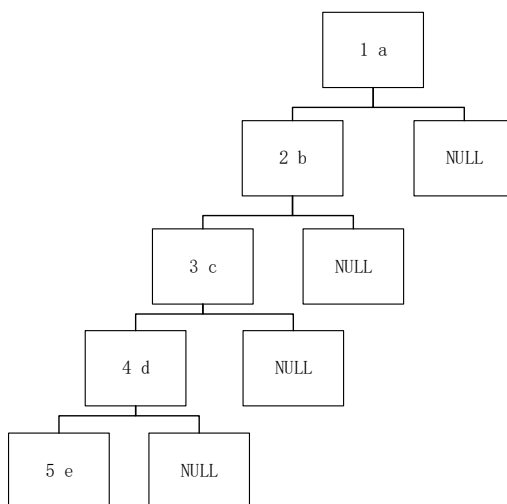
3.3 系统实现

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。

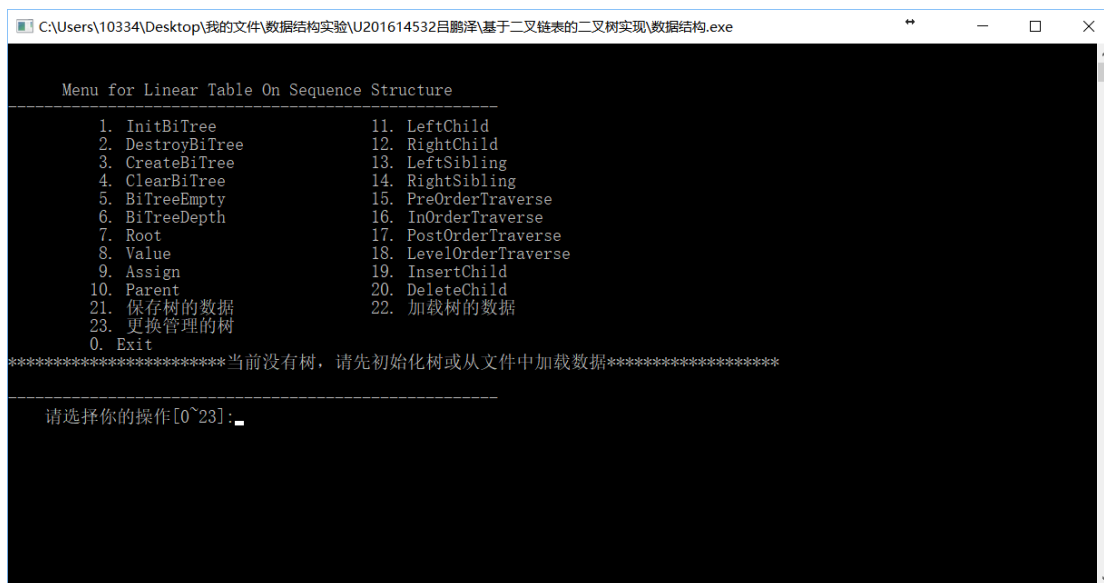
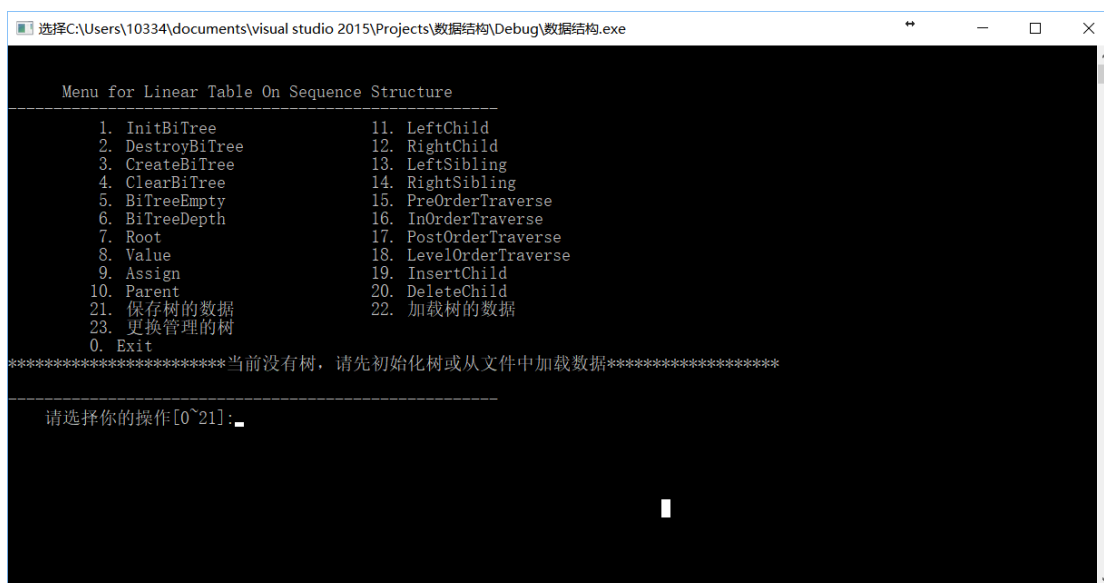


图 3-26 系统载入界面

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。



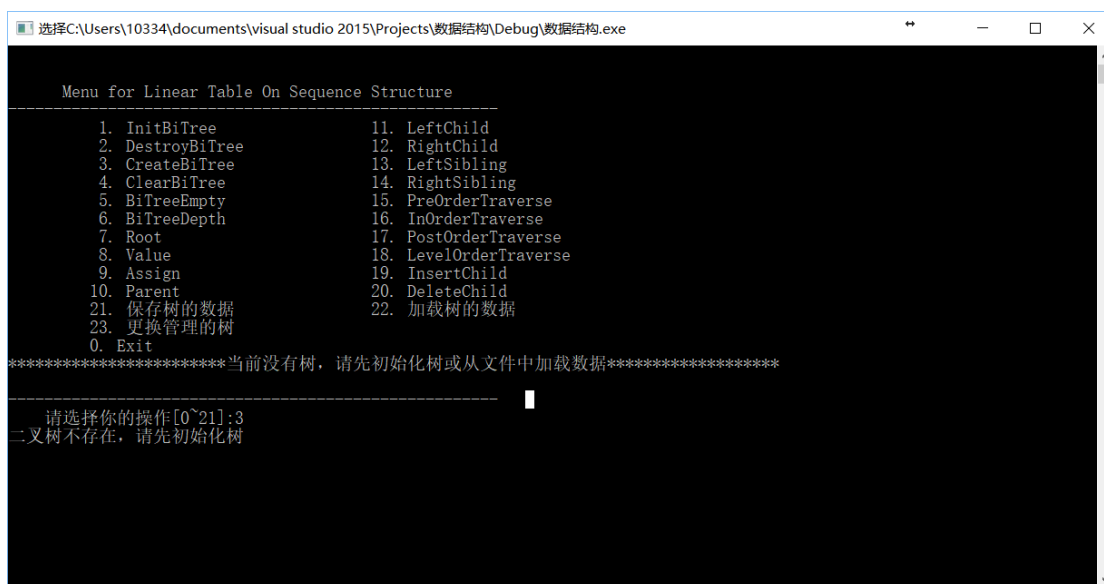
a




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



```

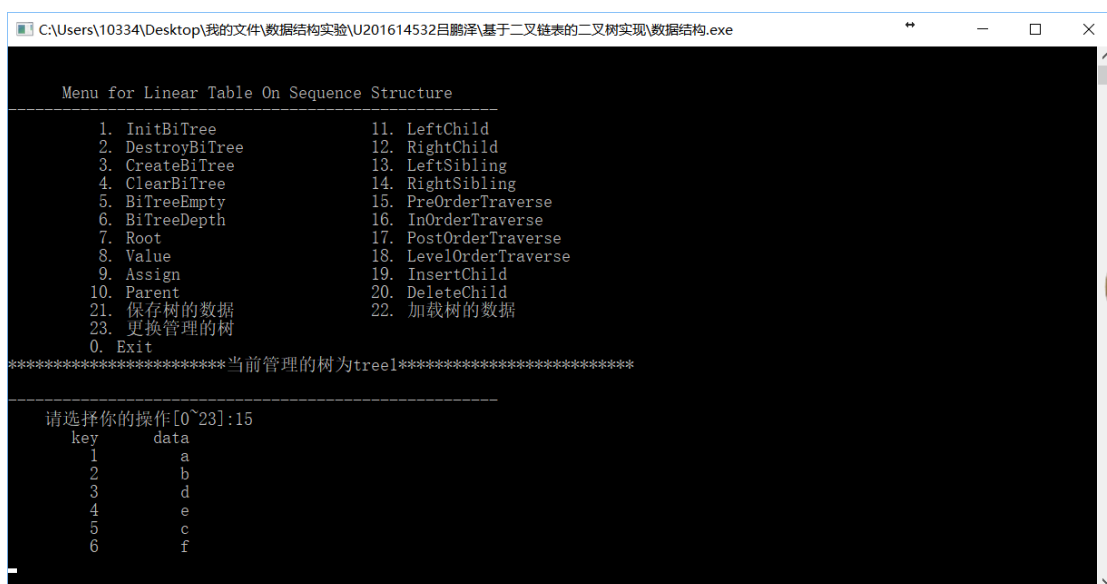
C:\Users\10334\documents\visual studio 2015\Projects\数据结构\Debug\数据结构.exe

Menu for Linear Table On Sequence Structure
-----
1. InitBiTree          11. LeftChild
2. DestroyBiTree       12. RightChild
3. CreateBiTree        13. LeftSibling
4. ClearBiTree         14. RightSibling
5. BiTreeEmpty         15. PreOrderTraverse
6. BiTreeDepth         16. InOrderTraverse
7. Root               17. PostOrderTraverse
8. Value              18. LevelOrderTraverse
9. Assign             19. InsertChild
10. Parent            20. DeleteChild
21. 保存树的数据      22. 加载树的数据
23. 更换管理的树
0. Exit

*****当前没有树，请先初始化树或从文件中加载数据*****

请选择你的操作[0~21]:22
数据加载成功
  
```

b



```

C:\Users\10334\Desktop\我的文件\数据结构实验\U201614532目鹏泽\基于二叉链表的二叉树实现\数据结构.exe

Menu for Linear Table On Sequence Structure
-----
1. InitBiTree          11. LeftChild
2. DestroyBiTree       12. RightChild
3. CreateBiTree        13. LeftSibling
4. ClearBiTree         14. RightSibling
5. BiTreeEmpty         15. PreOrderTraverse
6. BiTreeDepth         16. InOrderTraverse
7. Root               17. PostOrderTraverse
8. Value              18. LevelOrderTraverse
9. Assign             19. InsertChild
10. Parent            20. DeleteChild
21. 保存树的数据      22. 加载树的数据
23. 更换管理的树
0. Exit

*****当前管理的树为tree1*****

请选择你的操作[0~23]:15
key    data
1      a
2      b
3      d
4      e
5      c
6      f
  
```

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

```
C:\Users\10334\Desktop\我的文件\数据结构实验\U201614532吕鹏泽\基于二叉链表的二叉树实现\数据结构.exe

Menu for Linear Table On Sequence Structure
-----
1. InitBiTree          11. LeftChild
2. DestroyBiTree       12. RightChild
3. CreateBiTree        13. LeftSibling
4. ClearBiTree         14. RightSibling
5. BiTreeEmpty         15. PreOrderTraverse
6. BiTreeDepth         16. InOrderTraverse
7. Root               17. PostOrderTraverse
8. Value              18. LevelOrderTraverse
9. Assign             19. InsertChild
10. Parent            20. DeleteChild
21. 保存树的数据      22. 加载树的数据
23. 更换管理的树
0. Exit

*****当前管理的树为tree1*****

-----
请选择你的操作[0~23]:5
树tree1不为空
```

a

```
C:\Users\10334\Desktop\我的文件\数据结构实验\U201614532吕鹏泽\基于二叉链表的二叉树实现\数据结构.exe

Menu for Linear Table On Sequence Structure
-----
1. InitBiTree          11. LeftChild
2. DestroyBiTree       12. RightChild
3. CreateBiTree        13. LeftSibling
4. ClearBiTree         14. RightSibling
5. BiTreeEmpty         15. PreOrderTraverse
6. BiTreeDepth         16. InOrderTraverse
7. Root               17. PostOrderTraverse
8. Value              18. LevelOrderTraverse
9. Assign             19. InsertChild
10. Parent            20. DeleteChild
21. 保存树的数据      22. 加载树的数据
23. 更换管理的树
0. Exit

*****当前管理的树为tree2*****

-----
请选择你的操作[0~23]:5
树tree2不为空
```

b

```

C:\Users\10334\Desktop\我的文件\数据结构实验\U201614532吕鹏译\基于二叉链表的二叉树实现\数据结构.exe

Menu for Linear Table On Sequence Structure

1. InitBiTree          11. LeftChild
2. DestroyBiTree       12. RightChild
3. CreateBiTree        13. LeftSibling
4. ClearBiTree         14. RightSibling
5. BiTreeEmpty         15. PreOrderTraverse
6. BiTreeDepth         16. InOrderTraverse
7. Root               17. PostOrderTraverse
8. Value              18. LevelOrderTraverse
9. Assign             19. InsertChild
10. Parent            20. DeleteChild
21. 保存树的数据      22. 加载树的数据
23. 更换管理的树
0. Exit

*****当前管理的树为tree3*****

请选择你的操作[0~23]:5
树tree3为空
    
```

C

```

C:\Users\10334\Desktop\我的文件\数据结构实验\U201614532吕鹏译\基于二叉链表的二叉树实现\数据结构.exe

Menu for Linear Table On Sequence Structure

1. InitBiTree          11. LeftChild
2. DestroyBiTree       12. RightChild
3. CreateBiTree        13. LeftSibling
4. ClearBiTree         14. RightSibling
5. BiTreeEmpty         15. PreOrderTraverse
6. BiTreeDepth         16. InOrderTraverse
7. Root               17. PostOrderTraverse
8. Value              18. LevelOrderTraverse
9. Assign             19. InsertChild
10. Parent            20. DeleteChild
21. 保存树的数据      22. 加载树的数据
23. 更换管理的树
0. Exit

*****当前管理的树为tree1*****

请选择你的操作[0~23]:4
清空前树的前序遍历结果:
1      a
2      b
3      d
4      e
5      c
6      f
树tree1清空成功
清空后树的前序遍历结果:
    
```

```

C:\Users\10334\Desktop\我的文件\数据结构实验\U201614532吕鹏译\基于二叉链表的二叉树实现\数据结构.exe

Menu for Linear Table On Sequence Structure

1. InitBiTree          11. LeftChild
2. DestroyBiTree       12. RightChild
3. CreateBiTree        13. LeftSibling
4. ClearBiTree         14. RightSibling
5. BiTreeEmpty         15. PreOrderTraverse
6. BiTreeDepth         16. InOrderTraverse
7. Root               17. PostOrderTraverse
8. Value              18. LevelOrderTraverse
9. Assign             19. InsertChild
10. Parent            20. DeleteChild
21. 保存树的数据      22. 加载树的数据
23. 更换管理的树
0. Exit

*****当前管理的树为tree1*****

请选择你的操作[0~23]:5
树tree1为空
    
```


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

```
请选择你的操作[0~23]:6
树tree1的深度为3
```

a

```
请选择你的操作[0~23]:6
树tree2的深度为5
```

b

```
请选择你的操作[0~23]:6
树tree3的深度为0
```

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

```
请选择你的操作[0~23]:7
树tree1的根结点关键字为1，值为a
```

a

```
请选择你的操作[0~23]:7
树tree2的根结点关键字为1，值为a
```

b

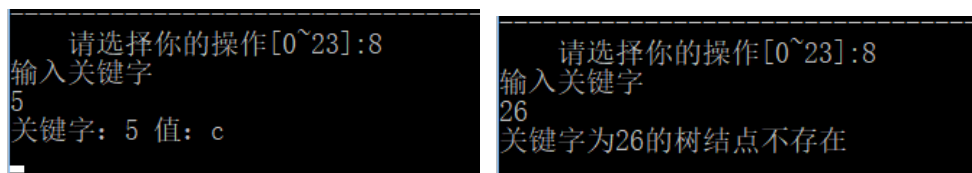
```
请选择你的操作[0~23]:7
树tree3为空
```

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

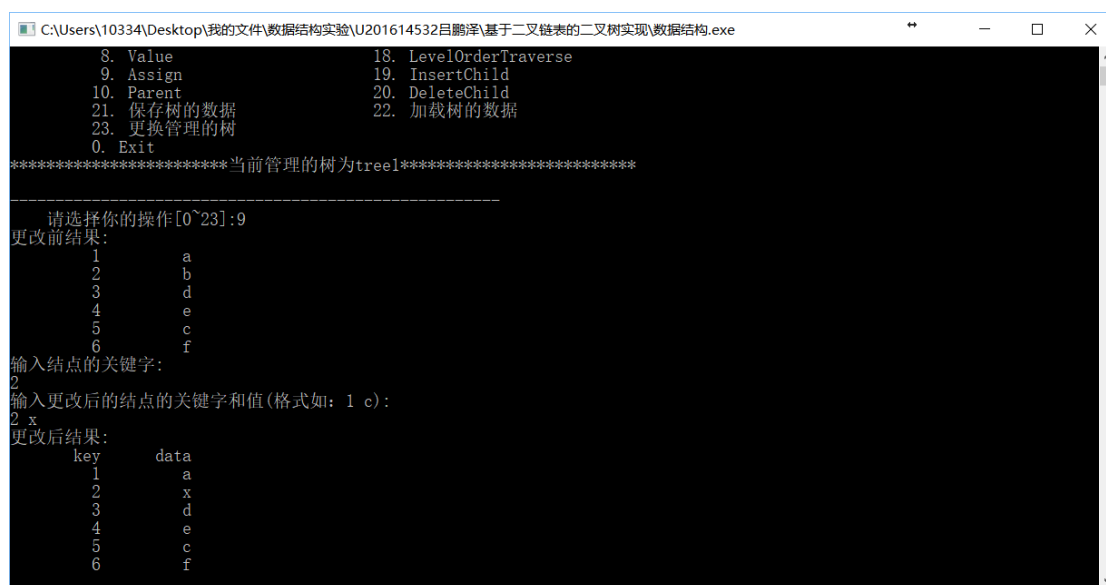


图 3-32 赋值树结点

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。

```

C:\Users\10334\Desktop\我的文件\数据结构实验\U201614532吕鹏译\基于二叉链表的二叉树实现\数据结构.exe

Menu for Linear Table On Sequence Structure
-----
1. InitBiTree          11. LeftChild
2. DestroyBiTree       12. RightChild
3. CreateBiTree        13. LeftSibling
4. ClearBiTree         14. RightSibling
5. BiTreeEmpty         15. PreOrderTraverse
6. BiTreeDepth         16. InOrderTraverse
7. Root               17. PostOrderTraverse
8. Value              18. LevelOrderTraverse
9. Assign             19. InsertChild
10. Parent            20. DeleteChild
21. 保存树的数据      22. 加载树的数据
23. 更换管理的树
0. Exit

*****当前管理的树为tree1*****

请选择你的操作[0~23]:10
输入关键字
1
无双亲结点
    
```

a

```

C:\Users\10334\Desktop\我的文件\数据结构实验\U201614532吕鹏译\基于二叉链表的二叉树实现\数据结构.exe

Menu for Linear Table On Sequence Structure
-----
1. InitBiTree          11. LeftChild
2. DestroyBiTree       12. RightChild
3. CreateBiTree        13. LeftSibling
4. ClearBiTree         14. RightSibling
5. BiTreeEmpty         15. PreOrderTraverse
6. BiTreeDepth         16. InOrderTraverse
7. Root               17. PostOrderTraverse
8. Value              18. LevelOrderTraverse
9. Assign             19. InsertChild
10. Parent            20. DeleteChild
21. 保存树的数据      22. 加载树的数据
23. 更换管理的树
0. Exit

*****当前管理的树为tree1*****

请选择你的操作[0~23]:10
输入关键字
4
双亲结点地址为: 030342A0, 数据为: 2 b
    
```

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。

```

*****当前管理的树为tree1*****

请选择你的操作[0~23]:11
输入关键字
5
左孩子结点地址为: 0303C3D8, 数据为: 6 f
    
```

a

```
*****当前管理的树为tree1*****
-----
      请选择你的操作[0~23]:12
输入关键字
5
无右孩子
```

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。

```
-----
      请选择你的操作[0~23]:13      请选择你的操作[0~23]:14
输入关键字      输入关键字
2              2
无左兄弟      该结点右兄弟地址为: 03034180, 数据为: 5 c
```

a

```
-----
      请选择你的操作[0~23]:13      请选择你的操作[0~23]:14
输入关键字      输入关键字
6              6
无左兄弟      无右兄弟
```

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

```
-----
      请选择你的操作[0~23]:15      请选择你的操作[0~23]:16
key      data      key      data
1         a         3         d
2         b         2         b
3         d         4         e
4         e         1         a
5         c         6         f
6         f         5         c
```

a

b

请选择你的操作[0~23]:17	
key	data
3	d
4	e
2	b
6	f
5	c
1	a

c

请选择你的操作[0~23]:18	
key	data
1	a
2	b
5	c
3	d
4	e
6	f

d

图 3-36 二叉树 tree1 遍历

请选择你的操作[0~23]:15	
key	data
1	a
2	b
3	c
4	d
5	e

a

请选择你的操作[0~23]:16	
key	data
5	e
4	d
3	c
2	b
1	a

b

请选择你的操作[0~23]:17	
key	data
5	e
4	d
3	c
2	b
1	a

c

请选择你的操作[0~23]:18	
key	data
1	a
2	b
3	c
4	d
5	e

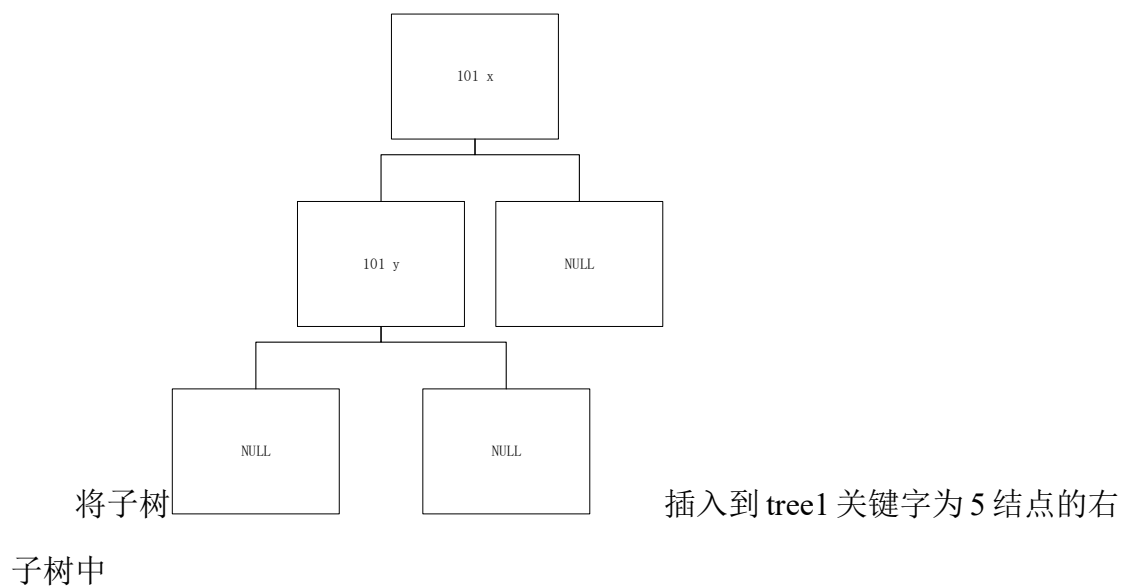
d

图 3-37 二叉树 tree2 遍历

请选择你的操作[0~23]:15	
二叉树tree3为空	

图 3-38 二叉树 tree3 遍历

14.插入子树



```

C:\Users\10334\Desktop\我的文件\数据结构实验\U201614532吕鹏泽\基于二叉链表的二叉树实现\数据结构.exe
0. Exit
*****当前管理的树为tree1*****
-----
请选择你的操作[0~23]:19
请输入将要创建的子树
输入数据组数:
5
输入数据
100 x
101 y
0 #
0 #
0 #
子树c创建成功
子树c的前序遍历结果:
100 x
101 y
输入要插入子树的结点的关键字:5
输入要插入的位置: 0. 左子树 1. 右子树1
子树插入成功, 插入后的前序遍历结果:
1 a
2 b
3 d
4 e
5 c
6 f
100 x
101 y
    
```

图 3-39 插入子树

15.删除子树

删除 tree1 关键字为 1 的左子树

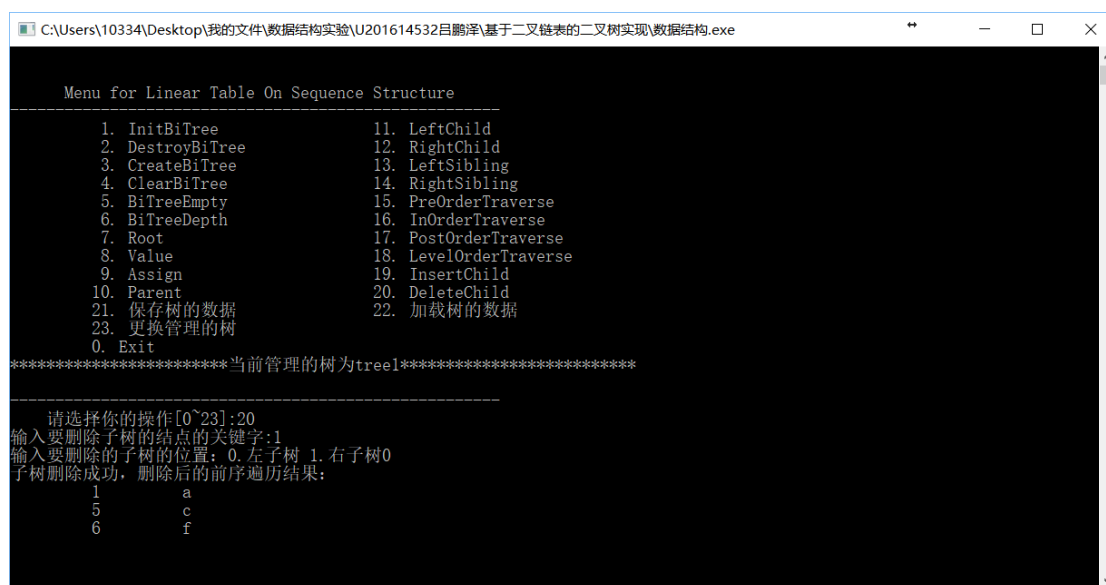


图 3-40 删除子树

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

在本次实验中，我以邻接表作为图的物理结构，用 C 语言实现了单个有向带权图的基本运算，并添加了数据保存和数据加载的功能。在实验过程中我将图顶点的数据域 data 抽象成为一个整型变量 key，代表该顶点的关键字，一个字符变量 c，代表该顶点保存的数据。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,我的程序实现了图的十三个基本运算:创建图,销毁图,查找顶点,获得顶点值,顶点赋值,获得第一邻接点,获得下一邻接点,插入顶点,删除顶点,插入弧,删除弧,深度优先搜索遍历,广度优先搜索遍历。两个附加功能:保存图数据、加载图数据,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,图顶点使用二进制的方式将整块线性表写入到文件中进行存储,存储效率高,速度快。邻接边将边数据输出到文件中进行存储,节省空间。(3)在调用功能函数前已对各功能的初始条件进行检验,确保程序不会异常退出。(4)运用了递归,栈,队列。

(说明:对于弧 $a \rightarrow b$,称 a 为弧尾, b 为弧头)

4.2 系统设计

4.2.1 系统总体设计:

在程序中实现消息处理与图基本运算的演示,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。

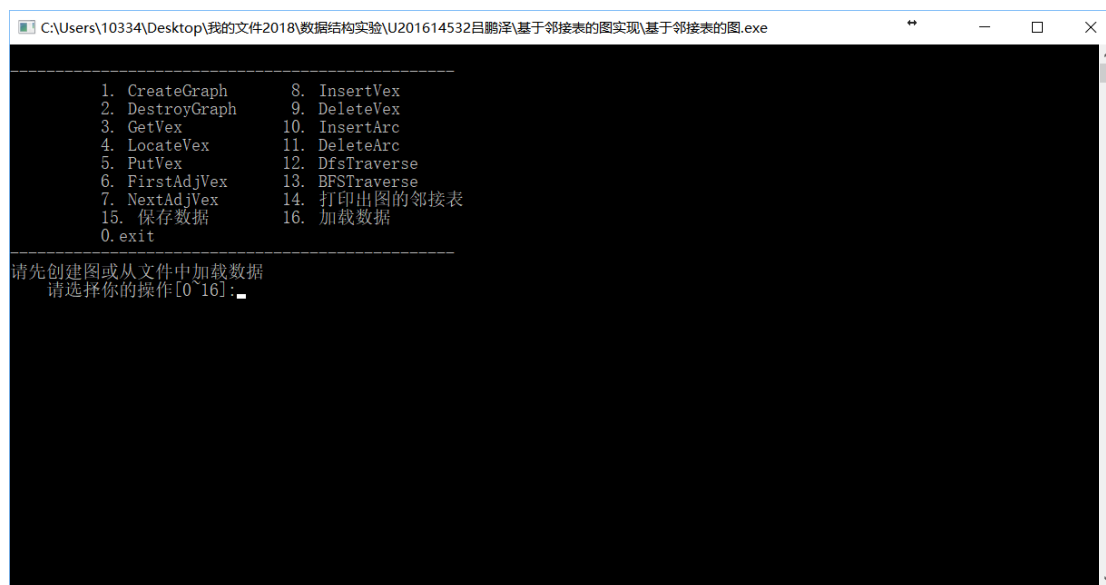


图 4-1 系统载入界面

用户通过输入在[0,16]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图的,用户可以选择功能 1 创

建图或者选择功能 16 从数据文件中加载上次保存的数据。此外，只有当不存在图时才能从文件中读取数据。当已存在图时，需要销毁后才能重新创建图。

演示的过程可抽象为如图 4-2 所示的流程图。

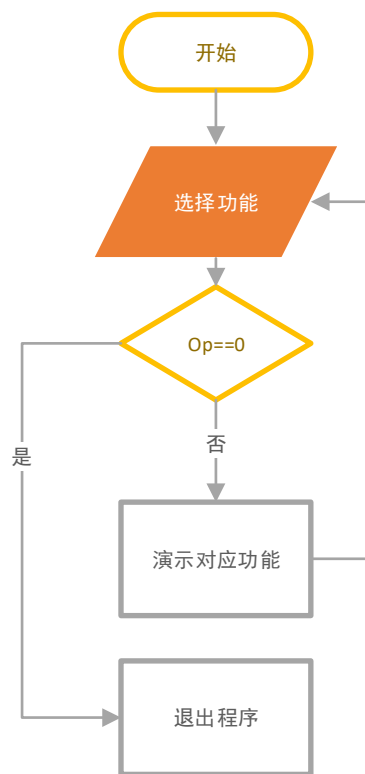


图 4-2 系统总体结构

4.2.2 图的物理结构

如图 4-3，表示的是基于邻接表图的物理结构，顺序表 L 记录着图顶点的信息，包括图顶点关键字，值，和该顶点的第一邻接边。对于一个顶点，其边结点包含该边的权和该顶点的邻接点，并指向下一条邻接边。

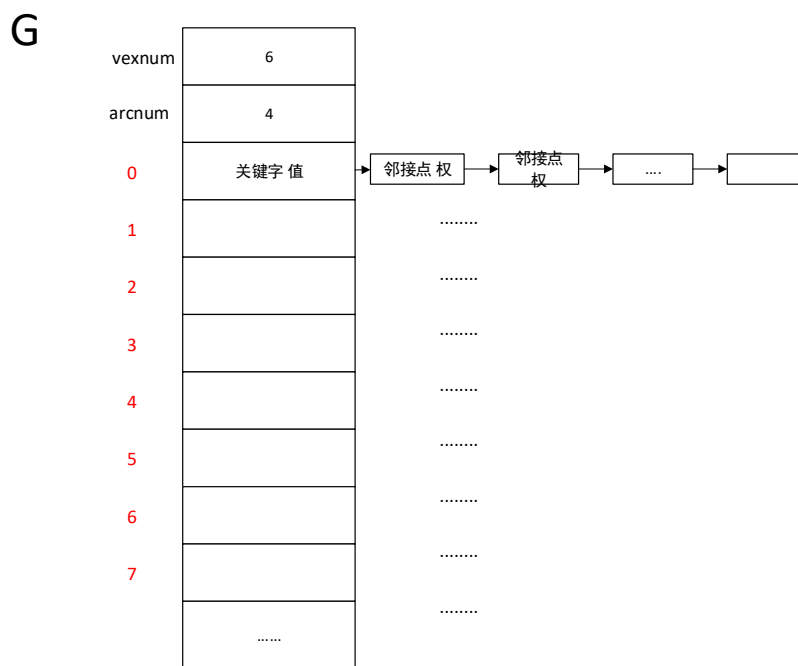


图 4-3 基于邻接表图的物理结构示意图

4.2.3 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

MAX_VERTEX_NUM 为最大顶点数，MAX_ARC_NUM 为最大弧数。
V_DATA 为顶点的数据文件，E_DATA 为弧的数据文件。C 语言描述如下所示：

```
#define MAX_VERTEX_NUM 200
#define MAX_ARC_NUM 2000
```

```
#define V_DATA "VData"
```

```
#define E_DATA "EData"
```

3.相关结构体定义

(1) 图结构

ALGraph 记录图的顶点数，边数和顶点信息。C 语言描述如下所示：

```
typedef struct {
    AdjList vertices;
    int vexnum, arcnum;//点数、边数
}ALGraph;
```

(2) 图顶点结构

图顶点包括指针域和数据域，指针指向其第一邻接边，数据域抽象为 VertexType 类型，包括树结点的关键字和数据，关键字为 KEY 类型，数据为 char 类型，我的程序中将关键字设为 int 型。C 语言描述如下所示：

```
typedef struct VNode {
    VertexType data;
    ArcNode *firstarc;
}VNode, AdjList[MAX_VERTEX_NUM];
typedef struct VertexType
{
    int key;
    char c;
}VertexType;
typedef int KEY;
```

(3) 边结点结构

adjeces 记录该弧的弧头顶点位置，nextarc 指向下一条有相同弧尾的弧，info 记录弧的信息，此处为弧的权值。C 语言描述如下所示：

```
typedef struct ArcNode {
    int adjves;
    struct ArcNode *nextarc;
```

```
    InfoType info;  
}ArcNode;
```

(4) 图关系结构

在创建图是需要用户输入顶点间的关系，用此结构记录，tail 为弧尾，head 为弧头，info 为弧权值。C 语言描述如下所示：

```
typedef struct {  
    KEY tail;  
    KEY head;  
    InfoType info;  
}VertexRalation;
```

4.2.4 算法设计

用 v 表示图的顶点数， e 表示图的边数。

(1) CreateCraph(&G,V,VR)

初始条件：V 是图的顶点集，VR 是图的关系集。

操作结果：按 V 和 VR 的定义构造图 G。

算法思想：1.创建顶点。2.按照关系 VR 创建弧

时间复杂度： $O(e+v)$

流程图：

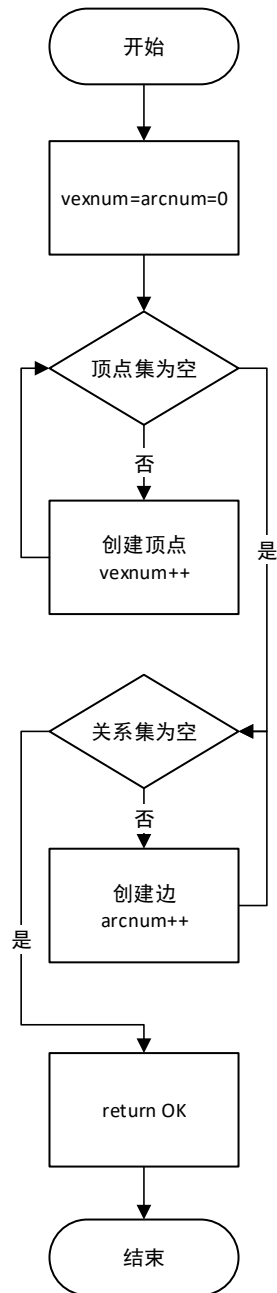


图 4-4 CreateCraph()流程图

(2) DestroyCraph(&G)

初始条件：图 G 存在。

操作结果：销毁图 G。

算法思想：1.释放边结点的空间 2.图顶点数和边数置 0

时间复杂度： $O(e+v)$

流程图：

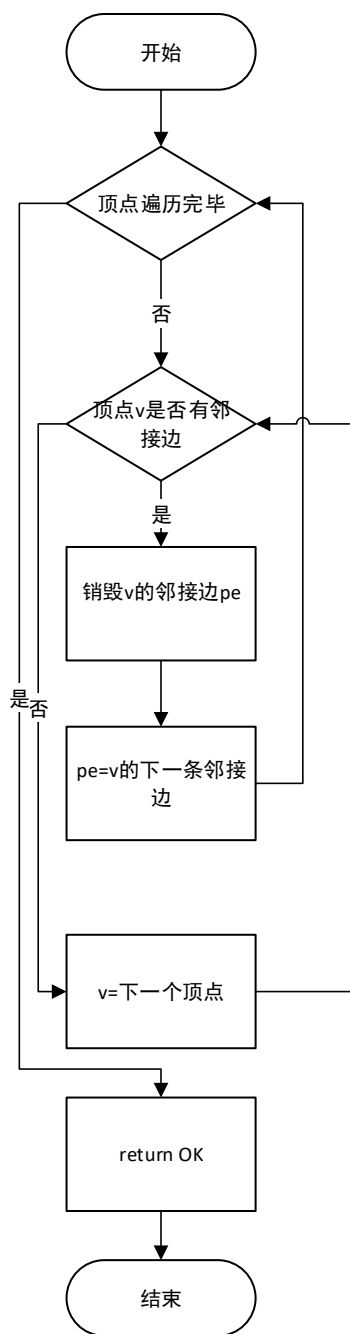


图 4-5 DestroyCraph()流程图

(3) LocateVex(G,u)

初始条件：图 G 存在，u 和 G 中的顶点具有相同特征。

操作结果：若 u 在图 G 中存在，返回顶点 u 的位置信息，否则返回其它信息。

算法思想：顺序查找顶点，找到返回顶点位置，未找到返回-1。

时间复杂度：O(v)

流程图：

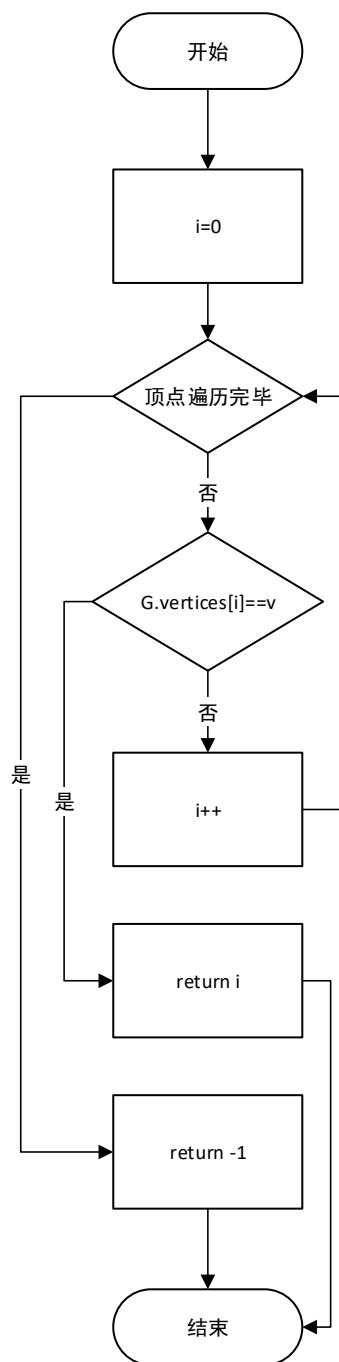


图 4-6 LocateVex()流程图

(4) GetVex(&G, v)

初始条件：图 G 存在，v 是 G 中的某个顶点。

操作结果：返回 v 的地址。

算法思想：顺序查找顶点，找到返回顶点地址，未找到返回 NULL。

时间复杂度：O(v)

流程图：

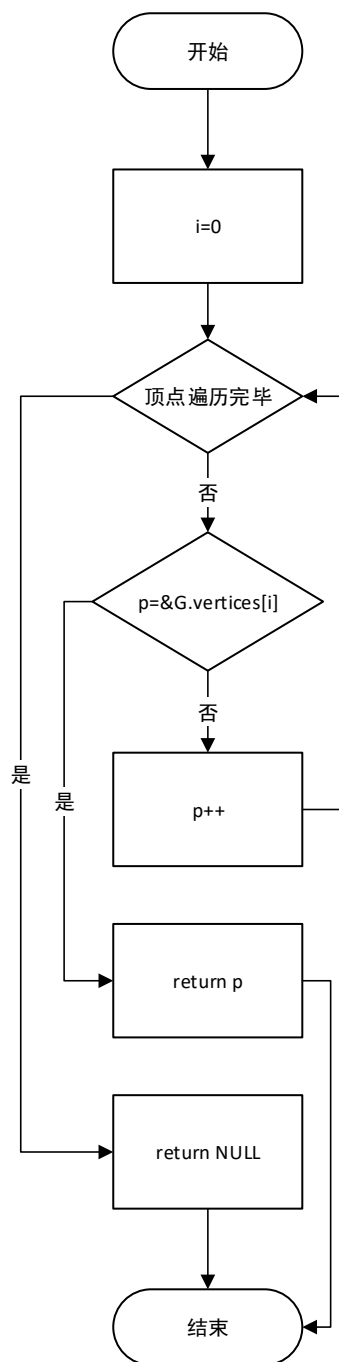


图 4-7 GetVex()流程图

(5) PutVex(&G, v)

初始条件：图 G 存在，v 是 G 中的某个顶点。

操作结果：对 v 赋值 value。

算法思想：1.使用 GetVex()获取 v 的地址。2.修改 v 的值

时间复杂度：O(v)

流程图：

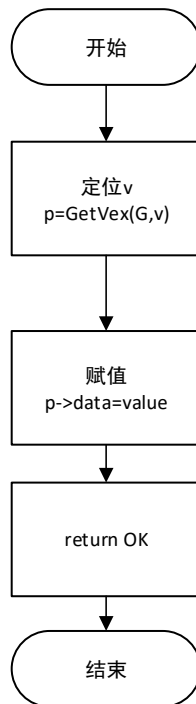


图 4-8 PutVex()流程图

(6) FirstAdjVex(&G, v)

初始条件：图 G 存在，v 是 G 的一个顶点。

操作结果：返回 v 的第一个邻接顶点，如果 v 没有邻接顶点，返回空。

时间复杂度：O(1)

算法思想：1.使用 GetVex()获取 v 的地址。2.返回 v 的第一邻接点。

流程图：

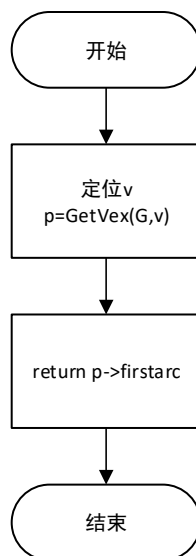


图 4-9 FirstAdjVex ()流程图

(7) NextAdjVex(&G, v, w)

初始条件：图 G 存在，v 是 G 的一个顶点,w 是 v 的邻接顶点。

操作结果：返回 v 的（相对于 w）下一个邻接顶点，如果 w 是最后一个邻接顶点，返回空。

算法思想：1.使用 GetVex()获取 v 的地址。2.查找 v 的邻接点 w。3.返回 w 的下一个邻接点

时间复杂度：O(e/v)

流程图：

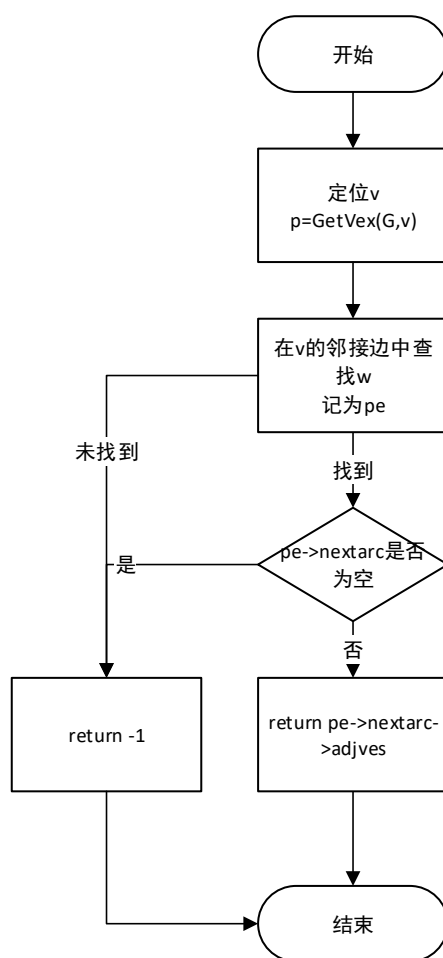


图 4-10 NextAdjVex ()流程图

(8) InsertVex(&G,v)

初始条件：图 G 存在，v 和 G 中的顶点具有相同特征。

操作结果：在图 G 中增加新顶点 v。

时间复杂度: $O(v)$

算法思想: 在 G 顶点的线性表的尾部插入 v

流程图:

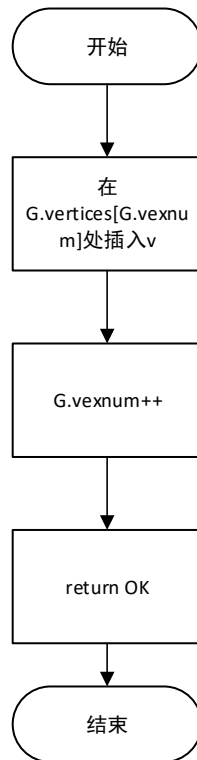


图 4-11 InsertVex ()流程图

(9) DeleteVex(&G,v)

初始条件: 图 G 存在, v 是 G 的一个顶点。

操作结果: 在图 G 中删除顶点 v 和与 v 相关的弧。

时间复杂度: $O(e+v)$

算法思想: 1.删除与 v 相关联的边 2.删除顶点 v 3.修改在 v 之后的顶点的位置

流程图:

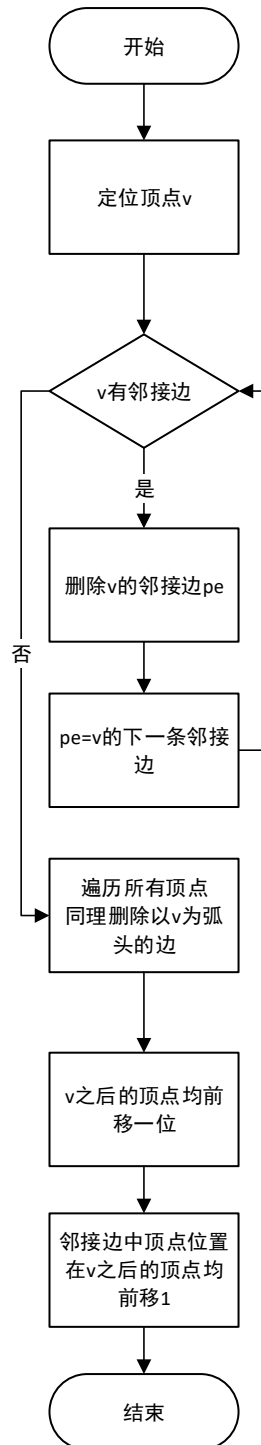


图 4-12 DeleteVex ()流程图

(10) InsertArc(&G,v,w)

初始条件：图 G 存在，v、w 是 G 的顶点。

操作结果：在图 G 中增加弧<v,w>，如果图 G 是无向图，还需要增加<w,v>。

时间复杂度：O(v)

算法思想：1.定位 v 2.在 v 的边的尾部插入弧 (v, w)

流程图：

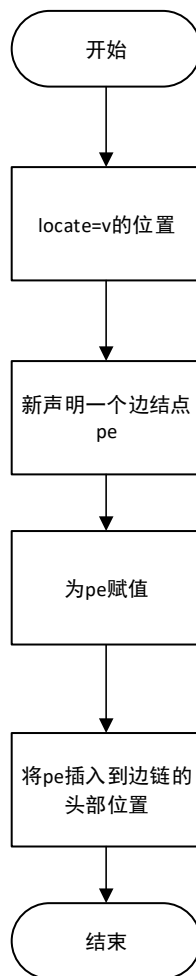


图 4-13 InsertArc ()流程图

(11) DeleteArc(&G,v,w)

初始条件：图 G 存在， v 、 w 是 G 的顶点。

操作结果：在图 G 中删除弧 $\langle v, w \rangle$ ，如果图 G 是无向图，还需要删除 $\langle w, v \rangle$ 。

时间复杂度： $O(v)$

算法思想：1.定位 v 2.查找连接 w 的邻接边 3.删除边 (v, w)

流程图：

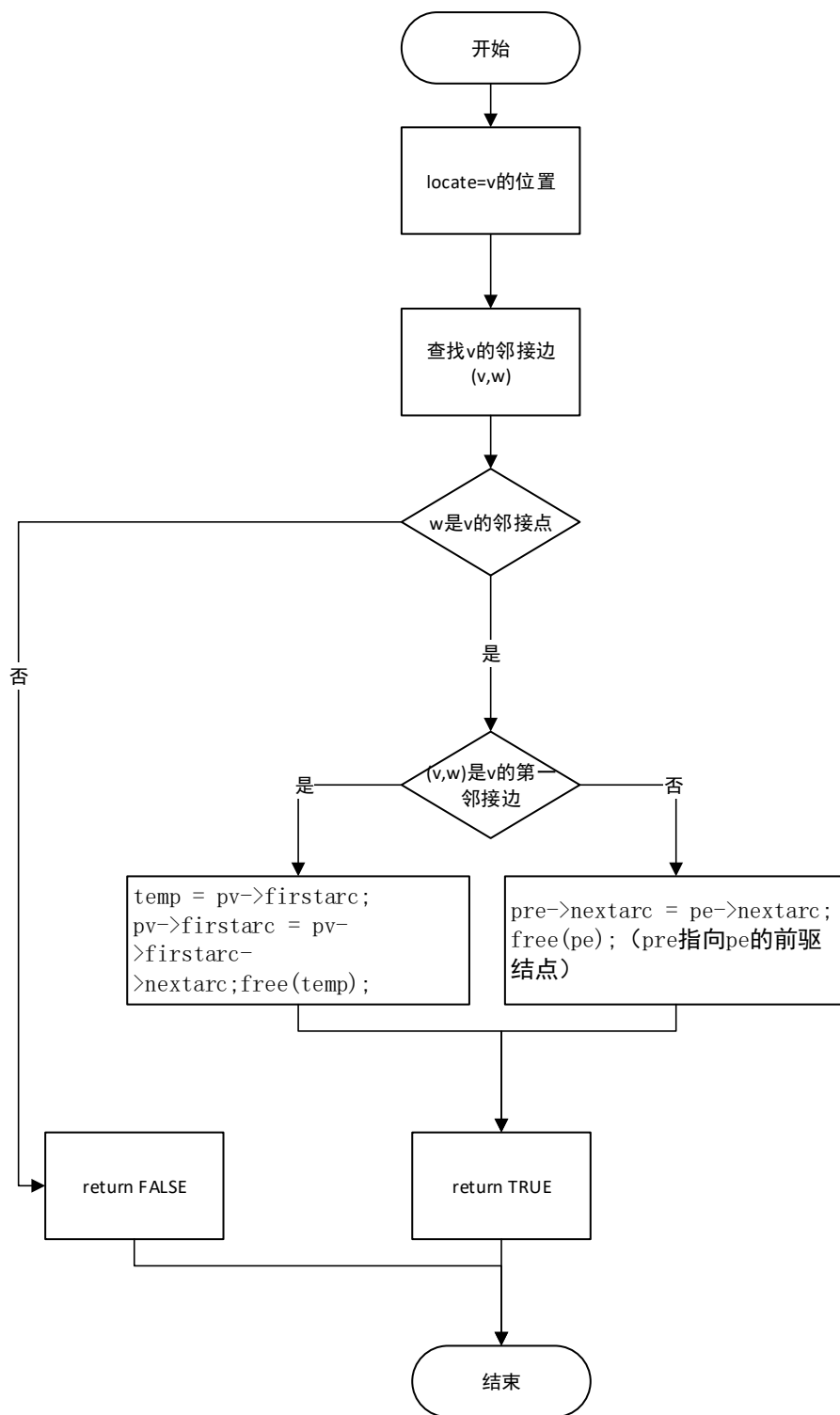


图 4-14 DeleteArc ()流程图

(12) DFSTraverse(G,visit())

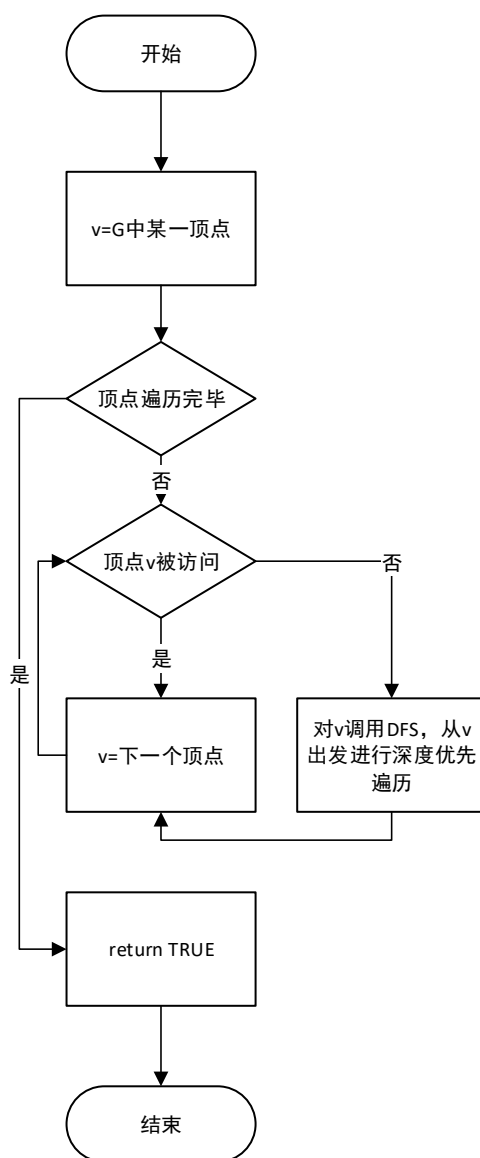
初始条件：图 G 存在。

操作结果：对图 G 进行深度优先搜索遍历，依次对图中的每一个顶点使用函数 `visit` 访问一次，且仅访问一次。

算法思想：(1) 访问出发顶点 v 、作访问标记、 v 入栈；(2) 当栈非空时，分下列情况处理，直到栈空为止：如果栈顶之顶点存在 未访问过的邻接顶点 w ，则访问顶点 w 、作访问标记、 w 入栈，否则出栈。

时间复杂度： $O(e+v)$

流程图：



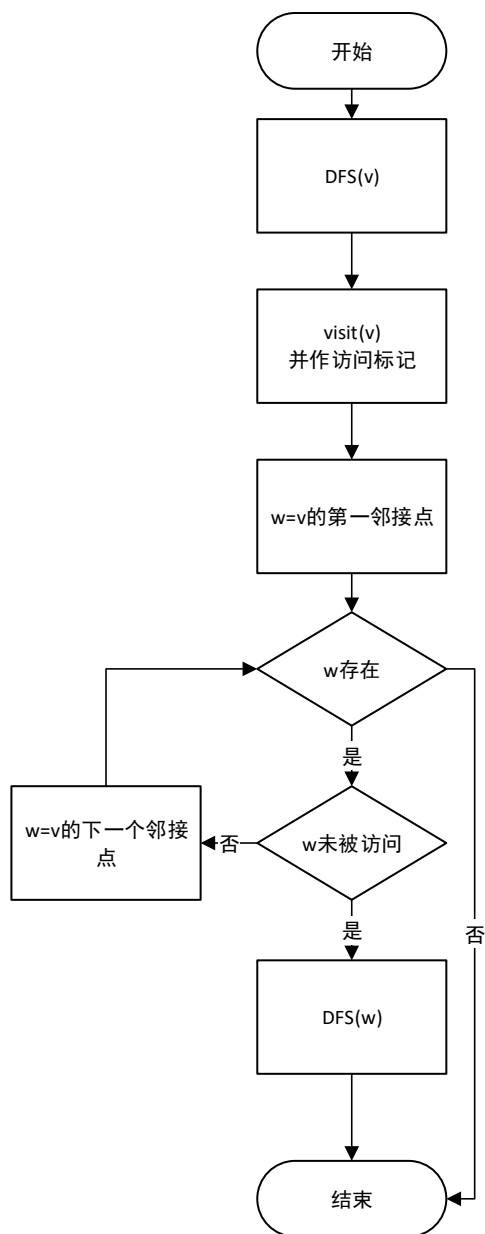


图 4-15 DFSTraverse ()流程图

(13) BFSTraverse(G,visit())

初始条件：图 G 存在。

操作结果：对图 G 进行广度优先搜索遍历，依次对图中的每一个顶点使用函数 visit 访问一次，且仅访问一次。

算法思想：：（1）访问出发顶点 v、作访问标记、v 入栈；（2）当栈非空时，分下列情况处理，直到栈空为止：如果栈顶之顶点存在未访问过的邻接顶点 w，则访问顶点 w、作访问标记、w 入栈，否则出栈。

时间复杂度: $O(e+v)$

流程图:

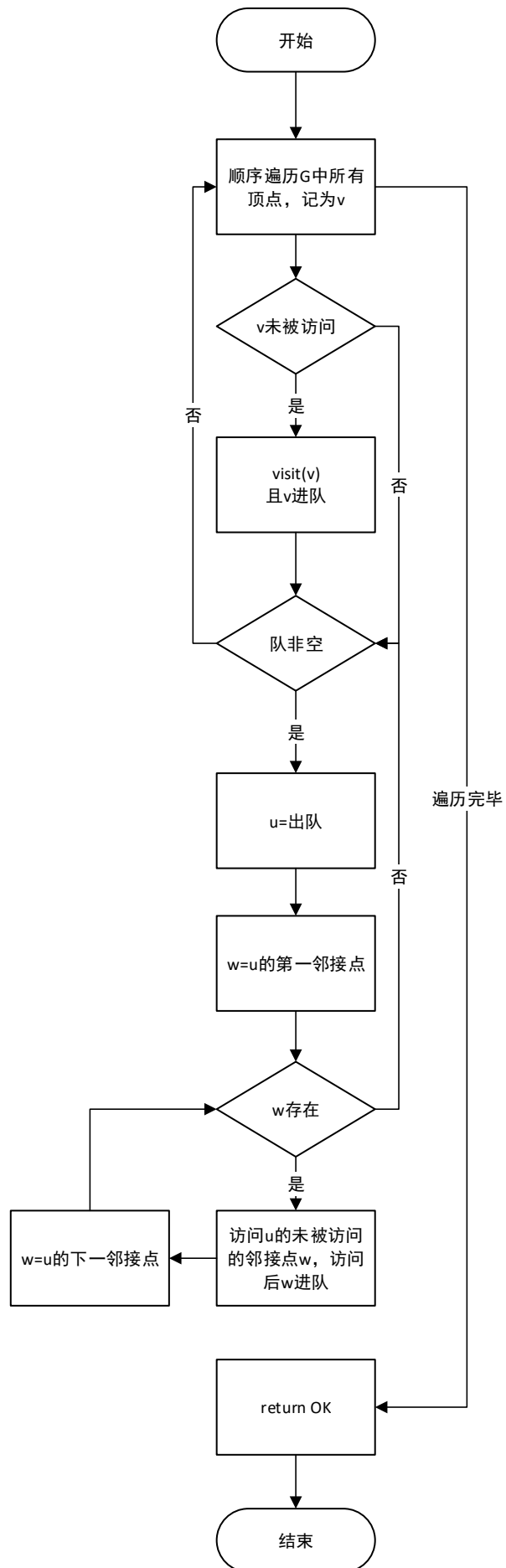


图 4-16 BFSTraverse ()流程图

(14) SaveData(G)

初始条件：图 G 存在。

操作结果：将 G 的数据写入到文件中。

算法思想：1.fwrite 写入顶点信息 2.fprintf 写入边信息

时间复杂度： $O(e+v)$

流程图：

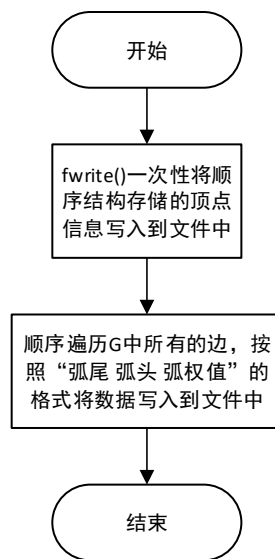


图 4-17 SaveData ()流程图

(15) LoadData(&G)

初始条件：图 G 不存在。

操作结果：从文件中读取数据对 G 进行初始化。

算法思想：1.fread 加载顶点信息 2.fscanf 加载边信息

时间复杂度： $O(e+v)$

流程图：

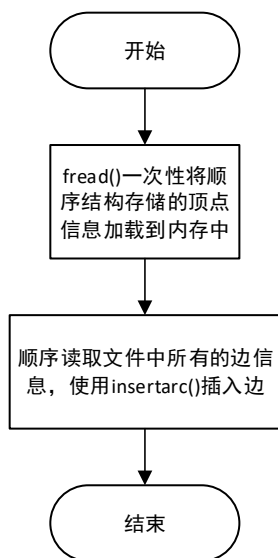
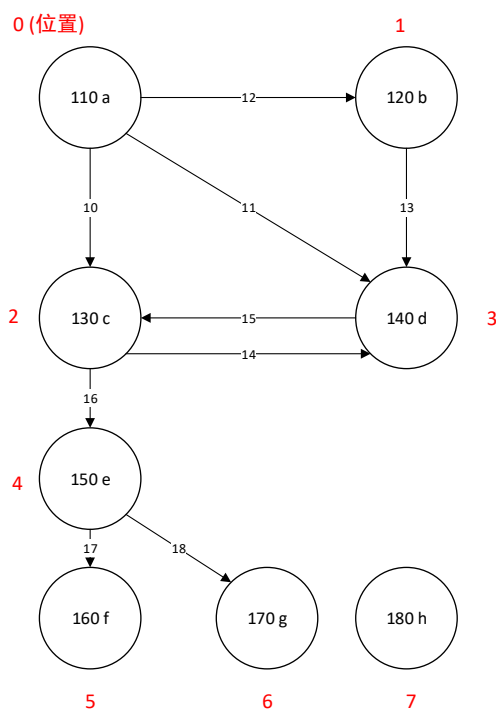


图 4-18 LoadData ()流程图

4.3 系统实现

说明：以下的测试数据基于图 4-19



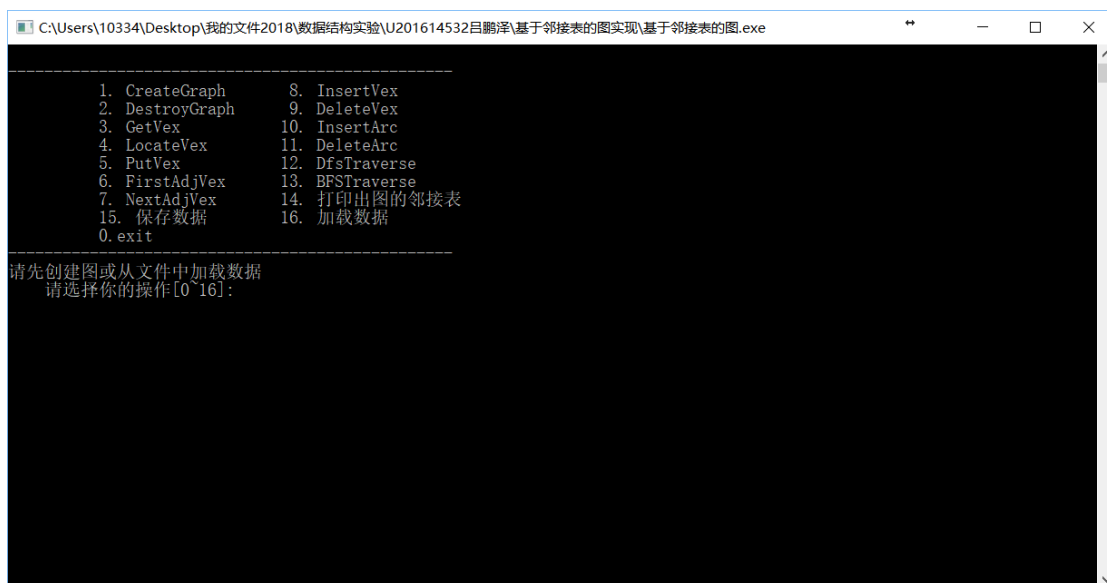
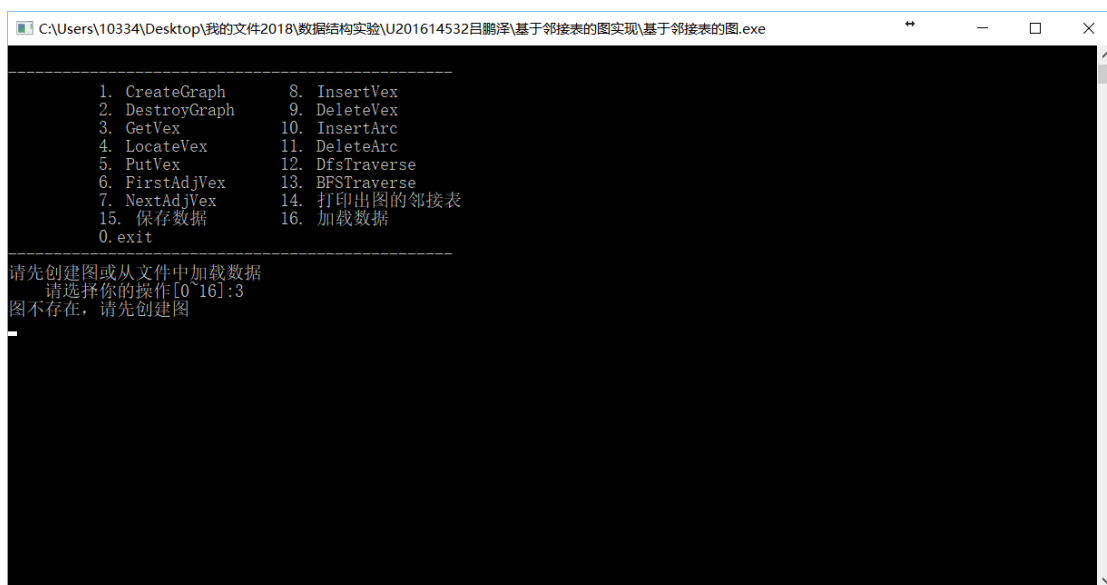


图 4-20 系统载入界面

2.创建图

如图 a，在图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，图创建成功。



a

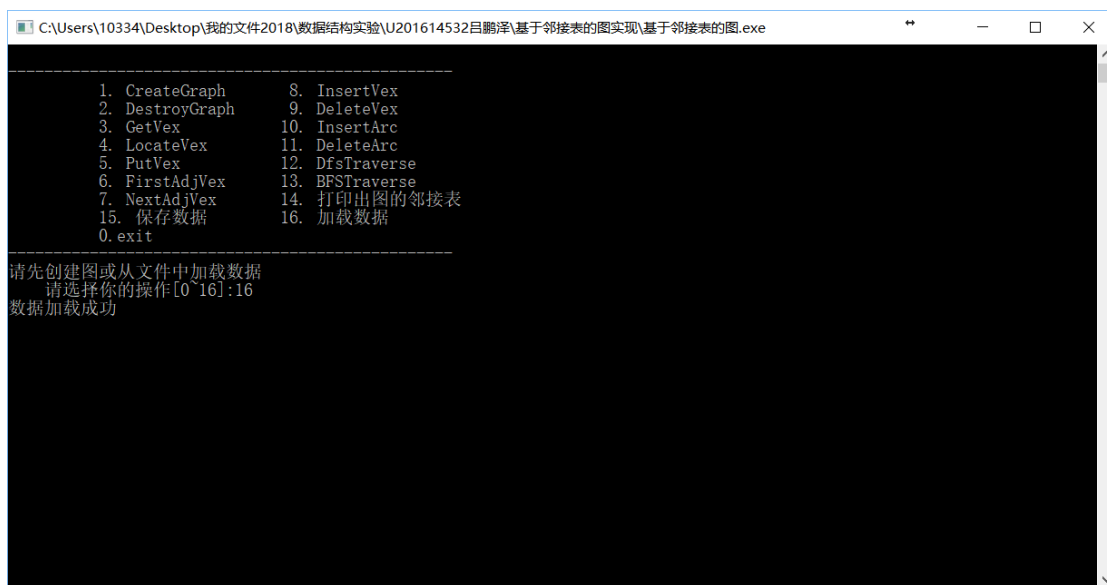


b

图 4-21 创建图操作演示

3.从文件中加载数据及遍历操作的演示

如图 a, 从文件中加载数据。如图 b, 深度优先遍历结果。如图 c, 广度优先遍历结果



a

```

C:\Users\10334\Desktop\我的文件2018\数据结构实验\U201614532吕鹏泽\基于邻接表的图实现\基于邻接表的图.exe

-----
1. CreateGraph      8. InsertVex
2. DestroyGraph    9. DeleteVex
3. GetVex          10. InsertArc
4. LocateVex       11. DeleteArc
5. PutVex          12. DfsTraverse
6. FirstAdjVex     13. BFsTraverse
7. NextAdjVex      14. 打印出图的邻接表
15. 保存数据       16. 加载数据
0. exit
-----
请选择你的操作[0~16]:12
110 a
130 c
150 e
160 f
170 g
140 d
120 b
180 h

```

b

```

C:\Users\10334\Desktop\我的文件2018\数据结构实验\U201614532吕鹏泽\基于邻接表的图实现\基于邻接表的图.exe

-----
1. CreateGraph      8. InsertVex
2. DestroyGraph    9. DeleteVex
3. GetVex          10. InsertArc
4. LocateVex       11. DeleteArc
5. PutVex          12. DfsTraverse
6. FirstAdjVex     13. BFsTraverse
7. NextAdjVex      14. 打印出图的邻接表
15. 保存数据       16. 加载数据
0. exit
-----
请选择你的操作[0~16]:13
110 a
130 c
140 d
120 b
150 e
160 f
170 g
180 h

```

c

图 4-22 遍历的演示

4.销毁图

如图 4-23，为销毁图的执行结果

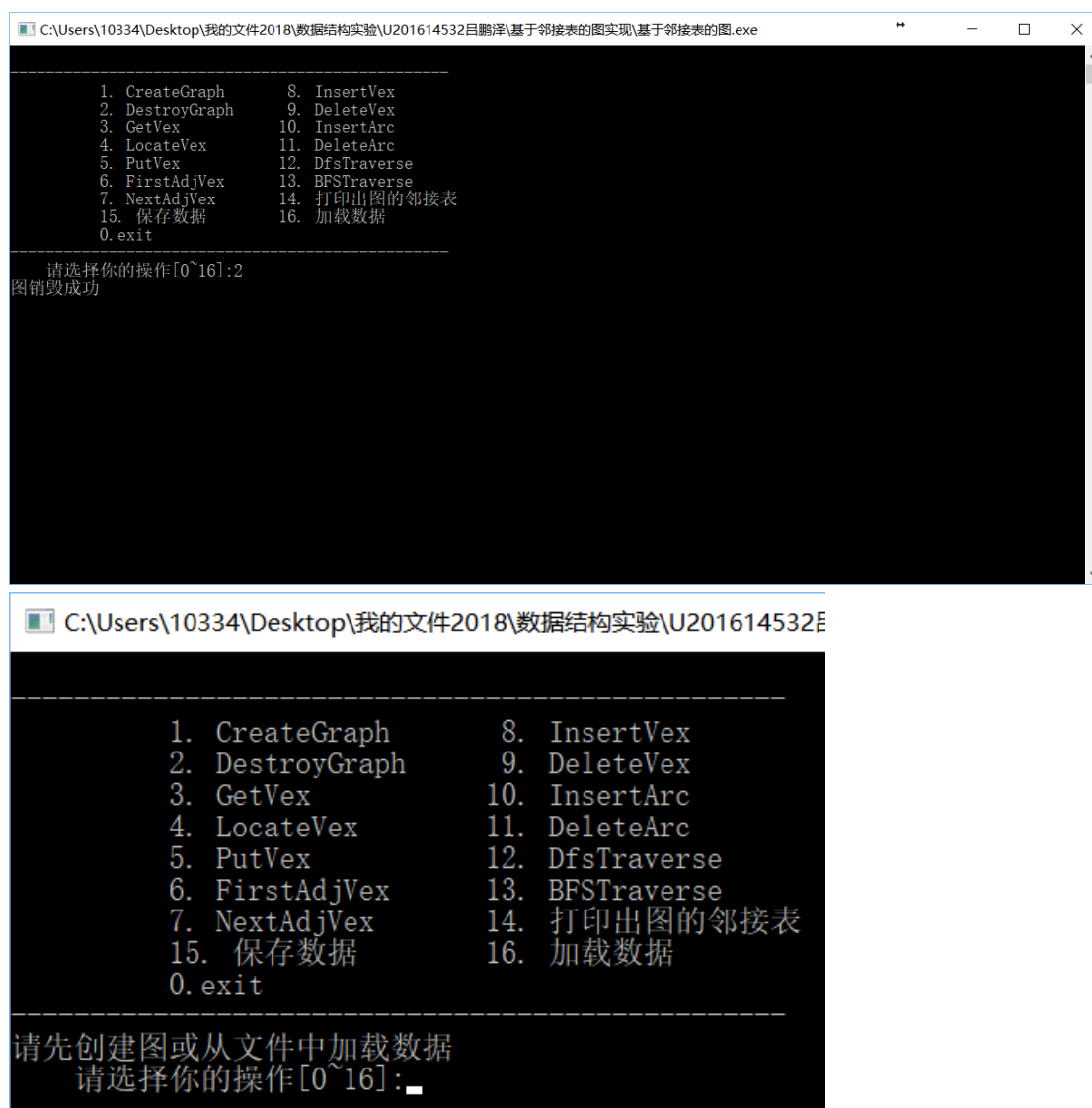
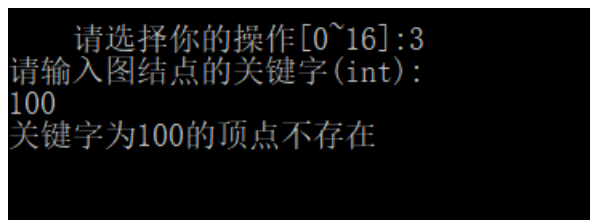


图 4-23 销毁图演示

5. 获取顶点

如图 a，顶点不在图中。如图 b，顶点在图中。



a

```

请选择你的操作[0~16]:3
请输入图结点的关键字(int):
120
关键字: 120    值: b
    
```

b

图 4-24 获取顶点

6.获取顶点位置

如图 a，顶点不在图中。如图 b，顶点在图中。

```

请选择你的操作[0~16]:4
请输入图结点的关键字(int):
1
关键字为1的顶点不存在
    
```

a

```

请选择你的操作[0~16]:4
请输入图结点的关键字(int):
110
关键字为110的顶点的位置为0
    
```

b

图 4-25 获取顶点位置

7.获取第一邻接点

如图 a，顶点存在第一邻接点。如图 b，顶点不存在第一邻接点。如图 c，顶点不在 G 中。

```

请选择你的操作[0~16]:6
请输入图顶点点的关键字(int):
110
关键字为110顶点的第一邻接顶点位置2 关键字130 值c
_
    
```

a

```

请选择你的操作[0~16]:6
请输入图顶点点的关键字(int):
180
关键字为180顶点的第一邻接顶点不存在
    
```

b

```

请选择你的操作[0~16]:6
请输入图顶点点的关键字(int):
200
关键字为200的顶点不存在
    
```

c

图 4-26 获取第一邻接点

8.获取下一邻接点

如图 4-27，将获取 110 的关于 140 的下一邻接点。

```

请选择你的操作[0~16]:7
请输入图结点的关键字(int):
110
请输入关键字为110结点的邻接结点关键字:
140
110结点相对于140结点的下一个邻接结点位置:1 关键字:120 值:b_
    
```

图 4-27 获取下一邻接点

9.插入顶点

如图 a，插入顶点成功。如图 b，插入后的结果。

```

请选择你的操作[0~16]:8
请输入新结点的关键字(int) 值(char)
1111 e
新节点1111 e插入成功
    
```

a

请选择你的操作[0~16]:14									
顶点数		边数							
9		9							
位置	关键字	值	邻接边						
0	110	a	->邻接点位置2	权10	->	邻接点位置3	权11	->	邻接点位置1
1	120	b	->邻接点位置3	权13	->	NULL			
2	130	c	->邻接点位置4	权16	->	邻接点位置3	权14	->	NULL
3	140	d	->邻接点位置2	权15	->	NULL			
4	150	e	->邻接点位置5	权17	->	邻接点位置6	权18	->	NULL
5	160	f	->NULL						
6	170	g	->NULL						
7	180	h	->NULL						
8	1111	e	->NULL						

b

图 4-28 插入顶点

10.删除顶点

如图 a，删除顶点成功。如图 b，删除后的结果。

```

请选择你的操作[0~16]:9
请输入要删除的图结点的关键字(int):
110
结点110删除成功

```

a

```

请选择你的操作[0~16]:14
顶点数 边数
7 6
位置 关键字 值 邻接边
-----
0| 120 b|->邻接点位置2 权13 -> NULL
-----
1| 130 c|->邻接点位置3 权16 -> 邻接点位置2 权14 -> NULL
-----
2| 140 d|->邻接点位置1 权15 -> NULL
-----
3| 150 e|->邻接点位置4 权17 -> 邻接点位置5 权18 -> NULL
-----
4| 160 f|->NULL
-----
5| 170 g|->NULL
-----
6| 180 h|->NULL
-----

```

b

图 4-29 删除顶点

11.插入边

如图 a，插入边成功。如图 b，插入边后的结果。

```

请选择你的操作[0~16]:10
请分别输入弧尾结点和弧头结点关键字
110 180
请输入弧的权值
33
弧(110, 180)插入成功

```

a

```

请选择你的操作[0~16]:14
顶点数 边数
8 10
位置 关键字 值 邻接边
-----
0| 110 a|->邻接点位置7 权33 -> 邻接点位置2 权10 -> 邻接点位置3 权11 -> 邻接点位置1 权12 -> NULL
-----
1| 120 b|->邻接点位置3 权13 -> NULL
-----
2| 130 c|->邻接点位置4 权16 -> 邻接点位置3 权14 -> NULL
-----
3| 140 d|->邻接点位置2 权15 -> NULL
-----
4| 150 e|->邻接点位置5 权17 -> 邻接点位置6 权18 -> NULL
-----
5| 160 f|->NULL
-----
6| 170 g|->NULL
-----
7| 180 h|->NULL
-----

```

b

图 4-30 插入边

13.删除边

如图 a，删除边成功。如图 b，删除边后的结果。如图 c，d，边不存在的两种结果

```

请选择你的操作[0~16]:11
请分别输入弧尾结点和弧头结点关键字
110 140
弧(110,140)删除成功
    
```

a

请选择你的操作[0~16]:14									
顶点数		边数							
8		8							
位置	关键字	值	邻接边						
0	110	a	->邻接点位置2	权10	->	邻接点位置1	权12	->	NULL
1	120	b	->邻接点位置3	权13	->	NULL			
2	130	c	->邻接点位置4	权16	->	邻接点位置3	权14	->	NULL
3	140	d	->邻接点位置2	权15	->	NULL			
4	150	e	->邻接点位置5	权17	->	邻接点位置6	权18	->	NULL
5	160	f	->NULL						
6	170	g	->NULL						
7	180	h	->NULL						

b

```

请选择你的操作[0~16]:11
请分别输入弧尾结点和弧头结点关键字
111 120
图中不存在顶点111
    
```

c

```

请选择你的操作[0~16]:11
请分别输入弧尾结点和弧头结点关键字
110 160
图中无弧(110,160)
    
```

d

图 4-31 删除边

4.4 实验小结

本次实验相比于二叉树来说稍微容易了一些,但在调试过程中遇到很多问题,在定义变量的时候结构体较多,在使用的时候搞混了,另外在引用时对于全局变量还是局部变量要区分一下。在编写函数时,一定要注意函数声明和函数定义的一致性,在调试过程中,我的一个函数的定义使用了引用符,而在声明中没有,导致了我的程序无法编译,我花费了许多时间才发现这一错误。

参考文献

- [1] 严蔚敏等. 数据结构(C 语言版). 清华大学出版社
- [2] Larry Nyhoff. ADTs, Data Structures, and Problem Solving with C++. Second Edition, Calvin College, 2005
- [3] 殷立峰. Qt C++ 跨平台图形界面程序设计基础. 清华大学出版社,2014:192~197
- [4] 严蔚敏等.数据结构题集(C 语言版). 清华大学出版社

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. InitiaList      7. LocateElem\n");
printf("      2. DestroyList     8. PriorElem\n");
printf("      3. ClearList         9. NextElem \n");
printf("      4. ListEmpty        10. ListInsert\n");
printf("      5. ListLength       11. ListDelete\n");
printf("      6. GetElem          12. ListTraverse\n");
printf("      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();
    }

```

序为 0
//输出第一个与 e 满足 compare 关系的元素位序，不存在位

```

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 (!ListTraverse(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 的数据元素的位置
    }
    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 ListTraverse(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 Elemtyp;

typedef struct Lnode

{

    Elemtyp data;

    Lnode * next;

}Lnode; //数据元素类型定义


typedef Lnode* SqList; //头指针
    
```

```

status InitiaList(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 ListTraverse(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. InitiaList      7. LocateElem\n");
printf("      2. DestroyList    8. PriorElem\n");
printf("      3. ClearList      9. NextElem \n");
printf("      4. ListEmpty      10. ListInsert\n");
printf("      5. ListLength     11. ListDelete\n");
printf("      6. GetElem        12. ListTraverse\n");
printf("      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 (InitiaList(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
```

```
    {
```

```
        Elemtyp 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
        {
            Elemtyp 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
        {
            Elemtyp 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 (!ListTraverse(*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 InitiaList(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 ListTraverse(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(SqlList &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

```

/*对于弧  $a \rightarrow b$ ，称  $a$  为弧尾， $b$  为弧头*/

#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 INFESTABLE -1

#define OVERFLOW -2


#define MAX_VERTEX_NUM 200//最大顶点数

#define MAX_ARC_NUM 2000//最大弧数

#define V_DATA "VData"//顶点数据文件

#define E_DATA "EData"//弧数据文件

typedef int status;

typedef int KEY;

typedef struct VertexType

{

    int key;

    char c;

}VertexType;

typedef int InfoType;


typedef struct ArcNode {

    int adjves;//该弧所指顶点的位置

```

```

    struct ArcNode *nextarc;//指向下一条弧的指针

    InfoType info;//弧的权值
}ArcNode;

typedef struct VNode {
    VertexType data;//顶点信息
    ArcNode *firstarc;//指向第一条依附该顶点的弧的指针
}VNode, AdjList[MAX_VERTEX_NUM];

typedef struct {
    AdjList vertices;
    int vexnum, arcnum;//点数、边数
}ALGraph;
typedef struct {
    KEY tail;//弧尾
    KEY head;//弧头
    InfoType info;//弧权值
}VertexRation;//图的关系集

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

status CreateGraph(ALGraph &G, VertexType * V, VertexRation * VR);
status DestroyGraph(ALGraph &G);
VNode * GetVex(ALGraph &G, KEY v);
status PutVex(ALGraph &G, KEY v, VertexType value);//将结点 v 的值赋为
value
status LocateVex(ALGraph &G, KEY v);
int FirstAdjVex(ALGraph &G, int v);//第 v 个顶点的第一邻接点
int NextAdjVex(ALGraph &G, int v, int w);//第 v 个顶点相对于第 w 顶点的下
一邻接点

```

```

status InsertVex(ALGraph &G, VertexType v);
status DeleteVex(ALGraph &G, KEY v);
status InsertArc(ALGraph &G, KEY v, KEY w, InfoType i);
status DeleteArc(ALGraph &G, KEY v, KEY w);
status DFSTraverse(ALGraph G, status(*visit)(VNode v));
void DFS(ALGraph G, int v, status(*visit)(VNode v));//从第 v 个顶点出发递归
地深度优先遍历图 G

status BFSTraverse(ALGraph G, status(*visit)(VNode v));
status visit(VNode v)
{
    printf("%d %c\n", v.data.key, v.data.c);
    return OK;
}
void SaveData(ALGraph G);
void LoadData(ALGraph &G);
void printG(ALGraph G)
{
    int i;
    ArcNode *p;
    printf("顶点数 边数\n");
    printf("%6d %4d\n", G.vexnum, G.arcnum);
    printf("位置 关键字 值 邻接边\n");
    printf("-----\n");
    for (i = 0; i < G.vexnum; i++)
    {
        printf("%4d|%6d %2c|->", i, G.vertices[i].data.key, G.vertices[i].data.c);
        for (p = G.vertices[i].firstarc; p; p = p->nextarc)
            printf("邻接点位置%-4d 权%-4d-> ", p->adjves, p->info);
    }
}

```

```

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

bool visited[MAX_VERTEX_NUM]; // 标记已被访问的结点
/*-----*/

void main(void) {
    ALGraph G;
    int op = 1;
    int exist = 0;
    while (op) {
        system("cls");
        printf("\n");
        printf("-----\n");
        printf("      1. CreateGraph      8. InsertVex\n");
        printf("      2. DestroyGraph     9. DeleteVex\n");
        printf("      3. GetVex           10. InsertArc\n");
        printf("      4. LocateVex        11. DeleteArc\n");
        printf("      5. PutVex           12. DfsTraverse\n");
        printf("      6. FirstAdjVex      13. BFSTraverse\n");
        printf("      7. NextAdjVex       14. 打印出图的邻接表\n");
        printf("      15. 保存数据        16. 加载数据\n");
        printf("      0.exit\n");
        printf("-----\n");
        if (exist == 0)
            printf("请先创建图或从文件中加载数据\n");
        printf("    请选择你的操作[0~16]:");
        if (exist == 1 && (G.arcnum >= MAX_ARC_NUM || G.vexnum >=

```

```

MAX_VERTEX_NUM))
    {
        printf("空间不足\n");
        exit(0);
    } //空间不足，异常退出
    scanf("%d", &op);
    if (exist == 0 && op != 1 && op != 16) //图不存在，需先执行操作 1
    {
        printf("图不存在，请先创建图\n");
        getchar(); getchar();
        continue;
    }
    else if (exist == 1 && (op == 1 || op == 16)) //图存在，并且选择了操作
1 或操作 16
    {
        printf("图已存在,请先销毁后再创建\n");
        getchar(); getchar();
        continue;
    }
    switch (op) {
    case 1:
    {
        int i = 0;
        VertexType V[MAX_VERTEX_NUM]; //图顶点
        printf("请顺序输入图顶点的关键字(int) 数据(char),以 0 #结束输入\n");

        while (1)
        {
            scanf("%d %c", &V[i].key, &V[i].c);

```



```

        if (V[i].key == 0 && V[i].c == '#')
            break;
        i++;
    }
    printf("图顶点信息录入完毕\n");
    i = 0;
    VertexRalation VR[MAX_ARC_NUM]; //顶点关系
    printf("请顺序输入每条弧的弧尾关键字(int) 弧头关键字(int) 权
值(int),以空格隔开,以 0 0 0 结束输入\n");

    while (1)
    {
        scanf("%d %d %d", &VR[i].tail, &VR[i].head, &VR[i].info);
        if (VR[i].tail == 0 && VR[i].head == 0 && VR[i].info == 0)
            break;
        i++;
    }
    printf("图顶点关系信息录入完毕\n");
    if (CreateGraph(G, V, VR))
        printf("图创建成功\n");
    else printf("图创建失败\n");
    exist = 1;
    getchar(); getchar();
    break;
}

case 2:
{
    DestroyGraph(G);
    printf("图销毁成功\n");
    exist = 0;
}

```

```
        getchar(); getchar();
        break;
    }
case 3:
{
    KEY key;
    printf("请输入图结点的关键字(int):\n");
    scanf("%d", &key);
    VNode *pv;
    pv = GetVex(G, key);
    if (pv == NULL)
        printf("关键字为%d 的顶点不存在\n", key);
    else
    {
        printf("关键字: %d    值: %c", pv->data.key, pv->data.c);
    }
    getchar(); getchar();
    break;
}
case 4:
{
    KEY key;
    printf("请输入图结点的关键字(int):\n");
    scanf("%d", &key);
    int locate;
    locate = LocateVex(G, key);
    if (locate == -1)
        printf("关键字为%d 的顶点不存在\n", key);
    else
```

```

    {
        printf("关键字为%d 的顶点的位置为%d\n", key, locate);
    }
    getchar(); getchar();
    break;
}
case 5:
{
    KEY key;
    printf("请输入图结点的关键字(int):\n");
    scanf("%d", &key);
    VNode *pv;
    pv = GetVex(G, key);
    if (pv == NULL)
    {
        printf("关键字为%d 的顶点不存在\n", key);
        getchar(); getchar();
        break;
    }
    VertexType value;
    printf("请输入更改后图结点关键字(int) 值(char)\n");
    scanf("%d %c", &value.key, &value.c);
    printf("更改前: %d %c\n", pv->data.key, pv->data.c);
    PutVex(G, key, value);
    printf("更改后: %d %c\n", pv->data.key, pv->data.c);
    getchar(); getchar();
    break;
}
case 6:

```

```

{
    KEY key;
    int locate;
    printf("请输入图顶点的关键字(int):\n");
    scanf("%d", &key);
    locate = LocateVex(G, key); // 顶点 v 的位置
    if (locate == -1)
        printf("关键字为%d 的顶点不存在\n", key);
    else
    {
        locate = FirstAdjVex(G, locate); // 顶点 v 的第一邻接点位置
        if (locate > -1)
            printf("关键字为%d 顶点的第一邻接顶点位置%d 关键字%d 值%c\n", key, locate, G.vertices[locate].data.key, G.vertices[locate].data.c);
        else printf("关键字为%d 顶点的第一邻接顶点不存在\n", key);
    }
    getchar(); getchar();
    break;
}

case 7:
{
    KEY v;
    printf("请输入图结点的关键字(int):\n");
    scanf("%d", &v);
    if (GetVex(G, v) == NULL)
        printf("关键字为%d 的顶点不存在\n", v);
    else
    {
        KEY w;

```

```

printf("请输入关键字为%d 结点的邻接结点关键字:\n", v);
scanf("%d", &w);
int vlocate, wlocate, Adj;
vlocate = LocateVex(G, v);
wlocate = LocateVex(G, w);
Adj = NextAdjVex(G, vlocate, wlocate);
if (Adj>-1)
    printf("%d 结点相对于%d 结点的下一个邻接结点位置 :%d 关键字 :%d 值 :%c", v, w, Adj, G.vertices[Adj].data.key, G.vertices[Adj].data.c);
else
    printf("%d 结点相对于%d 结点的下一个邻接结点不存在\n", v, w);

    getchar(); getchar();
    break;
}
}
case 8:
{
    VertexType v;
    printf("请输入新结点的关键字(int) 值(char)\n");
    scanf("%d %c", &v.key, &v.c);
    InsertVex(G, v);
    VNode *pv;
    pv = GetVex(G, v.key);
    printf("新节点%d %c 插入成功\n", pv->data.key, pv->data.c);
    getchar(); getchar();
    break;
}
}

```

case 9:

```
{
    KEY key;
    printf("请输入要删除的图结点的关键字(int):\n");
    scanf("%d", &key);
    if (GetVex(G, key) == NULL)
        printf("关键字为%d 的顶点不存在\n", key);
    else
    {
        DeleteVex(G, key);
        printf("结点%d 删除成功\n", key);
    }
    getchar(); getchar();
    break;
}
```

case 10:

```
{
    KEY v, w;
    printf("请分别输入弧尾结点和弧头结点关键字\n");
    scanf("%d %d", &v, &w);
    if (GetVex(G, v) == NULL)
        printf("图中不存在顶点%d\n", v);
    else if (GetVex(G, w) == NULL)
        printf("图中不存在顶点%d\n", w);
    else
    {
        InfoType i;
        printf("请输入弧的权值\n");
    }
}
```

```

        scanf("%d", &i);
        InsertArc(G, v, w, i);
        printf("弧(%d,%d)插入成功\n", v, w);
    }
    getchar(); getchar();
    break;
}
case 11:
{
    KEY v, w;
    printf("请分别输入弧尾结点和弧头结点关键字\n");
    scanf("%d %d", &v, &w);
    if (GetVex(G, v) == NULL)
        printf("图中不存在顶点%d\n", v);
    else if (GetVex(G, w) == NULL)
        printf("图中不存在顶点%d\n", w);
    else
    {
        if (DeleteArc(G, v, w))
            printf("弧(%d,%d)删除成功\n", v, w);
        else printf("图中无弧(%d,%d)\n", v, w);
    }
    getchar(); getchar();
    break;
}
case 12:
{
    DFSTraverse(G, visit);
    getchar(); getchar();

```

```
        break;
    }
case 13:
{
    BFSTraverse(G, visit);
    getchar(); getchar();
    break;
}
case 14:
{
    printG(G);
    getchar(); getchar();
    break;
}
case 15:
{
    SaveData(G);
    printf("数据保存成功\n");
    getchar(); getchar();
    break;
}
case 16:
{
    LoadData(G);
    exist = 1;
    printf("数据加载成功\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()

status CreateGraph(ALGraph &G, VertexType * V, VertexRelation * VR)
{
    /*创建顶点*/
    G.arcnum = G.vexnum = 0;
    int i;
    ArcNode *pe, *temp;
    VNode *pv; //顶点结点指针
    for (i = 0; V[i].key != 0 || V[i].c != '#'; i++)
    {
        G.vertices[i].data = V[i]; //创建一个顶点
        G.vertices[i].firstarc = NULL; //顶点邻接边置空
        G.vexnum++; //顶点数+1
    }
    /*创建边*/
    for (i = 0; VR[i].head || VR[i].tail || VR[i].info; i++)
    {
        pv = GetVex(G, VR[i].tail); //获取弧尾结点的地址
        temp = (ArcNode *)malloc(sizeof(ArcNode)); //创建一个边结点
        temp->adjves = LocateVex(G, VR[i].head);
        temp->info = VR[i].info; //录入边结点信息
        temp->nextarc = pv->firstarc;
    }
}

```

```

        pv->firstarc = temp;//将新的边结点放在边链的第一个结点
        G.arcnum++;//边数+1
    }
    return OK;
}

status DestroyGraph(ALGraph &G)
{
    int i;
    ArcNode *pe1, *pe2;
    for (i = 0; i < G.vexnum; i++)
    {
        pe2 = pe1 = G.vertices[i].firstarc;
        while (pe1)
        {
            pe1 = pe1->nextarc;
            free(pe2);
            pe2 = pe1;
        }//释放边结点
    }
    return OK;
}

VNode * GetVex(ALGraph &G, KEY v)
{
    VNode *p;
    for (p = G.vertices; p < G.vertices + G.vexnum; p++)
    {
        if (p->data.key == v)
            return p;
    }
}

```

```

        return NULL;
    }

    status PutVex(ALGraph &G, KEY v, VertexType value)
    {
        VNode *pv;
        pv = GetVex(G, v); // 获取结点
        pv->data = value; // 结点赋值
        return OK;
    }

    status LocateVex(ALGraph &G, KEY v)
    {
        int i;
        for (i = 0; i < G.vexnum; i++)
            if (G.vertices[i].data.key == v)
                return i; // 返回顶点的位置
        return -1; // 顶点不存在返回-1
    }

    int FirstAdjVex(ALGraph &G, int v)
    {
        VNode *pv;
        pv = &G.vertices[v];
        if (pv->firstarc == NULL) // 若 v 无邻接边
            return -1;
        else return pv->firstarc->adjves; // 返回 v 的第一邻接点位置
    }

    int NextAdjVex(ALGraph &G, int v, int w)
    {
        VNode *pv;
        pv = &G.vertices[v];
    }

```

```

    ArcNode *pe;
    for (pe = pv->firstarc; pe; pe = pe->nextarc)
    {
        if (pe->adjves == w)
        {
            if (pe->nextarc != NULL)//若 w 不是 v 的最后一个邻接点
                return pe->nextarc->adjves;
            else return -1;//w 是 v 的最后一个邻接点
        }
    }
    return -1;//w 不是 v 的邻接点
}

status InsertVex(ALGraph &G, VertexType v)
{
    G.vertices[G.vexnum].data = v;//赋值结点数据
    G.vertices[G.vexnum].firstarc = NULL;//新节点第一邻接边置空
    G.vexnum++;
    return OK;
}

status DeleteVex(ALGraph &G, KEY v)
{
    int locate = LocateVex(G, v);//定位顶点 v
    int i;
    VNode *pv;
    ArcNode *temp, *pe, *pre;//pe 遍历邻接边, pre 指向 pe 边结点的前驱结
点

    /*1.删除与顶点 v 关联的边*/

```

//1.1 删除以 v 为弧尾的边结点

pv = &G.vertices[locate];//指向顶点 v

pre = pe = pv->firstarc;

while (pe)

{

 pe = pe->nextarc;

 free(pre);

 pre = pe;

 G.arcnum--;

}

pv->firstarc = NULL;

//1.2 遍历 G 中所有顶点，删除以 v 为弧头的边结点

for (i = 0; i < G.vexnum; i++)

{

 DeleteArc(G, G.vertices[i].data.key, v);

}

/*2.删除顶点 v*/

for (i = locate; i < G.vexnum - 1; i++)

{

 G.vertices[i] = G.vertices[i + 1];

}

G.vexnum--;

/*3.更改 G 中边结点的邻接点的位置信息*/

for (i = 0; i < G.vexnum; i++)

{

 pv = &G.vertices[i];

 for (pe = pv->firstarc; pe; pe = pe->nextarc)

 {

```

        if (pe->adjves > locate)//v 结点之后的结点位置都前移 1
            pe->adjves--;
    }
}
return OK;
}

status InsertArc(ALGraph &G, KEY v, KEY w, InfoType i)
{
    int locate = LocateVex(G, v);
    ArcNode *pe;
    pe = (ArcNode *)malloc(sizeof(ArcNode));//分配一个边结点
                                           //初始化边结点

    pe->adjves = LocateVex(G, w);
    pe->info = i;
    //将新的边结点插入到边链的头部位置
    pe->nextarc = G.vertices[locate].firstarc;
    G.vertices[locate].firstarc = pe;
    G.arcnum++;
    return OK;
}

status DeleteArc(ALGraph &G, KEY v, KEY w)
{
    VNode *pv;
    ArcNode *pe, *temp, *pre;
    pv = GetVex(G, v);
    if (pv->firstarc == NULL)//v 无邻接点
        return FALSE;
    int wlocate = LocateVex(G, w);//w 的位置
    if (pv->firstarc->adjves == wlocate)//如果 w 是 v 的第一邻接点

```

```

    {
        temp = pv->firstarc;
        pv->firstarc = pv->firstarc->nextarc;
        free(temp);
        G.arcnum--;
        return TRUE;
    }
else//在 v 的非第一邻接结点中查找 w
{
    for (pre = pe = pv->firstarc; pe; pre = pe, pe = pe->nextarc)
    {
        if (pe->adjves == wlocate)//找到
        {
            pre->nextarc = pe->nextarc;//修改指针
            free(pe);//释放结点
            G.arcnum--;
            return TRUE;
        }
    }
    return FALSE;//弧(v,w)不存在
}

status DFSTraverse(ALGraph G, status(*visit)(VNode v))
{
    int v;
    for (v = 0; v < MAX_VERTEX_NUM; v++)//标记置空
        visited[v] = 0;
    for (v = 0; v < G.vexnum; v++)
        if (!visited[v]) DFS(G, v, visit);//对未访问的顶点调用 DFS
}

```

```

        return OK;
    }

void DFS(ALGraph G, int v, status(*visit)(VNode v))
{
    int w;
    visited[v] = TRUE;
    visit(G.vertices[v]); //访问第 v 个顶点
    for (w = FirstAdjVex(G, v); w >= 0; w = NextAdjVex(G, v, w))
        if (!visited[w]) DFS(G, w, visit); //对 v 的尚未访问的邻接顶点 w 递归
        调用 DFS
    }

status BFSTraverse(ALGraph G, status(*visit)(VNode v))
{
    int head, tail;
    int v, u, w;
    for (v = 0; v < MAX_VERTEX_NUM; v++) //标记置空
        visited[v] = 0;
    int queue[MAX_VERTEX_NUM];
    head = tail = 0;
    for (v = 0; v < G.vexnum; v++) //顺序遍历 G 中所有顶点
    {
        if (!visited[v]) //v 未被访问
        {
            visited[v] = TRUE; visit(G.vertices[v]); //访问 v
            queue[tail] = v; //v 入队
            tail = (tail + 1) % MAX_VERTEX_NUM;
            while (tail != head && (tail + 1) % MAX_VERTEX_NUM != head) //
                队非空
            {

```



```

        u = queue[head]; //出队
        head = (head + 1) % MAX_VERTEX_NUM;
        for (w = FirstAdjVex(G, u); w >= 0; w = NextAdjVex(G, u, w)) //u
的未被访问的邻接点依次进队
            if (!visited[w]) //u 的邻接点 w 未被访问
            {
                visited[w] = TRUE; //访问 w
                visit(G.vertices[w]);
                queue[tail] = w;
                tail = (tail + 1) % MAX_VERTEX_NUM; //w 进队
            }
        } //end while
    } //end if
} //end for
return OK;
}

void SaveData(ALGraph G)
{
    FILE *fp;
    /*储存顶点信息*/
    fp = fopen(V_DATA, "w");
    fwrite(&G, sizeof(ALGraph), 1, fp);
    fclose(fp);
    /*储存弧信息*/
    fp = fopen(E_DATA, "w");
    int i = 0;
    ArcNode *pe;
    for (i = 0; i < G.vexnum; i++)
    {

```

```

        for (pe = G.vertices[i].firstarc; pe; pe = pe->nextarc)
        {
            fprintf(fp, "%d %d %d\n", i, pe->adjves, pe->info); //按照弧尾位置
            //弧头位置 弧权值的格式将弧数据写入到文件中
        }
    }
    fclose(fp);
}

void LoadData(ALGraph &G)
{
    FILE *fp;
    int i;
    int tail, head, info; //弧尾, 弧头, 权值
    /*读取顶点信息*/
    fp = fopen(V_DATA, "r");
    fread(&G, sizeof(ALGraph), 1, fp);
    fclose(fp);
    for (i = 0; i < G.vexnum; i++)
        G.vertices[i].firstarc = NULL; //邻接边置空
    /*读取弧信息*/
    fp = fopen(E_DATA, "r");
    while (fscanf(fp, "%d %d %d", &tail, &head, &info) != EOF)
    { //使用修改的 InsertArc 插入弧
        ArcNode *pe;
        pe = (ArcNode *)malloc(sizeof(ArcNode)); //分配一个边结点
        //初始化边结点
        pe->adjves = head;
        pe->info = info;
        //将新的边结点插入到边链的头部位置
    }
}

```

```
    pe->nextarc = G.vertices[tail].firstarc;  
    G.vertices[tail].firstarc = pe;  
}  
fclose(fp);  
}
```

