

华中科技大学

课程实验报告

课程名称： 数据结构实验

专业班级 CS22XX

学 号 你的学号

姓 名 你的名字

指导教师 你的导师

报告日期 2023 年 5 月 25 日

目 录

1	基于链式存储结构的线性表实现.....	1
1.1	问题描述	1
1.2	系统设计	1
1.3	系统实现	2
1.4	系统测试	12
1.5	实验小结	16
2	基于二叉链表的二叉树实现	19
2.1	问题描述	19
2.2	系统设计	19
2.3	系统实现	21
2.4	系统测试	31
2.5	实验小结	40
3	课程的收获和建议	41
3.1	基于顺序存储结构的线性表实现	41
3.2	基于链式存储结构的线性表实现	41
3.3	基于二叉链表的二叉树实现	41
3.4	基于邻接表的图实现	42
4	参考文献	43
5	附录 A 基于顺序存储结构线性表实现的源程序	44
6	附录 B 基于链式存储结构线性表实现的源程序	68
7	附录 C 基于二叉链表二叉树实现的源程序	93
8	附录 D 基于邻接表图实现的源程序	128

1 基于链式存储结构的线性表实现

1.1 问题描述

本实验实现了线性表的链式存储，构造一个具有菜单功能的演示系统，实现了线性表的初始化、销毁、清空等基本功能和全部的 5 种附加功能，并实现了多线性表管理。

1.2 系统设计

链表作为一种线性结构，在数据结构中应用广泛，是最常用、最基本的结构类型之一。本次实验采用链表的线性存储方式，实现了链表基本的功能，比如增添、删除、遍历等。同时，还在此基础上添加了一些高级功能，比如翻转、倒序删除、排序等，更好地方便使用者对链表进行操作。该程序还使用语言文字提示，降低了程序的使用难度。

本实验涉及的头文件以及相关常量定义如下：

相关常量定义

```
1  /* 相关头文件 */
2  #include <stdio.h>
3  #include <malloc.h>
4  #include <stdlib.h>
5
6  /*-----相关的常量及定义 -----*/
7  #define TRUE 1//定义真值
8  #define FALSE 0//定制假值
9  #define OK 1//程序正常运行
10 #define ERROR 0//程序运行出错
11 #define INFEASTABLE -1 //没有实现的操作返回该标记
12 #define OVERFLOW -2 //数值溢出
13 #define MAX_NUM 10 //可管理线性表的数量
14 #define LIST_INIT_SIZE 100 //线性表的初始存储空间大小
15 #define LISTINCREMENT 10 //线性表存储空间不足时增加的存储空间量
16 FILE *fp; //文件指针，用于数据存储和读取
```

本系统的数据结构有两种：链式线性表和链式线性表的管理表。其具体定义如下。

相关数据结构定义

```
1 typedef int status; //定义所有状态码和返回值的类型为int
2 typedef int ElemType; //数据元素类型定义
3 typedef struct LNode{ //定义单链表节点结构体类型
4     ElemType data; //节点中存储的数据元素
5     struct LNode *next; //指向下一个节点的指针
6 }LNode, *LinkList;
```

系统的总体架构：界面上采用简易菜单演示系统，在 while 循环中建立菜单演示，op 代表用户选择的操作序号，程序将首先判断 op 的合法性，若合法则通过 switch 函数进入功能的选择，进入相关功能函数执行相关操作，操作完成后继续执行循环，直到用户输入 0 时，退出系统。

多线性表管理通过 ChooseList 函数实现，用户可以输入位序切换线性表，相关操作仅在当前线性表完成，而不会影响线性表组。

```
Menu for Linear Table On Sequence Structure
可在10个顺序表进行多表操作，初始化请先操作功能15,默认在第一个表上操作
-----
**          1. InitList      7. LocateElem      **
**          2. DestroyList  8. PriorElem       **
**          3. ClearList    9. NextElem        **
**          4. ListEmpty    10. ListInsert      **
**          5. ListLength   11. ListDelete     **
**          6. GetElem      12. ListTraverse   **
**          13. SaveList    14. LoadList      **
**          0.Exit          **
**          15.ChooseList(请先进行此选项以选择在哪个表上进行操作)
**          16.ReverseList  17. RemoveNthFromEnd
**          18.SortList    **
-----lbw-----
请选择你的操作[0--18]:
```

图 1-1 菜单演示系统

1.3 系统实现

本程序在 Windows 11 系统下采用 Dev-C++ 进行编译调试，语言选择 C 语言以下主要说明各个主要函数的实现思想，函数和系统实现的源代码放在附录中。

（本实验所有函数在实现功能之前会先对是否已有线性表进行判定，若无线性表，则返回 *INFEASIBLE*，在各函数具体设计思路中不再叙述此条。）

1. 初始化线性表

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

在设计链表结构体时，需要添加一个名为 `next` 的结构体指针，以实现链表结构体单元之间的关联。然而，该指针是初始化时未被赋值的，因此链表的初始化是一个重要步骤。为了给结构体指针赋值，需要使用 `malloc` 函数，它可以分配固定空间大小的地址。初始化链表时，我们需要创建一个头结点，这个头结点的 `next` 指针需要赋空值，以确保程序的稳定性。

复杂度：时间复杂度 $T(n) = O(1)$

2. 销毁线性表

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

在销毁链表时，不能直接清空头结点作为销毁，否则会导致程序内存泄漏，最终可能导致内存溢出和程序崩溃。正确的做法是使用 `free()` 函数释放每个结点的内存，并在清空所有结点后将头指针 `L` 赋值为空。这种做法能够及时释放占用的内存空间，使程序占用的空间保持稳定，有效提高程序的稳定性。清空过程可采用 `while` 循环遍历结点，并在出现空指针时结束循环，保证每个结点都得到了清空。

复杂度：时间复杂度 $T(n) = O(1)$

3. 清空线性表

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

该函数与销毁表函数唯一的区别在于是否考虑头结点。销毁表意味着整个链表都会彻底消失，而清空链表则是将链表中的所有数据清空，但链表本身仍然存在。因此，在清空链表之前必须确保链表的头结点存在。因此，该函数从头结点的下一个节点开始循环删除链表的所有节点，并将头结点的下一个节点赋值为空，以确保链表清空且仍然可用。

复杂度：时间复杂度 $T(n) = O(1)$

4. 判定线性表是否为空

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

简而言之，空表表示头结点存在但没有任何子节点。因此，当我们清空链表时，我们需要确定链表是否为空，这可以通过检查头结点的 `next` 指针是否为空来实现。如果链表为空，则不需要进行清空操作。否则，我们将从头结

点的下一个节点开始循环删除链表的所有节点，并将头结点的 `next` 指针赋值为空，以确保链表为空且仍然可用。值得注意的是，如果我们直接将头结点赋值为空，那么整个链表将无法访问并最终导致内存泄漏。

复杂度：时间复杂度 $T(n) = O(1)$

5. 求线性表的长度

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

判断链表的长度需要对链表进行一次遍历，每次遇到一个节点就将计数器加 1。当遍历到链表的末尾时，计数器的值就是链表的长度。

复杂度：时间复杂度 $T(n) = O(1)$

6. 获取元素

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

由于 `i` 的数值范围已设置，因此可以直接处理链表数据。引入一个 `int` 变量 `count`，用于记录当前遍历的链表位置，在 `count` 等于 `i` 时跳出循环。

为判断目标元素是否存在，我们设立了如下的判断条件：在循环结束后检查结构体指针是否为空值，若为空，则说明 `while` 循环遍历到链表尾部仍未找到目标元素，因此返回 `ERROR`。若不为空，则说明找到了目标元素，我们将目标元素的值赋给 `e`，并返回 `OK`，从而实现对该需求的满足。

复杂度：时间复杂度 $T(n) = O(1)$

7. 查找元素

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

如果线性表不存在，返回不可行。遍历线性表，记录遍历到的位置，如果遍历到的节点元素是要查找的位置，返回这个位置。如果遍历完成后依然未找到，返回不存在。

复杂度：时间复杂度 $T(n) = O(n)$

8. 获取前驱元素

函数名称是 `PriorElem (L,cur_e,pre_e)`；初始条件是线性表 `L` 已存在；操作结果是若 `cur_e` 是 `L` 的数据元素，且不是第一个，则用 `pre_e` 返回它的前驱，

否则操作失败，`pre_e` 无定义。

如果线性表不存在，返回不可行。从第一个元素开始遍历，如果遍历到的节点元素的 `next` 是要查找的元素，返回 `next` 所指元素。如果遍历完成后依然未找到，返回不存在。

复杂度：时间复杂度 $T(n) = O(n)$

9. 获取后继元素

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

如果线性表不存在，返回不可行。遍历线性表如果遍历到的节点元素是要查找的元素且元素的 `next` 存在，返回 `next` 所指元素。否则返回不存在。如果遍历完成后依然未找到，返回不存在。

复杂度：时间复杂度 $T(n) = O(n)$

10. 插入元素

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

如果线性表不存在，返回不可行。首先遍历链表寻找插入位置，如果索引不大于 0，返回索引错误，如果遍历完成后依然未找到，返回索引错误。否则分配空间并插入节点。

复杂度：时间复杂度 $T(n) = O(n)$

11. 删除元素

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

如果线性表不存在，返回不可行。首先遍历链表寻找删除位置，如果索引不大于 0，返回索引错误，如果遍历完成后依然未找到，返回索引错误。否则删除节点。

复杂度：时间复杂度 $T(n) = O(n)$

12. 遍历线性表

函数名称是 `ListTraverse(L, visit())`，初始条件是链表 `L` 已存在；操作结果是依次对 `L` 的每个数据元素调用函数 `visit()`。

如果线性表不存在，返回不可行。否则从第一个元素开始依次访问节点元

素直到节点的 next 指向 NULL。

复杂度：时间复杂度 $T(n) = O(n)$

13. 翻转线性表

链表翻转：函数名称是 `reverseList(L)`，初始条件是线性表 L 已存在，操作结果是将 L 翻转。

我们可以通过遍历链表，逐个改变节点之间的指针关系，实现了链表的翻转。在遍历过程中，通过使用三个指针变量，即前一个节点、当前节点和下一个节点，实现了节点指针的调整，使得链表的方向被逆序。最后，将链表的头节点指向翻转后的最后一个节点，完成了链表的翻转操作。

复杂度：时间复杂度 $T(n) = O(n)$

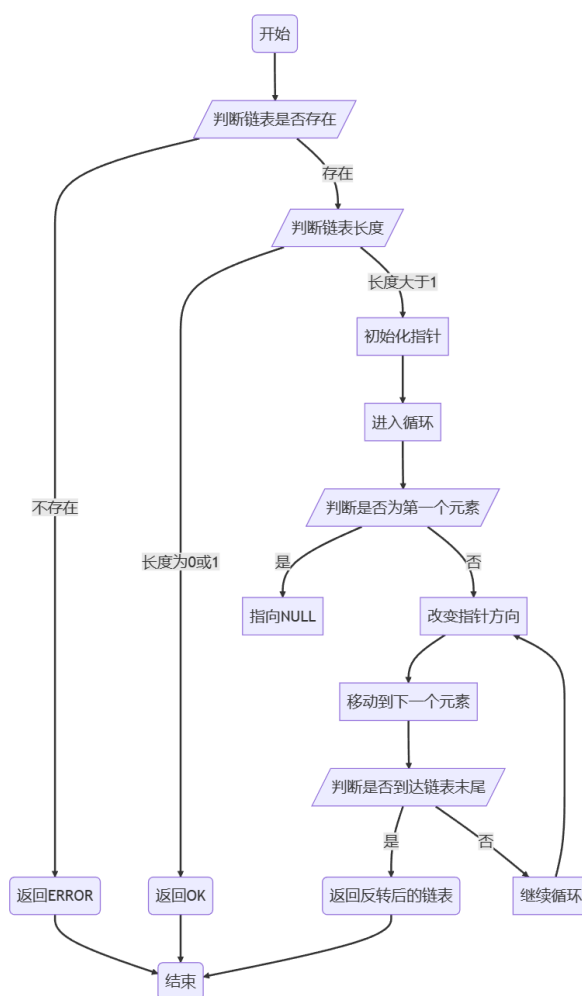


图 1-2 翻转线性表

14. 排序线性表

函数名称是 `sortList(L)`，初始条件是线性表 `L` 已存在；操作结果是将 `L` 由小到大排序；

如果线性表不存在，返回不可行。采用归并排序进行链表的排序，首先递归地从链表的中间节点拆分链表到单一节点，之后进行合并有序链表的操作。所有操作结束后即得到有序线性表。

复杂度：时间复杂度 $T(n) = O(n)$

15. 删除倒数第 n 个元素

函数名称是 `RemoveNthFromEnd(L,n)`；初始条件是线性表 `L` 已存在且非空，操作结果是该链表中倒数第 n 个节点；如果线性表不存在，返回不可行。否则遍历链表求出表长，根据表长获得倒数第 n 个元素的索引，根据这个索引遍历链表寻找删除位置，如果索引不大于 0，返回索引错误，如果遍历完成后依然未找到，返回索引错误。否则删除节点。

复杂度：时间复杂度 $T(n) = O(n)$

16. 线性表的文件操作

如果线性表不存在，返回不可行。打开只写文件，遍历线性表并依次写入元素。销毁线性表。接着初始化线性表，以只读模式打开刚才保存的文件，依次读入各元素，新建节点并插入线性表，直到读取到 EOF。

复杂度：时间复杂度 $T(n) = O(1)$

17. 实现多个线性表管理：设计相应的数据结构管理多个线性表的查找、添加、移除等功能。

针对多线性表的管理，实验设计一个新的结构体，其中定义了本实验链表的一个结构体数组，本质上是通过数组存储多个链表以此实现多线性表的管理，针对数组中每个线性表的管理和上述基础功能对单个链表的操作基本一致，不同在于，我们需要实现不同链表的切换，即找寻并切换至目标链表进行管理。

在设计过程中每一个链表都有一个 `name` 数组对链表以及对应的位序，因此，我们可以更改序号来切换不同链表，若没有找到目标或者查找的序号超过链表数组的最大数量，则返回 `ERROR`。

复杂度：时间复杂度 $T(n) = O(1)$

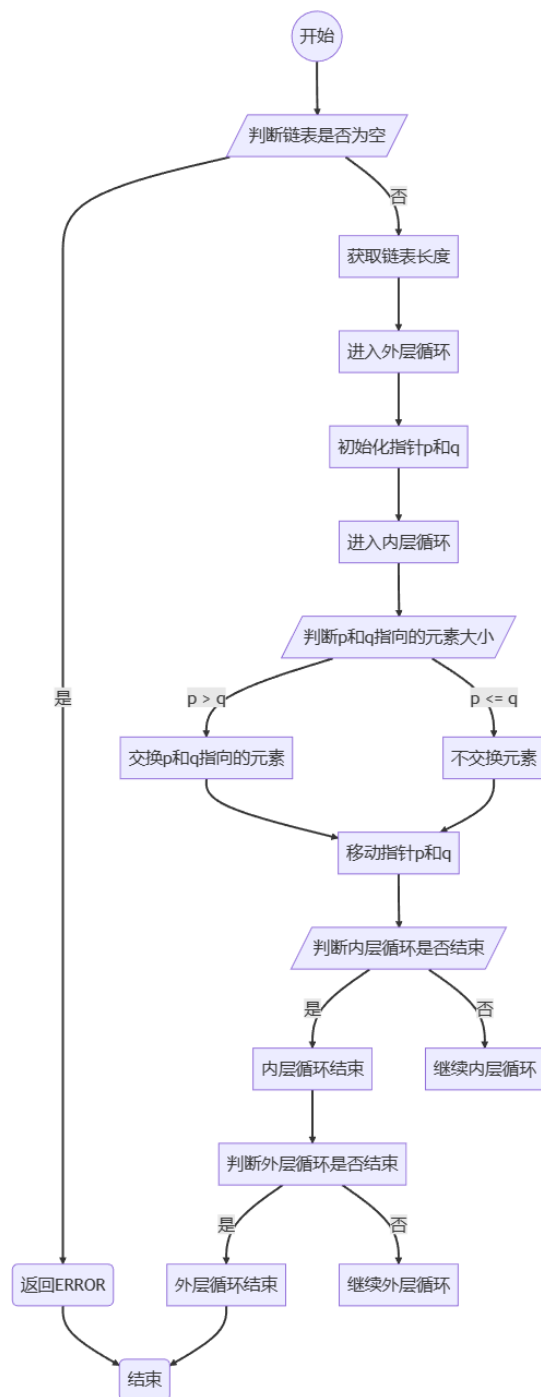


图 1-3 排序线性表

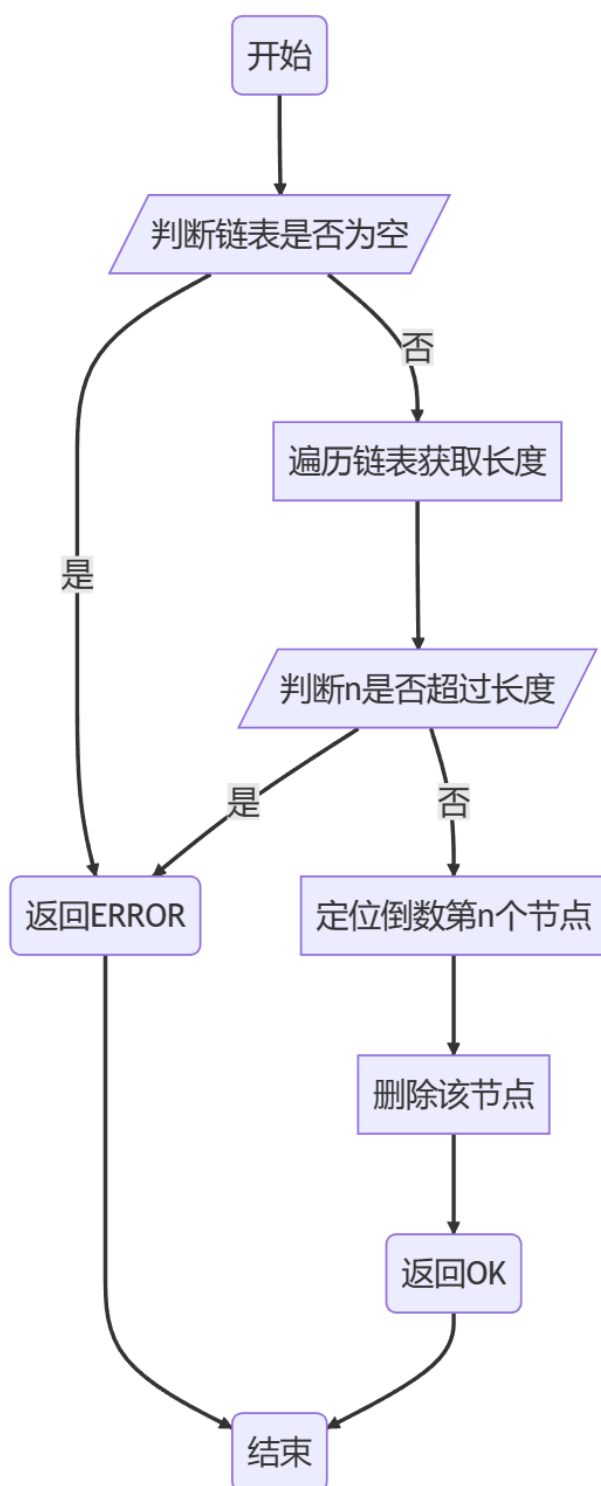


图 1-4 删除倒数第 n 个元素

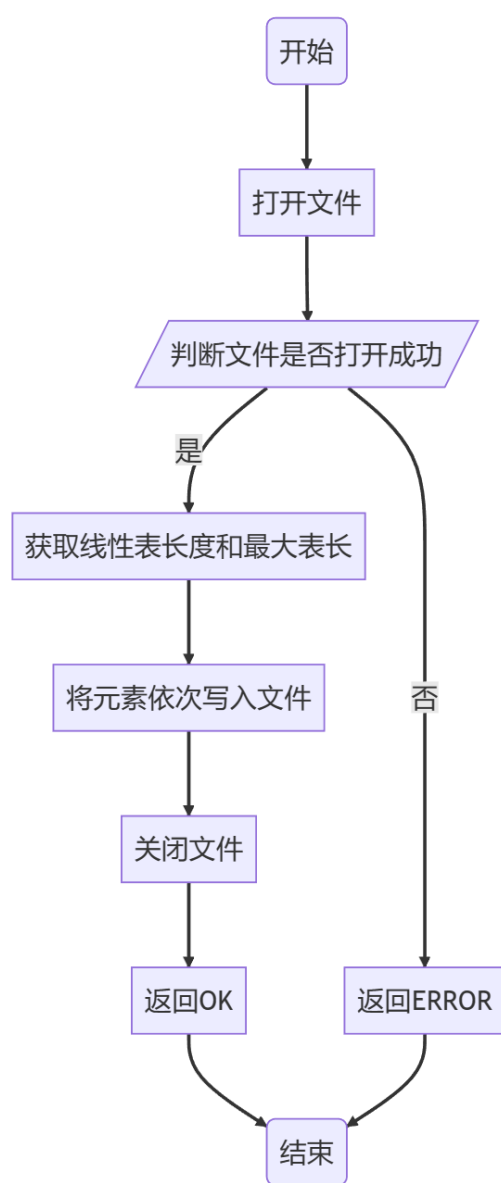


图 1-5 文件保存

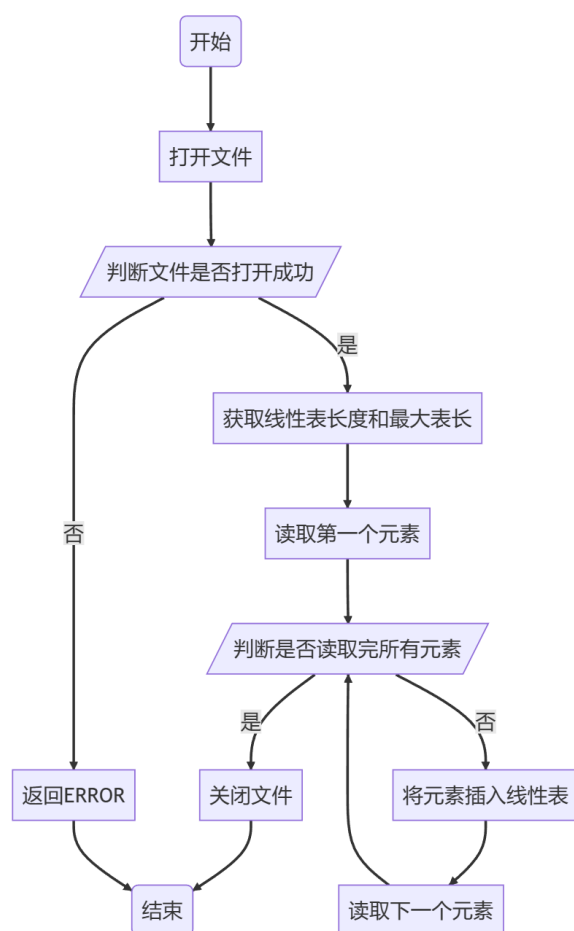


图 1-6 文件读取

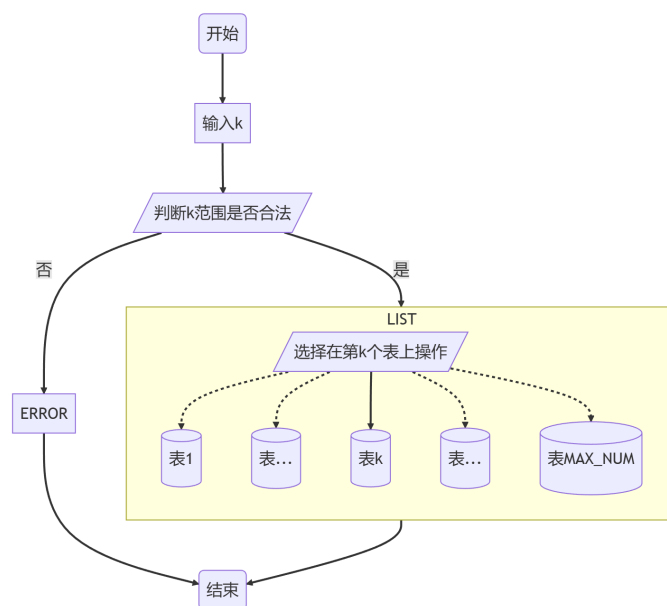


图 1-7 多表线性表管理的实现

1.4 系统测试

以下主要说明针对各个函数正常和异常的测试用例及测试结果。

1. 初始化线性表

若线性表未初始化，输出结果如下图：若线性表已初始化，输出结果如下图：

```
-----lbw-----
请选择你的操作[0--18]:
1
请输入要保存的线性表名称
aaa
线性表创建成功
```

图 1-8 初始化成功

```
-----lbw-----
请选择你的操作[0--18]:
1
请输入要保存的线性表名称
aaa
线性表创建失败
```

图 1-9 初始化失败

2. 销毁线性表

若线性表未销毁，输出结果如下图：若线性表已销毁，输出结果如下图：

```
-----lbw-----
请选择你的操作[0--18]:
2
销毁线性表成功！
```

图 1-10 销毁成功

3. 清空线性表

若线性表未清空，输出结果如下图：若线性表已清空，输出结果如下图：

4. 判定线性表是否为空

若线性表为空表，输出结果如下图：若线性表不为空表，输出结果如下图：

5. 求线性表的长度

输入 [1]，输出结果如下图：若线性表为空表，输出结果如下图：

6. 获取元素

输入 [1,2,3,4]，索引 1，输出结果如下图：输入 [1,2,3,4]，索引 0，输出结果如下图：

7. 查找元素

输入 [1,-2,3,4]，查找 1，输出结果如下图：输入 [1,-2,3,4]，查找 2，输出结果如下图：

8. 获取前驱元素

```
-----lbw-----
请选择你的操作[0--18]:
2
线性表不存在!
```

图 1-11 销毁失败

```
-----lbw-----
请选择你的操作[0--18]:
3
线性表重置成功!
```

图 1-12 清空成功

```
-----lbw-----
请选择你的操作[0--18]:
3
线性表不存在!
```

图 1-13 清空失败

```
-----lbw-----
请选择你的操作[0--18]:
4
线性表不存在!
```

图 1-14 判空失败

```
-----lbw-----
请选择你的操作[0--18]:
4
文件为空!
```

图 1-15 判空成功

```
-----lbw-----
请选择你的操作[0--18]:
5
线性表表长为1
```

图 1-16 求非空表的表长

```
-----lbw-----
请选择你的操作[0--18]:
5
线性表表长为0
```

图 1-17 求空表的表长

```
-----lbw-----
请选择你的操作[0--18]:
6
请输入要取结点的位置:
1
第1个结点的元素是: 1
```

图 1-18 获取元素成功

```
-----lbw-----
请选择你的操作[0--18]:
6
请输入要取结点的位置:
2
输入位置错误!
```

图 1-19 获取元素失败

```
-----lbw-----
请选择你的操作[0--18]:
7
请输入数据元素值:
1
1元素位于第1个位置!
```

图 1-20 查找元素成功

```
-----lbw-----
请选择你的操作[0--18]:
7
请输入数据元素值:
2
该元素不存在!
```

图 1-21 查找元素失败

输入 [1,2,3,4]，查找 2 的前驱元素，输出结果如下图：输入 [1,2,3,4]，查找

```
-----lbw-----
请选择你的操作[0--18]:
8
请输入数据元素:
2
其前驱元素为: 1
```

图 1-22 查找前驱元素成功

1 前驱元素，输出结果如下图：

```
-----lbw-----
请选择你的操作[0--18]:
8
请输入数据元素:
1
其不存在前驱元素!
```

图 1-23 查找前驱元素失败

9. 获取后继元素

输入 [1,2]，查找 1 的后继元素，输出结果如下图：输入 [1,2,3,4]，查找 2 的前驱元素，输出结果如下图：

输入 [1,2]，查找 2 的后继元素，输出结果如下图：

10. 插入元素

输入 [1,2]，插入元素 3 到位置 3，输出结果如下图：

输入 [1,2]，插入元素 3 到位置 4，输出结果如下图：

11. 删除元素

输入 [1,2,3]，删除位置 3 上的元素，输出结果如下图：

输入 [1,2,3]，删除位置 0 上的元素，输出结果如下图：

12. 遍历线性表

输入 [1,2]，遍历结果如下图：

13. 翻转线性表

输入 [1,2,3,4,-5,-4]，输出结果如下图：

14. 排序线性表

输入 [1,3,4,-5,-4]，输出结果如下图：

15. 删除倒数第 n 个元素

输入 [-4,-5,4,3,2,1]，删除倒数第 2 个元素，输出结果如下图：

输入 [-4,-5,4,3,2,1]，删除倒数第 7 个元素，输出结果如下图：


```
-----lbw-----
请选择你的操作[0--18]:
9
请输入数据元素:
1
其后继元素为: 2
```

图 1-24 查找后继元素成功

```
-----lbw-----
请选择你的操作[0--18]:
9
请输入数据元素:
2
其不存在后继元素!
```

图 1-25 查找后继元素失败

```
-----lbw-----
请选择你的操作[0--18]:
10
请输入您要插入的数据元素:
3
请输入您要插入的数据元素的位置:
3
插入数据元素成功!
```

图 1-26 插入元素成功

```
-----lbw-----
请选择你的操作[0--18]:
10
请输入您要插入的数据元素:
3
请输入您要插入的数据元素的位置:
4
插入数据元素失败!
```

图 1-27 插入元素失败

```
-----lbw-----
请选择你的操作[0--18]:
11
请输入您要删除的数据元素的位置:
3
删除数据元素成功!
```

图 1-28 删除元素成功

```
-----lbw-----
请选择你的操作[0--18]:
11
请输入您要删除的数据元素的位置:
0
删除数据元素失败!
```

图 1-29 删除元素失败

```
-----lbw-----
请选择你的操作[0--18]:
12

----- all elements -----
1 2
----- end -----
```

图 1-30 遍历线性表成功

```
-----lbw-----
请选择你的操作[0--18]:
16
线性表已翻转!

-----lbw-----
请选择你的操作[0--18]:
12

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

图 1-31 翻转线性表成功

```
-----lbw-----
请选择你的操作[0--18]:
18
排序已完成!

-----lbw-----
请选择你的操作[0--18]:
12

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

图 1-32 排序线性表成功

```
-----lbw-----
请选择你的操作[0--18]:
17
请输入要删除倒数第n个结点
2
结点已删除!

-----lbw-----
请选择你的操作[0--18]:
12

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

图 1-33 删除倒数第 n 个元素成功

16. 线性表的文件操作

文件保存：输入 [1,2,3,4]，保存为 test.txt 文件，文件内容如下：

文件读取：初始化表后，我们读入 test.txt 文件，输出结果如下图：

17. 多线性表管理

对第一个表，输入 [1,2,3,4]，切换到到第 2 个表并初始化。此时第一个表不为空表，而第二个表为空表，从而说明多线性表操作的独立性与正确性：

1.5 实验小结

通过本次实验，我加深了对链式存储的线性表的理解，并掌握了如何运用单链表解决实际问题。通过本次实验，我学到了：

1. 单链表的定义
2. 单链表的基本操作算法
3. 链式存储线性表的定义
4. 链式存储线性表的基本操作算法
5. 链式存储线性表的管理表的定义
6. 链式存储线性表的管理表的基本操作算法
7. 单链表的实际应用

```
-----lbw-----
请选择你的操作[0--18]:
17
请输入要删除倒数第n个结点
7
线性表不存在!
```

图 1-34 删除倒数第 n 个元素失败

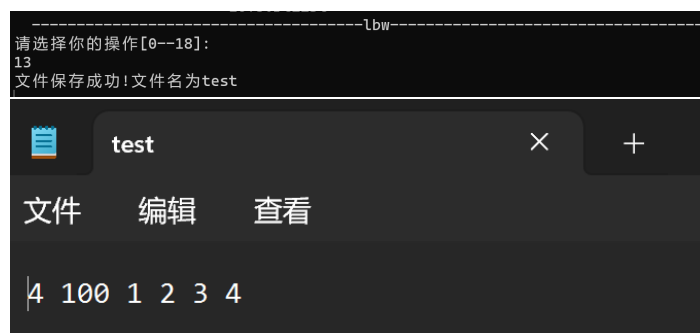


图 1-35 文件保存

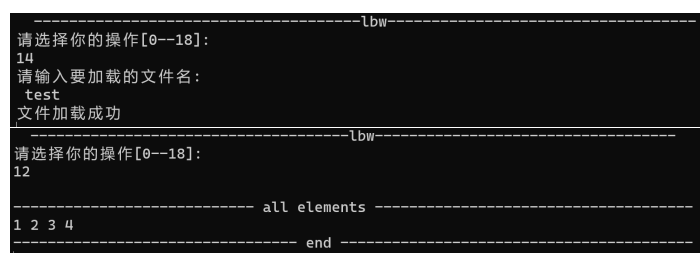


图 1-36 文件读取

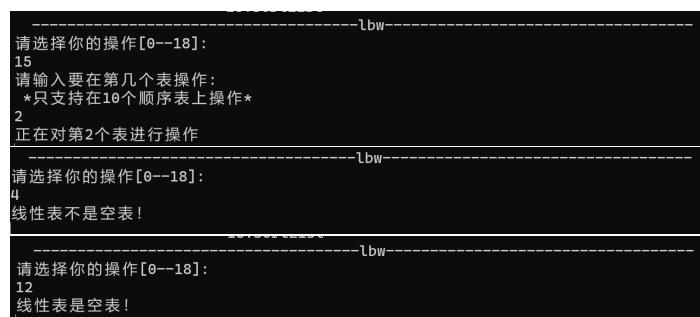


图 1-37 多线性表管理

通过本次实验，我认为我还有如下不足之处：

1. 对单链表的指针操作不太熟练
2. 不太明确线性表的实际应用

2 基于二叉链表的二叉树实现

2.1 问题描述

本实验实现了二叉树的二叉链表存储，构造一个具有菜单功能的演示系统，实现了二叉树的初始化、清空、求二叉树的深度等 14 种基本功能和全部的 5 种附加功能，还实现了多二叉树管理。

2.2 系统设计

基于二叉树的定义：二叉树是一种每个结点至多只有两个子树（即二叉树的每个结点的度不大于 2），并且二叉树的子树有左右之分，其次序不能任意颠倒。本次实验采用树的链式存储方式，实现了树的基本的功能，比如增添、删除、查找等。同时，还在此基础上添加了一些高级功能，比如翻转、求最大路径和、求 LCA 等。该程序还使用语言文字提示，降低了程序的使用难度。

本实验涉及的头文件以及相关常量定义如下：

相关常量定义

```
1  /* Linear Table On Sequence Structure */
2  #include <stdio.h>
3  #include <malloc.h>
4  #include <stdlib.h>
5  #include <stack>
6  #include <queue>
7  #include <stdlib.h>
8
9  // 定义每个节点的打印宽度为 5
10 #define NODE_WIDTH 5
11
12 // 定义每个节点的占位符为 _
13 #define NODE_PLACEHOLDER '_'
14
15 /*-----page 10 on textbook -----*/
16 #define TRUE 1//定义真值
17 #define FALSE 0//定制假值
```

```
18 #define OK 1//程序正常运行
19 #define ERROR 0//程序运行出错
20 #define INFEASTABLE -1//输入或输出不合法
21 #define OVERFLOW -2//数值溢出
22 #define MAX_NUM 10//最大个数
23 #define LIST_INIT_SIZE 100//最大尺寸
24 #define LISTINCREMENT 10//最大增加量
25
26 typedef int status; //定义所有状态码和返回值的类型为int
27 typedef char TElemType; //数据元素类型定义
28 status definition[1000]={0}; //定义并初始化关键字哈希表
```

本系统的数据结构有两种：二叉树和二叉树的管理表。其具体定义如下。

相关数据结构定义

```
1 // 定义二叉树的节点结构体
2 typedef struct BiTNode{
3     int key; // 用 key 作为标记，便于查找节点
4     TElemType data; // char 类型数据域
5     struct BiTNode *lchild, *rchild;
6     // 定义二叉链表的左孩子指针与右孩子指针
7 } BiTNode, *BiTree; // BiTNode 类型指针 BiTree
8
9 // 定义一个结构体，用于保存二叉树和树的名称
10 typedef struct {
11     BiTree T; // 创建二叉树用的指针 T
12     char name[20]; // 用于保存树的名称
13 } LElemType;
14
15 // 定义一个结构体，用于保存多个树进行操作
16 typedef struct {
17     LElemType tree[20]; // 多个树进行操作
18     int length;
19     int listsize;
20 } SqList;
```

系统的总体架构：界面上采用简易菜单演示系统，在 while 循环中建立菜单

演示，op 代表用户选择的操作序号，程序将首先判断 op 的合法性，若合法则通过 switch 函数进入功能的选择，进入相关功能函数执行相关操作，操作完成后继续执行循环，直到用户输入 0 时，退出系统。

多线性表管理通过 Choose 函数实现，用户可以输入位序切换线性表，相关操作仅在当前树完成，而不会影响森林。

```
Menu for Binary Tree On Binary Linked List
*****
1.  InitBiTree      2.  DestroyBiTree
3.  CreateBiTree    4.  ClearBiTree
5.  BiTreeEmpty     6.  BiTreeDepth
7.  Assign          8.  GetSibling
9.  InsertChild     10. DeleteChild
11. PreOrderTraverse 12. InOrderTraverse
13. PostOrderTraverse 14. LevelOrderTraverse
15. Choose(多树操作) 16. Save(保存文件)
17. Load(加载文件)  18. LowestCommonAncestor
19. TreeDisplay      20. InvertTree
21. MaxPathSum       22. LocateNode 0. Exit
*****
*****Powered By @_@||lbw*****
请选择你的操作[0~22]:
```

图 2-1 菜单演示系统

2.3 系统实现

本程序在 Windows 11 系统下采用 Dev-C++ 进行编译调试，语言选择 C 语言。以下主要说明各个主要函数的实现思想，函数和系统实现的源代码放在附录中。

（本实验所有函数在实现功能之前会先对是否已有树进行判定，若树不存在，则返回 *INFEASIBLE*，在各函数具体设计思路中不再叙述此条。）

1. 创建二叉树

创建二叉树：函数名称是 `CreateBiTree(T,definition)`；初始条件是 `definition` 给出二叉树 `T` 的定义，如带空子树的二叉树前序遍历序列、或前序 + 中序、或后序 + 中序；操作结果是按 `definition` 构造二叉树 `T`；

如果二叉树已存在，返回不可行。检查关键字是否重复，若重复返回错误，否则调用递归函数 `create`。递归函数通过 `definition` 数组检查输入的结点是否是空结点，若是则返回，否则创建并插入结点，再递归地创建左子树和右子树。

复杂度：时间复杂度 $T(n) = O(1)$

2. 清空二叉树

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

如果二叉树不存在，返回不可行。遇到空结点返回，递归地遍历左子树和右子树后释放结点空间。

复杂度：时间复杂度 $T(n) = O(n)$

3. 销毁二叉树

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

与清空操作类似，如果二叉树不存在，返回不可行。遇到空结点返回，递归地遍历左子树和右子树后释放结点空间。此外，不同于清空，我们还需要对头结点进行清除操作

复杂度：时间复杂度 $T(n) = O(n)$

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

仅需要判断头结点指针是否为空即可， T 指向 `NULL` 则返回 `TRUE`，不为空则返回 `FALSE`。

复杂度：时间复杂度 $T(n) = O(1)$

5. 求二叉树的深度

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

遇到空结点返回，使用递归方式计算左右子树的深度，然后取较大值加 1 作为当前树的深度。

复杂度：时间复杂度 $T(n) = O(n)$

6. 查找结点

函数名称是 `LocateNode(T,e)`；初始条件是二叉树 T 已存在， e 是和 T 中结点关键字类型相同的给定值；操作结果是返回查找到的结点指针，如无关键字为 e 的结点，返回 `NULL`；

该函数的返回值为指针类型，在查找过程中基于关键字的唯一性，所以通过比较语句发现关键字相同的结点即可把当前遍历的结点作为返回值返回，在循环外返回空指针，表示遍历过程如果没有正常 `return`，就说明树中没有对应的结点，返回空指针表示查找失败。

复杂度：时间复杂度 $T(n) = O(n)$

7. 结点赋值

函数名称是 `Assign(T,e,value)`；初始条件是二叉树 T 已存在， e 是和 T 中结点关键字类型相同的给定值；操作结果是关键字为 e 的结点赋值为 `value`；

首先检查关键字是否重复，然后查找结点。如果结点关键字重复或结点未找到，返回错误，否则把找到的结点的关键字和名称赋值成用户输入的值。

复杂度：时间复杂度 $T(n) = O(n)$

8. 获取兄弟结点

函数名称是 `GetSibling(T,e)`；初始条件是二叉树 T 存在， e 是和 T 中结点关键字类型相同的给定值；操作结果是返回关键字为 e 的结点的（左或右）兄弟结点指针。若关键字为 e 的结点无兄弟，则返回 `NULL`；

首先检查传入的结点是否是指定结点的双亲结点，如果是，返回其兄弟结点。如果结点为空，返回。否则递归地查找左子树和右子树。

复杂度：时间复杂度 $T(n) = O(n)$

9. 插入结点

函数名称是 `InsertNode(T,e,LR,c)`；初始条件是二叉树 T 存在， e 是和 T 中结点关键字类型相同的给定值， LR 为 0 或 1， c 是待插入结点；操作结果是根根据 LR 为 0 或者 1，插入结点 c 到 T 中，作为关键字为 e 的结点的左或右孩子结点，结点 e 的原有左子树或右子树则为结点 c 的右子树；

首先检查关键字是否重复，然后查找结点。如果结点关键字重复或结点未找到，返回错误。再根据插入方向 LR 的不同，将子树 c 插入到父节点 p 的相应位置。如果插入方向为左子树，则先将 p 的左子树作为 c 的右子树，然后将 c 作为 p 的新左子树；如果插入方向为右子树，则先将 p 的右子树作为 c 的右子树，然后将 c 作为 p 的新右子树。

复杂度：时间复杂度 $T(n) = O(n)$

10. 删除结点

函数名称是 `DeleteNode(T,e)`；初始条件是二叉树 T 存在， e 是和 T 中结点关键字类型相同的给定值。操作结果是删除 T 中关键字为 e 的结点；同时，如果关键字为 e 的结点度为 0，删除即可；如关键字为 e 的结点度为 1，用关键字为 e 的结点孩子代替被删除的 e 位置；如关键字为 e 的结点度为 2，用 e 的左孩子代替被删除的 e 位置， e 的右子树作为 e 的左子树中最右结点的右子树；

首先检查关键字是否重复，然后根据删除方向 LR 的不同，先将父节点 p 的相应子树存储到临时变量 $T1$ 中，再将父节点 p 的相应子树指针置为 `NULL`，最后调用 `DestroyBiTree` 函数释放临时变量 $T1$ 所指向的子树的内存空间。如

果双亲进度结点未找到且删除的是根结点，直接清空二叉树。若销毁子树操作成功，则返回 OK；否则返回 ERROR。

复杂度：时间复杂度 $T(n) = O(n)$

11. 先序遍历

函数名称是 `PreOrderTraverse(T,Visit)`；初始条件是二叉树 T 存在；操作结果：先序遍历，对每个结点调用函数 Visit 一次且一次。

首先访问当前结点，在递归访问左子树和右子树。

复杂度：时间复杂度 $T(n) = O(n)$

12. 中序遍历

函数名称是 `InOrderTraverse(T,Visit)`；初始条件是二叉树 T 存在；操作结果：中序遍历，对每个结点调用函数 Visit 一次且一次，

首先递归访问左子树，然后访问当前结点，再递归地访问右子树。

复杂度：时间复杂度 $T(n) = O(n)$

13. 后序遍历

函数名称是 `PostOrderTraverse(T,Visit)`；初始条件是二叉树 T 存在；操作结果：后序遍历，对每个结点调用函数 Visit 一次且一次，

首先递归访问左子树和右子树，再访问当前结点。

复杂度：时间复杂度 $T(n) = O(n)$

14. 按层遍历

函数名称是 `LevelOrderTraverse(T,Visit)`；初始条件是二叉树 T 存在；

采用广度优先搜索遍历二叉树。首先新建队列并将根结点入队，然后逐个访问队首元素，把队首元素的未入队子结点入队，弹出队首元素，直到队列为空。

复杂度：时间复杂度 $T(n) = O(n)$

15. 翻转二叉树

函数名称是 `InvertTree(T)`，初始条件是线性表 L 已存在；操作结果是将 T 翻转，使其所有节点的左右节点互换；

首先交换传入结点的两个子结点，如果是空结点则返回，然后递归地翻转左子树和右子树。

复杂度：时间复杂度 $T(n) = O(n)$

16. 最大路径和

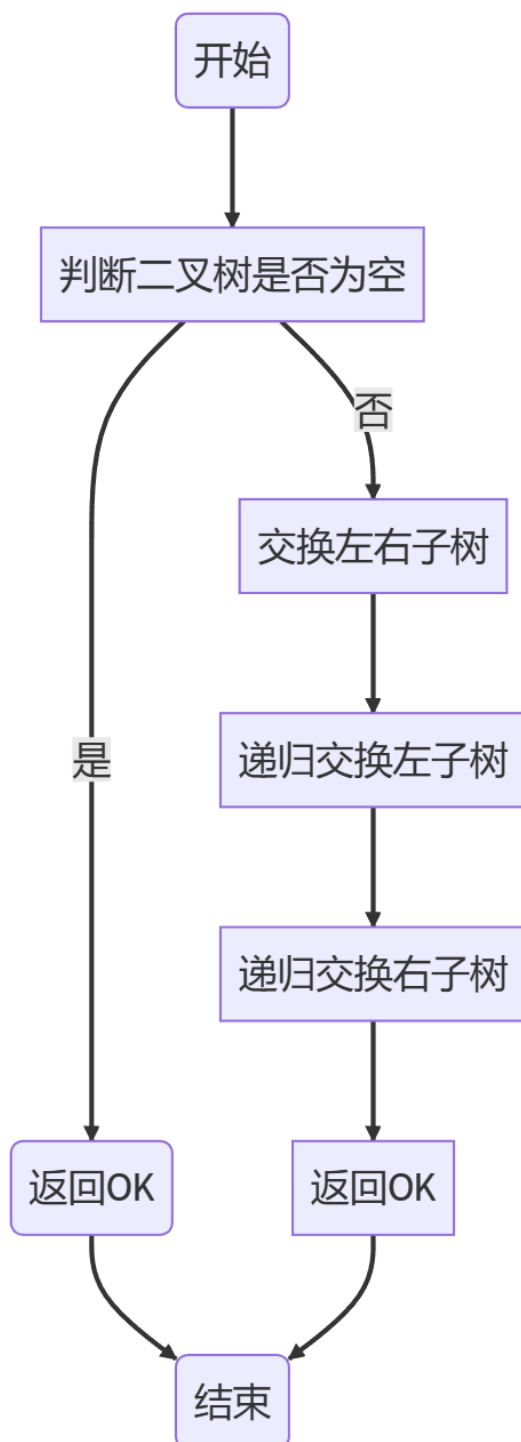


图 2-2 翻转二叉树

函数名称是 $\text{MaxPathSum}(T)$ ，初始条件是二叉树 T 存在；操作结果是返回根节点到叶子节点的最大路径；

遇到空结点返回，否则递归地计算左子树和右子树的最大路径和，取最大的一个的值加上传入顶点的权值后返回。

复杂度：时间复杂度 $T(n) = O(n)$

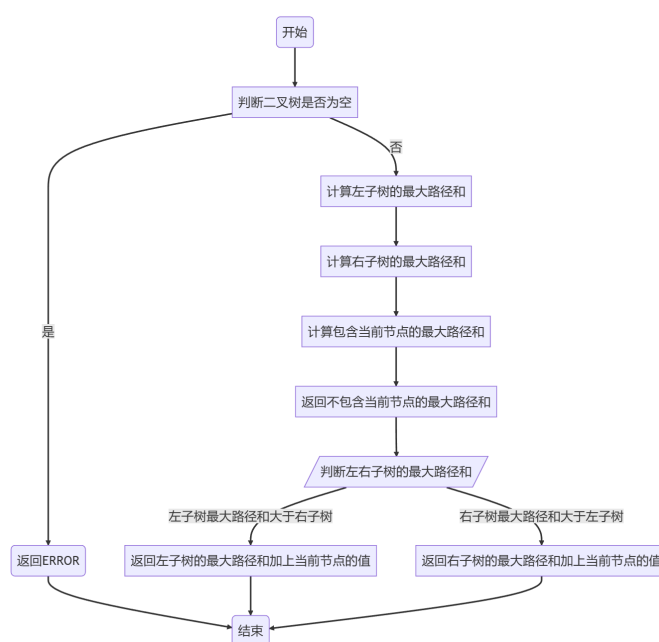


图 2-3 最大路径和

17. 最近公共祖先

函数名称是 $\text{LowestCommonAncestor}(T, e1, e2)$ ；初始条件是二叉树 T 存在；操作结果是该二叉树中 $e1$ 节点和 $e2$ 节点的最近公共祖先；

首先查找结点，如果两个结点中有未找到的，返回错误。然后调用递归函数 LCA ， LCA 中序遍历二叉树，从访问了一个结点开始计数，求这时开始到访问到第二个结点时到达的层数最小的结点，这个结点就是最近公共祖先。

复杂度：时间复杂度 $T(n) = O(n)$

18. 线性表的文件操作

如果线性表不存在，返回不可行。打开只写文件，调用递归函数 save ，如果传入结点为空，返回。写入传入的二叉树结点，然后递归地写入左子树和右子树。最后清空二叉树。接着创建新二叉树，以只读模式打开刚才保存的文件，调用递归函数 load ，新建结点，读入数据。插入新二叉树，然后递归地读取左子树和右子树。

复杂度：时间复杂度 $T(n) = O(1)$

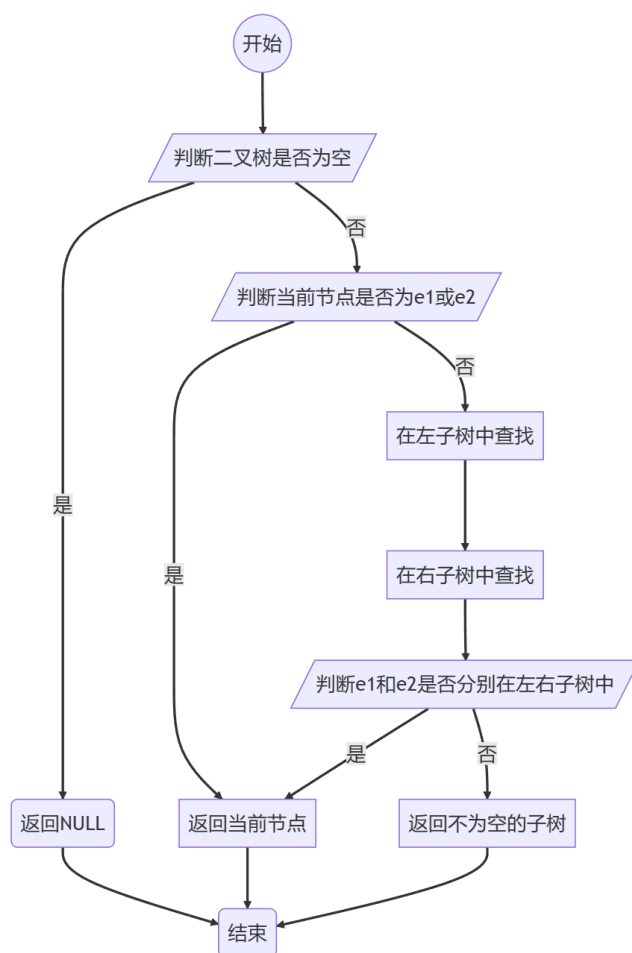


图 2-4 最近公共祖先

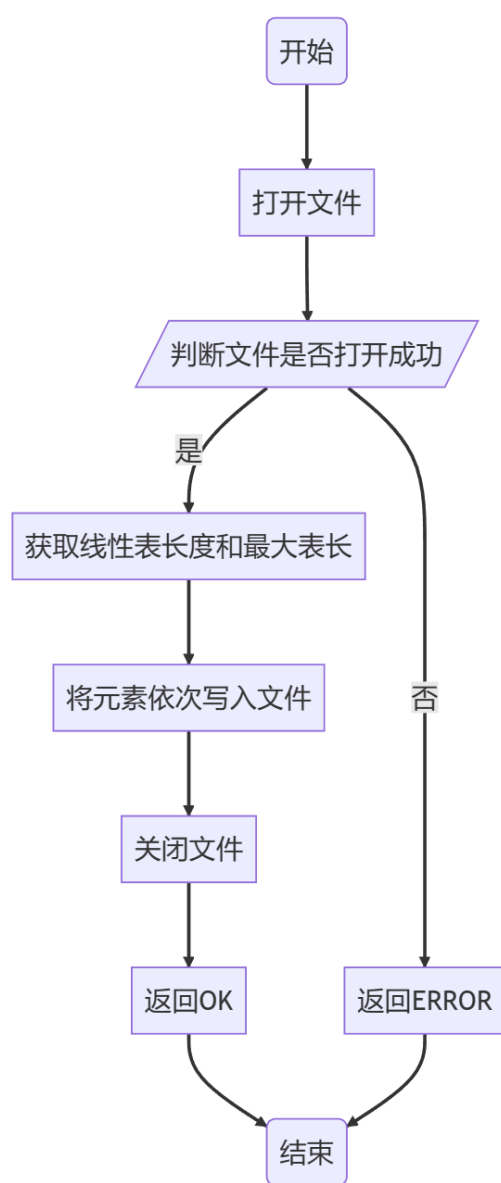


图 2-5 文件保存

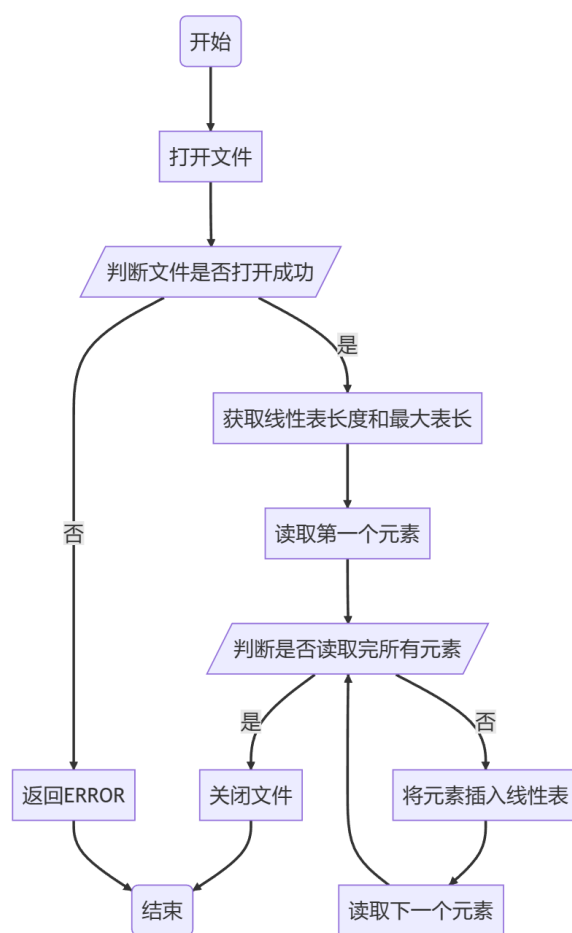


图 2-6 文件读取

19. 实现多个树管理：设计相应的数据结构管理多个树的查找、添加、移除等功能。

与多链表的管理类似，本实验设计一个结构体，其中定义了一个结构体数组，本质上是通过数组存储多个树以此实现多个树的管理，针对数组中每个树的管理和上述基础功能对单个树的操作基本一致，不同在于，我们需要实现不同树的切换，即找寻并切换至目标树进行管理。

在设计过程中每一个树都有一个 **name** 数组对链表以及对应的位序，因此，我们可以更改序号来切换不同链表，若没有找到目标或者查找的序号超过森林的最大数量，则返回 **ERROR**。

复杂度：时间复杂度 $T(n) = O(1)$

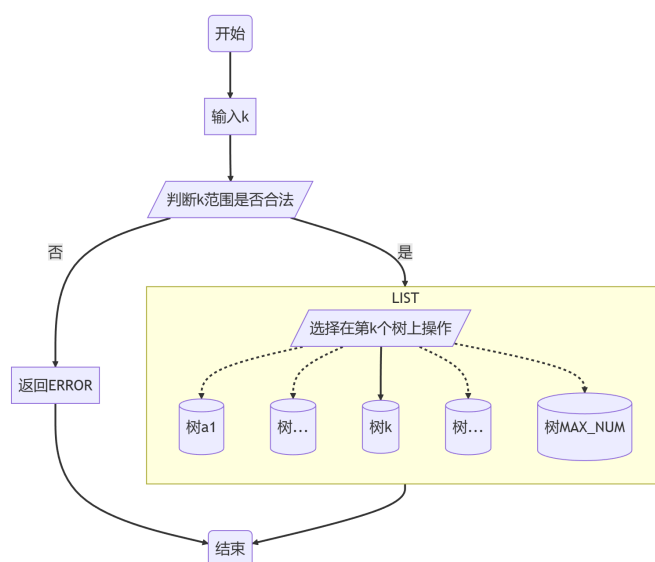


图 2-7 多树管理的实现

2.4 系统测试

以下主要说明针对各个函数正常和异常的测试用例及测试结果。测试开始以前序的方式输入序列 `1 1 a 2 2 b 3 3 c 6 4 d 7 5 e 0 0 null` 来创建二叉树 *A*，后续均进行连贯操作

1. 创建二叉树

输入二叉树 *A*，输出结果如下图：

```
请您输入数据。格式：节点关键字 节点名称。空节点：0 任意字符串。输入结束符：-1 任意字符串。
1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null
正在创建二叉树...
创建成功！
```

图 2-8 创建二叉树

2. 清空二叉树

若二叉树不存在，输出结果如下图：

```
请选择你的操作[0~22]:4
二叉树不存在！
```

图 2-9 二叉树清空失败

若二叉树存在，输出结果如下图：

```
请选择你的操作[0~22]:4
叉树清空成功！
```

图 2-10 二叉树清空成功

3. 求二叉树的深度


若二叉树不存在，输出结果如下图：

若二叉树不存在，输出结果如下图：

4. 查找结点

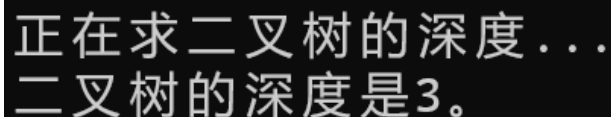
输入二叉树 *A*，查找关键字为 9 的节点，输出结果如下图：

输入二叉树 *A*，查找关键字为 1 的节点，输出结果如下图：



```
请选择你的操作[0~22]:6
二叉树不存在!
```

图 2-11 二叉树求深度失败

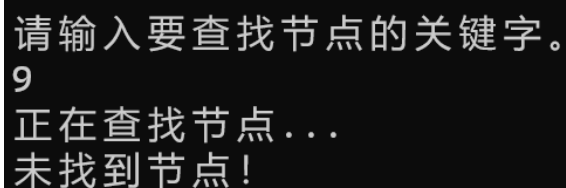


```
正在求二叉树的深度...
二叉树的深度是3。
```

图 2-12 二叉树求深度成功

5. 结点赋值

输入二叉树 A，将关键字为 1 的节点赋值为关键字为 4，节点名称赋值为 f，输出结果如下图：输入二叉树 A，将关键字为 1 的节点赋值为关键字为 4，节点名称赋值为 f，输出结果如下图：



```
请输入要查找节点的关键字。
9
正在查找节点...
未找到节点!
```

图 2-13 查找结点失败

```
请输入要查找节点的关键字。
```

```
1
```

```
正在查找节点...
```

```
节点的名称是a。
```

图 2-14 查找结点成功

```
请输入要赋值节点的key。
```

```
1
```

```
请输入新的值，格式：节点关键字 节点名称。
```

```
4 f
```

```
正在进行节点赋值...
```

```
名称重复或未找到！
```

图 2-15 结点赋值成功

6. 获取兄弟结点

输入二叉树 A，获取关键字为 3 节点的兄弟节点，输出结果如下图：

```
请输入要赋值节点的key。
```

```
1
```

```
请输入新的值，格式：节点关键字 节点名称。
```

```
6 f
```

```
正在进行节点赋值...
```

```
节点赋值成功！
```

图 2-16 结点赋值成功

```
请输入进行查找兄弟节点的节点的key。  
3  
正在查找节点...  
兄弟节点的关键字和名称是2 b。
```

图 2-17 获取兄弟结点成功

7. 插入结点

输入二叉树 A，插入节点 8 g 到关键字为 2 节点的右边，输出结果如下图：

```
请输入作为插入位置的节点的key。  
2  
请选择希望待插入节点成为上述节点的左孩子还是右孩子。左：0 右：11  
请输入待插入节点的值，格式：节点关键字 节点名称。  
8 g  
正在插入...  
插入成功！  
插入后的二叉树如下：  
      f  
     / \  
    b   c  
   / \ / \  
  g d e
```

图 2-18 插入结点成功

8. 删除结点

输入二叉树 A，删除关键字为 2 的节点，输出结果如下图：

```
请输入要删除节点的key。  
2  
正在删除节点...  
节点删除成功！  
删除后的二叉树如下：  
      f  
     / \  
    g   c  
       / \  
      d  e
```

图 2-19 删除结点成功

9. 先序遍历

输入二叉树 A，进行先序遍历, 输出结果如下图:

```
正在进行先序遍历...  
6,f 8,g 3,c 4,d 5,e
```

图 2-20 先序遍历

10. 中序遍历

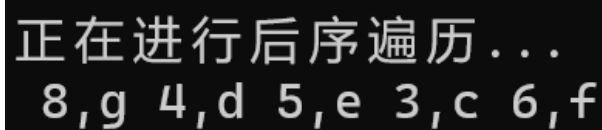
输入二叉树 A，进行中序遍历, 输出结果如下图:

```
正在进行中序遍历...  
8,g 6,f 4,d 3,c 5,e
```

图 2-21 中序遍历

11. 后序遍历

输入二叉树 A，进行先序遍历，输出结果如下图：

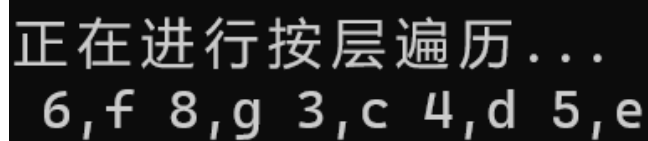


```
正在进行后序遍历...  
8,g 4,d 5,e 3,c 6,f
```

图 2-22 后序遍历

12. 层序遍历

输入二叉树 A，进行先序遍历，输出结果如下图：



```
正在进行按层遍历...  
6,f 8,g 3,c 4,d 5,e
```

图 2-23 层序遍历

13. 翻转二叉树

输入二叉树 A, 输出结果如下图:

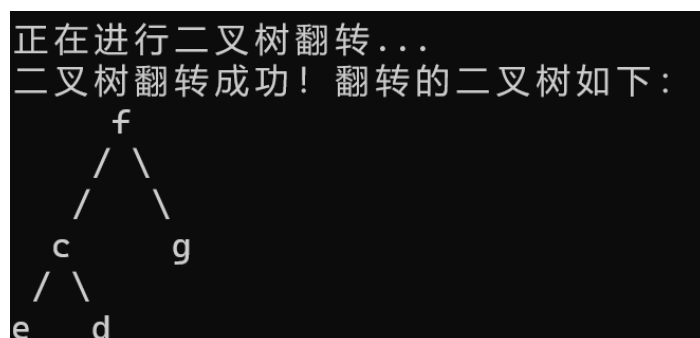


图 2-24 翻转二叉树

14. 最大路径和

输入二叉树 A, 输出结果如下图:

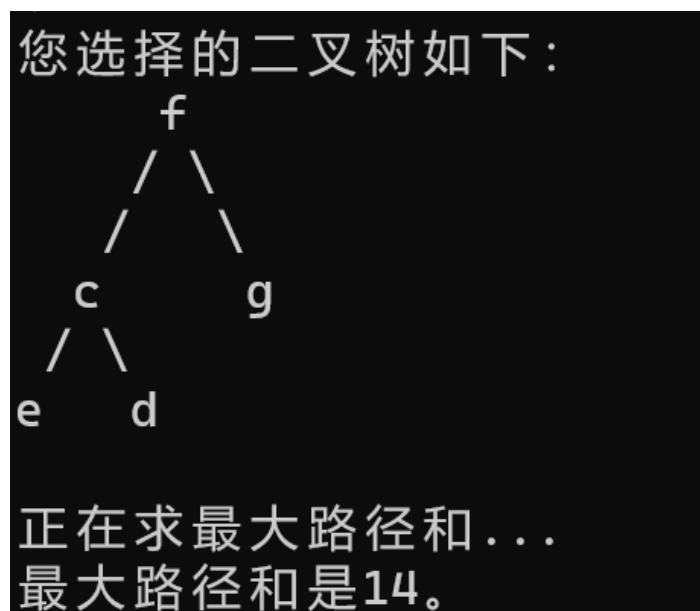


图 2-25 翻转二叉树

15. 最近公共祖先

输入二叉树 A, 输出结果如下图:

```
请输入节点1的值, 格式: 节点关键字 节点名称。  
3 c  
请输入节点2的值, 格式: 节点关键字 节点名称。  
8 g  
正在求最近公共祖先...  
最近公共祖先是6 f。
```

图 2-26 最近公共祖先

16. 二叉树的文件操作

文件保存：将以上二叉树 A 保存为 test.txt 文件，文件内容如下：



图 2-27 文件保存

文件读取：初始化二叉树后，我们读入 test.txt 文件，输出结果如下图：

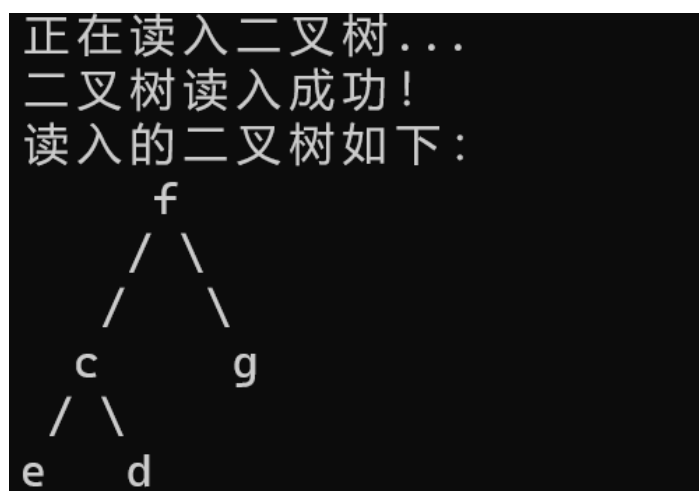


图 2-28 文件读取

17. 多线性表管理

对第一个树，读取 test.txt 文件，切换到第 2 个树并初始化。此时第一个树不为空树，而第二个表为空树，从而说明多线性表操作的独立性与正确性：

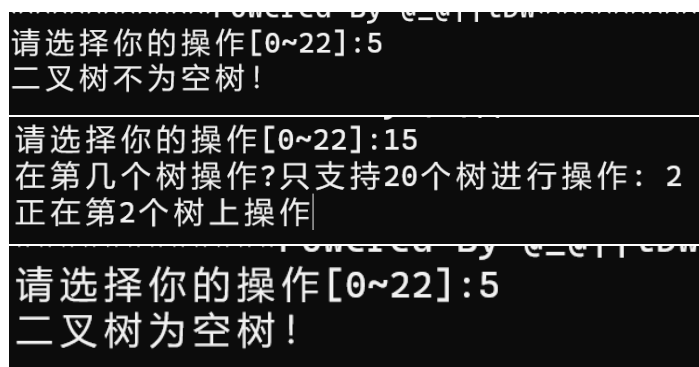


图 2-29 多树管理

2.5 实验小结

通过本次实验，我加深了对二叉链表存储的二叉树的理解，并掌握了如何运用二叉树解决实际问题。通过本次实验，我学到了：

1. 二叉链表的定义
2. 二叉链表的基本操作算法
3. 二叉链表存储的二叉树的定义
4. 二叉链表存储的二叉树的基本操作算法
5. 二叉树的管理表的定义
6. 二叉树的管理表的基本操作算法
7. 二叉树的实际应用

通过本次实验，我认为我还有如下不足之处：

1. 对其他结构存储的二叉树操作不够熟练
2. 对二叉树的复杂操作不够熟练

3 课程的收获和建议

3.1 基于顺序存储结构的线性表实现

我的收获：

1. 加深了对顺序表存储的线性表的理解
2. 掌握了如何运用线性表解决实际问题
3. 学到了顺序表的定义
4. 学到了顺序表的基本操作算法

我的建议：

1. 增加一些有实际应用背景的实验内容
2. 减少一些过于基础的实验内容

3.2 基于链式存储结构的线性表实现

我的收获：

1. 加深了对链式存储的线性表的理解
2. 掌握了如何运用线性表解决实际问题
3. 学到了单链表的定义
4. 学到了单链表的基本操作算法
5. 学到了单链表的管理表的定义
6. 学到了单链表的管理表的基本操作算法

我的建议：

1. 增加一些有实际应用背景的实验内容
2. 增加一些有关 next 指针操作的实验内容
3. 减少一些过于基础的实验内容

3.3 基于二叉链表的二叉树实现

我的收获：

1. 加深了对二叉链表存储的二叉树的理解

2. 掌握了如何运用二叉树解决实际问题
3. 学到了二叉链表的定义
4. 学到了二叉链表的基本操作算法
5. 学到了二叉树的管理表的定义
6. 学到了二叉树的管理表的基本操作算法

我的建议：

1. 增加一些有实际应用背景的实验内容
2. 增加一些以其它存储方式实现的二叉树的实验
3. 增加有关堆的操作

3.4 基于邻接表的图实现

我的收获：

1. 加深了对图的理解
2. 掌握了如何图解决实际问题
3. 学到了邻接表的定义
4. 学到了邻接表的基本操作算法
5. 学到了图的管理表的定义
6. 学到了图的管理表的基本操作算法

我的建议：

1. 增加一些有实际应用背景的实验内容
2. 增加一些以其它存储方式实现的图的实验

4 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. 严蔚敏等. 数据结构题集（C 语言版）. 清华大学出版社

5 附录 A 基于顺序存储结构线性表实现的源程序

相关定义

```
1  /* Linear Table On Sequence Structure */
2  #include <stdio.h>
3  #include <malloc.h>
4  #include <stdlib.h>
5
6  /*-----page 10 on textbook -----*/
7  /* 定义常量 */
8  #define TRUE 1
9  #define FALSE 0
10 #define OK 1
11 #define ERROR 0
12 #define INFEASTABLE -1
13 #define OVERFLOW -2
14 #define MAX_NUM 10
15 /* 定义数据类型 */
16 typedef int status;
17 typedef int ElemType; //数据元素类型定义
18
19 /*-----page 22 on textbook -----*/
20 /* 定义顺序表结构体 */
21 #define LIST_INIT_SIZE 100
22 #define LISTINCREMENT 10
23 typedef struct{ //顺序表（顺序结构）的定义
24     ElemType * elem;
25     int length;
26     int listsize;
27 }Sqlist;
28 /* 定义全局变量 */
29 FILE *fp;
30 /*-----page 19 on textbook -----*/
31 /* 函数声明 */
```

```
32 status InitiaList(Sqlist &L);
33 // 初始化顺序表
34 status DestroyList(Sqlist &L);
35 // 销毁顺序表
36 status ClearList(Sqlist &L);
37 // 清空顺序表
38 status ListEmpty(Sqlist L);
39 // 判断顺序表是否为空
40 int ListLength(Sqlist L);
41 // 获取顺序表长度
42 status GetElem(Sqlist L, int i, ElemType *e);
43 // 获取指定位置的元素
44 status LocateElem(Sqlist L, ElemType e, status(*compare)(ElemType
    , ElemType));
45 // 查找元素
46 status compare(ElemType a, ElemType b);
47 // 比较元素
48 status PriorElem(Sqlist L, ElemType cur_e, ElemType *pre_e);
49 // 获取指定元素的前一个元素
50 status NextElem(Sqlist L, ElemType cur_e, ElemType *next_e);
51 // 获取指定元素的后一个元素
52 status ListInsert(Sqlist &L, int i, ElemType e);
53 // 在指定位置插入元素
54 status ListDelete(Sqlist *L, int i, ElemType *e);
55 // 删除指定位置的元素
56 status ListTraverse(Sqlist L);
57 // 遍历顺序表
58 status SaveList(Sqlist L, char *filename);
59 // 将顺序表保存到文件
60 status LoadList(Sqlist *L, char *filename);
61 // 从文件中加载顺序表
62 int countSubarray(Sqlist L, int k);
63 // 计算顺序表中和为k的子数组个数
64 int maxSubArraySum(Sqlist L);
65 // 求顺序表中连续子数组的最大和
```

```
66 int SortList(SqlList &L);
67 //排序顺序表
68 int max(int a, int b) ;
69 //求两个数中的最大值
70 /*-----*/
```

演示系统

```
1  int main(void){
2      char filename[40];
3      int op=1;
4      int i;
5      int i_num=1;
6      SqlList L[MAX_NUM];
7      for(i=0;i<MAX_NUM;i++)
8      {
9          L[i].elem = NULL;
10         L[i].listsize = 0;
11         L[i].length = 0;
12     }
13     //上面的for循环是用来生成没有存储空间的线性表
14     ElemType e, cur_e , pre_e, next_e;
15     while(op){
16         /**
17         *利用最简单的printf来制作简易的菜单，可供选择；
18         *简洁美观的菜单有助于平复测试时的心情!!!
19         */
20         system("cls"); //用于清屏
21         printf("\n\n");
22         printf("          \t\t\tMenu for Linear Table On
                Sequence Structure \n");
23         printf("    可在%d个顺序表进行多表操作，初始化请先
                操作功能15,默认在第一个表上操作\n", MAX_NUM);
24         printf("
                -----
                n");
```



```
25         printf("**\t\t\t1. InitiaList          7. LocateElem\n");
26         printf("**\t\t\t2. DestroyList        8. PriorElem\n");
27         printf("**\t\t\t3. ClearList          9. NextElem\n");
28         printf("**\t\t\t4. ListEmpty          10. ListInsert\n");
29         printf("**\t\t\t5. ListLength         11. ListDelete\n");
30         printf("**\t\t\t6. GetElem            12.\n");
31         printf("**\t\t\t13. SaveList           14. LoadList\n");
32         printf("**\t\t\t0. Exit\n");
33         printf("**\t\t\t15. ChooseList(请先进行此选项以选\n");
34         printf("**\t\t\t16. countSubarray        17.\n");
35         printf("**\t\t\t18. SortList\n");
36         printf("-----\n");
37         printf("请选择你的操作[0--18]:\n");
38         scanf("%d",&op);//选择op的值,用于switch
39         switch(op){
40             case 1:
41                 //第一种情况是初始化线性表
42                 if(IntiaList(L[i_num])==OK)
43                 {
44
45                     printf("请输入要保存的线性表名称\n");
46                     scanf("%s", filename);
```

```
47         printf("线性表创建成功\n");
48     }
49     else printf("线性表创建失败! \n");
50     getchar();getchar();
51     break;
52
53     case 2:
54         //第二种情况是用来销毁线性表
55         if(L[i_num].elem == NULL)
56         {
57             printf("线性表不存在!\n");
58             getchar();getchar();
59             break;
60         }
61         if(DestroyList(L[i_num])==OK)
62         {
63             printf("销毁线性表成功!\n");
64         }
65         else printf("销毁线性表失败! \n");
66         getchar();getchar();
67         break;
68
69     case 3:
70         //用于重置线性表
71         if(L[i_num].elem == NULL)
72         {
73             printf("线性表不存在!\n");
74             getchar();getchar();
75             break;
76         }
77         if(ClearList(L[i_num])==OK)
78         {
79             printf("线性表重置成功! \n");
80         }
81         else printf("线性表重置失败! \n");
```

```
82         getchar();getchar();
83         break;
84
85         case 4:
86             //判断是否为空
87             if(L[i_num].elem == NULL)
88             {
89                 printf("线性表不存在!\n");
90                 getchar();getchar();
91                 break;
92             }
93             if(ListEmpty(L[i_num])==TRUE)
94             {
95                 printf("文件为空! \n");
96             }
97             else printf("线性表不是空表! \n");
98             getchar();getchar();
99             break;
100
101         case 5:
102             //得到线性表长度
103             if(L[i_num].elem == NULL)
104             {
105                 printf("线性表不存在!\n");
106                 getchar();getchar();
107                 break;
108             }
109             printf("线性表表长为%d\n",ListLength(L[
                i_num]));
110             getchar();getchar();
111             break;
112
113         case 6:
114             //得到某个元素
115             if(L[i_num].elem == NULL)
```

```
116         {
117             printf("线性表不存在!\n");
118             getchar();getchar();
119             break;
120         }
121     printf("请输入要取结点的位置: \n");
122     scanf("%d",&i);
123     if(GetElem(L[i_num],i,&e)==OK)
124     printf("第%d个结点的元素是: %d\n",i,e);
125     else printf("输入位置错误! \n");
126     getchar();getchar();
127     break;
128
129     case 7:
130         //确定元素位置, 容易出错
131         if(L[i_num].elem == NULL)
132         {
133             printf("线性表不存在!\n");
134             getchar();getchar();
135             break;
136         }
137         printf("请输入数据元素值: \n");
138         scanf("%d",&e);
139         if(i=LocateElem(L[i_num],e,compare))
140         printf("%d元素位于第%d个位置! \n",e,i);
141         else printf("该元素不存在!\n");
142         getchar();getchar();
143         break;
144
145     case 8:
146         //求出前驱结点
147         if(L[i_num].elem == NULL)
148         {
149             printf("线性表不存在!\n");
150             getchar();getchar();
```

```
151             break;
152         }
153         printf("请输入数据元素: \n");
154         scanf("%d",&cur_e);
155         PriorElem(L[i_num],cur_e,&pre_e);
156         if(PriorElem(L[i_num],cur_e,&pre_e)==OK)
157             printf("其前驱元素为: %d\n",pre_e);
158         else if(PriorElem(L[i_num],cur_e,&pre_e)
159                 ==OVERFLOW)
160             printf("顺序表中没有该元素! \n");
161         else printf("其不存在前驱元素! \n");
162         getchar();getchar();
163         break;
164
165     case 9:
166         //求出后置节点
167         if(L[i_num].elem == NULL)
168         {
169             printf("线性表不存在!\n");
170             getchar();getchar();
171             break;
172         }
173         printf("请输入数据元素: \n");
174         scanf("%d",&cur_e);
175         if(NextElem(L[i_num],cur_e,&next_e)==OK)
176             printf("其后继元素为: %d\n",next_e);
177         else if(NextElem(L[i_num],cur_e,&pre_e)==
178                 FALSE)
179             printf("其不存在后继元素! \n");
180         else
181             {printf("顺序表中没有该元素! \n");}
182         getchar();getchar();
183         break;
184
185     case 10:
```

```
184         //插入元素
185         if(L[i_num].elem == NULL)
186         {
187             printf("线性表不存在!\n");
188             getchar();getchar();
189             break;
190         }
191         printf("请输入您要插入的数据元素: \n");
192         scanf("%d",&e);
193         printf("请输入您要插入的数据元素的位置: \n");
194         scanf("%d",&i);
195         if(ListInsert(L[i_num],i,e)==OK)
196         printf("插入数据元素成功! \n");
197         else
198         printf("插入数据元素失败! \n");
199         getchar();getchar();
200         break;
201
202         case 11:
203             //删除元素
204             if(L[i_num].elem == NULL)
205             {
206                 printf("线性表不存在!\n");
207                 getchar();getchar();
208                 break;
209             }
210             printf("请输入您要删除的数据元素的位置: \n");
211             scanf("%d",&i);
212             if(ListDelete(&L[i_num],i,&e)==OK)
213             printf("删除数据元素成功! \n");
214             else
215             printf("删除数据元素失败! \n");
216             getchar();getchar();
```

```
217         break;
218     case 12:
219         //遍历线性表中的元素
220         if(L[i_num].elem == NULL)
221         {
222             printf("线性表不存在!\n");
223             getchar();getchar();
224             break;
225         }
226         if(!ListTraverse(L[i_num])) printf("线性
           表是空表! \n");
227         getchar();getchar();
228         break;
229
230     case 13:
231         //保存文件
232         if(L[i_num].elem == NULL)
233         {
234             printf("线性表不存在!\n");
235             getchar();getchar();
236             break;
237         }
238         if(SaveList(L[i_num], filename)==OK)
239             printf("文件保存成功\n文件名为%s\n",
                filename);
240         break;
241
242     case 14:
243         //加载文件，需要输入需要加载的名称
244         printf("请输入要加载的文件名:\n ");
245         scanf("%s", filename);
246         if(LoadList(&L[i_num], filename)==OK)
247         {
248             printf("文件加载成功\n");
249         }
```

```
250         break;
251     case 15:
252         //选择在哪个表进行操作
253         printf("请输入要在第几个表操作:\n ");
254         printf("*只支持在%d个顺序表上操作*\n",
                MAX_NUM);
255         scanf("%d",&i_num);
256         printf("正在对第%d个表进行操作\n",i_num);
257         if((i_num<1)|| (i_num>10))
258         {
259             printf("请选择正确范围! \n");
260             i_num=1;
261         }
262         getchar(); getchar();
263         break;
264         break;
265     case 16:
266         //和为k的子数组个数
267         int ret;
268         int k;
269         printf("请输入k\n ");
270         scanf("%d",&k);
271         ret= countSubarray(L[i_num], k);
272         if(ret==ERROR){
273             printf("线性表不存在");
274             getchar();getchar();
275             break;
276         }
277         printf("和为%d的子数组个数为%d\n",k,ret);
278         getchar();getchar();
279         break;
280     case 17:
281         //最大连续子数组和
282         ret= maxSubArraySum(L[i_num]);
283         if(ret==ERROR){
```



```
284         printf("线性表不存在");
285         getchar();getchar();
286         break;
287     }
288     printf("最大连续子数组和为%d\n",ret);
289     getchar();getchar();
290     break;
291     case 18:
292         //线性表排序
293         ret=SortList(L[i_num]);
294         if(ret==ERROR){
295             printf("线性表不存在");
296             getchar();getchar();
297             break;
298         }
299         printf("已完成排序\n");
300         getchar();getchar();
301         break;
302     case 0:
303         //退出菜单，退出整个程序
304         break;
305     }//end of switch
306 }//end of while
307 printf("欢迎下次再使用本辣鸡系统!\n");
308 }//end of main()
309 /*-----page 23 on textbook -----*/
```

函数实现

```
1  /*****
2  *函数名称: IntiaList
3  *函数功能: 构造一个空的线性表
4  *注释: 初始条件是线性表L不存在已存在; 操作结果是构造一个空的线性
        表。
5  *返回值类型: status类型
6  *****/
```

```
7  status InitiaList(Sqlist &L){
8      L.elem = (ElemType *)malloc( LIST_INIT_SIZE * sizeof (
          ElemType));
9      if(!L.elem) exit(OVERFLOW);//如果空间不足，创建失败
10     L.length=0;
11     L.listsize=LIST_INIT_SIZE;
12     return OK;
13 }
14
15
16 /*****
17 *函数名称: DestoryList
18 *函数功能: 销毁线性表
19 *注释: 初始条件是线性表L已存在；操作结果是销毁线性表L
20 *返回值类型: status类型
21 *****/
22 status DestroyList(Sqlist &L)
23 {
24     if(L.elem)
25         free(L.elem);
26     L.elem = NULL;
27     L.length = 0;
28     L.listsize = 0;
29     return OK;
30 }
31
32
33 /*****
34 *函数名称: ClearList
35 *函数功能: 重置顺序表
36 *注释: 初始条件是线性表L已存在；操作结果是将L重置为空表。
37 *返回值类型: status类型
38 *****/
39 status ClearList(Sqlist &L)
40 {
```

```
41         L.length=0;
42         return OK;
43     }
44
45
46
47 /*****
48 *函数名称: ListEmpty
49 *函数功能: 判断线性表是否为空
50 *注释: 初始条件是线性表L已存在; 操作结果是若L为空表则返回TRUE, 否则返回FALSE。
51 *返回值类型: status类型
52 *****/
53 status ListEmpty(Sqlist L)
54 {
55     if(L.length==0)
56     {
57         return TRUE;
58     }
59     return FALSE;
60 }
61
62
63
64 /*****
65 *函数名称: ListLength
66 *函数功能: 求线性表的表长
67 *注释: 初始条件是线性表已存在; 操作结果是返回L中数据元素的个数。
68 *返回值类型: int类型
69 *****/
70 int ListLength(Sqlist L)
71 {
72     return L.length;
73 }
74
```

```
75
76 /*****
77 *函数名称: GetElem
78 *函数功能: 得到某一个元素的值
79 *注释: 初始条件是线性表已存在, 1 ≤ i ≤ ListLength(L); 操作结果是用e返回L中第i个数据元素的值
80 *返回值类型: status类型
81 *****/
82 status GetElem(SqList L, int i, ElemType *e)
83 {
84     if(i < 1 || i > L.length)
85     {
86         return ERROR;
87     }
88     *e = L.elem[i-1];
89     return OK;
90 }
91
92
93
94 /*****
95 *函数名称: LocateElem
96 *函数功能: 查找元素
97 *注释: 初始条件是线性表已存在; 操作结果是返回L中第1个与e满足关系
      compare ()
98 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。
99 *返回值类型: status类型
100 *****/
101 status LocateElem(SqList L, ElemType e, status (*compare)(ElemType,
      ElemType))
102 {
103     int i;
104     for(i=0; i<L.length; i++)
105     {
106         if(compare(L.elem[i], e))
```

```
107             return ++i;
108     }
109     return 0;
110 }
111
112
113 /*****
114 *函数名称: compare
115 *函数功能: 比较大小, 服务于LocateList函数
116 *注释: 输入两个ElemType类型的值
117 *返回值类型: status类型
118 *****/
119 status compare(ElemType a, ElemType b)
120 {
121     if(a == b)
122         return TRUE;
123     else return FALSE;
124 }
125
126
127
128
129 /*****
130 *函数名称: PriorElem
131 *函数功能: 求元素的前驱
132 *注释: 初始条件是线性表L已存在; 操作结果是若cur_e是L的数据元素,
        且不是第一个,
133 则用pre_e返回它的前驱, 否则操作失败, pre_e无定义。
134 *返回值类型: status类型
135 *****/
136 status PriorElem(SqList L, ElemType cur_e, ElemType *pre_e)
137 {
138     int i;
139     for(i=0; i<L.length; i++)
140     {
```

```
141         if(L.elem[i]==cur_e && i==0)
142         {
143             return ERROR;
144         }
145         else if(L.elem[i]== cur_e)
146         {
147             *pre_e = L.elem[i-1];
148             return OK;
149         }
150     }
151     return OVERFLOW;
152 }
153
154
155
156
157
158 /*****
159 *函数名称: NextElem
160 *函数功能: 求后继节点
161 *输入输出: 初始条件是线性表L已存在; 操作结果是若cur_e是L的数据元
           素, 且不是最后一个,
162 则用next_e返回它的后继, 否则操作失败, next_e无定义。
163 *返回值类型: status类型
164 *****/
165 status NextElem(SqList L, ElemType cur_e, ElemType *next_e)
166 {
167     int i;
168     for(i=0; i<(L.length-1); i++)
169     {
170         if(L.elem[i]==cur_e)
171         {
172             *next_e = L.elem[i+1];
173             return OK;
174         }
```

```
175
176     }
177     if(i==L.length-1 && (L.elem[i]!=cur_e)) return OVERFLOW;
178     else return FALSE;
179 }
180
181
182
183 /*****
184 *函数名称: ListInsert
185 *函数功能: 插入元素
186 *注释: 初始条件是线性表L已存在且非空, 1 ≤ i ≤ ListLength(L)+1;
187 *      操作结果是在L的第i个位置之前插入新的数据元素e
188 *返回值类型: status类型
189 *****/
190 status ListInsert(SqList &L,int i,ElemType e)
191 {
192     int *p,*q,*newbase;
193     if(i<1||i>L.length+1)
194     {
195         printf("插入位置不正确!\n");
196         return ERROR;
197     }
198
199     if(L.length>=L.listsize){
200         newbase = (ElemType *)realloc(L.elem,(L.listsize
201             + LISTINCREMENT)*sizeof(ElemType));
202         if(!newbase) exit(OVERFLOW);
203         L.elem = newbase;
204         L.listsize += LISTINCREMENT;
205     }
206     q = &(L.elem[i-1]);
207     for(p=&(L.elem[L.length-1]);p>=q;--p) *(p+1) = *p;
208     *q=e;
209     ++L.length;
```

```
209         return OK;
210
211     }
212     //这是课本上面的关于插入算法的实现
213
214
215
216
217
218     /*****
219     *函数名称: ListDelete
220     *函数功能: 删除元素
221     *注释: 初始条件是线性表L已存在且非空, 1 ≤ i ≤ ListLength(L);
222     *      操作结果: 删除L的第i个数据元素, 用e返回其值。
223     *返回值类型: status类型
224     *****/
225     status ListDelete(Sqlist *L, int i, ElemType *e)
226     {
227         if(i < 1 || i > L->length)
228             return ERROR; //删除的位数不正确
229         int j;
230         *e = L->elem[i-1];
231         for (j = i - 1; j < L->length; j++)
232             L->elem[j] = L->elem[j + 1];
233         L->length--;
234         return OK;
235     }
236
237
238
239
240     /*****
241     *函数名称: ListTraverse
242     *函数功能: 遍历顺序表
243     *注释: 输出顺序表的值
```



```
244 *返回值类型: status类型
245 *****/
246 status ListTraverse(Sqlist L){
247     int i;
248     printf("\n-----all elements
           -----\\n");
249     for(i=0;i<L.length;i++) printf("%d ",L.elem[i]);
250     printf("\n----- end
           -----\\n");
251     return L.length;
252 }
253
254
255
256
257
258
259 /*****
260 *函数名称: SaveList
261 *函数功能: 保存线性表
262 *注释: 将线性表保存, 参考附录B, 其中关于写入元素个数和长度的问题
        理解不够清楚
263 *返回值类型:
264 *****/
265 status SaveList(Sqlist L, char* filename)
266 {
267     int i = 0;
268     if ((fp = fopen(filename, "w")) == NULL)
269     {
270         printf("文件保存失败\\n");
271         return ERROR;
272     }
273     fprintf(fp, "%d ", L.length); //保存的时候, 也将L的长度保
        存到了文件
274     fprintf(fp, "%d ", L.listsize); //将每个元素的大小也保存到
```

了文件里

```
275     while (i < L.length)
276         fprintf(fp, "%d ", L.elem[i++]); //利用循环, 将元素依次存
           进去
277     fclose(fp); //关闭文件
278     return OK;
279 }
280
281
282
283
284
285
286
287  /*****
288  *函数名称: LoadList
289  *函数功能: 加载文件
290  *注释: 加载文件, 以便功能的测试, 文件名要正确
291  *返回值类型: status类型
292  *****/
293  status LoadList(Sqlist *L, char *filename)
294  {
295      int i = 0;
296      if ((fp = fopen(filename, "r")) == NULL)
297      {
298          printf("文件加载失败\n");
299          return ERROR;
300      }
301      fscanf(fp, "%d ", &L->length);
302      fscanf(fp, "%d ", &L->listsize);
303      L->elem = (ElemType *)malloc(L->listsize * sizeof(
           ElemType));
304      if (!L->elem) exit(OVERFLOW);
305      while (i < L->length)
306          fscanf(fp, "%d ", &L->elem[i++]); //利用循环, 依次读出文件
```

中的内容

```
307         fclose(fp);
308         return OK;
309     }
310
311
312     /*****
313     函数名称: countSubarray
314     函数功能: 计算线性表中和为k的子数组个数
315     参数: SqList L - 线性表 int k - 和值
316     返回值类型: int - 符合条件的子数组个数
317     *****/
318     int countSubarray(SqList L, int k) {
319         if(L.length==0){
320             return ERROR;
321         }
322         int count = 0;
323         int sum = 0;
324         for (int i = 1; i <= L.length; i++) { // 枚举子数组的起
            始位置
325             sum = 0;
326             for (int j = i; j <= L.length; j++) { // 枚举子
                数组的终止位置
327                 sum += L.elem[j-1];
328                 if (sum == k) {
329                     count++;
330                 }
331             }
332         }
333         return count;
334     }
335
336
337     /*****
338     函数名称: maxSubArraySum
```

339 函数功能：计算线性表的最大子序和

340 参数：SqList L - 线性表

341 返回值类型：int - 线性表的最大子序和

342 *****/

```
343 int maxSubArraySum(SqList L) {
344     if(L.length==0){
345         return ERROR;
346     }
347     int max_so_far = L.elem[0];
348     int curr_max = L.elem[0];
349     int n = L.length;
350     for (int i = 1; i < n; i++) {
351         curr_max = max(L.elem[i], curr_max + L.elem[i]);
352         max_so_far = max(max_so_far, curr_max);
353     }
354     return max_so_far;
355 }
```

356

357 *****/

358 函数名称：SortList

359 函数功能：对线性表进行选择排序

360 参数：SqList &L - 线性表的引用

361 返回值类型：int - 返回值为排序是否成功的标志，0为成功，ERROR为失败

362 *****/

```
363 int SortList(SqList &L){
364     if(L.length==0){
365         return ERROR;
366     }
367     int i, j, min;
368     ElemType temp;
369     for(i = 0; i < L.length - 1; i++){
370         min = i;
371         for(j = i + 1; j < L.length; j++){
372             if(L.elem[j] < L.elem[min]){
```

```
373             min = j;
374         }
375     }
376     if(min != i){
377         temp = L.elem[min];
378         L.elem[min] = L.elem[i];
379         L.elem[i] = temp;
380     }
381 }
382 }
383 /*****
384 函数名称: max
385 函数功能: 返回两个整数中的最大值
386 参数: int a - 整数a
387 int b - 整数b
388 返回值类型: int - 返回a和b中的最大值
389 *****/
390 int max(int a, int b) {
391     return (a > b) ? a : b;
392 }
```

6 附录 B 基于链式存储结构线性表实现的源程序

相关定义

```
1  /* Linear Table On Sequence Structure */
2  #include <stdio.h>
3  #include <malloc.h>
4  #include <stdlib.h>
5
6  /*-----page 10 on textbook -----*/
7  #define TRUE 1//定义真值
8  #define FALSE 0//定制假值
9  #define OK 1//程序正常运行
10 #define ERROR 0//程序运行出错
11 #define INFEASTABLE -1 //输入或输出不合法
12 #define OVERFLOW -2 //数值溢出
13 #define MAX_NUM 10 //可管理线性表的数量
14
15 typedef int status; //定义所有状态码和返回值的类型为int
16 typedef int ElemType; //数据元素类型定义
17
18 /*-----page 22 on textbook -----*/
19 #define LIST_INIT_SIZE 100 //线性表的初始存储空间大小（数组长
    度）
20 #define LISTINCREMENT 10 //线性表存储空间不足时增加的存储空间量
21 typedef struct LNode{ //定义单链表节点结构体类型
22     ElemType data; //节点中存储的数据元素
23     struct LNode *next; //指向下一个节点的指针
24 }LNode, *LinkList;
25
26 FILE *fp; //文件指针，用于数据存储和读取
27
28 /*-----函数声明-----*/
29 status InitList(LinkList *L);
30 //初始化线性表
```

```
31 status DestroyList(LinkList *L);
32 //销毁线性表
33 status ClearList(LinkList *L);
34 //清空线性表
35 status ListEmpty(LinkList L);
36 //判断线性表是否为空
37 int ListLength(LinkList L);
38 //获取线性表长度
39 status GetElem(LinkList L, int i, ElemType *e);
40 //获取线性表中指定位置的数据元素
41 int LocateElem(LinkList L, ElemType e, status(*compare)(ElemType
    a, ElemType b));
42 //查找指定数据元素在线性表中第一次出现的位置
43 status compare(ElemType a, ElemType b);
44 //比较两个数据元素的大小
45 status PriorElem(LinkList L, ElemType cur_e, ElemType *pre_e);
46 //获取指定数据元素的前驱节点
47 status NextElem(LinkList L, ElemType cur_e, ElemType *next_e);
48 //获取指定数据元素的后继节点
49 status ListInsert(LinkList *L, int i, ElemType e);
50 //在线性表中指定位置插入一个数据元素
51 status ListDelete(LinkList *L, int i, ElemType *e);
52 //从线性表中删除指定位置的数据元素，并把它通过参数e返回
53 status ListTraverse(LinkList L);
54 //依次访问线性表中的每个元素，并输出到控制台
55 status SaveList(LinkList L, char* filename);
56 //将线性表中的数据元素以文本文件的形式存储到本地磁盘中
57 status LoadList(LinkList *L, char *filename);
58 //从本地磁盘中的文本文件中读取数据元素，并创建一个新的线性表
59 status ReverseList(LinkList &L);
60 //翻转链表
61 status RemoveNthFromEnd(LinkList &L, int n);
62 //删除倒数第n个节点
63 status SortList(LinkList L);
64 //对链表进行排序
```

演示系统

```
1  int main(){
2      char filename[40];
3      int op=1;
4      int i,i_num=1;
5      LinkList L[MAX_NUM];
6      for (i = 0; i<MAX_NUM; i++)
7      {
8          L[i]=NULL;
9      }
10     ElemType e, cur_e, pre_e, next_e;
11     while(op){
12         system("cls");
13         printf("\n\n");
14         printf("          \t\t\tMenu for Linear Table On
15                Sequence Structure \n");
16         printf("    可在%d个顺序表进行多表操作，初始化请先
17                操作功能15,默认在第一个表上操作\n", MAX_NUM);
18         printf("
19                -----
20                n");
21         printf("**\t\t\t1. InitList          7. LocateElem\
22                \t\t\t**\n");
23         printf("**\t\t\t2. DestroyList       8. PriorElem\
24                \t\t\t**\n");
25         printf("**\t\t\t3. ClearList        9. NextElem \
26                \t\t\t**\n");
27         printf("**\t\t\t4. ListEmpty       10. ListInsert
28                \t\t\t**\n");
29         printf("**\t\t\t5. ListLength     11. ListDelete
30                \t\t\t**\n");
31         printf("**\t\t\t6. GetElem        12.
32                ListTraverse\t\t\t**\n");
33         printf("**\t\t\t13.SaveList       14. LoadList\
34                \t\t\t**\n");
```



```
24         printf("**\t\t\t 0.Exit\n");
25         printf("**\t\t\t15.ChooseList(请先进行此选项以选\n");
26         printf("**\t\t\t16.ReverseList    17.\n");
27         printf("**\t\t\t18.SortList\n");
28         printf("-----\n");
29         printf("请选择你的操作[0--18]:\n");
30         scanf("%d",&op);
31         switch(op)
32         {
33             case 1:
34                 //第一种情况是初始化线性表
35                 if(InitList(&L[i_num])==OK)
36                 {
37
38                     printf("请输入要保存的线性表名称\n");
39                     scanf("%s", filename);
40                     printf("线性表创建成功\n");
41                 }
42                 else printf("线性表创建失败! \n");
43                 getchar();getchar();
44                 break;
45
46             case 2:
47                 //第二种情况是用来销毁线性表
48                 if(L[i_num] == NULL)
49                 {
50                     printf("线性表不存在!\n");
51                     getchar();getchar();
52                     break;
```

```
53         }
54         if(DestroyList(&L[i_num])==OK)
55         {
56             printf("销毁线性表成功!\n");
57         }
58         else printf("销毁线性表失败! \n");
59         getchar();getchar();
60         break;
61
62         case 3:
63             //用于重置线性表
64             if(L[i_num] == NULL)
65             {
66                 printf("线性表不存在!\n");
67                 getchar();getchar();
68                 break;
69             }
70             if(ClearList(&L[i_num])==OK)
71             {
72                 printf("线性表重置成功! \n");
73             }
74             else printf("线性表重置失败! \n");
75             getchar();getchar();
76             break;
77
78         case 4:
79             //判断是否为空
80             if(L[i_num] == NULL)
81             {
82                 printf("线性表不存在!\n");
83                 getchar();getchar();
84                 break;
85             }
86             if(ListEmpty(L[i_num])==TRUE)
87             {
```

```
88         printf("文件为空! \n");
89     }
90     else printf("线性表不是空表! \n");
91     getchar();getchar();
92     break;
93
94     case 5:
95         //得到线性表长度
96         if(L[i_num] == NULL)
97         {
98             printf("线性表不存在!\n");
99             getchar();getchar();
100             break;
101         }
102         printf("线性表表长为%d\n",ListLength(L[
            i_num]));
103         getchar();getchar();
104         break;
105
106     case 6:
107         //得到某个元素
108         if(L[i_num] == NULL)
109         {
110             printf("线性表不存在!\n");
111             getchar();getchar();
112             break;
113         }
114         printf("请输入要取结点的位置: \n");
115         scanf("%d",&i);
116         if(GetElem(L[i_num],i,&e)==OK)
117             printf("第%d个结点的元素是: %d\n",i,e);
118         else printf("输入位置错误! \n");
119         getchar();getchar();
120         break;
121
```

```
122         case 7:
123             //printf("\n----LocateElem功能待实现! \n
                ");
124             if(L[i_num] == NULL)
125             {
126                 printf("线性表不存在!\n");
127                 getchar();getchar();
128                 break;
129             }
130             printf("请输入数据元素值: \n");
131             scanf("%d",&e);
132             if(i=LocateElem(L[i_num],e,compare))
133                 printf("%d元素位于第%d个位置! \n",e,i);
134             else printf("该元素不存在!\n");
135             getchar();getchar();
136             break;
137
138         case 8:
139             //求出前驱结点
140             if(L[i_num] == NULL)
141             {
142                 printf("线性表不存在!\n");
143                 getchar();getchar();
144                 break;
145             }
146             printf("请输入数据元素: \n");
147             scanf("%d",&cur_e);
148             PriorElem(L[i_num],cur_e,&pre_e);
149             if(PriorElem(L[i_num],cur_e,&pre_e)==OK)
150                 printf("其前驱元素为: %d\n",pre_e);
151             else if(PriorElem(L[i_num],cur_e,&pre_e)
                ==OVERFLOW)
152                 printf("顺序表中没有该元素! \n");
153             else printf("其不存在前驱元素! \n");
154             getchar();getchar();
```

```
155         break;
156
157
158     case 9:
159         //求出后置节点
160         if(L[i_num] == NULL)
161         {
162             printf("线性表不存在!\n");
163             getchar();getchar();
164             break;
165         }
166         printf("请输入数据元素: \n");
167         scanf("%d",&cur_e);
168         if(NextElem(L[i_num],cur_e,&next_e)==OK)
169             printf("其后继元素为: %d\n",next_e);
170         else if(NextElem(L[i_num],cur_e,&pre_e)==
171             ERROR)
172             printf("顺序表中没有该元素! \n");
173         else
174             {printf("其不存在后继元素! \n");}
175         getchar();getchar();
176         break;
177
178     case 10:
179         //插入元素
180         if(L[i_num] == NULL)
181         {
182             printf("线性表不存在!\n");
183             getchar();getchar();
184             break;
185         }
186         printf("请输入您要插入的数据元素: \n");
187         scanf("%d",&e);
188         printf("请输入您要插入的数据元素的位置: \n");
```

```
188         scanf("%d",&i);
189         if(ListInsert(&L[i_num],i,e)==OK)
190             printf("插入数据元素成功! \n");
191         else
192             printf("插入数据元素失败! \n");
193         getchar();getchar();
194         break;
195
196     case 11:
197         //删除元素
198         if(L[i_num] == NULL)
199         {
200             printf("线性表不存在!\n");
201             getchar();getchar();
202             break;
203         }
204         printf("请输入您要删除的数据元素的位置: \n");
205         scanf("%d",&i);
206         if(ListDelete(&L[i_num],i,&e)==OK)
207             printf("删除数据元素成功! \n");
208         else
209             printf("删除数据元素失败! \n");
210         getchar();getchar();
211         break;
212
213     case 12:
214         //遍历线性表中的元素
215         if(L[i_num] == NULL)
216         {
217             printf("线性表不存在!\n");
218             getchar();getchar();
219             break;
220         }
221         if(!ListTraverse(L[i_num])) printf("线性
```

```
                表是空表!\n");
222         getchar();getchar();
223         break;
224
225         case 13:
226             //保存文件
227             if(L[i_num] == NULL)
228             {
229                 printf("线性表不存在!\n");
230                 getchar();getchar();
231                 break;
232             }
233             if(SaveList(L[i_num], filename)==OK)
234             printf("文件保存成功!文件名为%s\n",
                filename);
235             getchar();getchar();
236             break;
237
238         case 14:
239             //加载文件, 需要输入需要加载的名称
240             InitList(&L[i_num]);
241             printf("请输入要加载的文件名:\n ");
242             scanf("%s", filename);
243             if(LoadList(&L[i_num], filename)==OK)
244             {
245                 printf("文件加载成功\n");
246             }
247             getchar();getchar();
248             break;
249
250         case 15:
251             //选择在哪个表进行操作
252             printf("请输入要在第几个表操作:\n ");
253             printf("*只支持在%d个顺序表上操作*\n",
                MAX_NUM);
```

```
254         scanf("%d",&i_num);
255         printf("正在对第%d个表进行操作\n",i_num);
256         if((i_num<1)|| (i_num>10))
257         {
258             printf("超出选择范围\n ");
259             i_num=1;
260         }
261         getchar(); getchar();
262         break;
263         case 16:
264             //链表翻转
265             int ret;
266             ret= ReverseList(L[i_num]);
267             if(ret==ERROR){
268                 printf("线性表不存在!\n");
269                 getchar(); getchar();
270                 break;
271             }
272             printf("线性表已翻转!\n");
273             getchar(); getchar();
274             break;
275
276         case 17:
277             //删除链表的倒数第n个结点
278             int n;
279             printf("请输入要删除倒数第n个结点\n ");
280             scanf("%d",&n);
281             ret=RemoveNthFromEnd(L[i_num],n);
282             if(ret==ERROR){
283                 printf("线性表不存在!\n");
284                 getchar(); getchar();
285                 break;
286             }
287             printf("结点已删除!\n");
288             getchar(); getchar();
```



```
289             break;
290             case 18:
291                 // 链表排序
292                 ret=SortList(L[i_num]);
293                 if(ret==ERROR){
294                     printf("线性表不存在!\n");
295                     getchar(); getchar();
296                     break;
297                 }
298                 printf("排序已完成!\n");
299                 getchar(); getchar();
300                 break;
301
302             case 0:
303                 break;
304         } //end of switch
305     } //end of while
306     printf("\t\t欢迎下次再使用本系统! \n");
307 } //end of main()
308 /*-----page 23 on textbook -----*/
```

函数实现

```
1  /*****
2  *函数名称: InitList
3  *函数功能: 构造一个空的线性表
4  *注释: 初始条件是线性表L不存在已存在; 操作结果是构造一个空的线性
      表。
5  *返回值类型: status类型
6  *****/
7  status InitList(LinkList *L)
8  {
9      *L = (LinkList)malloc(sizeof(LNode)); //动态分配
10     if(*L == NULL)
11     {
12         exit(OVERFLOW); //如果没有足够的空间, 创建失败
```

```
13     }
14     (*L)->data = 0;
15     (*L)->next = NULL; //创建带有表头节点的链表，表头节点的数据域值为0
16     return OK;
17 }
18
19 /*****
20 *函数名称: DestroyList
21 *函数功能: 销毁线性表
22 *注释: 初始条件是线性表L已存在；操作结果是销毁线性表L
23 *返回值类型: status类型
24 *****/
25 status DestroyList(LinkList *L)
26 {
27     LinkList p, q; //指针p,q
28     p = *L; //将指针p指向表头节点
29     while(p)
30     {
31         q = p->next; //如果p不指向空，将q指向p的下一个节点
32         free(p); //然后释放p
33         p = q; //再将q所指向的节点赋给p
34     }
35     *L = NULL; //最后指针L指向空
36     return OK;
37 }
38
39 /*****
40 *函数名称: ClearList
41 *函数功能: 重置顺序表
42 *注释: 初始条件是线性表L已存在；操作结果是将L重置为空表。
43 *返回值类型: status类型
44 *****/
45 status ClearList(LinkList *L)
46 {
```

```
47     LinkList p, q; //创建两个指针p,q
48     p = (*L)->next; //将p指向第一个节点
49     while(p)
50     {
51         q = p->next; //当p不指向空时, q指向p的下一个节点
52         free(p); //释放p
53         p = q; //将q指向的节点赋给p
54     }
55     (*L)->next = NULL; //最后, 将表头节点的指针域指向空
56     return OK;
57 }
58
59 /*****
60 *函数名称: ListEmpty
61 *函数功能: 判断线性表是否为空
62 *注释: 初始条件是线性表L已存在; 操作结果是若L为空表则返回TRUE, 否则返回FALSE。
63 *返回值类型: status类型
64 *****/
65 status ListEmpty(LinkList L)
66 {
67     if(L->next)
68     {
69         return FALSE;
70     }
71     return TRUE; //如果表头节点的指针域指向空, 那么返回TRUE
72 }
73
74 /*****
75 *函数名称: ListLength
76 *函数功能: 求线性表的表长
77 *注释: 初始条件是线性表已存在; 操作结果是返回L中数据元素的个数。
78 *返回值类型: int类型
79 *****/
80 int ListLength(LinkList L)
```

```
81 {
82     int i = 0; // i用来统计次数，即表长
83     LinkList p = L->next; // 先将p指向表头节点的后一个节点，即
        第一个节点
84     while(p)
85     {
86         i++;
87         p = p->next; // 如果p指向不为空，i的次数加一，p指向
            下一个节点
88     }
89     return i; // 返回次数i，即表长
90 }
91
92
93 /*****
94 *函数名称：GetElem
95 *函数功能：得到某一个元素的值
96 *注释：初始条件是线性表已存在，1 ≤ i ≤ ListLength(L)；操作结果是用e返
        回L中第i个数据元素的值
97 *返回值类型：status类型
98 *****/
99 status GetElem(LinkList L, int i, ElemType *e)
100 {
101     int j = 1;
102     LinkList p;
103     p = L->next; // p指向表头节点后的第一个节点
104     while(p && j < i)
105     {
106         p = p->next; // 循环用来找到i位置节点
107         ++j;
108     }
109     if(!p || j > i)
110     {
111         return ERROR; // 用来判断输入位置是否正确，空表等
112     }
```

```
113         *e = p->data; //用e取节点的元素
114         return OK;
115     }
116
117     /*****
118     *函数名称: LocateElem
119     *函数功能: 查找元素
120     *注释: 初始条件是线性表已存在; 操作结果是返回L中第1个与e满足关系
121           compare ()
122           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。
123     *返回值类型: status类型
124     *****/
125     int LocateElem(LinkList L, ElemType e, status(*compare)(ElemType
126         a, ElemType b))
127     {
128         int i = 0;
129         LinkList p = L->next; //p指向第一个节点
130         while(p)
131         {
132             i++;
133             if((*compare)(p->data, e)) //通过遍历法比较得到所
134                 要找的元素
135             return i; //此时, 找到了元素所在位置
136             p = p->next; //没有找到时, p指向下一个节点, 循环
137         }
138         return 0;
139     }
140
141     /*****
142     *函数名称: compare
143     *函数功能: 比较大小, 服务于LocateList函数
144     *注释: 输入两个ElemType类型的值
145     *返回值类型: status类型
146     *****/
147     status compare(ElemType a, ElemType b)
```

```
145 {
146     if(a == b)
147         return TRUE;//比较输入的两个元素的大小，一样大则为TRUE
148     else
149         return FALSE;
150 }
151
152 /*****
153 *函数名称: PriorElem
154 *函数功能: 求元素的前驱
155 *注释: 初始条件是线性表L已存在；操作结果是若cur_e是L的数据元素，
        且不是第一个，
156 则用pre_e返回它的前驱，否则操作失败，pre_e无定义。
157 *返回值类型: status类型
158 *****/
159 status PriorElem(LinkList L, ElemType cur_e, ElemType *pre_e)
160 {
161     LinkList p = L->next;//p指向第一个节点
162     if(p->data==cur_e) return ERROR;//如果第一个节点就是要找
        的元素，则没有前驱
163     while(p->next != NULL && p->next->data != cur_e)
164     {
165         p = p->next;//通过循环，将p指针指向所要找的元素的前一个节点
166     }
167     if(p->next == NULL)//如果此时p指针指向空，则意味着表中没有该元素
168         return OVERFLOW;
169
170     *pre_e = p->data;//用pre_e取出p指向的节点的元素，即输入元素的前驱
171     return OK;
172 }
173
174 /*****
```

```
175 *函数名称: NextElem
176 *函数功能: 求后继节点
177 *输入输出: 初始条件是线性表L已存在; 操作结果是若cur_e是L的数据元
           素, 且不是最后一个,
178 则用next_e返回它的后继, 否则操作失败, next_e无定义。
179 *返回值类型: status类型
180 *****/
181 status NextElem(LinkList L, ElemType cur_e, ElemType *next_e)
182 {
183     LinkList p = L->next; //p指向第一个节点
184     while(p->next != NULL && p->data != cur_e)
185     {
186         p = p->next; //循环的方式找到所要找的元素的前一个
           节点
187     }
188     if(p->next == NULL && p->data != cur_e) //此时p指针指向
           空, p指向节点的值不是输入的元素, 那么没有输入的元素
189     return ERROR;
190     if(p->next == NULL && p->data == cur_e) //此时p指针指向
           空, p指向节点的值是输入的元素, 那么没有后继节点
191     return OVERFLOW;
192     *next_e = p->next->data; //剩余正常情况, 用next_e取出p所指
           向节点的下一个节点的值
193     return OK;
194 }
195
196 *****/
197 *函数名称: ListInsert
198 *函数功能: 插入元素
199 *注释: 初始条件是线性表L已存在且非空, 1 ≤ i ≤ ListLength(L)+1;
200 *      操作结果是在L的第i个位置之前插入新的数据元素e
201 *返回值类型: status类型
202 *****/
203 status ListInsert(LinkList *L, int i, ElemType e)
204 {
```

```
205     int j = 1;
206     LinkList p, q; //用两个指针确定插入位置的前后节点
207     p = *L; //p指向表头节点（考虑到表头插入）
208     while(p && j < i)
209     {
210         p = p->next; //找到插入位置
211         ++j;
212     }
213     if(!p || j > i)
214     {
215         return ERROR;
216     }
217     q = (LinkList)malloc(sizeof(LNode)); //给插入元素分配空间
218     if(q == NULL)
219     {
220         exit(OVERFLOW); //没分配成功
221     }
222     q->data = e;
223     q->next = p->next; //这三行代码是用来将元素存进去，同时将
        指针指向问题解决
224     p->next = q;
225     return OK;
226 }
227
228 /*****
229 *函数名称: ListDelete
230 *函数功能: 删除元素
231 *注释: 初始条件是线性表L已存在且非空, 1 ≤ i ≤ ListLength(L);
232 *      操作结果: 删除L的第i个数据元素, 用e返回其值。
233 *返回值类型: status类型
234 *****/
235 status ListDelete(LinkList *L, int i, ElemType *e)
236 {
237     int j = 1;
238     LinkList p, q;
```



```
239     p = *L; //p指向表头节点
240     while(p->next && j < i)
241     {
242         p = p->next; //找到删除位置，同时p的下一个节点不为
                空
243         ++j;
244     }
245     if(!(p->next) || j>i) //此时，根据p指针的情况 或者 j 的值
                判断
246     return ERROR; //即要么是空表，要么是删除位置不对。两者均可
                认为是删除位置不对
247
248     q = p->next;
249     p->next = q->next; //这四行代码是删除节点，并用e取出删除节
                点的值
250     *e = q->data;
251     free(q);
252
253     return OK;
254 }
255
256 /*****
257 *函数名称: ListTraverse
258 *函数功能: 遍历顺序表
259 *注释: 输出顺序表的值
260 *返回值类型: status类型
261 *****/
262 status ListTraverse(LinkList L)
263 {
264     LinkList p = L->next;
265     if(!p) //此时为空表
266     return ERROR;
267     printf("\n----- all elements
                -----\n");
268     while(p)
```

```
269         {
270             printf("%d ",p->data); //循环, 输出值
271             p = p->next;
272         }
273         printf("\n----- end
                ----- \n");
274         return OK;
275     }
276
277     /*****
278     *函数名称: SaveList
279     *函数功能: 保存线性表
280     *注释: 将线性表保存, 参考附录B, 其中关于写入元素个数和长度的问题
            理解不够清楚
281     *返回值类型:
282     *****/
283     status SaveList(LinkList L, char* filename)
284     {
285         LinkList p = L->next; //指针指向第一个节点
286         int listsize=LIST_INIT_SIZE; //最大表长
287         if ((fp = fopen(filename, "w")) == NULL)
288         {
289             printf("\t\t\t文件保存失败!"); //文件保存失败
290             return ERROR;
291         }
292         fprintf(fp, "%d ", ListLength(L)); //将元素个数写入
293         fprintf(fp, "%d ", listsize); //将最大长度写入
294         while(p)
295         {
296             fprintf(fp, "%d ", p->data); //指针op不指向空, 依
                次写入文件
297             p = p->next;
298         }
299         fclose(fp);
300         return OK;
```

```
301 }
302
303 /*****
304 *函数名称: LoadList
305 *函数功能: 加载文件
306 *注释: 加载文件, 以便功能的测试, 文件名要正确
307 *返回值类型: status类型
308 *****/
309 status LoadList(LinkList *L, char *filename)
310 {
311     int i = 1, length = 0, listsize;
312     ElemType e;
313     if ((fp = fopen(filename, "r")) == NULL)
314     {
315         printf("文件加载失败!");
316         return ERROR;
317     }
318     fscanf(fp, "%d ", &length);
319     fscanf(fp, "%d ", &listsize);
320     fscanf(fp, "%d ", &e);
321     while(i<=length)
322     {
323         ListInsert(L,i,e);
324         fscanf(fp, "%d ", &e);
325         i++;
326     }
327     fclose(fp);
328     return OK;
329 }
330
331 /*****
332 * 函数名称: ReverseList
333 * 函数功能: 翻转单链表
334 * 函数参数: 指向待翻转单链表的头结点的指针L的引用
335 * 注释: 如果链表为空或者只有一个元素, 则直接返回; 否则, 将链表中
```

的每个元素的指针方向翻转，使链表翻转。

```
336 * 返回值类型: status类型, 函数执行成功返回OK, 否则返回ERROR。
337 *****/
338 status ReverseList(LinkList &L) {
339     LNode*pr=L;
340     LNode*trans=NULL;
341     LNode*re=NULL;
342     if(L==NULL){return ERROR;}//链表不存在。
343     if(L->next==NULL||L->next->next==NULL){return OK;}//空链
        表或只含有一个元素的链表。
344     trans=L->next;
345     re=L->next->next;
346     while(trans!=NULL)
347     {
348         if(pr!=L){trans->next=pr;}//改变指针方向。
349         else{trans->next=NULL;}//第一个元素的next指向NULL
            。
350         pr=trans;//移动到下一个元素。
351         trans=re;
352         if(re!=NULL){re=re->next;}
353     }
354     L->next=pr;
355     return OK;
356 }
357 /*****
358 * 函数名称: RemoveNthFromEnd
359 * 函数功能: 删除单链表中倒数第n个节点
360 * 函数参数: 指向待删除节点所在的单链表L的指针的引用, 待删除节点的
        位置n
361 * 注释: 如果单链表为空, 则直接返回; 否则, 先遍历单链表获取单链表的
        长度len, 如果n超过len的范围, 则直接返回; 否则, 再次遍历单链
        表, 找到倒数第n个节点, 将其删除。
362 * 返回值类型: status类型, 函数执行成功返回OK, 否则返回ERROR。
363 *****/
364 status RemoveNthFromEnd(LinkList &L, int n)
```

```
365 {
366     int len=0;
367     LNode*trans=NULL;
368     LNode*locate=L;
369     if(L==NULL){return ERROR;}//空表。
370     trans=L->next;
371     while(trans!=NULL)
372     {
373         trans=trans->next;
374         len++;
375     }
376     if(n>len){return ERROR;}//索引越界。
377     while(len>n)//寻找元素。
378     {
379         len--;
380         locate=locate->next;
381     }
382     LNode*p=locate->next;//删除。
383     locate->next=p->next;
384     free(p);
385     return OK;
386 }
387
388 /*****
389 * 函数名称: SortList
390 * 函数功能: 对单链表进行升序排列
391 * 函数参数: 指向待排序单链表的头结点的指针L
392 * 注释: 如果链表为空, 则直接返回; 否则, 使用冒泡排序法将链表中的
      元素从小到大排序。
393 * 返回值类型: status类型, 函数执行成功返回OK, 否则返回ERROR。
394 *****/
395 status SortList(LinkList L) {
396     if(L==NULL){return ERROR;}//空表
397     int len = ListLength(L);
398     LNode *p, *q;
```

```
399     ElemType temp;
400     for (int i = 1; i < len; i++) {
401         p = L->next;
402         q = p->next;
403         for (int j = 1; j < len - i + 1; j++) {
404             if (p->data > q->data) {
405                 temp = p->data;
406                 p->data = q->data;
407                 q->data = temp;
408             }
409             p = q;
410             q = q->next;
411         }
412     }
413     return OK;
414 }
```

7 附录 C 基于二叉链表二叉树实现的源程序

相关定义

```
1  /* Linear Table On Sequence Structure */
2  #include <stdio.h>
3  #include <malloc.h>
4  #include <stdlib.h>
5  #include <stack>
6  #include <queue>
7  #include <stdlib.h>
8
9  // 定义每个节点的打印宽度为 5
10 #define NODE_WIDTH 5
11
12 // 定义每个节点的占位符为 _
13 #define NODE_PLACEHOLDER '_'
14
15 /*-----page 10 on textbook -----*/
16 #define TRUE 1//定义真值
17 #define FALSE 0//定制假值
18 #define OK 1//程序正常运行
19 #define ERROR 0//程序运行出错
20 #define INFESTABLE -1//输入或输出不合法
21 #define OVERFLOW -2//数值溢出
22 #define MAX_NUM 10//最大个数
23 #define LIST_INIT_SIZE 100//最大尺寸
24 #define LISTINCRE MENT 10//最大增加量
25
26 typedef int status;//定义所有状态码和返回值的类型为int
27 typedef char TElemType;//数据元素类型定义
28 status definition[1000]={0}; //定义并初始化关键字哈希表
29
30 // 定义二叉树的节点结构体
31 typedef struct BiTNode{
```

```
32         int key; // 用 key 作为标记, 便于查找节点
33         TElemType data; // char 类型数据域
34         struct BiTNode *lchild, *rchild; // 定义二叉链表的左孩子
           指针与右孩子指针
35 } BiTNode, *BiTree; // BiTNode 类型指针 BiTree
36
37 // 定义一个结构体, 用于保存二叉树和树的名称
38 typedef struct {
39     BiTree T; // 创建二叉树用的指针 T
40     char name[20]; // 用于保存树的名称
41 } LElemType;
42
43 // 定义一个结构体, 用于保存多个树进行操作
44 typedef struct {
45     LElemType tree[20]; // 多个树进行操作
46     int length;
47     int listsize;
48 } SqList;
49 /*-----函数声明-----*/
50 status InitBiTree(BiTree *T);
51 // 初始化二叉树
52 status DestroyBiTree(BiTree *T);
53 // 销毁二叉树
54 status CreateBiTree(BiTree *T);
55 // 创建二叉树
56 status ClearBiTree(BiTree T);
57 // 清空二叉树
58 status BiTreeEmpty(BiTree T);
59 // 判断二叉树是否为空
60 status BiTreeDepth(BiTree T);
61 // 计算二叉树的深度
62 char Root(BiTree T);
63 // 获取二叉树的根节点
64 char Value(BiTree T, int e);
65 // 获取二叉树中指定节点的值
```



```
66 status Assign(BiTree T, int e1, int e2, char ch);
67 // 给二叉树中指定节点赋值
68 BiTNode* Parent(BiTree T, int e);
69 // 获取二叉树中指定节点的父节点
70 BiTNode* LeftChild(BiTree T, int e);
71 // 获取二叉树中指定节点的左孩子节点
72 BiTNode* RightChild(BiTree T, int e);
73 // 获取二叉树中指定节点的右孩子节点
74 BiTNode* RightSibling(BiTree T, int e);
75 // 获取二叉树中指定节点的右兄弟节点
76 BiTNode* LeftSibling(BiTree T, int e);
77 // 获取二叉树中指定节点的左兄弟节点
78 BiTree Find(BiTree T, int key);
79 // 查找二叉树中指定 key 值的节点
80 status InsertChild(BiTree T, BiTree p, int LR, BiTree c);
81 // 在二叉树中指定节点的左/右侧插入子树
82 status DeleteChild(BiTree T, BiTree p, int LR);
83 // 删除二叉树中指定节点的左/右子树
84 status visit(char e);
85 // 访问二叉树节点
86 status PreOrderTraverse(BiTree T, status (* visit)(char e));
87 // 先序遍历二叉树
88 status InOrderTraverse(BiTree T, status (* visit)(char e));
89 // 中序遍历二叉树
90 status PostOrderTraverse(BiTree T, status (* visit)(char e));
91 // 后序遍历二叉树
92 status LevelOrderTraverse(BiTree T, status (* visit)(char e));
93 // 层序遍历二叉树
94 status TreeDisplay(BiTree T, int depth, status (* visit)(char e));
95 // 打印二叉树
96 status Save(BiTree T, FILE *fp);
97 // 将二叉树保存到文件中
98 status Load(BiTree *T, FILE *fp);
99 // 从文件中加载二叉树
100 BiTNode* LowestCommonAncestor(BiTree T, int e1, int e2);
```

```
101 // 查找二叉树中指定节点的最近公共祖先
102 BiTNode* LocateNode(BiTree T, int e);
103 // 查找二叉树中指定 key 值的节点
104 status InvertTree(BiTree &T);
105 // 翻转二叉树
106 status MaxPathSum(BiTree T);
107 // 计算二叉树中根节点到叶子节点的最大路径和
```

演示系统

```
1
2 int main(){
3     FILE *fp;
4     char filename[30];
5     SqList L;//相当于用数组构建多树操作的框架
6     BiTree T1, T2;//两个用BiTree构建的BiTNode指针
7     int op=1, key;//op用来case,便于用户操作;key是用来标记节点
8     int i, num=1, LR;//num默认为1,即默认在第一棵树进行操作,
        LR用来表示方向
9     char ch;//用来接收data域的值
10    for(i=0;i<20;i++)
11    {
12        L.tree[i].T = NULL;//相当于创建20个定义中类型的树的
            指针,同时指空,即20个树;
13    }
14
15    while(op){
16        system("cls");//系统函数,用于清屏
17        system("color F");
18        printf("\n\n");
19        printf("\t\t\t\t\t Menu for Binary Tree On Binary
            Linked List \n");
20        printf("\t\t\t\t\t *****
            n");
21        printf("\t\t\t\t\t 1. InitBiTree\t\t\t\t\t 2.
```

```
        DestroyBiTree\n");
22     printf("\t\t\t3.  CreateBiTree      \t  4.
        ClearBiTree\n");
23     printf("\t\t\t5.  BiTreeEmpty      \t  6.
        BiTreeDepth\n");
24     printf("\t\t\t7.  Assign          \t  8.
        GetSibling\n");
25     printf("\t\t\t9.  InsertChild      \t  10.
        DeleteChild\n");
26     printf("\t\t\t11. PreOrderTraverse      12.
        InOrderTraverse\n");
27     printf("\t\t\t13. PostOrderTraverse      14.
        LevelOrderTraverse\n");
28     printf("\t\t\t15. Choose(多树操作)      16. Save(
        保存文件)\n");
29     printf("\t\t\t17. Load(加载文件)      18.
        LowestCommonAncestor\n");
30     printf("\t\t\t19. TreeDisplay          20.
        InvertTree\n");
31     printf("\t\t\t21. MaxPathSum          \t  22.
        LocateNode 0.  Exit\n");
32     printf("\t*****Powered By
        @_@||lbw*****\n"
        );
33     printf("\t\t\t请选择你的操作[0~22]:");
34     scanf("%d",&op);
35     switch(op)
36     {
37
38         case 1:
39             InitBiTree(&(L.tree[num-1].T));
40             printf("\t\t\t二叉树创建成功!\n");
41             getchar();getchar();
42             break;
43
```

```
44         case 2:
45         if (L.tree[num-1].T == NULL)
46         {
47             printf("\t\t\t二叉树不存在! \n");
48             getchar();getchar();
49             break;
50         }
51         DestroyBiTree(&(L.tree[num-1].T));
52         printf("\t\t\t二叉树销毁成功! \n");
53         getchar();getchar();
54         break;
55
56         case 3:
57         if (L.tree[num-1].T == NULL)
58         {
59             printf("\t\t\t二叉树不存在! \n");
60             getchar();getchar();
61             break;
62         }
63         getchar();
64         printf("\t\t\t请用前序方式构建二叉树, #表示空结点! \n");
65         CreateBiTree(&L.tree[num-1].T);
66         printf("\t\t\t二叉树构造成功! \n");
67         getchar();getchar();
68         break;
69
70         case 4:
71         if (L.tree[num-1].T == NULL)
72         {
73             printf("\t\t\t二叉树不存在! \n");
74             getchar();getchar();
75             break;
76         }
77         if(ClearBiTree(L.tree[num-1].T) == 1)
```

```
78         printf("\t\t\t叉树清空成功! \n");
79         else
80         printf("\t\t\t二叉树清空失败! \n");
81         getchar();getchar();
82         break;
83
84     case 5:
85     if (L.tree[num-1].T==NULL)
86     {
87         printf("\t\t\t二叉树不存在! \n");
88         getchar();getchar();
89         break;
90     }
91     if(BiTreeEmpty(L.tree[num-1].T) == OK)
92     printf("\t\t\t二叉树为空树! \n");
93     else
94     printf("\t\t\t二叉树不为空树! \n");
95     getchar();getchar();
96     break;
97
98     case 6:
99     if (L.tree[num-1].T == NULL)
100    {
101        printf("\t\t\t二叉树不存在! \n");
102        getchar();getchar();
103        break;
104    }
105    if(BiTreeEmpty(L.tree[num-1].T) == OK)
106    printf("\t\t\t二叉树为空树! \n");
107    else
108    printf("\t\t\t二叉树的深度为:%d\n",
109        BiTreeDepth(L.tree[num-1].T));
109    getchar();getchar();
110    break;
111
```

```
112         case 7:
113             int newkey;
114             if (L.tree[num-1].T == NULL)
115             {
116                 printf("\t\t\t二叉树不存在! \n");
117                 getchar();getchar();
118                 break;
119             }
120             printf("\t\t\t请输入你要查找的key:");
121             scanf("%d", &key);
122             getchar();
123             printf("\t\t\t请输入你要重新赋值的data和
124                 key: (用空格隔开) ");
125             scanf("%c %d", &ch,&newkey);
126             if (Assign(L.tree[num-1].T, key,newkey,
127                 ch) != ERROR){
128                 printf("\t\t\t该key对应的data改为
129                     了: %c\n", ch);
130                 printf("\t\t\t该key对应的key改为
131                     了: %d\n", newkey);
132             }
133             else
134             printf("\t\t\t输入的key在树种不存在! \n")
135                 ;
136             getchar();getchar();
137             break;
138
139         case 8:
140             if (L.tree[num-1].T == NULL)
141             {
142                 printf("\t\t\t二叉树不存在! \n");
143                 getchar();getchar();
144                 break;
145             }
146             printf("\t\t\t请输入你要查找左右兄弟的key
```

```
        : ");
142     scanf("%d", &key);
143     if (LeftSibling(L.tree[num-1].T, key) !=
        NULL)
144     {
145         printf("\t\t\t左兄弟对应的data
            为: %c\n", LeftSibling(L.tree[
            num-1].T, key)->data);
146         printf("\t\t\t左兄弟对应的key为:
            %d\n", LeftSibling(L.tree[num
            -1].T, key)->key);
147     }
148     else{printf("\t\t\t左兄弟不存在! \n");}
149
150     if (RightSibling(L.tree[num-1].T, key) !=
        NULL)
151     {
152         printf("\t\t\t右兄弟对应的data
            为: %c\n", RightSibling(L.tree
            [num-1].T, key)->data);
153         printf("\t\t\t右兄弟对应的key为:
            %d\n", RightSibling(L.tree[num
            -1].T, key)->key);
154     }
155     else{printf("\t\t\t右兄弟不存在! \n");}
156     getchar();getchar();
157     break;
158
159     case 9:
160     if (L.tree[num-1].T == NULL)
161     {
162         printf("\t\t\t二叉树不存在! \n");
163         getchar();getchar();
164         break;
165     }
```

```
166         printf("\t\t\t请输入你要插入位置的key: ")
           ;
167         scanf("%d", &key);
168         getchar();
169         T1 = Find(L.tree[num-1].T, key);
170         printf("\t\t\t请输入你想要插入的方向,0表示
           左边,1表示右边: ");
171         scanf("%d", &LR);
172         getchar();
173         CreateBiTree(&T2);
174         if (InsertChild(L.tree[num-1].T, T1, LR,
           T2) == OK)
175         {
176             printf("\t\t\t插入成功!\n");
177             getchar(); getchar();
178         }
179         else
180         {
181             printf("\t\t\t插入失败!\n");
182             getchar(); getchar();
183         }
184         break;
185
186         case 10:
187         if (L.tree[num-1].T == NULL)
188         {
189             printf("\t\t\t二叉树不存在! \n");
190             getchar();getchar();
191             break;
192         }
193         printf("\t\t\t请输入你要删除位置的key: ")
           ;
194         scanf("%d", &key);
195         getchar();
196         T1 = Find(L.tree[num-1].T, key);
```



```
197         printf("\t\t\t请输入你想要删除的方向,0表  
           示左边,1表示右边: ");  
198         scanf("%d", &LR);  
199         getchar();  
200         if (DeleteChild(L.tree[num-1].T, T1, LR)  
           == OK)  
201         {  
202             printf("\t\t\t删除成功!\n");  
203             getchar(); getchar();  
204         }  
205         else  
206         {  
207             printf("\t\t\t删除失败!\n");  
208             getchar(); getchar();  
209         }  
210         break;  
211  
212         case 11:  
213         if (L.tree[num-1].T == NULL)  
214         {  
215             printf("\t\t\t二叉树不存在! \n");  
216             getchar();getchar();  
217             break;  
218         }  
219         printf("\t\t\t该二叉树的前序遍历为: \n");  
220         PreOrderTraverse(L.tree[num-1].T, visit);  
221         getchar();getchar();  
222         break;  
223  
224         case 12:  
225         if (!L.tree[num-1].T)  
226         {  
227             printf("\t\t\t二叉树不存在! \n");  
228             getchar();getchar();  
229             break;
```

```
230         }
231         printf("\t\t\t该二叉树的中序遍历为: \n");
232         InOrderTraverse(L.tree[num-1].T, visit);
233         getchar();getchar();
234         break;
235
236     case 13:
237         if (L.tree[num-1].T == NULL)
238         {
239             printf("\t\t\t二叉树不存在! \n");
240             getchar();getchar();
241             break;
242         }
243         printf("\t\t\t该二叉树的后序遍历为: \n");
244         PostOrderTraverse(L.tree[num-1].T, visit)
245             ;
246         getchar();getchar();
247         break;
248
249     case 14:
250         if (L.tree[num-1].T == NULL)
251         {
252             printf("\t\t\t二叉树不存在! \n");
253             getchar();getchar();
254             break;
255         }
256         printf("\t\t\t该二叉树的层序遍历为: \n");
257         LevelOrderTraverse(L.tree[num-1].T, visit
258             );
259         getchar();getchar();
260         break;
261
262     case 15:
263         printf("\t\t\t在第几个树操作?只支持20个树
264             进行操作: \n");
```

```
262         scanf("%d",&num);
263         if(num<1||num>20)
264         {
265             printf("\t\t\t请选择正确范围! \n"
266                 );
267             num=1;
268         }
269         getchar(); getchar();
270         break;
271
272         case 16:
273             printf("\t\t\t请输入要保存的文件名: ");
274             scanf("%s", filename);
275             if((fp=fopen(filename,"w"))==NULL) printf
276                 ("\t\t\t打开文件失败! \n");
277             else
278             {
279                 if(Save(L.tree[num-1].T,fp)==OK)
280                     printf("\t\t\t保存文件成功! \n");
281                 else printf("\t\t\t保存文件失败!
282                     \n");
283             }
284             fclose(fp);
285             getchar();getchar();
286             break;
287
288         case 17:
289             printf("\t\t\t请输入要加载的文件名: ");
290             scanf("%s",filename);
291             if((fp=fopen(filename,"r"))==NULL)
292                 printf("\t\t\t加载失败! \n");
293             else
294             {
295                 if(Load(&L.tree[num-1].T,fp)==OK)
296                     printf("\t\t\t加载成功! \n");
```

```
294         else printf("\t\t\t加载失败! \n")
                ;
295     }
296     fclose(fp);
297     getchar();getchar();
298     break;
299
300     case 18:
301     int e1,e2;
302     printf("\t\t\t请输入需要查找公共祖先的
        个元素的关键字! \n");
303     scanf("%d %d",&e1,&e2);
304     if(LowestCommonAncestor(L.tree[num-1].T,
        e1,e2)==NULL ){
305         printf("\t\t\t最近公共祖不存在! \
            n");
306         getchar();getchar();
307         break;
308     }
309     printf("\t\t\t最近公共祖先是%d %c!\n",
        LowestCommonAncestor(L.tree[num-1].T,
        e1,e2)->key,LowestCommonAncestor(L.
        tree[num-1].T, e1,e2)->data);
310     getchar();getchar();
311     break;
312
313     case 19:
314     if (!L.tree[num-1].T)
315     {
316         printf("\t\t\t二叉树不存在! \n");
317         getchar();getchar();
318         break;
319     }
320     if(BiTreeEmpty(L.tree[num-1].T) == OK)
321     printf("\t\t\t二叉树为空树! \n");
```

```
322         else{
323             printf("\t\t\t该二叉树为:\n");
324             TreeDisplay((L.tree[num-1].T),1,
                        visit);
325         }
326         getchar();getchar();
327         break;
328
329
330         case 20:
331         if(InvertTree(L.tree[num-1].T)==ERROR){
332             printf("\t\t\t二叉树为空树! \n");
333             getchar();getchar();
334             break;
335         }
336
337         printf("\t\t\t二叉树已翻转!\n");
338         printf("\t\t\t翻转后二叉树为:\n");
339         TreeDisplay((L.tree[num-1].T),1, visit);
340         getchar();getchar();
341         break;
342         case 21:
343         if(MaxPathSum(L.tree[num-1].T)==
            INFEASTABLE){
344             printf("\t\t\t二叉树为空树! \n");
345             getchar();getchar();
346             break;
347         }
348         printf("\t\t\t最大路径和为%d! \n",
            MaxPathSum(L.tree[num-1].T));
349         getchar();getchar();
350         break;
351         case 22:
352         printf("请输入要查找节点的关键字\n");
353         int e;
```

```
354         scanf("%d",&e);
355         if (LocateNode(L.tree[num-1].T,e)== NULL)
356             {
357                 printf("未找到节点! \n");
358                 getchar();getchar();
359                 break;
360             }
361         printf("节点的名称是%c。 \n",LocateNode(L.
362             tree[num-1].T,e)->data);
363         getchar();getchar();
364         break;
365     case 0:
366         break;
367 }//end of switch
368 }//end of while
369 printf("\n");
370 printf("\t\t\t欢迎下次再使用本系统! \n\n");
371 printf("\t\t\tPowered By@_||lbw\n\n");
372 //system("pause");
373 }//end of main()
374 /*-----page 23 on textbook -----*/
```

函数实现

```
1  /*-----page 23 on textbook -----*/
2  /**
3   * 函数名称: InitBiTree
4   * 初始条件: 二叉树T不存在
5   * 操作结果: 构造空树二叉树T
6   * 函数变量: BiTree *T
7   */
8  status InitBiTree(BiTree *T) {
9      *T = (BiTree)malloc(sizeof(BiTNode));
10     (*T)->lchild = NULL;
11     (*T)->rchild = NULL;
12     (*T)->data='#';//初始化二叉树,将左右指针指向空,data域设为
```

```

        #
13         return OK;
14     }
15
16     /**
17     * 函数名称: DestroyBiTree
18     * 初始条件: 二叉树T已存在
19     * 操作结果: 销毁二叉树T
20     *函数变量: BiTree *T
21     */
22     status DestroyBiTree(BiTree *T)
23     {
24         if (*T) {
25             if ((*T)->lchild)
26                 DestroyBiTree(&((*T)->lchild));
27             if ((*T)->rchild)
28                 DestroyBiTree(&((*T)->rchild));
29             free(*T);
30             (*T) = NULL; //使用递归依次释放左子树、右子树、根
                           节点指针
31         }
32         return OK;
33     }
34
35     /**
36     * 函数名称: CreateBiTree
37     * 初始条件: 二叉树T已存在
38     * 操作结果: 创建二叉树
39     *函数变量: BiTree *T
40     */
41     status CreateBiTree(BiTree *T)
42     {
43         char ch;
44         int key;
45         printf("\t\t\t请输入data: ");

```

```
46         scanf("%c", &ch);
47         getchar();
48         if(ch == '#')
49         {
50             *T = NULL;
51             return 0;
52         }
53         else
54         {
55             *T = (BiTree)malloc(sizeof(BiTNode));
56             (*T)->data = ch;
57             printf("\t\t\t请输入key:");
58             scanf("%d", &(*T)->key);
59             getchar();
60             CreateBiTree(&((*T)->lchild));
61             CreateBiTree(&((*T)->rchild));
62         }
63         return OK;
64     }
65     //取材于课本,依次输入节点值,并用key标记节点
66
67     /**
68     * 函数名称: ClearBiTree
69     * 初始条件: 二叉树T已初始化
70     * 操作结果: 构造二叉树
71     *函数变量:  BiTree *T
72     */
73     status ClearBiTree(BiTree T)
74     {
75         if(T)
76         {
77             T->lchild = NULL;
78             T->rchild = NULL;
79             T->data = '#';//清空二叉树,左右子树指针指向空,
                           data域设置为#
```



```
80         }
81         return OK;
82     }
83
84     /**
85     * 函数名称: BiTreeEmpty
86     * 初始条件: 二叉树T已存在
87     * 操作结果: 若T为空二叉树则返回TRUE, 否则返回FALSE
88     *函数变量:  BiTree *T
89     */
90     status BiTreeEmpty(BiTree T)
91     {
92         if(T->data=='#')//判断根节点的data域值是否为空
93         {
94             return OK;
95         }
96         else
97         {
98             return ERROR;//函数出错
99         }
100     }
101
102     /**
103     * 函数名称: BiTreeDepth
104     * 初始条件: 二叉树T已存在
105     * 操作结果: 返回T的深度
106     *函数变量:  BiTree *T, 深度depth
107     */
108     status BiTreeDepth(BiTree T)
109     {
110         int depth = 0;
111         if(T)
112         {
113             int lchilddepth = BiTreeDepth(T->lchild);
114             int rchilddepth = BiTreeDepth(T->rchild);
```

```
115         depth = (lchilddepth>=rchilddepth?(lchilddepth+1)
                  :(rchilddepth+1));
116     }//使用递归,得到左右子树的深度,并比较大小,之后返回最大的
        深度
117     return depth;
118 }
119
120 /**
121  * 函数名称: Root
122  * 初始条件: 二叉树T已存在
123  * 操作结果: 返回T的根
124  *函数变量:  BiTree *T
125  */
126 char Root(BiTree T)
127 {
128     return T->data;//T的data域的值即根节点
129 }
130
131 /**
132  * 函数名称: visit
133  * 操作结果: 打印字符e
134  */
135 status visit(char e)
136 {
137     printf("%c",e);//依次调用该函数,用来打印
138 }
139
140 /**
141  * 函数名称: Value
142  * 初始条件: 二叉树T已存在, e是T中的某个结点
143  * 操作结果: 返回e的值
144  *函数变量:  BiTree *T, 关键字e
145  */
146 char Value(BiTree T, int e)
147 {
```

```
148         if (!T) return ERROR;    //若二叉树为空,返回ERROR
149         BiTNode *st[10], *p;
150         int top = 0;    //置空栈
151         st[top++] = T;
152         while (top)
153         {
154             p = st[--top]; //先序遍历,弹出栈顶元素,判断是否有
                            //key的值与e相等
155             if (p->key == e)
156             {
157                 return p->data;
158             }
159
160             else {
161                 if(p->rchild!=NULL)
162                     st[top++] = p->rchild;
163                 if(p->lchild!=NULL)
164                     st[top++] = p->lchild;
165             }
166         }
167         return ERROR;
168     }
169
170     /**
171     * 函数名称: Assign
172     * 初始条件: 二叉树T已存在, e是T中的某个结点
173     * 操作结果: 结点e赋值为value
174     *函数变量: BiTree *T, 关键字e , 名称ch
175     */
176     status Assign(BiTree T, int e1, int e2, char ch)
177     {
178         if (!T) return ERROR;
179         BiTNode *st[10], *p;
180         int top = 0;
181         st[top++] = T;
```

```
182         while (top)
183         {
184             p = st[--top];
185             if (p->key == e1)
186             {
187                 p->data = ch;
188                 p->key == e2; //找到后进行复制,与Value函数
                                相似
189                 return OK;
190             }
191             else
192             {
193                 if (p->rchild != NULL)
194                     st[top++] = p->rchild;
195                 if (p->lchild != NULL)
196                     st[top++] = p->lchild;
197             }
198         }
199         return ERROR;
200     }
201
202     /**
203     * 函数名称: Parent
204     * 初始条件: 二叉树T已存在, e是T中的某个结点
205     * 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
206     *函数变量: BiTree *T, 关键字e
207     */
208     BiTNode* Parent(BiTree T, int e)
209     {
210         BiTree T1; //利用遍历和递归依次寻找左孩子右孩子对应的e值,
                符合条件便返回指针
211         if (T)
212         {
213             if ((T->lchild != NULL && T->lchild->key == e) || (T->
```

```
        rchild!=NULL&& T->rchild->key == e)) return T;
214         T1 = Parent(T->lchild, e);
215         if (T1 != NULL) return T1;
216         T1 = Parent(T->rchild, e);
217         if (T1 != NULL) return T1;
218     }
219     return NULL;
220 }
221
222 /**
223  * 函数名称: LeftChild
224  * 初始条件: 二叉树T已存在, e是T中的某个结点
225  * 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
226  *函数变量: BiTree *T, 关键字e
227  */
228 BiTNode* LeftChild(BiTree T, int e)
229 {
230     BiTree p;//利用递归找左孩子,思路简单
231     if (T)
232     {
233         if (T->key == e) return T->lchild;
234         p = LeftChild(T->lchild, e);
235         if (p != NULL) return p;
236         p = LeftChild(T->rchild, e);
237         if (p != NULL) return p;
238     }
239     return NULL;
240 }
241
242 /**
243  * 函数名称: RightChild
244  * 初始条件: 二叉树T已存在, e是T中的某个结点
245  * 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
246  *函数变量: BiTree *T, 关键字e
247  */
```

```
248 BiTNode* RightChild(BiTree T, int e)
249 {
250     BiTree p;//与上一个同理
251     if (T)
252     {
253         if (T->key == e) return T->rchild;
254         p = RightChild(T->lchild, e);
255         if (p != NULL) return p;
256         p = RightChild(T->rchild, e);
257         if (p != NULL) return p;
258     }
259     return NULL;
260 }
261
262 /**
263  * 函数名称: LeftSibling
264  * 初始条件: 二叉树T已存在, e是T中的某个结点
265  * 操作结果: 返回e的左兄弟结点指针。若e是T的左孩子或者无左兄弟, 则
                返回NULL
266  *函数变量: BiTree *T, 关键字e
267  */
268 BiTNode* LeftSibling(BiTree T, int e)
269 {
270     BiTree p=NULL;
271     if (T)
272     {
273         if (T->rchild!=NULL&&T->rchild->key == e)//如果右
                孩子不为空并且节点的值符合,那么便返回左孩子的
                指针
274         return T->lchild;
275         p = LeftSibling(T->lchild, e);
276         if (p != NULL) return p;
277         p = LeftSibling(T->rchild, e);
278         if (p != NULL) return p;
279     }
```

```
280         return NULL;
281     }
282
283     /**
284     * 函数名称: RightSibling
285     * 初始条件: 二叉树T已存在, e是T中的某个结点
286     * 操作结果: 返回e的左兄弟结点指针。若e是T的左孩子或者无左兄弟, 则
                返回NULL; 返回e的右兄弟结点指针。若e是T的右孩子或者无右兄弟, 则返
                回NULL
287     * 函数变量: BiTree *T, 关键字e
288     */
289     BiTNode* RightSibling(BiTree T, int e)
290     {
291         BiTree p=NULL;
292         if (T)
293         {
294             if (T->lchild!=NULL&&T->lchild->key == e)//与上一
                个函数类似,最后依次递归遍历左子树右子树
295                 return T->rchild;
296             p = RightSibling(T->lchild, e);
297             if (p != NULL) return p;
298             p = RightSibling(T->rchild, e);
299             if (p != NULL) return p;
300         }
301         return NULL;
302     }
303
304     BiTree Find(BiTree T, int key)
305     {
306         BiTree T1;
307         if (T == NULL) return NULL;
308         if (T->key == key) return T;
309         else
310         {
311             T1 = Find(T->lchild, key);
```

```
312         if (T1 != NULL) return T1;
313         T1 = Find(T->rchild, key);
314         if (T1 != NULL) return T1;
315     }
316     return NULL;
317 }
318
319 /**
320 * 函数名称: InsertChild
321 * 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
322 * 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
323 *函数变量: 指针BiTree T, 指针BiTree p, 左孩子 LR, 指针BiTree c
324 */
325 status InsertChild(BiTree T, BiTree p, int LR, BiTree c)
326 {
327     if (!T)
328     {
329         printf("\t\t\t二叉树不存在!");
330         return ERROR;
331     }
332     if (c->rchild != NULL)
333     {
334         printf("\t\t\t待插入二叉树的右子树不为空!");
335         return ERROR;
336     }
337     if (LR == 0)
338     {
339         c->rchild = p->lchild; //插入方向为左的情况
340         p->lchild = c;
341     }
342     else
343     {
344         c->rchild = p->rchild; //插入方向为右的情况
345         p->rchild = c;
```



```
346     }
347     return OK;
348 }
349
350 /**
351  * 函数名称: DeleteChild
352  * 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
353  * 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
354  * 指针BiTree T, 指针BiTree p, 左孩子int LR
355  */
356 status DeleteChild(BiTree T, BiTree p, int LR)
357 {
358     BiTree T1;
359     if (T == NULL)
360     {
361         printf("\t\t\t二叉树不存在! \n");
362         return ERROR;
363     }
364     if (LR == 0) //删除左子树
365     {
366         T1 = p->lchild;
367         p->lchild = NULL;
368         if (!DestroyBiTree(&T1)) return ERROR;
369     }
370     else //删除右子树
371     {
372         T1 = p->rchild;
373         p->rchild = NULL;
374         if (!DestroyBiTree(&T1)) return ERROR;
375     }
376     return OK;
377 }
378
379 /**
380  * 函数名称: PreOrderTraverse
```

```
381 * 初始条件：二叉树T已存在
382 * 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
383 *BiTree T, 函数指针* visit)(char e)
384 */
385 status PreOrderTraverse(BiTree T, status (* visit)(char e))
386 {
387     if(T)
388     {
389         if(visit(T->data))//先序遍历，利用递归方式
390             if(PreOrderTraverse(T->lchild, visit))
391                 if(PreOrderTraverse(T->rchild, visit))
392                     return 1;
393         return 0;
394     }
395     else return 1;
396 }
397
398 /**
399 * 函数名称：InOrderTraverse
400 * 初始条件：二叉树T已存在
401 * 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
402 *BiTree T, 函数指针* visit)(char e)
403 */
404 status InOrderTraverse(BiTree T, status (* visit)(char e))
405 {
406     if(T)
407     {
408         if(InOrderTraverse(T->lchild, visit))
409             if(visit(T->data))//中序遍历，利用递归方式
410                 if(InOrderTraverse(T->rchild, visit))
411                     return 1;
412         return 0;
413     }
```

```
414         else return 1;
415     }
416
417     /**
418     * 函数名称: PostOrderTraverse
419     * 初始条件: 二叉树T已存在
420     * 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
421     *BiTree T, 函数指针* visit)(char e)
422     */
423     status PostOrderTraverse(BiTree T, status (* visit)(char e))
424     {
425         if(T)
426         {
427             if(PostOrderTraverse(T->lchild, visit))
428             if(PostOrderTraverse(T->rchild, visit))
429             if(visit(T->data))//后序遍历, 利用递归方式
430             return 1;
431             return 0;
432         }
433         else return 1;
434     }
435
436     void level(BiTree T, int i)
437     {
438         if(T)
439         {
440             if(i == 1)
441             visit(T->data);//该函数用来辅助层序遍历
442             else
443             {
444                 level(T->lchild, i-1);
445                 level(T->rchild, i-1);
446             }
447         }
```

```
448 }
449
450 /**
451 * 函数名称: LevelOrderTraverse
452 * 初始条件: 二叉树T已存在
453 * 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
454 *BiTree T, 函数指针* visit)(char e)
455 */
456 status LevelOrderTraverse(BiTree T, status (* visit)(char e))
457 {
458     if(T)
459     {
460         int h = BiTreeDepth(T); //调用函数, 得到深度
461         int i;
462         for(i=1; i<=h; i++)
463         {
464             level(T, i); //对每一层进行访问
465         }
466         return OK;
467     }
468     else
469         return OK;
470 }
471
472 /**
473 * 函数名称: Save
474 * 初始条件: 二叉树T已存在
475 * 操作结果: 保存二叉树为文件
476 *BiTree T, 文件指针FILE *fp
477 */
478 status Save(BiTree T, FILE *fp)
479 {
480     int i = 0;
481     char ch = '#';
```

```
482         if(T)
483         {
484             fprintf(fp,"%d%c",T->key,T->data);//将key和data依
                次写入文件
485             if(Save(T->lchild,fp))
486             if(Save(T->rchild,fp)) return OK;
487             else return ERROR;
488         }
489     else
490     {
491         fprintf(fp,"%d%c",i,ch);//i和ch初始设置,意味着空
                节点的key为0,#代表空节点.
492         return OK;
493     }
494 }
495
496 /**
497 * 函数名称: Load
498 * 操作结果: 从文件中读取二叉树
499 *BiTree T,文件指针FILE *fp
500 */
501 status Load(BiTree *T,FILE *fp)
502 {
503     int i;//用i来表示key
504     char ch;//用ch来读取data域
505     if(feof(fp))
506     {
507         (*T)=NULL;
508         return OK;
509     }
510     fscanf(fp,"%d",&i);
511     fscanf(fp,"%c",&ch);
512     if(ch=='#')
513     {
514         (*T)=NULL;
```

```
515     }
516     else
517     {
518         (*T)=(BiTree)malloc(sizeof(BiTNode));
519         (*T)->key=i;
520         (*T)->data=ch;
521         Load(&((*T)->lchild),fp);
522         Load(&((*T)->rchild),fp);
523     }//相当于依次读取之后创建为二叉树
524     return OK;
525 }
526
527 /**
528 函数名称: TreeDisplay
529 初始条件: 二叉树T存在
530 操作结果: 按照树形结构打印二叉树
531 函数变量: BiTree T,文件指针FILE *fp, 深度depth
532 */
533 status TreeDisplay(BiTree T,int depth,status (* visit)(char e))
534 {
535     if(!T)
536     {
537         printf("\n");
538         return OK;
539     }
540     int i=0;
541     for(; i<depth; i++)
542         printf(" ");
543     visit(T->data);
544     printf("\n");
545     if(T->lchild||T->rchild)
546     {
547         TreeDisplay(T->lchild,depth+1,visit);
548         TreeDisplay(T->rchild,depth+1,visit);
549     }
```

```
550         return OK;
551     }
552     /**
553     函数名称: LowestCommonAncestor
554     初始条件: 二叉树T存在, e1和e2是二叉树中的两个节点
555     操作结果: 查找二叉树中指定节点的最近公共祖先
556     函数变量: BiTree T,关键字e1, 关键字e2
557     */
558     BiTNode* LowestCommonAncestor(BiTree T, int e1, int e2) {
559         if (T == NULL) {
560             return NULL;
561         }
562         if (T->key == e1 || T->key == e2) {
563             return T;
564         }
565         // 在左子树中查找
566         BiTNode* left = LowestCommonAncestor(T->lchild, e1, e2);
567         // 在右子树中查找
568         BiTNode* right = LowestCommonAncestor(T->rchild, e1, e2);
569         // 如果 e1 和 e2 分别在左右子树中, 则 T 是最近公共祖先
570         if (left != NULL && right != NULL) {
571             return T;
572         }
573         // 否则返回不为空的子树
574         return (left != NULL) ? left : right;
575     }
576
577     /**
578     函数名称: LocateNode
579     初始条件: 二叉树T存在, e是二叉树中的一个节点的key值
580     操作结果: 查找二叉树中指定key值的节点
581     函数变量: BiTree T,关键字e1
582     */
583     BiTNode* LocateNode(BiTree T, int e) {
584         if (T == NULL) {
```

```
585         return NULL;
586     }
587     std::queue<BiTree> q;
588     q.push(T);
589     while (!q.empty()) {
590         BiTree node = q.front();
591         q.pop();
592         if (node->key == e) {
593             return node;
594         }
595         // 将左右子树入队
596         if (node->lchild != NULL) {
597             q.push(node->lchild);
598         }
599         if (node->rchild != NULL) {
600             q.push(node->rchild);
601         }
602     }
603     // 未找到
604     return NULL;
605 }
606 /**
607
608 函数名称: InvertTree
609 初始条件: 二叉树T存在
610 操作结果: 翻转二叉树
611 函数变量: &BiTree T
612 */
613 status InvertTree(BiTree &T)
614 {
615     if (T == NULL) {
616         return OK;
617     }
618     // 递归交换左右子树
619     BiTree temp = T->lchild;
```



```
620         T->lchild = T->rchild;
621         T->rchild = temp;
622         InvertTree(T->lchild);
623         InvertTree(T->rchild);
624         return OK;
625     }
626
627     /**
628     函数名称: MaxPathSum
629     初始条件: 二叉树T存在
630     操作结果: 计算二叉树中根节点到叶子节点的最大路径和
631     函数变量: BiTree T
632     */
633     status MaxPathSum(BiTree T) {
634         if (T == NULL) {
635             return INFEASTABLE;
636         }
637         int left = MaxPathSum(T->lchild);
638         int right = MaxPathSum(T->rchild);
639         // 计算包含当前节点的最大路径和
640         int sum = T->key + ((left > 0) ? left : 0) + ((right > 0)
            ? right : 0);
641         // 返回不包含当前节点的最大路径和
642         return (left > right) ? ((left > 0) ? left : 0) + T->key
            : ((right > 0) ? right : 0) + T->key;
643     }
```

8 附录 D 基于邻接表图实现的源程序

相关定义

```
1  /* Linear Table On Sequence Structure */
2  #include<stdio>
3  #include<cstring>
4  #include<iostream>
5  #include<cstdlib>
6  #include<algorithm>
7  #include<malloc.h>
8  #include<numeric>
9
10 /*-----page 10 on textbook -----*/
11
12 #define TRUE 1//定义真值
13 #define FALSE 0//定制假值
14 #define OK 1//程序正常运行
15 #define ERROR 0//程序运行出错
16 #define INFEASTABLE -1//输入或输出不合法
17 #define OVERFLOW -2//数值溢出
18 #define MAX_VERTEX_NUM 20 // 最大顶点数
19 #define LIST_INIT_SIZE 100 // 邻接表的初始长度
20 #define LISTINCREMENT 10 // 邻接表的增量
21
22 typedef int status; // 定义所有状态码和返回值的类型为int
23 typedef int KeyType; // 定义类型 KeyType，表示顶点标识
24 typedef enum { DG, DN, UDG, UDN } GraphKind; // 定义一个枚举类型
    GraphKind
25 typedef int QElemType; // 定义QElemType，表示队列中的元素，是一个
    整型
26 typedef struct
27 {
28     KeyType key; // 数据项的唯一标识，整型
29     char others[20]; // 数据项的其他信息，字符型
```

```
30 } VertexType; // 定义VertexType, 表示顶点的数据结构, 包括唯一标识
    和其他信息
31
32 typedef struct ArcNode
33 {      // 定义 ArcNode, 表示边的表结点
34     int adjvex;          // 相邻顶点的位置编号
35     struct ArcNode* nextarc; // 下一个表结点指针
36 } ArcNode;
37
38 typedef struct VNode
39 {      // 定义一个结构体类型 VNode, 表示头结点及其数组
40     VertexType data;      // 顶点信息
41     ArcNode* firstarc;    // 指向第一条弧
42 } VNode, AdjList[MAX_VERTEX_NUM];
43 // 定义VNode, 表示顶点的结构体类型, 包括顶点信息和指向第一条弧的
    指针; 定义AdjList, 表示头结点的数组类型, 每个头结点的指针均指
    向一个链表, 构成一个更大链表
44
45 typedef struct { // 定义一个结构体类型 ALGraph, 表示邻接表的类
    型
46     AdjList vertices;      // 顶点数组表示邻接表
47     int vexnum, arcnum;    // 顶点数, 边数
48     GraphKind kind;       // 图的类型
49 } ALGraph;
50
51 typedef struct QNode { // 定义QNode, 表示队列结点
52     QElemType data; // 队列中的元素
53     struct QNode* next; // 下一个队列结点指针
54 } QNode, * QueuePtr;
55
56 typedef struct {
57     QueuePtr front1, rear; // 定义LinkQueue, 表示链式队列, 包
        括队头和队尾指针
58 } LinkQueue;
59 int book[MAX_VERTEX_NUM]; // 定义哈希表来判断顶点是否被访问过
```

```
60  int visited[MAX_VERTEX_NUM]; // 定义数组visited判断顶点是否被访问
    过
61
62  /*-----函数声明-----*/
63  status CreateGraph(ALGraph& G, VertexType V[], KeyType VR[][2]);
64  // 声明一个函数 CreateGraph, 用于创建图
65  status DestroyGraph(ALGraph& G);
66  // 声明一个函数 DestroyGraph, 用于销毁图
67  int LocateVex(ALGraph G, KeyType u);
68  // 声明一个函数 LocateVex, 用于查找顶点在图中的位置
69  status PutVex(ALGraph& G, KeyType u, VertexType value);
70  // 声明一个函数 PutVex, 用于修改顶点的值
71  int FirstAdjVex(ALGraph G, KeyType u);
72  // 声明一个函数 FirstAdjVex, 用于查找顶点的第一个邻接点
73  int NextAdjVex(ALGraph G, KeyType v, KeyType w);
74  // 声明一个函数 NextAdjVex, 用于查找顶点的下一个邻接点
75  status InsertVex(ALGraph& G, VertexType v);
76  // 声明一个函数 InsertVex, 用于插入顶点
77  status DeleteVex(ALGraph& G, KeyType v);
78  // 声明一个函数 DeleteVex, 用于删除顶点
79  status InsertArc(ALGraph& G, KeyType v, KeyType w);
80  // 声明一个函数 InsertArc, 用于插入边
81  status DeleteArc(ALGraph& G, KeyType v, KeyType w);
82  // 声明一个函数 DeleteArc, 用于删除边
83  void DFS(ALGraph G, int v, void (*visit)(VertexType)) ;
84  // 声明一个函数 DFS, 用于深度优先遍历图
85  void visit(VertexType v);
86  // 声明一个函数 visit, 用于访问顶点
87  status DFSTraverse(ALGraph& G, void (*visit)(VertexType));
88  // 声明一个函数 DFSTraverse, 用于深度优先遍历图
89  status InitQueue(LinkQueue& Q);
90  // 声明一个函数 InitQueue, 用于初始化队列
91  status DestroyQueue(LinkQueue& Q) ;
92  // 声明一个函数 DestroyQueue, 用于销毁队列
93  status ClearQueue(LinkQueue& Q) ;
```

```
94 // 声明一个函数 ClearQueue, 用于清空队列
95 status QueueEmpty(LinkQueue Q) ;
96 // 声明一个函数 QueueEmpty, 用于判断队列是否为空
97 int QueueLength(LinkQueue Q);
98 // 声明一个函数 QueueLength, 用于获取队列长度
99 status EnQueue(LinkQueue& Q, QElemType e) ;
100 // 声明一个函数 EnQueue, 用于入队
101 status DeQueue(LinkQueue& Q, QElemType& e);
102 // 声明一个函数 DeQueue, 用于出队
103 status BFSTraverse(ALGraph& G, void(*visit)(VertexType)) ;
104 // 声明一个函数 BFSTraverse, 用于广度优先遍历图
105 status SaveGraph(ALGraph G, char FileName[]);
106 // 声明一个函数 SaveGraph, 用于将图保存到文件中
107 status Load(ALGraph& G, char FileName[]);
108 // 声明一个函数 Load, 用于从文件中加载图
109 status LoadGraph(ALGraph& G, char FileName[]) ;
110 // 声明一个函数 LoadGraph, 用于从文件中加载图
111 status VerticesSetLessThanK(ALGraph G, KeyType key, int k) ;
112 // 声明一个函数 VerticesSetLessThanK, 用于查找度数小于 k 的顶点集
    合
113 status ShortestPathLength(ALGraph G, VertexType v, VertexType w);
114 // 声明一个函数 ShortestPathLength, 用于查找两个顶点之间的最短路
    径长度
115 void bfs(ALGraph G, int startnode);
116 // 声明一个函数 bfs, 用于广度优先遍历图
117 status ConnectedComponentsNums(ALGraph G);
118 // 声明一个函数 ConnectedComponentsNums, 用于查找图的连通分量个数
```

演示系统

```
1 int main() {
2     int op = 1, i = 0, i_num = 0;
3     KeyType key;
4     VertexType e;
5     ALGraph G[11];
6     for (i = 0; i < 10; i++){
```

```
7         G[i].kind = DG;
8         G[i].vexnum = 0;
9     }
10    while (op) {
11        system("cls");
12        printf("\n\n");
13        printf("\t\t\t\t\t Menu for Undirected Graph On
14        Chain Structure \n");
15        printf("\t\t\t\t\t *****
16        n");
17        printf("\t\t\t\t\t 1. CreateGraph \t 2. DestroyGraph
18        \n");
19        printf("\t\t\t\t\t 3. LocateVex \t 4. PutVex\n");
20        printf("\t\t\t\t\t 5. FirstAdjVex \t 6. NextAdjVex\n
21        ");
22        printf("\t\t\t\t\t 7. InsertVex \t 8. DeleteVex\n"
23        );
24        printf("\t\t\t\t\t 9. InsertArc \t 10. DeleteArc\n"
25        );
26        printf("\t\t\t\t\t 11. DFSTraverse \t 12. BFSTraverse\
27        n");
28        printf("\t\t\t\t\t 13. Save \t 14. Load\n");
29        printf("\t\t\t\t\t 15. Choose \t 0. Exit\n");
30        printf("\t\t\t\t\t -----\\
31        n");
32        printf("\t\t\t\t\t 16. VerticesSetLessThanK 17.
33        ShortestPathLength\n");
34        printf("\t\t\t\t\t 18. ConnectedComponentsNums\n");
35        printf("\t\t\t\t\t *****
36        n");
37        printf("\t\t\t\t\t 请选择你的操作[0-18]: ");
38        scanf("%d", &op);
```

```
29         switch (op) {
30             case 1: {
31                 VertexType V[30];
32                 KeyType VR[100][2];
33                 printf("请输入顶点序列和关系对序
34                        列:\n");
35                 i = 0;
36                 do {
37                     scanf("%d%s", &V[i].key,
38                           V[i].others);
39                 } while (V[i++].key != -1);
40                 i = 0;
41                 do {
42                     scanf("%d%d", &VR[i][0],
43                           &VR[i][1]);
44                 } while (VR[i++][0] != -1);
45                 if (CreateGraph(G[i_num], V, VR)
46                     == OK)
47                     printf("\t\t\t创建无向图成功!\n");
48                     ;
49                 else
50                     printf("\t\t\t创建无向图失败!\n");
51                     ;
52                 getchar(); getchar();
53                 break;
54             }
55             case 2: {
56                 if (G[i_num].vexnum == 0)
57                 {
58                     printf("\t\t\t无向图不存
59                            在!\n");
60                     getchar(); getchar();
61                     break;
62                 }
63                 if (DestroyGraph(G[i_num]) == OK)
```

```

{
57         printf("\t\t\t销毁无向图
           成功!\n");
58         for (i = i_num; i < 10; i
           ++ )
59             {
60                 G[i] = G[i + 1];
61             }
62         i_num = 0;
63     }
64     else
65         printf("\t\t\t销毁无向图失败!\n")
           ;
66         getchar(); getchar();
67         break;
68 }
69 case 3: {
70     if (G[i_num].vexnum == 0)
71     {
72         printf("\t\t\t无向图不存
           在!\n");
73         getchar(); getchar();
74         break;
75     }
76     printf("\t\t\t请输入所要查找的顶
           点的key:");
77     scanf("%d", &key);
78     if (LocateVex(G[i_num], key) ==
           -1)
79         printf("\t\t\t无向图中不存在该节
           点!\n");
80     else {
81         printf("\t\t\t该顶点的编
           号为%d ", LocateVex(G[
           i_num], key) + 1);

```



```
82             visit(G[i_num].vertices[
                    LocateVex(G[i_num],
                                key)]).data);
83         }
84         getchar(); getchar();
85         break;
86     }
87     case 4: {
88         if (G[i_num].vexnum == 0)
89         {
90             printf("\t\t\t无向图不存在!\n");
91             getchar(); getchar();
92             break;
93         }
94         printf("\t\t\t请输入要修改的key:
                    ");
95         scanf("%d", &key);
96         if (LocateVex(G[i_num], key) ==
                    -1)
97             printf("\t\t\t该顶点不存在!\n");
98         else
99         {
100             printf("\t\t\t请输入要将
                    该点修改为的key和值: "
                    );
101             scanf("%d%s", &e.key, e.
                    others);
102             if (PutVex(G[i_num], key,
                    e) == OK) printf("\t\t\t修改成功\n");
103             else printf("\t\t\t修改失败\n");
104         }
105         getchar(); getchar();
```

```
106             break;
107         }
108
109         case 5: {
110             if (G[i_num].vexnum == 0)
111             {
112                 printf("\t\t\t无向图不存在!\n");
113                 getchar(); getchar();
114                 break;
115             }
116             printf("\t\t\t请输入要查找的顶点的key: ");
117             scanf("%d", &key);
118             if (FirstAdjVex(G[i_num], key) == -1)
119                 printf("\t\t\t查找失败!\n");
120             else {
121                 printf("\t\t\t该顶点首个邻接顶点的序号为%d\n",
122                     FirstAdjVex(G[i_num], key) + 1);
123                 printf("\t\t\t该顶点首个邻接顶点的值为%d,%s\n",
124                     G[i_num].vertices[FirstAdjVex(G[i_num], key)].data.key, G[i_num].vertices[FirstAdjVex(G[i_num], key)].data.others);
125             }
126             getchar(); getchar();
127             break;
128         }
129         case 6: {
```

```
128         if (G[i_num].vexnum == 0)
129         {
130             printf("\t\t\t无向图不存在!\n");
131             getchar(); getchar();
132             break;
133         }
134         printf("\t\t\t请输入要查找的顶点的key:");
135         scanf("%d", &key);
136         printf("\t\t\t请输入和其相对的顶点的key: ");
137         scanf("%d", &i);
138         if (NextAdjVex(G[i_num], key, i) == -1)
139             printf("\t\t\t查找失败!\n");
140         else
141             printf("\t\t\tv相对于w的下一个邻接顶点为%d,%s\n", G[i_num].vertices[NextAdjVex(G[i_num], key, i)].data.key, G[i_num].vertices[NextAdjVex(G[i_num], key, i)].data.others);
142         getchar(); getchar();
143         break;
144     }
145     case 7: {
146         if (G[i_num].vexnum == 0)
147         {
148             printf("\t\t\t无向图不存在!\n");
149             getchar(); getchar();
150             break;
151         }
152         printf("\t\t\t请输入要添加的顶点
```

```
        的key和值: ");
153     scanf("%d%s", &e.key, e.others);
154     if (LocateVex(G[i_num], e.key) !=
        -1)
155         printf("\t\t\t该顶点已存在,添加失
            败!\n");
156     else
157     {
158         if (InsertVex(G[i_num], e
            ) == OK)
159             printf("\t\t\t添加顶点成
                功!\n");
160         else
161             printf("\t\t\t添加顶点失
                败!\n");
162     }
163     getchar(); getchar();
164     break;
165 }
166 case 8: {
167     if (G[i_num].vexnum == 0)
168     {
169         printf("\t\t\t无向图不存
            在!\n");
170         getchar(); getchar();
171         break;
172     }
173     printf("\t\t\t请输入要删除的顶点:
        ");
174     scanf("%d", &key);
175     if (DeleteVex(G[i_num], key) ==
        ERROR)
176         printf("\t\t\t所要删除的顶点不存
            在!\n");
177     else printf("\t\t\t删除顶点成功!\n");
```

```

n");
178         getchar(); getchar();
179         break;
180     }
181     case 9: {
182         if (G[i_num].vexnum == 0)
183         {
184             printf("\t\t\t无向图不存
                在!\n");
185             getchar(); getchar();
186             break;
187         }
188         printf("\t\t\t请输入边的头节点和
                尾节点的key:");
189         scanf("%d%d", &key, &i);
190         if (InsertArc(G[i_num], key, i)
            == ERROR)
191             printf("\t\t\t添加失败!\n");
192         else
193             printf("\t\t\t添加成功!\n");
194         getchar(); getchar();
195         break;
196     }
197     case 10: {
198         if (G[i_num].vexnum == 0)
199         {
200             printf("\t\t\t无向图不存
                在!\n");
201             getchar(); getchar();
202             break;
203         }
204         printf("\t\t\t请输入边的尾节点和
                头节点的key:");
205         scanf("%d%d", &key, &i);
206         if (DeleteArc(G[i_num], key, i)
```

```

207         == ERROR)
208         printf("\t\t\t删除失败!\n");
209     else
210         printf("\t\t\t删除成功!\n");
211         getchar(); getchar();
212         break;
213     }
214     case 11: {
215         if (G[i_num].vexnum == 0)
216         {
217             printf("\t\t\t无向图不存在!\n");
218             getchar(); getchar();
219             break;
220         }
221         printf("\t\t\t该无向图的深度优先
222             搜索遍历为: ");
223         DFSTraverse(G[i_num], visit);
224         getchar(); getchar();
225         break;
226     }
227     case 12: {
228         if (G[i_num].vexnum == 0)
229         {
230             printf("\t\t\t无向图不存在!\n");
231             getchar(); getchar();
232             break;
233         }
234         printf("\t\t\t该无向图的广度优先
235             搜索遍历为: ");
236         BFSTraverse(G[i_num], visit);
237         getchar(); getchar();
238         break;
239     }
240 }
```

```
237         case 13: {
238             char filename[100];
239             if (G[i_num].vexnum == 0)
240             {
241                 printf("\t\t\t无向图不存在!\n");
242                 getchar(); getchar();
243                 break;
244             }
245             printf("\t\t\t请输入文件名:");
246             scanf("%s", filename);
247             if (SaveGraph(G[i_num], filename)
                == OK)
248                 printf("\t\t\t文件保存成功!\n");
249             else
250                 printf("\t\t\t文件保存失败!\n");
251             getchar(); getchar();
252             break;
253         }
254         case 14: {
255             char filename[100];
256             printf("\t\t\t请输入文件名:");
257             scanf("%s", filename);
258             if (Load(G[i_num], filename) ==
                OK)
259                 printf("\t\t\t加载成功!\n");
260             else
261                 printf("\t\t\t加载失败!\n");
262             getchar(); getchar();
263             break;
264         }
265         case 15: {
266             printf("\t\t\t请输入要在第几个表操
                作,只支持在10个顺序表进行操作:
                ");
```

```
267         scanf("%d", &i_num);
268         if (i_num < 0 || i_num > 9)
269         {
270             printf("\t\t\t不支持在该
                表上进行操作!\n");
271             i_num = 0;
272         }
273         printf("\t\t\t在第%d个表操作",
                i_num);
274         getchar(); getchar();
275         break;
276     }
277     case 16: {
278         if (G[i_num].vexnum == 0)
279         {
280             printf("\t\t\t无向图不存
                在!\n");
281             getchar(); getchar();
282             break;
283         }
284         printf("\t\t\t请输入顶点的key和k:
                ");
285         scanf("%d%d", &key, &i);
286         VerticesSetLessThanK(G[i_num],
                key, i);
287         getchar(); getchar();
288         break;
289     }
290     case 17: {
291         if (G[i_num].vexnum == 0)
292         {
293             printf("\t\t\t无向图不存
                在!\n");
294             getchar(); getchar();
295             break;
```



```
296         }
297         printf("\t\t\t请输入两个顶点的key
           :");
298         VertexType i;
299         scanf("%d%d", &e.key, &i.key);
300         printf("\t\t\t距离为%d",
           ShortestPathLength(G[i_num], e
           , i));
301         getchar(); getchar();
302         break;
303     }
304     case 18: {
305         if (G[i_num].vexnum == 0)
306         {
307             printf("\t\t\t无向图不存
               在!\n");
308             getchar(); getchar();
309             break;
310         }
311         printf("\t\t\t连通分量个数为:%d",
           ConnectedComponentsNums(G[
           i_num]));
312         getchar(); getchar();
313         break;
314     }
315     case 0:
316         break;
317     } //end of switch
318 } //end of while
319 printf("\t\t\t欢迎下次使用本系统!\n\n");
320 } //end of main()
321 /*-----page 23 on textbook -----*/
```

函数实现

```
1 /**
```

2 函数名称: CreateGraph

3 初始条件: 邻接表G不存在, V为点, VR为图的树

4 操作结果: 创建邻接表G, 表示该图, 使得邻接表中每个元素分别存储所对应的顶点以及与其相邻的顶点集合

5 函数变量: ALGraph& G, VertexType V[], KeyType VR[][2]

6 */

```
7 status CreateGraph(ALGraph& G, VertexType V[], KeyType VR[][2]){
8     int i = 0;
9     memset(&G, 0, sizeof(G));
10    if (V[0].key == -1) return ERROR;
11    while (V[i].key != -1) {
12        G.vertices[i].data = V[i];
13        G.vertices[i].firstarc = NULL;
14        G.vexnum++;
15        i++;
16        if (G.vexnum > MAX_VERTEX_NUM) return ERROR;
17    }
18    i = 0;
19    while (VR[i][0] != -1) {
20        ArcNode* p; int first = 0, next = 0;
21        int flag1 = 0, flag2 = 0;
22        for (int j = 0; j < G.vexnum; j++)
23            if (VR[i][0] == G.vertices[j].data.key) {
24                flag1 = 1;
25                first = j;
26                break;
27            }
28        for (int j = 0; j < G.vexnum; j++)
29            if (VR[i][1] == G.vertices[j].data.key) {
30                flag2 = 1;
31                next = j;
32                break;
33            }
34        if (!flag1 || !flag2) return ERROR;
35        p = (ArcNode*)malloc(sizeof(ArcNode));
```

```
36         p->adjvex = next; p->nextarc = G.vertices[first].
           firstarc; G.vertices[first].firstarc = p;
37         p = (ArcNode*)malloc(sizeof(ArcNode));
38         p->adjvex = first; p->nextarc = G.vertices[next].
           firstarc; G.vertices[next].firstarc = p;
39         G.arcnum++;
40         i++;
41     }
42     return OK;
43 }
44 /**
45 函数名称: DestroyGraph
46 初始条件: 邻接表G存在
47 操作结果: 销毁G这个邻接表结构并释放相应的空间
48 函数变量: ALGraph& G
49 */
50 status DestroyGraph(ALGraph& G)
51 {
52     ArcNode* p, * q;
53     for (int i = 0; i < G.vexnum; i++) {
54         p = G.vertices[i].firstarc;
55         while (p) {
56             q = p->nextarc;
57             free(p);
58             p = q;
59         }
60     }
61     G.vexnum = 0;
62     G.arcnum = 0;
63     return OK;
64 }
65 /**
66 函数名称: LocateVex
67 初始条件: 邻接表G存在, Keytype的整数u
68 操作结果: 获取顶点U在邻接表G中的位置, 返回该点在邻接表G中的下标,
```

如果该点不存在返回-1

```
69 函数变量: ALGraph G, KeyType u
70 */
71 int LocateVex(ALGraph G, KeyType u)
72 {
73     for (int i = 0; i < G.vexnum; i++) {
74         if (u == G.vertices[i].data.key) return i;
75     }
76     return -1;
77 }
78 /**
79 函数名称: PutVex
80 初始条件: 邻接表G存在, 节点的值value存在
81 操作结果: 找到邻接表G中的顶点U, 并将其点值改为给定的value
82 函数变量: ALGraph& G, KeyType u, VertexType value
83 */
84 status PutVex(ALGraph& G, KeyType u, VertexType value){
85
86     for (int i = 0; i < G.vexnum; i++) {
87         if (value.key == G.vertices[i].data.key) {
88             return ERROR;
89         }
90     }
91     for (int i = 0; i < G.vexnum; i++) {
92         if (u == G.vertices[i].data.key) {
93             G.vertices[i].data = value;
94             return OK;
95         }
96     }
97     return ERROR;
98 }
99 /**
100 函数名称: FirstAdjVex
101 初始条件: 邻接表G存在, Keytype的整数u
102 操作结果: 找到顶点U的第一个邻接点并返回该邻接点的下标, 如果U不存
```

在或者U没有邻接点则返回-1

```

103 函数变量: ALGraph G, KeyType u
104 */
105 int FirstAdjVex(ALGraph G, KeyType u){
106     for (int i = 0; i < G.vexnum; i++) {
107         if (u == G.vertices[i].data.key) {
108             if (G.vertices[i].firstarc) return G.
                vertices[i].firstarc->adjvex;
109         }
110     }
111     return -1;
112 }
113 /**
114 函数名称: NextAdjVex
115 初始条件: 邻接表G存在, 节点v与节点w存在
116 操作结果: 找到与节点U相邻的节点V在与W的邻接表中的下一个邻接节点并
    返回该邻接点的下标,没有下一个则返回-1
117 函数变量: ALGraph G, KeyType v, KeyType w
118 */
119 int NextAdjVex(ALGraph G, KeyType v, KeyType w){
120     for (int i = 0; i < G.vexnum; i++) {
121         if (v == G.vertices[i].data.key) {
122             ArcNode* p = G.vertices[i].firstarc;
123             while (p) {
124                 if (G.vertices[p->adjvex].data.
                    key == w) {
125                     if (p->nextarc != NULL)
126                         return p->nextarc->
                            adjvex;
127                     else return -1;
128                 }
129                 p = p->nextarc;
130             }
131         }
132     }
133     return -1;
134 }

```

```
132         return -1;
133     }
134     /**
135     函数名称: InsertVex
136     初始条件: 邻接表G存在, 新顶点v存在
137     操作结果: 在邻接表G中插入一个新节点V
138     函数变量: ALGraph& G, VertexType v
139     */
140     status InsertVex(ALGraph& G, VertexType v){
141         if (G.vexnum == MAX_VERTEX_NUM) return ERROR;
142         if (LocateVex(G, v.key) != -1) return ERROR;
143         G.vertices[G.vexnum].data = v;
144         G.vertices[G.vexnum].firstarc = NULL;
145         G.vexnum++;
146         return OK;
147     }
148     /**
149     函数名称: DeleteVex
150     初始条件: 邻接表G存在, 需要删除的节点u存在
151     操作结果: 删除邻接表G中节点U以及所有与点U相关联的边
152     函数变量: ALGraph& G, KeyType u
153     */
154     status DeleteVex(ALGraph& G, KeyType v){
155         if (G.vexnum <= 1) return ERROR;
156         int i, j;
157         ArcNode* p, *q = NULL;
158         j = LocateVex(G, v);
159         if (j == -1) return ERROR;
160         p = G.vertices[j].firstarc;
161         while (p) {
162             q = p;
163             p = p->nextarc;
164             free(q);
165             G.arcnum--;
166         }
```

```
167         for (i = j; i < G.vexnum; i++) G.vertices[i] = G.vertices
           [i + 1];
168     G.vexnum--;
169     for (i = 0; i < G.vexnum; i++) {
170         p = G.vertices[i].firstarc;
171         while (p) {
172             if (p->adjvex == j) {
173                 if (p == G.vertices[i].firstarc)
174                     {
175                         G.vertices[i].firstarc =
176                             p->nextarc;
177                         free(p);
178                         p = G.vertices[i].
179                             firstarc;
180                     }
181                 else {
182                     q->nextarc = p->nextarc;
183                     free(p);
184                     p = q->nextarc;
185                 }
186             }
187             else {
188                 if (p->adjvex > j) p->adjvex--;
189                 q = p;
190                 p = p->nextarc;
191             }
192         }
193     }
194     return OK;
195 }
```

193 /**

194 函数名称: InsertArc

195 初始条件: 邻接表G存在, 节点U与节点V之间的边存在

196 操作结果: 在邻接表G中添加一条从节点U指向节点V的边

197 函数变量: ALGraph& G, KeyType u, KeyType v

```
198 */
199 status InsertArc(ALGraph& G, KeyType v, KeyType w){
200     ArcNode* p;
201     int i = 0, j = 0;
202     i = LocateVex(G, v);
203     j = LocateVex(G, w);
204     if (i == -1 || j == -1) return ERROR;
205     p = G.vertices[i].firstarc;
206     while (p) {
207         if (p->adjvex == j) return ERROR;
208         p = p->nextarc;
209     }
210     G.arcnum++;
211     p = (ArcNode*)malloc(sizeof(ArcNode));
212     p->adjvex = j;
213     p->nextarc = G.vertices[i].firstarc;
214     G.vertices[i].firstarc = p;
215     p = (ArcNode*)malloc(sizeof(ArcNode));
216     p->adjvex = i;
217     p->nextarc = G.vertices[j].firstarc;
218     G.vertices[j].firstarc = p;
219     return OK;
220 }
221 /**
222 函数名称: DeleteArc
223 初始条件: 邻接表G存在, 节点U与节点V之间的边存在
224 操作结果: 删除所有在邻接表G中从U指向V的边
225 函数变量: ALGraph& G, KeyType u, KeyType v
226 */
227 status DeleteArc(ALGraph& G, KeyType v, KeyType w){
228     ArcNode* p, *q = NULL;
229     int i = 0, j = 0;
230     i = LocateVex(G, v);
231     j = LocateVex(G, w);
232     if (i == -1 || j == -1) return ERROR;
```



```
233     p = G.vertices[i].firstarc;
234     while (p && p->adjvex != j) {
235         q = p;
236         p = p->nextarc;
237     }
238     if (p && p->adjvex == j) {
239         if (p == G.vertices[i].firstarc) G.vertices[i].
            firstarc = p->nextarc;
240         else q->nextarc = p->nextarc;
241         free(p);
242         G.arcnum--;
243     }
244     p = G.vertices[j].firstarc;
245     while (p && p->adjvex != i)
246     {
247         q = p;
248         p = p->nextarc;
249     }
250     if (p && p->adjvex == i) {
251         if (p == G.vertices[j].firstarc) G.vertices[j].
            firstarc = p->nextarc;
252         else q->nextarc = p->nextarc;
253         free(p);
254     }
255     return OK;
256 }
257 /**
258 函数名称: DFSTraverse
259 初始条件: 邻接表G存在, visit函数已经实现
260 操作结果: 对邻接表G进行深度优先遍历, 并打印遍历结果, visit函数用
            于对节点进行操作
261 函数变量: ALGraph& G, void (*visit)(VertexType)
262 */
263 void DFS(ALGraph G, int v, void (*visit)(VertexType)) {
264     visited[v] = 1;
```

```
265     visit(G.vertices[v].data);
266     for (int i = FirstAdjVex(G, G.vertices[v].data.key); i !=
        -1; i = NextAdjVex(G, G.vertices[v].data.key, G.
            vertices[i].data.key))
267         if (!visited[i]) DFS(G, i, visit);
268 }
269 void visit(VertexType v){
270     printf(" %d,%s", v.key, v.others);
271 }
272 status DFSTraverse(ALGraph& G, void (*visit)(VertexType))
273 {
274     memset(visited, 0, sizeof(visited));
275     for (int i = 0; i < G.vexnum; i++) if (!visited[i]) DFS(G
        , i, visit);
276     return OK;
277 }
278 status InitQueue(LinkQueue& Q) {
279     Q.front1 = Q.rear = (QueuePtr)malloc(sizeof(QNode));
280     if (!Q.front1) exit(OVERFLOW);
281     Q.front1->next = NULL;
282     return OK;
283 }
284 status DestroyQueue(LinkQueue& Q) {
285     while (Q.front1) {
286         Q.rear = Q.front1->next;
287         free(Q.front1);
288         Q.front1 = Q.rear;
289     }
290     return OK;
291 }
292 status ClearQueue(LinkQueue& Q) {
293     QueuePtr p, q;
294     Q.rear = Q.front1;
295     p = Q.front1->next;
296     Q.front1->next = NULL;
```

```
297     while (p) {
298         q = p;
299         p = p->next;
300         free(q);
301     }
302     return OK;
303 }
304 status QueueEmpty(LinkQueue Q) {
305     if (Q.front1 == Q.rear)
306         return TRUE;
307     else
308         return FALSE;
309 }
310 int QueueLength(LinkQueue Q) {
311     QueuePtr p;
312     p = Q.front1;
313     int i = 0;
314     while (Q.rear != p) {
315         i++;
316         p = p->next;
317     }
318     return i;
319 }
320 status EnQueue(LinkQueue& Q, QElemType e) {
321     QueuePtr p = (QueuePtr)malloc(sizeof(QNode));
322     if (!p) exit(OVERFLOW);
323     p->data = e;
324     p->next = NULL;
325     Q.rear->next = p;
326     Q.rear = p;
327     return OK;
328 }
329 status DeQueue(LinkQueue& Q, QElemType& e) {
330     QueuePtr p;
331     if (Q.front1 == Q.rear) return ERROR;
```

```
332     p = Q.front1->next;
333     e = p->data;
334     Q.front1->next = p->next;
335     if (Q.rear == p) Q.rear = Q.front1;
336     free(p);
337     return OK;
338 }
339 /**
340 函数名称: BFSTraverse
341 初始条件: 邻接表G存在, visit函数已经实现
342 操作结果: 对邻接表G进行广度优先遍历, 并打印遍历结果, visit函数用于对节点进行操作
343 函数变量: ALGraph& G, void (*visit)(VertexType)
344 */
345 status BFSTraverse(ALGraph& G, void(*visit)(VertexType)) {
346     int i, j, w;
347     LinkQueue Q;
348     memset(visited, 0, sizeof(visited));
349     InitQueue(Q);
350     for (i = 0; i < G.vexnum; i++)
351     if (!visited[i]) {
352         visited[i] = 1;
353         visit(G.vertices[i].data);
354         EnQueue(Q, i);
355         while (!QueueEmpty(Q)) {
356             DeQueue(Q, j);
357             for (w = FirstAdjVex(G, G.vertices[j].
                 data.key); w != -1; w = NextAdjVex(G,
                 G.vertices[j].data.key, G.vertices[w].
                 data.key))
358             if (!visited[w]) {
359                 visited[w] = 1;
360                 visit(G.vertices[w].data);
361                 EnQueue(Q, w);
362             }
```

```
363         }
364     }
365     return OK;
366 }
367 /**
368 函数名称: SaveGraph
369 初始条件: 邻接表G存在, 以及一个待存储为文件的文件名FileName[]
370 操作结果: 指定文件名把邻接表G存为文件
371 函数变量: ALGraph G, char FileName[]
372 */
373 status SaveGraph(ALGraph G, char FileName[]){
374     int i = 0;
375     FILE* fp = fopen(FileName, "w");
376     ArcNode* p;
377     fprintf(fp, "%d %d %d\n", G.vexnum, G.arcnum, G.kind);
378     for (i = 0; i < G.vexnum; i++) {
379         fprintf(fp, "%d %s\n", G.vertices[i].data.key, G.
            vertices[i].data.others);
380         if ((p = G.vertices[i].firstarc) != NULL) {
381             fprintf(fp, "%d ", p->adjvex);
382             while ((p = p->nextarc) != NULL) {
383                 fprintf(fp, "%d ", p->adjvex);
384             }
385         }
386         fprintf(fp, "%d\n", -1);
387     }
388     fclose(fp);
389     return OK;
390 }
391 status Load(ALGraph& G, char FileName[]){
392     int i, num = 0;
393     FILE* fp = fopen(FileName, "r");
394     ArcNode* p;
395     fscanf(fp, "%d %d %d", &G.vexnum, &G.arcnum, &G.kind);
396     for (i = 0; i < G.vexnum; i++) {
```

```
397         fscanf(fp, "%d %s", &G.vertices[i].data.key, &G.
           vertices[i].data.others);
398         fscanf(fp, "%d", &num);
399         G.vertices[i].firstarc = NULL;
400         while (num != -1) {
401             p = (ArcNode*)malloc(sizeof(ArcNode));
402             p->adjvex = num;
403             p->nextarc = G.vertices[i].firstarc;
404             G.vertices[i].firstarc = p;
405             fscanf(fp, "%d", &num);
406         }
407     }
408     fclose(fp);
409     return OK;
410 }
411 /**
412 函数名称: LoadGraph
413 初始条件: 一个待读取数据文件的文件名FileName[]
414 操作结果: 从指定的文件名中读取图的信息, 并生成一个邻接表G来表示该
           图
415 函数变量: ALGraph& G, char FileName[]
416 */
417 status LoadGraph(ALGraph& G, char FileName[]) {
418     Load(G, FileName);
419     SaveGraph(G, FileName);
420     Load(G, FileName);
421     return OK;
422 }
423 /**
424 函数名称: VerticesSetLessThanK
425 初始条件: 邻接表G存在, 节点v以及整数k存在
426 操作结果: 找到邻接表G中与节点v相连通且边权小于等于k的点集 (不包括
           v)
427 函数变量: ALGraph G, KeyType v, int k
428 */
```

```
429 status VerticesSetLessThanK(ALGraph G, KeyType key, int k) {
430     LinkQueue q, nextlevel;
431     InitQueue(q);
432     InitQueue(nextlevel);
433     int count = 0;
434     memset(visited, 0, sizeof(visited));
435     EnQueue(q, LocateVex(G, key));
436     visit(G.vertices[LocateVex(G, key)].data);
437     visited[LocateVex(G, key)] = 1;
438     while (count < k - 1) {
439         while (!QueueEmpty(q)) {
440             int j = 0;
441             DeQueue(q, j);
442             for (int w = FirstAdjVex(G, G.vertices[j]
                                     ].data.key); w != -1; w = NextAdjVex(G
                                     , G.vertices[j].data.key, G.vertices[w
                                     ].data.key))
443                 if (!visited[w]) {
444                     visited[w] = 1;
445                     visit(G.vertices[w].data);
446                     EnQueue(nextlevel, w);
447                 }
448             }
449         while (!QueueEmpty(nextlevel)) {
450             int n = 0;
451             DeQueue(nextlevel, n);
452             EnQueue(q, n);
453         }
454         ClearQueue(nextlevel);
455         count++;
456     }
457     return OK;
458 }
459 /**
460 函数名称: ShortestPathLength
```

```
461 初始条件：邻接表G存在，节点v以及节点w存在
462 操作结果：找到从节点v到节点w的路径的最短长度，并返回该最短路径长
    度
463 函数变量：ALGraph G, KeyType v, KeyType w
464 */
465 status ShortestPathLength(ALGraph G, VertexType v, VertexType w)
466 {
467     int i = 0, j = 0, locatev = -1, locatew = -1, minid = 0,
        min = -1;
468     const int inf = 1 << 30;
469     ArcNode* trans = NULL;
470     int dist[MAX_VERTEX_NUM] = { 0 };
471     int book[MAX_VERTEX_NUM] = { 0 };
472     if (G.vertices == NULL) { return -1; }
473     for (i = 0; i < G.vexnum; i++)
474     {
475         if (v.key == G.vertices[i].data.key) { locatev =
            i; }
476         if (w.key == G.vertices[i].data.key) { locatew =
            i; }
477     }
478     if (locatew == -1 || locatev == -1) { return -1; }
479     book[locatev]++;
480     for (i = 0; i < G.vexnum; i++) { dist[i] = inf; }
481     trans = G.vertices[locatev].firstarc;
482     dist[locatev] = 0;
483     while (trans != NULL)
484     {
485         dist[trans->adjvex] = 1;
486         trans = trans->nextarc;
487     }
488     for (i = 0; i < G.vexnum - 1; i++)
489     {
490         min = inf;
491         for (j = 0; j < G.vexnum; j++)
```



```
492         {
493             if (book[j] == 0 && min > dist[j])
494             {
495                 minid = j;
496                 min = dist[j];
497             }
498         }
499         book[minid]++;
500         trans = G.vertices[minid].firstarc;
501         while (trans != NULL)//尝试用这个点松弛和它邻近的
           点。
502         {
503             if (dist[trans->adjvex] > 1 + dist[minid
           ])
504             {
505                 dist[trans->adjvex] = 1 + dist[
           minid];
506             }
507             trans = trans->nextarc;
508         }
509     }
510     if (dist[locatew] == inf) { return -1; }//两点不连通。
511     return dist[locatew];
512 }
513 void bfs(ALGraph G, int startnode)
514 {
515     int head = 0, tail = 0;
516     int queue[MAX_VERTEX_NUM] = { startnode };
517     while (head <= tail)
518     {
519         ArcNode* trans = G.vertices[queue[head]].firstarc
           ;
520         while (trans != NULL)
521         {
522             if (book[trans->adjvex] == 0)
```

```
523         {
524             tail++;
525             queue[tail] = trans->adjvex;
526             book[trans->adjvex]++;
527         }
528         trans = trans->nextarc;
529     }
530     head++;
531 }
532 }
533 /**
534 函数名称: ConnectedComponentsNums
535 初始条件: 邻接表G存在
536 操作结果: 输出邻接表G的连通分量个数
537 函数变量: ALGraph G
538 */
539 status ConnectedComponentsNums(ALGraph G)
540 {
541     int i = 0, cnt = 0;
542     for (i = 0; i < MAX_VERTEX_NUM; i++) { book[i] = 0; }
543     if (G.vertices == NULL) { return 0; }
544     for (i = 0; i < G.vexnum; i++)
545     {
546         if (book[i] == 0)
547         {
548             book[i]++;
549             cnt++;
550             bfs(G, i);
551         }
552     }
553     return cnt;
554 }
```