

**课 程 实 验 报 告**

**课程名称： 数据结构实验**

**专业班级： 计算机科学与技术201904**

**学 号： U201915058**

**姓 名： 胡钧宇**

**指导教师： 袁凌**

**报告日期： 2020年 10月 11 日**

**计算机科学与技术学院**

目录

[1 基于顺序存储结构的线性表实现 3](#_Toc59554314)

[1.1 问题描述 3](#_Toc59554315)

[1.1.1 问题框架 3](#_Toc59554316)

[1.2 系统设计 3](#_Toc59554317)

[1.3 系统实现 4](#_Toc59554318)

[1.4 系统测试 15](#_Toc59554319)

[1.5 实验小结 20](#_Toc59554320)

[2 基于链式存储结构的线性表实现 22](#_Toc59554321)

[2.1 问题描述 22](#_Toc59554322)

[2.2 系统设计 22](#_Toc59554323)

[2.3 系统实现 23](#_Toc59554324)

[2.4 系统测试 35](#_Toc59554325)

[2.5 实验小结 39](#_Toc59554326)

[3 基于二叉链表的二叉树实现 41](#_Toc59554327)

[3.1 问题描述 41](#_Toc59554328)

[3.1.1 问题框架 41](#_Toc59554329)

[3.2 系统设计 41](#_Toc59554330)

[3.3 系统实现 42](#_Toc59554331)

[3.4 系统测试 52](#_Toc59554332)

[3.5 实验小结 62](#_Toc59554333)

[4 基于邻接表的图实现 64](#_Toc59554334)

[4.1 问题描述 64](#_Toc59554335)

[4.2 系统设计 64](#_Toc59554336)

[4.3 系统实现 66](#_Toc59554337)

[4.4 系统测试 72](#_Toc59554338)

[4.5 实验小结 84](#_Toc59554339)

[参考文献 86](#_Toc59554340)

[附录A 基于顺序存储结构线性表实现的源程序 87](#_Toc59554341)

[def.h 131](#_Toc59554342)

[Stack.h 132](#_Toc59554343)

[fun.h 133](#_Toc59554344)

[实验三.cpp 156](#_Toc59554345)

[附录B 基于链式存储结构线性表实现的源程序 163](#_Toc59554346)

[附录C：基于二叉树、二叉链表的源程序的实现 187](#_Toc59554347)

[def.h 187](#_Toc59554348)

[Stack.h 188](#_Toc59554349)

[fun.h 189](#_Toc59554350)

[实验三.cpp 211](#_Toc59554351)

[附录D 基于邻接表图实现的源程序 218](#_Toc59554352)

[def\_Graph.h 218](#_Toc59554353)

[tool.h 220](#_Toc59554354)

[func.h 224](#_Toc59554355)

[shiyan4.cpp 243](#_Toc59554356)

# 1 基于顺序存储结构的线性表实现

## 1.1 问题描述

线性表是顺序存储结构中最基本、最简单，也是最常用的一种数据结构为了加深我们对线性表的理解与运用，在这次实验中，我们采用物理结构，构造一个具有菜单的功能演示系统。其中，在主程序中完成函数调用所需实参值的准备和函数执行结果的显示。定义了线性表的初始化表、销毁线性表、清空线性表、判定空表、求表长和获得元素等基本操作对应的函数，并给出适当的操作提示显示，可以选择以文件的形式进行存储和加载，也可以将线性表存入相应的文件中，从文件中获得线性表

## 1.1.1 问题框架

根据线性表的顺序存储方式，即是顺序表，并结合其实际应用中数据管理方式，至少应定义：初始化表、销毁表、清空表、判定空表、求表长、获取指定位置元素、查找元素、获得前驱、获得后继、插入元素、删除元素、遍历这十二个基本操作。

## 1.2 系统设计

以上问题框架中的十二个基本操作，其具体运算功能定义如下。

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

⑵销毁表：函数名称是DestroyList(L)；初始条件是线性表L已存在；操作结果是销毁线性表L。

⑶清空表：函数名称是ClearList(L)；初始条件是线性表L已存在；操作结果是将L重置为空表。

⑷判定空表：函数名称是ListEmpty(L)；初始条件是线性表L已存在；操作结果是若L为空表则返回TRUE,否则返回FALSE。

⑸求表长：函数名称是ListLength(L)；初始条件是线性表已存在；操作结果是返回L中数据元素的个数。

⑹获得元素：函数名称是GetElem(L,i,e)；初始条件是线性表已存在，1≤i≤ListLength(L)；操作结果是用e返回L中第i个数据元素的值。

⑺查找元素：函数名称是LocateElem(L,e)；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满足相等的数据元素的位序，若这样的数据元素不存在，则返回值为0。

⑻获得前驱：函数名称是PriorElem(L,cur\_e,pre\_e)；初始条件是线性表L已存在；操作结果是若cur\_e是L的数据元素，且不是第一个，则用pre\_e返回它的前驱，否则操作失败，pre\_e无定义。

⑼获得后继：函数名称是NextElem(L,cur\_e,next\_e)；初始条件是线性表L已存在；操作结果是若cur\_e是L的数据元素，且不是最后一个，则用next\_e返回它的后继，否则操作失败，next\_e无定义。

⑽插入元素：函数名称是ListInsert(L,i,e)；初始条件是线性表L已存在，1≤i≤ListLength(L)+1；操作结果是在L的第i个位置之前插入新的数据元素e。

⑾删除元素：函数名称是ListDelete(L,i,e)；初始条件是线性表L已存在且非空，1≤i≤ListLength(L)；操作结果：删除L的第i个数据元素，用e返回其值。

⑿遍历表：函数名称是ListTraverse(L)，初始条件是线性表L已存在；操作结果是依次对L的每个数据元素调用函数。

## 1.3 系统实现

本系统通过命令式菜单来调用各函数功能，系统开始运行时会输出菜单界面，依照菜单中给出的各功能的指令输入命令，以实现相关功能的使用。

总体实现为：

首先根据每一个基本操作写出相关函数的基本功能实现，后再针对异常状况进行合理处理，完成每一个函数的具体实现。

后在main函数中定义一个op存储当前命令，通过switch语句，针对op的不同值在case op中调用对应函数以实现相应功能。

每个函数的具体实现方法如下：

⑴初始化表InitList(L)：首先利用malloc函数在内存中寻找一片大小为LIST\_INIT\_SIZE\*sizeof(ElemType)的存储空间，将其首地址赋给目前不存在的线性表L中的数组首地址，后将L.length置0，L.listsize置为LIST\_INIT\_SIZE，之后返回OK。(异常处理：当L已存在时，便不可再次初始化，此时需返回INFEASIBLE;)

(2) 销毁表DestroyList(L)：首先将线性表中数组的首地址利用free函数将其释放，之后将当前线性表中的数组首地址置为NULL，并将线性表长度置0，线性表大小置0。

⑶清空表ClearList(L)：首先将当前线性表摧毁(执行(2))，之后再次将线性表L初始化即可。

⑷判定空表ListEmpty(L)：当线性表不存在时，返回INFEASIBLE；当线性表长度为0且线性表存在时，即表明线性表为空，返回TRUE；当线性表长度不为0时，线性表非空，返回FALSE。

⑸求表长ListLength(L)：当线性表不存在时，函数返回INFEASIBLE；否则返回线性表长度即可。

⑹获得元素GetElem(L,i,e)：当线性表不存在时，函数返回INFEASIBLE；当线性表存在，但是i < 1或者i > L.length时，选取的位置非法，故返回ERROR；当线性表存在且i >= 1, i <= L.length时(即选取位置合法时)，将L.elem[i]中数据赋值到e，返回OK即可。

⑺查找元素LocateElem(L,e):当线性表不存在时，函数返回INFEASIBLE；当线性表存在时，扫描线性表中所有元素，并在扫描过程中将每一个已在线性表中的元素与e比较，如果相等则返回当前元素在数组中的序号+1，如果扫描过程中没有任何一个元素与e相等，查找失败，则返回ERROR。程序框图如图1-7：

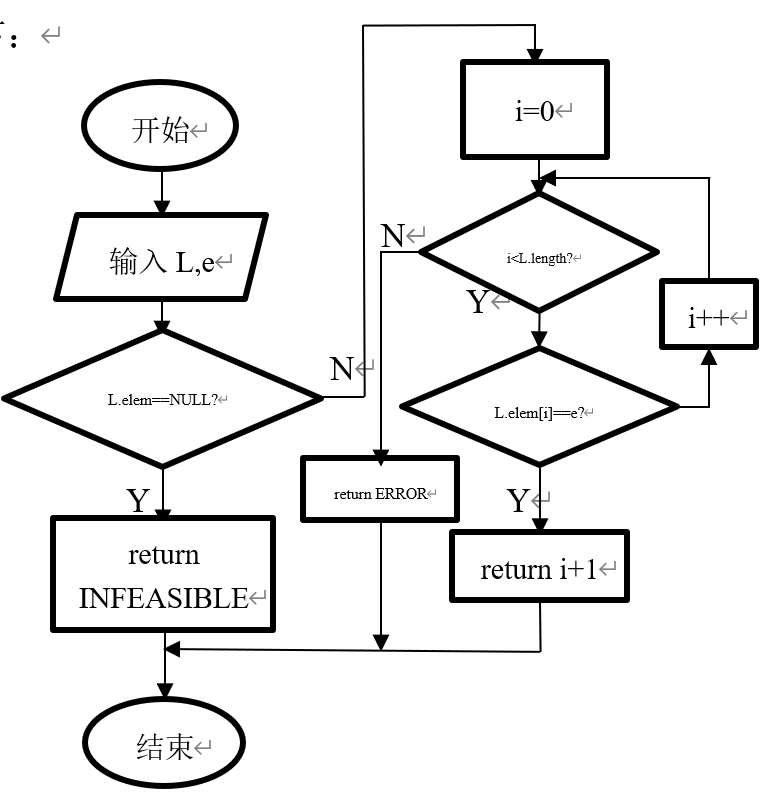


图1-7 查找元素

⑻获得前驱PriorElem(L,cur\_e,pre\_e)：当线性表不存在时，函数返回INFEASIBLE; 当线性表存在时，扫描线性表，且扫描过程中比较当前位置元素L.elem[i]是否与cur\_e相等，如果相同，则继续判断i是否为0，如果为0，则返回ERROR，表明此元素无前驱；如果i不为0，则将L.elem[i – 1]的值赋给pre\_e，并返回OK；此外，若线性表扫描过程中不存在i使得L.elem[i]等于cur\_e，则函数返回ERROR，表明不存在此元素，当然在线性表中也就不存在它的前驱。(框图如图1-8)

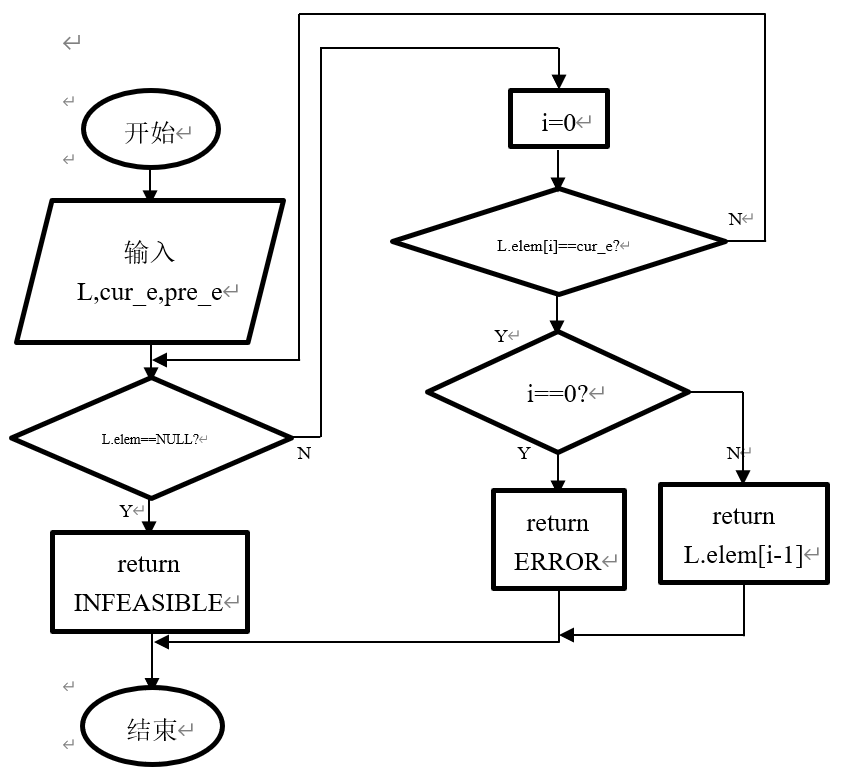


图1-8 获得前驱

⑼获得后继NextElem(L,cur\_e,next\_e)：当线性表不存在时，函数返回INFEASIBLE; 当线性表存在时，扫描线性表，且扫描过程中比较当前位置元素L.elem[i]是否与cur\_e相等，如果相同，则继续判断i是否为L.length - 1，如果为L.length - 1，表明这已经是线性表最后一个元素，无后继，函数返回ERROR；如果i不为L.length - 1，则将L.elem[i + 1]的值赋给next\_e，并返回OK；此外，若线性表扫描过程中不存在i使得L.elem[i]等于cur\_e，则函数返回ERROR，表明不存在此元素，当然在线性表中也就不存在它的后继。(框图如图1-9)

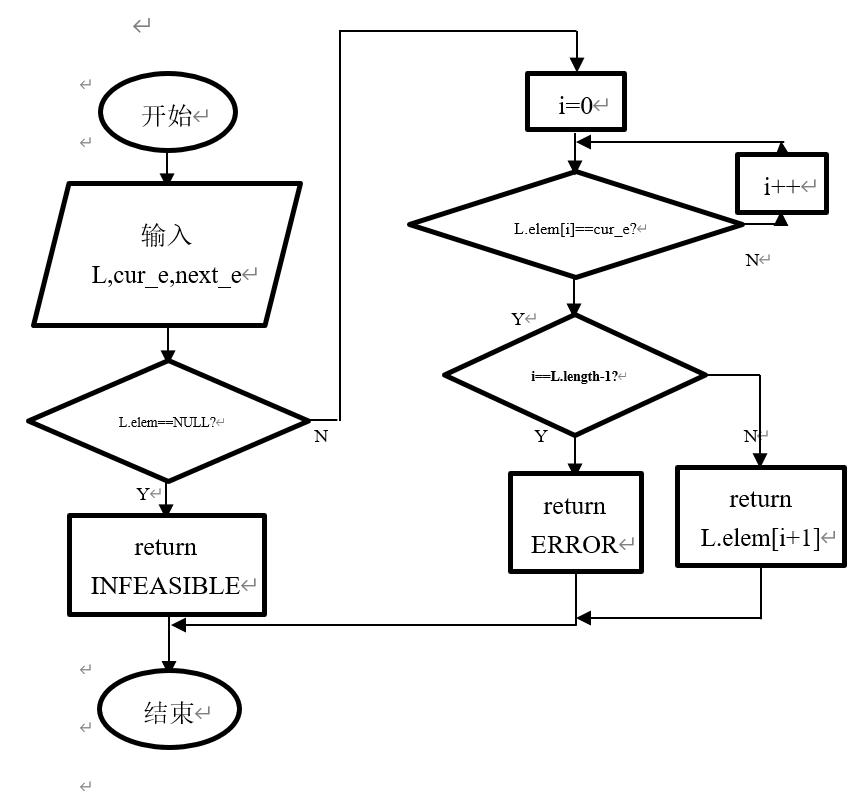


图1-9 获得后继

⑽插入元素ListInsert(L,i,e)：当线性表不存在时，函数返回INFEASIBLE；当线性表存在但是i < 1或者i > L.length + 1时，插入位置非法，函数返回ERROR；当线性表存在且i >= 1, i <= L.length + 1时，首先判断L.length + 1是否大于L.listsize，如果是，则应利用realloc函数对该线性表的数组重新分配大小为LIST\_INIT\_SIZE\*sizeof(ElemType)的存储空间，之后将原数组中地址位于[ L.elem + i - 1, L.elem + L.length – 1 ]中的元素全部向地址增大的方向移动一位，之后将e的值赋给L.elem[i – 1]，函数返回OK即可；如果L.length + 1 <= L.listsize，表示内存空间足够，只需要将原数组中地址位于[ L.elem + i, L.elem + L.length – 1 ]中的元素全部向地址增大的方向移动一位，之后将e的值赋给L.elem[i – 1]，函数返回OK即可。(框图如图1-10)

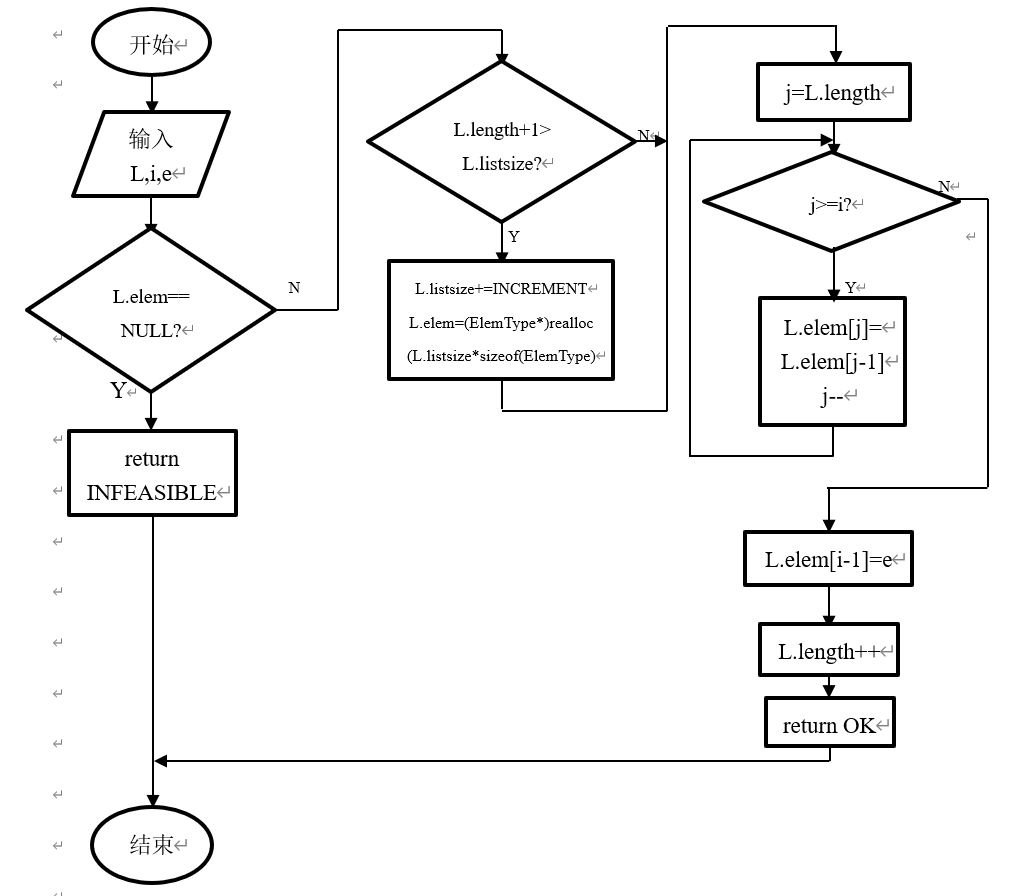


图1-10 插入元素

⑾删除元素ListDelete(L,i,e)：当线性表不存在时，函数返回INFEASIBLE；当线性表存在但是i < 1或者i > L.length时，删除位置非法，函数返回ERROR；当线性表存在且i >= 1, i <= L.length时，将数组中地址位于[ L.elem + i, L.elem + L.length – 1 ]的所有元素向地址减小的方向移动一位，后将原线性表长度减1即可。(框图如图1-11)

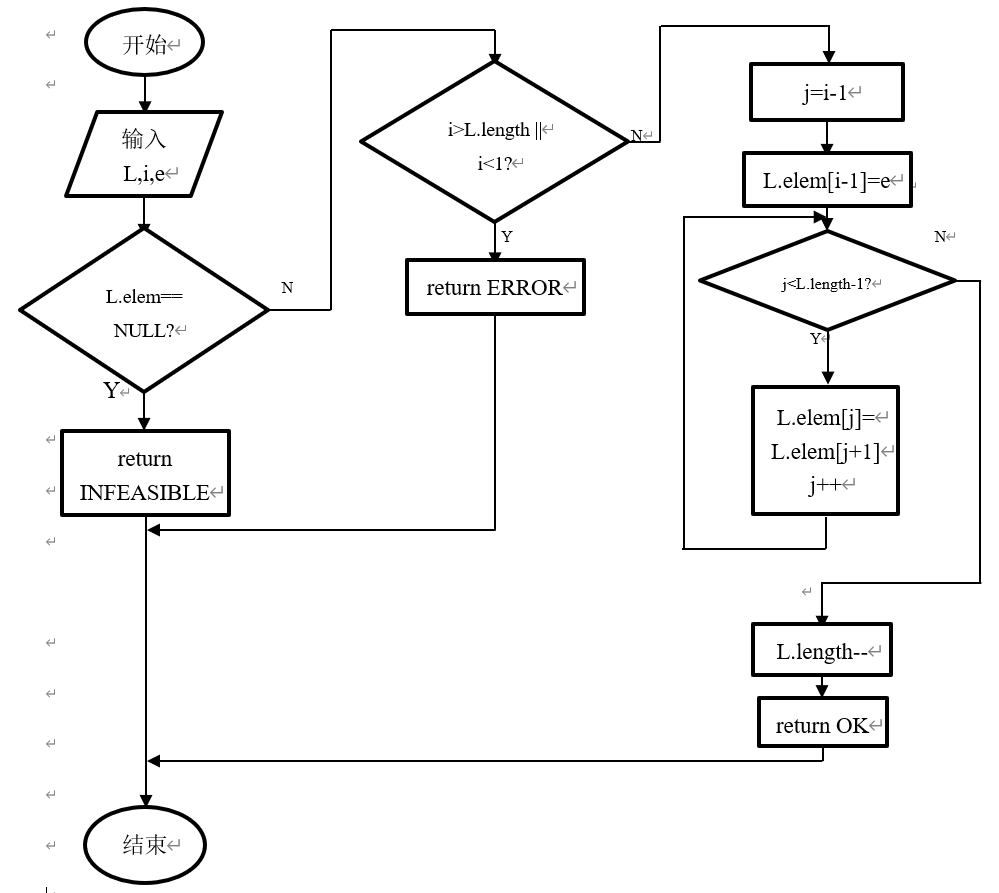


图1-11 删除元素

⑿遍历表ListTraverse(L)：当线性表不存在时，函数返回INFEASIBLE；当线性表存在且长度为0时，输出“空”，函数返回OK；当线性表存在且长度不为0时，利用循环结构将线性表扫描一遍，并输出线性表中每一个元素的值，函数返回OK。(框图如图1-12)

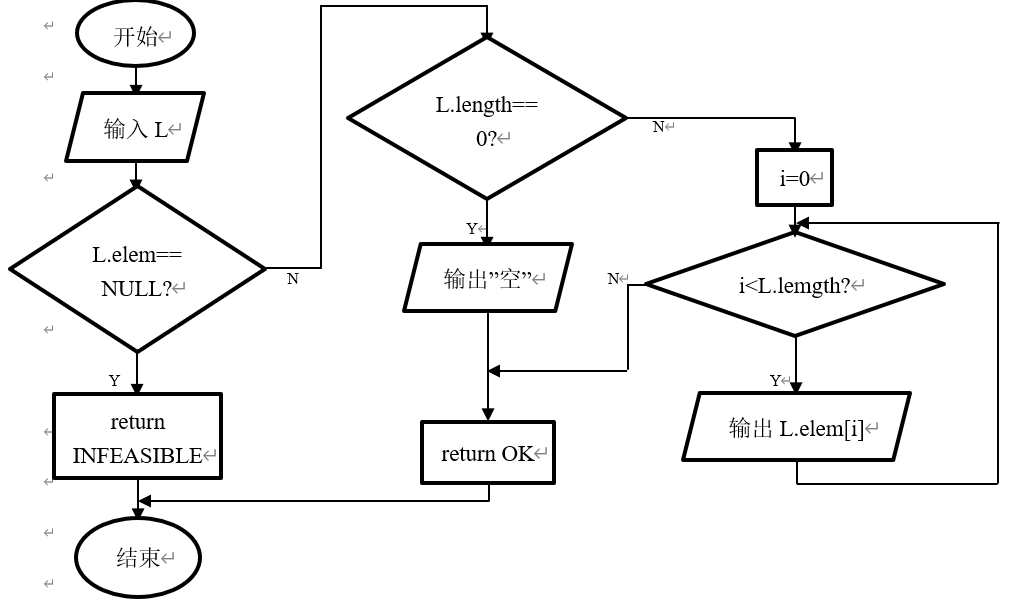


图1-12 遍历线性表

(13)输入一组元素InitInput(L):考虑到初始化时，用户可能会与此同时进行赋值操作，故它的调用是紧接在InitList(L)返回OK之后。此函数首先判断用户输入命令，命令为0时表示不输入数据，返回0即可；命令为1时表示输入数据，(此处假设用户输入数据不包含0)，则利用循环结构输入数据，同时将数据存入线性表中，并判断线性表是否已满，若线性表满，则需利用realloc进行扩容操作，直到输入为0时循环停止，函数返回OK。(如图1-13)

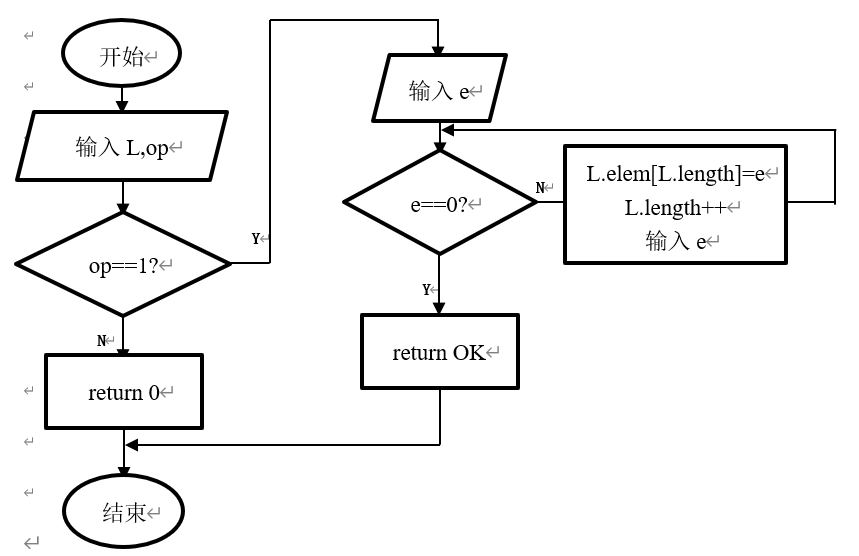


图1-13 输入一组元素进行初始化

此外，为了使系统功能更加丰富、更有实用性，我添加了部分功能模块，具体如下：

文件读取篇：

(13)线性表数据存入文件SaveList(L, SaveFile):利用文件指针以”w+”的形式打开(文件不存在时自动创建)文件名为SaveFile的文件，如果此步打开文件失败则函数返回ERROR，否则就将线性表L扫描一遍，并将每一个元素输入到目标文件中，且每输入一个元素后就立即输入一个空格，否则会导致存入文件中的数据连在一起难以取出。输入完毕后利用fclose函数关闭该目标文件。最后返回OK；

(14)读取文件中数据存入线性表LoadList(L, LoadFile)：利用文件指针以”r”的形式打开文件名为LoadFile的文件，如果此步打开文件失败则函数返回ERROR，否则就将文件中每一个数据利用fscanf函数读入线性表，读取完毕后利用fclose函数关闭该目标文件。最后返回OK；

线性表集合操作篇：

新定义了一个菜单函数gather\_operate(Lists),其大体结构与main函数相同，将八种线性表集合操作集于其中，返回值类型为一个线性表指针。(线性表修改操作需利用此返回值才得以实现)

(15)线性表集合初始化InitLists(Lists):首先判断原线性表集合中第一个线性表是否存在，若存在则表明线性表集合中可能已存在数据，不可再次进行初始化，返回INFEASIBLE;否则就将线性表集合长度置为0，对第一个线性表，利用之前已定义的InitList(L)函数将其初始化后，返回OK即可。

(16)向集合中新增线性表AddList(Lists, ListName)：首先判断，Lists当前长度是否已经是最大容量，如果是，则返回OVERFLOW，如果不是，则先利用strcpy函数将ListName拷贝至Lists.elem[Lists.length].name，再利用已定义的函数InitList(L)和InitInput(L)初始化Lists.elem[Lists.length].L，完成后将线性表集合的长度加一，再返回OK即可。(如图1-16)

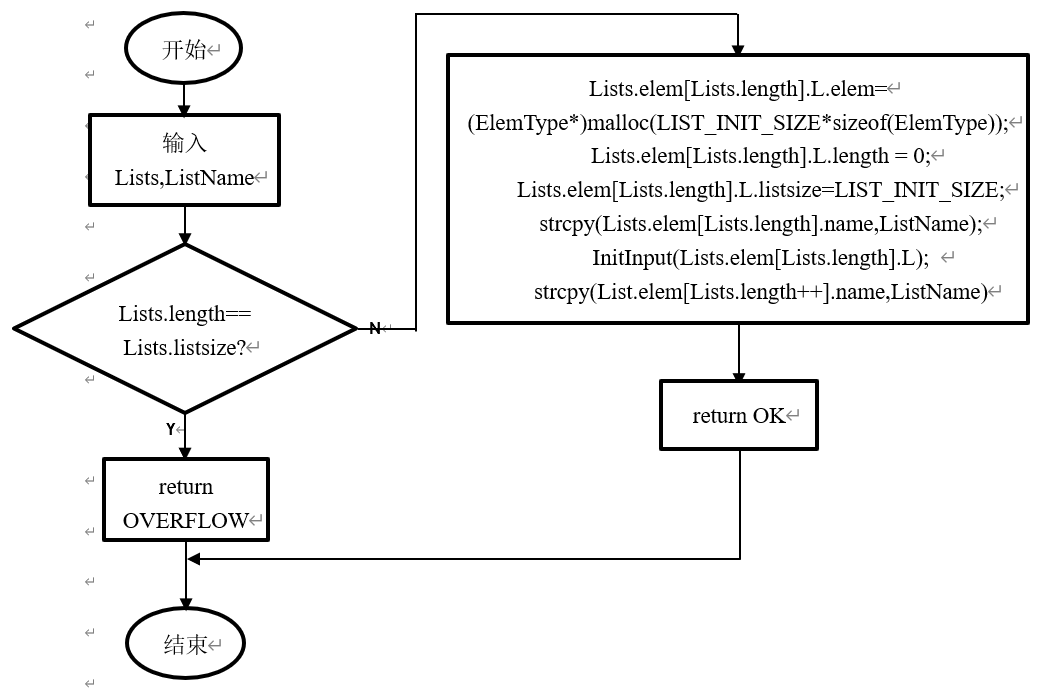


图1-16 新增线性表

(17)删除集合中名为ListName的线性表RemoveList(Lists, ListName):首先，扫描当前线性表集合，并在扫描过程中利用strcmp函数判断每一个线性表的名称是否与ListName相同，若相同则将之后的每一个线性表向前移动一位后，将集合长度减一，返回OK即可；如果扫描结束后仍未找到某个名为ListName的线性表，返回0即可。

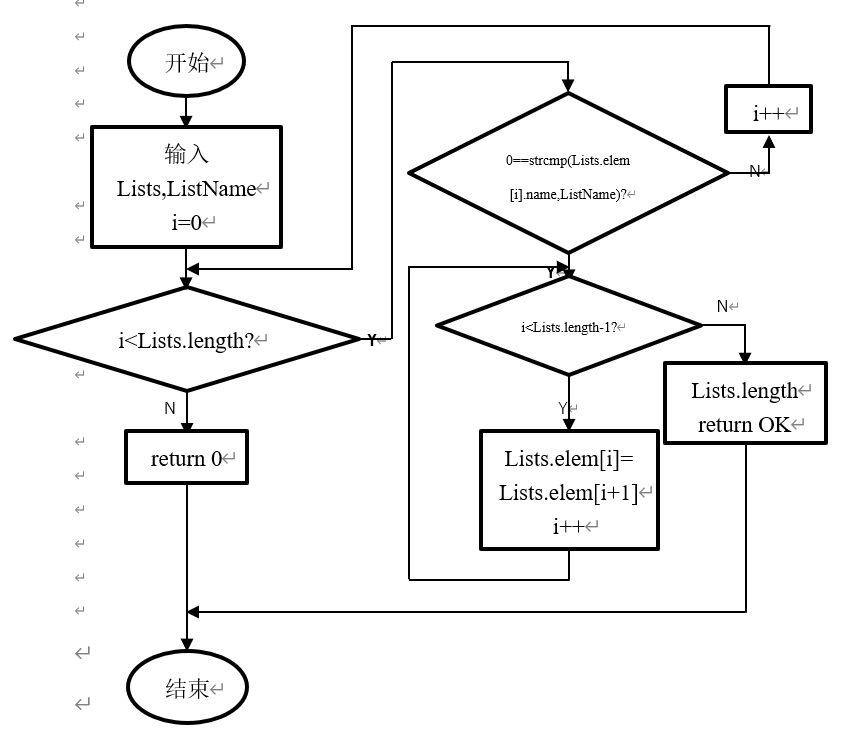


图1-17 删除线性表

(18)查找线性表LocateList(Lists, ListName)：扫描当前线性表集合，过程中利用strcmp函数判断当前线性表名称是否与ListName相同，若有相同则返回当前线性表的逻辑位置;若扫描结束仍未发现名为ListName的线性表，则返回0，表明查无此表。

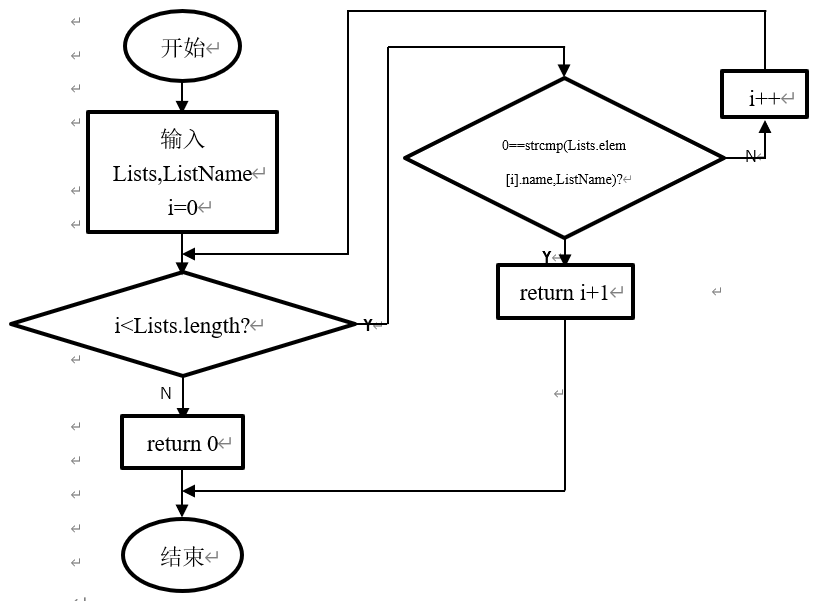


图1-18 查找线性表

(19)修改线性表：此功能无特定函数，是利用菜单函数gather\_operate(Lists)的返回值在main函数里实现，具体过程为：当用户选择修改功能时，先利用查找线性表函数LocateList(Lists,ListName)找到所要修改的表(若未找到则输出查无此表后break switch即可，无后续操作)，之后将查找到的线性表的地址作为gather\_operate函数的返回值返回到主函数中，之后，将主函数中原有线性表的地址存入一个新定义的临时线性表指针temp，再用gather\_operate函数的返回值覆盖原有线性表，之后便可对其执行线性表相关操作，操作结束后(即用户输入命令为0时)，再在main函数中将temp重新覆盖到原线性表L上即可。

(20)遍历集合ListsTraverse(Lists)：首先判断线性表集合是否存在，不存在则返回INFEASIBLE，若存在再判断长度是否为0，若为0则返回0，否则扫描线性表集合，并输出每一个线性表的名称和其中的数据，返回OK。

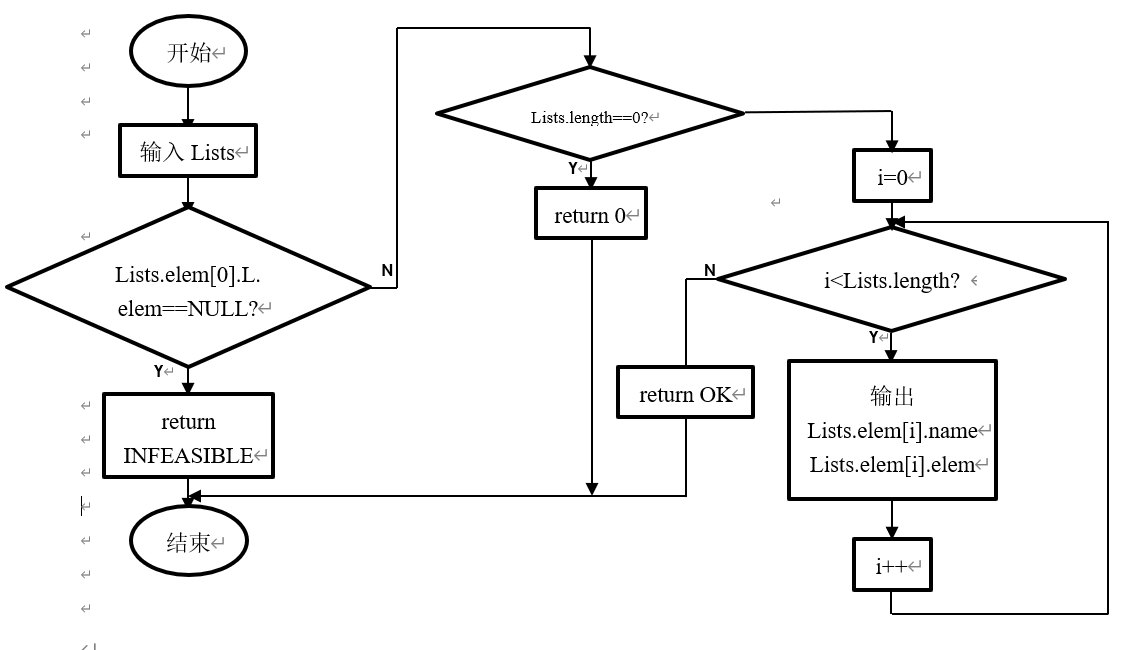


图1-20 遍历集合

(21)线性表集合存档SaveLists(Lists,FileName):首先判断集合是否存在或者表长是否为0，若是，则返回INFEASIBLE，表示当前集合内无数据，无需存档；否则就将其中每一个线性表的名称以及数据利用文件指针存入FileName文件，且存入名称之后需再写入一个’\n’换行符，便于后续读取中区分名称数据，存数据时，相邻数据间加空格即可。之后fclose关闭文件，返回OK，表示存储成功。

1.4 系统测试

操作总表1-1

|  |  |  |  |
| --- | --- | --- | --- |
| 测试功能及序号 | 管理线性表序号 | 输入函数的参数 | 预计输出 |
| 初始化 | 1 | 无 | 线性表初始化成功 |
| 销毁 | 1 | 无 | 线性表删除成功 |
| 清空 | 1 | 无 | 线性表清空成功 |
| 线性表判断为空 | 1 | 输入1：1，2，3，100 | 线性表非空 |
| 求表长 | 1 | 无 | 线性表表长为4 |
| 有位置求元素 | 1 | 输入位置为：3 | 元素值为3 |
| 查找元素 | 1 | 查找值为2的元素 | 存在 |
| 前驱元素 | 1 | 求1的前驱元素 | 不存在 |
| 后继元素 | 1 | 求50的后继元素 | 不存在 |
| 删除元素 | 1 | 删除100 | 删除成功 |
| 遍历线性表 | 1 | 无 | 无 |
| 插入元素 | 1 | 在第3个元素前面插入元素50 | 插入成功 |
| 保存线性表 | 1 | 保存到 | 成功保存 |
| 读取线性表 | 1 | 见下图 | 见下图 |
| 线性表集合操作 | 1 | 见下图 | 见下图 |

1. 输入对线性表1进行操作，进入菜单演示界面。执行功能1，初始化线性表，测试结果如1-21所示。

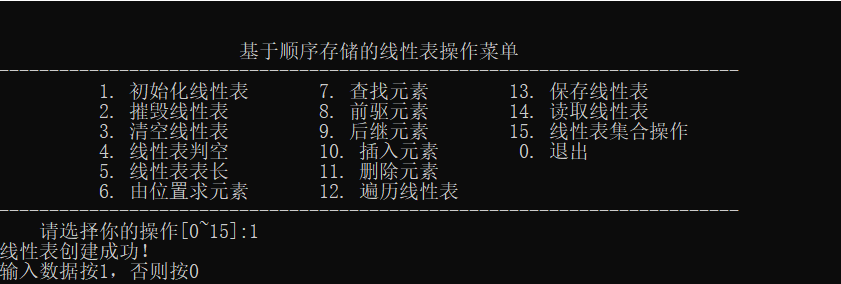


图1-21

1. 执行线性表销毁功能，测试结果如1-22

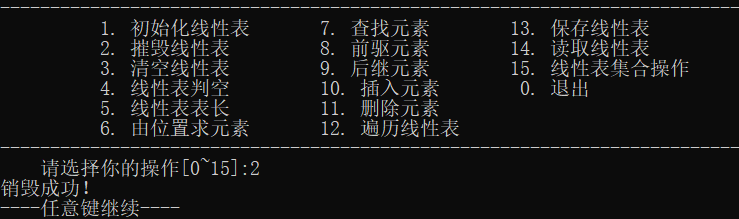


图1-22

1. 执行功能10，插入元素，所图1-23所示

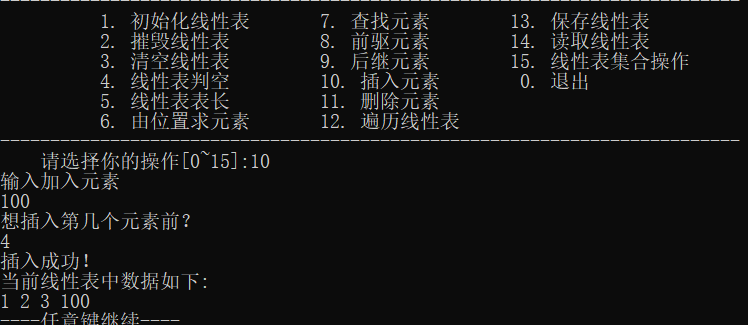


图1-23

1. 执行功能4，判断是否为空，如图1-24所示

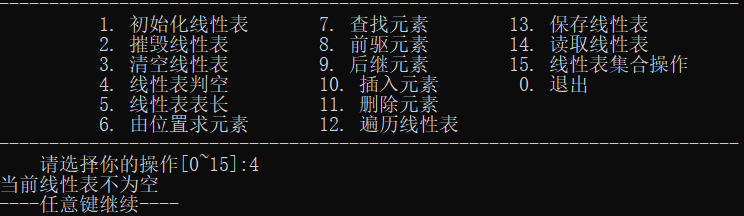


图1-24

1. 执行功能5，求表长，如图1-25所示

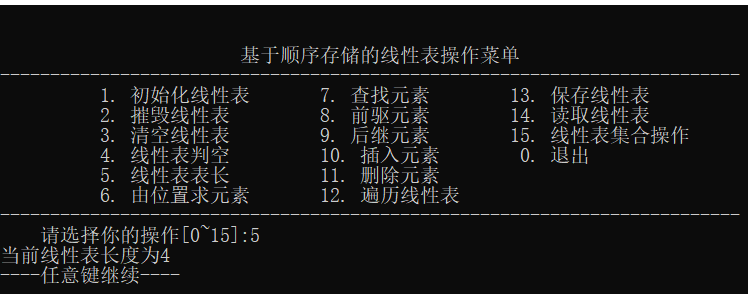


图1-25

1. 执行功能6，查找元素如图1-26所示

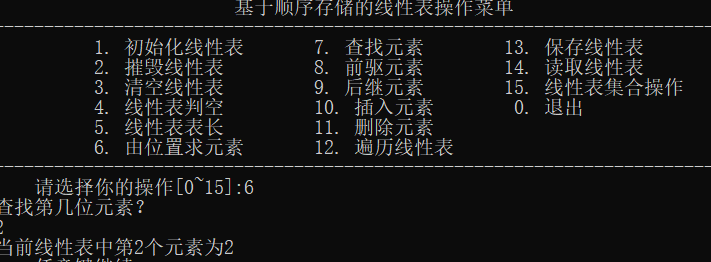


图1-26

1. 执行功能8，寻找前驱如图1-27所示

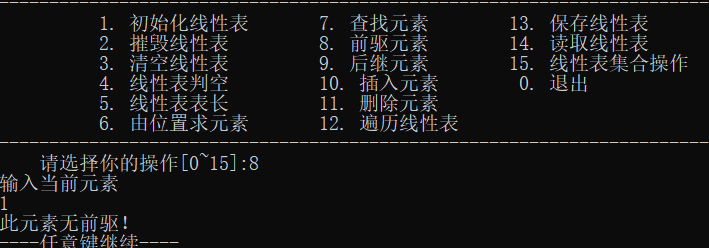


图1-27

1. 执行功能9，查找后继，如图1-28所示

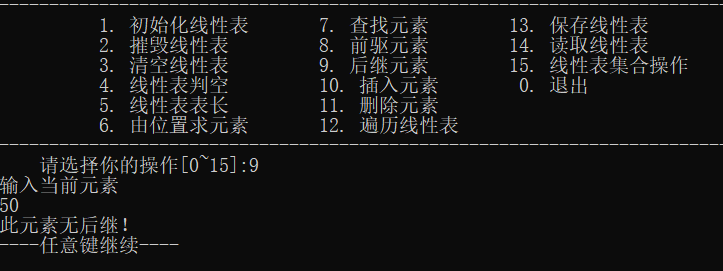


图1-28

1. 执行功能11，删除元素，如图1-29所示

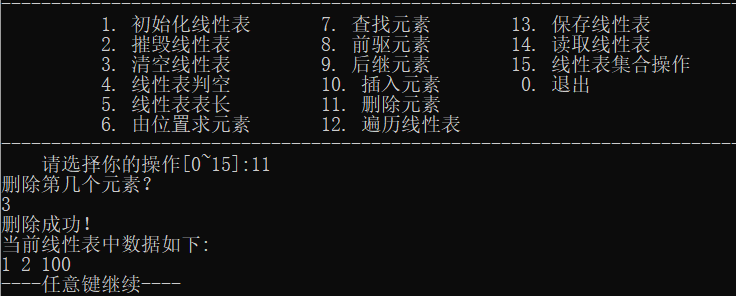


图1-29

1. 执行功能13到保存文件，如图1-30所示

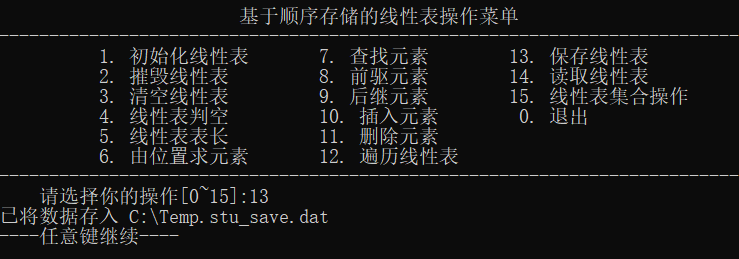


图1-30

11、执行功能12，遍历线性表，如图1-31

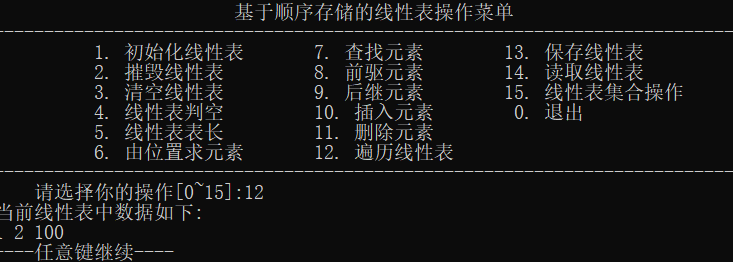
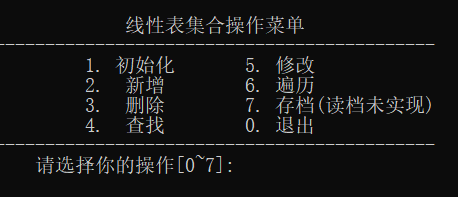
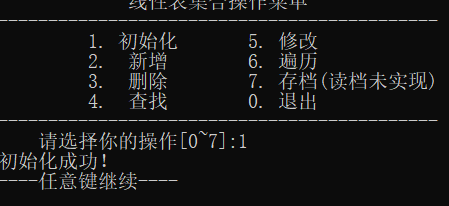


图1-31

1. 线性表的集合管理等操作如下图1-32



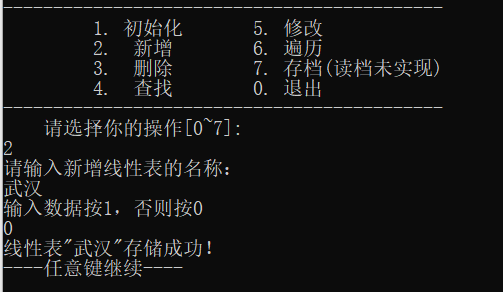


图1-32

## 1.5 实验小结

经过本次实验，加深了对线性表顺序存储结构的物理意义与逻辑意义上的理解。

1、在对顺序表的基本操作的实现过程中，亲身体会到了，找bug的苦恼，就让人心情郁闷烦躁，眼睛痛。但是也就是在这个痛苦的过程中，对线性表的初始化，清空，摧毁，求表长等等基本操作有了一定的认识。

2、比如这些操作中都有要首先考虑线性表是否存在，都有要考虑线性表运行过程中，输入的数据是否正确等等带来的影响，由这个影响由反作用与程序，去修正它。从而使它具有可读性和健壮性。不会太容易换个IDE就报错。

3、另外在这次实验中，我频繁用了i++这样的操作，以及while（）的循环操作，其中因为while循环里面的条件判断通常为（（）&&（））或者（（）||（）），由于i++在其中使得，边界条件很容易出错，有时候，因为双重错由便对了。所以，以后一定要慎用i++。

4、再者，当用指针了，有马上mallco的意识。之后，重温一遍文件读写的知识，确实是忘记了。最后加深了对“&e”的理解，这个用法使得在函数中调用主函数中的”e”时可以同时更改其值。

5、对于“销毁”和“清空”有了新的理解：实验之前对此二者的区别没有任何概念，只知道这么做，后来在和老师的交流中才理解到“清空”只是清除此线性表中数据，但是其存储空间和基地址仍被保留，而“销毁”则是删去原线性表中所有存储空间，同时其地址也会被丢失。

6、发现自己对于存档仍不够熟练，对于文件指针的掌握十分欠缺导致线性表集合读档没有实现成功，读取时仍会有文件指针未指向我所想象的位置，之后需要尽力改善，多多交流，多加练习。

7、对边界条件应当更加重视：程序的健壮性是一个程序成功与否的一个及其重要衡量标准，这次实验中，线性表(包括线性表集合)的操作中，如初始化函数中：“不可初始化已含有数据的线性表，这可能会导致数据的丢失”、插入函数中：“不可对一个不存在的线性表进行插入操作”、线性表集合插入函数：“不可向已满的集合中插入新的线性表”等等，处处体现了边界条件的重要

# 2 基于链式存储结构的线性表实现

## 2.1 问题描述

在链式存储结构中，基于链式存储的线性表是最基本、最简单，也是最常用的一种数据结构。故了解并熟练掌握其相关操作便很有必要，故本次实验的目的即为：(1)：加深对链式存储线性表的概念、基本运算的理解以及应用；(2)：熟练掌握链式存储线性表的逻辑结构与物理结构的关系；(3)：物理结构采用链表，熟练掌握线性表的基本运算的实现。

## 2.2 系统设计

本次实验采用含有头结点的链表作为基本存储结构，以上问题框架中的十二个基本操作，其具体运算功能定义如下。

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

⑵销毁表：函数名称是DestroyList(L)；初始条件是线性表L已存在；操作结果是销毁线性表L。

⑶清空表：函数名称是ClearList(L)；初始条件是线性表L已存在；操作结果是将L重置为空表。

⑷判定空表：函数名称是ListEmpty(L)；初始条件是线性表L已存在；操作结果是若L为空表则返回TRUE,否则返回FALSE。

⑸求表长：函数名称是ListLength(L)；初始条件是线性表已存在；操作结果是返回L中数据元素的个数。

⑹获得元素：函数名称是GetElem(L,I,e)；初始条件是线性表已存在，1≤i≤ListLength(L)；操作结果是用e返回L中第i个数据元素的值。

⑺查找元素：函数名称是LocateElem(L,e)；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满足相等的数据元素的位序，若这样的数据元素不存在，则返回值为0。

⑻获得前驱：函数名称是PriorElem(L,cur\_e,pre\_e)；初始条件是线性表L已存在；操作结果是若cur\_e是L的数据元素，且不是第一个，则用pre\_e返回它的前驱，否则操作失败，pre\_e无定义。

⑼获得后继：函数名称是NextElem(L,cur\_e,next\_e)；初始条件是线性表L已存在；操作结果是若cur\_e是L的数据元素，且不是最后一个，则用next\_e返回它的后继，否则操作失败，next\_e无定义。

⑽插入元素：函数名称是ListInsert(L,I,e)；初始条件是线性表L已存在，1≤i≤ListLength(L)+1；操作结果是在L的第i个位置之前插入新的数据元素e。

⑾删除元素：函数名称是ListDelete(L,I,e)；初始条件是线性表L已存在且非空，1≤i≤ListLength(L)；操作结果：删除L的第i个数据元素，用e返回其值。

⑿遍历表：函数名称是ListTraverse(L)，初始条件是线性表L已存在；操作结果是依次对L的每个数据元素调用函数。

## 2.3 系统实现

本系统通过命令式菜单来调用各函数功能，系统开始运行时会输出菜单界面，依照菜单中给出的各功能的指令输入命令，以实现相关功能的使用。

总体实现为：

首先根据每一个基本操作写出相关函数的基本功能实现，后再针对异常状况进行合理处理，完成每一个函数的具体实现。

后在main函数中定义一个op存储当前命令，通过switch语句，针对op的不同值在case op中调用对应函数以实现相应功能。

每个函数的具体实现方法如下：

(1)初始化表InitList(L)：首先利用malloc函数在内存中寻找一片大小为sizeof(Lnode)的存储空间，将其首地址赋给目前不存在的链表L的头指针L，后将L🡪next置为NULL，之后返回OK。(异常处理：当L已存在，即L !=NULL时便不可再次初始化，此时需返回INFEASIBLE😉

(2)销毁表DestroyList(L)：首先新定义一个链表指针p为NULL，之后在循环中将p置为L, L置为L🡪next, 将p指向的结点空间free，直到L==NULL时停止循环即可。

(3)清空表ClearList(L)：相似但不同于销毁，清空表首先需要新定义一个链表指针p指向NULL，后将L置为L🡪next, 再在循环中判断L是否为NULL，是则结束循环，不是就将p置为L，L置为L🡪next, 将p指向的节点空间free。

(4)判定空表ListEmpty(L)：当链表不存在时(即L==NULL时)，返回INFEASIBLE；当L🡪next==NULL时，即表明线性表为空，返回TRUE；除此以外，线性表非空，返回FALSE。

(5)求表长ListLength(L)：当线性表不存在时，函数返回INFEASIBLE；否则就定义计数变量count，初值为0，当L🡪next!=NULL时，L=L🡪next，count++。之后返回count即可。(框图如图2-5)

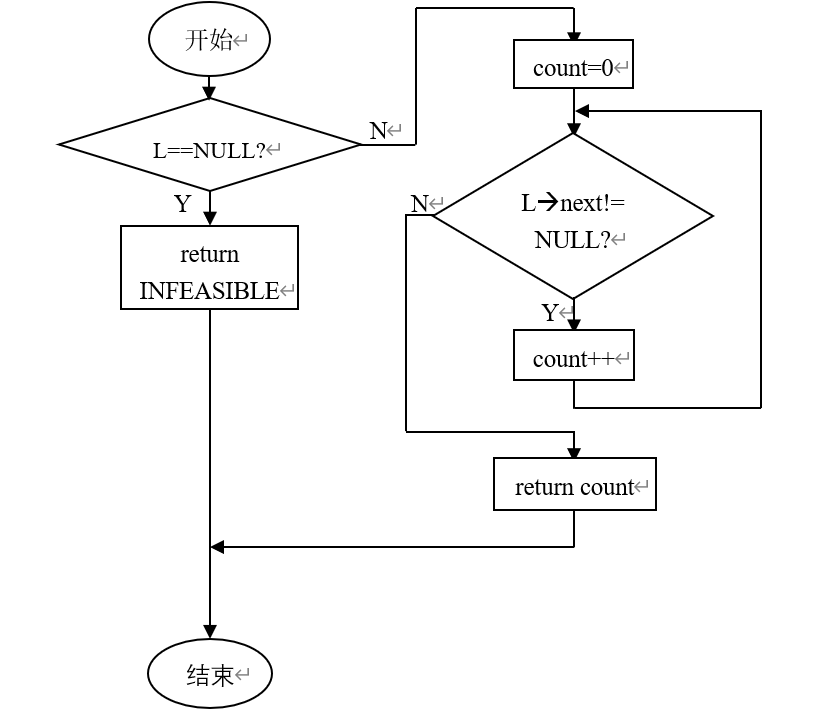


图2-5 求表长

(6)获得元素GetElem(L,I,e)：当线性表不存在时，函数返回INFEASIBLE；当线性表存在，但是I < 1时，查找位置非法，返回ERROR；否则，就定义计数变量count，初值为0，扫描一次链表，count随之加一，当count==i时，将当前结点内数据赋值到e后返回OK即可；若扫描完成之后仍为出现count==i，说明查找位置非法，返回ERROR。(框图如图2-6)

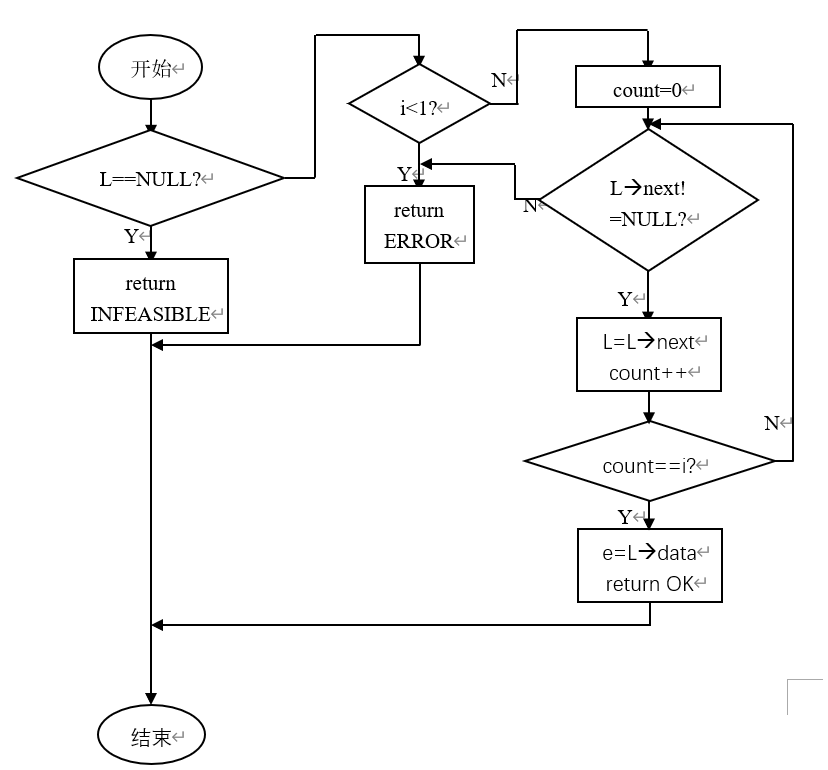


图2-6 获得元素

(7)查找元素LocateElem(L,e): 当线性表不存在时， 函数返回INFEASIBLE;当线性表存在时， 扫描线性表中所有元素，并在扫描过程中将每一个已在线性表中的元素与e比较， 如果相等则返回当前元素在链表中的序号+1，如果扫描过程中没有任何一个元素与e相等，查找失败，则返回ERROR。(程序框图如图2-7)

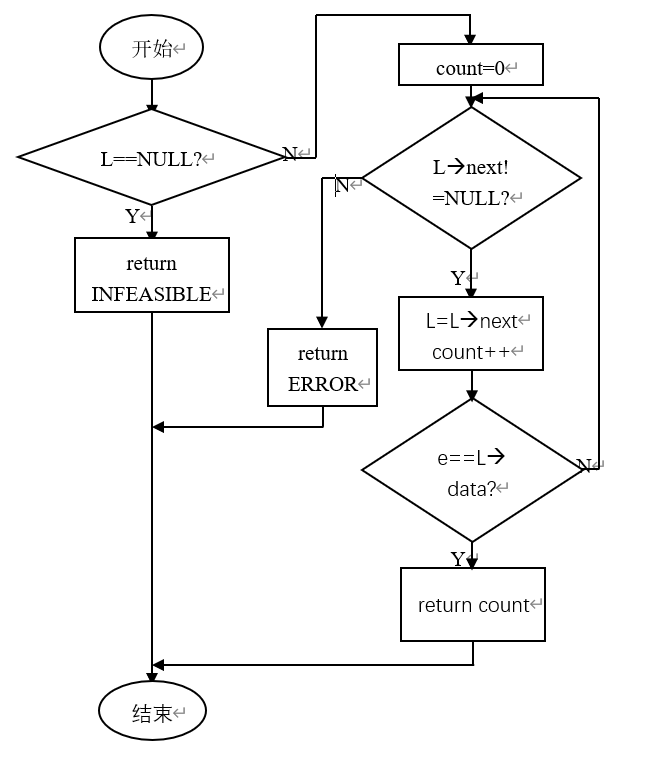


图2-7 查找元素

(8)获得前驱PriorElem(L,cur\_e,pre\_e)：当链表不存在时，函数返回INFEASIBLE; 当链表存在时，L=L🡪next跳过表头结点，在L!=NULL以及L🡪next!=NULL时扫描链表，且扫描过程中比较当前位置之后一个元素L🡪next🡪data是否与cur\_e相等，如果相同，就将L🡪data的值赋给pre，返回OK；若扫描结束后仍未找到则返回ERROR，表明此元素无前驱。(程序框图如图2-8)

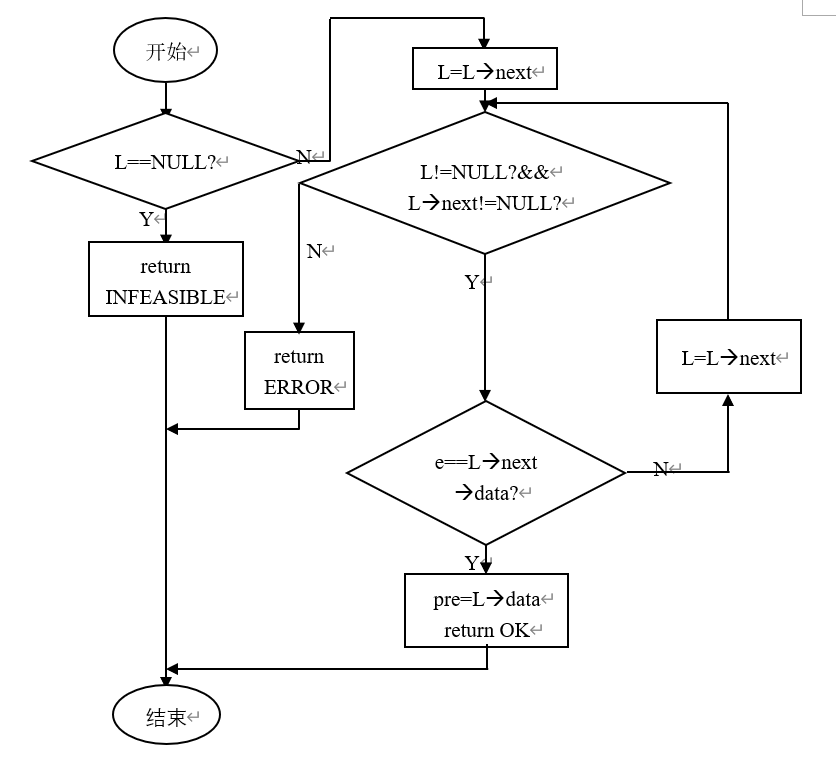


图2-8 前驱元素

(9)获得后继NextElem(L,cur\_e,next\_e)：当线性表不存在时，函数返回INFEASIBLE; 当线性表存在时，在L🡪next!=NULL的条件下扫描线性表，且扫描过程中比较当前位置元素是否与cur\_e相等，如果相同，返回L🡪next🡪data，扫描结束之后仍未找到时，返回ERROR。(框图如图2-9)

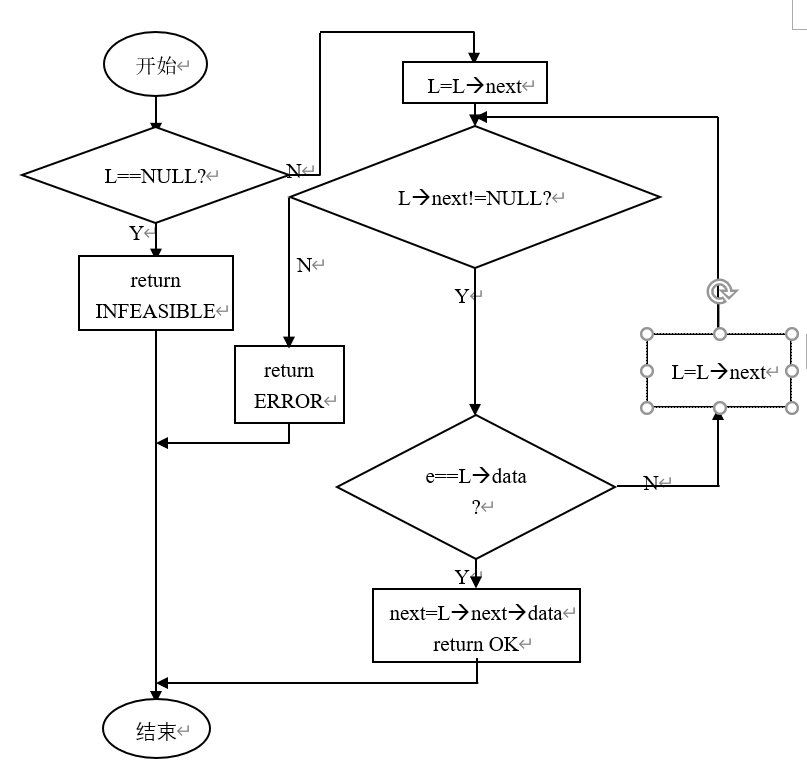


图2-9 后继元素

(10)插入元素ListInsert(L,I,e)：当线性表不存在时，函数返回INFEASIBLE；当线性表存在但是I < 1或者I > ListLength(L)+ 1时，插入位置非法，函数返回ERROR；当线性表存在且i >= 1, i <= ListLength(L)+ 1时，首先利用malloc函数寻找一片大小为sizeof(LNode)的空间，将此空间地址赋给结点指针p，将p🡪data赋值为e，p🡪next置为NULL以避免野指针，之后定义count=0，在L!=NULL时进行循环，判断count是否等于i-1，若是，则将p🡪next置为L🡪next，再将L🡪next置为p，之后return OK即可。若count!=i-1，则L=L🡪next，count++后，继续循环即可。(框图如2-10)

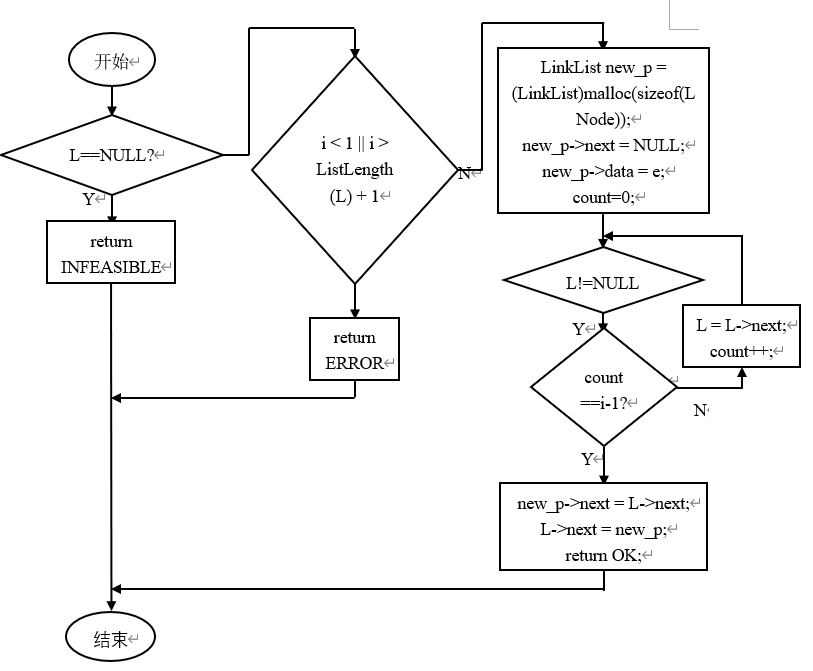


图2-10 插入元素

(11)删除元素ListDelete(L,i,e)：当线性表不存在时，函数返回INFEASIBLE；当线性表存在但是i < 1或者i > ListLength(L)时，删除位置非法，函数返回ERROR；当线性表存在且i >= 1, i <= ListLength(L)时，首先令count=0，结点指针p=NULL，在L!=NULL的情况下进行循环，循环中首先判断count是否等于i-1，若是，则令p = L->next，记录待删除结点；e = L->next->data，记录待删除结点内的数据；L->next = L->next->next，从链表中抹去待删除结点；free(p)，释放待删除结点空间；若count!=i-1，则L=L🡪next，count++，继续循环即可。(框图如图2-11)

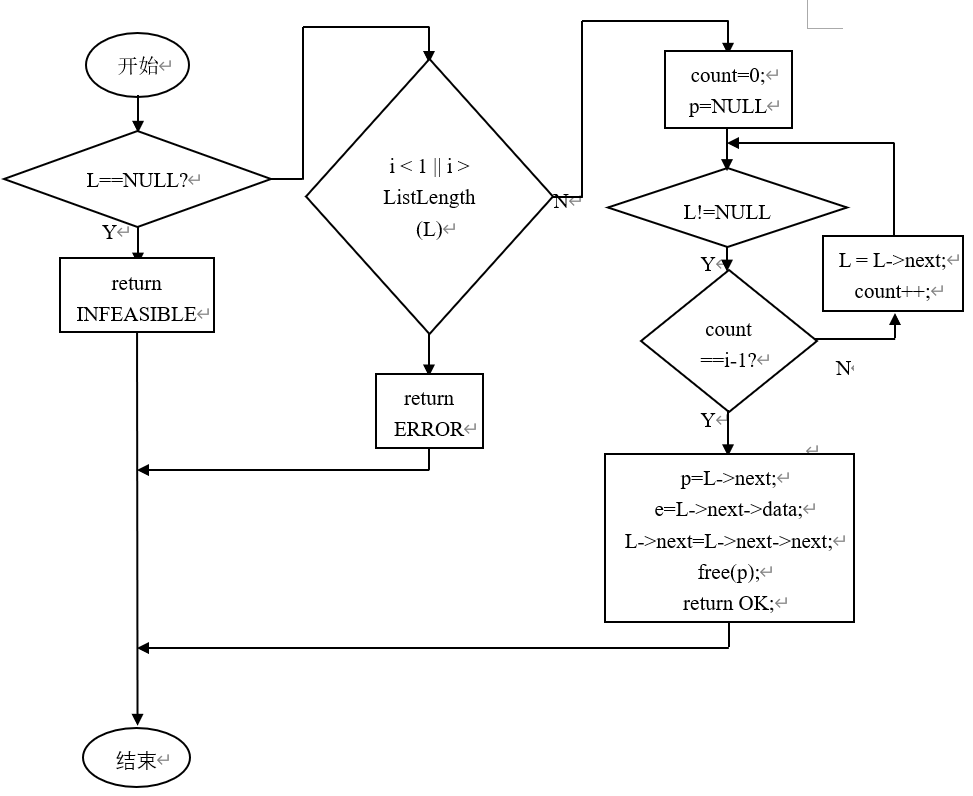


图2-11 删除节点

(12)遍历表ListTraverse(L)：当链表表不存在时，函数返回INFEASIBLE；当链表表存在且长度为0时，输出“空”，函数返回OK；当链表存在且长度不为0时，利用循环结构将链表表扫描一遍，并输出链表中每一个元素的值，函数返回OK。

(13)输入一组元素InitInput(L):考虑到初始化时，用户可能会与此同时进行赋值操作，故它的调用是紧接在InitList(L)返回OK之后。此函数首先判断用户输入命令，命令为0时表示不输入数据，返回0即可；命令为1时表示输入数据，(此处假设用户输入数据不包含0)，则利用循环结构和函数ListInsert(L,i,e)输入数据，同时将数据存入链表中，直到输入为0时循环停止，函数返回OK。

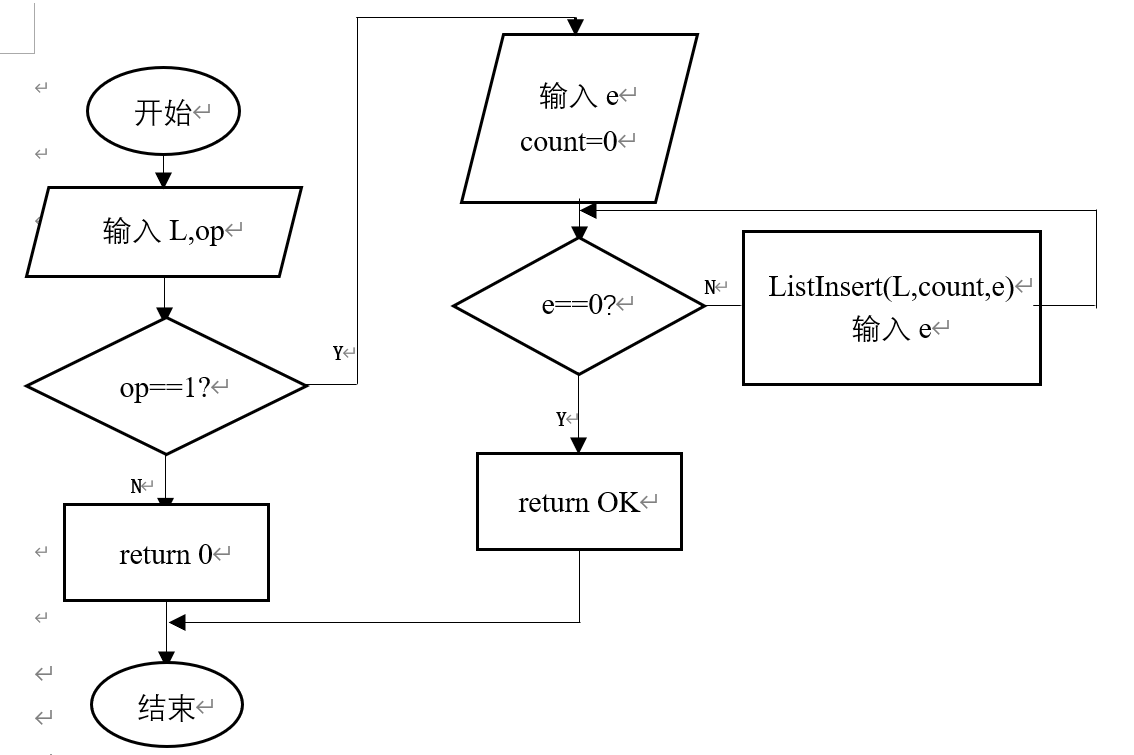


图2-13 初始化输入

此外，为了使系统功能更加丰富、更有实用性，我添加了部分功能模块，具体如下：

文件读取篇：

(13)线性表数据存入文件SaveList(L, SaveFile):利用文件指针以”w+”的形式打开(文件不存在时自动创建)文件名为SaveFile的文件，如果此步打开文件失败则函数返回ERROR，否则就将线性表L扫描一遍，并将每一个元素输入到目标文件中，且每输入一个元素后就立即输入一个空格，否则会导致存入文件中的数据连在一起难以取出。输入完毕后利用fclose函数关闭该目标文件。最后返回OK；

(14)读取文件中数据存入线性表LoadList(L, LoadFile)：利用文件指针以”r”的形式打开文件名为LoadFile的文件，如果此步打开文件失败则函数返回ERROR，否则就将文件中每一个数据利用fscanf函数读入线性表，读取完毕后利用fclose函数关闭该目标文件。最后返回OK；

线性表集合操作篇：(线性表集合的存储结构为链表，数据域为各链表头指针以及链表名)

新定义了一个菜单函数gather\_operate(Lists),其大体结构与main函数相同，将八种线性表集合操作集于其中，返回值类型为一个线性表指针。(线性表修改操作需利用此返回值才得以实现)

(15)线性表集合初始化InitLists(Lists):首先判断线性表集合中是否存在表头，若存在则表明线性表集合中可能已存在数据，不可再次进行初始化，返回INFEASIBLE;否则就Lists=(pLISTS)malloc(sizeof(LISTS))，Lists🡪L=NULL，InitList(Lists🡪L)，Lists🡪next=NULL，对第一个线性表，利用之前已定义的InitList(L)函数将其初始化后，返回OK即可。

(16)集合长度ListsLength(Lists)：当链表集合不存在时，函数返回INFEASIBLE；否则就定义计数变量count，初值为0，在Lists🡪next!=NULL时进入循环，L=L🡪next，count++。之后返回count即可。(框图如图2-16)

(17)向集合中新增链表AddList(Lists,i,ListName)：首先，若链表集合不存在，返回INFEASIBLE；若i<1或者i>ListsLength(Lists)+1时，新增位置非法，返回ERROR；否则，就利用循环找到第i-1个链表，记录当前链表在集合中的结点地址p，之后新定义集合结点new\_p，new\_p=(pLISTS)malloc(sizeof(LISTS))，再利用已定义的函数InitList(new\_p🡪L)和InitInput(new\_p🡪L)初始化new\_p，利用strcpy函数将ListName拷贝至new\_p🡪name，完成后将new\_p🡪next置为p🡪next，p🡪next置为new\_p, 再返回OK即可。(框图如图2-17)

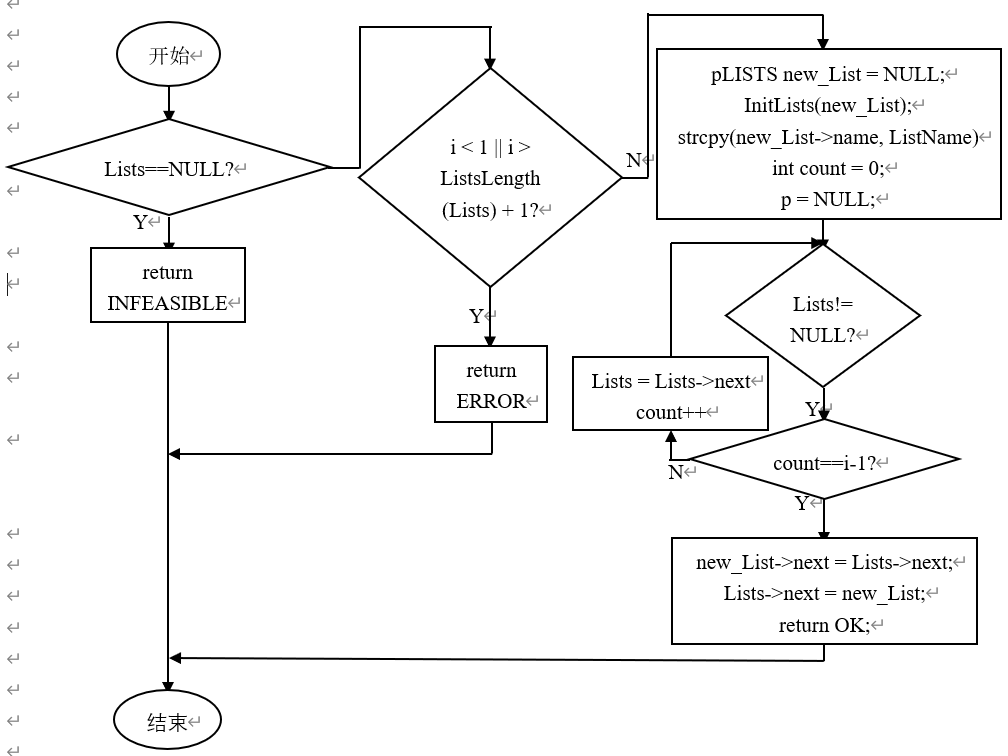


图2-17 新增链表

(18)删除集合中名为ListName的线性表RemoveList(Lists, ListName):首先，若当前线性表集合不存在，则返回INFEASIBLE；否则就扫描当前线性表集合，并在扫描过程中利用strcmp函数判断每一个结点的下一个链表的名称是否与ListName相同，定义一个指针p指向当前链表结点的后一个链表结点，若相同就Lists->next = Lists->next->next，抹去当前链表的下一个链表，之后DestroyList(p->L)，free(p)，返回OK即可；如果扫描结束后仍未找到某个名为ListName的线性表，返回0即可。(框图如图2-18)

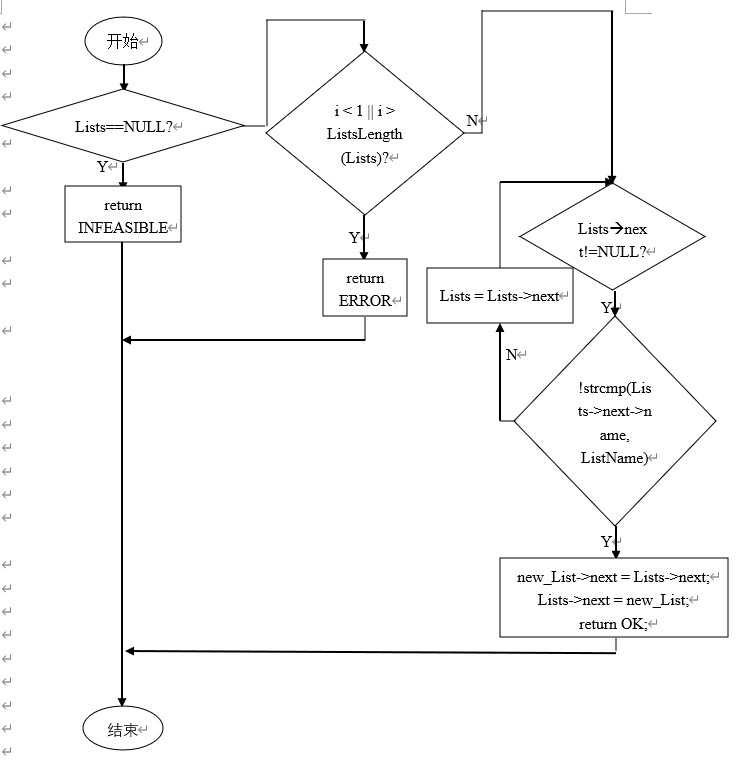


图2-18 删除链表

(19)查找线性表LocateList(Lists, ListName)：扫描当前线性表集合，过程中利用strcmp函数判断当前线性表名称是否与ListName相同，若有相同则返回当前线性表的逻辑位置;若扫描结束仍未发现名为ListName的线性表，则返回0，表明查无此表。

(20)修改线性表：此功能无特定函数，是利用菜单函数gather\_operate(Lists)的返回值在main函数里实现，具体过程为：当用户选择修改功能时，先利用查找线性表函数LocateList(Lists,ListName)找到所要修改的表(若未找到则输出查无此表后break switch即可，无后续操作)，之后将查找到的线性表的地址作为gather\_operate函数的返回值返回到主函数中，之后，将主函数中原有线性表的地址存入一个新定义的临时线性表指针temp，再用gather\_operate函数的返回值覆盖原有线性表，之后便可对其执行线性表相关操作，操作结束后(即用户输入命令为0时)，再在main函数中将temp重新覆盖到原线性表L上即可。

(21)遍历集合ListsTraverse(Lists)：首先判断线性表集合是否存在，不存在则返回INFEASIBLE，若存在再判断长度是否为0，若为0则返回0，否则扫描线性表集合，并输出每一个线性表的名称和其中的数据，返回OK。

## 2.4 系统测试

操作总表2-1

|  |  |  |  |
| --- | --- | --- | --- |
| 测试功能及序号 | 输入管理线性表序号 | 输入函数的参数 | 预计输出 |
| 初始化 | 1 | 无 | 线性表初始化成功 |
| 销毁 | 1 | 无 | 线性表删除成功 |
| 清空 | 1 | 无 | 线性表清空成功 |
| 线性表判断为空 | 1 | 输入1：1，2，3，100 | 线性表非空 |
| 求表长 | 1 | 无 | 线性表表长为4 |
| 有位置求元素 | 1 | 输入位置为：3 | 元素值为3 |
| 查找元素 | 1 | 查找值为2的元素 | 存在 |
| 前驱元素 | 1 | 求1的前驱元素 | 不存在 |
| 后继元素 | 1 | 求50的后继元素 | 不存在 |
| 插入元素 | 1 | 在第3个元素前面插入元素50 | 插入成功 |

|  |  |  |  |
| --- | --- | --- | --- |
| 删除元素 | 1 | 删除100 | 删除成功 |
| 遍历线性表 | 1 | 无 | 无 |
| 保存线性表 | 1 | 保存到 | 成功保存 |
| 读取线性表 | 1 | 见下图 | 见下图 |
| 线性表集合操作 | 1 | 见下图 | 见下图 |

1. 输入对线性表1进行操作，进入菜单演示界面。执行功能1，初始化线性表，测试结果如2-21所示。

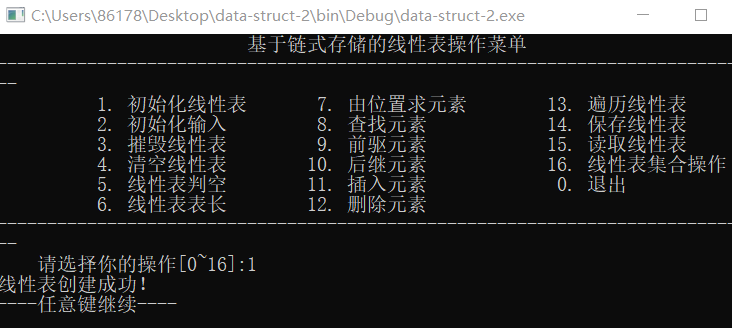


图2-21

1. 执行线性表销毁功能，测试结果如2-22

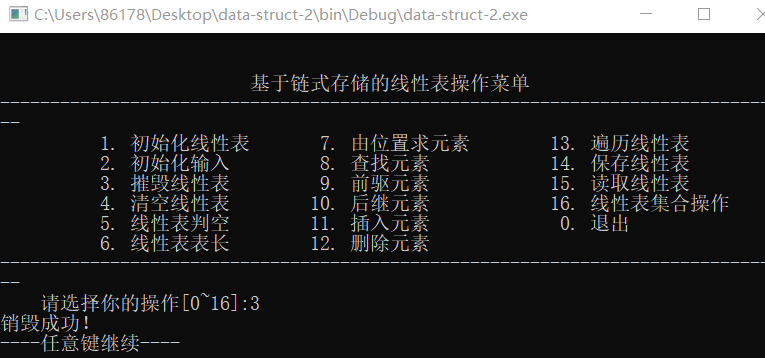


图2-22

1. 执行功能10，插入元素，所图2-23所示

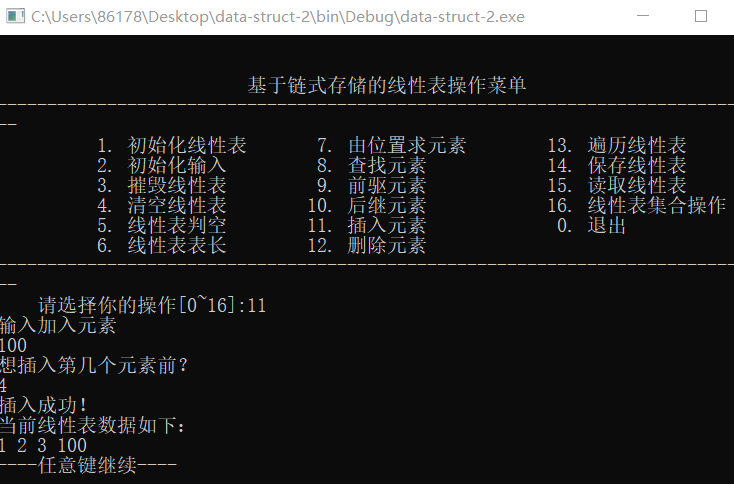


图2-23

1. 执行功能4，判断是否为空，如图2-24所示

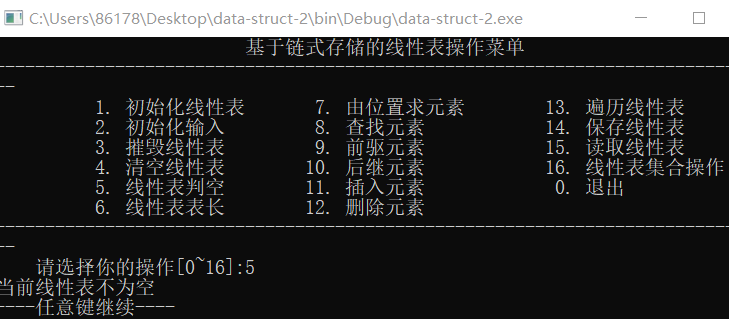


图2-24

1. 执行功能5，求表长，如图2-25所示

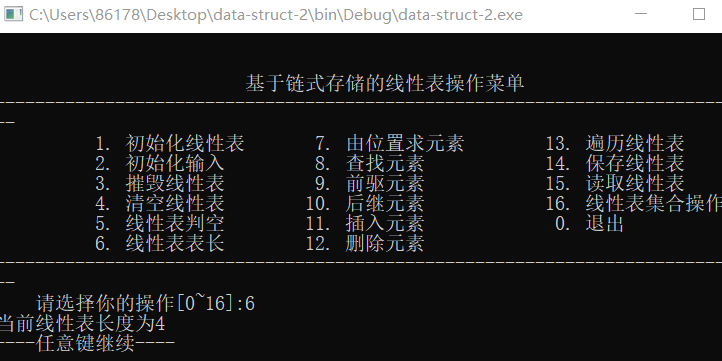


图2-25

1. 执行功能6，查找元素如图2-26所示



图2-26

1. 执行功能8，寻找前驱如图2-27所示



图2-27

1. 执行功能9，查找后继，如图2-28所示

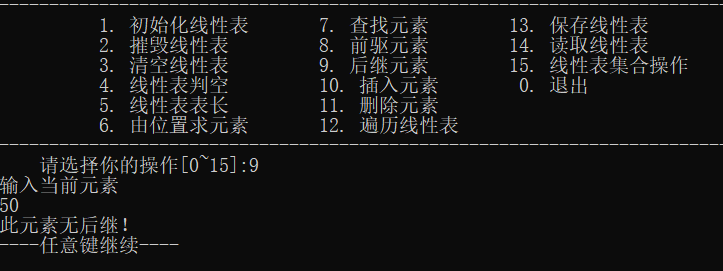


图2-28

1. 执行功能11，删除元素，如图2-29所示

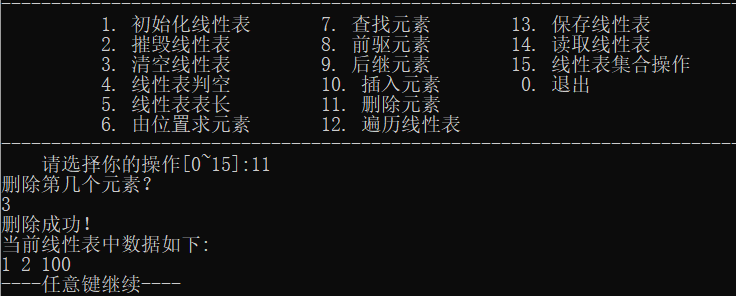


图2-29

1. 执行功能13到保存文件，如图2-30所示

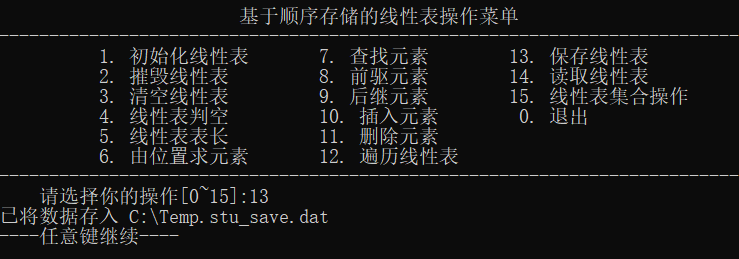


图2-30

11、执行功能12，遍历线性表，如图2-31

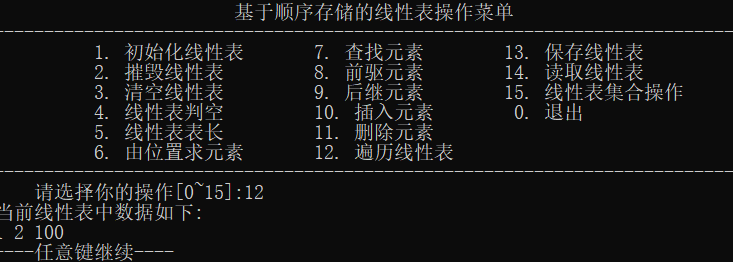
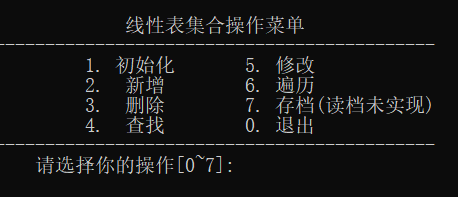
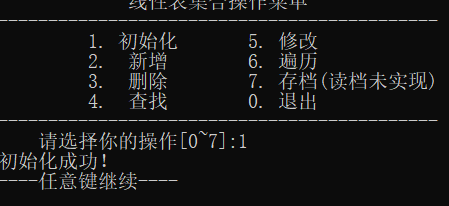


图2-31

1. 线性表的集合管理等操作如下图2-32





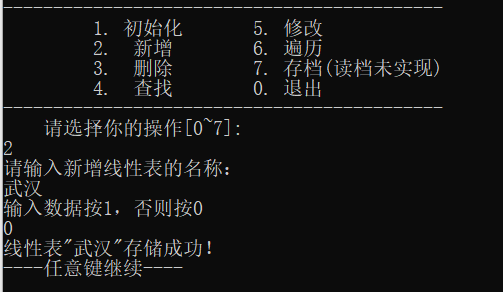


图2-32

## 2.5 实验小结

经过本次实验，加深了对线性表顺序存储结构的物理意义与逻辑意义上的理解。

1、在链表的基本操作的实现过程中，亲身体会到了，找bug的苦恼，就让人心情郁闷烦躁，眼睛痛。但是也就是在这个痛苦的过程中，对线性表的初始化，清空，摧毁，求表长等等基本操作有了一定的认识。

2、由于上次实验对顺序表进行了操作，这次实验中用链表进行，就快了很多，像清空、摧毁等操作的意义也清楚了很多

3、由于链表的不同，在某些细节的地方和顺序表还是有很大的区别

经过本次实验，加深了对线性表顺序存储结构的物理意义与逻辑意义上的理解。

4、在对顺序表的基本操作的实现过程中，亲身体会到了，找bug的苦恼，就让人心情郁闷烦躁，眼睛痛。但是也就是在这个痛苦的过程中，对线性表的初始化，清空，摧毁，求表长等等基本操作有了一定的认识。

5、比如这些操作中都有要首先考虑线性表是否存在，都有要考虑线性表运行过程中，输入的数据是否正确等等带来的影响，由这个影响由反作用与程序，去修正它。从而使它具有可读性和健壮性。不会太容易换个IDE就报错。

6、另外在这次实验中，我频繁用了i++这样的操作，以及while（）的循环操作，其中因为while循环里面的条件判断通常为（（）&&（））或者（（）||（）），由于i++在其中使得，边界条件很容易出错，有时候，因为双重错由便对了。所以，以后一定要慎用i++。

7、再者，当用指针了，有马上mallco的意识。之后，重温一遍文件读写的知识，确实是忘记了。最后加深了对“&e”的理解，这个用法使得在函数中调用主函数中的”e”时可以同时更改其值。

8、表最好设置一个头结点，这样可以更简便地进行插入和删除操作。另外，可以在头结点中存入一些信息，比如链表的长度信息。这样一来可以大大地简化某些函数的算法复杂度，比如求表长。原本需要对全表进行遍历，而若将表长信息存入头结点，那么直接读取长度信息即可。

9、另外，在给结点分配存储空间时，可以适当地增加一些空间，以免某些编译器会产生的溢出的现象。

10、在对链表的结点进行操作时（比如删除），务必注意要对特定结点进行重指向，比如让其指向NULL，否则会产生野指针，从而造成程序运行错误。

# 3 基于二叉链表的二叉树实现

## 3.1 问题描述

树形结构，是目前应用十分广泛的一种数据结构，像常见的电脑中文件的目录，都是基于树形结构所设计出来，而二叉树，作为树形结构的一个重要类型，也经常被用于解决许多实际问题，而且，即使是一般的树也能经过简单的转换变为二叉树。另外，二叉树的存储结构及其算法都较其它多叉树简单，因此掌握二叉树的数据存储方式以及相关基本运算对我们来说就显得尤为重要。而这，也就是本次实验的目的。

## 3.1.1 问题框架

根据二叉树的存储方式，并结合其实际应用中数据管理方式，至少应定义：初始化树、摧毁树、清空树、求树的深度、查找结点、结点赋值、获得兄弟结点、插入结点、删除结点、前序遍历、中序遍历、后序遍历、按层遍历这十二个基本操作。

## 3.2 系统设计

以上问题框架中的十二个基本操作，其具体运算功能定义如下。

⑴初始化树：函数名称是InitBiTree(T)；初始条件是树T不存在；操作结果是构造一个空树。

⑵销毁树：函数名称是ClearBiTree(T)；初始条件是树T已存在；操作结果是销毁树T。

⑶清空树：函数名称是ClearBiTree(T)；初始条件是树T已存在；操作结果是将T重置为空树。

⑷求树的深度：函数名称是BiTreeDepth (T)；初始条件是树已存在；操作结果是返回T的深度。

⑸结点查找：函数名称是LocateNode(T，key)；初始条件是树已存在；操作结果是返回键值为key的结点地址。

⑹结点赋值：函数名称是Assign(L,e,value)；初始条件是树已存在；操作结果是将树T中键值为e的结点内的数据替换为value。

⑺获得兄弟结点：函数名称是GetSibling(T, e)；初始条件是书存在；操作结果是将树T中键值为e的结点地址返回。

⑻插入节点：函数名称是InsertNode(T, e, LR, value)；初始条件是树T已存在；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；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，返回OK。如果插入失败，返回ERROR。特别地，当LR为-1时，作为根结点插入，原根结点作为c的右子树。

⑼删除结点：函数名称是DeleteNode(T, e)；初始条件是树T已存在；操作结果是若e是树T中某一结点的键值，则删除该结点，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。成功删除结点后返回OK，否则返回ERROR。

⑽前序遍历：函数名称是PreOrderTraverse(T,visit)；初始条件是树已存在；操作结果为按前序遍历方式对每个结点数据用visit访问出来。

⑾中序遍历：函数名称是InOrderTraverse(T,visit)；初始条件是树已存在；操作结果为按中序遍历方式对每个结点数据用visit访问出来。

⑿后序遍历：函数名称是PostOrderTraverse(T,visit)；初始条件是树已存在；操作结果为按后序遍历方式对每个结点数据用visit访问出来。

## 3.3 系统实现

⑴初始化树InitBiTree(T)：当T==NULL，即树不存在时，函数返回INFEASIBLE；否则就对T，利用malloc函数分配空间后，将T->lchild和T->rchild置为NULL，返回OK即可。

⑵销毁树ClearBiTree(T)：当T==NULL，即树不存在时，函数返回OK;否则就递归，ClearBiTree(T->lchild)，ClearBiTree(T->rchild)，之后判断当前结点左右孩子是否均为NULL，若是，则free(T)后将T置为NULL，return OK即可。

⑶清空树ClearBiTree(T)：此处清空树与销毁树共用同一个函数，只是传入的参数不同，此处传入参数为根节点的左孩子，相当于只保留二叉树的表头，将其存放数据的左子树销毁。

⑷求树的深度BiTreeDepth(T): 当T==NULL时，函数返回0; 否则返回BiTreeDepth(T->lchild) + 1和BiTreeDepth(T->rchild) + 1二者之间的最大者。

⑸结点查找LocateNode(T, e): 首先定义静态结点指针p=NULL，并进行如下操作：

1、判断若p!=NULL，就判断p指结点的键值是不是所要查找的键值，若不是就将p置为NULL；若是就说明已经找到了，返回p即可。

2、判断当前结点地址是否为空，若是就返回p；

3、判断e是否等于当前结点T的键值，若是，令p=T，返回p即可。

4、递归LocateNode(T->lchild，e), LocateNode(T->rchild，e)。

框图如图3-5：

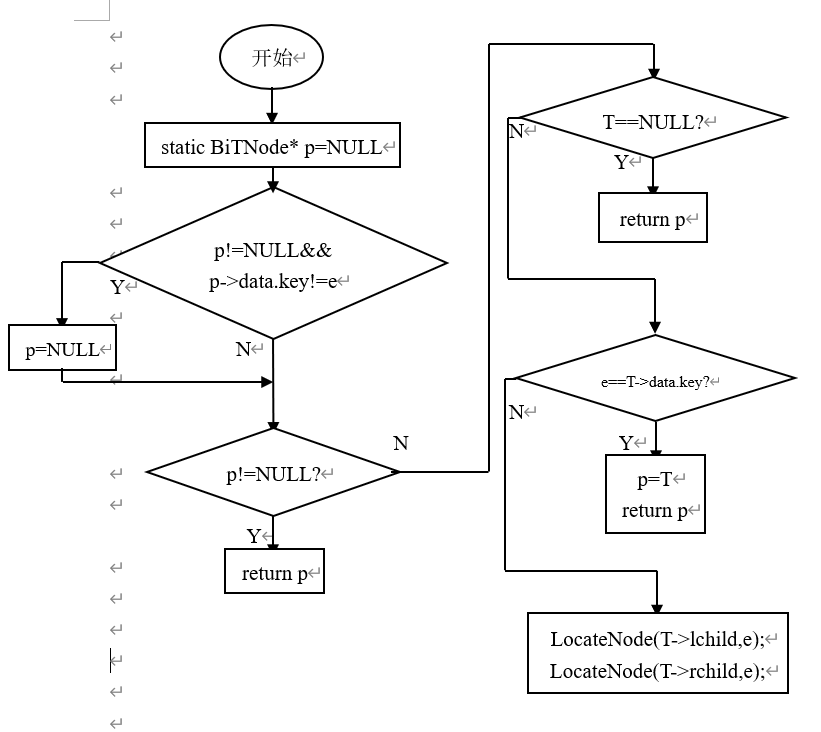


图3-5 查找结点

⑹结点赋值Assign(T, e, value)：当T==NULL，函数返回INFEASIBLE；否则，先利用LocateNode(T, value.key)是否为NULL，若不是，则应该返回ERROR，表示所要赋给目标结点的键值与原树中某一结点重复了，赋值失败；若是NULL，则再次利用LocateNode(T, e)找到目标结点地址赋值到p，后p->data.key=value.key, strcpy(p->data.others, value.others)，返回OK即可。框图如图3-6：

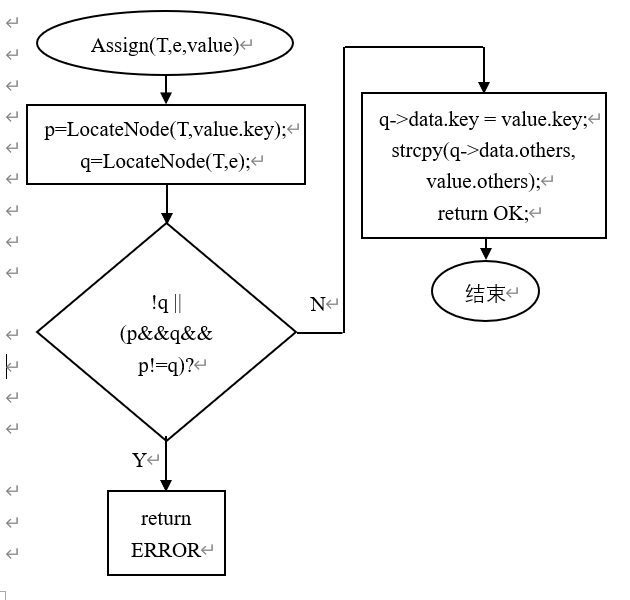


图3-6 结点赋值

⑺获得兄弟结点GetSibling(T, e): 定义结点指针rec=NULL

判断当前树中结点是否为NULL，若是，返回NULL；

判断T->lchild!=NULL && T->lchild->data.key==e，是，返回T->rchild;

判断T->rchild!=NULL && T->rchild->data.key==e，是，返回T->lchild;

不满足以上两种，用rec取得GetSibling(T->lchild, e)的返回值，不为NULL时返回rec，为NULL时，获取GetSibling(T->rchild, e)的返回值，为NULL时返回NULL，否则返回rec。框图如图3-7：

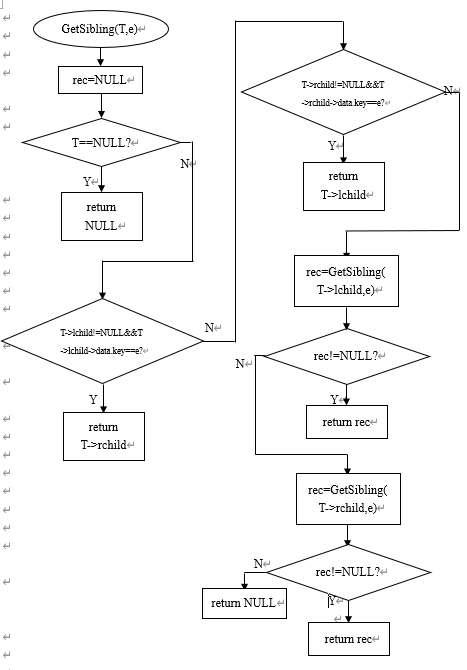


图3-7 获得兄弟结点

(8)插入结点InsertNode(T, e, LR, c): 定义结点指针aim=NULL，repeat=NULL,利用LocateNode(T, c.key)查找当前树中是否已经存在与待插入结点键值相同的结点，返回值赋值到repeat，若repeat!=NULL，函数返回ERROR；再利用LocateNode(T, e)查找树中是否有待查找结点，返回值赋值到aim，若aim== NULL, 返回ERROR；若不满足返回ERROR的以上情况，则利用malloc为新加入结点NewNode分配空间，并将c中数据填入NewNode的数据域中，之后判断LR值：

1.为0时：将NewNode->rchild置为aim->lchild; NewNode->lchild置为NULL，aim->lchild置为NewNode后，返回OK即可。

2.为1时:将NewNode->rchild置为aim->rchild; NewNode->lchild置为NULL; aim->rchild置为NewNode; 后返回OK即可。

3.为-1时:将NewNode->rchild置为T, NewNode->lchild置为NULL, T置为NewNode, 返回OK即可。框图如图3-8：

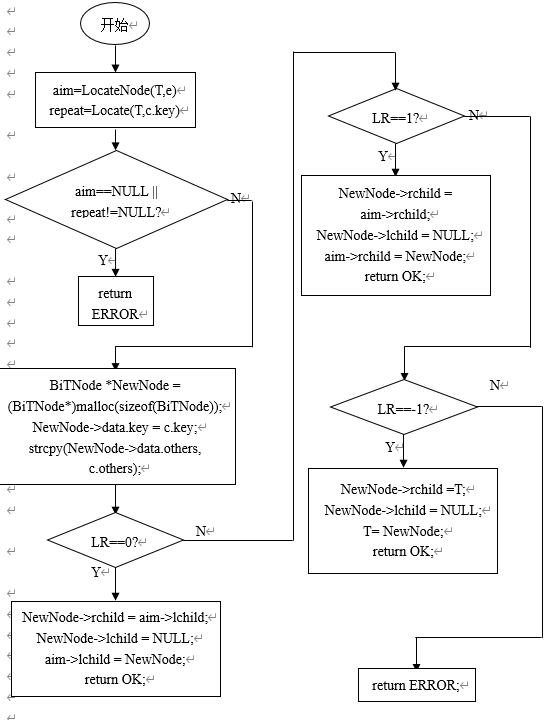


图3-8 插入节点

(9) 寻找双亲结点ParentNode(T,e,p): 为一辅助函数，用于寻找待删除结点的双亲结点p,由引用参数p带出，返回值为待删除结点关于双亲结点p的相对位置值，具体实现为：定义变量state=ERROR;

判断当前结点是否为NULL，是则函数返回ERROR，否则：

继续判断当前结点键值是否为e，是则令p=T, 函数返回OK，表明所要找的点为根节点，无双亲结点，若不是：

则判断T->lchild!=NULL && T->lchild->data.key==e，是，就令p=T,返回LEFT即可。若不是：

则判断T->rchild!=NULL && T->rchild->data.key==e，是，就令p=T,返回RIGHT即可。若不是：

对当前结点的左子树T->lchild：ParentNode(T->lchild, e, p)，返回值赋值到state，若state!=ERROR,返回state即可；否则就返回Parent(T->rchild, e, p)。

(10)删除结点DeleteNode(T, e)：定义结点指针p=NULL，del=NULL, 分别用于查找和记录待删除结点地址。定义变量state，赋值为ParentNode(T, e, p)的返回值，记录待删除结点是左结点、右结点还是根节点。若：

1.state为OK：判断p->lchild是否为空，若是，就将T置为T->rchild，释放p指结点，返回OK即可。否则，判断p->rchild是否为空，若是，T置为T->lchild, 释放p指结点，返回OK即可。否则将T置为T->lchild, 并通过循环找到根节点左子树的最右结点，将其rchild置为p->rchild后，释放p指结点，函数返回OK即可。

2.state为LEFT：置del为p->lchild，判断del是否有左子树，若有，将del->lchild的最右结点的rchild置为del->rchild；若没有，置p为del->lchild即可。最后将del所指结点空间释放，函数返回OK即可。

3.state为RIGHT：置del为p->rchild, 判断del->lchild是否存在，若存在就将del->lchild的最右结点的rchild置为del->rchild，p->rchild置为del->lchild即可。否则，将p->rchild置为del->rchild。最终将del指结点空间释放，函数返回OK即可。

4.state为ERROR：函数返回ERROR即可，表明未找到待删除结点。

框图如图3-10：

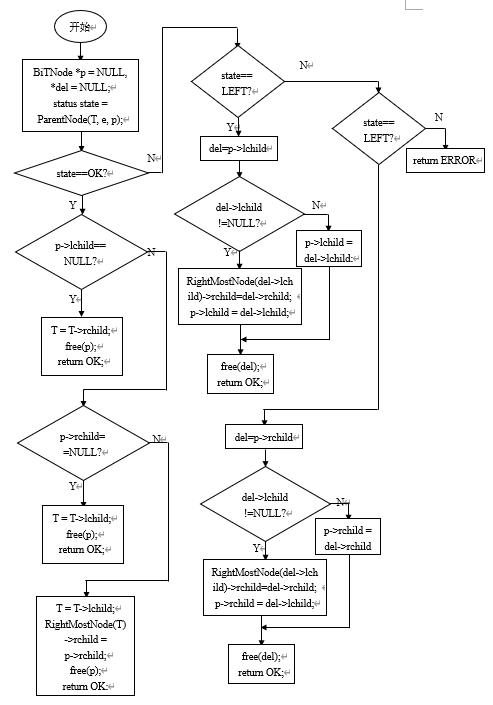


图3-10 删除结点

(11)前序遍历PreOrderTraverse(T, visit): 判断T是否为NULL，是就返回ERROR，否则就visit(T)，PreOrderTraverse(T->lchild, visit), PreOrderTraverse(T, visit)，返回OK即可。

(12)中序遍历InOrderTraverse(T, visit): 定义结点指针栈s，结点指针p = T;先初始化s，后进入循环：

循环条件为：p不为NULL 或者 栈s还非空。

执行：判断p是否为空，若非空，Push当前结点p进栈，p置为p->lchild

否则将当前栈顶元素Pop至p, visit(p)后，置p为p->rchild

循环结束后返回OK即可。

(13)后序遍历PostOrderTraverse(T, visit): 判断：当前结点是否为NULL，若是，则返回OK即可。否则，PostOrderTraverse(T->lchild, visit), PostOrderTraverse(T->rchild, visit), visit(T)，函数返回OK即可。

(14)按层遍历LevelOrderTraverse(T, visit):利用了

一个辅助结构：

typedef struct { //二叉树的结点definition，包含pos及pos对应的结点地址

int pos;

BiTNode \*node;

} def;

全局变量：整形i = 0，size=INIT\_DEF\_SIZE，position=1

def指针definition，指向一片大小为size\*sizeof(def)的空间

两个辅助函数：

1.树转definition：CreateDef(T, length)：

判断：当前结点为空时，函数返回NULL，否则执行后续步骤。

将definition[i].pos置为当前结点在满二叉树中对应的位置序号position，definition[i].node置为当前结点指针，后将i++，definition的length置为i。

将position \*= 2后，进入当前结点左子树CreateDef(T->lchild, length)，结束后将position/=2以还原成此结点的position。

将position置为2\*position+1，进入当前结点右子树CreateDef(T->rchild, length), 结束后position/=2

函数最后返回definition即可。

2.DefOrder(definition, length)：对definition按照pos值排序。此处采用冒泡排序法。

按层遍历LevelOrderTraverse(T, visit): 首先将全局变量全部置为初始状态，之后调用CreateDef(T, length)产生definition，再调用DefOrder函数对definition排序，之后利用for循环，以definition的length来遍历definition，对其中存储的树中结点信息调用visit函数，函数最后将definition的空间释放，返回OK即可。

此外，为了使系统功能更加丰富、更有实用性，我添加了部分功能模块，具体如下：

易于观察的遍历方式：

(15)树形遍历TreeLikeTraverse(T, visit)：基于按层遍历，后计算每一层的空格输出个数，输出斜线的部分利用函数数组存储每个结点左侧和右侧应有的输出状况(输出左斜线、输出右斜线、输出空格)

文件读取篇：

(16)树中的数据存入文件SaveList(T, SaveFile):利用文件指针以”w+”的形式打开(文件不存在时自动创建)文件名为SaveFile的文件，将二叉树T前序遍历扫描一遍，并将每一个结点的键值输入到目标文件中，且当前结点为NULL时，向文件内输入0 null, 每输入一个数据后就立即输入一个空格，否则会导致存入文件中的数据连在一起难以取出。输入完毕，over为1时，向文件内输入-1 null作为结束标志，利用fclose函数关闭该目标文件。最后返回OK；

(17)读取文件中数据存入线性表LoadList(T, LoadFile)：利用文件指针以”r”的形式打开文件名为LoadFile的文件，如果此步打开文件失败则函数返回ERROR，否则就将文件中每一组“键值-其他”数据利用fscanf函数读入一个TelemType类型的数组definition，读取完毕后先利用平台为我们提供的CreateBiTree(T, definition)函数创建二叉树后，利用fclose函数关闭该目标文件。最后返回OK；

森林操作篇：

新定义了一个菜单函数gather\_operate(Lists),其大体结构与main函数相同，将八种树操作集于其中，返回值类型为一个树的根节点指针。(树的修改操作需利用此返回值才得以实现)

typedef struct { //森林中每棵树的定义

BiTree T;

char name[30];

} Tree, \*pTree;

typedef struct { //森林定义(线性表结构存储)

pTree \*tree;

int length;

int size;

} FOREST, \*FST;

(18)森林初始化InitForest(fst):首先判断原森林是否存在，若存在则表明森林中可能已存在数据，不可再次进行初始化，返回INFEASIBLE;否则就对fst分配空间，将森林长度置为0，大小置为INIT\_SIZE，对其中的树指针分配空间后，返回OK即可。

(19)向集合中新增树AddList(fst,i,BiTreeName)：首先判断，fst是否为NULL，若是，森林不存在，返回INFEASIBLE；再判断fst当前长度是否已经是最大容量，如果是，则对当前森林扩容。若i为非法位置，直接返回ERROR；否则将i之后的树整体后移1位，再对fst->tree[i - 1]分配储存空间，利用strcpy函数将BiTreeName拷贝至fst->tree[i - 1]->name，之后利用已定义的函数InitBiTree (T)初始化fst->tree[i - 1]->T，完成后将森林的长度加一，再返回OK即可。

(20)移除树RemoveBiTree(fst, BiTreeName)：从森林中移除名为BiTreeName

的树。首先判断fst是否为NULL，若是，则返回INFEASIBLE；否则，遍历森林，比较每一棵树的名字，看是否与BiTreeName相同，若相同则利用ClearBiTree销毁这棵树，并释放其在森林中占据的空间，之后将原本在这棵树之后的树整体前移一个位置，再将森林长度减一，返回OK即可。若扫描结束后仍未发现同名树，则返回ERROR表示森林中不含名为BiTreeName的树。

(21)查找树LocateBiTree(fst,BiTreeName): 首先判断fst是否为NULL，若是，

则返回INFEASIBLE；遍历森林，比较每一棵树的名字，看是否与BiTreeName相同，若相同则返回其在数组中的序号加一即可，若扫描结束后仍未发现同名树，则返回ERROR表示森林中不含名为BiTreeName的树。

(22)遍历森林ForestTraverse(fst)：首先判断fst是否为NULL，是，则返回INFEASIBLE；不是，则执行如下步骤：

本函数内提供菜单，可以选择对每棵树的遍历方式(前序、中序、后序、层序、树形遍历)。具体实现中利用函数指针status (\*p)(BiTree, void (\*)(BiTree))，选择不同的遍历方式时，p指向相应的函数。

扫描森林，对每一棵树调用p指函数，之后返回OK即可。

(23)清空森林ClearForest(fst)：若fst==NULL，返回INFEASIBLE；否则扫描整个森林，对每一棵树，首先利用ClearForest函数销毁二叉树，之后释放该树在森林中占有的空间，并将该树的指针置为NULL。循环结束后，将森林的长度置为0，返回OK即可。

(24)销毁森林DestroyForest(fst)：若fst==NULL，返回INFEASIBLE；否则调用ClearForest函数清空森林后，释放森林所占空间，再将fst置为NULL后返回OK即可。

## 3.4 系统测试

系统的测试主要检查①系统是否能够不报错地正常运行；②二叉树的各项操作是否正确，包括创建二叉树、销毁二叉树、确定某树枝位置、遍历二叉树等等；③二叉树是否具有容错性；比如，在二叉树未初始化时，进行插入元素操作能够提示“二叉树不存在”，④二叉树的保存与读取是否正常。

1. 对初始状态的容错性进行测试

测试系统的容错性，看在二叉树未创建时，调用功能函数的提示是否正常。理论值应当提示“当前二叉树不存在，请先创建二叉树”。相关用例与结果如表3.2所示。

表3.2 初始状态容错性测试

|  |  |  |
| --- | --- | --- |
| 测试用例 | 程序输入 | 实验结果 |
| 用例1 | 输入‘4’ | 4 |
| 用例2 | 输入‘7’ | 7 |
| 用例3 | 输入‘12’ | 12 |
| 用例4 | 输入‘17’ | 17 |
| 用例5 | 输入‘20’ | 20 |
| 用例6 | 输入‘88’ | 88 |

3-8

1. 对二叉树的创建和二叉树的遍历的测试

创建一个二叉树，并对其进行遍历，看遍历结果是否符合理论结果。所创建的二叉树，如图3.5所示。

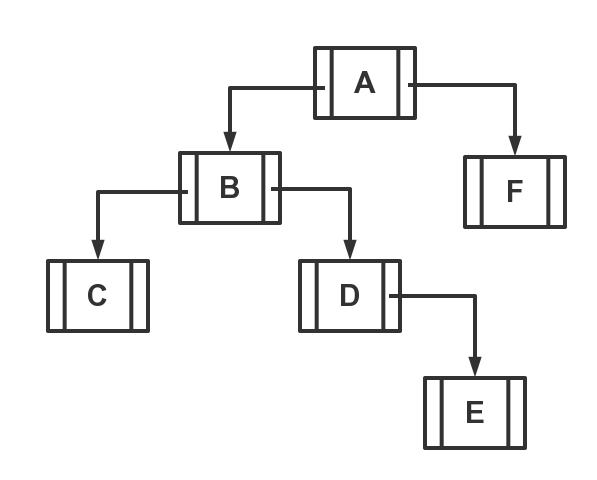


图3-8 创建的二叉树

按照上图创建二叉树，则输入ABC##D#E##F##，创建成功，结果如图3.6所示。

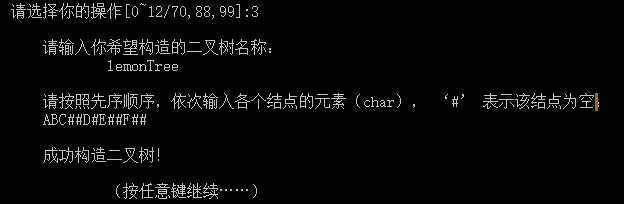


图3.6 成功构造二叉树

相关的遍历测试如表3.3所示。

表3.3 二叉树的遍历用例与结果

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程序输入 | 理论结果 | 实验结果 |
| 用例1 | 输入‘17’ | 打印序列  ABCDEF | 17 |
| 用例2 | 输入‘18’ | 打印序列  CBDEAF | 18 |
| 用例3 | 输入‘19’ | 打印序列  CEDBFA | 19 |
| 用例4 | 输入‘20’ | 打印序列  ABFCDE | 20 |

1. 对二叉树查询功能的测试

所涉及的功能包括，判空、求根结点、求深度、求特定结点的父母结点、求特定结点的左右子女结点、求特定结点的左右兄弟结点。相关用例与结果如表3.4和表3.5所示。

表3-9 二叉树判空/求深度/求根节点的用例与结果

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程序输入 | 理论结果 | 实验结果 |
| 用例1 | 输入‘5’ | 提示二叉树非空 | 非空 |
| 用例2 | 输入‘6’ | 提示深度为4 | 深度 |
| 用例3 | 输入‘7’ | 提示根节点为A | 根结点 |

表3-10 求各种结点的用例与结果

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程序输入 | 理论结果 | 实验结果 |
| 用例1 | 输入‘10’  再输入‘B’ | 返回A | 10B |
| 用例2 | 输入‘11’  再输入‘B’ | 返回‘C’ | 11C |
| 用例3 | 输入‘12’  再输入‘D’ | 返回‘E’ | 12 |
| 用例4 | 输入‘13’再输入‘B’ | 提示不存在 | 13A |
| 用例5 | 输入‘13’再输入‘F’ | 返回‘B’ | 13B |
| 用例6 | 输入‘14’再输入‘C’ | 返回‘D’ | 14D |

1. 对插子子树功能和删除子树功能的测试

插入子树功能和删除子树功能是二叉树功能函数中的两个重要组成部分。在本次测试中，试图在B的右结点下插入如下子树。

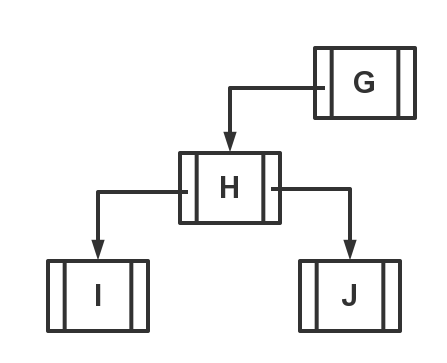


图3-11 插入的子树

得到新的树如图3-12所示。

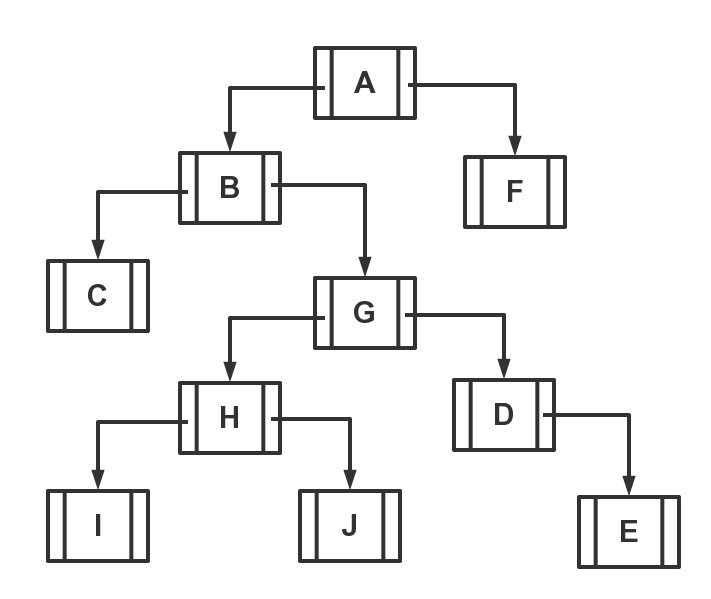


图3-12 得到的新树

按上图进行插入子树，提示插入成功。如图3-13所示。

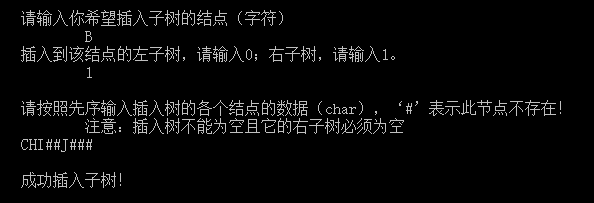


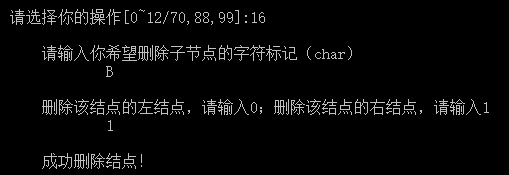
图3-13 成功插入子树

对新树进行遍历，测试用例如表3-14

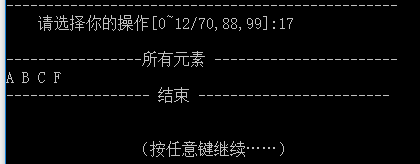
表3-14 对新树进行遍历的用例与结果

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程序输入 | 理论结果 | 实验结果 |
| 用例1 | 输入‘17’ | 打印序列  ABCGHIJDEF | 17 |
| 用例2 | 输入‘18’ | 打印序列  CBIHJGDEAF | 18 |
| 用例3 | 输入‘19’ | 打印序列  CIJHEDGBFA | 19 |
| 用例4 | 输入‘20’ | 打印序列  ABFCGHDIJE | 20 |

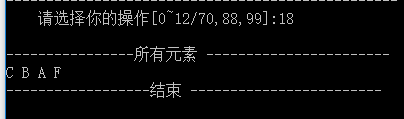
删除结点G，剩余的树的前序遍历应该是ABCF，中序遍历应该是CBAF。如图3.15(a)-(c)所示。



3-15(a)成功删除子节点



3-15(b)删除子节点后进行的先序遍历



3-15(c)删除子节点后进行的中序遍

1. 对树结点赋值的测试

对树节点进行赋值的用例如表3-16所示。

表3-16 对树结点进行赋值的用例与结果

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程序输入 | 理论结果 | 实验结果 |
| 用例1 | 输入‘9’输入‘A’，再输入’8‘ | 成功为“A”赋上‘8’值 | 赋值成功 |
| 用例2 | 输入‘8’再输入‘A’ | 成功打印出A的值‘8’ | 提取成功 |

（6）对多树切换和树的保存和读取进行测试

对当前树进行保存，保存成功如图3.17所示。

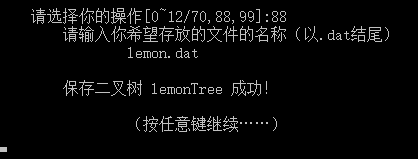


图3-17 保存二叉树成功

创建新树pearTree,创建功如图3-18所示。

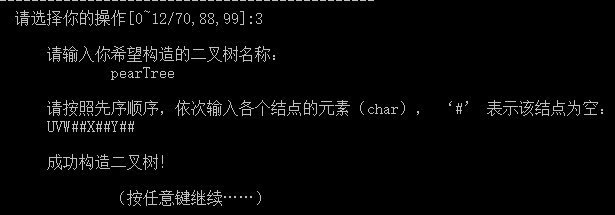


图3-18 pearTree创建成功

对新树进行先序遍历，如图3-19所示。

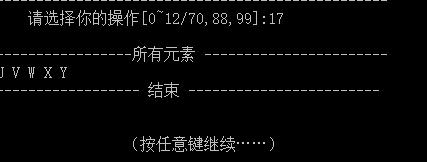


图3-19对新树进行的先序遍历

切换回原来的树，切换成功，并进行遍历，发现确实是原来的树。如图3-20所示。

|  |  |
| --- | --- |
| 成功转换树 | lemontree1 |

图3-20多树管理-切换树成功

对当前的lemonTree进行清空处理，并发现清空成功，如图3-21所示。

|  |  |
| --- | --- |
| 清空处理 | 空二叉树 |

图3-21 清空当前二叉树成功

重新载入先前保存的lemonTree结点信息，加载成功。如图3.16所示

|  |  |
| --- | --- |
| 载入新的二叉树 | 载入成功 |

图3-22 加载二叉树成功

## 3.5 实验小结

本次实验，主要操作是对基于二叉链表的二叉树这一存储结构来进行。在学习过程中，我有以下几点收获：

1. 灵活运用合适的数据结构真的很重要：不同于前两次的实验，从操作上来讲，这次实验中所用到的存储结构相较于前两次的线性存储结构明显要复杂一些，但此次实验也能看出其它数据结构的影子：非递归中序遍历(访问受限的线性表)、森林(利用线性表存储二叉树根结点地址)……而这，也让我感受到了在实际中对各种数据结构的灵活使用是多么重要。
2. 对递归这一方法有了更多的认识：此次实验中，不少函数用递归做起来是很舒服、很快的。以前学习递归其实就没有学明白导致实验最开始是很“畏惧”用递归的。但是，在结合了数据结构学习过程后，才初懂：递归其实就是一个函数里不断地产生“代码块”。之后结合离散数学上课所讲，辅以课外查找到的递归算法以及相关时间复杂度的分析后，我觉得我利用递归的能力又提高了一些。
3. 对静态变量多了一分警惕：由于此次实验，我初懂递归，便十分得瑟地大量地使用了递归，而且使用了很多全局变量，但是，第一次做完全没有注意到许多使用了静态变量的递归函数全都被我写成了“一次性函数”，即用过一次之后就直接“报废”的函数。之后没多久发现，递归也是一种多次调用啊！那么我在函数外部多次调用此函数，其中的静态变量没有手动将其还原成初始状态，导致非法访问、长度错误等等一系列问题。之后我将所有函数进行了静态变量处理，一趟大改，终于解决，我也对静态变量有了更加深入的了解与体会。
4. 对复杂类型的指针有了更深的理解：本次实验中，树形遍历函数，森林遍历函数这两个函数中，我用到了函数指针数组与函数指针。尤其是树形遍历函数，其中的斜线输出确实费了我不少心思：exe文件中我能控制的输出都只能是一行一行地输出，所以要想完整输出指定个数的斜线，必须要记录每一个结点的左右孩子状态，如果采用新的数据结构来记录，我写代码的开销势必又要加大，而且又会多出许多变量需要我来处理，很麻烦。于是我就想到，将输出函数直接存入一个函数地址数组中，遍历一遍树之后我就可以记录下各个结点下应该输出什么，很方便~很方便，唯一卡住的地方就是为这个数组动态分配存储空间(malloc)，我应该将其返回值强制转换成哪一种指针，怎么写？抱着试一试的心态，直接将定义去掉变量名填进去，编译，通过，运行理想，十分高兴。
5. 在本次实验中，对二叉树的基本操作有了一个整体的理解。其中，二叉树的构造和二叉树的遍历是两个的最为重要的内容。
6. 在本次实验中，二叉树的构造采用了先序构造的方式。构造时，路线沿着先序的方向，用‘#’号表示所指子树为NULL。若当前读取的字符不为空，则按先序依次构造树。在构造时，若上一次读取的字符不为空，则在左子树下插入本次读取的结点，否则，在右子树下插入。若当前读取的字符为空，则另当前所在结点指向NULL（上一字符不为空，左子树为NULL；反之，右子树为NULL）。
7. 除层序遍历外，二叉树的其他3种遍历都用到了递归的方式。事实上，三种遍历方法的算法基本相同，都是依次对左子树结点和右子树结点进行递归调用，只不过访问根结点的时机不同。先序遍历在两次调用递归前访问根节点，中序在中间，后序在最后。层序遍历主要用到了队列的思想，先入先出，从而使得每层能有序地输出。

# 4 基于邻接表的图实现

## 4.1 问题描述

在众多的数据结构中，图结构是应用最为广泛的一种。简单的，像地图导航、  
人物关系，复杂一点的如机器学习之类的，这些地方的数据存储都需要利用图来加以实现。具体实现中，由于图中各个点以及各条边的关联方式都与线性表不同，且通常一个点连几条边，每条边以及每个点的属性都是不能够立即确定下来的，所以，图是一种较线性表更为复杂的数据结构。且因其应用之广泛，所以对我们而言，熟练掌握图的存储方式及其基本操作就尤为重要，故本实验目的为：(1)加深对图的概念、基本运算的理解；⑵熟练掌握图的逻辑结构与物理结构的关系；⑶以邻接表作为物理结构，熟练掌握图基本运算的实现。

## 4.2 系统设计

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

(1) 创建图：函数名称是CreateCraph(G,V,VR)；初始条件是V是图的顶点集，

VR是图的关系集；操作结果是按V和VR的定义构造图G。

(2)销毁图：函数名称是DestroyGraph(G)；初始条件图G已存在；操作结果

是销毁图G。

(3)查找顶点：函数名称是LocateVex(G,u)；初始条件是图G存在，u是和G中顶点关键字类型相同的给定值；操作结果是若u在图G中存在，返回关键字为u的顶点位置序号（简称位序），否则返回其它表示“不存在”的信息。

(4)顶点赋值：函数名称是PutVex (G,u,value)；初始条件是图G存在，u是和G中顶点关键字类型相同的给定值；操作结果是对关键字为u的顶点赋值value。

(5)获得第一邻接点：函数名称是FirstAdjVex(G, u)；初始条件是图G存在，u是G中顶点的位序；操作结果是返回u对应顶点的第一个邻接顶点位序，如果u的顶点没有邻接顶点，否则返回其它表示“不存在”的信息。

(6)获得下一邻接点：函数名称是NextAdjVex(G, v, w)；初始条件是图G存在，v和w是G中两个顶点的位序，v对应G的一个顶点,w对应v的邻接顶点；操作结果是返回v的（相对于w）下一个邻接顶点的位序，如果w是最后一个邻接顶点，返回其它表示“不存在”的信息。

(7)插入顶点：函数名称是InsertVex(G,v)；初始条件是图G存在，v和G中的顶点具有相同特征；操作结果是在图G中增加新顶点v。（在这里也保持顶点关键字的唯一性）

(8)删除顶点：函数名称是DeleteVex(G,v)；初始条件是图G存在，v是和G中顶点关键字类型相同的给定值；操作结果是在图G中删除关键字v对应的顶点以及相关的弧。

(9)插入弧：函数名称是InsertArc(G,v,w)；初始条件是图G存在，v、w是和G中顶点关键字类型相同的给定值；操作结果是在图G中增加弧<v,w>，如果图G是无向图，还需要增加<w,v>。

(10)删除弧：函数名称是DeleteArc(G,v,w)；初始条件是图G存在，v、w是和G中顶点关键字类型相同的给定值；操作结果是在图G中删除弧<v,w>，如果图G是无向图，还需要删除<w,v>。

(11)深度优先搜索遍历：函数名称是DFSTraverse(G,visit())；初始条件是图G存在；操作结果是图G进行深度优先搜索遍历，依次对图中的每一个顶点使用函数visit访问一次，且仅访问一次。

(12)广深度优先搜索遍历：函数名称是BFSTraverse(G,visit())；初始条件是图G存在；操作结果是图G进行广度优先搜索遍历，依次对图中的每一个顶点使用函数visit访问一次，且仅访问一次。

## 4.3 系统实现

本系统通过命令式菜单来调用各函数功能，系统开始运行时会输出菜单界面，依照菜单中给出的各功能的指令输入命令，以实现相关功能的使用。总体实现为：首先根据每一个基本操作写出相关函数的基本功能实现，后再针对异常状况进行合理处理，完成每一个函数的具体实现。后在main函数中定义一个op存储当前命令，通过switch语句，针对op的不同值在case op中调用对应函数以实现相应功能。

每个函数的具体实现方法如下：

1. 创建图：函数名称是CreateCraph(G,V,VR); 首先判断所给点集是否超出最

大点容量，若是则返回ERROR，否则，继续查找点集内是否存在两个点key值相同，若存在亦返回ERROR，否则，就先将图G的点个数vexnum和边个数arcnum置0，遍历点集，将点集中点的信息存入图G中vertices的data域中，并同时将每个点指向连接它自己的第一条边的指针置为NULL。之后开始读取边集，对边集中每个二元组，利用查找结点的LocateNode函数，找到相应key值的点的序号，此步如果序号值为-1，说明图中没有该点，函数返回ERROR即可，否则，对两个点都要判断其后置链表中是否已存在对方结点的信息，若存在则跳过此步操作，不存在就在这两个点的后置链表中头插新节点，保存对方结点的序号，后执行下一轮操作，直到整个循环结束后，将循环次数除以2之后将其赋值给G.arcnum并返回OK即可。

2. 销毁图：函数名称是DestroyGraph(G); 循环遍历图中所有点，利用循环将

所有点的后置链表空间完全释放后将指针置为NULL。遍历结束后，将G.vexnum和G.arcnum全部置为0后返回OK即可。

3. 查找顶点：函数名称是LocateVex(G,u)；遍历图中现存所有点，对每一个点依次比较key值是否与u相同，相同则返回当前结点在图中的位序，若遍历结束之后仍然没有key值为u的点，则返回-1。

4. 顶点赋值：函数名称是PutVex (G,u,value)；首先在图中查找是否已经存在的key值为value.key的点，若存在，返回ERROR，否则，遍历当前图中所有点，并对每个点判断其键值是否为u，若当前点键值为u，则将此点的键值改为value.key，data.others改为value.others后返回OK即可。若遍历结束后仍未找到该点，返回ERROR。

5. 获得第一邻接点：函数名称是FirstAdjVex(G, u)；首先利用LocateVex函数查找键值为u的点在图中的位序pos，若pos==-1，说明图中没有键值为u的点，返回-1即可；若pos!=-1，返回G中第pos个点的firstarc所连接的点的序号。

6. 获得下一邻接点：函数名称是NextAdjVex(G, v, w)；首先在图G中利用LocateVex函数查找key值为v的点的位序pos，若pos为-1，返回-1即可，否则先用指针p指向G中第pos个点的第一条边，之后利用循环遍历第pos个点的后置链表，每次循环，将p置为pànextarc；若找到键值为w的点的位序，则将p置为链表中此结点的后继并跳出遍历，否则则循环到遍历结束为止。遍历结束后，判断p是否为NULL，是则返回-1，不是则返回pàadjvex。

7. 插入顶点：函数名称是InsertVex(G,v)；首先判断当前图中点数是否已达到MAX\_VERTEX\_NUM或者待插入结点的key值是否已经存在于当前图中，是则返回ERROR，否则就将点G.vertices[G.vexnum]的data域中的数据置为v中数据，并将该点的后接指针置为NULL，同时将图中点数加一后返回OK。

8. 删除顶点：函数名称是DeleteVex(G,v)；首先利用LocateVex函数查找G中是否有key值为v的点，若不存在或者当前图中只有一个结点，则返回ERROR；否则删除这个点的后置链表中存储的每一个位序所对应的结点的后置链表中这个点的位置存储结点，之后将这个点的后置链表清空，将图的边数减去此链表的长度，再将图中此点之后的点的位序全体前移1，同时将所有点的后置链表中存储的信息更改为新图中的各个点的序号，再将图的点数减一，之后返回OK即可。

9. 插入弧：函数名称是InsertArc(G,v,w)；首先判断当前图中点数是否小于2，若是，则返回ERROR；否则利用LocateVex函数继续判断v，w是否都存在于当前图中，若不是，则返回ERROR，否则继续判断v点后置链表中是否已存在w点的位置信息，若是则说明v和w点之间已存在连接，函数返回INFEASIBLE，否则就在v点的后置链表中头插插入w的位置信息结点，在w点的后置链表中头插插入v的位置信息结点，再将图的边数加一后函数返回OK即可。

10. 删除弧：函数名称是DeleteArc(G,v,w)；首先判断当前图中点数是否小于2，若是，则函数返回ERROR；否则利用LocateVex函数继续判断v，w是否都存在于当前图中，若不是，则返回ERROR，否则继续判断v点后置链表中是否存在w点的位置信息，若不存在则说明v和w点之间不存在连接，函数返回INFEASIBLE，否则就将v的后置链表中存储w的位置信息的结点删除，将w的后置链表中存储v的位置信息的结点删除，再将图的边数减一后函数返回OK即可。

11. 深度优先搜索遍历：函数名称是DFSTraverse(G,visit())；此函数递归实现

a.首先定义静态整型数组rec, 利用calloc函数分配大小为G.vexnum\* sizeof(int)的空间(用于记录每个点的访问状况，若第i个点被访问后rec[i]置1，未被访问保持rec[i]为0。)。之后定义静态整型变量pos，head，pos用于记录每个子分支的首结点序号，head用于记录每个连通分支的首结点序号，head与pos初值均为0；

b.首先判断当前图的结点数是否为0，若是，则直接return OK即可，否则，利用整型变量this\_pos记录pos，定义边指针p=G.[this\_pos].firsarc，即用于记录第this\_pos个点的第一邻接点，之后判断rec[this\_pos]是否为0，若是，则调用visit函数visit当前结点，并将rec[this\_pos]置为1，之后找到该点的后置链表中第一个尚未被访问的结点序号。

c.之后判断p是否等于NULL并且this\_pos是否不等于head，若是，则当前点的后置链表中所有点均已访问完毕，且当前联通分支仍未被访问完，return OK回溯；而如果p==NULL并且this\_pos==head，则说明当前连通分支已访问完成，{遍历点集，判断是否存在第i个点，使得rec[i]!=0，(即判断整个图中是否还有点未被访问)，若是，则将head置为当前为访问结点的序号，将pos置为head，flag置为1后跳出查找所用循环。若不是，则查找结束之后flag保持为0。之后在循环外判断flag是否为1，是，则说明还有其他连通分支，跳出深度遍历第二层循环，准备遍历另一个连通分支。否则就处理静态局部变量：将rec申请空间free，重新分配全0内存置rec，pos置0，head置0，整个函数返回OK即可。}

此外，如果p!=NULL，则将p指序号赋值到pos，深度遍历以序号为pos的点为起点的子图，最后将pos置回为当前子图的起点this\_pos。

12. 广深度优先搜索遍历：函数名称是BFSTraverse(G,visit())；基本思想是在visitNode函数内利用队列的存储结构进行处理，首先图的起点进队，之后将队首结点的邻接点中未被访问的结点进队后，访问队首结点，之后令其出队。循环上述过程，直至head==tail停止。思路可行，实现过程如下：首先定义数组Q，存储图结点后置链表指针，为避免野指针，仍利用calloc分配数组空间。之后定义记录数组rec，记录图中每个结点的访问状态；定义head、tail两整型变量，作为队列Q的队首和队尾，初值为0。之后for循环，遍历图中所有点，当存在某个点仍未被访问时，将此点入队后将其访问，并将该点访问状态改为1，之后进入函数visitNode(ALGraph G，ArcNode \*\*Q, int& head, int &tail, int \*rec, void (\*visit)(VertexType))，函数内部：while循环，终止条件为head==tail，循环内：定义指针p存储Q内队首元素，如果p==NULL，队列访问完毕，返回OK即可。否则进行判断，若p指结点未被访问，访问此结点，并将此结点访问状态置1；已被访问，跳过此步。之后遍历队首结点的全部邻接点，未被访问的全部进队。待此大循环结束后，图的广度优先遍历就完成了，函数visitNode结束，之后在BFSTraverse内将数组Q, rec所申请空间释放后，返回OK即可。

此外，为了使系统功能更加丰富、更有实用性，我添加了部分功能模块，具体如下：

易于观察的遍历方式：

(15)图形遍历GraphLikeTraverse(G, visit)：此处利用HavePython()函数和system(cmd)函数外调drawGraph.py文件(内部import了networkx和matplotlib.pyplot)，分别判断系统是否安装了python，尝试安装相关python库，以及画出当前图对应的无向图(数据通过存档文件来传递到py中)。(过程涉及命令行指令的部分，与本实验干系不大，不作赘述)

HavePython():首先获取当前系统的环境变量中的Path变量，遍历各个变量，将每个变量后接上python.exe，用fopen(路径，”r”);的返回值，判断是否有文件夹中是否有python.exe，若有，函数返回true，否则返回false。

绘图py：从文件读取图像信息组成点、边集，之后将其加入图中，利用plt中的show函数绘出图像。

文件读取篇：

(16)树中的数据存入文件SaveList(G, SaveFile):利用文件指针以”w+”的形式打开(文件不存在时自动创建)文件名为SaveFile的文件，将图G前序遍历扫描一遍，并将每一个结点的键值输入到目标文件中，且当前结点为NULL时，向文件内输入0 null, 每输入一个数据后就立即输入一个空格，否则会导致存入文件中的数据连在一起难以取出。输入完毕，over为1时，向文件内输入-1 null作为结束标志，利用fclose函数关闭该目标文件。最后返回OK；

(17)读取文件中数据存入线性表LoadList(T, LoadFile)：利用文件指针以”r”的形式打开文件名为LoadFile的文件，如果此步打开文件失败则函数返回ERROR，否则就将文件中每一组“键值-其他”数据利用fscanf函数读入一个TelemType类型的数组definition，读取完毕后先利用平台为我们提供的CreateBiTree(T, definition)函数创建二叉树后，利用fclose函数关闭该目标文件。最后返回OK；

多图操作篇：

新定义了一个菜单函数gather\_operate(GS),其大体结构与main函数相同，将七种图操作集于其中，返回值类型为一个图的指针。(图的修改操作需利用此返回值才得以实现)

typedef struct { //多图中每个图的定义

char name[30];

ALGraph G;

}Graph, \*G;

typedef struct { //多图定义

G \*elem;

int length;

int size;

}ALGraphs;

(18)多图集合初始化InitALGraphs(GS):首先判断原集合是否存在，若存在则表明集合中可能已存在数据，不可再次进行初始化，返回INFEASIBLE;否则就对GS.elem分配空间，将森林长度置为0，大小置为INIT后，返回OK即可。

(19)向集合中新增图AddGraph(GS,i,GraphName)：首先判断，GS是否为NULL，若是，集合不存在，返回INFEASIBLE；再判断GS当前长度是否已经是最大容量，如果是，则对当前集合扩容。若i为非法位置，直接返回ERROR；否则再遍历集合，查找是否有重名图，若有，返回FAIL，否则先建立一个新图指针p，malloc分配空间，之后利用CreateGraph函数为p指针所指图填入数据，之后将i之后的树整体后移1位，再将GS.elem[i – 1]置为p，将GS.length加一后返回OK即可。

(20)移除图RemoveGraph(GS, GraphName)：从集合中移除名为GraphName

的图。首先判断GS.elem是否为NULL，若是，则返回INFEASIBLE；否则，遍历集合，比较每一个图的名字，看是否与GraphName相同，若相同则利用DestroyGraph函数销毁这个图，并释放其空间，之后将原本在这个图之后的图整体前移一个位置，再将集合长度减一，返回OK即可。若扫描结束后仍未发现同名树，则返回ERROR表示集合中不含名为BiTreeName的图。

(21)查找图LocateGraph(GS,GraphName): 首先判断GS.elem是否为NULL，

若是，则返回INFEASIBLE；遍历集合，比较每一个图的名字，看是否与GraphName相同，若相同则返回其在数组中的序号加一即可，若扫描结束后仍未发现同名图，则返回ERROR表示集合中不含名为GraphName的图。

(22)遍历集合ALGraphsTraverse(GS)：首先判断GS.elem是否为NULL，是，则返回INFEASIBLE；不是，则执行如下步骤：

本函数内提供菜单，可以选择对每个图的遍历方式(深度优先、广度优先、图形、邻接表遍历)。具体实现中利用函数指针status (\*p)(ALGraph, void (\*visit)(VertexType))，选择不同的遍历方式时，利用switch结构将p指向相应的函数。

之后扫描集合，对每一个图调用p指函数，之后返回OK即可。

(23)清空集合ClearALGraphs(GS)：若GS.elem==NULL，返回INFEASIBLE；否则扫描整个集合，对每一个图，首先利用DestroyGraph函数销毁图，之后释放该树在集合中占有的空间，并将该树的指针置为NULL。循环结束后，将集合的长度置为0，返回OK即可。

## 4.4 系统测试

系统的测试主要检查①系统是否能够不报错地正常运行；②图的各项操作是否正确，包括创建图、销毁图、确定某顶点位置、遍历图等等；③图是否具有容错性；比如，在图未初始化时，进行插入元素操作能够提示“图尚未创建”

（1）对初始状态的容错性进行测试

测试系统的容错性，看在图未创建时，调用功能函数的提示是否正常。理论值应当提示“图还未创建，请先创建图”。相关用例与结果如表4.2所示。

表4.2 初始状态容错性测试

|  |  |  |
| --- | --- | --- |
| 测试用例 | 程序输入 | 实验结果 |
| 用例1 | 输入‘2’ | 2号 |
| 用例2 | 输入‘3’ | 3号 |
| 用例3 | 输入‘5’ | 5号 |
| 用例4 | 输入‘7’ | 7号 |
| 用例5 | 输入‘10’ | 10号 |
| 用例6 | 输入‘13’ | 13号 |

（2）对图的创建和图的遍历的测试

创建一个图，并对其进行遍历，看遍历结果是否符合理论结果。所创建的图，如图所示

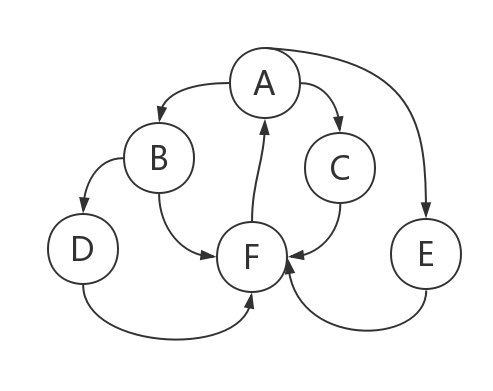


图4.4 实验用例中创建的图

按照上图所示，输入图的各项数据，包括顶点数、边数、顶点信息、弧的信息。创建成功，如图所示。

|  |  |
| --- | --- |
| 图的创建1370380 | 图的创建2280430 |
| 图的创建3A240389 | 图的创建4 |

图4.5 实验创建图的结果

分别对图进行深度优先遍历和广度优先遍历，遍历结果如表4.3所示。

表4.3 对创建图进行的深度优先遍历和广度优先遍历

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程序输入 | 理论结果 | 实验结果 |
| 用例1 | 输入‘12’ | 遍历序列为ABDFCE | 深度优先遍历 |
| 用例2 | 输入‘13’ | 遍历序列为ABCEDF | 广度优先遍历 |

1. 对图的查询功能进行测试

查询功能包括对顶点的查询、对顶点最近邻接顶点的查询、对顶点邻接顶点中某顶点的下一个顶点进行查询、对顶点的值进行查询（首先需要赋值），相关用例与测试结果，如表4.4所示。

表4.4 对图查询功能进行测试的用例与结果

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程序输入 | 理论结果 | 实验结果 |
| 用例1 | 输入‘3’  再输入‘C’ | 显示位于邻接表的第三个 | QQ截图20161208105159 |
| 用例2 | 输入‘6’再输入‘A’ | 显示邻接结点为‘B’ | 邻接结点 |
| 用例3 | 输入‘7’；输入‘A’再输入‘B’ | 显示邻接结点为  ‘C’ | 邻接结点为C |
| 用例4 | 输入‘7’；  输入‘A’再输入‘C’ | 显示邻接结点为‘E’ | 390邻接结点 |
| 用例5 | 输入‘5’；  输入‘B’再输入1995 | 成功为‘B’赋值1995 | 成功给B赋值 |
| 用例6 | 输入‘4’再输入‘B’ | 输出‘B’的值为1995 | B的值为1995 |

1. 对插入和删除功能的测试

在图中插入一个顶点G，并将G与原图作如下方式连接。

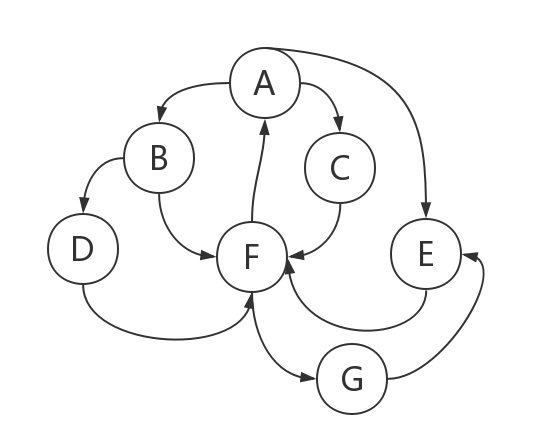


图4.6 插入一个顶点后的图模型

首先进行插入相关的测试，插入结点G，以及邻边G→E和F→G。相关插入用例如表4.5所示。

表4.5 插入结点G的相关用例

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程序输入 | 理论结果 | 实验结果 |
| 用例1 | 输入‘8’再输入‘G’ | 成功插入结点G | 成功插入结点G |
| 用例2 | 输入‘10’，再输入‘G’，‘E’ | 成功插入邻边  G→E | 成功在GE间插入弧 |
| 用例3 | 输入‘10’，再输入‘F’，‘G’ | 成功插入邻边  F→G | 成功在FG间插入弧 |

在插入G后，对插入后的图进行遍历。看插入后的图是否正常。相关用例如表4.6所示。

表4.6 对插入G后的图进行遍历

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程序输入 | 理论结果 | 实验结果 |
| 用例1 | 输入‘12’ | 遍历序列为ABDFGEC | 插入后的遍历1 |
| 用例2 | 输入‘13’ | 遍历序列为ABCEDFG | 插入后的遍历2 |

接着，对邻边和顶点进行删除操作，使得图恢复到插入G前的状态。删除操作的用例如表4.7所示。

表4.7 对结点G及相关邻边进行删除操作的用例

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程序输入 | 理论结果 | 实验结果 |
| 用例1 | 输入‘11’；输入‘G’、‘E’ | 成功删除邻边  G→E | 成功删除GE |
| 用例2 | 输入‘11’；  输入‘F’‘G’ | 成功删除邻边  F→G | C成功删除FG |
| 用例3 | 输入‘9’；输入‘G’ | 成功删除顶点 | 成功删除结点G |

对删除顶点G后的图再次进行遍历，以确定删除过程中没有出现问题。删除用例如表4.8所示。

表4.8 删除顶点G后对图进行遍历的测试用例及结果

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程序输入 | 理论结果 | 实验结果 |
| 用例1 | 输入‘12’ | 遍历序列为ABDFCE | 深度优先遍历 |
| 用例2 | 输入‘13’ | 遍历序列为ABCEDF | 广度优先遍历 |

1. 对图的销毁进行测试

对图的销毁进行测试，确保图能够被正常的销毁，销毁后选用其他功能时应提示“图还没有创建”。相关用例如表4.9所示。

表4.9 销毁图的用例与结果

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程序输入 | 理论结果 | 实验结果 |
| 用例1 | 输入‘2’ | 成功销毁图 | 成功销毁图 |
| 用例2 | 输入‘3’ | 提示图尚未创建 | 图还未创建 |

除上述功能外，该系统还支持对有向网进行操作；基本操作与有向图相同，唯一不同的是，在创建时需要输入权值。创建如图4.7所示的有向网。

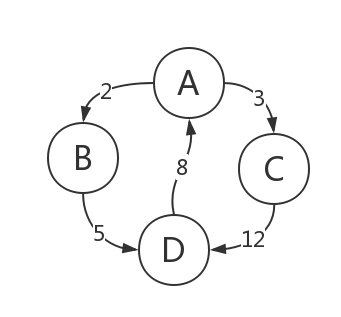


图4.7 实验用例有向网

对上图所示的有向网进行创建，分别输入图的基本信息、弧尾弧头以及权值等信息。创建过程如图4.8所示。

|  |  |
| --- | --- |
| 网1网3 | 有向网2 |

图4.8 创建网的过程

对有向网进行深度优先遍历和广度优先遍历，确认创建无误，相关用例如表4.10所示。

表4.10 对有向网进行遍历的用例与结果

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程序输入 | 理论结果 | 实验结果 |
| 用例1 | 输入‘12’ | 遍历序列为ACDB | 有向网遍历1 |
| 用例2 | 输入‘13’ | 遍历序列为ACBD | 有向网遍历2 |

## 4.5 实验小结

通过本次实验，实验者对以邻接表为物理结构的图有了更深入的认识。特别的，实验者对图的构建与图的遍历有了更好的理解。

在邻接表中，顶点是作为弧的头结点存放在线性表中的；而与顶点相连的弧则以链表的形式串在顶点所在的头结点上。利用这种储存形式，很容易求得顶点和边的各种信息。

深度优先遍历运用到了递归的思想。首先访问结点v，接着以同样的方式访问结点v的邻接结点。从本质上而言，深度优先遍历类似于树的先序遍历。

广度优先遍历用到了队列作为辅助。先入先出式的遍历，从一点为中心，层层铺设开来。从本质上而言，广度优先遍历类似于树的层序遍历。

# 参考文献

[1] 严蔚敏等. 数据结构(C语言版). 清华大学出版社

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

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

[4] 严蔚敏等.数据结构题集(C语言版). 清华大学出版社

# 附录A 基于顺序存储结构线性表实现的源程序

/\* Linear Table On Sequence Structure \*/

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <string.h>

#include <conio.h>

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

#define FAIL -3//预定义常量

typedef int status;//函数状态结果标志

typedef int ElemType; //数据元素类型定义

#define LIST\_INIT\_SIZE 100//线性表存储结构的最大分配量

#define LISTINCREMENT 10//线性表的分配增量

typedef struct{ //顺序表（顺序结构）的定义

ElemType \* elem;//抽象数据元素

int length;//表长

int listsize;//存储容量

}SqList;

typedef struct{ //线性表的管理表定义

struct { char name[30];

SqList L;

} elem[10];

int length;

int listsize;

}LISTS;

status InitList(SqList& L); //线性表初始化

status InitInput(SqList &L); //线性表输入数据

status DestroyList(SqList& L); //销毁线性表

status ClearList(SqList&L); //清空线性表

status ListEmpty(SqList L); //线性表判空 return status

int ListLength(SqList L); //求表长 return int

status GetElem(SqList L,int i,ElemType& e); //获取指定位置 i 处元素 e

status LocateElem(SqList L,ElemType e); //由元素内容e 查找元素位置return status

status PriorElem(SqList L,ElemType cur,ElemType &pre\_e);//由当前元素内容cur 获取其前驱元素pre\_e

status NextElem(SqList L,ElemType cur,ElemType &next\_e);//由当前元素内容cur 获取其后继元素next\_e

status ListInsert(SqList&L,int i,ElemType e); //向线性表指定位置 i 前插入元素e

status ListDelete(SqList&L,int i,ElemType& e); //删除位置 i 处元素存入 e 中

status ListTrabverse(SqList L); //遍历当前线性表

status SaveList(SqList L,char FileName[]); //当前线性表存档

status LoadList(SqList &L,char FileName[]); //当前线性表读档

status InitLists(LISTS &Lists); //线性表集合初始化

status AddList(LISTS &Lists, char ListName[]); //向集合中添加名为 ListsName 的线性表

status RemoveList(LISTS &Lists,char ListName[]);//删除集合中名为 ListsName 的线性表

int LocateList(LISTS Lists,char ListName[]); //在当前集合中查找名为 ListsName 的线性表

status ListsTraverse(LISTS Lists); //遍历当前集合

status SaveLists(LISTS Lists, char \*FileName); //线性表集合存档

status LoadLists(LISTS &Lists, char \*FileName); //线性表集合读档 （有问题）

status ChangeList(SqList &L); //修改集合中指定线性表

SqList \*gather\_operate(LISTS &Lists); //线性表集合操作

char\* SaveFile = "C:\\Temp.stu\_save.dat";

char\* LoadFile = "C:\\Temp.stu\_save.";

char\* SaveFile1 = "C:\\Temp.stu\_save.dat";

char\* LoadFile1 = "C:\\Temp.stu\_save.dat";

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

int main(){

SqList \*L = (SqList \*)malloc(sizeof(SqList)); int op=1; int flag = 0;

(\*L).elem = NULL;

LISTS Lists;

Lists.elem[0].L.elem = NULL;

SqList \*temp = (SqList \*)malloc(sizeof(SqList));;

SqList \*L\_ope = (SqList \*)malloc(sizeof(SqList));

(\*temp).elem = NULL;

while(op){

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

printf(" 基于顺序存储的线性表操作菜单\n");

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

printf(" 1. 初始化线性表 7. 查找元素 13. 保存线性表\n");

printf(" 2. 摧毁线性表 8. 前驱元素 14. 读取线性表\n");

printf(" 3. 清空线性表 9. 后继元素 15. 线性表集合操作\n");

printf(" 4. 线性表判空 10. 插入元素 0. 退出\n");

printf(" 5. 线性表表长 11. 删除元素\n");

printf(" 6. 由位置求元素 12. 遍历线性表\n");

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

if(op)

printf(" 请选择你的操作[0~15]:");

scanf("%d",&op);

switch(op){

case 1:

if(InitList(\*L)==OK)

{

printf("线性表创建成功！\n");

InitInput(\*L);

}

else printf("线性表创建失败！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 2:

int j;

j = DestroyList(\*L);

if(j == INFEASIBLE) printf("当前线性表不存在！\n");

else printf("销毁成功！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 3:

j = ClearList(\*L);

if(j == INFEASIBLE) printf("当前线性表不存在！\n");

else printf("清空成功！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 4:

j = ListEmpty(\*L);

if(j == INFEASIBLE) printf("当前线性表不存在\n");

else if(j) printf("当前线性表为空\n");

else printf("当前线性表不为空\n");

printf("----任意键继续----\n");

\_getch();

break;

case 5:j = ListLength(\*L);

if(j == INFEASIBLE) printf("当前线性表不存在！\n");

else printf("当前线性表长度为%d\n",j);

printf("----任意键继续----\n");

\_getch();

break;

case 6:

if((\*L).elem==NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

int i, e;

printf("查找第几位元素？\n");

scanf("%d",&i);

j = GetElem(\*L, i, e);

if(j == ERROR) printf("查找位置不合法！\n");

else printf("当前线性表中第%d个元素为%d\n", i, e);

printf("----任意键继续----\n");

\_getch();

break;

case 7:

printf("输入所需查找的元素内容\n");

scanf("%d", &e);

j = LocateElem(\*L, e);

if(j == ERROR) printf("表中不存在元素 %d\n",e);

else if(j == INFEASIBLE) printf("当前线性表为不存在！\n");

else printf("元素%d位于线性表中第%d个位置\n", e, j);

printf("----任意键继续----\n");

\_getch();

break;

case 8:

int prior;

printf("输入当前元素\n");

scanf("%d", &e);

j = PriorElem(\*L, e, prior);

if(j == ERROR) printf("此元素无前驱！\n");

else if(j == INFEASIBLE) printf("当前线性表不存在！\n");

else printf("当前元素前驱为%d\n", prior);

printf("----任意键继续----\n");

\_getch();

break;

case 9:

int next;

printf("输入当前元素\n");

scanf("%d", &e);

j = NextElem(\*L, e, next);

if(j == ERROR) printf("此元素无后继！\n");

else if(j == INFEASIBLE) printf("当前线性表不存在！\n");

else printf("当前元素后继为%d\n", next);

printf("----任意键继续----\n");

\_getch();

break;

case 10:

printf("输入加入元素\n");

scanf("%d", &e);

printf("想插入第几个元素前？\n");

scanf("%d", &i);

j = ListInsert(\*L, i, e);

if(j == ERROR) printf("插入位置非法！\n");

else if(j == INFEASIBLE) printf("当前线性表不存在！\n");

else

{

printf("插入成功！\n");

if(!ListTrabverse(\*L))

printf("当前线性表不存在！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 11:

printf("删除第几个元素？\n");

scanf("%d",&i);

j = ListDelete(\*L, i, e);

if(j == INFEASIBLE) printf("当前线性表不存在！\n");

else if(j == ERROR) printf("删除位置非法！\n");

else

{

printf("删除成功！\n");

if(!ListTrabverse(\*L))

printf("当前线性表不存在！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 12:

if(ListTrabverse(\*L) == INFEASIBLE) printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 13:

if(SaveList(\*L, SaveFile) == INFEASIBLE) printf("当前线性表不存在！\n");

else printf("已将数据存入 %s\n",SaveFile);

printf("----任意键继续----\n");

\_getch();

break;

case 14:

if((j = LoadList(\*L, LoadFile)) == ERROR) printf("当前线性表中已有数据！\n");

else if(j == OK) printf("数据已从 %s 中读出\n",LoadFile);

else if(j == INFEASIBLE) printf("当前线性表不存在！请先初始化！\n");

else if(j == FAIL) printf("存档文件建立失败！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 15:

if(flag == 1)

{

printf("当前有未结束的线性表修改，请先结束前一修改！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

L\_ope = gather\_operate(Lists);

if((\*L\_ope).elem != NULL)//只要选择了集合操作中的修改，就将主界面线性表置换成集合中所选线性表

{

temp = L; //temp暂存主界面线性表数据

L = L\_ope; //主界面后台线性表置换为集合内所选线性表

flag = 1; //系统状态置为集合操作态

}

break;

case 0:

break;

}//end of switch

if(flag && op == 0) //flag判断当前是否在线性表集合操作中，此时退出返回主界面

{

L = temp; //返还原始主界面线性表数据

op = 1; //保持不退出while

flag = 0; //状态置为初始界面

printf("原线性表数据已载入！\n");

\_getch();

}

}//end of while

printf("欢迎下次再使用本系统！\n");

printf("----任意键结束----\n");

\_getch();

return 0;

}//end of main()

status InitList(SqList& L)

// 线性表L不存在，构造一个空的线性表，返回OK，否则返回INFEASIBLE。

{

if(!L.elem)

{

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

L.length = 0;

L.listsize = LIST\_INIT\_SIZE;

return OK;

}

else

{

return INFEASIBLE;

}

}

status InitInput(SqList &L)

{

int ord, temp;

printf("输入数据按1，否则按0\n");

scanf("%d",&ord);

if(ord == 0)

{

return 0;

}

else

{

printf("请输入一组数据，0为结束标志\n");

scanf("%d",&temp);

while(temp != 0)

{

if(L.length > L.listsize)

{

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

L.listsize += LISTINCREMENT;

}

ListInsert(L, L.length + 1, temp);

scanf("%d",&temp);

}

printf("输入成功！\n");

return OK;

}

}

status DestroyList(SqList& L)

// 如果线性表L存在，销毁线性表L，释放数据元素的空间，返回OK，否则返回INFEASIBLE。

{

if(!L.elem)

{

return INFEASIBLE;

}

else

{

free(L.elem);

L.length = 0;

L.listsize = 0;

L.elem = NULL;

return OK;

}

}

status ClearList(SqList& L)

// 如果线性表L存在，删除线性表L中的所有元素，返回OK，否则返回INFEASIBLE。

{

if(!L.elem)

{

return INFEASIBLE;

}

else

{

free(L.elem);

L.length = 0;

L.listsize = LIST\_INIT\_SIZE;

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

return OK;

}

}

status ListEmpty(SqList L)

// 如果线性表L存在，判断线性表L是否为空，空就返回TRUE，否则返回FALSE；如果线性表L不存在，返回INFEASIBLE。

{

if(!L.elem)

{

return INFEASIBLE;

}

else

{

if(L.length == 0)

{

return TRUE;

}

else

{

return FALSE;

}

}

}

int ListLength(SqList L)

// 如果线性表L存在，返回线性表L的长度，否则返回INFEASIBLE。

{

if(!L.elem)

{

return INFEASIBLE;

}

else

{

return L.length;

}

}

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

// 如果线性表L存在，获取线性表L的第i个元素，保存在e中，返回OK；如果i不合法，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

{

if(!L.elem)

{

return INFEASIBLE;

}

else

{

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

{

return ERROR;

}

else

{

e = L.elem[i - 1];

return OK;

}

}

}

status LocateElem(SqList L,ElemType e)

// 如果线性表L存在，查找元素e在线性表L中的位置序号并返回OK；如果e不存在，返回ERROR；当线性表L不存在时，返回INFEASIBLE。

{

if(!L.elem)

{

return INFEASIBLE;

}

else

{

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

{

if(L.elem[i] == e)

{

return i + 1;

}

}

return ERROR;

}

}

status PriorElem(SqList L,ElemType e,ElemType &pre)

// 如果线性表L存在，获取线性表L中元素e的前驱，保存在pre中，返回OK；如果没有前驱，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

{

if(!L.elem)

{

return INFEASIBLE;

}

else

{

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

{

if(L.elem[i] == e)

{

if(i)

{

pre = L.elem[i - 1];

return OK;

}

else

{

return ERROR;

}

}

}

return ERROR;

}

}

status NextElem(SqList L,ElemType e,ElemType &next)

// 如果线性表L存在，获取线性表L元素e的后继，保存在next中，返回OK；如果没有后继，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

{

if(!L.elem)

{

return INFEASIBLE;

}

else

{

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

{

if(L.elem[i] == e)

{

if(i == L.length - 1)

{

return ERROR;

}

else

{

next = L.elem[i + 1];

return OK;

}

}

}

return ERROR;

}

}

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

// 如果线性表L存在，将元素e插入到线性表L的第i个元素之前，返回OK；当插入位置不正确时，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

{

if(!L.elem)

{

return INFEASIBLE;

}

else

{

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

{

return ERROR;

}

else

{

if(L.listsize < L.length + 1)

{

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

if(L.elem)

{

L.listsize += LISTINCREMENT;

}

else

{

return OVERFLOW;

}

}

for(int j = L.length; j >= i; j--)

{

L.elem[j] = L.elem[j - 1];

}

L.elem[i - 1] = e;

L.length++;

return OK;

}

}

}

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

// 如果线性表L存在，删除线性表L的第i个元素，并保存在e中，返回OK；当删除位置不正确时，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

{

if(!L.elem)

{

return INFEASIBLE;

}

else

{

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

{

return ERROR;

}

else

{

e = L.elem[i - 1];

for(int j = i; j < L.length; j++)

{

L.elem[j - 1] = L.elem[j];

}

L.length--;

return OK;

}

}

}

status ListTrabverse(SqList L)

// 如果线性表L存在，依次显示线性表中的元素，每个元素间空一格，返回OK；如果线性表L不存在，返回INFEASIBLE。

{

if(!L.elem)

{

return INFEASIBLE;

}

else

{

printf("当前线性表中数据如下:\n");

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

{

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

if(i != L.length - 1)

{

printf(" ");

}

else

{

printf("\n");

}

}

return OK;

}

}

status SaveList(SqList L,char FileName[])

// 如果线性表L存在，将线性表L的的元素写到FileName文件中，返回OK，否则返回INFEASIBLE。

{

if(!L.elem)

{

return INFEASIBLE;

}

else

{

FILE \*f;

f = fopen(FileName, "w+");

if(f)

{

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

{

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

}

fclose(f);

return OK;

}

else

{

return ERROR;

}

}

}

status LoadList(SqList &L,char FileName[])

// 如果线性表L为空，将FileName文件中的数据读入到线性表L中，返回OK，不存在返回INFEASIBLE，文件打开失败返回FAIL，已有数据返回ERROR。

{

if(L.elem == NULL)

{

return INFEASIBLE;

}

else if(L.length == 0)

{

FILE \*fp;

fp = fopen(FileName, "r");

if(fp)

{

L.length = 0;

L.listsize = LIST\_INIT\_SIZE;

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

for(; fscanf(fp, "%d", L.elem + L.length) != EOF; L.length++);

fclose(fp);

return OK;

}

else

{

return FAIL;

}

}

else

{

return ERROR;

}

}

status InitLists(LISTS &Lists)

//线性表集合初始化

{

if(Lists.elem[0].L.elem != NULL)

{

return INFEASIBLE;

}

else

{

Lists.length = 0;

Lists.listsize = 10;

InitList(Lists.elem[0].L);

return OK;

}

}

status AddList(LISTS &Lists, char ListName[])

// 在Lists中增加一个名称为ListName的空线性表后输入线性表数据

{

if(Lists.length == Lists.listsize) //集合溢出判断

{

return OVERFLOW;

}

Lists.elem[Lists.length].L.elem = (ElemType\*)malloc(LIST\_INIT\_SIZE \* sizeof(ElemType));

Lists.elem[Lists.length].L.length = 0;

Lists.elem[Lists.length].L.listsize = LIST\_INIT\_SIZE;

strcpy(Lists.elem[Lists.length].name ,ListName);

InitInput(Lists.elem[Lists.length].L);

Lists.length++;

return OK;

}

status RemoveList(LISTS &Lists,char ListName[])

// Lists中删除一个名称为ListName的线性表

{

for(int i = 0; i < Lists.length; i++)

{

if(!strcmp(ListName,Lists.elem[i].name))

{

for(; i < Lists.length - 1; i++)

{

Lists.elem[i] = Lists.elem[i + 1];

}

Lists.length--;

return OK;

}

}

return 0;

}

int LocateList(LISTS Lists,char ListName[])

// 在Lists中查找一个名称为ListName的线性表，成功返回逻辑序号，否则返回0

{

for(int i = 0; i < Lists.length; i++)

{

if(!strcmp(ListName, Lists.elem[i].name))

{

return i + 1;

}

}

return 0;

}

status ListsTraverse(LISTS Lists)

//遍历线性表集合，如果集合为空返回0，否则将集合中线性表的全部数据输出，并返回OK

{

if(Lists.elem[0].L.elem == NULL)

{

return INFEASIBLE;

}

if(Lists.length == 0)

{

return 0;

}

printf("当前线性表集合中共包含以下%d个线性表：\n", Lists.length);

for(int i = 0; i < Lists.length; i++)

{

printf("%s\n",Lists.elem[i].name);

if(Lists.elem[i].L.length == 0)

{

printf("空\n");

}

for(int j = 0; j < Lists.elem[i].L.length; j++)

{

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

if(j != Lists.elem[i].L.length - 1)

{

printf(" ");

}

else

{

printf("\n");

}

}

}

return OK;

}

status SaveLists(LISTS Lists, char \*FileName)

//将集合中数据存入文件 FileName 中，若线性表集合为空或不存在时，返回INFEASIBLE,否则返回OK

{

if(Lists.elem == NULL || Lists.length == 0)

{

return INFEASIBLE;

}

FILE \*fp = fopen(FileName,"w+");

char ch = '\0';

for(int i = 0; i < Lists.length; i++)

{

fprintf(fp, "%s", Lists.elem[i].name);

fprintf(fp,"%c",'\n');

for(int j = 0; j < Lists.elem[i].L.length; j++)

{

fprintf(fp, "%d", Lists.elem[i].L.elem[j]);

if(j == Lists.elem[i].L.length - 1)

{

fprintf(fp, "%c", '\n');

}

else

{

fprintf(fp, "%c", ' ');

}

}

}

fclose(fp);

return OK;

}

SqList \*gather\_operate(LISTS &Lists)

//线性表集合操作

{

int op = 1; int j = 0;

char ListName[30];

SqList L\_ope;

L\_ope.elem = NULL;

while(op)

{

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

printf(" 线性表集合操作菜单\n");

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

printf(" 1. 初始化 5. 修改\n");

printf(" 2. 新增 6. 遍历\n");

printf(" 3. 删除 7. 存档(读档未实现)\n");

printf(" 4. 查找 0. 退出\n");

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

printf(" 请选择你的操作[0~7]:");

scanf("%d", &op);

switch(op)

{

case 1:

if(InitLists(Lists) == INFEASIBLE) printf("当前线性表集合已被初始化！\n");

else printf("初始化成功！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 2:

if(Lists.elem[0].L.elem == NULL)

{

printf("当前线性表集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

char ListName[30];

printf("请输入新增线性表的名称：\n");

scanf("%s",ListName);

if(AddList(Lists, ListName) == OVERFLOW) printf("线性表集合已满，添加失败！\n");

else printf("线性表\"%s\"存储成功！\n", ListName);

printf("----任意键继续----\n");

\_getch();

break;

case 3:

if(Lists.elem[0].L.elem == NULL)

{

printf("当前线性表集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("请输入所要删除的线性表名称：\n");

scanf("%s",&ListName);

if(RemoveList(Lists, ListName))

{

printf("线性表%s删除成功！\n",ListName);

}

else

{

printf("查无此表！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 4:

if(Lists.elem[0].L.elem == NULL)

{

printf("当前线性表集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("请输入所要查找的线性表名称：\n");

scanf("%s",&ListName);

if(j = LocateList(Lists, ListName))

{

printf("线性表%s位于集合中第%d位\n", ListName, j);

}

else

{

printf("查无此表！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 5:

if(Lists.elem[0].L.elem == NULL)

{

printf("当前线性表集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("请输入所需更改的表名:\n");

scanf("%s",ListName);

j = LocateList(Lists, ListName);

if(j == 0)

{

printf("查无此表！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

else

{

return &(Lists.elem[j - 1].L);

}

case 6:

if((j = ListsTraverse(Lists)) == 0) printf("当前线性表集合为空！\n");

else if(j == INFEASIBLE) printf("当前线性表集合不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 7:

if(Lists.elem[0].L.elem == NULL)

{

printf("当前线性表集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

if(SaveLists(Lists, SaveFile1) == INFEASIBLE) printf("当前线性表中无数据！\n");

else printf("线性表集合中数据已被存入%s中！\n", SaveFile1);

printf("----任意键继续----\n");

\_getch();

break;

case 0:

break;

default:

printf("指令错误！\n");

printf("----任意键继续----\n");

\_getch();

}

}

return &L\_ope;

}附录B 基于链式存储结构线性表实现的源程序

/\* Linear Table On Linked Structure \*/

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <string.h>

#include <conio.h>

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

#define FAIL -3

typedef int status;

typedef int ElemType; //数据元素类型定义

typedef struct LNode{ //单链表（链式结构）结点的定义

ElemType data;

struct LNode \*next;

}LNode,\*LinkList;

typedef struct LISTS{

char name[30];

LinkList L = NULL;

LISTS \*next;

} LISTS, \*pLISTS;

status InitList(LinkList& L);// 线性表L不存在，构造一个空的线性表，返回OK，否则返回INFEASIBLE。

status InitInput(LinkList L);

status DestroyList(LinkList &L);

status ClearList(LinkList L);

status ListEmpty(LinkList L);

int ListLength(LinkList L);

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

status LocateElem(LinkList L,ElemType e); //简化过

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

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

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

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

status ListTraverse(LinkList L); //简化过

status SaveList(LinkList L,char FileName[]);

status LoadList(LinkList L,char FileName[]);

status InitLists(pLISTS &Lists);

int ListsLength(pLISTS Lists); //求线性表集合的长度

status AddList(pLISTS Lists, int i, char ListName[]);

status RemoveList(pLISTS Lists,char ListName[]);

int LocateList(pLISTS Lists,char ListName[]); //在当前集合中查找名为 ListsName 的线性表

status ListsTraverse(pLISTS Lists); //遍历当前集合

status SaveLists(pLISTS Lists, char \*FileName); //线性表集合存档

LinkList gather\_operate(pLISTS &Lists);

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

int main(){

LinkList L = NULL; int op=1;

int j = 0, flag = 0;

LinkList L\_ope = NULL, temp = NULL;

pLISTS Lists = NULL;

while(op){

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

printf(" 基于链式存储的线性表操作菜单\n");

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

printf(" 1. 初始化线性表 7. 由位置求元素 13. 遍历线性表\n");

printf(" 2. 初始化输入 8. 查找元素 14. 保存线性表\n");

printf(" 3. 摧毁线性表 9. 前驱元素 15. 读取线性表\n");

printf(" 4. 清空线性表 10. 后继元素 16. 线性表集合操作\n");

printf(" 5. 线性表判空 11. 插入元素 0. 退出\n");

printf(" 6. 线性表表长 12. 删除元素\n");

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

if(op)

printf(" 请选择你的操作[0~16]:");

scanf("%d",&op);

switch(op){

case 1:

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

else printf("线性表创建失败！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 2:

j = InitInput(L);

if(j == INFEASIBLE) printf("当前线性表不存在，请先初始化！\n");

else if(j == ERROR) printf("当前线性表非空，不可进行初始化输入！\n");

else printf("初始化输入成功！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 3:

j = DestroyList(L);

if(j == INFEASIBLE) printf("当前线性表不存在！\n");

else printf("销毁成功！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 4:

j = ClearList(L);

if(j == INFEASIBLE) printf("当前线性表不存在！\n");

else printf("清空成功！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 5:

j = ListEmpty(L);

if(j == INFEASIBLE) printf("当前线性表不存在!\n");

else if(j) printf("当前线性表为空\n");

else printf("当前线性表不为空\n");

printf("----任意键继续----\n");

\_getch();

break;

case 6:

j = ListLength(L);

if(j == INFEASIBLE) printf("当前线性表不存在！\n");

else printf("当前线性表长度为%d\n",j);

printf("----任意键继续----\n");

\_getch();

break;

case 7:

if(L == NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

int i, e;

printf("查找第几位元素？\n");

scanf("%d",&i);

j = GetElem(L, i, e);

if(j == ERROR) printf("查找位置不合法！\n");

else printf("当前线性表中第%d个元素为%d\n", i, e);

printf("----任意键继续----\n");

\_getch();

break;

case 8:

if(L == NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("输入所需查找的元素内容\n");

scanf("%d", &e);

j = LocateElem(L, e);

if(j == ERROR) printf("表中不存在元素 %d\n",e);

else printf("元素%d位于线性表中第%d个位置\n", e, j);

printf("----任意键继续----\n");

\_getch();

break;

case 9:

if(L == NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

int prior;

printf("输入当前元素\n");

scanf("%d", &e);

j = PriorElem(L, e, prior);

if(j == ERROR && LocateElem(L, e)) printf("此元素无前驱！\n");

else if(j != ERROR) printf("当前元素前驱为%d\n", prior);

else printf("此元素不在当前线性表中！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 10:

if(L == NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

int next;

printf("输入当前元素\n");

scanf("%d", &e);

j = NextElem(L, e, next);

if(j == ERROR && LocateElem(L, e)) printf("此元素无后继！\n");

else if(j != ERROR) printf("当前元素后继为%d\n", next);

else printf("此元素不在当前线性表中！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 11:

if(L == NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("输入加入元素\n");

scanf("%d", &e);

printf("想插入第几个元素前？\n");

scanf("%d", &i);

j = ListInsert(L, i, e);

if(j == ERROR) printf("插入位置非法！\n");

else

{

printf("插入成功！\n");

printf("当前线性表数据如下：\n");

ListTraverse(L);

}

printf("----任意键继续----\n");

\_getch();

break;

case 12:

if(L == NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("删除第几个元素？\n");

scanf("%d",&i);

j = ListDelete(L, i, e);

if(j == ERROR) printf("删除位置非法！\n");

else

{

printf("删除成功！\n");

if(ListEmpty(L)) printf("当前线性表已经被你删空了~\n");

else

{

printf("当前线性表中数据如下：\n");

ListTraverse(L);

}

}

printf("----任意键继续----\n");

\_getch();

break;

case 13:

ListTraverse(L);

printf("----任意键继续----\n");

\_getch();

break;

case 14:

if(L == NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

char SaveFile[40];

printf("输入存档文件路径(输入0选择默认路径):\n");

scanf("%s",SaveFile);

if(!strcmp(SaveFile,"0")) strcpy(SaveFile, "D:\\LinkList\_save.dat");

if(SaveList(L, SaveFile) == INFEASIBLE) printf("当前线性表不存在！\n");

else printf("已将数据存入 %s\n",SaveFile);

printf("----任意键继续----\n");

\_getch();

break;

case 15:

if(L == NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

else if(L->next != NULL)

{

printf("当前线性表非空！读取会出大问题！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

char LoadFile[40];

printf("输入读档文件路径(输入0选择默认路径):\n");

scanf("%s",LoadFile);

if(!strcmp(LoadFile,"0")) strcpy(LoadFile, "D:\\LinkList\_save.dat");

if((j = LoadList(L, LoadFile)) == ERROR) printf("当前线性表中已有数据！\n");

else if(j == OK) printf("数据已从 %s 中读出\n",LoadFile);

else if(j == INFEASIBLE) printf("当前线性表不存在！请先初始化！\n");

else if(j == FAIL) printf("存档文件建立失败！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 16:

if(flag == 1) //修改状态，不可再次选择线性表集合操作

{

printf("当前有未结束的线性表修改，请先结束前一修改！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

L\_ope = gather\_operate(Lists);

if(L\_ope != NULL)//只要选择了集合操作中的修改，就将主界面线性表置换成集合中所选线性表

{

temp = L; //temp暂存主界面线性表数据

L = L\_ope; //主界面后台线性表置换为集合内所选线性表

flag = 1; //系统状态置为集合操作态

}

break;

case 0:

break;

default:

printf("操作指令有误！\n");

printf("----任意键继续----\n");

\_getch();

break;

}//end of switch

if(flag && op == 0) //flag判断当前是否在线性表集合操作中，此时退出返回主界面

{

L = temp; //返还原始主界面线性表数据

op = 1; //保持不退出while

flag = 0; //状态置为初始界面

printf("原线性表数据已载入！\n");

\_getch();

}

}//end of while

printf("欢迎下次再使用本系统！\n");

printf("----任意键结束----\n");

\_getch();

return 0;

}//end of main()

status InitList(LinkList &L)

// 线性表L不存在，构造一个空的线性表，返回OK，否则返回INFEASIBLE。

{

if(L != NULL)

{

return INFEASIBLE;

}

else

{

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

L->next = NULL;

return OK;

}

}

status InitInput(LinkList L)

// 线性表不存在，返回INFEASIBLE；如果线性表为空表，输入数据；如果线性表不为空，返回ERROR

{

if(L == NULL)

{

return INFEASIBLE;

}

else if(L->next != NULL)

{

return ERROR;

}

else

{

ElemType temp = 0;

printf("请输入一组数据(0结束):\n");

scanf("%d",&temp);

while(temp)

{

L->next = (LinkList)malloc(sizeof(LNode));

L = L->next;

L->data = temp;

scanf("%d",&temp);

}

L->next = NULL;

}

}

status DestroyList(LinkList& L)

// 如果线性表L存在，销毁线性表L，释放数据元素的空间，返回OK，否则返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

else

{

LinkList p = NULL;

while(L != NULL)

{

p = L->next;

free(L);

L = p;

}

return OK;

}

}

status ClearList(LinkList L)

// 如果线性表L存在，删除线性表L中的所有元素，返回OK，否则返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

else

{

LinkList p = L->next, loc = L;

while(p != NULL)

{

L = p->next;

free(p);

p = L;

}

L = loc;

L->next = NULL;

return OK;

}

}

status ListEmpty(LinkList L)

// 如果线性表L存在，判断线性表L是否为空，空就返回TRUE，否则返回FALSE；如果线性表L不存在，返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

else if(L->next == NULL)

{

return TRUE;

}

else

{

return FALSE;

}

}

int ListLength(LinkList L)

// 如果线性表L存在，返回线性表L的长度，否则返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

int count = 0;

while(L->next != NULL)

{

L = L->next;

count++;

}

return count;

}

status GetElem(LinkList L,int i,ElemType &e)

// 如果线性表L存在，获取线性表L的第i个元素，保存在e中，返回OK；如果i不合法，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

{

if(L == NULL) //线性表不存在时

{

return INFEASIBLE;

}

else if(i < 1) //查找位置非法，返回ERROR

{

return ERROR;

}

else

{

int count = 0;

while(L->next != NULL) //L不为倒数第一个时

{

L = L->next;

count++;

if(count == i) //当前位置是否为所找位置

{

e = L->data;

return OK;

}

}

return ERROR; //查找位置非法，超出表长

}

}

status LocateElem(LinkList L,ElemType e)

// 如果线性表L存在，查找元素e在线性表L中的位置序号；如果e不存在，返回ERROR；当线性表L不存在时，返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

else

{

int count = 0;

while(L->next != NULL)

{

L = L->next;

count++;

if(L->data == e)

{

return count;

}

}

return ERROR;

}

}

status PriorElem(LinkList L,ElemType e,ElemType &pre)

// 如果线性表L存在，获取线性表L中元素e的前驱，保存在pre中，返回OK；如果没有前驱，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

else

{

L = L->next;

while(L != NULL && L->next != NULL) //当L是最后一个(表明倒数第二个的next里的data不等于e)时返回ERROR

{

if(L->next->data == e)

{

pre = L->data;

return OK;

}

L = L->next;

}

return ERROR;

}

}

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

// 如果线性表L存在，获取线性表L元素e的后继，保存在next中，返回OK；如果没有后继，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

else

{

L = L->next;

while(L->next != NULL)

{

if(L->data == cur)

{

next\_e = L->next->data;

return OK;

}

L = L->next;

}

return ERROR;

}

}

status ListInsert(LinkList L,int i,ElemType e)

// 如果线性表L存在，将元素e插入到线性表L的第i个元素之前，返回OK；当插入位置不正确时，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

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

{

return ERROR;

}

else

{

int count = 0;

//新建存储节点

LinkList new\_p = (LinkList)malloc(sizeof(LNode));

new\_p->next = NULL;

new\_p->data = e;

//查找插入位置

while(L != NULL)

{

if(count == i - 1) //查找位置的前一个

{

new\_p->next = L->next;

L->next = new\_p;

return OK;

}

L = L->next;

count++;

}

}

}

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

// 如果线性表L存在，删除线性表L的第i个元素，并保存在e中，返回OK；当删除位置不正确时，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

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

{

return ERROR;

}

else

{

int count = 0;

LinkList p = NULL;

while(L != NULL)

{

if(count == i - 1)

{

p = L->next;

e = L->next->data;

L->next = L->next->next;

free(p);

return OK;

}

L = L->next;

count++;

}

}

}

status ListTraverse(LinkList L)

// 如果线性表L存在，依次显示线性表中的元素，每个元素间空一格，返回OK；如果线性表L不存在，返回INFEASIBLE。

{

if(L == NULL) //线性表不存在时

{

printf("当前线性表不存在！请先初始化！\n");

return INFEASIBLE;

}

else if(L->next == NULL)

{

printf("别看了别看了，里边啥也没有~\n");

return 0;

}

else

{

L = L->next;

while(L != NULL)

{

printf("%d",L->data);

if(L->next != NULL)

{

printf(" ");

}

else

{

printf("\n");

}

L = L->next;

}

return OK;

}

}

status SaveList(LinkList L,char FileName[])

//线性表存档，线性表存在时将线性表中数据存入文件并返回OK，否则返回INFEASIBLE

{

if(L == NULL)

{

return INFEASIBLE;

}

else

{

FILE \*fp;

fp = fopen(FileName,"w+");

L = L->next;

while(L != NULL)

{

fprintf(fp,"%d ",L->data);

L = L->next;

}

fclose(fp);

return OK;

}

}

status LoadList(LinkList L,char FileName[])

//线性表读档，如果线性表不存在，返回INFEASIBLE，线性表为空时读入数据，非空时返回ERROR

{

if(L == NULL) //线性表不存在时需要先初始化才可读档

{

return INFEASIBLE;

}

else if(L->next != NULL) //线性表中已有数据时不可读档

{

return ERROR;

}

else

{

FILE \*fp = fopen(FileName,"r");

ElemType in;

LinkList track = NULL;//跟在 L后面

L->next = (LinkList)malloc(sizeof(LNode));

L = L->next;

while(fscanf(fp,"%d",&in) != EOF)

{

L->data = in;

L->next = (LinkList)malloc(sizeof(LNode));

track = L;

L = L->next;

}

track->next = NULL;

fclose(fp);

return OK;

}

}

status InitLists(pLISTS &Lists)

//初始化线性表集合

{

if(Lists == NULL)

{

Lists = (pLISTS)malloc(sizeof(LISTS));

Lists->L = NULL;

InitList(Lists->L);

Lists->next = NULL;

return OK;

}

else

{

return INFEASIBLE;

}

}

int ListsLength(pLISTS Lists)

{

if(Lists == NULL)

{

return INFEASIBLE;

}

int count = 0;

Lists = Lists->next;

while(Lists != NULL)

{

Lists = Lists->next;

count++;

}

return count;

}

status AddList(pLISTS Lists, int i, char ListName[])

//在线性表第i位插入一个名为ListName的线性表

{

if(Lists == NULL)

{

return INFEASIBLE;

}

if(i < 1)

{

printf("插入位置非法！\n");

return ERROR;

}

pLISTS new\_List = NULL;

InitLists(new\_List);

strcpy(new\_List->name, ListName);

int count = 0; //记录当前元素位置

pLISTS p = NULL;

while(Lists != NULL)

{

if(count == i-1)

{

new\_List->next = Lists->next;

Lists->next = new\_List;

printf("线性表%s已成功插入集合中第%d位！\n", ListName, i);

return OK;

}

Lists = Lists->next;

count++;

}

printf("插入位置非法！\n");

return ERROR;

}

status RemoveList(pLISTS Lists,char ListName[])

{

if(Lists == NULL)

{

return INFEASIBLE;

}

while(Lists->next != NULL)

{

if(!strcmp(Lists->next->name, ListName))

{

pLISTS p = Lists->next;

Lists->next = Lists->next->next;

DestroyList(p->L);

free(p);

printf("线性表%s已成功删除！\n",ListName);

return OK;

}

Lists = Lists->next;

}

printf("查无此表！\n");

return ERROR;

}

int LocateList(pLISTS Lists,char ListName[])

{

if(Lists == NULL)

{

return INFEASIBLE;

}

int count = 1;

Lists = Lists->next;

while(Lists != NULL)

{

if(!strcmp(Lists->name, ListName))

{

return count;

}

Lists = Lists->next;

count++;

}

}

status ListsTraverse(pLISTS Lists)

//遍历当前集合

{

if(Lists == NULL)

{

printf("当前线性表不存在！请先初始化！\n");

return INFEASIBLE;

}

if(ListsLength(Lists) == 0)

{

printf("阿巴阿巴...这个线性表集合里啥也没有啊~\n");

return 0;

}

printf("当前线性表集合中数据如下：\n");

Lists = Lists->next;

while(Lists != NULL)

{

printf("%s\n",Lists->name);

ListTraverse(Lists->L);

Lists = Lists->next;

}

return OK;

}

status SaveLists(pLISTS Lists, char \*FileName)

{

if(Lists == NULL)

{

return INFEASIBLE;

}

if(Lists->next == NULL)

{

return 0;

}

FILE \*fp = fopen(FileName,"w+");

Lists = Lists->next;

while(Lists != NULL)

{

fprintf(fp,"%s\n",Lists->name);

LinkList p = Lists->L->next;

int len = ListLength(Lists->L), count = 1;

while(p != NULL)

{

fprintf(fp, "%d", p->data);

if(count == len)

{

fprintf(fp, "%c", '\n');

}

else

{

fprintf(fp, "%c", ' ');

}

p = p->next;

count++;

}

Lists = Lists->next;

}

fclose(fp);

return OK;

}

LinkList gather\_operate(pLISTS &Lists)

//线性表集合操作

{

int op = 1, j = 0, i = 0;

char ListName[30];

LinkList L\_ope = NULL;

while(op)

{

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

printf(" 线性表集合操作菜单\n");

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

printf(" 1. 初始化 5. 修改\n");

printf(" 2. 插入 6. 遍历\n");

printf(" 3. 删除 7. 存档(读档未实现)\n");

printf(" 4. 查找 0. 退出\n");

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

printf(" 请选择你的操作[0~7]:");

scanf("%d", &op);

switch(op)

{

case 1:

if(InitLists(Lists) == INFEASIBLE) printf("当前线性表集合已被初始化！\n");

else printf("初始化成功！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 2:

if(Lists == NULL)

{

printf("当前线性表集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

char ListName[30];

printf("请输入新增线性表的名称：\n");

scanf("%s",ListName);

printf("想插入第几位线性表之前？\n");

scanf("%d",&i);

AddList(Lists, i, ListName);

printf("----任意键继续----\n");

\_getch();

break;

case 3:

if(Lists == NULL)

{

printf("当前线性表集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

if(ListsLength(Lists) == 0)

{

printf("别删了~里面已经啥也没有了~\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("请输入所要删除的线性表名称：\n");

scanf("%s",ListName);

RemoveList(Lists, ListName);

printf("----任意键继续----\n");

\_getch();

break;

case 4:

if(Lists == NULL)

{

printf("当前线性表集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("请输入所要查找的线性表名称：\n");

scanf("%s",ListName);

if(j = LocateList(Lists, ListName)) printf("线性表%s位于集合中第%d位\n", ListName, j);

else printf("查无此表！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 5:

if(Lists == NULL)

{

printf("当前线性表集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("请输入所需更改的表名:\n");

scanf("%s",ListName);

j = LocateList(Lists, ListName);

if(j == 0)

{

printf("查无此表！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

else

{

pLISTS p = Lists;

for(int i = 0; i < j; i++) p = p->next;

return p->L;

}

case 6:

ListsTraverse(Lists);

printf("----任意键继续----\n");

\_getch();

break;

case 7:

if(Lists == NULL)

{

printf("当前线性表集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

char FileName[40];

printf("输入存档文件路径(输入0选择默认路径):\n");

scanf("%s",FileName);

if(!strcmp(FileName,"0")) strcpy(FileName, "D:\\LISTS\_save.dat");

if(SaveLists(Lists, FileName) == INFEASIBLE) printf("当前线性表中无数据！\n");

else printf("线性表集合中数据已被存入%s中！\n", FileName);

printf("----任意键继续----\n");

\_getch();

break;

case 0:

break;

default:

printf("指令错误！\n");

printf("----任意键继续----\n");

\_getch();

}

}

return L\_ope;

}

/\*

## def.h

#include "stdio.h"

#include "stdlib.h"

#include "string.h"

#include "ctype.h"

#include <conio.h>

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

#define LEFT 4

#define RIGHT 6

#define max(a,b) ((a>b)?a:b)

//用于按层遍历中definition扩容

#define INIT\_DEF\_SIZE 64

#define INCREMENT 16

//用于森林扩容

#define INIT\_SIZE 100

typedef int status;

typedef int KeyType;

typedef struct {

KeyType key;

char others[20];

} TElemType; //二叉树结点类型定义

typedef struct BiTNode{ //二叉链表结点的定义

TElemType data;

struct BiTNode \*lchild,\*rchild;

} BiTNode, \*BiTree;

typedef struct {

int pos;

TElemType data;

} DEF;

typedef struct { //二叉树的结点definition，包含pos以及pos对应的结点地址，仅用于按层遍历

int pos;

BiTNode \*node;

} def;

typedef struct { //森林中每棵树的定义

BiTree T;

char name[30];

} Tree, \*pTree;

typedef struct { //森林定义(线性表结构存储)

pTree \*tree;

int length;

int size;

} FOREST, \*FST;

## Stack.h

//设计简易栈

#include"def.h"

#define INIT\_STACK\_SIZE 20

#define INCREMENT 16

typedef struct{

BiTree \*elem = NULL;

int top = 0;

int stacksize = 0;

} Stack;

status InitStack(Stack &s); //栈的初始化

status Push(Stack &s, BiTNode \*p); //p进栈

status Pop(Stack &s, BiTree &p); //出栈

status StackEmpty(Stack s); //栈判空

status InitStack(Stack &s)

//栈的初始化，s中有内容时返回INFEASIBLE表示初始化失败，否则返回OK表示初始化成功

{

if(s.elem != NULL) return INFEASIBLE;

s.stacksize = INIT\_STACK\_SIZE;

s.elem = (BiTree \*)calloc(s.stacksize, sizeof(BiTree)); //数组中全为地址，避免野指针

s.top = 0;

return OK;

}

status Push(Stack &s, BiTNode \*p)

//p进栈，栈不存在返回INFEASIBLE，栈满时扩容，成功之后返回OK

{

if(s.elem == NULL) return INFEASIBLE;

s.elem[s.top++] = p;

if(s.top >= s.stacksize) //栈满，realloc扩容

{

s.stacksize += INCREMENT;

s.elem = (BiTree \*)realloc(s.elem, s.stacksize \* sizeof(BiTree));

}

return OK;

}

status Pop(Stack &s, BiTree &p)

//出栈，栈不存在返回INFEASIBLE，栈空返回ERROR，否则返回OK

{

if(s.elem == NULL) return INFEASIBLE;

if(s.top == 0) return ERROR;

p = s.elem[--s.top];

s.elem[s.top + 1] = NULL; //保存的是地址值，不允许残留

return OK;

}

status StackEmpty(Stack s)

//栈判空，栈不存在返回INFEASIBLE，栈空返回TRUE，否则返回FALSE

{

if(s.elem == NULL) return INFEASIBLE;

if(s.top == 0) return TRUE;

return FALSE;

}

## fun.h

#include "Stack.h"

//按层遍历全局变量不易处理，提出

static int i = 0, size = INIT\_DEF\_SIZE;

static def \*definition = (def\*)malloc(size \* sizeof(def));

static int position = 1;

status InitBiTree(BiTree &T);

status CreateBiTree(BiTree &T,DEF definition[]);

status CreateBiTree(BiTree &T,TElemType definition[]);

/\*根据先序遍历序列definition创建二叉树 T，definition中如有重复则返回ERROR，否则返回OK\*/

status ClearBiTree(BiTree &T); //将二叉树设置成空，并删除所有结点，释放结点空间

bool IsRepeat(TElemType definition[]); //判断先序序列definition中是否有重复key值，用于创建树

int BiTreeDepth(BiTree T); //求二叉树T的深度

BiTNode\* LocateNode(BiTree T,KeyType e); //查找结点

status Assign(BiTree &T,KeyType e,TElemType value); //实现结点赋值。

BiTNode\* GetSibling(BiTree T,KeyType e); //实现获得兄弟结点

status InsertNode(BiTree &T,KeyType e,int LR,TElemType c); //实现插入节点，在e的LR(左右)孩子处插入c

status ParentNode(BiTree T, KeyType e, BiTree &p);

/\*寻找key值为e元素的双亲结点，由参数p带出函数，返回OK表示删除结点为根节点，

返回LEFT表示e为p的左孩子，返回RIGHT表示e为p的右孩子，返回ERROR表示没有此结点\*/

BiTNode\* RightMostNode(BiTree T); //寻找 T 最右端结点，用于删除

status DeleteNode(BiTree &T,KeyType e); //删除结点。

status PreOrderTraverse(BiTree T,void (\*visit)(BiTree)); //先序遍历二叉树T(递归)

status InOrderTraverse(BiTree T,void (\*visit)(BiTree)); //先序遍历二叉树T(非递归)

status PostOrderTraverse(BiTree T,void (\*visit)(BiTree)); //后序遍历二叉树T(递归)

def\* CreateDef(BiTree T, int &length); //按照结点位置构造二叉树的definition

void DefOrder(def \*definition, int length); //按结点位置对definition排序

status LevelOrderTraverse(BiTree T,void (\*visit)(BiTree)); //按层遍历二叉树T

status SaveBiTree(BiTree T, char FileName[]); //存档

status LoadBiTree(BiTree &T, char FileName[]); //读档

status TreeLikeTraverse(BiTree T, void (\*visit)(BiTree)); //树形遍历，输出树的图形表示

void visit(BiTree T);

BiTree\* gather\_operate(FST& fst);

//森林操作

status InitForest(FST& fst)

//初始化森林(以线性表形式定义)

{

if(fst != NULL) return INFEASIBLE;

fst = (FST)malloc(sizeof(FOREST));

fst->tree = (pTree\*)malloc(INIT\_SIZE \* sizeof(pTree));

fst->length = 0;

fst->size = INIT\_SIZE;

return OK;

}

status AddBiTree(const FST fst, int i, char BiTreeName[])

//向森林中第i位种入名为BiTreeName的树

{

if(fst == NULL) return INFEASIBLE;

if(i > fst->length + 1 || i < 1) return ERROR;

if(fst->length >= fst->size) //扩容

{

fst->size += INCREMENT;

fst->tree = (pTree\*)realloc(fst->tree, fst->size \* sizeof(pTree));

}

for(int j = fst->length; j >= i; j--) //移位置

{

fst->tree[j] = fst->tree[j - 1];

}

fst->tree[i - 1] = (pTree)malloc(sizeof(Tree));

strcpy(fst->tree[i - 1]->name, BiTreeName); //树命名 && 树初始化

fst->tree[i - 1]->T = NULL;

InitBiTree(fst->tree[i - 1]->T);

fst->length++;

return OK;

}

status RemoveBiTree(const FST fst, char BiTreeName[])

//砍去森林中名为BiTreeName的树

{

if(fst == NULL) return INFEASIBLE;

for(int i = 0; i < fst->length; i++)

{

if(!strcmp(BiTreeName, fst->tree[i]->name))

{

ClearBiTree(fst->tree[i]->T);

free(fst->tree[i]);

for(int a = i + 1; a < fst->length; a++)

fst->tree[a - 1] = fst->tree[a];

fst->length--;

return OK;

}

}

return ERROR;

}

int LocateBiTree(const FST fst, char BiTreeName[])

//查找当前森林中名为BiTreeName的树

{

if(fst == NULL) return INFEASIBLE;

for(int i = 0; i < fst->length; i++)

{

if(!strcmp(BiTreeName, fst->tree[i]->name)) return i + 1;

}

return ERROR;

}

status ForestTraverse(const FST fst)

//遍历森林

{

if(fst == NULL) return INFEASIBLE;

if(fst->length == 0) return ERROR;

int op = 1;

status (\*p)(BiTree, void (\*)(BiTree)) = NULL;

while(op)

{

system("cls");

printf(" 遍历方式选择\n");

printf(" 1.前序遍历\n");

printf(" 2.中序遍历\n");

printf(" 3.后序遍历\n");

printf(" 4.按层遍历\n");

printf(" 5.树形遍历\n");

printf(" 请输入您所选择遍历方式的相应序号[1-5]:");

scanf("%d", &op);

switch(op)

{

case 1: p = PreOrderTraverse; break;

case 2: p = InOrderTraverse; break;

case 3: p = PostOrderTraverse; break;

case 4: p = LevelOrderTraverse; break;

case 5: p = TreeLikeTraverse; break;

default: printf("别闹~序号还能输错？再给你个机会，回去重输~\n"); op = 6; \_getch(); break;

}

if(op != 6) op = 0;

}

printf("当前树中数据如下所示：\n");

for(int i = 0; i < fst->length; i++)

{

printf("第%d棵树：\n %s:\n", i + 1, fst->tree[i]->name);

p(fst->tree[i]->T->lchild, visit);

printf("\n\n");

}

return OK;

}

status ClearForest(const FST fst)

{

if(fst == NULL) return INFEASIBLE;

for(int i = 0; i < fst->length; i++)

{

ClearBiTree(fst->tree[i]->T); //ClearBiTree函数：参数为T时为销毁，参数为T->lchild时清空

free(fst->tree[i]);

fst->tree[i] = NULL;

}

fst->length = 0;

return OK;

}

status DestroyForest(FST &fst)

{

if(fst == NULL) return INFEASIBLE;

ClearForest(fst);

fst = NULL;

return OK;

}

//森林操作

bool IsRepeat(TElemType definition[])

//判断先序序列definition中是否有重复key值，用于创建树

{

int a = 0, b = 0;

for(int i = 0; (a = definition[i].key) != -1; i++)

{

for(int j = i + 1; (b = (definition[j].key)) != -1; j++)

{

if(a == b && a != 0)

{

return true;

}

}

}

return false;

}

status InitBiTree(BiTree &T)

{

if(T != NULL) return INFEASIBLE;

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

T->lchild = NULL;

T->rchild = NULL;

return OK;

}

status CreateBiTree(BiTree &T,DEF definition[])

{

int i=0,j;

static BiTNode \*p[100];

while (j=definition[i].pos)

{

p[j]=(BiTNode \*)malloc(sizeof(BiTNode));

p[j]->data=definition[i].data;

p[j]->lchild=NULL;

p[j]->rchild=NULL;

if (j!=1)

if (j%2) p[j/2]->rchild=p[j];

else p[j/2]->lchild=p[j];

i++;

}

T=p[1];

return OK;

}

status CreateBiTree(BiTree &T,TElemType definition[])

//根据带空枝的二叉树先根遍历序列definition构造一棵二叉树，将根节点指针赋值给T并返回OK，

//如果有相同的关键字，返回ERROR。

{

static int i = 0, flag = 0;//i为递归变量，遍历definition；flag控制IsRepeat函数只被调用一次；over处理二叉树结束标志-1

if(flag == 0 && (flag = 1) && IsRepeat(definition)) //判断是否有重复，并由于IsRepeat只需运行一次，通过flag使其成为一次性。

{

return ERROR;

}

T = (BiTree)malloc(sizeof(BiTNode)); //先对当前结点分配空间

//printf("In CreateBiTree, definition[%d].key = %d\n", i, definition[i].key);

if(definition[i].key == 0) //对于 0,null，直接回溯，将分配的空间free，T置NULL，definition后移

{

free(T);

T = NULL;

i++;

if(definition[i].key == -1)

{

i = 0; //函数结束，全局变量应恢复原状

flag = 0;

}

return OK;

}

else //数据进树

{

T->data.key = definition[i].key;

strcpy(T->data.others,definition[i].others);

}

i++; //definition后移

CreateBiTree(T->lchild, definition);

CreateBiTree(T->rchild, definition);

}

status ClearBiTree(BiTree &T)

//将二叉树设置成空，并删除所有结点，释放结点空间

{

// 请在这里补充代码，完成本关任务

/\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/

if(T == NULL)

{

return OK;

}

ClearBiTree(T->lchild);

ClearBiTree(T->rchild);

if(T->lchild == NULL && T->rchild == NULL)

{

free(T);

T = NULL;

return OK;

}

/\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/

}

int BiTreeDepth(BiTree T)

//求二叉树T的深度

{

// 请在这里补充代码，完成本关任务

/\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/

if(T == NULL)

{

return 0;

}

int LeftLen = 1 + BiTreeDepth(T->lchild);

int RightLen = 1 + BiTreeDepth(T->rchild);

return max(LeftLen, RightLen);

}

BiTNode\* LocateNode(BiTree T,KeyType e)

//查找结点

{

static BiTNode \*p = NULL;

if(p != NULL && p->data.key != e) p = NULL;

//在别处多次调用(非递归)时，防止上一次使用遗留下来的p未被置NULL而导致之后使用故障

if(p != NULL) return p;

//p不为NULL时，表明已找到目标节点，直接回溯即可

if(T == NULL) return p;

//当前T为空时，直接回溯即可

if(e == T->data.key)//当找到目标节点，将目标节点地址传至p，返回即可

{

p = T;

return p;

}

LocateNode(T->lchild,e);

LocateNode(T->rchild,e);

}

status Assign(BiTree &T,KeyType e,TElemType value)

//实现结点赋值。

{

BiTNode \*p = LocateNode(T, value.key);

//p记录树中key值为value.key的结点地址用于保证赋值后的树中无key值重复项

BiTNode \*q = LocateNode(T, e);

//q记录结点编号为e的结点地址以查重以及赋值

if(!q || (p && q && p != q)) //所有错误状况

//q为空，代表树中无编号为e结点，不可赋值；p!=q且p,q均不为空，说明赋值后会有key值重复

{

return ERROR;

}

else

{

q->data.key = value.key;

strcpy(q->data.others, value.others);

return OK;

}

/\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/

}

BiTNode\* GetSibling(BiTree T,KeyType e)

//实现获得兄弟结点

{

BiTNode \*rec = NULL; //记录返回值

if(T == NULL) //当前树为空时，返回空值，表明找不到，直接回溯

{

return NULL;

}

if(T->lchild != NULL && T->lchild->data.key == e) return T->rchild; //当前结点左儿子为所找结点时，返回当前结点右儿子

else if(T->rchild != NULL && T->rchild->data.key == e) return T->lchild;//当前结点左儿子为所找结点时，返回当前结点右儿子

else //不满足以上两种，继续深入搜索

{

if(rec = GetSibling(T->lchild, e)) return rec; //如果左边找到了，返回其返回值

else if(rec = GetSibling(T->rchild, e)) return rec;//如果右边找到了，返回其返回值

else return NULL; //都没有，返回NULL

}

}

status InsertNode(BiTree &T,KeyType e,int LR,TElemType c)

//实现插入节点，在e的LR(左右)孩子处插入c

{

BiTNode \*aim = LocateNode(T, e); //查找e所在结点

BiTNode \*repeat = LocateNode(T, c.key); //判断即将插入的结点key值在原树中是否已存在

if(aim == NULL || repeat != NULL) return ERROR; //当未找到 key=e 的结点或者将插入的结点key值在原树中已存在

else

{

BiTNode \*NewNode = (BiTNode\*)malloc(sizeof(BiTNode));

NewNode->data.key = c.key;

strcpy(NewNode->data.others, c.others); //新建结点，存储新结点值

if(LR == 0) //左插

{

NewNode->rchild = aim->lchild;

NewNode->lchild = NULL;

aim->lchild = NewNode;

return OK;

}

else if(LR == 1) //右插

{

NewNode->rchild = aim->rchild;

NewNode->lchild = NULL;

aim->rchild = NewNode;

return OK;

}

else if(LR == -1) //头插

{

NewNode->rchild = T;

NewNode->lchild = NULL;

T = NewNode;

return OK;

}

else return ERROR;

}

}

status ParentNode(BiTree T, KeyType e, BiTree &p)

/\*寻找key值为e元素的双亲结点，由参数p带出函数，返回OK表示删除结点为根节点，

返回LEFT表示e为p的左孩子，返回RIGHT表示e为p的右孩子，返回ERROR表示没有此结点\*/

{

status state = ERROR;

if(T == NULL) return ERROR;

if(T->data.key == e) {p = T; return OK;}

else if(T->lchild && T->lchild->data.key == e) {p = T; return LEFT;}

else if(T->rchild && T->rchild->data.key == e) {p = T; return RIGHT;}

state = ParentNode(T->lchild, e, p);

if(state) return state;

return ParentNode(T->rchild, e, p);

}

BiTNode\* RightMostNode(BiTree T)

//寻找 T 最右端结点

{

while(T->rchild != NULL) T = T->rchild;

return T;

}

status DeleteNode(BiTree &T,KeyType e)

//删除结点。

{

BiTNode \*p = NULL, \*del = NULL;

status state = ParentNode(T, e, p);

if(state == OK) //根节点为所要删除的结点

{

if(p->lchild == NULL)//根结点没有左孩子

{

T = T->rchild;

free(p);

return OK;

}

else if(p->rchild == NULL)//根结点没有右孩子

{

T = T->lchild;

free(p);

return OK;

}

else //根结点度为二

{

T = T->lchild;

RightMostNode(T)->rchild = p->rchild;//根节点左孩子的最右置为根节点右孩子

free(p);

return OK;

}

}

else if(state == LEFT) //待删除结点为p的左孩子时

{

del = p->lchild; //记录待删除结点

if(del->lchild != NULL)//如果要删除的结点有左孩子

{

RightMostNode(del->lchild)->rchild = del->rchild; //要删除的结点的右孩子移至要删除的结点的左孩子的最右结点的右子树处

p->lchild = del->lchild;

}

else p->lchild = del->rchild;

free(del);

return OK;

}

else if(state == RIGHT) //待删除结点为p的右孩子时

{

del = p->rchild; //记录待删除结点

if(del->lchild != NULL) //待删除结点有左孩子

{

RightMostNode(del->lchild)->rchild = del->rchild;//要删除的结点的右孩子移至要删除的结点的左孩子的最右结点的右子树处

p->rchild = del->lchild;

}

else p->rchild = del->rchild;//待删除结点没有左孩子

free(del);

return OK;

}

else return ERROR; //未查找到待删除结点

}

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

//先序遍历二叉树T(递归)

{

if(T == NULL) return OK;

visit(T);

PreOrderTraverse(T->lchild, visit);

PreOrderTraverse(T->rchild, visit);

return OK;

}

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

//中序遍历二叉树T(非递归)

{

Stack s; BiTNode \*p = T;

InitStack(s);

while(p != NULL || StackEmpty(s) == FALSE)

{

if(p != NULL) //当前结点地址不为空

{

Push(s, p); //当前结点作为根节点进栈

p = p->lchild; //遍历左子树

}

else

{

Pop(s, p); //空结点双亲结点出栈

visit(p); //输出当前空结点双亲结点中的信息

p = p->rchild; //遍历右子树

} // end if...else...

} //end while

return OK;

}

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

//后序遍历二叉树T

{

if(T == NULL) return OK;

PostOrderTraverse(T->lchild, visit);

PostOrderTraverse(T->rchild, visit);

visit(T);

return OK;

}

def\* CreateDef(BiTree T, int &length) //全局变量处理完成！！！！！！！！！！！！！！！！！

//按照结点位置构造二叉树的definition

{

if(T == NULL) return NULL;

if(i >= size) //扩容

{

size += INCREMENT;

definition = (def\*)realloc(definition, size \* sizeof(def));

}

definition[i].pos = position;

definition[i].node = T;

i++;

length = i;

{

position \*= 2;

CreateDef(T->lchild, length);

position /= 2;

}

{

position = position \* 2 + 1;

CreateDef(T->rchild, length);

position /= 2;

}

return definition;

}

void DefOrder(def \*definition, int length)

//按结点位置对definition排序

{

BiTNode \*temp = NULL;

int t = 0;

for(int a = 0; a < length; a++)

{

for(int b = a + 1; b < length; b++)

{

if(definition[a].pos > definition[b].pos)

{

//结点指针交换

temp = definition[a].node;

definition[a].node = definition[b].node;

definition[b].node = temp;

//结点位置交换

t = definition[a].pos;

definition[a].pos = definition[b].pos;

definition[b].pos = t;

}

}

}

temp = NULL;

}

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

//按层遍历二叉树T

{

//全局变量处理

if(i != 0)

{

i = 0;

size = INIT\_DEF\_SIZE;

definition = (def\*)malloc(size \* sizeof(def));

position = 1;

}

int length = 0;

//按照二叉树构造其definition

definition = CreateDef(T, length);

//按结点位置排序

DefOrder(definition, length);

//输出排好序的definition

for(int a = 0; a < length; a++) visit(definition[a].node);

free(definition);//释放由alloc系列函数所分配的空间

return OK;

}

int PreSpaceNum(int level, int width)

{

int num = 1;

for(int i = 1; i < level; i++) num \*= 2;

width -= 6 \* num;

for(int i = 0; i < level; i++) width /= 2;

return width;

}

void Left(int level, int width, int a)

{

for(int j = 0; j < 2 \* PreSpaceNum(level, width) + 6 - a; j++) printf(" ");

printf("/");

}

void Right(int level, int width, int a)

{

printf("\\");

for(int j = 0; j < 2 \* PreSpaceNum(level, width) + 6 - a; j++) printf(" ");

}

void sleep(int level, int width, int a)

{

for(int j = 0; j < 2 \* PreSpaceNum(level, width) + 7 - a; j++) printf(" ");

}

status TreeLikeTraverse(BiTree T, void (\*visit)(BiTree))

{

//空树

if(T == NULL)

{

printf("\n\n");

return OK;

}

//全局变量处理

if(i != 0)

{

i = 0;

size = INIT\_DEF\_SIZE;

definition = (def\*)malloc(size \* sizeof(def));

position = 1;

}

int length = 0;

//按照二叉树构造其definition

definition = CreateDef(T, length);

//按结点位置排序

DefOrder(definition, length);

//输出当前树形

for(int i = 0; i < length; i++) printf("%d,%d,%d,%s ",i, definition[i].pos, definition[i].node->data.key, definition[i].node->data.others);

printf("\n");

int depth = BiTreeDepth(T);

int level = 1, width = 1;

for(int i = 1; i < depth; i++) width \*= 2; width \*= 6; //计算最宽

void (\*\*p)(int level, int width, int a) = NULL; //函数集合

//函数集合空间分配

p = (void (\*\*)(int level, int width, int a))malloc(2 \* width \* sizeof(void (\*)(int level, int width, int a)));

//函数集合初始化

for(int i = 0; i < 2 \* width; i++) p[i] = sleep;

for(int i = 0; i < length; i++)

{

if(definition[i].node->lchild != NULL) p[2 \* (definition[i].pos - 1)] = Left;

if(definition[i].node->rchild != NULL) p[2 \* (definition[i].pos - 1) + 1] = Right;

}

int begin = 1, end = 2;

for(int i = 1; i < level; i++) begin \*= 2; //begin为当前层数level的首结点序号

end = 2 \* begin - 1; //end为当前层数level的尾序号

for(int i = 1, a = 0; i < definition[length - 1].pos, a < length; i++)//遍历整棵树(包含设想存在的空结点)

{

//输出全部数据

while((definition[a].pos == 1) || (definition[a].pos >= begin && definition[a].pos <= end))

{

for(int i = 0; i < definition[a].pos - begin; i++)

for(int j = 0; j < 2 \* PreSpaceNum(level, width) + 6; j++)

printf(" ");

//输出实际存在的结点的数据

for(int j = 0; j < PreSpaceNum(level, width); j++) printf(" ");

printf("%2d,%-3s", definition[a].node->data.key, definition[a].node->data.others);

for(int j = 0; j < PreSpaceNum(level, width); j++) printf(" ");

//while循环动力及结束时机

a++;

if(a == length) //definition已经读完

{

break;

}

//之前没有加前置判断才采用了每次begin加一， 而这样会导致非完全二叉树读取陷入死循环

begin = definition[a].pos;//definition没读完，就读definition的下一个

if(begin > end) //即将进入下一层了 (此条件不对)

{

level++;

begin = 1;

for(int i = 1; i < level; i++) begin \*= 2; //begin为当前层数level的首结点序号

end = 2 \* begin - 1;

printf("\n");

break;

}

}

//进入下一层之前先输出斜线

if(a == length) //definition读完了(树到了最后一行)就不需要输出斜线了

break;

//此层个数count的计算

int count = 1;

for(int i = 1; i < level - 1; i++) count \*= 2;

for(int a = 1; a < PreSpaceNum(level, width) + 3; a++) //控制小斜线个数

{

int b = begin / 2;

if(!b) b = 1;

//遍历函数列表，调用该层所属函数

for(int i = 0; i < count; i++)

{

//printf("p[%d] was called!\n",2 \* (b + i - 1));//后续bug检查时使用

p[2 \* (b + i - 1)](level, width, a);

for(int j = 0; j < a - 1; j++) printf(" ");

p[2 \* (b + i - 1) + 1](level, width, a);

}

printf("\n");

}

}

free(definition);//释放由alloc系列函数所分配的空间

return OK;

}

status SaveBiTree(BiTree T, char FileName[])

//将二叉树的结点数据写入到文件FileName中

{

static FILE \*fp = fopen(FileName, "w+");

static int flag = 1;

int over = 0;

int key = 0; char others[20] = "null";

if(flag == 1)

{

over = 1;

flag = 0;

}

if(T != NULL)

{

key = T->data.key;

strcpy(others, T->data.others);

}

fprintf(fp, "%d %s ", key, others);

if(T == NULL) return OK;

SaveBiTree(T->lchild, FileName);

SaveBiTree(T->rchild, FileName);

if(over)

{

fprintf(fp, "%d %s ", -1, "null");

fclose(fp);

}

return OK;

}

status LoadBiTree(BiTree &T, char FileName[])

//读入文件FileName的结点数据，创建二叉树

{

FILE \*fp = fopen(FileName, "r");

if(fp == NULL) return ERROR;

int i = 0;

int number = 0;

TElemType definition[100];

while(fscanf(fp, "%d", &definition[i].key) != EOF && fscanf(fp, "%s", &definition[i].others) != EOF) i++;

CreateBiTree(T, definition);

fclose(fp);

return OK;

}

void visit(BiTree T)

{

printf(" %d,%s",T->data.key,T->data.others);

}

BiTree\* gather\_operate(FST& fst)

//森林操作

{

int op = 1; int j = 0;

char BiTreeName[30];

while(op)

{

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

printf(" 森林操作菜单\n");

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

printf(" 1. 初始化 6. 修改\n");

printf(" 2. 新增 7. 遍历\n");

printf(" 3. 删除 8. 销毁\n");

printf(" 4. 查找 0. 退出\n");

printf(" 5. 清空\n");

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

printf(" 请选择你的操作[0~6]:");

scanf("%d", &op);

switch(op)

{

case 1:

if(InitForest(fst) == INFEASIBLE) printf("当前森林已被初始化！\n");

else printf("初始化成功！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 2:

if(fst == NULL) printf("当前森林不存在，请先初始化！\n");

else

{

char BiTreeName[30];

int pos = 0;

printf("请输入新增树的名称：\n");

scanf("%s",BiTreeName);

printf("请输入新树种在几号位？\n");

scanf("%d", &pos);

if(AddBiTree(fst, pos, BiTreeName) == ERROR) printf("种树位置不合法！第%d位之前仍有空位！\n");

else printf("树\"%s\"成功种进森林了！\n", BiTreeName);

}

printf("----任意键继续----\n");

\_getch();

break;

case 3:

if(fst == NULL) printf("当前森林不存在，请先初始化！\n");

else if(fst->length == 0) printf("当前森林里没有树！\n");

else

{

printf("请输入想砍掉的树的名称：\n");

scanf("%s",BiTreeName);

if(RemoveBiTree(fst, BiTreeName)) printf("树\"%s\"成功被砍没了！\n",BiTreeName);

else printf("查无此树！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 4:

if(fst == NULL) printf("当前森林不存在，请先初始化！\n");

else

{

printf("请输入所要查找的树的名称：\n");

scanf("%s",BiTreeName);

if(j = LocateBiTree(fst, BiTreeName)) printf("树%s位于森林中第%d位\n", BiTreeName, j);

else printf("查无此树！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 5:

if(ClearForest(fst) == INFEASIBLE) printf("当前森林不存在，请先初始化！\n");

else printf("清空成功！森林空了！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 6:

if(fst == NULL)

{

printf("当前森林不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

else

{

printf("请输入所需更改的树的名称:\n");

scanf("%s",BiTreeName);

j = LocateBiTree(fst, BiTreeName);

if(j == 0)

{

printf("查无此树！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

else

{

int ord = 0;

printf("是否要修改树的名字?(是，摁1 / 否，摁0)：");

scanf("%d", &ord);

if(ord)

{

printf("输入新名字：");

scanf("%s", fst->tree[j - 1]->name);

printf("修改成功！\n");

printf("继续修改数据吗？(是，摁1 / 否，摁0):");

scanf("%d", &ord);

if(!ord) break;

}

return &(fst->tree[j - 1]->T);

}

}

case 7:

if((j = ForestTraverse(fst)) == 0) printf("当前森林一棵树也没有！\n");

else if(j == INFEASIBLE) printf("当前森林不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 8:

if(DestroyForest(fst) == INFEASIBLE) printf("啧..你还是不记得初始化！\n");

else printf("销毁成功！森林没了！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 0:

break;

default:

printf("指令错误！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

}

return NULL;

}

## 实验三.cpp

#include"fun.h"

int main(){

BiTree T = NULL; //树

FST fst = NULL; //森林

int op = 1; //操作指令

int flag = 0; //森林操作完成标志

DEF definition[100]; //输入树中的数据载体

int e = 0; //键值载体

BiTNode \*node = NULL; //结点载体

BiTree \*T\_ope = NULL, temp = NULL;//森林操作树的载体

TElemType value; //结点数据载体

int LR; //插入节点：定左右

while(op){

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

printf(" 基于链式存储的线性表操作菜单\n");

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

printf(" 1. 初始化树 8. 获得兄弟结点 15. 树形遍历\n");

printf(" 2. 初始化输入 9. 插入节点 16. 存档\n");

printf(" 3. 摧毁树 10. 删除结点 17. 读档\n");

printf(" 4. 清空树 11. 前序遍历 18. 森林管理\n");

printf(" 5. 树的深度 12. 中序遍历 0. 退出\n");

printf(" 6. 结点查找 13. 后序遍历\n");

printf(" 7. 结点赋值 14. 按层遍历\n");

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

if(op)

printf(" 请选择你的操作[0~17]:");

scanf("%d",&op);

switch(op){

case 1:

if(InitBiTree(T) == OK) printf("初始化成功！\n");

else printf("当前的树里已经有点东西了~\n");

printf("----任意键继续----\n");

\_getch();

break;

case 2:

if(T == NULL) printf("初始化都没做的树不配存数据！\n");

else if(T->lchild != NULL) printf("当前的树里已经有点东西了~不可以进行初始化输入了\n");

else{

printf("输入数据：(格式：满二叉树中的位置序号+键值+数据)，位置序号为0时结束\n");

do {

scanf("%d%d%s",&definition[i].pos,&definition[i].data.key,definition[i].data.others);

} while (definition[i++].pos);

CreateBiTree(T->lchild,definition);

printf("输入成功！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 3:

if(T == NULL) printf("想销毁？先初始化一下再来~\n");

else if(flag == 1) printf("此处不支持销毁树，如需销毁，请选择森林管理中的“删除”功能！\n");

else{

ClearBiTree(T);

printf("销毁成功！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 4:

if(T == NULL) printf("又忘了初始化？！\n");

else{

ClearBiTree(T->lchild);

printf("清空成功！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 5:

if(T == NULL) printf("这里一棵树都不存在哦~\n");

else printf("当前树的深度为%d\n", BiTreeDepth(T->lchild));

printf("----任意键继续----\n");

\_getch();

break;

case 6:

if(T == NULL) printf("找啥找？树都没有\n");

else{

printf("输入键值：");

scanf("%d", &e);

node = LocateNode(T->lchild, e);

if(node == NULL) printf("当前树里面没有key值为%d的结点\n", e);

else printf("当前结点键值为:%d, 数据为:%s\n", node->data.key, node->data.others);

}

printf("----任意键继续----\n");

\_getch();

break;

case 7:

if(T == NULL) printf("当前树不存在！\n");

else{

printf("输入所要修改的结点键值：");

scanf("%d", &e);

node = LocateNode(T->lchild, e);

if(node == NULL) printf("当前树里面没有key值为%d的结点\n", e);

else

{

printf("请输入输入更改后的结点键值：");

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

printf("请输入更改后的数据：");

scanf("%s", value.others);

if(Assign(T->lchild, e, value) == ERROR)

printf("更改键值出错！因为原树中已存在其他结点的键值为%d！\n", value.key);

else printf("更改成功！\n");

}

}

printf("----任意键继续----\n");

\_getch();

break;

case 8:

if(T == NULL) printf("答应我，先初始化好嘛~\n");

else

{

printf("输入当前键值：");

scanf("%d", &e);

node = LocateNode(T->lchild, e);

if(node == NULL) printf("当前树里面没有key值为%d的结点\n", e);

else

{

node = GetSibling(T->lchild, e);

if(node == NULL) printf("这个结点没有兄弟~\n");

else printf("这个结点的兄弟结点的键值为：%d, 数据为：%s\n", node->data.key, node->data.others);

}

}

printf("----任意键继续----\n");

\_getch();

break;

case 9:

if(T == NULL) printf("哼！又不记得初始化\n");

else

{

printf("想给哪个结点添个孩子？(输入相应结点键值即可)\n");

scanf("%d", &e);

node = LocateNode(T->lchild, e);

if(node == NULL) printf("当前树里面没有key值为%d的结点\n", e);

else{

printf("想添个左孩子还是右孩子？(左孩子输0，右孩子输1，当祖宗输-1)\n");

scanf("%d", &LR);

printf("输入新增结点键值：");

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

printf("输入新增结点数据：");

scanf("%s", value.others);

if(InsertNode(T->lchild, e, LR, value) == ERROR) printf("新增键值出错！因为原树中已存在其他结点的键值为%d！\n", value.key);

else printf("插入节点成功！\n");

}

}

printf("----任意键继续----\n");

\_getch();

break;

case 10:

if(T == NULL) printf("请先初始化一棵树！\n");

else{

printf("输入待删除结点的键值：");

scanf("%d", &e);

if(DeleteNode(T, e) == ERROR) printf("当前树里面没有key值为%d的结点\n", e);

else printf("删除成功！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 11:

if(T == NULL) printf("先去初始化好嘛~\n");

else

{

printf("前序遍历当前树中结点信息如下：\n");

PreOrderTraverse(T->lchild, visit);

printf("\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 12:

if(T == NULL) printf("先去初始化好嘛~\n");

else

{

printf("中序遍历当前树中结点信息如下：\n");

InOrderTraverse(T->lchild, visit);

printf("\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 13:

if(T == NULL) printf("先去初始化好嘛~\n");

else

{

printf("后序遍历当前树中结点信息如下：\n");

PostOrderTraverse(T->lchild, visit);

printf("\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 14:

if(T == NULL) printf("先去初始化好嘛~\n");

else

{

printf("按层遍历当前树中结点信息如下：\n");

LevelOrderTraverse(T->lchild, visit);

printf("\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 15:

if(T == NULL) printf("先去初始化好嘛~\n");

else

{

printf("当前树就长下面这个样子：\n");

TreeLikeTraverse(T->lchild, visit);

printf("\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 16:

if(T == NULL) printf("您还未初始化！\n");

else{

char SaveFile[40];

printf("输入存档文件路径(输入0选择默认路径):\n");

scanf("%s",SaveFile);

if(!strcmp(SaveFile,"0")) strcpy(SaveFile, "D:\\BiTree\_save.dat");

SaveBiTree(T->lchild, SaveFile);

printf("已将数据存入 %s\n",SaveFile);

}

printf("----任意键继续----\n");

\_getch();

break;

case 17:

if(T == NULL) printf("树不存在树不存在树不存在！\n");

else if(T->lchild != NULL) printf("当前这颗树非空！不准你读档！\n");

else

{

char LoadFile[40];

printf("输入读档文件路径(输入0选择默认路径):\n");

scanf("%s",LoadFile);

if(!strcmp(LoadFile,"0")) strcpy(LoadFile, "D:\\BiTree\_save.dat");

if(LoadBiTree(T->lchild, LoadFile) == ERROR) printf("存档文件%s无法打开！\n", LoadFile);

else printf("数据已从 %s 中读出\n",LoadFile);

}

printf("----任意键继续----\n");

\_getch();

break;

case 18:

if(flag == 1) //修改状态，不可再次选择线性表集合操作

{

printf("当前有未结束的线性表修改，请先结束前一修改！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

T\_ope = gather\_operate(fst);

if(T\_ope != NULL)//只要选择了集合操作中的修改，就将主界面线性表置换成集合中所选线性表

{

temp = T; //temp暂存主界面线性表数据

T = \*T\_ope; //主界面后台线性表置换为集合内所选线性表

flag = 1; //系统状态置为集合操作态

}

break;

case 0:

break;

default:

printf("操作指令有误！\n");

printf("----任意键继续----\n");

\_getch();

break;

}//end of switch

if(flag && op == 0) //flag判断当前是否在线性表集合操作中，此时退出返回主界面

{

T = temp; //返还原始主界面线性表数据

op = 1; //保持不退出while

flag = 0; //状态置为初始界面

printf("原主界面的树中数据已载入！\n");

\_getch();

}

}//end of while

printf("欢迎下次再使用本系统！\n");

printf("----任意键结束----\n");

\_getch();

return 0;

}//end of main()

# 附录B 基于链式存储结构线性表实现的源程序

/\* Linear Table On Linked Structure \*/

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <string.h>

#include <conio.h>

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

#define FAIL -3

typedef int status;

typedef int ElemType; //数据元素类型定义

typedef struct LNode{ //单链表（链式结构）结点的定义

ElemType data;

struct LNode \*next;

}LNode,\*LinkList;

typedef struct LISTS{

char name[30];

LinkList L = NULL;

LISTS \*next;

} LISTS, \*pLISTS;

status InitList(LinkList& L);// 线性表L不存在，构造一个空的线性表，返回OK，否则返回INFEASIBLE。

status InitInput(LinkList L);

status DestroyList(LinkList &L);

status ClearList(LinkList L);

status ListEmpty(LinkList L);

int ListLength(LinkList L);

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

status LocateElem(LinkList L,ElemType e); //简化过

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

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

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

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

status ListTraverse(LinkList L); //简化过

status SaveList(LinkList L,char FileName[]);

status LoadList(LinkList L,char FileName[]);

status InitLists(pLISTS &Lists);

int ListsLength(pLISTS Lists); //求线性表集合的长度

status AddList(pLISTS Lists, int i, char ListName[]);

status RemoveList(pLISTS Lists,char ListName[]);

int LocateList(pLISTS Lists,char ListName[]); //在当前集合中查找名为 ListsName 的线性表

status ListsTraverse(pLISTS Lists); //遍历当前集合

status SaveLists(pLISTS Lists, char \*FileName); //线性表集合存档

LinkList gather\_operate(pLISTS &Lists);

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

int main(){

LinkList L = NULL; int op=1;

int j = 0, flag = 0;

LinkList L\_ope = NULL, temp = NULL;

pLISTS Lists = NULL;

while(op){

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

printf(" 基于链式存储的线性表操作菜单\n");

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

printf(" 1. 初始化线性表 7. 由位置求元素 13. 遍历线性表\n");

printf(" 2. 初始化输入 8. 查找元素 14. 保存线性表\n");

printf(" 3. 摧毁线性表 9. 前驱元素 15. 读取线性表\n");

printf(" 4. 清空线性表 10. 后继元素 16. 线性表集合操作\n");

printf(" 5. 线性表判空 11. 插入元素 0. 退出\n");

printf(" 6. 线性表表长 12. 删除元素\n");

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

if(op)

printf(" 请选择你的操作[0~16]:");

scanf("%d",&op);

switch(op){

case 1:

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

else printf("线性表创建失败！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 2:

j = InitInput(L);

if(j == INFEASIBLE) printf("当前线性表不存在，请先初始化！\n");

else if(j == ERROR) printf("当前线性表非空，不可进行初始化输入！\n");

else printf("初始化输入成功！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 3:

j = DestroyList(L);

if(j == INFEASIBLE) printf("当前线性表不存在！\n");

else printf("销毁成功！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 4:

j = ClearList(L);

if(j == INFEASIBLE) printf("当前线性表不存在！\n");

else printf("清空成功！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 5:

j = ListEmpty(L);

if(j == INFEASIBLE) printf("当前线性表不存在!\n");

else if(j) printf("当前线性表为空\n");

else printf("当前线性表不为空\n");

printf("----任意键继续----\n");

\_getch();

break;

case 6:

j = ListLength(L);

if(j == INFEASIBLE) printf("当前线性表不存在！\n");

else printf("当前线性表长度为%d\n",j);

printf("----任意键继续----\n");

\_getch();

break;

case 7:

if(L == NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

int i, e;

printf("查找第几位元素？\n");

scanf("%d",&i);

j = GetElem(L, i, e);

if(j == ERROR) printf("查找位置不合法！\n");

else printf("当前线性表中第%d个元素为%d\n", i, e);

printf("----任意键继续----\n");

\_getch();

break;

case 8:

if(L == NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("输入所需查找的元素内容\n");

scanf("%d", &e);

j = LocateElem(L, e);

if(j == ERROR) printf("表中不存在元素 %d\n",e);

else printf("元素%d位于线性表中第%d个位置\n", e, j);

printf("----任意键继续----\n");

\_getch();

break;

case 9:

if(L == NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

int prior;

printf("输入当前元素\n");

scanf("%d", &e);

j = PriorElem(L, e, prior);

if(j == ERROR && LocateElem(L, e)) printf("此元素无前驱！\n");

else if(j != ERROR) printf("当前元素前驱为%d\n", prior);

else printf("此元素不在当前线性表中！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 10:

if(L == NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

int next;

printf("输入当前元素\n");

scanf("%d", &e);

j = NextElem(L, e, next);

if(j == ERROR && LocateElem(L, e)) printf("此元素无后继！\n");

else if(j != ERROR) printf("当前元素后继为%d\n", next);

else printf("此元素不在当前线性表中！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 11:

if(L == NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("输入加入元素\n");

scanf("%d", &e);

printf("想插入第几个元素前？\n");

scanf("%d", &i);

j = ListInsert(L, i, e);

if(j == ERROR) printf("插入位置非法！\n");

else

{

printf("插入成功！\n");

printf("当前线性表数据如下：\n");

ListTraverse(L);

}

printf("----任意键继续----\n");

\_getch();

break;

case 12:

if(L == NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("删除第几个元素？\n");

scanf("%d",&i);

j = ListDelete(L, i, e);

if(j == ERROR) printf("删除位置非法！\n");

else

{

printf("删除成功！\n");

if(ListEmpty(L)) printf("当前线性表已经被你删空了~\n");

else

{

printf("当前线性表中数据如下：\n");

ListTraverse(L);

}

}

printf("----任意键继续----\n");

\_getch();

break;

case 13:

ListTraverse(L);

printf("----任意键继续----\n");

\_getch();

break;

case 14:

if(L == NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

char SaveFile[40];

printf("输入存档文件路径(输入0选择默认路径):\n");

scanf("%s",SaveFile);

if(!strcmp(SaveFile,"0")) strcpy(SaveFile, "D:\\LinkList\_save.dat");

if(SaveList(L, SaveFile) == INFEASIBLE) printf("当前线性表不存在！\n");

else printf("已将数据存入 %s\n",SaveFile);

printf("----任意键继续----\n");

\_getch();

break;

case 15:

if(L == NULL)

{

printf("当前线性表不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

else if(L->next != NULL)

{

printf("当前线性表非空！读取会出大问题！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

char LoadFile[40];

printf("输入读档文件路径(输入0选择默认路径):\n");

scanf("%s",LoadFile);

if(!strcmp(LoadFile,"0")) strcpy(LoadFile, "D:\\LinkList\_save.dat");

if((j = LoadList(L, LoadFile)) == ERROR) printf("当前线性表中已有数据！\n");

else if(j == OK) printf("数据已从 %s 中读出\n",LoadFile);

else if(j == INFEASIBLE) printf("当前线性表不存在！请先初始化！\n");

else if(j == FAIL) printf("存档文件建立失败！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 16:

if(flag == 1) //修改状态，不可再次选择线性表集合操作

{

printf("当前有未结束的线性表修改，请先结束前一修改！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

L\_ope = gather\_operate(Lists);

if(L\_ope != NULL)//只要选择了集合操作中的修改，就将主界面线性表置换成集合中所选线性表

{

temp = L; //temp暂存主界面线性表数据

L = L\_ope; //主界面后台线性表置换为集合内所选线性表

flag = 1; //系统状态置为集合操作态

}

break;

case 0:

break;

default:

printf("操作指令有误！\n");

printf("----任意键继续----\n");

\_getch();

break;

}//end of switch

if(flag && op == 0) //flag判断当前是否在线性表集合操作中，此时退出返回主界面

{

L = temp; //返还原始主界面线性表数据

op = 1; //保持不退出while

flag = 0; //状态置为初始界面

printf("原线性表数据已载入！\n");

\_getch();

}

}//end of while

printf("欢迎下次再使用本系统！\n");

printf("----任意键结束----\n");

\_getch();

return 0;

}//end of main()

status InitList(LinkList &L)

// 线性表L不存在，构造一个空的线性表，返回OK，否则返回INFEASIBLE。

{

if(L != NULL)

{

return INFEASIBLE;

}

else

{

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

L->next = NULL;

return OK;

}

}

status InitInput(LinkList L)

// 线性表不存在，返回INFEASIBLE；如果线性表为空表，输入数据；如果线性表不为空，返回ERROR

{

if(L == NULL)

{

return INFEASIBLE;

}

else if(L->next != NULL)

{

return ERROR;

}

else

{

ElemType temp = 0;

printf("请输入一组数据(0结束):\n");

scanf("%d",&temp);

while(temp)

{

L->next = (LinkList)malloc(sizeof(LNode));

L = L->next;

L->data = temp;

scanf("%d",&temp);

}

L->next = NULL;

}

}

status DestroyList(LinkList& L)

// 如果线性表L存在，销毁线性表L，释放数据元素的空间，返回OK，否则返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

else

{

LinkList p = NULL;

while(L != NULL)

{

p = L->next;

free(L);

L = p;

}

return OK;

}

}

status ClearList(LinkList L)

// 如果线性表L存在，删除线性表L中的所有元素，返回OK，否则返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

else

{

LinkList p = L->next, loc = L;

while(p != NULL)

{

L = p->next;

free(p);

p = L;

}

L = loc;

L->next = NULL;

return OK;

}

}

status ListEmpty(LinkList L)

// 如果线性表L存在，判断线性表L是否为空，空就返回TRUE，否则返回FALSE；如果线性表L不存在，返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

else if(L->next == NULL)

{

return TRUE;

}

else

{

return FALSE;

}

}

int ListLength(LinkList L)

// 如果线性表L存在，返回线性表L的长度，否则返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

int count = 0;

while(L->next != NULL)

{

L = L->next;

count++;

}

return count;

}

status GetElem(LinkList L,int i,ElemType &e)

// 如果线性表L存在，获取线性表L的第i个元素，保存在e中，返回OK；如果i不合法，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

{

if(L == NULL) //线性表不存在时

{

return INFEASIBLE;

}

else if(i < 1) //查找位置非法，返回ERROR

{

return ERROR;

}

else

{

int count = 0;

while(L->next != NULL) //L不为倒数第一个时

{

L = L->next;

count++;

if(count == i) //当前位置是否为所找位置

{

e = L->data;

return OK;

}

}

return ERROR; //查找位置非法，超出表长

}

}

status LocateElem(LinkList L,ElemType e)

// 如果线性表L存在，查找元素e在线性表L中的位置序号；如果e不存在，返回ERROR；当线性表L不存在时，返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

else

{

int count = 0;

while(L->next != NULL)

{

L = L->next;

count++;

if(L->data == e)

{

return count;

}

}

return ERROR;

}

}

status PriorElem(LinkList L,ElemType e,ElemType &pre)

// 如果线性表L存在，获取线性表L中元素e的前驱，保存在pre中，返回OK；如果没有前驱，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

else

{

L = L->next;

while(L != NULL && L->next != NULL) //当L是最后一个(表明倒数第二个的next里的data不等于e)时返回ERROR

{

if(L->next->data == e)

{

pre = L->data;

return OK;

}

L = L->next;

}

return ERROR;

}

}

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

// 如果线性表L存在，获取线性表L元素e的后继，保存在next中，返回OK；如果没有后继，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

else

{

L = L->next;

while(L->next != NULL)

{

if(L->data == cur)

{

next\_e = L->next->data;

return OK;

}

L = L->next;

}

return ERROR;

}

}

status ListInsert(LinkList L,int i,ElemType e)

// 如果线性表L存在，将元素e插入到线性表L的第i个元素之前，返回OK；当插入位置不正确时，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

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

{

return ERROR;

}

else

{

int count = 0;

//新建存储节点

LinkList new\_p = (LinkList)malloc(sizeof(LNode));

new\_p->next = NULL;

new\_p->data = e;

//查找插入位置

while(L != NULL)

{

if(count == i - 1) //查找位置的前一个

{

new\_p->next = L->next;

L->next = new\_p;

return OK;

}

L = L->next;

count++;

}

}

}

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

// 如果线性表L存在，删除线性表L的第i个元素，并保存在e中，返回OK；当删除位置不正确时，返回ERROR；如果线性表L不存在，返回INFEASIBLE。

{

if(L == NULL)

{

return INFEASIBLE;

}

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

{

return ERROR;

}

else

{

int count = 0;

LinkList p = NULL;

while(L != NULL)

{

if(count == i - 1)

{

p = L->next;

e = L->next->data;

L->next = L->next->next;

free(p);

return OK;

}

L = L->next;

count++;

}

}

}

status ListTraverse(LinkList L)

// 如果线性表L存在，依次显示线性表中的元素，每个元素间空一格，返回OK；如果线性表L不存在，返回INFEASIBLE。

{

if(L == NULL) //线性表不存在时

{

printf("当前线性表不存在！请先初始化！\n");

return INFEASIBLE;

}

else if(L->next == NULL)

{

printf("别看了别看了，里边啥也没有~\n");

return 0;

}

else

{

L = L->next;

while(L != NULL)

{

printf("%d",L->data);

if(L->next != NULL)

{

printf(" ");

}

else

{

printf("\n");

}

L = L->next;

}

return OK;

}

}

status SaveList(LinkList L,char FileName[])

//线性表存档，线性表存在时将线性表中数据存入文件并返回OK，否则返回INFEASIBLE

{

if(L == NULL)

{

return INFEASIBLE;

}

else

{

FILE \*fp;

fp = fopen(FileName,"w+");

L = L->next;

while(L != NULL)

{

fprintf(fp,"%d ",L->data);

L = L->next;

}

fclose(fp);

return OK;

}

}

status LoadList(LinkList L,char FileName[])

//线性表读档，如果线性表不存在，返回INFEASIBLE，线性表为空时读入数据，非空时返回ERROR

{

if(L == NULL) //线性表不存在时需要先初始化才可读档

{

return INFEASIBLE;

}

else if(L->next != NULL) //线性表中已有数据时不可读档

{

return ERROR;

}

else

{

FILE \*fp = fopen(FileName,"r");

ElemType in;

LinkList track = NULL;//跟在 L后面

L->next = (LinkList)malloc(sizeof(LNode));

L = L->next;

while(fscanf(fp,"%d",&in) != EOF)

{

L->data = in;

L->next = (LinkList)malloc(sizeof(LNode));

track = L;

L = L->next;

}

track->next = NULL;

fclose(fp);

return OK;

}

}

status InitLists(pLISTS &Lists)

//初始化线性表集合

{

if(Lists == NULL)

{

Lists = (pLISTS)malloc(sizeof(LISTS));

Lists->L = NULL;

InitList(Lists->L);

Lists->next = NULL;

return OK;

}

else

{

return INFEASIBLE;

}

}

int ListsLength(pLISTS Lists)

{

if(Lists == NULL)

{

return INFEASIBLE;

}

int count = 0;

Lists = Lists->next;

while(Lists != NULL)

{

Lists = Lists->next;

count++;

}

return count;

}

status AddList(pLISTS Lists, int i, char ListName[])

//在线性表第i位插入一个名为ListName的线性表

{

if(Lists == NULL)

{

return INFEASIBLE;

}

if(i < 1)

{

printf("插入位置非法！\n");

return ERROR;

}

pLISTS new\_List = NULL;

InitLists(new\_List);

strcpy(new\_List->name, ListName);

int count = 0; //记录当前元素位置

pLISTS p = NULL;

while(Lists != NULL)

{

if(count == i-1)

{

new\_List->next = Lists->next;

Lists->next = new\_List;

printf("线性表%s已成功插入集合中第%d位！\n", ListName, i);

return OK;

}

Lists = Lists->next;

count++;

}

printf("插入位置非法！\n");

return ERROR;

}

status RemoveList(pLISTS Lists,char ListName[])

{

if(Lists == NULL)

{

return INFEASIBLE;

}

while(Lists->next != NULL)

{

if(!strcmp(Lists->next->name, ListName))

{

pLISTS p = Lists->next;

Lists->next = Lists->next->next;

DestroyList(p->L);

free(p);

printf("线性表%s已成功删除！\n",ListName);

return OK;

}

Lists = Lists->next;

}

printf("查无此表！\n");

return ERROR;

}

int LocateList(pLISTS Lists,char ListName[])

{

if(Lists == NULL)

{

return INFEASIBLE;

}

int count = 1;

Lists = Lists->next;

while(Lists != NULL)

{

if(!strcmp(Lists->name, ListName))

{

return count;

}

Lists = Lists->next;

count++;

}

}

status ListsTraverse(pLISTS Lists)

//遍历当前集合

{

if(Lists == NULL)

{

printf("当前线性表不存在！请先初始化！\n");

return INFEASIBLE;

}

if(ListsLength(Lists) == 0)

{

printf("阿巴阿巴...这个线性表集合里啥也没有啊~\n");

return 0;

}

printf("当前线性表集合中数据如下：\n");

Lists = Lists->next;

while(Lists != NULL)

{

printf("%s\n",Lists->name);

ListTraverse(Lists->L);

Lists = Lists->next;

}

return OK;

}

status SaveLists(pLISTS Lists, char \*FileName)

{

if(Lists == NULL)

{

return INFEASIBLE;

}

if(Lists->next == NULL)

{

return 0;

}

FILE \*fp = fopen(FileName,"w+");

Lists = Lists->next;

while(Lists != NULL)

{

fprintf(fp,"%s\n",Lists->name);

LinkList p = Lists->L->next;

int len = ListLength(Lists->L), count = 1;

while(p != NULL)

{

fprintf(fp, "%d", p->data);

if(count == len)

{

fprintf(fp, "%c", '\n');

}

else

{

fprintf(fp, "%c", ' ');

}

p = p->next;

count++;

}

Lists = Lists->next;

}

fclose(fp);

return OK;

}

LinkList gather\_operate(pLISTS &Lists)

//线性表集合操作

{

int op = 1, j = 0, i = 0;

char ListName[30];

LinkList L\_ope = NULL;

while(op)

{

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

printf(" 线性表集合操作菜单\n");

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

printf(" 1. 初始化 5. 修改\n");

printf(" 2. 插入 6. 遍历\n");

printf(" 3. 删除 7. 存档(读档未实现)\n");

printf(" 4. 查找 0. 退出\n");

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

printf(" 请选择你的操作[0~7]:");

scanf("%d", &op);

switch(op)

{

case 1:

if(InitLists(Lists) == INFEASIBLE) printf("当前线性表集合已被初始化！\n");

else printf("初始化成功！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 2:

if(Lists == NULL)

{

printf("当前线性表集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

char ListName[30];

printf("请输入新增线性表的名称：\n");

scanf("%s",ListName);

printf("想插入第几位线性表之前？\n");

scanf("%d",&i);

AddList(Lists, i, ListName);

printf("----任意键继续----\n");

\_getch();

break;

case 3:

if(Lists == NULL)

{

printf("当前线性表集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

if(ListsLength(Lists) == 0)

{

printf("别删了~里面已经啥也没有了~\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("请输入所要删除的线性表名称：\n");

scanf("%s",ListName);

RemoveList(Lists, ListName);

printf("----任意键继续----\n");

\_getch();

break;

case 4:

if(Lists == NULL)

{

printf("当前线性表集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("请输入所要查找的线性表名称：\n");

scanf("%s",ListName);

if(j = LocateList(Lists, ListName)) printf("线性表%s位于集合中第%d位\n", ListName, j);

else printf("查无此表！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 5:

if(Lists == NULL)

{

printf("当前线性表集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("请输入所需更改的表名:\n");

scanf("%s",ListName);

j = LocateList(Lists, ListName);

if(j == 0)

{

printf("查无此表！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

else

{

pLISTS p = Lists;

for(int i = 0; i < j; i++) p = p->next;

return p->L;

}

case 6:

ListsTraverse(Lists);

printf("----任意键继续----\n");

\_getch();

break;

case 7:

if(Lists == NULL)

{

printf("当前线性表集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

char FileName[40];

printf("输入存档文件路径(输入0选择默认路径):\n");

scanf("%s",FileName);

if(!strcmp(FileName,"0")) strcpy(FileName, "D:\\LISTS\_save.dat");

if(SaveLists(Lists, FileName) == INFEASIBLE) printf("当前线性表中无数据！\n");

else printf("线性表集合中数据已被存入%s中！\n", FileName);

printf("----任意键继续----\n");

\_getch();

break;

case 0:

break;

default:

printf("指令错误！\n");

printf("----任意键继续----\n");

\_getch();

}

}

return L\_ope;

}

# 附录C：基于二叉树、二叉链表的源程序的实现

## def.h

#include "stdio.h"

#include "stdlib.h"

#include "string.h"

#include "ctype.h"

#include <conio.h>

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

#define LEFT 4

#define RIGHT 6

#define max(a,b) ((a>b)?a:b)

//用于按层遍历中definition扩容

#define INIT\_DEF\_SIZE 64

#define INCREMENT 16

//用于森林扩容

#define INIT\_SIZE 100

typedef int status;

typedef int KeyType;

typedef struct {

KeyType key;

char others[20];

} TElemType; //二叉树结点类型定义

typedef struct BiTNode{ //二叉链表结点的定义

TElemType data;

struct BiTNode \*lchild,\*rchild;

} BiTNode, \*BiTree;

typedef struct {

int pos;

TElemType data;

} DEF;

typedef struct { //二叉树的结点definition，包含pos以及pos对应的结点地址，仅用于按层遍历

int pos;

BiTNode \*node;

} def;

typedef struct { //森林中每棵树的定义

BiTree T;

char name[30];

} Tree, \*pTree;

typedef struct { //森林定义(线性表结构存储)

pTree \*tree;

int length;

int size;

} FOREST, \*FST;

## Stack.h

//设计简易栈

#include"def.h"

#define INIT\_STACK\_SIZE 20

#define INCREMENT 16

typedef struct{

BiTree \*elem = NULL;

int top = 0;

int stacksize = 0;

} Stack;

status InitStack(Stack &s); //栈的初始化

status Push(Stack &s, BiTNode \*p); //p进栈

status Pop(Stack &s, BiTree &p); //出栈

status StackEmpty(Stack s); //栈判空

status InitStack(Stack &s)

//栈的初始化，s中有内容时返回INFEASIBLE表示初始化失败，否则返回OK表示初始化成功

{

if(s.elem != NULL) return INFEASIBLE;

s.stacksize = INIT\_STACK\_SIZE;

s.elem = (BiTree \*)calloc(s.stacksize, sizeof(BiTree)); //数组中全为地址，避免野指针

s.top = 0;

return OK;

}

status Push(Stack &s, BiTNode \*p)

//p进栈，栈不存在返回INFEASIBLE，栈满时扩容，成功之后返回OK

{

if(s.elem == NULL) return INFEASIBLE;

s.elem[s.top++] = p;

if(s.top >= s.stacksize) //栈满，realloc扩容

{

s.stacksize += INCREMENT;

s.elem = (BiTree \*)realloc(s.elem, s.stacksize \* sizeof(BiTree));

}

return OK;

}

status Pop(Stack &s, BiTree &p)

//出栈，栈不存在返回INFEASIBLE，栈空返回ERROR，否则返回OK

{

if(s.elem == NULL) return INFEASIBLE;

if(s.top == 0) return ERROR;

p = s.elem[--s.top];

s.elem[s.top + 1] = NULL; //保存的是地址值，不允许残留

return OK;

}

status StackEmpty(Stack s)

//栈判空，栈不存在返回INFEASIBLE，栈空返回TRUE，否则返回FALSE

{

if(s.elem == NULL) return INFEASIBLE;

if(s.top == 0) return TRUE;

return FALSE;

}

## fun.h

#include "Stack.h"

//按层遍历全局变量不易处理，提出

static int i = 0, size = INIT\_DEF\_SIZE;

static def \*definition = (def\*)malloc(size \* sizeof(def));

static int position = 1;

status InitBiTree(BiTree &T);

status CreateBiTree(BiTree &T,DEF definition[]);

status CreateBiTree(BiTree &T,TElemType definition[]);

/\*根据先序遍历序列definition创建二叉树 T，definition中如有重复则返回ERROR，否则返回OK\*/

status ClearBiTree(BiTree &T); //将二叉树设置成空，并删除所有结点，释放结点空间

bool IsRepeat(TElemType definition[]); //判断先序序列definition中是否有重复key值，用于创建树

int BiTreeDepth(BiTree T); //求二叉树T的深度

BiTNode\* LocateNode(BiTree T,KeyType e); //查找结点

status Assign(BiTree &T,KeyType e,TElemType value); //实现结点赋值。

BiTNode\* GetSibling(BiTree T,KeyType e); //实现获得兄弟结点

status InsertNode(BiTree &T,KeyType e,int LR,TElemType c); //实现插入节点，在e的LR(左右)孩子处插入c

status ParentNode(BiTree T, KeyType e, BiTree &p);

/\*寻找key值为e元素的双亲结点，由参数p带出函数，返回OK表示删除结点为根节点，

返回LEFT表示e为p的左孩子，返回RIGHT表示e为p的右孩子，返回ERROR表示没有此结点\*/

BiTNode\* RightMostNode(BiTree T); //寻找 T 最右端结点，用于删除

status DeleteNode(BiTree &T,KeyType e); //删除结点。

status PreOrderTraverse(BiTree T,void (\*visit)(BiTree)); //先序遍历二叉树T(递归)

status InOrderTraverse(BiTree T,void (\*visit)(BiTree)); //先序遍历二叉树T(非递归)

status PostOrderTraverse(BiTree T,void (\*visit)(BiTree)); //后序遍历二叉树T(递归)

def\* CreateDef(BiTree T, int &length); //按照结点位置构造二叉树的definition

void DefOrder(def \*definition, int length); //按结点位置对definition排序

status LevelOrderTraverse(BiTree T,void (\*visit)(BiTree)); //按层遍历二叉树T

status SaveBiTree(BiTree T, char FileName[]); //存档

status LoadBiTree(BiTree &T, char FileName[]); //读档

status TreeLikeTraverse(BiTree T, void (\*visit)(BiTree)); //树形遍历，输出树的图形表示

void visit(BiTree T);

BiTree\* gather\_operate(FST& fst);

//森林操作

status InitForest(FST& fst)

//初始化森林(以线性表形式定义)

{

if(fst != NULL) return INFEASIBLE;

fst = (FST)malloc(sizeof(FOREST));

fst->tree = (pTree\*)malloc(INIT\_SIZE \* sizeof(pTree));

fst->length = 0;

fst->size = INIT\_SIZE;

return OK;

}

status AddBiTree(const FST fst, int i, char BiTreeName[])

//向森林中第i位种入名为BiTreeName的树

{

if(fst == NULL) return INFEASIBLE;

if(i > fst->length + 1 || i < 1) return ERROR;

if(fst->length >= fst->size) //扩容

{

fst->size += INCREMENT;

fst->tree = (pTree\*)realloc(fst->tree, fst->size \* sizeof(pTree));

}

for(int j = fst->length; j >= i; j--) //移位置

{

fst->tree[j] = fst->tree[j - 1];

}

fst->tree[i - 1] = (pTree)malloc(sizeof(Tree));

strcpy(fst->tree[i - 1]->name, BiTreeName); //树命名 && 树初始化

fst->tree[i - 1]->T = NULL;

InitBiTree(fst->tree[i - 1]->T);

fst->length++;

return OK;

}

status RemoveBiTree(const FST fst, char BiTreeName[])

//砍去森林中名为BiTreeName的树

{

if(fst == NULL) return INFEASIBLE;

for(int i = 0; i < fst->length; i++)

{

if(!strcmp(BiTreeName, fst->tree[i]->name))

{

ClearBiTree(fst->tree[i]->T);

free(fst->tree[i]);

for(int a = i + 1; a < fst->length; a++)

fst->tree[a - 1] = fst->tree[a];

fst->length--;

return OK;

}

}

return ERROR;

}

int LocateBiTree(const FST fst, char BiTreeName[])

//查找当前森林中名为BiTreeName的树

{

if(fst == NULL) return INFEASIBLE;

for(int i = 0; i < fst->length; i++)

{

if(!strcmp(BiTreeName, fst->tree[i]->name)) return i + 1;

}

return ERROR;

}

status ForestTraverse(const FST fst)

//遍历森林

{

if(fst == NULL) return INFEASIBLE;

if(fst->length == 0) return ERROR;

int op = 1;

status (\*p)(BiTree, void (\*)(BiTree)) = NULL;

while(op)

{

system("cls");

printf(" 遍历方式选择\n");

printf(" 1.前序遍历\n");

printf(" 2.中序遍历\n");

printf(" 3.后序遍历\n");

printf(" 4.按层遍历\n");

printf(" 5.树形遍历\n");

printf(" 请输入您所选择遍历方式的相应序号[1-5]:");

scanf("%d", &op);

switch(op)

{

case 1: p = PreOrderTraverse; break;

case 2: p = InOrderTraverse; break;

case 3: p = PostOrderTraverse; break;

case 4: p = LevelOrderTraverse; break;

case 5: p = TreeLikeTraverse; break;

default: printf("别闹~序号还能输错？再给你个机会，回去重输~\n"); op = 6; \_getch(); break;

}

if(op != 6) op = 0;

}

printf("当前树中数据如下所示：\n");

for(int i = 0; i < fst->length; i++)

{

printf("第%d棵树：\n %s:\n", i + 1, fst->tree[i]->name);

p(fst->tree[i]->T->lchild, visit);

printf("\n\n");

}

return OK;

}

status ClearForest(const FST fst)

{

if(fst == NULL) return INFEASIBLE;

for(int i = 0; i < fst->length; i++)

{

ClearBiTree(fst->tree[i]->T); //ClearBiTree函数：参数为T时为销毁，参数为T->lchild时清空

free(fst->tree[i]);

fst->tree[i] = NULL;

}

fst->length = 0;

return OK;

}

status DestroyForest(FST &fst)

{

if(fst == NULL) return INFEASIBLE;

ClearForest(fst);

fst = NULL;

return OK;

}

//森林操作

bool IsRepeat(TElemType definition[])

//判断先序序列definition中是否有重复key值，用于创建树

{

int a = 0, b = 0;

for(int i = 0; (a = definition[i].key) != -1; i++)

{

for(int j = i + 1; (b = (definition[j].key)) != -1; j++)

{

if(a == b && a != 0)

{

return true;

}

}

}

return false;

}

status InitBiTree(BiTree &T)

{

if(T != NULL) return INFEASIBLE;

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

T->lchild = NULL;

T->rchild = NULL;

return OK;

}

status CreateBiTree(BiTree &T,DEF definition[])

{

int i=0,j;

static BiTNode \*p[100];

while (j=definition[i].pos)

{

p[j]=(BiTNode \*)malloc(sizeof(BiTNode));

p[j]->data=definition[i].data;

p[j]->lchild=NULL;

p[j]->rchild=NULL;

if (j!=1)

if (j%2) p[j/2]->rchild=p[j];

else p[j/2]->lchild=p[j];

i++;

}

T=p[1];

return OK;

}

status CreateBiTree(BiTree &T,TElemType definition[])

//根据带空枝的二叉树先根遍历序列definition构造一棵二叉树，将根节点指针赋值给T并返回OK，

//如果有相同的关键字，返回ERROR。

{

static int i = 0, flag = 0;//i为递归变量，遍历definition；flag控制IsRepeat函数只被调用一次；over处理二叉树结束标志-1

if(flag == 0 && (flag = 1) && IsRepeat(definition)) //判断是否有重复，并由于IsRepeat只需运行一次，通过flag使其成为一次性。

{

return ERROR;

}

T = (BiTree)malloc(sizeof(BiTNode)); //先对当前结点分配空间

//printf("In CreateBiTree, definition[%d].key = %d\n", i, definition[i].key);

if(definition[i].key == 0) //对于 0,null，直接回溯，将分配的空间free，T置NULL，definition后移

{

free(T);

T = NULL;

i++;

if(definition[i].key == -1)

{

i = 0; //函数结束，全局变量应恢复原状

flag = 0;

}

return OK;

}

else //数据进树

{

T->data.key = definition[i].key;

strcpy(T->data.others,definition[i].others);

}

i++; //definition后移

CreateBiTree(T->lchild, definition);

CreateBiTree(T->rchild, definition);

}

status ClearBiTree(BiTree &T)

//将二叉树设置成空，并删除所有结点，释放结点空间

{

// 请在这里补充代码，完成本关任务

/\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/

if(T == NULL)

{

return OK;

}

ClearBiTree(T->lchild);

ClearBiTree(T->rchild);

if(T->lchild == NULL && T->rchild == NULL)

{

free(T);

T = NULL;

return OK;

}

/\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/

}

int BiTreeDepth(BiTree T)

//求二叉树T的深度

{

// 请在这里补充代码，完成本关任务

/\*\*\*\*\*\*\*\*\*\* Begin \*\*\*\*\*\*\*\*\*/

if(T == NULL)

{

return 0;

}

int LeftLen = 1 + BiTreeDepth(T->lchild);

int RightLen = 1 + BiTreeDepth(T->rchild);

return max(LeftLen, RightLen);

}

BiTNode\* LocateNode(BiTree T,KeyType e)

//查找结点

{

static BiTNode \*p = NULL;

if(p != NULL && p->data.key != e) p = NULL;

//在别处多次调用(非递归)时，防止上一次使用遗留下来的p未被置NULL而导致之后使用故障

if(p != NULL) return p;

//p不为NULL时，表明已找到目标节点，直接回溯即可

if(T == NULL) return p;

//当前T为空时，直接回溯即可

if(e == T->data.key)//当找到目标节点，将目标节点地址传至p，返回即可

{

p = T;

return p;

}

LocateNode(T->lchild,e);

LocateNode(T->rchild,e);

}

status Assign(BiTree &T,KeyType e,TElemType value)

//实现结点赋值。

{

BiTNode \*p = LocateNode(T, value.key);

//p记录树中key值为value.key的结点地址用于保证赋值后的树中无key值重复项

BiTNode \*q = LocateNode(T, e);

//q记录结点编号为e的结点地址以查重以及赋值

if(!q || (p && q && p != q)) //所有错误状况

//q为空，代表树中无编号为e结点，不可赋值；p!=q且p,q均不为空，说明赋值后会有key值重复

{

return ERROR;

}

else

{

q->data.key = value.key;

strcpy(q->data.others, value.others);

return OK;

}

/\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/

}

BiTNode\* GetSibling(BiTree T,KeyType e)

//实现获得兄弟结点

{

BiTNode \*rec = NULL; //记录返回值

if(T == NULL) //当前树为空时，返回空值，表明找不到，直接回溯

{

return NULL;

}

if(T->lchild != NULL && T->lchild->data.key == e) return T->rchild; //当前结点左儿子为所找结点时，返回当前结点右儿子

else if(T->rchild != NULL && T->rchild->data.key == e) return T->lchild;//当前结点左儿子为所找结点时，返回当前结点右儿子

else //不满足以上两种，继续深入搜索

{

if(rec = GetSibling(T->lchild, e)) return rec; //如果左边找到了，返回其返回值

else if(rec = GetSibling(T->rchild, e)) return rec;//如果右边找到了，返回其返回值

else return NULL; //都没有，返回NULL

}

}

status InsertNode(BiTree &T,KeyType e,int LR,TElemType c)

//实现插入节点，在e的LR(左右)孩子处插入c

{

BiTNode \*aim = LocateNode(T, e); //查找e所在结点

BiTNode \*repeat = LocateNode(T, c.key); //判断即将插入的结点key值在原树中是否已存在

if(aim == NULL || repeat != NULL) return ERROR; //当未找到 key=e 的结点或者将插入的结点key值在原树中已存在

else

{

BiTNode \*NewNode = (BiTNode\*)malloc(sizeof(BiTNode));

NewNode->data.key = c.key;

strcpy(NewNode->data.others, c.others); //新建结点，存储新结点值

if(LR == 0) //左插

{

NewNode->rchild = aim->lchild;

NewNode->lchild = NULL;

aim->lchild = NewNode;

return OK;

}

else if(LR == 1) //右插

{

NewNode->rchild = aim->rchild;

NewNode->lchild = NULL;

aim->rchild = NewNode;

return OK;

}

else if(LR == -1) //头插

{

NewNode->rchild = T;

NewNode->lchild = NULL;

T = NewNode;

return OK;

}

else return ERROR;

}

}

status ParentNode(BiTree T, KeyType e, BiTree &p)

/\*寻找key值为e元素的双亲结点，由参数p带出函数，返回OK表示删除结点为根节点，

返回LEFT表示e为p的左孩子，返回RIGHT表示e为p的右孩子，返回ERROR表示没有此结点\*/

{

status state = ERROR;

if(T == NULL) return ERROR;

if(T->data.key == e) {p = T; return OK;}

else if(T->lchild && T->lchild->data.key == e) {p = T; return LEFT;}

else if(T->rchild && T->rchild->data.key == e) {p = T; return RIGHT;}

state = ParentNode(T->lchild, e, p);

if(state) return state;

return ParentNode(T->rchild, e, p);

}

BiTNode\* RightMostNode(BiTree T)

//寻找 T 最右端结点

{

while(T->rchild != NULL) T = T->rchild;

return T;

}

status DeleteNode(BiTree &T,KeyType e)

//删除结点。

{

BiTNode \*p = NULL, \*del = NULL;

status state = ParentNode(T, e, p);

if(state == OK) //根节点为所要删除的结点

{

if(p->lchild == NULL)//根结点没有左孩子

{

T = T->rchild;

free(p);

return OK;

}

else if(p->rchild == NULL)//根结点没有右孩子

{

T = T->lchild;

free(p);

return OK;

}

else //根结点度为二

{

T = T->lchild;

RightMostNode(T)->rchild = p->rchild;//根节点左孩子的最右置为根节点右孩子

free(p);

return OK;

}

}

else if(state == LEFT) //待删除结点为p的左孩子时

{

del = p->lchild; //记录待删除结点

if(del->lchild != NULL)//如果要删除的结点有左孩子

{

RightMostNode(del->lchild)->rchild = del->rchild; //要删除的结点的右孩子移至要删除的结点的左孩子的最右结点的右子树处

p->lchild = del->lchild;

}

else p->lchild = del->rchild;

free(del);

return OK;

}

else if(state == RIGHT) //待删除结点为p的右孩子时

{

del = p->rchild; //记录待删除结点

if(del->lchild != NULL) //待删除结点有左孩子

{

RightMostNode(del->lchild)->rchild = del->rchild;//要删除的结点的右孩子移至要删除的结点的左孩子的最右结点的右子树处

p->rchild = del->lchild;

}

else p->rchild = del->rchild;//待删除结点没有左孩子

free(del);

return OK;

}

else return ERROR; //未查找到待删除结点

}

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

//先序遍历二叉树T(递归)

{

if(T == NULL) return OK;

visit(T);

PreOrderTraverse(T->lchild, visit);

PreOrderTraverse(T->rchild, visit);

return OK;

}

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

//中序遍历二叉树T(非递归)

{

Stack s; BiTNode \*p = T;

InitStack(s);

while(p != NULL || StackEmpty(s) == FALSE)

{

if(p != NULL) //当前结点地址不为空

{

Push(s, p); //当前结点作为根节点进栈

p = p->lchild; //遍历左子树

}

else

{

Pop(s, p); //空结点双亲结点出栈

visit(p); //输出当前空结点双亲结点中的信息

p = p->rchild; //遍历右子树

} // end if...else...

} //end while

return OK;

}

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

//后序遍历二叉树T

{

if(T == NULL) return OK;

PostOrderTraverse(T->lchild, visit);

PostOrderTraverse(T->rchild, visit);

visit(T);

return OK;

}

def\* CreateDef(BiTree T, int &length) //全局变量处理完成！！！！！！！！！！！！！！！！！

//按照结点位置构造二叉树的definition

{

if(T == NULL) return NULL;

if(i >= size) //扩容

{

size += INCREMENT;

definition = (def\*)realloc(definition, size \* sizeof(def));

}

definition[i].pos = position;

definition[i].node = T;

i++;

length = i;

{

position \*= 2;

CreateDef(T->lchild, length);

position /= 2;

}

{

position = position \* 2 + 1;

CreateDef(T->rchild, length);

position /= 2;

}

return definition;

}

void DefOrder(def \*definition, int length)

//按结点位置对definition排序

{

BiTNode \*temp = NULL;

int t = 0;

for(int a = 0; a < length; a++)

{

for(int b = a + 1; b < length; b++)

{

if(definition[a].pos > definition[b].pos)

{

//结点指针交换

temp = definition[a].node;

definition[a].node = definition[b].node;

definition[b].node = temp;

//结点位置交换

t = definition[a].pos;

definition[a].pos = definition[b].pos;

definition[b].pos = t;

}

}

}

temp = NULL;

}

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

//按层遍历二叉树T

{

//全局变量处理

if(i != 0)

{

i = 0;

size = INIT\_DEF\_SIZE;

definition = (def\*)malloc(size \* sizeof(def));

position = 1;

}

int length = 0;

//按照二叉树构造其definition

definition = CreateDef(T, length);

//按结点位置排序

DefOrder(definition, length);

//输出排好序的definition

for(int a = 0; a < length; a++) visit(definition[a].node);

free(definition);//释放由alloc系列函数所分配的空间

return OK;

}

int PreSpaceNum(int level, int width)

{

int num = 1;

for(int i = 1; i < level; i++) num \*= 2;

width -= 6 \* num;

for(int i = 0; i < level; i++) width /= 2;

return width;

}

void Left(int level, int width, int a)

{

for(int j = 0; j < 2 \* PreSpaceNum(level, width) + 6 - a; j++) printf(" ");

printf("/");

}

void Right(int level, int width, int a)

{

printf("\\");

for(int j = 0; j < 2 \* PreSpaceNum(level, width) + 6 - a; j++) printf(" ");

}

void sleep(int level, int width, int a)

{

for(int j = 0; j < 2 \* PreSpaceNum(level, width) + 7 - a; j++) printf(" ");

}

status TreeLikeTraverse(BiTree T, void (\*visit)(BiTree))

{

//空树

if(T == NULL)

{

printf("\n\n");

return OK;

}

//全局变量处理

if(i != 0)

{

i = 0;

size = INIT\_DEF\_SIZE;

definition = (def\*)malloc(size \* sizeof(def));

position = 1;

}

int length = 0;

//按照二叉树构造其definition

definition = CreateDef(T, length);

//按结点位置排序

DefOrder(definition, length);

//输出当前树形

for(int i = 0; i < length; i++) printf("%d,%d,%d,%s ",i, definition[i].pos, definition[i].node->data.key, definition[i].node->data.others);

printf("\n");

int depth = BiTreeDepth(T);

int level = 1, width = 1;

for(int i = 1; i < depth; i++) width \*= 2; width \*= 6; //计算最宽

void (\*\*p)(int level, int width, int a) = NULL; //函数集合

//函数集合空间分配

p = (void (\*\*)(int level, int width, int a))malloc(2 \* width \* sizeof(void (\*)(int level, int width, int a)));

//函数集合初始化

for(int i = 0; i < 2 \* width; i++) p[i] = sleep;

for(int i = 0; i < length; i++)

{

if(definition[i].node->lchild != NULL) p[2 \* (definition[i].pos - 1)] = Left;

if(definition[i].node->rchild != NULL) p[2 \* (definition[i].pos - 1) + 1] = Right;

}

int begin = 1, end = 2;

for(int i = 1; i < level; i++) begin \*= 2; //begin为当前层数level的首结点序号

end = 2 \* begin - 1; //end为当前层数level的尾序号

for(int i = 1, a = 0; i < definition[length - 1].pos, a < length; i++)//遍历整棵树(包含设想存在的空结点)

{

//输出全部数据

while((definition[a].pos == 1) || (definition[a].pos >= begin && definition[a].pos <= end))

{

for(int i = 0; i < definition[a].pos - begin; i++)

for(int j = 0; j < 2 \* PreSpaceNum(level, width) + 6; j++)

printf(" ");

//输出实际存在的结点的数据

for(int j = 0; j < PreSpaceNum(level, width); j++) printf(" ");

printf("%2d,%-3s", definition[a].node->data.key, definition[a].node->data.others);

for(int j = 0; j < PreSpaceNum(level, width); j++) printf(" ");

//while循环动力及结束时机

a++;

if(a == length) //definition已经读完

{

break;

}

//之前没有加前置判断才采用了每次begin加一， 而这样会导致非完全二叉树读取陷入死循环

begin = definition[a].pos;//definition没读完，就读definition的下一个

if(begin > end) //即将进入下一层了 (此条件不对)

{

level++;

begin = 1;

for(int i = 1; i < level; i++) begin \*= 2; //begin为当前层数level的首结点序号

end = 2 \* begin - 1;

printf("\n");

break;

}

}

//进入下一层之前先输出斜线

if(a == length) //definition读完了(树到了最后一行)就不需要输出斜线了

break;

//此层个数count的计算

int count = 1;

for(int i = 1; i < level - 1; i++) count \*= 2;

for(int a = 1; a < PreSpaceNum(level, width) + 3; a++) //控制小斜线个数

{

int b = begin / 2;

if(!b) b = 1;

//遍历函数列表，调用该层所属函数

for(int i = 0; i < count; i++)

{

//printf("p[%d] was called!\n",2 \* (b + i - 1));//后续bug检查时使用

p[2 \* (b + i - 1)](level, width, a);

for(int j = 0; j < a - 1; j++) printf(" ");

p[2 \* (b + i - 1) + 1](level, width, a);

}

printf("\n");

}

}

free(definition);//释放由alloc系列函数所分配的空间

return OK;

}

status SaveBiTree(BiTree T, char FileName[])

//将二叉树的结点数据写入到文件FileName中

{

static FILE \*fp = fopen(FileName, "w+");

static int flag = 1;

int over = 0;

int key = 0; char others[20] = "null";

if(flag == 1)

{

over = 1;

flag = 0;

}

if(T != NULL)

{

key = T->data.key;

strcpy(others, T->data.others);

}

fprintf(fp, "%d %s ", key, others);

if(T == NULL) return OK;

SaveBiTree(T->lchild, FileName);

SaveBiTree(T->rchild, FileName);

if(over)

{

fprintf(fp, "%d %s ", -1, "null");

fclose(fp);

}

return OK;

}

status LoadBiTree(BiTree &T, char FileName[])

//读入文件FileName的结点数据，创建二叉树

{

FILE \*fp = fopen(FileName, "r");

if(fp == NULL) return ERROR;

int i = 0;

int number = 0;

TElemType definition[100];

while(fscanf(fp, "%d", &definition[i].key) != EOF && fscanf(fp, "%s", &definition[i].others) != EOF) i++;

CreateBiTree(T, definition);

fclose(fp);

return OK;

}

void visit(BiTree T)

{

printf(" %d,%s",T->data.key,T->data.others);

}

BiTree\* gather\_operate(FST& fst)

//森林操作

{

int op = 1; int j = 0;

char BiTreeName[30];

while(op)

{

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

printf(" 森林操作菜单\n");

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

printf(" 1. 初始化 6. 修改\n");

printf(" 2. 新增 7. 遍历\n");

printf(" 3. 删除 8. 销毁\n");

printf(" 4. 查找 0. 退出\n");

printf(" 5. 清空\n");

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

printf(" 请选择你的操作[0~6]:");

scanf("%d", &op);

switch(op)

{

case 1:

if(InitForest(fst) == INFEASIBLE) printf("当前森林已被初始化！\n");

else printf("初始化成功！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 2:

if(fst == NULL) printf("当前森林不存在，请先初始化！\n");

else

{

char BiTreeName[30];

int pos = 0;

printf("请输入新增树的名称：\n");

scanf("%s",BiTreeName);

printf("请输入新树种在几号位？\n");

scanf("%d", &pos);

if(AddBiTree(fst, pos, BiTreeName) == ERROR) printf("种树位置不合法！第%d位之前仍有空位！\n");

else printf("树\"%s\"成功种进森林了！\n", BiTreeName);

}

printf("----任意键继续----\n");

\_getch();

break;

case 3:

if(fst == NULL) printf("当前森林不存在，请先初始化！\n");

else if(fst->length == 0) printf("当前森林里没有树！\n");

else

{

printf("请输入想砍掉的树的名称：\n");

scanf("%s",BiTreeName);

if(RemoveBiTree(fst, BiTreeName)) printf("树\"%s\"成功被砍没了！\n",BiTreeName);

else printf("查无此树！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 4:

if(fst == NULL) printf("当前森林不存在，请先初始化！\n");

else

{

printf("请输入所要查找的树的名称：\n");

scanf("%s",BiTreeName);

if(j = LocateBiTree(fst, BiTreeName)) printf("树%s位于森林中第%d位\n", BiTreeName, j);

else printf("查无此树！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 5:

if(ClearForest(fst) == INFEASIBLE) printf("当前森林不存在，请先初始化！\n");

else printf("清空成功！森林空了！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 6:

if(fst == NULL)

{

printf("当前森林不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

else

{

printf("请输入所需更改的树的名称:\n");

scanf("%s",BiTreeName);

j = LocateBiTree(fst, BiTreeName);

if(j == 0)

{

printf("查无此树！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

else

{

int ord = 0;

printf("是否要修改树的名字?(是，摁1 / 否，摁0)：");

scanf("%d", &ord);

if(ord)

{

printf("输入新名字：");

scanf("%s", fst->tree[j - 1]->name);

printf("修改成功！\n");

printf("继续修改数据吗？(是，摁1 / 否，摁0):");

scanf("%d", &ord);

if(!ord) break;

}

return &(fst->tree[j - 1]->T);

}

}

case 7:

if((j = ForestTraverse(fst)) == 0) printf("当前森林一棵树也没有！\n");

else if(j == INFEASIBLE) printf("当前森林不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 8:

if(DestroyForest(fst) == INFEASIBLE) printf("啧..你还是不记得初始化！\n");

else printf("销毁成功！森林没了！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 0:

break;

default:

printf("指令错误！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

}

return NULL;

}

## 实验三.cpp

#include"fun.h"

int main(){

BiTree T = NULL; //树

FST fst = NULL; //森林

int op = 1; //操作指令

int flag = 0; //森林操作完成标志

DEF definition[100]; //输入树中的数据载体

int e = 0; //键值载体

BiTNode \*node = NULL; //结点载体

BiTree \*T\_ope = NULL, temp = NULL;//森林操作树的载体

TElemType value; //结点数据载体

int LR; //插入节点：定左右

while(op){

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

printf(" 基于二叉链表的二叉树操作菜单\n");

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

printf(" 1. 初始化树 8. 获得兄弟结点 15. 树形遍历\n");

printf(" 2. 初始化输入 9. 插入节点 16. 存档\n");

printf(" 3. 摧毁树 10. 删除结点 17. 读档\n");

printf(" 4. 清空树 11. 前序遍历 18. 森林管理\n");

printf(" 5. 树的深度 12. 中序遍历 0. 退出\n");

printf(" 6. 结点查找 13. 后序遍历\n");

printf(" 7. 结点赋值 14. 按层遍历\n");

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

if(op)

printf(" 请选择你的操作[0~17]:");

scanf("%d",&op);

switch(op){

case 1:

if(InitBiTree(T) == OK) printf("初始化成功！\n");

else printf("当前的树里已经有点东西了~\n");

printf("----任意键继续----\n");

\_getch();

break;

case 2:

if(T == NULL) printf("初始化都没做的树不配存数据！\n");

else if(T->lchild != NULL) printf("当前的树里已经有点东西了~不可以进行初始化输入了\n");

else{

printf("输入数据：(格式：满二叉树中的位置序号+键值+数据)，位置序号为0时结束\n");

do {

scanf("%d%d%s",&definition[i].pos,&definition[i].data.key,definition[i].data.others);

} while (definition[i++].pos);

CreateBiTree(T->lchild,definition);

printf("输入成功！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 3:

if(T == NULL) printf("想销毁？先初始化一下再来~\n");

else if(flag == 1) printf("此处不支持销毁树，如需销毁，请选择森林管理中的“删除”功能！\n");

else{

ClearBiTree(T);

printf("销毁成功！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 4:

if(T == NULL) printf("又忘了初始化？！\n");

else{

ClearBiTree(T->lchild);

printf("清空成功！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 5:

if(T == NULL) printf("这里一棵树都不存在哦~\n");

else printf("当前树的深度为%d\n", BiTreeDepth(T->lchild));

printf("----任意键继续----\n");

\_getch();

break;

case 6:

if(T == NULL) printf("找啥找？树都没有\n");

else{

printf("输入键值：");

scanf("%d", &e);

node = LocateNode(T->lchild, e);

if(node == NULL) printf("当前树里面没有key值为%d的结点\n", e);

else printf("当前结点键值为:%d, 数据为:%s\n", node->data.key, node->data.others);

}

printf("----任意键继续----\n");

\_getch();

break;

case 7:

if(T == NULL) printf("当前树不存在！\n");

else{

printf("输入所要修改的结点键值：");

scanf("%d", &e);

node = LocateNode(T->lchild, e);

if(node == NULL) printf("当前树里面没有key值为%d的结点\n", e);

else

{

printf("请输入输入更改后的结点键值：");

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

printf("请输入更改后的数据：");

scanf("%s", value.others);

if(Assign(T->lchild, e, value) == ERROR)

printf("更改键值出错！因为原树中已存在其他结点的键值为%d！\n", value.key);

else printf("更改成功！\n");

}

}

printf("----任意键继续----\n");

\_getch();

break;

case 8:

if(T == NULL) printf("答应我，先初始化好嘛~\n");

else

{

printf("输入当前键值：");

scanf("%d", &e);

node = LocateNode(T->lchild, e);

if(node == NULL) printf("当前树里面没有key值为%d的结点\n", e);

else

{

node = GetSibling(T->lchild, e);

if(node == NULL) printf("这个结点没有兄弟~\n");

else printf("这个结点的兄弟结点的键值为：%d, 数据为：%s\n", node->data.key, node->data.others);

}

}

printf("----任意键继续----\n");

\_getch();

break;

case 9:

if(T == NULL) printf("哼！又不记得初始化\n");

else

{

printf("想给哪个结点添个孩子？(输入相应结点键值即可)\n");

scanf("%d", &e);

node = LocateNode(T->lchild, e);

if(node == NULL) printf("当前树里面没有key值为%d的结点\n", e);

else{

printf("想添个左孩子还是右孩子？(左孩子输0，右孩子输1，当祖宗输-1)\n");

scanf("%d", &LR);

printf("输入新增结点键值：");

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

printf("输入新增结点数据：");

scanf("%s", value.others);

if(InsertNode(T->lchild, e, LR, value) == ERROR) printf("新增键值出错！因为原树中已存在其他结点的键值为%d！\n", value.key);

else printf("插入节点成功！\n");

}

}

printf("----任意键继续----\n");

\_getch();

break;

case 10:

if(T == NULL) printf("请先初始化一棵树！\n");

else{

printf("输入待删除结点的键值：");

scanf("%d", &e);

if(DeleteNode(T, e) == ERROR) printf("当前树里面没有key值为%d的结点\n", e);

else printf("删除成功！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 11:

if(T == NULL) printf("先去初始化好嘛~\n");

else

{

printf("前序遍历当前树中结点信息如下：\n");

PreOrderTraverse(T->lchild, visit);

printf("\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 12:

if(T == NULL) printf("先去初始化好嘛~\n");

else

{

printf("中序遍历当前树中结点信息如下：\n");

InOrderTraverse(T->lchild, visit);

printf("\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 13:

if(T == NULL) printf("先去初始化好嘛~\n");

else

{

printf("后序遍历当前树中结点信息如下：\n");

PostOrderTraverse(T->lchild, visit);

printf("\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 14:

if(T == NULL) printf("先去初始化好嘛~\n");

else

{

printf("按层遍历当前树中结点信息如下：\n");

LevelOrderTraverse(T->lchild, visit);

printf("\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 15:

if(T == NULL) printf("先去初始化好嘛~\n");

else

{

printf("当前树就长下面这个样子：\n");

TreeLikeTraverse(T->lchild, visit);

printf("\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 16:

if(T == NULL) printf("您还未初始化！\n");

else{

char SaveFile[40];

printf("输入存档文件路径(输入0选择默认路径):\n");

scanf("%s",SaveFile);

if(!strcmp(SaveFile,"0")) strcpy(SaveFile, "D:\\BiTree\_save.dat");

SaveBiTree(T->lchild, SaveFile);

printf("已将数据存入 %s\n",SaveFile);

}

printf("----任意键继续----\n");

\_getch();

break;

case 17:

if(T == NULL) printf("树不存在树不存在树不存在！\n");

else if(T->lchild != NULL) printf("当前这颗树非空！不准你读档！\n");

else

{

char LoadFile[40];

printf("输入读档文件路径(输入0选择默认路径):\n");

scanf("%s",LoadFile);

if(!strcmp(LoadFile,"0")) strcpy(LoadFile, "D:\\BiTree\_save.dat");

if(LoadBiTree(T->lchild, LoadFile) == ERROR) printf("存档文件%s无法打开！\n", LoadFile);

else printf("数据已从 %s 中读出\n",LoadFile);

}

printf("----任意键继续----\n");

\_getch();

break;

case 18:

if(flag == 1) //修改状态，不可再次选择线性表集合操作

{

printf("当前有未结束的线性表修改，请先结束前一修改！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

T\_ope = gather\_operate(fst);

if(T\_ope != NULL)//只要选择了集合操作中的修改，就将主界面线性表置换成集合中所选线性表

{

temp = T; //temp暂存主界面线性表数据

T = \*T\_ope; //主界面后台线性表置换为集合内所选线性表

flag = 1; //系统状态置为集合操作态

}

break;

case 0:

break;

default:

printf("操作指令有误！\n");

printf("----任意键继续----\n");

\_getch();

break;

}//end of switch

if(flag && op == 0) //flag判断当前是否在线性表集合操作中，此时退出返回主界面

{

T = temp; //返还原始主界面线性表数据

op = 1; //保持不退出while

flag = 0; //状态置为初始界面

printf("原主界面的树中数据已载入！\n");

\_getch();

}

}//end of while

printf("欢迎下次再使用本系统！\n");

printf("----任意键结束----\n");

\_getch();

return 0;

}//end of main()

# 附录D 基于邻接表图实现的源程序

## def\_Graph.h

#pragma once

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

#define FAIL -3

#define MAX\_VERTEX\_NUM 20

#define INIT 20

#define INCREMENT 10

typedef int status;

typedef int KeyType;

typedef enum { DG, DN, UDG, UDN } GraphKind;

typedef struct {

KeyType key;

char others[20];

} VertexType; //顶点类型定义

typedef struct ArcNode { //表结点类型定义

int adjvex; //顶点位置编号

struct ArcNode\* nextarc; //下一个表结点指针

} ArcNode;

typedef struct VNode { //头结点及其数组类型定义

VertexType data; //顶点信息

ArcNode\* firstarc; //指向第一条弧

} VNode, AdjList[MAX\_VERTEX\_NUM];

typedef struct { //邻接表的类型定义

AdjList vertices; //头结点数组

int vexnum, arcnum; //顶点数、弧数

GraphKind kind; //图的类型

} ALGraph;

typedef struct {

char name[30];

ALGraph G;

}Graph, \*G;

typedef struct { //多图定义

G \*elem;

int length;

int size;

}ALGraphs;

## tool.h

#pragma once

int LocateNode(ALGraph G, KeyType key);

//查找键值为key的结点

bool IsRepeat(ArcNode\* arc, int num);

//在当前结点及其后续链表中是否存在num

bool IsRepeat(ALGraph G, KeyType key);

//图G中查找键值与key相同的点

status DeleteNode(ArcNode\*& arc, int del);

//删去v中序号为del结点

status visitNode(ALGraph G, ArcNode\*\* Q, int& head, int& tail, int\* rec, void (\*visit)(VertexType));

//广度优先遍历的子函数

void visit(VertexType v);

int LocateNode(ALGraph G, KeyType key)

//查找键值为key的结点

{

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

{

if (G.vertices[i].data.key == key)

return i + 1;

}

return ERROR;

}

bool IsRepeat(ArcNode\* arc, int num)

//在当前结点及其后续链表中是否存在num

{

while (arc != NULL)

{

if (arc->adjvex == num)

return true;

arc = arc->nextarc;

}

return false;

}

bool IsRepeat(ALGraph G, KeyType key)

//图G中查找键值与key相同的点

{

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

{

if (key == G.vertices[i].data.key)

return true;

}

return false;

}

status DeleteNode(ArcNode\*& arc, int del)

//删去v中序号为del结点

{

ArcNode\* p = arc;

//如果第一个结点的序号即为del

if (arc->adjvex == del)

{

arc = arc->nextarc;

free(p);

return OK;

}

//第一个结点的序号不为del

ArcNode\* del\_p = NULL;

while (p->nextarc != NULL)

{

if (p->nextarc->adjvex == del)

{

del\_p = p->nextarc;

p->nextarc = p->nextarc->nextarc;

return OK;

}

p = p->nextarc;

}

}

status visitNode(ALGraph G, ArcNode\*\* Q, int& head, int& tail, int\* rec, void (\*visit)(VertexType))

//广度优先遍历的子函数

{

while (head != tail)

{

//访问当前结点并出队

ArcNode\* p = Q[head++];

if(p == NULL) return OK;

if (!rec[p->adjvex])

{

visit(G.vertices[p->adjvex].data);

rec[p->adjvex] = 1;

}

//进队

while (p != NULL)

{

if (!rec[p->adjvex])

{

Q[tail++] = p;

}

p = p->nextarc;

}

}

return OK;

}

void visit(VertexType v)

{

printf(" %d %s", v.key, v.others);

}

bool HavePython()

{

//获取系统变量信息

char \*ret = NULL, \*buf = NULL, pack[80], pathSub[80];

extern char \*\*environ;

char \*\*env=environ;

bool result = false;

FILE \*fp = NULL;

//查找系统变量中PATH行内文件夹中是否有python

ret = getenv("PATH");

buf = ret;

if(ret != NULL)

{

while((ret = strstr(ret, ";")) != NULL)

{

//文件名储存数组置0

for(int i = 0; i < 80; i++) pack[i] = '\0', pathSub[i] = '\0';

//strcpy

for(int i = 0; buf != ret; i++, buf++)

{

pack[i] = \*buf;

pathSub[i] = \*buf;

}

strcat(pack, "python.exe");

//检测是否有python.exe

if((fp = fopen(pack, "r")) != NULL)

{

FILE \*f = fopen("statement.txt", "r");

if(f == NULL)

{

fclose(f);

FILE \*f = fopen("statement.txt", "w+");

printf("即将尝试为您安装相关python库(networkx, matplotlib.pyplot)，如已安装请无视命令行报错，可能会失败，谢谢！\n");

printf("P.S.个人测试时所用python版本为3.9，且安装命令成功运行\n");

char \*cmd = (char\*)calloc(200, sizeof(char));

strcat(cmd, "cd ");

strcat(cmd, pathSub);

strcat(cmd, "Scripts");

strcat(cmd, "\n");

strcat(cmd, "pip install networkx\n");

strcat(cmd, "pip install matplotlib.pyplot\n");

system(cmd);

}

result = true;

fclose(fp);

}

//判断是否读完

if(strlen(ret) > 2)

{

//跳过所有 ';'

ret++;

buf++;

}

else

break;

}

}

return result;

}

## func.h

#pragma once

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<stdio.h>

#include<stdlib.h>

#include"def\_Graph.h"

#include<string.h>

#include<unistd.h>

#include"tool.h"

#include<conio.h>

status CreateGraph(ALGraph &G);

status CreateGraph(ALGraph& G, VertexType V[], KeyType VR[][2]);

/\*根据V和VR构造图T并返回OK，如果V和VR不正确，返回ERROR;如果有相同的关键字，返回ERROR。\*/

status DestroyGraph(ALGraph& G);

/\*销毁无向图G\*/

int LocateVex(ALGraph G, KeyType u);

//根据u在图G中查找顶点，查找成功返回位序，否则返回-1；

status PutVex(ALGraph& G, KeyType u, VertexType value);

//根据u在图G中查找顶点，查找成功将该顶点值修改成value，返回OK；

//如果查找失败或关键字不唯一，返回ERROR

int FirstAdjVex(ALGraph G, KeyType u);

//根据u在图G中查找顶点，查找成功返回顶点u的第一邻接顶点位序，否则返回-1；

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

//根据u在图G中查找顶点，查找成功返回顶点v的邻接顶点相对于w的下一邻接顶点的位序，查找失败返回-1；

status InsertVex(ALGraph& G, VertexType v);

//在图G中插入顶点v，成功返回OK,否则返回ERROR

status DeleteVex(ALGraph& G, KeyType v);

//在图G中删除关键字v对应的顶点以及相关的弧，成功返回OK,否则返回ERROR

status InsertArc(ALGraph& G, KeyType v, KeyType w);

//在图G中增加弧<v,w>，成功返回OK,否则返回ERROR

status DeleteArc(ALGraph& G, KeyType v, KeyType w);

//在图G中删除弧<v,w>，成功返回OK,否则返回ERROR

status DFSTraverse(ALGraph G, void (\*visit)(VertexType));

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

status BFSTraverse(ALGraph G, void (\*visit)(VertexType));

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

status GraphLikeTraverse(ALGraph G, void(\*visit)(VertexType));

//输出图形

status LinearTableTraverse(ALGraph G, void(\*visit)(VertexType));

//输出邻接表

status SaveGraph(ALGraph G, char FileName[]);

//将图的数据写入到文件FileName中

status LoadGraph(ALGraph& G, char FileName[]);

//读入文件FileName的图数据，创建图的邻接表

/\*多图操作\*/

ALGraph\* gather\_operate(ALGraphs& GS);

status CreateGraph(ALGraph &G)

//输入创建图

{

if (G.vexnum != 0) return INFEASIBLE;

VertexType V[30];

KeyType VR[100][2];

int i = 0;

G.arcnum = G.vexnum = 0;

printf("请输入如下格式内容:\n\"键值 名称 键值 名称 ... -1 nil 连接键值 连接键值 ... -1 -1\"\n");

do {

scanf("%d%s", &V[i].key, V[i].others);

} while (V[i++].key != -1);

i = 0;

do {

scanf("%d%d", &VR[i][0], &VR[i][1]);

} while (VR[i++][0] != -1);

if (CreateGraph(G, V, VR) == ERROR) return ERROR;

else return OK;

}

status CreateGraph(ALGraph& G, VertexType V[], KeyType VR[][2])

/\*根据V和VR构造图T并返回OK，如果V和VR不正确，返回ERROR

如果有相同的关键字，返回ERROR。\*/

{

//判断溢出

for (int i = 0; V[i].key != -1; i++)

{

if (i >= MAX\_VERTEX\_NUM)

return ERROR;

}

//V[]内查重

for (int i = 0; V[i].key != -1; i++)

{

for (int j = i + 1; V[j].key != -1; j++)

{

if (V[i].key == V[j].key)

return ERROR;

}

}

//数据域赋值

G.vexnum = 0;

for (int i = 0; V[i].key != -1; i++)

{

G.vertices[i].data.key = V[i].key;

strcpy(G.vertices[i].data.others, V[i].others);

G.vertices[i].firstarc = NULL;

G.vexnum++;

}

//指针域链接

int count = 0; //记录图的度degree，每链接一次count++，最终count为degree的两倍

G.arcnum = 0;

for (int i = 0; VR[i][0] != -1; i++)

{

int root = LocateNode(G, VR[i][0]) - 1;

int branch = LocateNode(G, VR[i][1]) - 1;

if (root == -1 || branch == -1) //VR中包含V中不存在的点，错误

return ERROR; //此函数return ERROR之后，下一步应将数据域所赋的值free

//root后链branch

if (!IsRepeat(G.vertices[root].firstarc, branch) && root != branch)

{

count++;

ArcNode\* in = (ArcNode\*)malloc(sizeof(ArcNode));

in->nextarc = G.vertices[root].firstarc;

G.vertices[root].firstarc = in;

in->adjvex = branch;

}

//branch后链root

if (!IsRepeat(G.vertices[branch].firstarc, root) && root != branch)

{

count++;

ArcNode\* back = (ArcNode\*)malloc(sizeof(ArcNode));

back->nextarc = G.vertices[branch].firstarc;

G.vertices[branch].firstarc = back;

back->adjvex = root;

}

}

G.arcnum = count / 2;

return OK;

/\*\*\*\*\*\*\*\*\*\* End \*\*\*\*\*\*\*\*\*\*/

}

status DestroyGraph(ALGraph& G)

/\*销毁无向图G\*/

{

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

{

//释放每个结点后的链表

ArcNode\* p = G.vertices[i].firstarc, \* del = NULL;

while (p != NULL)

{

del = p;

p = p->nextarc;

free(del);

del = NULL;

}

}

G.vexnum = 0;

G.arcnum = 0;

return OK;

}

int LocateVex(ALGraph G, KeyType u)

//根据u在图G中查找顶点，查找成功返回位序，否则返回-1；

{

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

{

if (G.vertices[i].data.key == u)

return i;

}

return -1;

}

status PutVex(ALGraph& G, KeyType u, VertexType value)

//根据u在图G中查找顶点，查找成功将该顶点值修改成value，返回OK；

//如果查找失败或关键字不唯一，返回ERROR

{

if (IsRepeat(G, value.key)) return ERROR;

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

{

if (G.vertices[i].data.key == u)

{

G.vertices[i].data.key = value.key;

strcpy(G.vertices[i].data.others, value.others);

return OK;

}

}

return ERROR;

}

int FirstAdjVex(ALGraph G, KeyType u)

//根据u在图G中查找顶点，查找成功返回顶点u的第一邻接顶点位序，否则返回-1；

{

int pos = LocateVex(G, u);

if (pos == -1) return -1;

return G.vertices[pos].firstarc->adjvex;

}

int NextAdjVex(ALGraph G, KeyType v, KeyType w)

//根据u在图G中查找顶点，查找成功返回顶点v的邻接顶点相对于w的下一邻接顶点的位序，查找失败返回-1；

{

int pos = LocateVex(G, v);

if (pos == -1) return -1;

ArcNode\* p = G.vertices[pos].firstarc;

while (p != NULL)

{

int pos = p->adjvex;

p = p->nextarc;

if (G.vertices[pos].data.key == w)

break;

}

if (p == NULL) return -1; //没找到 w 或者 w没有下一个

return p->adjvex;

}

status InsertVex(ALGraph& G, VertexType v)

//在图G中插入顶点v，成功返回OK,否则返回ERROR

{

if (G.vexnum == MAX\_VERTEX\_NUM || IsRepeat(G, v.key)) return ERROR;

G.vertices[G.vexnum].data.key = v.key;

strcpy(G.vertices[G.vexnum].data.others, v.others);

G.vertices[G.vexnum].firstarc = NULL;

G.vexnum++;

return OK;

}

status DeleteVex(ALGraph& G, KeyType v)

//在图G中删除关键字v对应的顶点以及相关的弧，成功返回OK,否则返回ERROR

{

int aim = LocateVex(G, v);

//未找到key值为v的点，或者G中只有一个点

if (aim == -1 || G.vexnum == 1) return ERROR;

//删去该点的所有边

ArcNode\* p = G.vertices[aim].firstarc, \* del = NULL;

while (p != NULL)

{

//清除：与待删除结点 有关联的点 与待删除结点之间的边

//操作对象：与待删除结点关联的点内的链表

DeleteNode(G.vertices[p->adjvex].firstarc, aim);

//删除：待删除结点后链表中已处理完的结点

//操作对象：待删除结点内的链表

del = p;

p = p->nextarc;

free(del);

//每次循环，边数减 1

G.arcnum--;

}

G.vertices[aim].firstarc = NULL;

/\*\*\*待删除结点的链表处理完毕\*\*\*/

//从图中剔除该点

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

{

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

}

G.vexnum--;

//更改aim后的所有点在后置链表中的序号信息

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

{

//遍历第i个结点后所有链表

ArcNode\* p = G.vertices[i].firstarc;

while (p != NULL)

{

if (p->adjvex >= aim)

p->adjvex--;

p = p->nextarc;

}

}

return OK;

}

status InsertArc(ALGraph& G, KeyType v, KeyType w)

//在图G中增加弧<v,w>，成功返回OK,否则返回ERROR

{

//顶点数小于2的图直接pass

if (G.vexnum < 2) return ERROR;

//v和 w不同时存在于当前图，pass

int v\_pos = LocateVex(G, v), w\_pos = LocateVex(G, w);

if (v\_pos == -1 || w\_pos == -1) return ERROR;

//v和 w同时存在于当前图，但已有链接，直接pass

ArcNode\* p = G.vertices[v\_pos].firstarc;

while (p != NULL)

{

if (p->adjvex == w\_pos)

return INFEASIBLE;

p = p->nextarc;

}

//v后加 w

ArcNode\* new\_w = (ArcNode\*)malloc(sizeof(ArcNode));

new\_w->adjvex = w\_pos;

new\_w->nextarc = G.vertices[v\_pos].firstarc;

G.vertices[v\_pos].firstarc = new\_w;

//w后加 v

ArcNode\* new\_v = (ArcNode\*)malloc(sizeof(ArcNode));

new\_v->adjvex = v\_pos;

new\_v->nextarc = G.vertices[w\_pos].firstarc;

G.vertices[w\_pos].firstarc = new\_v;

G.arcnum++;

return OK;

}

status DeleteArc(ALGraph& G, KeyType v, KeyType w)

//在图G中删除弧<v,w>，成功返回OK,否则返回ERROR

{

//顶点数小于2的图直接pass

if (G.vexnum < 2) return ERROR;

//v和 w不同时存在于当前图，pass

int v\_pos = LocateVex(G, v), w\_pos = LocateVex(G, w);

if (v\_pos == -1 || w\_pos == -1) return ERROR;

//v和 w同时存在于当前图，但没有链接，直接pass

ArcNode\* p = G.vertices[v\_pos].firstarc;

bool flag = false;

while (p != NULL)

{

if (p->adjvex == w\_pos)

flag = true;

p = p->nextarc;

}

if (!flag) return INFEASIBLE;

//v里删除 w

DeleteNode(G.vertices[v\_pos].firstarc, w\_pos);

//w里删除 v

DeleteNode(G.vertices[w\_pos].firstarc, v\_pos);

G.arcnum--;

return OK;

}

status DFSTraverse(ALGraph G, void (\*visit)(VertexType))

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

{

static int\* rec = (int\*)calloc(G.vexnum, sizeof(int));//记录遍历状况

static int pos = 0; //子分支的首结点

static int head = 0; //不同连通分支的首结点

if(G.vexnum == 0) return OK;

while (true)

{

while (true)

{

int this\_pos = pos; //记录当前位置

ArcNode\* p = G.vertices[this\_pos].firstarc;

//part 1：访问当前结点 (与part 2互不影响)

if (!rec[this\_pos])

{

visit(G.vertices[this\_pos].data);

rec[this\_pos] = 1; //记录此结点已被访问

}

//part 2：找到链表中最浅的尚未被访问的点赋值到 p

while (p != NULL)

{

if (!rec[p->adjvex])

break;

p = p->nextarc;

}

//如果 p == NULL，说明此链表中所有结点已被访问完毕，返回即可

//并且仅当当前整个连通分支还未完全访问完，即回溯时才用return

if (p == NULL && this\_pos != head) return OK;

//当前连通分支访问完成，查找是否还有其它分支

else if (p == NULL && this\_pos == head)

{

int flag = 0;

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

{

//若还有其他分支，head变为该分支首结点，pos置为 head

if (!rec[i])

{

head = i;

pos = head;

flag = 1;

break;

}

}

//还有其他连通分支

if (flag)

break;

else

{

//全局变量处理

free(rec);

rec = (int\*)calloc(G.vexnum, sizeof(int));

pos = 0;

head = 0;

return OK;

}

}

//否则就深度优先搜索pos位置结点

pos = p->adjvex;

DFSTraverse(G, visit);

//一条路跑完了，回来了需将全局变量 pos置回 this\_pos

pos = this\_pos;

}

}

}

status BFSTraverse(ALGraph G, void (\*visit)(VertexType))

{

ArcNode\*\* Q = (ArcNode\*\*)calloc(G.vexnum, sizeof(ArcNode\*));

int\* rec = (int\*)calloc(G.vexnum, sizeof(int));

int head = 0, tail = 0;

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

{

if (!rec[i])

{

Q[tail++] = G.vertices[i].firstarc;

visit(G.vertices[i].data);

rec[i] = 1;

visitNode(G, Q, head, tail, rec, visit);

}

}

free(Q);

free(rec);

return OK;

}

status GraphLikeTraverse(ALGraph G, void (\*visit)(VertexType))

{

if(G.vexnum == 0) return OK;

if(HavePython())

{

SaveGraph(G, "D:\\drawing.txt");

system("python GraphDrawing\\drawGraph.py");

return OK;

}

else

{

printf("抱歉,检测到当前系统内未安装python,运行失败!\n");

}

}

status LinearTableTraverse(ALGraph G, void(\*visit)(VertexType))

{

int i = 0;

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

{

ArcNode\* p = G.vertices[i].firstarc;

printf("%d %s", G.vertices[i].data.key, G.vertices[i].data.others);

while (p)

{

printf(" %d", p->adjvex);

p = p->nextarc;

}

printf("\n");

}

return OK;

}

status SaveGraph(ALGraph G, char FileName[])

//将图的数据写入到文件FileName中

{

FILE\* fp = fopen(FileName, "w+");

if(G.vexnum > 1000)

{

printf("结点个数过大!存档存在危险!程序强行退出!\n");

exit -1;

}

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

{

fprintf(fp, "%d %s", G.vertices[i].data.key, G.vertices[i].data.others);

ArcNode\* p = G.vertices[i].firstarc;

while (p != NULL)

{

fprintf(fp, " %d", p->adjvex);

p = p->nextarc;

}

fprintf(fp, "\n");

}

fclose(fp);

return OK;

}

status LoadGraph(ALGraph& G, char FileName[])

//读入文件FileName的图数据，创建图的邻接表

{

FILE\* fp = fopen(FileName, "r");

if(fp == NULL) return ERROR;

int ch = 0;

G.vexnum = G.arcnum = 0;

for (int i = 0; ch != EOF; i++)

{

if(i >= MAX\_VERTEX\_NUM)

{

DestroyGraph(G);

return INFEASIBLE;

}

//读取部分

fscanf(fp, "%d %s", &G.vertices[i].data.key, G.vertices[i].data.others);

G.vexnum++;

ch = fgetc(fp); //读掉一个空格或换行

if (ch != '\n')

{

G.vertices[i].firstarc = (ArcNode\*)malloc(sizeof(ArcNode));

ArcNode\* p = G.vertices[i].firstarc;

//读一个空格，读一个数据，读到换行直接退，此行的链表构建完成

while ((ch != '\n') && (ch != EOF) && fscanf(fp, "%d", &ch) != EOF)

{

p->adjvex = ch;

ch = fgetc(fp);

if (ch == '\n' || ch == EOF)

p->nextarc = NULL;

else

{

p->nextarc = (ArcNode\*)malloc(sizeof(ArcNode));

p = p->nextarc;

p->nextarc = NULL;

}

G.arcnum++;

}

p = NULL;

}

else G.vertices[i].firstarc = NULL;

}

--G.vexnum;

G.arcnum /= 2;

fclose(fp);

return OK;

}

/\*多图操作\*/

status InitALGraphs(ALGraphs& GS)

//初始化多图(以顺序线性表形式定义)

{

if (GS.elem != NULL) return INFEASIBLE;

GS.elem = (G\*)calloc(INIT, sizeof(G));

GS.length = 0;

GS.size = INIT;

return OK;

}

status AddGraph(ALGraphs& GS, int i, char GraphName[])

//第i位加入名为GraphName的图

{

if (GS.elem == NULL) return INFEASIBLE;

if (i > GS.length + 1 || i < 1) return ERROR;

for(int i = 0; i < GS.length; i++)

{

if(!strcmp(GraphName, GS.elem[i]->name))

return FAIL;

}

//建立新图

G p = (Graph\*)malloc(sizeof(Graph));

int j = 0; char ord = '\0';

strcpy(p->name, GraphName); //图命名 && 图初始化

p->G.arcnum = p->G.vexnum = 0;

printf("是否需要立即输入图中点的信息?(Y/N)\n");

scanf("\n%c", &ord);

if(ord == 'y' || ord == 'Y')

{

j = CreateGraph(p->G);

if(j == ERROR) printf("输入数据有误，部分顶点键值重复，新增失败！\n");

}

if (GS.length >= GS.size) //扩容

{

GS.size += INCREMENT;

GS.elem = (G\*)realloc(GS.elem, GS.size \* sizeof(G));

}

for (int j = GS.length; j >= i; j--) //移位置

{

GS.elem[j] = GS.elem[j - 1];

}

GS.elem[i - 1] = p;

GS.length++;

return OK;

}

status RemoveGraph(ALGraphs& GS, char GraphName[])

//删去名为GraphName的图

{

if (GS.elem == NULL) return INFEASIBLE;

for (int i = 0; i < GS.length; i++)

{

if (!strcmp(GraphName, GS.elem[i]->name))

{

DestroyGraph(GS.elem[i]->G);

free(GS.elem[i]);

for (int a = i + 1; a < GS.length; a++)

GS.elem[a - 1] = GS.elem[a];

GS.length--;

return OK;

}

}

return ERROR;

}

int LocateGraph(ALGraphs GS, char GraphName[])

//查找当前森林中名为GraphName的树

{

if (GS.elem == NULL) return INFEASIBLE;

for (int i = 0; i < GS.length; i++)

{

if (!strcmp(GraphName, GS.elem[i]->name)) return i + 1;

}

return ERROR;

}

status ALGraphsTraverse(ALGraphs GS)

//遍历多图

{

if (GS.elem == NULL) return INFEASIBLE;

if (GS.length == 0) return ERROR;

int op = 1;

status(\*p)(ALGraph G, void (\*visit)(VertexType)) = NULL;

while (op)

{

system("cls");

printf(" 遍历方式选择\n");

printf(" 1.深度优先遍历\n");

printf(" 2.广度优先遍历\n");

printf(" 3.图形遍历\n");

printf(" 4.邻接表遍历\n");

printf(" 请输入您所选择遍历方式的相应序号[1-4]:");

scanf("%d", &op);

switch (op)

{

case 1: p = DFSTraverse; break;

case 2: p = BFSTraverse; break;

case 3: p = GraphLikeTraverse; break;

case 4: p = LinearTableTraverse; break;

default: printf("别闹~序号还能输错？再给你个机会，回去重输~\n"); op = 6; \_getch(); break;

}

if (op != 6) op = 0;

}

printf("当前集合中数据如下所示：\n");

for (int i = 0; i < GS.length; i++)

{

printf("第%d个图：\n %s:\n", i + 1, GS.elem[i]->name);

p(GS.elem[i]->G, visit);

printf("\n\n");

}

return OK;

}

status ClearALGraphs(ALGraphs& GS)

{

if (GS.elem == NULL) return INFEASIBLE;

for (int i = 0; i < GS.length; i++)

{

DestroyGraph(GS.elem[i]->G); //ClearBiTree函数：参数为T时为销毁，参数为T->lchild时清空

free(GS.elem[i]);

GS.elem[i] = NULL;

}

GS.length = 0;

return OK;

}

status DestroyALGraphs(ALGraphs& GS)

{

if (GS.elem == NULL) return INFEASIBLE;

ClearALGraphs(GS);

GS.size = 0;

free(GS.elem);

GS.elem = NULL;

return OK;

}

ALGraph\* gather\_operate(ALGraphs& GS)

{

int op = 1; int j = 0; int i = 0;

char GraphName[30];

while (op)

{

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

printf(" 多图集合操作菜单\n");

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

printf(" 1. 初始化 5. 修改\n");

printf(" 2. 新增 6. 遍历\n");

printf(" 3. 删除 7. 清空集合\n");

printf(" 4. 查找 0. 退出\n");

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

printf(" 请选择你的操作[0~7]:");

scanf("%d", &op);

switch (op)

{

case 1:

if (InitALGraphs(GS) == INFEASIBLE) printf("当前图集合已被初始化！\n");

else printf("初始化成功！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 2:

if (GS.elem == NULL)

{

printf("当前图集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("请输入新增图的名称：\n");

scanf("%s", GraphName);

printf("新增到第几位?");

scanf("%d", &i);

if ((j = AddGraph(GS, i, GraphName)) == ERROR) printf("插入位置非法，添加失败！\n");

else if (j == FAIL) printf("当前名称已被其他图占用!\n");

else printf("图\"%s\"存储成功！\n", GraphName);

printf("----任意键继续----\n");

\_getch();

break;

case 3:

if (GS.elem == NULL)

{

printf("当前图集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("请输入所要删除的图名称：\n");

scanf("%s", GraphName);

if (RemoveGraph(GS, GraphName) == OK)

{

printf("图\"%s\"删除成功！\n", GraphName);

}

else

{

printf("查无此图！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 4:

if (GS.elem == NULL)

{

printf("当前图集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("请输入所要查找的图名称：\n");

scanf("%s", &GraphName);

if (j = LocateGraph(GS, GraphName))

{

printf("图\"%s\"位于集合中第%d位\n", GraphName, j);

}

else

{

printf("查无此表！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 5:

if (GS.elem == NULL)

{

printf("当前图集合不存在，请先初始化！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

printf("请输入所需更改的图的名称:\n");

scanf("%s", GraphName);

j = LocateGraph(GS, GraphName);

if (j == 0)

{

printf("查无此图！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

else

{

return &(GS.elem[j - 1]->G);

}

case 6:

if ((j = ALGraphsTraverse(GS)) == ERROR) printf("当前图集合为空！\n");

else if (j == INFEASIBLE) printf("当前图集合不存在！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 7:

if(ClearALGraphs(GS) == INFEASIBLE) printf("当前集合尚未被初始化!\n");

else printf("清空成功!\n");

printf("----任意键继续----\n");

\_getch();

break;

case 0:

break;

default:

printf("指令错误！\n");

printf("----任意键继续----\n");

\_getch();

}

}

return NULL;

}

/\*多图操作\*/

## shiyan4.cpp

//散点测试集

//1 G1 2 G2 3 G3 4 G4 5 G5 6 G6 7 G7 8 G8 9 G9 10 G10 -1 nil -1 -1

//5 线性表 8 集合 7 二叉树 6 无向图 -1 nil 5 6 5 7 6 7 7 8 -1 -1

//1 a 2 b 3 c 4 d 5 e 6 f 7 g 8 h 9 k -1 nil 1 2 2 3 3 4 4 1 3 5 6 7 6 8 7 8 8 9 -1 -1

#include "func.h"

int main() {

ALGraph \*G = (ALGraph\*)malloc(sizeof(ALGraph)); //图

G->vexnum = 0;

G->arcnum = 0;

int op = 1; //操作指令

int j = 0, k = 0; //函数执行状态

KeyType key = 0, key1 = 0; //键值载体

VertexType value;

ALGraphs GS; //多图

GS.elem = NULL;

int flag = 0; //多图操作完成标志

ALGraph\* G\_ope = NULL, \*temp = NULL; //多图操作图的载体

while (op) {

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

printf(" 基于邻接表的图操作菜单\n");

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

printf(" 1. 创建图 7. 插入顶点 13. 图形遍历\n");

printf(" 2. 销毁图 8. 删除顶点 14. 输出邻接表\n");

printf(" 3. 查找顶点 9. 插入弧 15. 存档\n");

printf(" 4. 顶点赋值 10. 删除弧 16. 读档\n");

printf(" 5. 获得第一邻接点 11. 深度优先遍历 17. 多图管理\n");

printf(" 6. 获得下一邻接点 12. 广度优先遍历 0. 退出\n");

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

if (op)

printf(" 请选择你的操作[0~17]:");

scanf("%d", &op);

switch(op) {

case 1:

if((j = CreateGraph(\*G)) == OK) printf("创建成功！\n");

else if(j == INFEASIBLE) printf("当前的图里已经有点东西了~\n");

else if(j == ERROR) printf("输入数据有误，部分顶点键值重复，创建失败！\n");

printf("----任意键继续----\n");

\_getch();

break;

case 2:

if (G->vexnum == 0) printf("你不能销毁一个没有顶点的图！\n");

else

{

DestroyGraph(\*G);

printf("销毁成功！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 3:

if (G->vexnum == 0) printf("啊咧？当前图里一个点都没有，能找啥？\n");

else

{

printf("把结点键值报给我，我报号~\n");

scanf("%d", &key);

if((j = LocateVex(\*G, key)) == -1) printf("没找到键值是%d的点呢~\n", key);

else printf("找到了！键值为%d的结点在这个图里是第%d个点, 其中的数据是：key=%d, others=%s\n", key, j + 1, G->vertices[j].data.key, G->vertices[j].data.others);

}

printf("----任意键继续----\n");

\_getch();

break;

case 4:

if (G->vexnum == 0) printf("图是空的，没有点给你赋值~\n");

else {

printf("想改哪个顶点？键值报给我：");

scanf("%d", &key);

if (LocateVex(\*G, key) == -1)

{

printf("啊这...这个树里没有键值为%d的顶点诶...(为难.jpg)\n", key);

printf("----任意键继续----\n");

\_getch();

break;

}

else

{

printf("键值改成啥？");

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

if (LocateVex(\*G, value.key) != -1 && value.key != key)

{

printf("你这个不行啊，你这个键值被别的点占用了!\n");

printf("----任意键继续----\n");

\_getch();

break;

}

else {

printf("others改成啥？");

scanf("%s", value.others);

}

}

PutVex(\*G, key, value);

printf("更改成功！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 5:

if (G->vexnum == 0) printf("甭找了，图是空的~\n");

else

{

printf("你想获得哪个结点的第一邻接点呢？把键值告诉我，我给你找~\n");

scanf("%d", &key);

if ((j = FirstAdjVex(\*G, key)) == -1) printf("诶嘿~没找着你说的这个点\n");

else printf("键值为%d的点的第一邻接点是第%d号点, 其中的数据是：key=%d, others=%s\n", key, j + 1, G->vertices[j].data.key, G->vertices[j].data.others);

}

printf("----任意键继续----\n");

\_getch();

break;

case 6:

if (G->vexnum == 0) printf("找啥找？这图里现在一个点都没有\n");

else {

printf("输入键值：");

scanf("%d", &key);

if ((j = LocateVex(\*G, key)) != -1)

{

printf("你想在它的邻接链表里找哪一个点的下一邻接点？请输入其键值：\n");

scanf("%d", &key1);

if (IsRepeat(G->vertices[j].firstarc, LocateVex(\*G, key1)))

{

if ((j = NextAdjVex(\*G, key, key1)) == -1) printf("无下一邻接点！\n");

else printf("键值为%d的点相对于键值为%d的点的下一邻接点为第%d号点，其数据为：key=%d, others=%s\n", key, key1, j + 1, G->vertices[j].data.key, G->vertices[j].data.others);

}

else printf("键值为%d的点与键值为%d的点不邻接！\n",key, key1);

}

else

{

printf("图里面压根儿就没这么个点！\n");

}

}

printf("----任意键继续----\n");

\_getch();

break;

case 7:

if (G->vexnum == MAX\_VERTEX\_NUM) printf("图中点的个数已到达上限，加不进来了！\n");

else {

printf("请输入新增顶点的键值：");

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

if (IsRepeat(\*G, value.key)) printf("不行，这不能让你加，你这个键值已经被占用了！\n");

else

{

printf("请输入新增顶点的数据：");

scanf("%s", value.others);

InsertVex(\*G, value);

printf("添加成功！\n");

}

}

printf("----任意键继续----\n");

\_getch();

break;

case 8:

if (G->vexnum == 0) printf("别删了别删了，这图里已经没点了~\n");

else

{

printf("输入待删除顶点键值：");

scanf("%d", &key);

j = LocateVex(\*G, key);

if (j == -1) printf("当前图里面没有key值为%d的顶点\n", key);

else

{

DeleteVex(\*G, key);

printf("删除成功！\n");

}

}

printf("----任意键继续----\n");

\_getch();

break;

case 9:

if (G->vexnum < 2) printf("这图里总共都没有两个点，加个锤子啊~\n");

else

{

printf("想给哪两个点之间加段弧啊？输入它们的键值：");

scanf("%d%d", &key, &key1);

j = InsertArc(\*G, key, key1);

if(j == INFEASIBLE) printf("这两点之间已经有弧了!\n");

else if (j == ERROR)

{

if ((k = LocateVex(\*G, key)) == -1)

{

if (LocateVex(\*G, key1) == -1)

printf("这两个点都不在这个图里嗷~\n");

else

printf("键值为%d的点不在这个图里\n", key);

}

else if (k != -1) printf("键值为%d的点不在这个图里\n", key1);

}

else printf("添加成功！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 10:

if (G->vexnum < 2) printf("这图里总共都没有两个点，删个锤子啊~\n");

else

{

printf("想删哪两个点之间的弧啊？输入它们的键值：");

scanf("%d%d", &key, &key1);

j = DeleteArc(\*G, key, key1);

if(j == INFEASIBLE) printf("这两个点不邻接！\n");

else if (j == ERROR)

{

if ((k = LocateVex(\*G, key)) == -1)

{

if (LocateVex(\*G, key1) == -1)

printf("这两个点都不在这个图里嗷~\n");

else

printf("键值为%d的点不在这个图里\n", key);

}

else printf("键值为%d的点不在这个图里\n", key1);

}

else printf("删除成功！\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 11:

if (G->vexnum == 0) printf("太棒了，图里一个点都没有，不用干活了~\n");

else

{

printf("深度优先遍历时，当前图中所有点的信息如下：\n");

DFSTraverse(\*G, visit);

printf("\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 12:

if (G->vexnum == 0) printf("太棒了，图里一个点都没有，不用干活了~\n");

else

{

printf("广度优先遍历时，当前图中所有点的信息如下：\n");

BFSTraverse(\*G, visit);

printf("\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 13:

if (G->vexnum == 0) printf("太棒了，图里一个点都没有，不用干活了~\n");

else

{

printf("图形遍历中，当前图中所有点的信息如下：\n");

GraphLikeTraverse(\*G, visit);

printf("\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 14:

if(G->vexnum == 0) printf("太棒了，图里一个点都没有，不用干活了~\n");

else

{

printf("当前图的邻接表如下:\n");

LinearTableTraverse(\*G, visit);

printf("当前图中有%d个点, %d条边\n", G->vexnum, G->arcnum);

printf("\n");

}

printf("----任意键继续----\n");

\_getch();

break;

case 15:

if (G->vexnum == 0) printf("嘿嘿~一个点都没有，不用存档了~\n");

else {

char SaveFile[40];

printf("输入存档文件路径(输入0选择默认路径):\n");

scanf("%s", SaveFile);

if (!strcmp(SaveFile, "0")) strcpy(SaveFile, "D:\\Graph\_save.dat");

SaveGraph(\*G, SaveFile);

printf("已将数据存入 %s\n", SaveFile);

}

printf("----任意键继续----\n");

\_getch();

break;

case 16:

if(G->vexnum != 0) printf("当前图中已有其它结点!\n");

else

{

char LoadFile[40];

printf("输入读档文件路径(输入0选择默认路径):\n");

scanf("%s", LoadFile);

if (!strcmp(LoadFile, "0")) strcpy(LoadFile, "D:\\Graph\_save.dat");

if ((j = LoadGraph(\*G, LoadFile)) == ERROR) printf("存档文件%s无法打开！\n", LoadFile);

else if(j == INFEASIBLE) printf("存档文件中点的个数超出容量!读档失败!\n");

else printf("数据已从 %s 中读出\n", LoadFile);

printf("当前图中有%d个点, %d条边\n", G->vexnum, G->arcnum);

}

printf("----任意键继续----\n");

\_getch();

break;

case 17:

if (flag == 1) //修改状态，不可再次选择线性表集合操作

{

printf("当前有未结束的线性表修改，请先结束前一修改！\n");

printf("----任意键继续----\n");

\_getch();

break;

}

G\_ope = gather\_operate(GS);

if (G\_ope != NULL)//只要选择了集合操作中的修改，就将主界面的图置换成集合中所选图

{

temp = G; //temp暂存主界面线性表数据

G = G\_ope; //主界面后台线性表置换为集合内所选图

flag = 1; //系统状态置为集合操作态

}

break;

case 0:

break;

default:

printf("操作指令有误！\n");

printf("----任意键继续----\n");

\_getch();

break;

}//end of switch

if (flag && op == 0) //flag判断当前是否在线性表集合操作中，此时退出返回主界面

{

G = temp; //返还原始主界面线性表数据

op = 1; //保持不退出while

flag = 0; //状态置为初始界面

printf("原主界面的图中数据已载入！\n");

\_getch();

}

}//end of while

printf("欢迎下次再使用本系统！\n");

printf("----任意键结束----\n");

\_getch();

return 0;

}//end of main()