

**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202008**

**学 号： U202015533**

**姓 名： 徐瑞达**

**指导教师： 袁凌**

**报告日期： 2021年 5月6日**

**计算机科学与技术学院**

目 录

[1 基于顺序存储结构的线性表实现 2](#_Toc458159879)

[1.1 问题描述 2](#_Toc458159880)

[1.2 系统设计 2](#_Toc458159882)

[1.3 系统实现 2](#_Toc458159883)

[1.4 实验小结 2](#_Toc458159884)

[2 基于链式存储结构的线性表实现 2](#_Toc458159885)

[2.1 问题描述 2](#_Toc458159886)

[2.2 系统设计 2](#_Toc458159887)

[2.3 系统实现 2](#_Toc458159888)

[2.4 实验小结 2](#_Toc458159889)

[3 基于二叉链表的二叉树实现 2](#_Toc458159890)

[3.1 问题描述 2](#_Toc458159891)

[3.2 系统设计 2](#_Toc458159892)

[3.3 系统实现 2](#_Toc458159893)

[3.4 实验小结 2](#_Toc458159894)

[4 基于二叉链表的二叉树实现 2](#_Toc458159895)

[4.1 问题描述 2](#_Toc458159896)

[4.2 系统设计 2](#_Toc458159897)

[4.3 系统实现 2](#_Toc458159898)

[4.4 实验小结 2](#_Toc458159899)

[参考文献 2](#_Toc458159900)

[附录A 基于顺序存储结构线性表实现的源程序 2](#_Toc458159901)

[附录B 基于链式存储结构线性表实现的源程序 2](#_Toc458159902)

[附录C 基于二叉链表二叉树实现的源程序 2](#_Toc458159903)

[附录D 基于邻接表图实现的源程序 2](#_Toc458159904)

# 1 基于顺序存储结构的线性表实现

## 1.1 问题描述

该实验要解决的基本问题是实现线性表的各个基本功能，如初始化表、销毁表、清空表、判定空表、求表长和获得元素等等。

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。可选择以文件的形式进行存储和加载，也即将生成的线性表存入到相应的文件中，也可以从文件中获取线性表进行操作。

## 1.1.1需实现的基本运算

依据最小完备性和常用性相结合的原则，以函数形式定义了线性表的初始化表、销毁表、清空表、判定空表、求表长和获得元素等12种基本运算，具体运算功能定义如下。

⑴初始化表：函数名称是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,compare())；初始条件是线性表已存在；操作结果是返回L中第1个与e满足关系compare（）关系的数据元素的位序，若这样的数据元素不存在，则返回值为0。

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

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

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

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

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

⒀保存线性表：函数名称是SaveList(L,FileName)，初始条件是线性表L已存在；操作结果是将线性表元素按顺序保存至文件名为FileName的文件中。

⒁加载线性表：函数名称是LoadList(L,FileName)，初始条件是线性表L不存在；操作结果是将文件名为FileName的文件加载到线性表中。

⒂在管理表中插入一个线性表：函数名称是AddList(Lists,ListName)，无初始条件；操作结果是将文件名为ListName的线性表添加到管理表中。

⒃在管理表中删除一个线性表：函数名称是RemoveList(Lists, ListName)，初始条件是多线性表非空；操作结果是将文件名为ListName的线性表删除。

⒄在管理表中查找一个线性表：函数名称是LocateList(Lists, ListName)，初始条件是多线性表非空；操作结果是查找文件名为ListName的线性表并可对其进行操作。

## 1.2 系统设计

本次实验使用的数据结构定义为：

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

#define LIST\_INIT\_SIZE 10

#define LISTINCREMENT 10

typedef int status;

typedef int ElemType; //数据元素类型定义

typedef struct{ //顺序表（顺序结构）的定义

ElemType \* elem;

int length;

int listsize;

}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;

本次实验的系统设计如下：将菜单演示和用户选择写入到while循环中，用OP获取用户的选择，OP初始化为1,以便第一次能进入循环。进入循环后系统首先显示功能菜单，然后用户输入选择0-14,其中1-17分别代表对线性表或线性表的管理表的一个基本运算，在主函数中通过switch语句对应到相应的函数功能，执行完该功能后break跳出switch语句，继续执行while循环，直至用户输入0退出当前演示系统。 (演示系统结构如图1-1所示)

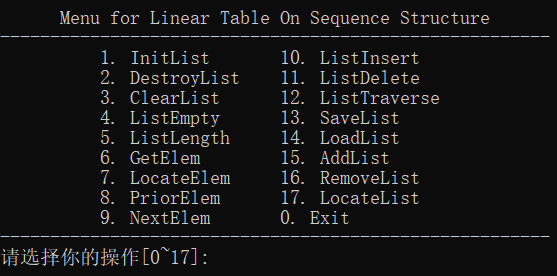


图1-1 演示系统结构图

## 1.3 系统实现

线性表运算算法思想与设计如下：

⑴初始化线性表思想：将线性表初始化过程写成函数，其中传入函数的参数是主函数中定义的结构型变量L的引用。在函数中，首先判断L是否创建，如果未创建则使用malloc函数分配LIST\_INIT\_SIZE大小的连续内存空间，如果分配成功则将newbase赋值给L.elem，由于线性表的长度为0，将 L.length初始化为0，L.listsize初始化为LIST\_INIT\_SIZE，即完成了线性表的初始化。经分析，算法的时间复杂度为O(1)。

⑵销毁线性表思想：将销毁线性表的过程写成函数，其中传入函数的参数是主函数中定义的结构性变量L的引用。在函数中，首先使用free函数释放掉以L.elem为首地址的连续内存空间，再将 L.length，L.listsize重新赋值为0。经分析，该算法的时间复杂度为O(1)。

⑶清空线性表的思想：将清空表的过程写成函数，其中将主函数中定义的 结构性变量L的引用作为函数参数。在函数中，由于清空操作并不释放内存空间，故只需将线性表的长度置为0即可。经分析，该算法的时间复杂度为O(1)。

⑷求线性表表长的思想：将求表长过程写成函数，其中主函数中定义的结 构性变量L的引用作为函数的参数，在函数中，直接返回L.length即为所求线性表的表长。经分析，该算法的时间复杂度为O(1)。

⑸获得元素的算法思想：将获得线性表元素写成函数，其中函数的参数是结构型变量L以及数据元素的序号i,由于采取的是线性存储结构，故直接通过访问数组的方式即L.elem[i-1]来获取元素，当然，在这之前需要判断合法性。经分析，该算法的时间复杂度为O(1)。

⑹查找元素的算法思想：将查找线性表特定值的数据元素写成函数，其中函数的参数是主函数中定义的结构类型变量L以及查找的数据元素的值，通过循环对线性表中的每一个元素与给定值比较是否相等，如果相等就返回该元素的位置。经分析，该算法的时间复杂度为O(n)。

⑺获得直接前驱算法思想：将获得直接前驱元素的过程写成函数，函数的参数是结构体类型变量以及特定数据元素的值，接受前驱的变量。在函数中，首先判断线性表是否为空，若不为空则调用获得元素的函数判断该线性表中特定数据元素的位序，如果位置为1，则返回ERROR, 否则将其前一个元素即L.elem[i-2]赋给接受前驱的变量。经分析，该算法的时间复杂度为O(n)。

⑻获得后继算法思想：将获得直接后继元素的过程写成函数，函数的参数是结构体类型变量以及特定数据元素的值，接受后继的变量。在函数中，首先判断线性表是否为空，若不为空则调用获得元素的函数判断该线性表中特定数据元素的位序。如果位置为L.length，则返回ERROR, 否则将其后一个元素即L.elem[i]赋给接受后继的变量。经分析，该算法的时间复杂度为O(n)。

⑼插入元素算法思想：将插入函数写成函数，函数的参数是结构型变量的引用，插入元素的值以及插入位置。在函数中，首先判断插入位置的合法性， 即是否在线性表中合适的位置，其次还要判断当前存储空间是否已满，如果存储空间已满则要用realloc函数重新分配空间，插入元素时要先从L.length开始将元素依次后移，然后将待插入元素插入到空位中。经分析，该算法的时间复杂度为O(n)。

⑽删除元素算法思想：将删除线性表中元素写成函数，函数的参数是结构类型变量的引用，待删除元素的位置，接受待删除元素的变量。在函数中，首先判断位序的合法性，然后直接将删除元素位置后一个元素直到最后一个元素从前往后向前移动一个单元。经分析，该算法的时间复杂度为O(n)。

⑾遍历线性表算法思想：将遍历线性表写成一个函数，函数的参数是结构类型变量，直接用一个循环来对线性表中的每一个元素进行操作（此处进行输出操作）。经分析，该算法的时间复杂度为 O(n)。

⑿保存线性表到文件中的算法思想：函数的参数是结构体变量L和待保存文件的路径。在函数中，创建一个指向待保存文件的文件指针，通过循环依次将线性表中的数据写入文件中。经分析，该算法的算法复杂度是O(n)。

⒀加载文件到线性表中的算法：函数的参数是结构体变量L的引用和待加载文件的路径。在函数中，创建一个指向待加载文件的文件指针，并要为线性表L分配空间，通过循环依次将文件中的数据读入线性表中，如果存储已满，还要增加分配空间。经分析，该算法的算法复杂度是O(n)。

⒁插入一个线性表算法思想：函数的参数是管理表的引用和待插入线性表的名称。在函数中，先在已存在的管理表中的末尾处开辟新的线性表的存储空间，并将管理表长度增加1。经分析，该算法的时间复杂度是O(1)。

⒂删除一个线性表算法思想：函数的参数是管理表的引用和待删除线性表的名称。在函数中，通过循环比较线性表名称和待删除线性表的名称，如果没有找到则返回ERROR，否则就先摧毁待删除线性表，然后自该位置起从前往后将下一个线性表复制到前一个线性表中。经分析，该算法复杂度是O（n2）。

⒃定位一个线性表算法思想：函数参数是管理表Lists和待查找的线性表名称。在函数中，通过循环比较待查找线性表名称和管理表中线性表的名称，如果匹配成功则返回该线性表位序，否则返回ERROR。经分析，该算法复杂度是O(n)。

**1.4 系统测试**

|  |  |  |
| --- | --- | --- |
| 测试顺序 | 调用函数说明 | 实际输出 |
| 2 | DestroyList销毁线性表 |  |
| 3 | ClearList清空线性表 |  |
| 5 | ListLength求表长 |  |
| 1 | InitList创建线性表 |  |
| 1 | 再次创建会报错 |  |
| 4 | ListEmpty判空，还未插入元素线性表为空 |  |
| 5 | 此时表长为0 |  |
| 10 | ListInsert插入元素 |  |
| 10 |  |  |
| 10 |  |  |
| 12 | ListTraverse遍历线性表 |  |
| 4 | 此时不为空 |  |
| 5 | 此时长度为2 |  |
| 6 | GetElem获取某位置的元素 |  |
| 6 |  |  |
| 6 |  |  |
| 7 | LocateElem查找某元素 |  |
| 7 |  |  |
| 8 | PriorElem获取直接前驱 |  |
| 8 |  |  |
| 8 |  |  |
| 9 | NextElem获取直接后继 |  |
| 9 |  |  |
| 9 |  |  |
| 11 | ListDelete删除元素 |  |
| 11 |  |  |
| 11 |  |  |
| 10 | 在第一个元素后依次插入5 7 9 11 23等5个元素，过程截图略 | |
| 12 | 遍历当前线性表 |  |
| 13 | SaveList保存文件 |  |
| 2 | 销毁线性表 |  |
| 14 | 加载文件 |  |
| 12 | 遍历当前线性表 |  |
| 15 | AddList添加线性表 |  |
| 16 | RemoveList移除线性表 |  |
| 16 |  |  |
| 17 | LocateList查找线性表 |  |
| 17 |  |  |

## 1.5 实验小结

在实验中，我遇到了如下几个错误：

（1）在功能演示系统中，调用 插入元素函数时将参数写反，导致出错，由于自己在测试时使用的数据中位置和元素一样未发现错误，后来在助教检查时方测试出来。这也给我提了醒，以后在测试时要尽量使测试数据随机。

（2）在添加case语句时，由于忘记添加break而导致了两个函数的测试过程相连接，经过同学提醒才意识到这一点。

（3）在代码健壮性方面，未能及时添加文件是否打开成功，文件指针的释放等代码，尽管在测试中可能影响不大，但在实际应用中可能是隐患。同时，对于判断分配空间是否成功这一点也是在后来优化代码时添加的。

（4）在代码优化部分，在对多线性表的管理程序的测试时发现每次输入数据过程繁琐，因此添加了从文件读入的选择，大大节省了测试时间。在以后的实验中，也要注意使用文件输入来进行测试。

# 2 基于链式存储结构的线性表实现

## 2.1 问题描述

该实验要解决的基本问题是实现单链表的各个基本功能，如初始化表、销毁表、清空表、判定空表、求表长和获得元素等等。

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。可选择以文件的形式进行存储和加载，也即将生成的单链表存入到相应的文件中，也可以从文件中获单链表进行操作。

**2.1.1需实现的基本运算**

依据最小完备性和常用性相结合的原则，以函数形式定义了单链表的初始化表、销毁表、清空表、判定空表、求表长和获得元素等12种基本运算，具体运算功能定义如下。

⑴初始化表：函数名称是InitList(L)；初始条件是单链表L不存在；操作结果是构造一个空的单链表。

⑵销毁表：函数名称是DestroyList(L)；初始条件是单链表L已存在；操作结果是销毁单链表L。

⑶清空表：函数名称是ClearList(L)；初始条件是单链表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)；初始条件是单链表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)；初始条件是单链表已存在；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)；初始条件是单链表已存在，1≤i≤ListLength(L)；操作结果是用e返回L中第i个数据元素的值。

⑺查找元素：函数名称是LocateElem(L,e,compare())；初始条件是单链表已存在；操作结果是返回L中第1个与e满足关系compare（）关系的数据元素的位序，若这样的数据元素不存在，则返回值为0。

⑻获得前驱：函数名称是PriorElem(L,cur\_e,pre\_e)；初始条件是单链表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)；初始条件是单链表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)；初始条件是单链表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)；初始条件是单链表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,visit())，初始条件是单链表L已存在；操作结果是依次对L的每个数据元素调用函数visit()。

⒀保存单链表：函数名称是SaveList(L,FileName)，初始条件是单链表L已存在；操作结果是将单链表元素按顺序保存至文件名为FileName的文件中。

⒁加载单链表：函数名称是LoadList(L,FileName)，初始条件是单链表L不存在；操作结果是将文件名为FileName的文件加载到单链表中。

⒂在管理表中插入一个单链表：函数名称是AddList(Lists,ListName)，无初始条件；操作结果是将文件名为ListName的单链表添加到管理表中。

⒃在管理表中删除一个单链表：函数名称是RemoveList(Lists, ListName)，初始条件是多单链表非空；操作结果是将文件名为ListName的单链表删除。

⒄在管理表中查找一个单链表：函数名称是LocateList(Lists, ListName)，初始条件是多单链表非空；操作结果是查找文件名为ListName的单链表并可对其进行操作。

## 2.2 系统设计

本次实验使用的数据结构定义为：

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

#define LIST\_INIT\_SIZE 10

#define LISTINCREMENT 10

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{ //单链表的管理表定义

struct { char name[30];

LinkList L;

}elem[10];

int length;

}LISTS;

本次实验的系统设计如下：将菜单演示和用户选择写入到while循环中，用OP获取用户的选择，OP初始化为1,以便第一次能进入循环。进入循环后系统首先显示功能菜单，然后用户输入选择0-17,其中1-17分别代表对单链表或单链表的管理表的一个基本运算，在主函数中通过switch语句对应到相应的函数功能，执行完该功能后break跳出switch语句，继续执行while循环，直至用户输入0退出当前演示系统。 (演示系统结构如图2-1所示)

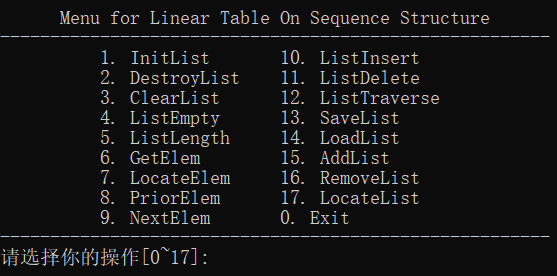


图2-1 演示系统结构图

## 2.3 系统实现

单链表运算算法思想与设计如下：

⑴初始化单链表思想：将单链表初始化过程写成函数，其中传入函数的参数是主函数中定义的结构型变量L的引用。在函数中，首先判断L是否创建，如果未创建则使用malloc函数分配一个结点的内存空间，并把L的next指针域置为空。经分析，算法的时间复杂度为O(1)。

⑵销毁单链表思想：将销毁单链表的过程写成函数，其中传入函数的参数是主函数中定义的结构性变量L的引用。在函数中，使用循环依次将每个结点的内存空间释放，最后将头指针置为空。经分析，该算法的时间复杂度为O(n)。

⑶清空单链表的思想：将清空单链表的过程写成函数，其中将主函数中定义的结构性变量L的引用作为函数参数。在函数中，使用循环从头结点开始依次释放内存，最后将L的指针域置为空。经分析，该算法的时间复杂度为O(n)。

⑷求单链表表长的思想：将求表长过程写成函数，其中主函数中定义的结 构性变量L的引用作为函数的参数，在函数中，通过循环和计数变量记录结点数，最后返回计数结果即为表长。经分析，该算法的时间复杂度为O(n)。

⑸获得元素的算法思想：将获得单链表元素写成函数，其中函数的参数是结构型变量L以及数据元素的序号i,由于采取的是链式存储结构，故通过循环依次访问结点并比较数据域的方式来获取元素，当然，如果位置不合法应返回ERROR。经分析，该算法的时间复杂度为O(n)。

⑹查找元素的算法思想：将查找单链表特定值的数据元素写成函数，其中函数的参数是主函数中定义的结构类型变量L以及查找的数据元素的值，通过循环对单链表中的每一个结点的数据域与给定值比较是否相等，如果相等就返回该元素的位置。经分析，该算法的时间复杂度为O(n)。

⑺获得直接前驱算法思想：将获得直接前驱元素的过程写成函数，函数的参数是结构体类型变量以及特定数据元素的值，接受前驱的变量。在函数中，首先判断单链表是否为空，若不为空则通过循环遍历元素比较，如果头结点元素即为所找元素则表明无直接前驱，如果头结点不是则用另一个指针指向当前遍历结点的前一个结点，当找到元素时将前一个结点的值赋给pre，若找不到则返回ERROR。经分析，该算法的时间复杂度为O(n)。

⑻获得直接后继算法思想：将获得直接后继元素的过程写成函数，函数的参数是结构体类型变量以及特定数据元素的值，接受后继的变量。在函数中，首先判断单链表是否为空，若不为空则通过循环遍历元素比较，当找到元素时将后一个结点的值赋给next，若该元素对应最后一个结点则表明无直接后继，若找不到元素则返回ERROR。经分析，该算法的时间复杂度为O(n)。

⑼插入元素算法思想：将插入元素写成函数，函数的参数是结构型变量的引用，插入元素的值以及插入位置。在函数中，首先判断插入位置的合法性， 即是否在单链表中合适的位置，插入元素时如果单链表结点数为0，则直接令L指向新创建的待插入结点，若不是则先定位至插入位置处更改结点的指针域来实现插入。经分析，该算法的时间复杂度为O(n)。

⑽删除元素算法思想：将删除单链表中元素写成函数，函数的参数是结构类型变量的引用，待删除元素的位置，接受待删除元素的变量。在函数中，先定位至待删除元素的位置，将其赋值给接受变量，然后通过改变指针域移除该结点，最后释放该结点的内存空间。经分析，该算法的时间复杂度为O(n)。

⑾遍历单链表算法思想：将遍历单链表写成一个函数，函数的参数是结构类型变量，直接用一个循环来对单链表中的每一个元素进行操作（此处进行输出操作）。经分析，该算法的时间复杂度为 O(n)。

⑿保存单链表到文件中的算法思想：函数的参数是结构体变量L和待保存文件的路径或名称。在函数中，创建一个指向待保存文件的文件指针，通过循环依次将单链表中的数据写入文件中。经分析，该算法的算法复杂度是O(n)。

⒀加载文件到单链表中的算法：函数的参数是结构体变量L的引用和待加载文件的路径。在函数中，创建一个指向待加载文件的文件指针，并根据读写逐个分配单链表L中结点的内存空间，最后将尾结点的指针域置为空并释放文件指针。经分析，该算法的算法复杂度是O(n)。

⒁插入一个单链表算法思想：函数的参数是管理表的引用和待插入单链表的名称。在函数中，先在已存在的管理表中的末尾处开辟新的单链表的存储空间，并将管理表长度增加1.经分析，该算法的时间复杂度是O(1)。

⒂删除一个单链表算法思想：函数的参数是管理表的引用和待删除单链表的名称。在函数中，通过循环比较单链表名称和待删除单链表的名称，如果没有找到则返回ERROR，否则就先摧毁待删除单链表，然后自该位置起从前往后将下一个单链表复制到前一个单链表中。经分析，该算法复杂度是O（n2）。

⒃定位一个单链表算法思想：函数参数是管理表Lists和待查找的单链表名称。在函数中，通过循环比较待查找单链表名称和管理表中单链表的名称，如果匹配成功则返回该单链表位序，否则返回ERROR。经分析，该算法复杂度是O(n)。

## 2.4 系统测试

|  |  |  |
| --- | --- | --- |
| 测试序号 | 调用函数说明 | 实际输出 |
| 2 | DestroyList销毁单链表 |  |
| 3 | ClearList清空单链表 |  |
| 5 | ListLength求表长 |  |
| 1 | InitList创建单链表 |  |
| 1 | 再次创建会报错 |  |
| 4 | ListEmpty判空，还未插入元素单链表为空 |  |
| 5 | 此时表长为0 |  |
| 10 | ListInsert插入元素 |  |
| 10 |  |  |
| 10 |  |  |
| 10 |  |  |
| 12 | ListTraverse遍历单链表 |  |
| 4 | 此时不为空 |  |
| 5 | 此时长度为3 |  |
| 6 | GetElem获取某位置的元素 |  |
| 6 |  |  |
| 6 |  |  |
| 7 | LocateElem查找某元素 |  |
| 7 |  |  |
| 8 | PriorElem获取直接前驱 |  |
| 8 |  |  |
| 8 |  |  |
| 9 | NextElem获取直接后继 |  |
| 9 |  |  |
| 9 |  |  |
| 11 | ListDelete删除元素 |  |
| 11 |  |  |
| 11 |  |  |
| 10 | 依次调用插入函数使单链表结点的数据域依次为3 5 7 11 13 17 19 23等8个元素，过程截图略 | |
| 12 | 遍历当前单链表 |  |
| 13 | SaveList保存文件 |  |
| 2 | 销毁线性表 |  |
| 14 | 加载文件 |  |
| 12 | 遍历当前线性表 |  |
| 15 | AddList添加线性表 |  |
| 16 | RemoveList移除线性表 |  |
| 16 |  |  |
| 17 | LocateList查找线性表 |  |
| 17 |  |  |

## 2.5 实验小结

在实验中，我遇到了如下几个错误：

（1）在写各个函数的代码中，总是因为混淆L和L->next而导致程序总是出现千奇百怪的输出结果，后来重新看了一遍关于单链表的结点的有关知识才避免了各种错误，这告诉我，在处理一种数据结构之前先要了解其逻辑结构。

（2）在编写程序时，我有几次都忘记了要释放链表中不需要的结点或者文件指针。这与上次实验中我犯的错误有相似之处，我会吸取教训，不再给代码埋下这种隐患。

# 3 基于二叉链表的二叉树实现

## 3.1 问题描述

该实验要解决的基本问题是实现二叉链的各个基本功能，如创建、销毁、清空、判空二叉树，求二叉树深度，查找结点，结点赋值，获得兄弟结点，插入和删除结点，前中后序遍历及其非递归算法，按层遍历等。

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。可选择以文件的形式进行存储和加载，也即将生成的二叉链表存入到相应的文件中，也可以从文件中获取二叉链表进行操作。同时也实现了多树管理。

### 3.1.1需实现的基本运算

依据最小完备性和常用性相结合的原则，以函数形式定义了二叉树的创建二叉树、销毁二叉树、清空二叉树、判定空二叉树和求二叉树深度等14种基本运算。同时又添加了求结点个数，获得父结点，获取从叶子结点到根结点的路径等功能。具体运算功能定义和说明如下。

⑴创建二叉树：函数名称是CreateBiTree(T,definition)；初始条件是definition 给出二叉树T的定义，如带空子树的二叉树前序遍历序列、或前序+中序、或后序+中序；操作结果是按definition构造二叉树T。

**注：**①要求T中各结点关键字具有唯一性。后面各操作的实现，也都要满足一棵二叉树中关键字的唯一性，不再赘述；②CreateBiTree中根据definition生成T，不应在CreateBiTree中输入二叉树的定义。

⑵销毁二叉树：函数名称是DestroyBiTree(T)；初始条件是二叉树T已存在；操作结果是销毁二叉树T。

⑶清空二叉树：函数名称是ClearBiTree (T)；初始条件是二叉树T存在；操作结果是将二叉树T清空。

⑷判定空二叉树：函数名称是BiTreeEmpty(T)；初始条件是二叉树T存在；操作结果是若T为空二叉树则返回TRUE，否则返回FALSE。

⑸求二叉树深度：函数名称是BiTreeDepth(T)；初始条件是二叉树T存在；操作结果是返回T的深度。

⑹查找结点：函数名称是LocateNode(T,e)；初始条件是二叉树T已存在，e是和T中结点关键字类型相同的给定值；操作结果是返回查找到的结点指针，如无关键字为e的结点，返回NULL。

⑺结点赋值：函数名称是Assign(T,e,value)；初始条件是二叉树T已存在，e是和T中结点关键字类型相同的给定值；操作结果是关键字为e的结点赋值为value。

⑻获得兄弟结点：函数名称是GetSibling(T,e)；初始条件是二叉树T存在，e是和T中结点关键字类型相同的给定值；操作结果是返回关键字为e的结点的（左或右）兄弟结点指针。若关键字为e的结点无兄弟，则返回NULL。

⑼插入结点：函数名称是InsertNode(T,e,LR,c)；初始条件是二叉树T存在，e是和T中结点关键字类型相同的给定值，LR为0或1，c是待插入结点；操作结果是根据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。

特殊情况，c插入作为根结点？可以考虑LR为-1时，作为根结点插入，原根结点作为c的右子树。

⑽删除结点：函数名称是DeleteNode(T,e)；初始条件是二叉树T存在，e是和T中结点关键字类型相同的给定值。操作结果是删除T中关键字为e的结点；同时，如果关键字为e的结点度为0，删除即可；如关键字为e的结点度为1，用关键字为e的结点孩子代替被删除的e位置；如关键字为e的结点度为2，用e的左孩子代替被删除的e位置，e的右子树作为e的左子树中最右结点的右子树。

⑾前序遍历：函数名称是PreOrderTraverse(T,Visit())；初始条件是二叉树T存在，Visit是一个函数指针的形参（可使用该函数对结点操作）；操作结果：先序遍历，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。

**注：**前序、中序和后序三种遍历算法，要求至少一个用非递归算法实现。

⑿中序遍历：函数名称是InOrderTraverse(T,Visit())；初始条件是二叉树T存在，Visit是一个函数指针的形参（可使用该函数对结点操作）；操作结果是中序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。

⒀后序遍历：函数名称是PostOrderTraverse(T,Visit())；初始条件是二叉树T存在，Visit是一个函数指针的形参（可使用该函数对结点操作）；操作结果是后序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。

⒁按层遍历：函数名称是LevelOrderTraverse(T,Visit())；初始条件是二叉树T存在，Visit是对结点操作的应用函数；操作结果是层序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。

⒂保存文件：函数名称是LevelOrderTraverse(T,FileName)；初始条件是二叉树T存在；操作结果是将二叉树结点按definition对二叉树的定义保存至文件名为FileName的文件中。

⒃加载文件：函数名称是LevelOrderTraverse(T,FileName)；初始条件是二叉树T存在；操作结果是将文件名为FileName的文件加载到二叉链表中。

⒄求结点个数：函数名称是CountNode (T)；初始条件是二叉树T存在；操作结果是得到二叉树中结点的数目。

⒅获得父结点：函数名称是GetParent (T,e)；初始条件是二叉树T存在；操作结果是找到关键字为e的结点并获得其父结点，否则返回NULL。

⒆插入一个树：函数名称是AddTree(Trees, TreeName);初始条件是多树管理表Trees存在；操作结果是在该管理表中插入一个二叉树。

⒇删除一个树：函数名称是RemoveTree(Trees,TreeName); 初始条件是多树管理表Trees存在；操作结果是在该管理表中删除和TreeName相同名称的二叉树，若没有匹配的树返回ERROR。

（21）查找一个树：函数名称是LocateTree(Trees,TreeName); 初始条件是多树管理表Trees存在；操作结果是在该管理表中查找和TreeName相同名称的二叉树并定位至该树对其进行操作，若没有匹配的树返回ERROR。

（22）获得叶子结点到根结点的路径：函数名称是PrintPath (T,path,pathlen)；初始条件是二叉树T存在；操作结果是输出每个叶子结点到根结点的路径。

## 3.2 系统设计

本次实验使用的数据结构定义为：

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

typedef int status;

typedef int KeyType;

typedef struct{

KeyType key;

char others[20];

}TElemType; //二叉树结点类型定义

TElemType definition[100];

typedef struct BiTNode{ //二叉链表结点的定义

TElemType data;

struct BiTNode \*lchild,\*rchild;

}BiTNode,\*BiTree;

typedef struct{ //二叉链表的管理表定义

struct { char name[30];

BiTree T;

}elem[10];

int length;

}LISTS;

本次实验的系统设计如下：将菜单演示和用户选择写入到while循环中，用OP获取用户的选择，OP初始化为1,以便第一次能进入循环。进入循环后系统首先显示功能菜单，然后用户输入选择0-24,其中1-24分别代表对二叉链表的一个基本运算，在主函数中通过switch语句对应到相应的函数功能，执行完该功能后break跳出switch语句，继续执行while循环，直至用户输入0退出当前演示系统。 (演示系统结构如图3-1所示)

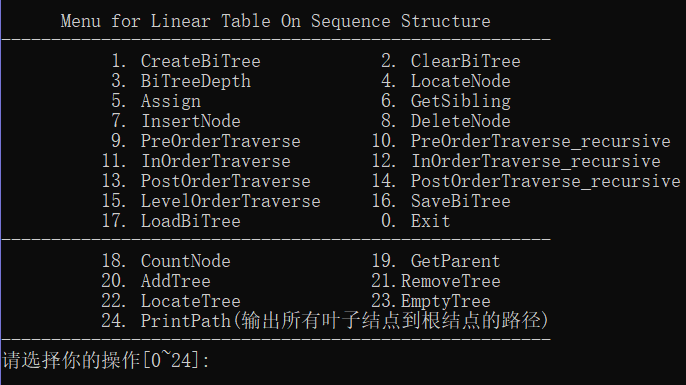


图3-1 演示系统结构图

## 3.3 系统实现

二叉链表运算算法思想与设计如下：

1. 创建二叉链表思想：将利用前序带空输入方式创建二叉链表过程写成函数，其中传入函数的参数是主函数中定义的结构型变量T的引用。在函数中，先判断是否有关键字冲突的情形，如果有返回ERROR；若关键字不冲突，则判断结点是否为空，若不为空则赋值并递归创建左右子树，若为空则将结点指针赋为空。经分析，算法的时间复杂度为O(n2)。
2. 清空二叉链表思想：将清空二叉链表的过程写成函数，其中传入函数的参数是主函数中定义的结构性变量T的引用。在函数中，首先判断树是否为空，如果为空则无需清空，如果不为空那么递归清空根结点的左右子树。最后释放根结点，并将T置为空。经分析，该算法的时间复杂度为O(n)。
3. 判空二叉链表的思想：将判空二叉链表的过程写成函数，其中将主函数中定义的结构性变量T的引用作为函数参数。在函数中，如果T==NULL则表明树为空返回OK，否则返回ERROR。经分析，该算法的时间复杂度为O(1)。
4. 求二叉链表深度的思想：将求深度过程写成函数，其中主函数中定义的结构性变量T作为函数的参数。在函数中，先判断树是否为空，为空则返回深度0，若不为空，递归计算左右子树的深度，最后取左右子树深度的最大值加1为二叉树的深度。经分析，该算法的时间复杂度为O(n)。
5. 查找元素的算法思想：将查找线性表元素写成函数，其中函数的参数是结构型变量T以及待查找结点的关键字e。在函数中若树不为空则递归查找结点及其左右子树，如果查找成功就返回查找到的结点的指针，否则返回NULL。经分析，该算法的时间复杂度为O(n)。
6. 给某个结点赋值的算法思想：将给树中的某个结点赋值的操作写成函数，其中函数的参数是主函数中定义的结构类型变量T的引用，待查找的结点的关键字e以及赋值数据项value。通过递归算法查找待赋值的关键字同时判断赋值后的关键字是否与树中的关键字冲突，查找成功则进行赋值，否则返回ERROR。经分析，该算法的时间复杂度为O(n)。
7. 获得兄弟结点算法思想：将获得兄弟结点的过程写成函数，函数的参数是结构体类型变量T以及待查找其兄弟结点的结点的关键字。在函数中，首先判断二叉链表是否为空，若不为空则判断当前结点是否有左右孩子及左右孩子是否为待查找结点，若是则返回兄弟结点，若不是则递归进行查找，若查找失败则返回NULL。经分析，该算法的时间复杂度为O(n)。
8. 插入一个结点算法思想：将插入一个结点的过程写成函数，函数的参数是结构体类型变量T的引用，待插入位置的结点的关键字，插入方式LR，待插入结点的数据。在函数中，首先判断树是否为空，若不为空则先创建待插入的结点，找到待插入位置的结点，判断待插入结点的关键字是否与原有树中的结点的关键字是否冲突，如果未找到结点或者关键字冲突则返回INFEASIBLE或ERROR，否则根据LR的不同分别做插入操作。经分析，该算法的时间复杂度为O(n)。
9. 删除结点算法思想：将删除结点写成函数，函数的参数是结构型变量T的引用，待删除结点的关键字。在函数中，首先树是否为空，若不为空则先获得关键字为e的结点及其父结点，并计算待删除结点的度。若度为0，则直接将父结点的对应孩子置为空并释放被删除结点，若度为1，则判断被删除结点是父结点的左孩子还是右孩子并分别做删除操作，若度为2，则将父结点的对应孩子置为待删除结点的左孩子，并通过循环操作找到待删除结点的左子树的最右结点，并将右子树连接至此。其中，每种情况均需要对待删除结点是根结点的情况进行特判。经分析，该算法的时间复杂度为O(n)。
10. 前序遍历算法思想：将对二叉树的前序遍历过程写成函数，函数的参数是结构类型变量T，遍历二叉树结点的visit函数的指针。对于递归算法，在函数中，首先判断树是否为空，若不为空则通过访问根结点，遍历左子树，遍历右子树的过程实现递归遍历。对于非递归算法，则利用栈的存储结构，依次将根结点，左子树，右子树入栈并出栈的循环操作实现遍历。经分析，递归与非递归算法的时间复杂度均为O(n)。
11. 中序遍历算法思想：将对二叉树的中序遍历过程写成函数，函数的参数是结构类型变量T，遍历二叉树结点的visit函数的指针。对于递归算法，在函数中，首先判断树是否为空，若不为空则通过遍历左子树，访问根结点，遍历右子树的过程实现递归遍历。对于非递归算法，则利用栈的存储结构，通过将结点按顺序入栈并出栈的操作实现遍历。经分析，递归与非递归算法的时间复杂度均为O(n)。
12. 后序遍历算法思想：将对二叉树的后序遍历过程写成函数，函数的参数是结构类型变量T，遍历二叉树结点的visit函数的指针。对于递归算法，在函数中，首先判断树是否为空，若不为空则通过访问根结点，遍历左子树，遍历右子树的过程实现递归遍历。对于非递归算法，则利用栈的存储结构，通过将结点按顺序入栈并出栈的操作实现遍历。其中，后序遍历比前两种遍历的非递归实现都要复杂。原因是，我们访问完左结点后，弹栈时并不能立即访问根结点的值，因为遍历顺序为：左->右->根，在访问根结点之前，我们还要确定根结点的右子树是否已经被访问过。因此可以利用flag来判断此结点是从左孩子结点还是右孩子结点来的。经分析，递归与非递归算法的时间复杂度均为O(n)。
13. 按层遍历算法思想：将对二叉树的按层遍历过程写成函数，函数的参数是结构类型变量T，遍历二叉树结点的visit函数的指针。在函数中，利用循环队列的数据结构来实现遍历。首先将根结点入队，接着通过判断队列是否为空来控制循环，在循环中，先将队首结点出队并访问，同时将其左右孩子依次入队。经分析，该算法的时间复杂度均为O(n)。
14. 保存文件算法思想：函数的参数是结构类型变量T和待保存文件的名称。在函数中，利用前序遍历的非递归算法过程依次将各个结点通过带空前序遍历序列的方式输出到文件FileName中。经分析，该算法的时间复杂度是O(n)。
15. 加载文件到二叉链表算法思想：函数的参数是结构类型变量T和待加载文件的名称。在函数中，先将文件读入到definition数组中，接着利用该数据定义创建二叉树。经分析，该算法的时间复杂度是O(n2)。

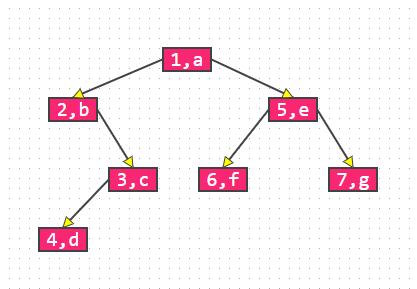
⒃添加一个二叉树的算法思想：函数参数是管理表Trees和待添加的二叉链表名称。在函数中，通过与创建二叉树类似的过程在Trees中添加一个二叉树。经分析，该算法复杂度是O(n2)。

1. 删除一个二叉树的算法思想：函数参数是管理表Trees的引用和待删除的二叉链表名称。在函数中，通过循环比较名称查找到待删除的二叉树，利用清空函数清空树，并利用循环从后至前覆盖被删除的二叉树。经分析，该算法复杂度是O(n2)。
2. 查找一个二叉树的算法思想：函数参数是管理表Trees和待查找的二叉链表名称。在函数中，通过循环比较名称查找到待删除的二叉树，并定位至该树，返回该树的位置，若未找到则返回ERROR。经分析，该算法复杂度是O(n)。
3. 计算结点数目的算法思想：函数参数是结构体变量T。在函数中，通过递归遍历左右子树将存放结点数目的变量依次增加，最终得到结点数目。经分析，该算法复杂度是O(n)。

⒇获得父结点的算法思想：将获得父结点的过程写成函数，函数的参数是结构体类型变量T以及待查找其父结点的结点的关键字。在函数中，首先判断二叉链表是否为空，若不为空则判断当前结点是否有左右孩子及左右孩子是否为待查找结点，若是则返回该结点，若不是则递归进行查找，若查找失败则返回NULL。经分析，该算法的时间复杂度为O(n)。

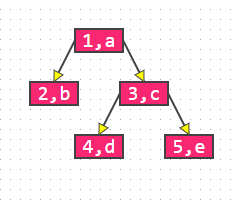
## 3.4 系统测试

说明，该全局测试所用的二叉树如图3-2和3-3



**图3-2 用到的二叉树1**

输入的前序带空序列为：1 a 2 b 0 null 3 c 4 d 0 null 0 null 0 null 5 e 6 f 0 null 0 null 7 g 0 null 0 null -1 null



**图3-3 用到的二叉树2**

输入的前序带空序列为： 1 a 2 b 0 null 0 null 3 c 4 d 0 null 0 null 5 e 0 null 0 null -1 null

|  |  |  |  |
| --- | --- | --- | --- |
| 测试序号 | 调用函数说明 | 预期输出 | 实际输出 |
| 2 | ClearBiTree清空二叉链表 | 无需清空 |  |
| 1 | CreateBiTree创建二叉链表（该树为二叉树1） | 二叉链表创建成功 |  |
| 3 | BiTreeDepth求深度 | 二叉树的深度为4 |  |
| 9 | PreOrderTraverse前序遍历 | 1,a 2,b 3,c 4,d 5,e 6,f 7,g |  |
| 11 | InOrderTraverse中序遍历 | 2,b 4,d 3,c 1,a 6,f 5,e 7,g |  |
| 14 | PostOrderTraverse后序非递归算法遍历 | 4,d 3,c 2,b 6,f 7,g 5,e 1,a |  |
| 15 | LevelOrderTraverse按层遍历 | 1,a 2,b 5,e 3,c 6,f 7,g 4,d |  |
| 18 | CountNode求结点数目 | 二叉树中结点个数为7 |  |
| 23 | EmptyTree判空树 | 二叉树不为空 |  |
| 4 | LocateNode查找结点 | 见截图 |  |
| 4 | LocateNode查找结点 | 见截图 |  |
| 6 | GetSibling查找兄弟结点 | 见截图 |  |
| 6 | GetSibling查找兄弟结点 | 见截图 |  |
| 6 | GetSibling查找兄弟结点 | 见截图 |  |
| 19 | GetParent查找父结点 | 见截图 |  |
| 19 | GetParent查找父结点 | 见截图 |  |
| 19 | GetParent查找父结点 | 见截图 |  |
| 5 | Assign赋值 | 见截图 |  |
| 5 | Assign赋值 | 见截图 |  |
| 5 | Assign赋值 | 见截图 |  |
| 9 | PreOrderTraverse前序遍历 | 1,a 2,b 3,c 4,d 5,e 6,f 7,new |  |
| 7 | InsertNode插入结点 | 见截图 |  |
| 7 | InsertNode插入结点 | 见截图 |  |
| 7 | InsertNode插入结点 | 见截图 |  |
| 7 | InsertNode插入结点 | 见截图 |  |
| 9 | PreOrderTraverse前序遍历 | 8,root 1,a 2,b 3,c 4,d 9,h 5,e 6,f 7,new |  |
| 8 | DeleteNode删除结点 | 见截图 |  |
| 8 | DeleteNode删除结点 | 见截图 |  |
| 8 | DeleteNode删除结点 | 见截图 |  |
| 9 | PreOrderTraverse前序遍历 | 1,a 2,b 3,c 4,d 5,e 6,f 7,new |  |
| 16 | SaveBiTree保存文件 | 成功保存至文件test |  |
| 2 | ClearBiTree清空树 | 二叉链表清空成功 |  |
| 3 | BiTreeDepth求深度 | 二叉树的深度为0 |  |
| 23 | EmptyTree判空树 | 二叉树为空 |  |
| 9 | PreOrderTraverse前序遍历 | 由于二叉树已经清空，将无输出 |  |
| 17 | LoadBiTree加载文件 | 成功加载文件test至二叉链表 |  |
| 10 | PreOrderTraverse\_recursive前序递归算法遍历 | 1,a 2,b 3,c 4,d 5,e 6,f 7,new |  |
| 12 | InOrderTraverse\_recursive中序递归算法遍历 | 2,b 4,d 3,c 1,a 6,f 5,e 7,new |  |
| 24 | PrintPath输出所有叶子结点到根结点的路径 | 4,d 3,c 2,b 1,a  6,f 5,e 1,a  7,new 5,e 1,a |  |
| 20 | AddTree添加树 | 见截图 |  |
| 21 | RemoveTree移除树 | 见截图 |  |
| 21 | RemoveTree移除树 | 见截图 |  |
| 22 | LocateTree查找树 | 见截图 |  |
| 22 | LocateTree查找树 | 见截图 |  |
| 10 | LevelOrderTraverse按层遍历当前定位到的树 | 1,a 2,b 3,c 4,d 5,e 6,f 7,g |  |

## 3.5 实验小结

在实验中，我遇到了如下几个错误：

（1本次实验中有许多函数都是利用递归实现的，在这个过程中，我因为不清楚递归的过程，而导致返回值时有错误的现象。同时，也因为不清楚如何计算递归方式实现的函数的时间复杂度而浪费了很多时间。

（2）在创建二叉树的函数中，为了实现递归过程中使用definition定义的正确性，我使用到了全局变量i，因此在每次调用该函数时都应该将全局变量i重置，但在需要调用的过程中总是因为忘记重置而导致测试时出现问题。

# 4 基于邻接表的图实现

## 4.1 问题描述

## 4.2 系统设计

## 4.3 系统实现

## 4.4 系统测试

## 4.5 实验小结

# 参考文献

[1] 严蔚敏等. 数据结构(C语言版). 清华大学出版社

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

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

[4] 严蔚敏等.数据结构题集(C语言版). 清华大学出版社

# 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 <ctype.h>

#include <stdlib.h>

#include <string.h>

/\*-------- Macro Definition ---------\*/

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

#define LIST\_INIT\_SIZE 10

#define LISTINCREMENT 10

/\*-------- Declare the type ---------\*/

typedef int status;

typedef int ElemType; //数据元素类型定义

typedef struct{ //顺序表（顺序结构）的定义

ElemType \* elem;

int length;

int listsize;

}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;

SqList L;

/\*-----Function Declaration---------\*/

status InitList(SqList &L);

status DestroyList(SqList &L);

status ClearList(SqList &L);

status ListEmpty(SqList L);

status ListLength(SqList L);

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

status LocateElem(SqList L,ElemType e);

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

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

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

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

status ListTraverse(SqList L);

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

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

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

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

status LocateList(LISTS Lists,char ListName[]);

/\*----- The main function --------\*/

int main(void){

L.elem=NULL;

LISTS Lists;

int op=1,res=0;

while(op){

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

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

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

printf(" 1. InitList 10. ListInsert\n");

printf(" 2. DestroyList 11. ListDelete\n");

printf(" 3. ClearList 12. ListTraverse\n");

printf(" 4. ListEmpty 13. SaveList\n");

printf(" 5. ListLength 14. LoadList\n");

printf(" 6. GetElem 15. AddList\n");

printf(" 7. LocateElem 16. RemoveList\n");

printf(" 8. PriorElem 17. LocateList\n");

printf(" 9. NextElem 0. Exit\n");

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

printf("请选择你的操作[0~17]:");

scanf("%d",&op);

switch(op){

case 1:

res=InitList(L);

if(res==OK) printf("线性表创建成功");

else if(res==INFEASIBLE) printf("线性表已创建");

else printf("线性表创建失败");

getchar();getchar();

break;

case 2:

res=DestroyList(L);

if(res==OK)

{

if(L.elem) printf("线性表销毁失败");

else printf("线性表销毁成功");

}

else if(res==INFEASIBLE) printf("线性表未创建，无需销毁");

else printf("线性表销毁失败");

getchar();getchar();

break;

case 3:

res=ClearList(L);

if (L.length||(!L.elem)) printf("线性表清除失败");

if (res==OK) printf("线性表清除成功");

if(res==INFEASIBLE) printf("线性表未创建");

getchar();getchar();

break;

case 4:

res=ListEmpty(L);

if (res==TRUE) printf("线性表为空表");

else if(res==FALSE)printf("线性表不为空表");

else printf("线性表未创建");

getchar();getchar();

break;

case 5:

res=ListLength(L);

if(res==L.length) printf("线性表的长度为%d",ListLength(L));

else printf("线性表未创建");

getchar();getchar();

break;

case 6:

int pos,e;

printf("请输入元素位置:");

scanf("%d",&pos);

res=GetElem(L,pos,e);

if(res==OK)printf("线性表中第%d个元素是%d",pos,e);

else if(res==ERROR) printf("输入位置不合法");

else printf("线性表未创建");

getchar();getchar();

break;

case 7:

int data;

printf("请输入元素:");

scanf("%d",&data);

res=LocateElem(L,data);

if(res!=ERROR) printf("线性表中元素%d的位置是%d",data,LocateElem(L,data));

else if(res==ERROR)printf("该元素不在线性表中");

else printf("线性表未创建");

getchar();getchar();

break;

case 8:

int data1,pre;

printf("请输入元素:");

scanf("%d",&data1);

res=PriorElem(L,data1,pre);

if(res==INFEASIBLE)printf("该元素无直接前驱");

else if(res!=ERROR)printf("线性表中元素%d的直接前驱元素是%d",data1,pre);

else printf("该元素不在线性表中");

getchar();getchar();

break;

case 9:

int data2,next;

printf("请输入元素:");

scanf("%d",&data2);

res=NextElem(L,data2,next);

if(res==INFEASIBLE) printf("该元素无直接后继");

else if(res!=ERROR)printf("线性表中元素%d的直接后继元素是%d\n",data2,next);

else printf("该元素不在线性表中");

getchar();getchar();

break;

case 10:

// for(int i=1;i<=30;i++) ListInsert(L,i,i);//此行代码用于快速添加元素以便进行测试

int data3,pos1;

printf("请输入插入的元素及插入的位置:");

scanf("%d%d",&data3,&pos1);

res=ListInsert(L,pos1,data3);

if(res&&res!=INFEASIBLE)printf("插入元素成功，当前线性表的长度为%d",ListLength(L));

else if(res==ERROR) printf("元素位置不合法");

else printf("线性表未创建");

getchar();getchar();

break;

case 11:

int data4,pos2;

printf("请输入删除的元素位置:");

scanf("%d",&pos2);

res=ListDelete(L,pos2,data4);

if(res&&res!=INFEASIBLE)printf("删除元素成功，被删除元素为%d，当前线性表的长度为%d",data4,ListLength(L));

else if(res==ERROR) printf("元素位置不合法");

else printf("线性表未创建");

getchar();getchar();

break;

case 12:

res=ListTraverse(L);

if(!res) printf("线性表是空表");

else if(res==INFEASIBLE) printf("线性表未创建");

getchar();getchar();

break;

case 13:

char FileName[30];

printf("请输入文件保存路径或文件名:");

scanf("%s",FileName);

res=SaveList(L,FileName);

if(res) printf("文件成功保存至%s\n",FileName);

getchar();getchar();

break;

case 14:

char FileName1[30];

printf("请输入待加载文件路径或文件名:");

scanf("%s",FileName1);

res=LoadList(L,FileName1);

if(res) printf("文件成功加载至当前线性表\n");

else printf("当前线性表未销毁，不能进行加载操作");

getchar();getchar();

break;

case 15:

int n,e1,f;

char name[30];

Lists.length=0;

printf("请选择从文件加载(0)或键盘输入(1)：");

scanf("%d",&f);

if(!f)

{

char FileName2[50];

printf("请输入待加载文件路径或文件名:");

scanf("%s",FileName2);

FILE \*fp=fopen(FileName2,"r");

if(fp==NULL) printf("未成功打开文件");

while(!feof(fp))

{

fscanf(fp,"%d",&n);

while(n--)

{

fscanf(fp,"%s",name);

AddList(Lists,name);

fscanf(fp,"%d",&e1);

while (e1)

{

ListInsert(Lists.elem[Lists.length-1].L,Lists.elem[Lists.length-1].L.length+1,e1);

fscanf(fp,"%d",&e1);

}

}

}

fclose(fp);

printf("插入线性表完毕\n当前管理表为:\n");

}

else

{

printf("请输入要添加的线性表个数:");

scanf("%d",&n);

printf("请依次输入线性表名称和线性表元素:");

while(n--)

{

scanf("%s",name);

AddList(Lists,name);

scanf("%d",&e1);

while (e1)

{

ListInsert(Lists.elem[Lists.length-1].L,Lists.elem[Lists.length-1].L.length+1,e1);

scanf("%d",&e1);

}

}

printf("插入线性表完毕\n当前管理表为:\n");

}

for(n=0;n<Lists.length;n++)

{

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

ListTraverse(Lists.elem[n].L);

putchar('\n');

}

getchar();getchar();

break;

case 16:

char check[30];

printf("请输入要删除的线性表名称:");

scanf("%s",check);

if (RemoveList(Lists,check)==OK)

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

{

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

ListTraverse(Lists.elem[n].L);

putchar('\n');

}

else printf("删除线性表失败");

getchar();getchar();

break;

case 17:

char check1[30];

printf("请输入要查找的线性表名称:");

scanf("%s",check1);

if(n=LocateList(Lists,check1))

{

printf("%s ",Lists.elem[n-1].name);

ListTraverse(Lists.elem[n-1].L);

putchar('\n');

}

else printf("该线性表不在此管理表中");

getchar();getchar();

break;

case 0:

break;

// printf("请选择你的操作[0~17]:");

}//end of switch

}//end of while

printf("\nWelcome to use this system again next time!\n");

return 0;

//system("pause");

}//end of main()

/\*----- Implementation of functions --------\*/

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

函数功能:初始化线性表

函数参数:引用线性表L的地址

函数返回值:返回int类型，创建成功返回1

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

status InitList(SqList &L)

{

if(L.elem==NULL)

{

ElemType \*newbase=(ElemType \*)malloc(sizeof(ElemType)\*LIST\_INIT\_SIZE);

if(!newbase) exit(OVERFLOW);

L.elem=newbase;

L.length=0;

L.listsize=LIST\_INIT\_SIZE;

return OK;

}

else return INFEASIBLE;

}

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

函数功能:销毁线性表

函数参数:引用线性表L的地址

函数返回值:返回int类型，销毁成功返回1

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

status DestroyList(SqList &L)

{

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

else

{

free(L.elem);

L.elem=NULL;

L.length=0;

L.listsize=0;

return OK;

}

}

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

函数功能:清空线性表

函数参数:引用线性表L的地址

函数返回值:返回int类型，清空成功返回1

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

status ClearList(SqList &L)

{

if(L.elem!=NULL)

{

L.length=0;

return OK;

}

else return INFEASIBLE;

}

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

函数功能:线性表判空

函数参数:线性表L

函数返回值:返回int类型，线性表为空返回1

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

status ListEmpty(SqList L)

{

if(L.elem!=NULL)

{

if(L.length==0) return TRUE;

else return FALSE;

}

else return INFEASIBLE;

}

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

函数功能:求线性表表长

函数参数:线性表L

函数返回值:返回int类型的线性表长度

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

status ListLength(SqList L)

{

if(L.elem!=NULL) return L.length;

else return INFEASIBLE;

}

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

函数功能:得到线性表中的元素并将该元素赋值给e

函数参数:线性表L，元素的位置i，引用元素e的地址

函数返回值:返回int类型，获得成功返回1

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

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

{

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

else if(i<1||i>L.length) return ERROR;

else

{

e=L.elem[i-1];

return OK;

}

}

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

函数功能:查找元素

函数参数:线性表L，待查找元素e

函数返回值:返回int类型，查找成功返回元素位置，否则返回ERROR

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

status LocateElem(SqList L,ElemType e)

{

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

else

{

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

if(L.elem[i-1]==e) return i;

return ERROR;

}

}

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

函数功能:得到元素的直接前驱并赋值给pre

函数参数:线性表L，元素e，引用待返回元素的地址

函数返回值:返回int类型，得到前驱成功返回1

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

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

{

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

else

{

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

{//此处也可以直接调用LocateElem函数得到元素e的位置

if(L.elem[i-1]==e)

{

if(i==1) return INFEASIBLE;

else {pre=L.elem[i-2];return OK;}

}

}

return ERROR;

}

}

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

函数功能:得到元素的直接后继并赋值给next

函数参数:线性表L，元素e，引用待返回元素的地址

函数返回值:返回int类型，得到后继成功返回1

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

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

{

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

else

{

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

{

if(L.elem[i-1]==e)

{

if(i==L.length) return INFEASIBLE;

else {next=L.elem[i];return OK;}

}

}

return ERROR;

}

}

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

函数功能:插入元素在第i个元素前

函数参数:引用线性表L的地址，待插入元素元素e，待插入位置i

函数返回值:返回int类型，插入成功返回1

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

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

{

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

else

{

if(i>L.length+1||i<=0) return ERROR;

if(L.length==L.listsize)

{

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

if(!newbase) exit(OVERFLOW);

L.elem=newbase;

L.listsize+=LISTINCREMENT;

}

for(int j=L.length;j>=i;j--) L.elem[j]=L.elem[j-1];

L.elem[i-1]=e;

L.length++;

return OK;

}

}

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

函数功能:删除第i个元素并得到被删除元素的值

函数参数:引用线性表L的地址，引用返回元素e的地址，待删除位置i

函数返回值:返回int类型，插入成功返回1

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

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

{

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

else

{

if(i>=L.length+1||i<=0) return ERROR;

e=L.elem[i-1];

for(int j=i;j<=L.length;j++) L.elem[j-1]=L.elem[j];

L.length--;

return OK;

}

}

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

函数功能:遍历输出线性表中的元素

函数参数:线性表L的

函数返回值:返回int类型，遍历成功返回1

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

status ListTraverse(SqList L)

{

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

else

{

for(int i=1;i<=L.length;i++) printf("%d ",L.elem[i-1]);

if(L.length==0) return ERROR;

}

return OK;

}

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

函数功能:保存线性表到指定文件中

函数参数:线性表L，待保存文件的路径FileName

函数返回值:返回int类型，保存成功返回1

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

status SaveList(SqList L,char FileName[])

{

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

else

{

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

if(fout==NULL) {printf("未成功打开文件");return ERROR;}

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

fprintf(fout,"%d ",L.elem[i-1]);

fclose(fout);

return OK;

}

}

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

函数功能:加载文件到线性表中

函数参数:线性表L，待加载文件的路径FileName

函数返回值:返回int类型，加载成功返回1

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

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

{

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

else

{

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

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

if(fp==NULL) {printf("未成功打开文件");return ERROR;}

int i=1,flag=0;

L.length=0;

L.listsize=LIST\_INIT\_SIZE;

while(!feof(fp))

{

if(L.length==L.listsize)

{

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

if(!newbase) exit(OVERFLOW);

L.elem=newbase;

L.listsize+=LISTINCREMENT;

}

fscanf(fp,"%d",&L.elem[i-1]);

i++;

L.length++;

}

L.length--;

fclose(fp);

return OK;

}

}

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

函数功能:在线性表的管理表中插入一个新的线性表

函数参数:引用线性表的管理表Lists的地址，待插入线性表的名称FileName

函数返回值:返回int类型，插入成功返回1

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

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

{

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

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

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

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

Lists.length++;

return OK;

}

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

函数功能:在线性表的管理表中删除一个线性表

函数参数:引用线性表的管理表Lists的地址，待删除线性表的名称FileName

函数返回值:返回int类型，删除成功返回1

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

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

{

for(int j=1;j<=Lists.length;j++)

{

if(strcmp(ListName,Lists.elem[j-1].name)==0)

{

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

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

{

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

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

Lists.elem[i-1].L.length=Lists.elem[i].L.length;

Lists.elem[i-1].L.listsize=Lists.elem[i].L.listsize;

}

Lists.length--;

return OK;

}

}

return ERROR;

}

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

函数功能:在线性表的管理表中查找线性表

函数参数:线性表的管理表Lists，待查找线性表的名称FileName

函数返回值:返回int类型，查找成功返回该线性表的位置

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

int LocateList(LISTS Lists,char ListName[])

{

for(int i=1;i<=Lists.length;i++)

if(strcmp(ListName,Lists.elem[i-1].name)==0)

{

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

return i;

}

return ERROR;

}

# 附录B 基于链式存储结构线性表实现的源程序

/\* Linear Table On Sequence Structure \*/

#include <stdio.h>

#include <ctype.h>

#include <stdlib.h>

#include <string.h>

/\*-------- Macro Definition ---------\*/

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

#define LIST\_INIT\_SIZE 10

#define LISTINCREMENT 10

//#define LL Lists.elem[num-1].L

/\*-------- Declare the type ---------\*/

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{ //单链表的管理表定义

struct { char name[30];

LinkList L;

}elem[10];

int length;

}LISTS;

LinkList L;

/\*-----Function Declaration---------\*/

status InitList(LinkList &L);

status DestroyList(LinkList &L);

status ClearList(LinkList &L);

status ListEmpty(LinkList L);

status ListLength(LinkList L);

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

status LocateElem(LinkList L,ElemType e);

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

status NextElem(LinkList L,ElemType e,ElemType &next);

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

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

status ListTraverse(LinkList L);

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

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

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

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

status LocateList(LISTS Lists,char ListName[]);

/\*----- The main function --------\*/

int main(void){

L=NULL;

LISTS Lists;

int op=1,res=0;

while(op){

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

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

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

printf(" 1. InitList 10. ListInsert\n");

printf(" 2. DestroyList 11. ListDelete\n");

printf(" 3. ClearList 12. ListTraverse\n");

printf(" 4. ListEmpty 13. SaveList\n");

printf(" 5. ListLength 14. LoadList\n");

printf(" 6. GetElem 15. AddList\n");

printf(" 7. LocateElem 16. RemoveList\n");

printf(" 8. PriorElem 17. LocateList\n");

printf(" 9. NextElem 0. Exit\n");

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

printf("请选择你的操作[0~17]:");

scanf("%d",&op);

switch(op){

case 1:

res=InitList(L);

if(res==OK) printf("单链表创建成功");

else if(res==INFEASIBLE) printf("单链表已创建");

else printf("单链表创建失败");

getchar();getchar();

break;

case 2:

res=DestroyList(L);

if(res==OK)

{

if(L) printf("单链表销毁失败");

else printf("单链表销毁成功");

}

else if(res==INFEASIBLE) printf("单链表未创建，无需销毁");

else printf("单链表销毁失败");

getchar();getchar();

break;

case 3:

res=ClearList(L);

if(res==INFEASIBLE) printf("单链表未创建,");

if (!L) printf("单链表清除失败");

if (res==OK) printf("单链表清除成功");

getchar();getchar();

break;

case 4:

res=ListEmpty(L);

if (res==TRUE) printf("单链表为空表");

else if(res==FALSE)printf("单链表不为空表");

else printf("单链表未创建");

getchar();getchar();

break;

case 5:

res=ListLength(L);

if(res!=-1) printf("单链表的长度为%d",ListLength(L));

else printf("单链表未创建");

getchar();getchar();

break;

case 6:

int pos,e;

printf("请输入元素位置:");

scanf("%d",&pos);

res=GetElem(L,pos,e);

if(res==OK)printf("单链表中第%d个元素是%d",pos,e);

else if(res==ERROR) printf("输入位置不合法");

else printf("单链表未创建");

getchar();getchar();

break;

case 7:

int data;

printf("请输入元素:");

scanf("%d",&data);

res=LocateElem(L,data);

if(res!=ERROR) printf("单链表中元素%d的位置是%d",data,LocateElem(L,data));

else if(res==ERROR)printf("该元素不在单链表中");

else printf("单链表未创建");

getchar();getchar();

break;

case 8:

int data1,pre;

printf("请输入元素:");

scanf("%d",&data1);

res=PriorElem(L,data1,pre);

if(res==INFEASIBLE)printf("该元素无直接前驱");

else if(res!=ERROR)printf("单链表中元素%d的直接前驱元素是%d",data1,pre);

else printf("该元素不在单链表中");

getchar();getchar();

break;

case 9:

int data2,next;

printf("请输入元素:");

scanf("%d",&data2);

res=NextElem(L,data2,next);

if(res==INFEASIBLE) printf("该元素无直接后继");

else if(res!=ERROR)printf("单链表中元素%d的直接后继元素是%d\n",data2,next);

else printf("该元素不在单链表中");

getchar();getchar();

break;

case 10:

// for(int i=1;i<=30;i++) ListInsert(L,i,i);//此行代码用于快速添加元素以便进行测试

int data3,pos1;

printf("请输入插入的元素及插入的位置:");

scanf("%d%d",&data3,&pos1);

res=ListInsert(L,pos1,data3);

if(res&&res!=INFEASIBLE)printf("插入元素成功，当前单链表的长度为%d",ListLength(L));

else if(res==ERROR) printf("元素位置不合法");

else printf("单链表未创建");

getchar();getchar();

break;

case 11:

int data4,pos2;

printf("请输入删除的元素位置:");

scanf("%d",&pos2);

res=ListDelete(L,pos2,data4);

if(res&&res!=INFEASIBLE)printf("删除元素成功，被删除元素为%d，当前单链表的长度为%d",data4,ListLength(L));

else if(res==ERROR) printf("元素位置不合法");

else printf("单链表未创建");

getchar();getchar();

break;

case 12:

res=ListTraverse(L);

if(!res) printf("单链表是空表");

else if(res==INFEASIBLE) printf("单链表未创建");

getchar();getchar();

break;

case 13:

char FileName[30];

printf("请输入文件保存路径或文件名:");

scanf("%s",FileName);

res=SaveList(L,FileName);

if(res) printf("文件成功保存至%s\n",FileName);

getchar();getchar();

break;

case 14:

char FileName1[30];

printf("请输入待加载文件路径或文件名:");

scanf("%s",FileName1);

res=LoadList(L,FileName1);

if(res) printf("文件成功加载至当前单链表\n");

else printf("当前单链表未销毁，不能进行加载操作");

getchar();getchar();

break;

case 15:

int n,e1,f;

char name[30];

Lists.length=0;

printf("请选择从文件加载(0)或键盘输入(1)：");

scanf("%d",&f);

if(!f)

{

char FileName2[50];

printf("请输入待加载文件路径或文件名:");

scanf("%s",FileName2);

FILE \*fp=fopen(FileName2,"r");

if(fp==NULL) printf("未成功打开文件");

while(!feof(fp))

{

fscanf(fp,"%d",&n);

while(n--)

{

fscanf(fp,"%s",name);

AddList(Lists,name);

fscanf(fp,"%d",&e1);

while (e1)

{

ListInsert(Lists.elem[Lists.length-1].L,ListLength(Lists.elem[Lists.length-1].L)+1,e1);

fscanf(fp,"%d",&e1);

}

}

}

fclose(fp);

printf("插入单链表完毕\n当前管理表为:\n");

}

else

{

printf("请输入要添加的单链表个数:");

scanf("%d",&n);

printf("请依次输入单链表名称和单链表元素:");

while(n--)

{

scanf("%s",name);

AddList(Lists,name);

scanf("%d",&e1);

while(e1)

{

ListInsert(Lists.elem[Lists.length-1].L,ListLength(Lists.elem[Lists.length-1].L)+1,e1);

scanf("%d",&e1);

}

}

printf("插入单链表完毕\n当前管理表为:\n");

}

for(n=0;n<Lists.length;n++)

{

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

ListTraverse(Lists.elem[n].L);

putchar('\n');

}

getchar();getchar();

break;

case 16:

char check[30];

printf("请输入要删除的单链表名称:");

scanf("%s",check);

if (RemoveList(Lists,check)==OK)

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

{

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

ListTraverse(Lists.elem[n].L);

putchar('\n');

}

else printf("删除单链表失败");

getchar();getchar();

break;

case 17:

char check1[30];

printf("请输入要查找的单链表名称:");

scanf("%s",check1);

if(n=LocateList(Lists,check1))

{

printf("%s ",Lists.elem[n-1].name);

ListTraverse(Lists.elem[n-1].L);

putchar('\n');

}

else printf("该单链表不在此管理表中");

getchar();getchar();

break;

case 0:

break;

// printf("请选择你的操作[0~17]:");

}//end of switch

}//end of while

printf("\nWelcome to use this system again next time!\n");

return 0;

//system("pause");

}//end of main()

/\*----- Implementation of functions --------\*/

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

函数功能:初始化单链表

函数参数:引用单链表L的地址

函数返回值:返回int类型，创建成功返回1

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

status InitList(LinkList &L)

{

if(L==NULL)

{

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

L->next=NULL;

return OK;

}

else return INFEASIBLE;

}

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

函数功能:销毁单链表

函数参数:引用单链表L的地址

函数返回值:返回int类型，销毁成功返回1

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

status DestroyList(LinkList &L)

{

if(L==NULL) return INFEASIBLE;

else

{

LinkList q=L->next;

while(q!=NULL){

free(L);

L=q;

q=L->next;

}

}

L=NULL;

return OK;

}

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

函数功能:清空单链表

函数参数:引用单链表L的地址

函数返回值:返回int类型，清空成功返回1

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

status ClearList(LinkList &L)

{

if(L!=NULL)

{

LinkList pre,p;

p = L->next;

while(p!=NULL) {

pre=p;

p=p->next;

free(pre);

}

L->next=NULL;

return OK;

}

else return INFEASIBLE;

}

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

函数功能:单链表判空

函数参数:单链表L

函数返回值:返回int类型，单链表为空返回1

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

status ListEmpty(LinkList L)

{

if(L!=NULL&&L->next==NULL) return TRUE;

else if(L==NULL)return INFEASIBLE;

else return FALSE;

}

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

函数功能:求单链表表长

函数参数:单链表L

函数返回值:返回int类型的单链表长度

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

status ListLength(LinkList L)

{

if(L==NULL)return INFEASIBLE;

else

{

int i=0;

LinkList p=L->next;

while(p)

{

i++;

p=p->next;

}

return i;

}

}

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

函数功能:得到单链表中的元素并将该元素赋值给e

函数参数:单链表L，元素的位置i，引用元素e的地址

函数返回值:返回int类型，获得成功返回1

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

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

{

if(L==NULL)return INFEASIBLE;

else

{

if(L->next==NULL)return ERROR;

LinkList p=L->next;

int j=1;

while(p&&j<i)

{

++j;

p=p->next;

}

if(p==NULL||j>i) return ERROR;

e=p->data;

return OK;

}

}

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

函数功能:查找元素

函数参数:单链表L，待查找元素e

函数返回值:返回int类型，查找成功返回元素位置，否则返回ERROR

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

status LocateElem(LinkList L,ElemType e)

{

if(L==NULL)return INFEASIBLE;

else

{

if(L->next==NULL)return ERROR;

LinkList p=L->next;

int j=1;

while(p)

{

if(p->data==e) return j;

p=p->next;

j++;

}

return ERROR;

}

}

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

函数功能:得到元素的直接前驱并赋值给pre

函数参数:单链表L，元素e，引用待返回元素的地址

函数返回值:返回int类型，得到前驱成功返回1

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

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

{

if(L==NULL)return INFEASIBLE;

else

{

if(L->next==NULL)return ERROR;

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

if(p->data==e)return INFEASIBLE;

while(p)

{

if(p->data==e)

{

if(p==L->next)return ERROR;

pre=q->data;

return OK;

}

q=p;

p=p->next;

}

return ERROR;

}

}

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

函数功能:得到元素的直接后继并赋值给next

函数参数:单链表L，元素e，引用待返回元素的地址

函数返回值:返回int类型，得到后继成功返回1

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

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

{

if(L==NULL)return INFEASIBLE;

else

{

if(L->next==NULL)return ERROR;

LinkList p=L->next;

while(p)

{

if(p->data==e)

{

if(p->next==NULL)return INFEASIBLE;

next=p->next->data;

return OK;

}

p=p->next;

}

return ERROR;

}

}

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

函数功能:插入元素在第i个元素前

函数参数:引用单链表L的地址，待插入元素元素e，待插入位置i

函数返回值:返回int类型，插入成功返回1

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

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

{

if(L==NULL)return INFEASIBLE;

else

{

if(i<=0)return ERROR;

LinkList p=L;

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

q->data=e;

int j=1;

if(p->next==NULL)

{

L->next=q;

q->next=NULL;

return OK;

}

while(p&&j<i)

{

p=p->next;

j++;

}

if(p==NULL)return ERROR;

q->next=p->next;

p->next=q;

return OK;

}

}

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

函数功能:删除第i个元素并得到被删除元素的值

函数参数:引用单链表L的地址，引用返回元素e的地址，待删除位置i

函数返回值:返回int类型，插入成功返回1

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

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

{

if(L==NULL)return INFEASIBLE;

else

{

if(i<=0||L->next==NULL)return ERROR;

LinkList p=L,q;

int j=1;

while(p&&j<i)

{

p=p->next;

if(p==NULL)return ERROR;

j++;

}

if(p->next==NULL)return ERROR;

q=p->next;

e=q->data;

p->next=q->next;

free(q);

return OK;

}

}

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

函数功能:遍历输出单链表中的元素

函数参数:单链表L的

函数返回值:返回int类型，遍历成功返回1

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

status ListTraverse(LinkList L)

{

if(L==NULL)return INFEASIBLE;

else

{

LinkList p=L->next;

while(p)

{

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

if(p->next!=NULL)printf(" ");

p=p->next;

}

return OK;

}

}

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

函数功能:保存单链表到指定文件中

函数参数:单链表L，待保存文件的路径FileName

函数返回值:返回int类型，保存成功返回1

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

status SaveList(LinkList L,char FileName[])

{

if(L==NULL)return INFEASIBLE;

else

{

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

LinkList q=L->next;

while(q)

{

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

if(q->next!=NULL)fprintf(fp," ");

q=q->next;

}

fclose(fp);

return OK;

}

}

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

函数功能:加载文件到单链表中

函数参数:单链表L，待加载文件的路径FileName

函数返回值:返回int类型，加载成功返回1

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

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

{

if(L!=NULL)return INFEASIBLE;

else

{

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

if(!fin)return ERROR;

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

LinkList q=L;

while(!feof(fin))

{

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

fscanf(fin,"%d",&p->data);

q->next=p;

q=q->next;

}

q->next=NULL;

fclose(fin);

return OK;

}

}

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

函数功能:在单链表的管理表中插入一个新的单链表

函数参数:引用单链表的管理表Lists的地址，待插入单链表的名称FileName

函数返回值:返回int类型，插入成功返回1

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

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

{

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

Lists.elem[Lists.length].L=(LNode\*)malloc(sizeof(LNode));

Lists.elem[Lists.length].L->next=NULL;

Lists.length++;

return OK;

}

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

函数功能:在单链表的管理表中删除一个单链表

函数参数:引用单链表的管理表Lists的地址，待删除单链表的名称FileName

函数返回值:返回int类型，删除成功返回1

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

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

{

for(int j=1;j<=Lists.length;j++)

{

if(strcmp(ListName,Lists.elem[j-1].name)==0)

{

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

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

{

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

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

}

Lists.length--;

return OK;

}

}

return ERROR;

}

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

函数功能:在单链表的管理表中查找单链表

函数参数:单链表的管理表Lists，待查找单链表的名称FileName

函数返回值:返回int类型，查找成功返回该单链表的位置

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

int LocateList(LISTS Lists,char ListName[])

{

for(int i=1;i<=Lists.length;i++)

if(strcmp(ListName,Lists.elem[i-1].name)==0)

{

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

return i;

}

return ERROR;

}

# 附录C 基于二叉链表二叉树实现的源程序

/\* Linear Table On Sequence Structure \*/

#include <stdio.h>

#include <ctype.h>

#include <stdlib.h>

#include <string.h>

/\*-------- Macro Definition ---------\*/

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

/\*-------- Declare the type ---------\*/

typedef int status;

typedef int KeyType;

typedef struct{

KeyType key;

char others[20];

}TElemType; //二叉树结点类型定义

TElemType definition[100];

typedef struct BiTNode{ //二叉链表结点的定义

TElemType data;

struct BiTNode \*lchild,\*rchild;

}BiTNode,\*BiTree;

typedef struct{ //二叉链表的管理表定义

struct { char name[30];

BiTree T;

}elem[10];

int length;

}LISTS;

int i=0;

BiTree T=NULL;

int num=0;//声明全局变量

/\*-----Function Declaration---------\*/

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

status ClearBiTree(BiTree&T);

status EmptyTree(BiTree&T);

int BiTreeDepth(BiTree T);

BiTNode\* LocateNode(BiTree T,KeyType e);

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

BiTNode\* GetSibling(BiTree T,KeyType e);

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

status DeleteNode(BiTree &T,KeyType e);

void visit(BiTree T);

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

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

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

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

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

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

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

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

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

status AddTree(LISTS &Trees,char TreeName[]);

status RemoveTree(LISTS &Trees,char TreeName[]);

status LocateTree(LISTS Trees,char TreeName[]);

/\*-----Aiddtion of functions --------\*/

int CountNode(BiTree T);

BiTNode \*GetParent(BiTree&T,KeyType e);

void PrintPath(BiTree T,BiTree\*path,int pathlen);

/\*----- The main function --------\*/

int main(){

BiTNode\*ptr;

KeyType e;

TElemType value;

LISTS Trees;

int LR=0,op=1,ans=0,j=0,n;

char FileName[20];

while(op){

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

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

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

printf(" 1. CreateBiTree 2. ClearBiTree\n");

printf(" 3. BiTreeDepth 4. LocateNode\n");

printf(" 5. Assign 6. GetSibling\n");

printf(" 7. InsertNode 8. DeleteNode\n");

printf(" 9. PreOrderTraverse 10. PreOrderTraverse\_recursive\n");

printf(" 11. InOrderTraverse 12. InOrderTraverse\_recursive\n");

printf(" 13. PostOrderTraverse 14. PostOrderTraverse\_recursive\n");

printf(" 15. LevelOrderTraverse 16. SaveBiTree\n");

printf(" 17. LoadBiTree 0. Exit\n");

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

printf(" 18. CountNode 19. GetParent\n");

printf(" 20. AddTree 21.RemoveTree\n");

printf(" 22. LocateTree 23.EmptyTree\n");

printf(" 24. PrintPath(输出所有叶子结点到根结点的路径)\n");

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

printf("请选择你的操作[0~24]:");

scanf("%d",&op);

switch(op){

case 1:

j=0;

printf("请依次输入各结点的关键字和数据：\n");

do{

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

}while(definition[j++].key!=-1);

if(T!=NULL) printf("二叉链表已存在");

else

{

i=0;

ans=CreateBiTree(T,definition);

if(ans==OK) printf("二叉链表创建成功");

else printf("关键字不唯一，二叉链表创建失败");

}

getchar();getchar();

break;

case 2:

ans=ClearBiTree(T);

if(ans==OK)

{

if(T) printf("二叉链表清空失败");

else printf("二叉链表清空成功");

}

else if(ans==INFEASIBLE) printf("二叉链表已空，无需清空");

else printf("二叉表清空失败");

getchar();getchar();

break;

case 3:

ans=BiTreeDepth(T);

printf("二叉树的深度为%d",ans);

getchar();getchar();

break;

case 4:

printf("请输入待查找结点的关键字：");

scanf("%d",&e);

ptr=LocateNode(T,e);

if (ptr==NULL) printf("查找失败，二叉链表中无此结点");

else printf("该结点为%d,%s",ptr->data.key,ptr->data.others);

getchar();getchar();

break;

case 5:

printf("请输入待查找结点的关键字：");

scanf("%d",&e);

printf("请输入待赋值的结点数据：");

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

ans=Assign(T,e,value);

if(ans==INFEASIBLE) printf("赋值失败，待赋值的结点关键字与当前二叉树中的结点关键字冲突");

else if(ans==OK) printf("赋值成功");

else if(ans==ERROR) printf("赋值失败，二叉链表中无待查找结点");

getchar();getchar();

break;

case 6:

printf("请输入待查找结点的关键字:");

scanf("%d",&e);

ptr=LocateNode(T,e);

if(ptr==NULL)printf("二叉链表中无待查找结点");

else

{

ptr=GetSibling(T,e);

if(ptr==NULL)printf("无兄弟结点");

else printf("兄弟结点为%d,%s",ptr->data.key,ptr->data.others);

}

getchar();getchar();

break;

case 7:

int data;

printf("请输入待查找结点关键字:");

scanf("%d",&e);

printf("请输入插入结点方式:");

scanf("%d",&LR);

printf("请输入待插入结点的数据:");

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

ans=InsertNode(T,e,LR,value);

if(ans==OK) printf("插入成功");

else if(ans==INFEASIBLE) printf("插入失败,待插入结点与二叉链表中结点关键字冲突");

else printf("插入失败,无待查找结点");

getchar();getchar();

break;

case 8:

printf("请输入待删除结点关键字:");

scanf("%d",&e);

ans=DeleteNode(T,e);

if(ans==INFEASIBLE)printf("该二叉链表中无该结点");

else if(ans==OK)printf("成功删除结点");

else printf("删除失败");

getchar();getchar();

break;

case 9:

ans=PreOrderTraverse(T,visit);

getchar();getchar();

break;

case 10:

ans=PreOrderTraverse\_recursive(T,visit);

getchar();getchar();

break;

case 11:

ans=InOrderTraverse(T,visit);

getchar();getchar();

break;

case 12:

ans=InOrderTraverse\_recursive(T,visit);

getchar();getchar();

break;

case 13:

ans=PostOrderTraverse(T,visit);

getchar();getchar();

break;

case 14:

ans=PostOrderTraverse\_recursive(T,visit);

getchar();getchar();

break;

case 15:

ans=LevelOrderTraverse(T,visit);

getchar();getchar();

break;

case 16:

printf("请输入文件名称:");

scanf("%s",FileName);

ans=SaveBiTree(T,FileName);

if(ans==OK) printf("成功保存至文件%s",FileName);

else if(ans==INFEASIBLE) printf("未成功打开文件");

else printf("二叉树为空");

getchar();getchar();

break;

case 17:

printf("请输入文件名称:");

scanf("%s",FileName);

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

ans=LoadBiTree(T,FileName);

if(ans==OK) printf("成功加载文件%s至二叉链表",FileName);

else if(ans==INFEASIBLE) printf("未成功打开文件");

getchar();getchar();

break;

case 18:

ans=CountNode(T);

printf("二叉树中结点个数为%d",ans);

getchar();getchar();

break;

case 19:

printf("请输入待查找结点关键字:");

scanf("%d",&e);

ptr=GetParent(T,e);

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

else printf("二叉链表中无待查找结点或查找结点为根结点");

getchar();getchar();

break;

case 20:

char name[30],others[30];

Trees.length=0;

printf("请输入要添加的二叉链表个数:");

scanf("%d",&n);

printf("请依次输入二叉链表名称和二叉链表结点:");

while(n--)

{

scanf("%s",name);

AddTree(Trees,name);

}

printf("\n插入二叉链表完毕\n当前管理表为(依次输出每个树的前序和中序遍历):\n");

for(n=0;n<Trees.length;n++)

{

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

PreOrderTraverse(Trees.elem[n].T,visit);

putchar('\n');

InOrderTraverse(Trees.elem[n].T,visit);

putchar('\n');

}

getchar();getchar();

break;

case 21:

char check[30];

printf("请输入要删除的二叉链表名称:");

scanf("%s",check);

if (RemoveTree(Trees,check)==OK)

for(int n=0;n<Trees.length;n++)

{

printf("%s ",Trees.elem[n].name);

PreOrderTraverse(Trees.elem[n].T,visit);

putchar('\n');

InOrderTraverse(Trees.elem[n].T,visit);

putchar('\n');

putchar('\n');

}

else printf("删除二叉链表失败");

getchar();getchar();

break;

case 22:

char check1[30];

printf("请输入要查找的二叉链表名称:");

scanf("%s",check1);

if(n=LocateTree(Trees,check1))

{

printf("%s\n",Trees.elem[n-1].name);

PreOrderTraverse(Trees.elem[n-1].T,visit);

putchar('\n');

InOrderTraverse(Trees.elem[n-1].T,visit);

putchar('\n');

putchar('\n');

}

else printf("该二叉链表不在此管理表中");

getchar();getchar();

break;

case 23:

ans=EmptyTree(T);

if(ans)printf("二叉树为空");

else printf("二叉树不为空");

getchar();getchar();

break;

case 24:

BiTree path[100];

PrintPath(T,path,0);

getchar();getchar();

break;

case 0:

break;

// printf("请选择你的操作[0~17]:");

}//end of switch

}//end of while

printf("\nWelcome to use this system again next time!\n");

return 0;

//system("pause");

}//end of main()

/\*----- Implementation of functions --------\*/

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

函数功能:实现遍历时所需的功能

函数参数:二叉树T

函数返回值:无返回值

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

void visit(BiTree T)

{

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

}

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

函数功能:创建二叉链表

函数参数:二叉树T的引用，二叉树结点数组

函数返回值:返回int类型，创建成功返回1

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

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

{

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

{

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

if(definition[k].key==definition[j].key&&definition[j].key)return ERROR;

}

if(definition[i].key==0)

{

T=NULL;

i++;

}

else

{

if(!(T=(BiTNode\*)malloc(sizeof(BiTNode))))exit(OVERFLOW);

T->data=definition[i++];

CreateBiTree(T->lchild,definition);

CreateBiTree(T->rchild,definition);

}

return OK;

}

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

函数功能:清空二叉链表

函数参数:二叉树T的引用

函数返回值:返回int类型，清空成功返回1

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

status ClearBiTree(BiTree &T)

{

if(T==NULL)return INFEASIBLE;//如果T已为空，无需清空

if(T)

{//递归清空根结点中的左右子树

if(T->lchild!=NULL) ClearBiTree(T->lchild);

if(T->rchild!=NULL) ClearBiTree(T->rchild);

free(T);//释放根结点

T=NULL;

}

return OK;

}

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

函数功能:判空二叉链表

函数参数:二叉树T的引用

函数返回值:返回int类型,树为空返回1

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

status EmptyTree(BiTree&T)

{

if(T==NULL)return OK;

else return ERROR;

}

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

函数功能:求二叉树的深度

函数参数:二叉树T

函数返回值:返回int类型，二叉树为空返回0，否则返回深度

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

int BiTreeDepth(BiTree T)

{

int ld,rd;

if(T==NULL)return 0;//如果二叉树为空则深度为0

else {

ld=BiTreeDepth(T->lchild);//递归计算最左结点的深度

rd=BiTreeDepth(T->rchild);//递归计算最右结点的深度

return (ld>=rd?ld:rd)+1;//左右深度的最大值再加上根结点的层数即为二叉树深度

}

}

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

函数功能:查找二叉链表结点

函数参数:二叉树T，待查找关键字e

函数返回值:返回二叉链表结点的指针类型，指示被找到的结点

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

BiTNode\* LocateNode(BiTree T,KeyType e)

{

if(T==NULL)return NULL;

else

{

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

BiTree pl=LocateNode(T->lchild,e);

BiTree pr=LocateNode(T->rchild,e);

if(pl!=NULL)return pl;

else if(pr!=NULL)return pr;

else return NULL;

}

}

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

函数功能:给某个结点赋值

函数参数:二叉树T的引用，被赋值的结点的关键字，待赋值的结点

函数返回值:返回int类型，赋值成功返回1

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

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

{

if(T==NULL)return ERROR;

else

{

if(T->data.key==value.key&&e!=value.key) return INFEASIBLE;

//如果当前结点与待赋值的关键字相同且待查找关键字与待赋值的关键字不同时返回INFEASIBLE

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

{//否则进行赋值

T->data=value;

return OK;

}

int flag1=Assign(T->lchild,e,value);//查找左子树进行查找

if(flag1==INFEASIBLE)return INFEASIBLE;//如果查找左子树时遇到关键字冲突则返回ERROR

Assign(T->rchild,e,value);

}

}

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

函数功能:得到兄弟结点

函数参数:二叉树T，待查找兄弟结点的结点的关键字e

函数返回值:返回二叉链表结点的指针类型，若有兄弟结点返回指向兄弟结点指针，否则返回NULL

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

BiTNode\* GetSibling(BiTree T,KeyType e)

{

if(T==NULL)return NULL;

else

{

if(T->lchild!=NULL&&T->lchild->data.key==e) return T->rchild;

if(T->rchild!=NULL&&T->rchild->data.key==e) return T->lchild;

GetSibling(T->lchild,e);

GetSibling(T->rchild,e);

}

}//得到关键字为e的结点的兄弟结点

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

函数功能:插入结点

函数参数:二叉树T的引用，被插入结点的关键字e，指示插入类型的整型参数，待插入结点

函数返回值:返回int类型，插入成功返回1

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

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

{

if(T==NULL) return ERROR;

else

{

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

ctr->data=c;//创建待插入的结点

ctr->lchild=NULL;

ctr->rchild=NULL;

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

BiTNode\*ptr\_1=LocateNode(T,c.key);

if(ptr\_1!=NULL)return INFEASIBLE;

if(ptr==NULL)return ERROR;

//如果未找到被插入的结点或者插入会造成关键字不唯一时返回ERROR

if(LR==0)

{//待插入结点作为被插入结点的左孩子，原有左子树变为新结点的右子树

if(ptr->lchild!=NULL)ctr->rchild=ptr->lchild;

ptr->lchild=ctr;

}

else if(LR==1)

{//待插入结点作为被插入结点的右孩子，原有右子树变为新结点的右子树

if(ptr->rchild!=NULL) ctr->rchild=ptr->rchild;

ptr->rchild=ctr;

}

else if(LR==-1)

{//待插入结点成为新的根结点，原有的二叉树作为新结点的右子树

ctr->rchild=T;

ctr->lchild=NULL;

T=ctr;

}

return OK;

}

}

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

函数功能:删除结点

函数参数:二叉树T的引用，待删除结点的关键字

函数返回值:返回int类型，遍历成功返回1

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

status DeleteNode(BiTree &T,KeyType e)

{

if(T==NULL) return ERROR;

else

{

BiTNode \*ptr=LocateNode(T,e);//获取关键字为e的结点

BiTNode \*parent=GetParent(T,e);//得到关键字为e的结点的父结点

if(ptr==NULL)return INFEASIBLE;

int deg=0;

if(ptr->lchild==NULL&&ptr->rchild==NULL) deg=0;

else if(ptr->lchild&&ptr->rchild) deg=2;

else deg=1;//判断度

switch(deg)

{

case 0:

if(parent==NULL)T=NULL;

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

else if(parent&&parent->rchild!=NULL&&parent->rchild->data.key==e) parent->rchild=NULL;

free(ptr);

ptr=NULL;

break;

case 1:

if(parent==NULL)

{

if(ptr->lchild!=NULL) T=ptr->lchild;

else if(ptr->rchild!=NULL) T=ptr->rchild;

return OK;

}//如果是根结点

if(parent->lchild!=NULL&&parent->lchild->data.key==e)

{

if(ptr->lchild!=NULL) parent->lchild=ptr->lchild;

else parent->lchild=ptr->rchild;

}//如果被删除结点是左孩子

else if(parent->rchild!=NULL&&parent->rchild->data.key==e)

{

if(ptr->lchild!=NULL) parent->rchild=ptr->lchild;

else parent->rchild=ptr->rchild;

}//如果被删除结点是右孩子

free(ptr);

ptr==NULL;

break;

case 2:

if(parent==NULL)T=ptr->lchild;

//如果父结点为空，即待删除的结点为根结点

else

{//如果不为空

if(parent->lchild!=NULL&&parent->lchild->data.key==e) parent->lchild=ptr->lchild;

//如果父结点的左孩子是待删除结点，则父结点的左孩子为待删除结点的左孩子

else if(parent->rchild!=NULL&&parent->rchild->data.key==e) parent->rchild=ptr->lchild;

//如果父结点的右孩子是待删除结点，则父结点的左孩子为待删除结点的左孩子

}

BiTNode \*cur=ptr->lchild;

while(cur->rchild) cur=cur->rchild;//找到待删除结点的左子树的最右结点

cur->rchild=ptr->rchild;//将待删除结点的右子树接到左子树的最右结点处

free(ptr);//释放被删除结点的空间

ptr=NULL;

break;

}

return OK;

}

}

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

函数功能:递归实现先序遍历

函数参数:二叉树T，访问函数的指针visit

函数返回值:返回int类型，遍历成功返回1

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

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

{

if(T==NULL)return ERROR;

else

{

visit(T);

PreOrderTraverse(T->lchild,visit);

PreOrderTraverse(T->rchild,visit);

return OK;

}

}

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

函数功能:非递归实现先序遍历

函数参数:二叉树T，访问函数的指针visit

函数返回值:返回int类型，遍历成功返回1

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

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

{

struct BiTNode \*St[100],\*p;

int top=0;

if (T!=NULL)

{

St[top++]=T;

while(top)

{

p=St[--top];

visit(p);

if(p->rchild!=NULL) St[top++]=p->rchild;

if(p->lchild!=NULL) St[top++]=p->lchild;

}

}

}

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

函数功能:递归实现中序遍历

函数参数:二叉树T，访问函数的指针visit

函数返回值:返回int类型，遍历成功返回1

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

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

{

if(T==NULL)return ERROR;

else

{

InOrderTraverse(T->lchild,visit);

visit(T);

InOrderTraverse(T->rchild,visit);

return OK;

}

}

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

函数功能:非递归实现中序遍历

函数参数:二叉树T，访问函数的指针visit

函数返回值:返回int类型，遍历成功返回1

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

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

{

BiTNode \*st[100];

int top=0;

do

{

while(T)

{

if(top==100) exit(OVERFLOW);

st[top++]=T;

T=T->lchild;

}

if (top)

{

T=st[--top];

visit(T);

T=T->rchild;

}

}while(top||T);

}

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

函数功能:递归实现后序遍历

函数参数:二叉树T，访问函数的指针visit

函数返回值:返回int类型，遍历成功返回1

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

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

{

if(T==NULL)return ERROR;

else

{

PostOrderTraverse(T->lchild,visit);

PostOrderTraverse(T->rchild,visit);

visit(T);

return OK;

}

}

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

函数功能:非递归实现后序遍历

函数参数:二叉树T，访问函数的指针visit

函数返回值:返回int类型，遍历成功返回1

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

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

{

BiTNode \*St[100],\*pre;

int flag,top=0;

if(T!=NULL)

{

do

{

while(T!=NULL)

{

St[top++]=T;

T=T->lchild;

}

pre=NULL;

flag=1;

while(top&&flag)

{

T=St[top-1];

if(T->rchild==pre)

{

visit(T);

top--;

pre=T;

}

else

{

T=T->rchild;

flag=0;

}

}

}while(top);

}

}

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

函数功能:按层遍历二叉树

函数参数:二叉树T，访问函数的指针visit

函数返回值:返回int类型，遍历成功返回1

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

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

{

if(T==NULL) return ERROR;

else

{

BiTree elem[100],e;

int front=0,length=0;

elem[front+length]=T;

length++;

while(length!=0)

{

e=elem[front];

front=(front+1)%100;

length--;

visit(e);

if(e->lchild!=NULL)

{

elem[front+length]=e->lchild;

length++;

}

if(e->rchild!=NULL)

{

elem[front+length]=e->rchild;

length++;

}

}

return OK;

}

}

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

函数功能:保存二叉树到文件中

函数参数:二叉树T，待保存文件的路径名称

函数返回值:返回int类型，保存成功返回1

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

status SaveBiTree(BiTree T,char FileName[])

{//利用非递归的先序带空序列将二叉树保存到文件中

if(T==NULL)return ERROR;

else

{

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

if(fp==NULL)return INFEASIBLE;

struct BiTNode \*St[100],\*p;

int top=0;

St[top++]=T;

while(top)

{

p=St[--top];

if(p!=NULL)fprintf(fp,"%d %s ",p->data.key,p->data.others);

else {fprintf(fp,"0 null ");continue;}

if(p->rchild!=NULL) St[top++]=p->rchild;

else if(p->rchild==NULL)St[top++]=NULL;

if(p->lchild!=NULL) St[top++]=p->lchild;

else if(p->lchild==NULL)St[top++]=NULL;

}

fprintf(fp,"-1 null ");

fclose(fp);

return OK;

}

}

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

函数功能:加载文件到二叉树中

函数参数:二叉树T，待加载文件的路径名称

函数返回值:返回int类型，加载成功返回1

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

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

{

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

if(fp==NULL)return INFEASIBLE;

int j=0;

TElemType definition[100];

do{

fscanf(fp,"%d%s",&definition[j].key,definition[j].others);

}while(definition[j++].key!=-1);

i=0;

CreateBiTree(T,definition);

fclose(fp);

return OK;

}

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

函数功能:计算二叉树结点数目

函数参数:二叉树T

函数返回值:返回int类型的结点数目

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

int CountNode(BiTree T)

{

int n=0;

if(T)

{

n=1;

n+=CountNode(T->lchild);

n+=CountNode(T->rchild);

}

return n;

}

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

函数功能:得到父结点

函数参数:二叉树T的引用，待得到父结点的结点的关键字

函数返回值:返回二叉树结点的指针类型

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

BiTNode \*GetParent(BiTree&T,KeyType e)

{

if(T==NULL)return NULL;

else

{

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

if(T->lchild!=NULL&&T->lchild->data.key==e) return T;

if(T->rchild!=NULL&&T->rchild->data.key==e) return T;

BiTree p=GetParent(T->lchild,e);

BiTree q=GetParent(T->rchild,e);

if(p!=NULL)return p;

if(q!=NULL)return q;

}

}//得到关键字为e的结点的父结点

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

函数功能:在二叉链表的管理表中添加一个二叉链表

函数参数:引用二叉链表的管理表Trees的地址，待添加二叉链表的名称FileName

函数返回值:返回int类型，添加成功返回1

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

status AddTree(LISTS &Trees,char TreeName[])

{

strcpy(Trees.elem[Trees.length].name,TreeName);

int r=-1;

TElemType trees\_definition[100];

do{

r++;

scanf("%d%s",&trees\_definition[r].key,trees\_definition[r].others);

}while(trees\_definition[r].key!=-1);

i=0;

CreateBiTree(Trees.elem[Trees.length].T,trees\_definition);

Trees.length++;

return OK;

}

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

函数功能:在二叉链表的管理表中删除一个二叉链表

函数参数:引用二叉链表的管理表Trees的地址，待删除二叉链表的名称FileName

函数返回值:返回int类型，删除成功返回1

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

status RemoveTree(LISTS &Trees,char TreeName[])

{

for(int j=1;j<=Trees.length;j++)

{

if(strcmp(TreeName,Trees.elem[j-1].name)==0)

{

ClearBiTree(Trees.elem[j-1].T);

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

{

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

Trees.elem[i-1].T=Trees.elem[i].T;

}

Trees.length--;

return OK;

}

}

return ERROR;

}

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

函数功能:在二叉链表的管理表中查找二叉链表

函数参数:二叉链表的管理表Trees，待查找二叉链表的名称FileName

函数返回值:返回int类型，查找成功返回该二叉链表的位置

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

int LocateTree(LISTS Trees,char TreeName[])

{

for(int i=1;i<=Trees.length;i++)

if(strcmp(TreeName,Trees.elem[i-1].name)==0)

{

T=Trees.elem[i-1].T;

return i;

}

return ERROR;

}

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

函数功能:在函数中输出二叉树中从每个叶子结点到根结点的路径

函数参数:二叉树T，存放路径的指向二叉结点的指针类型数组path，路径长度pathlen

函数返回值:返回

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

void PrintPath(BiTree T,BiTree \*path,int pathlen)

{

int i;

if(T!=NULL)

{

path[pathlen]=T;

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

{

for(i=pathlen;i>=0;i--)

printf("%d,%s ",path[i]->data.key,path[i]->data.others);

printf("\n");

}

else

{

PrintPath(T->lchild,path,pathlen+1);

PrintPath(T->rchild,path,pathlen+1);

}

}

}

# 附录D 基于邻接表图实现的源程序