

**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201609**

**学 号： U2016XXXX**

**姓 名： 通 哥 哥**

**指导教师： 袁 凌**

**报告日期： 2017年 1月 19 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](#_Toc504127748)

[1.1 问题描述 3](#_Toc504127749)

[1.2 系统设计 5](#_Toc504127753)

[1.3 系统实现 9](#_Toc504127756)

[1.4 实验小结 18](#_Toc504127760)

[2 基于链式存储结构的线性表实现 20](#_Toc504127761)

[2.1 问题描述 20](#_Toc504127762)

[2.2 系统设计 22](#_Toc504127766)

[2.3 系统实现 27](#_Toc504127770)

[2.4 实验小结 36](#_Toc504127774)

[3 基于二叉链表的二叉树实现 37](#_Toc504127775)

[3.1 问题描述 37](#_Toc504127776)

[3.2 系统设计 40](#_Toc504127780)

[3.3 系统实现 45](#_Toc504127784)

[3.4 实验小结 61](#_Toc504127788)

[4 基于邻接表的图实现 62](#_Toc504127789)

[4.1 问题描述 62](#_Toc504127790)

[4.2 系统设计 64](#_Toc504127794)

[4.3 系统实现 70](#_Toc504127798)

[4.4 实验小结 84](#_Toc504127802)

[参考文献 85](#_Toc504127803)

[附录A 基于顺序存储结构线性表实现的源程序 88](#_Toc504127804)

[附录B 基于链式存储结构线性表实现的源程序 95](#_Toc504127805)

[附录C 基于二叉链表二叉树实现的源程序 103](#_Toc504127806)

[附录D 基于邻接表图实现的源程序 116](#_Toc504127807)

# 1 基于顺序存储结构的线性表实现

## 1.1 问题描述

采用顺序表的物理结构，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、销毁表、清空表、判定空表、求表长和获得元素等基本运算对应的函数，并给出适当的操作提示显示，可选择以文件的形式进行存储和加载，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 线性表的基本概念

线性表是最常用且最简单的一种数据结构，即n个数据元素的有限序列。线性表中元素的个数n定义为线性表的长度，n=0时成为空表。在非空表中的每个数据元素都有一个确定的位置，如a1是第一个数据元素，an是最后一个数据元素，ai是第i个数据元素。线性表的存储结构分为线性存储和链式存储。

## 1.1.2 逻辑结构与基本运算

线性表的数据逻辑结构定义如下:

ADT List｛

数据对象：D=｛ai|ai∈ElemSet，i=1，2，……，n，n≥0｝

数据关系：R1=｛ <ai-1，ai> | ai-1，ai∈D，i=2，……，n｝

｝

依据最小完备性和常用性相结合的原则，以函数形式定义了包括线性表的初

始化表、加载表、保存表、销毁表、清空表、判定空表、求表长、获得元素、查

找元素、获得前驱、获得后继、插入元素、删除元素、遍历表 14 个基本运算，

要求分别定义函数来实现上述功能，具体功能运算如下：

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

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已存在且非空，1≤i≤ListLength(L)+1；操作结果是在L的第i个位置之前插入新的元素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文件读取：函数名称是SaveData (L)，初始条件是线性表L已存在；操作结果是从“file”文件中读取数据并按顺序保存到线性表中。

⒁文件保存：函数名称是LoadData (L)，初始条件是线性表L已存在；操作结果是将当前线性表中的数据保存到“file”文件中。

⒂切换顺序表：函数名称是ChangeSqList(&L,Lists,&current)，初始条件是线性表L已存在；操作结果是切换到用户所要求的线性表中。

## 1.1.3 演示系统与数据文件

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，并给出适当的操作提示显示。附录 A 提

供了简易菜单的框架。

本演示系统实现线性表的文件形式保存。其中，①需要设计文件数据记

录格式，以高效保存线性表数据逻辑结构(D,{R})的完整信息；②需要设计线性

表文件保存和加载操作合理模式。附录 B 提供了文件存取的方法。

本演示系统完美实现多个线性表管理。

## 1.2 系统设计

1.2.1 数据结构的定义

#define LIST\_INIT\_SIZE 100

#define LISTICREMENT 10

typedef struct {

ElemType \*elem;

int length;

int listsize;

}SQList

typedef struct {

char c; int d; float f;

} ElemType;

1.2.2系统框架

本系统大体上可以分为三个部分。

第一部分是获取线性表的属性，包括求出表长、求第i个位置元素的值、求出某元素的前驱、后继的值、获取元素序位、遍历线性表；

第二部分是对线性表进行操作，包括初始化线性表、销毁线性表、清空线性表、插入一个元素、删除一个元素；

第三部分是附加部分，分为多线性表管理和文件形式的保存。需要说明的是，多线性表管理和文件形式保存两者功能室并行的，可以互相使用，但是文件中只能保存一张表。

系统结构图如图1.1所示：



图1.1 系统结构图

**1.2.3 算法的设计**

1. 初始化线性表

算法思路：开辟一个存储区并将首地址赋值给空指针L，同时将length属性值设置为0，listsize属性值设置为MAXLEN。

2. 销毁线性表

算法思路：释放开辟的空间，同时将指针设置为NULL。

3. 清空线性表

算法思路：直接将length属性值设置为0，之前占用的成为自由区。

4. 判断线性表是否为空

算法思路：如果length属性值为0，则为空，否则不为空

5. 求表长

算法思路：返回length属性值。

6. 获取元素

**Input**:元素的序位i

**Output**:元素的值

**Procedure**:

Step1:判断线性表是否为空

Step2：直接返回数组第i-1个元素的值

7. 获取序位

**Input**:元素的值

**Output**:元素的序位

**Procedure**：

Step1:判断序位是否合法

Step2:从0—length-1开始循环查找，如果成功则找到，否则查找失败

8. 获取前驱元素

算法思路：先判断是否存在前驱元素，接下来过程和获取序位类似，从0—length-1开始遍历，如果找到，则返回前一个元素的值。

9. 获得后继元素

算法思路：先判断是否存在后继元素，接下来过程和获取序位类似，从0—length-1开始遍历，如果找到，则返回后一个元素的值。

10. 插入元素

**Input:**元素e和序位i

**Output:**插入后的顺序表

**Procedure:**

Step1:判断顺序表是否存在，不存在则出错。

Step2:判断序位i是否合法，不合法则出错。

Step3:从最后一个元素开始，依次将后面n-i个元素向后移动一个位置。

Step4:将空出来的位置插入为e。

Step5:将顺序表的length属性值+1。

算法流程图如图1.2：



图1.2 插入元素算法流程图

11. 删除元素

**Input:**序位i

**Output:**删除第i个元素后的顺序表

**Procedure:**

Step1:判断顺序表是否存在，不存在则出错。

Step2:判断序位i是否合法，不合法则出错。

Step3:需要删除的元素位置的下一个开始，依次将下一个元素覆盖为上一个元素

Step4:将顺序表的length属性值-1。

算法流程图如图1.3。

12. 遍历表

算法思路：从第一个元素开始，依次访问每个元素。

13. 加载文件

首先需要构建存储结构，因为是线性表，已经省去了线性化过程，可以直接将数组中的元素一一对应。为了安全性，程序内部将存储文件封装起来放在根目录下，命名为file文件。元素内容紧密相连，利用循环，每次取出sizeof(ElemType)字节的数据，然后调用ListInsert函数，将新产生的元素插入到第length+1的位置上去。



图1.3 删除元素算法流程图

14. 保存文件

算法思路：首先检查线性表是否存在，如果存在，那么以”wb”的方式打开一个文件，命名为file。线性表本身就是线性的，那么我们就省去了线性化过程，因为数据存储必须是线性的。然后循环length次，每一次过程中，都将这一次所对应的序位的数据保存在file文件中。循环结束，关闭文件。

15. 多表管理

算法思路：在全局变量中定义一个SqList \* 类型的数组，程序中假定的数组的最大值为10，也就是说能够管理10个线性表。然后再主程序中定义一个SqList\*类型的线性表L，L的意义是当前线性表的地址。那么每当切换线性表的时候，只需要将线性表数组中对应的地址取出来赋值给L，那么L就代表了数组中对应的线性表，通过对L进行操作，即实现了对线性表的多表管理。

## 1.3 系统实现

## 1.3.1 编程环境、运行环境、项目工程描述

本次实验采用JetBrain CLion编程软件编写，并用gcc进行编译运行，项目名称是SquenceList。

## 1.3.2 头文件及预定义常量说明

1.头文件

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

2.预定义常量

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

3.类型表达式

typedef int Status;

typedef struct ElemType{

char c;

int d;

float f;

}ElemType;//基础数据类型

注：程序代码详见附录

1.3.3 演示系统操作

系统运行主界面如图1.4所示：

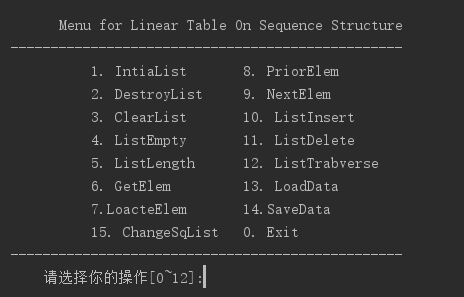


图1.4 主界面

功能1：初始化如图1.5：



图1.5 初始化

功能2：插入元素：

注：为提高程序的健壮性，提高测试标准，第四个数据元素实际上是不合法的，这是用来检测程序识别不合法数据的能力。

表1.1 插入元素样例

|  |  |  |  |
| --- | --- | --- | --- |
| 元素位序 | C值 | D值 | F值 |
| 1 | A | 5 | 3.14 |
| 2 | B | 8 | 2.718 |
| 3 | C | 100 | 1.44 |
| 5 | Q | 888 | 1.01 |

结果如图1.6：



图1.5 插入元素测试样例

功能3：判断线性表是否为空，如图1.6：

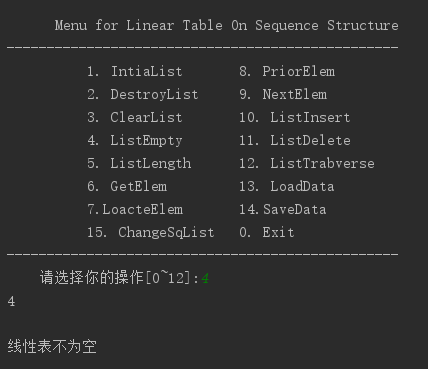


图1.6 线性表为空

功能4：求表长，如图1.7：

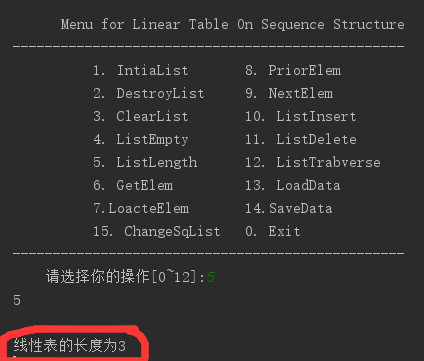


图1.7 求表长

功能5：通过位序获取元素，输入3，结果如图1.8：



图1.8 获取元素

功能6：给出元素，获得其在线性表中的位序，输入第一号元素的各个值，其位序是1，结果如图1.9：



图1.9 获取序位

功能7：获取前驱和后继，输入2号元素：B 8 2.718,结果如图1.10：



图1.10（a） 获得前驱、后继元素

这里，求第一号元素的前驱和最后一号元素的后继，是用来检测程序的健壮性，理论上应该给出错误提示信息。

测试失败样例如图：



图 1.10（b）测试失败样例

功能8：遍历元素，将其打印出来，如图1.11：

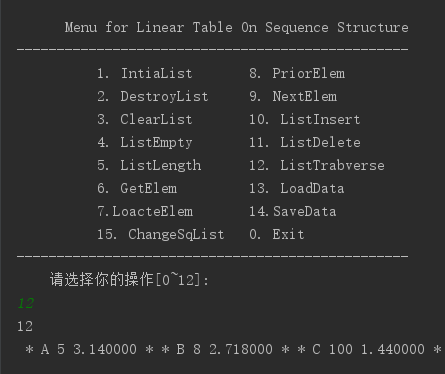


图1.11 遍历线性表

功能9：删除元素，选择删除第二个元素，如图1.12：



图1.12（a） 删除第2个元素

为增强程序的健壮性，我们故意删除一个不存在的元素，比如第8个元素。测试结果如图1.12(b)：



图1.12（b）测试失败样例

我们重新调用一遍GetElem函数，结果如图1.13：



图1.13 获取新的2号位元素

功能10：调用清空线性表，然后查看表长，如图1.14：

清空线性表后，数据全部为零，原来的数据区变成自由区，长度自动设置为0.但是线性表并没有被销毁，内存还正在被占用。

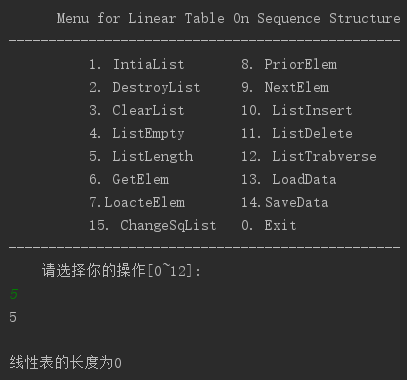


图1.14 清空线性表操作

附加功能11：文件保存和读取，数据还取自于上面的，先对数据进行存盘，调用14，然后退出程序，重新进入并初始化后，调用13，再进行遍历，操作结果如图1.15，图1.16，图1.17：



图1.15 数据存盘

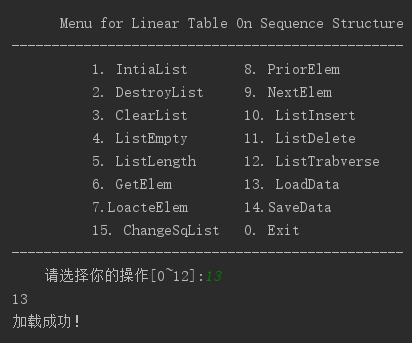


图1.16 数据加载

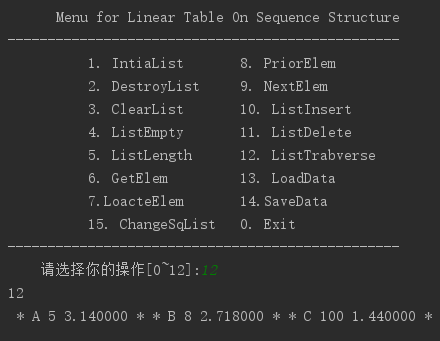


图1.17 遍历顺序表

附加功能12：多表管理：系统默认进入第一个线性表，如果切换则输入15，切换到1—10中任意一张表，比如我们切换到第5张表，对它初始化，输入一些数据，对其进行遍历，输入数据如表1.2：

表1.2 第5张表的数据

|  |  |  |  |
| --- | --- | --- | --- |
| 元素位序 | C值 | D值 | F值 |
| 1 | X | 1998 | 0.01 |
| 2 | Y | 4 | 0.11 |
| 3 | Z | 1 | 1.11 |

操作结果如图1.18、图1.19、图1.20：



图1.18 创建新表5

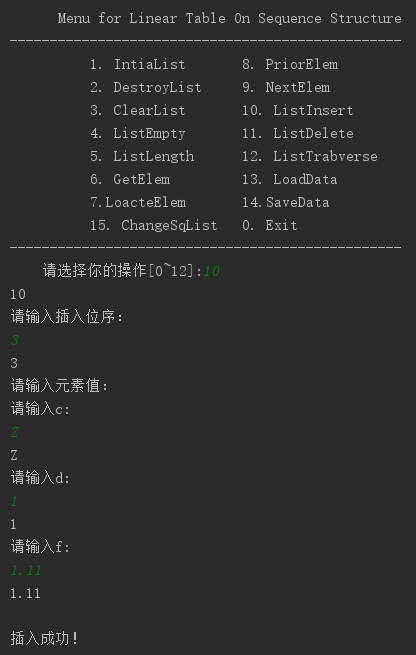


图1.19 输入元素

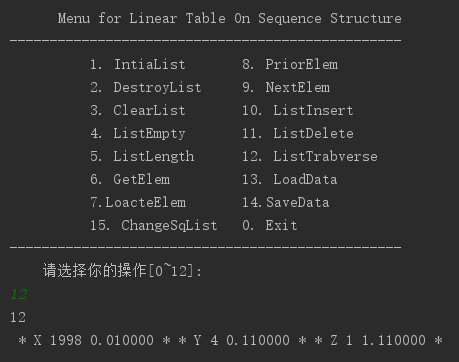


图1.20 遍历新表5

至此，演示系统就结束了！

## 1.4 实验小结

本次实验是数据结构课程的第一个实验，难度不是很大。重点是理解顺序表的物理结构，学会使用相关的方法，实现相应的算法。其中插入和删除等操作应该重点掌握，理解移动元素的方法，以及算法的时间复杂度等等。实现过程中，越界条件很重要，这增加了程序的稳定，符合算法设计的要求之一——健壮性。

实验过程虽然很顺利，但是还有一些问题，比如处理边界条件上。调用DeleteList函数时，如果序位传了负数，那么应当有相应的提示。事实上，程序还可以更加完善，当出现不同的错误时，应该返回不同的状态码，然后调用者根据不同的状态码来处理不同的异常操作。

还有一点，程序的界面应该设计得美观，提高程序交互性，这样才能使程序更加完美。

# 2 基于链式存储结构的线性表实现

## 2.1 问题描述

通过实验达到⑴加深对线性表的概念、基本运算的理解；⑵熟练掌握线性表的逻辑结构与物理结构的关系；⑶物理结构采用单链表,熟练掌握线性表的基本运算的实现。

## 2.1.1线性表的基本概念

线性表的链式存储结构的特点是一组任意的存储单元存储线性表的数据元素（这组存储单元可以是连续的，也可以是不连续的）。因此，为了表示每个数据元素与其直接后继的逻辑关系，对数据元素来说，存储本身信息的同时，还需要存储存储其直接后继的指针。它包含两个域，存储信息的域叫作数据域，存储下一节点地址的域叫作指针域。

2.1.2 逻辑结构

线性表的数据逻辑结构定义如下:

ADT List｛

数据对象：D=｛ai|ai∈ElemSet，i=1，2，……，n，n≥0｝

数据关系：R1=｛ <ai-1，ai> | ai-1，ai∈D，i=2，……，n｝

｝

依据最小完备性和常用性相结合的原则，以函数形式定义了包括线性表的初

始化表、加载表、保存表、销毁表、清空表、判定空表、求表长、获得元素、查

找元素、获得前驱、获得后继、插入元素、删除元素、遍历表 14 个基本运算，

要求分别定义函数来实现上述功能，具体功能运算如下：

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

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已存在且非空，1≤i≤ListLength(L)+1；操作结果是在L的第i个位置之前插入新的元素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文件读取：函数名称是SaveData (L)，初始条件是线性表L已存在；操作结果是从“file”文件中读取数据并按顺序保存到线性表中。

⒁文件保存：函数名称是LoadData (L)，初始条件是线性表L已存在；操作结果是将当前线性表中的数据保存到“file”文件中。

⒂切换链表：函数名称是ChangeLinkedList(&L,Lists,&current)，初始条件是线性表L已存在；操作结果是切换到用户所要求的线性表中。

## 2.1.3 演示系统与数据文件

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，并给出适当的操作提示显示。附录 A 提

供了简易菜单的框架。

本演示系统实现线性表的文件形式保存。其中，①需要设计文件数据记

录格式，以高效保存线性表数据逻辑结构(D,{R})的完整信息；②需要设计线性

表文件保存和加载操作合理模式。附录 B 提供了文件存取的方法。

本演示系统已经实现多个线性表管理。

## 2.2 系统设计

2.2.1 数据结构的物理结构

typedef struct{

struct node \* head;//头指针，指向表头节点

struct node \* tail;//尾指针，指向表尾节点

int length; //当前表长度，不包括表头节点

}LinkedList;

typedef struct node{

ElemType data; //数据域

struct node \* next;//节点域

}node;

2.2.2系统框架

本系统大体上可以分为三个部分。

第一部分是获取线性表的属性，包括求出表长、求第i个位置元素的值、求出某元素的前驱、后继的值、获取元素序位、遍历线性表；

第二部分是对线性表进行操作，包括初始化线性表、销毁线性表、清空线性表、插入一个元素、删除一个元素；

第三部分是附加部分，分为多线性表管理和文件形式的保存。需要说明的是，多线性表管理和文件形式保存两者功能室并行的，可以互相使用，但是文件中只能保存一张表。

系统结构图如图1.1所示：



图2.1 系统结构图

2.2.3 运算算法思想与设计

1. 初始化线性表

算法思路：开辟一个存储区并将首地址赋值给空指针L，同时将length属性值设置为0，head属性设置为NULL，tail属性设置为NULL。

2. 销毁线性表

算法思路：释放开辟的空间，同时将指针设置为NULL。

3. 清空线性表

算法思路：释放所有的结点空间，同时设置链表的length属性值为0，head指针设置为NULL，tail指针设置为NULL。

4. 判断线性表是否为空

算法思路：如果length属性值为0，则为空，否则不为空

5. 求表长

算法思路：返回length属性值。

6. 获取元素

**Input**:元素的序位i

**Output**:元素的值

**Procedure**:

Step1:判断线性表是否为空，如果为空返回状态码。

Step2:从头指针开始计数，如果计数的值等于i，则返回头指针所指元素的值，否则移动指针到下一个结点。

7. 获取序位

**Input**:元素的值

**Output**:元素的序位

**Procedure**：

Step1:判断序位是否合法

Step2:从0—length-1开始循环查找，如果成功则找到，否则查找失败

8. 获取前驱元素

算法思路：先判断是否存在前驱元素，接下来过程和获取序位类似，从0—length-1开始遍历，如果找到，则返回前一个元素的值。此算法需要使用两个变量p，q来指示当天元素和当前元素的前一个元素。

9. 获得后继元素

算法思路：先判断是否存在后继元素，接下来过程和获取序位类似，从0—length-1开始遍历，如果找到，则返回后一个元素的值。此算法需要使用两个变量p，q分别指示当前变量和当前变量的下一个元素。

10. 插入元素

**Input:**元素e和序位i

**Output:**插入后的链表

**Procedure:**

Step1:判断顺序表是否存在，不存在则出错。

Step2:判断序位i是否合法，不合法则出错。

Step3:开辟新空间，并将数据域赋值成为e，指针域设置为NULL；

Step4:从第一个元素开始，将指针p从头结点依次移动i次，然后执行f->next = p->next;p->next = f;

Step5:将链表的length属性值+1。

算法流程图如图1.2：



图2.2 插入元素算法流程图

11. 删除元素

**Input:**序位i

**Output:**删除第i个元素后的顺序表

**Procedure:**

Step1:判断顺序表是否存在，不存在则出错。

Step2:判断序位i是否合法，不合法则出错。

Step3:执行q->next = p->next,free(p);其中p指向当前元素，q指向p的上一个元素

Step4:将链表的length属性值-1。

算法流程图如图1.3。

12. 遍历表

算法思路：从第一个元素开始，依次访问每个元素。

13. 加载文件

首先需要构建存储结构，因为是线性表，已经省去了线性化过程，可以直接将数组中的元素一一对应。为了安全性，程序内部将存储文件封装起来放在根目录下，命名为file文件。元素内容紧密相连，利用循环，每次取出sizeof(ElemType)字节的数据，然后调用ListInsert函数，将新产生的元素插入到第length+1的位置上去。



图1.3 删除元素算法流程图

14. 保存文件

算法思路：首先检查线性表是否存在，如果存在，那么以”wb”的方式打开一个文件，命名为file。线性表本身就是线性的，那么我们就省去了线性化过程，因为数据存储必须是线性的。然后循环length次，每一次过程中，都将这一次所对应的序位的数据保存在file文件中。循环结束，关闭文件。

15. 多表管理

算法思路：在全局变量中定义一个LinkedList \* 类型的数组，程序中假定的数组的最大值为10，也就是说能够管理10个线性表。然后再主程序中定义一个LinkedList\*类型的线性表L，L的意义是当前线性表的地址。那么每当切换线性表的时候，只需要将线性表数组中对应的地址取出来赋值给L，那么L就代表了数组中对应的线性表，通过对L进行操作，即实现了对线性表的多表管理。

## 2.3 系统实现

## 2.3.1 编程环境、运行环境、项目工程描述

本次实验采用JetBrain CLion编程软件编写，并用gcc进行编译运行，项目名称是LinkedList。

## 2.3.2 头文件及预定义常量说明

1.头文件

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

2.预定义常量

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

3.类型表达式

typedef int Status;

typedef struct int ElemType;//基础数据类型

注：详细代码见附录

2.3.3演示系统操作

系统运行主界面如图1.4所示：



图2.4 主界面

功能1：初始化如图1.5：



图2.5 初始化

功能2：插入元素：

注：为提高程序的健壮性，提高测试标准，第四个数据元素实际上是不合法的，这是用来检测程序识别不合法数据的能力。

表2.1 插入元素样例

|  |  |
| --- | --- |
| 元素位序 | 元素值 |
| 1 | 100 |
| 2 | 200 |
| 3 | 300 |
| 5 | 500 |

结果如图2.6：



图2.6 插入元素测试样例

功能3：判断线性表是否为空，如图2.7：

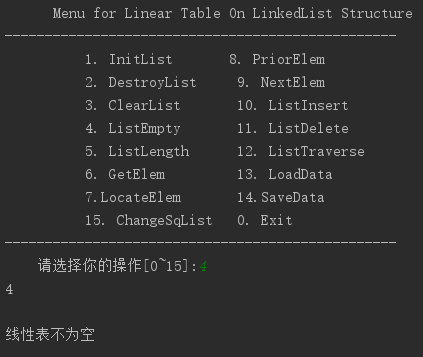


图2.7 判断线性表是否为空

功能4：求表长，如图2.8：



图2.8 求表长

功能5：通过位序获取元素，输入3，结果如图2.9：



图2.9 获取元素

功能6：给出元素，获得其在线性表中的位序，输入第一号元素的各个值，其位序是1，结果如图2.10：



图2.10 获取序位

功能7：获取前驱和后继，输入2号元素：200，结果如图2.11：



图2.10（a） 获得前驱、后继元素

这里，求第一号元素的前驱和最后一号元素的后继，是用来检测程序的健壮性，理论上应该给出错误提示信息。

测试失败样例如图2.10：



图 2.10（b）测试失败样例

功能8：遍历元素，将其打印出来，如图2.11：



图2.11 遍历线性表

功能9：删除元素，选择删除第三个元素，如图2.12：



图2.12（a） 删除第3个元素

为增强程序的健壮性，进行删除一个不存在的元素，比如第8个元素。测试结果如图2.12(b)：



图2.12（b）测试失败样例

我们重新调用一遍ListLength函数，结果如图2.13：



图2.13 求表长

功能10：调用清空线性表，然后查看表长，如图2.14：

清空线性表后，结点数据全部被清空，长度自动设置为0.但是线性表并没有被销毁，head指针设置成NULL，tail指针设置为NULL。

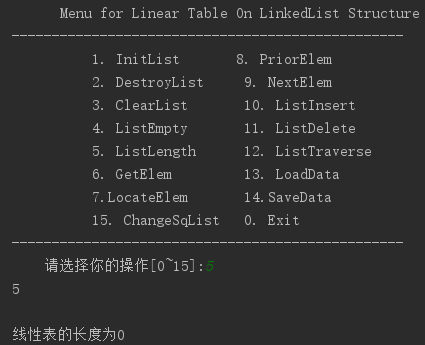
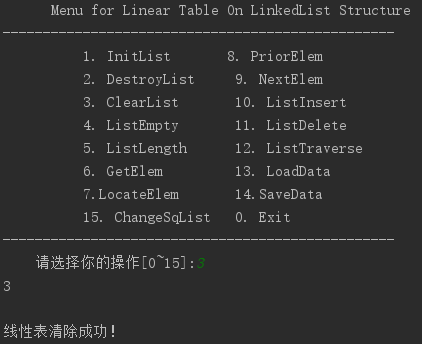


图2.14 清空线性表操作

附加功能11：文件保存和读取，数据还取自于上面的，先对数据进行存盘，调用14，然后退出程序，重新进入并初始化后，调用13，再进行遍历，操作结果如图2.15，图2.16，图2.17：



图2.15 数据存盘

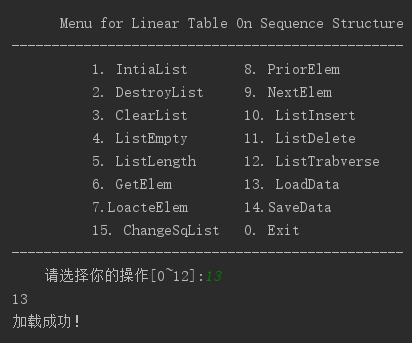


图2.16 数据加载

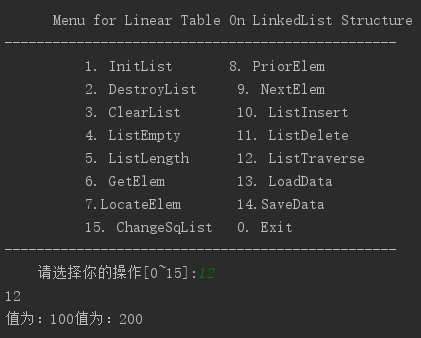


图2.17 遍历链表

附加功能12：多表管理：系统默认进入第一个线性表，如果切换则输入15，切换到1—10中任意一张表，比如我们切换到第5张表，对它初始化，输入一些数据，对其进行遍历，输入数据如表1.2：

表1.2 第5张表的数据

|  |  |
| --- | --- |
| 元素位序 | 元素值 |
| 1 | 123 |
| 2 | 456 |
| 3 | 789 |

操作结果如图1.18、图1.19、图1.20：

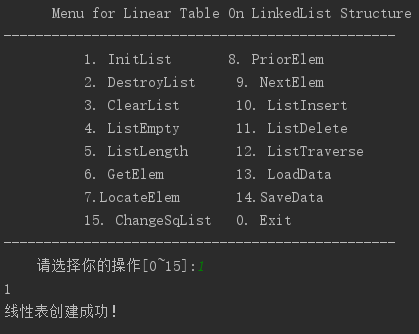


图1.18 创建新表5

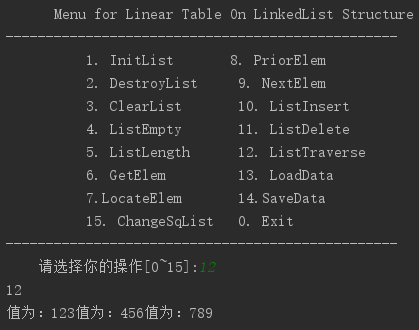


图1.19 输入元素

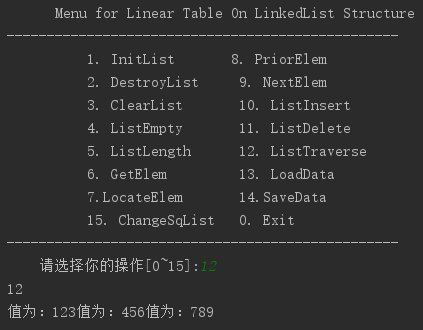


图1.20 遍历新表5

至此，演示系统就结束了！

## 2.4 实验小结

本次实验是数据结构第二次实验，内容是线性表的链式结构。实验难度较上一次有所增加，其中插入元素和删除元素的算法和顺序表的方法完全不一样。链式结构的插入元素首先要找到插入点，然后修改指针即可，而顺序表进行插入怎要移动元素。删除时，链表需要注意的是，要设置两个指针，其中一个指示当前指针的上一个，因为单链表的特点，不能知道该结点的上一个结点，而只能知道该结点的下一个结点。

还有一点，单链表因为加入了指针操作，导致程序的错误率比较高，而我写程序的时候，出错比较多的是有没有处理悬空指针、指针赋值错误等等，这些都是些细节的东西，稍有不慎就会出现Bug，因此，写程序需要细心。

# 3 基于二叉链表的二叉树实现

## 3.1 问题描述

采用二叉链表作为二叉树的物理结构，实现二叉树的基本运算，数据元素的类型名可自行定义。要求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，并给出适当的操作提示。

3.1.1 二叉树的基本概念

二叉树是一种树型结构，即n个结点的有限集，它的特点是每个结点至多只有两棵子树（即二叉树中不存在度大于2的结点），并且，二叉树的子树有左右之分，其次序不能任意颠倒。

3.1.2 逻辑结构与基本运算

抽象数据类型二叉树的定义如下：

ADT BinaryTree {

数据对象D：D是具有相同特性的数据元素的集合。

数据关系R:

若D=Φ，则R=Φ，称BinaryTree为空二叉树；

若D≠Φ，则R={H},H是如下二元关系：

1. 在D中存在唯一的成为根的数据元素root，它在关系H中无前驱；
2. 若D-{root}≠Φ,则存在D-{root}={D1,Dr},且D1∩Dr=Φ;
3. 若D1≠Φ，则D1中存在唯一的元素X1，<root,X1>∈H，且存在D1上的关系H1包含于H；若Dr≠Φ,则Dr中存在唯一的元素Xr，<root,Xr>∈H,且存在Dr上的关系属于H；
4. （D，{H1}）是一棵符合本定义的二叉树，称为根的左子树，（Dr，{Hr}）是一棵符合本定义的二叉树，称为根的右子树。

}

依据最小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初始化、销毁二叉树、创建二叉树、清空二叉树、判定空二叉树和求二叉树深度等20种基本运算，具体运算功能定义如下：

⑴初始化二叉树：函数名称是InitBiTree(T)；初始条件是二叉树T不存在；操作结果是构造空二叉树T。

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；操作结果是按definition构造二叉树T。

⑷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。

⑸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。

⑹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。

⑺获得根结点：函数名称是Root(T)；初始条件是二叉树T已存在；操作结果是返回T的根。

⑻获得结点：函数名称是Value(T,e)；初始条件是二叉树T已存在，e是T中的某个结点；操作结果是返回e的值。

⑼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T中的某个结点；操作结果是结点e赋值为value。

⑽获得双亲结点：函数名称是Parent(T,e) ；初始条件是二叉树T已存在，e是T中的某个结点；操作结果是若e是T的非根结点，则返回它的双亲结点指针，否则返回NULL。

⑾获得左孩子结点：函数名称是LeftChild(T,e)；初始条件是二叉树T存在，e是T中某个节点；操作结果是返回e的左孩子结点指针。若e无左孩子，则返回NULL。

⑿获得右孩子结点：函数名称是RightChild(T,e)；初始条件是二叉树T已存在，e是T中某个结点；操作结果是返回e的右孩子结点指针。若e无右孩子，则返回NULL。

⒀获得左兄弟结点：函数名称是LeftSibling(T,e)；初始条件是二叉树T存在，e是T中某个结点；操作结果是返回e的左兄弟结点指针。若e是T的左孩子或者无左兄弟，则返回NULL。

⒁获得右兄弟结点：函数名称是RightSibling(T,e)；初始条件是二叉树T已存在，e是T中某个结点；操作结果是返回e的右兄弟结点指针。若e是T的右孩子或者无有兄弟，则返回NULL。

⒂插入子树：函数名称是InsertChild(T,p,LR,c)；初始条件是二叉树T存在，p指向T中的某个结点，LR为0或1，,非空二叉树c与T不相交且右子树为空；操作结果是根据LR为0或者1，插入c为T中p所指结点的左或右子树，p 所指结点的原有左子树或右子树则为c的右子树。

⒃删除子树：函数名称是DeleteChild(T.p.LR)；初始条件是二叉树T存在，p指向T中的某个结点，LR为0或1。 操作结果是根据LR为0或者1，删除c为T中p所指结点的左或右子树。

⒄前序遍历：函数名称是PreOrderTraverse(T,Visit())；初始条件是二叉树T存在，Visit是对结点操作的应用函数；操作结果：先序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。

⒅中序遍历：函数名称是InOrderTraverse(T,Visit))；初始条件是二叉树T存在，Visit是对结点操作的应用函数；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。

⒆后序遍历：函数名称是PostOrderTraverse(T,Visit))；初始条件是二叉树T存在，Visit是对结点操作的应用函数；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。

⒇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。

## 3.1.3演示系统与数据文件

要求构造一个具有菜单的功能演示系统。其中，在主函数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，并给出适当的操作提示。附录A中给了简单菜单的框架。演示系统已经实现了线性表的文件形式保存、多个线性表的理这两个附加功能。

## 3.2 系统设计

3.2.1 数据的物理结构

typedef struct ElemType{

int key;//关键值

char c;//字符

}ElemType;//基础数据类型

typedef struct BiTree{

struct node \* root;

int numNode;//节点数

}BiTree; //二叉树

typedef struct node{

ElemType data;//数据域

struct node \* lchild;

struct node \* rchild;

}node;//结点数据类型

## 3.2.2 演示系统

将菜单演示和用户选择输入写入到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while循环后，系统首先显示功能菜单，然后提示用户输入选择（0-20），其中1-23对应二叉树的一个基本操作，分别对应一个函数，通过switch语句将用户输入的序号对应到相应的函数功能，执行完该语句后break跳出switch语句，执行while循环，直到用户输入0选择退出，退出系统。系统的框架如图3.1：

## 3.2.3 运算算法思想与设计

二叉树运算算法思想与设计如下：

1. 初始化二叉树算法思想：将初始化二叉树过程写成函数，函数的参数是主函数中所定义的结构类型指针T（参数传递为指针），T所指向的是二叉树的根结点，将T赋值为NULL即完成了二叉树的初始化。经分析，该

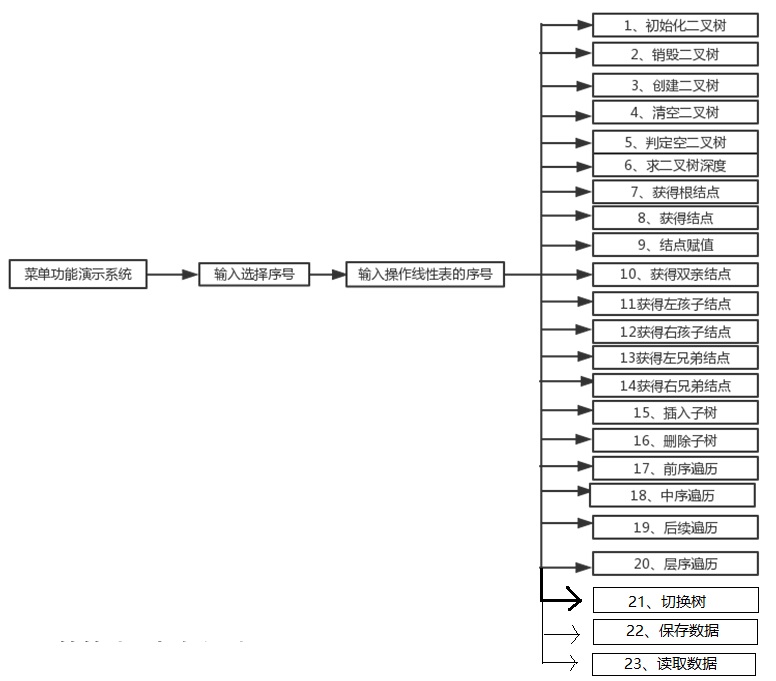


图 3.1 程序框架

算法的时间复杂度为O(1)。

2. 销毁二叉树算法思想：将二叉树的销毁过程写成函数，函数的参数是指向二叉树根结点的结构类型指针T，采取递归的方式先销毁二叉树的左子树，在销毁二叉树的右子树，最后用free函数释放掉根结点对应的内存空间。经分析，该算法的时间复杂度为O(n)。

3. 创建二叉树算法思想：将二叉树的创建过程写成函数，函数的参数是指向二叉树根结点的结构类型指针T，按照先序次序输入二叉树中结点的值，如果第一个字符为0，则T为空二叉树。否则，malloc函数分配一个单元的空间作为树的根结点，并为其赋值，采取递归的方式继续创建根结点的左子树和右子树。经分析，该算法的时间复杂度为O(n)。

4. 清空二叉树算法思想：将二叉树的清空过程写成函数，函数的参数同上，将根结点的左右孩子指针置为空，此时其左右子树的存储空间并未释放掉。经分析，该算法的时间复杂度为O(n)。算法流程图如图3.2：



图3.2 删除树

5. 判定空二叉树的算法思想：将判定空二叉树写成函数，对于一个二叉树，若根结点不存在则为空二叉树，否则不是空二叉树，那么只需要判断指向根结点的结构类型指针T是否为空即可。经分析，该算法的时间复杂度为O(1)。

6. 求二叉树的深度算法思想：将求二叉树的深度写成函数，采取递归的方式求二叉树的深度，如果根结点的左右孩子都不存在，则返回树的深度为1，否则的话返回根结点左右子树的深度的最大加上一。经分析，该算法的时间复杂度为O(n)。



图 3.3 求树的深度

7. 获得二叉树根结点的算法思想：将求二叉树根结点写成函数，传入头结点，若其头指针不为空，则返回该结点的关键字信息。经分析，该算法的时间复杂度为O(1)。

8. 获得结点的算法思想是：将获得关键字结点写成函数，函数的参数是二叉树的头指针以及主函数中输入的关键字，通过递归先序遍历二叉树，即先比较根结点的关键字与给定是否一致，若一致，返回该结点的INDEX值，否则继续分别递归遍历其左子树和右子树。或者采取非递归的方式，即使用堆栈来存储根结点的信息，先将根结点入栈，在循环中弹出栈顶元素，比较关键字，在分别将其右子树和左子树的根结点入栈。经分析，该算法的时间复杂度为O(n)。

9. 结点赋值的算法思想是：将结点赋值写成函数，函数的参数是二叉树的头指针，主函数中输入的关键字，根据关键字获取到根结点，并对根结点赋值，思想同8，不在赘述。经分析，该算法的时间复杂度为O(n)。

10. 获得双亲结点的算法思想是：将获得双亲结点写成函数，函数的参数是二叉树的头指针，若头指针为空，则返回空；否则，如果头指针左孩子或右孩子不为空且关键字与给定关键字相同，返回根结点，如果不同，采取递归的方式调用原函数，传入的参数分别为根结点的左右子树的根结点。经分析，该算法的时间复杂度为O(n)。

11. 获得左兄弟结点的算法思想是：将获得左孩子结点写成函数，函数的参数是二叉树的头指针，如果头指针为空，返回NULL；否则，如果二叉树的根结点的右孩子关键字与给定关键字一致，返回根结点的左孩子。如果不一致，继续递归调用原函数，传入的参数为二叉树根结点的左子树的根结点指针，如果函数的返回值非空，则返回该值，否则继续调用原函数，传入的参数是二叉树根结点的右子树的根结点指针，如果函数的返回值为非空，则返回该值，否则返回NULL。经分析，该算法的时间复杂度为O(n)。

12. 获得右兄弟结点的算法思想是:算法思想同上。经分析，该算法的时间复杂度为O(n)。

13. 获得左孩子结点的算法思想是：将获得左孩子结点写成函数，函数的参数是二叉树的头指针。如果根结点的关键字与给定关键字一致，且其左孩子存在，则返回其左孩子关键字，否则递归调用原函数分别将根结点的左右子树根结点指针作为形参，若返回值非空则返回该值。经分析，该算法的时间复杂度为O(n)。

14. 获得右孩子结点的算法思想是：算法思想同上。经分析，该算法的时间复杂度为O(n)。

15. 插入子树的算法思想是：将插入子树写成函数，函数的形参是二叉树的头指针T，指向特定二叉树结点的指针p（在主函数中要求输入关键字，通过FIND函数找到插入点位置并将其值记录），再调用CreateBiTNode函数创建根结点c的右子树为空的二叉树，插入的过程即将P所指向结点的左子树或者右子树根结点指针赋值给c的右孩子指针域，而c赋值给p所指结点的左指针域或者右指针域。经分析，该算法的时间复杂度为O(n)。

16. 删除子树的算法思想是：将删除子树写成函数，函数的形参是二叉树的头指针，特定结点的指针，根据指向特定结点的指针找到给结点并将其左右孩子指针域置为空，对应的还应该在这之前调用DESTROY函数将其左子树或右子树销毁。经分析，该算法的时间复杂度为O(n)。

17. 前序遍历的算法思想是：将前序遍历写成函数，函数的形参是二叉树的头指针，首先访问根结点，并对其执行遍历操作，操作成功后采取递归的方式遍历根结点的左子树，返回值为OK时继续遍历其右子树，最终遍历完成。递归方法前序遍历、中序遍历以及后序遍历的根本区别在与访问根结点的操作是在两次递归操作之间之前还是之后。经分许，该算法的时间复杂度为O(n)。

18. 非递归前序遍历的算法思想是：将非递归前序遍历写成函数，函数的形参是二叉树的头指针，仿照递归过程堆栈的使用，首先将根结点的值压入堆栈，执行循环体，堆栈非空，弹出根结点并执行访问操作，将其右子树根结点指针压入堆栈，再将其左子树根结点指针压入堆栈，执行循环。经分析，该算法的时间复杂度为O(n)。

19. 非递归中序遍历的算法思想是：将非递归中序遍历写成函数，函数的形参是二叉树的头指针，首先将根结点的值压入堆栈，然后一直向左，将根结点依次压入堆栈，判断堆栈非空，将栈顶元素弹出，访问该结点，如果其右子树非空，对其右子树执行循环操作。经分析，该算法的时间复杂度为O(n)。

20. 层序遍历的算法思想是：将层序遍历写成函数，函数的形参是二叉树的头指针，借助队列，使根结点进入队列，根结点出队列，执行访问操作，此时将根结点的左右子树根结点依次进入队列，即队列中的某个元素出队列时将其左右子树根结点放进队列，循环即可一层一层的访问二叉树。经分析，该算法的时间复杂度为O(n)。层序遍历流程图如图3.4：



图3.4 程序流程图

## 3.3 系统实现

## 3.3.1 编程环境、运行环境、项目工程描述

本次实验采用JetBrains CLion 2017.2编程软件编写，并用GCC进行编译运行，项目名称是BiTree。

## 3.3.2 头文件及预定义常量说明

1、头文件

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

2、预定义常量

（1）函数结果状态宏定义

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define OVERFLOW -2

（2）类型表达式

typedef struct ElemType{

int key;//关键值

char c;//字符

}ElemType;//基础数据类型

typedef struct BiTree{

struct node \* root;

int numNode;//节点数

}BiTree;

typedef struct node{

ElemType data;//数据域

struct node \* lchild;

struct node \* rchild;

}node;

## 3.3.3 演示系统操作

首先打开主界面如图3.4所示：



图3.5 主界面

默认进入的是第1棵树，首先需要对二叉树进行初始化，初始化操作如图3.5：



图3.6 初始化二叉树

选择第三个选项，创建一棵二叉树，二叉树模型如图3.7：



图3.7 树模型

程序按照前序方式输入序列，输入如图3.8：

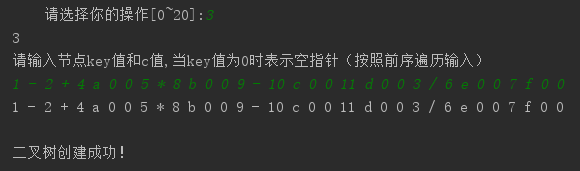


图3.8 创造树

首先把二叉树遍历一遍，遍历的方式有四种，分别是前序遍历、中序遍历、后序遍历、层序遍历。前序遍历是指：先遍历根节点，然后遍历左子树和右子树。中序遍历是指：先遍历左子树，然后遍历根节点，最后遍历右子树。后序遍历是指：先遍历左子树和右子树，然后遍历根节点。层序遍历是指：根据二叉树的深度，一层一层的遍历。

本程序二叉树的原型是根据《数据结构与算法分析》第129页的表达式二叉树模型，其中前序遍历结果也被称作波兰表达式，后序遍历结果被称作是逆波兰表达式。

理论上图3.7的二叉树四种遍历结果为- + a \* b – c d / e f ,

a + b \* c – d – e / f , a b c d - \* + e f / , - + / a \* e f b – c d。

程序遍历操作结果如图3.9：

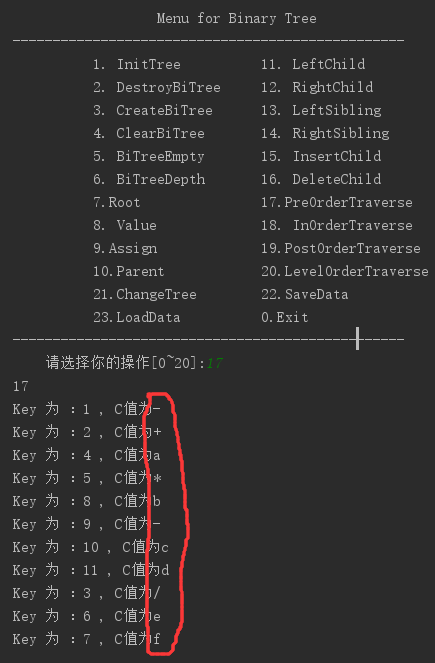


图3.9（a）前序遍历

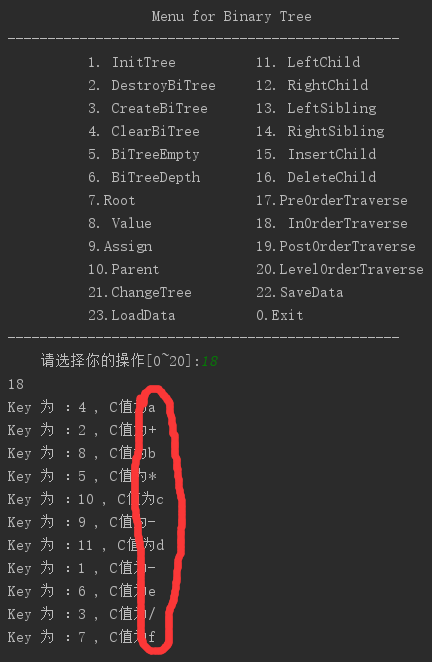


图3.9（b）中序遍历

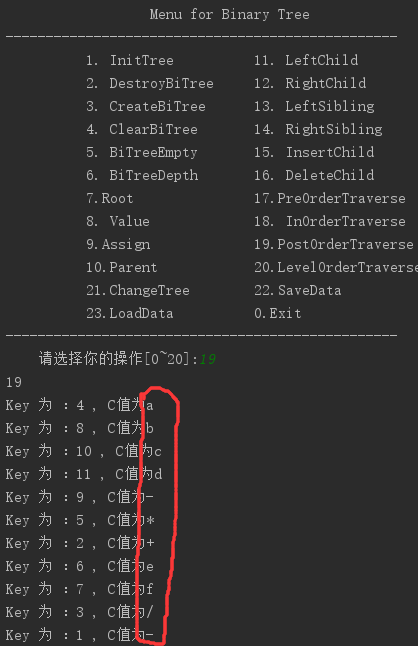


图3.9（c）后序遍历

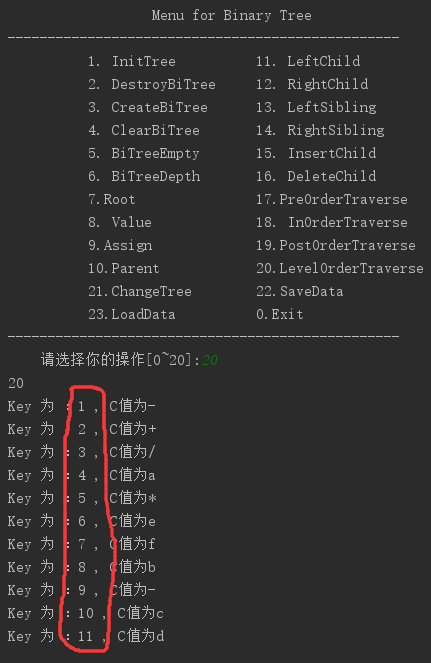


图3.9（d）层序遍历

从图中红色线圈出的即是遍历顺序，可以看出程序正确无误。

接下来判断二叉树是否为空，继续沿用上一个二叉树，理论上二叉树不为空，其结果如图3.10所示：



图3.10 判断二叉树是否为空

第四步求树的深度，根据图3.7所示的树模型，理论上树的深度是5，实际测试结果见图3.11所示：



图3.11 树的深度

输入第6号指令，返回二叉树的深度是5，结果和理论值相符合，测试正确。

第五步求根节点，根节点的值已知是“-”，其结果如图3.12所示：



图3.12 树根

树根为“-”，符合预期。

第六步根据关键字获取值，选取key值为4的节点，那么c值为a。在主菜单中输入8号指令，输入key值为4，结果如图3.13所示：

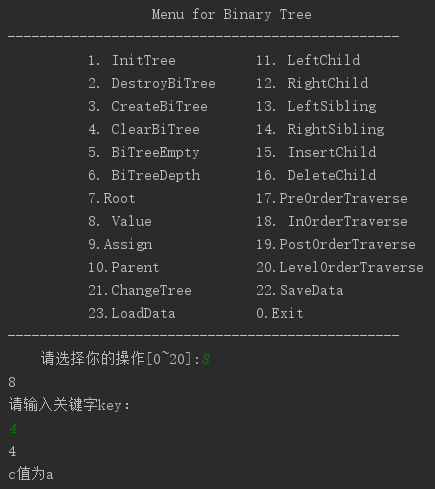


图3.13 根据关键字求值

上图所示的值为a，测试正确。

第七步是向图中节点赋值，输入9号指令，将key值为10的元素替换为“X”，操作过程及结果如图3.14所示：



图 3.14 节点赋值

为了验证程序的正确性，层序遍历这棵二叉树，原来a的位置被替换为X，理论上层序遍历的结果是- + / X \* e f b – c d，实际结果如图3.15所示：

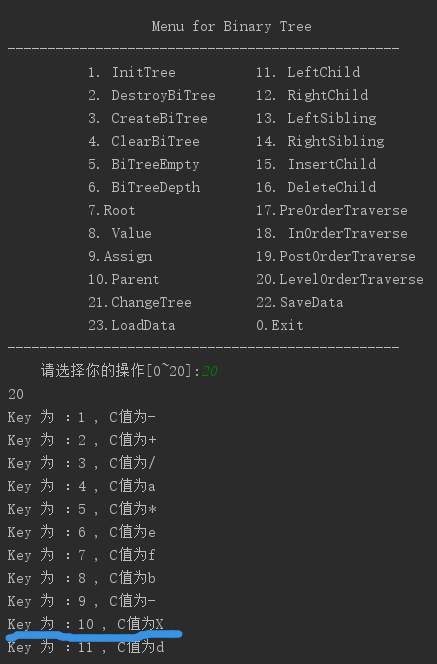


图3.15 重新层序遍历

图中划出来的一行由原来的a变为X，结果符合预期。

第八步是获取节点的双亲结点，沿用上面的二叉树，求key值为9的双亲结点，操作过程及结果如图3.16所示：



图3.16 key为10的双亲结点

从图3.7中的模型来看，key为10的双亲结点是9，结果正确。

第九步是获取左孩子节点，为了验证程序的正确性，我们选取key值为9的节点，求它的左孩子节点，操作结果如图3.17所示：

第十步是获取右孩子节点，key选取为9，求它的右孩子，结果如图3.7：

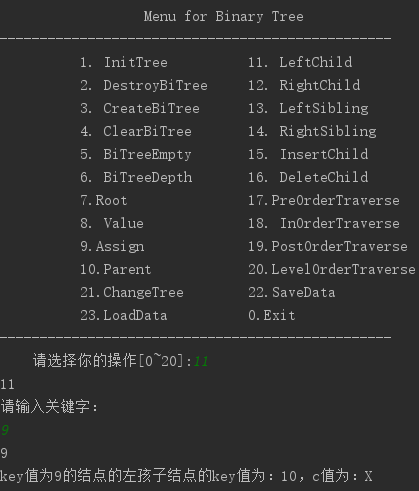


图3.17（a）获取左孩子



图3.17（b） 获取右孩子

为增强实验的鲁棒性，本程序还能处理操作者的错误指令，比如获取一个没有右孩子节点的右孩子，理论上应当返回相应的错误，其结果如图3.18所示：



图3.18 错误操作处理

第十一步和第十二步是获取节点的左兄弟和右兄弟，操作结果如图3.19：



图3.19（a） 获取左兄弟

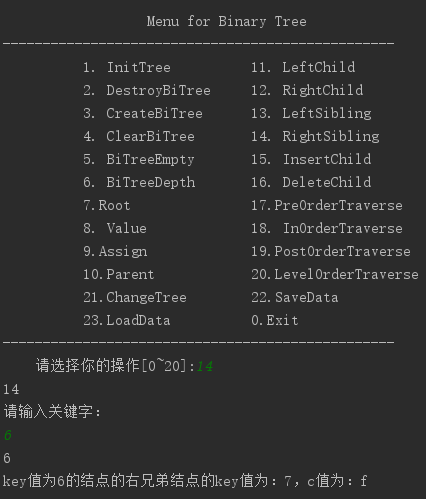


图3.19（b） 获取右兄弟

第十三步是删除孩子节点，选择删除key值为5的左孩子节点，操作结果如图3.20所示：



图3.20 删除5号节点的左孩子

为验证程序的正确性，进行一遍层序遍历，遍历结果如图3.21所示：

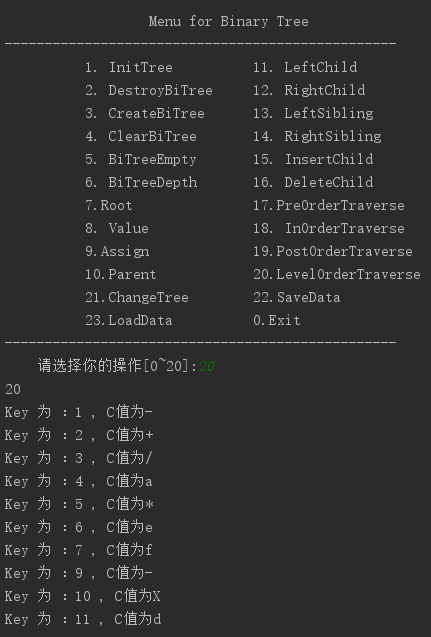


图3.21 遍历节点验证程序正确性

图中缺少了8号节点，有之前的树模型可知，程序正确。

第十四步插入孩子节点，我们将刚才被删除的节点补充回来，即key值为8，c值为b，操作过程及结果如图3.22所示：

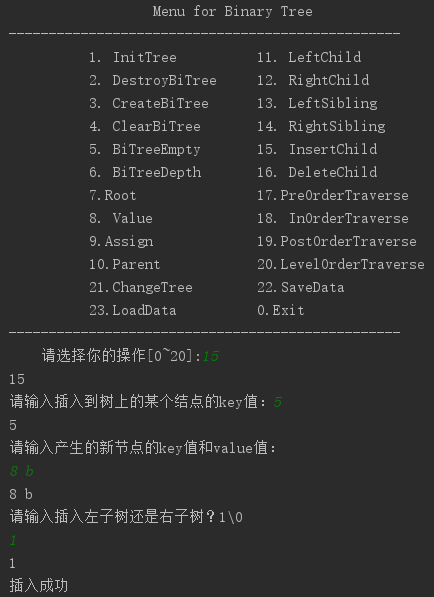


图3.22 插入节点

与删除操作同样，我们需要验证程序的正确性，进行层序遍历结果如图3.23：

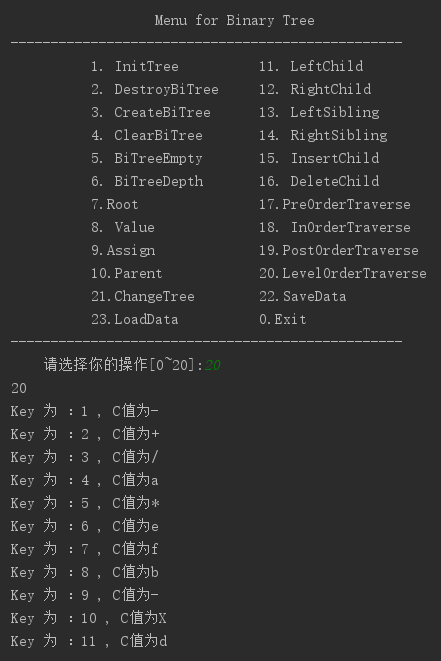


图3.23 层序遍历

第十五步是附加功能存盘、读取、多树管理。

首先把我们刚才构建好的二叉树存盘，操作结果如图3.24所示：



图 3.24 保存数据

然后切换到任意一棵二叉树，并进行初始化。将数据读入到新的二叉树中，操作结果如图3.25所示：



图3.25 切换二叉树

输入指令23，加载数据，如图3.26所示：



图3.26 加载数据

为了验证程序的正确性，我们需要进行一次前序遍历和层序遍历以确保加载后的二叉树是正确的，结果如图3.27所示：

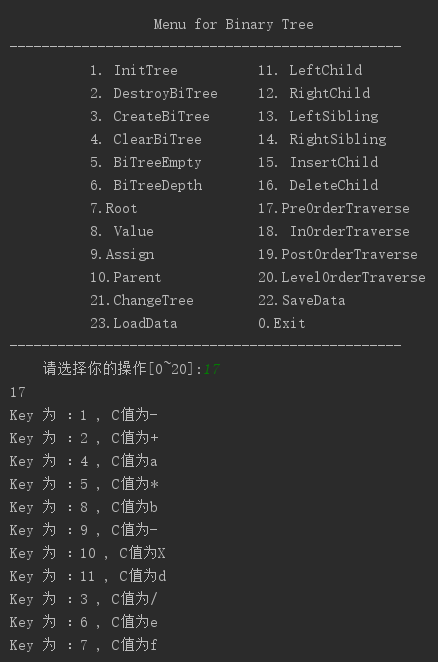


图3.27（a） 前序遍历

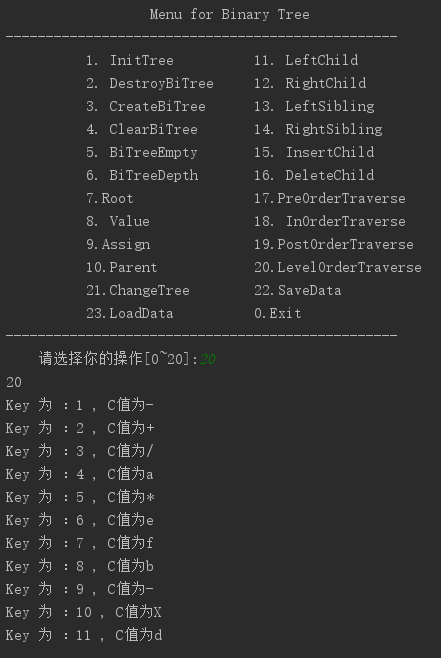


图3.27（b）层序遍历

从上图的结果看，结果完全正确。

第十六步是销毁二叉树，输入指令2，结果如图3.28所示：



图3.28 销毁二叉树

为了验证二叉树确实被销毁，调用5号指令判断二叉树是否为空，结果如图3.29：



图3.29 验证二叉树销毁

从图中看，二叉树已经为空，说明二叉树已经被销毁了。

至此，整个程序的功能演示完毕。

## 3.4 实验小结

二叉树的难度相较于两个线性表的难度有所上升，但是其中蕴含了许多编程思想。我觉得其中最重要的两个经典算法是前序遍历和层序遍历。因为剩下的算法都是围绕着这两个算法来展开拓延的。前序遍历教会了我们利用递归的形式操作二叉树，这是以二叉链表为基础的主要操作。比如求树的深度，我们可以先求左子树和右子树的深度，然后找到两者的最大值再加一即可得到这棵树的深度。还有获取孩子节点、兄弟节点等等算法都是以递归的思想来展开的，这里就不予赘述。另外一个是层序遍历，这种思路是完全不同于递归的思想。层序遍历用到了队列，每当一个元素出队列后，就将其两个孩子节点放到队列中，这样就可以实现层序遍历。这种思路来自于图的广度优先搜索，事实上二叉树就是一个简单图，层序遍历就是其进行广度优先搜索的结果。利用这一思想，我们还可以求出一棵二叉树的最大宽度等等。

# 4 基于邻接表的图实现

## 4.1 问题描述

通过实验达到⑴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。

4.1.1图的基本概念

一个图(G)定义为一个偶对(V,E) ，记为G=(V,E) 。其中： V是顶点(Vertex)的非空有限集合，记为V(G)；E是无序集V&V的一个子集，记为E(G) ，其元素是图的弧(Arc)。将顶点集合为空的图称为空图。

4.1.2 逻辑结构与基本运算

图形式化定义为：

ADT Graph {

数据对象：G=(V ，E); V={v|data object}; E={<v,w>| v,w∈V∧p(v,w)}

数据关系R:

P(v,w)表示从顶点v到顶点w有一条直接通路。

弧(Arc) ：表示两个顶点v和w之间存在一个关系，用顶点偶对<v,w>表示。通常根据图的顶点偶对将图分为有向图和无向图。

有向图(Digraph)： 若图G的关系集合E(G)中，顶点偶对<v,w>的v和w之间是有序的，称图G是有向图。

在有向图中，若 <v,w>∈E(G) ，表示从顶点v到顶点w有一条弧。 其中：v称为弧尾(tail)或始点(initial node)，w称为弧头(head)或终点(terminal node) 。

无向图(Undigraph)： 若图G的关系集合E(G)中，顶点偶对<v,w>的v和w之间是无序的，称图G是无向图。

在无向图中，若"<v,w>∈E(G) ，有<w,v>∈E(G) ，即E(G)是对称，则用无序对(v,w) 表示v和w之间的一条边(Edge)，因此(v,w) 和(w,v)代表的是同一条边。

依据最小完备性和常用性相结合的原则，以函数形式定义了包括创建图、销毁图、查找顶点、获得顶点值、顶点赋值、获得第一邻接点、获得下一邻接点、插入顶点、删除顶点、插入弧、删除弧、深度优先遍历、广度优先遍历、多图管理、文件存储 15 个基本运算，

要求分别定义函数来实现上述功能，具体功能运算如下：

⑴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。

⑵销毁图：树函数名称是DestroyGraph(T)；初始条件图G已存在；操作结果是销毁图G。

⑶查找顶点：函数名称是LocateVex(G,u)；初始条件是图G存在，u和G中的顶点具有相同特征；操作结果是若u在图G中存在，返回顶点u的位置信息，否则返回其它信息。

⑷获得顶点值：函数名称是GetVex (G,v)；初始条件是图G存在，v是G中的某个顶点；操作结果是返回v的值。

⑸顶点赋值：函数名称是PutVex (G,v,value)；初始条件是图G存在，v是G中的某个顶点；操作结果是对v赋值value。

⑹获得第一邻接点：函数名称是FirstAdjVex(&G, v)；初始条件是图G存在，v是G的一个顶点；操作结果是返回v的第一个邻接顶点，如果v没有邻接顶点，返回“空”。

⑺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是G的一个顶点,w是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，如果w是最后一个邻接顶点，返回“空”。

⑻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。

⑼删除顶点：函数名称是DeleteVex(&G,v)；初始条件是图G存在，v是G的一个顶点；操作结果是在图G中删除顶点v和与v相关的弧。

⑽插入弧：函数名称是InsertArc(&G,v,w)；初始条件是图G存在，v、w是G的顶点；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>。

⑾删除弧：函数名称是DeleteArc(&G,v,w)；初始条件是图G存在，v、w是G的顶点；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>。

⑿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。

⒀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。

4.1.3 系统演示及数据文件

要求构造一个具有菜单的功能演示系统。其中，在主函数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，并给出适当的操作提示。附录A中给了简单菜单的框架。演示系统已经实现了线性表的文件形式保存、多个线性表的理这两个附加功能。

## 4.2 系统设计

4.2.1 系统的物理结构

typedef int Status; //函数返回值的状态码

typedef struct ElemType{

char c[2];//一个字符

int n;//一个数字

}ElemType;//基础数据类型

typedef struct InfoType{

int weigtht;//存放弧的权值

}InfoType;//弧信息

typedef struct ArcNode{

int adjvex; //该弧指向的顶点位置

struct ArcNode \* nextarc; //指向下一条弧

InfoType info; //弧的信息

}ArcNode;//边类型

typedef struct Vnode{

ElemType data; //顶点信息

ArcNode \* firstarc;//指向第一条弧头节点

ArcNode \* lastArc; //最后一条弧头节点，用于方便插入

ArcNode \* cur; //指向当前弧头节点，用于寻找第一个、下一个邻接点

int tag; //顶点的标志域，用于判断顶点是否被遍历到

}Vnode;//顶点类型

typedef struct Graph{

Vnode vertices[MAX\_VERTEX\_NUM];//顶点集

int vexnum,arcnum; //边和顶点的数量

int kind; //图的种类标志

}Graph;//图类型

4.2.2 演示系统

将菜单演示和用户选择输入写入到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while循环后，系统首先显示功能菜单，然后提示用户输入选择（0-16），其中1-16对应二叉树的一个基本操作，分别对应一个函数，通过switch语句将用户输入的序号对应到相应的函数功能，执行完该语句后break跳出switch语句，执行while循环，直到用户输入0选择退出，退出系统。系统的框架如图4.1：



图4.1 系统框架

4.2.3 运算算法设计与思想

基于邻接表的图算法思想与设计如下：

1. 创建图

算法思想：从图的构成要素出发，可以知道图是由顶点集合边集构成，边集是建立在顶点集之上的，所以应当先生成顶点集，然后再生成边集。故创建两个函数：生成顶点集、生成边集。

用户先选择是有向图还是无向图，然后输入顶点数量，用数组存放顶点，用户输入顶点key值、顶点n值和顶点c值，系统读取后，将顶点集的tag由UNEXIST设置为NEWLY，赋值给第key个顶点。当用户的key值为-1时，输入停止。

然后输入边集，用户输入弧尾节点，然后输入弧头节点，和权值。生成对应的领接表后系统将领接表的地址赋值给相应的节点。当输入检测到含有-1时，输入停止。系统流程大致如图4.2：



图4.2 创建图

2. 销毁图

算法思想：销毁图即把申请的所有空间都释放掉，然后将对应的地址G设置为空，并给出用户提示。

3. 查找顶点

算法思想：查找顶点是根据C值获得顶点的key值， 用数组存放顶点，所以可以选择直接从数组的第0个元素开始，顺序查找到第MAX\_VEXNUM个元素，当顶点的C值等于函数传入的C值后，函数返回元素在数组的下标，即key值。

4. 获得顶点

算法思想：根据key值获得顶点的Vnode对应的地址，那么首先判断传入的参数是否存在，如果参数合法则直接返回G->vertics[key]。

5. 顶点赋值

算法思想：首先提示用户输入需要修改的顶点的key值和修改后的n值和c值，然后判断用户传入的参数是否合法，如果合法则生成对应的ElemType类型的数据元素，然后将G->vertics[key].data赋值为新生成的ElemType类型的数据。

6. 获得第一邻接点

算法思想：首先相应用户输入的key值，然后判断是否合法，如果合法则返回第一个节点的firstarc所指向的邻接表的adjvex值。获得第一邻接点流程图如图4.3：



图4.3 获得第一个邻接点

7. 获得下一个邻接点

算法思想：首先获得用户的输入key值，如果合法则返回cur指针的adjvex值，并让cur指向下一个节点。

8. 插入顶点

算法思想：首先接受用户输入的key值，如果合法，则产生新的节点，并让用户输入data域。

9. 删除顶点

算法思想：首先接受用户输入的key值，如果合法，则将对应的key值的顶点的tag域设置为UNEXIST即可。

10. 插入弧

算法思想：首先判断图是否存在，如果存在则判断图是否存在没如果存在则生成一个邻接表结点，结点的adjvex域存放弧头节点的key值，将该邻接表结点插入到弧尾节点的邻接表域，如果邻接表结点已经存在该条弧，那么直接返回OK并释放空间，否则进行插入。当插入完毕时，判断该图是否是无向图，如果是无向图那么用同样的方法生成以相反的邻接表进行插入。插入弧算法流程图如图4.4：



图4.4 插入弧

11. 删除弧

算法思想：根据用户输入的弧尾节点找到该节点的领接表结点，在邻接表中找到弧头节点的邻接表结点，运用线性表的删除操作删除该邻接表结点，然后判断该图是否为无向图，如果是无向图则进行相反的操作删除另外一条对应的弧。

12. 深度优先遍历

算法思想：深度优先遍历有两种方法，递归方法，一种是非递归方法。这里介绍非递归思想。将遍历起点入栈，然后进入循环，循环条件是栈非空，在循环内，首先出栈一个节点，如果该节点没有被遍历，则遍历该节点，然后将其tag域设置为OLD，并将其所有邻接点中未被遍历的节点入栈。直至循环结束。算法的程序流程图如图4.5所示：



图4.5 深度优先遍历（非递归）

13. 广度优先遍历

算法思想：广度优先遍历与深度优先遍历的非递归法非常像，只需要将栈换成队列即可。将遍历起点入队列，然后进入循环，循环条件是队列非空，在循环内，首先出队列一个节点，如果该节点没有被遍历，则遍历该节点，然后将其tag域设置为OLD，并将其所有邻接点中未被遍历的节点入队列。直至循环结束。

14. 多图管理

算法思想：首先定义一个图指针数组，每当切换图时，只需要将当前的图的地址改成图指针数组中对应的地址即可实现。

15. 文件存储

算法思想：文件存储和图的遍历比较像，本质上都是将图数据结构线性化过程。从图的定义出发，可以知道只需要保存点和边集即可实现。用两个文件分别保存顶点集和边集，然后再末尾加上-1，表示文件读取结束。

16. 文件读取

算法思想：根据算法15的保存形式，只需要将点集和边集读取到内存中即可，该算法和创建图操作的算法非常相似，只是输入流由键盘变成了文件流而已。

## 4.3 系统实现

4.3.1 编程环境、运行环境、项目工程描述

本次实验采用JetBrains CLion 2017.2编程软件编写，并用GCC进行编译运行，项目名称是BiTree。

4.3.2 头文件及预定义常量说明

#define MAX\_VERTEX\_NUM 20

#define DIRECTION\_GRAPH 1

#define UNDIRECTION\_GRAPH 0

#define UNEXIST -1

#define NEWLY 0

#define OLD 1

typedef int Status; //函数的状态码

typedef struct ElemType{

char c[2];//一个字符

int n;//一个数字

}ElemType;//基础数据类型

typedef struct InfoType{

int weigtht;//存放弧的权值

}InfoType;//弧信息

typedef struct ArcNode{

int adjvex; //该弧指向的顶点位置

struct ArcNode \* nextarc; //指向下一条弧

InfoType info; //弧的信息

}ArcNode;//边类型

typedef struct Vnode{

ElemType data; //顶点信息

ArcNode \* firstarc;//指向第一条弧头节点

ArcNode \* lastArc; //最后一条弧头节点，用于方便插入

ArcNode \* cur; //指向当前弧头节点，用于寻找第一个、下一个邻接点

int tag; //顶点的标志域，用于判断顶点是否被遍历到

}Vnode;//顶点类型

typedef struct Graph{

Vnode vertices[MAX\_VERTEX\_NUM];//顶点集

int vexnum,arcnum; //边和顶点的数量

int kind; //图的种类标志

}Graph;//图类型

4.3.3 演示系统操作

首先打开主界面，如图4.6所示：

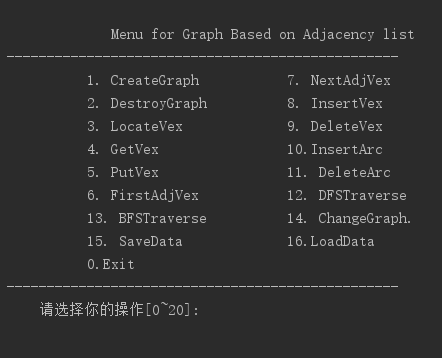


图4.6 主界面

功能1：创建图。创建图要求用户输入点集和边集，根据之前设置的数据类型，用户输入点集时需输入三个数据项，分别是key，n，c，其中key代表可元素的关键值，n和c表示节点的数据值。边集有三部分构成，分别是弧尾节点，弧头节点，权值。

在输入点集和边集之前，需要输入图的基本信息，选择是无向图还是有向图，输入顶点的数量。

创建图操作如图4.7所示：

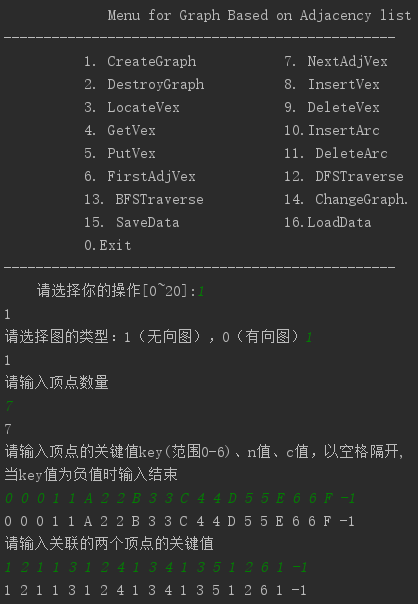


图4.7 创建图

创建图的逻辑结构如图4.8所示：



图4.8 图的逻辑结构

功能2：深度优先遍历：

在创建完成之后，首先对图进行遍历，测试图是否输入成功。图的遍历方式有两种，一种是深度优先遍历，一种是广度优先遍历。深度优先遍历是直接从一个顶点出发遍历到底，直至所有顶点都被遍历到。广度优先遍历是按照层次顺序一层一层进行遍历。

按照图4.8的逻辑结构对图进行深度优先遍历结果如图4.9所示：



图4.9 图的深度优先遍历

从结果上看，可以知道遍历方式正确。其中，最后一个元素是0号元素，从逻辑结构图中可以知道0号元素是不连通的，所以其在最后被遍历。

功能3：广度优先遍历

广度优先遍历的结果如图4.10所示：



图4.10 广度优先遍历

从结果上看，广度优先遍历无误。

功能4：获取元素的key值。

不妨获取c值为E的元素的key值，从预先的逻辑结构图中可以知道该节点的key值为5，测试结果如图4.11所示：



图4.11 获取节点的key值

功能5：根据key值获取元素数据

设获取的元素节点的key值为4，操作结果如图4.12所示：



图4.12 获取节点值

从图4.12结果看，key值为4的节点的n值为4，c值为D，符合逻辑结构图中预先设定好的值，结果正确。

功能6：修改节点数据

为了不影响后面的测试内容，我们选择对0号元素进行修改，将0号元素的c值由‘0’修改为‘Z’，操作结果如图4.13所示：



图4.13 修改节点的数据

为验证修改成功，我们选择再进行一次广度优先搜索，搜索结果如图4.14：



图4.14 验证修改节点操作

在红色部分圈出来的地方，可以知道节点的c值被修改为Z，程序正确。

功能7、功能8：通过key获得第一邻接点和下一邻接点

设获得获得key值为3的邻接点的值，从上面的逻辑结构图中可以知道，key值为3的节点的邻接点的值分别为1、4、5，但是逻辑结构图中邻接点的顺序不能确定，而计算机内部的邻接表内的顺序随输入顺序的不同而不同，所以以下结果我们只用关注值，而不用关注节点key的顺序。

获得第一邻接点的操作结果如图4.15所示：



图4.15 第一邻接点的值

第一邻接点的值为1，结果正确，接下来是连着三次调用下一邻接点操作，操作结果如图4.16所示：



图4.16（a）



图4.16（b）

上图结果分别为4、5，结果符合预期。为了增强程序的健壮性，我们不妨再调用一次下一邻接点，看一下结果会返回什么，操作结果如图4.16（c）所示：



图4.16（c）

结果返回“节点不存在”，这说明已经到了邻接表的表尾。

功能9：增加节点

增加节点，那么需要用户输入节点的key值，n值和c值，程序为了让用户操作更简单，有提示，当key值直接接入到原有节点的下一个位置时，可以直接输入-1。演示结果如图4.17所示：

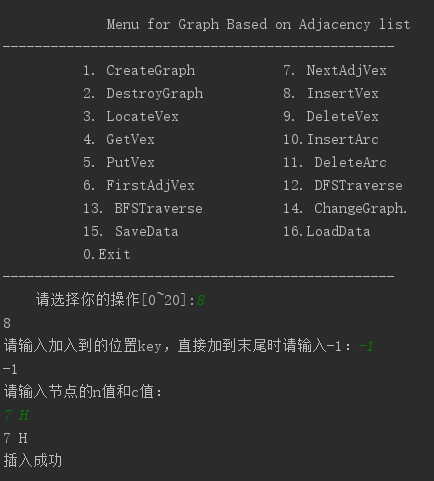


图4.17 插入节点

插入节点后，为了验证程序的正确性，需要对节点再进行一次广度优先遍历，遍历的结果如图4.18所示：



图4.18 验证插入节点操作

可以看到，在0号元素下面多了7号元素。此时该无向图含有三个极大联通子图。

功能10：插入弧

在原来的基础上，插入两条弧，将C与H、F与H连接起来，此时逻辑结构图已经变为图4.19所示：



图4.19 新的逻辑结构图

调用程序的10号操作，结果如图4.20所示：



图4.20 增加边

需要指出的是，尽管增加边的时候只是增加了有向弧，但是因为该图是无向图，自动增加相应的另外一条边，这一点从前面的算法中可以体现。

为了验证程序的正确性，对该图进行一次广度优先遍历。因为C节点的度为4，那么这次不妨从C节点开始遍历，C节点的key值为3，遍历的结果如图4.21所示：

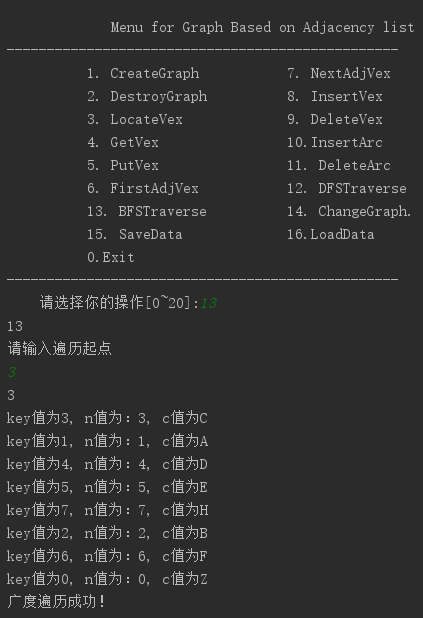


图4.21 验证增加弧的正确性

上图中7号元素的位置已经改变，根据新的逻辑结构图可以知道此时遍历的结果是正确的。

功能11：删除边

沿用图4.19的逻辑结构图，设删除的边为C与H所在的边，操作结果如图4.22所示：

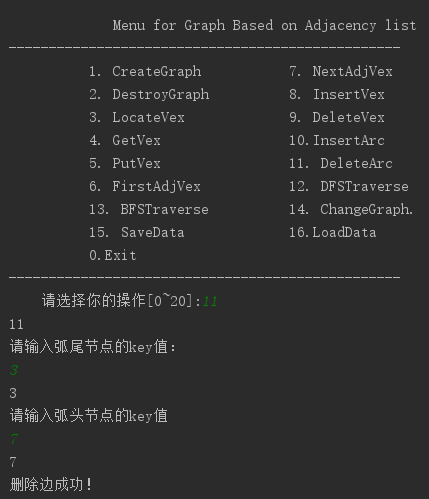


图4.22 删除边

需要说明的是本程序自动检测图是否是无向图，如果是无向图当删除一条弧时，对应的另外一条弧也被删除。

为了检测删除边的正确性，这次依然进行广度优先遍历，遍历的起点为C。广度优先遍历结果如图4.23所示：

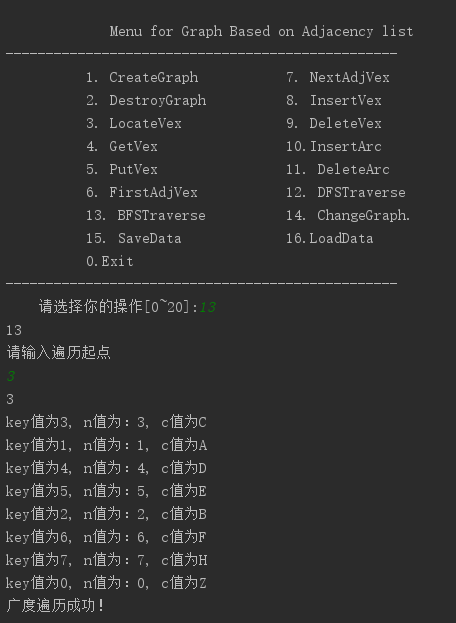


图4.23 验证删除弧

从上图观察可以发现，7号元素的位置已经改变，7号元素不再是3号元素的邻接点，说明删除边是正确的。此时的逻辑结构图为图4.24所示：



图4.24 删除边后的逻辑结构图

功能12：删除顶点

设删除的顶点为7号元素H，那么删除后的逻辑结构图又成为了第一次与设定的逻辑结构图（图4.8）。操作结果如图4.25所示：



图4.25 删除节点

为了验证删除节点是否正确，需要对该图进行一次广度优先遍历，遍历起点设置为1号节点，遍历结果如图4.26所示：



图4.26 测试删除节点

功能13：文件保存

调用结果如图4.27所示：

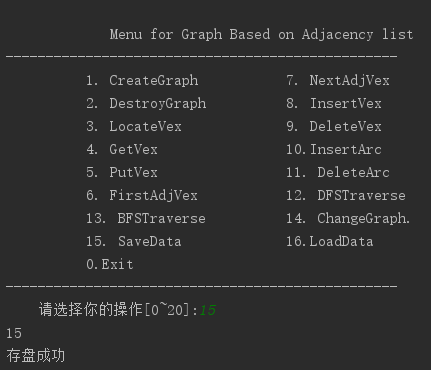


图4.27 文件保存

需要说明的是，本文件保存默认只能保存当前的图。

功能14：多图管理

调用多图管理后，输入相应的图编号即可进入相应的图。操作结果如图4.28：



图4.28 多图管理

此时第三张图是空图，没有任何信息。

功能15：文件读取

利用上面的操作，将保存在文件中的第一张图从文件中提取到第三张图中。在这操作之前，需要先验证第三张图是空图，验证结果如图4.29所示：

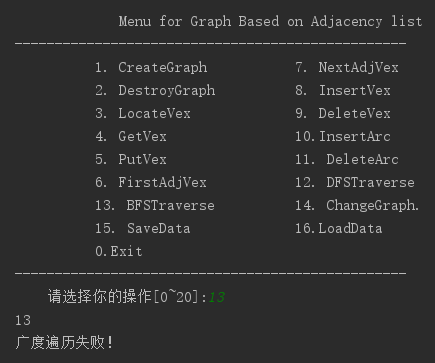


图4.29 验证空图

上面的结果显示，第三张图是空图。

接下来将刚刚存在文件中的数据提取出来，提取结果如图4.30所示：

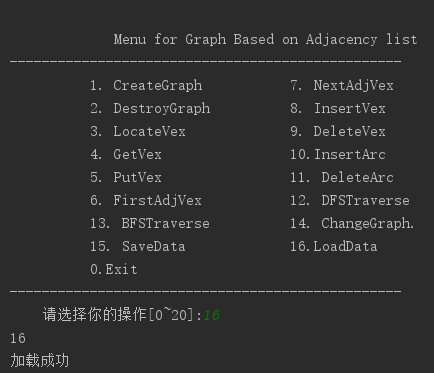


图4.30 文件读取

文件读取成功，那么对其进行广度优先遍历，遍历结果如图4.31所示：



图4.31 广度优先遍历

从遍历的结果上看，与之前的结果符合。

功能16：销毁图

演示的最后就是将图进行销毁，销毁结果如图4.32所示：



图4.32 销毁图

至此，整个功能的演示完毕。

## 4.4 实验小结

图的实验相对来说有点复杂，因为图的结构本身来说就比较复杂。其中困扰我最长时间的是深度优先遍历。开始时，我想尽量使用非递归的方式写，可是总是出各种Bug，有的是条件判断不足，有的是逻辑上的错误。后来实在没有办法就只好改成递归的形式，程序的代码量顿时小了起来，测试也通过了。可是还是不太甘心，就又写了一遍非递归的方法，此时思路比较清晰，测试也通过了。有些时候，递归不是万能的，最好的方式是两种方法都掌握。

还有一个问题就是关于删除节点后，节点的位置是否移动，经过和老师的讨论，老师建议我将删除的节点后面的节点向前移动一个位置。但是呢，因为问老师这个问题的时候我已经完成了实验，所以就没这么做。我的解决办法是设置一个tag域，因为遍历的时候也要设置一个tag域，不妨借用这个tag就好，所以我设置节点的tag域域三种状态，分别是：UNEXIST，NEWLY，OLD，分别表示：节点不存在，节点未被遍历，节点已被遍历。

随着实验的结束，我对数据结构的理解更加深刻了。

# 参考文献

[1] 严蔚敏等. 数据结构(C语言版). 清华大学出版社

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

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

[4] 严蔚敏等.数据结构题集(C语言版). 清华大学出版社

指导教师评定意见

一、对实验报告的评语

|  |
| --- |
|  |

二、对实验报告评分

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

# 附录A 基于顺序存储结构线性表实现的源程序

//

// Created by tonggege on 17-10-14.

//

#include "SequenceList.h"

#include <stdlib.h>

#include <stdio.h>

/\*\*一些说明：

\* ①线性表序号从1开始，底层数组从0开始

\*/

/\*\*func: 初始化链表

\* @操作结果 构造一个空的线性表L

\* @param L 顺序表的地址

\* @return 状态码

\*/

Status InitList(SqList \*\* L){

if(\*L!=NULL) {

printf("线性表已存在！");

return ERROR;

}

//构造一个线性表

if(!((\*L) = (SqList \*)malloc(sizeof(SqList)))) return ERROR;

(\*L)->elem = (ElemType \*)malloc(LIST\_INIT\_SIZE\*sizeof(ElemType)); //分配空间

if((\*L)->elem == NULL) exit(OVERFLOW); //如果分配失败则退出函数并返回错误码

(\*L)->length = 0; //初始长度为0

(\*L)->listsize = LIST\_INIT\_SIZE; //初始存储容量,以sizeof(ElemType)为单位

return OK;

}

/\*\*func: 销毁线性表

\* @初始条件 :线性表L已存在

\* @操作结果 :销毁线性表L

\* @param L 顺序表的地址

\* @return 状态码

\*/

Status DestroyList(SqList \*\* L){

if((\*L)==NULL) return ERROR;

if((\*L)->elem ==NULL || (\*L)->listsize == 0) exit(ERROR); //判断线性表L是否已经存在

free((\*L)->elem); //释放空间

(\*L)->listsize = 0; //设置存储容量为0

(\*L)->length = 0; //设置线性表长度为0

free(\*L);

return OK; //返回状态码

}

/\*\*func 清空线性表

\* @初始条件 线性表L已存在

\* @param L 线性表

\* @return 状态码

\*/

Status ClearList(SqList \* L){

if(L==NULL) return ERROR;

if(L->elem ==NULL || L->listsize == 0) exit(ERROR); //判断线性表L是否已经存在

free(L->elem); //释放空间

L->elem = (ElemType \* )malloc(LIST\_INIT\_SIZE \* sizeof(ElemType)); //重新分配初始量

L->listsize = LIST\_INIT\_SIZE;

L->length = 0;

return OK;

}

/\*\*func 判断线性表是否为空

\* @param L

\* @return

\*/

int ListEmpty(SqList \* L){

if(L->length == 0 || L->listsize == 0) return TRUE;

else return FALSE;

}

/\*\*func 获取线性表长度

\* @param L

\* @return 状态码

\*/

int ListLength(const SqList \* L){

if(L->listsize!=0) return L->length;

else return 0;

}

/\*\*func 获得第i个元素并存储到e中

\* @初始条件 线性表存在 索引i<= 表长length

\* @param L 线性表L

\* @param i 索引i

\* @param e 获得元素的地址e

\* @return 状态码

\*/

Status GetElem(const SqList \* L, int i, ElemType \* e){

if(!L) return ERROR;

if(L->listsize <= 0) exit(ERROR); //判断线性表是否存在

if(i> ListLength(L)) exit(ERROR); //判断是否为空表

ElemType \* List = L->elem; //获取基地址

\*e = List[i-1]; //线性表获取元素的时间的复杂度是常数O(1)

return OK;

}

/\*\*func 返回L中第1个与e满足关系compare的数据源素的位序.若这样的数据元素不存在,则返回值为0

\* @初始条件 线性表L已存在,compare已定义

\* @param L 线性表L

\* @param e 待比较的元素

\* @param compare 比较函数

\* @return 位序

\*/

int LocateElem(SqList \* L, ElemType e, void \* compare){

if(!L) return ERROR;

if(L->listsize == 0 || L->length == 0) return 0;

int ( \*cmp)(ElemType a,ElemType b);

cmp = (int (\*)(ElemType ,ElemType))compare;

ElemType \*List = L->elem; //获取基地址

int flag = FALSE; //判断当前是否找到这样的元素

int index = -1; //存放查找到的索引

for(int i = 0;i<ListLength(L) && flag != TRUE;i++){

if(cmp(List[i],e) == 0){

flag = TRUE; //找到标志

index = i+1; //底层数组下标和实际位序差1

}

if(flag != 0) break; //如果找到这样的元素就不继续查找了,跳出循环

}

if(flag) return index;

else return 0;

}

/\*\*func 获得cur\_e的前驱

\* @初始条件 L存在,cur\_e不是首元素,cur\_e在L中

\* @param L 线性表

\* @param cur\_e 元素

\* @param pre\_e 前驱指针

\* @return 状态码

\*/

Status PriorElem(const SqList \* L, ElemType cur\_e, ElemType \* pre\_e){

if(L->listsize == 0 || L->length <= 1) return (ERROR); //如果表不存在或者表长小于1,那么返回错误码

const ElemType \* List = L->elem;

if(compare(List[0],cur\_e)==0) return (ERROR); //如果该元素是第一个元素,则返回错误码

int flag = FALSE; //判断当前是否查找到

for(int i = 0;i<ListLength(L) && flag !=TRUE;i++){

if(compare(List[i],cur\_e)){

flag = TRUE; //找到标志

if(pre\_e!=NULL) \*pre\_e = List[i-1];

}

}

return OK;

}

/\*\*func 获得cur\_e的后继

\* @初始条件 L存在,cur\_e不是最后一个元素,cur\_e在L中

\* @param L 线性表

\* @param cur\_e 元素

\* @param next\_e 前驱指针

\* @return 状态码

\*/

Status NextElem(const SqList \* L, const ElemType cur\_e, ElemType \* next\_e){

if(L->listsize == 0 || L->length <=1 ) return(ERROR); //如果表不存在或者表长小于1,那么返回错误码

const ElemType \* List = L->elem;

if(compare(List[L->length-1] , cur\_e)==0) return(ERROR); //如果该元素最后一个元素,则返回错误码

int flag = FALSE; //判断当前是否查找到

for(int i = 0;i< ListLength(L) && flag != TRUE; i++){ //退出循环条件是表尾或找到元素

if(compare(List[i],cur\_e)==0){ //如果找到

flag = TRUE;

if(next\_e!=NULL) \*next\_e = List[i+1];

}

}

return OK;

}

/\*\*func:在第i个位置插入e

\* @param L 顺序表

\* @param i 位置

\* @param e 元素

\* @return 状态码

\*/

Status ListInsert(SqList \* L, int i, ElemType e){

if(!L) return ERROR;

if(L->listsize == 0) return(ERROR); //判断线性表是否存在

if(i<=0 || i > L->length + 1) return(ERROR); //判断位序是否是合法

if(L->listsize==L->length){

L->elem = (ElemType \*)realloc(L->elem,L->listsize+LISTINCREMENT);

}

ElemType \* List = L->elem; //获取 空间基地址

int j = ListLength(L); //从后往前移动

for(; j >= i; j--){

List[j] = List[j-1];

}

List[j] = e; //进行插入

L->length++;//长度加一

return OK;

}

/\*\*func:删除第i个元素

\*

\* @param L 线性表

\* @param i 序号

\* @param e 保存被删除元素的地址

\* @return 状态码

\*/

Status ListDelete(SqList \* L,int i,ElemType \* e){

if(L->listsize<=0||L->elem==NULL) return ERROR;

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

if(e!=NULL) \*e = L->elem[i-1];//定位删除位置点

for(int j = i-1;j<ListLength(L)-1;j++){ //从删除位置开始依次向右移动

L->elem[j] = L->elem[j+1];

}

L->length--;//更新表长

return OK;

}

/\*\*

\* func 使用visit函数便利每一个元素

\* @param L 线性表

\* @param visit 遍历函数

\* @return 状态码

\*/

Status ListTraverse(SqList \* L,void \* visit){

if(L==NULL) return ERROR;

if(L->listsize<=0||L->length<=0) return ERROR;

void (\*vis)(ElemType) = (void (\*)(ElemType))(visit);//遍历函数

for(int i = 0;i<L->length;i++) vis(L->elem[i]);

return OK;

}

/\*\*

\* func:加载文件

\* @param L 线性表地址

\* @return 状态码

\*/

Status LoadData(SqList \* L){

if(L==NULL) return ERROR;

ElemType e;

FILE \* fp;

if(!(fp = fopen("file","rb"))) return ERROR; //打开文件

int i = 1;

while(fread(&e,1,sizeof(ElemType),fp)!=0){//读取文件并插入

ListInsert(L,i,e);

i++;

}

fclose(fp);//关闭文件

return OK;

}

/\*\*

\* func 保存信息

\* @param L 线性表地址

\* @return 状态码

\*/

Status SaveData(SqList \* L){

if(L==NULL)return ERROR;

ElemType e;//获取元素暂存的单元

FILE \* fp;

fp = fopen("file","wb");//打开文件

for(int i = 1;i<=L->length;i++){

GetElem(L,i,&e);//获取元素

fwrite(&e,1,sizeof(ElemType),fp);//写入文件

}

fclose(fp);//关闭文件

return OK;

}

/\*\*

\* func:切换线性表

\* @param Prior ：本线性表地址

\* @param lists 列表

\* @param cur 当前列表序号

\* @return 状态码

\*/

Status ChangeSqList(SqList \*\* Prior,SqList \*\* lists,int \*cur){

int n = 1;

printf("您切换到第几个顺序表？");

scanf("%d",&n);//获取线性表序号，1—10

if(n<=0 || n>=11) return ERROR;

lists[(\*cur)-1] = \*Prior;//更新上一个

\*cur = n;

(\*Prior) = lists[n-1];

printf("切换成功!");

return OK;

}

# 附录B 基于链式存储结构线性表实现的源程序

//

// Created by tonggege on 17-11-6.

//

#include <malloc.h>

#include "LinkedList.h"

#include <stdio.h>

void visit(ElemType e){

printf("值为：%d",e);

}

int compare(ElemType a,ElemType b){

return a==b?1:0;

}

int Input(ElemType \*e){

printf("请输入值:\n");scanf("%d",e);

}

int Output(ElemType e){

printf("值为%d",e);

}

/\*

\* func:初始化链表

\* @param L 链式线性表的指针

\* @return 状态码

\*/

Status InitList(LinkedList \*\* list){

if((\*list)!=NULL) return ERROR;

\*list = (LinkedList \*)malloc(sizeof(LinkedList));

LinkedList \* L = \*list;

L->length = 0;

//初始化失败返回错误状态码

if((L->tail=L->head = (node \* )malloc(sizeof(node))) == NULL) return ERROR;

L->head->next = NULL;//为新产生的表头节点赋初值

return OK;

}

/\*\*

\* func： 销毁线性表

\* @param L

\* @return 状态码

\*/

Status DestroyList(LinkedList \*\* list){

LinkedList \* L = \*list;

if(!L||L->head == NULL) return ERROR;//处理表指针为空情况

node \* head = L->head;

node \* tmp = NULL;//用于删除的中间变量

while(head!=NULL){

tmp = head;

head = head->next;

free(tmp);

}

free(L);

\*list = NULL;

return OK;

}

/\*\*

\* func：清空列表

\* @param L 表指针

\* @return 状态码

\*/

Status ClearList(LinkedList\*L){

if(!L) return ERROR;

node \* p = L->head->next;//p指向第一个节点

while(p){

node \* tmp = p->next;

free(p);//释放节点

p = tmp;

}

L->length = 0;//长度设置为0

L->tail = L->head;//修改表尾指针

return OK;

}

/\*\*

\* func:判断列表是否为空

\* @param L

\* @return 是，返回true，否则false

\*/

int ListEmpty(LinkedList \* L){

if(!L) return TRUE;

if(L->length==0|| L->head == L->tail) return TRUE;//空表或者表不存在视为空

return FALSE;

}

/\*\*

\* func:获取线性表的长度

\* @param L 线性表

\* @return 长度

\*/

int ListLength(LinkedList \* L){

if(L==NULL) return 0;

return L->length;

}

/\*\*

\* func:获得第i个元素

\* @param L 链表

\* @param i 序号

\* @param e 保存元素的指针

\* @return 状态码

\*/

Status GetElem(LinkedList \*L,int i,ElemType \* e){

if(L==NULL) return ERROR;

if(i<=0 && i>ListLength(L)) return ERROR;//序号不正确返 ````````······························· 回错误状态码

node \* p = L->head->next;//p循环对比

for(int j = 1;j<=ListLength(L);j++,p = p->next){

if(i==j){ //如果序号正确则返回

if(e!=NULL) \*e = p->data;

break;

}

}

return p==NULL?ERROR:OK;

}

/\*\*

\* func:获得与e元素相等的元素的序位

\* @param L 链表

\* @param e 欲对比的元素

\* @param compare 比较函数，=0时相等，<0时右边大，>0时左边大

\* @return 元素序位

\*/

int LocateElem(LinkedList \*L,ElemType e,void \* compare){

if(L==NULL) return ERROR;

//将比较函数转换成合适的格式

int (\*com)(ElemType,ElemType) = (int (\*)(ElemType,ElemType))compare;

node \* p = L->head->next;

int i;//循环变量

for(i = 1;i<=ListLength(L)||p;i++,p = p->next){

if(com(p->data,e)==0){

return i;

}

}

return 0;

}

/\*\*

\* func 获取前驱节点数据

\* @param L 链表

\* @param cur\_e 查询节点

\* @param pre\_e 获取指针

\* @return 状态码

\*/

Status PriorElem(LinkedList \*L,ElemType cur\_e,ElemType \* pre\_e){

if(L==NULL) return ERROR;

if(L->head->next==NULL) return ERROR;//如果线性表不存在，则报错

if(L->head->next->data==cur\_e) return ERROR;//第一个节点不存在前驱

//q来配对,p来获得前驱

node \* p = L->head;

node \* q = L->head->next;

while(q&&p){

if(q->data==cur\_e){

if(pre\_e!=NULL) \*pre\_e=p->data;

return OK;

}

p = p->next;q = q->next;

}

return ERROR;

}

/\*\*

\* func 获取后继节点

\* @param L 线性表

\* @param cur\_e 当前待比较节点

\* @param next\_e 后继节点指针

\* @return 状态码

\*/

Status NextElem(LinkedList \*L,ElemType cur\_e,ElemType \* next\_e){

if(L==NULL) return ERROR;

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

if(L->tail->data==cur\_e) return ERROR;//尾节点没有后继

//p来配对，q来获取节点

node \* p = L->head->next;

node \* q = p->next;

while(p&&q){

if(p->data==cur\_e){

if(next\_e!=NULL) \*next\_e=q->data;

return OK;

}

p = p->next;q = q->next;

}

return ERROR;//不存在cur\_e

}

/\*\*

\* func : 将e节点插入到位置i中，1<=i<=length+1

\* @param L 线性表指针

\* @param i 位序

\* @param e 元素

\* @return 状态码

\*/

Status ListInsert(LinkedList \*L,int i,ElemType e){

if(!L) return ERROR;//线性表不存在

if(i<=0 || i> ListLength(L)+1) return ERROR;//序号不合法

//三部曲——产生节点

node \* f = NULL;

if(!(f=(node\*)malloc(sizeof(node)))) return ERROR;

f->data = e;

f->next = NULL;

node \* p = L->head;

node \* q = L->head->next;

for(int j = 1;j<=ListLength(L);j++,p = p->next,q = q->next){

//插入到1-length位置

if(j==i){

f->next = q;

p->next = f;

L->length++;

return OK;

}

}

//如果插入到尾部，则需要修改尾指针

p->next = f;

L->tail = f;

L->length++;

return OK;

}

/\*\*

\* func ：删除第i个元素并用e返回

\* @param L 线性表L

\* @param i 位序i

\* @param e 返回元素的指针

\* @return 状态码

\*/

Status ListDelete(LinkedList \*L,int i,ElemType \*e){

if(L==NULL) return ERROR;

if(i<=0 || i>ListLength(L)) return ERROR;

node \* p = L->head;//q指向被删除的元素的前一个

node \* q = L->head->next;//p指向被删除的元素

for(int j = 1;i<=ListLength(L);j++,p = p->next,q = q->next){

if(j == i){

p->next = q->next;

if(e) \*e = q->data;//赋值返回

free(q);

if(j==ListLength(L)){//如果删除最后一个元素，则需要修改尾指针

L->tail = p;

}

L->length--;

return OK;

}

}

}

/\*\*

\* func:遍历列表

\* @param L 线性表

\* @param visit 访问函数

\* @return 状态码

\*/

Status ListTraverse(LinkedList \* list,void \* visit){

if(list==NULL) return ERROR;

LinkedList L = \*list;

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

void (\*vst)(ElemType );

vst = (void (\*)(ElemType )) visit;

node \* p = L.head->next;

while(p){

vst(p->data);

p = p->next;

}

return OK;

}

Status ChangeSqList(LinkedList \*\* Prior,LinkedList \*\* lists,int \*cur){

int n = 1;

printf("您切换到第几个顺序表？");

scanf("%d",&n);//获取线性表序号，1—10

if(n<=0 || n>=11) return ERROR;

lists[(\*cur)-1] = \*Prior;//更新上一个

\*cur = n;

(\*Prior) = lists[n-1];

printf("切换成功!");

return OK;

}

Status LoadData(LinkedList \* L){

if(L==NULL) return ERROR;

ElemType e;

FILE \* fp;

if(!(fp = fopen("file","rb"))) return ERROR; //打开文件

int i = 1;

while(fread(&e,1,sizeof(ElemType),fp)!=0){//读取文件并插入

ListInsert(L,i,e);

i++;

}

fclose(fp);//关闭文件

return OK;

}

/\*\*

\* func 保存信息

\* @param L 线性表地址

\* @return 状态码

\*/

Status SaveData(LinkedList \* L){

if(L==NULL)return ERROR;

ElemType e;//获取元素暂存的单元

FILE \* fp;

fp = fopen("file","wb");//打开文件

for(int i = 1;i<=L->length;i++){

GetElem(L,i,&e);//获取元素

fwrite(&e,1,sizeof(ElemType),fp);//写入文件

}

fclose(fp);//关闭文件

return OK;

}

# 附录C 基于二叉链表二叉树实现的源程序

//

// Created by tonggege on 17-11-10.

//

#include "BiTree.h"

#include <malloc.h>

#include<queue>

#include <iostream>

#include <cstdio>

#include <io.h>

using namespace std;

/\*\*

\* func：初始化树

\* @param T 树

\* @return 状态码

\*/

Status InitBiTree(BiTree \*\*T){

if(\*T) {

cout << "二叉树已经存在！"<<endl;

}

\*T = (BiTree \* )malloc(sizeof(BiTree)); //分配空间

(\*T)->root = NULL;

return OK;

}

/\*\*

\* func:销毁树

\* @param n 节点指针

\* @return 状态码

\*/

Status DestroyTreeNode(node \* n){

//递归销毁二叉树节点，先销毁左孩子

// 再销毁右孩子，最后销毁根节点

if(n){

DestroyTreeNode(n->lchild);

DestroyTreeNode(n->rchild);

free(n);

}

return OK;

}

Status DestroyBiTree(BiTree \*\*T){

if(!(\*T)) return ERROR;

DestroyTreeNode((\*T)->root);

free(\*T);

\*T = NULL;

return OK;

}

/\*\*

\* func 生成二叉树

\* @param T 二叉树

\* @return 二叉树根节点

\*/

node \* CreateBiTreeRec();

Status CreateBiTree(BiTree \*T){

if(T==NULL) return ERROR; //判断传入的参数是否合法

char c;int key;

cout<<"请输入节点key值和c值,当key值为0时表示空指针（按照前序遍历输入）"<<endl;

cin >>key; //输入key值

cin.get();

if(key == 0){ //如果key为0，那么结束输入

T->root = NULL;

return OK;

}

else{

cin >> c;

T->root = (node \* )malloc(sizeof(node));

T->root->data.key = key;

T->root->data.c = c;

T->root->lchild = NULL;

T->root->rchild = NULL;

}

T->root->lchild = CreateBiTreeRec(); //递归地创造左孩子

T->root->rchild = CreateBiTreeRec();//递归地创造右孩子

return OK;

}

node \* CreateBiTreeRec(){

char c;int key;

node \* N = NULL;

cin >> key;cin.get();

if(key==0) return NULL;

cin >> c;

if(!(N = (node \*)malloc(sizeof(node)))) return NULL;

N->data.key = key;

N->data.c = c;

N->lchild = CreateBiTreeRec();

N->rchild = CreateBiTreeRec();

return N;

}

/\*\*

\* func:销毁树

\* @param T 树指针

\* @return 状态码

\*/

Status ClearBiTree(BiTree \*T){

if(T==NULL)return ERROR;

DestroyTreeNode(T->root);

T->root = NULL;

return OK;

}

/\*\*

\* func:判断树是否为空

\* @param T

\* @return

\*/

int BiTreeEmpty(BiTree \*T){

if(T==NULL) return TRUE;

int flag = (T->root == NULL?TRUE:FALSE); //如果为二叉树不存在则直接返回TRUE

return flag;

}

/\*\*

\* func:求树的深度

\* @param n 节点

\* @return 深度

\*/

//思路：

/\*\*

\* 一棵树的深度可以定义为左子树和右子树的深度最大值+1,其中，空树深度为0

\*/

int nodeDepth(node \* n){

if(n){

int count1,count2;

count1 = 1+nodeDepth(n->lchild); //计算左孩子的深度+1

count2 = 1+nodeDepth(n->rchild);//计算右孩子的深度+1

return (count1>count2?count1:count2);//返回两者的最大值

}

return 0;

}

int BiTreeDepth(BiTree \*T){

if(T==NULL) return 0;//如果二叉树不存在则深度为0

return nodeDepth(T->root);//递归求数的深度

}

/\*\*

\* func:求树的根节点

\* @param T

\* @return

\*/

node \* Root(BiTree \*T){

if(T==NULL) return NULL;

return T->root;

}

/\*\*

\* func 获取值

\* @param BiTree 二叉树

\* @param key 关键值

\* @return

\*/

node \* nodeValue(node \* n,int key);

char Value(BiTree \*T,int key){

if(T==NULL) return ERROR; //判断条件

return nodeValue(T->root,key)->data.c; //递归获取待查找的节点

}

node \* nodeValue(node \* n,int key){

if(n){

if(n->data.key==key) return n; //如果找到则直接返回key值

else{

node \* cl = nodeValue(n->lchild,key); //否则递归查找左孩子

node \* cr = nodeValue(n->rchild,key);//递归查找右孩子

return cl==NULL ? cr:cl; //返回两个的查找结果

}

}

else{

return NULL;

}

}

/\*\*

\* func 赋值，如果e在T中，则将value赋值为T

\* @param root 根节点

\* @param e 待赋值节点

\* @return 状态码

\*/

node \* isInTree(node \* root,int key);

Status Assign(BiTree \*T,int key,char value){

if(T==NULL) return ERROR;

node \* N = isInTree(T->root,key);

if(!N) return ERROR; //如果e不存在树上，则返回错误码

else{

N->data.c = value;

return OK;

}

}

//返回节点e是否在root中

node \* isInTree(node \* root,int key){

if(!root) return NULL;//如果根节点为空，则不在这棵树上

if(root->data.key==key) return root;//如果根节点是，则返回TRUE

//返回左节点或右节点

node \* l = isInTree(root->lchild,key);//递归查找左孩子

node \* r = isInTree(root->rchild,key);//递归查找右孩子

return l==NULL?r:l; //返回查找结果

}

/\*\*

\* func:获得双亲节点

\* @param T ：树

\* @param e ：e的根节点

\* @return e的根节点

\*/

node \* ParentRec(node \* root,int key);

node \* Parent(BiTree \*T, int key){

if(T==NULL) return NULL;

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

node \* l = ParentRec(T->root->lchild,key); //获取左孩子的双亲节点

node \* r = ParentRec(T->root->rchild,key); //获取右孩子的双亲节点

return l?l:r;

}

//递归获取孩子的双亲节点

node \* ParentRec(node \* root,int key){

if(root){

//如果找到，则返回找到的节点

if(root->lchild&&root->lchild->data.key==key||root->rchild&&root->rchild->data.key==key){

return root;

}

else{//不相等则继续往下递归搜索

node \* l = ParentRec(root->lchild,key);

node \* r = ParentRec(root->rchild,key);

return l?l:r;

}

}

return NULL;

}

/\*\*

\* func:获得左孩子

\* @param T 树

\* @param key 节点关键值

\* @return 节点e的左孩子

\*/

node \* LeftChildRec(node \* root, int key);

node \* LeftChild(BiTree \*T,int key){

if(T==NULL) return NULL;

return LeftChildRec(T->root,key);

}

node \* LeftChildRec(node \* root,int key){

if(root){

if(root->data.key == key){ //找到的情况

return root->lchild;

}

else{

//没找到继续递归的向下查找

node \* l = LeftChildRec(root->lchild,key);

node \* r = LeftChildRec(root->rchild,key);

return l?l:r;

}

}

return NULL;

}

/\*\*

\* func:返回节点e的右孩子

\* @param T 树

\* @param e 节点数据域

\* @return 节点e的右孩子

\*/

node \* RightChildRec(node \* root,int key);

node \* RightChild(BiTree \*T, int key){

if(T==NULL) return NULL;

return RightChildRec(T->root,key);

}

node \* RightChildRec(node \* root,int key){

if(root){

if(root->data.key == key){

return root->rchild;//查找到的情况

}

else{//递归查找

node \* l = RightChildRec(root->lchild,key);

node \* r = RightChildRec(root->rchild,key);

return l?l:r;

}

}

return NULL;

}

/\*\*

\* func:返回e左兄弟

\* @param T：树

\* @param e：e节点

\* @return ：返回e的右孩子

\*/

node \* LeftSiblingRec(node \* root,int key);

node \* LeftSibling(BiTree \*T,int key){

if(T==NULL) return NULL;

return LeftSiblingRec(T->root, key);

}

node \* LeftSiblingRec(node \* root,int key){

if(root){

//如果本身就是左孩子则不存在左兄弟

if(root->lchild&&root->lchild->data.key==key) return NULL;

//如果是右孩子并且找到，则返回左孩子

if(root->rchild&&root->rchild->data.key==key) return root->lchild;

//如果没有查找到，则继续向下深入查找

node \* l = LeftSiblingRec(root->lchild,key);

node \* r = LeftSiblingRec(root->rchild,key);

return l?l:r;

}

return NULL;

}

/\*\*

\* func:返回右兄弟

\* @param T：树

\* @param e：e节点

\* @return ：返回e的左孩子

\*/

node \* RightSiblingRec(node \* root,int key);

node \* RightSibling(BiTree \*T,int key){

if(T==NULL) return NULL;

return RightSiblingRec(T->root,key);

}

node \* RightSiblingRec(node \* root,int key){

if(root){

if(root->lchild&&root->lchild->data.key==key) return root->rchild;

if(root->rchild&&root->rchild->data.key==key) return NULL;

node \* l = RightSiblingRec(root->lchild,key);

node \* r = RightSiblingRec(root->rchild,key);

return l?l:r;

}

return NULL;

}

/\*\*

\* func:根据LR为0或1，插入c为T中p所指节点的左或右子树。p所指节点的原有左或右子树则成为c的右子树

\* 初始条件：二叉树T存在，p指向T中某个节点，LR为0或1，非空二叉树c与T不相交且右子树为空

\* @param T 树

\* @param LR

\* @param c

\* @return

\*/

Status InsertChild(BiTree \*T,node \*p ,int LR,node \* c){

if(T==NULL) return ERROR;

//判断传入的信息是否正确

if(!p) return ERROR;

if(!c) return ERROR;

if(c->rchild!=NULL) return ERROR;

if(LR==1){//如果插入左孩子

c->rchild = p->lchild;

p->lchild = c;

}

else{//否则

c->rchild = p->rchild;

p->rchild = c;

}

return OK;

}

/\*\*

\* func:删除孩子节点

\* @param T

\* @param p

\* @param LR

\* @return

\*/

Status DeleteChild(BiTree \*T,node \* p,int LR){

if(T==NULL) return ERROR;

if(!p) return ERROR;

if(LR==1){//如果删除左孩子

DestroyTreeNode(p->lchild);

p->lchild = NULL;

}

else{//否则

DestroyTreeNode(p->rchild);

p->rchild = NULL;

}

return OK;

}

/\*\*

\* func:前序遍历

\* @param T ：树

\* @param Visit ：访问函数

\* @return 状态码

\*/

Status PreTraverseRec(node \* root,void \* Visit);

Status PreOrderTraverse(BiTree \*T,void \* Visit){

if(T==NULL) return ERROR;

return PreTraverseRec(T->root,Visit);

}

//递归遍历

Status PreTraverseRec(node \* root,void \* Visit){

Status (\*vst)(node\*) = (Status (\*)(node \*)) Visit;

if(root){

if(!vst(root)) return ERROR; //先遍历根节点，再遍历左右子树

PreTraverseRec(root->lchild,(void\*)vst);

PreTraverseRec(root->rchild,(void\*)vst);

}

}

/\*\*

\* func:中序遍历

\* @param T ：树

\* @param Visit ：访问函数

\* @return 状态码

\*/

Status InTraverseRec(node \* root,void \* Visit);

Status InOrderTraverse(BiTree \*T,void \* Visit){

if(T==NULL) return ERROR;

return InTraverseRec(T->root,Visit);

}

//递归遍历

Status InTraverseRec(node \* root,void \* Visit){

Status (\*vst)(node\*) = (Status (\*)(node \*)) Visit;

if(root){

InTraverseRec(root->lchild,(void\*)vst); //遍历左子树

if(!vst(root)) return ERROR; //访问根节点

InTraverseRec(root->rchild,(void\*)vst);//遍历右子树

}

}

/\*\*

\* func:后序遍历

\* @param T ：树

\* @param Visit ：访问函数

\* @return 状态码

\*/

Status PostTraverseRec(node \* root,void \* Visit);

Status PostOrderTraverse(BiTree \*T,void \* Visit){

if(T==NULL) return ERROR;

return PostTraverseRec(T->root,Visit);

}

//递归遍历

Status PostTraverseRec(node \* root,void \* Visit){

Status (\*vst)(node\*) = (Status (\*)(node \*)) Visit;

if(root){

PostTraverseRec(root->lchild,(void\*)vst);//遍历左子树

PostTraverseRec(root->rchild,(void\*)vst);//遍历右子树

if(!vst(root)) return ERROR;//遍历根节点

}

}

/\*\*

\* func：层序遍历

\* @param T

\* @param Visit

\* @return

\*/

Status LevelOrderTraverse(BiTree \*T,void \* Visit){

std::queue <node \*>Q; //利用了C++的队列

Status (\*vst)(node \* ) = (Status(\*)(node \*)) Visit; //声明访问函数

Q.push(T->root);node \* self = NULL; //将根节点如队列

while(!Q.empty()){

self = Q.front();Q.pop(); //出队列

//将两个孩子放入到队列中

if(self->lchild) Q.push(self->lchild);

if(self->rchild) Q.push(self->rchild);

//访问该节点

if(!vst(self)) return ERROR;

}

return OK;

}

//visit函数

void visitNode(node \* N){

printf("Key 为 ：%d , C值为%c \n",N->data.key,N->data.c);

}

Status ChangeTree(BiTree \* tree[], BiTree \*\* T, int num){

if(!tree[num]) (\*T) = tree[num];

else return ERROR;

return OK;

}

Status LoadData(BiTree \* T){

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

int key;char c;int flag = 0;

flag = fscanf(fp,"%d",&key);

if(key==0||flag!=1){

T->root = NULL;

fclose(fp);

return OK;

}

//加载数据和创造树非常类似，只是输入流换成了文件而已

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

T->root = (node \*)malloc(sizeof(node));

T->root->data.key = key;

T->root->data.c = c;

T->root->lchild = LoadDataRec(fp);

T->root->rchild = LoadDataRec(fp);

fclose(fp);

return OK;

}

node \* LoadDataRec(FILE \* fp){

int key;

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

if(key == 0) return NULL;

node \* root = (node \* )malloc(sizeof(node));

root->data.key = key;

fscanf(fp," %c",&root->data.c);

root->lchild = LoadDataRec(fp);

root->rchild = LoadDataRec(fp);

return root;

}

Status SaveData(BiTree \* T){

if(T->root == NULL) return ERROR;

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

fprintf(fp,"%d %c ",T->root->data.key,T->root->data.c); //写入到文件

SaveDataRec(T->root->lchild,fp);//递归向下

SaveDataRec(T->root->rchild,fp);//递归向下

fclose(fp);

return OK;

}

Status SaveDataRec(node \* root,FILE \* fp){

if(!root){

fprintf(fp,"%d ",0);

return OK;

}

else{

fprintf(fp,"%d %c ",root->data.key,root->data.c);

SaveDataRec(root->lchild,fp);

SaveDataRec(root->rchild,fp);

}

return OK;

}

# 附录D 基于邻接表图实现的源程序

//

// Created by Administrator on 2017/12/1.

//

#include "Graph.h"

#include <stdio.h>

#include <iostream>

#include <cstring>

#include <stack>

#include <queue>

using namespace std;

/\*\*

\* func:访问函数

\* @param G 图地址G

\* @param node 顶点类型

\* @return 状态码

\*/

int visit(Graph \* G,Vnode \* node){

if(!G) return ERROR;

if(!node||node->tag<0) return -1;

cout << "key值为"<<LocateVex(G,node->data.c,(void \* )(&strcmp))<<", n值为："<< node->data.n<<", c值为"<<node->data.c<<endl;

return OK;

}

/\*\*

\* func: 创造节点

\* @param G 图地址

\* @return 状态码

\*/

Status CreateVex(Graph \* G){

if(!G) return ERROR;

Vnode \* vex = G->vertices;

int count = 0;int type;

int index = -1;

int flag = 1;//判断输入是否合法

cout << "请输入顶点数量"<< endl;

cin >> count;

G->vexnum = (count > 0?count:0); //设置图顶点的数量

if(count <=0 ) return ERROR;

cout << "请输入顶点的关键值key(范围0-"<<count-1<<")、n值、c值，以空格隔开,\n当key值为负值时输入结束"<< endl;

int key;

while(1){

cin >> key;

if(key < 0) break;//当k为负值时，输入结束

//输入数据

cin >> G->vertices[key].data.n;

cin >> G->vertices[key].data.c;

//初始时将顶点的指针域都设置为NULL

G->vertices[key].firstarc = G->vertices[key].lastArc = G->vertices[key].cur = NULL;

G->vertices[key].tag = NEWLY; //设置顶点的标志域为NEWLY

}

return OK;

}

/\*\*

\* func: 产生边

\* @param tail 以tail为尾的边

\* @return 边地址

\*/

ArcNode \* GenerateArc(int tail,int weight){

//生成一个以弧尾节点为tail权重为weight的邻接点

ArcNode \* newArc = (ArcNode\*)malloc(sizeof(ArcNode));//分配空间

//赋值

newArc->nextarc = NULL;

newArc->adjvex = tail;

newArc->info.weigtht = weight;

return newArc;

}

/\*\*

\* func: 创造边

\* @param G 图地址

\* @return 状态码

\*/

Status CreateArc(Graph \* G){

if(!G) return ERROR;

int head;//弧尾

int tail;//弧头

ArcNode \* p = NULL;

ArcNode \* q;

//给出提示

(G->kind==UNDIRECTION\_GRAPH)?(cout << "请输入关联的两个顶点的关键值\n"):(cout<<"请输入起点和终点的关键值\n");

while(1){

cin >> head;

if(head<0) break;//当输入为负值的时候退出循环

cin>>tail;

//当不满足条件的时候返回错误码

if(head >=MAX\_VERTEX\_NUM||tail < 0 || tail >= MAX\_VERTEX\_NUM||

G->vertices[tail].tag<0||G->vertices[head].tag<0) return ERROR;

int weight;

cout << "请输入权值："<<endl;

cin >> weight;

p = GenerateArc(tail,weight);//产生一个邻接点

if(G->vertices[head].firstarc==NULL) { //插入点是首元素

G->vertices[head].firstarc = p;

G->vertices[head].lastArc = p;

}

else{//插入点不是首元素

G->vertices[head].lastArc->nextarc = p;

G->vertices[head].lastArc = p;

}

G->arcnum++;

//处理无向图情形，补充相反的边即可

if(G->kind==UNDIRECTION\_GRAPH){

q = GenerateArc(head,weight);

if(G->vertices[tail].firstarc==NULL){

G->vertices[tail].firstarc = q;

G->vertices[tail].lastArc = q;

}

else{

G->vertices[tail].lastArc->nextarc = q;

G->vertices[tail].lastArc = q;

}

G->arcnum++;

}

}

return OK;

}

/\*\*

\* func：产生图

\* @param Gra

\* @return 状态码

\*/

Status CreateGraph(Graph \*\*Gra) {

if(\*Gra) return ERROR;

Graph \* G = (Graph \*)malloc(sizeof(Graph));

(\*Gra) = G;

if(!G) return ERROR;

int type;

std::cout << "请选择图的类型：1（无向图），0（有向图）";

std::cin >> type;

//获取类型

G->kind = (type==1?UNDIRECTION\_GRAPH:DIRECTION\_GRAPH);

G->vexnum = 0;G->arcnum = 0;

//将所有顶点设置成不存在

for(int i = 0;i<MAX\_VERTEX\_NUM;i++) G->vertices[i].tag = UNEXIST;

int flag1,flag2;

flag1 = CreateVex(G);//产生顶点

flag2 = CreateArc(G);//产生边（注：顺序不能颠倒）

//将cur域归位成第一个，cur是为了first和nextAdj准备的

for(int i = 0;i<MAX\_VERTEX\_NUM;i++){

G->vertices[i].cur = G->vertices[i].firstarc;

}

return flag1 && flag2;

}

/\*\*

\* func:销毁图

\* @param Gra

\* @return

\*/

Status DestroyGraph(Graph \*\*Gra){

Graph \*G = \*Gra;

ArcNode \* p = NULL;

for(int i = 0;i<MAX\_VERTEX\_NUM;i++){

G->vertices[i].tag = -1;

p = G->vertices[i].firstarc;

ArcNode \* tmp;

//依次释放邻接表节点

while(p){

tmp = p;

p = p->nextarc;

free(tmp);

}

}

free(G);//释放分配的空间

(\*Gra) = NULL;//销毁完后要归零

return OK;

}

/\*\*

\* 获取顶点地址

\* @param G 图

\* @param Info 待比较信息

\* @param Gcompare 比较函数

\* @param node 返回的顶点地址

\* @return 位置或者-1

\*/

int LocateVex(Graph \* G,char \* cmp,void \*Gcompare){

if(!G) return ERROR;

//直接从顶点数组依次向下寻找，不用BFS和DFS

int (\*Gcmp)(char \*,char \*);//函数指针

Gcmp = (int(\*)(char \*, char \*))(Gcompare);

for(int i = 0;i<MAX\_VERTEX\_NUM;i++){

if(G->vertices[i].tag==UNEXIST) continue;//遇见不存在的位置直接跳过

if(Gcmp(G->vertices[i].data.c,cmp)==0){

return i;//查找到直接返回序位

}

}

return -1;//查找不成功返回-1

}

Vnode \* GetVex(Graph \*G,int key){

if(!G||key<0||key>=MAX\_VERTEX\_NUM||G->vertices[key].tag<0) return NULL;

return &G->vertices[key];//根据key直接获得顶点，时间复杂度为O(1)

}

/\*\*

\* func ：给顶点赋值

\* @param G 图G

\* @param key 待赋值顶点的编号

\* @param e 赋值的data域

\* @return 状态码

\*/

Status PutVex(Graph \* G,int key,ElemType e){

if(!G||key<0||key>=MAX\_VERTEX\_NUM||G->vertices[key].tag<0) return ERROR;

G->vertices[key].data = e;

return OK;

}

/\*\*

\* func:获取第一个邻结点

\* @param G

\* @param key

\* @return

\*/

int FirstAdjVex(Graph \*G,int key){

if(!G||key<0||key>MAX\_VERTEX\_NUM||G->vertices[key].tag<0||!G->vertices[key].firstarc) return -1;

G->vertices[key].cur = G->vertices[key].firstarc;//cur归零

if(!G->vertices[key].firstarc) return -1;//如果first域为空，那么返回-1

else{

G->vertices[key].cur = G->vertices[key].firstarc->nextarc;

return G->vertices[key].firstarc->adjvex;

}//否则返回第一个邻接点

}

/\*\*

\* func : 获得下一个领结点

\* @param G 图指针

\* @param key 关键值

\* @return 节点

\*/

int NextAdjVex(Graph \*G,int key){

if(G==NULL||key<0||key>=MAX\_VERTEX\_NUM||G->vertices[key].tag<0) return -1;

if(G->vertices[key].cur ==NULL) return -1;//当前域cur为空，则返回-1

int kkk = G->vertices[key].cur->adjvex;//保存待返回的值

G->vertices[key].cur = G->vertices[key].cur->nextarc;//当前指针移向下一个

return kkk;

}

/\*\*

\* func : 插入节点

\* @param G 图G

\* @param v 顶点指针（必须非空）

\* @param key ; 当key为-1时，直接放在后面

\* @return 状态码

\*/

Status InsertVex(Graph \* G,Vnode \*v,int key){

if(!G||!v) return ERROR;

v->firstarc = v->cur = v->lastArc = NULL;//新产生的顶点的邻接域全部置为零

v->tag = NEWLY;

if(key == -1){//-1代表直接插入到尾部

G->vertices[G->vexnum] = (\*v);//进行赋值

}

else{

if(G->vertices[key].tag>0) return ERROR;//如果插入点有顶点，则返回错误码

G->vertices[key] = (\*v);//进行赋值

}

G->vexnum++;

return OK;

}

/\*\*

\* func :根据key值删除边

\* @param G 图地址

\* @param head 弧尾节点

\* @param tail 弧头节点

\* @return 状态码

\*/

Status DeleteArcByKey(Graph \* G,int head,int tail){

if(!G) return ERROR;

ArcNode \* tmp,\*p;

//删除所有以tail为弧头的节点，用循环，以防止有多个边

while(G->vertices[head].firstarc&&G->vertices[head].firstarc->adjvex==tail){

tmp = G->vertices[head].firstarc;

G->vertices[head].firstarc = G->vertices[head].firstarc->nextarc;

free(tmp);

G->arcnum--;

}

//删除与tail连接的边，其中tail不是第一个邻接表节点

for(p = G->vertices[head].firstarc;p&&p->nextarc;p = p->nextarc){

if(p->nextarc->adjvex==tail){

tmp = p->nextarc;

p->nextarc = tmp->nextarc;

free(tmp);

G->arcnum--;

}

}

return OK;

}

/\*\*

\* func :删除顶点

\* @param G 图地址

\* @param v 顶点

\* @return 状态码

\*/

Status DeleteVex(Graph \*G,Vnode \*v){

if(!G||!v) return ERROR;

//删除顶点的方式就是将其tag域置为UNEXIST

int d\_key = LocateVex(G,v->data.c,(void \*)&strcmp);

v->tag = -1;

for(int i = 0;i<MAX\_VERTEX\_NUM;i++){

//循环查找

DeleteArcByKey(G,d\_key,i); //删除边

DeleteArcByKey(G,i,d\_key);

}

return OK;

}

/\*\*

\* func: 判断是否存在以key\_head为弧尾，以key\_tail为弧头的边

\* @param G 图地址

\* @param key\_head 弧尾节点

\* @param key\_tail 弧头节点

\* @return 状态码

\*/

int HasArc(Graph \* G, int key\_head, int key\_tail){

if(!G) return ERROR;

Vnode \*head = GetVex(G,key\_head);//获取弧尾节点

ArcNode \* p = NULL;

for(p = head->firstarc;p;p = p->nextarc){

if(p->adjvex==key\_tail) return 1;

}

return 0;

}

/\*\*

\* func: 插入边

\* @param G 图地址

\* @param key\_head 弧尾节点（出发点）

\* @param key\_tail 弧头节点（结束点）

\* @param weight 权重

\* @return 状态码

\*/

Status InsertArc(Graph \*G,int key\_head,int key\_tail,int weight){

if(!G) return ERROR;

Vnode \* head = GetVex(G,key\_head);//根据key值获取节点

Vnode \* tail = GetVex(G,key\_tail);

if((!head)||(head->tag==UNEXIST)||(!tail)||tail->tag==UNEXIST) return ERROR;

if(HasArc(G,key\_head,key\_tail)) return OK;//如果已经存在边，那么直接返回OK

ArcNode \* p = (ArcNode \* )malloc(sizeof(ArcNode));

if(!p) return ERROR;

p->nextarc = NULL;p->adjvex = key\_tail; //对新产生的边进行赋值

if(head->firstarc==NULL){ //如果first域为空，那么新产生的边就是first

head->lastArc = head->cur = head->firstarc = p;

}

else{//否则新产生的边插入到尾部

head->lastArc->nextarc = p;

head->lastArc = head->lastArc->nextarc;

}

p->info.weigtht = weight;//进行权重赋值

G->arcnum++;

//处理无向图

if(G->kind==UNDIRECTION\_GRAPH){

if(HasArc(G,key\_tail,key\_head)) return OK;

//当所有条件都符合的时候，就产生一条边

ArcNode \* q = (ArcNode \*)malloc(sizeof(ArcNode));

q->nextarc = NULL;

q->adjvex = key\_head;

q->info.weigtht = p->info.weigtht;

if(tail->firstarc==NULL){

tail->cur = tail->lastArc = tail->firstarc = q;

} else{

tail->lastArc->nextarc = q;

tail->lastArc = tail->lastArc->nextarc;

}

G->arcnum++;

}

return OK;

}

/\*\*

\* func : 删除边

\* @param G 图地址

\* @param key\_head 弧尾节点

\* @param key\_tail 弧头节点

\* @return 状态码

\*/

Status DeleteArc(Graph \*G,int key\_head,int key\_tail){

if(!G) return ERROR;

//获取节点

Vnode \* head = GetVex(G,key\_head);

Vnode \* tail = GetVex(G,key\_tail);

if(!head||!tail) return ERROR;

//删除边

int flag1 = DeleteArcByKey(G,key\_head,key\_tail);

int flag2;

if(G->kind==UNDIRECTION\_GRAPH) flag2 = DeleteArcByKey(G,key\_tail,key\_head);

return flag1&&flag2;

}

Status DFSOneVex(Graph \* G,void \* visit,int first\_key);

/\*\*

\* func：深度优先遍历

\* @param G 图地址

\* @param visit 访问函数指针

\* @return 状态码

\*/

Status DFSTraverse(Graph \*G,void \* visit){

if(!G) return ERROR;

//初始化节点

for(int i = 0;i<MAX\_VERTEX\_NUM;i++){

if(G->vertices[i].tag!=UNEXIST){

G->vertices[i].tag=NEWLY;

}

}

cout << "请输入遍历起点"<<endl;

int first\_key;

cin >> first\_key;

//从一个顶点开始遍历

int flag = DFSOneVex(G,visit,first\_key);

//对于非连通图，需要检查是否还有未被遍历的节点

for(int i = 0;i<MAX\_VERTEX\_NUM;i++){

if(G->vertices[i].tag==NEWLY)

DFSOneVex(G,visit,i);

}

return OK;

}

//从一个顶点开始遍历（递归形式）

Status DFSOneVexRec(Graph \* G,void \*visit,int first\_key){

int (\* vst)(Graph \* G, Vnode \* node);

vst = (int (\* )(Graph \* G,Vnode \*node))visit;//访问函数

if(first\_key<0||first\_key>=MAX\_VERTEX\_NUM||G->vertices[first\_key].tag==UNEXIST) return ERROR;

if(G->vertices[first\_key].tag==NEWLY){

vst(G,&G->vertices[first\_key]);

G->vertices[first\_key].tag=OLD;

for(int key = FirstAdjVex(G,first\_key);key>=0;key = NextAdjVex(G,first\_key)){

DFSOneVex(G,visit,key);

}

}

return OK;

}

//从一个顶点开始遍历（非递归形式）

Status DFSOneVex(Graph \* G,void \*visit,int first\_key){

int (\* vst)(Graph \* G, Vnode \* node);

vst = (int (\* )(Graph \* G,Vnode \*node))visit;//访问函数

int key;

if(first\_key<0||first\_key>=MAX\_VERTEX\_NUM||G->vertices[first\_key].tag==UNEXIST) return ERROR;

stack<int> st;

st.push(first\_key);

do{

first\_key = st.top();st.pop();

if(GetVex(G,first\_key)->tag!=NEWLY) continue;

vst(G,&G->vertices[first\_key]);

G->vertices[first\_key].tag=OLD;

for(int key = FirstAdjVex(G,first\_key);key>=0;key = NextAdjVex(G,first\_key)){

if(GetVex(G,key)->tag==NEWLY) st.push(key);

}

}while(!st.empty());

return OK;

}

Status BFSOneVex(Graph \* G,void \* visit,int first\_key);

/\*\*

\* func:广度优先遍历

\* @param G 图地址

\* @param visit 访问函数

\* @return 状态码

\*/

Status BFSTraverse(Graph \*G,void \* visit){

if(!G) return ERROR;

//初始化节点的tag域为NEWLY

for(int i = 0;i<MAX\_VERTEX\_NUM;i++){

if(G->vertices[i].tag!=UNEXIST){

G->vertices[i].tag=NEWLY;

}

}

cout << "请输入遍历起点"<<endl;

int first\_key;

cin >> first\_key;

if(first\_key<0||first\_key>=MAX\_VERTEX\_NUM||G->vertices[first\_key].tag==UNEXIST) return ERROR;

BFSOneVex(G,visit,first\_key);//从一个节点开始进行广度遍历

//当图为非连通图的时候，再进行遍历

for(int i = 0;i<MAX\_VERTEX\_NUM;i++){

if(G->vertices[i].tag==NEWLY) BFSOneVex(G,visit,i);

}

return OK;

}

//从一个节点开始进行广度遍历

Status BFSOneVex(Graph \* G,void \* visit,int first\_key){

int (\* vst)(Graph \* G, Vnode \*node);

vst = (int (\* )(Graph \* G,Vnode \*node))visit;

queue<int> Q;//借助队列实现广度遍历

Q.push(first\_key);

do{

first\_key = Q.front();Q.pop(); //获取队首元素，并出队列

//依次遍历出队列的邻接点，并加入到队列中

for(int key = FirstAdjVex(G,first\_key);key>= 0;key = NextAdjVex(G,first\_key)){

if(GetVex(G,key)->tag!=NEWLY) continue;

Q.push(key);

}

//如果该元素未被遍历，那么进行遍历

if(GetVex(G,first\_key)->tag==NEWLY){

vst(G,GetVex(G,first\_key));

G->vertices[first\_key].tag=OLD;

}

}while(!Q.empty());//循环结束条件是队列空

return OK;

}

/\*\*

\* func: 切换图

\* @param graph 图数组

\* @param G 图指针的地址

\* @param num 图编号

\* @return 状态码

\*/

Status ChangeGraph(Graph \* graph[], Graph \*\* G, int num){

if(!graph[num]) (\*G) = graph[num];

else return ERROR;

return OK;

}

/\*\*

\* func: 保存数据

\* @param G 图地址

\* @return 状态码

\*/

Status SaveData(Graph \* G){

FILE \* fvex, \* farc, \*finfo;

fvex = fopen("vex","w");

if(!fvex) return ERROR;

//依次写入顶点信息

for(int i = 0;i<MAX\_VERTEX\_NUM;i++){

if(G->vertices[i].tag!=UNEXIST){

fprintf(fvex,"%d %d %s ",i,G->vertices[i].data.n,G->vertices[i].data.c);

}

}

fprintf(fvex,"%d",-1);//文件末尾-1

fclose(fvex);//关闭文件

farc = fopen("arc","w");

if(!farc) return ERROR;

ArcNode \* p;

//依次写入边的信息

for(int i = 0;i<MAX\_VERTEX\_NUM;i++){

if(G->vertices[i].tag!=UNEXIST){

p = G->vertices[i].firstarc;

while(p){

fprintf(farc,"%d %d %d ",i,p->adjvex,p->info.weigtht);

p = p->nextarc;

}

}

}

fprintf(farc,"%d",-1);//文件末尾-1

fclose(farc);//关闭文件

finfo = fopen("info","w");

if(!finfo) return ERROR;

//写入图的信息

fprintf(finfo,"%d %d %d",G->kind,G->vexnum, G->arcnum);

fclose(finfo);//关闭文件

return OK;

}

/\*\*

\* func:加载数据

\* @param Gra 图指针的地址

\* @return 状态码

\*/

Status LoadData(Graph \*\* Gra){

//如果图不存在，则分配空间

if(\*Gra==NULL) \*Gra = (Graph \*)malloc(sizeof(Graph));

Graph \* G = \*Gra;

//初始化节点全部为不存在

for(int i = 0;i<MAX\_VERTEX\_NUM;i++) G->vertices[i].tag = UNEXIST;

FILE \* fvex, \*farc, \*finfo;

//只需要读取各个点，然后边集通过InsertArc完成

finfo = fopen("info","r");

if(!finfo) return ERROR;

//读取图信息

fscanf(finfo,"%d%d%d",&G->kind,&G->vexnum,&G->arcnum);

fclose(finfo);

fvex = fopen("vex","r");

if(!fvex) return ERROR;

int key;

//循环读取顶点信息

while(fscanf(fvex,"%d",&key) && key>=0&&key<MAX\_VERTEX\_NUM){

fscanf(fvex,"%d%s",&G->vertices[key].data.n,G->vertices[key].data.c);

G->vertices[key].tag=NEWLY;

G->vertices[key].firstarc = G->vertices[key].cur = G->vertices[key].cur = NULL;

}

fclose(fvex);//关闭文件

farc = fopen("arc","r");

if(!farc) return ERROR;

int head,tail,weight;

//循环插入边

while(fscanf(farc,"%d",&head)&&head>=0&&head<MAX\_VERTEX\_NUM){

fscanf(farc,"%d%d",&tail,&weight);

InsertArc(G,head,tail,weight);

}

fclose(farc);

//将cur域归位成第一个，cur是为了first和nextAdj准备的

for(int i = 0;i<MAX\_VERTEX\_NUM;i++){

G->vertices[i].cur = G->vertices[i].firstarc;

}

return OK;

}